Announcing Native LLM APIs in Ray Data and Ray Serve

By The Anyscale Team | April 2, 2025

Table of contents

Sharing

Sign up for product updates

Email*

Submit

Recommended content

Introducing the Ray for Practitioners Course and

Introduction

Today, we're excited to announce native APIs for LLM inference with Ray Data and Ray Serve.

As LLMs become increasingly central to modern AI infrastructure deployments, platforms require the ability to deploy and scale these models efficiently. While Ray Data and Ray Serve are suitable for this purpose, developers have to write a sizable amount of boilerplate in order to leverage the libraries for scaling LLM applications.

In Ray 2.44, we're announcing **Ray Data LLM** and **Ray Serve LLM**.

- **Ray Data LLM** provides APIs for offline batch inference with LLMs within existing Ray Data pipelines
- **Ray Serve LLM** provides APIs for deploying LLMs for online inference in Ray Serve applications.

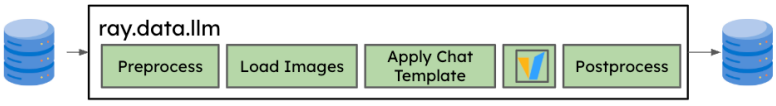
Both modules offer first-class integration for vLLM and OpenAI compatible endpoints.

Ray Data LLM

The `ray.data.llm` module integrates with key large language model (LLM) inference engines and deployed models to enable LLM batch inference.

Ray Data LLM is designed to address several common developer pains around batch inference:

- We saw that many users were building ad-hoc solutions for high-throughput batch inference. These solutions would entail launching many online inference servers and build extra proxying/load balancing utilities to maximize throughput. To address this, we wanted to leverage Ray Data and take advantage of pre-built distributed data loading and processing functionality.
- We saw common patterns of users sending batch data to an existing inference server. To address this, we wanted to make sure that users could integrate their data pipelines with an OpenAI compatible API endpoint, and provide the flexibility for the user to be able to templatize the query sent to the server.
- We saw that users were integrating LLMs into existing Ray Data pipelines (chaining LLM post-processing stages). To address this, we wanted to make sure that the API was compatible with the existing lazy and functional Ray Data API.



With Ray Data LLM, users create a Processor object, which can be called on a Ray Data Dataset and will return a Ray Data dataset. The processor object will contain configuration like:

- Prompt and template
- OpenAI compatible sampling parameters, which can be specified per row

- vLLM engine configuration, if applicable

```
import ray
from ray.data.llm import vLLMEngineProcessorCon
import numpy as np

config = vLLMEngineProcessorConfig(
    model="meta-llama/Llama-3.1-8B-Instruct",
    engine_kwargs={
        "enable_chunked_prefill": True,
        "max_num_batched_tokens": 4096,
        "max_model_len": 16384,
    },
    concurrency=1,
    batch_size=64,
)
processor = build_llm_processor(
    config,
    preprocess=lambda row: dict(
        messages=[
            {"role": "system", "content": "You
            {"role": "user", "content": row["it
        ],
        sampling_params=dict(
            temperature=0.3,
            max_tokens=250,
        )
    ),
    postprocess=lambda row: dict(
        answer=row["generated_text"],
        **row  # This will return all the origi
    ),
)

ds = ray.data.from_items(["Start of the haiku i

ds = processor(ds)
ds.show(limit=1)
```

In this particular example, the Processor object will:

- Perform the necessary preprocessing and postprocessing to handle LLM outputs properly
- Instantiate and configure multiple vLLM replicas, depending on specified concurrency and provided engine configurations. Each of these replicas can themselves be distributed as well.
- Continuously feed each replica by leveraging async actors in Ray to take advantage of continuous batching and maximize throughput
- Invoke various Ray Data methods (`map` , `map_batches`) which can be fused and optimized with other preprocessing stages in the pipeline by Ray Data during execution.

As you can see, Ray Data LLM can easily simplify the usage of LLMs within your existing data pipelines. See the documentation for more details.

Ray Serve LLM

Ray Serve LLM APIs allow users to deploy multiple LLM models together with a familiar Ray Serve API, while providing compatibility with the OpenAI API.

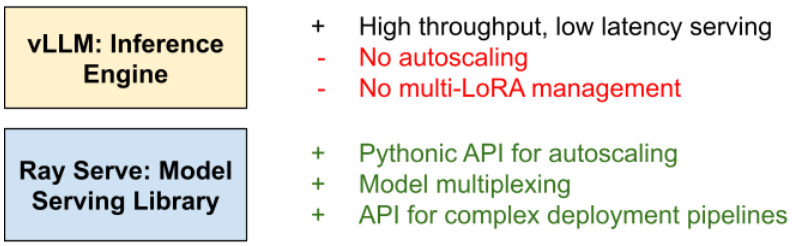
Ray Serve LLM is designed with the following features:

- Automatic scaling and load balancing
- Unified multi-node multi-model deployment
- OpenAI compatibility
- Multi-LoRA support with shared base models
- Deep integration with inference engines (vLLM to start)
- Composable multi-model LLM pipelines

While vLLM has grown rapidly over the last year, we have seen a significant uptick of users leveraging Ray Serve to deploy vLLM for

multiple models and program more complex pipelines.

For production deployments, Ray Serve + vLLM are great complements.



vLLM provides a simple abstraction layer to serve hundreds of different models with high throughput and low latency. However, vLLM is only responsible for single model replicas, and for production deployments you often need an orchestration layer to be able to autoscale, handle different fine-tuned adapters, handle distributed model-parallelism, and author multi-model, compound AI pipelines that can be quite complex.

Ray Serve is built to address the gaps that vLLM has for scaling and productionization. Ray Serve offers:

- Pythonic API for autoscaling
- Built-in support for model multiplexing
- Provides a Pythonic, imperative way to write complex multi-model / deployment pipelines
- Has first-class support for distributed model parallelism by leveraging Ray.

Below is a simple example of deploying a Qwen model with Ray Serve on a local machine with two GPUs behind an OpenAI-compatible router, then querying it with the OpenAI client.

```
1  from ray import serve
2  from ray.serve.llm import LLMConfig, LLMServer,
3
4  llm_config = LLMConfig(
```

```
    model_loading_config=dict(
        model_id="qwen-0.5b",
        model_source="Qwen/Qwen2.5-0.5B-Instruc
    ),
    deployment_config=dict(
        autoscaling_config=dict(
            min_replicas=1, max_replicas=2,
        )
    ),
    # Pass the desired accelerator type (e.g. A
    accelerator_type="A10G",
    # You can customize the engine arguments (e
    engine_kwargs=dict(
        tensor_parallel_size=2,
    ),
)

# Deploy the application
deployment = LLMServer.as_deployment(
    llm_config.get_serve_options(name_prefix="v
llm_app = LLMRouter.as_deployment().bind([deplo
serve.run(llm_app)
```

And then you can query this with the OpenAI Python API:

```
from openai import OpenAI

# Initialize client
client = OpenAI(base_url="http://localhost:8000

# Basic chat completion with streaming
response = client.chat.completions.create(
    model="qwen-0.5b",
    messages=[{"role": "user", "content": "Hell
    stream=True
)

for chunk in response:
    if chunk.choices[0].delta.content is not No
```

```
15         print(chunk.choices[0].delta.content, e
16
```

Visit the documentation for more details.

Ray Serve LLM can also be deployed on Kubernetes by using KubeRay. Take a look at the Ray Serve production guide for more details

Future Developments

Give these new features a spin and let us know your feedback! If you're interested in chatting with developers, feel free to join the Ray Slack or participate on Discourse, and follow the roadmap for Ray Serve LLM and Ray Data LLM here for future updates.