lmdeploy Documentation

Release 0.13.0

LMDeploy Contributors

May 12, 2026

GET STARTED

LMDeploy has the following core features:

- **Efficient Inference**: LMDeploy delivers up to 1.8x higher request throughput than vLLM, by introducing key features like persistent batch(a.k.a. continuous batching), blocked KV cache, dynamic split&fuse, tensor parallelism, high-performance CUDA kernels and so on.
- **Effective Quantization**: LMDeploy supports weight-only and k/v quantization, and the 4-bit inference performance is 2.4x higher than FP16. The quantization quality has been confirmed via OpenCompass evaluation.
- **Effortless Distribution Server**: Leveraging the request distribution service, LMDeploy facilitates an easy and efficient deployment of multi-model services across multiple machines and cards.
- **Excellent Compatibility**: LMDeploy supports KV Cache Quant, AWQ and Automatic Prefix Caching to be used simultaneously.

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

LMDeploy is a python library for compressing, deploying, and serving Large Language Models(LLMs) and Vision-Language Models(VLMs). Its core inference engines include TurboMind Engine and PyTorch Engine. The former is developed by C++ and CUDA, striving for ultimate optimization of inference performance, while the latter, developed purely in Python, aims to decrease the barriers for developers.

It supports LLMs and VLMs deployment on both Linux and Windows platform, with minimum requirement of CUDA version 11.3. Furthermore, it is compatible with the following NVIDIA GPUs:

- Volta(sm70): V100
- Turing(sm75): 20 series, T4
- Ampere(sm80,sm86): 30 series, A10, A16, A30, A100
- Ada Lovelace(sm89): 40 series

1.1.1 Install with pip (Recommend)

It is recommended installing lmdeploy using pip in a conda environment (python 3.10 - 3.13):

```
conda create -n lmdeploy python=3.12 -y
conda activate lmdeploy
pip install lmdeploy
```

1.1.2 Install from source

By default, LMDeploy will build with NVIDIA CUDA support, utilizing both the Turbomind and PyTorch backends. Before installing LMDeploy, ensure you have successfully installed the CUDA Toolkit.

Once the CUDA toolkit is successfully set up, you can build and install LMDeploy with a single command:

```
pip install git+https://github.com/InternLM/lmdeploy.git
```

You can also explicitly disable the Turbomind backend to avoid CUDA compilation by setting the `DISABLE_TURBOMIND` environment variable:

```
DISABLE_TURBOMIND=1 pip install git+https://github.com/InternLM/lmdeploy.git
```

If you prefer a specific version instead of the `main` branch of LMDeploy, you can specify it in your command:

```
pip install https://github.com/InternLM/lmdeploy/archive/refs/tags/v0.11.0.zip
```

If you want to build LMDeploy with support for Ascend, Cambricon, or MACA, install LMDeploy with the corresponding `LMDEPLOY_TARGET_DEVICE` environment variable.

LMDeploy also supports installation on AMD GPUs with ROCm.

```
#The recommended way is to use the official ROCm PyTorch Docker image with pre-installed␣
↪dependencies:
docker run -it \
    --cap-add=SYS_PTRACE \
    --security-opt seccomp=unconfined \
    --device=/dev/kfd \
    --device=/dev/dri \
    --group-add video \
    --ipc=host \
    --network=host \
    --shm-size 32G \
    -v /root:/workspace \
    rocm/pytorch:latest


#Once inside the container, install LMDeploy with ROCm support:
LMDEPLOY_TARGET_DEVICE=rocm pip install  git+https://github.com/InternLM/lmdeploy.git
```

1.2 Quick Start

This tutorial shows the usage of LMDeploy on CUDA platform:

- Offline inference of LLM model and VLM model
- Serve a LLM or VLM model by the OpenAI compatible server
- Console CLI to interactively chat with LLM model

Before reading further, please ensure that you have installed lmdeploy as outlined in the *installation guide*

1.2.1 Offline batch inference

LLM inference

```
from lmdeploy import pipeline
pipe = pipeline('internlm/internlm2_5-7b-chat')
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

When constructing the `pipeline`, if an inference engine is not designated between the TurboMind Engine and the PyTorch Engine, LMDeploy will automatically assign one based on *their respective capabilities*, with the TurboMind Engine taking precedence by default.

However, you have the option to manually select an engine. For instance,

```
from lmdeploy import pipeline, TurbomindEngineConfig
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=TurbomindEngineConfig(
                    max_batch_size=32,
                    enable_prefix_caching=True,
                    cache_max_entry_count=0.8,
                    session_len=8192,
                ))
```

or,

```
from lmdeploy import pipeline, PytorchEngineConfig
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=PytorchEngineConfig(
                    max_batch_size=32,
                    enable_prefix_caching=True,
                    cache_max_entry_count=0.8,
                    session_len=8192,
                ))
```

Note

The parameter "cache_max_entry_count" significantly influences the GPU memory usage. It means the proportion of FREE GPU memory occupied by the K/V cache after the model weights are loaded.

The default value is 0.8. The K/V cache memory is allocated once and reused repeatedly, which is why it is observed that the built pipeline and the "api_server" mentioned later in the next consumes a substantial amount of GPU memory.

If you encounter an Out-of-Memory(OOM) error, you may need to consider lowering the value of "cache_max_entry_count".

When use the callable `pipe()` to perform token generation with given prompts, you can set the sampling parameters via `GenerationConfig` as below:

```
from lmdeploy import GenerationConfig, pipeline

pipe = pipeline('internlm/internlm2_5-7b-chat')
prompts = ['Hi, pls intro yourself', 'Shanghai is']
response = pipe(prompts,
                gen_config=GenerationConfig(
                    max_new_tokens=1024,
                    top_p=0.8,
                    top_k=40,
                    temperature=0.6
                ))
```

In the `GenerationConfig`, `top_k=1` or `temperature=0.0` indicates greedy search.

For more information about pipeline, please read the *detailed tutorial*

VLM inference

The usage of VLM inference pipeline is akin to that of LLMs, with the additional capability of processing image data with the pipeline. For example, you can utilize the following code snippet to perform the inference with an InternVL model:

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2-8B')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

In VLM pipeline, the default image processing batch size is 1. This can be adjusted by `VisionConfig`. For instance, you might set it like this:

```
from lmdeploy import pipeline, VisionConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2-8B',
                vision_config=VisionConfig(
                    max_batch_size=8
                ))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

However, the larger the image batch size, the greater risk of an OOM error, because the LLM component within the VLM model pre-allocates a massive amount of memory in advance.

We encourage you to manually choose between the TurboMind Engine and the PyTorch Engine based on their respective capabilities, as detailed in *the supported-models matrix*. Additionally, follow the instructions in *LLM Inference* section to reduce the values of memory-related parameters

1.2.2 Serving

As demonstrated in the previous *offline batch inference* section, this part presents the respective serving methods for LLMs and VLMs.

Serve a LLM model

```
lmdeploy serve api_server internlm/internlm2_5-7b-chat
```

This command will launch an OpenAI-compatible server on the localhost at port `23333`. You can specify a different server port by using the `--server-port` option. For more options, consult the help documentation by running `lmdeploy serve api_server --help`. Most of these options align with the engine configuration.

To access the service, you can utilize the official OpenAI Python package `pip install openai`. Below is an example demonstrating how to use the entrypoint `v1/chat/completions`

```
from openai import OpenAI
client = OpenAI(
    api_key='YOUR_API_KEY',
    base_url="http://0.0.0.0:23333/v1"
)
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
  model=model_name,
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": " provide three suggestions about time management"},
  ],
    temperature=0.8,
    top_p=0.8
)
print(response)
```

We encourage you to refer to the detailed guide for more comprehensive information about *serving with Docker*, *function calls* and other topics

Serve a VLM model

```
lmdeploy serve api_server OpenGVLab/InternVL2-8B
```

> **Note**
>
> LMDeploy reuses the vision component from upstream VLM repositories. Each upstream VLM model may have different dependencies. Consequently, LMDeploy has decided not to include the dependencies of the upstream VLM repositories in its own dependency list. If you encounter an "ImportError" when using LMDeploy for inference with VLM models, please install the relevant dependencies yourself.

After the service is launched successfully, you can access the VLM service in a manner similar to how you would access the `gptv4` service by modifying the `api_key` and `base_url` parameters:

```python
from openai import OpenAI

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role':
        'user',
        'content': [{
            'type': 'text',
            'text': 'Describe the image please',
        }, {
            'type': 'image_url',
            'image_url': {
                'url':
                'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/
↪tiger.jpeg',
            },
        }],
    }],
    temperature=0.8,
    top_p=0.8)
print(response)
```

1.2.3 Inference with Command line Interface

LMDeploy offers a very convenient CLI tool for users to chat with the LLM model locally. For example:

```
lmdeploy chat internlm/internlm2_5-7b-chat --backend turbomind
```

It is designed to assist users in checking and verifying whether LMDeploy supports their model, whether the chat template is applied correctly, and whether the inference results are delivered smoothly.

Another tool, `lmdeploy check_env`, aims to gather the essential environment information. It is crucial when reporting an issue to us, as it helps us diagnose and resolve the problem more effectively.

If you have any doubt about their usage, you can try using the `--help` option to obtain detailed information.

1.3 On Other Platforms

1.3.1 Get Started with Huawei Ascend

We currently support running lmdeploy on **Atlas 800T A3, Atlas 800T A2 and Atlas 300I Duo**. The usage of lmdeploy on a Huawei Ascend device is almost the same as its usage on CUDA with PytorchEngine in lmdeploy. Please read the original *Get Started* guide before reading this tutorial.

Here is the *supported model list*.

> [!IMPORTANT] We have uploaded a docker image with KUNPENG CPU to aliyun. Please try to pull the image by following command:
>
> Atlas 800T A3:

```
docker pull crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
lmdeploy_dlinfer/ascend:a3-latest
```

(Atlas 800T A3 currently supports only the Qwen-series with eager mode.)

Atlas 800T A2:

```
docker pull crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
lmdeploy_dlinfer/ascend:a2-latest
```

300I Duo:

```
docker pull crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
lmdeploy_dlinfer/ascend:300i-duo-latest
```

(Atlas 300I Duo currently works only with graph mode.)

To build the environment yourself, refer to the Dockerfiles *here*.

Offline batch inference

LLM inference

Set `device_type="ascend"` in the `PytorchEngineConfig`:

```
from lmdeploy import pipeline
from lmdeploy import PytorchEngineConfig
pipe = pipeline("internlm/internlm2_5-7b-chat",
        backend_config=PytorchEngineConfig(tp=1, device_type="ascend"))
question = ["Shanghai is", "Please introduce China", "How are you?"]
response = pipe(question)
print(response)
```

VLM inference

Set `device_type="ascend"` in the `PytorchEngineConfig`:

```
from lmdeploy import pipeline, PytorchEngineConfig
from lmdeploy.vl import load_image
pipe = pipeline('OpenGVLab/InternVL2-2B',
        backend_config=PytorchEngineConfig(tp=1, device_type='ascend'))
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Online serving

Serve a LLM model

Add `--device ascend` in the serve command.

```
lmdeploy serve api_server --backend pytorch --device ascend internlm/internlm2_5-7b-chat
```

Run the following commands to launch docker container for lmdeploy LLM serving:

```
docker run -it --net=host crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
↪lmdeploy_dlinfer/ascend:a2-latest \
    bash -i -c "lmdeploy serve api_server --backend pytorch --device ascend internlm/
↪internlm2_5-7b-chat"
```

Serve a VLM model

Add `--device ascend` in the serve command

```
lmdeploy serve api_server --backend pytorch --device ascend OpenGVLab/InternVL2-2B
```

Run the following commands to launch docker container for lmdeploy VLM serving:

```
docker run -it --net=host crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
↪lmdeploy_dlinfer/ascend:a2-latest \
    bash -i -c "lmdeploy serve api_server --backend pytorch --device ascend OpenGVLab/
↪InternVL2-2B"
```

Inference with Command line Interface

Add `--device ascend` in the serve command.

```
lmdeploy chat internlm/internlm2_5-7b-chat --backend pytorch --device ascend
```

Run the following commands to launch lmdeploy chatting after starting container:

```
docker run -it crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/lmdeploy_
↪dlinfer/ascend:a2-latest \
    bash -i -c "lmdeploy chat --backend pytorch --device ascend internlm/internlm2_5-7b-
↪chat"
```

Quantization

w4a16 AWQ

Run the following commands to quantize weights on Atlas 800T A2.

```
lmdeploy lite auto_awq $HF_MODEL --work-dir $WORK_DIR --device npu
```

Please check *supported_models* before use this feature.

w8a8 SMOOTH_QUANT

Run the following commands to quantize weights on Atlas 800T A2.

```
lmdeploy lite smooth_quant $HF_MODEL --work-dir $WORK_DIR --device npu
```

Please check *supported_models* before use this feature.

int8 KV-cache Quantization

Ascend backend has supported offline int8 KV-cache Quantization on eager mode.

Please refer this doc for details.

Limitations on 300I Duo

1. only support dtype=float16.
2. only support graph mode, please do not add –eager-mode.

1.3.2 MetaX-tech

The usage of lmdeploy on a MetaX-tech device is almost the same as its usage on CUDA with PytorchEngine in lmdeploy. Please read the original *Get Started* guide before reading this tutorial.

Here is the *supported model list*.

> [!IMPORTANT] We have uploaded a docker image to aliyun. Please try to pull the image by following command:
>
> `docker pull crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/lmdeploy_dlinfer/maca:latest`

Offline batch inference

LLM inference

Set `device_type="maca"` in the `PytorchEngineConfig`:

```
from lmdeploy import pipeline
from lmdeploy import PytorchEngineConfig
pipe = pipeline("internlm/internlm2_5-7b-chat",
        backend_config=PytorchEngineConfig(tp=1, device_type="maca"))
question = ["Shanghai is", "Please introduce China", "How are you?"]
response = pipe(question)
print(response)
```

VLM inference

Set `device_type="maca"` in the `PytorchEngineConfig`:

```
from lmdeploy import pipeline, PytorchEngineConfig
from lmdeploy.vl import load_image
pipe = pipeline('OpenGVLab/InternVL2-2B',
        backend_config=PytorchEngineConfig(tp=1, device_type='maca'))
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Online serving

Serve a LLM model

Add `--device maca` in the serve command.

```
lmdeploy serve api_server --backend pytorch --device maca internlm/internlm2_5-7b-chat
```

Run the following commands to launch docker container for lmdeploy LLM serving:

```
docker run -it --net=host crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
↪lmdeploy_dlinfer/maca:latest \
    bash -i -c "lmdeploy serve api_server --backend pytorch --device maca internlm/
↪internlm2_5-7b-chat"
```

Serve a VLM model

Add `--device maca` in the serve command

```
lmdeploy serve api_server --backend pytorch --device maca OpenGVLab/InternVL2-2B
```

Run the following commands to launch docker container for lmdeploy VLM serving:

```
docker run -it --net=host crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
↪lmdeploy_dlinfer/maca:latest \
    bash -i -c "lmdeploy serve api_server --backend pytorch --device maca OpenGVLab/
↪InternVL2-2B"
```

Inference with Command line Interface

Add `--device maca` in the serve command.

```
lmdeploy chat internlm/internlm2_5-7b-chat --backend pytorch --device maca
```

Run the following commands to launch lmdeploy chatting after starting container:

```
docker run -it crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/lmdeploy_
↪dlinfer/maca:latest \
    bash -i -c "lmdeploy chat --backend pytorch --device maca internlm/internlm2_5-7b-
↪chat"
```

1.3.3 Cambricon

The usage of lmdeploy on a Cambricon device is almost the same as its usage on CUDA with PytorchEngine in lmdeploy. Please read the original *Get Started* guide before reading this tutorial.

Here is the *supported model list*.

> [!IMPORTANT] We have uploaded a docker image to aliyun. Please try to pull the image by following command:
>
> `docker pull crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/lmdeploy_dlinfer/camb:latest`
>
> [!IMPORTANT] Currently, launching multi-device inference on Cambricon accelerators requires manually starting Ray.
>
> Below is an example for a 2-devices setup
>
> ```
> export MLU_VISIBLE_DEVICES=0,1
> ray start --head --resources='{"MLU": 2}'
> ```

Offline batch inference

LLM inference

Set `device_type="camb"` in the `PytorchEngineConfig`:

```
from lmdeploy import pipeline
from lmdeploy import PytorchEngineConfig
pipe = pipeline("internlm/internlm2_5-7b-chat",
        backend_config=PytorchEngineConfig(tp=1, device_type="camb"))
question = ["Shanghai is", "Please introduce China", "How are you?"]
response = pipe(question)
print(response)
```

VLM inference

Set `device_type="camb"` in the `PytorchEngineConfig`:

```
from lmdeploy import pipeline, PytorchEngineConfig
from lmdeploy.vl import load_image
pipe = pipeline('OpenGVLab/InternVL2-2B',
        backend_config=PytorchEngineConfig(tp=1, device_type='camb'))
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Online serving

Serve a LLM model

Add `--device camb` in the serve command.

```
lmdeploy serve api_server --backend pytorch --device camb internlm/internlm2_5-7b-chat
```

Run the following commands to launch docker container for lmdeploy LLM serving:

```
docker run -it --net=host crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
↪lmdeploy_dlinfer/camb:latest \
    bash -i -c "lmdeploy serve api_server --backend pytorch --device camb internlm/
↪internlm2_5-7b-chat"
```

Serve a VLM model

Add `--device camb` in the serve command

```
lmdeploy serve api_server --backend pytorch --device camb OpenGVLab/InternVL2-2B
```

Run the following commands to launch docker container for lmdeploy VLM serving:

```
docker run -it --net=host crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/
↪lmdeploy_dlinfer/camb:latest \
    bash -i -c "lmdeploy serve api_server --backend pytorch --device camb OpenGVLab/
↪InternVL2-2B"
```

Inference with Command line Interface

Add `--device camb` in the serve command.

```
lmdeploy chat internlm/internlm2_5-7b-chat --backend pytorch --device camb
```

Run the following commands to launch lmdeploy chatting after starting container:

```
docker run -it crpi-4crprmm5baj1v8iv.cn-hangzhou.personal.cr.aliyuncs.com/lmdeploy_
↪dlinfer/camb:latest \
    bash -i -c "lmdeploy chat --backend pytorch --device camb internlm/internlm2_5-7b-
↪chat"
```

1.4 Supported Models

The following tables detail the models supported by LMDeploy's TurboMind engine and PyTorch engine across different platforms.

1.4.1 TurboMind on CUDA Platform

Model	Size	Type	FP16/BF16	KV INT8	KV INT4	W4A16
Llama	7B - 65B	LLM	Yes	Yes	Yes	Yes
Llama2	7B - 70B	LLM	Yes	Yes	Yes	Yes
Llama3	8B, 70B	LLM	Yes	Yes	Yes	Yes
Llama3.1	8B, 70B	LLM	Yes	Yes	Yes	Yes
Llama3.2[2]	1B, 3B	LLM	Yes	Yes*	Yes*	Yes
InternLM	7B - 20B	LLM	Yes	Yes	Yes	Yes
InternLM2	7B - 20B	LLM	Yes	Yes	Yes	Yes
InternLM2.5	7B	LLM	Yes	Yes	Yes	Yes
InternLM3	8B	LLM	Yes	Yes	Yes	Yes
InternLM-XComposer2	7B, 4khd-7B	MLLM	Yes	Yes	Yes	Yes
InternLM-XComposer2.5	7B	MLLM	Yes	Yes	Yes	Yes
Intern-S1	241B	MLLM	Yes	Yes	Yes	No
Intern-S1-mini	8.3B	MLLM	Yes	Yes	Yes	No
Qwen	1.8B - 72B	LLM	Yes	Yes	Yes	Yes
Qwen1.5[1]	1.8B - 110B	LLM	Yes	Yes	Yes	Yes
Qwen2[2]	0.5B - 72B	LLM	Yes	Yes*	Yes*	Yes
Qwen2-MoE	57BA14B	LLM	Yes	Yes	Yes	Yes
Qwen2.5[2]	0.5B - 72B	LLM	Yes	Yes*	Yes*	Yes
Qwen3	0.6B-235B	LLM	Yes	Yes	Yes*	Yes*
Qwen3.5[3]	0.8B-397B	MLLM	Yes	Yes	No	Yes
Mistral[1]	7B	LLM	Yes	Yes	Yes	No
Mixtral	8x7B, 8x22B	LLM	Yes	Yes	Yes	Yes
DeepSeek-V2	16B, 236B	LLM	Yes	Yes	Yes	No
DeepSeek-V2.5	236B	LLM	Yes	Yes	Yes	No
Qwen-VL	7B	MLLM	Yes	Yes	Yes	Yes
DeepSeek-VL	7B	MLLM	Yes	Yes	Yes	Yes
Baichuan	7B	LLM	Yes	Yes	Yes	Yes
Baichuan2	7B	LLM	Yes	Yes	Yes	Yes
Code Llama	7B - 34B	LLM	Yes	Yes	Yes	No
YI	6B - 34B	LLM	Yes	Yes	Yes	Yes
LLaVA(1.5,1.6)	7B - 34B	MLLM	Yes	Yes	Yes	Yes
InternVL	v1.1 - v1.5	MLLM	Yes	Yes	Yes	Yes
InternVL2[2]	1 - 2B, 8B - 76B	MLLM	Yes	Yes*	Yes*	Yes
InternVL2.5(MPO)[2]	1 - 78B	MLLM	Yes	Yes*	Yes*	Yes
InternVL3[2]	1 - 78B	MLLM	Yes	Yes*	Yes*	Yes
InternVL3.5[3]	1 - 241BA28B	MLLM	Yes	Yes*	Yes*	No
ChemVLM	8B - 26B	MLLM	Yes	Yes	Yes	Yes
MiniCPM-Llama3-V-2_5	-	MLLM	Yes	Yes	Yes	Yes
MiniCPM-V-2_6	-	MLLM	Yes	Yes	Yes	Yes
GLM4	9B	LLM	Yes	Yes	Yes	Yes
CodeGeeX4	9B	LLM	Yes	Yes	Yes	-
Molmo	7B-D,72B	MLLM	Yes	Yes	Yes	No
gpt-oss	20B,120B	LLM	Yes	Yes	Yes	Yes

"-" means not verified yet.

Note

- [1] The TurboMind engine doesn't support window attention. Therefore, for models that have applied window attention and have the corresponding switch "use_sliding_window" enabled, such as Mistral, Qwen1.5 and etc., please choose the PyTorch engine for inference.
- [2] When the head_dim of a model is not 128, such as llama3.2-1B, qwen2-0.5B and internvl2-1B, turbomind doesn't support its kv cache 4/8 bit quantization and inference
- [3] TurboMind does not currently support the vision encoder for the Qwen3.5 series.

1.4.2 PyTorchEngine on CUDA Platform

Model	Size	Type	FP16/BF16	KV INT8	KV INT4	W8A8	W4A16
Llama	7B - 65B	LLM	Yes	Yes	Yes	Yes	Yes
Llama2	7B - 70B	LLM	Yes	Yes	Yes	Yes	Yes
Llama3	8B, 70B	LLM	Yes	Yes	Yes	Yes	Yes
Llama3.1	8B, 70B	LLM	Yes	Yes	Yes	Yes	Yes
Llama3.2	1B, 3B	LLM	Yes	Yes	Yes	Yes	Yes
Llama4	Scout, Maverick	MLLM	Yes	Yes	Yes	-	-
InternLM	7B - 20B	LLM	Yes	Yes	Yes	Yes	Yes
InternLM2	7B - 20B	LLM	Yes	Yes	Yes	Yes	Yes
InternLM2.5	7B	LLM	Yes	Yes	Yes	Yes	Yes
InternLM3	8B	LLM	Yes	Yes	Yes	Yes	Yes
Intern-S1	241B	MLLM	Yes	Yes	Yes	Yes	-
Intern-S1-mini	8.3B	MLLM	Yes	Yes	Yes	Yes	-
Intern-S1-Pro	1TB	MLLM	Yes	-	-	-	No
Baichuan2	7B	LLM	Yes	Yes	Yes	Yes	No
Baichuan2	13B	LLM	Yes	Yes	Yes	No	No
ChatGLM2	6B	LLM	Yes	Yes	Yes	No	No
YI	6B - 34B	LLM	Yes	Yes	Yes	Yes	Yes
Mistral	7B	LLM	Yes	Yes	Yes	Yes	Yes
Mixtral	8x7B, 8x22B	LLM	Yes	Yes	Yes	No	No
QWen	1.8B - 72B	LLM	Yes	Yes	Yes	Yes	Yes
QWen1.5	0.5B - 110B	LLM	Yes	Yes	Yes	Yes	Yes
QWen1.5-MoE	A2.7B	LLM	Yes	Yes	Yes	No	No
QWen2	0.5B - 72B	LLM	Yes	Yes	No	Yes	Yes
Qwen2.5	0.5B - 72B	LLM	Yes	Yes	No	Yes	Yes
Qwen3	0.6B - 235B	LLM	Yes	Yes	Yes*	-	Yes*
QWen3-Next	80B	LLM	Yes	No	No	No	No
QWen2-VL	2B, 7B	MLLM	Yes	Yes	No	No	Yes
QWen2.5-VL	3B - 72B	MLLM	Yes	No	No	No	No
QWen3-VL	2B - 235B	MLLM	Yes	No	No	No	No
QWen3.5	0.8B-397B	MLLM	Yes	No	No	No	No
DeepSeek-MoE	16B	LLM	Yes	No	No	No	No
DeepSeek-V2	16B, 236B	LLM	Yes	No	No	No	No
DeepSeek-V2.5	236B	LLM	Yes	No	No	No	No
DeepSeek-V3	685B	LLM	Yes	No	No	No	No
DeepSeek-V3.2	685B	LLM	Yes	No	No	No	No
DeepSeek-VL2	3B - 27B	MLLM	Yes	No	No	No	No
MiniCPM3	4B	LLM	Yes	Yes	Yes	No	No
MiniCPM-V-2_6	8B	LLM	Yes	No	No	No	Yes

continues on next page

Table 2 – continued from previous page

Model	Size	Type	FP16/BF16	KV INT8	KV INT4	W8A8	W4A16
Gemma	2B-7B	LLM	Yes	Yes	Yes	No	No
StarCoder2	3B-15B	LLM	Yes	Yes	Yes	No	No
Phi-3-mini	3.8B	LLM	Yes	Yes	Yes	Yes	Yes
Phi-3-vision	4.2B	MLLM	Yes	Yes	Yes	-	-
Phi-4-mini	3.8B	LLM	Yes	Yes	Yes	Yes	Yes
CogVLM-Chat	17B	MLLM	Yes	Yes	Yes	-	-
CogVLM2-Chat	19B	MLLM	Yes	Yes	Yes	-	-
LLaVA(1.5,1.6)[2]	7B-34B	MLLM	No	No	No	No	No
InternVL(v1.5)	2B-26B	MLLM	Yes	Yes	Yes	No	Yes
InternVL2	1B-76B	MLLM	Yes	Yes	Yes	-	-
InternVL2.5(MPO)	1B-78B	MLLM	Yes	Yes	Yes	-	-
InternVL3	1B-78B	MLLM	Yes	Yes	Yes	-	-
InternVL3.5	1B-241BA28B	MLLM	Yes	Yes	Yes	No	No
Mono-InternVL[1]	2B	MLLM	Yes	Yes	Yes	-	-
ChemVLM	8B-26B	MLLM	Yes	Yes	No	-	-
Gemma2	9B-27B	LLM	Yes	Yes	Yes	-	-
Gemma3	1B-27B	MLLM	Yes	Yes	Yes	-	-
GLM-4	9B	LLM	Yes	Yes	Yes	No	No
GLM-4-0414	9B	LLM	Yes	Yes	Yes	-	-
GLM-4V	9B	MLLM	Yes	Yes	Yes	No	Yes
GLM-4.1V-Thinking	9B	MLLM	Yes	Yes	Yes	-	-
GLM-4.5	355B	LLM	Yes	Yes	Yes	-	-
GLM-4.5-Air	106B	LLM	Yes	Yes	Yes	-	-
CodeGeeX4	9B	LLM	Yes	Yes	Yes	-	-
Phi-3.5-mini	3.8B	LLM	Yes	Yes	No	-	-
Phi-3.5-MoE	16x3.8B	LLM	Yes	Yes	No	-	-
Phi-3.5-vision	4.2B	MLLM	Yes	Yes	No	-	-
SDAR	1.7B-30B	LLM	Yes	Yes	No	-	-
GLM-4.7-Flash	30B	LLM	Yes	No	No	No	No
GLM-5	754B	LLM	Yes	No	No	No	No

Note

- [1] Currently Mono-InternVL does not support FP16 due to numerical instability. Please use BF16 instead.
- [2] PyTorch engine removes the support of original llava models after v0.6.4. Please use their corresponding transformers models instead, which can be found in https://huggingface.co/llava-hf Starting from version 0.11.1, PytorchEngine no longer provides support for mllama.

1.4.3 PyTorchEngine on Other Platforms

			Atlas 800T A2	Atlas 800T A2	Atlas 800T A2	Atlas 800T A2	Atlas 300I Duo	Atlas 800T A3	Maca C500	Cambricon
Model	Size	Type	FP16/BF16	FP16/BF16	W8A8(gr	W4A16(e	FP16(grap	FP16/BF16	BF/FP1	BF/FP16
Llama2	7B - 70B	LLM	Yes	Yes	Yes	Yes	-	Yes	Yes	Yes
Llama3	8B	LLM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Llama3.1	8B	LLM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
InternLM2	7B - 20B	LLM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
InternLM2.	7B - 20B	LLM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
InternLM3	8B	LLM	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Mixtral	8x7B	LLM	Yes	Yes	No	No	Yes	-	Yes	Yes
QWen1.5 MoE	A2.7B	LLM	Yes	-	No	No	-	-	Yes	-
QWen2(.5	7B	LLM	Yes	Yes	Yes	Yes	Yes	-	Yes	Yes
QWen2-VL	2B, 7B	MLL	Yes	Yes	-	-	-	-	Yes	No
QWen2.5 VL	3B - 72B	MLL	Yes	Yes	-	-	Yes	-	Yes	No
QWen2-MoE	A14.57	LLM	Yes	-	No	No	-	-	Yes	-
QWen3	0.6B-235B	LLM	Yes	Yes	No	No	Yes	Yes	Yes	Yes
DeepSeek V2	16B	LLM	No	Yes	No	No	-	-	-	-
InternVL(v	2B-26B	MLL	Yes	-	Yes	Yes	-	-	Yes	-
InternVL2	1B-40B	MLL	Yes	Yes	Yes	Yes	Yes	-	Yes	Yes
InternVL2.5	1B-78B	MLL	Yes	Yes	Yes	Yes	Yes	-	Yes	Yes
InternVL3	1B-78B	MLL	Yes	Yes	Yes	Yes	Yes	-	Yes	Yes
CogVLM chat	19B	MLL	Yes	No	-	-	-	-	Yes	-
GLM4V	9B	MLL	Yes	No	-	-	-	-	-	-

1.5 Reward Models

LMDeploy supports reward models, which are detailed in the table below:

Model	Size	Supported Inference Engine
Qwen2.5-Math-RM	72B	PyTorch
InternLM2-Reward	1.8B, 7B, 20B	PyTorch
POLAR	1.8B, 7B	PyTorch

1.5.1 Offline Inference

We take `internlm/internlm2-1_8b-reward` as an example:

```
from transformers import AutoTokenizer
from lmdeploy import pipeline, PytorchEngineConfig

model_path = "internlm/internlm2-1_8b-reward"
chat = [
    {"role": "system", "content": "Please reason step by step, and put your final answer within \\boxed{}."},
    {"role": "user", "content": "Janet's ducks lay 16 eggs per day. She eats three for breakfast every morning and bakes muffins for her friends every day with four. She sells the remainder at the farmers' market daily for $2 per fresh duck egg. How much in dollars does she make every day at the farmers' market?"},
    {"role": "assistant", "content": "To determine how much Janet makes from selling the duck eggs at the farmers' market, we need to follow these steps:\n\n1. Calculate the total number of eggs laid by the ducks each day.\n2. Determine how many eggs Janet eats and bakes for herself each day.\n3. Find out how many eggs are left to be sold.\n4. Calculate the revenue from selling the remaining eggs at $2 per egg.\n\nLet's start with the first step:\n\n1. Janet's ducks lay 16 eggs per day.\n\nNext, we calculate how many eggs Janet eats and bakes for herself each day:\n\n2. Janet eats 3 eggs for breakfast every morning.\n3. Janet bakes 4 eggs for her friends every day.\n\nSo, the total number of eggs Janet eats and bakes for herself each day is:\n\\[ 3 + 4 = 7 \\text{ eggs} \\]\n\nNow, we find out how many eggs are left to be sold:\n\\[ 16 - 7 = 9 \\text{ eggs} \\]\n\nFinally, we calculate the revenue from selling the remaining eggs at $2 per egg:\n\\[ 9 \\times 2 = 18 \\text{ dollars} \\]\n\nTherefore, Janet makes 18 dollars every day at the farmers' market."}
]

tokenizer = AutoTokenizer.from_pretrained(model_path, trust_remote_code=True)

conversation_str = tokenizer.apply_chat_template(
    chat,
    tokenize=False,
    add_generation_prompt=False
)

input_ids = tokenizer.encode(
    conversation_str,
    add_special_tokens=False
```

(continues on next page)

(continued from previous page)

```
)


if __name__ == '__main__':
    engine_config = PytorchEngineConfig(tp=tp)
    with pipeline(model_path, backend_config=engine_config) as pipe:
        score = pipe.get_reward_score(input_ids)
        print(f'score: {score}')
```

1.5.2 Online Inference

Start the API server:

```
lmdeploy serve api_server internlm/internlm2-1_8b-reward --backend pytorch
```

Get the reward score from the `/pooling` API endpoint:

```
curl http://0.0.0.0:23333/pooling \
  -H "Content-Type: application/json" \
  -d '{
    "model": "internlm/internlm2-1_8b-reward",
    "input": "Who are you?"
  }'
```

1.6 Offline Inference Pipeline

In this tutorial, We will present a list of examples to introduce the usage of `lmdeploy.pipeline`.

You can overview the detailed pipeline API in this guide.

1.6.1 Usage

A 'Hello, world' example

```
from lmdeploy import pipeline

pipe = pipeline('internlm/internlm2_5-7b-chat')
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

In this example, the pipeline by default allocates a predetermined percentage of GPU memory for storing k/v cache. The ratio is dictated by the parameter `TurbomindEngineConfig.cache_max_entry_count`.

There have been alterations to the strategy for setting the k/v cache ratio throughout the evolution of LMDeploy. The following are the change histories:

1. `v0.2.0 <= lmdeploy <= v0.2.1`

   `TurbomindEngineConfig.cache_max_entry_count` defaults to 0.5, indicating 50% GPU **total memory** allocated for k/v cache. Out Of Memory (OOM) errors may occur if a 7B model is deployed on a GPU with

memory less than 40G. If you encounter an OOM error, please decrease the ratio of the k/v cache occupation as follows:

```
from lmdeploy import pipeline, TurbomindEngineConfig

# decrease the ratio of the k/v cache occupation to 20%
backend_config = TurbomindEngineConfig(cache_max_entry_count=0.2)

pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

2. `lmdeploy > v0.2.1`

   The allocation strategy for k/v cache is changed to reserve space from the **GPU free memory** proportionally. The ratio `TurbomindEngineConfig.cache_max_entry_count` has been adjusted to 0.8 by default. If OOM error happens, similar to the method mentioned above, please consider reducing the ratio value to decrease the memory usage of the k/v cache.

Set tensor parallelism

```
from lmdeploy import pipeline, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
print(response)
```

Set sampling parameters

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'],
                gen_config=gen_config)
print(response)
```

Apply OpenAI format prompt

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
response = pipe(prompts,
                gen_config=gen_config)
print(response)
```

Apply streaming output

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(tp=2)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
for item in pipe.stream_infer(prompts, gen_config=gen_config):
    print(item)
```

Get logits for generated tokens

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('internlm/internlm2_5-7b-chat')

gen_config=GenerationConfig(output_logits='generation',
                            max_new_tokens=10)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'],
                gen_config=gen_config)
logits = [x.logits for x in response]
```

Get last layer's hidden states for generated tokens

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('internlm/internlm2_5-7b-chat')

gen_config=GenerationConfig(output_last_hidden_state='generation',
                            max_new_tokens=10)
response = pipe(['Hi, pls intro yourself', 'Shanghai is'],
                gen_config=gen_config)
hidden_states = [x.last_hidden_state for x in response]
```

Calculate ppl

```
from transformers import AutoTokenizer
from lmdeploy import pipeline


model_repoid_or_path = 'internlm/internlm2_5-7b-chat'
pipe = pipeline(model_repoid_or_path)
tokenizer = AutoTokenizer.from_pretrained(model_repoid_or_path, trust_remote_code=True)
messages = [
   {"role": "user", "content": "Hello, how are you?"},
]
input_ids = tokenizer.apply_chat_template(messages)

# ppl is a list of float numbers
ppl = pipe.get_ppl(input_ids)
print(ppl)
```

Note

- When input_ids is too long, an OOM (Out Of Memory) error may occur. Please apply it with caution
- get_ppl returns the cross entropy loss without applying the exponential operation afterwards

Use PyTorchEngine

```
pip install triton>=2.1.0
```

```
from lmdeploy import pipeline, GenerationConfig, PytorchEngineConfig

backend_config = PytorchEngineConfig(session_len=2048)
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('internlm/internlm2_5-7b-chat',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': 'Hi, pls intro yourself'
}], [{
    'role': 'user',
    'content': 'Shanghai is'
}]]
response = pipe(prompts, gen_config=gen_config)
print(response)
```

Inference with LoRA

```
from lmdeploy import pipeline, GenerationConfig, PytorchEngineConfig

backend_config = PytorchEngineConfig(session_len=2048,
                                     adapters=dict(lora_name_1='chenchi/lora-chatglm2-6b-
↪guodegang'))
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
pipe = pipeline('THUDM/chatglm2-6b',
                backend_config=backend_config)
prompts = [[{
    'role': 'user',
    'content': ''
}]]
response = pipe(prompts, gen_config=gen_config, adapter_name='lora_name_1')
print(response)
```

Release pipeline

You can release the pipeline explicitly by calling its `close()` method, or alternatively, use the `with` statement as demonstrated below:

```
from lmdeploy import pipeline

with pipeline('internlm/internlm2_5-7b-chat') as pipe:
    response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
    print(response)
```

1.6.2 FAQs

- **RuntimeError: An attempt has been made to start a new process before the current process has finished its bootstrapping phase**.

  If you got this for tp>1 in pytorch backend. Please make sure the python script has following

  ```
  if __name__ == '__main__':
  ```

  Generally, in the context of multi-threading or multi-processing, it might be necessary to ensure that initialization code is executed only once. In this case, `if __name__ == '__main__':` can help to ensure that these initialization codes are run only in the main program, and not repeated in each newly created process or thread.

- To customize a chat template, please refer to *chat_template.md*.
- If the weight of lora has a corresponding chat template, you can first register the chat template to lmdeploy, and then use the chat template name as the adapter name.

1.7 OpenAI Compatible Server

This article primarily discusses the deployment of a single LLM model across multiple GPUs on a single node, providing a service that is compatible with the OpenAI interface, as well as the usage of the service API. For the sake of convenience, we refer to this service as `api_server`. Regarding parallel services with multiple models, please refer to the guide about *Request Distribution Server*.

In the following sections, we will first introduce methods for starting the service, choosing the appropriate one based on your application scenario.

Next, we focus on the definition of the service's RESTful API, explore the various ways to interact with the interface, and demonstrate how to try the service through the Swagger UI or LMDeploy CLI tools.

1.7.1 Launch Service

Take the internlm2_5-7b-chat model hosted on huggingface hub as an example, you can choose one the following methods to start the service.

Option 1: Launching with lmdeploy CLI

```
lmdeploy serve api_server internlm/internlm2_5-7b-chat --server-port 23333
```

The arguments of `api_server` can be viewed through the command `lmdeploy serve api_server -h`, for instance, `--tp` to set tensor parallelism, `--session-len` to specify the max length of the context window, `--cache-max-entry-count` to adjust the GPU mem ratio for k/v cache etc.

Option 2: Deploying with docker

With LMDeploy official docker image, you can run OpenAI compatible server as follows:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server internlm/internlm2_5-7b-chat
```

The parameters of `api_server` are the same with that mentioned in "*option 1*" section

Option 3: Deploying to Kubernetes cluster

Connect to a running Kubernetes cluster and deploy the internlm2_5-7b-chat model service with kubectl command-line tool (replace `<your token>` with your huggingface hub token):

```
sed 's/{{HUGGING_FACE_HUB_TOKEN}}/<your token>/' k8s/deployment.yaml | kubectl create -f␣
↪- \
    && kubectl create -f k8s/service.yaml
```

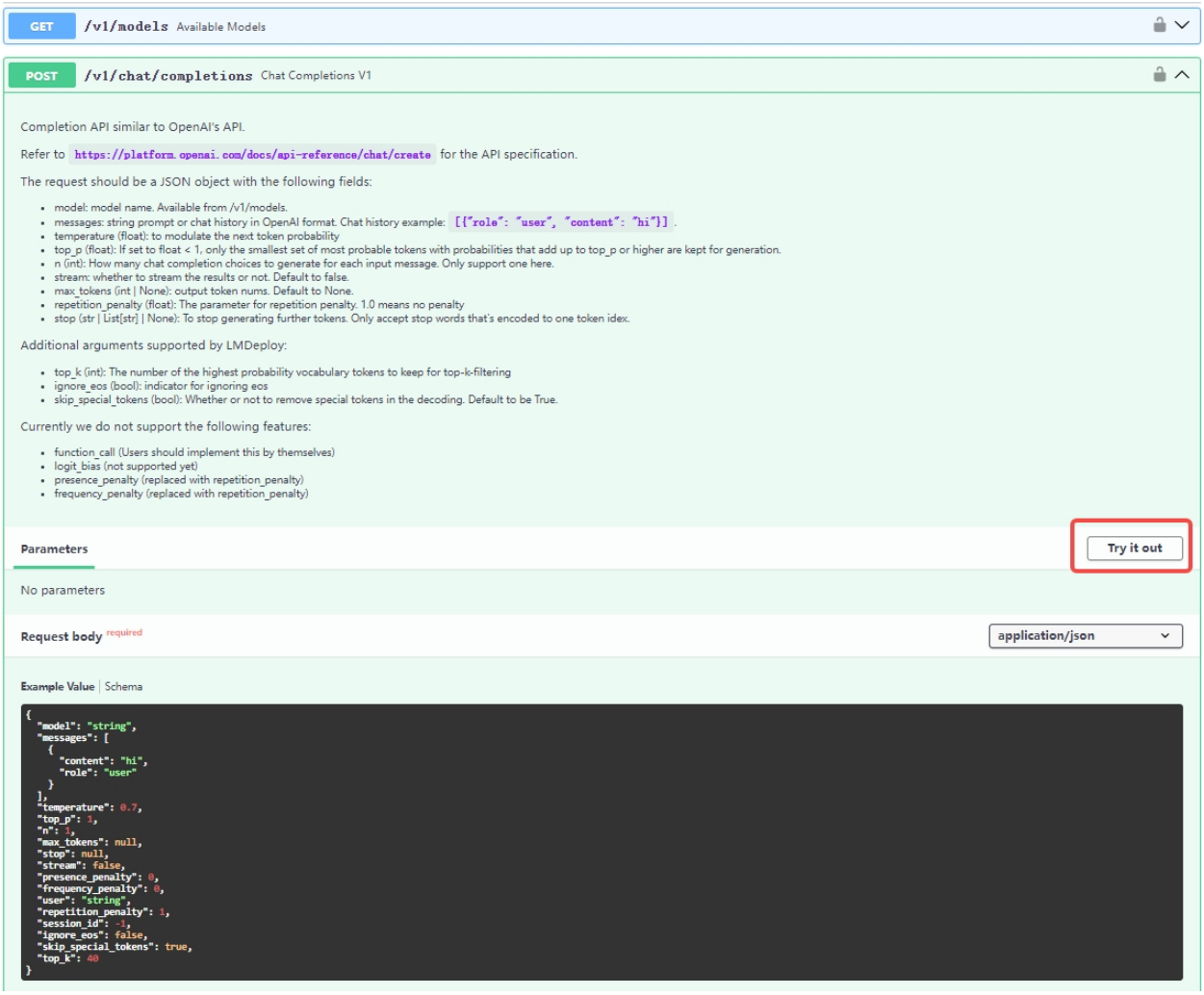
In the example above the model data is placed on the local disk of the node (hostPath). Consider replacing it with high-availability shared storage if multiple replicas are desired, and the storage can be mounted into container using PersistentVolume.

1.7.2 RESTful API

LMDeploy's RESTful API is compatible with the following three OpenAI interfaces:

- /v1/chat/completions
- /v1/models
- /v1/completions

You can overview and try out the offered RESTful APIs by the website `http://0.0.0.0:23333` as shown in the below image after launching the service successfully.



If you need to integrate the service into your own projects or products, we recommend the following approach:

Integrate with `OpenAI`

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```
from openai import OpenAI
client = OpenAI(
    api_key='YOUR_API_KEY',
    base_url="http://0.0.0.0:23333/v1"
)
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
  model=model_name,
  messages=[
    {"role": "system", "content": "You are a helpful assistant."},
    {"role": "user", "content": " provide three suggestions about time management"},
  ],
    temperature=0.8,
```

(continues on next page)

(continued from previous page)

```
    top_p=0.8
)
print(response)
```

If you want to use async functions, may try the following example:

```
import asyncio
from openai import AsyncOpenAI

async def main():
    client = AsyncOpenAI(api_key='YOUR_API_KEY',
                         base_url='http://0.0.0.0:23333/v1')
    model_cards = await client.models.list()._get_page()
    response = await client.chat.completions.create(
        model=model_cards.data[0].id,
        messages=[
            {
                'role': 'system',
                'content': 'You are a helpful assistant.'
            },
            {
                'role': 'user',
                'content': ' provide three suggestions about time management'
            },
        ],
        temperature=0.8,
        top_p=0.8)
    print(response)

asyncio.run(main())
```

You can invoke other OpenAI interfaces using similar methods. For more detailed information, please refer to the OpenAI API guide

Integrate with lmdeploy `APIClient`

Below are some examples demonstrating how to visit the service through `APIClient`

If you want to use the `/v1/chat/completions` endpoint, you can try the following code:

```
from lmdeploy.serve.openai.api_client import APIClient
api_client = APIClient('http://{server_ip}:{server_port}')
model_name = api_client.available_models[0]
messages = [{"role": "user", "content": "Say this is a test!"}]
for item in api_client.chat_completions_v1(model=model_name, messages=messages):
    print(item)
```

For the `/v1/completions` endpoint, you can try:

```
from lmdeploy.serve.openai.api_client import APIClient
api_client = APIClient('http://{server_ip}:{server_port}')
model_name = api_client.available_models[0]
```

(continues on next page)

(continued from previous page)

```
for item in api_client.completions_v1(model=model_name, prompt='hi'):
    print(item)
```

Tools

May refer to *api_server_tools*.

Anthropic-Compatible Endpoints

May refer to *api_server_anthropic*.

Integrate with Java/Golang/Rust

May use openapi-generator-cli to convert `http://{server_ip}:{server_port}/openapi.json` to java/rust/golang client. Here is an example:

```
$ docker run -it --rm -v ${PWD}:/local openapitools/openapi-generator-cli generate -i /
↪local/openapi.json -g rust -o /local/rust

$ ls rust/*
rust/Cargo.toml  rust/git_push.sh  rust/README.md

rust/docs:
ChatCompletionRequest.md  EmbeddingsRequest.md  HttpValidationError.md  LocationInner.md␣
↪ Prompt.md
DefaultApi.md             GenerateRequest.md    Input.md                Messages.md     ␣
↪ ValidationError.md

rust/src:
apis  lib.rs  models
```

Integrate with cURL

cURL is a tool for observing the output of the RESTful APIs.

- list served models `v1/models`

```
curl http://{server_ip}:{server_port}/v1/models
```

- chat `v1/chat/completions`

```
curl http://{server_ip}:{server_port}/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "internlm-chat-7b",
    "messages": [{"role": "user", "content": "Hello! How are you?"}]
  }'
```

- text completions `v1/completions`

```
curl http://{server_ip}:{server_port}/v1/completions \
  -H 'Content-Type: application/json' \
  -d '{
  "model": "llama",
  "prompt": "two steps to build a house:"
}'
```

1.7.3 Launch multiple api servers

Following are two steps to launch multiple api servers through torchrun. Just create a python script with the following codes.

1. Launch the proxy server through `lmdeploy serve proxy`. Get the correct proxy server url.
2. Launch the script through `torchrun --nproc_per_node 2 script.py InternLM/internlm2-chat-1_8b --proxy_url http://{proxy_node_name}:{proxy_node_port}`.**Note**: Please do not use `0.0.0.0:8000` here, instead, we input the real ip name, `11.25.34.55:8000` for example.

```
import os
import socket
from typing import List, Literal

import fire


def get_host_ip():
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(('8.8.8.8', 80))
        ip = s.getsockname()[0]
    finally:
        s.close()
    return ip


def main(model_path: str,
         tp: int = 1,
         proxy_url: str = 'http://0.0.0.0:8000',
         port: int = 23333,
         backend: Literal['turbomind', 'pytorch'] = 'turbomind'):
    local_rank = int(os.environ.get('LOCAL_RANK', -1))
    world_size = int(os.environ.get('WORLD_SIZE', -1))
    local_ip = get_host_ip()
    if isinstance(port, List):
        assert len(port) == world_size
        port = port[local_rank]
    else:
        port += local_rank * 10
    if (world_size - local_rank) % tp == 0:
        rank_list = ','.join([str(local_rank + i) for i in range(tp)])
        command = f'CUDA_VISIBLE_DEVICES={rank_list} lmdeploy serve api_server {model_
↪path} '\
```

(continues on next page)

(continued from previous page)

```
                f'--server-name {local_ip} --server-port {port} --tp {tp} '\
                f'--proxy-url {proxy_url} --backend {backend}'
        print(f'running command: {command}')
        os.system(command)


if __name__ == '__main__':
    fire.Fire(main)
```

1.7.4 FAQ

1. When user got `"finish_reason":"length"`, it means the session is too long to be continued. The session length can be modified by passing `--session_len` to api_server.
2. When OOM appeared at the server side, please reduce the `cache_max_entry_count` of `backend_config` when launching the service.
3. Regarding the stop words, we only support characters that encode into a single index. Furthermore, there may be multiple indexes that decode into results containing the stop word. In such cases, if the number of these indexes is too large, we will only use the index encoded by the tokenizer. If you want use a stop symbol that encodes into multiple indexes, you may consider performing string matching on the streaming client side. Once a successful match is found, you can then break out of the streaming loop.
4. To customize a chat template, please refer to *chat_template.md*.

1.8 Tools Calling

LMDeploy supports tools for InternLM2, InternLM2.5, llama3.1 and Qwen2.5 models. Please use `--tool-call-parser` to specify which parser to use when launching the api_server. Supported names are:

1. internlm
2. qwen
3. llama3

1.8.1 Single Round Invocation

Please start the service of models before running the following example.

```
from openai import OpenAI

tools = [
  {
    "type": "function",
    "function": {
      "name": "get_current_weather",
      "description": "Get the current weather in a given location",
      "parameters": {
        "type": "object",
        "properties": {
```

(continues on next page)

(continued from previous page)

```
          "location": {
            "type": "string",
            "description": "The city and state, e.g. San Francisco, CA",
          },
          "unit": {"type": "string", "enum": ["celsius", "fahrenheit"]},
        },
        "required": ["location"],
      },
    }
  }
]
messages = [{"role": "user", "content": "What's the weather like in Boston today?"}]

client = OpenAI(api_key='YOUR_API_KEY',base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response)
```

1.8.2 Multiple Round Invocation

InternLM

A complete toolchain invocation process can be demonstrated through the following example.

```
from openai import OpenAI


def add(a: int, b: int):
    return a + b


def mul(a: int, b: int):
    return a * b


tools = [{
    'type': 'function',
    'function': {
        'name': 'add',
        'description': 'Compute the sum of two numbers',
        'parameters': {
            'type': 'object',
            'properties': {
                'a': {
                    'type': 'int',
```

(continues on next page)

(continued from previous page)

```
                    'description': 'A number',
                },
                'b': {
                    'type': 'int',
                    'description': 'A number',
                },
            },
            'required': ['a', 'b'],
        },
    }
}, {
    'type': 'function',
    'function': {
        'name': 'mul',
        'description': 'Calculate the product of two numbers',
        'parameters': {
            'type': 'object',
            'properties': {
                'a': {
                    'type': 'int',
                    'description': 'A number',
                },
                'b': {
                    'type': 'int',
                    'description': 'A number',
                },
            },
            'required': ['a', 'b'],
        },
    }
}]
messages = [{'role': 'user', 'content': 'Compute (3+5)*2'}]

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response)
func1_name = response.choices[0].message.tool_calls[0].function.name
func1_args = response.choices[0].message.tool_calls[0].function.arguments
func1_out = eval(f'{func1_name}(**{func1_args})')
print(func1_out)

messages.append(response.choices[0].message)
messages.append({
    'role': 'tool',
    'content': f'3+5={func1_out}',
```

(continues on next page)

(continued from previous page)

```
    'tool_call_id': response.choices[0].message.tool_calls[0].id
})
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response)
func2_name = response.choices[0].message.tool_calls[0].function.name
func2_args = response.choices[0].message.tool_calls[0].function.arguments
func2_out = eval(f'{func2_name}(**{func2_args})')
print(func2_out)
```

Using the InternLM2-Chat-7B model to execute the above example, the following results will be printed.

```
ChatCompletion(id='1', choices=[Choice(finish_reason='tool_calls', index=0,␣
↪logprobs=None, message=ChatCompletionMessage(content='', role='assistant', function_
↪call=None, tool_calls=[ChatCompletionMessageToolCall(id='0',␣
↪function=Function(arguments='{"a": 3, "b": 5}', name='add'), type='function')]))],␣
↪created=1722852901, model='/nvme/shared_data/InternLM/internlm2-chat-7b', object='chat.
↪completion', system_fingerprint=None, usage=CompletionUsage(completion_tokens=25,␣
↪prompt_tokens=263, total_tokens=288))
8
ChatCompletion(id='2', choices=[Choice(finish_reason='tool_calls', index=0,␣
↪logprobs=None, message=ChatCompletionMessage(content='', role='assistant', function_
↪call=None, tool_calls=[ChatCompletionMessageToolCall(id='1',␣
↪function=Function(arguments='{"a": 8, "b": 2}', name='mul'), type='function')]))],␣
↪created=1722852901, model='/nvme/shared_data/InternLM/internlm2-chat-7b', object='chat.
↪completion', system_fingerprint=None, usage=CompletionUsage(completion_tokens=25,␣
↪prompt_tokens=293, total_tokens=318))
16
```

Llama 3.1

Meta announces in Llama3's official user guide that,

> There are three built-in tools (brave_search, wolfram_alpha, and code interpreter) can be turned on using the system prompt:
>
> 1. Brave Search: Tool call to perform web searches.
> 2. Wolfram Alpha: Tool call to perform complex mathematical calculations.
> 3. Code Interpreter: Enables the model to output python code.

Additionally, it cautions: "**Note:** We recommend using Llama 70B-instruct or Llama 405B-instruct for applications that combine conversation and tool calling. Llama 8B-Instruct can not reliably maintain a conversation alongside tool calling definitions. It can be used for zero-shot tool calling, but tool instructions should be removed for regular conversations between the model and the user."

Therefore, we utilize Meta-Llama-3.1-70B-Instruct to show how to invoke the tool calling by LMDeploy `api_server`.

On a A100-SXM-80G node, you can start the service as follows:

```
lmdeploy serve api_server /the/path/of/Meta-Llama-3.1-70B-Instruct/model --tp 4
```

For an in-depth understanding of the api_server, please refer to the detailed documentation available *here*.

The following code snippet demonstrates how to utilize the 'Wolfram Alpha' tool. It is assumed that you have already registered on the Wolfram Alpha website and obtained an API key. Please ensure that you have a valid API key to access the services provided by Wolfram Alpha

```
from openai import OpenAI
import requests


def request_llama3_1_service(messages):
    client = OpenAI(api_key='YOUR_API_KEY',
                    base_url='http://0.0.0.0:23333/v1')
    model_name = client.models.list().data[0].id
    response = client.chat.completions.create(
        model=model_name,
        messages=messages,
        temperature=0.8,
        top_p=0.8,
        stream=False)
    return response.choices[0].message.content


# The role of "system" MUST be specified, including the required tools
messages = [
    {
        "role": "system",
        "content": "Environment: ipython\nTools: wolfram_alpha\n\n Cutting Knowledge␣
↪Date: December 2023\nToday Date: 23 Jul 2024\n\nYou are a helpful Assistant." # noqa
    },
    {
        "role": "user",
        "content": "Can you help me solve this equation: x^3 - 4x^2 + 6x - 24 = 0"  #␣
↪noqa
    }
]

# send request to the api_server of llama3.1-70b and get the response
# the "assistant_response" is supposed to be:
# <|python_tag|>wolfram_alpha.call(query="solve x^3 - 4x^2 + 6x - 24 = 0")
assistant_response = request_llama3_1_service(messages)
print(assistant_response)

# Call the API of Wolfram Alpha with the query generated by the model
app_id = 'YOUR-Wolfram-Alpha-API-KEY'
params = {
    "input": assistant_response,
    "appid": app_id,
    "format": "plaintext",
    "output": "json",
}
```

(continues on next page)

(continued from previous page)

```
wolframalpha_response = requests.get(
    "https://api.wolframalpha.com/v2/query",
    params=params
)
wolframalpha_response = wolframalpha_response.json()

# Append the contents obtained by the model and the wolframalpha's API
# to "messages", and send it again to the api_server
messages += [
    {
        "role": "assistant",
        "content": assistant_response
    },
    {
        "role": "ipython",
        "content": wolframalpha_response
    }
]

assistant_response = request_llama3_1_service(messages)
print(assistant_response)
```

Qwen2.5

Qwen2.5 supports multi tool calling, which means that multiple tool requests can be initiated in one request

```
from openai import OpenAI
import json

def get_current_temperature(location: str, unit: str = "celsius"):
    """Get current temperature at a location.

    Args:
        location: The location to get the temperature for, in the format "City, State,␣
↪Country".
        unit: The unit to return the temperature in. Defaults to "celsius". (choices: [
↪"celsius", "fahrenheit"])

    Returns:
        the temperature, the location, and the unit in a dict
    """
    return {
        "temperature": 26.1,
        "location": location,
        "unit": unit,
    }


def get_temperature_date(location: str, date: str, unit: str = "celsius"):
    """Get temperature at a location and date.
```

(continues on next page)

(continued from previous page)

```
    Args:
        location: The location to get the temperature for, in the format "City, State,␣
↪Country".
        date: The date to get the temperature for, in the format "Year-Month-Day".
        unit: The unit to return the temperature in. Defaults to "celsius". (choices: [
↪"celsius", "fahrenheit"])

    Returns:
        the temperature, the location, the date and the unit in a dict
    """
    return {
        "temperature": 25.9,
        "location": location,
        "date": date,
        "unit": unit,
    }

def get_function_by_name(name):
    if name == "get_current_temperature":
        return get_current_temperature
    if name == "get_temperature_date":
        return get_temperature_date

tools = [{
    'type': 'function',
    'function': {
        'name': 'get_current_temperature',
        'description': 'Get current temperature at a location.',
        'parameters': {
            'type': 'object',
            'properties': {
                'location': {
                    'type': 'string',
                    'description': 'The location to get the temperature for, in the␣
↪format \'City, State, Country\'.'
                },
                'unit': {
                    'type': 'string',
                    'enum': [
                        'celsius',
                        'fahrenheit'
                    ],
                    'description': 'The unit to return the temperature in. Defaults to \
↪'celsius\'.'
                }
            },
            'required': [
                'location'
            ]
        }
    }
```

(continues on next page)

(continued from previous page)

```
}, {
    'type': 'function',
    'function': {
        'name': 'get_temperature_date',
        'description': 'Get temperature at a location and date.',
        'parameters': {
            'type': 'object',
            'properties': {
                'location': {
                    'type': 'string',
                    'description': 'The location to get the temperature for, in the␣
↪format \'City, State, Country\'.'
                },
                'date': {
                    'type': 'string',
                    'description': 'The date to get the temperature for, in the format \
↪'Year-Month-Day\'.'
                },
                'unit': {
                    'type': 'string',
                    'enum': [
                        'celsius',
                        'fahrenheit'
                    ],
                    'description': 'The unit to return the temperature in. Defaults to \
↪'celsius\'.'
                }
            },
            'required': [
                'location',
                'date'
            ]
        }
    }
}]
messages = [{'role': 'user', 'content': 'Today is 2024-11-14, What\'s the temperature in␣
↪San Francisco now? How about tomorrow?'}]

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response.choices[0].message.tool_calls)
messages.append(response.choices[0].message)

for tool_call in response.choices[0].message.tool_calls:
    tool_call_args = json.loads(tool_call.function.arguments)
```

(continues on next page)

(continued from previous page)

```
    tool_call_result =  get_function_by_name(tool_call.function.name)(**tool_call_args)
    messages.append({
        'role': 'tool',
        'name': tool_call.function.name,
        'content': tool_call_result,
        'tool_call_id': tool_call.id
    })

response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    top_p=0.8,
    stream=False,
    tools=tools)
print(response.choices[0].message.content)
```

Using the Qwen2.5-14B-Instruct, similar results can be obtained as follows

```
[ChatCompletionMessageToolCall(id='0', function=Function(arguments='{"location": "San␣
↪Francisco, California, USA"}', name='get_current_temperature'), type='function'),
 ChatCompletionMessageToolCall(id='1', function=Function(arguments='{"location": "San␣
↪Francisco, California, USA", "date": "2024-11-15"}', name='get_temperature_date'),␣
↪type='function')]

The current temperature in San Francisco, California, USA is 26.1°C. For tomorrow, 2024-
↪11-15, the temperature is expected to be 25.9°C.
```

It is important to note that in scenarios involving multiple tool calls, the order of the tool call results can affect the response quality. The tool_call_id has not been correctly provided to the LLM.

1.9 Reasoning Outputs

For models that support reasoning capabilities, such as DeepSeek R1, LMDeploy can parse reasoning outputs on the server side and expose them via `reasoning_content`.

1.9.1 Examples

DeepSeek R1

We can start DeepSeek R1's `api_server` like other models, but we need to specify the `--reasoning-parser` argument.

```
lmdeploy serve api_server deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B --reasoning-parser␣
↪deepseek-r1
```

Then, we can call the service's functionality from the client:

```python
from openai import OpenAI

openai_api_key = "Your API key"
openai_api_base = "http://0.0.0.0:23333/v1"

client = OpenAI(
    api_key=openai_api_key,
    base_url=openai_api_base,
)

models = client.models.list()
model = models.data[0].id

messages = [{"role": "user", "content": "9.11 and 9.8, which is greater?"}]
response = client.chat.completions.create(model=model, messages=messages, stream=True)
for stream_response in response:
    print('reasoning content: ',stream_response.choices[0].delta.reasoning_content)
    print('content: ', stream_response.choices[0].delta.content)

response = client.chat.completions.create(model=model, messages=messages, stream=False)
reasoning_content = response.choices[0].message.reasoning_content
content = response.choices[0].message.content

print("reasoning_content:", reasoning_content)
print("content:", content)
```

1.9.2 Custom parser

Built-in reasoning parser names include:

- `qwen-qwq`
- `qwen3`
- `intern-s1`
- `deepseek-r1`
- `deepseek-v3`
- `gpt-oss`

Notes

- `deepseek-v3`: starts in reasoning mode only when `enable_thinking=True`. When `enable_thinking` is `None` (default), output is usually plain content without a reasoning segment.
- `gpt-oss`: parses OpenAI Harmony channels:
  - `final` -> `content`
  - `analysis` -> `reasoning_content`
  - `commentary` with `functions.*` recipient -> `tool_calls`

Add a custom parser

Add a parser class under `lmdeploy/serve/openai/reasoning_parser/` and register it with `ReasoningParserManager`.

```
from lmdeploy.serve.openai.reasoning_parser import (
    ReasoningParser, ReasoningParserManager
)

@ReasoningParserManager.register_module(["example"])
class ExampleParser(ReasoningParser):
    def __init__(self, tokenizer: object, **kwargs):
        super().__init__(tokenizer, **kwargs)

    def get_reasoning_open_tag(self) -> str | None:
        return "<think>"

    def get_reasoning_close_tag(self) -> str | None:
        return "</think>"

    def starts_in_reasoning_mode(self) -> bool:
        return True
```

Then start the service with:

```
lmdeploy serve api_server $model_path --reasoning-parser example
```

1.10 Anthropic-Compatible Endpoints

LMDeploy provides a lightweight Anthropic-compatible surface for easier integration with Anthropic-style clients and gateways.

1.10.1 Supported Endpoints

- `POST /v1/messages`
- `POST /v1/messages/count_tokens`
- `GET /anthropic/v1/models`

1.10.2 Required Headers

For Anthropic `POST` endpoints, include:

- `content-type: application/json`
- `anthropic-version: 2023-06-01` (or another accepted version string)

1.10.3 Notes and Current Limits

- `POST /v1/messages` supports text output, reasoning output (`thinking` blocks), and tool-use output (`tool_use` blocks).
- Tool use requires launching API server with a configured tool parser (`--tool-call-parser ...`).
- Reasoning block extraction depends on parser configuration (same parser stack used by OpenAI-compatible chat endpoint).
- `count_tokens` is tokenizer/chat-template based and is intended for practical estimation.

1.10.4 Example: `/v1/messages`

```
curl http://{server_ip}:{server_port}/v1/messages \
  -H "content-type: application/json" \
  -H "anthropic-version: 2023-06-01" \
  -d '{
    "model": "internlm-chat-7b",
    "max_tokens": 128,
    "messages": [{"role": "user", "content": "Hello from Anthropic client"}]
  }'
```

1.10.5 Example: `/v1/messages` with tools

```
curl http://{server_ip}:{server_port}/v1/messages \
  -H "content-type: application/json" \
  -H "anthropic-version: 2023-06-01" \
  -d '{
    "model": "internlm-chat-7b",
    "max_tokens": 128,
    "messages": [{"role": "user", "content": "Find lmdeploy docs"}],
    "tools": [{
      "name": "search",
      "description": "Search docs",
      "input_schema": {
        "type": "object",
        "properties": {
          "query": {"type": "string"}
        },
        "required": ["query"]
      }
    }],
    "tool_choice": {"type": "auto"}
  }'
```

1.10.6 Streaming Events (SSE)

When `stream=true`, the endpoint returns `text/event-stream` events such as:

- `message_start`
- `content_block_start`
- `content_block_delta` (`text_delta`, `thinking_delta`, `input_json_delta`)
- `content_block_stop`
- `message_delta`
- `message_stop`

1.10.7 Example: `/v1/messages/count_tokens`

```
curl http://{server_ip}:{server_port}/v1/messages/count_tokens \
  -H "content-type: application/json" \
  -H "anthropic-version: 2023-06-01" \
  -d '{
    "model": "internlm-chat-7b",
    "system": "You are a helpful assistant.",
    "messages": [{"role": "user", "content": "Count these tokens"}]
  }'
```

1.11 Serving LoRA

1.11.1 Launch LoRA

LoRA is currently only supported by the PyTorch backend. Its deployment process is similar to that of other models, and you can view the commands using lmdeploy `serve api_server -h`. Among the parameters supported by the PyTorch backend, there are configuration options for LoRA.

```
PyTorch engine arguments:
  --adapters [ADAPTERS [ADAPTERS ...]]
                        Used to set path(s) of lora adapter(s). One can input key-value␣
↪pairs in xxx=yyy format for multiple lora adapters. If only have one adapter, one can␣
↪only input the path of the adapter.. Default:
                        None. Type: str
```

The user only needs to pass the Hugging Face model path of the LoRA weights in the form of a dictionary to `--adapters`.

```
lmdeploy serve api_server THUDM/chatglm2-6b --adapters mylora=chenchi/lora-chatglm2-6b-
↪guodegang
```

After the service starts, you can find two available model names in the Swagger UI: 'THUDM/chatglm2-6b' and 'mylora'. The latter is the key in the `--adapters` dictionary.

1.11.2 Client usage

CLI

When using the OpenAI endpoint, the `model` parameter can be used to select either the base model or a specific LoRA weight for inference. The following example chooses to use the provided `chenchi/lora-chatglm2-6b-guodegang` for inference.

```
curl -X 'POST' \
  'http://localhost:23334/v1/chat/completions' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "model": "mylora",
  "messages": [
    {
      "content": "hi",
      "role": "user"
    }
  ]
}'
```

And here is the output:


```
{
  "id": "2",
  "object": "chat.completion",
  "created": 1721377275,
  "model": "mylora",
  "choices": [
    {
      "index": 0,
      "message": {
        "role": "assistant",
        "content": " “”",
        "tool_calls": null
      },
      "logprobs": null,
      "finish_reason": "stop"
    }
  ],
  "usage": {
    "prompt_tokens": 17,
    "total_tokens": 43,
    "completion_tokens": 26
  }
}
```

python

```
from openai import OpenAI
client = OpenAI(
    api_key='YOUR_API_KEY',
    base_url="http://0.0.0.0:23333/v1"
)
model_name = 'mylora'
response = client.chat.completions.create(
  model=model_name,
  messages=[
    {"role": "user", "content": "hi"},
  ],
    temperature=0.8,
    top_p=0.8
)
print(response)
```

The printed response content is:

```
ChatCompletion(id='4', choices=[Choice(finish_reason='stop', index=0, logprobs=None,␣
↪message=ChatCompletionMessage(content=' ', role='assistant', function_call=None, tool_
↪calls=None))], created=1721377497, model='mylora', object='chat.completion', service_
↪tier=None, system_fingerprint=None, usage=CompletionUsage(completion_tokens=22, prompt_
↪tokens=17, total_tokens=39))
```

1.12 Request Distributor Server

The request distributor service can parallelize multiple api_server services. Users only need to access the proxy URL, and they can indirectly access different api_server services. The proxy service will automatically distribute requests internally, achieving load balancing.

1.12.1 Startup

Start the proxy service:

```
lmdeploy serve proxy --server-name {server_name} --server-port {server_port} --routing-
↪strategy "min_expected_latency" --serving-strategy Hybrid
```

After startup is successful, the URL of the proxy service will also be printed by the script. Access this URL in your browser to open the Swagger UI. Subsequently, users can add it directly to the proxy service when starting the `api_server` service by using the `--proxy-url` command. For example: `lmdeploy serve api_server InternLM/internlm2-chat-1_8b --proxy-url http://0.0.0.0:8000` In this way, users can access the services of the `api_server` through the proxy node, and the usage of the proxy node is exactly the same as that of the `api_server`, both of which are compatible with the OpenAI format.

- /v1/models
- /v1/chat/completions
- /v1/completions

1.12.2 Node Management

Through Swagger UI, we can see multiple APIs. Those related to api_server node management include:

- /nodes/status
- /nodes/add
- /nodes/remove

They respectively represent viewing all api_server service nodes, adding a certain node, and deleting a certain node.

Node Management through curl

```
curl -X 'GET' \
  'http://localhost:8000/nodes/status' \
  -H 'accept: application/json'
```

```
curl -X 'POST' \
  'http://localhost:8000/nodes/add' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "url": "http://0.0.0.0:23333"
}'
```

```
curl -X 'POST' \
  'http://localhost:8000/nodes/remove?node_url=http://0.0.0.0:23333' \
  -H 'accept: application/json' \
  -d ''
```

Node Management through python

```
# query all nodes
import requests
url = 'http://localhost:8000/nodes/status'
headers = {'accept': 'application/json'}
response = requests.get(url, headers=headers)
print(response.text)
```

```
# add a new node
import requests
url = 'http://localhost:8000/nodes/add'
headers = {
    'accept': 'application/json',
    'Content-Type': 'application/json'
}
data = {"url": "http://0.0.0.0:23333"}
response = requests.post(url, headers=headers, json=data)
print(response.text)
```

```
# delete a node
import requests
url = 'http://localhost:8000/nodes/remove'
headers = {'accept': 'application/json',}
params = {'node_url': 'http://0.0.0.0:23333',}
response = requests.post(url, headers=headers, data='', params=params)
print(response.text)
```

1.12.3 Serving Strategy

LMDeploy currently supports two serving strategies:

- Hybrid: Does not distinguish between Prefill and Decoding instances, following the traditional inference deployment mode.
- DistServe: Separates Prefill and Decoding instances, deploying them on different service nodes to achieve more flexible and efficient resource scheduling and scalability.

1.12.4 Dispatch Strategy

The current distribution strategies of the proxy service are as follows:

- random dispatches based on the ability of each api_server node provided by the user to process requests. The greater the request throughput, the more likely it is to be allocated. Nodes that do not provide throughput are treated according to the average throughput of other nodes.
- min_expected_latency allocates based on the number of requests currently waiting to be processed on each node, and the throughput capability of each node, calculating the expected time required to complete the response. The shortest one gets allocated. Nodes that do not provide throughput are treated similarly.
- min_observed_latency allocates based on the average time required to handle a certain number of past requests on each node. The one with the shortest time gets allocated.

1.13 Offline Inference Pipeline

LMDeploy abstracts the complex inference process of multi-modal Vision-Language Models (VLM) into an easy-to-use pipeline, similar to the Large Language Model (LLM) inference *pipeline*.

The supported models are listed *here*. We genuinely invite the community to contribute new VLM support to LMDeploy. Your involvement is truly appreciated.

This article showcases the VLM pipeline using the OpenGVLab/InternVL2_5-8B model as a case study. You'll learn about the simplest ways to leverage the pipeline and how to gradually unlock more advanced features by adjusting engine parameters and generation arguments, such as tensor parallelism, context window sizing, random sampling, and chat template customization. Moreover, we will provide practical inference examples tailored to scenarios with multiple images, batch prompts etc.

Using the pipeline interface to infer other VLM models is similar, with the main difference being the configuration and installation dependencies of the models. You can read here for environment installation and configuration methods for different models.

> **See also:** *Multi-Modal Inputs* — message format reference for all modalities (image, video, time series) with OpenAI-style examples.

1.13.1 A 'Hello, world' example

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

If `ImportError` occurs while executing this case, please install the required dependency packages as prompted.

In the above example, the inference prompt is a tuple structure consisting of (prompt, image). Besides this structure, the pipeline also supports prompts in the OpenAI format:

```
from lmdeploy import pipeline

pipe = pipeline('OpenGVLab/InternVL2_5-8B')

prompts = [
    {
        'role': 'user',
        'content': [
            {'type': 'text', 'text': 'describe this image'},
            {'type': 'image_url', 'image_url': {'url': 'https://raw.githubusercontent.
↪com/open-mmlab/mmdeploy/main/tests/data/tiger.jpeg'}}
        ]
    }
]
response = pipe(prompts)
print(response)
```

Set tensor parallelism

Tensor paramllelism can be activated by setting the engine parameter `tp`

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(tp=2))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Set context window size

When creating the pipeline, you can customize the size of the context window by setting the engine parameter `session_len`.

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(session_len=8192))

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Set sampling parameters

You can change the default sampling parameters of pipeline by passing `GenerationConfig`

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(tp=2, session_len=8192))
gen_config = GenerationConfig(top_k=40, top_p=0.8, temperature=0.6)
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image), gen_config=gen_config)
print(response)
```

Customize image token position

By default, LMDeploy inserts the special image token into the user prompt following the chat template defined by the upstream algorithm repository. However, for certain models where the image token's position is unrestricted, such as deepseek-vl, or when users require a customized image token placement, manual insertion of the special image token into the prompt is necessary. LMDeploy use `<IMAGE_TOKEN>` as the special image token.

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image
from lmdeploy.vl.constants import IMAGE_TOKEN

pipe = pipeline('deepseek-ai/deepseek-vl-1.3b-chat')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image{IMAGE_TOKEN}', image))
print(response)
```

Set chat template

While performing inference, LMDeploy identifies an appropriate chat template from its builtin collection based on the model path and subsequently applies this template to the input prompts. However, when a chat template cannot be told from its model path, users have to specify it. For example, liuhaotian/llava-v1.5-7b employs the 'llava-v1' chat template, if user have a custom folder name instead of the official 'llava-v1.5-7b', the user needs to specify it by setting 'llava-v1' to `ChatTemplateConfig` as follows:

```
from lmdeploy import pipeline, ChatTemplateConfig
from lmdeploy.vl import load_image
pipe = pipeline('local_model_folder',
                chat_template_config=ChatTemplateConfig(model_name='llava-v1'))
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

For more information about customizing a chat template, please refer to *this* guide

Setting vision model parameters

The default parameters of the visual model can be modified by setting `VisionConfig`.

```
from lmdeploy import pipeline, VisionConfig
from lmdeploy.vl import load_image
vision_config=VisionConfig(max_batch_size=16)
pipe = pipeline('liuhaotian/llava-v1.5-7b', vision_config=vision_config)
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Output logits for generated tokens

```
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl import load_image
pipe = pipeline('OpenGVLab/InternVL2_5-8B')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')

response = pipe(('describe this image', image),
                gen_config=GenerationConfig(output_logits='generation'))
logits = response.logits
print(logits)
```

1.13.2 Multi-images inference

When dealing with multiple images, you can put them all in one list. Keep in mind that multiple images will lead to a higher number of input tokens, and as a result, the size of the *context window* typically needs to be increased.

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(session_len=8192))

image_urls=[
    'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/human-
↪pose.jpg',
    'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/det.jpg'
]

images = [load_image(img_url) for img_url in image_urls]
response = pipe(('describe these images', images))
print(response)
```

1.13.3 Batch prompts inference

Conducting inference with batch prompts is quite straightforward; just place them within a list structure:

```
from lmdeploy import pipeline, TurbomindEngineConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(session_len=8192))

image_urls=[
    "https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/human-
↪pose.jpg",
    "https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/resources/det.jpg"
]
prompts = [('describe this image', load_image(img_url)) for img_url in image_urls]
response = pipe(prompts)
print(response)
```

1.13.4 Multi-turn conversation

There are two ways to do the multi-turn conversations with the pipeline. One is to construct messages according to the format of OpenAI and use above introduced method, the other is to use the `pipeline.chat` interface.

```
from lmdeploy import pipeline, TurbomindEngineConfig, GenerationConfig
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2_5-8B',
                backend_config=TurbomindEngineConfig(session_len=8192))
```

(continues on next page)

(continued from previous page)

```
image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/
↪resources/human-pose.jpg')
gen_config = GenerationConfig(top_k=40, top_p=0.8, temperature=0.8)
sess = pipe.chat(('describe this image', image), gen_config=gen_config)
print(sess.response.text)
sess = pipe.chat('What is the woman doing?', session=sess, gen_config=gen_config)
print(sess.response.text)
```

1.13.5 Release pipeline

You can release the pipeline explicitly by calling its `close()` method, or alternatively, use the `with` statement as demonstrated below:

```
from lmdeploy import pipeline

from lmdeploy import pipeline
from lmdeploy.vl import load_image

with pipeline('OpenGVLab/InternVL2_5-8B') as pipe:
    image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
    response = pipe(('describe this image', image))
    print(response)

# Clear the torch cache and perform garbage collection if needed
import torch
import gc
torch.cuda.empty_cache()
gc.collect()
```

1.14 OpenAI Compatible Server

This article primarily discusses the deployment of a single large vision language model across multiple GPUs on a single node, providing a service that is compatible with the OpenAI interface, as well as the usage of the service API. For the sake of convenience, we refer to this service as `api_server`. Regarding parallel services with multiple models, please refer to the guide about *Request Distribution Server*.

In the following sections, we will first introduce two methods for starting the service, choosing the appropriate one based on your application scenario.

Next, we focus on the definition of the service's RESTful API, explore the various ways to interact with the interface, and demonstrate how to try the service through the Swagger UI or LMDeploy CLI tools.

1.14.1 Launch Service

Take the llava-v1.6-vicuna-7b model hosted on huggingface hub as an example, you can choose one the following methods to start the service.

Option 1: Launching with lmdeploy CLI

```
lmdeploy serve api_server liuhaotian/llava-v1.6-vicuna-7b --server-port 23333
```

The arguments of `api_server` can be viewed through the command `lmdeploy serve api_server -h`, for instance, `--tp` to set tensor parallelism, `--session-len` to specify the max length of the context window, `--cache-max-entry-count` to adjust the GPU mem ratio for k/v cache etc.

Option 2: Deploying with docker

With LMDeploy official docker image, you can run OpenAI compatible server as follows:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server liuhaotian/llava-v1.6-vicuna-7b
```

The parameters of `api_server` are the same with that mentioned in "*option 1*" section

Each model may require specific dependencies not included in the Docker image. If you run into issues, you may need to install those yourself on a case-by-case basis. If in doubt, refer to the specific model's project for documentation.

For example, for Llava:

```
FROM openmmlab/lmdeploy:latest

RUN apt-get update && apt-get install -y python3 python3-pip git

WORKDIR /app

RUN pip3 install --upgrade pip
RUN pip3 install timm
RUN pip3 install git+https://github.com/haotian-liu/LLaVA.git --no-deps

COPY . .

CMD ["lmdeploy", "serve", "api_server", "liuhaotian/llava-v1.6-34b"]
```

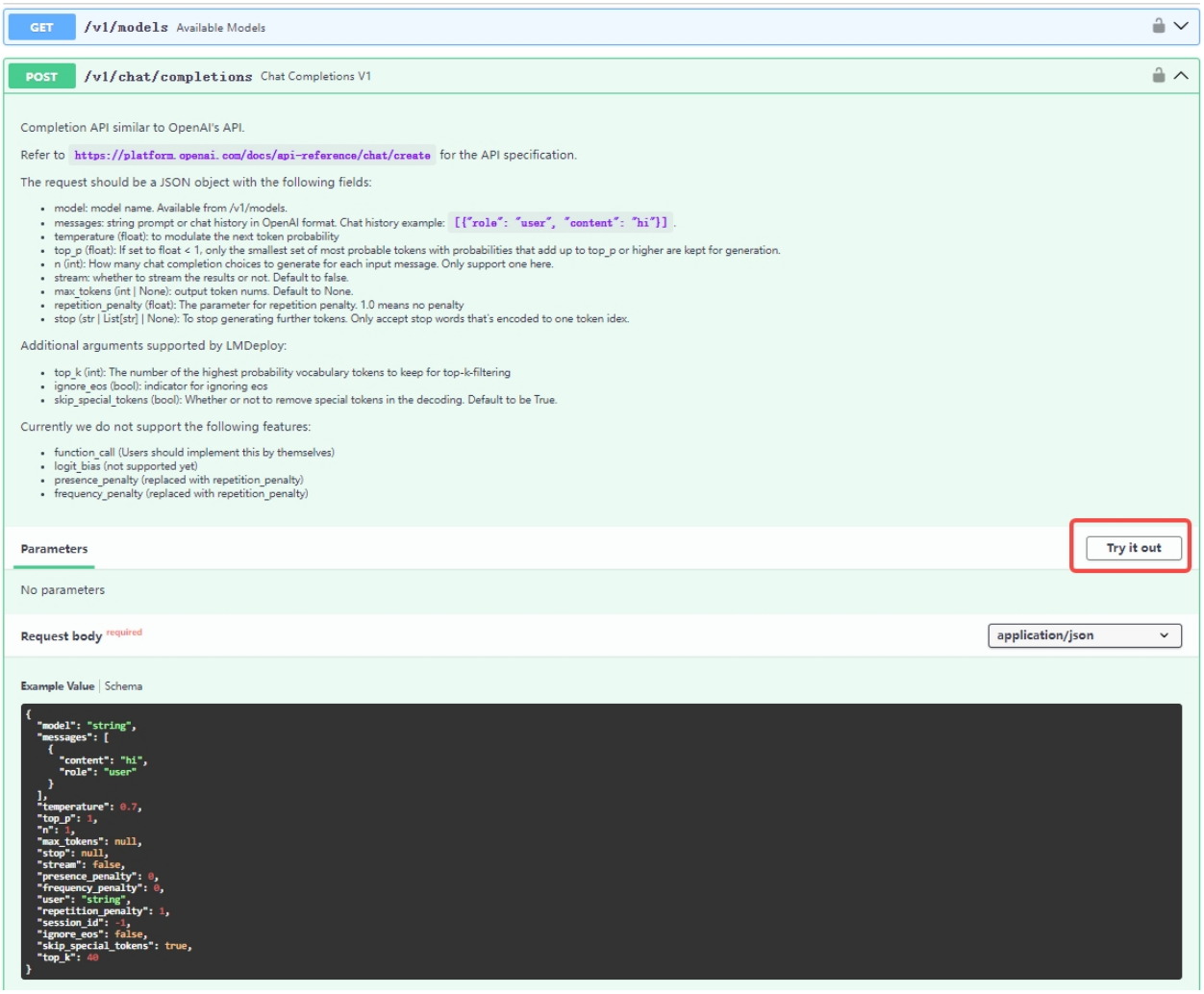
1.14.2 RESTful API

LMDeploy's RESTful API is compatible with the following three OpenAI interfaces:

- /v1/chat/completions
- /v1/models
- /v1/completions

The interface for image interaction is `/v1/chat/completions`, which is consistent with OpenAI.

You can overview and try out the offered RESTful APIs by the website `http://0.0.0.0:23333` as shown in the below image after launching the service successfully.



If you need to integrate the service into your own projects or products, we recommend the following approach:

Integrate with `OpenAI`

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```
from openai import OpenAI

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role':
        'user',
        'content': [{
            'type': 'text',
            'text': 'Describe the image please',
        }, {
            'type': 'image_url',
            'image_url': {
                'url':
                'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/
↪tiger.jpeg',
            },
        }],
    }],
    temperature=0.8,
    top_p=0.8)
print(response)
```

Integrate with lmdeploy `APIClient`

Below are some examples demonstrating how to visit the service through `APIClient`

If you want to use the `/v1/chat/completions` endpoint, you can try the following code:

```
from lmdeploy.serve.openai.api_client import APIClient

api_client = APIClient(f'http://0.0.0.0:23333')
model_name = api_client.available_models[0]
messages = [{
    'role':
    'user',
    'content': [{
        'type': 'text',
        'text': 'Describe the image please',
    }, {
        'type': 'image_url',
        'image_url': {
            'url':
            'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/tiger.
↪jpeg',
        },
```

(continues on next page)

(continued from previous page)

```
    }]
}]
for item in api_client.chat_completions_v1(model=model_name,
                                           messages=messages):
    print(item)
```

Integrate with Java/Golang/Rust

May use openapi-generator-cli to convert `http://{server_ip}:{server_port}/openapi.json` to java/rust/golang client. Here is an example:

```
$ docker run -it --rm -v ${PWD}:/local openapitools/openapi-generator-cli generate -i /
↪local/openapi.json -g rust -o /local/rust

$ ls rust/*
rust/Cargo.toml  rust/git_push.sh  rust/README.md

rust/docs:
ChatCompletionRequest.md  EmbeddingsRequest.md  HttpValidationError.md  LocationInner.md␣
↪ Prompt.md
DefaultApi.md             GenerateRequest.md    Input.md                Messages.md     ␣
↪ ValidationError.md

rust/src:
apis  lib.rs  models
```

1.15 Vision-Language Models

1.15.1 Multi-Modal Inputs

LMDeploy uses the OpenAI message format for all modalities. Each content item in a message is a dict with a `type` field that determines how it is decoded.

Quick reference:

Modality	`type` key	URL field
Text	`text`	—
Image	`image_url`	`image_url.url`
Video	`video_url`	`video_url.url`
Time Series	`time_series_url`	`time_series_url.url`

All examples below target the lmdeploy OpenAI-compatible API server. Start it with:

```
lmdeploy serve api_server <model_path> --server-port 23333
```

Text

```
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [{
            'type': 'text',
            'text': 'Who are you?',
        }],
    }],
    temperature=0.8,
    top_p=0.8,
)
print(response.choices[0].message.content)
```

Single Image

```
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'image_url',
                'image_url': {
                    'url': 'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/
↪tests/data/tiger.jpeg',
                },
            },
            {
                'type': 'text',
                'text': 'Describe this image.',
            },
        ],
    }],
    temperature=0.8,
    top_p=0.8,
)
print(response.choices[0].message.content)
```

Multiple Images

```
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'image_url',
                'image_url': {
                    'url': 'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/
↪tests/data/tiger.jpeg',
                },
            },
            {
                'type': 'image_url',
                'image_url': {
                    'url': 'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/
↪tests/data/tiger.jpeg',
                },
            },
            {
                'type': 'text',
                'text': 'Compare these two images. What are the similarities and␣
↪differences?',
            },
        ],
    }],
    temperature=0.8,
    top_p=0.8,
)
print(response.choices[0].message.content)
```

Single Video

Note: Native video input is currently supported for **Qwen3-VL**, **Qwen3.5**, and **InternS1-Pro** models only.

```
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id
```

(continues on next page)

(continued from previous page)

```
video_url = 'https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen2-VL/space_woaudio.mp4'

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'video_url',
                'video_url': {
                    'url': video_url,
                },
            },
            {
                'type': 'text',
                'text': "What's in this video?",
            },
        ],
    }],
    temperature=0.8,
    top_p=0.8,
    max_completion_tokens=256,
)
print(response.choices[0].message.content)
```

Multiple Videos

Note: Native video input is currently supported for **Qwen3-VL**, **Qwen3.5**, and **InternS1-Pro** models only.

```
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

video_url_1 = 'https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen2-VL/space_woaudio.mp4
↪'
video_url_2 = 'https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen2-VL/space_woaudio.mp4
↪'

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'video_url',
                'video_url': {'url': video_url_1},
            },
            {
```

(continues on next page)

(continued from previous page)

```
                'type': 'video_url',
                'video_url': {'url': video_url_2},
            },
            {
                'type': 'text',
                'text': 'Compare these two videos. What are the similarities and␣
↪differences?',
            },
        ],
    }],
    temperature=0.8,
    top_p=0.8,
    max_completion_tokens=256,
)
print(response.choices[0].message.content)
```

Mixed Image and Video

Note: Native video input is currently supported for **Qwen3-VL**, **Qwen3.5**, and **InternS1-Pro** models only.

```
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

image_url = 'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/tiger.
↪jpeg'
video_url = 'https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen2-VL/space_woaudio.mp4'

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'image_url',
                'image_url': {'url': image_url},
            },
            {
                'type': 'video_url',
                'video_url': {'url': video_url},
            },
            {
                'type': 'text',
                'text': 'Describe both the image and the video.',
            },
        ],
    }],
    temperature=0.8,
```

(continues on next page)

(continued from previous page)

```
    top_p=0.8,
    max_completion_tokens=256,
)
print(response.choices[0].message.content)
```

Time Series

Note: Time series input is currently supported for the **InternS1-Pro** model only.

The `time_series_url` content item requires a `sampling_rate` field (in Hz) alongside the URL.

```
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'text',
                'text': ('Please determine whether an Earthquake event has occurred. '
                         'If so, specify P-wave and S-wave starting indices.'),
            },
            {
                'type': 'time_series_url',
                'time_series_url': {
                    'url': 'https://raw.githubusercontent.com/CUHKSZzxy/Online-Data/main/
↪0092638_seism.npy',
                    'sampling_rate': 100,
                },
            },
        ],
    }],
    temperature=0.8,
    top_p=0.8,
    max_completion_tokens=256,
)
print(response.choices[0].message.content)
```

Local Files and Base64

In addition to HTTP URLs, lmdeploy accepts:

- **Local file paths** via `file://` scheme: `file:///absolute/path/to/file.jpg`
- **Base64-encoded data** via data URLs: `data:<mime>;base64,<encoded_data>`

Use the helpers in `lmdeploy.vl.utils` to encode local files:

```
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'image_url',
                'image_url': {
                    'url': 'file:///path/to/your/image.jpg',
                },
            },
            {'type': 'text', 'text': 'Describe this image.'},
        ],
    }],
)
print(response.choices[0].message.content)
```

```
from openai import OpenAI
from lmdeploy.vl.utils import encode_image_base64

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

b64 = encode_image_base64('/path/to/your/image.jpg')
image_url = f'{b64}'

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'image_url',
                'image_url': {'url': image_url},
            },
            {'type': 'text', 'text': 'Describe this image.'},
        ],
    }],
)
print(response.choices[0].message.content)
```

```python
from openai import OpenAI
from lmdeploy.vl.utils import encode_video_base64

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

# num_frames controls how many frames to sample before encoding
b64 = encode_video_base64('/path/to/your/video.mp4', num_frames=16)
video_url = f'data:video/mp4;base64,{b64}'

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {
                'type': 'video_url',
                'video_url': {'url': video_url},
            },
            {'type': 'text', 'text': 'Describe this video.'},
        ],
    }],
)
print(response.choices[0].message.content)
```

```python
from openai import OpenAI
from lmdeploy.vl.utils import encode_time_series_base64

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

b64 = encode_time_series_base64('/path/to/your/data.npy')
ts_url = f'data:application/octet-stream;base64,{b64}'

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {'type': 'text', 'text': 'Analyze this time series.'},
            {
                'type': 'time_series_url',
                'time_series_url': {
                    'url': ts_url,
                    'sampling_rate': 100,
                },
            },
        ],
    }],
)
print(response.choices[0].message.content)
```

Processor and IO kwargs

Two optional parameters let you control media processing:

- **mm_processor_kwargs**: controls vision token resolution (min/max pixels per image or video frame)
- **media_io_kwargs**: controls how media is loaded (e.g. video frame sampling rate and count)

Both are passed as extra fields in the API request body via `extra_body`, or directly to `pipe()` when using the pipeline API.

```python
from openai import OpenAI

client = OpenAI(api_key='EMPTY', base_url='http://localhost:23333/v1')
model_name = client.models.list().data[0].id

video_url = 'https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen2-VL/space_woaudio.mp4'

response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role': 'user',
        'content': [
            {'type': 'video_url', 'video_url': {'url': video_url}},
            {'type': 'text', 'text': 'Describe this video.'},
        ],
    }],
    max_completion_tokens=256,
    extra_body={
        'mm_processor_kwargs': {
            'video': {
                'min_pixels': 4 * 32 * 32,
                'max_pixels': 256 * 32 * 32,
            },
        },
        'media_io_kwargs': {
            'video': {
                'num_frames': 16,
                'fps': 2,
            },
        },
    },
)
print(response.choices[0].message.content)
```

```python
from lmdeploy import pipeline, PytorchEngineConfig

pipe = pipeline('<model_path>', backend_config=PytorchEngineConfig(tp=1))

video_url = 'https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen2-VL/space_woaudio.mp4'

messages = [{
    'role': 'user',
    'content': [
        {'type': 'video_url', 'video_url': {'url': video_url}},
```

(continues on next page)

(continued from previous page)

```
        {'type': 'text', 'text': 'Describe this video.'},
    ],
}]

response = pipe(
    messages,
    mm_processor_kwargs={
        'video': {
            'min_pixels': 4 * 32 * 32,
            'max_pixels': 256 * 32 * 32,
        },
    },
    media_io_kwargs={
        'video': {
            'num_frames': 16,
            'fps': 2,
        },
    },
)
print(response)
```

1.15.2 DeepSeek-VL2

Introduction

DeepSeek-VL2, an advanced series of large Mixture-of-Experts (MoE) Vision-Language Models that significantly improves upon its predecessor, DeepSeek-VL. DeepSeek-VL2 demonstrates superior capabilities across various tasks, including but not limited to visual question answering, optical character recognition, document/table/chart understanding, and visual grounding.

LMDeploy supports deepseek-vl2-tiny, deepseek-vl2-small and deepseek-vl2 in PyTorch engine.

Quick Start

Install LMDeploy by following the *installation guide*.

Prepare

When deploying the **DeepSeek-VL2** model using LMDeploy, you must install the official GitHub repository and related 3-rd party libs. This is because LMDeploy reuses the image processing functions provided in the official repository.

```
pip install git+https://github.com/deepseek-ai/DeepSeek-VL2.git --no-deps
pip install attrdict timm 'transformers<4.48.0'
```

Worth noticing that it may fail with `transformers>=4.48.0`, as known in this issue.

Offline inference pipeline

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*.

To construct valid DeepSeek-VL2 prompts with image inputs, users should insert <IMAGE_TOKEN> manually.

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image


if __name__ == "__main__":
    pipe = pipeline('deepseek-ai/deepseek-vl2-tiny')

    image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
    response = pipe(('<IMAGE_TOKEN>describe this image', image))
    print(response)
```

1.15.3 LLaVA

LMDeploy supports the following llava series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
llava-hf/Llava-interleave-qwen-7b-hf	7B	TurboMind, PyTorch
llava-hf/llava-1.5-7b-hf	7B	TurboMind, PyTorch
llava-hf/llava-v1.6-mistral-7b-hf	7B	PyTorch
llava-hf/llava-v1.6-vicuna-7b-hf	7B	PyTorch
liuhaotian/llava-v1.6-mistral-7b	7B	TurboMind
liuhaotian/llava-v1.6-vicuna-7b	7B	TurboMind

The next chapter demonstrates how to deploy an Llava model using LMDeploy, with llava-hf/llava-interleave as an example.

Note

PyTorch engine removes the support of original llava models after v0.6.4. Please use their corresponding transformers models instead, which can be found in https://huggingface.co/llava-hf

Installation

Please install LMDeploy by following the *installation guide*.

Or, you can go with office docker image:

```
docker pull openmmlab/lmdeploy:latest
```

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import GenerationConfig, TurbomindEngineConfig, pipeline
from lmdeploy.vl import load_image


pipe = pipeline("llava-hf/llava-interleave-qwen-7b-hf", backend_
↪config=TurbomindEngineConfig(cache_max_entry_count=0.5),
    gen_config=GenerationConfig(max_new_tokens=512))

image = load_image('https://qianwen-res.oss-cn-beijing.aliyuncs.com/Qwen-VL/assets/demo.
↪jpeg')
prompt = 'Describe the image.'
print(f'prompt:{prompt}')
response = pipe((prompt, image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('llava-hf/llava-interleave-qwen-7b-hf', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server llava-hf/llava-interleave-qwen-7b-hf
```

You can also start the service using the aforementioned built docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
```

(continues on next page)

(continued from previous page)

```
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server llava-hf/llava-interleave-qwen-7b-hf
```

The docker compose is another option. Create a `docker-compose.yml` configuration file in the root directory of the lmdeploy project as follows:

```
version: '3.5'

services:
  lmdeploy:
    container_name: lmdeploy
    image: openmmlab/lmdeploy:latest
    ports:
      - "23333:23333"
    environment:
      HUGGING_FACE_HUB_TOKEN: <secret>
    volumes:
      - ~/.cache/huggingface:/root/.cache/huggingface
    stdin_open: true
    tty: true
    ipc: host
    command: lmdeploy serve api_server llava-hf/llava-interleave-qwen-7b-hf
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: "all"
              capabilities: [gpu]
```

Then, you can execute the startup command as below:

```
docker-compose up -d
```

If you find the following logs after running `docker logs -f lmdeploy`, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.15.4 InternVL

LMDeploy supports the following InternVL series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
InternVL	13B-19B	TurboMind
InternVL1.5	2B-26B	TurboMind, PyTorch
InternVL2	4B	PyTorch
InternVL2	1B-2B, 8B-76B	TurboMind, PyTorch
InternVL2.5/2.5-MPO/3	1B-78B	TurboMind, PyTorch
Mono-InternVL	2B	PyTorch

The next chapter demonstrates how to deploy an InternVL model using LMDeploy, with InternVL2-8B as an example.

Installation

Please install LMDeploy by following the *installation guide*, and install other packages that InternVL2 needs

```
pip install timm
# It is recommended to find the whl package that matches the environment from the␣
↪releases on https://github.com/Dao-AILab/flash-attention.
pip install flash-attn
```

Or, you can build a docker image to set up the inference environment. If the CUDA version on your host machine is >=12.4, you can run:

```
docker build --build-arg CUDA_VERSION=cu12 -t openmmlab/lmdeploy:internvl . -f ./docker/
↪InternVL_Dockerfile
```

Otherwise, you can go with:

```
git clone https://github.com/InternLM/lmdeploy.git
cd lmdeploy
docker build --build-arg CUDA_VERSION=cu11 -t openmmlab/lmdeploy:internvl . -f ./docker/
↪InternVL_Dockerfile
```

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('OpenGVLab/InternVL2-8B')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl.constants import IMAGE_TOKEN

pipe = pipeline('OpenGVLab/InternVL2-8B', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text=f'{IMAGE_TOKEN}{IMAGE_TOKEN}\nDescribe the two images in␣
↪detail.'),
        dict(type='image_url', image_url=dict(max_dynamic_patch=12, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image1.jpg')),
        dict(type='image_url', image_url=dict(max_dynamic_patch=12, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image2.jpg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

```
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl.constants import IMAGE_TOKEN

pipe = pipeline('OpenGVLab/InternVL2-8B', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text=f'Image-1: {IMAGE_TOKEN}\nImage-2: {IMAGE_TOKEN}\
↪nDescribe the two images in detail.'),
        dict(type='image_url', image_url=dict(max_dynamic_patch=12, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image1.jpg')),
        dict(type='image_url', image_url=dict(max_dynamic_patch=12, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image2.jpg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

```
import numpy as np
from lmdeploy import pipeline, GenerationConfig
from decord import VideoReader, cpu
from lmdeploy.vl.constants import IMAGE_TOKEN
from lmdeploy.vl import encode_image_base64
from PIL import Image
pipe = pipeline('OpenGVLab/InternVL2-8B', log_level='INFO')


def get_index(bound, fps, max_frame, first_idx=0, num_segments=32):
```

(continues on next page)

(continued from previous page)

```
    if bound:
        start, end = bound[0], bound[1]
    else:
        start, end = -100000, 100000
    start_idx = max(first_idx, round(start * fps))
    end_idx = min(round(end * fps), max_frame)
    seg_size = float(end_idx - start_idx) / num_segments
    frame_indices = np.array([
        int(start_idx + (seg_size / 2) + np.round(seg_size * idx))
        for idx in range(num_segments)
    ])
    return frame_indices


def load_video(video_path, bound=None, num_segments=32):
    vr = VideoReader(video_path, ctx=cpu(0), num_threads=1)
    max_frame = len(vr) - 1
    fps = float(vr.get_avg_fps())
    pixel_values_list, num_patches_list = [], []
    frame_indices = get_index(bound, fps, max_frame, first_idx=0, num_segments=num_
↪segments)
    imgs = []
    for frame_index in frame_indices:
        img = Image.fromarray(vr[frame_index].asnumpy()).convert('RGB')
        imgs.append(img)
    return imgs


video_path = 'red-panda.mp4'
imgs = load_video(video_path, num_segments=8)

question = ''
for i in range(len(imgs)):
    question = question + f'Frame{i+1}: {IMAGE_TOKEN}\n'

question += 'What is the red panda doing?'

content = [{'type': 'text', 'text': question}]
for img in imgs:
    content.append({'type': 'image_url', 'image_url': {'max_dynamic_patch': 1, 'url': f
↪'{encode_image_base64(img)}'}})

messages = [dict(role='user', content=content)]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='Describe this video in detail. Don\'t repeat.
↪'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server OpenGVLab/InternVL2-8B
```

You can also start the service using the aforementioned built docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:internvl \
    lmdeploy serve api_server OpenGVLab/InternVL2-8B
```

The docker compose is another option. Create a `docker-compose.yml` configuration file in the root directory of the lmdeploy project as follows:

```
version: '3.5'

services:
  lmdeploy:
    container_name: lmdeploy
    image: openmmlab/lmdeploy:internvl
    ports:
      - "23333:23333"
    environment:
      HUGGING_FACE_HUB_TOKEN: <secret>
    volumes:
      - ~/.cache/huggingface:/root/.cache/huggingface
    stdin_open: true
    tty: true
    ipc: host
    command: lmdeploy serve api_server OpenGVLab/InternVL2-8B
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: "all"
              capabilities: [gpu]
```

Then, you can execute the startup command as below:

```
docker-compose up -d
```

If you find the following logs after running `docker logs -f lmdeploy`, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
```

(continues on next page)

(continued from previous page)

```
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.15.5 InternLM-XComposer-2.5

Introduction

InternLM-XComposer-2.5 excels in various text-image comprehension and composition applications, achieving GPT-4V level capabilities with merely 7B LLM backend. IXC-2.5 is trained with 24K interleaved image-text contexts, it can seamlessly extend to 96K long contexts via RoPE extrapolation. This long-context capability allows IXC-2.5 to perform exceptionally well in tasks requiring extensive input and output contexts. LMDeploy supports model internlm/internlm-xcomposer2d5-7b in TurboMind engine.

Quick Start

Installation

Please install LMDeploy by following the *installation guide*, and install other packages that InternLM-XComposer-2.5 needs

```
pip install decord
```

Offline inference pipeline

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image
from lmdeploy.vl.constants import IMAGE_TOKEN

pipe = pipeline('internlm/internlm-xcomposer2d5-7b')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

Lora Model

InternLM-XComposer-2.5 trained the LoRA weights for webpage creation and article writing. As TurboMind backend doesn't support slora, only one LoRA model can be deployed at a time, and the LoRA weights need to be merged when deploying the model. LMDeploy provides the corresponding conversion script, which is used as follows:

```
export HF_MODEL=internlm/internlm-xcomposer2d5-7b
export WORK_DIR=internlm/internlm-xcomposer2d5-7b-web
export TASK=web
python -m lmdeploy.vl.tools.merge_xcomposer2d5_task $HF_MODEL $WORK_DIR --task $TASK
```

Quantization

The following takes the base model as an example to show the quantization method. If you want to use the LoRA model, please merge the LoRA model according to the previous section.

```
export HF_MODEL=internlm/internlm-xcomposer2d5-7b
export WORK_DIR=internlm/internlm-xcomposer2d5-7b-4bit

lmdeploy lite auto_awq \
   $HF_MODEL \
  --work-dir $WORK_DIR
```

More examples

The following uses the `pipeline.chat` interface api as an example to demonstrate its usage. Other interfaces apis also support inference but require manually splicing of conversation content.

```
from lmdeploy import pipeline, GenerationConfig
from transformers.dynamic_module_utils import get_class_from_dynamic_module

HF_MODEL = 'internlm/internlm-xcomposer2d5-7b'
load_video = get_class_from_dynamic_module('ixc_utils.load_video', HF_MODEL)
frame2img = get_class_from_dynamic_module('ixc_utils.frame2img', HF_MODEL)
Video_transform = get_class_from_dynamic_module('ixc_utils.Video_transform', HF_MODEL)
get_font = get_class_from_dynamic_module('ixc_utils.get_font', HF_MODEL)

video = load_video('liuxiang.mp4') # https://github.com/InternLM/InternLM-XComposer/raw/
↪main/examples/liuxiang.mp4
img = frame2img(video, get_font())
img = Video_transform(img)

pipe = pipeline(HF_MODEL)
gen_config = GenerationConfig(top_k=50, top_p=0.8, temperature=1.0)
query = 'Here are some frames of a video. Describe this video in detail'
sess = pipe.chat((query, img), gen_config=gen_config)
print(sess.response.text)

query = 'tell me the athlete code of Liu Xiang'
sess = pipe.chat(query, session=sess, gen_config=gen_config)
print(sess.response.text)
```

```
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl.constants import IMAGE_TOKEN
from lmdeploy.vl import load_image

query = f'Image1 {IMAGE_TOKEN}; Image2 {IMAGE_TOKEN}; Image3 {IMAGE_TOKEN}; I want to␣
↪buy a car from the three given cars, analyze their advantages and weaknesses one by one
↪'

urls = ['https://raw.githubusercontent.com/InternLM/InternLM-XComposer/main/examples/
↪cars1.jpg',
        'https://raw.githubusercontent.com/InternLM/InternLM-XComposer/main/examples/
↪cars2.jpg',
        'https://raw.githubusercontent.com/InternLM/InternLM-XComposer/main/examples/
↪cars3.jpg']
images = [load_image(url) for url in urls]

pipe = pipeline('internlm/internlm-xcomposer2d5-7b', log_level='INFO')
output = pipe((query, images), gen_config=GenerationConfig(top_k=0, top_p=0.8, random_
↪seed=89247526689433939))
```

Since LMDeploy does not support beam search, the generated results will be quite different from those using beam search with transformers. It is recommended to turn off top_k or use a larger top_k sampling to increase diversity.

Please first convert the web model using the instructions above.

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('/nvme/shared/internlm-xcomposer2d5-7b-web', log_level='INFO')
pipe.chat_template.meta_instruction = None

query = 'A website for Research institutions. The name is Shanghai AI lab. Top␣
↪Navigation Bar is blue.Below left, an image shows the logo of the lab. In the right,␣
↪there is a passage of text below that describes the mission of the laboratory.There␣
↪are several images to show the research projects of Shanghai AI lab.'
output = pipe(query, gen_config=GenerationConfig(max_new_tokens=2048))
```

When using transformers for testing, it is found that if repetition_penalty is set, there is a high probability that the decode phase will not stop if `num_beams` is set to 1. As LMDeploy does not support beam search, it is recommended to turn off repetition_penalty when using LMDeploy for inference.

Please first convert the write model using the instructions above.

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('/nvme/shared/internlm-xcomposer2d5-7b-write', log_level='INFO')
pipe.chat_template.meta_instruction = None

query = 'Please write a blog based on the title: French Pastries: A Sweet Indulgence'
output = pipe(query, gen_config=GenerationConfig(max_new_tokens=8192))
```

1.15.6 CogVLM

Introduction

CogVLM is a powerful open-source visual language model (VLM). LMDeploy supports CogVLM-17B models like THUDM/cogvlm-chat-hf and CogVLM2-19B models like THUDM/cogvlm2-llama3-chat-19B in PyTorch engine.

Quick Start

Install LMDeploy by following the *installation guide*

Prepare

When deploying the **CogVLM** model using LMDeploy, it is necessary to download the model first, as the **CogVLM** model repository does not include the tokenizer model. However, this step is not required for **CogVLM2**.

Taking one **CogVLM** model `cogvlm-chat-hf` as an example, you can prepare it as follows:

```
huggingface-cli download THUDM/cogvlm-chat-hf --local-dir ./cogvlm-chat-hf --local-dir-
↪use-symlinks False
huggingface-cli download lmsys/vicuna-7b-v1.5 special_tokens_map.json tokenizer.model␣
↪tokenizer_config.json --local-dir ./cogvlm-chat-hf --local-dir-use-symlinks False
```

Offline inference pipeline

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image


if __name__ == "__main__":
    pipe = pipeline('cogvlm-chat-hf')

    image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
    response = pipe(('describe this image', image))
    print(response)
```

1.15.7 MiniCPM-V

LMDeploy supports the following MiniCPM-V series of models, which are detailed in the table below:

Model	Supported Inference Engine
MiniCPM-Llama3-V-2_5	TurboMind
MiniCPM-V-2_6	TurboMind

The next chapter demonstrates how to deploy an MiniCPM-V model using LMDeploy, with MiniCPM-V-2_6 as an example.

Installation

Please install LMDeploy by following the *installation guide*.

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('openbmb/MiniCPM-V-2_6')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('openbmb/MiniCPM-V-2_6', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(max_slice_nums=9, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image1.jpg')),
        dict(type='image_url', image_url=dict(max_slice_nums=9, url='https://raw.
↪githubusercontent.com/OpenGVLab/InternVL/main/internvl_chat/examples/image2.jpg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
print(out.text)

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
print(out.text)
```

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('openbmb/MiniCPM-V-2_6', log_level='INFO')

question = "production date"
messages = [
    dict(role='user', content=[
```

(continues on next page)

(continued from previous page)

```
        dict(type='text', text=question),
        dict(type='image_url', image_url=dict(url='example1.jpg')),
    ]),
    dict(role='assistant', content='2023.08.04'),
    dict(role='user', content=[
        dict(type='text', text=question),
        dict(type='image_url', image_url=dict(url='example2.jpg')),
    ]),
    dict(role='assistant', content='2007.04.24'),
    dict(role='user', content=[
        dict(type='text', text=question),
        dict(type='image_url', image_url=dict(url='test.jpg')),
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
print(out.text)
```

```
from lmdeploy import pipeline, GenerationConfig
from lmdeploy.vl import encode_image_base64
import torch
from PIL import Image
from transformers import AutoModel, AutoTokenizer
from decord import VideoReader, cpu    # pip install decord

pipe = pipeline('openbmb/MiniCPM-V-2_6', log_level='INFO')

MAX_NUM_FRAMES=64 # if cuda OOM set a smaller number
def encode_video(video_path):
    def uniform_sample(l, n):
        gap = len(l) / n
        idxs = [int(i * gap + gap / 2) for i in range(n)]
        return [l[i] for i in idxs]
    vr = VideoReader(video_path, ctx=cpu(0))
    sample_fps = round(vr.get_avg_fps() / 1)  # FPS
    frame_idx = [i for i in range(0, len(vr), sample_fps)]
    if len(frame_idx) > MAX_NUM_FRAMES:
        frame_idx = uniform_sample(frame_idx, MAX_NUM_FRAMES)
    frames = vr.get_batch(frame_idx).asnumpy()
    frames = [Image.fromarray(v.astype('uint8')) for v in frames]
    print('num frames:', len(frames))
    return frames

video_path="video_test.mp4"
frames = encode_video(video_path)
question = "Describe the video"

content=[dict(type='text', text=question)]
for frame in frames:
    content.append(dict(type='image_url', image_url=dict(use_image_id=False, max_slice_
↪nums=2,
        url=f'{encode_image_base64(frame)}')))
```

(continues on next page)

(continued from previous page)

```
messages = [dict(role='user', content=content)]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
print(out.text)
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server openbmb/MiniCPM-V-2_6
```

You can also start the service using the official lmdeploy docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server openbmb/MiniCPM-V-2_6
```

The docker compose is another option. Create a `docker-compose.yml` configuration file in the root directory of the lmdeploy project as follows:

```
version: '3.5'

services:
  lmdeploy:
    container_name: lmdeploy
    image: openmmlab/lmdeploy:latest
    ports:
      - "23333:23333"
    environment:
      HUGGING_FACE_HUB_TOKEN: <secret>
    volumes:
      - ~/.cache/huggingface:/root/.cache/huggingface
    stdin_open: true
    tty: true
    ipc: host
    command: lmdeploy serve api_server openbmb/MiniCPM-V-2_6
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: "all"
              capabilities: [gpu]
```

Then, you can execute the startup command as below:

```
docker-compose up -d
```

If you find the following logs after running `docker logs -f lmdeploy`, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.15.8 Phi-3 Vision

Introduction

Phi-3 is a family of small language and multi-modal models from MicroSoft. LMDeploy supports the multi-modal models as below.

Model	Size	Supported Inference Engine
microsoft/Phi-3-vision-128k-instruct	4.2B	PyTorch
microsoft/Phi-3.5-vision-instruct	4.2B	PyTorch

The next chapter demonstrates how to deploy an Phi-3 model using LMDeploy, with microsoft/Phi-3.5-vision-instruct as an example.

Installation

Please install LMDeploy by following the *installation guide* and install the dependency Flash-Attention

```
# It is recommended to find the whl package that matches the environment from the␣
↪releases on https://github.com/Dao-AILab/flash-attention.
pip install flash-attn
```

Offline inference

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('microsoft/Phi-3.5-vision-instruct')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe(('describe this image', image))
print(response)
```

Online serving

Launch Service

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server microsoft/Phi-3.5-vision-instruct
```

Integrate with `OpenAI`

Here is an example of interaction with the endpoint `v1/chat/completions` service via the openai package. Before running it, please install the openai package by `pip install openai`

```
from openai import OpenAI

client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=[{
        'role':
        'user',
        'content': [{
            'type': 'text',
            'text': 'Describe the image please',
        }, {
            'type': 'image_url',
            'image_url': {
                'url':
                'https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/data/
↪tiger.jpeg',
            },
        }],
    }],
    temperature=0.8,
    top_p=0.8)
print(response)
```

1.15.9 Qwen2-VL

LMDeploy supports the following Qwen-VL series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
Qwen-VL-Chat	-	TurboMind
Qwen2-VL	2B, 7B	PyTorch

The next chapter demonstrates how to deploy an Qwen-VL model using LMDeploy, with Qwen2-VL-7B-Instruct as an example.

Installation

Please install LMDeploy by following the *installation guide*, and install other packages that Qwen2-VL needs

```
pip install qwen_vl_utils
```

Or, you can build a docker image to set up the inference environment. If the CUDA version on your host machine is >=12.4, you can run:

```
git clone https://github.com/InternLM/lmdeploy.git
cd lmdeploy
docker build --build-arg CUDA_VERSION=cu12 -t openmmlab/lmdeploy:qwen2vl . -f ./docker/
↪Qwen2VL_Dockerfile
```

Otherwise, you can go with:

```
docker build --build-arg CUDA_VERSION=cu11 -t openmmlab/lmdeploy:qwen2vl . -f ./docker/
↪Qwen2VL_Dockerfile
```

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('Qwen/Qwen2-VL-2B-Instruct')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('Qwen/Qwen2-VL-2B-Instruct', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
```

(continues on next page)

(continued from previous page)

```
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('Qwen/Qwen2-VL-2B-Instruct', log_level='INFO')

min_pixels = 64 * 28 * 28
max_pixels = 64 * 28 * 28
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(min_pixels=min_pixels, max_pixels=max_
↪pixels, url='https://raw.githubusercontent.com/QwenLM/Qwen-VL/master/assets/mm_
↪tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(min_pixels=min_pixels, max_pixels=max_
↪pixels, url='https://raw.githubusercontent.com/QwenLM/Qwen-VL/master/assets/mm_
↪tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server Qwen/Qwen2-VL-2B-Instruct
```

You can also start the service using the aforementioned built docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
    --ipc=host \
    openmmlab/lmdeploy:qwen2vl \
    lmdeploy serve api_server Qwen/Qwen2-VL-2B-Instruct
```

The docker compose is another option. Create a `docker-compose.yml` configuration file in the root directory of the lmdeploy project as follows:

```
version: '3.5'

services:
  lmdeploy:
    container_name: lmdeploy
```

(continues on next page)

(continued from previous page)

```
    image: openmmlab/lmdeploy:qwen2vl
    ports:
      - "23333:23333"
    environment:
      HUGGING_FACE_HUB_TOKEN: <secret>
    volumes:
      - ~/.cache/huggingface:/root/.cache/huggingface
    stdin_open: true
    tty: true
    ipc: host
    command: lmdeploy serve api_server Qwen/Qwen2-VL-2B-Instruct
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: "all"
              capabilities: [gpu]
```

Then, you can execute the startup command as below:

```
docker-compose up -d
```

If you find the following logs after running `docker logs -f lmdeploy`, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.15.10 Qwen2.5-VL

LMDeploy supports the following Qwen-VL series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
Qwen2.5-VL	3B, 7B, 32B, 72B	PyTorch

The next chapter demonstrates how to deploy a Qwen-VL model using LMDeploy, with Qwen2.5-VL-7B-Instruct as an example.

Installation

Please install LMDeploy by following the *installation guide*, and install other packages that Qwen2.5-VL needs

```
# Qwen2.5-VL requires the latest transformers (transformers >= 4.49.0)
pip install git+https://github.com/huggingface/transformers
# It's highly recommended to use `[decord]` feature for faster video loading.
pip install qwen-vl-utils[decord]==0.0.8
```

Offline inference

The following sample code shows the basic usage of the VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('Qwen/Qwen2.5-VL-7B-Instruct')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('Qwen/Qwen2.5-VL-7B-Instruct', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('Qwen/Qwen2.5-VL-7B-Instruct', log_level='INFO')

min_pixels = 64 * 28 * 28
max_pixels = 64 * 28 * 28
messages = [
```

(continues on next page)

(continued from previous page)

```
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(min_pixels=min_pixels, max_pixels=max_
↪pixels, url='https://raw.githubusercontent.com/QwenLM/Qwen-VL/master/assets/mm_
↪tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(min_pixels=min_pixels, max_pixels=max_
↪pixels, url='https://raw.githubusercontent.com/QwenLM/Qwen-VL/master/assets/mm_
↪tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

```
import numpy as np
from lmdeploy import pipeline, GenerationConfig
from decord import VideoReader, cpu
from lmdeploy.vl.constants import IMAGE_TOKEN
from lmdeploy.vl import encode_image_base64
from PIL import Image
pipe = pipeline('Qwen/Qwen2.5-VL-7B-Instruct', log_level='INFO')


def get_index(bound, fps, max_frame, first_idx=0, num_segments=32):
    if bound:
        start, end = bound[0], bound[1]
    else:
        start, end = -100000, 100000
    start_idx = max(first_idx, round(start * fps))
    end_idx = min(round(end * fps), max_frame)
    seg_size = float(end_idx - start_idx) / num_segments
    frame_indices = np.array([
        int(start_idx + (seg_size / 2) + np.round(seg_size * idx))
        for idx in range(num_segments)
    ])
    return frame_indices


def load_video(video_path, bound=None, num_segments=32):
    vr = VideoReader(video_path, ctx=cpu(0), num_threads=1)
    max_frame = len(vr) - 1
    fps = float(vr.get_avg_fps())
    pixel_values_list, num_patches_list = [], []
    frame_indices = get_index(bound, fps, max_frame, first_idx=0, num_segments=num_
↪segments)
    imgs = []
    for frame_index in frame_indices:
        img = Image.fromarray(vr[frame_index].asnumpy()).convert('RGB')
        imgs.append(img)
```

(continues on next page)

(continued from previous page)

```
    return imgs


video_path = 'red-panda.mp4'
imgs = load_video(video_path, num_segments=8)

question = ''
for i in range(len(imgs)):
    question = question + f'Frame{i+1}: {IMAGE_TOKEN}\n'

question += 'What is the red panda doing?'

content = [{'type': 'text', 'text': question}]
for img in imgs:
    content.append({'type': 'image_url', 'image_url': {'max_dynamic_patch': 1, 'url': f
↪'{encode_image_base64(img)}'}})

messages = [dict(role='user', content=content)]
out = pipe(messages, gen_config=GenerationConfig(top_k=1))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='Describe this video in detail. Don\'t repeat.
↪'))
out = pipe(messages, gen_config=GenerationConfig(top_k=1))
```

1.15.11 Molmo

LMDeploy supports the following molmo series of models, which are detailed in the table below:

Model	Size	Supported Inference Engine
Molmo-7B-D-0924	7B	TurboMind
Molmo-72-0924	72B	TurboMind

The next chapter demonstrates how to deploy a molmo model using LMDeploy, with Molmo-7B-D-0924 as an example.

Installation

Please install LMDeploy by following the *installation guide*

Offline inference

The following sample code shows the basic usage of VLM pipeline. For detailed information, please refer to *VLM Offline Inference Pipeline*

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image

pipe = pipeline('allenai/Molmo-7B-D-0924')

image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
response = pipe((f'describe this image', image))
print(response)
```

More examples are listed below:

```
from lmdeploy import pipeline, GenerationConfig

pipe = pipeline('allenai/Molmo-7B-D-0924', log_level='INFO')
messages = [
    dict(role='user', content=[
        dict(type='text', text='Describe the two images in detail.'),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Beijing_Small.jpeg')),
        dict(type='image_url', image_url=dict(url='https://raw.githubusercontent.com/
↪QwenLM/Qwen-VL/master/assets/mm_tutorial/Chongqing_Small.jpeg'))
    ])
]
out = pipe(messages, gen_config=GenerationConfig(do_sample=False))

messages.append(dict(role='assistant', content=out.text))
messages.append(dict(role='user', content='What are the similarities and differences␣
↪between these two images.'))
out = pipe(messages, gen_config=GenerationConfig(do_sample=False))
```

Online serving

You can launch the server by the `lmdeploy serve api_server` CLI:

```
lmdeploy serve api_server allenai/Molmo-7B-D-0924
```

You can also start the service using the docker image:

```
docker run --runtime nvidia --gpus all \
    -v ~/.cache/huggingface:/root/.cache/huggingface \
    --env "HUGGING_FACE_HUB_TOKEN=<secret>" \
    -p 23333:23333 \
```

(continues on next page)

(continued from previous page)

```
    --ipc=host \
    openmmlab/lmdeploy:latest \
    lmdeploy serve api_server allenai/Molmo-7B-D-0924
```

If you find the following logs, it means the service launches successfully.

```
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
HINT:    Please open  http://0.0.0.0:23333   in a browser for detailed api usage!!!
INFO:     Started server process [2439]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on  http://0.0.0.0:23333  (Press CTRL+C to quit)
```

The arguments of `lmdeploy serve api_server` can be reviewed in detail by `lmdeploy serve api_server -h`.

More information about `api_server` as well as how to access the service can be found from *here*

1.15.12 Gemma3

Introduction

Gemma is a family of lightweight, state-of-the-art open models from Google, built from the same research and technology used to create the Gemini models. Gemma 3 models are multimodal, handling text and image input and generating text output, with open weights for both pre-trained variants and instruction-tuned variants. Gemma 3 has a large, 128K context window, multilingual support in over 140 languages, and is available in more sizes than previous versions. Gemma 3 models are well-suited for a variety of text generation and image understanding tasks, including question answering, summarization, and reasoning. Their relatively small size makes it possible to deploy them in environments with limited resources such as laptops, desktops or your own cloud infrastructure, democratizing access to state of the art AI models and helping foster innovation for everyone.

Quick Start

Install LMDeploy by following the *installation guide*.

Prepare

When deploying the **Gemma3** model using LMDeploy, please install the latest transformers.

Offline inference pipeline

The following sample code shows the basic usage of VLM pipeline. For more examples, please refer to *VLM Offline Inference Pipeline*.

```
from lmdeploy import pipeline
from lmdeploy.vl import load_image


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    pipe = pipeline('google/gemma-3-12b-it')

    image = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/tests/
↪data/tiger.jpeg')
    response = pipe(('describe this image', image))
    print(response)
```

1.16 AWQ/GPTQ

LMDeploy TurboMind engine supports the inference of 4bit quantized models that are quantized both by AWQ and GPTQ, but its quantization module only supports the AWQ quantization algorithm.

The following NVIDIA GPUs are available for AWQ/GPTQ INT4 inference:

- V100(sm70): V100
- Turing(sm75): 20 series, T4
- Ampere(sm80,sm86): 30 series, A10, A16, A30, A100
- Ada Lovelace(sm89): 40 series

Before proceeding with the quantization and inference, please ensure that lmdeploy is installed by following the *installation guide*

The remainder of this article is structured into the following sections:

- *Quantization*
- *Evaluation*
- *Inference*
- *Service*
- *Performance*

1.16.1 Quantization

A single command execution is all it takes to quantize the model. The resulting quantized weights are then stored in the $WORK_DIR directory.

```
export HF_MODEL=internlm/internlm2_5-7b-chat
export WORK_DIR=internlm/internlm2_5-7b-chat-4bit

lmdeploy lite auto_awq \
   $HF_MODEL \
  --calib-dataset 'wikitext2' \
  --calib-samples 128 \
  --calib-seqlen 2048 \
  --w-bits 4 \
  --w-group-size 128 \
  --batch-size 1 \
  --work-dir $WORK_DIR
```

Typically, the above command doesn't require filling in optional parameters, as the defaults usually suffice. For instance, when quantizing the internlm/internlm2_5-7b-chat model, the command can be condensed as:

```
lmdeploy lite auto_awq internlm/internlm2_5-7b-chat --work-dir internlm2_5-7b-chat-4bit
```

Note:

- We recommend that you specify the –work-dir parameter, including the model name as demonstrated in the example above. This facilitates LMDeploy in fuzzy matching the –work-dir with an appropriate built-in chat template. Otherwise, you will have to designate the chat template during inference.
- If the quantized model's accuracy is compromised, it is recommended to enable –search-scale for re-quantization and increase the –batch-size, for example, to 8. When search_scale is enabled, the quantization process will take more time. The –batch-size affects the amount of memory used, which can be adjusted according to actual conditions as needed.

Upon completing quantization, you can engage with the model efficiently using a variety of handy tools. For example, you can initiate a conversation with it via the command line:

```
lmdeploy chat ./internlm2_5-7b-chat-4bit --model-format awq
```

1.16.2 Evaluation

Please refer to OpenCompass about model evaluation with LMDeploy. Here is the guide

1.16.3 Inference

Trying the following codes, you can perform the batched offline inference with the quantized model:

```
from lmdeploy import pipeline, TurbomindEngineConfig
engine_config = TurbomindEngineConfig(model_format='awq')
pipe = pipeline("./internlm2_5-7b-chat-4bit", backend_config=engine_config)
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

For more information about the pipeline parameters, please refer to *here*.

In addition to performing inference with the quantized model on localhost, LMDeploy can also execute inference for the 4bit quantized model derived from AWQ algorithm available on Huggingface Hub, such as models from the lmdeploy space and TheBloke space

```
# inference with models from lmdeploy space
from lmdeploy import pipeline, TurbomindEngineConfig
pipe = pipeline("lmdeploy/llama2-chat-70b-4bit",
                backend_config=TurbomindEngineConfig(model_format='awq', tp=4))
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)

# inference with models from thebloke space
from lmdeploy import pipeline, TurbomindEngineConfig, ChatTemplateConfig
pipe = pipeline("TheBloke/LLaMA2-13B-Tiefighter-AWQ",
                backend_config=TurbomindEngineConfig(model_format='awq'),
                chat_template_config=ChatTemplateConfig(model_name='llama2')
                )
```

(continues on next page)

(continued from previous page)

```
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

1.16.4 Service

LMDeploy's `api_server` enables models to be easily packed into services with a single command. The provided RESTful APIs are compatible with OpenAI's interfaces. Below are an example of service startup:

```
lmdeploy serve api_server ./internlm2_5-7b-chat-4bit --backend turbomind --model-format␣
↪awq
```

The default port of `api_server` is `23333`. After the server is launched, you can communicate with server on terminal through `api_client`:

```
lmdeploy serve api_client http://0.0.0.0:23333
```

You can overview and try out `api_server` APIs online by swagger UI at `http://0.0.0.0:23333`, or you can also read the API specification from *here*.

1.16.5 Performance

We benchmarked the Llama-2-7B-chat and Llama-2-13B-chat models with 4-bit quantization on NVIDIA GeForce RTX 4090. And we measure the token generation throughput (tokens/s) by setting a single prompt token and generating 512 tokens. All the results are measured for single batch inference.

model	llm-awq	mlc-llm	turbomind
Llama-2-7B-chat	112.9	159.4	206.4
Llama-2-13B-chat	N/A	90.7	115.8

1.16.6 FAQs

1. Out of Memory error during quantization due to insufficient GPU memory: This can be addressed by reducing the parameter `--calib-seqlen`, increasing the parameter `--calib-samples`, and set `--batch-size` to 1.

1.17 SmoothQuant

LMDeploy provides functions for quantization and inference of large language models using 8-bit integers(INT8). For GPUs such as Nvidia H100, lmdeploy also supports 8-bit floating point(FP8).

And the following NVIDIA GPUs are available for INT8/FP8 inference respectively:

- INT8
  - V100(sm70): V100
  - Turing(sm75): 20 series, T4
  - Ampere(sm80,sm86): 30 series, A10, A16, A30, A100
  - Ada Lovelace(sm89): 40 series

  - Hopper(sm90): H100

- FP8
  - Ada Lovelace(sm89): 40 series
  - Hopper(sm90): H100

First of all, run the following command to install lmdeploy:

```
pip install lmdeploy[all]
```

1.17.1 8-bit Weight Quantization

Performing 8-bit weight quantization involves three steps:

1. **Smooth Weights**: Start by smoothing the weights of the Language Model (LLM). This process makes the weights more amenable to quantizing.
2. **Replace Modules**: Locate DecoderLayers and replace the modules RSMNorm and nn.Linear with QRSMNorm and QLinear modules respectively. These 'Q' modules are available in the lmdeploy/pytorch/models/q_modules.py file.
3. **Save the Quantized Model**: Once you've made the necessary replacements, save the new quantized model.

lmdeploy provides `lmdeploy lite smooth_quant` command to accomplish all three tasks detailed above. Note that the argument `--quant-dtype` is used to determine if you are doing int8 or fp8 weight quantization. To get more info about usage of the cli, run `lmdeploy lite smooth_quant --help`

Here are two examples:

- int8

```
lmdeploy lite smooth_quant internlm/internlm2_5-7b-chat --work-dir ./internlm2_5-7b-
↪chat-int8 --quant-dtype int8
```

- fp8

```
lmdeploy lite smooth_quant internlm/internlm2_5-7b-chat --work-dir ./internlm2_5-7b-
↪chat-fp8 --quant-dtype fp8
```

1.17.2 Inference

Trying the following codes, you can perform the batched offline inference with the quantized model:

```
from lmdeploy import pipeline, PytorchEngineConfig

engine_config = PytorchEngineConfig(tp=1)
pipe = pipeline("internlm2_5-7b-chat-int8", backend_config=engine_config)
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

1.17.3 Service

LMDeploy's `api_server` enables models to be easily packed into services with a single command. The provided RESTful APIs are compatible with OpenAI's interfaces. Below are an example of service startup:

```
lmdeploy serve api_server ./internlm2_5-7b-chat-int8 --backend pytorch
```

The default port of `api_server` is `23333`. After the server is launched, you can communicate with server on terminal through `api_client`:

```
lmdeploy serve api_client http://0.0.0.0:23333
```

You can overview and try out `api_server` APIs online by swagger UI at `http://0.0.0.0:23333`, or you can also read the API specification from *here*.

1.18 INT4/INT8 KV Cache

Since v0.4.0, LMDeploy has supported **online** key-value (kv) cache quantization with int4 and int8 numerical precision, utilizing an asymmetric quantization method that is applied on a per-head, per-token basis. The original kv offline quantization method has been removed.

Intuitively, quantization is beneficial for increasing the number of kv block. Compared to fp16, the number of kv block for int4/int8 kv can be increased by 4 times and 2 times respectively. This means that under the same memory conditions, the system can support a significantly increased number of concurrent operations after kv quantization, thereby ultimately enhancing throughput.

However, quantization typically brings in some loss of model accuracy. We have used OpenCompass to evaluate the accuracy of several models after applying int4/int8 quantization. int8 kv keeps the accuracy while int4 kv has slight loss. The detailed results are presented in the *Evaluation* section. You can refer to the information and choose wisely based on your requirements.

LMDeploy inference with quantized kv supports the following NVIDIA GPU models:

- Volta architecture (sm70): V100
- Turing architecture (sm75): 20 series, T4
- Ampere architecture (sm80, sm86): 30 series, A10, A16, A30, A100
- Ada Lovelace architecture (sm89): 40 series
- Hopper architecture (sm90): H100, H200

In summary, LMDeploy kv quantization has the following advantages:

1. data-free online quantization
2. Supports all nvidia GPU models with Volta architecture (sm70) and above
3. KV int8 quantization has almost lossless accuracy, and KV int4 quantization accuracy is within an acceptable range
4. Efficient inference, with int8/int4 kv quantization applied to llama2-7b, RPS is improved by round 30% and 40% respectively compared to fp16

1.18.1 TurboQuant

LMDeploy supports KV quantization based on Google Research's TurboQuant technology (to be presented at ICLR 2026), achieving higher compression ratio with near-zero accuracy loss through K=4bit QJL4 + V=2bit MSE combination.

Principles

TurboQuant achieves efficient compression through two key steps:

1. **High-quality compression (PolarQuant method)**: First randomly rotates the data vectors (using orthogonal transforms like Hadamard transform). This clever step simplifies the data's geometry, making it easy to apply a standard, high-quality quantizer to each part of the vector individually. This stage uses most of the compression power (the majority of the bits) to capture the main concept and strength of the original vector.
2. **Eliminating hidden errors (QJL method)**: Uses a small, residual amount of compression power (just 1 bit) to apply the QJL (Quantized Johnson-Lindenstrauss) algorithm to the tiny amount of error left over from the first stage. The QJL stage acts as a mathematical error-checker that eliminates bias, leading to more accurate attention scores.

K/V Quantization Scheme

- **K Path - QJL4 Quantization**:
  - Uses 3-bit Lloyd-Max codebook for MSE quantization (captures main information)
  - Uses 1-bit QJL to store residual sign (eliminates error bias)
  - Each token's K is compressed to 4-bit
- **V Path - MSE int2 Quantization**:
  - Uses 2-bit Lloyd-Max codebook for MSE quantization
  - Each token's V is compressed to 2-bit
  - Stores normalization coefficients for dequantization

Advantages

- **Zero accuracy loss**: Through PolarQuant + QJL combination, achieves high compression rate while maintaining model accuracy
- **Higher compression ratio**: K 4bit + V 2bit = average 3bit, further compression compared to int4's 4bit
- **Eliminates quantization bias**: QJL algorithm acts as error-checker, effectively eliminating quantization-induced bias

Performance Benchmark

Tested on H200 with Qwen3-30B-A3B-Base model and ShareGPT dataset:

Metric	Baseline (quant_policy=0)	TurboQuant (quant_policy=42)	Change
Input throughput	2368.8 tok/s	2195.8 tok/s	-7.3%
Output throughput	2186.7 tok/s	2027.0 tok/s	-7.3%
Request throughput	10.74 req/s	9.96 req/s	-7.3%
Mean E2E latency	5.888s	6.348s	+7.8%
Mean TTFT	1.139s	1.235s	+8.4%
Mean TPOT	0.024s	0.026s	+8.3%
Mean ITL	0.059s	0.059s	~unchanged

Test configuration: GPU: H200, Model: Qwen3-30B-A3B-Base, Dataset: ShareGPT, Concurrency: 64, Requests: 5000

Takeaway: TurboQuant K4V2 achieves ~5x KV cache memory reduction with about 7%-8% end-to-end performance overhead, which looks like a reasonable trade-off for memory-bound serving scenarios.

Limitations

- **PytorchEngine only**: TurboQuant currently only supports PyTorch engine, not Turbomind engine
- **MLA not supported**: Does not support Multi-head Latent Attention architecture
- **Speculative decoding not supported**: Does not support speculative decoding
- Requires head_dim to be a power of 2
- Requires `fast_hadamard_transform` package for best performance (optional)

Optional Dependency

TurboQuant uses Hadamard transform to accelerate the quantization process. Installing `fast_hadamard_transform` provides better performance:

```
pip install fast_hadamard_transform
```

Without this dependency, TurboQuant still works correctly, but performance may be slightly reduced.

In the next section, we will take `internlm2-chat-7b` model as an example, introducing the usage of kv quantization and inference of lmdeploy. But before that, please ensure that lmdeploy is installed.

```
pip install lmdeploy
```

1.18.2 Usage

Applying kv quantization and inference via LMDeploy is quite straightforward. Simply set the `quant_policy` parameter.

LMDeploy specifies that `quant_policy=4` stands for 4-bit kv, `quant_policy=8` indicates 8-bit kv, and `quant_policy=42` indicates TurboQuant.

Offline inference

```
from lmdeploy import pipeline, TurbomindEngineConfig
engine_config = TurbomindEngineConfig(quant_policy=8)
pipe = pipeline("internlm/internlm2_5-7b-chat", backend_config=engine_config)
response = pipe(["Hi, pls intro yourself", "Shanghai is"])
print(response)
```

Serving

```
lmdeploy serve api_server internlm/internlm2_5-7b-chat --quant-policy 8
```

TurboQuant

TurboQuant uses `quant_policy=42`, **PytorchEngine only**:

```
from lmdeploy import pipeline, PytorchEngineConfig
engine_config = PytorchEngineConfig(
    tp=1,
    cache_max_entry_count=0.8,
    quant_policy=42  # TurboQuant: K=4bit QJL4 + V=2bit MSE
)
pipe = pipeline("Qwen/Qwen3-8B", backend_config=engine_config)
response = pipe.infer("Hello, how are you?", max_new_tokens=30)
print(response.text)
```

1.18.3 Evaluation

We apply kv quantization of LMDeploy to several LLM models and utilize OpenCompass to evaluate the inference accuracy. The results are shown in the table below:

-	-	-	llama2-	-	-	internlm	-	-	internlm2	-	-	qwen1.5	-	-
			7b-chat			chat-7b			chat-7b			7b-chat		
datase	ver-sion	met-ric	kv fp16	kv int8	kv int4	kv fp16	kv int8	kv int4	kv fp16	kv int8	kv int4	fp16	kv int8	kv int4
ce-val	-	naive_	28.42	27.96	27.58	60.45	60.88	60.28	78.06	77.87	77.05	70.56	70.49	68.62
mmlu	-	naive_	35.64	35.58	34.79	63.91	64	62.36	72.30	72.27	71.17	61.48	61.56	60.65
triv-iaqa	2121	score	56.09	56.13	53.71	58.73	58.7	58.18	65.09	64.87	63.28	44.62	44.77	44.04
gsm8k	1d7fe	ac-cu-racy	28.2	28.05	27.37	70.13	69.75	66.87	85.67	85.44	83.78	54.97	56.41	54.74
race-middl	9a54l	ac-cu-racy	41.57	41.78	41.23	88.93	88.93	88.93	92.76	92.83	92.55	87.33	87.26	86.28
race-high	9a54l	ac-cu-racy	39.65	39.77	40.77	85.33	85.31	84.62	90.51	90.42	90.42	82.53	82.59	82.02

For detailed evaluation methods, please refer to *this* guide. Remember to pass `quant_policy` to the inference engine in the config file.

1.18.4 Performance

model	kv type	test settings	RPS	v.s. kv fp16
llama2-chat-7b	fp16	tp1 / ratio 0.8 / bs 256 / prompts 10000	14.98	1.0
-	int8	tp1 / ratio 0.8 / bs 256 / prompts 10000	19.01	1.27
-	int4	tp1 / ratio 0.8 / bs 256 / prompts 10000	20.81	1.39
llama2-chat-13b	fp16	tp1 / ratio 0.9 / bs 128 / prompts 10000	8.55	1.0
-	int8	tp1 / ratio 0.9 / bs 256 / prompts 10000	10.96	1.28
-	int4	tp1 / ratio 0.9 / bs 256 / prompts 10000	11.91	1.39
internlm2-chat-7b	fp16	tp1 / ratio 0.8 / bs 256 / prompts 10000	24.13	1.0
-	int8	tp1 / ratio 0.8 / bs 256 / prompts 10000	25.28	1.05
-	int4	tp1 / ratio 0.8 / bs 256 / prompts 10000	25.80	1.07

The performance data is obtained by `benchmark/profile_throughput.py`

1.19 llm-compressor Support

This guide aims to introduce how to use LMDeploy's TurboMind inference engine to run models quantized by the vllm-project/llm-compressor tool.

Currently supported `llm-compressor` quantization types include:

- int4 quantization (e.g., AWQ, GPTQ)

These quantized models can run via the TurboMind engine on the following NVIDIA GPU architectures:

Compute Capability	Micro-architecture	GPUs
7.0	Volta	V100
7.2	Volta	Jetson Xavier
7.5	Turing	GeForce RTX 20 series, T4
8.0	Ampere	A100, A800, A30
8.6	Ampere	GeForce RTX 30 series, A40, A10
8.7	Ampere	Jetson Orin
8.9	Ada Lovelace	GeForce RTX 40 series, L40, L20
9.0	Hopper	H20, H200, H100, GH200
12.0	Blackwell	GeForce RTX 50 series

LMDeploy will continue to follow up and expand support for the `llm-compressor` project.

The remainder of this document consists of the following sections:

- *Model Quantization*
- *Model Deployment*
- *Accuracy Evaluation*

1.19.1 Model Quantization

`llm-compressor` provides a wealth of model quantization examples. Please refer to its tutorials to select a quantization algorithm supported by LMDeploy to complete your model quantization work.

LMDeploy also provides a built-in script for AWQ quantization of **Qwen3-30B-A3B** using `llm-compressor` for your reference:

```
# Create conda environment
conda create -n lmdeploy python=3.10 -y
conda activate lmdeploy

# Install llm-compressor
pip install llmcompressor

# Clone lmdeploy source code and run the quantization example
git clone https://github.com/InternLM/lmdeploy
cd lmdeploy
python examples/lite/qwen3_30b_a3b_awq.py --work-dir ./qwen3_30b_a3b_awq
```

In the following sections, we will use this quantized model as an example to introduce model deployment and accuracy evaluation methods.

1.19.2 Model Deployment

Offline Inference

With the quantized model, offline batch processing can be implemented with just a few lines of code:

```
from lmdeploy import pipeline, TurbomindEngineConfig

engine_config = TurbomindEngineConfig()
with pipeline("./qwen3_30b_a3b_4bit", backend_config=engine_config) as pipe:
    response = pipe(["Hi, pls intro yourself", "Shanghai is"])
    print(response)
```

For a detailed introduction to the pipeline, please refer to here.

Online Serving

LMDeploy api_server supports encapsulating the model as a service with a single command. The provided RESTful APIs are compatible with OpenAI interfaces. Below is an example of starting the service:

```
lmdeploy serve api_server ./qwen3_30b_a3b_4bit --backend turbomind
```

The default service port is 23333. After the server starts, you can access the service via the OpenAI SDK. For command arguments and methods to access the service, please read this document.

1.19.3 Accuracy Evaluation

Aftering deploying AWQ symmetric/asymmetric quantized models of Qwen3-8B (Dense) and Qwen3-30B-A3B (MoE) as services via LMDeploy, we evaluated their accuracy on several academic datasets using opencompass. Results indicate that, for Qwen3-8B, asymmetric quantization generally outperforms symmetric quantization, while Qwen3-30B-A3B shows no substantial difference between symmetric and asymmetric quantization. Compared with BF16, Qwen3-8B shows a smaller accuracy gap under both symmetric and asymmetric quantization than Qwen3-30B-A3B. Compared with BF16, accuracy drops significantly on long-output datasets such as aime2025 (avg 17,635 tokens) and LCB (avg 14,157 tokens), while on medium/short-output datasets like ifeval (avg 1,885 tokens) and mmlu_pro (avg 2,826 tokens), the accuracy is as expected.

dataset	Qwen3-8B			Qwen3-30B-A3B		
	bf16	awq sym	awq asym	bf16	awq sym	awq asym
ifeval	85.58	83.73	85.77	86.32	84.10	84.29
hle	5.05	5.05	5.24	7.00	5.47	5.65
gpqa	59.97	56.57	59.47	61.74	57.95	57.07
aime2025	69.48	64.38	63.96	73.44	64.79	66.67
mmlu_pro	73.69	71.73	72.34	77.85	75.77	75.69
LCBCodeGeneration	50.86	44.10	46.95	56.67	50.86	49.24

For reproduction methods, please refer to this document.

1.20 Benchmark

Please install the lmdeploy precompiled package and download the script and the test dataset:

```
pip install lmdeploy
# clone the repo to get the benchmark script
git clone --depth=1 https://github.com/InternLM/lmdeploy
cd lmdeploy
# switch to the tag corresponding to the installed version:
git fetch --tags
# Check the installed lmdeploy version:
pip show lmdeploy | grep Version
# Then, check out the corresponding tag (replace <version> with the version string):
git checkout <version>
# download the test dataset
wget https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered/resolve/
↪main/ShareGPT_V3_unfiltered_cleaned_split.json
```

1.20.1 Benchmark offline pipeline API

```
python3 benchmark/profile_pipeline_api.py ShareGPT_V3_unfiltered_cleaned_split.json meta-
↪llama/Meta-Llama-3-8B-Instruct
```

For a comprehensive list of available arguments, please execute `python3 benchmark/profile_pipeline_api.py -h`

1.20.2 Benchmark offline engine API

```
python3 benchmark/profile_throughput.py ShareGPT_V3_unfiltered_cleaned_split.json meta-
↪llama/Meta-Llama-3-8B-Instruct
```

Detailed argument specification can be retrieved by running `python3 benchmark/profile_throughput.py -h`

1.20.3 Benchmark online serving

Launch the server first (you may refer *here* for guide) and run the following command:

```
python3 benchmark/profile_restful_api.py --backend lmdeploy --num-prompts 5000 --dataset-
↪path ShareGPT_V3_unfiltered_cleaned_split.json
```

For detailed argument specification of `profile_restful_api.py`, please run the help command `python3 benchmark/profile_restful_api.py -h`.

1.21 Model Evaluation Guide

This document describes how to evaluate a model's capabilities on academic datasets using OpenCompass and LMDeploy. The complete evaluation process consists of two main stages: inference stage and evaluation stage.

During the inference stage, the target model is first deployed as an inference service using LMDeploy. OpenCompass then sends dataset content as requests to this service and collects the generated responses.

In the evaluation stage, the OpenCompass evaluation model `opencompass/CompassVerifier-32B` is deployed as a service via LMDeploy. OpenCompass subsequently submits the inference results to this service to obtain final evaluation scores.

If sufficient computational resources are available, please refer to the *End-to-End Evaluation* section for complete workflow execution. Otherwise, we recommend following the *Step-by-Step Evaluation* section to execute both stages sequentially.

1.21.1 Environment Setup

```
pip install lmdeploy
pip install "opencompass[full]"

# Download the lmdeploy source code, which will be used in subsequent steps to access␣
↪eval script and configuration
git clone --depth=1 https://github.com/InternLM/lmdeploy.git
```

It is recommended to install LMDeploy and OpenCompass in separate Python virtual environments to avoid potential dependency conflicts.

1.21.2 End-to-End Evaluation

1. **Deploy Target Model**

```
lmdeploy serve api_server <model_path> --server-port 10000 <--other-options>
```

2. **Deploy Evaluation Model (Judger)**

```
lmdeploy serve api_server opencompass/CompassVerifier-32B --server-port 20000 --tp 2
```

3. **Generate Evaluation Configuration and Execute**

```
cd {the/root/path/of/lmdeploy/repo}

## Specify the dataset path. OC will download the datasets automatically if they are
## not found in the path
export HF_DATASETS_CACHE=/nvme4/huggingface_hub/datasets
export COMPASS_DATA_CACHE=/nvme1/shared/opencompass/.cache
python eval/eval.py {task_name} \
    --mode all \
    --api-server http://{api-server-ip}:10000 \
    --judger-server http://{judger-server-ip}:20000 \
    -w {oc_output_dir}
```

For detailed usage instructions about `eval.py`, such as specifying evaluation datasets, please run `python eval/eval.py --help`.

After evaluation completion, results are saved in `{oc_output_dir}/{yyyymmdd_hhmmss}`, where `{yyyymmdd_hhmmss}` represents the task timestamp.

1.21.3 Step-by-Step Evaluation

Inference Stage

This stage generates model responses for the dataset.

1. **Deploy Target Model**

```
lmdeploy serve api_server <model_path> --server-port 10000 <--other-options>
```

2. **Generate Inference Configuration and Execute**

```
cd {the/root/path/of/lmdeploy/repo}

## Specify the dataset path. OC will download the datasets automatically if they are
## not found in the path
export COMPASS_DATA_CACHE=/nvme1/shared/opencompass/.cache
export HF_DATASETS_CACHE=/nvme4/huggingface_hub/datasets
# Run inference task
python eval/eval.py {task_name} \
    --mode infer \
    --api-server http://{api-server-ip}:10000 \
    -w {oc_output_dir}
```

For detailed usage instructions about `eval.py`, such as specifying evaluation datasets, please run `python eval/eval.py --help`.

Evaluation Stage

This stage uses the evaluation model (Judger) to assess the quality of inference results.

1. **Deploy Evaluation Model (Judger)**

```
lmdeploy serve api_server opencompass/CompassVerifier-32B --server-port 20000 --tp 2 --
↪session-len 65536
```

2. **Generate Evaluation Configuration and Execute**

```
cd {the/root/path/of/lmdeploy/repo}

## Specify the dataset path. OC will download the datasets automatically if they are
## not found in the path
export COMPASS_DATA_CACHE=/nvme1/shared/opencompass/.cache
export HF_DATASETS_CACHE=/nvme4/huggingface_hub/datasets
# Run evaluation task
opencompass /path/to/judger_config.py -m eval -w {oc_output_dir} -r {yyyymmdd_hhmmss}
```

Important Notes:

- `task_name` must be identical to the one used in the inference stage
- The `oc_output_dir` specified with `-w` must match the directory used in the inference stage
- The `-r` parameter indicates "previous outputs & results" and should specify the timestamp directory generated during the inference stage (the subdirectory under `{oc_output_dir}`)

For detailed usage instructions about `eval.py`, such as specifying evaluation datasets, please run `python eval/eval.py --help`.

1.22 Multi-Modal Model Evaluation Guide

This document describes how to evaluate multi-modal models' capabilities using VLMEvalKit and LMDeploy.

1.22.1 Environment Setup

```
pip install lmdeploy

git clone https://github.com/open-compass/VLMEvalKit.git
cd VLMEvalKit && pip install -e .
```

It is recommended to install LMDeploy and VLMEvalKit in separate Python virtual environments to avoid potential dependency conflicts.

1.22.2 Evaluations

1. **Deploy Large Multi-Modality Models (LMMs)**

```
lmdeploy serve api_server <model_path> --server-port 23333 <--other-options>
```

2. **Config the Evaluation Settings**

Modify `VLMEvalKit/vlmeval/config.py`, add following LMDeploy API configurations in the `api_models` dictionary.

The `<task_name>` is a custom name for your evaluation task (e.g., `lmdeploy_qwen3vl-4b`). The `model` parameter should match the `<model_path>` used in the `lmdeploy serve` command.

```
// filepath: VLMEvalKit/vlmeval/config.py
// ...existing code...
api_models = {
    # lmdeploy api
    ...,
    "<task_name>": partial(
        LMDeployAPI,
        api_base="http://0.0.0.0:23333/v1/chat/completions",
        model="<model_path>",
        retry=4,
        timeout=1200,
        temperature=0.7, # modify if needed
        max_new_tokens=16384, # modify if needed
    ),
```

(continues on next page)

(continued from previous page)

```
    ...
}
// ...existing code...
```

3. **Start Evaluations**

```
cd VLMEvalKit
python run.py --data OCRBench --model <task_name> --api-nproc 16 --reuse --verbose --api␣
↪123
```

The <task_name> should match the one used in the above config file.

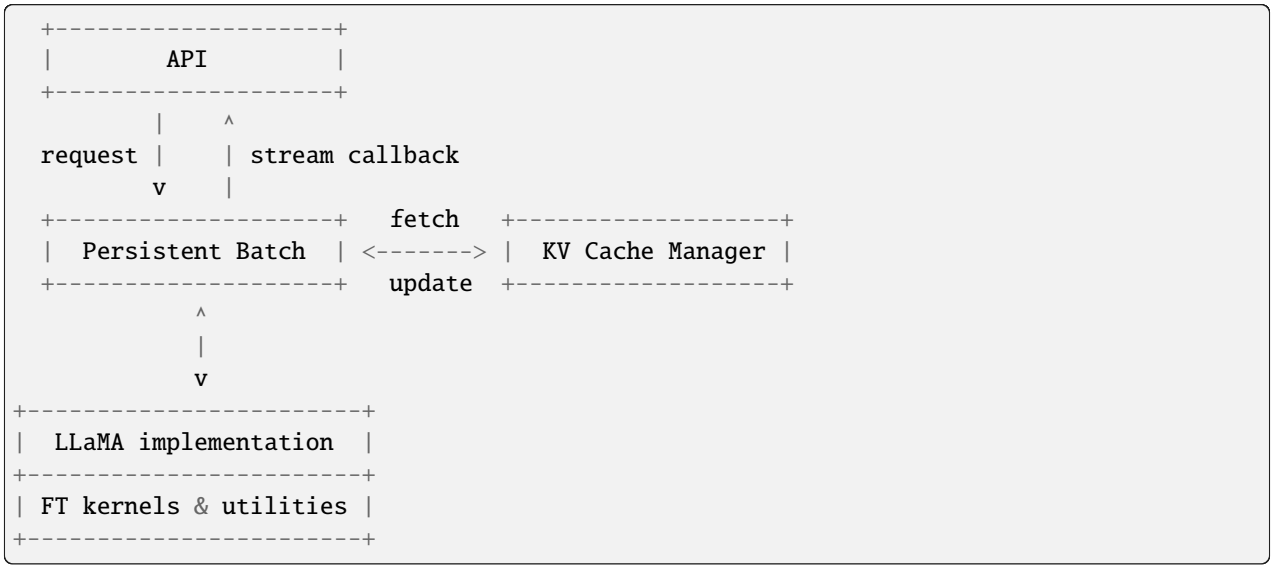
Parameter explanations:

- `--data`: Specify the dataset for evaluation (e.g., `OCRBench`).
- `--model`: Specify the model name, which must match the <task_name> in your `config.py`.
- `--api-nproc`: Specify the number of parallel API calls.
- `--reuse`: Reuse previous inference results to avoid re-running completed evaluations.
- `--verbose`: Enable verbose logging.

1.23 Architecture of TurboMind

TurboMind is an inference engine that supports high throughput inference for conversational LLMs. It's based on NVIDIA's FasterTransformer. Major features of TurboMind include an efficient LLaMa implementation, the persistent batch inference model and an extendable KV cache manager.

1.23.1 High level overview of TurboMind



1.23.2 Persistent Batch

You may recognize this feature as "continuous batching" in other repos. But during the concurrent development of the feature, we modeled the inference of a conversational LLM as a persistently running batch whose lifetime spans the entire serving process, hence the name "persistent batch". To put it simply

- The persistent batch as N pre-configured batch slots.
- Requests join the batch when there are free slots available. A batch slot is released and can be reused once the generation of the requested tokens is finished.
- **On cache-hits (see below), history tokens don't need to be decoded in every round of a conversation; generation of response tokens will start instantly.**
- The batch grows or shrinks automatically to minimize unnecessary computations.

1.23.3 KV Cache Manager

The KV cache manager of TurboMind is a memory-pool-liked object that also implements LRU policy so that it can be viewed as a form of **cache of KV caches**. It works in the following way

- All device memory required for KV cache is allocated by the manager. A fixed number of slots is pre-configured to match the memory size of the system. Each slot corresponds to the memory required by the KV cache of a single sequence. Allocation chunk-size can be configure to implement pre-allocate/on-demand style allocation policy (or something in-between).
- When space for the KV cache of a new sequence is requested but no free slots left in the pool, the least recently used sequence is evicted from the cache and its device memory is directly reused by the new sequence. However, this is not the end of the story.
- Fetching sequence currently resides in one of the slots resembles a *cache-hit*, the history KV cache is returned directly and no context decoding is needed.
- Victim (evicted) sequences are not erased entirely but converted to its most compact form, i.e. token IDs. When the same sequence id is fetched later (*cache-miss*) the token IDs will be decoded by FMHA backed context decoder and converted back to KV cache.
- The eviction and conversion are handled automatically inside TurboMind and thus transparent to the users. **From the user's aspect, system that use TurboMind has access to infinite device memory.**

1.23.4 LLaMa implementation

Our implementation of the LLaMa family models is modified from Gpt-NeoX model in FasterTransformer. In addition to basic refactoring and modifications to support the LLaMa family, we made some improvements to enable high performance inference of conversational models, most importantly:

- To support fast context decoding in multi-round conversations. We replaced the attention implementation in context decoder with a cutlass-based FMHA implementation that supports mismatched Q/K lengths.
- We introduced indirect buffer pointers in both context FMHA and generation FMHA to support the discontinuity in KV cache within the batch.
- To support concurrent inference with persistent batch, new synchronization mechanism was designed to orchestrate the worker threads running in tensor parallel mode.
- To maximize the throughput, we implement INT8 KV cache support to increase the max batch size. It's effective because in real-world serving scenarios, KV cache costs more memory and consumes more memory bandwidth than weights or other activations.

- We resolved an NCCL hang issue when running multiple model instances in TP mode within a single process, NCCL APIs are now guarded by host-side synchronization barriers.

1.23.5 API

TurboMind supports a Python API that enables streaming output and tensor parallel mode.

1.23.6 Difference between FasterTransformer and TurboMind

Apart of the features described above, there are still many minor differences that we don't cover in this document. Notably, many capabilities of FT are dropped in TurboMind because of the difference in objectives (e.g. prefix prompt, beam search, context embedding, sparse GEMM, GPT/T5/other model families, etc)

1.23.7 FAQ

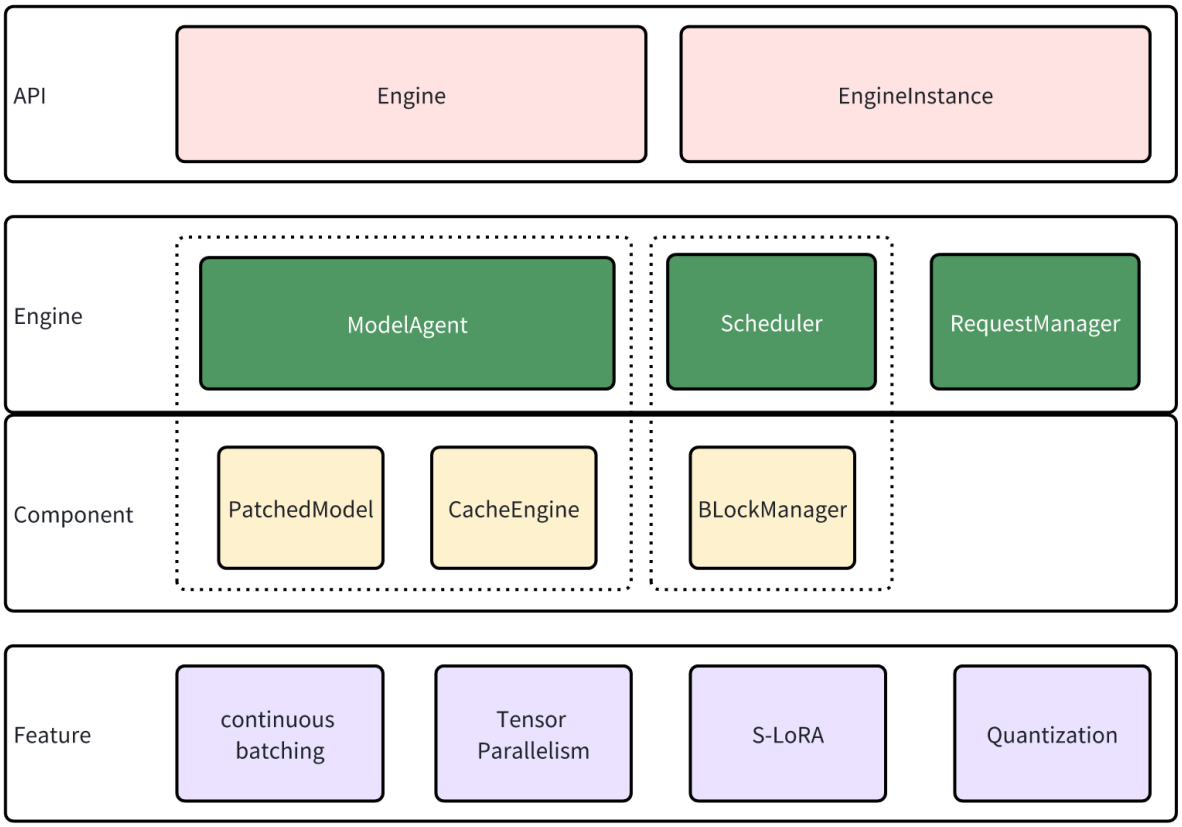
Supporting Huggingface models

For historical reasons, TurboMind's weight layout is based on the original LLaMa implementation (differ only by a transpose). The implementation in huggingface transformers uses a different layout for `W_q` and `W_k` which is handled in deploy.py.

1.24 Architecture of lmdeploy.pytorch

`lmdeploy.pytorch` is an inference engine in LMDeploy that offers a developer-friendly framework to users interested in deploying their own models and developing new features.

1.24.1 Design



1.24.2 API

`lmdeploy.pytorch` shares service interfaces with `Turbomind`, and the inference service is implemented by `Engine` and `EngineInstance`.

`EngineInstance` acts as the sender of inference requests, encapsulating and sending requests to the `Engine` to achieve streaming inference. The inference interface of `EngineInstance` is thread-safe, allowing instances in different threads to initiate requests simultaneously. The `Engine` will automatically perform batch processing based on the current system resources.

Engine is the request receiver and executor. It contain modules:

- `ModelAgent` serves as a wrapper for the model, handling tasks such as loading model/adapters, managing the cache, and implementing tensor parallelism.
- The `Scheduler` functions as the sequence manager, determining the sequences and adapters to participate in the current step, and subsequently allocating resources for them.
- `RequestManager` is tasked with sending and receiving requests. acting as the bridge between the `Engine` and `EngineInstance`.

1.24.3 Engine

The Engine responses to requests in a sub-thread, following this looping sequence:

1. Get new requests through `RequestManager`. These requests are cached for now.
2. The `Scheduler` performs scheduling, deciding which cached requests should be processed and allocating resources for them.
3. `ModelAgent` swaps the caches according to the information provided by the Scheduler, then performs inference with the patched model.
4. The `Scheduler` updates the status of requests based to the inference results from `ModelAgent`.
5. `RequestManager` responds to the sender (`EngineInstance`), and the process return to step 1.

Now, Let's delve deeper into the modules that participate in these steps.

Scheduler

In LLM inference, caching history key and value states is a common practice to prevent redundant computation. However, as history lengths vary in a batch of sequences, we need to pad the caches to enable batching inference. Unfortunately, this padding can lead to significant memory wastage, limiting the transformer's performance.

vLLM employs a paging-based strategy, allocating caches in page blocks to minimize extra memory usage. Our Scheduler module in the Engine shares a similar design, allocating resources based on sequence length in blocks and evicting unused blocks to support larger batching and longer session lengths.

Additionally, we support S-LoRA, which enables the use of multiple LoRA adapters on limited memory.

ModelAgent

`lmdeploy.pytorch` supports Tensor Parallelism, which leads to complex model initialization, cache allocation, and weight partitioning. ModelAgent is designed to abstract these complexities, allowing the Engine to focus solely on maintaining the pipeline.

ModelAgent consists of two components:

1. **`patched_model**: : This is the transformer model after patching. In comparison to the original model, the patched model incorporates additional features such as Tensor Parallelism, quantization, and high-performance kernels.
2. **cache_engine**: This component manages the caches. It receives commands from the Scheduler and performs host-device page swaps. Only GPU blocks are utilized for caching key/value pairs and adapters.

1.24.4 Features

`lmdeploy.pytorch` supports new features including:

- **Continuous Batching**: As the sequence length in a batch may vary, padding is often necessary for batching inference. However, large padding can lead to additional memory usage and unnecessary computation. To address this, we employ continuous batching, where all sequences are concatenated into a single long sequence to avoid padding.
- **Tensor Parallelism**: The GPU memory usage of LLM might exceed the capacity of a single GPU. Tensor parallelism is utilized to accommodate such models on multiple devices. Each device handles parts of the model simultaneously, and the results are gathered to ensure correctness.

- **S-LoRA**: LoRA adapters can be used to train LLM on devices with limited memory. While it's common practice to merge adapters into the model weights before deployment, loading multiple adapters in this way can consume a significant amount of memory. We support S-LoRA, where adapters are paged and swapped in when necessary. Special kernels are developed to support inference with unmerged adapters, enabling the loading of various adapters efficiently.
- **Quantization**: Model quantization involves performing computations with low precision. `lmdeploy.pytorch` supports w8a8 quantization. For more details, refer to *w8a8*.

1.25 lmdeploy.pytorch New Model Support

lmdeploy.pytorch is designed to simplify the support for new models and the development of prototypes. Users can adapt new models according to their own needs.

1.25.1 Model Support

Configuration Loading (Optional)

lmdeploy.pytorch initializes the engine based on the model's config file. If the parameter naming of the model to be integrated differs from common models in transformers, parsing errors may occur. A custom ConfigBuilder can be added to parse the configuration.

```
# lmdeploy/pytorch/configurations/gemma.py

from lmdeploy.pytorch.config import ModelConfig

from .builder import AutoModelConfigBuilder


class GemmaModelConfigBuilder(AutoModelConfigBuilder):

    @classmethod
    def condition(cls, hf_config):
        # Check if hf_config is suitable for this builder
        return hf_config.model_type in ['gemma', 'gemma2']

    @classmethod
    def build(cls, hf_config, model_path: str = None):
        # Use the hf_config loaded by transformers
        # Construct the ModelConfig for the pytorch engine
        return ModelConfig(hidden_size=hf_config.hidden_size,
                           num_layers=hf_config.num_hidden_layers,
                           num_attention_heads=hf_config.num_attention_heads,
                           num_key_value_heads=hf_config.num_key_value_heads,
                           bos_token_id=hf_config.bos_token_id,
                           eos_token_id=hf_config.eos_token_id,
                           head_dim=hf_config.head_dim,
                           vocab_size=hf_config.vocab_size)
```

The `lmdeploy.pytorch.check_env.check_model` function can be used to verify if the configuration can be parsed correctly.

Implementing the Model

After ensuring that the configuration can be parsed correctly, you can start implementing the model logic. Taking the implementation of llama as an example, we need to create the model using the configuration file from transformers.

```
class LlamaForCausalLM(nn.Module):

    # Constructor, builds the model with the given config
    # ctx_mgr is the context manager, which can be used to pass engine configurations or␣
↪additional parameters
    def __init__(self,
                 config: LlamaConfig,
                 ctx_mgr: StepContextManager,
                 dtype: torch.dtype = None,
                 device: torch.device = None):
        super().__init__()
        self.config = config
        self.ctx_mgr = ctx_mgr
        # build LLamaModel
        self.model = LlamaModel(config, dtype=dtype, device=device)
        # build lm_head
        self.lm_head = build_rowwise_linear(config.hidden_size,
                                            config.vocab_size,
                                            bias=False,
                                            dtype=dtype,
                                            device=device)

    # Model inference function
    # It is recommended to use the same parameters as below
    def forward(
        self,
        input_ids: torch.Tensor,
        position_ids: torch.Tensor,
        past_key_values: List[List[torch.Tensor]],
        attn_metadata: Any = None,
        inputs_embeds: torch.Tensor = None,
        **kwargs,
    ):
        hidden_states = self.model(
            input_ids=input_ids,
            position_ids=position_ids,
            past_key_values=past_key_values,
            attn_metadata=attn_metadata,
            inputs_embeds=inputs_embeds,
        )

        logits = self.lm_head(hidden_states)
        logits = logits.float()
        return logits
```

In addition to these, the following content needs to be added:

```
class LlamaForCausalLM(nn.Module):
```

(continues on next page)

(continued from previous page)

```
    ...

    # Indicates whether the model supports cudagraph
    # Can be a callable object, receiving forward inputs
    # Dynamically determines if cudagraph is supported
    support_cuda_graph = True

    # Builds model inputs
    # Returns a dictionary, the keys of which must be inputs to forward
    def prepare_inputs_for_generation(
        self,
        past_key_values: List[List[torch.Tensor]],
        inputs_embeds: Optional[torch.Tensor] = None,
        context: StepContext = None,
    ):
        ...

    # Loads weights
    # The model's inputs are key-value pairs of the state dict
    def load_weights(self, weights: Iterable[Tuple[str, torch.Tensor]]):
        ...
```

We have encapsulated many fused operators to simplify the model construction. These operators better support various functions such as tensor parallelism and quantization. We encourage developers to use these ops as much as possible.

```
# Using predefined build_merged_colwise_linear, SiluAndMul, build_rowwise_linear
# Helps us build the model faster and without worrying about tensor concurrency,␣
↪quantization, etc.
class LlamaMLP(nn.Module):

    def __init__(self,
                 config: LlamaConfig,
                 dtype: torch.dtype = None,
                 device: torch.device = None):
        super().__init__()
        quantization_config = getattr(config, 'quantization_config', None)
        # gate up
        self.gate_up_proj = build_merged_colwise_linear(
            config.hidden_size,
            [config.intermediate_size, config.intermediate_size],
            bias=config.mlp_bias,
            dtype=dtype,
            device=device,
            quant_config=quantization_config,
            is_tp=True,
        )

        # silu and mul
        self.act_fn = SiluAndMul(inplace=True)

        # down
        self.down_proj = build_rowwise_linear(config.intermediate_size,
```

(continues on next page)

(continued from previous page)

```
                                                config.hidden_size,
                                                bias=config.mlp_bias,
                                                quant_config=quantization_config,
                                                dtype=dtype,
                                                device=device,
                                                is_tp=True)

    def forward(self, x):
        """forward."""
        gate_up = self.gate_up_proj(x)
        act = self.act_fn(gate_up)
        return self.down_proj(act)
```

Model Registration

To ensure that the developed model implementation can be used normally, we also need to register the model in `lmdeploy/pytorch/models/module_map.py`

```
MODULE_MAP.update({
    'LlamaForCausalLM':
    f'{LMDEPLOY_PYTORCH_MODEL_PATH}.llama.LlamaForCausalLM',
})
```

If you do not wish to modify the model source code, you can also pass a custom module map from the outside, making it easier to integrate into other projects.

```
from lmdeploy import PytorchEngineConfig, pipeline

backend_config = PytorchEngineConfig(custom_module_map='/path/to/custom/module_map.py')
generator = pipeline(model_path, backend_config=backend_config)
```

1.26 Context length extrapolation

Long text extrapolation refers to the ability of LLM to handle data longer than the training text during inference. TurboMind engine now support LlamaDynamicNTKScalingRotaryEmbedding and the implementation is consistent with huggingface.

1.26.1 Usage

You can enable the context length extrapolation abality by modifying the TurbomindEngineConfig. Edit the `session_len` to the expected length and change `rope_scaling_factor` to a number no less than 1.0.

Take `internlm2_5-7b-chat-1m` as an example, which supports a context length of up to **1 million tokens**:

```
from lmdeploy import pipeline, GenerationConfig, TurbomindEngineConfig

backend_config = TurbomindEngineConfig(
        rope_scaling_factor=2.5,
        session_len=1000000,
```

(continues on next page)

(continued from previous page)

```
        max_batch_size=1,
        cache_max_entry_count=0.7,
        tp=4)
pipe = pipeline('internlm/internlm2_5-7b-chat-1m', backend_config=backend_config)
prompt = 'Use a long prompt to replace this sentence'
gen_config = GenerationConfig(top_p=0.8,
                              top_k=40,
                              temperature=0.8,
                              max_new_tokens=1024)
response = pipe(prompt, gen_config=gen_config)
print(response)
```

1.26.2 Evaluation

We use several methods to evaluate the long-context-length inference ability of LMDeploy, including *passkey retrieval*, *needle in a haystack* and computing *perplexity*

Passkey Retrieval

You can try the following code to test how many times LMDeploy can retrieval the special key.

```
import numpy as np
from lmdeploy import pipeline
from lmdeploy import TurbomindEngineConfig
import time

session_len = 1000000
backend_config = TurbomindEngineConfig(
        rope_scaling_factor=2.5,
        session_len=session_len,
        max_batch_size=1,
        cache_max_entry_count=0.7,
        tp=4)
pipe = pipeline('internlm/internlm2_5-7b-chat-1m', backend_config=backend_config)


def passkey_retrieval(session_len, n_round=5):
    # create long context input
    tok = pipe.tokenizer
    task_description = 'There is an important info hidden inside a lot of irrelevant␣
↪text. Find it and memorize them. I will quiz you about the important information there.
↪'
    garbage = 'The grass is green. The sky is blue. The sun is yellow. Here we go. There␣
↪and back again.'

    for _ in range(n_round):
        start = time.perf_counter()
        n_times = (session_len - 1000) // len(tok.encode(garbage))
        n_garbage_prefix = np.random.randint(0, n_times)
        n_garbage_suffix = n_times - n_garbage_prefix
```

(continues on next page)

(continued from previous page)

```
        garbage_prefix = ' '.join([garbage] * n_garbage_prefix)
        garbage_suffix = ' '.join([garbage] * n_garbage_suffix)
        pass_key = np.random.randint(1, 50000)
        information_line = f'The pass key is {pass_key}. Remember it. {pass_key} is the␣
↪pass key.'  # noqa: E501
        final_question = 'What is the pass key? The pass key is'
        lines = [
            task_description,
            garbage_prefix,
            information_line,
            garbage_suffix,
            final_question,
        ]

        # inference
        prompt = ' '.join(lines)
        response = pipe([prompt])
        print(pass_key, response)
        end = time.perf_counter()
        print(f'duration: {end - start} s')

passkey_retrieval(session_len, 5)
```

This test takes approximately 364 seconds per round when conducted on A100-80G GPUs

Needle In A Haystack

OpenCompass offers very useful tools to perform needle-in-a-haystack evaluation. For specific instructions, please refer to the guide.

Perplexity

The following codes demonstrate how to use LMDeploy to calculate perplexity.

```
from transformers import AutoTokenizer
from lmdeploy import TurbomindEngineConfig, pipeline
import numpy as np

# load model and tokenizer
model_repoid_or_path = 'internlm/internlm2_5-7b-chat-1m'
backend_config = TurbomindEngineConfig(
        rope_scaling_factor=2.5,
        session_len=1000000,
        max_batch_size=1,
        cache_max_entry_count=0.7,
        tp=4)
pipe = pipeline(model_repoid_or_path, backend_config=backend_config)
tokenizer = AutoTokenizer.from_pretrained(model_repoid_or_path, trust_remote_code=True)

# get perplexity
text = 'Use a long prompt to replace this sentence'
```

(continues on next page)

(continued from previous page)

```
input_ids = tokenizer.encode(text)
ppl = pipe.get_ppl(input_ids)[0]
print(ppl)
```

1.27 Customized chat template

The effect of the applied chat template can be observed by **setting log level** `INFO`.

LMDeploy supports two methods of adding chat templates:

- One approach is to utilize an existing conversation template by directly configuring a JSON file like the following.

```
{
    "model_name": "your awesome chat template name",
    "system": "<|im_start|>system\n",
    "meta_instruction": "You are a robot developed by LMDeploy.",
    "eosys": "<|im_end|>\n",
    "user": "<|im_start|>user\n",
    "eoh": "<|im_end|>\n",
    "assistant": "<|im_start|>assistant\n",
    "eoa": "<|im_end|>",
    "separator": "\n",
    "capability": "chat",
    "stop_words": ["<|im_end|>"]
}
```

  The new chat template would be applied like this:

```
{system}{meta_instruction}{eosys}{user}{user_content}{eoh}{assistant}{assistant_
↪content}{eoa}{separator}{user}...
```

  When using the CLI tool, you can pass in a custom chat template with `--chat-template`, for example.

```
lmdeploy serve api_server internlm/internlm2_5-7b-chat --chat-template ${JSON_FILE}
```

  You can also pass it in through the interface function, for example.

```
from lmdeploy import ChatTemplateConfig, serve
serve('internlm/internlm2_5-7b-chat',
      chat_template_config=ChatTemplateConfig.from_json('${JSON_FILE}'))
```

- Another approach is to customize a Python chat template class like the existing LMDeploy chat templates. It can be used directly after successful registration. The advantages are a high degree of customization and strong controllability. Below is an example of registering an LMDeploy chat template.

```
from lmdeploy.model import MODELS, BaseChatTemplate


@MODELS.register_module(name='customized_model')
class CustomizedModel(BaseChatTemplate):
    """A customized chat template."""
```

(continues on next page)

(continued from previous page)

```
    def __init__(self,
                 system='<|im_start|>system\n',
                 meta_instruction='You are a robot developed by LMDeploy.',
                 user='<|im_start|>user\n',
                 assistant='<|im_start|>assistant\n',
                 eosys='<|im_end|>\n',
                 eoh='<|im_end|>\n',
                 eoa='<|im_end|>',
                 separator='\n',
                 stop_words=['<|im_end|>', '<|action_end|>']):
        super().__init__(system=system,
                         meta_instruction=meta_instruction,
                         eosys=eosys,
                         user=user,
                         eoh=eoh,
                         assistant=assistant,
                         eoa=eoa,
                         separator=separator,
                         stop_words=stop_words)


from lmdeploy import ChatTemplateConfig, pipeline

messages = [{'role': 'user', 'content': 'who are you?'}]
pipe = pipeline('internlm/internlm2_5-7b-chat',
                chat_template_config=ChatTemplateConfig('customized_model'))
for response in pipe.stream_infer(messages):
    print(response.text, end='')
```

In this example, we register a LMDeploy chat template that sets the model to be created by LMDeploy, so when the user asks who the model is, the model will answer that it was created by LMDeploy.

1.28 How to debug Turbomind

Turbomind is implemented in C++, which is not as easy to debug as Python. This document provides basic methods for debugging Turbomind.

1.28.1 Prerequisite

First, complete the local compilation according to the commands in *Install from source*.

1.28.2 Configure Python debug environment

Since many large companies currently use Centos 7 for online production environments, we will use Centos 7 as an example to illustrate the process.

Obtain `glibc` and `python3` versions

```
rpm -qa | grep glibc
rpm -qa | grep python3
```

The result should be similar to this:

```
[username@hostname workdir]# rpm -qa | grep glibc
glibc-2.17-325.el7_9.x86_64
glibc-common-2.17-325.el7_9.x86_64
glibc-headers-2.17-325.el7_9.x86_64
glibc-devel-2.17-325.el7_9.x86_64

[username@hostname workdir]# rpm -qa | grep python3
python3-pip-9.0.3-8.el7.noarch
python3-rpm-macros-3-34.el7.noarch
python3-rpm-generators-6-2.el7.noarch
python3-setuptools-39.2.0-10.el7.noarch
python3-3.6.8-21.el7_9.x86_64
python3-devel-3.6.8-21.el7_9.x86_64
python3.6.4-sre-1.el6.x86_64
```

Based on the information above, we can see that the version of `glibc` is `2.17-325.el7_9.x86_64` and the version of `python3` is `3.6.8-21.el7_9.x86_64`.

Download and install `debuginfo` library

Download `glibc-debuginfo-common-2.17-325.el7.x86_64.rpm`, `glibc-debuginfo-2.17-325.el7.x86_64.rpm`, and `python3-debuginfo-3.6.8-21.el7.x86_64.rpm` from http://debuginfo.centos.org/7/x86_64.

```
rpm -ivh glibc-debuginfo-common-2.17-325.el7.x86_64.rpm
rpm -ivh glibc-debuginfo-2.17-325.el7.x86_64.rpm
rpm -ivh python3-debuginfo-3.6.8-21.el7.x86_64.rpm
```

Upgrade GDB

```
sudo yum install devtoolset-10 -y
echo "source scl_source enable devtoolset-10" >> ~/.bashrc
source ~/.bashrc
```

Verification

```
gdb python3
```

The output should be similar to this:

```
[username@hostname workdir]# gdb python3
GNU gdb (GDB) Red Hat Enterprise Linux 9.2-10.el7
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
   <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from python3...
(gdb)
```

If it shows `Reading symbols from python3`, the configuration has been successful.

For other operating systems, please refer to DebuggingWithGdb.

1.28.3 Set up symbolic links

After setting up symbolic links, there is no need to install it locally with `pip` every time.

```
# Change directory to lmdeploy, e.g.
cd /workdir/lmdeploy

# Since it has been built in the build directory
# Link the lib directory
cd lmdeploy && ln -s ../build/lib . && cd ..
# (Optional) Link compile_commands.json for clangd index
ln -s build/compile_commands.json .
```

1.28.4 Start debugging

```
# Use gdb to start the API server with Llama-2-13b-chat-hf, e.g.
gdb --args python3 -m lmdeploy serve api_server /workdir/Llama-2-13b-chat-hf

# Set directories in gdb
Reading symbols from python3...
(gdb) set directories /workdir/lmdeploy

# Set a breakpoint using the relative path, e.g.
(gdb) b src/turbomind/models/llama/BlockManager.cc:104

# When it shows
# ```
# No source file named src/turbomind/models/llama/BlockManager.cc.
# Make breakpoint pending on future shared library load? (y or [n])
# ```
# Just type `y` and press enter

# Run
(gdb) r

# (Optional) Use https://github.com/InternLM/lmdeploy/blob/main/benchmark/profile_
↪restful_api.py to send a request

python3 profile_restful_api.py --backend lmdeploy --dataset-path /workdir/ShareGPT_V3_
↪unfiltered_cleaned_split.json --num_prompts 1
```

1.28.5 Using GDB

Refer to GDB Execution Commands and happy debugging.

1.29 Structured output

Structured output, also known as guided decoding, forces the model to generate text that exactly matches a user-supplied JSON schema, grammar, or regex. Both the PyTorch and Turbomind backends now support structured (schema-constrained) generation. Below are examples for the pipeline API and the API server.

1.29.1 pipeline

```
from lmdeploy import pipeline
from lmdeploy.messages import GenerationConfig, PytorchEngineConfig

model = 'internlm/internlm2-chat-1_8b'
guide = {
    'type': 'object',
    'properties': {
        'name': {
            'type': 'string'
```

(continues on next page)

(continued from previous page)

```
        },
        'skills': {
            'type': 'array',
            'items': {
                'type': 'string',
                'maxLength': 10
            },
            'minItems': 3
        },
        'work history': {
            'type': 'array',
            'items': {
                'type': 'object',
                'properties': {
                    'company': {
                        'type': 'string'
                    },
                    'duration': {
                        'type': 'string'
                    }
                },
                'required': ['company']
            }
        }
    },
    'required': ['name', 'skills', 'work history']
}
pipe = pipeline(model, backend_config=PytorchEngineConfig(), log_level='INFO')
gen_config = GenerationConfig(
    response_format=dict(type='json_schema', json_schema=dict(name='test',␣
↪schema=guide)))
response = pipe(['Make a self introduction please.'], gen_config=gen_config)
print(response)
```

1.29.2 api_server

Firstly, start the api_server service for the InternLM2 model.

```
lmdeploy serve api_server internlm/internlm2-chat-1_8b --backend pytorch
```

The client can test using OpenAI's python package: The output result is a response in JSON format.

```
from openai import OpenAI
guide = {
    'type': 'object',
    'properties': {
        'name': {
            'type': 'string'
        },
        'skills': {
            'type': 'array',
```

(continues on next page)

(continued from previous page)

```
            'items': {
                'type': 'string',
                'maxLength': 10
            },
            'minItems': 3
        },
        'work history': {
            'type': 'array',
            'items': {
                'type': 'object',
                'properties': {
                    'company': {
                        'type': 'string'
                    },
                    'duration': {
                        'type': 'string'
                    }
                },
                'required': ['company']
            }
        }
    },
    'required': ['name', 'skills', 'work history']
}
response_format=dict(type='json_schema',  json_schema=dict(name='test',schema=guide))
messages = [{'role': 'user', 'content': 'Make a self-introduction please.'}]
client = OpenAI(api_key='YOUR_API_KEY', base_url='http://0.0.0.0:23333/v1')
model_name = client.models.list().data[0].id
response = client.chat.completions.create(
    model=model_name,
    messages=messages,
    temperature=0.8,
    response_format=response_format,
    top_p=0.8)
print(response)
```

1.30 PyTorchEngine Multi-Node Deployment Guide

To support larger-scale model deployment requirements, PyTorchEngine provides multi-node deployment support. Below are the detailed steps for deploying a `tp=16` model across two 8-GPU nodes.

1.30.1 1. Create Docker Containers (Optional)

To ensure consistency across the cluster environment, it is recommended to use Docker to set up the cluster. Create containers on each node as follows:

```
docker run -it \
    --network host \
    -v $MODEL_PATH:$CONTAINER_MODEL_PATH \
    openmmlab/lmdeploy:latest
```

[!IMPORTANT] Ensure that the model is placed in the same directory on all node containers.

1.30.2 2. Set Up the Cluster Using Ray

2.1 Start the Head Node

Select one node as the **head node** and run the following command in its container:

```
ray start --head --port=$DRIVER_PORT
```

2.2 Join the Cluster

On the other nodes, use the following command in their containers to join the cluster created by the head node:

```
ray start --address=$DRIVER_NODE_ADDR:$DRIVER_PORT
```

run `ray status` on head node to check the cluster.

[!IMPORTANT] Ensure that `DRIVER_NODE_ADDR` is the address of the head node and `DRIVER_PORT` matches the port number used during the head node initialization.

1.30.3 3. Use LMDeploy Interfaces

In the head node's container, you can use all functionalities of PyTorchEngine as usual.

3.1 Start the Server

```
lmdeploy serve api_server \
    $CONTAINER_MODEL_PATH \
    --backend pytorch \
    --tp 16
```

3.2 Use the Pipeline

```
from lmdeploy import pipeline, PytorchEngineConfig

if __name__ == '__main__':
    model_path = '/path/to/model'
    backend_config = PytorchEngineConfig(tp=16)
    with pipeline(model_path, backend_config=backend_config) as pipe:
        outputs = pipe('Hakuna Matata')
```

> [!NOTE] PyTorchEngine will automatically choose the appropriate launch method (single-node/multi-node) based on the `tp` parameter and the number of devices available in the cluster. If you want to enforce the use of the Ray cluster, you can configure `distributed_executor_backend='ray'` in `PytorchEngineConfig` or use the environment variable `LMDEPLOY_EXECUTOR_BACKEND=ray`.

By following the steps above, you can successfully deploy PyTorchEngine in a multi-node environment and leverage the Ray cluster for distributed computing.

> [!WARNING] To achieve better performance, we recommend users to configure a higher-quality network environment (such as using InfiniBand) to improve engine efficiency.

1.31 PyTorchEngine Profiling

We provide multiple profiler to analysis the performance of PyTorchEngine.

1.31.1 PyTorch Profiler

We have integrated the PyTorch Profiler. You can enable it by setting environment variables when launching the pipeline or API server:

```
# enable profile cpu
export LMDEPLOY_PROFILE_CPU=1
# enable profile cuda
export LMDEPLOY_PROFILE_CUDA=1
# profile would start after 3 seconds
export LMDEPLOY_PROFILE_DELAY=3
# profile 10 seconds
export LMDEPLOY_PROFILE_DURATION=10
# prefix path to save profile files
export LMDEPLOY_PROFILE_OUT_PREFIX="/path/to/save/profile_"
```

After the program exits, the profiling data will be saved to the path specified by `LMDEPLOY_PROFILE_OUT_PREFIX` for performance analysis.

1.31.2 Nsight System

We also support using Nsight System to profile NVIDIA devices.

Single GPU

For single-GPU scenarios, simply use `nsys profile`:

```
nsys profile python your_script.py
```

Multi-GPU

When using multi-GPU solutions like DP/TP/EP, set the following environment variables:

```
# enable nsight system
export LMDEPLOY_RAY_NSYS_ENABLE=1
# prefix path to save profile files
export LMDEPLOY_RAY_NSYS_OUT_PREFIX="/path/to/save/profile_"
```

Then launch the script or API server as usual (Do **NOT** use nsys profile here).

The profiling results will be saved under `LMDEPLOY_RAY_NSYS_OUT_PREFIX`. If `LMDEPLOY_RAY_NSYS_OUT_PREFIX` is not configured, you can find the results in `/tmp/ray/session_xxx/nsight`.

1.31.3 Ray timeline

We use `ray` to support multi-device deployment. You can get the ray timeline with the environments below.

```
export LMDEPLOY_RAY_TIMELINE_ENABLE=1
export LMDEPLOY_RAY_TIMELINE_OUT_PATH="/path/to/save/timeline.json"
```

1.32 Production Metrics

LMDeploy exposes a set of metrics via Prometheus, and provides visualization via Grafana.

1.32.1 Setup Guide

This section describes how to set up the monitoring stack (Prometheus + Grafana) provided in the `lmdeploy/monitoring` directory.

1.32.2 Prerequisites

- Docker and Docker Compose installed
- LMDeploy server running with metrics system enabled

1.32.3 Usage (DP = 1)

1. **Start your LMDeploy server with metrics enabled**

```
lmdeploy serve api_server Qwen/Qwen2.5-7B-Instruct --enable-metrics
```

Replace the model path according to your needs. By default, the metrics endpoint will be available at `http://<lmdeploy_server_host>:23333/metrics`.

2. **Navigate to the monitoring directory**

```
cd lmdeploy/monitoring
```

3. **Start the monitoring stack**

```
docker compose up
```

This command will start Prometheus and Grafana in the background.

4. **Access the monitoring interfaces**

- Prometheus: Open your web browser and go to http://localhost:9090.
- Grafana: Open your web browser and go to http://localhost:3000.

5. **Log in to Grafana**

- Default Username: `admin`
- Default Password: `admin` You will be prompted to change the password upon your first login.

6. **View the Dashboard**

The LMDeploy dashboard is pre-configured and should be available automatically.

1.32.4 Usage (DP > 1)

1. **Start your LMDeploy server with metrics enabled**

As an example, we use the model `Qwen/Qwen2.5-7B-Instruct` with `DP=2, TP=2`. Start the service as follows:

```
# Proxy server
lmdeploy serve proxy --server-port 8000 --routing-strategy 'min_expected_latency' --
↪serving-strategy Hybrid --log-level INFO

# API server
LMDEPLOY_DP_MASTER_ADDR=127.0.0.1 \
LMDEPLOY_DP_MASTER_PORT=29555 \
lmdeploy serve api_server \
    Qwen/Qwen2.5-7B-Instruct \
    --backend pytorch \
    --tp 2 \
```

(continues on next page)

(continued from previous page)

```
    --dp 2 \
    --proxy-url http://0.0.0.0:8000 \
    --nnodes 1 \
    --node-rank 0 \
    --enable-metrics
```

You should be able to see multiple API servers added to the proxy server list. Details can be found in `lmdeploy/serve/proxy/proxy_config.json`.

For example, you may have the following API servers:

```
http://$host_ip:$api_server_port1

http://$host_ip:$api_server_port2
```

2. **Modify the Prometheus configuration**

When `DP > 1`, LMDeploy will launch one API server for each DP rank. If you want to monitor a specific API server, e.g. `http://$host_ip:$api_server_port1`, modify the configuration file `lmdeploy/monitoring/prometheus.yaml` as follows.

> Note that you should use the actual host machine IP instead of `127.0.0.1` here, since LMDeploy starts the API server using the actual host IP when `DP > 1`

```
global:
  scrape_interval: 5s
  evaluation_interval: 30s

scrape_configs:
  - job_name: lmdeploy
    static_configs:
      - targets:
          - '$host_ip:$api_server_port1' # <= Modify this
```

3. **Navigate to the monitoring folder and perform the same steps as described above**

1.32.5 Troubleshooting

1. **Port conflicts**

Check if any services are occupying ports `23333` (LMDeploy server port), `9090` (Prometheus port), or `3000` (Grafana port). You can either stop the conflicting running ports or modify the config files as follows:

- Modify LMDeploy server port for Prometheus scrape

In `lmdeploy/monitoring/prometheus.yaml`

```
global:
  scrape_interval: 5s
  evaluation_interval: 30s

scrape_configs:
  - job_name: lmdeploy
    static_configs:
      - targets:
```

(continues on next page)

(continued from previous page)

```
          - '127.0.0.1:23333' # <= Modify this LMDeploy server port 23333, need to match␣
↪the running server port
```

- Modify Prometheus port

In `lmdeploy/monitoring/grafana/datasources/datasource.yaml`

```
apiVersion: 1
datasources:
  - name: Prometheus
    type: prometheus
    access: proxy
    url: http://localhost:9090 # <= Modify this Prometheus interface port 9090
    isDefault: true
    editable: false
```

- Modify Grafana port:

In `lmdeploy/monitoring/docker-compose.yaml`, for example, change the port to `3090`

Option 1: Add `GF_SERVER_HTTP_PORT` to the environment section.

```
  environment:
- GF_AUTH_ANONYMOUS_ENABLED=true
- GF_SERVER_HTTP_PORT=3090  # <= Add this line
```

Option 2: Use port mapping.

```
grafana:
  image: grafana/grafana:latest
  container_name: grafana
  ports:
  - "3090:3000"  # <= Host:Container port mapping
```

2. **No data on the dashboard**

- Create traffic

Try to send some requests to the LMDeploy server to create certain traffic

```
python3 benchmark/profile_restful_api.py --backend lmdeploy --num-prompts 5000 --dataset-
↪path ShareGPT_V3_unfiltered_cleaned_split.json
```

After refreshing, you should be able to see data on the dashboard.

1.33 Context Parallel

When the memory on a single GPU is insufficient to deploy a model, it is often deployed using tensor parallelism (TP), which generally requires `num_key_value_heads` to be divisible by `TP`. If you want to deploy with `TP` > `num_key_value_heads`, the kv-heads should be duplicated to meet the divisibility requirement. However, this has two disadvantages:

1. The amount of available kv_cache is halved, which reducing the maximum supported session length.
2. The maximum inference batch size is reduced, leading to lower throughput.

To address this issue, the TurboMind inference backend supports setting `attn_dp_size`, which avoids creating copies of kv-heads, but this introduces data imbalance. To eliminate data imbalance, TurboMind supports sequence parallelism, which allowing kv_cache to be stored interleaved on different cp_ranks. See the example below:

```
cp_rank=2, prompt_len=5, generation_len=4
kv_cache stored on cp_rank0: 0, 2, 4, 6, 8
kv_cache stored on cp_rank1: 1, 3, 5, 7
```

1.33.1 Usage

Taking Intern-S1 / Qwen3-235B-A22B as an example, their `num_key_value_heads` is 4. If you want to deploy with `TP=8` and avoid duplication of kv_cache, you can deploy in the following way:

```
lmdeploy serve api_server internlm/Intern-S1 --tp 8 --cp 2

lmdeploy serve api_server Qwen/Qwen3-235B-A22B --tp 8 --cp 2
```

1.34 Speculative Decoding

Speculative decoding is an optimization technique that introcude a lightweight draft model to propose multiple next tokens and then, the main model verify and choose the longest matched tokens in a forward pass. Compared with standard auto-regressive decoding, this methold lets the system generate multiple tokens at once.

[!NOTE] This is an experimental feature in lmdeploy.

1.34.1 Examples

Here are some examples.

Eagle 3

Prepare

Install flash-atten3

```
git clone --depth=1 https://github.com/Dao-AILab/flash-attention.git
cd flash-attention/hopper
python setup.py install
```

pipeline

```
from lmdeploy import PytorchEngineConfig, pipeline
from lmdeploy.messages import SpeculativeConfig


if __name__ == '__main__':

```

(continues on next page)

(continued from previous page)

```
    model_path = 'meta-llama/Llama-3.1-8B-Instruct'
    spec_cfg = SpeculativeConfig(
        method='eagle3',
        num_speculative_tokens=3,
        model='yuhuili/EAGLE3-LLaMA3.1-Instruct-8B',
    )
    pipe = pipeline(model_path, backend_config=PytorchEngineConfig(max_batch_size=128),␣
↪speculative_config=spec_cfg)
    response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
    print(response)
```

serving

```
lmdeploy serve api_server \
meta-llama/Llama-3.1-8B-Instruct \
--backend pytorch \
--server-port 24545 \
--speculative-draft-model yuhuili/EAGLE3-LLaMA3.1-Instruct-8B \
--speculative-algorithm eagle3 \
--speculative-num-draft-tokens 3 \
--max-batch-size 128 \
--enable-metrics
```

Deepseek MTP

Prepare

Install FlashMLA

```
git clone https://github.com/deepseek-ai/FlashMLA.git flash-mla
cd flash-mla
git submodule update --init --recursive
pip install -v .
```

pipeline

```
from lmdeploy import PytorchEngineConfig, pipeline
from lmdeploy.messages import SpeculativeConfig


if __name__ == '__main__':

    model_path = 'deepseek-ai/DeepSeek-V3'
    spec_cfg = SpeculativeConfig(
        method='deepseek_mtp',
        num_speculative_tokens=3,
```

(continues on next page)

(continued from previous page)

```
    )
    pipe = pipeline(model_path,
                    backend_config=PytorchEngineConfig(tp=16, max_batch_size=128),
                    speculative_config=spec_cfg)
    response = pipe(['Hi, pls intro yourself', 'Shanghai is'])
    print(response)
```

serving

```
lmdeploy serve api_server \
deepseek-ai/DeepSeek-V3 \
--backend pytorch \
--server-port 24545 \
--tp 16 \
--speculative-algorithm deepseek_mtp \
--speculative-num-draft-tokens 3 \
--max-batch-size 128 \
--enable-metrics
```

1.35 Update Weights

LMDeploy supports update model weights online for scenes such as RL training. Here are the steps to do so.

1.35.1 Step 1: Launch server

For pytorch backend you have to add `--distributed-executor-backend ray`.

```
lmdeploy serve api_server internlm/internlm2_5-7b-chat --server-port 23333 --distributed-
↪executor-backend ray # for pytorch backend
```

1.35.2 Step 2: Offloads weights & kv cache

Before update model weights, the server should offloads weights and kv cache.

```
from lmdeploy.utils import serialize_state_dict
import requests

BASE_URL = 'http://0.0.0.0:23333'
api_key = 'sk-xxx'

headers = {
                "Content-Type": "application/json",
                "Authorization": f"Bearer {api_key}",
            }
```

(continues on next page)

(continued from previous page)

```
# offloads weights and kv cache with level=2
response = requests.post(f"{BASE_URL}/sleep", headers=headers, params=dict(tags=['weights
↪', 'kv_cache'], level=2))
assert response.status_code == 200, response.status_code

# wake up weights, the server is ready for update weights
response = requests.post(f"{BASE_URL}/wakeup", headers=headers, params=dict(tags=[
↪'weights']))
assert response.status_code == 200, response.status_code
```

1.35.3 Step 3: Update weights

Split model weights into multi segments and update through `update_weights` endpoint.

```
segmented_state_dict: List[Dict[str, torch.Tensor]] = ...
num_segment = len(segmented_state_dict)
for seg_idx in range(num_segment):
    serialized_data = serialize_state_dict(segmented_state_dict[seg_idx])
    data = dict(serialized_named_tensors=serialized_data, finished=seg_idx == num_
↪segment-1)
    response = requests.post(f"{BASE_URL}/update_weights", headers=headers, json=data)
    assert response.status_code == 200, f"response.status_code = {response.status_code}"
```

Note: For pytorch backend, lmdeploy also supports flattened bucket tensors:

```
from lmdeploy.utils import serialize_state_dict, FlattenedTensorBucket,␣
↪FlattenedTensorMetadata

segmented_state_dict: List[Dict[str, torch.Tensor]] = ...
num_segment = len(segmented_state_dict)
for seg_idx in range(num_segment):
    named_tensors = list(segmented_state_dict[seg_idx].items())
    bucket = FlattenedTensorBucket(named_tensors=named_tensors)
    metadata = bucket.get_metadata()
    flattened_tensor_data = dict(flattened_tensor=bucket.get_flattened_tensor(),␣
↪metadata=metadata)
    serialized_data = serialize_state_dict(flattened_tensor_data)
    data = dict(serialized_named_tensors=serialized_data, finished=seg_idx == num_
↪segment-1, load_format='flattened_bucket')
    response = requests.post(f"{BASE_URL}/update_weights", headers=headers, json=data)
    assert response.status_code == 200, f"response.status_code = {response.status_code}"
```

1.35.4 Step 4: Wakeup server

After update model weights, the server should onloads kv cache and provide serving again with the new updated weights.

```
response = requests.post(f"{BASE_URL}/wakeup", headers=headers, params=dict(tags=['kv_
↪cache']))
assert response.status_code == 200, response.status_code
```

1.36 Inference pipeline

1.36.1 Pipeline

lmdeploy.**pipeline**(*model_path*, *backend_config=None*, *chat_template_config=None*, *log_level='WARNING'*, *max_log_len=None*, *trust_remote_code=False*, *speculative_config=None*, ***kwargs*)

Create a pipeline for inference.

Parameters

- **model_path** (str) – the path of a model. It could be one of the following options:
  - i) A local directory path of a turbomind model which is converted by `lmdeploy convert` command or download from ii) and iii).
  - ii) The model_id of a lmdeploy-quantized model hosted inside a model repo on huggingface.co, such as `InternLM/internlm-chat-20b-4bit`, `lmdeploy/llama2-chat-70b-4bit`, etc.
  - iii) The model_id of a model hosted inside a model repo on huggingface.co, such as `internlm/internlm-chat-7b`, `Qwen/Qwen-7B-Chat`, `baichuan-inc/Baichuan2-7B-Chat` and so on.
- **backend_config** (Union[*TurbomindEngineConfig*, *PytorchEngineConfig*, None]) – backend config instance. Default to None.
- **chat_template_config** (Optional[*ChatTemplateConfig*]) – chat template configuration. Default to None.
- **log_level** (str) – set log level whose value among [CRITICAL, ERROR, WARNING, INFO, DEBUG]
- **max_log_len** (Optional[int]) – Max number of prompt characters or prompt tokens being printed in log.
- **trust_remote_code** (bool) – whether to trust remote code from model repositories.
- **speculative_config** (Optional[SpeculativeConfig]) – speculative decoding configuration.
- ****kwargs** – additional keyword arguments passed to the pipeline.

Returns

a pipeline instance for inference.

Return type

Pipeline

Examples

```
# LLM
import lmdeploy
pipe = lmdeploy.pipeline('internlm/internlm-chat-7b')
response = pipe(['hi','say this is a test'])
print(response)

# VLM
from lmdeploy.vl import load_image
from lmdeploy import pipeline, TurbomindEngineConfig, ChatTemplateConfig
pipe = pipeline('liuhaotian/llava-v1.5-7b',
                backend_config=TurbomindEngineConfig(session_len=8192),
                chat_template_config=ChatTemplateConfig(model_name='vicuna'))
im = load_image('https://raw.githubusercontent.com/open-mmlab/mmdeploy/main/demo/
↪resources/human-pose.jpg')
response = pipe([('describe this image', [im])])
print(response)
```

class lmdeploy.**Pipeline**(*model_path*, *backend_config=None*, *chat_template_config=None*, *log_level='WARNING'*, *max_log_len=None*, *trust_remote_code=False*, *speculative_config=None*, ***kwargs*)

Bases: object

Pipeline - User-facing API layer for inference.

__init__(*model_path*, *backend_config=None*, *chat_template_config=None*, *log_level='WARNING'*, *max_log_len=None*, *trust_remote_code=False*, *speculative_config=None*, ***kwargs*)

Initialize Pipeline.

Parameters

- **model_path** (str) – Path to the model.
- **backend_config** (Union[*TurbomindEngineConfig*, *PytorchEngineConfig*, None]) – Backend configuration.
- **chat_template_config** (Optional[*ChatTemplateConfig*]) – Chat template configuration.
- **log_level** (str) – Log level.
- **max_log_len** (Optional[int]) – Max number of prompt characters or prompt tokens being printed in log.
- **trust_remote_code** (bool) – whether to trust remote code from model repositories.
- **speculative_config** (Optional[SpeculativeConfig]) – Speculative decoding configuration.
- ****kwargs** – Additional keyword arguments.

infer(*prompts*, *gen_config=None*, *do_preprocess=True*, *adapter_name=None*, *use_tqdm=False*, ***kwargs*)

Inference prompts.

Parameters

- **prompts** (list[str] | str | list[dict] | list[list[dict]] | tuple | list[tuple]) – Prompts for inference. It can be a single prompt, a list of prompts, a list of tuples, or a tuple. tuple can be (prompt, image or [images]) or (image or [images], prompt).

- **gen_config** (Union[*GenerationConfig*, list[*GenerationConfig*], None]) – Generation configuration(s).
- **do_preprocess** (bool) – Whether to pre-process messages.
- **adapter_name** (Optional[str]) – Adapter name.
- **use_tqdm** (bool) – Whether to use progress bar.
- ****kwargs** – Additional keyword arguments.

Returns
A single response or a list of responses.

Return type
Response | list[Response]

stream_infer(*prompts*, *sessions=None*, *gen_config=None*, *do_preprocess=True*, *adapter_name=None*, *stream_response=True*, ***kwargs*)

Stream inference.

Parameters

- **prompts** (list[str] | str | list[dict] | list[list[dict]] | tuple | list[tuple]) – Prompts to inference. It can be a single prompt, a list of prompts, a list of tuples, or a tuple. tuple can be (prompt, image or [images]) or (image or [images], prompt).
- **sessions** (Union[Session, list[Session], None]) – Sessions. Each of which corresponds to a prompt.
- **gen_config** (Union[*GenerationConfig*, list[*GenerationConfig*], None]) – Generation configuration(s).
- **do_preprocess** (bool) – Whether to pre-process messages.
- **adapter_name** (Optional[str]) – Adapter name.
- **stream_response** (bool) – Whether to stream the response. If True, the generator will stream the response. Otherwise, the generator will run until finish and return the final response. This argument is introduced to support the streaming and non-streaming modes of Pipeline.chat.
- ****kwargs** – Additional keyword arguments.

Returns
A generator that yields the output (i.e. instance of class Response) of the inference.

Return type
Iterator

chat(*prompt*, *session=None*, *gen_config=None*, *stream_response=False*, *adapter_name=None*, ***kwargs*)

Chat.

Parameters

- **prompt** (str | tuple[str, Image | list[Image]]) – prompt string or a tuple of (prompt, image or [images]).
- **session** – the chat session.
- **gen_config** (Optional[*GenerationConfig*]) – an instance of GenerationConfig. Default to None.
- **stream_response** – whether to stream the response.

- **adapter_name** – adapter name.
- ****kwargs** – additional keyword arguments.

Returns
the updated session, or a streaming iterator if stream_response is True.

Return type
Session | Iterator

`get_ppl`(*input_ids*)

Get perplexity scores given a list of input tokens that have to be of the same length.

Parameters
input_ids (list[int] | list[list[int]]) – the batch of input token ids.

Returns
A list of perplexity scores.

Return type
list[float]

1.36.2 Config

`class` lmdeploy.**PytorchEngineConfig**(*dtype='auto'*, *tp=1*, *dp=1*, *dp_rank=0*, *ep=1*, *session_len=None*, *max_batch_size=None*, *attn_tp_size=None*, *mlp_tp_size=None*, *moe_tp_size=None*, *cache_max_entry_count=0.8*, *prefill_interval=16*, *block_size=64*, *kernel_block_size=-1*, *num_cpu_blocks=0*, *num_gpu_blocks=0*, *adapters=None*, *max_prefill_token_num=8192*, *thread_safe=False*, *enable_prefix_caching=False*, *device_type='cuda'*, *eager_mode=False*, *custom_module_map=None*, *download_dir=None*, *revision=None*, *quant_policy=QuantPolicy.NONE*, *distributed_executor_backend=None*, *empty_init=False*, *enable_microbatch=False*, *enable_eplb=False*, *enable_mp_engine=False*, *mp_engine_backend='mp'*, *model_format=None*, *enable_metrics=True*, *hf_overrides=None*, *disable_vision_encoder=False*, *logprobs_mode=None*, *enable_return_routed_experts=False*, *enable_transfer_obj_ref=False*, *dllm_block_length=None*, *dllm_unmasking_strategy='low_confidence_dynamic'*, *dllm_denoising_steps=None*, *dllm_confidence_threshold=0.85*, *role=EngineRole.Hybrid*, *migration_backend=MigrationBackend.DLSlime*)

PyTorch Engine Config.

Parameters

- **dtype** (str) – data type for model weights and activations. It can be one of the following values, [‘auto’, ‘float16’, ‘bfloat16’] The *auto* option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models.
- **tp** (int) – Tensor Parallelism. default 1.
- **dp** (int) – Data Parallelism. default 1.
- **dp_rank** (int) – rank of dp.
- **ep** (int) – Expert Parallelism. default 1.

- **session_len** (Optional[int]) – Max session length. Default None.
- **max_batch_size** (Optional[int]) – Max batch size. If it is not specified, the engine will automatically set it according to the device
- **attn_tp_size** (Optional[int]) – tp size for attention, only works for dp>1
- **mlp_tp_size** (Optional[int]) – tp size for mlp, only works for dp>1
- **moe_tp_size** (Optional[int]) – tp size for moe, only works for dp>1
- **cache_max_entry_count** (float) – the percentage of gpu memory occupied by the k/v cache. For lmdeploy versions greater than *v0.2.1*, it defaults to 0.8, signifying the percentage of FREE GPU memory to be reserved for the k/v cache
- **prefill_interval** (int) – Interval to perform prefill, Default 16.
- **block_size** (int) – paging cache block size, default 64.
- **num_cpu_blocks** (int) – Num cpu blocks. If num is 0, cache would be allocate according to current environment.
- **num_gpu_blocks** (int) – Num gpu blocks. If num is 0, cache would be allocate according to current environment.
- **adapters** (Optional[dict[str, str]]) – The path configs to lora adapters.
- **max_prefill_token_num** (int) – tokens per iteration.
- **thread_safe** (bool) – thread safe engine instance.
- **enable_prefix_caching** (bool) – Enable token match and sharing caches.
- **device_type** (str) – The inference device type, options [‘cuda’]
- **eager_mode** (bool) – Enable “eager” mode or not
- **custom_module_map** (Optional[dict[str, str]]) – nn module map customized by users. Once provided, the original nn modules of the model will be substituted by the mapping ones
- **download_dir** (Optional[str]) – Directory to download and load the weights, default to the default cache directory of huggingface.
- **revision** (Optional[str]) – The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- **quant_policy** (QuantPolicy) – default to 0. When k/v is quantized into 4 or 8 bit, set it to 4 or 8, respectively
- **distributed_executor_backend** (Optional[str]) – backend of distributed backend, options: [‘uni’, ‘mp’, ‘ray’]
- **empty_init** (bool) – Whether to load the model weights, you should set it to True if you want to update weights after create the pipeline
- **enable_microbatch** (bool) – enable microbatch for specified model
- **enable_eplb** (bool) – enable eplb for specified model
- **enable_metrics** (bool) – enable metrics system
- **role** (EngineRole) – role of engin, options: [‘Hybrid’, ‘Prefill’, ‘Decode’]. Default to *EngineRole.Hybrid*.
- **migration_backend** (MigrationBackend) – migration backend. options: [‘DLSlime’]. Default to *MigrationBackend.DLSlime*.

- **enable_mp_engine** (bool) – run engine in multi-process mode.
- **mp_engine_backend** (str) – backend of mp engine, options: ['mp', 'ray']. Default to *mp*.
- **model_format** (Optional[str]) – weight quantization policy, options: ['fp8'].
- **hf_overrides** (Optional[dict[str, Any]]) – Huggingface overrides for the model. It can be used to override the default config of the model,
- **disable_vision_encoder** (bool) – Whether to disable loading vision encoder. Default to False.
- **logprobs_mode** (Optional[str]) – The mode of logprob, options: ['raw_logits', 'raw_logprobs']
- **dllm_block_length** (Optional[int]) – Block size of block diffusion model.
- **dllm_unmasking_strategy** (str) – Dllm unmasking strategy, options: ['low_confidence_dynamic', 'low_confidence_static', 'sequential'].
- **dllm_denoising_steps** (Optional[int]) – Dllm denoising steps.
- **dllm_confidence_threshold** (float) – dllm unmasking threshold for dynamic unmasking.

class lmdeploy.**TurbomindEngineConfig**(**args*, ***kwargs*)

TurboMind Engine config.

Parameters

- **dtype** – data type for model weights and activations. It can be one of the following values, ['auto', 'float16', 'bfloat16'] The *auto* option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models.
- **model_format** – the layout of the deployed model. It can be one of the following values [hf, awq, gptq, compressed-tensors, fp8, mxfp4]. *hf* means a Hugging Face model (.bin, .safetensors), *awq* and *gptq* mean grouped 4-bit weight-only checkpoints, *compressed-tensors* means pack-quantized grouped int4 checkpoints and is usually auto-detected from the input model config, *fp8* means blocked fp8 checkpoints, and *mxfp4* means MXFP4 expert weights. If it is not specified, i.e. None, it will be extracted from the input model
- **tp** – the number of GPU cards used in tensor parallelism, default to 1
- **session_len** – the max session length of a sequence, default to None
- **max_batch_size** – the max batch size during inference. If it is not specified, the engine will automatically set it according to the device
- **cache_max_entry_count** – the percentage of gpu memory occupied by the k/v cache. For versions of lmdeploy between *v0.2.0* and *v0.2.1*, it defaults to 0.5, depicting the percentage of TOTAL GPU memory to be allocated to the k/v cache. For lmdeploy versions greater than *v0.2.1*, it defaults to 0.8, signifying the percentage of FREE GPU memory to be reserved for the k/v cache. When it's an integer > 0, it represents the total number of k/v blocks.
- **cache_chunk_size** – The policy to apply for KV block from the block manager, default to -1.
- **cache_block_seq_len** – the length of the token sequence in a k/v block, default to 64
- **enable_prefix_caching** – enable cache prompts for block reuse, default to False
- **quant_policy** – default to 0. When k/v is quantized into 4 or 8 bit, set it to 4 or 8, respectively

- **rope_scaling_factor** – scaling factor used for dynamic ntk, default to 0. TurboMind follows the implementation of transformer LlamaAttention
- **use_logn_attn** – whether or not to use log attn: default to False
- **download_dir** – Directory to download and load the weights, default to the default cache directory of huggingface.
- **revision** – The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- **max_prefill_token_num** – the number of tokens each iteration during prefill, default to 8192
- **num_tokens_per_iter** – the number of tokens processed in each forward pass. Working with *max_prefill_iters* enables the "Dynamic SplitFuse"-like scheduling
- **max_prefill_iters** – the max number of forward pass during prefill stage
- **async** – enable async execution, default to 1 (enabled)
- **devices** – the used devices
- **empty_init** – Whether to load the model weights, you should set it to True if you want to update weights after create the pipeline
- **hf_overrides** – Huggingface overrides for the model. It can be used to override the default config of the model
- **enable_metrics** – enable metrics system

class lmdeploy.**GenerationConfig**(*n=1*, *max_new_tokens=512*, *do_sample=False*, *top_p=1.0*, *top_k=50*, *min_p=0.0*, *temperature=0.8*, *repetition_penalty=1.0*, *ignore_eos=False*, *random_seed=None*, *stop_words=None*, *bad_words=None*, *stop_token_ids=None*, *bad_token_ids=None*, *min_new_tokens=None*, *skip_special_tokens=True*, *spaces_between_special_tokens=True*, *logprobs=None*, *response_format=None*, *logits_processors=None*, *output_logits=None*, *output_last_hidden_state=None*, *include_stop_str_in_output=False*, *with_cache=False*, *preserve_cache=False*, *migration_request=None*, *return_routed_experts=False*, *repetition_ngram_size=0*, *repetition_ngram_threshold=0*)

Generation parameters used by inference engines.

Parameters

- **n** (int) – Define how many chat completion choices to generate for each input message. **Only 1** is supported now.
- **max_new_tokens** (int) – The maximum number of tokens that can be generated in the chat completion
- **do_sample** (bool) – Whether or not to use sampling, use greedy decoding otherwise. Default to be False.
- **top_p** (float) – An alternative to sampling with temperature, called nucleus sampling, where the model considers the results of the tokens with top_p probability mass
- **top_k** (int) – An alternative to sampling with temperature, where the model considers the top_k tokens with the highest probability
- **min_p** (float) – Minimum token probability, which will be scaled by the probability of the most likely token. It must be a value between 0 and 1. Typical values are in the 0.01-0.2

range, comparably selective as setting *top_p* in the 0.99-0.8 range (use the opposite of normal *top_p* values)

- **temperature** (float) – Sampling temperature
- **repetition_penalty** (float) – Penalty to prevent the model from generating repeated words or phrases. A value larger than 1 discourages repetition
- **ignore_eos** (bool) – Indicator to ignore the eos_token_id or not
- **random_seed** (Optional[int]) – Seed used when sampling a token
- **stop_words** (Optional[list[str]]) – Words that stop generating further tokens
- **bad_words** (Optional[list[str]]) – Words that the engine will never generate
- **stop_token_ids** (Optional[list[int]]) – list of tokens that stop the generation when they are generated. The returned output will not contain the stop tokens.
- **bad_token_ids** (Optional[list[int]]) – list of tokens that the engine will never generate.
- **min_new_tokens** (Optional[int]) – The minimum numbers of tokens to generate, ignoring the number of tokens in the prompt.
- **skip_special_tokens** (bool) – Whether or not to remove special tokens in the decoding. Default to be True.
- **spaces_between_special_tokens** (bool) – Whether or not to add spaces around special tokens. The behavior of Fast tokenizers is to have this to False. This is setup to True in slow tokenizers.
- **logprobs** (Optional[int]) – Number of log probabilities to return per output token.
- **response_format** (Optional[dict]) – Generate responses according to given formatting. Examples:

```
{
    "type": "json_schema",
    "json_schema": {
        "name": "test",
        "schema": {
        "properties": {
            "name": {
            "type": "string"
            }
        },
        "required": ["name"],
        "type": "object"
        }
    }
}
```

or,

```
{
    "type": "regex_schema",
    "regex_schema": "call me [A-Za-z]{1,10}"
}
```

- **logits_processors** (Optional[list[Callable[[Tensor, Tensor], Tensor]]]) – Custom logit processors.

- **repetition_ngram_size** (int) – The size of n-grams to consider for repetition early stop. Must be non-negative; values below 0 are treated as 0.
- **repetition_ngram_threshold** (int) – The number of times an n-gram must be repeated to trigger early stop. Must be non-negative; values below 0 are treated as 0.

class lmdeploy.**ChatTemplateConfig**(*model_name*, *model_path=None*, *system=None*, *meta_instruction=None*, *eosys=None*, *user=None*, *eoh=None*, *assistant=None*, *eoa=None*, *tool=None*, *eotool=None*, *separator=None*, *capability=None*, *stop_words=None*)

Parameters for chat template.

Parameters

- **model_name** (str) – the name of the deployed model. Determine which chat template will be applied. All the chat template names: `lmdeploy list`
- **system** (Optional[str]) – begin of the system prompt.
- **meta_instruction** (Optional[str]) – system prompt.
- **eosys** (Optional[str]) – end of the system prompt.
- **user** (Optional[str]) – begin of the user prompt.
- **eoh** (Optional[str]) – end of the user prompt.
- **assistant** (Optional[str]) – begin of the assistant prompt.
- **eoa** (Optional[str]) – end of the assistant prompt.
- **tool** (Optional[str]) – begin of the tool prompt.
- **eotool** (Optional[str]) – end of the tool prompt.
- **capability** (Optional[Literal['completion', 'infilling', 'chat', 'python']]) – the capability of the model, one of 'completion', 'infilling', 'chat', 'python'. Default to None.
- **stop_words** (Optional[list[str]]) – list of stop words. Default to None.

1.37 OpenAPI Endpoints

1.37.1 OpenAI Compatible API Endpoints

POST /abort_request

Abort Request

Abort an ongoing request.

Request body:

```
{
  "abort_all":{
    "default":false,
    "title":"Abort All",
    "type":"boolean"
  },
  "abort_message":{
```

(continues on next page)

(continued from previous page)


```
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"Abort Message"
  },
  "finished_reason":{
    "anyOf":[
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "title":"Finished Reason"
  },
  "session_id":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":-1,
    "title":"Session Id"
  }
}
```


Example request:

```
POST /abort_request HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "abort_all": true,
    "abort_message": "string",
    "finished_reason": {},
    "session_id": 1
}
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

- 422 Unprocessable Entity – Validation Error

Example response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```

`GET /distserve/engine_info`

Engine Info

Example request:

```
GET /distserve/engine_info HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`POST /distserve/free_cache`

Free Cache

Request body:

```
{
  "remote_engine_id":{
```

(continues on next page)

(continued from previous page)

```
    "title":"Remote Engine Id",
    "type":"string"
  },
  "remote_session_id":{
    "title":"Remote Session Id",
    "type":"integer"
  }
}
```

Example request:

```
POST /distserve/free_cache HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "remote_engine_id": "string",
    "remote_session_id": 1
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
      "detail": [
          {
              "ctx": {},
              "input": {},
              "loc": [
                  "string",
                  1
              ],
              "msg": "string",
              "type": "string"
          }
      ]
  }
  ```

`POST /distserve/p2p_connect`

P2P Connect

Request body:


```
{
  "protocol":{
    "description":"Migration Transport Protocol.\n\nAttributes:\n    RDMA: IB or␣
↪RoCEv1/v2.\n    NVLINK: High device-to-device link.\n\nWarning: By now, only `GPU␣
↪Directed RDMA` is supported in DistServe.\n    We preserve several protocol and␣
↪will be implemented in the future.",
    "enum":[
      1,
      2,
      3
    ],
    "title":"MigrationProtocol",
    "type":"integer"
  },
  "remote_engine_endpoint_info":{
    "properties":{
      "zmq_address":{
        "title":"Zmq Address",
        "type":"string"
      }
    },
    "required":[
      "zmq_address"
    ],
    "title":"DistServeEngineEndpointInfo",
    "type":"object"
  },
  "remote_engine_id":{
    "title":"Remote Engine Id",
    "type":"string"
  },
  "remote_kvtransfer_endpoint_info":{
    "items":{
      "properties":{
        "endpoint_info":{
          "title":"Endpoint Info",
          "type":"string"
        },
        "protocol":{
          "description":"Migration Transport Protocol.\n\nAttributes:\n    RDMA: IB␣
↪or RoCEv1/v2.\n    NVLINK: High device-to-device link.\n\nWarning: By now, only␣
↪`GPU Directed RDMA` is supported in DistServe.\n    We preserve several protocol␣
↪and will be implemented in the future.",
          "enum":[
            1,
            2,
            3
          ],
          "title":"MigrationProtocol",
```


(continues on next page)

(continued from previous page)

```
          "type":"integer"
        }
      },
      "required":[
        "protocol",
        "endpoint_info"
      ],
      "title":"DistServeKVTransferEndpointInfo",
      "type":"object"
    },
    "title":"Remote Kvtransfer Endpoint Info",
    "type":"array"
  }
}
```

Example request:

```
POST /distserve/p2p_connect HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "protocol": 1,
    "remote_engine_endpoint_info": {
        "zmq_address": "string"
    },
    "remote_engine_id": "string",
    "remote_kvtransfer_endpoint_info": [
        {
            "endpoint_info": "string",
            "protocol": 1
        }
    ]
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
  ```

(continues on next page)

(continued from previous page)


```
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```


`POST /distserve/p2p_drop_connect`

P2P Drop Connect

Request body:


```
{
  "engine_id":{
    "title":"Engine Id",
    "type":"string"
  },
  "remote_engine_id":{
    "title":"Remote Engine Id",
    "type":"string"
  }
}
```


Example request:

```
POST /distserve/p2p_drop_connect HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "engine_id": "string",
    "remote_engine_id": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

Example response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```

`POST /distserve/p2p_initialize`

P2P Initialize

Request body:

```
{
  "local_engine_config":{
    "description":"DistServe Engine Config.\n\nIn Disaggregated LLM Serving, we␣
↪need to get engine info of each\nPD Peer for the following reason:\n    1. Cache:␣
↪The stride of cache block for correct offset of KV Transfer.\n    2. Parallel:␣
↪Prefill and decode use different parallel strategy to\n        achieve high SLO␣
↪Attainment or high throughput. In this situation,\n        we need to caclculate␣
↪which prefill-decode worker peers need to connect.\n        For example, prefill␣
↪worker use pp4 and decode worker use tp2pp2,\n        the perfill-decode worker␣
↪conn peer is (0, 0), (0, 1), (1, 0), (1, 1),\n        (2, 2), (2, 3), (3, 2), (3,␣
↪3). Instead, under the situation of\n        (tp4, tp4), perfill-decode worker␣
↪conn peer is (0, 0), (1, 1), (2, 2),\n        (3, 3).",
    "properties":{
      "block_size":{
        "title":"Block Size",
        "type":"integer"
      },
      "dp_rank":{
        "title":"Dp Rank",
        "type":"integer"
      },
      "dp_size":{
        "title":"Dp Size",
        "type":"integer"
      },
      "ep_size":{
        "title":"Ep Size",
        "type":"integer"
```

(continues on next page)

(continued from previous page)


```
      },
      "num_cpu_blocks":{
        "title":"Num Cpu Blocks",
        "type":"integer"
      },
      "num_gpu_blocks":{
        "title":"Num Gpu Blocks",
        "type":"integer"
      },
      "pp_size":{
        "anyOf":[
          {
            "type":"integer"
          },
          {
            "type":"null"
          }
        ],
        "title":"Pp Size"
      },
      "tp_size":{
        "title":"Tp Size",
        "type":"integer"
      }
    },
    "required":[
      "tp_size",
      "ep_size",
      "dp_size",
      "pp_size",
      "dp_rank",
      "block_size",
      "num_cpu_blocks",
      "num_gpu_blocks"
    ],
    "title":"DistServeEngineConfig",
    "type":"object"
  },
  "local_engine_id":{
    "title":"Local Engine Id",
    "type":"string"
  },
  "nvlink_config":{
    "anyOf":[
      {
        "description":"TODO: Add NVLink Protocol",
        "properties":{},
        "title":"DistServeNVLinkConfig",
        "type":"object"
      },
      {
        "type":"null"
```


(continues on next page)

(continued from previous page)


```
      }
    ]
  },
  "protocol":{
    "description":"Migration Transport Protocol.\n\nAttributes:\n    RDMA: IB or␣
↪RoCEv1/v2.\n    NVLINK: High device-to-device link.\n\nWarning: By now, only `GPU␣
↪Directed RDMA` is supported in DistServe.\n    We preserve several protocol and␣
↪will be implemented in the future.",
    "enum":[
      1,
      2,
      3
    ],
    "title":"MigrationProtocol",
    "type":"integer"
  },
  "rank":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "title":"Rank"
  },
  "rdma_config":{
    "anyOf":[
      {
        "description":"DistServe RDMA Config.\n\nArgs:\n    with_gdr: default to␣
↪True.\n    link_type: default to `RDMALinkType.RoCE`.\n\nWarning: Only GDR is␣
↪supported by now.\nWarning: Technically, both RoCE and IB are supported.\n    ␣
↪However, IB mode is not tested because of unavailable\n    testing envoriment.",
        "properties":{
          "link_type":{
            "description":"RDMA Link Type.",
            "enum":[
              1,
              2
            ],
            "title":"RDMALinkType",
            "type":"integer"
          },
          "with_gdr":{
            "default":true,
            "title":"With Gdr",
            "type":"boolean"
          }
        },
        "title":"DistServeRDMAConfig",
        "type":"object"
```


(continues on next page)

(continued from previous page)


```
      },
      {
        "type":"null"
      }
    ]
  },
  "remote_engine_config":{
    "description":"DistServe Engine Config.\n\nIn Disaggregated LLM Serving, we␣
↪need to get engine info of each\nPD Peer for the following reason:\n    1. Cache:␣
↪The stride of cache block for correct offset of KV Transfer.\n    2. Parallel:␣
↪Prefill and decode use different parallel strategy to\n        achieve high SLO␣
↪Attainment or high throughput. In this situation,\n        we need to caclculate␣
↪which prefill-decode worker peers need to connect.\n        For example, prefill␣
↪worker use pp4 and decode worker use tp2pp2,\n        the perfill-decode worker␣
↪conn peer is (0, 0), (0, 1), (1, 0), (1, 1),\n        (2, 2), (2, 3), (3, 2), (3,␣
↪3). Instead, under the situation of\n        (tp4, tp4), perfill-decode worker␣
↪conn peer is (0, 0), (1, 1), (2, 2),\n        (3, 3).",
    "properties":{
      "block_size":{
        "title":"Block Size",
        "type":"integer"
      },
      "dp_rank":{
        "title":"Dp Rank",
        "type":"integer"
      },
      "dp_size":{
        "title":"Dp Size",
        "type":"integer"
      },
      "ep_size":{
        "title":"Ep Size",
        "type":"integer"
      },
      "num_cpu_blocks":{
        "title":"Num Cpu Blocks",
        "type":"integer"
      },
      "num_gpu_blocks":{
        "title":"Num Gpu Blocks",
        "type":"integer"
      },
      "pp_size":{
        "anyOf":[
          {
            "type":"integer"
          },
          {
            "type":"null"
          }
        ],
        "title":"Pp Size"
```


(continues on next page)

(continued from previous page)

```
      },
      "tp_size":{
        "title":"Tp Size",
        "type":"integer"
      }
    },
    "required":[
      "tp_size",
      "ep_size",
      "dp_size",
      "pp_size",
      "dp_rank",
      "block_size",
      "num_cpu_blocks",
      "num_gpu_blocks"
    ],
    "title":"DistServeEngineConfig",
    "type":"object"
  },
  "remote_engine_id":{
    "title":"Remote Engine Id",
    "type":"string"
  },
  "tcp_config":{
    "anyOf":[
      {
        "description":"TODO: Add TCP Protocol",
        "properties":{},
        "title":"DistServeTCPConfig",
        "type":"object"
      },
      {
        "type":"null"
      }
    ]
  }
}
```

Example request:

```
POST /distserve/p2p_initialize HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "local_engine_config": {
        "block_size": 1,
        "dp_rank": 1,
        "dp_size": 1,
        "ep_size": 1,
        "num_cpu_blocks": 1,
        "num_gpu_blocks": 1,
```

(continues on next page)

(continued from previous page)

```
        "pp_size": 1,
        "tp_size": 1
    },
    "local_engine_id": "string",
    "protocol": 1,
    "rank": 1,
    "rdma_config": {
        "link_type": 1,
        "with_gdr": true
    },
    "remote_engine_config": {
        "block_size": 1,
        "dp_rank": 1,
        "dp_size": 1,
        "ep_size": 1,
        "num_cpu_blocks": 1,
        "num_gpu_blocks": 1,
        "pp_size": 1,
        "tp_size": 1
    },
    "remote_engine_id": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
      "detail": [
          {
              "ctx": {},
              "input": {},
              "loc": [
                  "string",
                  1
              ],
              "msg": "string",
              "type": "string"
          }
  ```

(continues on next page)

(continued from previous page)

```
    ]
}
```

`POST /generate`

Generate

Request body:

```
{
  "ignore_eos":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Ignore Eos"
  },
  "image_data":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "items":{
          "anyOf":[
            {
              "type":"string"
            },
            {
              "additionalProperties":true,
              "type":"object"
            }
          ]
        },
        "type":"array"
      },
      {
        "type":"null"
      }
    ],
    "title":"Image Data"
  },
  "include_stop_str_in_output":{
    "anyOf":[
```

(continues on next page)

(continued from previous page)

```
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Include Stop Str In Output"
  },
  "input_ids":{
    "anyOf":[
      {
        "items":{
          "type":"integer"
        },
        "type":"array"
      },
      {
        "type":"null"
      }
    ],
    "title":"Input Ids"
  },
  "max_tokens":{
    "default":128,
    "title":"Max Tokens",
    "type":"integer"
  },
  "media_io_kwargs":{
    "anyOf":[
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "description":"Additional kwargs to pass to the media IO processing, keyed by␣
↪modality.",
    "title":"Media Io Kwargs"
  },
  "min_p":{
    "default":0.0,
    "title":"Min P",
    "type":"number"
  },
  "mm_processor_kwargs":{
    "anyOf":[
      {
        "additionalProperties":true,
```

(continues on next page)

(continued from previous page)


```
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "description":"Additional kwargs to pass to the HF processor",
    "title":"Mm Processor Kwargs"
  },
  "prompt":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"Prompt"
  },
  "repetition_ngram_size":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Size",
    "type":"integer"
  },
  "repetition_ngram_threshold":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Threshold",
    "type":"integer"
  },
  "repetition_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":1.0,
    "title":"Repetition Penalty"
  },
  "return_logprob":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
```


(continues on next page)

(continued from previous page)


```
    ],
    "title":"Return Logprob"
  },
  "return_routed_experts":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Return Routed Experts"
  },
  "session_id":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":-1,
    "title":"Session Id"
  },
  "skip_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Skip Special Tokens"
  },
  "spaces_between_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Spaces Between Special Tokens"
  },
  "stop":{
```


(continues on next page)

(continued from previous page)


```
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
      },
      {
        "type":"null"
      }
    ],
    "title":"Stop"
  },
  "stop_token_ids":{
    "anyOf":[
      {
        "items":{
          "type":"integer"
        },
        "type":"array"
      },
      {
        "type":"null"
      }
    ],
    "title":"Stop Token Ids"
  },
  "stream":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Stream"
  },
  "temperature":{
    "default":1.0,
    "title":"Temperature",
    "type":"number"
  },
  "top_k":{
    "default":0,
    "title":"Top K",
    "type":"integer"
  },
```


(continues on next page)

(continued from previous page)

```
  "top_p":{
    "default":1.0,
    "title":"Top P",
    "type":"number"
  }
}
```

Example request:

```
POST /generate HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "ignore_eos": true,
    "image_data": "string",
    "include_stop_str_in_output": true,
    "input_ids": [
        1
    ],
    "max_tokens": 1,
    "media_io_kwargs": {},
    "min_p": 1.0,
    "mm_processor_kwargs": {},
    "prompt": "string",
    "repetition_ngram_size": 1,
    "repetition_ngram_threshold": 1,
    "repetition_penalty": 1.0,
    "return_logprob": true,
    "return_routed_experts": true,
    "session_id": 1,
    "skip_special_tokens": true,
    "spaces_between_special_tokens": true,
    "stop": "string",
    "stop_token_ids": [
        1
    ],
    "stream": true,
    "temperature": 1.0,
    "top_k": 1,
    "top_p": 1.0
}
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

- 422 Unprocessable Entity – Validation Error

Example response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```

`GET /health`

Health

Health check.

Example request:

```
GET /health HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`GET /is_sleeping`

Is Sleeping

Example request:

```
GET /is_sleeping HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`GET /metrics`

Metrics

[Optional] Prometheus metrics endpoint.

Example request:

```
GET /metrics HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Prometheus metrics data
- 404 Not Found – Metrics Endpoint not enabled

`POST /pooling`

Pooling

Pooling prompts for reward model.

In vLLM documentation, https://docs.vllm.ai/en/latest/serving/openai_compatible_server.html#pooling-api_1, the input format of Pooling API is the same as Embeddings API.

Go to https://platform.openai.com/docs/api-reference/embeddings/create for the Embeddings API specification.

The request should be a JSON object with the following fields:

- **model** (str): model name. Available from /v1/models.
- **input** (list[int] | list[list[int]] | str | list[str]): input text to be embed

Request body:

```
{
  "dimensions":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "title":"Dimensions"
  },
  "encoding_format":{
    "default":"float",
    "enum":[
      "float",
      "base64"
    ],
```

(continues on next page)

(continued from previous page)


```
    "title":"Encoding Format",
    "type":"string"
  },
  "input":{
    "anyOf":[
      {
        "items":{
          "type":"integer"
        },
        "type":"array"
      },
      {
        "items":{
          "items":{
            "type":"integer"
          },
          "type":"array"
        },
        "type":"array"
      },
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
      }
    ],
    "title":"Input"
  },
  "model":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"Model"
  },
  "user":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
```


(continues on next page)

(continued from previous page)

```
    "title":"User"
  }
}
```

Example request:

```
POST /pooling HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "dimensions": 1,
    "encoding_format": "float",
    "input": [
        1
    ],
    "model": "string",
    "user": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
      "detail": [
          {
              "ctx": {},
              "input": {},
              "loc": [
                  "string",
                  1
              ],
              "msg": "string",
              "type": "string"
          }
      ]
  }
  ```

`POST /sleep`

Sleep

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`GET /terminate`

Terminate

Terminate server.

Example request:

```
GET /terminate HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`POST /update_weights`

Update Params

Update weights for the model.

Request body:

```
{
  "finished":{
    "default":false,
    "title":"Finished",
    "type":"boolean"
  },
  "load_format":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
```

(continues on next page)

(continued from previous page)

```
    "title":"Load Format"
  },
  "serialized_named_tensors":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
      },
      {
        "additionalProperties":true,
        "type":"object"
      }
    ],
    "title":"Serialized Named Tensors"
  }
}
```

Example request:

```
POST /update_weights HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "finished": true,
    "load_format": "string",
    "serialized_named_tensors": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
      "detail": [
  ```

(continues on next page)

(continued from previous page)

```
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```

POST /v1/chat/completions

Chat Completions V1

Completion API similar to OpenAI's API.

Refer to https://platform.openai.com/docs/api-reference/chat/create for the API specification.

The request should be a JSON object with the following fields:

- **model**: model name. Available from /v1/models.
- **messages**: string prompt or chat history in OpenAI format. Chat history example: `[{"role": "user", "content": "hi"}]`.
- **temperature** (float): to modulate the next token probability
- **top_p** (float): If set to float < 1, only the smallest set of most probable tokens with probabilities that add up to top_p or higher are kept for generation.
- **n** (int): How many chat completion choices to generate for each input message. **Only support one here**.
- **stream**: whether to stream the results or not. Default to false.
- **stream_options**: Options for streaming response. Only set this when you set stream: true.
- **max_completion_tokens** (int | None): output token nums. Default to None.
- **max_tokens** (int | None): output token nums. Default to None. Deprecated: Use max_completion_tokens instead.
- **repetition_penalty** (float): The parameter for repetition penalty. 1.0 means no penalty
- **stop** (str | list[str] | None): To stop generating further tokens. Only accept stop words that's encoded to one token idex.
- **response_format** (dict | None): To generate response according to given schema. Examples:

```
{
  "type": "json_schema",
  "json_schema":{
    "name": "test",
    "schema":{
      "properties":{
        "name":{"type":"string"}
      },
      "required":["name"],
```

(continues on next page)

(continued from previous page)

```
      "type":"object"
    }
  }
}
```

or `{"type": "regex_schema", "regex_schema": "call me [A-Za-z]{1,10}"}`

- **logit_bias** (dict): Bias to logits. Only supported in pytorch engine.
- **tools** (list): A list of tools the model may call. Currently, only internlm2 functions are supported as a tool. Use this to specify a list of functions for which the model can generate JSON inputs.
- **tool_choice** (str | object): Controls which (if any) tool is called by the model. *none* means the model will not call any tool and instead generates a message. Specifying a particular tool via `{"type": "function", "function": {"name": "my_function"}}` forces the model to call that tool. *auto* or *required* will put all the tools informationto the model.

Additional arguments supported by LMDeploy:

- **top_k** (int): The number of the highest probability vocabulary tokens to keep for top-k-filtering
- **ignore_eos** (bool): indicator for ignoring eos
- **skip_special_tokens** (bool): Whether or not to remove special tokens in the decoding. Default to be True.
- **spaces_between_special_tokens** (bool): Whether or not to add spaces around special tokens. The behavior of Fast tokenizers is to have this to False. This is setup to True in slow tokenizers.
- **min_new_tokens** (int): To generate at least numbers of tokens.
- **min_p** (float): Minimum token probability, which will be scaled by the probability of the most likely token. It must be a value between 0 and 1. Typical values are in the 0.01-0.2 range, comparably selective as setting *top_p* in the 0.99-0.8 range (use the opposite of normal *top_p* values)
- **repetition_ngram_size** (int): N-gram length for repetition early stop (PyTorch engine). `0` disables.
- **repetition_ngram_threshold** (int): How many times that n-gram must repeat to trigger early stop. `0` disables.

Currently we do not support the following features:

- **presence_penalty** (replaced with repetition_penalty)
- **frequency_penalty** (replaced with repetition_penalty)

Request body:

```
{
  "chat_template_kwargs":{
    "anyOf":[
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "description":"Additional keyword args to pass to the template renderer. Will␣
↪be accessible by the chat template.",
```

(continues on next page)

(continued from previous page)

```
    "title":"Chat Template Kwargs"
  },
  "do_preprocess":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Do Preprocess"
  },
  "enable_thinking":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "title":"Enable Thinking"
  },
  "frequency_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.0,
    "title":"Frequency Penalty"
  },
  "ignore_eos":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Ignore Eos"
  },
  "include_stop_str_in_output":{
    "anyOf":[
      {
```

(continues on next page)

(continued from previous page)

```
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Include Stop Str In Output"
  },
  "logit_bias":{
    "anyOf":[
      {
        "additionalProperties":{
          "type":"number"
        },
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ],
    "title":"Logit Bias"
  },
  "logprobs":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Logprobs"
  },
  "max_completion_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "description":"An upper bound for the number of tokens that can be generated␣
↪for a completion, including visible output tokens and reasoning tokens",
    "examples":[
      null
    ],
```

(continues on next page)

(continued from previous page)

```
    "title":"Max Completion Tokens"
  },
  "max_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "deprecated":true,
    "examples":[
      null
    ],
    "title":"Max Tokens"
  },
  "media_io_kwargs":{
    "anyOf":[
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "description":"Additional kwargs to pass to the media IO processing, keyed by␣
↪modality.",
    "title":"Media Io Kwargs"
  },
  "messages":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "additionalProperties":true,
          "type":"object"
        },
        "type":"array"
      }
    ],
    "examples":[
      [
        {
          "content":"hi",
          "role":"user"
        }
      ]
    ],
```

(continues on next page)

(continued from previous page)


```
    "title":"Messages"
  },
  "min_new_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ],
    "title":"Min New Tokens"
  },
  "min_p":{
    "default":0.0,
    "title":"Min P",
    "type":"number"
  },
  "mm_processor_kwargs":{
    "anyOf":[
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "description":"Additional kwargs to pass to the HF processor",
    "title":"Mm Processor Kwargs"
  },
  "model":{
    "title":"Model",
    "type":"string"
  },
  "n":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":1,
    "title":"N"
  },
  "presence_penalty":{
    "anyOf":[
```


(continues on next page)

(continued from previous page)


```
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.0,
    "title":"Presence Penalty"
  },
  "reasoning_effort":{
    "anyOf":[
      {
        "enum":[
          "low",
          "medium",
          "high"
        ],
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"Reasoning Effort"
  },
  "repetition_ngram_size":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Size",
    "type":"integer"
  },
  "repetition_ngram_threshold":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Threshold",
    "type":"integer"
  },
  "repetition_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":1.0,
    "title":"Repetition Penalty"
  },
  "response_format":{
    "anyOf":[
```


(continues on next page)

(continued from previous page)


```
    {
      "properties":{
        "json_schema":{
          "anyOf":[
            {
              "properties":{
                "description":{
                  "anyOf":[
                    {
                      "type":"string"
                    },
                    {
                      "type":"null"
                    }
                  ],
                  "title":"Description"
                },
                "name":{
                  "title":"Name",
                  "type":"string"
                },
                "schema":{
                  "anyOf":[
                    {
                      "additionalProperties":true,
                      "type":"object"
                    },
                    {
                      "type":"null"
                    }
                  ],
                  "examples":[
                    null
                  ],
                  "title":"Schema"
                },
                "strict":{
                  "anyOf":[
                    {
                      "type":"boolean"
                    },
                    {
                      "type":"null"
                    }
                  ],
                  "default":false,
                  "title":"Strict"
                }
              },
              "required":[
                "name"
              ],
```


(continues on next page)

(continued from previous page)

```
                "title":"JsonSchema",
                "type":"object"
              },
              {
                "type":"null"
              }
            ]
          },
          "regex_schema":{
            "anyOf":[
              {
                "type":"string"
              },
              {
                "type":"null"
              }
            ],
            "title":"Regex Schema"
          },
          "type":{
            "enum":[
              "text",
              "json_object",
              "json_schema",
              "regex_schema"
            ],
            "title":"Type",
            "type":"string"
          }
        },
        "required":[
          "type"
        ],
        "title":"ResponseFormat",
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ]
  },
  "return_token_ids":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
```

(continues on next page)

(continued from previous page)


```
    ],
    "default":false,
    "title":"Return Token Ids"
  },
  "seed":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "title":"Seed"
  },
  "session_id":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":-1,
    "title":"Session Id"
  },
  "skip_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Skip Special Tokens"
  },
  "spaces_between_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Spaces Between Special Tokens"
  },
  "stop":{
```


(continues on next page)

(continued from previous page)

```
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ],
    "title":"Stop"
  },
  "stream":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Stream"
  },
  "stream_options":{
    "anyOf":[
      {
        "description":"The stream options.",
        "properties":{
          "include_usage":{
            "anyOf":[
              {
                "type":"boolean"
              },
              {
                "type":"null"
              }
            ],
            "default":false,
            "title":"Include Usage"
          }
        },
        "title":"StreamOptions",
        "type":"object"
      },
```

(continues on next page)

(continued from previous page)


```
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ]
  },
  "temperature":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.7,
    "title":"Temperature"
  },
  "tool_choice":{
    "anyOf":[
      {
        "description":"The tool choice definition.",
        "properties":{
          "function":{
            "description":"The name of tool choice function.",
            "properties":{
              "name":{
                "title":"Name",
                "type":"string"
              }
            },
            "required":[
              "name"
            ],
            "title":"ToolChoiceFuncName",
            "type":"object"
          },
          "type":{
            "const":"function",
            "default":"function",
            "examples":[
              "function"
            ],
            "title":"Type",
            "type":"string"
          }
        },
        "required":[
          "function"
        ],
```


(continues on next page)

(continued from previous page)

```
        "title":"ToolChoice",
        "type":"object"
      },
      {
        "enum":[
          "auto",
          "required",
          "none"
        ],
        "type":"string"
      }
    ],
    "default":"auto",
    "examples":[
      "none"
    ],
    "title":"Tool Choice"
  },
  "tools":{
    "anyOf":[
      {
        "items":{
          "description":"Function wrapper.",
          "properties":{
            "function":{
              "description":"Function descriptions.",
              "properties":{
                "description":{
                  "anyOf":[
                    {
                      "type":"string"
                    },
                    {
                      "type":"null"
                    }
                  ],
                  "examples":[
                    null
                  ],
                  "title":"Description"
                },
                "name":{
                  "title":"Name",
                  "type":"string"
                },
                "parameters":{
                  "anyOf":[
                    {
                      "additionalProperties":true,
                      "type":"object"
                    },
                    {
```

(continues on next page)

(continued from previous page)


```
                    "type":"null"
                  }
                ],
                "title":"Parameters"
              }
            },
            "required":[
              "name"
            ],
            "title":"Function",
            "type":"object"
          },
          "type":{
            "default":"function",
            "examples":[
              "function"
            ],
            "title":"Type",
            "type":"string"
          }
        },
        "required":[
          "function"
        ],
        "title":"Tool",
        "type":"object"
      },
      "type":"array"
    },
    {
      "type":"null"
    }
  ],
  "examples":[
    null
  ],
  "title":"Tools"
},
"top_k":{
  "anyOf":[
    {
      "type":"integer"
    },
    {
      "type":"null"
    }
  ],
  "default":40,
  "title":"Top K"
},
"top_logprobs":{
  "anyOf":[
```


(continues on next page)

(continued from previous page)

```
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "title":"Top Logprobs"
  },
  "top_p":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":1.0,
    "title":"Top P"
  },
  "user":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"User"
  }
}
```

Example request:

```
POST /v1/chat/completions HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "chat_template_kwargs": {},
    "do_preprocess": true,
    "enable_thinking": true,
    "frequency_penalty": 1.0,
    "ignore_eos": true,
    "include_stop_str_in_output": true,
    "logit_bias": {},
    "logprobs": true,
    "max_completion_tokens": 1,
    "max_tokens": 1,
    "media_io_kwargs": {},
```

(continues on next page)

(continued from previous page)

```
    "messages": "string",
    "min_new_tokens": 1,
    "min_p": 1.0,
    "mm_processor_kwargs": {},
    "model": "string",
    "n": 1,
    "presence_penalty": 1.0,
    "reasoning_effort": "low",
    "repetition_ngram_size": 1,
    "repetition_ngram_threshold": 1,
    "repetition_penalty": 1.0,
    "response_format": {
        "json_schema": {
            "description": "string",
            "name": "string",
            "schema": {},
            "strict": true
        },
        "regex_schema": "string",
        "type": "text"
    },
    "return_token_ids": true,
    "seed": 1,
    "session_id": 1,
    "skip_special_tokens": true,
    "spaces_between_special_tokens": true,
    "stop": "string",
    "stream": true,
    "stream_options": {
        "include_usage": true
    },
    "temperature": 1.0,
    "tool_choice": {
        "function": {
            "name": "string"
        },
        "type": "string"
    },
    "tools": [
        {
            "function": {
                "description": "string",
                "name": "string",
                "parameters": {}
            },
            "type": "string"
        }
    ],
    "top_k": 1,
    "top_logprobs": 1,
    "top_p": 1.0,
    "user": "string"
```

(continues on next page)

(continued from previous page)

```
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

- 422 Unprocessable Entity – Validation Error

  Example response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```

`POST /v1/completions`

Completions V1

Completion API similar to OpenAI's API.

Go to https://platform.openai.com/docs/api-reference/completions/create for the API specification.

The request should be a JSON object with the following fields:

- **model** (str): model name. Available from /v1/models.
- **prompt** (str): the input prompt.
- **suffix** (str): The suffix that comes after a completion of inserted text.
- **max_completion_tokens** (int | None): output token nums. Default to None.
- **max_tokens** (int | None): output token nums. Default to 16. Deprecated: Use max_completion_tokens instead.
- **temperature** (float): to modulate the next token probability

- **top_p** (float): If set to float < 1, only the smallest set of most probable tokens with probabilities that add up to top_p or higher are kept for generation.
- **n** (int): How many chat completion choices to generate for each input message. **Only support one here**.
- **stream**: whether to stream the results or not. Default to false.
- **stream_options**: Options for streaming response. Only set this when you set stream: true.
- **repetition_penalty** (float): The parameter for repetition penalty. 1.0 means no penalty
- **user** (str): A unique identifier representing your end-user.
- **stop** (str | list[str] | None): To stop generating further tokens. Only accept stop words that's encoded to one token idex.

Additional arguments supported by LMDeploy:

- **ignore_eos** (bool): indicator for ignoring eos
- **skip_special_tokens** (bool): Whether or not to remove special tokens in the decoding. Default to be True.
- **spaces_between_special_tokens** (bool): Whether or not to add spaces around special tokens. The behavior of Fast tokenizers is to have this to False. This is setup to True in slow tokenizers.
- **top_k** (int): The number of the highest probability vocabulary tokens to keep for top-k-filtering
- **min_p** (float): Minimum token probability, which will be scaled by the probability of the most likely token. It must be a value between 0 and 1. Typical values are in the 0.01-0.2 range, comparably selective as setting *top_p* in the 0.99-0.8 range (use the opposite of normal *top_p* values)
- **repetition_ngram_size** (int): N-gram length for repetition early stop (PyTorch engine). `0` disables.
- **repetition_ngram_threshold** (int): How many times that n-gram must repeat to trigger early stop. `0` disables.

Currently we do not support the following features:

- **logprobs** (not supported yet)
- **presence_penalty** (replaced with repetition_penalty)
- **frequency_penalty** (replaced with repetition_penalty)

Request body:


```
{
  "echo":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Echo"
  },
  "frequency_penalty":{
    "anyOf":[
      {
        "type":"number"
```


(continues on next page)

(continued from previous page)

```
      },
      {
        "type":"null"
      }
    ],
    "default":0.0,
    "title":"Frequency Penalty"
  },
  "ignore_eos":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Ignore Eos"
  },
  "logprobs":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "title":"Logprobs"
  },
  "max_completion_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "description":"An upper bound for the number of tokens that can be generated␣
↪for a completion, including visible output tokens and reasoning tokens",
    "examples":[
      null
    ],
    "title":"Max Completion Tokens"
  },
  "max_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
```

(continues on next page)

(continued from previous page)


```
      {
        "type":"null"
      }
    ],
    "default":16,
    "deprecated":true,
    "examples":[
      16
    ],
    "title":"Max Tokens"
  },
  "min_p":{
    "default":0.0,
    "title":"Min P",
    "type":"number"
  },
  "model":{
    "title":"Model",
    "type":"string"
  },
  "n":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":1,
    "title":"N"
  },
  "presence_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.0,
    "title":"Presence Penalty"
  },
  "prompt":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{},
        "type":"array"
```


(continues on next page)

(continued from previous page)


```
      }
    ],
    "title":"Prompt"
  },
  "repetition_ngram_size":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Size",
    "type":"integer"
  },
  "repetition_ngram_threshold":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Threshold",
    "type":"integer"
  },
  "repetition_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":1.0,
    "title":"Repetition Penalty"
  },
  "return_token_ids":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Return Token Ids"
  },
  "seed":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "title":"Seed"
  },
  "session_id":{
```


(continues on next page)

(continued from previous page)


```
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":-1,
    "title":"Session Id"
  },
  "skip_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Skip Special Tokens"
  },
  "spaces_between_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Spaces Between Special Tokens"
  },
  "stop":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
```


(continues on next page)

(continued from previous page)


```
    ],
    "title":"Stop"
  },
  "stream":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Stream"
  },
  "stream_options":{
    "anyOf":[
      {
        "description":"The stream options.",
        "properties":{
          "include_usage":{
            "anyOf":[
              {
                "type":"boolean"
              },
              {
                "type":"null"
              }
            ],
            "default":false,
            "title":"Include Usage"
          }
        },
        "title":"StreamOptions",
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ]
  },
  "suffix":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
```


(continues on next page)

(continued from previous page)


```
    ],
    "title":"Suffix"
  },
  "temperature":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.7,
    "title":"Temperature"
  },
  "top_k":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":40,
    "title":"Top K"
  },
  "top_p":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":1.0,
    "title":"Top P"
  },
  "user":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"User"
  }
}
```


Example request:

```
POST /v1/completions HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "echo": true,
    "frequency_penalty": 1.0,
    "ignore_eos": true,
    "logprobs": 1,
    "max_completion_tokens": 1,
    "max_tokens": 1,
    "min_p": 1.0,
    "model": "string",
    "n": 1,
    "presence_penalty": 1.0,
    "prompt": "string",
    "repetition_ngram_size": 1,
    "repetition_ngram_threshold": 1,
    "repetition_penalty": 1.0,
    "return_token_ids": true,
    "seed": 1,
    "session_id": 1,
    "skip_special_tokens": true,
    "spaces_between_special_tokens": true,
    "stop": "string",
    "stream": true,
    "stream_options": {
        "include_usage": true
    },
    "suffix": "string",
    "temperature": 1.0,
    "top_k": 1,
    "top_p": 1.0,
    "user": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  ```

(continues on next page)

(continued from previous page)


```
{
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```


`POST /v1/embeddings`

Create Embeddings

Creates embeddings for the text.

Request body:


```
{
  "input":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
      }
    ],
    "title":"Input"
  },
  "model":{
    "title":"Model",
    "type":"string"
  },
  "user":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"User"
  }
}
```

Example request:

```
POST /v1/embeddings HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "input": "string",
    "model": "string",
    "user": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
      "detail": [
          {
              "ctx": {},
              "input": {},
              "loc": [
                  "string",
                  1
              ],
              "msg": "string",
              "type": "string"
          }
      ]
  }
  ```

`POST /v1/encode`

Encode

Encode prompts.

The request should be a JSON object with the following fields:

- **input**: the prompt to be encoded. In str or list[str] format.
- **do_preprocess**: whether do preprocess or not. Default to False.
- **add_bos**: True when it is the beginning of a conversation. False when it is not. Default to True.

Request body:


```
{
  "add_bos":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Add Bos"
  },
  "do_preprocess":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Do Preprocess"
  },
  "input":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
      }
    ],
    "title":"Input"
  }
}
```


Example request:

```
POST /v1/encode HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "add_bos": true,
    "do_preprocess": true,
    "input": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

- 422 Unprocessable Entity – Validation Error

  Example response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```

`GET /v1/models`

Available Models

Show available models.

Example request:

```
GET /v1/models HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Successful Response

  Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`POST /wakeup`

Wakeup

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

1.37.2 Proxy Server API

`POST /distserve/connection_warmup`

Connection Warmup

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`POST /distserve/gc`

Cache Block Gc To Be Migrated

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`POST /nodes/add`

Add Node

Add a node to the manager.

- **url** (str): A http url. Can be the url generated by *lmdeploy serve api_server*.
- **status** (dict): The description of the node. An example: `{models: ['internlm-chat-7b], speed: 1}`. The speed here can be RPM or other metric. All the values of nodes should be the same metric.

Request body:

```
{
  "status":{
    "anyOf":[
      {
        "description":"Status protocol consists of models' information.",
```

(continues on next page)

(continued from previous page)


```
        "properties":{
          "latency":{
            "default":[],
            "examples":[
              []
            ],
            "items":{},
            "title":"Latency",
            "type":"array"
          },
          "models":{
            "default":[],
            "examples":[
              []
            ],
            "items":{
              "type":"string"
            },
            "title":"Models",
            "type":"array"
          },
          "role":{
            "description":"Role of Engine.\n\nNote: In the implementation of␣
↪LMDeploy-Distserve, all engine is hybrid\n    engine technically, the role of␣
↪engine is up to what kind of request is\n    sent to the engine. However, taking␣
↪implementation into the consideration,\n    the role is still need to be␣
↪identified when starting the engine server\n    for the following reasons:\n    ␣
↪  1. Make sure the engine can be correctly discovered by the proxy.\n        2.␣
↪The create of ModelInputs is different among hybrid, prefill and\n          ␣
↪decode engines in DP Engine (DSV3 DP + EP).",
            "enum":[
              1,
              2,
              3
            ],
            "title":"EngineRole",
            "type":"integer"
          },
          "speed":{
            "anyOf":[
              {
                "type":"integer"
              },
              {
                "type":"null"
              }
            ],
            "examples":[
              null
            ],
            "title":"Speed"
          },
```


(continues on next page)

(continued from previous page)

```
          "unfinished":{
            "default":0,
            "title":"Unfinished",
            "type":"integer"
          }
        },
        "title":"Status",
        "type":"object"
      },
      {
        "type":"null"
      }
    ]
  },
  "url":{
    "title":"Url",
    "type":"string"
  }
}
```

Example request:

```
POST /nodes/add HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "status": {
        "latency": [
            {}
        ],
        "models": [
            "string"
        ],
        "role": 1,
        "speed": 1,
        "unfinished": 1
    },
    "url": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

Example response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```

`POST /nodes/remove`

Remove Node

Show available models.

Request body:

```
{
  "status":{
    "anyOf":[
      {
        "description":"Status protocol consists of models' information.",
        "properties":{
          "latency":{
            "default":[],
            "examples":[
              []
            ],
            "items":{},
            "title":"Latency",
            "type":"array"
          },
          "models":{
            "default":[],
            "examples":[
              []
            ],
            "items":{
              "type":"string"
            },
            "title":"Models",
            "type":"array"
          },
          "role":{
```

(continues on next page)

(continued from previous page)


```
            "description":"Role of Engine.\n\nNote: In the implementation of␣
↪LMDeploy-Distserve, all engine is hybrid\n    engine technically, the role of␣
↪engine is up to what kind of request is\n    sent to the engine. However, taking␣
↪implementation into the consideration,\n    the role is still need to be␣
↪identified when starting the engine server\n    for the following reasons:\n    ␣
↪  1. Make sure the engine can be correctly discovered by the proxy.\n        2.␣
↪The create of ModelInputs is different among hybrid, prefill and\n          ␣
↪decode engines in DP Engine (DSV3 DP + EP).",
            "enum":[
              1,
              2,
              3
            ],
            "title":"EngineRole",
            "type":"integer"
          },
          "speed":{
            "anyOf":[
              {
                "type":"integer"
              },
              {
                "type":"null"
              }
            ],
            "examples":[
              null
            ],
            "title":"Speed"
          },
          "unfinished":{
            "default":0,
            "title":"Unfinished",
            "type":"integer"
          }
        },
        "title":"Status",
        "type":"object"
      },
      {
        "type":"null"
      }
    ]
  },
  "url":{
    "title":"Url",
    "type":"string"
  }
}
```


Example request:

```
POST /nodes/remove HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "status": {
        "latency": [
            {}
        ],
        "models": [
            "string"
        ],
        "role": 1,
        "speed": 1,
        "unfinished": 1
    },
    "url": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
      "detail": [
          {
              "ctx": {},
              "input": {},
              "loc": [
                  "string",
                  1
              ],
              "msg": "string",
              "type": "string"
          }
      ]
  }
  ```

`GET /nodes/status`

Node Status

Show nodes status.

Example request:

```
GET /nodes/status HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`POST /nodes/terminate`

Terminate Node

Terminate nodes.

Request body:


```
{
  "status":{
    "anyOf":[
      {
        "description":"Status protocol consists of models' information.",
        "properties":{
          "latency":{
            "default":[],
            "examples":[
              []
            ],
            "items":{},
            "title":"Latency",
            "type":"array"
          },
          "models":{
            "default":[],
            "examples":[
              []
            ],
            "items":{
              "type":"string"
            },
            "title":"Models",
            "type":"array"
          },
          "role":{
            "description":"Role of Engine.\n\nNote: In the implementation of␣
↪LMDeploy-Distserve, all engine is hybrid\n    engine technically, the role of␣
↪engine is up to what kind of request is\n    sent to the engine. However, taking␣
```


(continues on next page)

(continued from previous page)

```
↪implementation into the consideration,\n    the role is still need to be␣
↪identified when starting the engine server\n    for the following reasons:\n     ␣
↪  1. Make sure the engine can be correctly discovered by the proxy.\n        2.␣
↪The create of ModelInputs is different among hybrid, prefill and\n           ␣
↪decode engines in DP Engine (DSV3 DP + EP).",
            "enum":[
              1,
              2,
              3
            ],
            "title":"EngineRole",
            "type":"integer"
          },
          "speed":{
            "anyOf":[
              {
                "type":"integer"
              },
              {
                "type":"null"
              }
            ],
            "examples":[
              null
            ],
            "title":"Speed"
          },
          "unfinished":{
            "default":0,
            "title":"Unfinished",
            "type":"integer"
          }
        },
        "title":"Status",
        "type":"object"
      },
      {
        "type":"null"
      }
    ]
  },
  "url":{
    "title":"Url",
    "type":"string"
  }
}
```

Example request:

```
POST /nodes/terminate HTTP/1.1
Host: example.com
Content-Type: application/json
```

(continues on next page)

(continued from previous page)


```
{
    "status": {
        "latency": [
            {}
        ],
        "models": [
            "string"
        ],
        "role": 1,
        "speed": 1,
        "unfinished": 1
    },
    "url": "string"
}
```


Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:


  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
      "detail": [
          {
              "ctx": {},
              "input": {},
              "loc": [
                  "string",
                  1
              ],
              "msg": "string",
              "type": "string"
          }
      ]
  }
  ```


`GET /nodes/terminate_all`

Terminate Node All

Terminate nodes.

Example request:

```
GET /nodes/terminate_all HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

`POST /v1/chat/completions`

Chat Completions V1

Completion API similar to OpenAI's API.

Refer to https://platform.openai.com/docs/api-reference/chat/create for the API specification.

The request should be a JSON object with the following fields:

- **model**: model name. Available from /v1/models.
- **messages**: string prompt or chat history in OpenAI format. Chat history example: *[{"role": "user", "content": "hi"}]*.
- **temperature** (float): to modulate the next token probability
- **top_p** (float): If set to float < 1, only the smallest set of most probable tokens with probabilities that add up to top_p or higher are kept for generation.
- **n** (int): How many chat completion choices to generate for each input message. **Only support one here**.
- **stream**: whether to stream the results or not. Default to false.
- **max_completion_tokens** (int | None): output token nums. Default to None.
- **max_tokens** (int | None): output token nums. Default to None. Deprecated: Use max_completion_tokens instead.
- **repetition_penalty** (float): The parameter for repetition penalty. 1.0 means no penalty
- **stop** (str | list[str] | None): To stop generating further tokens. Only accept stop words that's encoded to one token idex.
- **response_format** (dict | None): To generate response according to given schema. Examples:

```
{
  "type": "json_schema",
  "json_schema":{
    "name": "test",
    "schema":{
      "properties":{
        "name":{"type":"string"}
      },
      "required":["name"],
      "type":"object"
    }
```

(continues on next page)

(continued from previous page)

```
  }
}
```

or `{"type": "regex_schema", "regex_schema": "call me [A-Za-z]{1,10}"}`

- **logit_bias** (dict): Bias to logits. Only supported in pytorch engine.
- **tools** (list): A list of tools the model may call. Currently, only internlm2 functions are supported as a tool. Use this to specify a list of functions for which the model can generate JSON inputs.
- **tool_choice** (str | object): Controls which (if any) tool is called by the model. *none* means the model will not call any tool and instead generates a message. Specifying a particular tool via `{"type": "function", "function": {"name": "my_function"}}` forces the model to call that tool. *auto* or *required* will put all the tools information to the model.

Additional arguments supported by LMDeploy:

- **top_k** (int): The number of the highest probability vocabulary tokens to keep for top-k-filtering
- **ignore_eos** (bool): indicator for ignoring eos
- **skip_special_tokens** (bool): Whether or not to remove special tokens in the decoding. Default to be True.
- **min_new_tokens** (int): To generate at least numbers of tokens.
- **min_p** (float): Minimum token probability, which will be scaled by the probability of the most likely token. It must be a value between 0 and 1. Typical values are in the 0.01-0.2 range, comparably selective as setting *top_p* in the 0.99-0.8 range (use the opposite of normal *top_p* values)

Currently we do not support the following features:

- **presence_penalty** (replaced with repetition_penalty)
- **frequency_penalty** (replaced with repetition_penalty)

Request body:

```
{
  "chat_template_kwargs":{
    "anyOf":[
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "description":"Additional keyword args to pass to the template renderer. Will␣
↪be accessible by the chat template.",
    "title":"Chat Template Kwargs"
  },
  "do_preprocess":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
```

(continues on next page)

(continued from previous page)


```
      }
    ],
    "default":true,
    "title":"Do Preprocess"
  },
  "enable_thinking":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "title":"Enable Thinking"
  },
  "frequency_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.0,
    "title":"Frequency Penalty"
  },
  "ignore_eos":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Ignore Eos"
  },
  "include_stop_str_in_output":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Include Stop Str In Output"
  },
```


(continues on next page)

(continued from previous page)


```
  "logit_bias":{
    "anyOf":[
      {
        "additionalProperties":{
          "type":"number"
        },
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ],
    "title":"Logit Bias"
  },
  "logprobs":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Logprobs"
  },
  "max_completion_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "description":"An upper bound for the number of tokens that can be generated␣
↪for a completion, including visible output tokens and reasoning tokens",
    "examples":[
      null
    ],
    "title":"Max Completion Tokens"
  },
  "max_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
```


(continues on next page)

(continued from previous page)


```
      }
    ],
    "deprecated":true,
    "examples":[
      null
    ],
    "title":"Max Tokens"
  },
  "media_io_kwargs":{
    "anyOf":[
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "description":"Additional kwargs to pass to the media IO processing, keyed by␣
↪modality.",
    "title":"Media Io Kwargs"
  },
  "messages":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "additionalProperties":true,
          "type":"object"
        },
        "type":"array"
      }
    ],
    "examples":[
      [
        {
          "content":"hi",
          "role":"user"
        }
      ]
    ],
    "title":"Messages"
  },
  "min_new_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
```


(continues on next page)

(continued from previous page)


```
      }
    ],
    "examples":[
      null
    ],
    "title":"Min New Tokens"
  },
  "min_p":{
    "default":0.0,
    "title":"Min P",
    "type":"number"
  },
  "mm_processor_kwargs":{
    "anyOf":[
      {
        "additionalProperties":true,
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "description":"Additional kwargs to pass to the HF processor",
    "title":"Mm Processor Kwargs"
  },
  "model":{
    "title":"Model",
    "type":"string"
  },
  "n":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":1,
    "title":"N"
  },
  "presence_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.0,
    "title":"Presence Penalty"
```


(continues on next page)

(continued from previous page)


```
  },
  "reasoning_effort":{
    "anyOf":[
      {
        "enum":[
          "low",
          "medium",
          "high"
        ],
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"Reasoning Effort"
  },
  "repetition_ngram_size":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Size",
    "type":"integer"
  },
  "repetition_ngram_threshold":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Threshold",
    "type":"integer"
  },
  "repetition_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":1.0,
    "title":"Repetition Penalty"
  },
  "response_format":{
    "anyOf":[
      {
        "properties":{
          "json_schema":{
            "anyOf":[
              {
                "properties":{
                  "description":{
                    "anyOf":[
                      {
```


(continues on next page)

(continued from previous page)


```
                    "type":"string"
                  },
                  {
                    "type":"null"
                  }
                ],
                "title":"Description"
              },
              "name":{
                "title":"Name",
                "type":"string"
              },
              "schema":{
                "anyOf":[
                  {
                    "additionalProperties":true,
                    "type":"object"
                  },
                  {
                    "type":"null"
                  }
                ],
                "examples":[
                  null
                ],
                "title":"Schema"
              },
              "strict":{
                "anyOf":[
                  {
                    "type":"boolean"
                  },
                  {
                    "type":"null"
                  }
                ],
                "default":false,
                "title":"Strict"
              }
            },
            "required":[
              "name"
            ],
            "title":"JsonSchema",
            "type":"object"
          },
          {
            "type":"null"
          }
        ]
      },
      "regex_schema":{
```


(continues on next page)

(continued from previous page)


```
            "anyOf":[
              {
                "type":"string"
              },
              {
                "type":"null"
              }
            ],
            "title":"Regex Schema"
          },
          "type":{
            "enum":[
              "text",
              "json_object",
              "json_schema",
              "regex_schema"
            ],
            "title":"Type",
            "type":"string"
          }
        },
        "required":[
          "type"
        ],
        "title":"ResponseFormat",
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ]
  },
  "return_token_ids":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Return Token Ids"
  },
  "seed":{
    "anyOf":[
      {
        "type":"integer"
      },
```


(continues on next page)

(continued from previous page)

```
      {
        "type":"null"
      }
    ],
    "title":"Seed"
  },
  "session_id":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":-1,
    "title":"Session Id"
  },
  "skip_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Skip Special Tokens"
  },
  "spaces_between_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Spaces Between Special Tokens"
  },
  "stop":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
```

(continues on next page)

(continued from previous page)

```
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ],
    "title":"Stop"
  },
  "stream":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Stream"
  },
  "stream_options":{
    "anyOf":[
      {
        "description":"The stream options.",
        "properties":{
          "include_usage":{
            "anyOf":[
              {
                "type":"boolean"
              },
              {
                "type":"null"
              }
            ],
            "default":false,
            "title":"Include Usage"
          }
        },
        "title":"StreamOptions",
        "type":"object"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ]
  },
  "temperature":{
```

(continues on next page)

(continued from previous page)


```
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.7,
    "title":"Temperature"
  },
  "tool_choice":{
    "anyOf":[
      {
        "description":"The tool choice definition.",
        "properties":{
          "function":{
            "description":"The name of tool choice function.",
            "properties":{
              "name":{
                "title":"Name",
                "type":"string"
              }
            },
            "required":[
              "name"
            ],
            "title":"ToolChoiceFuncName",
            "type":"object"
          },
          "type":{
            "const":"function",
            "default":"function",
            "examples":[
              "function"
            ],
            "title":"Type",
            "type":"string"
          }
        },
        "required":[
          "function"
        ],
        "title":"ToolChoice",
        "type":"object"
      },
      {
        "enum":[
          "auto",
          "required",
          "none"
        ],
```


(continues on next page)

(continued from previous page)

```
        "type":"string"
      }
    ],
    "default":"auto",
    "examples":[
      "none"
    ],
    "title":"Tool Choice"
  },
  "tools":{
    "anyOf":[
      {
        "items":{
          "description":"Function wrapper.",
          "properties":{
            "function":{
              "description":"Function descriptions.",
              "properties":{
                "description":{
                  "anyOf":[
                    {
                      "type":"string"
                    },
                    {
                      "type":"null"
                    }
                  ],
                  "examples":[
                    null
                  ],
                  "title":"Description"
                },
                "name":{
                  "title":"Name",
                  "type":"string"
                },
                "parameters":{
                  "anyOf":[
                    {
                      "additionalProperties":true,
                      "type":"object"
                    },
                    {
                      "type":"null"
                    }
                  ],
                  "title":"Parameters"
                }
              },
              "required":[
                "name"
              ],
```

(continues on next page)

(continued from previous page)


```
            "title":"Function",
            "type":"object"
          },
          "type":{
            "default":"function",
            "examples":[
              "function"
            ],
            "title":"Type",
            "type":"string"
          }
        },
        "required":[
          "function"
        ],
        "title":"Tool",
        "type":"object"
      },
      "type":"array"
    },
    {
      "type":"null"
    }
  ],
  "examples":[
    null
  ],
  "title":"Tools"
},
"top_k":{
  "anyOf":[
    {
      "type":"integer"
    },
    {
      "type":"null"
    }
  ],
  "default":40,
  "title":"Top K"
},
"top_logprobs":{
  "anyOf":[
    {
      "type":"integer"
    },
    {
      "type":"null"
    }
  ],
  "title":"Top Logprobs"
},
```


(continues on next page)

(continued from previous page)


```
  "top_p":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":1.0,
    "title":"Top P"
  },
  "user":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"User"
  }
}
```


Example request:

```
POST /v1/chat/completions HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "chat_template_kwargs": {},
    "do_preprocess": true,
    "enable_thinking": true,
    "frequency_penalty": 1.0,
    "ignore_eos": true,
    "include_stop_str_in_output": true,
    "logit_bias": {},
    "logprobs": true,
    "max_completion_tokens": 1,
    "max_tokens": 1,
    "media_io_kwargs": {},
    "messages": "string",
    "min_new_tokens": 1,
    "min_p": 1.0,
    "mm_processor_kwargs": {},
    "model": "string",
    "n": 1,
    "presence_penalty": 1.0,
    "reasoning_effort": "low",
    "repetition_ngram_size": 1,
```

(continues on next page)

(continued from previous page)

```
    "repetition_ngram_threshold": 1,
    "repetition_penalty": 1.0,
    "response_format": {
        "json_schema": {
            "description": "string",
            "name": "string",
            "schema": {},
            "strict": true
        },
        "regex_schema": "string",
        "type": "text"
    },
    "return_token_ids": true,
    "seed": 1,
    "session_id": 1,
    "skip_special_tokens": true,
    "spaces_between_special_tokens": true,
    "stop": "string",
    "stream": true,
    "stream_options": {
        "include_usage": true
    },
    "temperature": 1.0,
    "tool_choice": {
        "function": {
            "name": "string"
        },
        "type": "string"
    },
    "tools": [
        {
            "function": {
                "description": "string",
                "name": "string",
                "parameters": {}
            },
            "type": "string"
        }
    ],
    "top_k": 1,
    "top_logprobs": 1,
    "top_p": 1.0,
    "user": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

(continues on next page)

(continued from previous page)

```
{}
```

- 422 Unprocessable Entity – Validation Error

  Example response:

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: application/json

{
    "detail": [
        {
            "ctx": {},
            "input": {},
            "loc": [
                "string",
                1
            ],
            "msg": "string",
            "type": "string"
        }
    ]
}
```

`POST /v1/completions`

Completions V1

Completion API similar to OpenAI's API.

Go to https://platform.openai.com/docs/api-reference/completions/create for the API specification.

The request should be a JSON object with the following fields:

- **model** (str): model name. Available from /v1/models.
- **prompt** (str): the input prompt.
- **suffix** (str): The suffix that comes after a completion of inserted text.
- **max_completion_tokens** (int | None): output token nums. Default to None.
- **max_tokens** (int): output token nums. Default to 16. Deprecated: Use max_completion_tokens instead.
- **temperature** (float): to modulate the next token probability
- **top_p** (float): If set to float < 1, only the smallest set of most probable tokens with probabilities that add up to top_p or higher are kept for generation.
- **n** (int): How many chat completion choices to generate for each input message. **Only support one here**.
- **stream**: whether to stream the results or not. Default to false.
- **repetition_penalty** (float): The parameter for repetition penalty. 1.0 means no penalty
- **user** (str): A unique identifier representing your end-user.
- **stop** (str | list[str] | None): To stop generating further tokens. Only accept stop words that's encoded to one token idex.

Additional arguments supported by LMDeploy:

- **ignore_eos** (bool): indicator for ignoring eos
- **skip_special_tokens** (bool): Whether or not to remove special tokens in the decoding. Default to be True.
- **top_k** (int): The number of the highest probability vocabulary tokens to keep for top-k-filtering

Currently we do not support the following features:

- **logprobs** (not supported yet)
- **presence_penalty** (replaced with repetition_penalty)
- **frequency_penalty** (replaced with repetition_penalty)

Request body:


```
{
  "echo":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Echo"
  },
  "frequency_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.0,
    "title":"Frequency Penalty"
  },
  "ignore_eos":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Ignore Eos"
  },
  "logprobs":{
    "anyOf":[
      {
        "type":"integer"
```


(continues on next page)

(continued from previous page)


```
      },
      {
        "type":"null"
      }
    ],
    "title":"Logprobs"
  },
  "max_completion_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "description":"An upper bound for the number of tokens that can be generated␣
↪for a completion, including visible output tokens and reasoning tokens",
    "examples":[
      null
    ],
    "title":"Max Completion Tokens"
  },
  "max_tokens":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":16,
    "deprecated":true,
    "examples":[
      16
    ],
    "title":"Max Tokens"
  },
  "min_p":{
    "default":0.0,
    "title":"Min P",
    "type":"number"
  },
  "model":{
    "title":"Model",
    "type":"string"
  },
  "n":{
    "anyOf":[
      {
        "type":"integer"
```


(continues on next page)

(continued from previous page)

```
      },
      {
        "type":"null"
      }
    ],
    "default":1,
    "title":"N"
  },
  "presence_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":0.0,
    "title":"Presence Penalty"
  },
  "prompt":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{},
        "type":"array"
      }
    ],
    "title":"Prompt"
  },
  "repetition_ngram_size":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Size",
    "type":"integer"
  },
  "repetition_ngram_threshold":{
    "default":0,
    "minimum":0.0,
    "title":"Repetition Ngram Threshold",
    "type":"integer"
  },
  "repetition_penalty":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
```

(continues on next page)

(continued from previous page)


```
    ],
    "default":1.0,
    "title":"Repetition Penalty"
  },
  "return_token_ids":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Return Token Ids"
  },
  "seed":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "title":"Seed"
  },
  "session_id":{
    "anyOf":[
      {
        "type":"integer"
      },
      {
        "type":"null"
      }
    ],
    "default":-1,
    "title":"Session Id"
  },
  "skip_special_tokens":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Skip Special Tokens"
  },
  "spaces_between_special_tokens":{
```


(continues on next page)

(continued from previous page)


```
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":true,
    "title":"Spaces Between Special Tokens"
  },
  "stop":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "items":{
          "type":"string"
        },
        "type":"array"
      },
      {
        "type":"null"
      }
    ],
    "examples":[
      null
    ],
    "title":"Stop"
  },
  "stream":{
    "anyOf":[
      {
        "type":"boolean"
      },
      {
        "type":"null"
      }
    ],
    "default":false,
    "title":"Stream"
  },
  "stream_options":{
    "anyOf":[
      {
        "description":"The stream options.",
        "properties":{
          "include_usage":{
            "anyOf":[
              {
                "type":"boolean"
```


(continues on next page)

(continued from previous page)

```
            },
            {
              "type":"null"
            }
          ],
          "default":false,
          "title":"Include Usage"
        }
      },
      "title":"StreamOptions",
      "type":"object"
    },
    {
      "type":"null"
    }
  ],
  "examples":[
    null
  ]
},
"suffix":{
  "anyOf":[
    {
      "type":"string"
    },
    {
      "type":"null"
    }
  ],
  "title":"Suffix"
},
"temperature":{
  "anyOf":[
    {
      "type":"number"
    },
    {
      "type":"null"
    }
  ],
  "default":0.7,
  "title":"Temperature"
},
"top_k":{
  "anyOf":[
    {
      "type":"integer"
    },
    {
      "type":"null"
    }
  ],
```

(continues on next page)

(continued from previous page)


```
    "default":40,
    "title":"Top K"
  },
  "top_p":{
    "anyOf":[
      {
        "type":"number"
      },
      {
        "type":"null"
      }
    ],
    "default":1.0,
    "title":"Top P"
  },
  "user":{
    "anyOf":[
      {
        "type":"string"
      },
      {
        "type":"null"
      }
    ],
    "title":"User"
  }
}
```


Example request:

```
POST /v1/completions HTTP/1.1
Host: example.com
Content-Type: application/json

{
    "echo": true,
    "frequency_penalty": 1.0,
    "ignore_eos": true,
    "logprobs": 1,
    "max_completion_tokens": 1,
    "max_tokens": 1,
    "min_p": 1.0,
    "model": "string",
    "n": 1,
    "presence_penalty": 1.0,
    "prompt": "string",
    "repetition_ngram_size": 1,
    "repetition_ngram_threshold": 1,
    "repetition_penalty": 1.0,
    "return_token_ids": true,
    "seed": 1,
    "session_id": 1,
```

(continues on next page)

(continued from previous page)

```
    "skip_special_tokens": true,
    "spaces_between_special_tokens": true,
    "stop": "string",
    "stream": true,
    "stream_options": {
        "include_usage": true
    },
    "suffix": "string",
    "temperature": 1.0,
    "top_k": 1,
    "top_p": 1.0,
    "user": "string"
}
```

Status Codes

- 200 OK – Successful Response

  Example response:

  ```
  HTTP/1.1 200 OK
  Content-Type: application/json

  {}
  ```

- 422 Unprocessable Entity – Validation Error

  Example response:

  ```
  HTTP/1.1 422 Unprocessable Entity
  Content-Type: application/json

  {
      "detail": [
          {
              "ctx": {},
              "input": {},
              "loc": [
                  "string",
                  1
              ],
              "msg": "string",
              "type": "string"
          }
      ]
  }
  ```

`GET /v1/models`

Available Models

Show available models.

Example request:

```
GET /v1/models HTTP/1.1
Host: example.com
```

Status Codes

- 200 OK – Successful Response

Example response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{}
```

1.38 Command-line Tools

1.38.1 lmdeploy - CLI interface

The CLI provides a unified API for converting, compressing and deploying large language models.

```
lmdeploy [-h] [-v] {lite,serve,check_env,chat} ...
```

lmdeploy options

- ***-h***, ***--help*** - show this help message and exit
- ***-v***, ***--version*** - show program's version number and exit

lmdeploy lite

Compressing and accelerating LLMs with lmdeploy.lite module

```
lmdeploy lite [-h] {auto_awq,auto_gptq,calibrate,smooth_quant} ...
```

lmdeploy lite options

- ***-h***, ***--help*** - show this help message and exit

lmdeploy lite auto_awq

Perform weight quantization using AWQ algorithm.

```
lmdeploy lite auto_awq [-h] [--revision REVISION] [--download-dir DOWNLOAD_DIR]
                       [--work-dir WORK_DIR]
                       [--calib-dataset {wikitext2,c4,pileval,gsm8k,neuralmagic_
↪calibration,open-platypus,openwebtext}]
                       [--calib-samples CALIB_SAMPLES] [--calib-seqlen CALIB_SEQLEN]
                       [--batch-size BATCH_SIZE] [--search-scale]
```

(continues on next page)

(continued from previous page)

```
                        [--dtype {auto,float16,bfloat16}] [--trust-remote-code]
                        [--device DEVICE] [--w-bits W_BITS] [--w-sym]
                        [--w-group-size W_GROUP_SIZE]
                        model
```

lmdeploy lite auto_awq positional arguments

- ***model*** - The path of model in hf format (default: `None`)

lmdeploy lite auto_awq options

- ***-h***, ***--help*** - show this help message and exit
- ***--revision*** `REVISION` - The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- ***--download-dir*** `DOWNLOAD_DIR` - Directory to download and load the weights, default to the default cache directory of huggingface.
- ***--work-dir*** `WORK_DIR` - The working directory to save results (default: `./work_dir`)
- ***--calib-dataset*** `CALIB_DATASET` - The calibration dataset name. (default: `wikitext2`)
- ***--calib-samples*** `CALIB_SAMPLES` - The number of samples for calibration (default: `128`)
- ***--calib-seqlen*** `CALIB_SEQLEN` - The sequence length for calibration (default: `2048`)
- ***--batch-size*** `BATCH_SIZE` - The batch size for running the calib samples. Low GPU mem requires small batch_size. Large batch_size reduces the calibration time while costs more VRAM (default: `1`)
- ***--search-scale*** - Whether search scale ratio. Default to be disabled, which means only smooth quant with 0.5 ratio will be applied
- ***--dtype*** `DTYPE` - data type for model weights and activations. The `"auto"` option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. This option will be ignored if the model is a quantized model (default: `auto`)
- ***--trust-remote-code*** - Whether to trust remote code from model repositories.
- ***--device*** `DEVICE` - Device for weight quantization (cuda or npu) (default: `cuda`)
- ***--w-bits*** `W_BITS` - Bit number for weight quantization (default: `4`)
- ***--w-sym*** - Whether to do symmetric quantization
- ***--w-group-size*** `W_GROUP_SIZE` - Group size for weight quantization statistics (default: `128`)

lmdeploy lite auto_gptq

Perform weight quantization using GPTQ algorithm.

```
lmdeploy lite auto_gptq [-h] [--revision REVISION] [--work-dir WORK_DIR]
                        [--calib-dataset {wikitext2,c4,pileval,gsm8k,neuralmagic_
↪calibration,open-platypus,openwebtext}]
                        [--calib-samples CALIB_SAMPLES] [--calib-seqlen CALIB_SEQLEN]
                        [--batch-size BATCH_SIZE] [--dtype {auto,float16,bfloat16}]
                        [--trust-remote-code] [--w-bits W_BITS] [--w-group-size W_GROUP_
↪SIZE]
                        model
```

lmdeploy lite auto_gptq positional arguments

- ***model*** - The path of model in hf format (default: `None`)

lmdeploy lite auto_gptq options

- ***-h***, ***--help*** - show this help message and exit
- ***--revision*** `REVISION` - The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- ***--work-dir*** `WORK_DIR` - The working directory to save results (default: `./work_dir`)
- ***--calib-dataset*** `CALIB_DATASET` - The calibration dataset name. (default: `wikitext2`)
- ***--calib-samples*** `CALIB_SAMPLES` - The number of samples for calibration (default: `128`)
- ***--calib-seqlen*** `CALIB_SEQLEN` - The sequence length for calibration (default: `2048`)
- ***--batch-size*** `BATCH_SIZE` - The batch size for running the calib samples. Low GPU mem requires small batch_size. Large batch_size reduces the calibration time while costs more VRAM (default: `1`)
- ***--dtype*** `DTYPE` - data type for model weights and activations. The `"auto"` option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. This option will be ignored if the model is a quantized model (default: `auto`)
- ***--trust-remote-code*** - Whether to trust remote code from model repositories.
- ***--w-bits*** `W_BITS` - Bit number for weight quantization (default: `4`)
- ***--w-group-size*** `W_GROUP_SIZE` - Group size for weight quantization statistics (default: `128`)

lmdeploy lite calibrate

Perform calibration on a given dataset.

```
lmdeploy lite calibrate [-h] [--work-dir WORK_DIR]
                        [--calib-dataset {wikitext2,c4,pileval,gsm8k,neuralmagic_
↪calibration,open-platypus,openwebtext}]
                        [--calib-samples CALIB_SAMPLES] [--calib-seqlen CALIB_SEQLEN]
                        [--batch-size BATCH_SIZE] [--search-scale]
                        [--dtype {auto,float16,bfloat16}] [--trust-remote-code]
                        model
```

lmdeploy lite calibrate positional arguments

- ***model*** - The name or path of the model to be loaded (default: `None`)

lmdeploy lite calibrate options

- ***-h***, ***--help*** - show this help message and exit
- ***--work-dir*** `WORK_DIR` - The working directory to save results (default: `./work_dir`)
- ***--calib-dataset*** `CALIB_DATASET` - The calibration dataset name. (default: `wikitext2`)
- ***--calib-samples*** `CALIB_SAMPLES` - The number of samples for calibration (default: `128`)
- ***--calib-seqlen*** `CALIB_SEQLEN` - The sequence length for calibration (default: `2048`)
- ***--batch-size*** `BATCH_SIZE` - The batch size for running the calib samples. Low GPU mem requires small batch_size. Large batch_size reduces the calibration time while costs more VRAM (default: `1`)
- ***--search-scale*** - Whether search scale ratio. Default to be disabled, which means only smooth quant with 0.5 ratio will be applied
- ***--dtype*** `DTYPE` - data type for model weights and activations. The `"auto"` option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. This option will be ignored if the model is a quantized model (default: `auto`)
- ***--trust-remote-code*** - Whether to trust remote code from model repositories.

lmdeploy lite smooth_quant

Perform w8a8 quantization using SmoothQuant.

```
lmdeploy lite smooth_quant [-h] [--work-dir WORK_DIR] [--device DEVICE]
                           [--calib-dataset {wikitext2,c4,pileval,gsm8k,neuralmagic_
↪calibration,open-platypus,openwebtext}]
                           [--calib-samples CALIB_SAMPLES] [--calib-seqlen CALIB_SEQLEN]
                           [--batch-size BATCH_SIZE] [--search-scale]
                           [--dtype {auto,float16,bfloat16}]
                           [--quant-dtype {int8,float8_e4m3fn,float8_e5m2,fp8}]
                           [--revision REVISION] [--download-dir DOWNLOAD_DIR]
                           [--trust-remote-code]
                           model
```

lmdeploy lite smooth_quant positional arguments

- ***model*** - The name or path of the model to be loaded (default: `None`)

lmdeploy lite smooth_quant options

- ***-h***, ***--help*** - show this help message and exit
- ***--work-dir*** WORK_DIR - The working directory for outputs. defaults to "./work_dir"
- ***--device*** DEVICE - Device for weight quantization (cuda or npu) (default: cuda)
- ***--calib-dataset*** CALIB_DATASET - The calibration dataset name. (default: wikitext2)
- ***--calib-samples*** CALIB_SAMPLES - The number of samples for calibration (default: 128)
- ***--calib-seqlen*** CALIB_SEQLEN - The sequence length for calibration (default: 2048)
- ***--batch-size*** BATCH_SIZE - The batch size for running the calib samples. Low GPU mem requires small batch_size. Large batch_size reduces the calibration time while costs more VRAM (default: 1)
- ***--search-scale*** - Whether search scale ratio. Default to be disabled, which means only smooth quant with 0.5 ratio will be applied
- ***--dtype*** DTYPE - data type for model weights and activations. The "auto" option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. This option will be ignored if the model is a quantized model (default: auto)
- ***--quant-dtype*** QUANT_DTYPE - data type for the quantized model weights and activations.Note "fp8" is the short version of "float8_e4m3fn" (default: int8)
- ***--revision*** REVISION - The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- ***--download-dir*** DOWNLOAD_DIR - Directory to download and load the weights, default to the default cache directory of huggingface.
- ***--trust-remote-code*** - Whether to trust remote code from model repositories.

lmdeploy serve

Serve LLMs with openai API

```
lmdeploy serve [-h] {api_server,proxy} ...
```

lmdeploy serve options

- ***-h***, ***--help*** - show this help message and exit

lmdeploy serve api_server

Serve LLMs with restful api using fastapi.

```
lmdeploy serve api_server [-h] [--server-name SERVER_NAME] [--server-port SERVER_PORT]
                          [--allow-origins ALLOW_ORIGINS [ALLOW_ORIGINS ...]]
                          [--allow-credentials]
                          [--allow-methods ALLOW_METHODS [ALLOW_METHODS ...]]
                          [--allow-headers ALLOW_HEADERS [ALLOW_HEADERS ...]]
                          [--proxy-url PROXY_URL]
                          [--max-concurrent-requests MAX_CONCURRENT_REQUESTS]
```

(continues on next page)

(continued from previous page)

```
                            [--backend {pytorch,turbomind}]
                            [--log-level {CRITICAL,FATAL,ERROR,WARN,WARNING,INFO,DEBUG,
↪NOTSET}]
                            [--api-keys [API_KEYS ...]] [--ssl] [--model-name MODEL_NAME]
                            [--max-log-len MAX_LOG_LEN] [--disable-fastapi-docs]
                            [--allow-terminate-by-client] [--enable-abort-handling]
                            [--trust-remote-code] [--chat-template CHAT_TEMPLATE]
                            [--tool-call-parser TOOL_CALL_PARSER]
                            [--reasoning-parser REASONING_PARSER] [--revision REVISION]
                            [--download-dir DOWNLOAD_DIR] [--adapters [ADAPTERS ...]]
                            [--device {cuda,ascend,maca,camb}] [--eager-mode]
                            [--disable-vision-encoder]
                            [--logprobs-mode {None,raw_logits,raw_logprobs}]
                            [--dllm-block-length DLLM_BLOCK_LENGTH]
                            [--dllm-unmasking-strategy {low_confidence_dynamic,low_
↪confidence_static,sequential}]
                            [--dllm-denoising-steps DLLM_DENOISING_STEPS]
                            [--dllm-confidence-threshold DLLM_CONFIDENCE_THRESHOLD]
                            [--enable-return-routed-experts]
                            [--distributed-executor-backend {uni,mp,ray}]
                            [--kernel-block-size KERNEL_BLOCK_SIZE]
                            [--dtype {auto,float16,bfloat16}] [--tp TP]
                            [--session-len SESSION_LEN] [--max-batch-size MAX_BATCH_SIZE]
                            [--cache-max-entry-count CACHE_MAX_ENTRY_COUNT]
                            [--cache-block-seq-len CACHE_BLOCK_SEQ_LEN]
                            [--enable-prefix-caching]
                            [--max-prefill-token-num MAX_PREFILL_TOKEN_NUM]
                            [--quant-policy {QuantPolicy.NONE,QuantPolicy.INT4,QuantPolicy.
↪INT8,QuantPolicy.TURBO_QUANT}]
                            [--model-format {hf,awq,gptq,compressed-tensors,fp8,mxfp4}]
                            [--hf-overrides HF_OVERRIDES] [--disable-metrics] [--dp DP]
                            [--ep EP] [--enable-microbatch] [--enable-eplb]
                            [--role {Hybrid,Prefill,Decode}]
                            [--migration-backend {DLSlime,Mooncake}] [--node-rank NODE_
↪RANK]
                            [--nnodes NNODES] [--cp CP]
                            [--rope-scaling-factor ROPE_SCALING_FACTOR]
                            [--num-tokens-per-iter NUM_TOKENS_PER_ITER]
                            [--max-prefill-iters MAX_PREFILL_ITERS] [--async {0,1}]
                            [--communicator {nccl,native,cuda-ipc}]
                            [--dist-init-addr DIST_INIT_ADDR]
                            [--vision-max-batch-size VISION_MAX_BATCH_SIZE]
                            [--speculative-algorithm {eagle,eagle3,deepseek_mtp,qwen3_5_
↪mtp}]
                            [--speculative-draft-model SPECULATIVE_DRAFT_MODEL]
                            [--speculative-num-draft-tokens SPECULATIVE_NUM_DRAFT_TOKENS]
                            model_path
```

lmdeploy serve api_server positional arguments

- ***model_path*** - The path of a model. it could be one of the following options: - i) a local directory path of a turbomind model which is converted by *lmdeploy convert* command or download from ii) and iii). - ii) the model_id of a lmdeploy-quantized model hosted inside a model repo on huggingface.co, such as "internlm/internlm-chat-20b-4bit", "lmdeploy/llama2-chat-70b-4bit", etc. - iii) the model_id of a model hosted inside a model repo on huggingface.co, such as "internlm/internlm-chat-7b", "qwen/qwen-7b-chat ", "baichuan-inc/baichuan2-7b-chat" and so on (default: None)

lmdeploy serve api_server options

- ***-h***, ***--help*** - show this help message and exit
- ***--server-name*** SERVER_NAME - Host ip for serving (default: 0.0.0.0)
- ***--server-port*** SERVER_PORT - Server port (default: 23333)
- ***--allow-origins*** ALLOW_ORIGINS - A list of allowed origins for cors (default: ['*'])
- ***--allow-credentials*** - Whether to allow credentials for cors
- ***--allow-methods*** ALLOW_METHODS - A list of allowed http methods for cors (default: ['*'])
- ***--allow-headers*** ALLOW_HEADERS - A list of allowed http headers for cors (default: ['*'])
- ***--proxy-url*** PROXY_URL - The proxy url for api server. (default: None)
- ***--max-concurrent-requests*** MAX_CONCURRENT_REQUESTS - This refers to the number of concurrent requests that the server can handle. The server is designed to process the engine's tasks once the maximum number of concurrent requests is reached, regardless of any additional requests sent by clients concurrently during that time. Default to None. (default: None)
- ***--backend*** BACKEND - Set the inference backend (default: turbomind)
- ***--log-level*** LOG_LEVEL - Set the log level (default: WARNING)
- ***--api-keys*** API_KEYS - Optional list of space separated API keys (default: None)
- ***--ssl*** - Enable SSL. Requires OS Environment variables 'SSL_KEYFILE' and 'SSL_CERTFILE'
- ***--model-name*** MODEL_NAME - The name of the served model. It can be accessed by the RESTful API */v1/models*. If it is not specified, *model_path* will be adopted (default: None)
- ***--max-log-len*** MAX_LOG_LEN - Max number of prompt characters or prompt tokens being printed in log. Default: Unlimited (default: None)
- ***--disable-fastapi-docs*** - Disable FastAPI's OpenAPI schema, Swagger UI, and ReDoc endpoint
- ***--allow-terminate-by-client*** - Enable server to be terminated by request from client
- ***--enable-abort-handling*** - Enable server to handle client abort requests
- ***--trust-remote-code*** - Whether to trust remote code from model repositories.
- ***--chat-template*** CHAT_TEMPLATE - A JSON file or string that specifies the chat template configuration. Please refer to https://lmdeploy.readthedocs.io/en/latest/advance/chat_template.html for the specification (default: None)
- ***--tool-call-parser*** TOOL_CALL_PARSER - The registered tool parser name dict_keys(['glm47', 'internlm', 'intern-s1', 'qwen3coder', 'interns2-preview', 'llama3', 'qwen2d5', 'qwen', 'qwen3']). Default to None. (default: None)

- ***--reasoning-parser*** REASONING_PARSER - The registered reasoning parser name: dict_keys(['default', 'deepseek-v3']). Legacy names: ['qwen-qwq', 'intern-s1', 'deepseek-r1']. Default to None. (default: None)
- ***--revision*** REVISION - The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- ***--download-dir*** DOWNLOAD_DIR - Directory to download and load the weights, default to the default cache directory of huggingface.

lmdeploy serve api_server PyTorch engine arguments

- ***--adapters*** ADAPTERS - Used to set path(s) of lora adapter(s). One can input key-value pairs in xxx=yyy format for multiple lora adapters. If only have one adapter, one can only input the path of the adapter. (default: None)
- ***--device*** DEVICE - The device type of running (default: cuda)
- ***--eager-mode*** - Whether to enable eager mode. If True, cuda graph would be disabled
- ***--disable-vision-encoder*** - disable multimodal encoder
- ***--logprobs-mode*** LOGPROBS_MODE - The mode of logprobs. (default: None)
- ***--dllm-block-length*** DLLM_BLOCK_LENGTH - Block length for dllm (default: None)
- ***--dllm-unmasking-strategy*** DLLM_UNMASKING_STRATEGY - The unmasking strategy for dllm. (default: low_confidence_dynamic)
- ***--dllm-denoising-steps*** DLLM_DENOISING_STEPS - The number of denoising steps for dllm. (default: None)
- ***--dllm-confidence-threshold*** DLLM_CONFIDENCE_THRESHOLD - The confidence threshold for dllm. (default: 0.85)
- ***--enable-return-routed-experts*** - Whether to output routed expert ids for replay
- ***--distributed-executor-backend*** DISTRIBUTED_EXECUTOR_BACKEND - The distributed executor backend for pytorch engine. (default: None)
- ***--kernel-block-size*** KERNEL_BLOCK_SIZE - The length of the token sequence in a k/v block for kernels. Only supported by Pytorch Engine. When set to a different value than –cache-block-seq-len, memory allocators and prefix cache use –cache-block-seq-len as the block size, while kernels use –kernel-block-size. (default: -1)
- ***--dtype*** DTYPE - data type for model weights and activations. The "auto" option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. This option will be ignored if the model is a quantized model (default: auto)
- ***--tp*** TP - GPU number used in tensor parallelism. Should be 2^n (default: 1)
- ***--session-len*** SESSION_LEN - The max session length of a sequence (default: None)
- ***--max-batch-size*** MAX_BATCH_SIZE - Maximum batch size. If not specified, the engine will automatically set it according to the device (default: None)
- ***--cache-max-entry-count*** CACHE_MAX_ENTRY_COUNT - The percentage of free gpu memory occupied by the k/v cache, excluding weights (default: 0.8)
- ***--cache-block-seq-len*** CACHE_BLOCK_SEQ_LEN - The length of the token sequence in a k/v block. For Turbomind Engine, if the GPU compute capability is >= 8.0, it should be a multiple of 32, otherwise it should be a multiple of 64. For Pytorch Engine, if Lora Adapter is specified, this parameter will be ignored (default: 64)
- ***--enable-prefix-caching*** - Enable cache and match prefix

- *--max-prefill-token-num* MAX_PREFILL_TOKEN_NUM - the max number of tokens per iteration during prefill (default: 8192)
- *--quant-policy* QUANT_POLICY - KV cache quantization policy. 0: no quantization; 4: 4-bit; 8: 8-bit; 42: TurboQuant (K4V2) (default: 0)
- *--model-format* MODEL_FORMAT - The format of input model. *hf* means *hf_llama*, *awq* and *gptq* refer to 4-bit grouped quantization, *compressed-tensors* refers to pack-quantized grouped int4 checkpoints and is usually auto-detected from the model config, *fp8* refers to blocked fp8 checkpoints, and *mxfp4* refers to MXFP4 expert weights. (default: None)
- *--hf-overrides* HF_OVERRIDES - Extra arguments to be forwarded to the HuggingFace config. (default: None)
- *--disable-metrics* - disable metrics system
- *--dp* DP - data parallelism. dp_rank is required when pytorch engine is used. (default: 1)
- *--ep* EP - expert parallelism. dp is required when pytorch engine is used. (default: 1)
- *--enable-microbatch* - enable microbatch for specified model
- *--enable-eplb* - enable eplb for specified model
- *--role* ROLE - Hybrid for Non-Disaggregated Engine; Prefill for Disaggregated Prefill Engine; Decode for Disaggregated Decode Engine (default: Hybrid)
- *--migration-backend* MIGRATION_BACKEND - kvcache migration management backend when PD disaggregation (default: DLSlime)
- *--node-rank* NODE_RANK - The current node rank. (default: 0)
- *--nnodes* NNODES - The total node nums (default: 1)

lmdeploy serve api_server TurboMind engine arguments

- *--dtype* DTYPE - data type for model weights and activations. The "auto" option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. This option will be ignored if the model is a quantized model (default: auto)
- *--tp* TP - GPU number used in tensor parallelism. Should be 2^n (default: 1)
- *--session-len* SESSION_LEN - The max session length of a sequence (default: None)
- *--max-batch-size* MAX_BATCH_SIZE - Maximum batch size. If not specified, the engine will automatically set it according to the device (default: None)
- *--cache-max-entry-count* CACHE_MAX_ENTRY_COUNT - The percentage of free gpu memory occupied by the k/v cache, excluding weights (default: 0.8)
- *--cache-block-seq-len* CACHE_BLOCK_SEQ_LEN - The length of the token sequence in a k/v block. For Turbomind Engine, if the GPU compute capability is >= 8.0, it should be a multiple of 32, otherwise it should be a multiple of 64. For Pytorch Engine, if Lora Adapter is specified, this parameter will be ignored (default: 64)
- *--enable-prefix-caching* - Enable cache and match prefix
- *--max-prefill-token-num* MAX_PREFILL_TOKEN_NUM - the max number of tokens per iteration during prefill (default: 8192)
- *--quant-policy* QUANT_POLICY - KV cache quantization policy. 0: no quantization; 4: 4-bit; 8: 8-bit; 42: TurboQuant (K4V2) (default: 0)
- *--model-format* MODEL_FORMAT - The format of input model. *hf* means *hf_llama*, *awq* and *gptq* refer to 4-bit grouped quantization, *compressed-tensors* refers to pack-quantized grouped int4 checkpoints and is usually

auto-detected from the model config, *fp8* refers to blocked fp8 checkpoints, and *mxfp4* refers to MXFP4 expert weights. (default: None)

- ***--nnodes*** NNODES - The total node nums (default: 1)
- ***--node-rank*** NODE_RANK - The current node rank. (default: 0)
- ***--hf-overrides*** HF_OVERRIDES - Extra arguments to be forwarded to the HuggingFace config. (default: None)
- ***--disable-metrics*** - disable metrics system
- ***--dp*** DP - data parallelism. dp_rank is required when pytorch engine is used. (default: 1)
- ***--cp*** CP - context parallelism size in attention for turbomind backend, tp must be a multiple of cp. (default: 1)
- ***--rope-scaling-factor*** ROPE_SCALING_FACTOR - Rope scaling factor (default: 0.0)
- ***--num-tokens-per-iter*** NUM_TOKENS_PER_ITER - the number of tokens processed in a forward pass (default: 0)
- ***--max-prefill-iters*** MAX_PREFILL_ITERS - the max number of forward passes in prefill stage (default: 1)
- ***--async*** ASYNC_ - Enable async execution (default: 1, enabled). Set to 0 to disable async mode, 1 to enable it. (default: 1)
- ***--communicator*** COMMUNICATOR - Communication backend for multi-GPU inference. The "native" option is deprecated and serves as an alias for "cuda-ipc" (default: nccl)
- ***--dist-init-addr*** DIST_INIT_ADDR (default: None)

lmdeploy serve api_server Vision model arguments

- ***--vision-max-batch-size*** VISION_MAX_BATCH_SIZE - the vision model batch size (default: 1)

lmdeploy serve api_server Speculative decoding arguments

- ***--speculative-algorithm*** SPECULATIVE_ALGORITHM - The speculative algorithm to use. *None* means speculative decoding is disabled (default: None)
- ***--speculative-draft-model*** SPECULATIVE_DRAFT_MODEL - The path to speculative draft model (default: None)
- ***--speculative-num-draft-tokens*** SPECULATIVE_NUM_DRAFT_TOKENS - The number of speculative tokens to generate per step (default: 1)

lmdeploy serve proxy

Proxy server that manages distributed api_server nodes.

```
lmdeploy serve proxy [-h] [--server-name SERVER_NAME] [--server-port SERVER_PORT]
                     [--serving-strategy {Hybrid,DistServe}] [--dummy-prefill]
                     [--routing-strategy {random,min_expected_latency,min_observed_
↪latency}]
                     [--disable-cache-status] [--migration-protocol {RDMA,NVLINK}]
                     [--link-type {RoCE,IB}] [--disable-gdr] [--api-keys [API_KEYS ...]]
                     [--ssl]
                     [--log-level {CRITICAL,FATAL,ERROR,WARN,WARNING,INFO,DEBUG,NOTSET}]
```

lmdeploy serve proxy options

- ***-h***, ***--help*** - show this help message and exit
- ***--server-name*** SERVER_NAME - Host ip for proxy serving (default: `0.0.0.0`)
- ***--server-port*** SERVER_PORT - Server port of the proxy (default: `8000`)
- ***--serving-strategy*** SERVING_STRATEGY - the strategy to serve, Hybrid for colocating Prefill and Decode-workloads into same engine, DistServe for Prefill-Decode Disaggregation (default: `Hybrid`)
- ***--dummy-prefill*** - dummy prefill for performance profiler
- ***--routing-strategy*** ROUTING_STRATEGY - the strategy to dispatch requests to nodes (default: `min_expected_latency`)
- ***--disable-cache-status*** - Whether to disable cache status of the proxy. If set, the proxy will forget the status of the previous time
- ***--migration-protocol*** MIGRATION_PROTOCOL - transport protocol of KV migration (default: `RDMA`)
- ***--link-type*** LINK_TYPE - RDMA Link Type (default: `RoCE`)
- ***--disable-gdr*** - with GPU Direct Memory Access
- ***--api-keys*** API_KEYS - Optional list of space separated API keys (default: `None`)
- ***--ssl*** - Enable SSL. Requires OS Environment variables `'SSL_KEYFILE'` and `'SSL_CERTFILE'`
- ***--log-level*** LOG_LEVEL - Set the log level (default: `WARNING`)

lmdeploy check_env

Check the environmental information.

```
lmdeploy check_env [-h] [--dump-file DUMP_FILE]
```

lmdeploy check_env options

- ***-h***, ***--help*** - show this help message and exit
- ***--dump-file*** DUMP_FILE - The file path to save env info. Only support file format in *json*, *yml*, *pkl* (default: `None`)

lmdeploy chat

```
lmdeploy chat [-h] [--backend {pytorch,turbomind}] [--chat-template CHAT_TEMPLATE]
              [--revision REVISION] [--download-dir DOWNLOAD_DIR] [--trust-remote-code]
              [--adapters [ADAPTERS ...]] [--device {cuda,ascend,maca,camb}] [--eager-
↪mode]
              [--dllm-block-length DLLM_BLOCK_LENGTH] [--dtype {auto,float16,bfloat16}]
              [--tp TP] [--session-len SESSION_LEN]
              [--cache-max-entry-count CACHE_MAX_ENTRY_COUNT] [--enable-prefix-caching]
              [--quant-policy {QuantPolicy.NONE,QuantPolicy.INT4,QuantPolicy.INT8,
↪QuantPolicy.TURBO_QUANT}]
              [--model-format {hf,awq,gptq,compressed-tensors,fp8,mxfp4}]
```

(continues on next page)

(continued from previous page)

```
              [--rope-scaling-factor ROPE_SCALING_FACTOR]
              [--communicator {nccl,native,cuda-ipc}] [--cp CP] [--async {0,1}]
              [--speculative-algorithm {eagle,eagle3,deepseek_mtp,qwen3_5_mtp}]
              [--speculative-draft-model SPECULATIVE_DRAFT_MODEL]
              [--speculative-num-draft-tokens SPECULATIVE_NUM_DRAFT_TOKENS]
              model_path
```

lmdeploy chat positional arguments

- ***model_path*** - The path of a model. it could be one of the following options: - i) a local directory path of a turbomind model which is converted by *lmdeploy convert* command or download from ii) and iii). - ii) the model_id of a lmdeploy-quantized model hosted inside a model repo on huggingface.co, such as "internlm/internlm-chat-20b-4bit", "lmdeploy/llama2-chat-70b-4bit", etc. - iii) the model_id of a model hosted inside a model repo on huggingface.co, such as "internlm/internlm-chat-7b", "qwen/qwen-7b-chat ", "baichuan-inc/baichuan2-7b-chat" and so on (default: None)

lmdeploy chat options

- ***-h***, ***--help*** - show this help message and exit
- ***--backend*** BACKEND - Set the inference backend (default: turbomind)
- ***--chat-template*** CHAT_TEMPLATE - A JSON file or string that specifies the chat template configuration. Please refer to https://lmdeploy.readthedocs.io/en/latest/advance/chat_template.html for the specification (default: None)
- ***--revision*** REVISION - The specific model version to use. It can be a branch name, a tag name, or a commit id. If unspecified, will use the default version.
- ***--download-dir*** DOWNLOAD_DIR - Directory to download and load the weights, default to the default cache directory of huggingface.
- ***--trust-remote-code*** - Whether to trust remote code from model repositories.

lmdeploy chat PyTorch engine arguments

- ***--adapters*** ADAPTERS - Used to set path(s) of lora adapter(s). One can input key-value pairs in xxx=yyy format for multiple lora adapters. If only have one adapter, one can only input the path of the adapter. (default: None)
- ***--device*** DEVICE - The device type of running (default: cuda)
- ***--eager-mode*** - Whether to enable eager mode. If True, cuda graph would be disabled
- ***--dllm-block-length*** DLLM_BLOCK_LENGTH - Block length for dllm (default: None)
- ***--dtype*** DTYPE - data type for model weights and activations. The "auto" option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. This option will be ignored if the model is a quantized model (default: auto)
- ***--tp*** TP - GPU number used in tensor parallelism. Should be 2^n (default: 1)
- ***--session-len*** SESSION_LEN - The max session length of a sequence (default: None)
- ***--cache-max-entry-count*** CACHE_MAX_ENTRY_COUNT - The percentage of free gpu memory occupied by the k/v cache, excluding weights (default: 0.8)

- ***--enable-prefix-caching*** - Enable cache and match prefix
- ***--quant-policy*** QUANT_POLICY - KV cache quantization policy. 0: no quantization; 4: 4-bit; 8: 8-bit; 42: TurboQuant (K4V2) (default: 0)

lmdeploy chat TurboMind engine arguments

- ***--dtype*** DTYPE - data type for model weights and activations. The "auto" option will use FP16 precision for FP32 and FP16 models, and BF16 precision for BF16 models. This option will be ignored if the model is a quantized model (default: auto)
- ***--tp*** TP - GPU number used in tensor parallelism. Should be 2^n (default: 1)
- ***--session-len*** SESSION_LEN - The max session length of a sequence (default: None)
- ***--cache-max-entry-count*** CACHE_MAX_ENTRY_COUNT - The percentage of free gpu memory occupied by the k/v cache, excluding weights (default: 0.8)
- ***--enable-prefix-caching*** - Enable cache and match prefix
- ***--quant-policy*** QUANT_POLICY - KV cache quantization policy. 0: no quantization; 4: 4-bit; 8: 8-bit; 42: TurboQuant (K4V2) (default: 0)
- ***--model-format*** MODEL_FORMAT - The format of input model. *hf* means *hf_llama*, *awq* and *gptq* refer to 4-bit grouped quantization, *compressed-tensors* refers to pack-quantized grouped int4 checkpoints and is usually auto-detected from the model config, *fp8* refers to blocked fp8 checkpoints, and *mxfp4* refers to MXFP4 expert weights. (default: None)
- ***--rope-scaling-factor*** ROPE_SCALING_FACTOR - Rope scaling factor (default: 0.0)
- ***--communicator*** COMMUNICATOR - Communication backend for multi-GPU inference. The "native" option is deprecated and serves as an alias for "cuda-ipc" (default: nccl)
- ***--cp*** CP - context parallelism size in attention for turbomind backend, tp must be a multiple of cp. (default: 1)
- ***--async*** ASYNC_ - Enable async execution (default: 1, enabled). Set to 0 to disable async mode, 1 to enable it. (default: 1)

lmdeploy chat Speculative decoding arguments

- ***--speculative-algorithm*** SPECULATIVE_ALGORITHM - The speculative algorithm to use. *None* means speculative decoding is disabled (default: None)
- ***--speculative-draft-model*** SPECULATIVE_DRAFT_MODEL - The path to speculative draft model (default: None)
- ***--speculative-num-draft-tokens*** SPECULATIVE_NUM_DRAFT_TOKENS - The number of speculative tokens to generate per step (default: 1)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- search
- routingtable

HTTP ROUTING TABLE

INDEX

Symbols

C

G

I

P

S

T