(openai-compatible-server)=

OpenAI-Compatible Server

vLLM provides an HTTP server that implements OpenAI's [Completions API](https://platform.openai.com/docs/api-reference/completions), [Chat API](https://platform.openai.com/docs/api-reference/chat), and more!

You can start the server via the [`vllm serve`](#vllm-serve) command, or through [Docker](#deployment-docker):

```bash
vllm serve NousResearch/Meta-Llama-3-8B-Instruct --dtype auto --api-key token-abc123
```

To call the server, you can use the [official OpenAI Python client](https://github.com/openai/openai-python), or any other HTTP client.

```python
from openai import OpenAI
client = OpenAI(
    base_url="http://localhost:8000/v1",
    api_key="token-abc123",
)

completion = client.chat.completions.create(
  model="NousResearch/Meta-Llama-3-8B-Instruct",
```

```
  messages=[
    {"role": "user", "content": "Hello!"}
  ]
)

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

Supported APIs

We currently support the following OpenAI APIs:

- [Completions API](#completions-api) (`/v1/completions`)
  - Only applicable to [text generation models](../models/generative_models.md) (`--task generate`).
  - *Note: `suffix` parameter is not supported.*
- [Chat Completions API](#chat-api) (`/v1/chat/completions`)
  - Only applicable to [text generation models](../models/generative_models.md) (`--task generate`) with a [chat template](#chat-template).
  - *Note: `parallel_tool_calls` and `user` parameters are ignored.*
- [Embeddings API](#embeddings-api) (`/v1/embeddings`)
  - Only applicable to [embedding models](../models/pooling_models.md) (`--task embed`).

In addition, we have the following custom APIs:

- [Tokenizer API](#tokenizer-api) (`/tokenize`, `/detokenize`)
  - Applicable to any model with a tokenizer.
- [Pooling API](#pooling-api) (`/pooling`)

  - Applicable to all [pooling models](../models/pooling_models.md).
- [Score API](#score-api) (`/score`)
  - Only applicable to [cross-encoder models](../models/pooling_models.md) (`--task score`).
- [Re-rank API](#rerank-api) (`/rerank`, `/v1/rerank`, `/v2/rerank`)
  - Implements [Jina AI's v1 re-rank API](https://jina.ai/reranker/)
    - Also compatible with [Cohere's v1 & v2 re-rank APIs](https://docs.cohere.com/v2/reference/rerank)
  - Jina and Cohere's APIs are very similar; Jina's includes extra information in the rerank endpoint's response.
  - Only applicable to [cross-encoder models](../models/pooling_models.md) (`--task score`).

(chat-template)=

Chat Template

In order for the language model to support chat protocol, vLLM requires the model to include
a chat template in its tokenizer configuration. The chat template is a Jinja2 template that
specifies how are roles, messages, and other chat-specific tokens are encoded in the input.

An example chat template for `NousResearch/Meta-Llama-3-8B-Instruct` can be found [here](https://github.com/meta-llama/llama3?tab=readme-ov-file#instruction-tuned-models)

Some models do not provide a chat template even though they are instruction/chat fine-tuned. For those model,
you can manually specify their chat template in the `--chat-template` parameter with the file path to the chat
template, or the template in string form. Without a chat template, the server will not be able to

process chat

and all chat requests will error.

```bash
vllm serve <model> --chat-template ./path-to-chat-template.jinja
```

vLLM community provides a set of chat templates for popular models. You can find them under the <gh-dir:examples> directory.

With the inclusion of multi-modal chat APIs, the OpenAI spec now accepts chat messages in a new format which specifies

both a `type` and a `text` field. An example is provided below:

```python
completion = client.chat.completions.create(
  model="NousResearch/Meta-Llama-3-8B-Instruct",
  messages=[
    {"role": "user", "content": [{"type": "text", "text": "Classify this sentiment: vLLM is wonderful!"}]}
  ]
)
```

Most chat templates for LLMs expect the `content` field to be a string, but there are some newer models like

`meta-llama/Llama-Guard-3-1B` that expect the content to be formatted according to the OpenAI schema in the

request. vLLM provides best-effort support to detect this automatically, which is logged as a string like
"Detected the chat template content format to be...", and internally converts incoming requests to match
the detected format, which can be one of:

- `"string"`: A string.
  - Example: `"Hello world"`
- `"openai"`: A list of dictionaries, similar to OpenAI schema.
  - Example: `[{"type": "text", "text": "Hello world!"}]`

If the result is not what you expect, you can set the `--chat-template-content-format` CLI argument to override which format to use.

Extra Parameters

vLLM supports a set of parameters that are not part of the OpenAI API.
In order to use them, you can pass them as extra parameters in the OpenAI client.
Or directly merge them into the JSON payload if you are using HTTP call directly.

```python
completion = client.chat.completions.create(
  model="NousResearch/Meta-Llama-3-8B-Instruct",
  messages=[
    {"role": "user", "content": "Classify this sentiment: vLLM is wonderful!"}
  ],
  extra_body={
```

```
    "guided_choice": ["positive", "negative"]
  }
)
```

Extra HTTP Headers

Only `X-Request-Id` HTTP request header is supported for now. It can be enabled
with `--enable-request-id-headers`.

> Note that enablement of the headers can impact performance significantly at high QPS
> rates. We recommend implementing HTTP headers at the router level (e.g. via Istio),
> rather than within the vLLM layer for this reason.
> See [this PR](https://github.com/vllm-project/vllm/pull/11529) for more details.

```python
completion = client.chat.completions.create(
  model="NousResearch/Meta-Llama-3-8B-Instruct",
  messages=[
    {"role": "user", "content": "Classify this sentiment: vLLM is wonderful!"}
  ],
  extra_headers={
    "x-request-id": "sentiment-classification-00001",
  }
)
print(completion._request_id)
```

```
completion = client.completions.create(
  model="NousResearch/Meta-Llama-3-8B-Instruct",
  prompt="A robot may not injure a human being",
  extra_headers={
    "x-request-id": "completion-test",
  }
)
print(completion._request_id)
```

CLI Reference

(vllm-serve)=

`vllm serve`

The `vllm serve` command is used to launch the OpenAI-compatible server.

:::{argparse}
:module: vllm.entrypoints.openai.cli_args
:func: create_parser_for_docs
:prog: vllm serve
:::

Configuration file

You can load CLI arguments via a [YAML](https://yaml.org/) config file.

The argument names must be the long form of those outlined [above](#vllm-serve).

For example:

```yaml
# config.yaml

host: "127.0.0.1"
port: 6379
uvicorn-log-level: "info"
```

To use the above config file:

```bash
vllm serve SOME_MODEL --config config.yaml
```

:::{note}
In case an argument is supplied simultaneously using command line and the config file, the value from the command line will take precedence.
The order of priorities is `command line > config file values > defaults`.
:::

API Reference

(completions-api)=

Completions API

Our Completions API is compatible with [OpenAI's Completions API](https://platform.openai.com/docs/api-reference/completions);
you can use the [official OpenAI Python client](https://github.com/openai/openai-python) to interact with it.

Code example: <gh-file:examples/online_serving/openai_completion_client.py>

Extra parameters

The following [sampling parameters](#sampling-params) are supported.

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-completion-sampling-params
:end-before: end-completion-sampling-params
:::

The following extra parameters are supported:

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-completion-extra-params
:end-before: end-completion-extra-params
:::

(chat-api)=

Chat API

Our Chat API is compatible with [OpenAI's Chat Completions API](https://platform.openai.com/docs/api-reference/chat);
you can use the [official OpenAI Python client](https://github.com/openai/openai-python) to interact with it.

We support both [Vision](https://platform.openai.com/docs/guides/vision)- and [Audio](https://platform.openai.com/docs/guides/audio?audio-generation-quickstart-example=audio-in)-related parameters;
see our [Multimodal Inputs](#multimodal-inputs) guide for more information.
- *Note: `image_url.detail` parameter is not supported.*

Code example: <gh-file:examples/online_serving/openai_chat_completion_client.py>

Extra parameters

The following [sampling parameters](#sampling-params) are supported.

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-chat-completion-sampling-params
:end-before: end-chat-completion-sampling-params
:::

The following extra parameters are supported:

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-chat-completion-extra-params
:end-before: end-chat-completion-extra-params
:::

(embeddings-api)=

Embeddings API

Our Embeddings API is compatible with [OpenAI's Embeddings API](https://platform.openai.com/docs/api-reference/embeddings);
you can use the [official OpenAI Python client](https://github.com/openai/openai-python) to interact with it.

If the model has a [chat template](#chat-template), you can replace `inputs` with a list of `messages` (same schema as [Chat API](#chat-api))
which will be treated as a single prompt to the model.

:::{tip}
This enables multi-modal inputs to be passed to embedding models, see [this page](#multimodal-inputs) for details.
:::

Code example: <gh-file:examples/online_serving/openai_embedding_client.py>

Extra parameters

The following [pooling parameters](#pooling-params) are supported.

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-embedding-pooling-params
:end-before: end-embedding-pooling-params
:::

The following extra parameters are supported by default:

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-embedding-extra-params
:end-before: end-embedding-extra-params
:::

For chat-like input (i.e. if `messages` is passed), these extra parameters are supported instead:

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-chat-embedding-extra-params
:end-before: end-chat-embedding-extra-params
:::

(tokenizer-api)=

Tokenizer API

Our Tokenizer API is a simple wrapper over [HuggingFace-style tokenizers](https://huggingface.co/docs/transformers/en/main_classes/tokenizer).
It consists of two endpoints:

- `/tokenize` corresponds to calling `tokenizer.encode()`.
- `/detokenize` corresponds to calling `tokenizer.decode()`.

(pooling-api)=

Pooling API

Our Pooling API encodes input prompts using a [pooling model](../models/pooling_models.md) and returns the corresponding hidden states.

The input format is the same as [Embeddings API](#embeddings-api), but the output data can contain an arbitrary nested list, not just a 1-D list of floats.

Code example: <gh-file:examples/online_serving/openai_pooling_client.py>

(score-api)=

Score API

Our Score API applies a cross-encoder model to predict scores for sentence pairs.

Usually, the score for a sentence pair refers to the similarity between two sentences, on a scale of 0 to 1.

You can find the documentation for these kind of models at [sbert.net](https://www.sbert.net/docs/package_reference/cross_encoder/cross_encoder.html).

Code example: <gh-file:examples/online_serving/openai_cross_encoder_score.py>

Single inference

You can pass a string to both `text_1` and `text_2`, forming a single sentence pair.

Request:

```bash
curl -X 'POST' \
  'http://127.0.0.1:8000/score' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "model": "BAAI/bge-reranker-v2-m3",
  "encoding_format": "float",
  "text_1": "What is the capital of France?",
  "text_2": "The capital of France is Paris."
}'
```

```

Response:

```bash
{
  "id": "score-request-id",
  "object": "list",
  "created": 693447,
  "model": "BAAI/bge-reranker-v2-m3",
  "data": [
    {
      "index": 0,
      "object": "score",
      "score": 1
    }
  ],
  "usage": {}
}
```

Batch inference

You can pass a string to `text_1` and a list to `text_2`, forming multiple sentence pairs
where each pair is built from `text_1` and a string in `text_2`.
The total number of pairs is `len(text_2)`.
```

Request:

```bash
curl -X 'POST' \
  'http://127.0.0.1:8000/score' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "model": "BAAI/bge-reranker-v2-m3",
  "text_1": "What is the capital of France?",
  "text_2": [
    "The capital of Brazil is Brasilia.",
    "The capital of France is Paris."
  ]
}'
```

Response:

```bash
{
  "id": "score-request-id",
  "object": "list",
  "created": 693570,
  "model": "BAAI/bge-reranker-v2-m3",
  "data": [
    {
```

```
      "index": 0,
      "object": "score",
      "score": 0.001094818115234375
    },
    {
      "index": 1,
      "object": "score",
      "score": 1
    }
  ],
  "usage": {}
}
```


You can pass a list to both `text_1` and `text_2`, forming multiple sentence pairs
where each pair is built from a string in `text_1` and the corresponding string in `text_2` (similar to `zip()`).
The total number of pairs is `len(text_2)`.

Request:

```bash
curl -X 'POST' \
  'http://127.0.0.1:8000/score' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
```

```
  "model": "BAAI/bge-reranker-v2-m3",
  "encoding_format": "float",
  "text_1": [
    "What is the capital of Brazil?",
    "What is the capital of France?"
  ],
  "text_2": [
    "The capital of Brazil is Brasilia.",
    "The capital of France is Paris."
  ]
}'
```

Response:

```bash
{
  "id": "score-request-id",
  "object": "list",
  "created": 693447,
  "model": "BAAI/bge-reranker-v2-m3",
  "data": [
    {
      "index": 0,
      "object": "score",
      "score": 1
    },
```

```
    {
      "index": 1,
      "object": "score",
      "score": 1
    }
  ],
  "usage": {}
}
```

Extra parameters

The following [pooling parameters](#pooling-params) are supported.

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-score-pooling-params
:end-before: end-score-pooling-params
:::

The following extra parameters are supported:

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py
:language: python
:start-after: begin-score-extra-params
:end-before: end-score-extra-params
:::

(rerank-api)=

Re-rank API

Our Re-rank API applies a cross-encoder model to predict relevant scores between a single query, and
each of a list of documents. Usually, the score for a sentence pair refers to the similarity between two sentences, on
a scale of 0 to 1.

You can find the documentation for these kind of models at [sbert.net](https://www.sbert.net/docs/package_reference/cross_encoder/cross_encoder.html).

The rerank endpoints support popular re-rank models such as `BAAI/bge-reranker-base` and other models supporting the
`score` task. Additionally, `/rerank`, `/v1/rerank`, and `/v2/rerank`
endpoints are compatible with both [Jina AI's re-rank API interface](https://jina.ai/reranker/) and
[Cohere's re-rank API interface](https://docs.cohere.com/v2/reference/rerank) to ensure compatibility with
popular open-source tools.

Code example: <gh-file:examples/online_serving/jinaai_rerank_client.py>

Example Request

Note that the `top_n` request parameter is optional and will default to the length of the `documents`

field.

Result documents will be sorted by relevance, and the `index` property can be used to determine original order.

Request:

```bash
curl -X 'POST' \
  'http://127.0.0.1:8000/v1/rerank' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "model": "BAAI/bge-reranker-base",
  "query": "What is the capital of France?",
  "documents": [
    "The capital of Brazil is Brasilia.",
    "The capital of France is Paris.",
    "Horses and cows are both animals"
  ]
}'
```

Response:

```bash
{
  "id": "rerank-fae51b2b664d4ed38f5969b612edff77",
```

```
  "model": "BAAI/bge-reranker-base",
  "usage": {
    "total_tokens": 56
  },
  "results": [
    {
      "index": 1,
      "document": {
        "text": "The capital of France is Paris."
      },
      "relevance_score": 0.99853515625
    },
    {
      "index": 0,
      "document": {
        "text": "The capital of Brazil is Brasilia."
      },
      "relevance_score": 0.0005860328674316406
    }
  ]
}
```


Extra parameters

The following [pooling parameters](#pooling-params) are supported.

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py

:language: python

:start-after: begin-rerank-pooling-params

:end-before: end-rerank-pooling-params

:::

The following extra parameters are supported:

:::{literalinclude} ../../../vllm/entrypoints/openai/protocol.py

:language: python

:start-after: begin-rerank-extra-params

:end-before: end-rerank-extra-params

:::