CallNavi, A Challenge and Empirical Study on LLM Function Calling and Routing

Yewei Song[1], Xunzhu Tang[1], Cedric Lothritz[2]
Saad Ezzini[3,5], Jacques Klein[1], Tegawendé F. Bissyandé[1]
Andrey Boytsov[4], Ulrick Ble [4], Anne Goujon[4]
[1]University of Luxembourg, [2]Luxembourg Institute of Science and Technology,
[3]Department of Information and Computer Science, KFUPM, [4]BGL BNP PARIBAS, [5]Interdisciplinary Research Center for Intelligent Manufacturing and Robotics, KFUPM,

Abstract

API-driven chatbot systems are increasingly integral to software engineering applications, yet their effectiveness hinges on accurately generating and executing API calls. This is particularly challenging in scenarios requiring multi-step interactions with complex parameterization and nested API dependencies. Addressing these challenges, this work contributes to the evaluation and assessment of AI-based software development through three key advancements: (1) the introduction of a novel dataset specifically designed for benchmarking API function selection, parameter generation, and nested API execution; (2) an empirical evaluation of state-of-the-art language models, analyzing their performance across varying task complexities in API function generation and parameter accuracy; and (3) a hybrid approach to API routing, combining general-purpose large language models for API selection with fine-tuned models and prompt engineering for parameter generation. These innovations significantly improve API execution in chatbot systems, offering practical methodologies for enhancing software design, testing, and operational workflows in real-world software engineering contexts.

CCS Concepts

• **Software and its engineering** → **Empirical software validation**; *Software design engineering*; • **Computing methodologies** → *Information extraction.*

Keywords

Function Calling, Large Language Models, Chatbot, Benchmark

ACM Reference Format:
Yewei Song[1], Xunzhu Tang[1], Cedric Lothritz[2], Saad Ezzini[3,5], Jacques Klein[1], Tegawendé F. Bissyandé[1], Andrey Boytsov[4], Ulrick Ble [4], Anne Goujon[4], [1]University of Luxembourg, [2]Luxembourg Institute of Science and Technology,, [3]Department of Information and Computer Science, KFUPM, [4]BGL BNP PARIBAS, [5]Interdisciplinary Research Center for Intelligent Manufacturing and Robotics, KFUPM, . 2025. CallNavi, A Challenge and Empirical Study on LLM Function Calling and Routing. In *Proceedings of The 29th International Conference on Evaluation and Assessment in Software Engineering (EASE 2025).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXXX.XXXXXXX

EASE 2025, Istanbul, Türkiye

ACM ISBN 978-1-4503-XXXX-X/2025/XX
https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Modern conversational AI systems, such as chatbots, rely on accurate API calling to enable effective user interactions, as shown in Figure 1. Beyond generating simple API calls, models must handle complex scenarios involving selecting the correct API from extensive lists, orchestrating multiple sequential calls, and managing nested API interactions. While progress has been made in generating syntactically correct **single** API calls, there is limited focus on generating **sequences** of API calls with logical dependencies in long description context, a crucial requirement for real-world applications.

Large Language Models (LLMs), such as GPT-4 [2] and Llama [35], have demonstrated impressive capabilities in various natural language processing tasks. These models excel at generating coherent and contextually relevant responses, but their ability to produce structured outputs, such as API calls, program code, or other machine-readable formats, remains a challenging frontier. Structured output generation requires adherence to predefined syntactic and semantic rules, making it more constrained than generating free-form text[20].

Recent advancements have explored structured output generation in applications such as code generation [7], table completion [15], and multi-turn dialogue [6]. Tools like CodeX [7] and AlphaCode [19] focus on generating functionally valid code, while methods like chain-of-thought prompting [37] and tool-augmented reasoning frameworks [28] enhance reasoning in complex tasks. These methods highlight the potential of LLMs for tasks requiring step-by-step reasoning and structured output generation.

Early studies, such as BotBase [41], explored translating natural language into API calls, laying the groundwork for automating tool use. More recent benchmarks, including API-Bank [18], ToolEyes [40], and BFCL [38], evaluate LLMs on API execution. However, these datasets often operate with small API candidate pools or lack scenarios involving nested or interdependent API calls. For example, API-Bank assesses tool-augmented models but limits API candidates to fewer than five per task. Similarly, ToolBench [21] and ToolEyes evaluate multi-tool scenarios but do not support tasks requiring highly interdependent API calls.

To address these gaps, we propose CallNavi, a novel benchmark designed to evaluate LLMs on:

- Selecting APIs from an unfiltered list of over 100 candidates;
- Executing multiple sequential API calls;
- Handling nested API interactions.

CallNavi introduces real-world complexity by simulating unfiltered API selection and combining generation with routing tasks in a long context input. It categorizes questions into easy, medium, and hard levels, enabling a granular evaluation of model capabilities across varying task complexities. Additionally, we propose new metrics, including a stability score, to measure prediction consistency across multiple runs.

We benchmark 18 LLMs, encompassing commercial, general-purpose, and fine-tuned models, on CallNavi. Our findings provide insights into the strengths and limitations of current models, laying a foundation for advancing API selection and function calling.

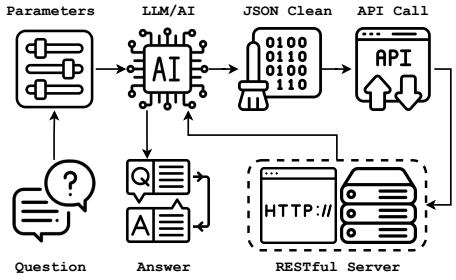

Figure 1: Example of API Calling pipeline via LLM

2 Related Work

Generating and executing accurate API calls is crucial to integrating LLM into real-world conversation applications. Existing benchmarks, such as API-Bank [18], ToolEyes [40], and ToolBench [21], evaluate API selection and execution capabilities but often rely on prefiltered API candidate pools, lack nested API tasks, or focus on narrow domain coverage. In contrast, CallNavi introduces unfiltered API selection with over 100 candidates, multi-call tasks, and nested API scenarios across 10 diverse domains. Table 1 highlights these distinctions, demonstrating how CallNavi addresses limitations in existing benchmarks by introducing realistic complexity and structured difficulty levels.

Structured output generation, a critical capability for API function calling, has seen significant advancements. MetaGPT [16] and CodeAgent [32] emphasize task decomposition and multi-step reasoning, improving performance in complex workflows. Techniques like constrained generation [5] and grammar-aware Seq2Seq models [9] improve structured output reliability, aligning with CallNavi's focus on evaluating structured reasoning and accuracy.

Stability in LLM predictions is another vital area of research. Although traditional metrics such as *freq@topk* assess prediction reliability, they fail to capture consistency across multiple runs fully. Inspired by prior stability-focused studies [10, 34], CallNavi introduces a stability score to quantify prediction consistency, complementing traditional metrics like AST match and exact match.

From a software engineering perspective, function-calling tasks align with modularity and abstraction principles, emphasizing decomposition into manageable sub-tasks. Early works, such as BotBase [41], synthesized API calls from natural language, laying the groundwork for modern tools like Gorilla [17], ToolLLM [28], and ToolAlpaca [31]. Recent efforts like StableToolBench [13] and τ-bench [39] highlight challenges in tool learning and real-world tool-agent-user interactions.

API recommendation systems, such as those explored by Peng et al. [27], provide insights into ranking and selecting APIs. These systems complement CallNavi, which emphasizes multi-step workflows requiring careful API selection. Similarly, abstract syntax networks [29] and benchmarks like BigCodeBench [46] advance structured code generation and semantic parsing, aligning with CallNavi's emphasis on reasoning and logical consistency in nested tasks.

In summary, while prior work has laid a strong foundation for API function calling, CallNavi advances the field by addressing critical gaps such as unfiltered API selection, nested tasks, and stability evaluation. These contributions provide a robust framework for benchmarking LLMs in realistic and complex scenarios.

Table 1: Comparison of CallNavi with existing API function-calling benchmarks test set.

Benchmark	Domains	Questions	Max API Candidates	Multi-Call	Nested
CallNavi	10	729	115	Yes	Yes
API-Bank	8	753	<5	Yes	Yes
ToolEyes	41	382	<20	Yes	No
ToolBench	8	795	32	Yes	No
BFCL (API)	N/A	70	<5	Yes	No

3 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?
- **Optimization** How to enhance API function calling ability for zero/few-shot LLM?

4 CallNavi Dataset

To create our dataset, we adopted a hybrid approach that combines automated generation with manual validation and construction to ensure high-quality and diverse data across different levels of difficulty. The process consisted of the following steps:

Initial API Function Generation. Using GPT-4o, we generated API function names, descriptions, parameters, and return values based on a variety of scenario descriptions spanning multiple domains. These domains were selected to reflect realistic use cases across 10 common chatbot application areas, as described in Table 2. This ensures that the dataset evaluates CallNavi in scenarios requiring advanced task routing, contextualization, and regulatory compliance.

Validation and Refinement. All generated API functions were manually reviewed for accuracy, consistency, and relevance. i.e.:

- Parameters were checked to ensure they aligned with real-world API design conventions.
- Ambiguities or redundancies in function descriptions were resolved.
- Naming conventions for parameters and return values were standardized to ensure consistency across the dataset.

Generation of Easy Questions. For the easy subset, we used GPT-4o to generate questions related to API usage. These questions were subsequently validated to ensure:

- Relevance to the provided APIs,
- Syntactic and semantic correctness, and
- Coverage of straightforward, single API usage scenarios.

Manual Construction of Medium and Hard Questions. Medium and hard questions were manually crafted to reflect increasingly complex API calling scenarios. The criteria and considerations for these levels were as follows:

- **Medium Questions:** Focused on multi-step tasks requiring the use of multiple APIs in sequence. These tasks test the model's ability to identify dependencies between API calls while maintaining logical flow.
- **Hard Questions:** Designed to address edge cases, ambiguous queries, and nested API calls requiring advanced reasoning. Scenarios simulate real-world challenges, such as incomplete user inputs or conflicting requirements.

Quality Control. The dataset underwent a multi-stage quality assurance process to ensure its reliability:

- Each generated instance was cross-checked by multiple annotators for correctness and consistency.
- For manually written instances, authors verified adherence to the design criteria.
- Errors, ambiguities, and inconsistencies were flagged and resolved iteratively.

Summary. The `CallNavi` dataset combines automation with human oversight, resulting in a benchmark that is both realistic and challenging. By spanning easy, medium, and hard tasks across diverse real-world domains, as outlined in Table 2, the dataset evaluates LLM capabilities in scenarios requiring robust task routing, contextual understanding, and API management.

The first part of the metadata is a long JSON file with the API name, description, and parameters in the following format.

```
{
    "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"
},
...
```

Table 2: CallNavi dataset domains, questions and difficulties statistics table.

Domain	API Functions	Questions	Difficulty			Max Input Tokens
			Easy	Medium	Hard	
Banking	91	115	70	28	17	6517
Shopping	81	65	41	17	7	5195
Logistics	46	65	40	17	8	3434
Aviation	48	80	44	24	12	3461
Healthcare	20	47	31	10	6	1788
Public Services	82	85	50	27	8	6249
Human Resources	20	35	21	13	1	1863
Hotel Industry	49	65	40	19	6	3811
Insurance	42	60	40	11	9	3452
Telecommunications	100	112	79	22	11	6374
Overall	579	729	456	188	85	6517

4.1 Dataset

[1] The *CallNavi* dataset evaluates LLMs' task routing and API calling capabilities across multiple domains. As shown in Table 2, 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 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 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.

[1] https://github.com/Etamin/CallNavi

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.

5 Metrics and Evaluation

5.1 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 4 [38]. 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 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, is identical to the ground truth.

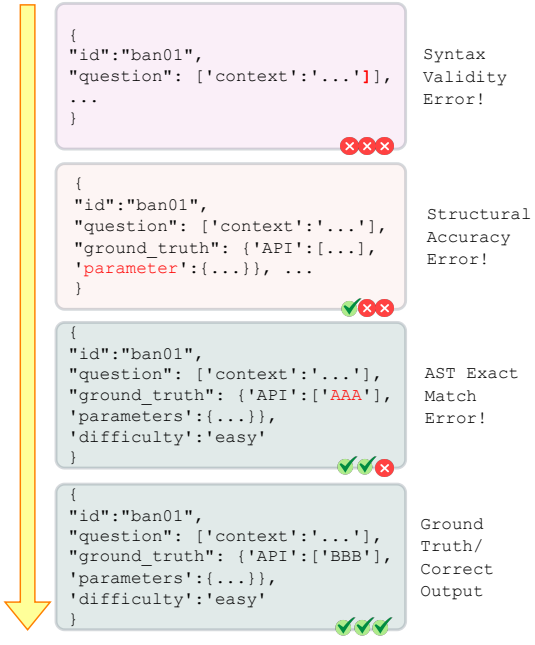

Figure 2: Example of Evaluation Pipeline

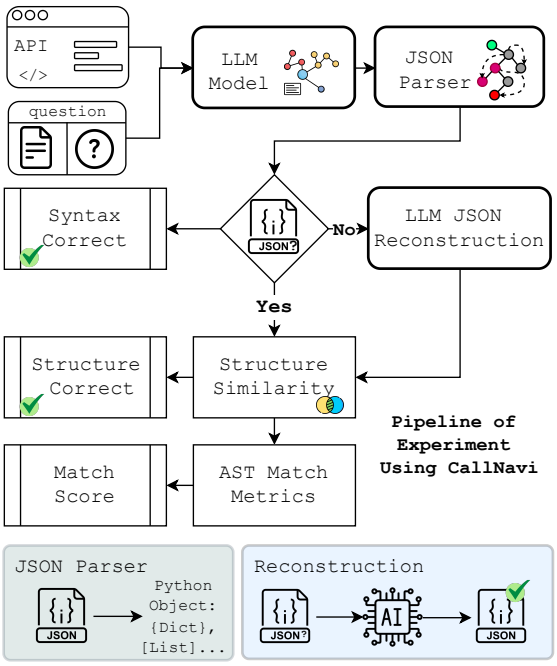

Figure 3: Pipeline of AST Match Score

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 multistep evaluation ensures a thorough assessment of the accuracy of API function calls and the structural integrity of the parameters, allowing for a granular analysis of the performance.

5.2 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 [45]. This approach aims to observe if LLMs can perform such evaluation tasks with high precision. Using an LLM for this purpose, we assess its ability to compare and validate structured data, thereby determining its effectiveness in automating the evaluation process.

5.3 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.

While 'freq@topk' is often used to evaluate the performance of model stability, it does not capture the stability in LLM output. For example, if a model produces the sequence "AABBC" across multiple runs, 'freq@topk' might assign a high score of 0.4 because the most frequent token ("A") appears 40% of the time. However, this sequence is unstable as **no single output** consistently dominates. In contrast, our stability score focuses on the dominance of the most frequent output, offering a better measure of a model's reliability in structured tasks.

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. The stability score is set to 0 to reflect neither majority nor consensus in such cases.
- **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 from 0 to 1; 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**:
$$\text{Stability Score} = \frac{2 - 1}{5 - 1} = \frac{1}{4} \approx 0.25$$
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**:
$$\text{Stability Score} = \frac{3 - 1}{5 - 1} = \frac{1}{2} \approx 0.5$$
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 unique outputs (e.g., $['A', 'B', 'C', 'D', 'E']$).
  - $F_1 = F_2 = 1$.
  - **Stability Score**:
$$\text{Stability Score} = 0$$
Complete instability due to a lack of consensus.

Interpretation:

- **Stability Score of 1**: Perfect stability; all outputs are same.
- **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.

To further evaluate the stability of the model's outputs, we calculate the average Levenshtein distance between the first answer and each subsequent output[43]. We normalize Levenshtein distance using the following formula.

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

6 Experiments and Benchmark

6.1 Models

To evaluate the benchmark, we selected models based on their performance, architecture, and relevance to function-calling tasks. The selection criteria focused on general-purpose and fine-tuned models optimized for function calling or JSON generation, ensuring a well-rounded comparison between zero-shot and fine-tuned capabilities. Models like BART and traditional retrieval-based approaches were excluded as they lack the ability to select APIs from extensive lists, which is critical for the complexity of this task.

Table 3 organizes the selected models into four groups: commercial models, medium-large models (10B + parameters), small models (5B-10B parameters) and light models (with parameters below 5B).

Model Name	Origin	Size	Context Limits
GPT-4o	OpenAI [2]	N/A	128K
GPT-4o-mini	OpenAI [2]	N/A	128K
Gemini 1.5 Flash	Google [30]	N/A	1M
LLaMA 3.1	Meta AI [22]	70B	128K
Command-R	Command AI [8]	35B	128K
Gemma2	Google [33]	27B	8K
Mistral-Small	Mistral AI [3]	22B	128K
Phi3	Microsoft [1]	14B	128K
Mistral-Nemo	Mistral AI [4]	12B	1M
Gemma2	Google [33]	9B	8K
LLaMA 3.1	Meta AI [22]	8B	128K
xLAM	Salesforce [42]	7B	4K
DeepSeek R1	DeepSeek-AI [12]	7B	128K
NemoTron-Mini	NVIDIA [26]	4B	4K
Phi3.5	Microsoft [36]	3B	128K
LLaMA 3.2	Meta AI [23]	3B	128K
NexusRaven	Nexusflow.ai [24]	13B	16K
Gorilla	Berkeley [25]	7B	4K

Table 3: Comparison of General Purpose LLMs.

We chose 2 fine-tuned Function Calling models for testing, which have top performance on the BFCL leaderboard: NexusRaven and Gorilla OpenFunctions v2. Then we found that some models cannot input long lists, e.g. Firefunction v2 [11].

6.2 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.

6.3 Pipeline

Our evaluation pipeline begins with the creation of prompts based on two templates. The first template 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 5.1: **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 multi-step process ensures comprehensive evaluation across various levels of output complexity.

6.4 Benchmarks Results

The results presented in Table 4 highlight the performance of various models in 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. Gemini 1.5 Flash follows these metrics closely.

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.

Our analysis demonstrates that the Pearson correlation between the "GPT Score" and the "All Avg." column in the "Parameter AST Match" section is **0.934**, with a p-value of **4.40e-08**. This indicates a very strong positive correlation, suggesting that higher GPT scores are closely associated with better average AST match performance. The results in our table are closely aligned with those of the Berkeley

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**
	DeepSeek R1 7B	0.250	0.271	0.082	0.235	0.902	0.117	0.129	0.042	0.047	0.097	0.073	0.289
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**

Table 4: Benchmark Results Table. Macro Avg. means the arithmetic mean of 3 difficulties.

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.

Answer to RQ1: 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.

6.4.1 Stability Test. In our stability experiments, we ran 5 times referring to the previous study [34], and the stability results are in Table 5. By these metrics results, we obtain a numerical comparison that reflects the stability differences between outputs.

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.

As mentioned in Section 5.3, 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 in Table 5 below.

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

Answer to RQ2: 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.

Table 5: 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
DeepSeek R1	7B	0.058	0.501

7 Zero Shot Improvement

7.1 Calling + Parameters 2 Steps Generation

We observed that most general-purpose LLMs perform better when generating only API names(routing) rather than both names and

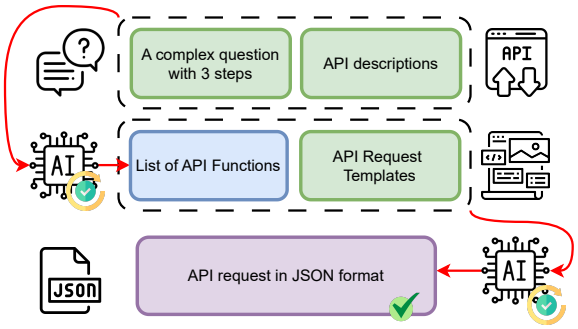

Figure 4: 2-Steps Generation Pipeline

parameters simultaneously (see Table 4). 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 shown in Figure4. 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 6, 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.

Table 6: 2 Steps Generation results for LLMs

	Models	easy	medium	hard	overall
Fine-Tuned	NexusRaven13B	0.657	0.457	0.188	0.551
Model w/	Gorilla v27B	0.682	0.005	0.000	0.427
GPT routing	xLAM-fc-7B	0.714	0.462	0.188	0.588
	Gemma2:27b	0.633	0.617	0.341	0.595
	Gemma2	0.723	0.457	0.164	0.589
General	llama3.1	0.714	0.457	0.164	0.584
Large	mistral-small	0.728	0.436	0.294	0.602
Language	mistral-nemo	0.712	0.308	0.141	0.541
Models w/	phi3:14b	0.019	0.005	0.011	0.015
GPT routing	command-r	0.633	0.547	0.223	0.563
	llama3.2	0.462	0.297	0.082	0.375
	nemotron-mini	0.208	0.01	0.000	0.133
LLM w/	Gemma2:27b	0.598	0.59	0.341	0.566
itself as	Mistral-small	0.684	0.382	0.235	0.554
as router	Command-r	0.621	0.505	0.247	0.547

7.2 Backward Inference Thinking

To optimize the API selection and calling process, we implement a **Backward Thinking** approach, inspired by CauseJudger [14] and Reverse Chain [44], as illustrated in Figure 5. This approach enables the model to construct a sequence of API calls more systematically by working backwards from the final goal rather than following a purely forward selection strategy.

Table 7: 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

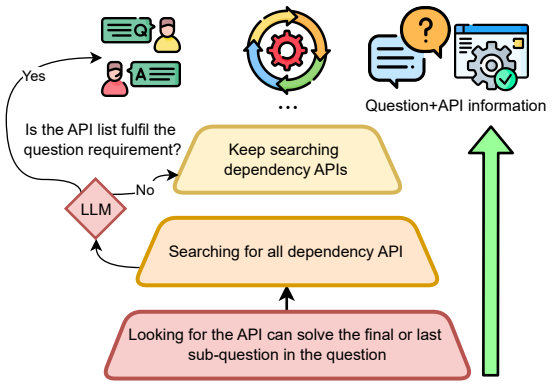

Figure 5: Backward Thinking Pipeline

The process follows these steps:

(1) **Identifying the Final API Call**: The model first determines the ultimate API needed to answer the user's query. This API must provide the final required information or action.
(2) **Checking Parameter Completeness**: The model verifies whether all required parameters for the final API are available. If any essential information is missing, the model does not proceed with execution but instead considers the necessary steps to obtain the missing data.
(3) **Determining Supporting API Calls**: If missing parameters are identified, the model searches for additional APIs that can retrieve the necessary data. These supporting API calls are planned in reverse order, ensuring that the final API call has all the required inputs.
(4) **Iterative Refinement**: This process continues iteratively. Each newly identified API is analyzed for its own dependencies, ensuring that all required information is recursively retrieved before execution.

By breaking the task into smaller, dependency-aware steps, this method allows the model to effectively plan and execute multi-step API calls, improving accuracy in complex scenarios. As shown in Table 7, this approach yields a 30% improvement in hard-level API calling tasks. The backward inference mechanism significantly enhances the model's ability to handle intricate, real-world API calling scenarios, reducing failure cases caused by missing or misordered API dependencies.

Answer to RQ3: 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.

8 Case Study

8.1 Insufficient Context Limit

We observe that models like xLAM and NemoTron-Mini, which have a 4K context limit, struggle with longer API calls in CallNavi, where some inputs exceed 6K tokens. This limitation leads to truncated inputs, causing incorrect API selection and missing parameters in multi-call sequences. While models with higher context limits generally perform better, we also find that context length alone does not guarantee success—models must still effectively manage dependencies and navigate complex API workflows. These findings highlight the need for both expanded context windows and improved structured reasoning in function-calling tasks.

8.2 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.

8.3 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.

8.4 Logical Errors in Hard Questions

Logical errors are particularly prevalent in hard questions, where the task involves multiple dependent API calls or complex reasoning. These errors include incorrect sequencing of API calls, failure to propagate parameters correctly, or omitting necessary steps. e.g.:

- **Example**: When asked to retrieve a user's transaction history and compute their monthly spending, the model retrieves the transactions but fails to invoke an API for computation, leaving the task incomplete.
- **Impact**: Logical errors highlight the limitations of current models in handling multistep tasks' dependency reasoning.

8.5 Impact of JSON and YAML on Model Performance

To analyze the influence of input and output formats on model performance, we conducted experiments using JSON and YAML, two widely used structured data formats. These formats differ significantly in syntax and structure, which could affect the ability of models to interpret, process, and generate outputs accurately. We tested four configurations:

- **YAML to YAML**: Both input and output are YAML.
- **JSON to JSON**: Both input and output are JSON.
- **YAML to JSON**: Input data is formatted in YAML, and output data is in JSON.
- **JSON to YAML**: Input data is formatted in JSON, and output data is in YAML.

The results of these experiments are shown in Table 8.

Table 8: Compare JSON or YAML as input/output performance differences.

Input/ Output	Model	Syntax Acc	Structure Acc	Easy	Medium	Hard	Overall	GPT-Score
YAML to YAML	LLAMA3.1	0.525	0.076	0.081	0.042	0.011	0.063	0.183
	mistral-small	0	0.183	0.23	0.047	0.035	0.16	0.241
	Gemma2:27b	0.938	0.097	0.12	0.037	0	0.085	0.238
	command-r	0.883	0.096	0.105	0.042	0.011	0.078	0.241
JSON to JSON	LLAMA3.1	0.925	0.207	0.223	0.058	0.059	0.162	0.422
	mistral-small	0.986	0.196	0.201	0.106	0.059	0.16	0.417
	Gemma2:27b	0.982	0.226	0.217	0.143	0.07	0.181	0.476
	command-r	0.969	0.189	0.167	0.095	0.047	0.134	0.4
YAML to JSON	LLAMA3.1	0.88	0.194	0.208	0.106	0.023	0.16	0.347
	mistral-small	0.995	0.179	0.173	0.112	0.058	0.144	0.333
	Gemma2:27b	0.984	0.212	0.195	0.159	0.082	0.172	0.414
	command-r	0.967	0.198	0.168	0.122	0.071	0.145	0.322
JSON to YAML	LLAMA3.1	0.598	0.104	0.14	0.026	0	0.094	0.32
	mistral-small	0	0.218	0.263	0.079	0	0.185	0.332
	Gemma2:27b	0.931	0.128	0.153	0.026	0	0.103	0.419
	command-r	0.853	0.091	0.123	0.011	0	0.079	0.363

Analysis of Results. **1. JSON Outperforms YAML.** Across all configurations, models achieved higher syntax, structure, and task-specific accuracies with JSON as both the input and output format. For example, the **JSON to JSON** configuration resulted in the highest Syntax Accuracy (e.g., 0.986 for Mistral-Small) and Structure Accuracy (e.g., 0.226 for Gemma2:27B), highlighting JSON's straightforward syntax and reduced ambiguity.

2. YAML Challenges. Models struggled significantly with YAML, particularly in the **YAML to YAML** configuration, which had the lowest performance across metrics. For instance, LLAMA 3.1 achieved a Syntax Accuracy of 0.525, and Structure Accuracy remained poor across models. YAML's indentation-sensitive syntax and verbosity likely contribute to these challenges.

3. Mixed Configurations Mitigate Errors. Configurations with mixed input and output formats (e.g., **YAML to JSON**) performed better than pure YAML setups. JSON as an output format simplified generation tasks, as evidenced by improved metrics compared to YAML outputs.

4. JSON to YAML is Challenging. The **JSON to YAML** configuration showed decreased performance compared to **JSON to JSON**, particularly in Syntax Accuracy (e.g., 0.598 for LLAMA 3.1). This indicates that YAML's complexity as an output format negatively affects model performance.

9 Discussion and Conclusion

9.1 Discussion

Our dataset introduces significant challenges, particularly in **medium** and **hard** questions, where models must select APIs from a large pool and generate parameters in **multi-step** and **nested contexts**. This complexity highlights the limitations of fine-tuned models trained on smaller API sets and underscores the need for more diverse and robust training paradigms.

We observe distinct model behaviours in API routing and parameter JSON generation. **GPT-4o** excels in both tasks, while models like **LLaMA 3.1** and **Gemma2** perform well in API routing but struggle with parameter generation, making them suitable for routing-centric applications. In contrast, smaller models (<10B parameters) exhibit instability, often producing inconsistent or incomplete outputs, limiting their effectiveness in complex, multi-step scenarios.

Long-context processing remains a significant bottleneck. Although models with larger context windows better handle structured inputs, they still struggle with simultaneous logical inference and structured JSON generation. Our findings suggest that merely increasing context size does not fully resolve multi-step reasoning challenges, emphasizing the need for improved architectures and reasoning strategies such as **Chain of Thought**.

Despite advancements, no model, including **GPT-4o**, fully solves intricate API calling tasks, reinforcing the need for further research in LLM-driven function calling.

9.2 Conclusion

This work introduces **`CallNavi`**, a benchmark evaluating API function calling in LLMs across **500 APIs** and **700 questions**. We assess general-purpose and fine-tuned models, revealing key limitations in **API selection, parameter generation, and multi-step reasoning**.

To improve function calling accuracy, we propose **2-steps generation** and **backward inference**, enhancing structured API selection. While larger models like **GPT-4o** perform well, they still struggle with **long-context input processing**, particularly in tasks requiring both **logical inference and structured JSON generation**. Models with <6K token limits often truncate inputs, leading to incomplete API calls and degraded performance.

Our findings contribute to the broader field of **software engineering evaluation and assessment**, particularly in automated API function that calls for AI-based software design, stability evaluation and structured reasoning. Future work should focus on improving LLM robustness in real-world deployments, integrating **retrieval-augmented techniques**, and expanding function-calling benchmarks to incorporate real-time constraints such as **error handling, authentication, and API versioning**.

Threats To Validity

Internal Validity. One key limitation is context length constraints, where models like xLAM and NemoTron(4K) struggle with inputs exceeding 6K tokens in CallNavi, leading to truncation and incomplete API calls. While models with longer context windows perform better, our results suggest that context size alone is insufficient without strong reasoning and structured generation capabilities.

The complexity and variability of CallNavi, particularly in multi-step and nested API tasks, pose additional challenges. Fine-tuned models, often trained on smaller API sets, may struggle to generalize. Additionally, LLM-as-a-judge introduces potential subjectivity in evaluation. Our optimization strategies like 2-steps generation and backward inference—improve multi-step API selection, but their effectiveness may vary across different architectures.

External Validity. While CallNavi spans 500+ APIs and 700+ questions across 10 domains, it does not cover all real-world constraints, such as authentication, error handling, or evolving API versions. Future LLMS with longer context, hybrid architectures, etc., may demonstrate different performance trends.

Additionally, real-world API integration involves challenges beyond our benchmark, such as network failures, rate limits, and dynamic tool adaptation. While our evaluation covers syntax validity, AST match, and stability, future extensions should explore

live production-level testing to better assess real-world deployment challenges.

Acknowledgments

We thank our collaborator BGL BNP Paribas for their support. The FNR funded this research under grants NCER22/IS/16570468/NCERFT and BRIDGES2021/IS/16229163/LuxemBERT.

References

[1] 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* (2024).

[2] 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* (2023).

[3] Mistral Ai. 2024. AI in abundance. https://mistral.ai/news/september-24-release/

[4] Mistral Ai. 2024. Mistral NeMo. https://mistral.ai/news/mistral-nemo/

[5] Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2024. Guiding LLMs The Right Way: Fast, Non-Invasive Constrained Generation. *arXiv preprint arXiv:2403.06988* (2024).

[6] Paweł Budzianowski, Tsung-Hsien Wen, Bo-Hsiang Tseng, Inigo Casanueva, Stefan Ultes, Osman Ramadan, and Milica Gašić. 2018. Multiwoz–a large-scale multi-domain wizard-of-oz dataset for task-oriented dialogue modelling. *arXiv preprint arXiv:1810.00278* (2018).

[7] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[8] Cohere. [n. d.]. The Command R model — Cohere. https://docs.cohere.com/v2/docs/command-r

[9] Yihong Dong, Ge Li, and Zhi Jin. 2023. CODEP: grammatical seq2seq model for general-purpose code generation. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 188–198.

[10] Mengnan Du, Fengxiang He, Na Zou, Dacheng Tao, and Xia Hu. 2023. Shortcut Learning of Large Language Models in Natural Language Understanding. arXiv:2208.11857 [cs.CL] https://arxiv.org/abs/2208.11857

[11] 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. https://fireworks.ai/blog/firefunction-v2-launch-post

[12] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *arXiv preprint arXiv:2501.12948* (2025).

[13] Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. 2024. StableToolBench: Towards Stable Large-Scale Benchmarking on Tool Learning of Large Language Models. *arXiv preprint arXiv:2403.07714* (2024).

[14] Jinwei He and Feng Lu. 2024. CauseJudger: Identifying the Cause with LLMs for Abductive Logical Reasoning. *arXiv preprint arXiv:2409.05559* (2024).

[15] Jonathan Herzig, Paweł Krzysztof Nowak, Thomas Müller, Francesco Piccinno, and Julian Martin Eisenschlos. 2020. TaPas: Weakly supervised table parsing via pre-training. *arXiv preprint arXiv:2004.02349* (2020).

[16] 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* (2023).

[17] Charlie Cheng-Jie Ji, Huanzhi Mao, Shishir G. Patil Fanjia Yan, Tianjun Zhang, Ion Stoica, and Joseph E. Gonzalez. 2024. Gorilla OpenFunctions v2. https://gorilla.cs.berkeley.edu//blogs/7_open_functions_v2.html.

[18] 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* (2023).

[19] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097.

[20] Michael Xieyang Liu, Frederick Liu, Alexander J. Fiannaca, Terry Koo, Lucas Dixon, Michael Terry, and Carrie J. Cai. 2024. "We Need Structured Output": Towards User-centered Constraints on Large Language Model Output. In *Extended Abstracts of the CHI Conference on Human Factors in Computing Systems* (Honolulu, HI, USA) *(CHI EA '24)*. Association for Computing Machinery, New York, NY, USA, Article 10, 9 pages. doi:10.1145/3613905.3650756

[21] 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* (2024).

[22] Meta-AI. 2024. Introducing Llama 3.1: our most capable models to date. https://ai.meta.com/blog/meta-llama-3-1/

[23] Meta-AI. 2024. Llama 3.2: Revolutionizing edge AI and vision with open, customizable models. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/

[24] Nexusflow.ai. 2023. NexusRaven-V2: Surpassing GPT-4 for Zero-shot Function Calling. https://nexusflow.ai/blogs/ravenv2

[25] 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* (2023).

[26] Jason Paul. 2024. NVIDIA announces first digital Human Technologies On-Device Small Language Model, improving conversation for game characters | NVIDIA blog. https://blogs.nvidia.com/blog/digital-human-technology-mecha-break/

[27] Yun Peng, Shuqing Li, Wenwei Gu, Yichen Li, Wenxuan Wang, Cuiyun Gao, and Michael R Lyu. 2022. Revisiting, benchmarking and exploring API recommendation: How far are we? *IEEE Transactions on Software Engineering* 49, 4 (2022), 1876–1897.

[28] 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* (2023).

[29] Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract syntax networks for code generation and semantic parsing. *arXiv preprint arXiv:1704.07535* (2017).

[30] 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* (2024).

[31] 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* (2023).

[32] 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. arXiv:2402.02172 [cs.SE] https://arxiv.org/abs/2402.02172

[33] 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* (2024).

[34] 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* (2023).

[35] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[36] Adina Trufinescu. 2024. Discover the new Multi-Lingual, High-Quality PHI-3.5 SLMS. https://techcommunity.microsoft.com/t5/ai-azure-ai-services-blog/discover-the-new-multi-lingual-high-quality-phi-3-5-slms/ba-p/4225280

[37] 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 (2022), 24824–24837.

[38] 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.

[39] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. *arXiv preprint arXiv:2406.12045* (2024).

[40] 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* (2024).

[41] Shayan Zamanirad, Boualem Benatallah, Moshe Chai Barukh, Fabio Casati, and Carlos Rodriguez. 2017. Programming bots by synthesizing natural language expressions into API invocations. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 832–837.

[42] 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* (2024).

[43] 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)*. IEEE, 2247–2251.

[44] Yinger Zhang, Hui Cai, Xeirui Song, Yicheng Chen, Rui Sun, and Jing Zheng. 2023. Reverse chain: A generic-rule for llms to master multi-api planning. *arXiv preprint arXiv:2310.04474* (2023).

[45] 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 (2023), 46595–46623.

[46] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. 2024. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877* (2024).