APIGen: Automated PIpeline for Generating Verifiable and Diverse Function-Calling Datasets

Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh Murthy, Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, Caiming Xiong
Salesforce AI Research, USA
{zuxin.liu, thai.hoang, jianguozhang}@salesforce.com

Abstract

The advancement of function-calling agent models requires diverse, reliable, and high-quality datasets. This paper presents APIGen, an automated data generation pipeline designed to synthesize high-quality datasets for function-calling applications. We leverage APIGen and collect 3,673 executable APIs across 21 different categories to generate diverse function-calling datasets in a scalable and structured manner. Each data in our dataset is verified through three hierarchical stages: format checking, actual function executions, and semantic verification, improving its reliability and correctness. We demonstrate that models trained with our curated datasets, even with only 7B parameters, can achieve state-of-the-art performance on the Berkeley Function-Calling Benchmark, outperforming multiple GPT-4 models. Moreover, our 1B model achieves exceptional performance, surpassing GPT-3.5-Turbo and Claude-3 Haiku. We release a dataset containing 60,000 high-quality entries, aiming to advance the field of function-calling agent domains. The dataset is available on Huggingface [1] and the project homepage [2].

1 Introduction

Function-calling agents represent a significant advancement in artificial intelligence, specifically within the realm of Large Language Models (LLMs). These models, such as GPT4 [1], Gemini [2], and Mistral [3], have evolved to not only understand and generate human-like text but also to execute functional API calls based on natural language instructions. For instance, consider a user requesting the weather in Palo Alto, as illustrated in Fig. 1. The function-calling agent interprets this query, accesses the relevant API—such as `get_weather("Palo Alto", "today")`—and retrieves the weather information, all in real-time. This capability extends the utility of LLMs beyond simple conversation tasks to include dynamic interactions with a variety of digital services and applications, ranging from social media platforms to financial services [4, 5, 6, 7, 8, 9, 10].

Despite their growing popularity and potential, the deployment of function-calling agents is often hampered by the quality of the datasets used for training. Current datasets are largely static and lack comprehensive verification, leading to potential inaccuracies and inefficiencies of model fine-tuning in real-world applications [11, 12, 13, 14]. This limitation is particularly evident when models trained on these datasets encounter new, unseen APIs. For example, a model trained primarily on restaurant booking APIs may struggle when suddenly tasked with retrieving stock market data, as it lacks the specific training data or the adaptability to handle new domains.

[1] `https://huggingface.co/datasets/Salesforce/xlam-function-calling-60k`
[2] `https://apigen-pipeline.github.io/`

38th Conference on Neural Information Processing Systems (NeurIPS 2024) Track on Datasets and Benchmarks.

https://doi.org/10.52202/079017-1725

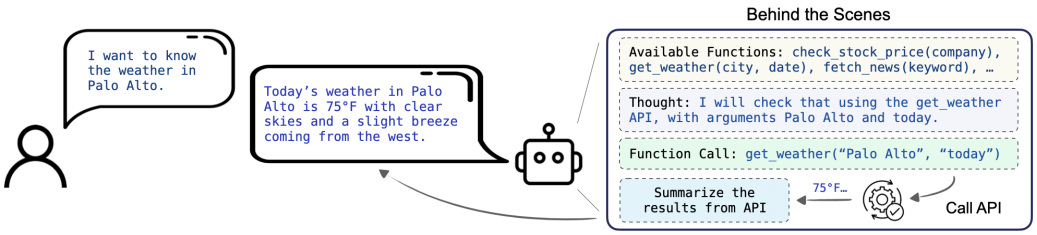

Figure 1: Workflow of an LLM-based function-calling agent.

To address these challenges, we introduce APIGen, an **A**utomated **PI**peline for **Gen**erating verifiable and diverse function-calling datasets. Our framework is designed to facilitate the fine-tuning of function-calling LLMs by providing high-quality, diverse datasets that better reflect the variability and complexity of real-world API use. Crucially, each generated data point undergoes rigorous multi-stage verification processes—format, execution, and semantic—to improve accuracy and applicability. We fine-tune function-calling models using the dataset generated by APIGen. The results show the strong performance of our models, surpassing many existing powerful LLMs with much fewer parameters, highlighting the effectiveness of APIGen and the high quality of the dataset it produces.

With APIGen, we release a comprehensive dataset containing 60,000 entries with 3,673 APIs across 21 categories. They include various query styles, such as parallel function calling data (asking the agent to produce multiple concurrent function calls in a single response) [11], which is rarely found in public datasets, to the best of our knowledge. This large-scale synthetic dataset is intended to catalyze further research and development in the field of function-calling agents, offering researchers and developers a foundation for training and testing their models. The data is available on Huggingface and our project homepage.

The contributions of this work are summarized as follows:

- We introduce APIGen, a function-calling data generation pipeline that features quality, scalability, and diversity of the data. APIGen is compatible with a range of models and APIs to construct high-quality synthetic function-calling datasets.
- We train two function-calling models of different sizes, 1.3B and 6.7B, using APIGen-constructed training data. Extensive experiments demonstrate that the 6.7B model achieves a rank of 3rd on the Berkeley Function-Calling Leaderboard [11], surpassing GPT-4o and Gemini-1.5-Pro, while the 1.3B model outperforms GPT-3.5-Turbo.
- We also release a synthetic function-calling dataset containing 60,000 high-quality data generated by APIGen using several strong open-source LLMs, which can potentially benefit the research community in developing advanced function-calling models.

2 Related Work

Tool-use Agent. Recent works have developed frameworks and models that enable LLMs to interact with APIs and tools [15, 16, 17, 18, 19, 20, 21, 22]. RestGPT [23] connects LLMs to RESTful APIs using a Planner, API selector, and API executor to handle complex instructions. Toolformer [24] is an early work that enables agents to use tools like Question Answering, Calculator, and Wikipedia Search through a supervised-finetuned model. [25, 26] propose the xLAM model series, showing strong tool usage capability across several benchmarks. Octopus-v4 [27] presents a methodology to incorporate multiple specialized language models to solve corresponding tasks. While NexusRaven [5] and Gorilla OpenFunctions-v2 [28] are strong open-sourced models that focus on function calling, neither provides access to their training datasets.

Agent Datasets. Several datasets have been created to support the development of agent models. AgentInstruct [20] consists of 6 datasets for different agent tasks, including AlfWorld [29], WebShop [30], Mind2Web [31], Knowledge Graph, Operating System, and Database [32]. APIBank [14] is a benchmark designed for tool-augmented LLMs, providing a training set containing tool-use dialogues from various APIs. Toolalpaca [33] constructs a varied and well-structured tool-use dataset by randomly selecting APIs and generating documentation using ChatGPT. ToolBench [12] creates

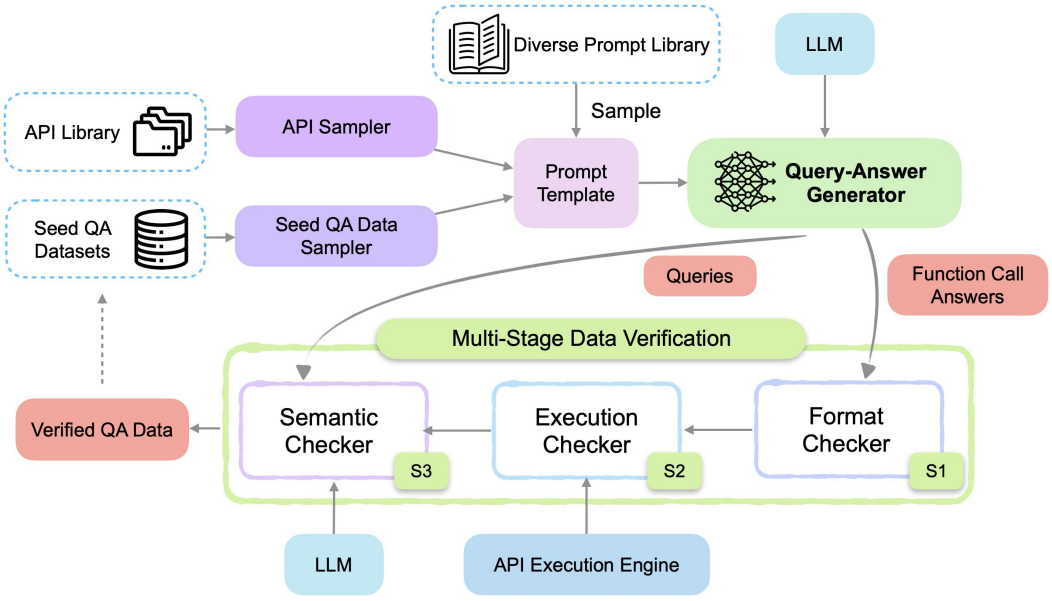

Figure 2: Illustration of the post-process filters.

an instruction-tuning dataset for tool use by collecting diverse REST APIs and generating their descriptions using ChatGPT. AgentOhana [26] and Lumos [34] design a unified data and training pipeline for efficient agent learning, covering multiple different datasets and environments. However, most of these datasets were not rigorously verified, and usually contain noisy data.

Benchmarks. Recent studies have established several benchmarks to assess agent abilities on various tasks such as web interactions, reasoning, decision making, function calling, code generation, and tool usage [30, 8, 32, 7, 35, 12, 13, 36, 37, 9, 38, 39]. Specifically, AgentBoard [37] includes 9 tasks, with ToolOperation and ToolQuery designed to evaluate agent ability on multi-turn interaction with external tools. ToolEval [12] assesses functional calling capabilities via RapidAPI, containing around 1,000 test cases and asking GPT-3.5 to assess the Win Rate. Furthermore, the Berkeley Function-Calling Leaderboard (BFCL) [11] provides a robust and comprehensive framework to evaluate models' abilities to call functions, with 1,700 test cases covering a wide range of scenarios. We use BFCL as our testing ground as it provides the most thorough comparison among popular LLMs.

3 APIGen Framework

This section introduces the detailed design of APIGen, an **A**utomated **PI**peline for **Gen**erating verifiable and diverse function-calling datasets. Our framework is designed with three key factors in mind: data quality, data diversity, and collection scalability. We achieve these through the key modules shown in Fig. 2: the multi-stage data verification process ensures data quality, the seed QA (query-answer) data sampler, API sampler, and various prompt templates ensure diversity, and our structured modular design using a unified format enables the system to scale to diverse API sources, including but not limited to Python functions and representational state transfer (REST) APIs.

3.1 Data Generation Overview

Figure 2 outlines the data generation process using the APIGen framework, which begins by sampling one or more APIs and example query-answer (QA) pairs (seed data) from the library, then formatting them into a standardized JSON format (see Fig. 3 for examples). A prompt template is selected based on the desired data generation objectives, which steers the LLM in generating relevant query-answer pairs. Each answer in the generated pairs is a function call formatted in JSON.

https://doi.org/10.52202/079017-1725

Figure 3: JSON data format examples.

The adoption of a standardized JSON format for APIs, function calls, and generator outputs (as shown in Figure 3) provides several advantages. Firstly, it establishes a structural way to verify whether the generator's output contains all necessary fields. Outputs that fail to comply with these format requirements are discarded. Secondly, the JSON structure enables efficient checking of function calls for correct parsing and validity of arguments. Calls that include arguments not present in the API library or hallucinate non-existent functions are excluded, enhancing the overall quality of the dataset. Another key benefit is the scalability it enables. With this uniform format, APIGen can easily incorporate data from diverse sources (Python functions, REST APIs, etc) by developing format converters that adapt them into these basic JSON elements, without modifying other core components, such as the prompting library, making the framework highly adaptable and extensible.

The generated function calls are subjected to a multi-stage verification process to improve their correctness and relevance. First, a format checker verifies correct JSON formatting and parseability. Next, the API execution engine processes the calls and sends the results and queries to a semantic checker, another LLM, which assesses alignment between the function calls, execution results, and query objectives. Data points passing all stages are added back to the seed dataset as high-quality examples to enhance future generation diversity. We detail each checker in the next section.

3.2 Multi-Stage Data Verification

Prioritizing quality is crucial, as previous research has shown that small amounts of high-quality fine-tuning data can substantially enhance model performance on domain-specific tasks [40]. This motivates our multi-stage dataset verification process to align large language models effectively.

The key insight driving our framework design is that, unlike synthetic chat data which can be difficult to evaluate, function-calling answers can be directly executed via their corresponding APIs. This enables checking if the output API and parameters' formats are correct, if the generated API calls are executable, and if execution results match the query's intent, etc. Based on this observation, we propose a three-stage verification process:

Stage 1: Format Checker: This stage performs sanity checks to filter out poorly formatted or incomplete data. The LLM output must strictly follow a JSON format with the "query" and "answer" fields, as shown in Fig. 3. We usually also include an additional "thought" field, which is known as the chain-of-thought (CoT) prompting technique [41], to increase the pass rate of the generated data. The data is discarded if these fields cannot be properly extracted for function calls. Additionally, the function calls are checked for correct JSON parsing and valid arguments. Generated calls whose arguments or functions are not present in the given APIs are eliminated to reduce hallucination and improve data quality.

Stage 2: Execution Checker: Well-formatted function calls from Stage 1 are executed against the appropriate backend (e.g. Python functions are directly imported and executed in a separate subprocess, while REST APIs are called to obtain results and status codes). Unsuccessful executions are filtered out, and fine-grained error messages are provided for failures, including argument type errors, invalid parameters, runtime errors, timeout, syntax errors, missing arguments, etc.

Stage 3: Semantic Checker: Successful Stage 2 execution results, available functions, and the generated query are formatted and passed to another LLM to assess if the results semantically align

with the query's objective. Query-answer pairs that execute successfully but produce meaningless results due to infeasible queries or incorrect arguments are filtered out. The main decision factors for this stage are: 1) whether the function call aligns with the query's objective and has proper arguments; 2) whether the function call and arguments are appropriately chosen from the available functions; 3) whether the number of function calls matches the user's intent; 4) whether the execution results contain errors or indicate unsuccessful function execution; 5) whether the execution results are relevant and match the query's purpose. APIGen's design offers the flexibility to select one or multiple LLMs as checkers, and the filtering rules can be readily adjusted—either tightened or relaxed—depending on specific use cases. Though the final stage can not guarantee correctness, the execution feedback information from stage 2 allows the checker to better assess the quality of the data, thus improving the decision accuracy.

Data points that pass all three verification stages are regarded as high-quality and added back to improve future diverse data generation. This multi-stage verification process is the key to ensuring the APIGen framework produces a dataset that is not only diverse but also of a high degree of confidence in data quality, enabling more effective fine-tuning of LLMs to domain-specific API-related tasks.

3.3 Methods to Improve Dataset Diversity

Encouraging diversity in training datasets is crucial for developing robust function-calling agents that can handle a wide range of real-world scenarios. In APIGen, we promote data diversity through multiple perspectives, including query style diversity, sampling diversity, and API diversity.

Query Style Diversity. APIGen's dataset is structured into four main categories: simple, multiple, parallel, and parallel multiple, each designed to challenge and enhance the model's capabilities in different usage scenarios. These categories are inspired by the Berkeley function-calling benchmark [11] and are controlled by corresponding prompts and seed data. We show examples of them in the supplementary material. The categories are as follows:

- **Simple:** This query style includes straightforward scenarios where a single function call is made based on the user's input with a single provided JSON format API description.
- **Multiple:** In this style, user queries could be answered by one of several function calls. The challenge lies in selecting the most appropriate function from multiple provided APIs. It represents one of the most common real-world use cases.
- **Parallel:** This query style requires executing multiple function calls simultaneously in response to a single user query, which may consist of one or more sentences but with only one API provided. For instance, if the user wants to know the weather in both `Palo Alto` and `Paris`, the model should call the `get_weather` function twice with corresponding city names in a single response.
- **Parallel Multiple:** This query style combines the parallel and multiple categories, where multiple function and API documents are provided, and each function call might be invoked multiple times based on the query's requirements.

While there exist publicly available training data for *simple* and *multiple* categories [42, 12], however, to the best of our knowledge, we offer the first large-scale and high-quality datasets that include the *parallel*-related function-calling scenario.

Sampling Diversity. APIGen utilizes a sampling system designed to maximize the diversity and relevance of the generated datasets, which include three main components, as shown in Fig. 2:

- **API Sampler:** This module extracts one or more function descriptions from executable API libraries, standardizing them into a uniform JSON format. The diverse sources of APIs ensure a wide range of function calls are available for inclusion in the training dataset.
- **Example Sampler:** It samples a specified number of seed examples corresponding to the different categories. These examples are transformed into structured queries, function descriptions, and answers, serving as an important few-shot reference for data generation.
- **Prompt Sampler:** This sampler draws from a diverse prompt library to generate a variety of query-answer pairs. The prompts for each query style contain different contexts, ranging from simple, concise query-answer pairs to more realistic scenarios, such as ambiguous or misspelled user requests, enhancing the model's ability to handle real-world interactions.

 https://doi.org/10.52202/079017-1725

We provide some prompt templates and seed data in the supplementary material. In APIGen, the number of examples and APIs sampled for each dataset iteration is randomly chosen from a predefined range. This randomization enhances dataset variability by preventing repetitive patterns and ensuring a broad coverage of scenarios. We next introduce our API diversity.

4 Dataset Preparation and Collection

We begin by discussing our dataset preparation process, which includes selecting and cleaning API libraries. Then we present our dataset collection setup and an overview of the resulting dataset.

4.1 Dataset API Sources

To ensure a high-quality and diverse dataset, we focused on collecting real-world APIs that could be readily executed and came with thorough documentation. We primarily sourced APIs from ToolBench [12], a comprehensive tool-use dataset that includes 16,464 REST APIs across 49 coarse-grained categories from RapidAPI Hub. This hub is a leading marketplace featuring a vast array of developer-contributed APIs. To further enhance the usability and quality of the APIs, we perform the following filtering and cleaning procedures on the ToolBench dataset:

- **Data Quality Filtering:** We remove APIs with incorrectly parsed documentation and those lacking required or optional parameters. APIs requiring no parameters were excluded to maintain the challenge level appropriate for our dataset needs.
- **API Accessibility Testing:** We tested API accessibility by making requests to each endpoint using example parameters provided in the dataset and through the Stable Toolbench server [42]. APIs that could not be executed or returned errors, such as timeouts or invalid endpoints, were discarded.
- **Docstring Regeneration:** To improve the quality of API documentation, we regenerated docstrings for the APIs that have noisy and unusable descriptions.

After cleaning, we obtain 3,539 executable REST APIs with good documentation. Additionally, we incorporated Python functions as another API type, inspired by the executable evaluation categories of the Berkeley function-calling benchmark [11]. We collected 134 well-documented Python functions covering diverse fields such as mathematics, finance, and data management. Sample API examples are provided in the supplementary material.

The original ToolBench dataset contained semantically overlapping categories such as `Finance` and `Financial`. We consolidated these into 21 distinct categories to ensure clarity and balance across the dataset. Figure 4 illustrates the distribution of the 3,673 executable APIs across these redefined categories, spanning sectors like technology, social sciences, education, and sports. This diverse collection of APIs provides a strong foundation for synthetic data generation and is a valuable asset for ensuring data quality and reliability.

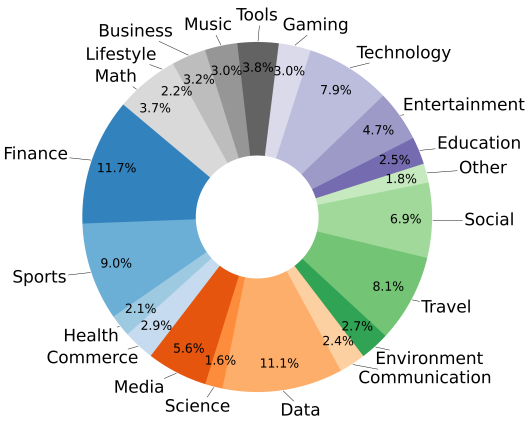

Figure 4: The category distribution of the 3,673 executable APIs.

4.2 Collection Setup and Dataset Details

To validate the effectiveness of the APIGen framework, we generated datasets targeting various query styles as outlined in Section 3.3. We utilized several base LLMs for data generation, including DeepSeek-V2-Chat (236B) [43], DeepSeek-Coder-33B-Inst [44], Mixtral-8x22B-Inst, and Mixtral-8x7B-Inst [3]. For each model, our target was to generate 40,000 data points by sampling different combinations of APIs, seed data, and prompt templates. To foster diversity in the generated responses,

we set the generation temperature to 0.7 across all models. Examples of the prompt templates and APIs used are provided in the supplementary materials for reference.

Table 1 presents statistics for the data generation process with different models, including the total verified data point count and the number of filtered data points at each verification stage. The filtering process successfully removes many low-quality data points due to formatting issues, execution errors, or failure to pass the semantic check. The first two stages, format checker and execution checker, typically filter out the majority of low-quality data. These data points often have infeasible argument ranges, incorrect types, missing required parameters, or more severe issues such as hallucination of function calls or parameters. Our systematic verification process provides a rigorous way to reduce the occurrence of these situations.

Table 1: Filtering statistics for the generated datasets using different base LLMs.

Model	Verified Data	Fail Format	Fail Execution	Fail Semantic	Pass Rate
DeepSeek-Coder-33B-Inst	13,769	4,311	15,496	6,424	34.42%
Mixtral-8x7B-Inst	15,385	3,311	12,341	7,963	38.46%
Mixtral-8x22B-Inst	26,384	1,680	5,073	6,863	65.96%
DeepSeek-V2-Chat (236B)	33,659	817	3,359	2,165	84.15%

The semantic checker also plays a crucial role in filtering generated data that does not align with the query's objectives. For instance, if a user's query contains multiple requests, but the returned results only address one, or if the generated function-call data and execution results do not match the user's query, the data point will be filtered out. Including these data points in the training set for model training could potentially harm the performance, as demonstrated in the experiments.

We observe that stronger models like DeepSeek-V2-Chat and Mixtral-8x22B-Inst have better format-following capabilities and higher pass rates, while the two relatively smaller models have a much higher likelihood of producing data that cannot be executed. This suggests that when using weaker models to generate data, a strict verification process is recommended to filter out low-quality data.

We are releasing approximately 60,000 high-quality function-calling datasets generated from the two strongest models: Mixtral-8x22B-Inst and DeepSeek-V2-Chat (236B). These datasets include all the query styles mentioned in Sec. 3.3 and cover a wide range of practical situations, with 3,673 diverse APIs across 21 categories. Each data point has been verified with high confidence of correctness using real-world API executions and the semantic checker. We also conducted human inspection on 600 sampled data. The results show that over 95% of the data are correct (details in Appendix A.3), showing the effectiveness of the framework. By making this dataset publicly available, we aim to benefit the research community and facilitate future work in this area.

5 Experiments

5.1 Experiment Setup

To evaluate the utility and effectiveness of the collected dataset, we conducted experiments by training function-calling models with the generated data. Our aim is to answer two key questions: 1) To what extent can the generated data boost the model's function-calling capability, and how does it compare to existing models? 2) How effective is the APIGen framework in filtering out low-quality data?

To address these questions, we train two versions of base models: DeepSeek-Coder-1.3B-instruct and DeepSeek-Coder-7B-instruct-v1.5 [44] using the xLAM (large action model) training pipeline proposed in [25, 26]. We refer to these models as xLAM-1B (FC) and xLAM-7B (FC), where FC stands for the Function-Calling mode, similar to this mode in other existing models that output JSON-format function calls [1, 28, 45, 4]. We compare the performance of these small-sized models against state-of-the-art models, including different versions of GPT-4 series [1], Claude-3 series [46], Gemini series [2], Llama3 [47], Mixtral [3], OpenFunctions-v2 [28], Command R+ [45], etc.

Benchmark. We evaluate the trained models' performance on the Berkeley Function-Calling Benchmark (BFCL) [11], which provides a comprehensive evaluation framework for assessing the function-calling capabilities of LLMs across various programming languages and application domains. Designed to reflect real-world use cases, the BFCL includes 1,700 testing cases, covering

 https://doi.org/10.52202/079017-1725

complex scenarios such as parallel and multiple-function calls. The benchmark contains diverse API sources like Java, JavaScript, and Python, offering a detailed analysis of each model's ability to correctly interpret and execute commands under different conditions. BFCL serves as a highly detailed and scalable benchmark for evaluating LLMs' function-calling capabilities and provides a leaderboard to track the most recent and powerful LLMs, both commercialized and open-source.

Evaluation Metrics. The Berkeley Function-Calling Leaderboard (BFCL) evaluates LLMs using two main categories: Abstract Syntax Tree (AST) Evaluation and Executable Function Evaluation. The AST evaluation focuses on the syntactic accuracy of the generated function calls, ensuring that the model's output matches a predefined function documentation in structure and parameters. This includes checks for correct function names, required parameters, and appropriate data types. The Executable Function Evaluation goes a step further by running the generated function calls to verify their operational correctness. This executable test ensures that the functions not only compile but also execute correctly, providing the expected results, which is crucial for practical applications where real-time performance is essential.

5.2 Experiment Results Analysis

Can the generated data improve the model's function-calling capability and how does it compare to other most powerful models? The performance of our models, xLAM-7B and xLAM-1B, as presented in Table 2, highlights the effectiveness of our APIGen framework and the quality of the datasets produced. Notably, our xLAM-7B model ranks 3rd among the most powerful LLMs listed on the BFCL leaderboard, surpassing several versions of GPT-4 (GPT-4o, GPT4-Turbo-FC), Llama3-70B, multiple Claude-3 models, and a series of strong models which are known for their exceptional capabilities in various tasks, including function-calling. This achievement demonstrates the significant impact of our high-quality dataset on the model's function-calling performance.

Table 2: Performance comparison of different models on BFCL leaderboard (as of date 07/18/2024). The rank is based on the overall accuracy, which is a weighted average of different evaluation categories. "FC" stands for function-calling mode in contrast to using a customized "prompt" to extract the function calls.

Rank	Overall Accuracy	Model	Abstract Syntax Tree (AST) Evaluation				Evaluation by Executing APIs				Relevance Detection
			Simple	Multiple	Parallel	Parallel Multiple	Simple	Multiple	Parallel	Parallel Multiple	
1	90.18	Claude-3.5-Sonnet (Prompt)	86.73	95.5	92.5	92	100	96	82	80	85.42
2	88.29	GPT-4-0125-Preview (Prompt)	88.36	95	92	92	99.41	94	84	75	70.42
3	88.24	xLAM-7b-fc-r (FC)	85.64	94	91	87	96.47	88	84	80	84.58
4	87.71	Claude-3-Opus-20240229 (Prompt)	86.73	94	86.5	89	97.65	92	80	75	80.42
5	86.53	Nemotron-4-340b-instruct (Prompt)	83.45	92.5	90.5	85.5	98.24	96	82	77.5	78.33
6	86.35	Gemini-1.5-Pro-Preview-0514 (FC)	80.18	92	91.5	88	91.76	88	76	77.5	89.58
7	85.88	Gemini-1.5-Pro-Preview-0409 (FC)	80	92.5	90.5	87.5	90	90	74	77.5	88.75
8	85.88	GPT-4-1106-Preview (FC)	84	91.5	92.5	86.5	89.41	92	78	67.5	80.42
9	85.88	GPT-4-turbo-2024-04-09 (Prompt)	86.55	95	91	90	97.65	94	80	72.5	62.5
10	84.65	Gorilla-OpenFunctions-v2 (FC)	88	95	87.5	86.5	94.71	94	70	67.5	61.25
11	84.59	GPT-4-0125-Preview (FC)	80.18	93	90.5	84.5	83.53	92	86	77.5	82.92
12	84	Meta-Llama-3-70B-Instruct (Prompt)	81.45	93	91.5	86	91.76	88	84	77.5	69.17
13	83	GPT-4o-2024-05-13 (FC)	78.91	90	88	84.5	86.47	78	82	75	81.25
14	82.94	GPT-4-turbo-2024-04-09 (FC)	74.73	90	90	88	82.94	88	76	67.5	88.75
...											
22	80.29	Gemini-1.5-Flash-Preview-0514 (FC)	80.91	93.5	78	73	81.76	90	54	72.5	79.58
23	79.88	Functionary-Small-v2.4 (FC)	82.18	88.5	82	81	78.24	82	80	65	67.92
24	79.76	Command-R-Plus (FC) (Optimized)	79.09	91	88.5	82	81.18	86	74	67.5	63.75
25	78.94	xLAM-1b-fc-r (FC)	81.27	87	78	76	82.94	94	84	75	63.75
26	77.76	Claude-3-Opus (FC tools-2024-04-04)	82.73	91.5	58.5	62	90.59	94	38	62.5	82.5
27	76.71	Claude-instant-1.2 (Prompt)	79.82	85.5	83	69.5	84.71	80	82	65	57.5
28	76.47	Claude-3.5-Sonnet-20240620 (FC)	85.27	92	59	54.5	97.06	88	18	35	78.33
29	74.35	Claude-3-Haiku-20240307 (Prompt)	84.91	91.5	84.5	56	92.94	94	70	25	34.58
30	71.47	Claude-2.1 (Prompt)	80.18	76	55.5	53	71.18	84	46	47.5	83.33
31	70.88	Command-R-Plus (FC) (Original)	74.91	90	82	76	81.76	88	68	55	24.17
32	68.76	Mistral-large-2402 (FC Auto)	66.91	94.5	25.5	72	83.53	96	8	52.5	84.17
33	68.06	Nexusflow-Raven-v2 (FC)	75.27	86	44.5	61	67.06	92	74	62.5	57.5
34	67.06	Gemini-1.0-Pro-001 (FC)	79.09	92.5	30.5	25.5	86.47	84	44	12.5	80
35	66.06	DBRX-Instruct (Prompt)	64	71.5	72.5	60	71.18	86	80	62.5	55.83
36	65.12	Snowflake-arctic-instruct (Prompt)	62.36	69	59	53.5	87.65	86	74	72.5	59.58
37	64.29	Mistral-large-2402 (FC Any)	81.45	93.5	31.5	79	94.71	92	8	65	0
38	63.94	GPT-3.5-Turbo-0125 (FC)	61.45	66	91	81	93.53	80	82	70	2.08

Our smaller xLAM-1B model also shows remarkable results, securing the 25th position and outperforming many larger models, such as Claude-3 Haiku [46], Command-R-Plus [45], DBRX-Instruct [48], Mistral-large [3], and GPT-3.5-Turbo-0125. The results highlight the effectiveness of the APIGen pipeline in enhancing a model's function-calling capabilities, even with a much smaller size. Both xLAM-7B and xLAM-1B demonstrate substantial improvements in handling complex query types, particularly in the *parallel* and *multiple* function-calling scenarios, which are typically

underrepresented in existing publicly available dataset. This validates the value of our pipeline and datasets in addressing practical scenarios involving complex API interactions and multiple concurrent API calls, especially considering that the base model, DeepSeek-Coder-v1.5, only ranks 45th on the leaderboard and performs poorly in these categories.

Next, we answer the question: **how effective is the APIGen framework in filtering out low-quality data?** We conducted an ablation study by adding the datasets that were filtered out by stage 3 (semantic checker) and stage 2 (execution checker) back to the training set, simulating situations where generated data is used without the rigorous verification process. The performance comparison on the BFCL benchmark, shown in Fig. 5, reveals that using these filtered datasets for training harms the final performance, with a more significant impact on the smaller model. This indicates that directly using generated data might not yield the best results and demonstrates the effectiveness of our APIGen framework in filtering out low-quality data.

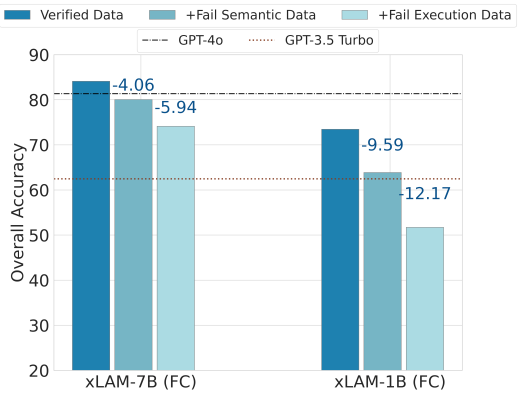

Figure 5: Performance comparison of using different stage's datasets from APIGen. "+Fail Semantic Data" and "+Fail Execution Data" meaning adding the filtered dataset from stage 3 and stage 2 to the training set.

These results provide compelling evidence for the effectiveness of the APIGen framework in generating high-quality, diverse datasets for function-calling tasks. The impressive performance achieved by our small-sized models highlights the efficiency of our approach, demonstrating that by focusing on data quality and diversity, we can effectively boost the performance of smaller models, making them competitive with much larger ones in this function-calling agent domain.

6 Conclusion

In this paper, we introduced APIGen, a novel framework that generates reliable and diverse function-calling datasets by leveraging a multi-stage verification process. Our experiments demonstrate the effectiveness of APIGen in producing high-quality datasets from a wide range of executable APIs. This has significant implications for the development of more efficient and accessible language models, as it shows that high-quality data can be as important as model size in achieving strong performance. By enabling smaller models to achieve competitive results and significantly enhancing their function-calling capabilities, our approach and released dataset open up new possibilities for the development of efficient and powerful language models in the agent tool-use domains.

However, the current version of APIGen and the generated dataset have some limitations. Presently, the framework and dataset only consider REST APIs and Python functions. Additionally, although APIGen is a general framework, it currently only implements the generation procedure for single-turn function calling. Future work will focus on extending APIGen to support more scenarios, programming languages, and APIs. We also plan to extend the framework to handle multi-turn and more complex interactions between agents, human users, and tools. Despite these limitations, we believe that APIGen and the generated dataset represent a significant step forward in the development of efficient and effective function-calling agents.

References

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

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

https://doi.org/10.52202/079017-1725

[3] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

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

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

[6] Zhiwei Liu, Weiran Yao, Jianguo Zhang, Le Xue, Shelby Heinecke, Rithesh Murthy, Yihao Feng, Zeyuan Chen, Juan Carlos Niebles, Devansh Arpit, et al. Bolaa: Benchmarking and orchestrating llm-augmented autonomous agents. *arXiv preprint arXiv:2308.05960*, 2023.

[7] Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chen Qian, Chi-Min Chan, Yujia Qin, Yaxi Lu, Ruobing Xie, et al. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors in agents. *ICLR*, 2023.

[8] Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

[9] Zhiwei Liu, Weiran Yao, Jianguo Zhang, Liangwei Yang, Zuxin Liu, Juntao Tan, Prafulla K Choubey, Tian Lan, Jason Wu, Huan Wang, et al. Agentlite: A lightweight library for building and advancing task-oriented llm agent system. *arXiv preprint arXiv:2402.15538*, 2024.

[10] Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, et al. Diversity empowers intelligence: Integrating expertise of software engineering agents. *arXiv preprint arXiv:2408.07060*, 2024.

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

[12] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world apis. *ICLR*, 2024.

[13] Xingyao Wang, Zihan Wang, Jiateng Liu, Yangyi Chen, Lifan Yuan, Hao Peng, and Heng Ji. Mint: Evaluating llms in multi-turn interaction with tools and language feedback. *arXiv preprint arXiv:2309.10691*, 2023.

[14] Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[15] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

[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. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

[17] Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*, 2023.

[18] Mengdi Xu, Peide Huang, Wenhao Yu, Shiqi Liu, Xilun Zhang, Yaru Niu, Tingnan Zhang, Fei Xia, Jie Tan, and Ding Zhao. Creative robot tool use with large language models. *arXiv preprint arXiv:2310.13065*, 2023.

[19] Yiheng Xu, Hongjin Su, Chen Xing, Boyu Mi, Qian Liu, Weijia Shi, Binyuan Hui, Fan Zhou, Yitao Liu, Tianbao Xie, et al. Lemur: Harmonizing natural language and code for language agents. *arXiv preprint arXiv:2310.06830*, 2023.

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

[21] Zuxin Liu, Jesse Zhang, Kavosh Asadi, Yao Liu, Ding Zhao, Shoham Sabach, and Rasool Fakoor. Tail: Task-specific adapters for imitation learning with large pretrained models. *arXiv preprint arXiv:2310.05905*, 2023.

[22] Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

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

[24] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36, 2024.

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

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

[27] Wei Chen and Zhiyuan Li. Octopus v4: Graph of language models. *arXiv preprint arXiv:2404.19296*, 2024.

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

[29] Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

[30] Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

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

[32] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

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

[34] Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. Lumos: Learning Agents with Unified Data, Modular Design, and Open-Source LLMs. *arXiv preprint arXiv:2311.05657*, 2023.

[35] Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, et al. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *arXiv preprint arXiv:2310.03128*, 2023.

 https://doi.org/10.52202/079017-1725

[36] Zuxin Liu, Zijian Guo, Haohong Lin, Yihang Yao, Jiacheng Zhu, Zhepeng Cen, Hanjiang Hu, Wenhao Yu, Tingnan Zhang, Jie Tan, et al. Datasets and benchmarks for offline safe reinforcement learning. *arXiv preprint arXiv:2306.09303*, 2023.

[37] Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. Agentboard: An analytical evaluation board of multi-turn llm agents. *arXiv preprint arXiv:2401.13178*, 2024.

[38] Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

[39] 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. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

[40] Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. Lima: Less is more for alignment. *Advances in Neural Information Processing Systems*, 36, 2024.

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

[42] Zhicheng Guo, Sijie Cheng, Hao Wang, Shihao Liang, Yujia Qin, Peng Li, Zhiyuan Liu, Maosong Sun, and Yang Liu. Stabletoolbench: Towards stable large-scale benchmarking on tool learning of large language models, 2024.

[43] DeepSeek-AI. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model, 2024.

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

[45] Cohere. Command r plus: Enhanced retrieval-augmented generation with microsoft azure. `https://cohere.com/blog/command-r-plus-microsoft-azure`, 2024. Accessed: 2024-04-04.

[46] AI Anthropic. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card*, 2024.

[47] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[48] Databricks. Introducing dbrx: A new state-of-the-art open llm. `https://www.databricks.com/blog/introducing-dbrx-new-state-art-open-llm`, 2024. Accessed: 2024-03-27.

A Dataset Documentation and Accessibility

A.1 Dataset Documentation and Intended Uses

The dataset generated using the APIGen framework is intended for training and evaluating function-calling agents. The dataset consists of diverse query-answer pairs, where the answers are verified function calls that could address the requested query with provided APIs. The APIs and function calls are in a standardized JSON format, as demonstrated in the main paper Fig. 3. More details of the format and examples are available in Appendix A.2. The dataset covers a wide range of API categories and includes various query styles, such as simple, multiple, parallel, and parallel multiple function calls, as introduced in [11].

Hosting, Licensing, and Maintenance Plan. The dataset currently can be viewed and downloaded from our project homepage [3] or via Huggingface [4]. All datasets are licensed under the Creative Commons Attribution 4.0 License (CC BY). We also plan to open-source the trained models on Huggingface once after the company's legal approval. As for maintenance, we have established a long-term plan to keep the datasets up-to-date, correct any potential issues, and provide support to users. We also aim to expand these datasets further based on new advances in the field, thus continually promoting progress in the field of function-calling agent training.

Author Responsibility Statement. As the authors, we hereby affirm that we bear full responsibility for the datasets provided in this submission. We confirm that to the best of our knowledge, no rights are violated in the collection, distribution, and use of these datasets.

A.2 JSON Data Format and Examples

This JSON data format is used to represent a query along with the available tools and the corresponding answers. Here's a description of the format:

A.2.1 Dataset Structure

The JSON data structure comprises three main keys: `query`, a string representing the problem statement; `tools`, an array of tools each defined by properties such as `name`, `description`, and `parameters` that further describe each tool's required and optional parameters with their types and descriptions; and `answers`, an array detailing responses with the tool used (`name`) and the arguments provided (`arguments`) for each answer, thereby aligning tools with their respective query intentions. The detailed description of each data point's entries is as follows.

- `query` (string): The query or problem statement.
- `tools` (array): An array of available tools that can be used to solve the query.

  Each tool is represented as an object with the following properties:
  - `name` (string): The name of the tool.
  - `description` (string): A brief description of what the tool does.
  - `parameters` (object): An object representing the parameters required by the tool.
    - Each parameter is represented as a key-value pair, where the key is the parameter name and the value is an object with the following properties:
      - `type` (string): The data type of the parameter (e.g., "integer", "float", "array").
      - `description` (string): A brief description of the parameter.
      - `required` (boolean): Indicates whether the parameter is required or optional.
- `answers` (array): An array of answers corresponding to the query.
  - Each answer is represented as an object with the following properties:
    - `name` (string): The name of the tool used to generate the answer.
    - `arguments` (object): An object representing the arguments passed to the tool to generate the answer.
      - Each argument is represented as a key-value pair, where the key is the parameter name and the value is the corresponding value.

[3] `https://apigen-pipeline.github.io/`

[4] `https://huggingface.co/datasets/Salesforce/xlam-function-calling-60k`

https://doi.org/10.52202/079017-1725

A.2.2 Example Data

Here's an example JSON data for the simplest scenario.

```
{
  "query": "What is the weather in Palo Alto?",
  "tools": [
    {
      "name": "weather_api.get_current_weather",
      "description": "Retrieves the current weather conditions
   for a specified location.",
      "parameters": {
        "location": {
          "type": "string",
          "description": "The name of the city or geographic
   location.",
          "required": true
        },
        "units": {
          "type": "string",
          "description": "The units for temperature measurement
    (e.g., 'Celsius', 'Fahrenheit').",
          "required": false
        }
      }
    }
  ],
  "answers": [
    {
      "name": "weather_api.get_current_weather",
      "arguments": {
        "location": "Palo Alto",
        "units": "Celsius"
      }
    }
  ]
}
```

In this example, the query asks about the current weather in Palo Alto. The tools array contains a single entry for `weather_api.get_current_weather`, describing the tool used to retrieve weather data, including parameters for location and units. The answers array lists the specific API call made with the location set as `"Palo Alto"` and units as `"Celsius"`.

Here's an example JSON data for the parallel function-calling category, i.e., the user's query contains multiple intentions and the answers contain multiple parallel tool calls:

```
{
  "query": "Find the sum of all the multiples of 3 and 5
   between 1 and 1000. Also find the product of the first five
    prime numbers.",
  "tools": [
    {
      "name": "math_toolkit.sum_of_multiples",
      "description": "Find the sum of all multiples of
   specified numbers within a specified range.",
      "parameters": {
        "lower_limit": {
          "type": "integer",
          "description": "The start of the range (inclusive).",
          "required": true
        },
```

```
        "upper_limit": {
          "type": "integer",
          "description": "The end of the range (inclusive).",
          "required": true
        },
        "multiples": {
          "type": "array",
          "description": "The numbers to find multiples of.",
          "required": true
        }
      }
    },
    {
      "name": "math_toolkit.product_of_primes",
      "description": "Find the product of the first n prime
   numbers.",
      "parameters": {
        "count": {
          "type": "integer",
          "description": "The number of prime numbers to
   multiply together.",
          "required": true
        }
      }
    }
  ],
  "answers": [
    {
      "name": "math_toolkit.sum_of_multiples",
      "arguments": {
        "lower_limit": 1,
        "upper_limit": 1000,
        "multiples": [3, 5]
      }
    },
    {
      "name": "math_toolkit.product_of_primes",
      "arguments": {
        "count": 5
      }
    }
  ]
}
```


In this example, the query asks to find the sum of multiples of 3 and 5 between 1 and 1000, and also find the product of the first five prime numbers. The available tools are `math_toolkit.sum_of_multiples` and `math_toolkit.product_of_primes`, along with their parameter descriptions. The `answers` array provides the specific tool and arguments used to generate each answer.

A.3 Human Evaluation of Dataset Quality

To ensure that the three-stage verification process employed by APIGen produces a high-quality dataset, we conduct a human evaluation on a sample of the generated data. We engage three human evaluators to manually inspect a total of 600 samples from our released dataset. The evaluators assess the quality of each sample based on factors such as the accuracy of parameter values and the appropriateness of the number of API calls.

https://doi.org/10.52202/079017-1725

The results of the human evaluation reveal that only 28 out of the 600 inspected samples have minor issues, such as inaccurate parameter values or more API calls than expected. This means that the majority of the data, approximately 95.3%, are of very high quality. The high quality of the dataset can be attributed to the format and execution checkers implemented in the APIGen pipeline.

The format checker ensures that the generated data adheres to the specified JSON format and contains all the necessary fields. This step helps to filter out poorly formatted or incomplete data points. The execution checker, on the other hand, executes the generated function calls against the appropriate backend and verifies their successful execution. By providing real execution results, the execution checker plays a crucial role in filtering out cases that might be difficult to identify by an LLM-based semantic checker alone.

The combination of these two checkers, along with the final semantic checker, creates a robust verification process that effectively filters out low-quality data points. The human evaluation results confirm the effectiveness of this approach, demonstrating that APIGen is capable of generating high-quality datasets for training function-calling agents.

B Dataset Generation and Experiment Details

B.1 Generator LLM Prompt

Example Prompt for the Generator to Generate Parallel Function-Calling Data

```
"""
You are a data labeler. The responsibility for you is to
    generate a set of diverse queries and corresponding
    answers for the given functions in JSON format.

Construct queries and answers that exemplifies how to use
    these functions in a practical scenario. Include in each
    query specific, plausible values for each parameter. For
    instance, if the function requires a date, use a typical
    and reasonable date.

Ensure the query:
- Is clear and concise
- Contain multiple parallel queries in natural language for
    the given functions, they could use either the same
    function with different arguments or different functions
- Demonstrates typical use cases
- Includes all necessary parameters in a meaningful way. For
    numerical parameters, it could be either numerals or words
- Across a variety level of difficulties, ranging from
    beginner and advanced use cases
- The corresponding result's parameter types and ranges match
     with the functions descriptions.

Ensure the answer:
- Is a list of function calls in JSON format.
- The length of the answer list should be equal to the number
     of requests in the query
- Can solve all the requests in the query effectively

Here are examples of queries and corresponding answers for
    similar functions:
{examples}
```

```
Note that the query could be interpreted as a combination of
    several independent requests.

Based on these examples and the above instructions, generate
    {number} diverse query and answer pairs for the functions
    `{func_name}`.
The detailed functions description is as follows:
{func_desc}

{format_inst}

Now please generate {number} diverse query and answer pairs
    following the above format.
"""
```

The template provided outlines the prompt for an LLM to generate datasets as data labelers, emphasizing the diversity of query types and complexity to ensure thorough coverage of potential real-world applications. It specifies the importance of generating clear, concise queries and precisely formatted JSON responses. Sampled data, used to populate the `examples` field, and API information, filling the `func_name` and `func_desc` fields, enable a structured approach to dataset generation. The `format_inst` specifies the enforced JSON output format, as shown below.

Example Format Instruction to Generate Parallel Function-Calling Data

````
The output MUST strictly adhere to the following JSON format,
    and NO other text MUST be included:
```
[
  {
    "query": "The generated query.",
    "answers": [
      {
        "name": "api_name",
        "arguments": {
          "arg_name": "value",
          ... (more arguments as required)
        }
      },
      ... (more API calls as required)
    ]
  }
]
```
````

The enforced JSON output format facilitates efficient data extraction and cost-effective generation. By requesting multiple query-answer pairs in a single inference with the `number` field—referred to here as a "batching" technique—token usage and costs are significantly reduced.

B.2 Semantic Checker LLM Prompt

We prompted another LLM as the semantic checker to evaluate whether the execution results and the tool calls align with the user query. We could use multiple LLMs with different prompts as checkers here to increase the credibility of this verification stage. We provide one example prompt as follows.

https://doi.org/10.52202/079017-1725

Example Prompt for the Semantic Checker to Verify the Data

```
"""
As a data quality evaluator, you must assess the alignment
    between a user query, corresponding function calls, and
    their execution results.
These function calls and results are generated by other
    models, and your task is to ensure these results
    accurately reflect the user's intentions.

Do not pass if:

1. The function call does not align with the query's
    objective, or the input arguments appear incorrect.
2. The function call and arguments are not properly chosen
    from the available functions.
3. The number of function calls does not correspond to the
    user's intentions.
4. The execution results are irrelevant and do not match the
    function's purpose.
5. The execution results contain errors or reflect that the
    function calls were not executed successfully.

Given Information:
- All Available Functions:
  {func_desc}

- User Query: {query}

- Generated Function Calls: {func_call}

- Execution Results: {execution_result}

Note: The query may have multiple intentions. Functions may
    be placeholders, and execution results may be truncated
    due to length, which is acceptable and should not cause a
    failure.
The main decision factor is wheather the function calls
    accurately reflect the query's intentions and the function
     descriptions.

Provide your reasoning in the thought section and decide if
    the data passes (answer yes or no).
If not passing, concisely explain your reasons in the thought
     section; otherwise, leave this section blank.

Your response MUST strictly adhere to the following JSON
    format, and NO other text MUST be included.
```
```
{{
  "thought": "Concisely describe your reasoning here",
  "pass": "yes" or "no"
}}
```
```
"""
```

Here, the `func_desc` field is the same as the generator, while the `func_call` and `execution_result` are the key fields to determine whether the generated data successfully address the `query`'s intention. We also enforce the model to output a JSON-formatted string, and then extract whether we should give a pass to this data point.

B.3 Model Training

We train two function-calling models of different sizes, xLAM-1B (FC) and xLAM-7B (FC), using the dataset generated by APIGen. The training pipeline mainly follows the AgentOhana paper [26]. We use 8 NVIDIA A100 40GB GPUs for training both models.

Since the Berkeley Function-Calling Benchmark [11] contains a relevance detection category, which evaluates a model's ability to distinguish non-relevant queries and tools, we extend APIGen to generate relevance detection data points from the generated datasets. These data points cover two types of scenarios:

- The provided tools cannot solve the query (e.g., query: "I want to know the weather in Palo Alto on Dec 25, 2023," provided tool: `get_house_price(city)`).
- The provided tools are missing key arguments to solve the query (e.g., query: "I want to know the weather in Palo Alto on Dec 25, 2023," provided tool: `get_weather(city)`).

In both cases, the correct output is an empty tool call or a concise explanation indicating that the model should refuse to answer due to insufficient or irrelevant information.

We create 8,000 such data points from the collected dataset by 1) randomly discarding some tools that will be called in the answer or 2) randomly dropping some required parameters that were used in the generated tool calls. Then we relabel the answer to be an empty tool call or with a concise explanation. By incorporating relevance detection data points into our training datasets, we can enhance the model's performance in determining when the provided tools are not suitable for addressing a given query. This enables the training of agents that can effectively assess the relevance of the available tools and respond appropriately, either by utilizing the relevant tools or by refraining from answering when the necessary information is lacking.

When training the model, we fill in the sampled query and available tools to the training prompt template, and then ask the model to predict the corresponding tool calls in specified JSON format. The training prompt template is as follows:

Model Training Prompt

```
"""
[BEGIN OF TASK INSTRUCTION]
You are an expert in composing functions. You are given a
    question and a set of possible functions.
Based on the question, you will need to make one or more
    function/tool calls to achieve the purpose.
If none of the function can be used, point it out and refuse
    to answer.
If the given question lacks the parameters required by the
    function, also point it out.
[END OF TASK INSTRUCTION]

[BEGIN OF AVAILABLE TOOLS]
{func_desc}
[END OF AVAILABLE TOOLS]

[BEGIN OF FORMAT INSTRUCTION]
The output MUST strictly adhere to the following JSON format,
     and NO other text MUST be included.
```


https://doi.org/10.52202/079017-1725

```
The example format is as follows. Please make sure the
   parameter type is correct. If no function call is needed,
   please make tool_calls an empty list '[]'
```
```
{{
  "tool_calls": [
    {{"name": "func_name1", "arguments": {{"argument1": "
   value1", "argument2": "value2"}}}},
    ... (more tool calls as required)
  ]
}}
```
```
[END OF FORMAT INSTRUCTION]

[BEGIN OF QUERY]
User Query: {query}
[END OF QUERY]
"""
```

The training hyperparameters for our models include a learning rate of 5×10^{-6}, four epochs, and use of the AdamW optimizer. Other settings include a cutoff length of 2048, a per-device batch size of six, two gradient accumulation steps, a cosine learning rate scheduler with 50 warmup steps, and the bfloat16 (BF16) data type.