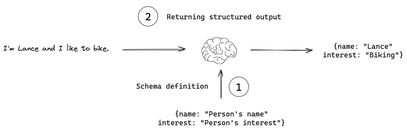
Structured outputs

Overview

For many applications, such as chatbots, models need to respond to users directly in natural language. However, there are scenarios where we need models to output in a *structured format*. For example, we might want to store the model output in a database and ensure that the output conforms to the database schema. This need motivates the concept of structured output, where models can be instructed to respond with a particular output structure.



Key concepts

(1) Schema definition: The output structure is represented as a schema, which can be defined in several ways. **(2) Returning structured output:** The model is given this schema, and is instructed to return output that conforms to it.

Recommended usage

This pseudo-code illustrates the recommended workflow when using structured output. LangChain provides a method, `withStructuredOutput()`, that automates the process of binding the schema to the model and parsing the output. This helper function is available for all model providers that support structured output.

```
// Define schema
const schema = { foo: "bar" };
// Bind schema to model
const modelWithStructure = model.withStructuredOutput(schema);
// Invoke the model to produce structured output that matches the schema
const structuredOutput = await modelWithStructure.invoke(userInput);
```

Schema definition

The central concept is that the output structure of model responses needs to be represented in some way. While types of objects you can use depend on the model you're working with, there are common types of objects that are typically allowed or recommended for structured output in TypeScript.

The simplest and most common format for structured output is a Zod schema definition:

```
import { z } from "zod";

const ResponseFormatter = z.object({
  answer: z.string().describe("The answer to the user's question"),
  followup_question: z
    .string()
    .describe("A followup question the user could ask"),
});
```

You can also define a JSONSchema object, which is what Zod schemas are converted to internally before being sent to the model provider:

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/product.schema.json",
  "title": "ResponseFormatter",
  "type": "object",
  "properties": {
    "answer": {
      "description": "The answer to the user's question",
      "type": "string"
    },
    "followup_question": {
      "description": "A followup question the user could ask",
      "type": "string"
    }
  },
  "required": ["answer", "followup_question"]
}
```

Returning structured output

With a schema defined, we need a way to instruct the model to use it. While one approach is to include this schema in the prompt and *ask nicely* for the model to use it, this is not recommended. Several more powerful methods that utilizes native features in the model provider's API are available.

Using tool calling

Many model providers support tool calling, a concept discussed in more detail in our tool calling guide. In short, tool calling involves binding a tool to a model and, when appropriate, the model can *decide* to call this tool and ensure its response conforms to the tool's schema. With this in mind, the central concept is straightforward: *create a tool with our schema and bind it to the model!* Here is an example using the `ResponseFormatter` schema defined above:

```
import { ChatOpenAI } from "@langchain/openai";

const model = new ChatOpenAI({
  modelName: "gpt-4",
  temperature: 0,
});

// Create a tool with ResponseFormatter as its schema.
const responseFormatterTool = tool(async () => {}, {
  name: "responseFormatter",
  schema: ResponseFormatter,
});

// Bind the created tool to the model
const modelWithTools = model.bindTools([responseFormatterTool]);

// Invoke the model
const aiMsg = await modelWithTools.invoke(
  "What is the powerhouse of the cell?"
);
```

JSON mode

In addition to tool calling, some model providers support a feature called `JSON mode`. This supports JSON schema definition as input and enforces the model to produce a conforming JSON output. You can find a table of model providers that support JSON mode here. Here is an example of how to use JSON mode with OpenAI:

```
import { ChatOpenAI } from "@langchain/openai";

const model = new ChatOpenAI({
  model: "gpt-4",
}).bind({
  response_format: { type: "json_object" },
});

const aiMsg = await model.invoke(
  "Return a JSON object with key 'random_nums' and a value of 10 random numbers in [0-99]"
);
console.log(aiMsg.content);
// Output: {
//   "random_nums": [23, 47, 89, 15, 34, 76, 58, 3, 62, 91]
// }
```

One important point to flag: the model *still* returns a string, which needs to be parsed into a JSON object. This can, of course, simply use the `json` library or a JSON output parser if you need more advanced functionality. See this how-to guide on the JSON output parser for more details.

```
import json
const jsonObject = JSON.parse(aiMsg.content)
// {'random_ints': [23, 47, 89, 15, 34, 76, 58, 3, 62, 91]}
```

Structured output method

There are a few challenges when producing structured output with the above methods:

(1) If using tool calling, tool call arguments needs to be parsed from an object back to the original schema.

(2) In addition, the model needs to be instructed to *always* use the tool when we want to enforce structured output, which is a provider specific setting.

(3) If using JSON mode, the output needs to be parsed into a JSON object.

With these challenges in mind, LangChain provides a helper function (`withStructuredOutput()`) to streamline the process.

This both binds the schema to the model as a tool and parses the output to the specified output schema.

```
// Bind the schema to the model
const modelWithStructure = model.withStructuredOutput(ResponseFormatter);
// Invoke the model
const structuredOutput = await modelWithStructure.invoke(
  "What is the powerhouse of the cell?"
);
// Get back the object
console.log(structuredOutput);
// { answer: "The powerhouse of the cell is the mitochondrion. Mitochondria are organelles that
```

[FURTHER READING]

For more details on usage, see our how-to guide.

Was this page helpful?

You can also leave detailed feedback on GitHub.