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 [1].

Hosting, Licensing, and Maintenance Plan. The dataset currently can be viewed and downloaded from our project homepage [1] or via Huggingface [2]. 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:

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

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

        * `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.

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.

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.

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 [2]. We use 8 NVIDIA A100 40GB GPUs for training both models.

Since the Berkeley Function-Calling Benchmark [1] 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.
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}, two 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.

References

[1] 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.

[2] 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.