NESTFUL: A Benchmark for Evaluating LLMs on Nested Sequences of API Calls

Kinjal Basu[1*], **Ibrahim Abdelaziz**[1*], **Kiran Kate**[1], **Mayank Agarwal**[1], **Maxwell Crouse**[1], **Yara Rizk**[1], **Kelsey Bradford**[2], **Asim Munawar**[1], **Sadhana Kumaravel**[1], **Saurabh Goyal**[1], **Xin Wang**[1], **Luis A. Lastras**[1], **and Pavan Kapanipathi**[1]

[1]IBM Research, USA [2]Georgia Institute of Technology

kinjal.basu@ibm.com, ibrahim.abdelaziz1@ibm.com

Abstract

The resurgence of autonomous agents built using large language models (LLMs) to solve complex real-world tasks has brought increased focus on LLMs' fundamental ability of tool or function calling. At the core of these agents, an LLM must plan, execute, and respond using external tools, APIs, and custom functions. Research on tool calling has gathered momentum, but evaluation benchmarks and datasets representing the complexity of the tasks have lagged behind. In this work, we focus on one such complexity, nested sequencing, with the goal of extending existing benchmarks and evaluations. Specifically, we present NESTFUL, a benchmark to evaluate LLMs on nested sequences of API calls, i.e., sequences where the output of one API call is passed as input to a subsequent call. NESTFUL contains 1800+ nested sequences where all the function calls are executable. Experimental results on a variety of models show that the best-performing model (GPT-4o) achieves a full sequence match accuracy of 28% and a win-rate of 60%, necessitating a large scope for improvement in the nested sequencing aspect of function calling. Our analysis of these results provides possible future research directions for the community, in addition to a benchmark to track progress. We have released the NESTFUL dataset under the Apache 2.0 license at https://github.com/IBM/NESTFUL.

1 Introduction

Autonomous agents, built with Large language models (LLMs), are gaining popularity in solving complex, real-world problems (Yao et al., 2023; Deng et al., 2024). LLMs handle a user's request by understanding their intents, planning the required tasks to address it, executing those tasks step by step, and providing a response. For most real-world problems (Jimenez et al.; Roy et al., 2024; Thakur et al., 2023), LLMs must interact with external environments through tool, function, and API calls (Application Programming Interface), which primarily leverages LLMs' tool calling abilities[1].

The significant reliance on LLMs' function-calling abilities led recent research to continuously improve this dimension of LLMs. On one hand, approaches to improve function calling have exploded (Abdelaziz et al., 2024; Liu et al., 2024; Srinivasan et al., 2023); on the other hand, benchmarks and evaluations are lagging behind. For instance, BFCL v1 and v2 focused on evaluating single, multiple, and parallel function calling tasks for both non-executable and executable versions (Yan et al., 2024). Works such as API-Blend (Basu et al., 2024) complement prior work by introducing granular task evaluation, such as slot-filling, API-detection, and sequencing. BFCL v3 has progressed further into agentic use cases with multi-step and multi-turn function calling evaluation.

However, benchmarks for fundamental but complex tasks such as the sequencing of functions have not been well explored yet, which forms the basis of this work. Existing evaluation benchmarks pose sequencing as the prediction of single or multiple isolated API calls, where the output of any particular API call within that sequence is considered irrelevant. In contrast, for many real-world tasks, a sequence of API calls is nested, i.e., the output of some API calls is used in the arguments of subsequent API calls. Figure 1 shows one such example of a nested sequence of APIs.

In this paper, we present NESTFUL, a benchmark specifically designed to evaluate models on nested API calls with over 1800 nested sequences. It consists of: (1) user queries, (2) a catalog of APIs and their specifications, (3) the sequence of API calls and the corresponding parameters, and

[*]These authors contributed equally to this work

[1]API, function calling, and tool-use are used interchangeably throughout the paper

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing, pages 33538–33547

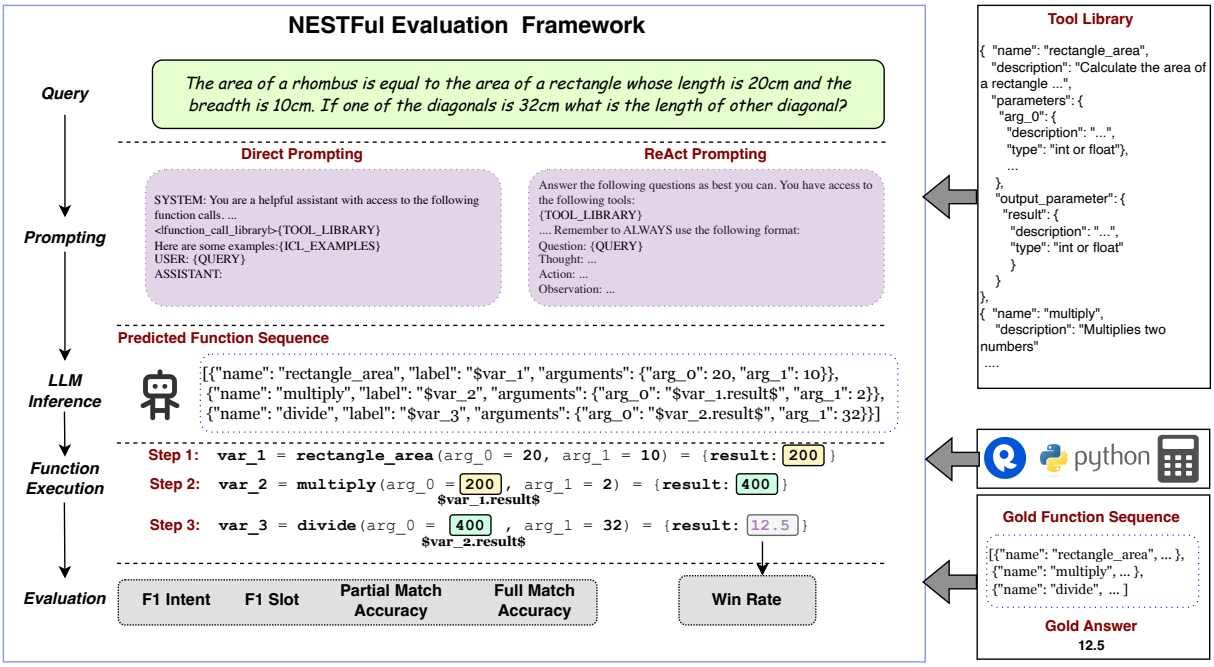


Figure 1: End-to-End Evaluation Pipeline: NESTFUL provides a test set of input queries and their corresponding list of nested function calls. It also provides executable implementations for each tool in the library and allows for evaluating models in direct prompting or REACT styles. Given a query, the pipeline infers the input LLM to generate the required sequence of function calls, execute those functions (taking into account nested variables), and compare the final answer with the gold one.

(4) the expected output response. The datasets are based on the MathQA (Amini et al., 2019) and StarCoder2-Instruct (Wei et al., 2024) datasets, which are commonly used in the literature but are missing the executable component. Figure 1 shows the end-to-end evaluation pipeline of NESTFUL.

We evaluated the dataset on 19 state-of-the-art models from the literature and exposed the gaps of these models in handling complex function calling sequences. GPT-4o achieved the best performance, but did not exceed 28% full sequence match nor 60% on win rate metrics. Models struggled as the nesting got deeper and the data dependencies increased. To further advance research in this area, we will publicly release the NESTFUL dataset with executable Python implementation for each tool and the evaluation code for all models.

2 Related Work

The best way to enable API function calling in LLMs remains an active area of research. Methods that utilize large, general-purpose proprietary models (e.g., Gemini (Team et al., 2023) or GPT (Achiam et al., 2023)) typically make use of carefully constructed prompts and in-context learning examples, e.g., (Song et al., 2023). Smaller, more specialized models often start from a strong-performing code model (e.g., DeepSeek-Coder (Guo et al., 2024), CodeLlama (Roziere et al., 2023), or Granite Code (Mishra et al., 2024)) and fine-tune primarily on highly curated datasets (Srinivasan et al., 2023; Ji et al., 2024; Abdelaziz et al., 2024) that have been extended with synthetic data (Zhang et al., 2024a).

Most of these existing works initially focused on basic function-calling abilities that did not involve much complexity. Recent advances in enabling LLMs to handle complex multi-API interactions have introduced structured methods like Reverse Chain (Zhang et al., 2024c). This approach employs backward reasoning to optimize multi-step API planning, allowing LLMs to effectively manage nested workflows by aligning intermediate steps with the final goal. Such methods highlight LLMs' potential to perform efficient, target-driven planning in resource-constrained environments.

To evaluate and enhance these aforementioned approaches, numerous works released training and benchmarking data in service of API function calling, such as ToolLLM (Qin et al., 2023); APIBench (Patil et al., 2023); APIGen (Liu et al., 2024); or API-BLEND (Basu et al., 2024). While these

benchmarks focus on simpler or isolated API calls, NesTools (Han et al., 2024) and our work target more complex, real-world tasks involving interdependent, nested tool use. Unlike the fully synthetic NesTools, NESTFUL is built from established datasets and has longer average call sequences (4.36 vs. 3.04). Other benchmarks like SealTool (Wu et al., 2024) and BFCL-v3 (Yan et al., 2024) include some nesting but are smaller and not specifically designed for it.

ToolBench (Xu et al., 2023) includes 205 nested samples from the Webshop and TableTop datasets, compared to over 1800 in NESTFUL. Moreover, ToolBench supports only 34 APIs, whereas NESTFUL features more than 900 unique functions. ShortcutsBench (Shen et al., 2024) has multi-step tool calls but is tailored specifically to the Apple Shortcuts feature, with evaluation data generated in a proprietary format that is difficult to interpret and does not follow a standard JSON structure. Finally, AgentBoard is a broad evaluation framework for general-purpose agents for multi-step planning tasks with fine-grained metrics and visual tools, while NESTFUL focuses specifically on tool-augmented LLMs with detailed offline metrics like F1 score and accuracy alongside win rate.

3 NESTFUL Dataset Curation

NESTFUL comprises more than 1,800 instances designed for benchmarking tool calling in LLMs on nested sequencing. Each instance consists of (1) a user query, (2) a list of all available tools for the model to choose from, (3) the gold sequence of tools and their arguments needed to answer the user query, and (4) the final answer that should be obtained once the tools are executed. The dataset also contains corresponding Python code for every API in the library and a mechanism to run the input query via any LLM, execute the tools predicted by the LLM, and provide the final answer.

3.1 Nested Function Calling Data Schema

NESTFUL's data schema, demonstrated in Figure 2, showcases the template used for representing the *Input*, *Tool Library*, *Output*, and *Python Code*. An important aspect of nested function calling is to enable a mechanism for tool reference; i.e., a subsequent tool call using that reference to access the output of the previous tool execution. To do so, we assign a *unique variable name* to each tool, which distinctly identifies each tool, even when two identical tools with different arguments appear in the same sequence (parallel API calls). For example, in Figure 1, "rectangle_area" tool was assigned "label": "$var_1". This allows the next tool "multiply" to use the output of "rectangle_area" as an argument: "arg_0": "$var_1.result$".

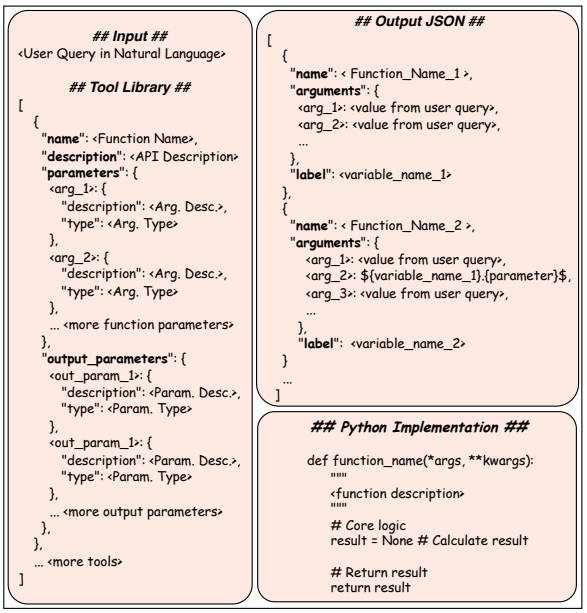


Figure 2: NESTFUL Data Schema for *Input*, *Tool Library*, *Output*, and *Python implementation*. In *Output JSON*, the `arg_2` of `Function_Name_2` showcases how the variable assignments are used to create a nested sequence of function calls.

```
{
  "name": "surface_sphere",
  "description": "Calculate the surface area of a sphere",
  "parameters": {
    "arg_0": {
      "description": "Radius of the sphere",
      "type": "float"
    }
  },
  "output_parameters": {
    "result": {
      "description": "Surface area of the sphere",
      "type": "float"
    }
  }
}
```

Figure 3: Sample specification for tools from MathQA

3.2 NESTFUL Data Domains

NESTFUL is composed of data from two domains; 1) mathematical reasoning data and 2) generic tools from the coding domain. We describe below the process followed to create each data category.

3.2.1 Mathematical Reasoning Data

For the first part of NESTFUL, we relied on datasets that test the model for nested function calling in the math domain. We build on

Problem:
An artist wishes to paint a circular region on a square poster that is 3.4 feet on a side. if the area of the circular region is to be 1/2 the area of the poster, what must be the radius of the circular region in feet?

Operation Stack:
`square_area(3.4) → divide(#0, 2) → divide(#1, const_pi) → sqrt(#2)`

Tool Call format:
```
[{"name": "square_area", "arguments": {"arg_0": 3.4}, "label": "var_0"},
{"name": "divide", "arguments": {"arg_0": "$var_0.result$", arg_1: 2} "label": "var_1"},
{"name": "divide", "arguments": {"arg_0": "$var_1.result$", "arg_1": "pi"}, "label": "var_2"},
{"name": "sqrt", "arguments": {"arg_0": "$var_2.result$"}, "label": "var_3"}]
```

Final Answer:
1.3564

Figure 4: Sample problem from the MathQA dataset. The *Operation Stack* provides an ordered sequence of nested tool calls which we transform into a *Tool Call format* for the NESTFUL dataset.

MathQA (Amini et al., 2019), a benchmark designed to evaluate AI models' ability to solve mathematical word problems. It consists of questions that test numerical reasoning and problem-solving skills, requiring models to both understand the text of a word problem and perform mathematical operations to arrive at the correct solution.

Tool specifications Since MathQA provides only the tool names, we manually created specifications for all the tools in the dataset. This covers 40 tools in total; e.g., divide, remainder, volume_cylinder, permutation, etc. For each tool, we define the name, tool description, and detailed outline of the tool input and output parameters, including the parameter data type and description as shown in Figure 3.

Tool calling input-output pairs To build the test data, we used the test set of MathQA where the "problem" definition is the query and parsed the "annotated formula" into a nested sequence of tool calls. An example is shown in Figure 4.

Executable Code and Filtering For each tool, we also generated its corresponding implementation in Python. This allows us to execute the nested call sequence and match the execution result with the gold answer. It also ensures the correctness of the set of corresponding tools and the code execution too. We then filtered out any samples where we could not reproduce the gold answer from executing the nested tool calls. This process resulted in 1,415 test samples spanning 40 tools with an average of 5.1 tool calls per sample.

3.2.2 Coding Data

We also curated test examples based on generic Python functions from the StarCoder2-Instruct dataset (Wei et al., 2024). This dataset has a total of 50K Python functions and covers a wide range of tools that can be used. We started by collecting tool instructions and their Python implementations, followed by using Mixtral-8x22B to infer parameter type hints. Any functions that were syntactically incorrect or non-executable were filtered out. Next, a synthetic data generation pipeline was used to create instruction–nested call pairs using the valid seed tools and examples. This pipeline included a validator to ensure all parameters and tool names were accurate and complete. Finally, execution-based filtering was applied to verify that the generated samples produce the correct final output. We elaborate on each step in the following sections.

Tool specifications We leveraged StarCoder2-Instruct's Python implementations and docstrings to create API specifications, see Figure 5 for an example. For each Python function, (1) we used Mixtral 8x22B[2] to generate the possible Python types of the input and output arguments, and (2) we validated and executed the function code itself to ensure it does not produce any errors. After both steps, we generated a corresponding JSON specification for each function documenting its input-output arguments. Figure 5 shows an example.

Tool calling input-output pairs To create input-output pairs, we leveraged DiGiT[3] synthetic data generation framework. DiGiT allows for defining various synthetic data generation pipelines given seed examples of input/output pairs. In particular, we used 10 seed examples and used Mixtral-8x22B as the teacher model. We also implemented a function calling validator that applies various heuristics to check the quality of the synthetic data, ensuring function calls adhere to the given specifications. In

[2] https://huggingface.co/mistralai/Mixtral-8x22B-Instruct-v0.1

[3] https://github.com/foundation-model-stack/fms-dgt

Python Implementation

```
def angle_rotation(arr, k):
    """Rotates the elements in the array by `k` positions
       in the right direction.
    Args:
        arr: The array of numbers.
        k:   The number of positions to rotate the elements
             in the array.
    Returns:
        The array after rotating the elements by `k`
        positions in the right direction.
    """
    if len(arr) < k:
        k = k % len(arr)
    return arr[len(arr)-k:] + arr[:len(arr)-k]
```

Tool Specification

```
{
  "name": "angle_rotation",
  "description": "Rotates the elements in the array by `k`…",
  "parameters": {
    "properties": {
      "arr": {
        "description": "The array of numbers.",
        "items": {
          "type": "integer"
        },
        "type": "array"
      },
      "k": {
        "description": "The number of positions to rotate …",
        "type": "integer"
      }
    },
    "required": [
      "arr",
      "k"
    ],
    "type": "object"
  },
  "output_parameters": {
    "properties": {
      "output_0": {
        "description": "The array after rotating …",
        "type": "array"
      }
    }
  }
}
```

Figure 5: [*top*] An example function (*angle_rotation*) from StarCoder2-Instruct dataset with its docstring documentation. [*bottom*] Tool specification for "angle_rotation" after inferring the different data types and creating its input/output parameters.

particular, we have validations to ensure the tools and parameters used are not hallucinated, required parameters are specified, and there is at least one nested tool call in the output sequence.

Filtering We further filtered the generated input-output pairs by executing their gold nested API sequence to ensure they execute and attach the result as the gold answer. For a function that has a randomness element (e.g., generating a random list), we set a fixed seed for all our experiments and re-execute all those cases to ensure that we are getting the same response all the time. From this category of data, we generated 446 test examples covering more than 881 distinct tools with an average tool sequence length of 2.1.

3.3 Dataset Quality

Our benchmark builds on MathQA and StarCoder2-Instruct. MathQA is a well-known mathematical reasoning dataset that was manually validated by humans, providing the input, the correct sequence of math operations, and the final answer. After converting it into a nested tool sequence, we further validate it by executing the sequence to ensure it produces the original gold answer. However, since the coding dataset is synthetically generated, we also implemented multiple automatic validations at various stages. This includes checking that the nested tool sequences align with tool specifications (input/output) and that they execute correctly to produce the expected final output.

4 Evaluation

4.1 Baselines

We extensively evaluated NESTFUL on 19 proprietary and open-source models, ranging in size from 1B to 685B parameters. This selection includes top tool-calling LLMs featured on the Berkeley Function-Calling Leaderboard (BFCL) (Yan et al., 2024), as well as state-of-the-art models known for strong function-calling capabilities. Among the tool-calling models, we include the xLAM (Zhang et al., 2024b; Liu et al., 2024), Hammer (Lin et al., 2024), ToolAce (Liu et al.) model families, and Granite-20B-FunctionCalling (Abdelaziz et al., 2024). We also evaluate a range of foundation models, including multiple sizes of LLAMA 3.1 (Dubey et al., 2024), Mixtral[4], and DeepSeek-V3 (Guo et al., 2024), and the state-of-the-art proprietary model GPT-4o (Hurst et al., 2024). To explore how an agentic LLM performs on NESTFUL, we also include AgentLM-13B (Zeng et al., 2023), which has been instruction-tuned using interaction trajectories from diverse agentic tasks.

4.2 Experimental Settings

The experiments are carried out with a temperature of 0.0 in one-shot and three-shot settings, i.e., the prompt contains one or three in-context learning (ICL) examples, respectively. To the best of our knowledge, all 18 open models were not trained with the *label* assignment syntax in the output API sequence, so it was crucial to have ICL examples to get the best results. For each model, we used its specified prompt along with the special tags. Context length limitations prevented the inclusion of the entire API library in the prompt. Instead, we pre-processed the data to create a shorter API list for each sample. This list ensured the inclusion

[4] https://huggingface.co/mistralai/

Model	#Parameters	One-shot ICL					Three-shots ICL				
		F1 Func.	F1 Param.	Part. Acc.	Full Acc.	Win Rate	F1 Func.	F1 Param.	Part. Acc.	Full Acc.	Win Rate
xLAM-1b-fc-r	1B	0.19	0.08	0.09	0.00	0.01	0.22	0.09	0.09	0.03	0.02
xLAM-2-1b-fc-r	1B	0.41	0.13	0.13	0.00	0.00	0.43	0.12	0.13	0.00	0.00
xLAM-7b-fc-r	7B	0.49	0.17	0.15	0.00	0.03	0.55	0.23	0.23	0.15	0.14
xLAM-2-8b-fc-r	8B	0.48	0.15	0.14	0.00	0.01	0.47	0.17	0.15	0.04	0.04
Hammer2.0-7b	7B	0.56	0.24	0.21	0.07	0.16	0.61	0.30	0.29	0.22	0.25
Hammer2.1-7b	7B	0.10	0.05	0.05	0.01	0.01	0.16	0.10	0.11	0.08	0.08
Llama-3-1-8B-Instruct	8B	0.64	0.19	0.17	0.06	0.06	0.63	0.22	0.22	0.16	0.11
ToolACE-8B	8B	0.43	0.13	0.13	0.00	0.00	0.50	0.15	0.13	0.00	0.00
ToolACE-2-Llama-3.1-8B	8B	0.28	0.10	0.13	0.00	0.00	0.29	0.10	0.13	0.00	0.00
Granite-20B-FunctionCalling	20B	0.64	0.20	0.17	0.02	0.05	0.61	0.25	0.26	0.21	0.20
Mixtral-8x7B-Instruct-v0.1	46.7B	0.22	0.07	0.05	0.00	0.01	0.32	0.13	0.14	0.09	0.09
xLAM-8x7b-fc-r	46.7B	0.40	0.15	0.16	0.01	0.01	0.43	0.16	0.17	0.02	0.03
Llama-3-1-70B-Instruct	70B	0.41	0.19	0.15	0.04	0.09	0.33	0.17	0.15	0.07	0.11
Mixtral-8x22B-Instruct-v0.1	141B	0.49	0.21	0.17	0.06	0.07	0.65	0.29	0.28	0.21	0.23
xLAM-8x22b-fc-r	141B	0.53	0.21	0.22	0.12	0.03	0.50	0.23	0.25	0.17	0.06
Llama-3-1-405B-Instruct-fp8	405B	0.41	0.14	0.08	0.03	0.10	0.41	0.18	0.13	0.07	0.14
DeepSeek-V3	685B	0.69	0.36	0.27	0.09	0.43	0.69	**0.42**	0.37	**0.29**	**0.60**
GPT-4o (2024-08-06)	UNK	**0.73**	0.41	**0.38**	**0.28**	**0.59**	**0.74**	0.41	**0.38**	0.28	**0.60**

Table 1: Evaluation results on NESTFUL on state-of-the-art LLMs with Direct Prompting technique. Models are sorted by their size. Experiments are done in one-shot and three-shot ICL settings. The best performance is highlighted in **bold**; the second best is underlined. **Partial Sequence Accuracy (Part. Acc.)** denotes the percentage of calling the correct API sequence (API names and arguments), whereas **Full Sequence Accuracy (Full Acc.)** counts the percentage of times where the model gets the entire sequence of APIs correctly. Both scores range from 0 to 1. We also report **Win Rate**, which measures whether all the predicted APIs by the model are executable and lead to an exact match with the gold answer.

of the gold APIs, the APIs used in the in-context learning (ICL) examples, and some random APIs, keeping the total prompt length under 4K tokens. On average, each input includes 11.2 tools, with a minimum of 7 and a maximum of 21, and all the models are evaluated on the exact same pre-processed tool lists, and the data remains fixed across evaluations. Output API calls were extracted from the model's response as a list of JSON objects, taking into account the specific prompt format and output structure for each model. Finally, we evaluated a zero-shot ReAct (Yao et al., 2022) agent with the best 4 open models based on the win-rate and AgentLM-13B, limiting max steps to 10.

4.3 Metrics

For a detailed evaluation, we use the following metrics: 1) F1 score for function and parameter names generation (Abdelaziz et al., 2024), 2) ***Partial*** and ***Full Match Accuracy***, and 3) ***Win Rate***.

LLM model response is a sequence of API calls, with each call consisting of an API name and its argument-value pairs. We use the ***Partial Sequence Matching*** metric to determine how many predicted APIs (with their argument-value pairs) in a sequence match with the gold API sequence. In contrast, the ***Full Sequence Matching*** metric evaluates whether the model predicts the exact full sequence of APIs, including both the API names and their argument-value pairs, when compared to the gold API sequence. In both cases, we calculate the scores for each sample and then compute the statistical mean across the entire dataset as the final score. However, F1 and Accuracy metrics can unfairly penalize valid alternative API sequences that differ from the gold sequence but still produce the correct result. To address this limitation, we introduce the ***Win Rate*** metric. Win Rate measures whether the predicted APIs are valid and, when executed, lead to the gold answer. In this way, Win Rate focuses solely on the correctness of the final output, regardless of the specific execution path. A win is recorded if the predicted answer matches the gold answer.

4.4 Results

Table 1 presents a comparison of different baselines on the NESTFUL dataset with one-shot and three-shot ICL example settings. The low numbers of the best function calling models depict the complexity and toughness of the nested sequencing problem. GPT-4o and DeepSeek-V3 achieve the highest win-rate of 60%, which is significantly below the acceptable numbers for real-world applications in general. This clearly depicts the significant scope for improvements for the models in various aspects of function calling, including nested sequencing. We inspected the models' outputs and identified several common issues across them. These models struggle with tasks such as assigning variables, utilizing output parameter details from the API specifications, and correctly passing

variable names and corresponding output parameters to subsequent APIs, even with ICL examples[5].

As anticipated, in most of our experiments in Table 1, the models are doing better across all the metrics when they are provided with three ICL examples in the prompt instead of one example. Across all models, *Partial Sequence Match* scores are consistently higher than *Full Sequence Match* scores, which is expected, as the latter is a stricter metric than the former. In many cases, the *Win Rate* is higher than the *Full Accuracy* because models may follow alternative reasoning paths/trajectories to arrive at the correct final answer. While such deviations can penalize full or partial accuracy scores, they are still credited under the win rate for successfully reaching the gold answer. Hammer2.0-7b, despite being a smaller model, outperforms several larger tool-augmented LLMs. DeepSeek-V3 emerges as the strongest open-source model, closely matching GPT-4o's performance in the three-shot ICL setting, although it trails slightly in the one-shot setup. These results highlight that model size or architectural complexity is not the primary determinant of performance; rather, the ability to effectively follow instructions and leverage in-context examples plays a more critical role. This is evident as some large models like xLAM-8x22b-fc-r and Llama-3-1-405B-Instruct-fp8 underperform, while smaller models like Hammer2.0-7b achieve exceptional results.

ReAct-based Evaluation: The previous results showed how poorly direct prompting of LLMs performed on NESTFUL. The literature has shown that agentic approaches have resulted in better performance on complex tasks. While many agentic architectures exist, we selected ReAct due to its popularity and its ability to reason over the output of the tool that is added to the prompt at each turn.

Table 2 summarizes the results of the ReAct agent (Yao et al., 2023) compared to one-shot ICL direct prompting. Note that there was no ICL example provided in the ReAct case, as the expected output does not need to follow the *label* assignment syntax. We only report *Win Rate*, which checks if the trajectory of output tools leads to the gold answer, due to the single-step planning and execution approach of REACT as opposed to planning the entire API sequence at once in direct prompting.

[5]Note: While we acknowledge that these models were not trained using the robust data schema outlined in Section 3.1, the challenges associated with nested sequencing persist regardless of the schema used and remain an area where LLMs need improvement.

Model	Direct Prompting (One-shot ICL)	ReAct Agent (Zero-shot)
Hammer2.0-7b	**0.16**	0.07
AgentLM-13B	0.00	0.00
Mixtral-8x22B-Instruct-v0.1	0.07	**0.30**
DeepSeek-V3	0.43	**0.46**

Table 2: Evaluation results (**Win Rate**) on NESTFUL comparing the performance of a ReAct Agent (zero-shot) to the Direct Prompting with a one-shot ICL example. For each model, the best performance is highlighted in bold for comparison.

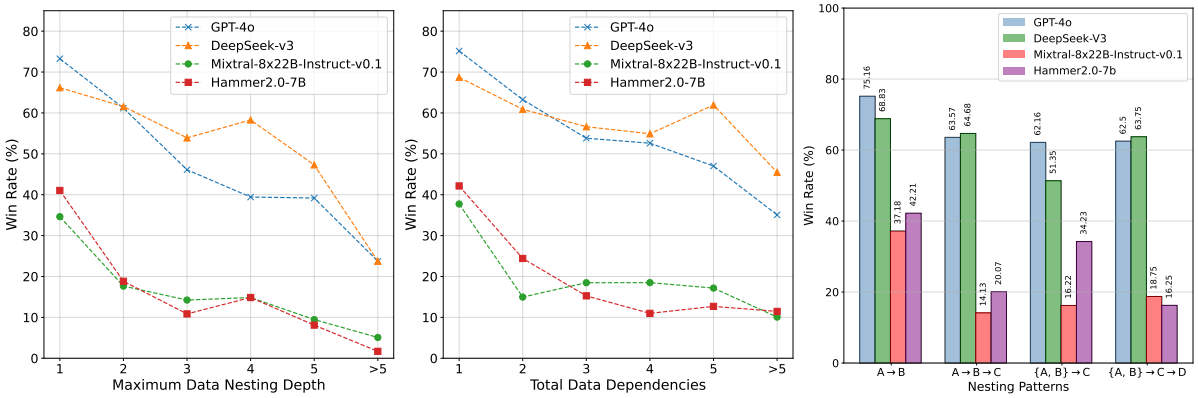
For larger models like Mixtral-8x22B-Instruct-v0.1 and DeepSeek-V3, the ReAct approach outperforms direct prompting, though there is still room for improvement. Notably, Mixtral-8x22B-Instruct-v0.1 shows the highest win-rate gain of 30% with ReAct. Hammer2.0-7B performs better with direct prompting compared to the ReAct approach. Although AgentLM-13B is specifically trained for agentic tasks, it does not demonstrate strong performance on NESTFUL, indicating that agent-specific training or architectures do not always guarantee improved results in this setting. Output analysis reveals that the top-performing models exhibited better alignment with the ReAct format and occasionally relied on their parametric knowledge to replicate API functionalities. As a result, they achieved correct final outcomes, reflected in a higher *win-rate* despite inconsistencies in intermediate steps.

4.5 Dataset Analysis

To analyze the results, we model the samples in the NESTFUL as a Directed Acyclic Graph (DAG) where nodes are individual function calls and the edges are data dependencies between two nodes.

Data Nesting Depth Analysis In Figure 6a, we present the win rate for top-performing models against varying levels of maximum depth in the DAG structure, which corresponds to the longest nested data dependency flow in a sample. We observe that all models perform well for samples with maximum single nesting depth. However, the performance drops sharply with depths of two or more, suggesting that long nested sequences present difficult scenarios for current models.

Total Data Dependency Analysis In Figure 6b, we present the win rate compared to the total number of data dependencies within a sample (a representation of the complexity of the sequence). The



(a) Data Nesting Depth Analysis (b) Total Data Dependencies Analysis (c) Nesting Pattern Analysis

Figure 6: Top 4 models' performances (on the three-shot setting) with a varying number of (a) longest data dependencies, (b) the total number of data dependencies, and (c) common data nesting patterns in NESTFUL. We observe that models perform well for the simple data samples, however, the performance drops sharply with complicated structural patterns in the data.

trends here are similar to the ones we observed in the Nesting Depth Analysis section. Models perform well for singular nested data dependency in the sample, achieving around 75% *win-rate*, but the performance drops sharply with two or more nested data dependencies.

Nesting Patterns Analysis We identify common nesting patterns and analyze model performances for individual nesting patterns. Results are shown in Figure 6c. We observe that for a simple pattern such as `A` $\rightarrow$ `B`, where the output of `A` is used as input by `B`, all models perform fairly well. However, for complex patterns such as `{A, B}` $\rightarrow$ `C`, where the outputs of both `A` and `B` are used as input by `C`, model performance decreases significantly. This suggests that models currently struggle with more complex patterns present in NESTFUL.

5 Challenges

Results show that NESTFUL posed a challenge for state-of-the-art LLMs for several reasons.

Data-type Adherence for the Input/Output Parameters In the API specification, we define the data type for all parameters. The 'type' field specifies the data type, such as string, number, list, etc. Since APIs follow a strict structure for both input and output, it is crucial for the model to adhere to these specified formats. If the model fails to follow this, especially in cases involving nested functions where the output of one API is used as the input for another, the process will fail if the output type doesn't match the expected input type.

Variable Assignments As discussed in Section 3.1, we add variable assignments for each API in the output to manage parallel function calls, which is very common in real-life applications. Below is an example of parallel nested function calls:

```
{"input": "What is the difference between the
     squares of 4 and 3?"
 "output": [
    {"name": "square", "arguments": {arg_0: 4},
      "label": "$var_1"},
    {"name": "square", "arguments": {arg_0: 3},
      "label": "$var_2"},
    {"name": "subtraction", "arguments": {arg_0:
        $var_1.result$, arg_1: $var_2.result$},
      "label": "var_3"},
 ]}
```

The example highlights the complexity of distinguishing repeated functions with different outputs, which models struggle with due to a lack of schema training—a challenge also supported by our qualitative analysis in Section 4.4.

Implicit API calling Implicit function calling occurs when a model must identify and invoke the appropriate APIs to solve a user query, even though the query doesn't explicitly mention them. This requires understanding the problem, selecting the correct functions, and filling in parameters using query details or previous outputs—adding significant complexity to the task. Figure 1 demonstrates an example of implicit function calling, where the user query presents an arithmetic problem without explicitly stating the APIs involved.

6 Conclusion

In this work, we introduced NESTFUL, a new benchmark for evaluating the LLMs on nested sequences of API function calling. Existing LLMs

perform poorly on this dataset as compared to their performance on existing benchmarks. We also studied their performance and identified several modes of failure. In addition, we outlined the many challenges this dataset poses to LLM function calling approaches. By making this dataset available publicly under a permissive open-source license, we aim to push tool calling capabilities in new directions and unlock solutions to more realistic and challenging tasks.

Limitations

A limitation of our benchmark, NESTFUL, is that it does not involve environment interactions for AI agents or the execution of real-world APIs. In future work, it would be interesting to explore the setting of embodied agents, where API calls produce changes in a grounded environment. We also plan to expand the NESTFUL dataset with real-world API integrations (e.g., via RapidAPI).

Ethics Statement

Our dataset does not pose any ethical concerns for the community. All of the functions we consider are either related to the Math domain or generic functions that do not deal with personally identifiable information or functions that could be used maliciously. Furthermore, the user queries do not contain any hate, profanity or toxic content.

References

Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Sadhana Kumaravel, Matthew Stallone, Rameswar Panda, Yara Rizk, GP Bhargav, Maxwell Crouse, Chulaka Gunasekara, et al. 2024. Granite-function calling model: Introducing function calling abilities via multi-task learning of granular tasks. *arXiv preprint arXiv:2407.00121*.

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

Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *Preprint*, arXiv:1905.13319.

Kinjal Basu, Ibrahim Abdelaziz, Subhajit Chaudhury, Soham Dan, Maxwell Crouse, Asim Munawar, Sadhana Kumaravel, Vinod Muthusamy, Pavan Kapanipathi, and Luis A. Lastras. 2024. Api-blend: A comprehensive corpora for training and benchmarking api llms. *Preprint*, arXiv:2402.15491.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2024. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36.

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

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*.

Han Han, Tong Zhu, Xiang Zhang, Mengsong Wu, Hao Xiong, and Wenliang Chen. 2024. Nestools: A dataset for evaluating nested tool learning abilities of large language models. *Preprint*, arXiv:2410.11805.

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

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

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

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, et al. Toolace: Winning the points of llm function calling. 2024. *URL https://arxiv. org/abs/2409.00920*.

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, et al. 2024. Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, et al. 2024. Granite code models: A family of open foundation models for code intelligence. *arXiv preprint arXiv:2405.04324*.

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

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

Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring llm-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 208–219.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Haiyang Shen, Yue Li, Desong Meng, Dongqi Cai, Sheng Qi, Li Zhang, Mengwei Xu, and Yun Ma. 2024. Shortcutsbench: a large-scale real-world benchmark for api-based agents. *arXiv preprint arXiv:2407.00132*.

Yifan Song, Weimin Xiong, Dawei Zhu, Wenhao Wu, Han Qian, Mingbo Song, Hailiang Huang, Cheng Li, Ke Wang, Rong Yao, et al. 2023. Restgpt: Connecting large language models with real-world restful apis. *arXiv preprint arXiv:2306.06624*.

Venkat Krishna Srinivasan, Zhen Dong, Banghua Zhu, Brian Yu, Damon Mosk-Aoyama, Kurt Keutzer, Jiantao Jiao, and Jian Zhang. 2023. Nexusraven: a commercially-permissive language model for function calling. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*.

Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. 2023. A language-agent approach to formal theorem-proving. In *The 3rd Workshop on Mathematical Reasoning and AI at NeurIPS'23*.

Yuxiang Wei, Federico Cassano, Jiawei Liu, Yifeng Ding, Naman Jain, Zachary Mueller, Harm de Vries, Leandro von Werra, Arjun Guha, and Lingming Zhang. 2024. Selfcodealign: Self-alignment for code generation. *arXiv preprint arXiv:2410.24198*.

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 *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 372–384. Springer.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023. On the tool manipulation capability of open-source large language models. *arXiv preprint arXiv:2305.16504*.

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Berkeley function calling leaderboard. https://gorilla.cs.berkeley.edu/blogs/8_berkeley_function_calling_leaderboard.html.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*.

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

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*.

Jianguo Zhang, Tian Lan, Rithesh Murthy, Zhiwei Liu, Weiran Yao, Juntao Tan, Thai Hoang, Liangwei Yang, Yihao Feng, Zuxin Liu, et al. 2024a. Agentohana: Design unified data and training pipeline for effective agent learning. *arXiv preprint arXiv:2402.15506*.

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

Yinger Zhang, Hui Cai, Xeirui Song, Yicheng Chen, Rui Sun, and Jing Zheng. 2024c. Reverse chain: A generic-rule for llms to master multi-api planning. *Preprint*, arXiv:2310.04474.