Generating Structured Outputs from LLMs

An overview of popular techniques to confine LLMs' output to a predefined schema

Ibrahim Habib

2025-07-26

Table of contents

Today, the most common interface for interacting with LLMs is through the classic chat UI found in ChatGPT, Gemini, or DeepSeek. The interface is quite simple, where the user inputs a body of text and the model responds with another body, which may or may not follow a specific structure. Since humans can understand unstructured natural language, this interface is suitable and quite effective for the target audience it was designed for.

However, the user base of LLMs is much larger than the 8 billion humans living on Earth. It expands to millions of software programs that can potentially harness the power of such large generative models. Unlike humans, software programs cannot understand unstructured data, preventing them from exploiting the knowledge generated by these neural networks.

To address this issue, various techniques have been developed to generate outputs from LLMs following a predefined schema. This article will overview three of the most popular approaches for producing structured outputs from LLMs. It is written for engineers interested in integrating LLMs into their software applications.

1 Structured Output Generation

Structured output generation from LLMs involves using these models to produce data that adheres to a predefined schema, rather than generating unstructured text. The schema can be defined in various formats, with JSON and regex being the most common. For example, when utilizing JSON format, the schema specifies the expected keys and the data types (such as int, string, float, etc.) for each value. The LLM then outputs a JSON object that includes only the defined keys and correctly formatted values.

There are various situations where structured output is needed from LLMs. Formatting unstructured bodies of text is one large application area of this technology. You can use a model to extract specific information from large bodies of text or even images (using VLMs). For example, you can use a general VLM to extract the purchase date, total price, and store name from receipts.

There are various techniques to generate structured outputs from LLMs. This article will discuss three.

1. Relying on API Providers
2. Prompting and Reprompting Strategies
3. Constrained Decoding

2 Relying on API Providers *'Magic'*

Multiple LLM service API providers, including OpenAI and Google's Gemini, allow users to define a schema for the model's output. This schema is usually defined using a Pydantic class and provided to the API endpoint. If you are using LangChain, you can follow this tutorial to integrate structured outputs into your application.

Simplicity is the greatest aspect of this particular approach. You define the required schema in a manner familiar to you, pass it to the API provider, and sit back and relax as the service provider performs all the *magic* for you.

Using this technique, however, will limit you to using only API providers that provide the described service. This limits the growth and flexibility of your projects, as it shuts the door to using multiple models, particularly open source ones. If the API providers suddenly decide to spike the price of the service, you will be forced either to accept the extra costs or look for another provider.

Moreover, it isn't exactly *Hogwarts Magic* that the service provider does. The provider follows a certain approach to generate the structured output for you. Knowledge of the underlying technology will facilitate the app development and accelerate the debugging process and error understanding. For the mentioned reasons, grasping the underlying science is probably worth the effort.

3 Prompting and Reprompting-Based Techniques

If you have chatted with an LLM before, then this technique is probably on your mind. If you want a model to follow a certain structure, just tell it to do so! In the system prompt, instruct the model to follow a certain structure, provide a few examples, and ask it not to add any additional text or description.

After the model responds to the user request and the system receives the output, you should use a parser to transform the sequence of bytes to an appropriate representation in the system. If parsing succeeds, then congratulate yourself and thank the power of prompt engineering. If parsing fails, then your system will have to recover from the error.

3.1 Prompting is Not Enough

The problem with prompting is unreliability. On its own, prompting is not enough to trust a model to follow a required structure. It might add extra explanation, disregard certain fields, and use an incorrect data type. Prompting can be and should be coupled with error recovery techniques that handle the case where the model defies the schema, which is detected by parsing failure.

Some people might think that a parser acts like a boolean function. It takes a string as input, checks its adherence to predefined grammar rules, and returns a simple *'yes'* or *'no'* answer. In reality, parsers are more complex than that and provide much richer information than '*follows'* or *'does not follow'* structure.

Parsers can detect mistakes and incorrect tokens in input text according to grammar rules (Aho et al., 2007, pp. 192–196). This information provides us with valuable information on the specifics of misalignments in the input string. For example, the parser is what detects a missing semicolon error when you're running Java code.

Armed with knowledge of mistakes in the generated outputs, we can reprompt the model to generate the outputs in the desired format while avoiding the detected mistake. The new prompt is made by augmenting the original prompt with specific information on the observed mistake and explicitly asking the model to avoid it.

Figure 1 depicts the flow used in the prompting-based techniques.

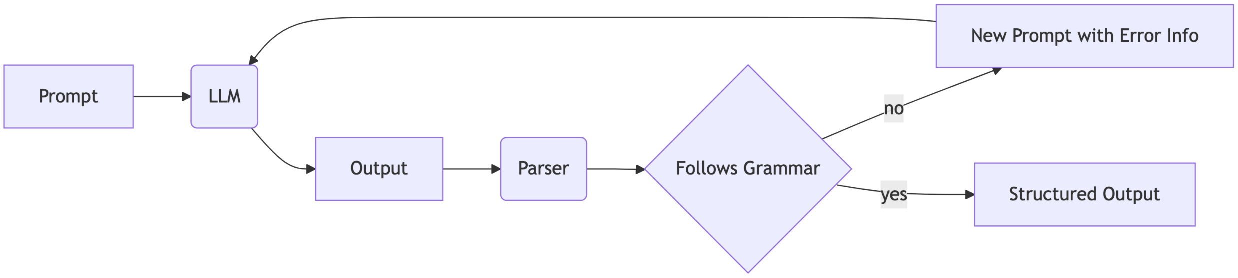

Figure 1: General Flow of Prompting and Reprompting techniques

3.2 Prompting Tools

One of the most popular libraries for prompt based structured output generation from LLMs is instructor. Instructor is a Python library with over 11k stars on GitHub. It supports data definition with Pydantic, integrates with over 15 providers, and provides automatic retries on parsing failure. In addition to Python, the package is also avillable in TypeScript, Go, Ruby, and Rust. [1]

The beauty of Instructor lies in its simplicity. All you need is to define a Pydantic class, initialize a client using only its name and API key (if required), and pass your request. The sample code below, from the docs, displays the simplicity of Instructor.

```
import instructor
from pydantic import BaseModel
from openai import OpenAI


class Person(BaseModel):
    name: str
    age: int
    occupation: str


client = instructor.from_openai(OpenAI())
person = client.chat.completions.create(
    model="gpt-4o-mini",
    response_model=Person,
    messages=[
```

[1] All information on the package is based on its documentation at the time of writing this article.

```
        {
          "role": "user",
          "content": "Extract: John is a 30-year-old software engineer"
        }
    ],
)
print(person)  # Person(name='John', age=30, occupation='software engineer')
```

3.3 The Cost of Reprompting

As convenient as the reprompting technique might be, it comes at a hefty cost. LLM usage cost, either service provider API costs or GPU usage, scales linearly with the number of input tokens and the number of generated tokens.

As mentioned earlier prompting based techniques might require reprompting. The reprompt will have roughly the same cost as the original one. Hence, the cost scales linearly with the number of reprompts.

If you're going to use this technique, you have to keep the cost problem in mind. No one wants to be surprised by a large bill from an API provider. One idea to help cut surprising costs is to put emergency brakes into the system by applying a hard-coded limit on the number of allowed reprompts. This will help you put an upper limit on the costs of a single prompt and reprompt cycle.

4 Constrained Decoding

Unlike the prompting, constrained decoding doesn't need retries to generate a valid, structure-following output. Constrained decoding utilizes computational linguistics techniques and knowledge of the token generation process in LLMs to generate outputs that are guaranteed to follow the required schema.

4.1 How it works?

LLMs are autoregressive models. They generate one token at a time and the generated tokens are used as inputs to the same model.

The last layer of an LLM is basically a logistic regression model that calculates for each token in the model's vocabulary the probability of it following the input sequence. The model calculates the logits value for each token, then using the softmax function, these value are scaled and transformed to probability values.

Constrained decoding produces structured outputs by limiting the available tokens at each generation step. The tokens are picked so that the final output obeys the required structure. To figure out how the set of possible next tokens can be determined, we need to visit RegEx.

Regular expressions, RegEx, are used to define specific patterns of text. They are used to check if a sequence of text matches an expected structure or schema. So basically, RegEx is a language that can be used to define expected structures from LLMs. Because of its popularity, there is a wide array of tools and libraries that transforms other forms of data structure definition like Pydantic classes and JSON to RegEx. Because of its flexibility and the wide availability of conversion tools, we can transform our goal now and focus on using LLMs to generate outputs following a RegEx pattern.

4.1.1 Deterministic Finite Automata (DFA)

One of the ways a RegEx pattern can be compiled and tested against a body of text is by transforming the pattern into a deterministic finite automata (DFA). A DFA is simply a state machine that is used to check if a string follows a certain structure or pattern.

A DFA consists of 5 components.

1. A set of tokens (called the alphabet of the DFA)
2. A set of states
3. A set of transitions. Each transition connects two states (maybe connecting a state with itself) and is annotated with a token from the alphabet
4. A start state (marked with an input arrow)
5. One or more final states (marked as double circles)

A string is a sequence of tokens. To test a string against the pattern defined by a DFA, you begin at the start state and loop over the string's tokens, taking the transition corresponding to the token at each move. If at any point you have a token for which no corresponding transition exists from the current state, parsing fails and the string defies the schema. If parsing ends at one of the final states, then the string matches the pattern; otherwise it also fails.

For example, the string `abab` matches the pattern in Figure 2 because starting at `q0` and following the transitions marked with `a`, `b`, `a`, and `b` in this order will land us at `q2`, which is a final state.

On the other hand, the string `abba` doesn't match the pattern because its path ends at `q0` which isn't a final state.

A great thing about RegEx is that it can be compiled into a DFA; after all they are just two different ways to specify patterns. Discussion of such transformation is out of scope for this article. The interested reader can check Aho et al. (2007, pp. 152–166) for a discusion of 2 techniques to perform the transformation.

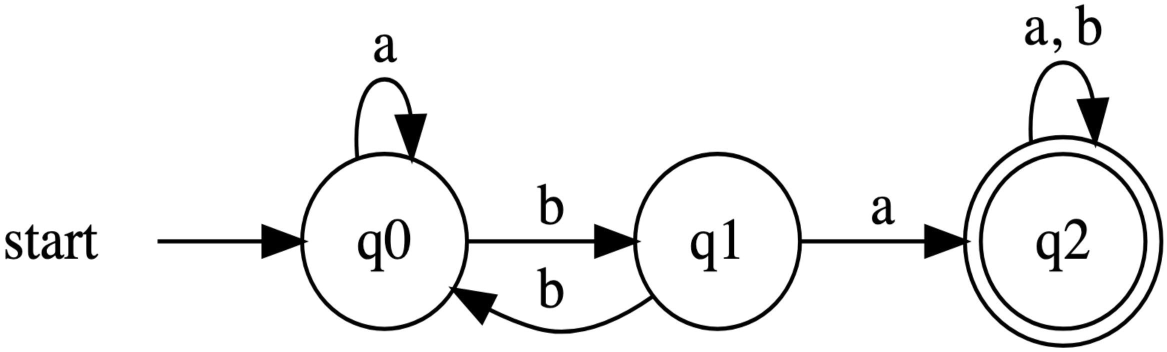

Figure 2: Example for a DFA with the alphabet `{a, b}`, states `{q0, q1, q2}`, and a single finale state, `q2`.

4.1.2 DFA for Valid Next Tokens Set

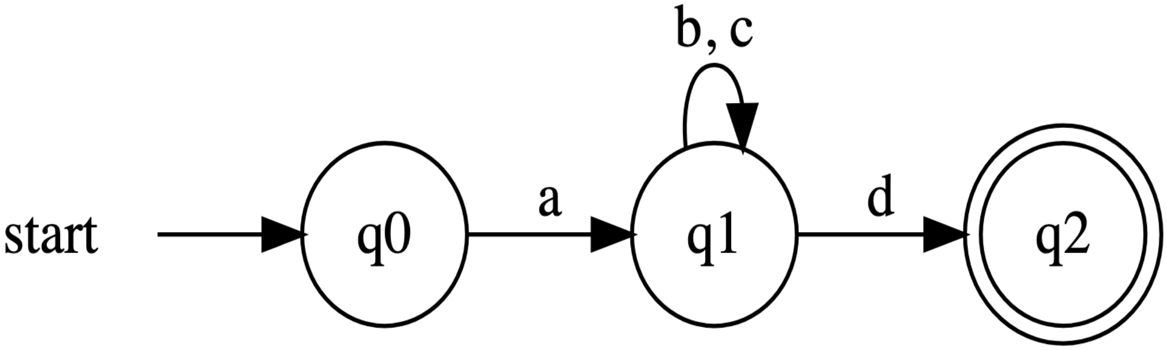

Figure 3: Example for a DFA generated from the RegEx `a(b|c)*d`

Let's recap what we have reached so far. We wanted a technique to figure out the set of valid next tokens to follow a certain schema. We defined the schema using RegEx and transformed it into a DFA. Now we are going to show that a DFA informs us of the set of possible tokens at any point during parsing, fitting our requirements and needs.

After building the DFA, we can easily determine in $O(1)$ the set of valid next tokens while standing at any state. It is the set of tokens annotating any transition exiting from the current state.

Consider the DFA in Figure 3, for example. The following table shows the set of valid next token while standing at each state.

State	Valid Next Tokens
q0	{a}
q1	{b, c, d}
q2	{}

4.1.3 Applying the DFA to LLMs

Getting back to our structured output from LLMs problem, we can transform our schema to a RegEx then to a DFA. The alphabet of this DFA will be set to the LLM's vocabulary (the set of all tokens the model can generate). While the model generates tokens, we will move through the DFA, starting at the start state. At each step, we will be able to determine the set of valid next tokens.

The trick now happens at the softmax scaling stage. By zeroing out the logits of all tokens that are not in the valid tokens set, we will calculate probabilities only for valid tokens, forcing the model to generate a sequence of tokens that follows the schema. That way, we can generate structured outputs with zero additional costs!

4.2 Constrained Decoding Tools

One of the most popular Python libraries for constrained decoding is Outlines (Willard & Louf, 2023). It is very simple to use and integrates with many LLM providers like OpenAI, Anthropic, Ollama, and vLLM.

You can define the schema using a Pydantic class, for which the library handles the RegEx transformation, or directly using a RegEx pattern.

```
from pydantic import BaseModel
from typing import Literal
import outlines
import openai

class Customer(BaseModel):
    name: str
    urgency: Literal["high", "medium", "low"]
    issue: str

client = openai.OpenAI()
model = outlines.from_openai(client, "gpt-4o")

customer = model(
    "Alice needs help with login issues ASAP",
    Customer
)
#   Always returns valid Customer object
#   No parsing, no errors, no retries
```

The code snippet above from the docs displays the simplicity of using Outlines. For more information on the library, you can check the docs and the dottxt blogs.

5 Conclusion

Structured output generation from LLMs is a powerful tool that expands the possible use cases of LLMs beyond the simple human chat. This article discussed three approaches: relying on API providers, prompting and reprompting strategies, and constrained decoding. For most scenarios, constrained decoding is the favoured method because of its flexibility and low cost. Moreover, the existence of popular libraries like Outlines simplifies the introduction of constrained decoding to software projects.

If you want to learn more about constrained decoding, then I would highly recommend this course from deeplearning.ai and dottxt, the creators of Outlines library. Using videos and code examples, this course will help you get hands-on experience getting structured outputs from LLMs using the techniques discussed in this post.

5.1 References

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, techniques, & tools*. Pearson/Addison Wesley.

Willard, B. T., & Louf, R. (2023). *Efficient guided generation for large language models*. https://arxiv.org/abs/2307.09702