




UbuCon India²⁵

Scaling AI Infra: Data Pipelines, Orchestration, and Distributed Training with Ray

Prarabdha Srivastava , IISc Bangalore

Qualcomm

 Canonical

IT'S FOSS





Pinterest: “improve dataset **iteration speed** from **days to hours**, while improving our **GPU utilization to over 90%**”
“iterate quickly with **web-scale data**... thousands of features... petabytes of data”



Uber: “it has allowed us to significantly improve performance and fault tolerance, while also reducing the complexity of our technology stack.”



ByteDance

Canva

- OpenAI uses Ray to train some of the largest models, including ChatGPT, citing the framework’s ability to accelerate iteration at scale as a critical factor in their success.
- Uber has adopted Ray as the unified compute backend for its machine learning and deep learning platforms, praising its performance improvements and reduced complexity.
- Ant Group deployed Ray Serve on a massive scale for model serving during the world’s largest online shopping day, achieving unprecedented transaction throughput.



Link to the github repo used for this workshop

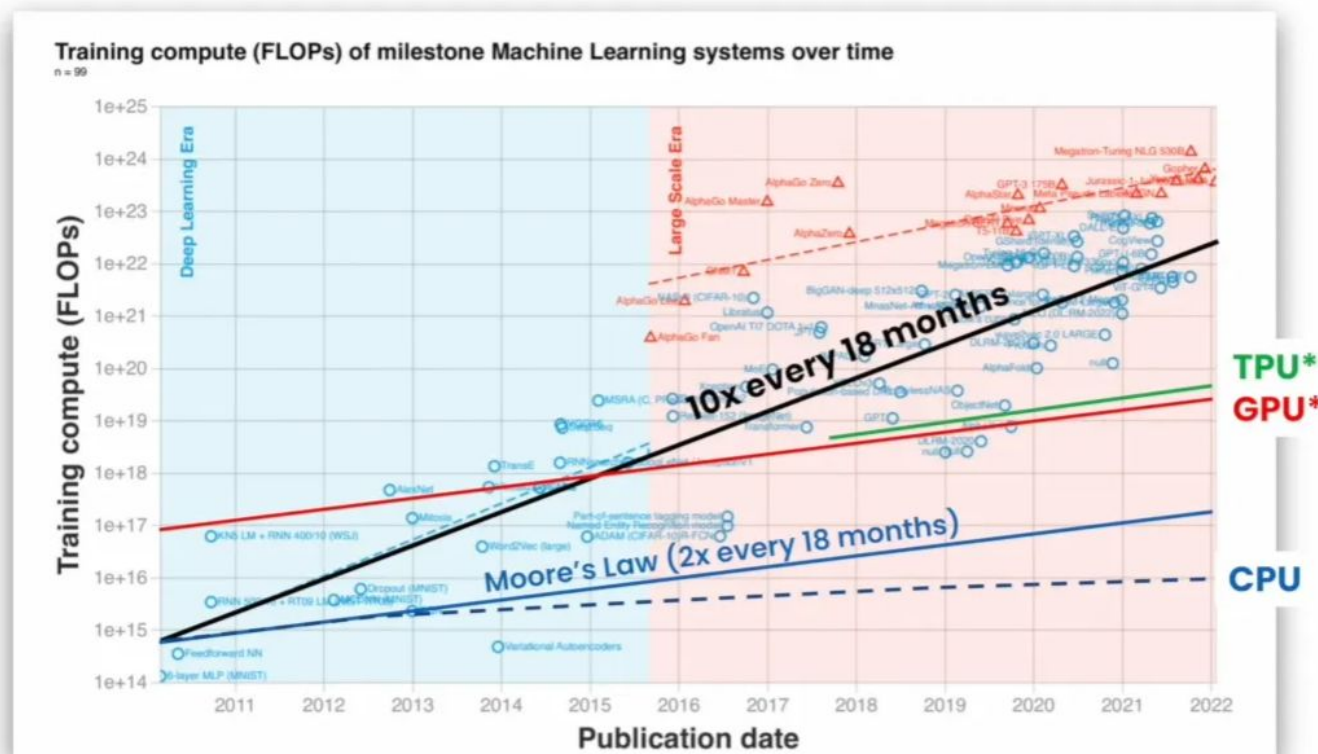
Why AI Infrastructure is the Real Bottleneck

Training AI models has become more accessible with advances in frameworks and hardware, but scaling the underlying infrastructure remains a critical challenge. Managing terabyte-scale multimodal datasets combining text and images demands robust, efficient pipelines and orchestration workflows. These are often underestimated in research and industry yet can make or break AI projects.

the cluster

- State-of-the-art: cluster data ingestion tasks and coordinate across data

Specialized hardware not good enough



"Compute trends across three eras of machine learning", J. Sevilla, <https://arxiv.org/html/2202.05924>

- Training is straightforward; scaling infrastructure is difficult and complex.
- Handling terabyte-scale text and image data pushes the limits of existing systems.
- Data ingestion, preprocessing, orchestration, and real-time inference require careful design for reliability and cost-effectiveness.

Common pain points include slow S3-based data ingestion, GPUs underutilized due to pipeline stalls, and frequent expensive reprocessing cycles caused by unclear orchestration.

Why Ray ?

Ray's Vision : Make Distributed Computing Accesible for every ml devs. So you can actually focus on ml

One Framework to scale everything - “Turn single-machine Python code into a distributed system — **without rewriting it.**”

Scale training, preprocessing, tuning, and serving using a single unified runtime.

Seamless Integration with the Tools You Already Use - “Ray doesn't replace your stack — it **supercharges** it.”

*Works out-of-the-box with: **PyTorch**, **TensorFlow**, **Transformers**, **scikit-learn**, **vLLM**, **PEFT**, **MLflow***

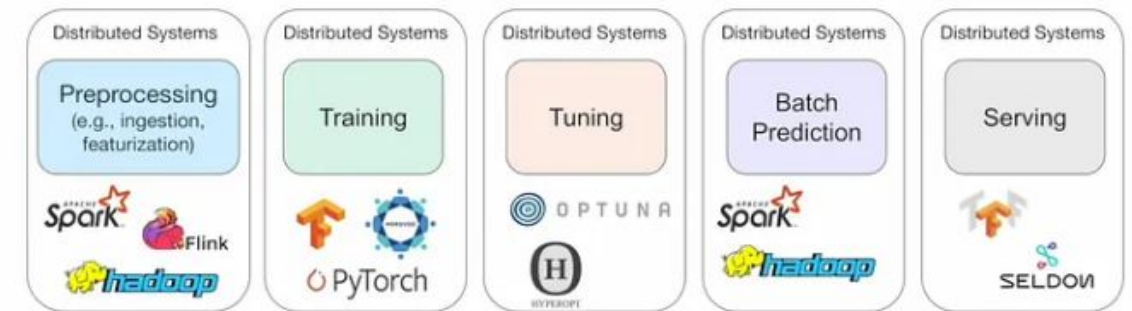
Fault-Tolerance, Checkpointing & Elastic Scaling — Built In - “Train at scale without fearing a crash.”

*Ray handles **distributed memory**, **communication**, and **resource scheduling** for you.*

Focus on ML — Not Infrastructure - “Ray abstracts away distributed systems so you can focus on models.”

*Ray handles **distributed memory**, **communication**, and **resource scheduling** for you.*

Challenge: need to scale every stage!



Need to stitch together a bunch of disparate systems

- **Hard** to develop
- **Hard** to deploy
- **Hard** to manage
- Slow

Ray Ecosystem

Ray Data - Distributed data loading and preprocessing (streams from S3, Parquet, etc.)

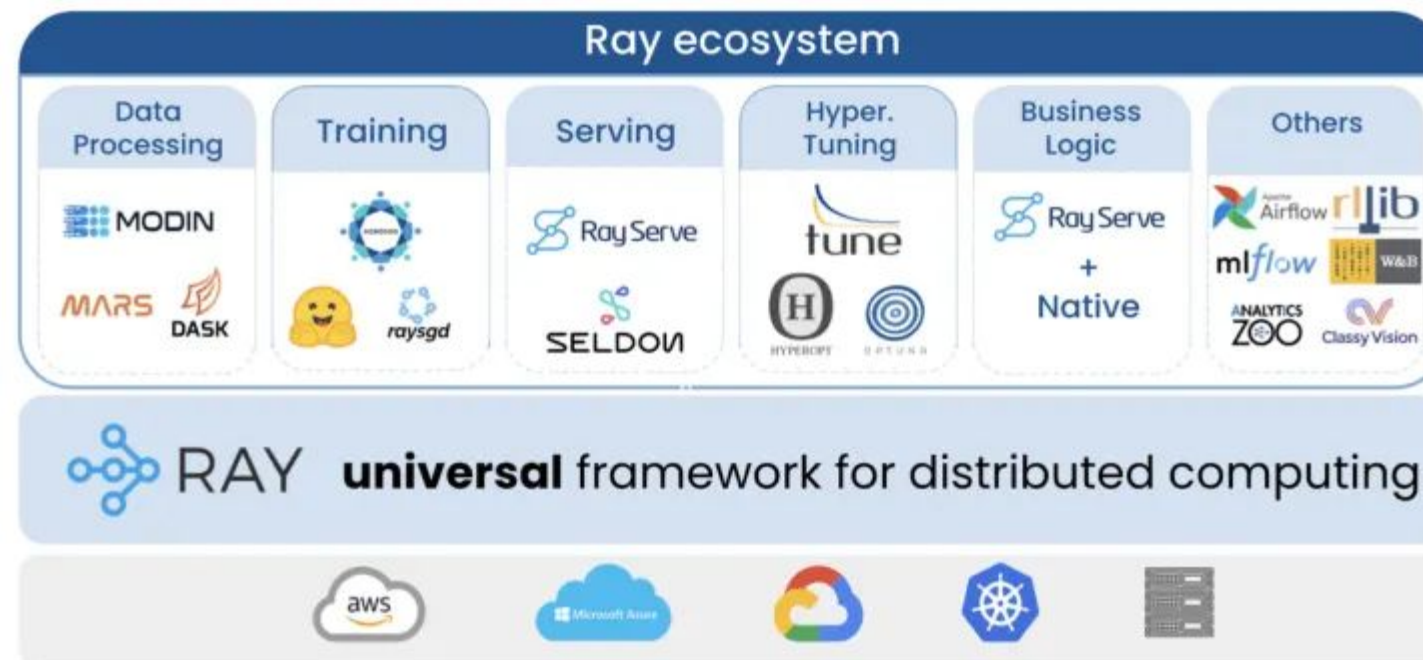
Ray Train - Multi-GPU / multi-node training (PyTorch, TF, XGBoost, PEFT)

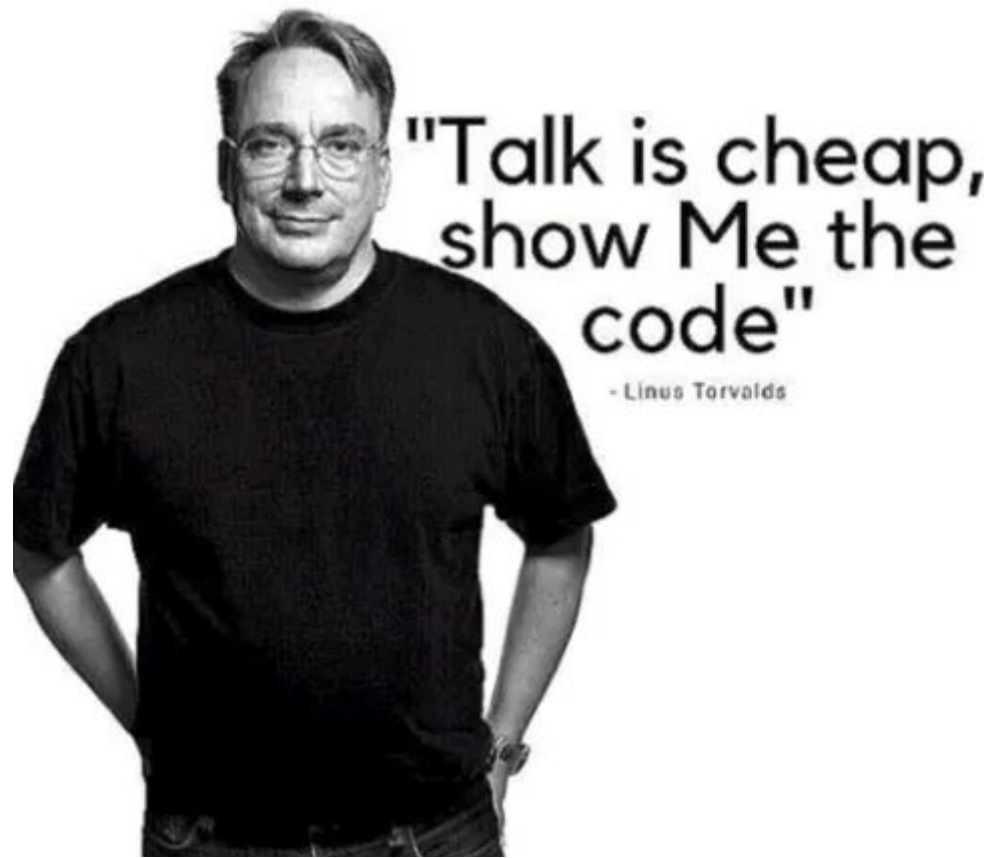
Ray Tune - Hyperparameter tuning at cluster scale

Ray Serve - Production-grade model serving with autoscaling

Ray AIR - Unified APIs for end-to-end workflows

RLlib - Industry grade Scalable Reinforcement Learning





Ray Core

Ok Mr. Linus , We will take a **slow single Threaded** Python Function

.... And **Transform** it step by step

... Into a **distributed** Application

```
8 def process_images(image: np.ndarray) -> np.ndarray:
9     '''A dummy slow image filter'''
10    time.sleep(1)
11    return 255 - image
```

1 Image = 1 seconds
10k Images = 10k seconds = 2 hours 45 mins

This is completely unacceptable for any real world application. How to resolve

Key Abstraction : **Ray Tasks**

A ray task is a python function marked as runnable in parallel

- initialize ray
- Decorate your function with `@ray.remote`
- Ray executes this function in a separate process in all the available cores
- Ray scheduler handles all the complexity of distributing the work

| Primitive | Role | Example |
|--------------------------|--|---|
| <code>ray.init()</code> | Starts the Ray runtime on your machine. | <code>ray.init()</code> |
| <code>@ray.remote</code> | A decorator that turns a Python function into a Ray Task. | <code>@ray.remote def my_func(): ...</code> |
| <code>.remote()</code> | The suffix used to execute a remote function asynchronously. | <code>my_func.remote()</code> |
| ObjectRef | A future/promise returned by a <code>.remote()</code> call. | <code>ref = my_func.remote()</code> |
| <code>ray.get()</code> | A blocking call to retrieve the actual result from an ObjectRef . | <code>result = ray.get(ref)</code> |

But just Like in real World , Every small win is immediately followed by a new , harder feature request

Now your manager wants a dashboard to see total number of pixels processed

... IN REAL TIME

Let's try to solve this in old fashioned pythonic way

To solve the problem of mutable shared state ...

We need a new Abstraction : **Ray Actors**

The Actor provided a centralized stateful service

.. That all our stateless tasks can communicate with

.... Correctly solving the **shared state problem**

The Reason for Failure: Process Isolation

Because each Ray Task runs in its **own separate process**.

Each process gets its **own copy** of the `total_pixels_processed` variable.

Each task increments its private copy from 0 to 300, which is then discarded.

The original variable in our main script is **never touched**

A Task

... is like a temporary gig worker

They do one job really fast, report the results and are gone. They have **no Memory**

An Actor

... is like a full time employee with an office

They are long running services that can **update it's private state** (memory)

Core Primitives for Ray Actor

A ray task is a python function marked as runnable in parallel

- Define Actor Class using `@ray.remote` decorator

```
@ray.remote
class PixelCounter:
    def __init__(self):
        # This is the actor's private, internal state
        self.total_pixels = 0

    def add(self, num_pixels: int):
        # This method will update the actors's state
        self.total_pixels += num_pixels

    def get_total(self) -> int:
        return self.total_pixels
```

- Create an instance to start the service

```
#This create the actor and returns a handle to it
counter = PixelCounter.remote()
```

- Ray executes this function in a separate process in all the available cores

```
@ray.remote
def process_images(image: np.ndarray , counter_actor: "ActorHandle") -> np.ndarray:
    '''A dummy slow image filter'''
    counter_actor.add.remote(image.size)
    time.sleep(1)
    return 255 - image
```

| Primitive | Role | Example |
|-------------------------------------|--|---|
| <code>@ray.remote</code> (on class) | Turns a Python class into a Ray Actor factory. | <code>@ray.remote class MyActor: ...</code> |
| Actor Handle | A remote control for a running actor instance. | <code>my_actor = MyActor.remote()</code> |
| <code>.remote()</code> (on method) | Asynchronously calls a method on an Actor. | <code>my_actor.my_method.remote(arg)</code> |

We now have parallel , stateful computation working perfectly on our machine

We can go **fast** and we can **coordinate**

But now our Workload is exploded with data. We are now not processing just 8 images but 100k images. Let's do the maths

8 Images on a 6 core cpu = 1.3 sec
100k images = 4 hours and 30 mins

Again this is not **acceptable**

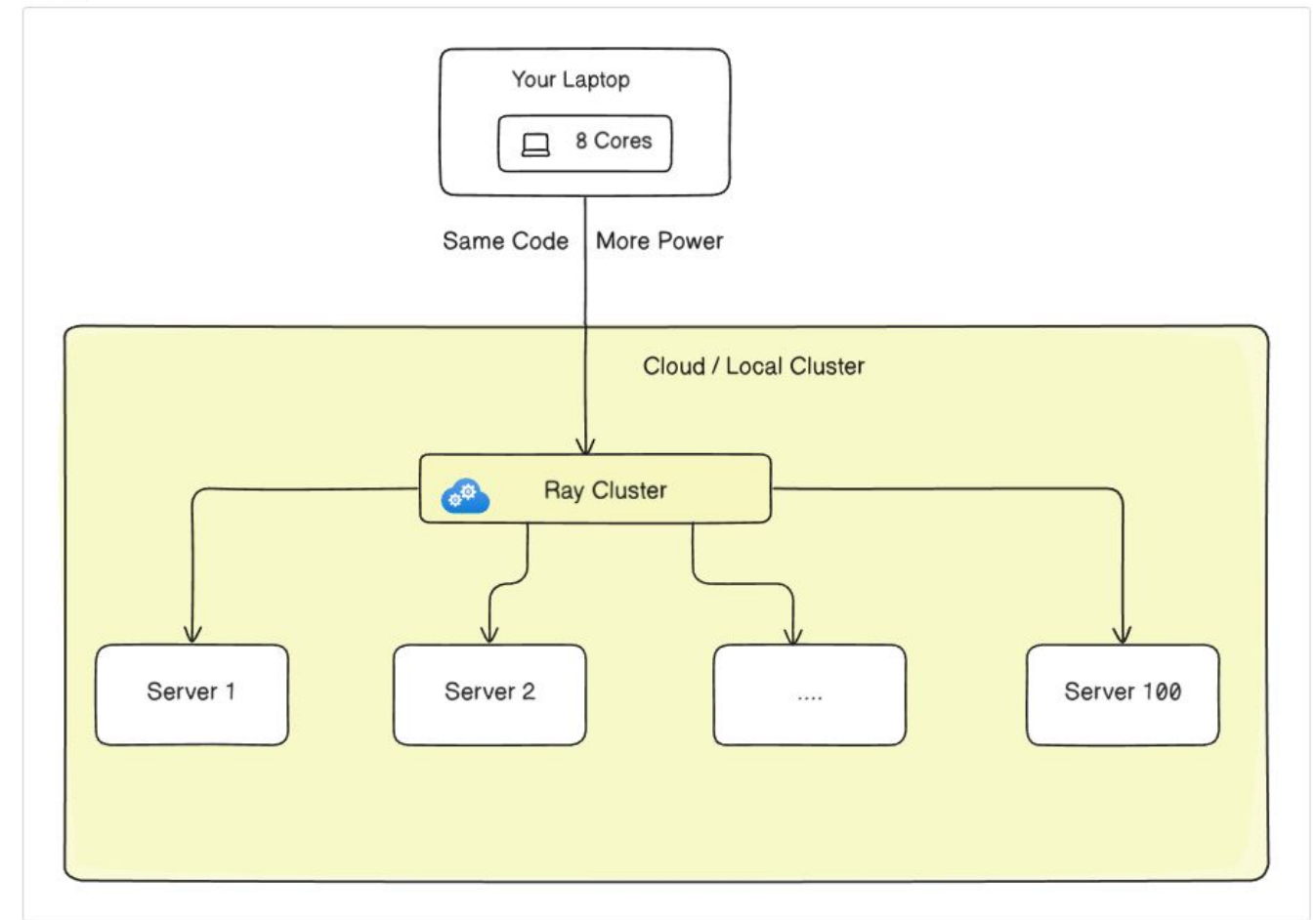
We need more compute Power than a single Laptop can offer
We need to go from **6 Cores to 600 Cores**

Ray Clusters

The big picture is scaling from a Single-Node “Cluster” - Which is just your laptop

To a massive , Multi-Node Cluster in the Cloud

... Without Changing our Application Logic



The code we wrote in the previous Section

is already

Cluster Aware by default

A Single Head Node

This is the manager. It runs your main script , takes commands and tells other nodes what to do

Multiple Worker Nodes

This is the Muscle . They are pure Compute Power that executes task and host actors

How to Connect to a cluster

Step 1 : Write a yaml describing your cluster

```
! local_cluster.yaml
...
1 cluster_name: local-cluster-1
2
3 provider:
4   type: local
5   head_ip: 192.168.1.10      #
6   worker_ips:
7     - 192.168.1.11
8     - 192.168.1.12
9     - 192.168.1.13
10    - 192.168.1.14
11
12 # Describe the head node
13 head_node_type:
14   name: head_node
15   resources: {"CPU": 4}
16
17 # Describe the worker nodes
18 worker_node_types:
19   - name: worker_nodes
20     resources: {"CPU": 8, "GPU": 1}
21     min_workers: 4
22     max_workers: 4
23
24 # Setup commands for local nodes
25 setup_commands:
26   - sudo apt update && sudo apt install -y python3-pip
27   - pip install -U ray
28
29 # Start Ray commands
30 head_start_ray_commands:
31   - ray stop
32   - ray start --head --port=6379 --dashboard-host=0.0.0.0
33
34 worker_start_ray_commands:
35   - ray stop
36   - ray start --address=192.168.1.10:6379
```

```
! gcp_cluster.yaml
...
1 cluster_name: gcp-cluster-1
2
3 provider:
4   type: gcp
5   region: us-central1
6   availability_zone: us-central1-a
7   project_id: your-gcp-project-id
8
9 head_node_type:
10   name: head_node
11   machineType: n1-standard-4
12   disks:
13     - boot: true
14       autoDelete: true
15       initializeParams:
16         diskSizeGb: 100
17         sourceImage: projects/debian-cloud/global/images/family/debian-11
18   labels:
19     role: head
20
21 # Describe the worker nodes (the muscle)
22 worker_node_types:
23   - name: worker_nodes
24     machineType: n1-standard-8
25     min_workers: 10
26     max_workers: 10
27     disks:
28       - boot: true
29         autoDelete: true
30         initializeParams:
31           diskSizeGb: 100
32           sourceImage: projects/debian-cloud/global/images/family/debian-11
33     labels:
34       role: worker
35
36 # Docker image / setup commands (optional)
37 setup_commands:
38   - sudo apt update && sudo apt install -y python3-pip
39   - pip install -U ray
40
```

Deploying a Distributed System used to be a nightmare

... A job dedicated for a team of devops engineer

Ray turn it into a couple of Command-Line Instructions

Step 2 : Launch the cluster with one command

```
prarabdha-srivastava@prarabdha-srivastava-HP-Pavilion-Laptop-14-ec0xxx:~/Desktop/iisc/ubucon$ ray up gcp_cluster.yaml
...
- ✓ Head node configured.
- ✓ 10 worker nodes configured.
- All nodes are running.
- Cluster is up.
```

Your 80 Core
supercomputer is
ready

Step 3 : The **Wow Moment** - Run your code unchanged



```
ray submit gcp_cluster.yaml main.py
```

Step 4 : Don't forget to shut down the cluster



```
ray down gcp_cluster.yaml
```

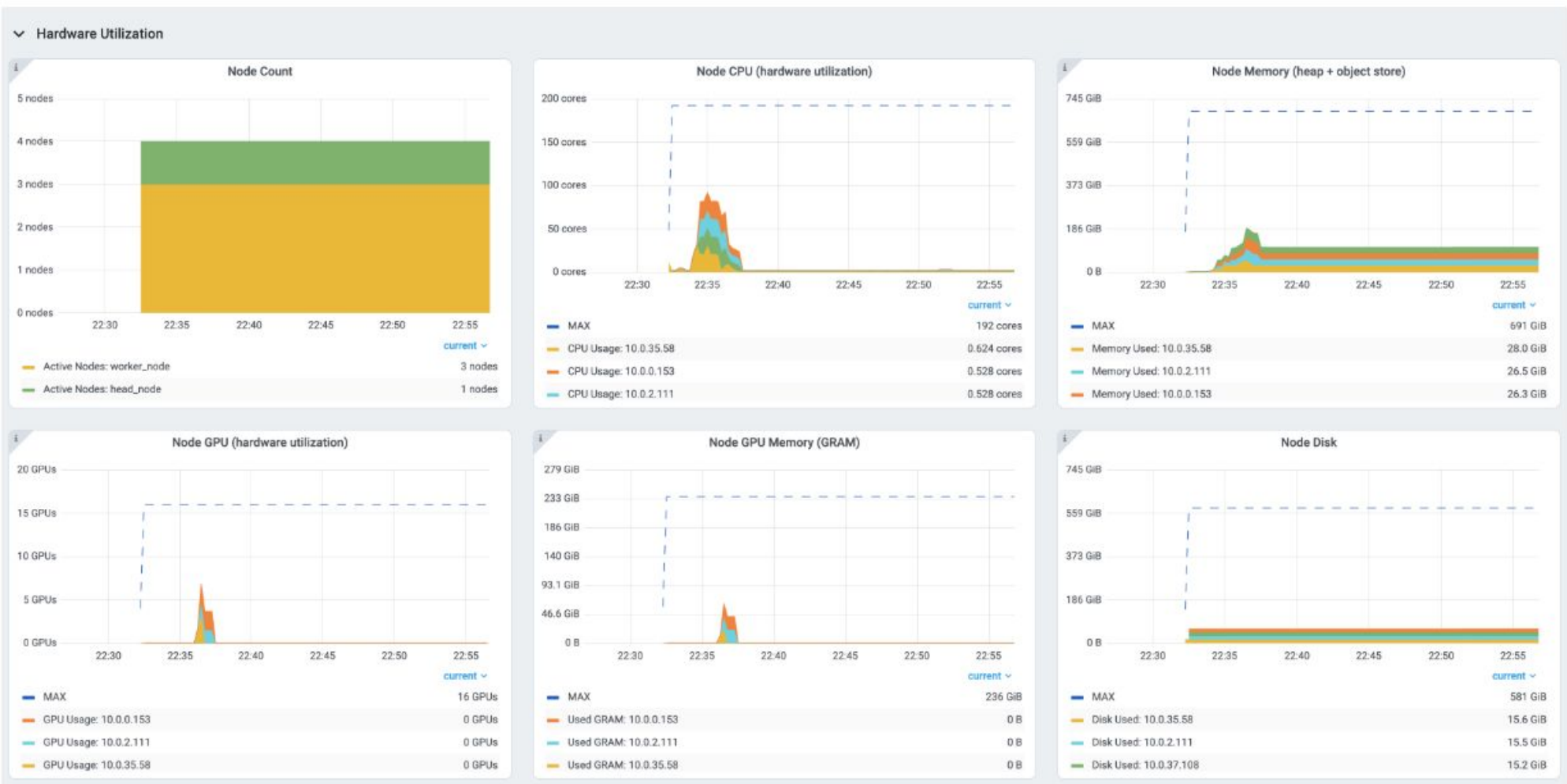
The exact same [main.py](#)

Now seamlessly distributes it's
10,000 tasks across 10 worker
nodes

WHEN YOU RUN **ray up**, THE OUTPUT GIVES YOU A LINK TO YOUR MISSION CONTROL.

The dashboard gives you a live look into your cluster:

- A visual map of all your nodes.
- CPU and memory usage per machine.
- The status of all running tasks and actors.
- Logs from all worker machines, all in one place.



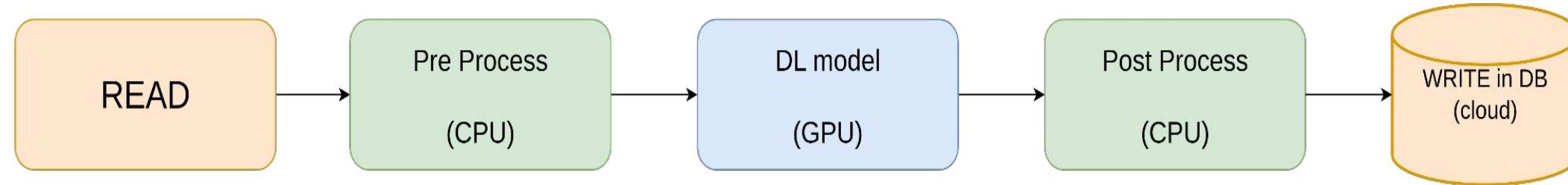
| Primitive / Tool | Role | Example |
|------------------------------|---|--|
| cluster.yaml | A declarative file defining your cluster's cloud resources. | Defines instance types, node counts, provider. |
| ray up <config> | CLI command to create or update a Ray cluster from a config file. | ray up my-cluster.yaml |
| ray down <config> | CLI command to terminate the cluster and release cloud resources. | ray down my-cluster.yaml |
| ray submit <config> <script> | CLI command to submit your Python script to the cluster's head node. | ray submit my-cluster.yaml app.py |
| ray.init(address='auto') | Connects your script to an existing Ray cluster instead of a local one. | ray.init(address='auto') |

CORE PRIMITIVES FOR THIS SECTION

Ray Data

New Scenario : **Handle a massive Dataset**

Your Manager tells you to load a 500 gb image data from S3 -> Process that image using a deep learning model -> Store the processed images back on s3



1. Support for multimodal data

- Native Tensor datatype support (fixed, variable shaped, ragged, etc)
- Backend based on PyArrow, Numpy, Pandas
- Readers for images, video, audio

2. Robust support for accelerators

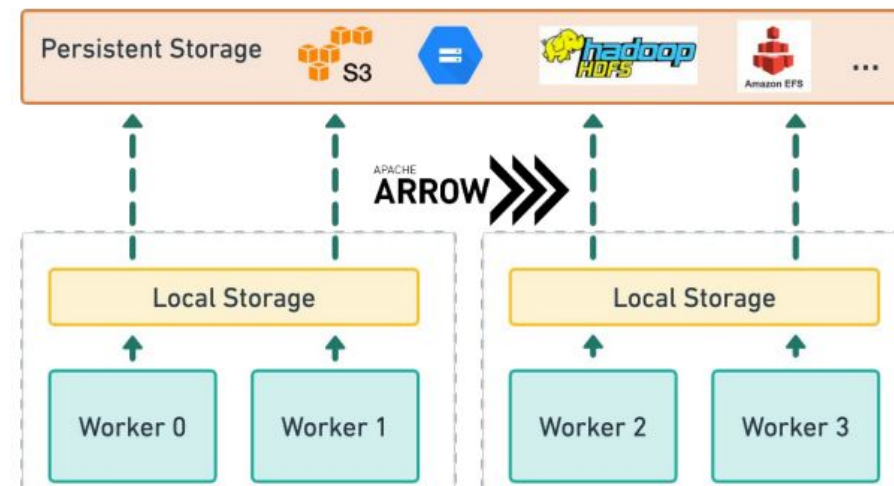
- GPU-based scheduling
- Accelerator observability
- Support for GPUs, TPUs, etc
- Heterogeneous device pipelines

3. Stateful operators

- Stateful operators for expensive GPU state
- Actor placement groups for large models
- Streaming execution for reduced memory pressure

4. Ecosystem interoperability

- First-class integration with major data sources (parquet, iceberg, delta, hudi, etc)
- Integration with Pytorch, Tensorflow, TFRecords
- vLLM + SGLang support



RAM of head node is 16gb and data is 500 gb . So how to solve this problem ?

The solution is a fundamental mental shift

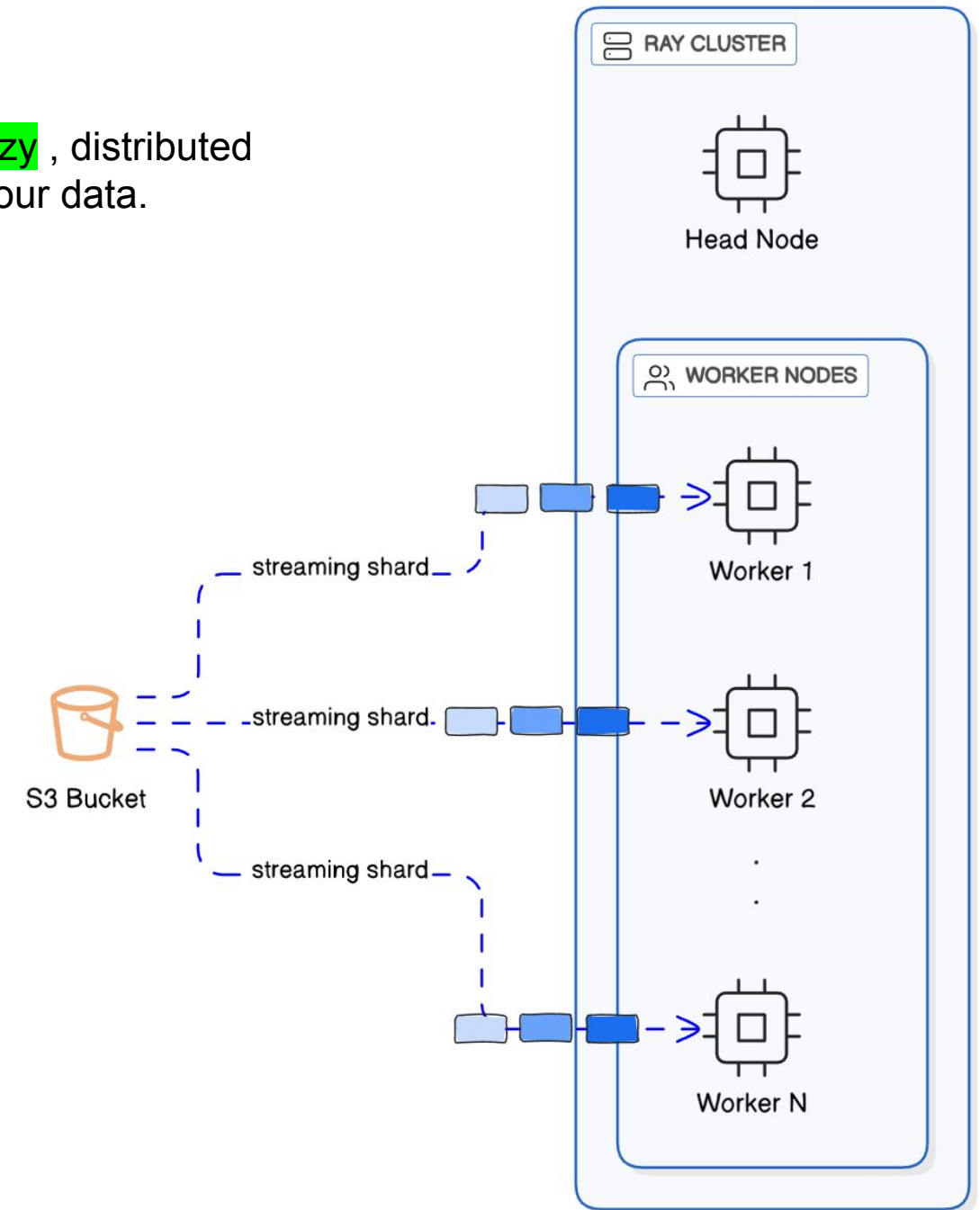
We need to stop thinking data as a single monolithic block ...

... And instead treat it as a **Distributed Stream**

THREE KEY IDEAS:

- 1. Lazy Execution:** Creates a *plan*, not loading data. Loads only when needed.
- 2. Sharding & Streaming:** Splits data into blocks, streams directly to workers (bypassing head node).
- 3. Built-in Parallelism:** Operations like `.map()` run as parallel tasks across the cluster.

A ray Dataset is a **lazy** , distributed reference to your data.



Step 1 : Create a Dataset : get reference of data almost zero data utilization of head node

```
15 # Doesn't read the whole data just creates a plan
16 ds = ray.data.read_images("gs://bucket-name/raw-images/")
17
18 print(ds)
19 #Output: Dataset(num_blocks = .. , num_rows = ... , schema = {image: ArrowTensorType(...)})
20
```

Step 2 : Apply Transformations with .map()

```
15 @ray.remote
16 def process_image(row: dict) -> dict:
17     image = row['image']
18     time.sleep(1)
19     inverted_image = 255 - image
20     return {'processed_image' : inverted_image , "original_id": row.get('id', None)}
21
22 # This defines the computation graph. It is still LAZY . Nothing has happen yet.
23 processed_ds = ds.map(process_image)
```

Step 3 : Trigger the Computation and save the results

```
25 # This call triggers the distributed computation:
26 # 1. Ray Data starts reading blocks of images from S3 in parallel.
27 # 2. It sends these blocks directly to `process_image` tasks on the cluster.
28 # 3. It gathers the results and writes them back to S3.
29
30 processed_ds.write_parquet("gs://bucket-name/processed-images/")
```

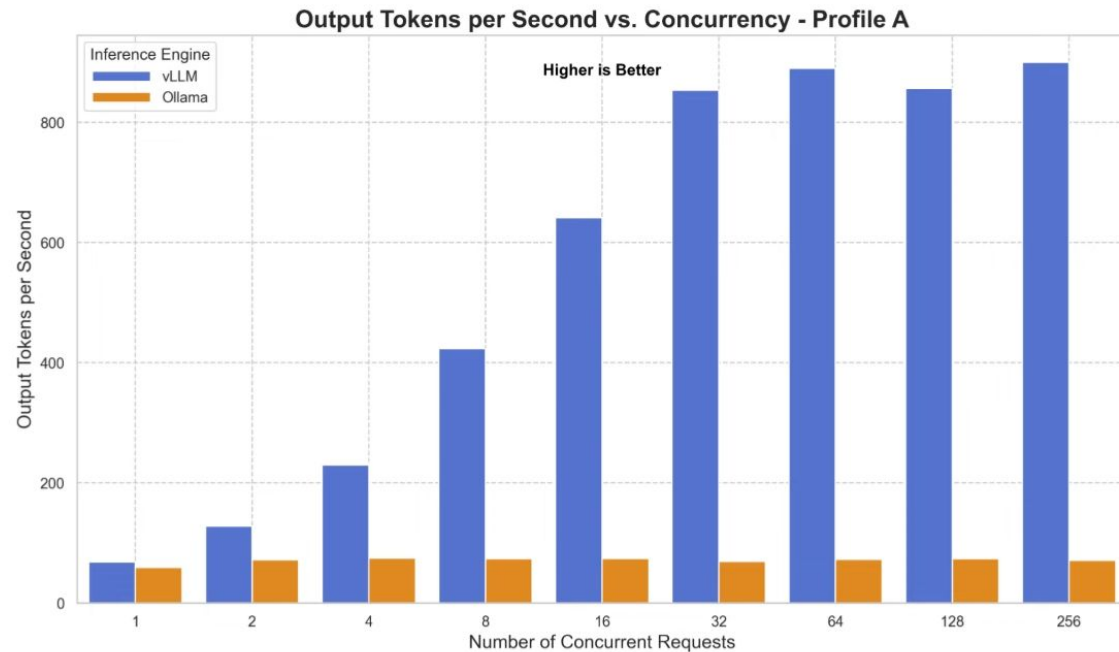


| Primitive | Role | Example |
|-----------------------|--|--|
| ray.data.Datas et | A lazy, distributed pointer to a large dataset. | ds = ray.data.read_parquet(...) |
| ray.data.read_ *() | Functions to create a Dataset from storage. | read_images(), read_csv() |
| Dataset.map() | Applies a function to each row of the Dataset in parallel. | ds.map(my_remote_func) |
| Dataset.write_ *() | Triggers computation and saves the resulting Dataset. | ds.write_parquet(...) |

A little Bonus for guys working with Gen AI : vLLM

Output Tokens per Second

Results - Default Ollama vs vLLM : Standard settings



If you're not using **vLLM** yet... what are you even doing?

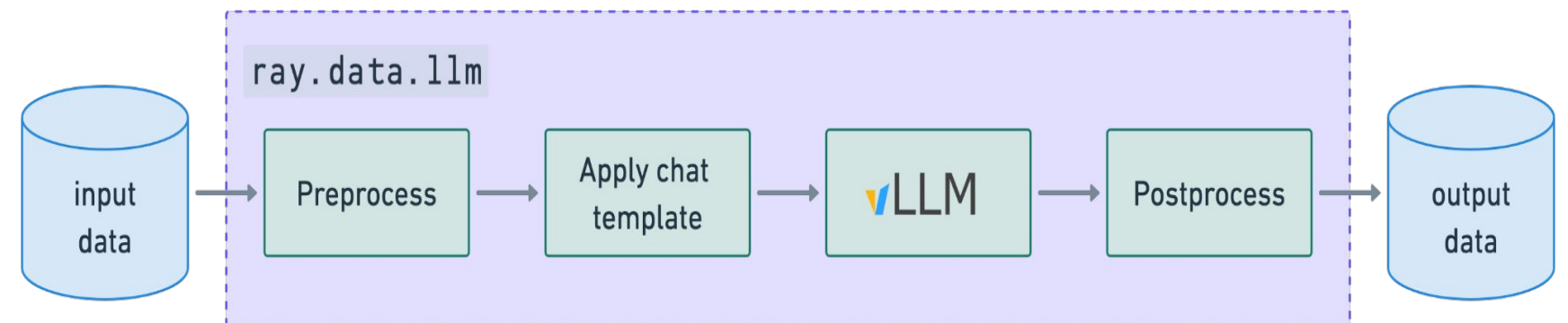
And when you combine **vLLM + Ray**, you get the same scaling superpowers that big tech relies on for massive AI workflows.

Take-Home Task:

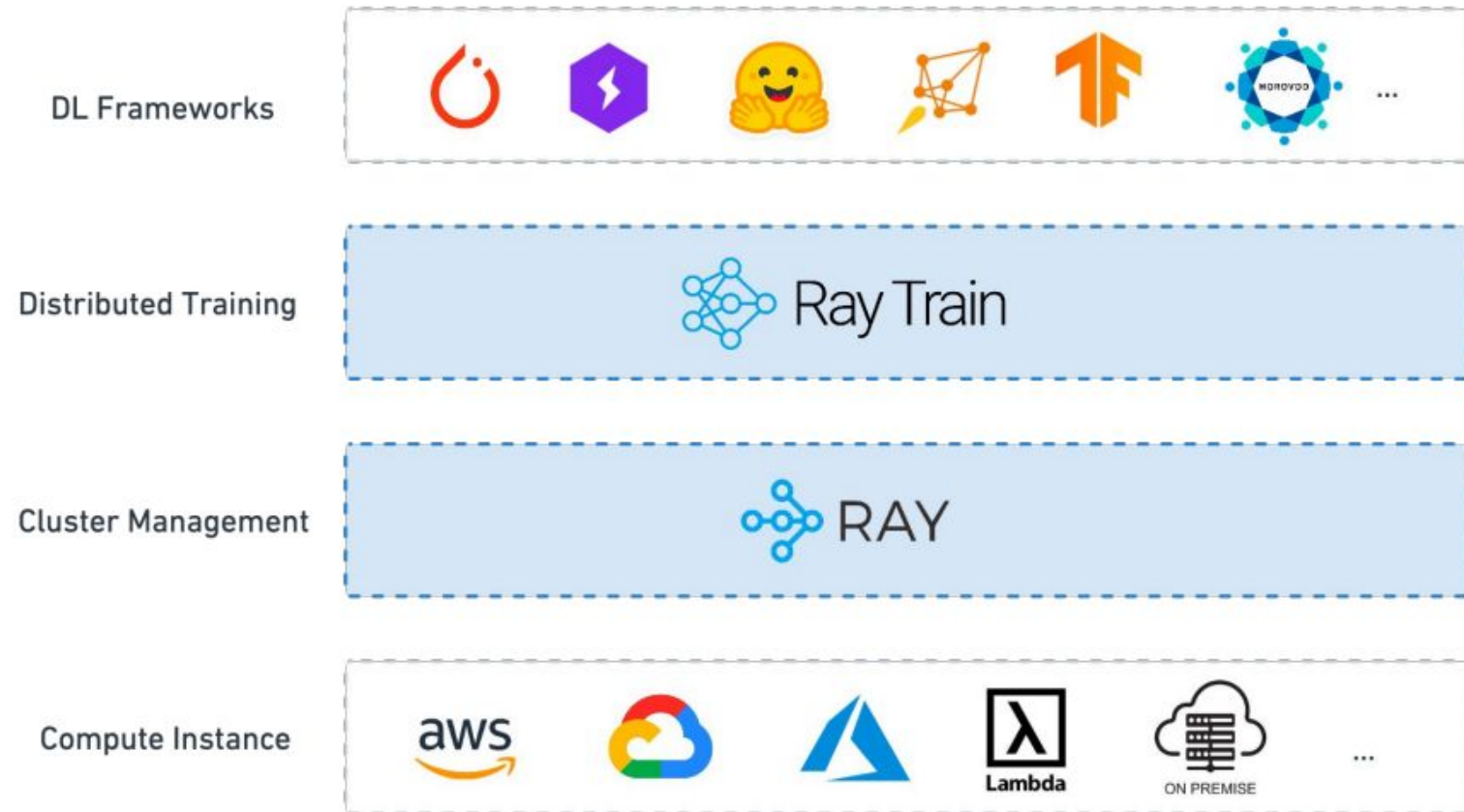
Now that you understand how Ray and vLLM work, try implementing this full pipeline on the **GSM-8K** dataset from Hugging Face.

vLLM – The Most Popular LLM Inference Framework

- >30k stars on Github
- >200 merged PR every month
- **Notable contributors:**
Anyscale, AI21, Alibaba, AWS, ByteDance, CentML, Databricks, Deepinfra, Microsoft, Mistral, NVIDIA, Oracle, Parasail, Snowflake, Tencent, ...



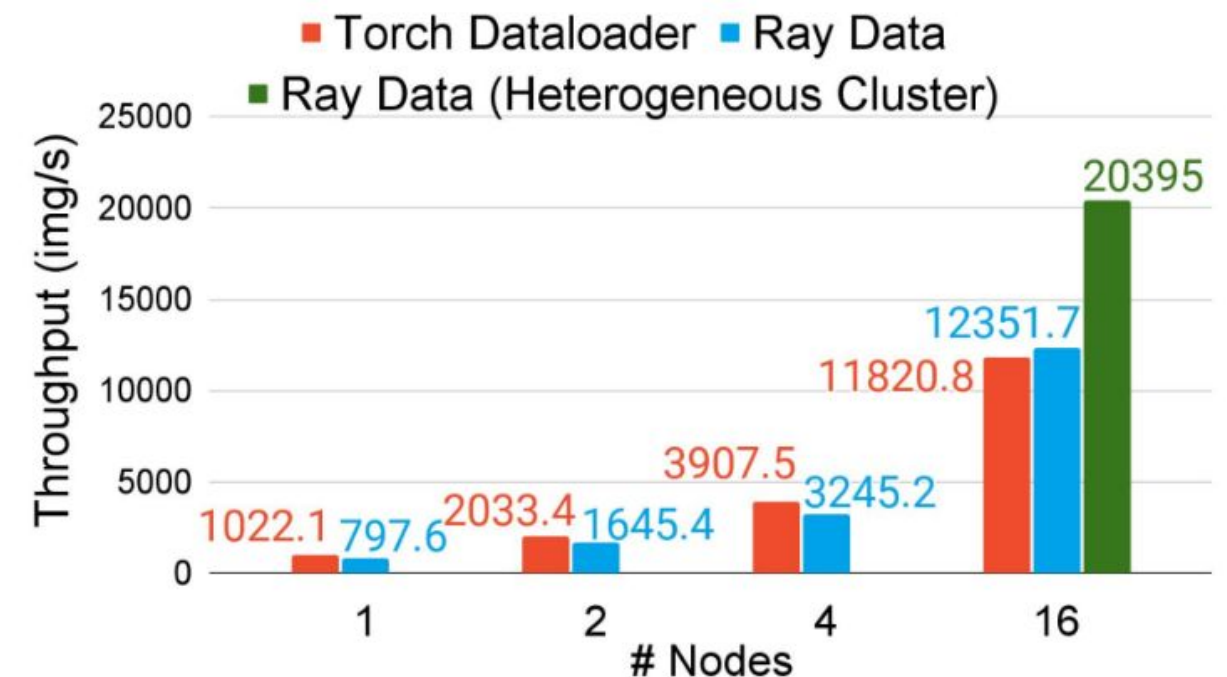
Ray Train

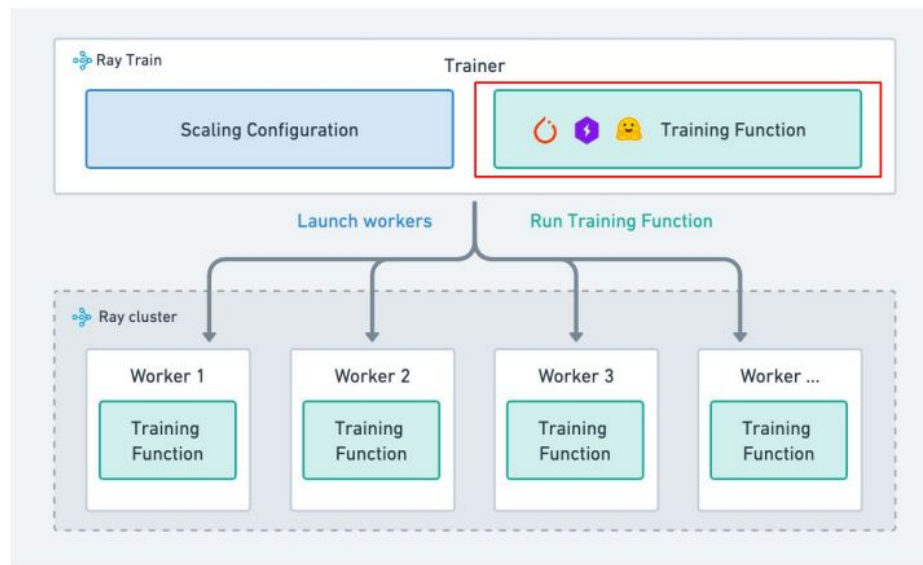


Ray Train is a library for **developing, orchestrating, and scaling** distributed deep learning applications.

| Feature | torchrun | Ray Train |
|--------------------------|--------------------------------------|--|
| Cluster management | You must manually SSH & set env vars | Ray handles multi-node orchestration automatically |
| Checkpoint management | Manual | Automatic per-epoch management |
| Resuming job | Manual code or CLI | <code>.restore()</code> or automatic resume |
| Distributed logs/metrics | Manual gathering | Integrated experiment tracking |
| Elastic scaling | Fixed world size | Can dynamically scale workers |
| Failure retries | None | <code>max_failures</code> configurable |

Ray Data vs. Torch DataLoader (No caching)





```

from ray.train.torch import TorchTrainer
from ray.train import ScalingConfig

def train_func(config):
    # Your PyTorch/Lightning/Hugging Face training code here.

scaling_config = ScalingConfig(num_workers=4, use_gpu=True)
trainer = TorchTrainer(train_func, scaling_config=scaling_config)
result = trainer.fit()

```



Hugging Face Transformers

```

from datasets import load_dataset
from transformers import Trainer, TrainingArgument

def train_func(config):
    model = AutoModel.from_pretrained(...)
    dataset = load_dataset(...)

    trainer = Trainer(
        model=model,
        args=TrainingArgument(...),
        train_dataset=dataset["train"],
        eval_dataset=dataset["eval"]
    )

    trainer.train()

```



```

from datasets import load_dataset
from transformers import Trainer, TrainingArgument
from ray.train.huggingface.transformers import prepare_trainer

def train_func(config):
    model = AutoModel.from_pretrained(...)
    dataset = load_dataset(...)

    trainer = Trainer(
        model=model,
        args=TrainingArgument(...),
        train_dataset=dataset["train"],
        eval_dataset=dataset["eval"]
    )
    trainer = prepare_trainer(trainer)
    trainer.train()

```

```

from ray.train.torch import TorchTrainer
from ray.train import ScalingConfig

trainer = TorchTrainer(
    train_func,
    scaling_config=ScalingConfig(num_workers=16, use_gpu=True)
)
trainer.fit()

```

Ray Train : **Compatibility**

Integration with deep learning frameworks



Setup distributed env

Setup DDP model

Setup distributed sampler

Move batches to GPU

```
def train_func(config):
    dist.init_process_group("nccl")
    rank = os.environ["LOCAL_RANK"]
    device = torch.device(f"cuda:{rank}")

    model = MyTorchModel(...)
    model = model.to(device)
    model = DDP(model, device_ids=[device])

    sampler = DistributedSampler(
        dataset,
        rank=os.environ["RANK"],
        num_replicas=os.environ["WORLD_SIZE"],
        shuffle=True,
    )
    dataloader = DataLoader(
        ...,
        sampler=sampler
    )

    # Training
    for epoch in range(num_epochs):
        for inputs, labels in dataloader:
            # train batch
            inputs = inputs.to(device)
            labels = labels.to(device)
            ...
```

```
from ray.train.torch import prepare_model, prepare_data_loader

def train_func(config):
    model = MyTorchModel(...)
    model = prepare_model(model)

    dataloader = DataLoader(...)
    dataloader = prepare_data_loader(dataloader)

    # Training
    for epoch in range(num_epochs):
        for inputs, labels in dataloader:
            ...
```



```
from ray.train.torch import TorchTrainer
from ray.train import ScalingConfig

trainer = TorchTrainer(
    train_func,
    scaling_config=ScalingConfig(num_workers=16, use_gpu=True)
)
trainer.fit()
```

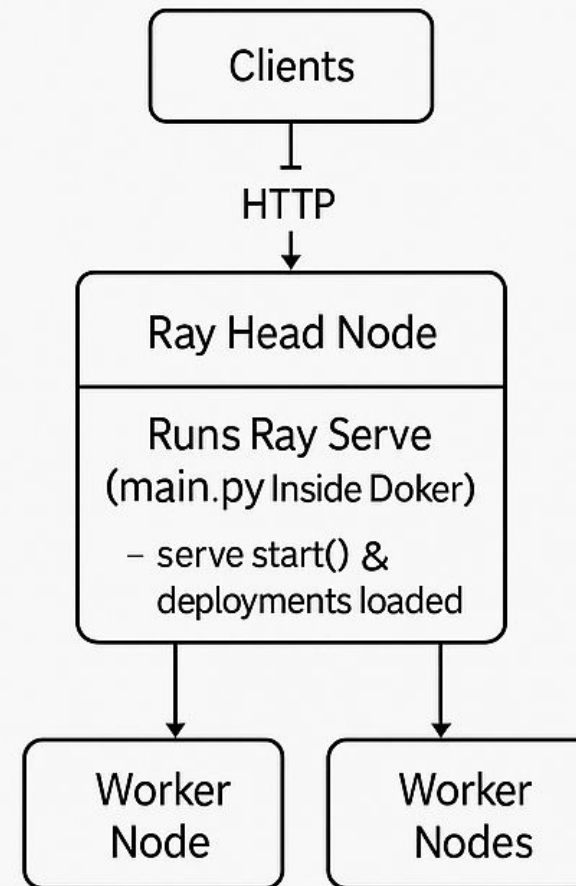
Ray Serve

A very simple vLLM based API on Ray Server

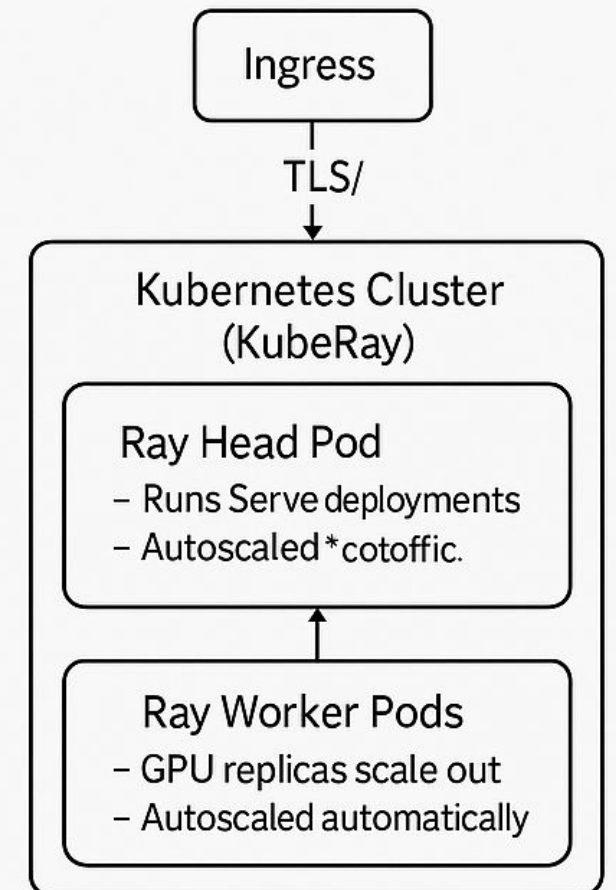
```
1 import ray
2 from ray import serve
3 from vllm import LLM, SamplingParams
4
5 ray.init()
6 serve.start()
7
8 @serve.deployment(ray_actor_options={"num_gpus": 1})
9 class VLLMDeployment:
10     def __init__(self):
11         self.llm = LLM(model="Qwen/Qwen2.5-0.5B")
12         self.params = SamplingParams(max_tokens=100)
13
14     def __call__(self, prompt: str):
15         output = self.llm.generate([prompt], self.params)
16         return output[0].outputs[0].text
17
18 VLLMDeployment.deploy()
```

How to Deploy Ray Serve in Production

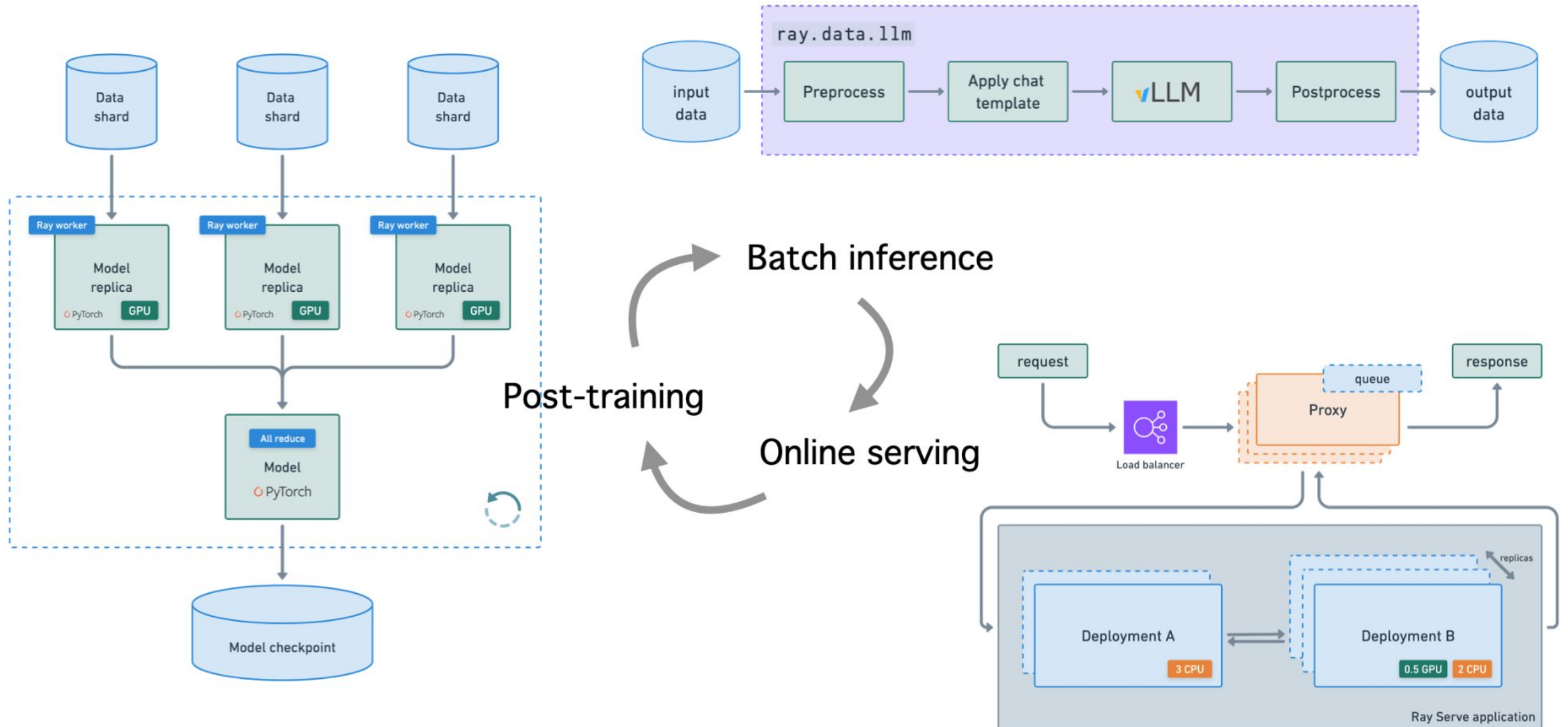
Option A – Ray Cluster on VMs



Option B – Kubernetes + KubeRay



Ray Serve



Conclusion : Please Use **RAY**

This workflow gives you a clear path to building scalable AI pipelines on familiar infrastructure.

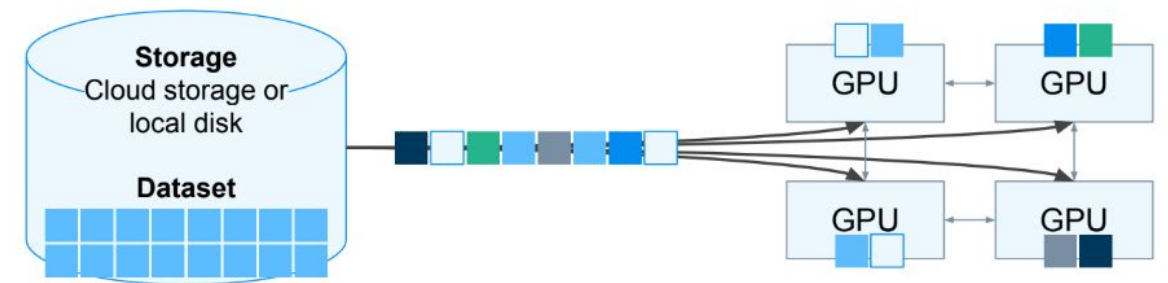
Stream TB-scale data from S3 storage using [ray.data](https://docs.ray.io/en/latest/data/index.html), enabling high-throughput, fault-tolerant loading.

Efficient preprocessing with `map_batches` and object spilling to memory or disk, minimizing overhead and reducing costly recomputations.

Distributed training leveraging Ray's TorchTrainer across multiple GPU nodes to scale model training and reduce wall-clock time.

Deploy inference with Ray Serve for scalable, real-time predictions with low latency.

This architecture leverages abstractions away a lot of infra management, letting you focus on experimentation & building impactful projects.



Needs to be **fast**, to maximize GPU utilization.

Needs to **scale** to large datasets and clusters.

Needs to be **flexible**, to support arbitrary preprocessing.


→ Data can have different: storage, modality, preprocessing, memory footprint, ordering, ...



UbuCon India²⁵

Thank You!

Qualcomm

 Canonical

IT'S FOSS

