1.llm推理框架简单总结

下面首先来总结一下这些框架的特点，如下表所示：

Framework	Tokens Per Second	Query per second	Latency	Adapters	Quantisation	Variable precision	Batching	Distributed Inference	Custom models	Token streaming	Prometheus metrics
vLLM	115	0.94	4.8 s	✗	✗	✗	✓	✓	✓	✓	✗
Text generation inference	50	0.26	4.8 s	✗	✓	✓	✓	✓	✓	✓	✓
CTranslate2	93	0.55	4.5 s	✗	✓	✓	✓	✓	✓	✓	✗
DeepSpeed-MII	80	0.47	2.5 s	✗	✗	✓	✓	✓	✗	✗	✗
OpenLLM	30	0.15	6.6 s	✓	✓	✓	✗	✗	✓	✗	✓
Ray Serve	28	0.15	7.1 s	✗	✓	✓	✓	✓	✓	✗	✓
MLC LLM	25	0.13	7.2 s	✗	✓	✗	✗	✗	✓	✓	✗

LLM推理有很多框架，各有其特点，下面分别介绍一下表中七个框架的关键点：

1. vLLM：适用于大批量Prompt输入，并对推理速度要求高的场景；
2. Text generation inference：依赖HuggingFace模型，并且不需要为核心模型增加多个adapter的场景；
3. CTranslate2：可在CPU上进行推理；
4. OpenLLM：为核心模型添加adapter并使用HuggingFace Agents，尤其是不完全依赖PyTorch；
5. Ray Serve：稳定的Pipeline和灵活的部署，它最适合更成熟的项目；
6. MLC LLM：可在客户端（边缘计算）（例如，在Android或iPhone平台上）本地部署LLM；
7. DeepSpeed-MII：使用DeepSpeed库来部署LLM；

下面在内存容量为40GB的A100 GPU上，并且使用LLaMA-1 13b模型（因为列表中的所有库都支持它）进行七个部署框架的对比。

1.vLLM

vLLM的吞吐量比HuggingFace Transformers（HF）高14x-24倍，比HuggingFace Text Generation Inference（TGI）高2.2x-2.5倍。

1.1 使用

离线批量推理

```
# pip install vllm
from vllm import LLM, SamplingParams

prompts = [
    "Funniest joke ever:",
    "The capital of France is",
```

```
    "The future of AI is",
]
sampling_params = SamplingParams(temperature=0.95, top_p=0.95, max_tokens=200)
llm = LLM(model="huggyllama/llama-13b")
outputs = llm.generate(prompts, sampling_params)

for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt!r}, Generated text: {generated_text!r}")Copy to
clipboardErrorCopiedCopy to clipboardErrorCopied
```

API Server

```
# Start the server:
python -m vllm.entrypoints.api_server --env MODEL_NAME=huggyllama/llama-13b

# Query the model in shell:
curl http://localhost:8000/generate \
    -d '{
        "prompt": "Funniest joke ever:",
        "n": 1,
        "temperature": 0.95,
        "max_tokens": 200
    }'
```

1.2 功能

问题背景：传统推理为何“卡”在大量请求下?

在传统的 LLM 推理系统中（比如 Hugging Face Transformers）：

- 每个请求的 **KV 缓存**（Key-Value Cache）必须预先分配一块**连续的 GPU 内存**，大小等于最大可能的上下文长度（比如 2048 tokens）。
- 即使一个请求只用了 100 个 token，系统也会为它预留 2048 个 token 的空间 → **内存浪费严重。**
- 批处理（batching）是“静态”的：一批请求必须**同时开始、同时结束。**如果某个请求早早生成完（比如只输出 10 个 token），它仍占用 GPU 资源直到整批中最长的请求结束 → **GPU 利用率低。**
- 当请求量大、长度差异大时，GPU 很快被“碎片化”或“占满”，无法塞进新请求 → **吞吐骤降，延迟飙升。**

vllm的功能:

- Continuous batching：有iteration-level的调度机制，每次迭代batch大小都有所变化，因此vLLM在大量查询下仍可以很好的工作。

> **Note**
>
> **Continuous Batching：每次迭代都“动态调仓”**
>
> > 也叫 **iteration-level scheduling**
>
> - 不再等待整批请求全部完成才处理下一批。
> - 每生成一个 token 后
>
>   ，系统立即检查：

    - 哪些请求已经结束？→ **释放它们的内存页。**
    - 哪些新请求来了？→ **立刻塞进当前 batch，只要还有空闲页。**
  - 所有请求"异步"推进，互不阻塞。

- PagedAttention：受操作系统中虚拟内存和分页的经典思想启发的注意力算法，这就是模型加速的秘诀。

> **Note**
>
> **PagedAttention：像操作系统一样管理内存**
>
> > **灵感来源**：操作系统的 **虚拟内存 + 分页机制**（paging）
>
> - 将每个请求的 KV 缓存**切分成固定大小的"页"（blocks）**，比如每页 16 个 token。
> - 这些页**不需要在 GPU 内存中连续存放**，就像硬盘上的文件可以分散存储一样。
> - 系统维护一个"页表"（block table），记录逻辑页 → 物理页的映射。
> - **内存按需分配**：生成第 17 个 token 时，才分配第 2 个页；不需要提前预留 2048 个 token 的空间。

内存共享：并行采样也不怕

- 当一个 prompt 需要生成多个输出（如 `n=3` 个回答），传统方法会为每个输出**重复计算并存储相同的 prompt KV 缓存。**
- vLLM 通过 PagedAttention 的**页表共享机制**，让多个输出**共享 prompt 对应的物理页。**
- 采用 **Copy-on-Write**（写时复制）策略，确保安全。

1.3 优点

- **文本生成的速度**： 实验多次，发现vLLM的推理速度是最快的；
- **高吞吐量服务**： 支持各种解码算法，比如parallel sampling, beam search等；
- **与OpenAI API兼容**： 如果使用OpenAI API，只需要替换端点的URL即可；

1.4 缺点

- **添加自定义模型**：虽然可以合并自己的模型，但如果模型没有使用与vLLM中现有模型类似的架构，则过程会变得更加复杂。例如，增加Falcon的支持，这似乎很有挑战性；
- **缺乏对适配器（LoRA、QLoRA等）的支持**：当针对特定任务进行微调时，开源LLM具有重要价值。然而，在当前的实现中，没有单独使用模型和适配器权重的选项，这限制了有效利用此类模型的灵活性。
- **缺少权重量化**：有时，LLM可能不需要使用GPU内存，这对于减少GPU内存消耗至关重要。**如果能对模型权重进行量化（如 INT8、INT4），就可以显著减少 GPU 内存占用，从而在相同硬件上服务更多请求。而 vLLM 早期版本不支持权重量化，因此在内存受限场景下不如支持量化的框架灵活**

这是LLM推理最快的库。得益于其内部优化，它显著优于竞争对手。尽管如此，它在支持有限范围的模型方面确实存在弱点。

使用vLLM的开发路线可以参考：https://github.com/vllm-project/vllm/issues/244

2.Text generation inference

Text generation inference是用于文本生成推断的Rust、Python和gRPC服务器，在HuggingFace中已有LLM 推理API使用。

这其实是说 **TGI 的架构由多种语言和技术组成**，各司其职：

1. **Rust**：高性能核心调度器

- TGI 的 **调度逻辑、批处理引擎、内存管理** 等关键性能模块是用 **Rust** 写的。
- 为什么用 Rust?
  - **内存安全**（不会像 C/C++ 那样容易出错）
  - **高性能**（接近 C，远快于 Python）
  - **适合系统级编程**（比如管理 GPU 请求队列、实现 continuous batching）
- Rust 部分通过 **PyO3** 桥接，被 Python 调用。

2. **Python**：模型加载与推理接口

- 实际的 **模型加载、前向计算（forward pass）** 仍然用 **PyTorch + Transformers**（Python 生态）。
- Python 负责：
  - 加载 Hugging Face Hub 上的模型
  - 调用 CUDA 进行 GPU 推理
  - 处理 tokenizer、生成逻辑等
- Rust 调度器会把请求“交给”Python 执行推理。

3. **gRPC**：高性能通信协议

- TGI 对外提供 **gRPC 接口**（同时也支持 HTTP/REST）。
- gRPC 是什么?
  - 由 Google 开发的 **远程过程调用**（RPC）框架。
  - 基于 **Protocol Buffers**（protobuf）定义接口，**二进制传输**，比 JSON+HTTP 更快、更省带宽。
  - 支持 **流式响应**（streaming），非常适合 LLM 逐 token 生成的场景。
- 举例：你的客户端可以“订阅”一个流，TGI 一边生成 token 一边推送给你，而不用等全部生成完。

> 所以，TGI 既支持：
>
> - **HTTP/REST API**（方便调试，兼容 OpenAI 格式）
> - **gRPC API**（高性能、低延迟，适合内部微服务调用）

2.1使用

使用docker运行web server

```
mkdir data
docker run --gpus all --shm-size 1g -p 8080:80 \
-v data:/data ghcr.io/huggingface/text-generation-inference:0.9 \
  --model-id huggyllama/llama-13b \
  --num-shard 1
```

查询实例

```
# pip install text-generation
from text_generation import Client

client = Client("http://127.0.0.1:8080")
prompt = "Funniest joke ever:"
print(client.generate(prompt, max_new_tokens=17 temperature=0.95).generated_text)
```

2.2功能

- **内置服务评估**：可以监控服务器负载并深入了解其性能；
- **使用flash attention（和v2）优化transformer推理代码**： 并非所有模型都内置了对这些优化的支持，该技术可以对未使用该技术的模型可以进行优化；TGI 使用 FlashAttention（v1/v2）等优化技术来加速 Attention 计算。这些优化是在**推理引擎层实现的**，因此即使原始模型（如 Hugging Face 上的 LLaMA）没有内置 FlashAttention，TGI 也能在加载时自动启用它，从而提升推理速度和显存效率。

功能	说明	用户价值
内置服务评估	提供 Prometheus metrics、结构化日志、实时负载监控	便于生产部署、故障排查、容量规划
自动应用 FlashAttention	即使原始模型没优化，TGI 也能在推理时启用 FlashAttention 加速	开箱即用高性能，无需找“魔改版”模型

2.3 优点

- **所有的依赖项都安装在Docker中：** 会得到一个现成的环境；
- **支持HuggingFace模型：** 轻松运行自己的模型或使用任何HuggingFace模型中心；
- **对模型推理的控制**：该框架提供了一系列管理模型推理的选项，包括精度调整、量化、张量并行性、重复惩罚等；

2.4 缺点

- **缺乏对适配器的支持：** 需要注意的是，尽管可以使用适配器部署LLM（可以参考 https://www.youtube.com/watch?v=HI3cYN0c9ZU），但目前还没有官方支持或文档；
- **从源代码（Rust+CUDA内核）编译：** 对于不熟悉Rust的人，将客户化代码纳入库中变得很有挑战性；
- **文档不完整**：所有信息都可以在项目的自述文件中找到。尽管它涵盖了基础知识，但必须在问题或源代码中搜索更多细节；

使用Text generation inference的开发路线可以参考：https://github.com/huggingface/text-generation-inference/issues/232

3.CTranslate2

CTranslate2是一个C++和Python库，用于使用Transformer模型进行高效推理。**CTranslate2** 是由 OpenNMT 团队开发的一个**专为 Transformer 模型设计的高性能推理引擎**，主要特点包括：

- **不依赖 PyTorch/TensorFlow**：它使用自己的轻量级运行时（基于 C++ 和自定义 CUDA kernel）。
- **支持 CPU 和 GPU**（NVIDIA CUDA）。
- **专为推理优化**：训练不行，但推理极快。
- **模型需转换格式**：原始 PyTorch/HF 模型需先用 `ct2-transformers-converter` 转为 CTranslate2 的二进制格式（`.ct2`）。
- **广泛用于生产**：Hugging Face 的 `transformers` 库可通过 `optimum` 集成 CTranslate2；也被用于 Whisper、MarianMT、LLaMA 等模型的高效部署。

3.1 使用

转换模型

```
pip install -qqq transformers ctranslate2

# The model should be first converted into the CTranslate2 model format:
ct2-transformers-converter --model huggyllama/llama-13b --output_dir llama-13b-ct2 --force
```

查询实例

```
import ctranslate2
import transformers

generator = ctranslate2.Generator("llama-13b-ct2", device="cuda", compute_type="float16")
tokenizer = transformers.AutoTokenizer.from_pretrained("huggyllama/llama-13b")

prompt = "Funniest joke ever:"
tokens = tokenizer.convert_ids_to_tokens(tokenizer.encode(prompt))
results = generator.generate_batch(
    [tokens],
    sampling_topk=1,
    max_length=200,
)
tokens = results[0].sequences_ids[0]
output = tokenizer.decode(tokens)
print(output)
```

3.2 功能

- **在CPU和GPU上快速高效地执行：** 得益于内置的一系列优化：层融合、填充去除、批量重新排序、原位操作、缓存机制等。推理LLM更快，所需内存更少；

> **Note**
>
> 1. **层融合**（Layer Fusion）
>
> - **问题**：标准 Transformer 中，每个子层（如 Linear → ReLU → Dropout）是分开执行的，每次都要读写中间结果，增加内存带宽压力。
> - CTranslate2 做法：将多个操作融合成一个 CUDA kernel 或 CPU 函数。
>   - 例如：`MatMul + Bias + GELU` 合并为一个 kernel。
> - 效果：
>   - 减少 GPU/CPU 的 kernel 启动开销；
>   - 减少中间张量的显存/内存占用；
>   - 提高计算密度 → **提速 + 省内存。**
>
> 2. **填充去除**（Padding Removal）
>
> - **问题**：在 batch 推理中，短序列会被 pad 到最长序列长度，导致大量"无效计算"（对 pad token 做 attention 和 FFN）。
> - CTranslate2 做法：
>   - 在 batch 内**动态移除 padding**；
>   - 使用 **packed representation**（紧凑表示）只处理有效 token。
> - 效果：
>   - 避免对 pad token 的无意义计算；
>   - 尤其在输入长度差异大时（如聊天场景），**提速显著**（实测可快 2–5 倍）。
>
> **3. 批量重新排序（Batch Reordering）**
>
> - **问题**：如果 batch 中序列长度差异大，GPU 并行效率低（"长拖短"）。
> - CTranslate2 做法：
>   - 在 batch 内按长度**排序或分组**（如 bucketing）；
>   - 减少 padding，提升计算一致性。
> - **效果**：配合 padding removal，进一步提升硬件利用率。
>
> 4. **原位操作**（In-place Operations）
>
> - **问题**：很多操作（如激活函数）会创建新张量，增加内存峰值。
> - CTranslate2 做法：
>   - 尽可能**复用输入内存**进行计算（如 `x = gelu(x)` 直接覆盖原内存）。
> - **效果**：**显著降低内存峰值**，尤其在长上下文推理中。
>
> 5. **缓存机制**（KV Cache Optimization）
>
> - 虽然 CTranslate2 早期主要用于翻译（非自回归），但对 LLM（自回归生成）也做了优化：
>   - 高效管理 **Key-Value 缓存**；
>   - 支持增量解码（incremental decoding）；

> - 缓存结构紧凑，减少内存碎片。
>
> 🌟 注意：CTranslate2 **没有 PagedAttention**（那是 vLLM 的专利），但它通过上述通用优化，在 CPU 上表现尤为突出。

- **动态内存使用率：** 由于CPU和GPU上都有缓存分配器，内存使用率根据请求大小动态变化，同时仍能满足性能要求；
- **支持多种CPU体系结构：** 该项目支持x86-64和AArch64/ARM64处理器，并集成了针对这些平台优化的多个后端：英特尔MKL、oneDNN、OpenBLAS、Ruy和Apple Accelerate；

3.3 优点

- **并行和异步执行**：可以使用多个GPU或CPU核心并行和异步处理多个批处理；
- **Prompt缓存**：在静态提示下运行一次模型，缓存模型状态，并在将来使用相同的静态提示进行调用时重用；
- **磁盘上的轻量级**：量化可以使模型在磁盘上缩小4倍，而精度损失最小；

3.4 缺点

- **没有内置的REST服务器**：尽管仍然可以运行REST服务器，但没有具有日志记录和监控功能的现成服务
- **缺乏对适配器（LoRA、QLoRA等）的支持**

4.DeepSpeed-MII

在DeepSpeed支持下，DeepSpeed-MII可以进行低延迟和高通量推理。

4.1 使用

运行web服务

```
# DON'T INSTALL USING pip install deepspeed-mii
# git clone https://github.com/microsoft/DeepSpeed-MII.git
# git reset --hard 60a85dc3da5bac3bcefa8824175f8646a0f12203
# cd DeepSpeed-MII && pip install .
# pip3 install -U deepspeed

# ... and make sure that you have same CUDA versions:
# python -c "import torch;print(torch.version.cuda)" == nvcc --version
import mii

mii_configs = {
    "dtype": "fp16",
    'max_tokens': 200,
    'tensor_parallel': 1,
    "enable_load_balancing": False
}
mii.deploy(task="text-generation",
```

```
            model="huggyllama/llama-13b",
            deployment_name="llama_13b_deployment",
            mii_config=mii_configs)Copy to clipboardErrorCopiedCopy to
clipboardErrorCopied
```

查询实例

```
import mii

generator = mii.mii_query_handle("llama_13b_deployment")
result = generator.query(
  {"query": ["Funniest joke ever:"]},
  do_sample=True,
  max_new_tokens=200
)
print(result)
```

4.2 功能

- **多个副本上的负载平衡：** 这是一个非常有用的工具，可用于处理大量用户。负载均衡器在各种副本之间高效地分配传入请求，从而缩短了应用程序的响应时间。

> **Note**
>
> - 它内置了一个智能负载均衡器（Load Balancer）：
>   - 自动将新请求分配给**当前最空闲的副本**（比如队列最短、GPU 利用率最低）。
>   - 支持多种策略：轮询（round-robin）、最少连接（least connections）等。
> - 结果：
>   - 所有 GPU **并行工作，无闲置**；
>   - 用户请求**平均响应时间显著下降**；
>   - 系统整体吞吐线性扩展（2 副本 ≈ 2 倍吞吐）。

- **非持久部署：** 目标环境的部署不是永久的，需要经常更新的，这在资源效率、安全性、一致性和易管理性至关重要的情况下，这是非常重要的。

> **Tip**
>
> - 指的是：模型服务不是“一直开着、永不重启”的，而是：
>   - 按需启动（on-demand）；
>   - 使用完可自动释放资源；
>   - 支持快速滚动更新（比如换新模型版本）；
>   - 甚至可以“用完即焚”（ephemeral）。

4.3 优点

- **支持不同的模型库：** 支持多个开源模型库，如Hugging Face、FairSeq、EluetherAI等；
- **量化延迟和降低成本：** 可以显著降低非常昂贵的语言模型的推理成本；
- **Native和Azure集成：** 微软开发的MII框架提供了与云系统的出色集成；

4.4 缺点

- **支持模型的数量有限：** 不支持Falcon、LLaMA2和其他语言模型；
- **缺乏对适配器（LoRA、QLoRA等）的支持；**

5.OpenLLM

OpenLLM是一个用于在生产中操作大型语言模型（LLM）的开放平台。

5.1 使用

运行web服务

```
pip install openllm scipy
openllm start llama --model-id huggyllama/llama-13b \
  --max-new-tokens 200 \
  --temperature 0.95 \
  --api-workers 1 \
  --workers-per-resource 1Copy to clipboardErrorCopiedCopy to
clipboardErrorCopied
```

查询实例

```
import openllm

client = openllm.client.HTTPClient('http://localhost:3000')
print(client.query("Funniest joke ever:"))Copy to clipboardErrorCopiedCopy to
clipboardErrorCopied
```

5.2 功能

- **适配器支持：** 可以将要部署的LLM连接多个适配器，这样可以只使用一个模型来执行几个特定的任务；
- **支持不同的运行框架：** 比如Pytorch（pt）、Tensorflow（tf）或Flax（亚麻）；
- **HuggingFace Agents：** 连接HuggingFace上不同的模型，并使用LLM和自然语言进行管理；

5.3 优点

- **良好的社区支持：** 不断开发和添加新功能；
- **集成新模型：** 可以添加用户自定义模型；
- **量化：** OpenLLM支持使用bitsandbytes[12]和GPTQ[13]进行量化；

> **Note**
>
> - **bitsandbytes**（简称 bnb）是一个开源 Python 库，由 Tim Dettmers 团队开发，专注于**高效、低精度的神经网络计算。**

- 它最广为人知的功能是实现了 **8-bit 和 4-bit 量化**，尤其以 **LLM.int8()** 和 **4-bit NormalFloat（NF4）** 闻名。
- 与 Hugging Face Transformers 深度集成，只需一行代码即可启用。

核心技术亮点：

1. LLM.int8（8-bit）：
   - 将权重拆分为“大值”和“小值”两部分；
   - 大值用 int8 存储，小值保留 float16；
   - 在推理时动态反量化，几乎**无精度损失**（尤其对 >6B 的模型）。
2. **4-bit NormalFloat** (NF4)：
   - 专为 LLM 设计的 4-bit 数据类型；
   - 比传统 int4 更适合权重分布（非对称、重尾）；
   - 配合 **Double Quantization**（对量化参数再量化），进一步节省内存。

> **Tip**
>
> - **GPTQ** 是一种**后训练量化**（Post-Training Quantization, PTQ）方法，专为 **decoder-only LLM**（如 LLaMA、OPT、Falcon）设计。
> - 目标：将模型权重压缩到 **4-bit**，同时**最小化精度损失。**
> - 由论文《GPTQ: Accurate Post-Training Quantization for Generative Pretrained Transformers》提出。
>
> 核心思想：
>
> - **逐层量化**：一次只量化一层，用少量校准数据（如 128 个样本）补偿量化误差；
> - **贪心优化**：对每个权重，选择最优的 4-bit 表示，使得输出误差最小；
> - **保留 scale/zero-point**：类似 int4，但更精细。

- **LangChain集成：** 可以使用LangChain与远程OpenLLM服务器进行交互；

5.4 缺点

- **缺乏批处理支持：** 对于大量查询，这很可能会成为应用程序性能的瓶颈；
- **缺乏内置的分布式推理**：如果你想在多个GPU设备上运行大型模型，你需要额外安装OpenLLM的服务组件Yatai；

6.Ray Serve

Ray Serve是一个可扩展的模型服务库，用于构建在线推理API。Serve与框架无关，因此可以使用一个工具包来为深度学习模型的所有内容提供服务。**Ray Serve** 是构建在 **Ray 分布式计算框架** 之上的一个**可扩展、可编程的模型服务库**，专为部署机器学习和 LLM 应用而设计。

- 它不是传统意义上的“服务器软件”，而是一个 **Python 库**，让你用几行代码就能部署可伸缩的 API 服务。

- 支持 **单机到多机集群**，无缝扩展。
- 被 Hugging Face、Anyscale、vLLM（通过 Ray 集成）等广泛用于生产环境。

6.1 使用

运行web服务

```
# pip install ray[serve] accelerate>=0.16.0 transformers>=4.26.0 torch starlette
pandas
# ray_serve.py
import pandas as pd

import ray
from ray import serve
from starlette.requests import Request

@serve.deployment(ray_actor_options={"num_gpus": 1})
class PredictDeployment:
    def __init__(self, model_id: str):
        from transformers import AutoModelForCausalLM, AutoTokenizer
        import torch

        self.model = AutoModelForCausalLM.from_pretrained(
            model_id,
            torch_dtype=torch.float16,
            device_map="auto",
        )
        self.tokenizer = AutoTokenizer.from_pretrained(model_id)

    def generate(self, text: str) -> pd.DataFrame:
        input_ids = self.tokenizer(text, return_tensors="pt").input_ids.to(
            self.model.device
        )
        gen_tokens = self.model.generate(
            input_ids,
            temperature=0.9,
            max_length=200,
```

```
        )
        return pd.DataFrame(
            self.tokenizer.batch_decode(gen_tokens), columns=["responses"]
        )

    async def __call__(self, http_request: Request) -> str:
        json_request: str = await http_request.json()
        return self.generate(prompt["text"])

deployment = PredictDeployment.bind(model_id="huggyllama/llama-13b")

# then run from CLI command:
# serve run ray_serve:deployment
```

查询实例

```
import requests

sample_input = {"text": "Funniest joke ever:"}
output = requests.post("http://localhost:8000/", json=[sample_input]).json()
print(output)
```

6.2 功能

- **监控仪表板和Prometheus度量：** 可以使用Ray仪表板来获得Ray集群和Ray Serve应用程序状态；

> **Note**
>
> - Ray Dashboard（仪表板）：
>   - 启动 Ray 集群后，会自动开启一个 Web UI（默认 `http://localhost:8265`）。
>   - 你可以实时看到：
>   - 当前有多少 **Serve 副本**（replicas）在运行；
>   - 每个副本的 **CPU/GPU 利用率、内存占用**；
>   - **请求队列长度、吞吐量、延迟分布**；
>   - 任务失败日志、资源瓶颈等。
> - Prometheus Metrics：
>   - Ray Serve 自动暴露标准 Prometheus 指标（如 `serve_deployment_request_latency_ms`）。
>   - 可接入 Grafana，构建企业级监控告警系统。

- **跨多个副本自动缩放：** Ray通过观察队列大小并做出添加或删除副本的缩放决策来调整流量峰值；
- **动态请求批处理：** 当模型使用成本很高，为最大限度地利用硬件，可以采用该策略；

6.3 优点

- **文档支持：** 开发人员几乎为每个用例撰写了许多示例；
- **支持生产环境部署：** 这是本列表中所有框架中最成熟的；
- **本地LangChain集成：** 您可以使用LangChain与远程Ray Server进行交互；

6.4 缺点

- **缺乏内置的模型优化：** Ray Serve不专注于LLM，它是一个用于部署任何ML模型的更广泛的框架，必须自己进行优化；
- **入门门槛高：** 该库功能多，提高了初学者进入的门槛；

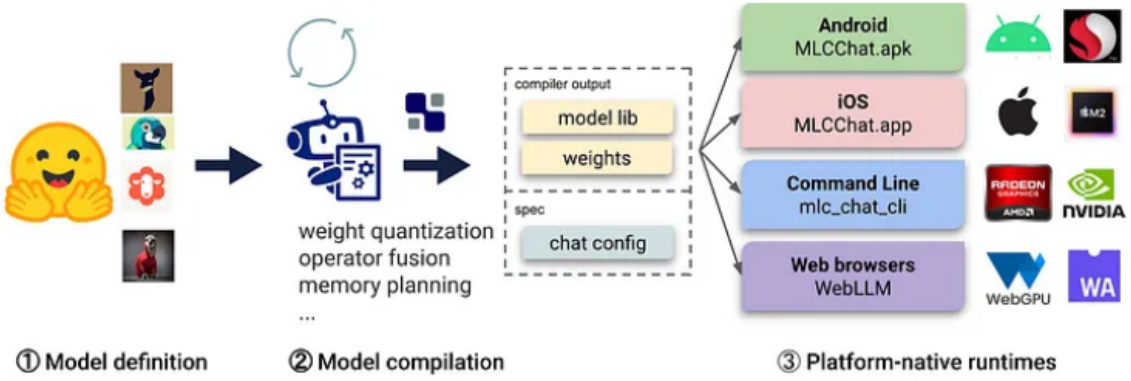
如果需要最适合生产的解决方案，而不仅仅是深度学习，Ray Serve是一个不错的选择。它最适合于可用性、可扩展性和可观察性非常重要的企业。此外，还可以使用其庞大的生态系统进行数据处理、模型训练、微调和服务。最后，从OpenAI到Shopify和Instacart等公司都在使用它。

7.MLC LLM

LLM的机器学习编译（MLC LLM）是一种通用的部署解决方案，MLC LLM 的目标是：**在手机、浏览器、嵌入式设备等无 GPU 环境下高效运行 LLM。**

为此，它做了深度工程优化：

- 使用 **Apache TVM / MLIR** 编译模型为设备原生代码（如 Metal for iPhone, Vulkan for Android, WebGPU for browser）；
- 量化必须**在编译时静态确定**，不能依赖运行时反量化（像 bitsandbytes 那样）；
- 分组量化（尤其是 4-bit）：
  - 精度高（接近 FP16）；
  - 内存占用极低（4-bit + 小 scale 张量）；
  - **可被 TVM 高效编译为硬件友好的 kernel。**



7.1 使用

运行web服务

```
# 1. Make sure that you have python >= 3.9
# 2. You have to run it using conda:
conda create -n mlc-chat-venv -c mlc-ai -c conda-forge mlc-chat-nightly
conda activate mlc-chat-venv

# 3. Then install package:
pip install --pre --force-reinstall mlc-ai-nightly-cu118 \
```

```
  mlc-chat-nightly-cu118 \
  -f https://mlc.ai/wheels

# 4. Download the model weights from HuggingFace and binary libraries:
git lfs install && mkdir -p dist/prebuilt && \
  git clone https://github.com/mlc-ai/binary-mlc-llm-libs.git dist/prebuilt/lib
&& \
  cd dist/prebuilt && \
  git clone https://huggingface.co/huggyllama/llama-13b dist/ && \
  cd ../..


# 5. Run server:
python -m mlc_chat.rest --device-name cuda --artifact-path dist
```

查询实例

```
import requests

payload = {
   "model": "lama-30b",
   "messages": [{"role": "user", "content": "Funniest joke ever:"}],
   "stream": False
}
r = requests.post("http://127.0.0.1:8000/v1/chat/completions", json=payload)
print(r.json()['choices'][0]['message']['content'])
```

7.2 功能

- **平台本机运行时：** 可以部署在用户设备的本机环境上，这些设备可能没有现成的Python或其他必要的依赖项。应用程序开发人员只需要将MLC编译的LLM集成到他们的项目中即可；
- **内存优化：** 可以使用不同的技术编译、压缩和优化模型，从而可以部署在不同的设备上；

7.3 优点

- **所有设置均可在JSON配置中完成：** 在单个配置文件中定义每个编译模型的运行时配置；
- **预置应用程序：** 可以为不同的平台编译模型，比如C++用于命令行，JavaScript用于web，Swift用于iOS，Java/Kotlin用于Android；

7.4 缺点

- **使用LLM模型的功能有限**：不支持适配器，无法更改精度等，该库主要用于编译不同设备的模型；
- **只支持分组量化：** 这种方法表现良好，但是在社区中更受欢迎的其他量化方法（bitsandbytes和GPTQ）不支持；

> **Note**
>
> **分组量化** 是一种比“逐层量化”（per-layer）更精细、比“逐通道量化”（per-channel）更高效、专为大模型设计的**4-bit 量化策略。**
>
> 核心思想：
>
> - 不是对整个权重矩阵用**同一个缩放因子**（scale）进行量化（那样会损失精度）；
> - 而是把权重矩阵**按行或按列切成小块**（比如每 128 个元素为一组）；

> - **每组独立计算自己的 scale 和 zero-point**，然后进行 4-bit 量化。
>
> > 🔍 举个例子：
> > 一个 Linear 层的权重是 `[1024, 4096]` 的矩阵。
> > 分组量化可能把它按行切成每组 128 列 → 共 32 组。
> > 每组有自己的 scale，比如第 1 组 scale=0.8，第 2 组 scale=1.2……
> > 这样能更准确地保留原始权重的分布特性。

- **复杂的安装：** 安装需要花几个小时，不太适合初学者开发人员；

如果需要在iOS或Android设备上部署应用程序，这个库正是你所需要的。它将允许您快速地以本机方式编译模型并将其部署到设备上。但是，如果需要一个高负载的服务器，不建议选择这个框架。

参考资料：

- Frameworks for Serving LLMs.
- LLM七种推理服务框架总结
- 目前业界大模型推理框架很多，各有什么优缺点，应该如何选择