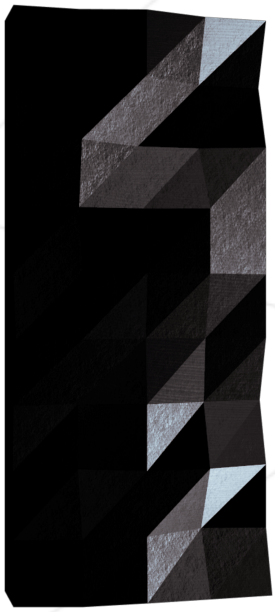

call me maybe

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

Version: 1.1

Contents

Chapter I

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 II

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Introduction

III.1 What is Function Calling?

Large Language Models (LLMs) are powerful at understanding and generating human language, but they don't naturally produce structured, machine-executable output. Function calling bridges this gap by translating natural language requests into precise function calls with typed arguments.

Consider this example: Natural Language to Function Call

```
User: "What is the sum of 40 and 2?"

Traditional LLM: "The sum of 40 and 2 is 42."

Function Calling System:
{
  "function": "add_numbers",
  "arguments": {"a": 40, "b": 2}
}
```

The function calling system doesn't answer the question directly. Instead, it provides the **tools** to solve it: the right function name and the correct arguments with proper types.

III.2 Why is This Important?

Function calling enables LLMs to:

- **Interact with external systems**: Call APIs, query databases, control devices
- **Execute code**: Perform calculations, data transformations, file operations
- **Chain operations**: Break complex tasks into executable steps
- **Provide structured output**: Generate JSON, XML, or other machine-readable formats
- **Extract structured data from unstructured text**: For example, given a large book, extract fields such as `{protagonist name, protagonist sex, protagonist age}`

This technology powers modern AI assistants, code generation tools, and autonomous agents, while also enabling tasks like automatic information extraction and knowledge structuring from raw text.

III.3 The Challenge

Small language models (like the 0.6B parameter model you'll use) are notoriously unreliable at generating structured output. When prompted to produce JSON, they might succeed only 30% of the time. Yet production systems achieve 99%+ reliability with these same small models.

How? The answer lies in **constrained decoding** — a technique that guides the model's output token-by-token to guarantee valid structure, without relying on prompting alone.

Chapter IV

Common Instructions

IV.1 General Rules

- Your project must be written in **Python 3.10 or later**.
- Your project must adhere to the **flake8** coding standard.
- Your functions should handle exceptions gracefully to avoid crashes. Use `try-except` blocks to manage potential errors. Prefer context managers for resources like files or connections to ensure automatic cleanup. 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. Use context managers where possible for automatic handling.
- Your code must include type hints for function parameters, return types, and variables where applicable (using the `typing` module). Use `mypy` for static type checking. All functions must pass mypy without errors.
- Include docstrings in functions and classes following PEP 257 (e.g., Google or NumPy style) to document purpose, parameters, and returns.

IV.2 Makefile

Include a `Makefile` in your project to automate common tasks. It must contain the following rules (mandatory lint implies the specified flags; it is strongly recommended to try `-strict` for enhanced checking):

- **install**: Install project dependencies using `pip`, `uv`, `pipx`, or any other package manager of your choice.
- **run**: Execute the main script of your project (e.g., via Python interpreter).
- **debug**: Run the main script in debug mode using Python's built-in debugger (e.g., pdb).
- **clean**: Remove temporary files or caches (e.g., __pycache__, .mypy_cache) to keep the project environment clean.

- **lint**: Execute the commands `flake8 .` and `mypy . --warn-return-any --warn-unused-ignores --ignore-missing-imports --disallow-untyped-defs --check-untyped-defs`
- **lint-strict** (optional): Execute the commands `flake8 .` and `mypy . --strict`

IV.3 Additional Guidelines

- Create test programs to verify project functionality (not submitted or graded). Use frameworks like `pytest` or `unittest` for unit tests, covering edge cases.
- Include a `.gitignore` file to exclude Python artifacts.
- It is recommended to use virtual environments (e.g., venv or conda) for dependency isolation during development.

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

IV.3.1 Additional Requirements

- 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, outlines, etc.
- You need to use the following models:
  - **Qwen/Qwen3-0.6B** (default)
  - You can use other models as long as your project works 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 as the one `src` is in.
- The reviewer, as well as the moulinette, will just run `uv sync`.
- All errors should be handled gracefully. Your program must never crash unexpectedly and must always provide clear error messages to the user.

IV.3.2 Usage

Your program must be run using the following command (where `src` is the folder containing your files):
Running the program

```
uv run python -m src [--input <input_file>] [--output <output_file>]
```

By default, the program will read input files from the data/input/ directory and write output to the data/output/ directory. You can optionally specify custom paths using the --input and --output arguments. For example:

```
uv run python -m src --input data/input/example.json --output data/output/function_calling_results.
    json
```

Chapter V

Mandatory part

V.1 Summary

In this project, you will create a function calling tool that translates natural language prompts into structured function calls. Given a question like "What is the sum of 40 and 2?", your solution should not return 42, but instead provide:

- The function name: `fn_add_numbers`
- The arguments: `{"a": 40, "b": 2}`

Your implementation must use **constrained decoding** to guarantee 100% valid JSON output, ensuring near-perfect reliability even with a small 0.5B parameter model.

V.2 Input Files

Your solution will process two input files located in the `data/input/` directory:

- `function_calling_tests.json`: contains a JSON array of natural language prompts that your system must process.
  Example: function_calling_tests.json

```
[
    "What is the sum of 2 and 3?",
    "Reverse the string 'hello'",
    "Calculate the factorial of 5"
]
```

- `function_definitions.json`: contains the available functions your system can call. Each function includes:
  - Function name
  - Argument names and types
  - Return type
  - Description

  Example: function_definitions.json

```
[
    {
        "name": "fn_add_numbers",
        "description": "Add two numbers",
        "parameters": {
            "a": {"type": "number"},
            "b": {"type": "number"}
        },
        "returns": {"type": "number"}
    },
    {
        "name": "fn_reverse_string",
        "description": "Reverse a string",
        "parameters": {
            "s": {"type": "string"}
        },
        "returns": {"type": "string"}
    }
]
```

These examples establish the expected complexity level. However, your solution will be tested with different prompts and function sets. You must implement proper JSON error handling for input files, as they may contain invalid JSON or be missing entirely.

V.3 LLM Interaction

V.3.1 The LLM SDK

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

The SDK provides several essential methods:

- `get_logits_from_input_ids(input_ids: Tensor) -> Tensor`
  Takes an `input_ids` tensor and returns the raw logits after calling the LLM model.
- `get_path_to_vocabulary_json() -> str`
  Returns the path to a JSON file containing the structured correspondence between `input_ids` and tokens.
- `encode(text: str) -> List[int]`
  Encodes a text string into its corresponding list of token IDs using the model's tokenizer.
- `decode(token_ids: List[int]) -> str` *(optional)*
  Optionally decodes a list of token IDs back into a text string.

V.3.2 The Generation Pipeline

The LLM generation process follows these steps:

1. **Prompt**: Your natural language question
   Example: *"What is the sum of 2 and 3?"*

2. Tokenization: The text is broken into subword units (tokens). Unlike simple word splitting, tokenizers often include leading spaces, handle punctuation, and split words into smaller components using algorithms such as BPE or SentencePiece.
   Example (realistic): *["What", "Ġis", "Ġthe", "Ġsum", "Ġof", "Ġ2", "Ġand", "Ġ3", "?"]*
   Note: The symbol "Ġ", indicates a preceding space; real tokenizers preserve such details to reconstruct text accurately.

3. **Input IDs**: Tokens are converted to numerical IDs the model understands.
   Example (illustrative): *[892, 318, 262, 4771, 286, 16, 290, 17, 30]*

4. **LLM Processing**: The model processes these numbers through its neural network.

5. **Logits**: The model outputs probability scores for each possible next token.
   Example: *token_5: 0.001, token_42: 0.85, token_100: 0.02, ...*

6. **Token Selection**: The next token is chosen based on these probabilities, usually the one with the highest score.
   *At this stage, techniques like **constrained decoding** can be applied to restrict the token choices and ensure outputs follow a specific structure, such as generating 100% valid JSON.*

Important: This process repeats token-by-token. Each generated token is added to the prompt, and steps 2-6 repeat until the complete response is generated.

Simplified view:

```
Prompt -> Tokenization -> Input IDs -> LLM -> Logits -> Next Token Selection
```

V.3.3 Understanding Constrained Decoding

Language models generate text one token at a time. At each step, the model produces a probability distribution (logits) over all possible next tokens. Normally, you would sample from this distribution or pick the highest probability token.

Constrained decoding intervenes in this process by modifying the logits *before* token selection:

1. The model produces logits for all possible tokens.
2. You identify which tokens would maintain both a valid JSON structure *and compliance with the expected schema.*
3. You set logits for invalid tokens (those breaking the schema or structure) to negative infinity.
4. You sample only from the remaining valid tokens.

In this project, constrained decoding must not only ensure syntactically valid JSON but also enforce a specific schema. For instance, if the JSON field `"firmware"` can only take a few predefined values, the decoder should restrict token selection to those allowed options. This guarantees that every generated token maintains both structural and semantic validity, enforcing the required schema. As a result, the produced JSON is 100% retrievable and can always be parsed without errors.

Your solution must NOT rely on the model spontaneously producing correct JSON from a prompt. Prompting the model with function definitions and hoping for structured output is not reliable, and it is not the skill we expect you to develop here.

Think about how you can use the vocabulary JSON file to map between tokens and their string representations. This is crucial for determining which tokens are valid at each generation step.

V.4 Output File Format

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

- `prompt` (string): The original natural-language request
- `fn_name` (string): The name of the function to call
- `args` (object): All required arguments with the correct types

V.4.1 Example Output

```
[
    {
        "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"}
    }
]
```

V.4.2 Validation Rules

- The file must be valid JSON (no trailing commas, no comments)
- Keys and types must match the schema in `function_definitions.json` exactly
- No extra keys or prose are allowed anywhere in the output
- All required arguments must be present
- Argument types must match the function definition (number, string, boolean, etc.)

The given input files may change during the peer review. Do not hardcode solutions based on the provided examples.

V.5 Performance and Reliability

Your implementation should achieve:

- **Near-perfect accuracy**: 95%+ correct function selection and argument extraction
- **100% valid JSON**: Every output must be parseable and schema-compliant
- **Reasonable speed**: Process all test prompts in under 5 minutes on standard hardware
- **Robust error handling**: Gracefully handle malformed inputs, missing files, and edge cases

The Qwen3-0.6B model has only 500 million parameters, yet with proper constrained decoding, it can achieve reliability comparable to much larger models. This demonstrates the power of structural guidance over raw model size.

V.6 Testing Your Implementation

To verify your solution works correctly:

1. Ensure input files are in the `input/` directory
2. Run: `uv run python -m src`
3. Check that `output/function_calling_results.json` is created
4. Validate the JSON structure and content
5. Verify function names and argument types match the definitions

Test with various edge cases: empty strings, large numbers, special characters, ambiguous prompts, and functions with multiple parameters.

Chapter VI

Readme Requirements

A `README.md` file must be provided at the root of your Git repository. Its purpose is to allow anyone unfamiliar with the project (peers, staff, recruiters, etc.) to quickly understand what the project is about, how to run it, and where to find more information on the topic.
The `README.md` must include at least:

- The very first line must be italicized and read: *This project has been created as part of the 42 curriculum by <login1>[, <login2>[, <login3>[...]]].*
- A "**Description**" section that clearly presents the project, including its goal and a brief overview.
- An "**Instructions**" section containing any relevant information about compilation, installation, and/or execution.
- A "**Resources**" section listing classic references related to the topic (documentation, articles, tutorials, etc.), as well as a description of how AI was used — specifying for which tasks and which parts of the project.

➠ **Additional sections may be required depending on the project** (e.g., usage examples, feature list, technical choices, etc.).

Any required additions will be explicitly listed below.

For this project, the `README.md` must also include:

- **Algorithm explanation**: Describe your constrained decoding approach in detail
- **Design decisions**: Explain key choices in your implementation
- **Performance analysis**: Discuss accuracy, speed, and reliability of your solution
- **Challenges faced**: Document difficulties encountered and how you solved them
- **Testing strategy**: Describe how you validated your implementation
- **Example usage**: Provide clear examples of running your program

Your README must be written in English.

Chapter VII

Submission and peer review

Submit your assignment in your `Git` repository as usual. Only the work inside your repository will be reviewed during the defense. Don't hesitate to double-check the names of your files to ensure they are correct.

Your repository must contain:

- `src/` directory with your implementation
- `pyproject.toml` and `uv.lock` for dependency management
- `llm_sdk/` directory (copied from the provided package)
- `data/input/` directory with test files (for demonstration)
- `README.md` with comprehensive documentation
- Any additional files needed to run your solution

`Do not include the output/ directory in your repository. It will be generated during the peer review.`

During the evaluation, a brief **modification of the project** may occasionally be requested. This could involve a minor behaviour change, a few lines of code to write or rewrite, or an easy-to-add feature.

While this step may **not be applicable to every project**, you must be prepared for it if it is mentioned in the evaluation guidelines.

This step is meant to verify your actual understanding of a specific part of the project. The modification can be performed in any development environment you choose (e.g., your usual setup), and it should be feasible within a few minutes — unless a specific time frame is defined as part of the evaluation.

You can, for example, be asked to make a small update to a function or script, modify a display, or adjust a data structure to store new information, etc.

The details (scope, target, etc.) will be specified in the **evaluation guidelines** and may vary from one evaluation to another for the same project.