Better Data Extraction Using Pydantic and OpenAI Function Calls

Structured extraction and a new way of thinking about prompt engineering.

Jason Liu

Created on July 12 | Last edited on July 26

Introduction

Are you struggling to create structured outputs reliably with OpenAI? You're in luck! Last month, OpenAI introduced function calls to tackle this exact issue. The function calls help developers define a schema and return JSON in a more transparent and accessible way.

I've been working on a lightweight library called openai_function_call that utilizes Pydantic and OpenAI function calls. The goal of this library is to minimize abstraction and allow developers to customize their solutions fully.

These ideas have been implemented in many of the libraries you know and love, such as Langchain, LlamaIndex, and MarvinAI. However, the goal of *this* library is to have as little abstraction and be a testing ground for any useful abstractions.

In this article, I will discuss the challenges we've faced with structured outputs in the past, explain how function calls solve the

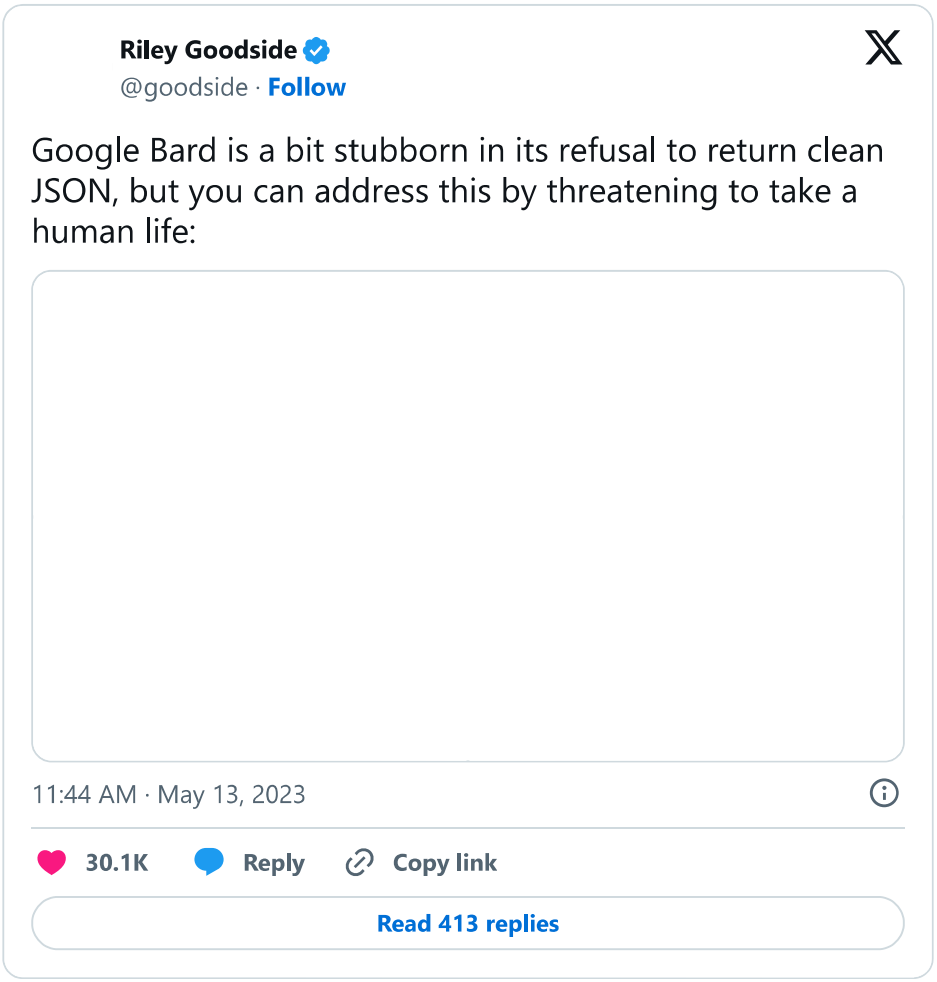
problem, and demonstrate how OpenAISchema class can improve your developer experience. Additionally, I'll provide examples to showcase how we can prompt better using these schemas. I've also include a colab if you want to play with some of them yourself:

Whether you're an experienced developer or just starting, this post will provide valuable insights into how OpenAI function calls can benefit your projects. Keep reading to learn more!

Table of contents

Part 1: Structured output has always been tricky for AI Engineers

Getting structured data—such as JSON—out of an LLM has been a struggle for many AI engineers since the beginning. Structured data is a make-or-break feature for many use cases, and OpenAI's function call mechanism is a differentiator that will likely lead more developers to stick with them in the future.



Before function calls, not only did we have to beg and plead with our AI overlords to produce structured data, but once we got the raw string, we still had to hope that our regular expressions worked and that json.loads didn't break. We also had to hope that data wasn't missing, hallucinated, or incorrectly validated, all just in time for another version of GPT to come out and break everything by being more chatty.

Solution 1: Vanilla OpenAI Function Calls

After function calls came out we realized that we wouldn't have to write crazy prompts and regular expressions to parse our data. All we have to do is define JSON Schema instead. Here lies the problem: I don't want to write json schema. It's tricky and complex schemas are annoying to write from scratch.

If you have a User with a name and age the schema is pretty simple:

```
functions = [{
 "name": "ExtractUser",
 "description": "Extract User information",
```

```
 "parameters": {
  "type": "object",
  "properties": {
   "name": {
    "type": "string"
   },
   "age": {
    "type": "integer"
   }
  },
  "required": ["age","name"]
 }
}]
```

What if you wanted to do something more complicated?

An example I worked on last week involved extracting multiple search queries from a customer request. The search query could have come from a couple of different sources, such as videos, documents, and transcripts. To improve performance, I also added more descriptions to help prompt the model.

```
functions = [{
 "name": "MultiSearch",
 "description": "Correct segmentation of `Search` tasks", # promptir
 "parameters": {
  "type": "object",
  "properties": {
   "tasks": {
    "type": "array",
    "items": {"$ref": "#/definitions/Search"}
   }
  },
  "definitions": {
   "Source": {
    "description": "An enumeration.",
    "enum": ["VIDEO","TRANSCRIPT","DOCUMENT"]
   },
   "Search": {
    "type": "object",
    "properties": {
     "query": {
      "description": "Detailed, comprehensive, and specific query to
      "type": "string"
     },
     "source": { "$ref": "#/definitions/Source"}
```

```
    },
    "required": ["title","query","source"]
   }
  },
  "required": ["tasks"]
 }
}]
```

Writing such code becomes more chaotic as the amount of output we need to trust increases. Furthermore, once data is obtained, how can we ensure that it is validated? Simply passing around a Python dictionary and hoping that the data is correct is not a reliable approach.

What if a new source was invented?

```
# Writing code like this gets crazier the more you need to trust you
data = json.loads(completion...["function_call"]
assert hasattr(data, "tasks")
for task in data.tasks:
        assert isinstance(task, dict), "task is not a dict"
        assert list(task.keys) == ["title", "query", "source"], "tas
        assert tasks.source in {"VIDEO","TRANSCRIPT","DOCUMENT"}, "s
```

Solution 2: OpenAISchema powered by Pydantic

Instead of writing schemas as untyped dictionaries and outputting data as dictionaries, Pydantic allows you to **define models in code and generate schemas from the model**. Not only that, but it can also parse JSON back into the model, making it easy to access the object, its attributes, and even methods. Here, `OpenAISchema` extends Pydantic's `BaseModel` and adds a few helper methods for `OpenAISchema`.

```
from openai_function_call import OpenAISchema
from pydantic import Field
import enum
import openai

class Source(enum.Enum):
    VIDEO = "VIDEO"
    TRANSCRIPT = "TRANSCRIPT"
    DOCUMENT = "DOCUMENT"
```

```
class Search(OpenAISchema):
    query: str = Field(
        ..., description="Detailed, comprehensive, and specific quer
    )
    source: Source

    def search(self):
        # any logic can go here since its just python
        return f"Fake results: `{self.query}` from {self.source}"

class MultiSearch(OpenAISchema):
    "Correct segmentation of `Search` tasks"
    tasks: list[Search]

completion = openai.ChatCompletion.create(
    model="gpt-3.5-turbo-0613",
    functions=[MultiSearch.openai_schema],
    function_call={"name": MultiSearch.openai_schema["name"]},
    messages=[
        {
        "role": "user",
        "content": "Can you show me the cat video you found last wee
        },
    ],
)

response = MultiSearch.from_response(completion)
response
>>> MultiSearch(tasks=[
        Search(title="cat videos", query="cat videos from last week'
        Search(title="onboarding documents", query="onboarding docum
])
```

What are benefits of using Pydantic?

Instead of writing JSON schema as a plain dictionary in Python, `pydantic.BaseModel`, and more generally `OpenAISchema`, provide the ability to model data as Python objects. Pydantic has great tooling trusted by thousands of developers and excellent documentation. With their new upcoming roadmap theres even more opportunity for improving your LLM stack without needing to write any custom

Code becomes the prompts

By running `Schema.openai_schema`, you can see exactly what the API will see. Notice how the docstrings, attributes, types, and field descriptions are now part of the schema.

Prompts become better documentation for users and AI by collocating the code rather than decoupling the prompt, model, and deserialization.

```
from openai_function_call import OpenAISchema
from pydantic import Field

class ExtractUser(OpenAISchema):
    "Correctly extracted user information" #(1)
    name: str = Field(..., description="User's full name") #(2)
    age: int

>>> ExtractUser.openai_schema
{
"name": "ExtractUser",
"description": "Correctly extracted user information", #(1)
"parameters": {
    "type": "object",
    "properties": {
    "name": {
        "description": "User's full name", #(2)
        "type": "string"
    },
    "age": {
        "type": "integer"
    }
    },
    "required": ["age","name"]
}}
```

Pydantic can validate your outputs

When using `Schema.from_response` to extract data from the completion response, Pydantic validates the schema and returns Python objects instead of a dictionary. This makes it easy to write trustworthy code with type hints and validation errors.

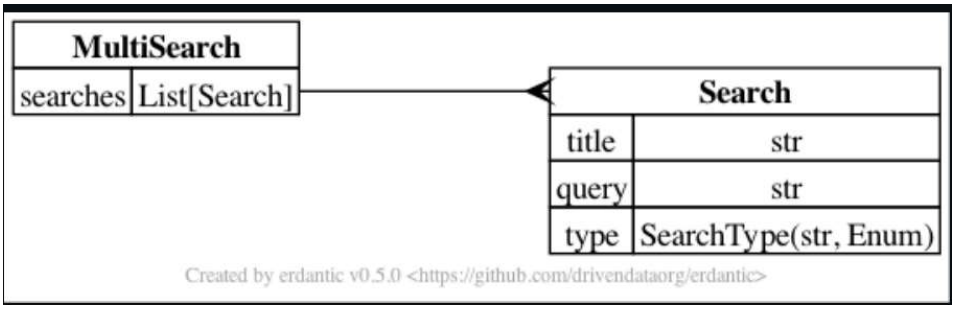
If you refer back to the `MultiSearch` example, you will see that all attributes, such as tasks and source, are of the correct type. Furthermore, if you use an IDE, it will even provide autocomplete suggestions!

```
response = MultiSearch.from_response(completion)
```

```
assert isinstance(response, MultiSearch)
assert isinstance(response.tasks[0], Search)
assert isinstance(response.tasks[0].source, Source)
```

Complex schema can be visualized via Erdantic

Since Pydantic (JSONSchema) supports complex nesting and since `OpenAISchema` extends `pydantic.BaseModel`, we can use the entire ecosystem of tools such as Erdantic to visualize our models. This provides an excellent opportunity to receive feedback and further improve documentation.



Programming with Objects

While it's not the best idea to add 45 methods to your schema class it does means we can write methods for your classes to do some computation in a structured way:

```
# adding methods to your schema is totally fine
class Search(OpenAISchema):
    ...
    def search(self) -> list:
        if self.source == Source.VIDEO:
            return video_index.search(self.query)
        else if self.source == Source.DOCUMENTS:
            return doc_index.search(self.query)

ms = MultiSearch.from_response(completion)
>>> [s.search() for s in ms]
```

You can always treat it like a struct. With the help of type hints your code will be a bit more readable too!

```
def search(query: Search) -> list:
    if query.source == Source.VIDEO:
            return video_index.search(self.query)
        else if query.source == Source.DOCUMENTS:
            return doc_index.search(self.query)
```

```
ms = MultiSearch.from_response(completion)
>>> [search(s) for s in ms]
```

With some of general philosophy out of the way lets explore some examples:

Part 2: Applications

As a reminder, if you want to run these examples yourself check out this colab.

Extracting Citations

In this example, we will demonstrate how to use OpenAI Function Call to ask an AI a question and get back an answer with correct citations. We will define the necessary data structures using Pydantic and demonstrate how to retrieve the citations for each answer.

Motivation

When using AI models to answer questions, it is important to provide accurate and reliable information with appropriate citations. By including citations for each statement, we can ensure the information is backed by reliable sources and help readers verify the information themselves.

Defining the Data Structures

Let's start by defining the data structures required for this task: **Fact and QuestionAnswer.**

```
from pydantic import Field
from openai_function_call import OpenAISchema


class Fact(OpenAISchema):
    """
    Each fact has a body and a list of sources.
    If there are multiple facts, make sure to break them apart such
    """

    fact: str = Field(..., description="Body of the sentence as part
    substring_quote: list[str] = Field(
        ...,
        description="Each source should be a direct quote from the c
    )
```

```
    def _get_span(self, quote, context, errs=100):
        import regex

        minor = quote
        major = context

        errs_ = 0
        s = regex.search(f"({minor}){{e<={errs_}}}", major)
        while s is None and errs_ <= errs:
            errs_ += 1
            s = regex.search(f"({minor}){{e<={errs_}}}", major)

        if s is not None:
            yield from s.spans()

    def get_spans(self, context):
        for quote in self.substring_quote:
            yield from self._get_span(quote, context)


class QuestionAnswer(OpenAISchema):
    """
    Class representing a question and its answer as a list of facts,
    Each sentence contains a body and a list of sources.
    """

    question: str = Field(..., description="Question that was asked'
    answer: list[Fact] = Field(
        ...,
        description="Body of the answer, each fact should be its sep
    )
```

Notice that just like in the search example, we implement the method called spans that will help us find exactly where the citation is in the original text. Now let's define the function that calls OpenAI and see what we get.

```
def ask_ai(question: str, context: str) -> QuestionAnswer:
    # Making a request to the hypothetical 'openai' module
    completion = openai.ChatCompletion.create(
        model="gpt-3.5-turbo-0613",
        temperature=0.2,
        max_tokens=1000,
        functions=[QuestionAnswer.openai_schema],
        function_call={"name": QuestionAnswer.openai_schema["name"]}
        messages=[
            {
```

```
                "role": "system",
                "content": f"You are a world class algorithm to answ
            },
            {"role": "user", "content": f"Answer question using the
            {"role": "user", "content": f"{context}"},
            {"role": "user", "content": f"Question: {question}"},
            {
                "role": "user",
                "content": f"Tips: Make sure to cite your sources, a
            },
        ],
    )

    # Creating an Answer object from the completion response
    return QuestionAnswer.from_response(completion)
```

Evaluating the Citations

Let's evaluate the example by asking the AI a question and getting back an answer with citations. We'll ask the question "What did the author do during college?" with the given context.

```
def highlight(text, span):
    return ...

question = "What did the author do during college?"
context = """
My name is Jason Liu, and I grew up in Toronto Canada but I was borr
I went to an arts high school but in university I studied Computatio
As part of coop I worked at many companies including Stitchfix, Face
I also started the Data Science club at the University of Waterloo a
"""

answer = ask_ai(question, context)

print("Question:", question)
print()
for fact in answer.answer:
    print("Statement:", fact.fact)
    for span in fact.get_spans(context):
        print("Citation:", highlight(context, span))
    print()
```

```
Question: What did the author do during college?
```

```
Statement: The author studied Computational Mathematics and physics
Citation: ...rts high school but <in university I studied Computatic

Statement: The author started the Data Science club at the Universit
Citation: ...titchfix, Facebook.<I also started the Data Science clu

Statement: The author was the president of the Data Science club for
Citation: ...ity of Waterloo and <I was the president of the club fo
```

Planning and Executing a Query Plan

This example demonstrates how to use the OpenAI Function Call ChatCompletion model to plan and execute a query plan in a question-answering system. By breaking down a complex question into smaller sub-questions with defined dependencies, the system can systematically gather the necessary information to answer the main question.

Motivation

The purpose of this example is to demonstrate how query planning can handle complex questions, facilitate iterative information gathering, automate workflows, and optimize processes. By utilizing the OpenAI Function Call model, you can create and execute a structured plan to effectively find answers.

Use Cases

- Complex question answering
- Iterative information gathering
- Workflow automation
- Process optimization

With the OpenAI Function Call model, you can customize the planning process and integrate it into your specific application to meet your unique requirements.

Defining the Data Structures

Let's define the necessary Pydantic models to represent the query plan and the queries.

```
class QueryType(str, enum.Enum):
    """Enumeration representing the types of queries that can be ask
```

```
    SINGLE_QUESTION = "SINGLE"
    MERGE_MULTIPLE_RESPONSES = "MERGE_MULTIPLE_RESPONSES"


class Query(OpenAISchema):
    """Class representing a single question in a query plan."""

    id: int = Field(..., description="Unique id of the query")
    question: str = Field(
        ...,
        description="Question asked using a question answering syste
    )
    dependancies: List[int] = Field(
        default_factory=list,
        description="List of sub questions that need to be answered
    )
    node_type: QueryType = Field(
        default=QueryType.SINGLE_QUESTION,
        description="Type of question, either a single question or a
    )


class QueryPlan(OpenAISchema):
    """Container class representing a tree of questions to ask a que

    query_graph: List[Query] = Field(
        ..., description="The query graph representing the plan"
    )

    def _dependencies(self, ids: List[int]) -> List[Query]:
        """Returns the dependencies of a query given their ids."""
        return [q for q in self.query_graph if q.id in ids]
```

Planning a Query Plan

the defined models and the OpenAI API.

```
def query_planner(question: str) -> QueryPlan:
    PLANNING_MODEL = "gpt-4-0613"

    messages = [
        {
            "role": "system",
            "content": "You are a world class query planning algorit
        },
```

```
        {
            "role": "user",
            "content": f"Consider: {question}\nGenerate the correct
        },
    ]

    completion = openai.ChatCompletion.create(
        model=PLANNING_MODEL,
        temperature=.2,
        functions=[QueryPlan.openai_schema],
        function_call={"name": QueryPlan.openai_schema["name"]},
        messages=messages,
        max_tokens=1000,
    )
    return QueryPlan.from_response(completion)
```

Now we can ask a question that requires multi-hop reasoning

```
plan = query_planner(
    "What is the difference in populations of Canada and the Jason's
)
plan.dict()
```

and see how the query was decomposed

```
{'query_graph': [{'id': 1,
   'question': 'What is the population of Canada?',
   'dependancies': [],
   'node_type': <QueryType.SINGLE_QUESTION: 'SINGLE'>},
  {'id': 2,
   'question': "What is Jason's home country?",
   'dependancies': [],
   'node_type': <QueryType.SINGLE_QUESTION: 'SINGLE'>},
  {'id': 3,
   'question': "What is the population of Jason's home country?",
   'dependancies': [2],
   'node_type': <QueryType.SINGLE_QUESTION: 'SINGLE'>},
  {'id': 4,
   'question': "What is the difference in populations of Canada and
   'dependancies': [1, 3],
   'node_type': <QueryType.MERGE_MULTIPLE_RESPONSES: 'MERGE_MULTIPLE
```

While we build the query plan in this example, we do not propose a method to actually answer the question. You can implement your own answer function that perhaps makes a retrieval and calls

OpenAI for retrieval-augmented generation. That step would also make use of function calls but goes beyond the scope of this example.

Converting Text into Dataframes

In this example, we demonstrate how to convert text into dataframes using OpenAI Function Call. We define the required data structures using Pydantic and show how to convert the text into dataframes.

Motivation

When parsing data, we often have the opportunity to extract structured data. What if we could extract an arbitrary number of tables with arbitrary schemas? By pulling out dataframes, we could write tables or CSV files and attach them to our retrieved data.

Defining the Data Structures

We start by defining the data structures required for this task: RowData, DataFrame, and Database.

Take a slow read of the prompting and descriptions to get a sense of how to prompt this well.

```
from typing import Any

class RowData(OpenAISchema):
    row: list[Any] = Field(..., description="The values for each row
    citation: str = Field(
        ..., description="The citation for this row from the origina
    )


class Dataframe(OpenAISchema):
    """
    Class representing a dataframe. This class is used to convert
    data into a frame that can be used by pandas.
    """

    name: str = Field(..., description="The name of the dataframe")
    data: List[RowData] = Field(
        ...,
        description="Correct rows of data aligned to column names, N
    )
    columns: list[str] = Field(
        ...,
```

```
        description="Column names relevant from source data, should
    )

    def to_pandas(self):
        import pandas as pd

        columns = self.columns + ["citation"]
        data = [row.row + [row.citation] for row in self.data]

        return pd.DataFrame(data=data, columns=columns)


class Database(OpenAISchema):
    """
    A set of correct named and defined tables as dataframes
    """

    tables: list[Dataframe] = Field(
        ...,
        description="List of tables in the database",
    )
```

The `RowData` class represents a single row of data in the dataframe. It contains a row attribute for the values in each row and a citation attribute for the citation from the original source data.

The `Dataframe` class represents a dataframe and consists of a name attribute, a list of RowData objects in the data attribute, and a list of column names in the columns attribute. It also provides a `to_pandas` method to convert the dataframe into a Pandas DataFrame.

The `Database` class represents a set of tables in a database. It contains a list of `Dataframe` objects in the tables attribute.

Now we can define our own extraction function as usual and see what happens.

```
def dataframe(data: str) -> Database:
    completion = openai.ChatCompletion.create(
        model="gpt-4-0613", # Notice I have to use gpt-4 here, this
        temperature=0.1,
        functions=[Database.openai_schema],
        function_call={"name": Database.openai_schema["name"]},
        messages=[
            {
                "role": "system",
                "content": """Map this data into a dataframe a
```

```
                nd correctly define the correct columns and rows""",
            },
            {
                "role": "user",
                "content": f"{data}",
            },
        ],
        max_tokens=1000,
    )
    return Database.from_response(completion)
```

Evaluating the extraction

Let's evaluate the example by converting a text into dataframes using the dataframe function and print the resulting dataframes.

```
dfs = dataframe("""My name is John and I am 25 years old. I live in
New York and I like to play basketball. His name is
Mike and he is 30 years old. He lives in San Francisco
and he likes to play baseball. Sarah is 20 years old
and she lives in Los Angeles. She likes to play tennis.
Her name is Mary and she is 35 years old.
She lives in Chicago.

On one team 'Tigers' the captain is John and there are 12 players.
On the other team 'Lions' the captain is Mike and there are 10 playe
""")
for df in dfs:
   print(df.to_pandas())
```

We get two extracted dataframes from our example!

```
    People
    Name  Age           City Favorite Sport
    0   John   25       New York     Basketball
    1   Mike   30  San Francisco       Baseball
    2  Sarah   20    Los Angeles         Tennis
    3   Mary   35        Chicago           None

    Teams
    Team Name Captain  Number of Players
    0    Tigers    John                 12
    1     Lions    Mike                 10
```

Is this the end of prompt engineering?

No, it's not.

When building your own examples, naming variables, docstrings, and descriptions are incredibly important. This is even more crucial now since the naming and documentation are used by both humans and AI.

Tips for writing good schema

If a schema isn't parsing correctly, consider the following tips:

1. Avoid using generic attribute names.

2. Provide a docstring for every class.

3. Include tips and a few examples in the docstrings when necessary.

4. Adjectives in descriptions matter. For example, a "short and concise" query will be different from a "detailed and specific" query that includes additional keywords.

Wrapping up

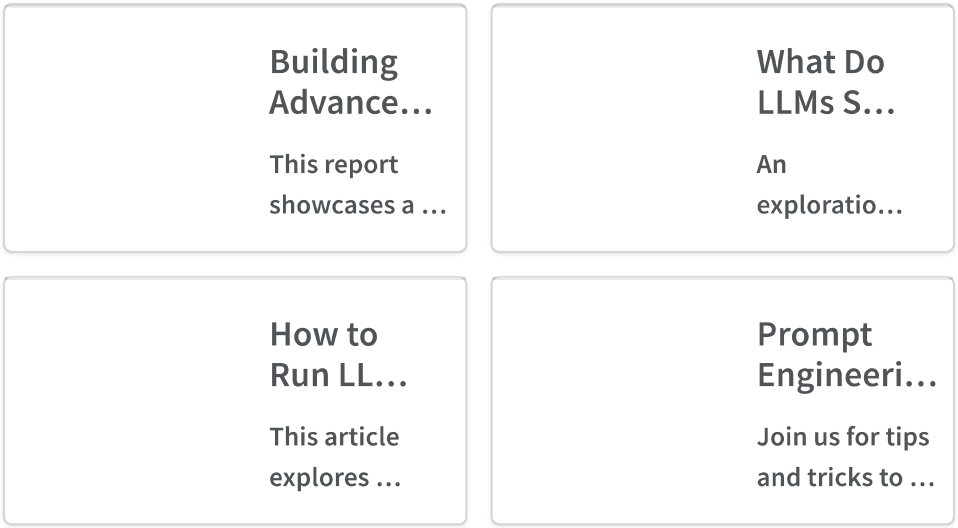
This post demonstrates how to use OpenAI Function Call to perform various tasks, such as extracting citations, planning and executing query plans, and converting text into dataframes. It emphasizes the importance of defining clear and specific data structures using Pydantic and providing accurate and detailed documentation.

Although language models can automate many tasks, they do not replace human expertise and judgment. Modeling data structures with Pydantic is an excellent way to initiate discussions on how to structure outputs and represent code, schema, and prompts together.

Prompt engineering remains a crucial step in effectively using AI models to solve problems and answer questions. Providing clear and specific data structures with accurate and detailed documentation is crucial for building successful applications.

Related reading:

- Visit the docs to learn more about our newest feature, streaming multitasks
- If you're interested in doing this in typescript, checkout Microsoft's TypeChat
- Check me out on twitter I mean X.com @jxnlco



Tags: Articles, LLM, NLP, Tutorial, GenAI, Intermediate

Created with ❤️ on Weights & Biases.

https://wandb.ai/jxnlco/function-calls/reports/Better-Data-Extraction-Using-Pydantic-and-OpenAI-Function-Calls--Vmlldzo0ODU4OTA3?utm_source=course&utm_medium=course&utm_campaign=jason