(structured-outputs)=

Structured Outputs

vLLM supports the generation of structured outputs using [outlines](https://github.com/dottxt-ai/outlines), [lm-format-enforcer](https://github.com/noamgat/lm-format-enforcer), or [xgrammar](https://github.com/mlc-ai/xgrammar) as backends for the guided decoding.
This document shows you some examples of the different options that are available to generate structured outputs.

Online Serving (OpenAI API)

You can generate structured outputs using the OpenAI's [Completions](https://platform.openai.com/docs/api-reference/completions) and [Chat](https://platform.openai.com/docs/api-reference/chat) API.

The following parameters are supported, which must be added as extra parameters:

- `guided_choice`: the output will be exactly one of the choices.
- `guided_regex`: the output will follow the regex pattern.
- `guided_json`: the output will follow the JSON schema.
- `guided_grammar`: the output will follow the context free grammar.
- `guided_whitespace_pattern`: used to override the default whitespace pattern for guided json decoding.
- `guided_decoding_backend`: used to select the guided decoding backend to use.

You can see the complete list of supported parameters on the [OpenAI-Compatible Server](#openai-compatible-server)page.

Now let´s see an example for each of the cases, starting with the `guided_choice`, as it´s the easiest one:

```python
from openai import OpenAI
client = OpenAI(
    base_url="http://localhost:8000/v1",
    api_key="-",
)

completion = client.chat.completions.create(
    model="Qwen/Qwen2.5-3B-Instruct",
    messages=[
        {"role": "user", "content": "Classify this sentiment: vLLM is wonderful!"}
    ],
    extra_body={"guided_choice": ["positive", "negative"]},
)
print(completion.choices[0].message.content)
```

The next example shows how to use the `guided_regex`. The idea is to generate an email address, given a simple regex template:

```python
```

completion = client.chat.completions.create(
    model="Qwen/Qwen2.5-3B-Instruct",
    messages=[
        {
            "role": "user",
            "content": "Generate an example email address for Alan Turing, who works in Enigma. End in .com and new line. Example result: alan.turing@enigma.com\n",
        }
    ],
    extra_body={"guided_regex": "\w+@\w+\.com\n", "stop": ["\n"]},
)
print(completion.choices[0].message.content)
```

One of the most relevant features in structured text generation is the option to generate a valid JSON with pre-defined fields and formats.

For this we can use the `guided_json` parameter in two different ways:

- Using directly a [JSON Schema](https://json-schema.org/)
- Defining a [Pydantic model](https://docs.pydantic.dev/latest/) and then extracting the JSON Schema from it (which is normally an easier option).

The next example shows how to use the `guided_json` parameter with a Pydantic model:

```python
from pydantic import BaseModel
from enum import Enum

```python
class CarType(str, Enum):
    sedan = "sedan"
    suv = "SUV"
    truck = "Truck"
    coupe = "Coupe"


class CarDescription(BaseModel):
    brand: str
    model: str
    car_type: CarType


json_schema = CarDescription.model_json_schema()

completion = client.chat.completions.create(
    model="Qwen/Qwen2.5-3B-Instruct",
    messages=[
        {
            "role": "user",
             "content": "Generate a JSON with the brand, model and car_type of the most iconic car from the 90's",
        }
    ],
    extra_body={"guided_json": json_schema},
)
```

print(completion.choices[0].message.content)

```

:::{tip}

While not strictly necessary, normally it´s better to indicate in the prompt that a JSON needs to be generated and which fields and how should the LLM fill them.

This can improve the results notably in most cases.

:::

Finally we have the `guided_grammar`, which probably is the most difficult one to use but it´s really powerful, as it allows us to define complete languages like SQL queries.

It works by using a context free EBNF grammar, which for example we can use to define a specific format of simplified SQL queries, like in the example below:

```python
simplified_sql_grammar = """
    ?start: select_statement

    ?select_statement: "SELECT " column_list " FROM " table_name

    ?column_list: column_name ("," column_name)*

    ?table_name: identifier

    ?column_name: identifier

    ?identifier: /[a-zA-Z_][a-zA-Z0-9_]*/

```
"""

completion = client.chat.completions.create(
    model="Qwen/Qwen2.5-3B-Instruct",
    messages=[
        {
            "role": "user",
            "content": "Generate an SQL query to show the 'username' and 'email' from the 'users' table.",
        }
    ],
    extra_body={"guided_grammar": simplified_sql_grammar},
)
print(completion.choices[0].message.content)
```

Full example: <gh-file:examples/online_serving/openai_chat_completion_structured_outputs.py>

Experimental Automatic Parsing (OpenAI API)

This section covers the OpenAI beta wrapper over the `client.chat.completions.create()` method that provides richer integrations with Python specific types.

At the time of writing (`openai==1.54.4`), this is a "beta" feature in the OpenAI client library. Code reference can be found [here](https://github.com/openai/openai-python/blob/52357cff50bee57ef442e94d78a0de38b4173fc2/src/openai/resources/beta/chat/completions.py#L100-L104).

For the following examples, vLLM was setup using `vllm serve meta-llama/Llama-3.1-8B-Instruct`

Here is a simple example demonstrating how to get structured output using Pydantic models:

```python
from pydantic import BaseModel
from openai import OpenAI


class Info(BaseModel):
    name: str
    age: int


client = OpenAI(base_url="http://0.0.0.0:8000/v1", api_key="dummy")
completion = client.beta.chat.completions.parse(
    model="meta-llama/Llama-3.1-8B-Instruct",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "My name is Cameron, I'm 28. What's my name and age?"},
    ],
    response_format=Info,
    extra_body=dict(guided_decoding_backend="outlines"),
)

message = completion.choices[0].message
```

print(message)

assert message.parsed

print("Name:", message.parsed.name)

print("Age:", message.parsed.age)

```

Output:

```console
ParsedChatCompletionMessage[Testing](content='{"name": "Cameron", "age": 28}', refusal=None, role='assistant', audio=None, function_call=None, tool_calls=[], parsed=Testing(name='Cameron', age=28))
Name: Cameron
Age: 28
```

Here is a more complex example using nested Pydantic models to handle a step-by-step math solution:

```python
from typing import List
from pydantic import BaseModel
from openai import OpenAI


class Step(BaseModel):
    explanation: str

```
    output: str


class MathResponse(BaseModel):
    steps: List[Step]
    final_answer: str


client = OpenAI(base_url="http://0.0.0.0:8000/v1", api_key="dummy")
completion = client.beta.chat.completions.parse(
    model="meta-llama/Llama-3.1-8B-Instruct",
    messages=[
        {"role": "system", "content": "You are a helpful expert math tutor."},
        {"role": "user", "content": "Solve 8x + 31 = 2."},
    ],
    response_format=MathResponse,
    extra_body=dict(guided_decoding_backend="outlines"),
)

message = completion.choices[0].message
print(message)
assert message.parsed
for i, step in enumerate(message.parsed.steps):
    print(f"Step #{i}:", step)
print("Answer:", message.parsed.final_answer)
```

Output:

```console
ParsedChatCompletionMessage[MathResponse](content='{ "steps": [{ "explanation": "First, let\'s isolate the term with the variable \'x\'. To do this, we\'ll subtract 31 from both sides of the equation.", "output": "8x + 31 - 31 = 2 - 31"}, { "explanation": "By subtracting 31 from both sides, we simplify the equation to 8x = -29.", "output": "8x = -29"}, { "explanation": "Next, let\'s isolate \'x\' by dividing both sides of the equation by 8.", "output": "8x / 8 = -29 / 8"}], "final_answer": "x = -29/8" }', refusal=None, role='assistant', audio=None, function_call=None, tool_calls=[], parsed=MathResponse(steps=[Step(explanation="First, let's isolate the term with the variable 'x'. To do this, we'll subtract 31 from both sides of the equation.", output='8x + 31 - 31 = 2 - 31'), Step(explanation='By subtracting 31 from both sides, we simplify the equation to 8x = -29.', output='8x = -29'), Step(explanation="Next, let's isolate 'x' by dividing both sides of the equation by 8.", output='8x / 8 = -29 / 8')], final_answer='x = -29/8'))
Step #0: explanation="First, let's isolate the term with the variable 'x'. To do this, we'll subtract 31 from both sides of the equation." output='8x + 31 - 31 = 2 - 31'
Step #1: explanation='By subtracting 31 from both sides, we simplify the equation to 8x = -29.' output='8x = -29'
Step #2: explanation="Next, let's isolate 'x' by dividing both sides of the equation by 8." output='8x / 8 = -29 / 8'
Answer: x = -29/8
```

Offline Inference

Offline inference allows for the same types of guided decoding.

To use it, we´ll need to configure the guided decoding using the class `GuidedDecodingParams` inside `SamplingParams`.

The main available options inside `GuidedDecodingParams` are:

- `json`
- `regex`
- `choice`
- `grammar`
- `backend`
- `whitespace_pattern`

These parameters can be used in the same way as the parameters from the Online Serving examples above.

One example for the usage of the `choices` parameter is shown below:

```python
from vllm import LLM, SamplingParams
from vllm.sampling_params import GuidedDecodingParams

llm = LLM(model="HuggingFaceTB/SmolLM2-1.7B-Instruct")

guided_decoding_params = GuidedDecodingParams(choice=["Positive", "Negative"])
sampling_params = SamplingParams(guided_decoding=guided_decoding_params)
outputs = llm.generate(
    prompts="Classify this sentiment: vLLM is wonderful!",
    sampling_params=sampling_params,
)
print(outputs[0].outputs[0].text)
```

```

Full example: <gh-file:examples/offline_inference/structured_outputs.py>
```