A Gentle Introduction to Structured Generation with Anthropic API

Building Reproducible LLM Applications

Welcome to our short series of articles on structured generation, a paradigm designed to reduce the unpredictability of large language models (LLMs) by ensuring that their outputs adhere to predefined formats.

The goal of the series is to show how structured generation can be implemented in Python using Anthropic's LLMs inference API. As a first step, this post introduces the basic concepts behind structured generation. In addition, it provides a practical example that guides Anthropic's Claude 3.5 Sonnet model to produce structured data in many formats.

In follow-up tutorials, we will explore assistant response prefilling and function calling, as more advanced techniques for structured generation. We'll also demonstrate how to generate and validate schemas efficiently with the Python Pydantic library, and we will wrap up the series with a complete example of legal text classification.

By the end of it, you will have gained concrete strategies for building more robust LLM-powered applications, and a theoretical understanding of LLMs.

Enrica Troiano[1] and Tommaso Furlanello[12]

[1] HK3Lab [2] Tribe AI

Correspondence: {name}.{surname}@hk3lab.ai

You can find the raw markdown file for the post and the complete code for the examples in our github repository. To run the code yourself, simply clone the repository and execute the Jupyter notebook gentle-intro.ipynb.

A GENTLE INTRODUCTION TO STRUCTURED GENERATION

In December 2022, we hailed a new generation of large language models (LLMs) as a godsend for far-reaching technological revolutions. The linguistic skills of these models had become so astoundingly human-like, and their knowledge so exceptional in breadth, that they could feature at the same time buddies to chat with and powerful tools to lead industrial progress. Usurprisingly, today LLMs represent a key workforce in many enterprises, but the very feature that makes them so powerful (their ability to generate unstructured text in natural language) poses new challenges.

The issue with LLMs is that they can be inconsistent. For example, anyone who has used Claude (Anthropic's LLM-powered virtual assistant) knows that it can provide different answers when asked the same question repeatedly. Moreover, the format of the answers doesn't always align with the users' requests: often, a direct yes-or-no question is met with an elaborate explanation, and questions expecting a digit (e.g., 6) elicit responses in string form ("six"). While these irregularities aren't a big deal for human

readers, they can become problematic when chatbots are integrated with other systems that require consistent output formats.

This is where **structured generation** comes into play. Structured generation is a solution to make the LLMs' output format more predictable. Its fundamental idea is to narrow the possible LLM's answers down to the answers that have certain characteristics. In other words, we want to force the models to respect the structural constraints that we prefer. For instance, from a customer support chatbot we may require a response composed of two parts, one that is a direct answer for the customer's question, and the other that gives a detailed log and explanation for the company's database.

Implicitly, we exert some control over a LLM text generation process as soon as we prompt it with our queries: a prompt is the input condition that determines what tokens the model returns, i.e., what answer (from the space all possible answers) is appropriate to the user's input. But finding a prompt that elicits the desired output format can be a long trial-and-error process. We must experiment with prompt variations and see if the model's responses change accordingly. We may eventually find the wording that works the best for a specific task, and yet have no guarantee that it will systematically lead to the desired output format.

Structured generation complements and enhances prompt engineering, because it represents a more methodical approach to specify our expectations for the models, and thus control their outputs.

MOTIVATING EXAMPLES

Let's have a look at two examples to illustrate the most immediate advantage of obtaining structured outputs (i.e., their machine readability), as opposed to collecting LLM answers as they are.

Example 1: Without structured generation.

```
User: "What's the status of my order #12345?"

LLM: "I've checked your order #12345, and it's currently in transit. It
was shipped yesterday and is expected to arrive within 3-5 business days.
Is there anything else I can help you with?"
```

Processing this response would require complex parsing with custom regex to extract different types of information:

```
import re

def process_unstructured_response(response: str):
    """Only working for the specific example"""
    # Extract order number
    order_number_match = re.search(r'order #(\d+)', response)
    order_number = order_number_match.group(1) if order_number_match else
None

    # Extract status
    status_match = re.search(r"it's currently ([\w\s]+)", response)
```

```
    status = status_match.group(1) if status_match else None

    # Extract shipping information
    shipped_match = re.search(r'It was shipped (\w+)', response)
    shipped_date = shipped_match.group(1) if shipped_match else None

    # Extract estimated delivery time
    delivery_match = re.search(r'expected to arrive within (\d+-\d+)
(\w+)', response)
    if delivery_match:
        delivery_time = delivery_match.group(1)
        delivery_unit = delivery_match.group(2)
    else:
        delivery_time = None
        delivery_unit = None

    return {
        "order_number": order_number,
        "status": status,
        "shipped_date": shipped_date,
        "estimated_delivery": f"{delivery_time} {delivery_unit}" if
delivery_time and delivery_unit else None
    }

# Test the function
response = "I've checked your order #12345, and it's currently in transit.
It was shipped yesterday and is expected to arrive within 3-5 business
days. Is there anything else I can help you with?"
result = process_unstructured_response(response)
print(result)
```

```
{'order_number': '12345', 'status': 'in transit', 'shipped_date':
'yesterday', 'estimated_delivery': '3-5 business'}
```

In this example, the LLM provides a human-friendly response that answers the question but doesn't follow any particular structure. This is fine for direct human interaction but can be challenging for automated systems to parse and process.

Example 2: With structured generation.

```
User: "What's the status of my order #12345?"

LLM:
{
  "customer_response": "Your order #12345 is currently in transit. It was
shipped yesterday and is expected to arrive within 3-5 business days.",
  "internal_log": {
    "order_number": "12345",
```

```
    "status": "in_transit",
    "ship_date": "2023-04-15",
    "estimated_delivery": "2023-04-18 to 2023-04-20",
    "last_update": "2023-04-16T09:30:00Z",
    "notes": "Package scanned at distribution center in Atlanta, GA"
  }
}
```

Processing this response is straightforward:

```
import json

def process_structured_response(response: str):
    data = json.loads(response)
    return_to_user(data['customer_response'])
    save_to_database(data['internal_log'])

def return_to_user(message: str):
    print(f"Sending to user: {message}")

def save_to_database(log: dict):
    print(f"Saving to database: {log}")

process_structured_response(LLM_response)
```

In this example, which uses structured generation, the LLM's response is formatted in a specific JSON structure. The format clearly separates the customer-facing response from the internal log information, which makes it easy for the system to:

1. Extract the customer response for display
2. Store detailed order information in a database
3. Update internal tracking systems
4. Trigger automated processes based on the order status

Overall, processing a structured response is much simpler and less error-prone than trying to parse an unstructured response. That is desirable when LLMs are used in combination with other software components or databases, as in real-world applications.

KEY ASPECTS OF STRUCTURED GENERATION

Structured generation can be implemented explicitly (through careful prompting of the model) or implicitly (with an algorithm called guided decoding, which manipulates the model generation at each token to enforce the desired structure).

To achieve it, we must consider two main aspects:

- Defining output schemas: We specify the exact format and structure we expect from the model's output. This helps the model understand our requirements and constrains its responses accordingly.

- Validation and processing: We implement mechanisms to validate the model's output against our defined schema, ensuring it meets our criteria before further processing or merging into other systems.

In what follows, we moslty focus on the first aspect and exploit prompt engineering for explicit structured generation.

Obtaining Structured Outputs via Prompt Engineering with Anthropic's Claude API

We'll implement structured generation with Python in the Anthropic API.

1. Setting Up Our Environment

To set up the environment, install the Anthropic library from the terminal.

```
pip install anthropic
```

Next, set up your API key (keys can be obtained on Anthropic API). To do that safely, you can proceed in either of two ways.

First option: register it in a global environment variable and run the following command in the terminal:

```
export ANTHROPIC_API_KEY='YOUR_API_KEY'
```

Then, load the key with:

```
import os
key = os.environ["ANTHROPIC_API_KEY"]
if key is None:
    raise ValueError("Error: ANTHROPIC_API_KEY not found")
```

Second option: store the key in a .env file. If you are following the tutorial from the cloned github repository, you can find a .env.copy file in the root of the repository: please rename it to .env after editing it with your API key. Otherwise, you can create the file yourself and add the following line in it:

```
ANTHROPIC_API_KEY='YOUR_API_KEY'
```

Then, load the key with the Python library `python-dotenv`, which can be installed with the following command:

```
pip install python-dotenv
```

Check if Python can find the key:

```
from dotenv import load_dotenv
import os
load_dotenv()
key = os.environ["ANTHROPIC_API_KEY"]
if key is None:
    raise ValueError("Error: ANTHROPIC_API_KEY not found")
```

Assuming that you have set up the key properly, you are ready to import the Anthropic library and initialize the client with your API key.

```
import anthropic
from dotenv import load_dotenv
import os
load_dotenv()
key = os.environ["ANTHROPIC_API_KEY"]
if key is None:
    raise ValueError("Error: ANTHROPIC_API_KEY not found")
client = anthropic.Anthropic(
    api_key=key,
)
```

2. Testing Anthropic API with Claude 3.5 Sonnet

For all our examples, we will use Claude 3.5 Sonnet, a model that has been extensively trained on structured generation tasks and is highly versatile across different data structures, like JSON, YAML, XML and CSV. The latest model version at the moment of writing is `claude-3-5-sonnet-20240620`.

To test that everything is properly installed and that the API key is working, we can start with a simple example where we ask the model what a JSON schema is.

```
response = client.messages.create(
    model="claude-3-5-sonnet-20240620",
    max_tokens=200,
    messages=[
        {"role": "user", "content": "What is a JSON schema in a
sentence?"}
    ],
)
```

This code snippet will return a `Message` object with the following structure:

```
Message = {
    "id": str,
    "content": [
        {
            "text": str,
            "type": str
        }
    ],
    "model": str,
    "role": str,
    "stop_reason": str,
    "stop_sequence": None,  # or potentially str
    "type": str,
    "usage": {
        "input_tokens": int,
        "output_tokens": int
    }
}
```

In order to extract the text content of the message, we can index into the content key, treating it as a Python dictionary.

```
text_response = response.content[0].text
print(f"Claude response: {text_response}")
```

```
Claude response: A JSON schema is a declarative format for describing the
structure, content, and validation rules of JSON data.
```

This is how we access the LLM's responses.

3. Using the System Prompt to Guide the Output Format

We now want to obtain more structured outputs, for instance a response that contains a topic, citations and a short answer. So we pose the same question on the JSON schema, but this time we provide the model with an example of the desired output format directly in the prompt, separately including an example Python dictionary, a JSON string and a YAML string.

For the model to be able and output a Python dictionary, we include the following example in the prompt:

1. Python Dictionary:

```
example_dictionary = {
    "topic": "zip format",
    "citations": [{"citation_number": 1, "source":
"https://example.com"}],
    "answer": "The .zip format is a compressed file format that groups
```

```
multiple files into a single archive, with the files inside the archive
appearing as if they were not compressed."
}
```

Dictionaries are native Python data structures. They are easy to work with in Python, but not easily interchangeable with other programming languages. A more exchangeable format is:

2. JSON (JavaScript Object Notation):

```
example_json_string = '{"topic": "zip format", "citations":
[{"citation_number": 1, "source": "https://example.com"}], "answer": "The
.zip format is a compressed file format that groups multiple files into a
single archive, with the files inside the archive appearing as if they
were not compressed."}'
```

JSON is easy for humans to read and write, and easy for machines to parse and generate. It's language-independent and widely used for API responses and configuration files. An even more human-friendly format is:

3. YAML (YAML Ain't Markup Language):

```
example_yaml_string = """topic: zip format
citations:
  - citation_number: 1
    source: https://example.com
answer: The .zip format is a compressed file format that groups multiple
files into a single archive, with the files inside the archive appearing
as if they were not compressed.
"""
```

YAML is data serialization standard, often used for configuration files and in applications where data is stored or transmitted. It is more readable than JSON for complex structures, but can be prone to errors due to its reliance on indentation.

Now, let's use these three examples in our prompts.

Anthropic's models are trained to receive instructions through the system prompt, which is a message that is prepended to the user's message and sent along with it to help guide the model's response. In order to access the system prompt, we use the `system` argument in the `messages.create` function.

```
response_list = []
for example in [example_dictionary, example_json_string,
example_yaml_string]:
    response = client.messages.create(
        model="claude-3-5-sonnet-20240620",
        system=f"You are a helpful assistant that responds in the same
format as the following example: {example}",
```

```
        messages=[
            {"role": "user", "content": "What is a JSON schema in a
sentence?"}
        ],
        max_tokens=200,
    )
    response_list.append(response.content[0].text)
    print(f"Claude response: {response.content[0].text}")
```

This will give us the following output in a Python dictionary format:

```
{
  "topic": "JSON schema",
  "citations": [
    {
      "citation_number": 1,
      "source": "https://json-schema.org/understanding-json-schema/"
    }
  ],
  "answer": "A JSON schema is a declarative language that allows you to
annotate and validate JSON documents, defining the structure, constraints,
and documentation of JSON data."
}
```

Here's the answer from the JSON string format:

```
{
  "topic": "JSON schema",
  "citations": [
    {
      "citation_number": 1,
      "source": "https://json-schema.org/understanding-json-schema/"
    }
  ],
  "answer": "A JSON schema is a declarative language that allows you to
annotate and validate JSON documents, defining the structure, constraints,
and documentation of JSON data."
}
```

And this is the answer from the YAML string format:

```
topic: JSON schema

citations:
  - citation_number: 1
    source: https://json-schema.org/understanding-json-schema/
```

```
answer: A JSON schema is a declarative language that allows you to
annotate and validate JSON documents, defining the structure, constraints,
and documentation of JSON data.
```

Parsing each of these responses demonstrates how we can work with different formats. Note that executing generated Python code can be risky in general. We're only doing this for illustrative purposes, and with a model as safe and reliable as Claude 3.5 Sonnet. In real-world applications, always validate and sanitize any data before processing.

```
import json
import yaml
import ast

def parse_response(response, format_type):
    try:
        if format_type == 'dict':
            # WARNING: ast.literal_eval is safer than eval, but still use
caution
            return ast.literal_eval(response)
        elif format_type == 'json':
            return json.loads(response)
        elif format_type == 'yaml':
            return yaml.safe_load(response)
    except Exception as e:
        print(f"Error parsing {format_type} response: {e}")
        return None

# Parse and print each response
for response, format_type in zip(response_list, ['dict', 'json', 'yaml']):
    parsed = parse_response(response, format_type)
    if parsed:
        print(f"\nParsed {format_type.upper()} response:")
        print(f"Topic: {parsed.get('topic')}")
        print(f"Citation: {parsed.get('citations')[0] if
parsed.get('citations') else 'No citation'}")
        print(f"Answer: {parsed.get('answer')}")
    else:
        print(f"\nFailed to parse {format_type.upper()} response")
```

This code shows how to parse each type of response. For the Python dictionary, we use `ast.literal_eval()`. For JSON, we use the built-in `json` module, and for YAML, we use the `pyyaml` library's `safe_load()` function.

By using these different formats and parsing methods, structured generation produces outputs that are not only human-readable, but also easily processable by machines. This flexibility is what allows us to integrate LLM outputs into broader workflows.

As a final example, we will process a larger list of file formats, create a dictionary of their responses indexed by the topic with answers as values, and save it to disk as a JSON file.

```
file_formats = [
    "zip", "tar", "rar", "7z", "iso", "gz", "bz2", "xz", "pdf", "docx"
]

format_info = {}

for format in file_formats:
    response = client.messages.create(
        model="claude-3-5-sonnet-20240620",
        system=f"You are a helpful assistant that responds in the same
format as the following example: {example_json_string}",
        messages=[
            {"role": "user", "content": f"What is the {format} file format
in one sentence?"}
        ],
        max_tokens=200,
    )

    try:
        parsed_response = json.loads(response.content[0].text)
        format_info[parsed_response['topic']] = parsed_response['answer']
    except json.JSONDecodeError:
        print(f"Error parsing response for {format} format")

# Save the dictionary to a JSON file
with open('file_formats_info.json', 'w') as f:
    json.dump(format_info, f, indent=2)

print("File format information has been saved to file_formats_info.json")
```

Overall, for simple tasks like the one above, prompt engineering proves an efficient strategy to guide the model towards the desired output format.

Conclusion

In this tutorial, we've presented a practical implementation of structured generation via prompt engineering. Using Anthropic's Claude API, we have successfully steered LLM outputs into predefined formats (Python dictionaries, JSON, and YAML).

Key takeaways:

1. Structured generation addresses the challenge of inconsistent LLM outputs, as a crucial step for integrating AI into enterprise systems.
2. Combining prompt engineering with structured generation techniques offers fine-grained control over model responses.
3. Different output formats (Python dict, JSON, YAML) cater to various use cases and system integration needs.
4. Parsing and processing structured outputs enables seamless incorporation into broader workflows.

As we advance in this series, we'll look into more sophisticated techniques like assistant response prefilling, function calling, and we'll leverage Pydantic for schema generation and validation. These methods will further refine our ability harness the full potential of AI-generated content, while maintaining the consistency and reliability required for production-grade systems.

Learning More

Here are some valuable resources on structured generation and related topics.

- **Anthropic Cookbook**: Explore practical examples of using Claude for structured JSON data extraction. The cookbook covers tasks like summarization, entity extraction, and sentiment analysis, from basic implementations to advanced use cases.
- **Instructor Library**: A powerful tool for generating structured outputs with LLMs, built on top of Pydantic.
- **Outlines Library**: An open-source implementation for structured generation with multiple model integrations.
- **Pydantic Documentation**: For in-depth information on schema generation and validation.

If you found this tutorial helpful, please consider showing your support:

- Star our GitHub repository: StructuredGenTutorial
- Stay tuned for our next blog post on tribe.ai/blog
- Follow us on Twitter:
  - @hyp_enri
  - @cyndesama