(offline-inference)=

Offline Inference

You can run vLLM in your own code on a list of prompts.

The offline API is based on the {class}`~vllm.LLM` class.
To initialize the vLLM engine, create a new instance of `LLM` and specify the model to run.

For example, the following code downloads the [`facebook/opt-125m`](https://huggingface.co/facebook/opt-125m) model from HuggingFace and runs it in vLLM using the default configuration.

```python
llm = LLM(model="facebook/opt-125m")
```

After initializing the `LLM` instance, you can perform model inference using various APIs.
The available APIs depend on the type of model that is being run:

- [Generative models](#generative-models) output logprobs which are sampled from to obtain the final output text.
- [Pooling models](#pooling-models) output their hidden states directly.

Please refer to the above pages for more details about each API.

:::{seealso}

[API Reference](/api/offline_inference/index)

:::

Configuration Options

This section lists the most common options for running the vLLM engine.

For a full list, refer to the [Engine Arguments](#engine-args) page.

(model-resolution)=

Model resolution

vLLM loads HuggingFace-compatible models by inspecting the `architectures` field in `config.json` of the model repository

and finding the corresponding implementation that is registered to vLLM.

Nevertheless, our model resolution may fail for the following reasons:

- The `config.json` of the model repository lacks the `architectures` field.
- Unofficial repositories refer to a model using alternative names which are not recorded in vLLM.
- The same architecture name is used for multiple models, creating ambiguity as to which model should be loaded.

To fix this, explicitly specify the model architecture by passing `config.json` overrides to the `hf_overrides` option.

For example:

```python
```

model = LLM(
    model="cerebras/Cerebras-GPT-1.3B",
    hf_overrides={"architectures": ["GPT2LMHeadModel"]}, # GPT-2
)
```

Our [list of supported models](#supported-models) shows the model architectures that are recognized by vLLM.

### Reducing memory usage

Large models might cause your machine to run out of memory (OOM). Here are some options that help alleviate this problem.

#### Tensor Parallelism (TP)

Tensor parallelism (`tensor_parallel_size` option) can be used to split the model across multiple GPUs.

The following code splits the model across 2 GPUs.

```python
llm = LLM(model="ibm-granite/granite-3.1-8b-instruct",
          tensor_parallel_size=2)
```

:::{important}
```

To ensure that vLLM initializes CUDA correctly, you should avoid calling related functions (e.g. {func}`torch.cuda.set_device`)
before initializing vLLM. Otherwise, you may run into an error like `RuntimeError: Cannot re-initialize CUDA in forked subprocess`.

To control which devices are used, please instead set the `CUDA_VISIBLE_DEVICES` environment variable.
:::

Quantization

Quantized models take less memory at the cost of lower precision.

Statically quantized models can be downloaded from HF Hub (some popular ones are available at [Neural Magic](https://huggingface.co/neuralmagic))
and used directly without extra configuration.

Dynamic quantization is also supported via the `quantization` option -- see [here](#quantization-index) for more details.

Context length and batch size

You can further reduce memory usage by limiting the context length of the model (`max_model_len` option)
and the maximum batch size (`max_num_seqs` option).

```python
```

llm = LLM(model="adept/fuyu-8b",

          max_model_len=2048,

          max_num_seqs=2)

```

### Performance optimization and tuning

You can potentially improve the performance of vLLM by finetuning various options.

Please refer to [this guide](#optimization-and-tuning) for more details.
```