Empowering Full Stack Apps with LLM Function Calling

By Sunil Prajapati

12th Nov 2025

What is function calling?

What does it do?

A structured way for an LLM to trigger real-world operations through code rather than just generating text.

Function calling allows the model to respond with a well-defined JSON object that specifies *which* backend function should be executed and *what parameters* to use, rather than producing free-form natural language. This bridges the gap between “understanding” and “acting,” letting the LLM interface directly with your application logic, APIs, or databases through a controlled schema.

The LLM becomes a reasoning layer that decides when and how to invoke available tools.
Instead of executing functions itself, the model analyzes the user’s intent and selects an appropriate function from a list of developer-defined options. The backend then safely executes that function, returns the result, and the model uses the output to craft a final, human-readable response. This design keeps decision-making intelligent but execution secure.

When to use function calling?

What applications gets the benefit?

- **When your application needs to turn user intent into concrete backend actions.**

Function calling is ideal when users express requests in natural language but the system must respond by performing real operations such as querying a database, calling an external API, or executing a computation. It's most useful in apps where users say *what they want*, and the LLM intelligently decides *how to get it done* (e.g., "Book a flight from Bangkok to Tokyo tomorrow morning" → triggers your flight search API).

- **When you're building dynamic, data-driven, or multi-step assistant features.**

Any system that depends on live or contextual data like weather, finance, travel, health, or IoT dashboards. The LLM can interpret complex user prompts, choose which tools to call, and combine the responses into conversational answers. This enables features such as smart dashboards, customer service bots, AI tutors, or productivity assistants that interact with structured systems in real time.

How it works?

Architectual Workflow

- **1. Understanding the Intent**

The user provides a natural-language prompt such as *“Show me today’s weather in Kathmandu.”*

The LLM analyzes the text, extracts intent and key parameters (e.g., city: "Kathmandu"), and determines whether this request matches any available functions defined by the developer.

2. Deciding and Returning a Function Call

Instead of generating a plain-text answer, the LLM returns a **structured JSON** describing which function to call and with what arguments, such as:

```
{ "name": "getWeather", "arguments": { "city": "Kathmandu" } }
```

The backend receives this JSON, verifies its validity, and executes the corresponding local or remote function safely.

- **3. Executing and Responding Back**

The backend runs the actual function (e.g., calls a weather API), retrieves the result, and sends it back to the LLM.

Examples and Demo

Clone the repo

Available on GitHub

- https://github.com/scherbatsky-jr/function-calling
- Follow the instructions in Readme
- The chat controller is available in llmcontroller.js

Building JSON definition of function

Components

- Name
- Description
- Parameters
  - Type
  - Required
  - Additional info

Example methods and description

Dummy Methods

- For weather info:

```
function getWeather({ city, date }) {
  const base = store.weather[city] || { summary: "clear", tempC: 25 };
  return { city, date, ...base };
}
```

- JSON Function description:

```
{
    name: "getWeather",
    description: "Get current weather for a city (optionally for a date).",
    parameters: {
      type: "object",
      properties: {
        city: { type: "string", minLength: 1 },
```

More Examples

Multiple parameters

```
{
    name: "convertCurrency",
    description: "Convert an amount from one currency to another.",
    parameters: {
      type: "object",
      properties: {
      amount: { type: "number", minimum: 0 },
        from: { type: "string", minLength: 3, maxLength: 3 },
        to: { type: "string", minLength: 3, maxLength: 3 }
     },
      required: ["amount", "from", "to"],
      additionalProperties: false
   }
  },
```

Thank you