Version: 1.0

call_me_maybe

Introduction to function calling in LLMs

Summary

Does LLMs Speak the language of computers? We'll find out.
Made in collaboration with @ldevelle, @pcamaren, @crfernan

#LLM #AI #PARSING

Intellectual Property Disclaimer

All content presented in this training module, including but not limited to texts, images, graphics, and other materials, is protected by intellectual property rights held by Association 42.

Terms of Use:

- **Personal use:** You are permitted to use the contents of this module solely for personal purpose. Any commercial use, reproduction, distribution, modification, or public display is strictly prohibited without prior written permission from Association 42.
- **Respect for Integrity:** You must not alter, transform, or adapt the content in any way that could harm its integrity.

Protection of Rights:

Any violation of these terms constitutes an infringement of intellectual property rights and may result in legal action. We reserve the right to take all necessary measures to protect our rights, including but not limited to claims for damages.

For any questions regarding the use of the content or to obtain authorization, please contact: `legal@42.fr`

Contents

1 Foreword **1**

2 Common Instructions **2**

2.1 General Rules 2

2.2 Makefile 2

2.3 Additional Guidelines 2

2.3.1 Additional instructions 3

3 Do LLMs Speak the language of computers? **4**

3.0.1 Summary 4

3.0.2 Input 4

3.0.3 LLM interaction 5

3.0.4 Output File Format 5

4 Beta **7**

Chapter 1

Foreword

Roman engineers carved stone tablets with perfect grids to track grain shipments across the empire, so no delivery was lost to bad handwriting. In the 1800s, sailors logged ocean currents in structured tables so precise that some are still used in navigation today.

The first weather forecasts were sent as telegrams in a fixed code — a few numbers that could describe an entire sky. In the 1960s, NASA's mission control worked from laminated flowcharts that told them exactly what each blinking light meant, no matter who was on shift. Barcode scanners in supermarkets translated black-and-white stripes into inventory data long before most homes had computers. Even beekeepers have used standardized forms for decades, recording hive health, nectar flow, and queen lineage in tiny boxes only they could read at a glance.

Humans have always built structures to make information reliable, shareable, and usable. Which brings us to today, where the goal is to make AI speak the language of computers.

Chapter 2

Common Instructions

2.1 General Rules

- Your project must be written in **Python 3.11 or later**.
- Your project must adhere to the **flake8** coding standard. Bonus files are also subject to this standard.
- Your functions should handle exceptions gracefully to avoid crashes. Use `try-except` blocks to manage potential errors. If your program crashes due to unhandled exceptions during the review, it will be considered non-functional.
- All resources (e.g., file handles, network connections) must be properly managed to prevent leaks.

2.2 Makefile

Include a `Makefile` in your project to automate common tasks. It must contain the following rules:

- **install**: Install project dependencies using `pip`, `uv`, `pipx`, or any other package manager of your choice.
- **run**: Execute the main script of your project.
- **debug**: Run the main script in debug mode using Python's built-in debugger.
- **clean**: Remove temporary files or caches to keep the project environment clean.
- **lint**: Lint your code using `flake8` to ensure it meets coding standards.

2.3 Additional Guidelines

- Create test programs to verify project functionality (not submitted or graded).
- Submit your work to the assigned Git repository. Only the content in this repository will be graded.

If any additional project-specific requirements apply, they will be stated immediately below this section.

2.3.1 Additional instructions

- All classes must use **pydantic** for validation.
- You can use the **numpy** and **json** packages.
- The use of **dspy** (or any similar package) is completely forbidden, including pytorch, huggingface package, transformers etc.
- You need to use the following models:
  - **Qwen/Qwen3-0.6B** (default)
  - You can use other models as long as it is working with **Qwen/Qwen3-0.6B**
- The function to call should be chosen using the LLM, not with heuristics or any other sort of medieval magic.
- It is forbidden to use any private methods or attributes from the `LLM_SDK` package.
- You should create a virtual environment and install the packages **numpy**, and **pydantic** using **uv**. To use **llm_sdk** you can copy it in the same directory than the one `src` is in.
- The evaluators, as well as the moulinette, will just run `uv sync`.
- Your program must be run using the following command (where `src` is the folder containing your files):

```
uv run python -m src
```

- All errors should be handled gracefully. It must never crash unexpectedly, and must always provide a clear error message to the user.

Chapter 3

Do LLMs Speak the language of computers?

3.0.1 Summary

In this project, you should create a function calling tool. What is that, you might be asking yourself? It is a program which takes a question, and instead of returning the answer to that question, provides a toolkit to solve it.

For example, given the question "What is the sum of 40 and 2?", the solution should not return 42, but instead, provides a function to call, in this particular case, fn_add_numbers, and the passed arguments should be 40 and 2.

3.0.2 Input

As much as 42 has high expectations for its students, you are not asked to code the function-calling-crystal-ball. Or at least not yet.

In order to have an idea of the types of problems that your solution should solve, you are given the file `input/function_calling_tests.json` containing a **JSON structured list** of possible **prompts** to input to your function.

On the file `input/function_definitions.json`, you will find a list of functions among which your solution should decide the best one to call in order to solve the given prompt. This list includes the definition of the functions' argument(s) name(s) and type(s); and the functions' return type.

These examples are provided to establish the expected level of complexity for the problems. Keep in mind, however, that your solution will be tested with different prompts and varying sets of functions. Otherwise, the exercise would not be particularly engaging. You must implement proper JSON error handling for input files, as they may contain invalid JSON or be missing.

3.0.3 LLM interaction

We have highlighted the word prompt, therefore, you might have already imagined that you will be interacting with an AI somehow. And guess what? Once again, you are right!

The expected solution should interact with an LLM in order to produce the toolkit we mentioned earlier.

Attached to this project, you'll find a wrapper class **Small_LLM_Model** that you can use to interact with the LLM.

There is two methods that you can use to interact with the LLM.

- `get_logits_from_input_ids` : It takes an `input_ids` tensor and returns the raw logits after calling the LLM model.
- `get_path_to_vocabulary_json` : It returns the location in your computer where the structured JSON correspondence between the `input_ids` and the tokens is located.

The following diagram illustrates the process of interacting with the LLM.

```
Prompt → Tokens → Inputs_Ids → LLM Interaction → Logits → Next Token
```

Your solution must not rely on including the entire function definitions JSON content directly in the prompt sent to the LLM. This approach would be considered an invalid solution as it bypasses the core challenge of making the LLM understand and reason about function calling through proper prompt engineering and token-level interactions.

3.0.4 Output File Format

Your program will produce a single JSON file: `output/function_calling_name.json`. For each prompt add a JSON object to this file. Each object in the array must contain exactly the following keys:

- `str` : The original natural-language request.
- `str` : The name of the function to call.
- `object` : all required arguments with the correct types.

For example:

```
output/function_calling_name.json
[
    {
        "prompt": "What is the sum of 2 and 3?"
        "fn_name": "fn_add_numbers",
        "args": { "a": 2.0, "b": 3.0 }
    },
    {
        "prompt": "Reverse the string 'hello'",
```

```
        "fn_name": "fn_reverse_string",
        "args": { "s": "hello" }
    }
]
```

Validation rules:

- The file must be valid JSON (no trailing commas, no comments).
- Keys and types must match the schema in `functions_definitions.json` exactly.
- No extra keys or prose are allowed anywhere in the output.

The file is generated at runtime by the scaffold; your job is to guide the model so that each predicted `fn_name` and `args` passes validation and ends up in `output/function_calling_name.json` as shown above.

The given files may change during the evaluation, so you should not use them as a reference.

Chapter 4

Beta

Thank you for participating in the beta of this project. We are looking forward to your feedback.