CallNavi, A challenge and empirical study on LLM function calling and routing

Anonymous ACL submission

Abstract

Interacting with a software system via a chatbot can be challenging, especially when the chatbot needs to generate API calls, in the right order and with the right parameters, to communicate with the system. API calling in chatbot systems poses significant challenges, particularly in complex, multi-step tasks requiring accurate API selection and execution. We contribute to this domain in three ways: first, by introducing a novel dataset designed to assess models on API function selection, parameter generation, and nested API calls; second, by benchmarking state-of-the-art language models across varying levels of complexity to evaluate their performance in API function generation and parameter accuracy; and third, by proposing an enhanced API routing method that combines general-purpose large language models for API selection with fine-tuned models for parameter generation and some prompt engineering approach. These approaches lead to substantial improvements in handling complex API tasks, offering practical advancements for real-world API-driven chatbot systems.

1 Introduction

In modern conversational AI systems, such as chatbots, the ability to accurately call APIs is critical for ensuring effective interactions with users. This is especially true in complex scenarios where selecting API, handling multiple calls, and managing nested calls are required to retrieve or manipulate data dynamically. While research on API function generation has progressed significantly, much of the focus has been on generating syntactically correct **single** API calls rather than addressing the challenge of generating a **sequence** of complex API calls with logical dependencies. These aspects are important in scenarios that involve integration with retrieval-augmented generation (RAG) systems, where selecting & executing the right API within the context of external knowledge retrieval is key to task completion (Lewis et al., 2020).

To address this gap, our work introduces CallNavi, a novel, complex dataset designed to test large language models (LLMs) on API selection, multiple API calls, and nested calling structures. Unlike previous datasets that focus on generating single, isolated API calls, our dataset simulates real-world complexity by combining API generation with selection challenges. We comprehensively benchmark existing models, comparing their performance across different difficulty levels and task types. We also introduce function calling routing, which involves finding the correct API names and the order in which to call them, essentially acting like a task router in RAG. This serves as a key sub-task in determining the correct sequence of API calls, aligning with the focus on multiple API calling scenarios. Additionally, we explore optimization approaches that improve API selection accuracy and performance in nested call scenarios.

Our findings also reveal interesting insights through case studies, highlighting the limitations of current models and potential areas for improvement. For example, we demonstrate that general-purpose models tend to struggle with long or complex API lists, whereas a combination of general LLMs for API selection and fine-tuned models for specific call generation yields better results. This work not only fills a critical gap in existing research but also provides a foundation for future exploration in API calling and selection tasks within AI systems.

2 Research Questions

- **Benchmark** Which LLMs have the best performance for function calling in a real-world scenario?
- **Evaluation** Which is the best way to evaluate the API function calling ability of LLMs?

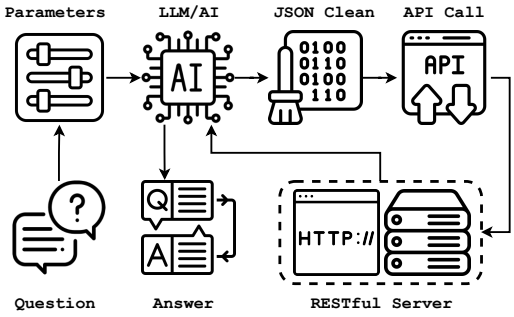

Figure 1: Pipeline of API calling while using LLMs

- **Optimization** How to enhance API function calling ability for zero/few-shot LLM?

3 CallNavi Dataset

3.1 Domains

To develop robust LLMs for complex API function calling, it's crucial to evaluate performance across diverse, real-world domains. The `CallNavi` dataset includes 10 common chatbot domains (see Table 1), reflecting various user queries and tasks and serving as a vital benchmark. These high-demand domains require advanced task routing and present unique challenges in language understanding, contextualization, task management, and compliance with regulations like GDPR. By encompassing a diverse set of domains, `CallNavi` provides a comprehensive framework for assessing LLM capabilities in real-world scenarios, ensuring that function calling systems are scalable and adaptable.

3.2 Generate and Meta Data

To create our dataset, we first used GPT-4o to generate API function names and parameters based on scenario descriptions provided. After the initial generation, we manually checked all the generated parameters to ensure accuracy and consistency with the intended functions. For more challenging tasks, we manually wrote all medium and hard-difficulty questions to better reflect complex API calling scenarios. To build the easier subset of the dataset, we again used GPT-4o to generate the questions related to API calling, which were then manually verified to ensure relevance and correctness. This hybrid approach allowed us to create a diverse and well-structured dataset across varying levels of difficulty, balancing automation with human oversight for high-quality results.

The format of samples in our dataset is shown in the code block below. The first part is the API name, description, and parameters.

```
{
    "name": "getAccountBalance",
    "parameters": ["accountID"],
    "description": "Retrieves the
        current balance for a
       specific account.",
    "returnParameter": {
      "Balance": "number"
    }
},
...
```

We then format each question as shown in the example below, which includes the user query, the ground truth API call in JSON format, and the difficulty level:

```
{
  "id": "ban01",
  "question": [
    {"role": "user",
    "content": "What is the
       balance for the account
       with ID 987654?"}],
  "ground_truth": {
    "API": ["getAccountBalance"],
    "parameters":
        {"accountID": "987654"}},
  "difficulty": "easy"
},
...
```

3.3 Size & Difficulty

Table 1: CallNavi dataset domains, questions and difficulties statistics table.

Domain	API Functions	Questions	Difficulty		
			Easy	Medium	Hard
Banking	91	115	70	28	17
Shopping	81	65	41	17	7
Logistics	46	65	40	17	8
Aviation	48	80	44	24	12
Healthcare	20	47	31	10	6
Public Services	82	85	50	27	8
Human Resources	20	35	21	13	1
Hotel Industry	49	65	40	19	6
Insurance	42	60	40	11	9
Telecommunications	100	112	79	22	11
Overall	579	729	456	188	85

3.4 Dataset

The *CallNavi* dataset evaluates LLMs' task routing and API calling capabilities across multiple domains. As shown in Table 1, it contains **729 questions** of varying difficulty and API interaction complexity, along with **579 distinct API functions**. Questions are categorized into **easy**, **medium**, and **hard** levels:

- **Easy (456 questions)**: Require a **single API** call to fulfill the task.
  Example: A user checking their bank account balance with one straightforward API call.

- **Medium (188 questions)**: Involve **multiple APIs** within the same question, with all necessary parameters provided in the context.
  Example: A shopping query needing product details and stock availability via two independent API calls.

- **Hard (85 questions)**: Require **multiple API calls** where some parameters depend on **responses from previous calls**, adding complexity. **5** steps maximum of API.
  Example: Updating delivery status by first retrieving a package ID, then using it to fetch the delivery status through sequential API calls.

This dataset tests LLMs' ability to perform function-calling routing and parameters generation across varying difficulties, assessing both basic single-call handling and complex multi-step nested requests. Zero-shot and few-shot models must infer correct API interactions only from the question context from different difficulties.

4 Metrics and Evaluation

4.1 API Parameters AST Match

Our study utilizes Abstract Syntax Tree (AST) evaluation to assess models' ability to generate accurate JSON outputs for API calls. The format of the output JSON and parameters follows a structure similar to the BFCL dataset in Section 3.2 (Yan et al., 2024). We parse the generated JSON string into an object and compare each component with the ground truth, such as the API list and parameters.

In scenarios involving multi-step API calls where parameters depend on previous steps or where text inputs may not have a single definitive answer, placeholder tokens are used for parameters. These tokens are positionally aligned with

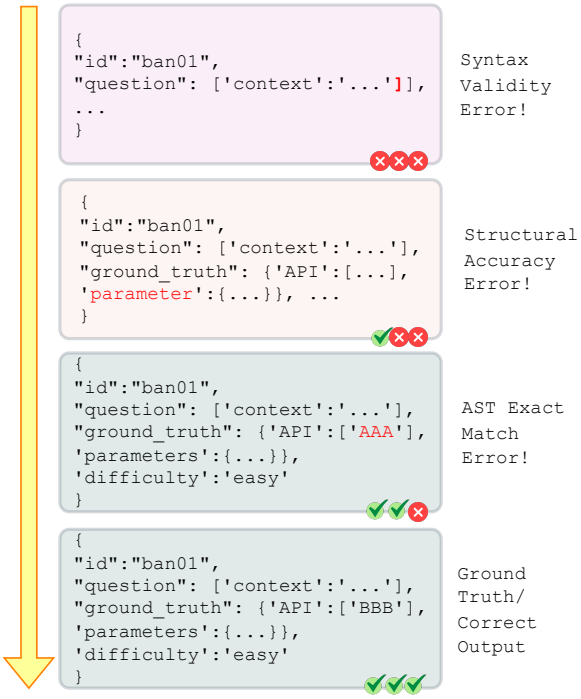

Figure 2: Example of evaluation pipeline

the ground truth, and we exclude them from strict comparisons during evaluation.

Our AST evaluation process is based on three key criteria, examples in Figure 2:

- **Syntax Validity**: Whether the JSON string can be correctly parsed into an object without syntax errors.

- **Structural Accuracy**: Whether the parsed API calls match the ground truth and include the correct parameter names(keys).

- **AST Exact Match**: Whether the entire parsed object, including its structure and content(values), is identical to the ground truth.

As Figure 3 shows, we begin by checking the syntax validity of the generated JSON structure. If syntax errors are detected, we apply a JSON fix prompt to repair the structure or convert alternative formats (such as function-calling syntax) into valid JSON. Once the structure is valid, we assess structural accuracy by comparing the predicted JSON with the ground truth. A structural match is scored as 1. Finally, we convert both the predicted and ground truth JSONs into object trees, comparing each node and leaf. A perfect match across all nodes results in a score of 1 for AST Exact Match.

This multi-step evaluation ensures a thorough assessment of both the accuracy of API function

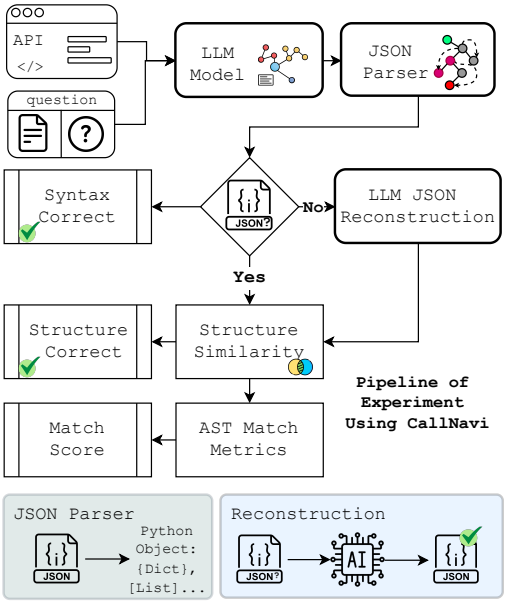

Figure 3: Pipeline of AST Match Score

calls and the structural integrity of the parameters, allowing for a granular analysis of the model's performance.

4.2 LLM-as-a-Judge Evaluation

We also use GPT-4o language models to evaluate whether the generated JSON outputs correspond accurately to the ground truth (Zheng et al., 2023). This approach aims to observe if LLMs can perform such evaluation tasks with high precision. By leveraging an LLM for this purpose, we assess its capability to compare and validate structured data, thereby determining its effectiveness in automating the evaluation process. An example prompt is shown in appendix A.2.

4.3 Stability Score

4.4 Election Stability Score

In chatbot systems, consistent outputs for identical inputs are crucial to ensure reliability and user trust, especially in professional settings. Users expect the same accurate response each time they ask the same question. Inconsistencies can cause confusion, erode confidence, and lead to errors, particularly in critical fields like finance or healthcare.

We propose an **Election Stability Score** to evaluate the consistency of API outputs across multiple runs for the same input. This score mirrors an election process, selecting the majority output as the final answer. To calculate the score, we define:

- N: Total number of outputs (samples).
- F_1: Maximum frequency among all unique outputs (count of the most frequent output).
- F_2: Second maximum frequency (count of the second most frequent output).

The stability score is calculated as:

$$\text{Stability Score} = \frac{F_1 - F_2}{N - F_2}$$

This quantifies the consistency of the model's outputs: If there's a tie for the most frequent output ($F_1 = F_2$), the stability score is set to 0, indicating no consensus; If the most frequent output is unique ($F_1 > F_2$), the score ranges from 0 to 1, reflecting the dominance of the most frequent output.

To ensure reliable comparisons and reduce errors, we preprocess the outputs by removing unnecessary spaces, newlines, and formatting inconsistencies, converting the text to lowercase, and stripping extraneous characters that could cause mismatches. More detailed analysis can be found in the appendix C.

To further evaluate the stability of the model's outputs, we calculate the average Levenshtein distance between the first answer and each subsequent output(Zhang et al., 2017). We normalize Levenshtein distance using the following formula.

$$Score_{Lev} = \frac{1}{n}\sum_{i=1}^{n}(1 - \frac{lev(x_0, x_i)}{Max(len(x_0), len(x_i))})$$

5 Experiments and Benchmark

5.1 Models

For the benchmark, we selected models based on their performance, architecture, and popularity on platforms like the Hugging Face LLM leaderboard[1] and BFCL leader board of function calling. These models include general-purpose and fine-tuned versions optimized for function calling or JSON generation tasks, providing a well-rounded comparison between zero-shot and fine-tuned capabilities. Traditional retrieval models and language models like BART were not included because they are unsuitable for the specific requirements of our task. Retrieval models are typically designed to fetch relevant information based on a query but cannot

[1] https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard

select from multiple APIs or arrange API calls in a logical order, which are critical for complex API calling tasks. Our task requires models not only to retrieve the correct APIs but also to structure them sequentially based on dependencies and logic, a challenge that retrieval-based approaches cannot handle. Hence, we focused on models with capabilities more aligned with dynamic function generation and multi-step reasoning.

5.1.1 General Purpose

Table 2 organizes the selected models into four distinct groups based on size and purpose. The commercial models group includes GPT-4o and GPT-4o-mini from OpenAI and Gemini 1.5 Flash from Google (Reid et al., 2024), which are widely recognized for their state-of-the-art performance in general-purpose language tasks, though their parameter sizes are undisclosed. The middle-large models (with more than 10B parameters) feature LLaMA 3.1 (70B) (Meta-AI, 2024a), Command-R (35B) (Cohere), Gemma2 (27B) (Team et al., 2024), Mistral-Small (22B) (Ai, 2024a), Phi3 (14B) (Abdin et al., 2024), and Mistral-Nemo (12B) (Ai, 2024b). These models are designed for more complex and resource-intensive tasks, making them suitable for complex API function-calling tasks.

In the small models group (with parameters between 5B and 10B), Gemma2 (9B) and LLaMA 3.1 (8B) stand out, providing a balance between efficiency and performance for API-related tasks that require less computational power. We also used the Large Action Model xLAM-fc based on its performance on various benchmarks (Zhang et al., 2024). Finally, the lightweight models group (less than 5B parameters) includes NemoTron-Mini (4B)(Paul, 2024), Phi3.5 (3B)(Trufinescu, 2024), and LLaMA 3.2 (3B)(Meta-AI, 2024b), which are optimized for tasks with stricter resource constraints, offering reasonable performance for lightweight applications. This categorization helps differentiate models based on their potential applications and scalability in real-world tasks.

5.1.2 Fine-tuned Models

We choose 2 fine-tuned Function Calling models for testing which have top performance on the BFCL leaderboard: NexusRaven and Gorilla OpenFunctions v2 (Nexusflow.ai, 2023; Ji et al., 2024). Then we found that Firefunction v2 on the leaderboard cannot accept that long API calling list in the pilot test (Garbacki and Chen, 2024).

Model Name	Origin	Parameters
GPT-4o	OpenAI	N/A
GPT-4o-mini	OpenAI	N/A
Gemini 1.5 Flash	Google	N/A
LLaMA 3.1	Meta AI	70B
Command-R	Command AI	35B
Gemma2	Google	27B
Mistral-Small	Mistral AI	22B
Phi3	Microsoft	14B
Mistral-Nemo	Mistral AI	12B
Gemma2	Google	9B
LLaMA 3.1	Meta AI	8B
xLAM	Salesforce	7B
NemoTron-Mini	NVIDIA	4B
Phi3.5	Microsoft	3B
LLaMA 3.2	Meta AI	3B
NexusRaven	Nexusflow.ai	13B
Gorilla	Berkeley	7B

Table 2: Comparison of General Purpose LLMs.

5.2 Environment

All our local models run with 4-bit Quantization, running on the default Ollama platform settings without any optimization for JSON generation. We do our experiments on NVIDIA-V100 GPU.

5.3 Pipeline

Our evaluation pipeline begins with the creation of prompts based on two templates, as detailed in the appendix. The first template in the appendix A.1 focuses on retrieving the **API calling list**, which corresponds to the "API" list in the ground truth. This prompt instructs the model to identify which API calls should be used and in what order. The second template is designed to generate the full **API calling JSON**, including the parameters for each call.

Once the prompts are generated, we run them through each model to obtain predictions. In the first part of the evaluation, where API calls are generated without parameters, we directly calculate the exact match between the predicted API list and the ground truth and make them called **API Calling Routing**. For the second part, where full JSON outputs are provided, the results are evaluated using the three AST-related scores outlined in Section 4.1: **Syntax Validity**, **Structural Accuracy**, and **AST Exact Match**.

Finally, we employ an LLM-as-a-judge approach, using GPT-4o to calculate a score for comparison, providing a final measure of how well the model's outputs align with the ground truth. This

Category	Models	API Calling Routing Exact Match				Syntax Validity	Structural Accuracy	API Calling with Parameters AST Match					Overall GPT Score
		Easy	Medium	Hard	All			Easy	Medium	Hard	All Avg.	Macro Avg.	
Commercial Models	GPT4o	0.978	0.914	0.611	**0.919**	0.993	**0.887**	0.802	0.638	0.388	**0.711**	**0.609**	**0.913**
	GPT4o mini	0.971	0.930	0.564	0.913	**0.994**	0.869	0.800	0.648	0.364	0.710	0.604	0.908
	Gemini 1.5 Flash	0.973	0.904	0.564	0.908	0.945	0.806	0.728	0.462	0.258	0.604	0.483	0.876
Large General LLMs	LLAMA3.1 70B	0.945	0.835	0.470	0.861	0.967	**0.299**	0.296	0.191	0.094	**0.245**	**0.194**	**0.583**
	CommandR 35B	0.789	0.877	0.529	0.781	0.969	0.189	0.167	0.095	0.047	0.134	0.103	0.400
	Gemma2 27B	0.945	0.877	0.552	**0.882**	0.982	0.226	0.217	0.143	0.070	0.181	0.143	0.476
	Mistral-Small 22B	0.885	0.819	0.494	0.823	**0.986**	0.196	0.201	0.106	0.059	0.160	0.122	0.417
	Phi3 14B	0.050	0.032	0.011	0.041	0.283	0.021	0.019	0.010	0.0	0.015	0.010	0.082
	Mistral-Nemo 12B	0.927	0.808	0.470	0.843	0.842	0.271	0.296	0.127	0.035	0.222	0.153	0.524
Middle LLMs	Gemma2 9B	0.962	0.845	0.506	**0.879**	0.983	0.220	0.241	0.095	0.059	0.182	0.132	0.488
	LLAMA3.1 8B	0.916	0.813	0.552	0.847	0.925	0.207	0.223	0.058	0.059	0.162	0.113	0.422
	xLAM-fc 7B	0.642	0.377	0.188	0.521	**0.990**	**0.271**	0.307	0.117	0.058	**0.229**	**0.161**	**0.554**
Light Models	nemotron-mini 4B	0.644	0.287	0.094	0.488	0.529	0.080	0.067	0.010	0.012	0.047	0.030	0.271
	LLAMA3.2 3B	0.842	0.622	0.400	**0.733**	**0.917**	0.063	0.052	0.021	0.035	**0.042**	**0.036**	**0.353**
	Phi3.5 3B	0.723	0.340	0.188	0.562	0.004	0.0	0.0	0.0	0.0	0.0	0.0	0.002
Fine-Tuned	NexusRaven 13B	0.210	0.148	0.082	0.179	N/A	N/A	0.160	0.074	0.047	0.124	0.094	0.254
	Gorilla v2 7B	0.616	0.005	0.0	**0.387**	N/A	N/A	0.524	0.005	0.0	**0.329**	**0.176**	**0.518**
Asynchronous Generation	*NexusRaven 13B	N/A	N/A	N/A	N/A	N/A	N/A	0.657	0.457	0.188	0.551	0.434	0.795
	*Gorilla v2 7B	N/A	N/A	N/A	N/A	N/A	N/A	0.682	0.005	0.0	0.427	0.229	0.626
	*xLAM-fc 7B	N/A	N/A	N/A	N/A	N/A	0.777	0.714	0.462	0.188	**0.588**	**0.455**	**0.835**

Table 3: Benchmark Results Table, Fine-tuned model with star* means using Asynchronous Generation with GPT4o calling prediction. Macro Avg. means the arithmetic mean of 3 difficulties.

multi-step process ensures comprehensive evaluation across various levels of output complexity.

5.4 Benchmarks Results

The results presented in Table 3 highlight the performance of various models across different aspects of API function calling, including API calling routing accuracy, syntax validity, structural accuracy, and API parameter match through AST evaluation. OpenAI's models, GPT4o and GPT4o mini, consistently outperform the others, particularly in syntax validity (0.993 and 0.994, respectively) and overall GPT score (0.913 and 0.908). Both models also demonstrate strong structural accuracy and API parameter AST match, especially in easier tasks. And Gemini 1.5 Flash follow closely in these metrics.

Among the large general-purpose open LLMs, LLAMA3.1 (70B) performs well in API calling with an exact match score of 0.945 in easy tasks, though its performance drops significantly in harder cases (0.470). It also achieves the second-highest overall GPT score (0.583), largely due to high syntax validity (0.967). However, its structural accuracy and parameter AST match are weaker, with significant drops in harder tasks. But middle-size LLMs show strong potential ability, very close to the larger group performance such as Gemma2(9B) and xLAM(7B).

The other models and fine-tuned models generally struggle across all indicators. For example, CommandR (35B) shows relatively strong performance in medium API calling tasks (0.877) but performs poorly in structural accuracy (0.189) and API parameter AST match (0.134). Similarly, Mistal models show moderate performance, but the smaller models (e.g., Phi3, LLAMA3.2) display particularly low overall GPT scores and poor performance in most tasks.

The results in our table closely align with those from the Berkeley Function Calling Leaderboard[2], which assesses LLMs' performance in API or function-calling tasks. In both evaluations, models like OpenAI's GPT4o stand out for their high syntax validity and overall accuracy, as reflected in our table where GPT4o scores above 0.9 in both categories. This matches the leaderboard's top models, which also excel in AST evaluations and execution accuracy. In contrast, fine-tuned models such as FireFunction V2 in our results show weaker performance in API calling accuracy and AST matching, a trend similarly observed with fine-tuned models like Gorilla OpenFunctions on the Berkeley leaderboard, particularly in more complex API scenarios. Both evaluations emphasize the challenges faced by fine-tuned models in handling complex function scenarios or multi-step API calls, highlighting the need for improvement in these areas.

[2]https://gorilla.cs.berkeley.edu/leaderboard.html

Table 4: Stability Test Results

Model	Size	Election Stability Score	Levenshtein Stability Score
GPT4o	N/A	0.674	0.972
GPT4o mini	N/A	0.855	0.984
Gemini 1.5 Flash	N/A	0.825	0.946
LLAMA3.1	70B	0.407	0.841
LLAMA3.1	8B	0.332	0.740
Mistral-Small	22B	0.208	0.719
Mistral-Nemo	12B	0.365	0.734
CommandR	35B	0.325	0.754
Gemma2	27B	0.609	0.890
Gemma2	9B	0.355	0.864
nemotron-mini	4B	0.013	0.527
LLAMA3.2	3B	0.085	0.613
Phi3.5	3B	0.909	0.637
xLAM-fc	7B	0.782	0.948

RQ1: Which LLM have the best performance for function calling in a reality scenario?

Answer: Based on the results of our benchmark, we can still claim that OpenAI GPT models are the best solution to solve this kind of challenge. But we can see if the test only by calling the API name list, open-source models can have a closer performance to the state-of-the-art.

5.4.1 Stability Test

In our stability experiments, we ran 5 times referring to the previous study (Tian et al., 2023), and the stability results are in Table 4. By these metrics results, we obtain a numerical comparison that reflects the stability differences between outputs.

As mentioned in Section 4.3, a higher Election Stability Score indicates greater absolute consistency in the model's outputs across multiple runs. A high Levenshtein Stability Score means similar between the same input in text generation output. **Commercial models** also perform better.

RQ2: Which way is the best way to evaluate the API function calling ability of LLMs?

Answer: The best way to evaluate the API function calling ability of LLMs is still **AST evaluation** with parameters. However, our **Election Stability Score** provides additional insights into output stability, revealing differences that traditional metrics may overlook.

6 Zero Shot Improvement

6.1 Calling + Parameters Asynchronous Generation

We observed that most general-purpose LLMs perform better when generating only API names(routing) rather than both names and parameters simultaneously (see Table 3). The added complexity of producing detailed parameters alongside API calls can negatively impact overall performance. Additionally, fine-tuned models struggle with long lists of APIs, limiting their effectiveness in scenarios requiring multiple API calls.

To address these challenges, we propose combining the strengths of general-purpose LLMs and fine-tuned models. Specifically, a general LLM selects the relevant APIs based on the input prompt, leveraging its superior understanding in identifying appropriate API calls. These selected APIs are then provided to a fine-tuned/LAM model, which focuses on generating the correct API calls along with the necessary parameters. This sequential process allows the general LLM to efficiently handle API selection, while the fine-tuned model concentrates on accurately producing API calls and parameters within a more manageable context.

As demonstrated in Table 3 (models marked with an asterisk*), this combined approach with **GPT-4o routing** significantly improves performance. Separating the tasks of API selection and parameter generation enhances the models' ability to handle complex API calling tasks more effectively.

6.2 Backward Inference Thinking

To optimize the API selection and calling process, we implemented a novel approach that prompts the LLM to first consider which API should be the final one executed, based on the given task. The model is then guided to confirm whether this final API satisfies all the requirements in the question and whether all necessary information is included in the parameters. If the model identifies missing information, it is prompted to think through which additional API calls could provide the missing data. This iterative process continues until all information required by the question is gathered and all requirements are met. By breaking the task into these smaller, more manageable steps, the LLM can effectively plan and execute a series of API calls that better address the complexity of the scenario. This method has led to a performance improvement of approximately **30%** in hard-level questions API callings in Table 5 test with GPT-4o/mini, where the challenges of multiple selections and nested calls are more prevalent. This optimization enhances the model's ability to handle intricate, real-world API calling tasks more effectively.

Table 5: Backward Thinking performance in High difficulty calling in GPT-4o and GPT-4o-mini.

Model	API Calling Routing		API Calling with Parameters	
	Original	Backward Thinking	Original	Backward Thinking
GPT4o	0.611	0.894	0.388	0.729
GPT4o mini	0.564	0.847	0.364	0.482

RQ3:How to enhance API function calling routing ability for zero/few-shot LLM?

Answer: We have tested 2 different ways to enhance the performance of function calling in our test, both can significantly increase performance in function calling routing and JSON generation.

7 Related Work

The task of function calling in LLMs has recently gained significant attention, particularly in enabling LLMs to interface with external tools or APIs. We reviewed several relevant works, such as ToolLLM, ToolAlpaca, Gorilla, and Granite (Qin et al., 2023b; Tang et al., 2023; Patil et al., 2023; Abdelaziz et al., 2024), along with benchmarks and datasets related to function calling, including BFCL, ToolEyes, API-Bank, ToolACE, and ToolBench (Yan et al., 2024; Ye et al., 2024; Li et al., 2023; Liu et al., 2024; Qin et al., 2023a). However, our benchmark is unique in that it offers LLMs a list of over 100 APIs simultaneously, presenting an unprecedented challenge in API selection and execution.

A pivotal technique in prompting LLMs for complex tasks is the Chain of Thought approach, which guides models to reason step-by-step before arriving at a final decision (Wei et al., 2022). Additionally, agent systems that generate structured outputs, such as MetaGPT and CodeAgent, have inspired us to explore enhancements in structured task generation (Hong et al., 2023; Tang et al., 2024).

Ensuring stability across multiple runs remains a key challenge in function-calling tasks. We reviewed prior work addressing LLM stability, and we drew inspiration from existing stability evaluation methods to refine our approach (Du et al., 2023; Tian et al., 2023). These methods are critical for evaluating the reliability of models in real-world API-driven applications.

8 Disscussion

8.1 Dataset

Our dataset poses significant challenges for most models, particularly in **medium** and **hard** questions. Unlike simpler datasets with limited API options, ours introduces a wider range of APIs and complex scenarios requiring accurate API selection from a giant number of candidates and parameter generation in **multi-step** and **nested contexts**. This complexity is novel for fine-tuned models historically trained on smaller API sets, causing them to struggle with the higher **variability** and **complexity** in our data. This underscores the need for further improvements in fine-tuning models on larger and more diverse API-related tasks.

8.2 Models

In our task of **API calling routing** and **parameter JSON generation**, we observed distinct behaviours across models. **GPT-4o** consistently demonstrated advanced capabilities in both **routing** and **parameter generation**, reinforcing its position as a top choice for complex language tasks. In contrast, open-source LLMs like **LLaMA 3.1** and **Gemma2** performed competently in API routing but struggled with the more intricate task of parameter generation. These models are promising alternatives when the focus is on selecting the correct APIs, offering a cost-effective solution without greatly compromising routing accuracy.

However, **smaller models** under **10B parameters**, such as **Gemma2 (8B)** and **LLaMA 3.2 (3B)**, tend to exhibit instability in complex API tasks, leading to inconsistent or incomplete outputs. This makes them less reliable for tasks requiring high accuracy and multiple API steps.

Overall, while progress has been made, understanding and generating correct API calls in hard, complex scenarios remains a significant challenge. None of the models, including **GPT-4o**, deliver perfect results, indicating we are still far from fully mastering intricate API calling tasks.

9 Conclusion

In conclusion, our paper tackles the challenges of API function calling routing and parameters generation in LLMs by introducing a **novel dataset**, offering a comprehensive **benchmark**. We contribute a unique dataset with over **500 APIs** and **700 questions**, along with benchmarks for both general-purpose and fine-tuned models. Additionally, we present **two optimization strategies** that enhance multi-step API selection and generation via asynchronous generation and backward thinking, advancing research in LLM-driven function calling and its practical applications.

Limitations

While our work presents important advancements in API function calling routing and parameters generation for LLMs, there are several limitations to acknowledge. First, the scenarios we used in our dataset are simplified versions of real-world situations and may not fully capture the complexities that arise in actual applications. This could limit the practical applicability of our results in highly dynamic or unpredictable environments. Additionally, our team has limited expertise in designing datasets across the many diverse domains included in this study, which may have led to oversights in representing certain domain-specific challenges accurately.

Another limitation is related to the performance of the models we tested. While we selected well-regarded LLMs for benchmarking, they may not have performed at their best due to factors such as suboptimal configurations or fine-tuning for different tasks. Lastly, we acknowledge the absence of a mock server in our experimental setup, which could have enhanced the realism of the API calling process. Incorporating a mock server to simulate real-time API interactions would provide a more accurate assessment of how these models handle real-world API calling tasks. Future iterations of this work should address these limitations to provide more comprehensive and applicable insights.

Ethic Statment

In the development of our dataset, we ensured that no private or sensitive information was used, and all data was mock information, generated in compliance with GDPR. This guarantees that no data leakage or privacy concerns are associated with our work. Additionally, we made efforts to minimize the environmental impact of our research by optimizing model training and evaluation processes, thus reducing the carbon footprint associated with our benchmarking tasks. Where possible, we utilized energy-efficient infrastructure to further support this initiative. Our commitment to ethical practices extends across all aspects of this research, ensuring data privacy and environmental responsibility.

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*.

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. 2024. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*.

Mistral Ai. 2024a. AI in abundance.

Mistral Ai. 2024b. Mistral NeMo.

Cohere. The Command R model — Cohere.

Mengnan Du, Fengxiang He, Na Zou, Dacheng Tao, and Xia Hu. 2023. Shortcut learning of large language models in natural language understanding. *Preprint*, arXiv:2208.11857.

Pawel Garbacki and Benny Chen. 2024. Firefunction-v2: Function calling capability on par with gpt4o at 2.5x the speed and 10% of the cost.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

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

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*.

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

Meta-AI. 2024a. Introducing Llama 3.1: our most capable models to date.

Meta-AI. 2024b. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models.

Nexusflow.ai. 2023. Nexusraven-v2: Surpassing gpt-4 for zero-shot function calling.

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*.

Jason Paul. 2024. NVIDIA announces first digital Human Technologies On-Device Small Language Model, improving conversation for game characters | NVIDIA blog.

Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, Yi Ren Fung, Yusheng Su, Huadong Wang, Cheng Qian, Runchu Tian, Kunlun Zhu, Shihao Liang, Xingyu Shen, Bokai Xu, Zhen Zhang, Yining Ye, Bowen Li, Ziwei Tang, Jing Yi, Yuzhang Zhu, Zhenning Dai, Lan Yan, Xin Cong, Yaxi Lu, Weilin Zhao, Yuxiang Huang, Junxi Yan, Xu Han, Xian Sun, Dahai Li, Jason Phang, Cheng Yang, Tongshuang Wu, Heng Ji, Zhiyuan Liu, and Maosong Sun. 2023a. Tool learning with foundation models. *Preprint*, arXiv:2304.08354.

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

Machel Reid, Nikolay Savinov, Denis Teplyashin, Dmitry Lepikhin, Timothy Lillicrap, Jean-baptiste Alayrac, Radu Soricut, Angeliki Lazaridou, Orhan Firat, Julian Schrittwieser, et al. 2024. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *arXiv preprint arXiv:2403.05530*.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.

Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawende F. Bissyande. 2024. Codeagent: Autonomous communicative agents for code review. *Preprint*, arXiv:2402.02172.

Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. 2024. Gemma 2: Improving open language models at a practical size. *arXiv preprint arXiv:2408.00118*.

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*.

Adina Trufinescu. 2024. Discover the new Multi-Lingual, High-Quality PHI-3.5 SLMS.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Fanjia Yan, Huanzhi Mao, Charlie Cheng-Jie Ji, Tianjun Zhang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. 2024. Berkeley function calling leaderboard.

Junjie Ye, Guanyu Li, Songyang Gao, Caishuang Huang, Yilong Wu, Sixian Li, Xiaoran Fan, Shihan Dou, Qi Zhang, Tao Gui, et al. 2024. Tooleyes: Fine-grained evaluation for tool learning capabilities of large language models in real-world scenarios. *arXiv preprint arXiv:2401.00741*.

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

Shengnan Zhang, Yan Hu, and Guangrong Bian. 2017. Research on string similarity algorithm based on levenshtein distance. In *2017 IEEE 2nd Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, pages 2247–2251. IEEE.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36:46595–46623.

A Prompts

A.1 Model prediction

Prompt: Give the API list with descriptions below, then give the question in a chatbot, please give me the correct API that should be called.
=======API list start=======
{JSON API list}
=======API list end=======
=======Question start=======
{Question}
=======Question end=======
Given the user question, and the APIs, classify and give a correct API name to call.
The answer should be formatted in only one line, and only API names in brackets look like "['getCustomerDetails', 'depositFunds']" or "['getCustomerDetails']". NO explanation and NO parameters in the answer!"

Prompt: Give the API list with descriptions below, then give the question in a chatbot, please give me the correct API that should be called.
=======API list start=======
{JSON API list}
=======API list end=======
=======Question start=======
{Questions}
=======Question end=======

Given the user question, and the APIs, classify and give a correct API name and parameters to call. The answer should be formatted including API names and parameters in JSON style, which looks like:
{'API': ['getCustomerDetails', 'depositFunds'],
'parameters':[{"parameter1ForCall1": "***" },{"parameter1ForCall2": "***", "parameter2ForCall2": "***"}]}
If we cannot get some parameter information from the question, set these parameters to "$$$".
NO explanation/notes in the answer!

The first prompt is designed for API calling list generating only in the experiment 'API Calling Exact Match' from Table 3. The second prompt is related to generating full JSON output like ground truth in Section 3.2.

A.2 LLM-as-judge

We use the following prompt to generate GPT Score of evaluation in Section 4.2

Prompt: You will be given a predicted API calling JSON and ground truth couple. Your task is to provide a 'total rating' scoring on how similar the predicted API calling JSON is to ground truth JSON. Give your answer as a float on a scale of 0 to 1 with 3 decimals, where 0 means that the predicted JSON is not related to ground truth, and 1.0 means that the answer is completely the same as ground truth.
You should consider if the API calling list is the same or not, and whether all parameters match or not.
Only give a score in [[]], looks like [[1.00]]:
======Begin predict JSON======
{predict JSON}
======End of predict JSON======
=====Begin of ground truth JSON=====
{ground truth JSON}
=====End of ground truth JSON=====
Response:

A.3 Backward Thinking

Prompt: Give the API list of a chatbot system with descriptions below, then give the question in chatbot, please first give me the one API that can answer the final task in the question.
=======API list start=======
{JSON API list}
=======API list end=======
=======Question start=======
{Questions}
=======Question end=======
Given the user question, and the APIs, classify and give a correct last one task API name and parameters to call.

The answer should be formatted into the following JSON format:{format JSON}

We will solve the question in the next steps.
NO explanation in the answer! "'

After initial the session, give the following prompt to get a full API calling JSON.

> **Prompt:** Then please based on the answer before, try to find APIs to fill in the missing information needed to complete the final API call list from the previous answer.
>
> Please also check if the API list has solved all sub-questions in the question. If not, please add other APIs and parameters related to the question in the correct order.
>
> The answer should be formatted in the JSON format used in the previous talk.
>
> Check if we can get all input parameters from the questions or not, if we can't please don't put <!stop> after the API list.
>
> If you think all the information is completed to solve the question, please write a <!stop> after the full API list.
>
> NO explanation in the answer! Only give API calling JSON!

B Case Study

B.1 Hallucination

In one of our test cases, the **Phi3:14b** model produced an incorrect API function call in response to a baggage tracking scenario. The predicted output was as follows:

```
{
    "API": ["getLostBaggageReport
       ", "updateBaggageStatus"],
    "parameters": [{"baggageId":
       "BAG123"}, {}]
}
```

However, the ground truth was:

```
{
    "API": ["getBaggageStatus"],
    "parameters": [{"baggageId":
       "BAG123"}]
}
```

In this case, the model hallucinated two API calls, `"getLostBaggageReport"` and `"updateBaggageStatus"`, which were not part of the provided API list. This hallucination led the model to predict incorrect API calls, deviating from the expected function `"getBaggageStatus"`. Although the model correctly captured the parameter `baggageId: "BAG123"`, it introduced an unnecessary second parameter block as an empty dictionary, further reducing the accuracy of the output.

This example highlights a common issue with current large language models in complex tasks: their tendency to hallucinate irrelevant API calls when uncertain. Such behavior emphasizes the need for improved mechanisms to ensure more accurate API function routing and parameter generation in these models.

B.2 JSON Generation

In another example, the **Mistral-Nemo** model generated an incorrect output, which included unwanted notes in the result, rendering it invalid as a JSON. The predicted output was:

```
{
    "API": ["
       getCustomerCreditCards"],
    "parameters": [{"customerID":
        "123456"}]
}
(Assuming that ATM cards are
   considered credit cards for
   this specific API)
```

The ground truth, however, was:

```
{
    "API": ["getATMCardList"],
    "parameters": [{"accountID":
       "123456"}]
}
```

In this case, the model incorrectly generated an API call for `"getCustomerCreditCards"` instead of the correct API `"getATMCardList"`. Additionally, the model included an unwanted note—`"(Assuming that ATM cards are considered credit cards for this specific API)"`—which made the output non-compliant with JSON formatting, as this additional text was outside the structure of the JSON object.

This example illustrates the challenge of maintaining output fidelity in models when they generate explanations or assumptions within the response, which should be avoided in strict JSON-

formatted outputs. Such behavior disrupts the automation of API calls and highlights the need for better prompt engineering to ensure models only return valid JSON results without extraneous content.

C Stability Research

The **stability score** is calculated in Section 4.3. To give a clear example of calculating the stability of the model's outputs, we analyze the frequency distribution of the results obtained from 5 times runs. Let's review the variable settings:

- N be the total number of outputs (samples).
- F_1 be the **maximum frequency** of any unique output (the most frequent output).
- F_2 be the **second maximum frequency** (the frequency of the second most frequent output).

Explanation:

- **Tie Situations** ($F_1 = F_2$): When the maximum frequency is equal to the second maximum frequency, it indicates a tie for the most frequent output. In such cases, the stability score is set to 0 to reflect the lack of a clear majority or consensus.
- **No Tie Situations** ($F_1 > F_2$): The numerator $F_1 - F_2$ measures the dominance of the most frequent output over the second most frequent one. The denominator $N - F_2$ normalizes this difference relative to the total number of outputs excluding those of the second most frequent output. The resulting score ranges between 0 and 1, where higher values indicate greater stability.

Examples:

- **All Outputs Identical:**
  - **Results**: All outputs are the same (e.g., $['A','A','A','A','A']$).
  - $F_1 = N$, $F_2 = 0$ (since there's only one unique output).
  - **Stability Score**:

$$\text{Stability Score} = \frac{N-0}{N-0} = \frac{N}{N} = 1$$

    Indicates perfect stability.

- **Tie Situation (e.g., 2 vs 2 vs 1):**
  - **Results**: Two outputs occur twice, and one occurs once (e.g., $['A','A','B','B','C']$).
  - $F_1 = 2$, $F_2 = 2$ (tie between 'A' and 'B').
  - **Stability Score**:

$$\text{Stability Score} = 0$$

    Reflects the lack of consensus due to the tie.

- **Minority Advantage (e.g., 2 vs 1 vs 1 vs 1):**
  - **Results**: One output occurs two times, another occurs once each (e.g., $['A','A','B','C','D']$).
  - $F_1 = 2$, $F_2 = 1$.
  - **Stability Score**:

$$\text{Stability Score} = \frac{2-1}{5-1} = \frac{1}{3} \approx 0.25$$

    Indicates a little stability.

- **Partial Agreement(Strong Opposition) (e.g., 3 vs 2):**
  - **Results**: One output occurs three times, another occurs twice (e.g., $['A','A','A','B','B']$).
  - $F_1 = 3$, $F_2 = 2$.
  - **Stability Score**:

$$\text{Stability Score} = \frac{3-2}{5-2} = \frac{1}{3} \approx 0.333$$

    Indicates moderate stability.

- **Partial Agreement(Weak Opposition) (e.g., 3 vs 1 vs 1):**
  - **Results**: One output occurs three times, another occurs once each (e.g., $['A','A','A','B','C']$).
  - $F_1 = 3$, $F_2 = 1$.
  - **Stability Score**:

$$\text{Stability Score} = \frac{3-1}{5-1} = \frac{1}{3} \approx 0.5$$

    Indicates higher moderate stability.

- **High Majority (e.g., 4 vs 1):**

- **Results**: One output occurs four times, another occurs once (e.g., $['A', 'A', 'A', 'A', 'B']$).
- $F_1 = 4$, $F_2 = 1$.
- **Stability Score**:

$$\text{Stability Score} = \frac{4-1}{5-1} = \frac{3}{4} = 0.75$$

Indicates strong stability.

- **All Outputs Unique:**
  - **Results**: All outputs are different (e.g., $['A', 'B', 'C', 'D', 'E']$).
  - $F_1 = F_2 = 1$.
  - **Stability Score**:

$$\text{Stability Score} = 0$$

Reflects complete instability due to lack of any consensus.

Interpretation:

- **Stability Score of 1**: Perfect stability; all outputs are identical.
- **Stability Score of 0**: No stability; either all outputs are unique or there's a tie for the most frequent output.
- **Stability Scores Between 0 and 1**: Partial stability; higher scores indicate greater agreement among outputs.

Usage:

This stability score provides a quantitative measure of the consistency of the model's outputs across multiple runs. It accounts for both the dominance of the most frequent output and the impact of significant minority outputs, offering a nuanced assessment of model stability.