anyscale

Ray: 大模型时代的 AI Infra

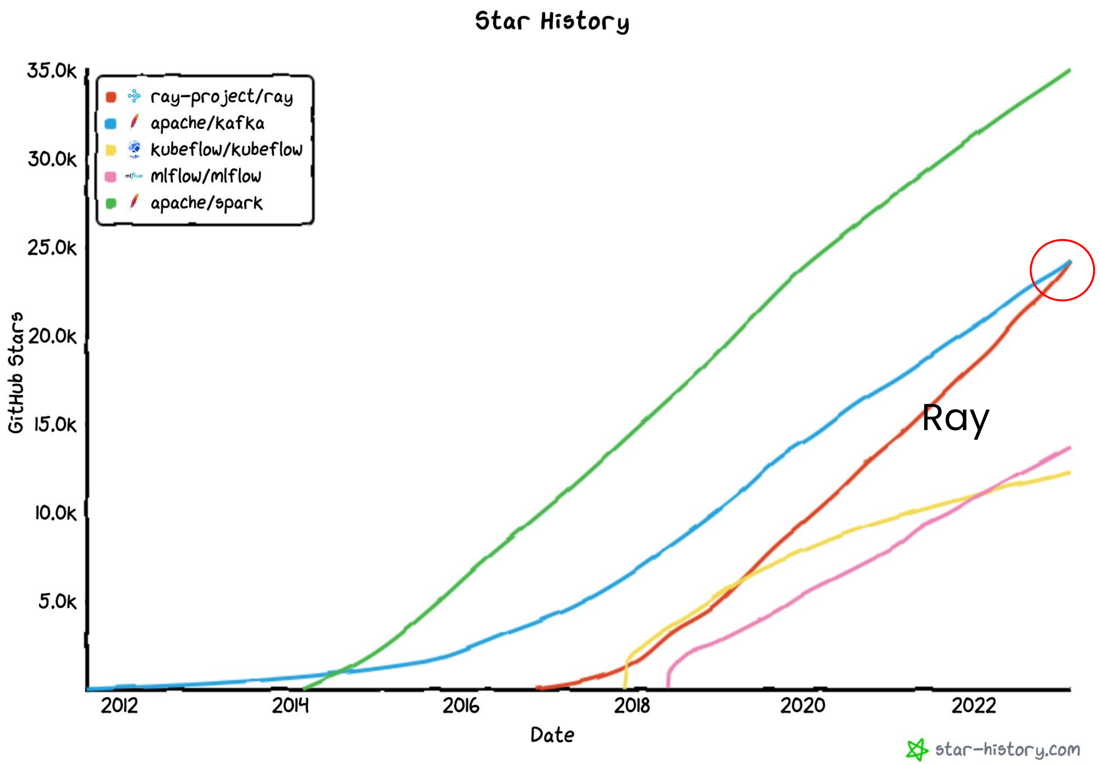
张喆, Anyscale Ray团队负责人

zhz@anyscale.com

zhethoughts

07.02.2023

Ray: Fastest Growing Distributed Compute Framework



Github stars超过Apache Kafka

Ray: Fastest Growing Distributed Compute Framework

amazon	McKinsey & Company	Oak Ridge National Laboratory	ERICSSON	WILD LIFE	cruise
Morgan Stanley	rbi restaurant brands international	shopify	J.P.Morgan	KOCH	RIOT GAMES
intel	Microsoft	蚂蚁金服 ANT FINANCIAL	Uber	NetEase Games	RICARDO
OpenAI	ByteDance	lyft	instacart	verizon	Spotify

25,000+
GitHub
stars

820+
Community
Contributors

5,000+
Repositories
Depend on Ray

1,000+
Organizations
Using Ray

ChatGPT / GPT-4 Trained on Ray!

Fast iterations at Hyper-**scale**!

*"We looked at a half-dozen distributed computing projects, and **Ray** was by far the winner. … We are using **Ray** to train our largest models. It's been very helpful to us to be able to scale up to unprecedented scale. … Ray owns a whole layer, and not in an opaque way."*

This Talk

Adoption Highlights

<u>Ray Intro</u>

Use Case Details

Getting Hands Dirty

Ray: a short history

2016: Started as a class project

- Goal: scale real time ML and RL

2017: RLlib and Ray Tune released

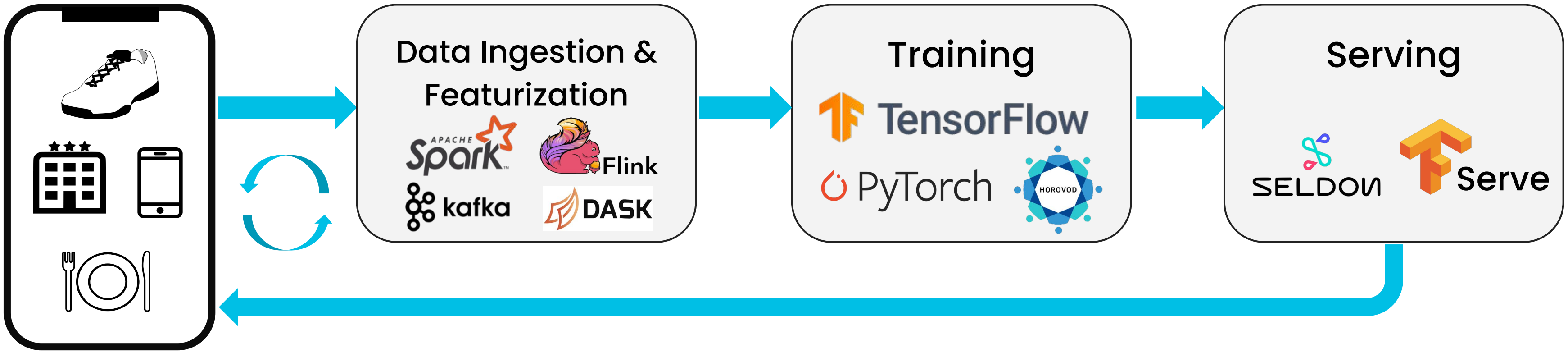
2019: Anyscale founded (company behind Ray)

2020: Ray v1.0 release; Ray Serve released

2022: Ray v2.0 release; Ray AI Runtime

2023: LLM / Generative AI

Example Use Case: Online learning



Solution: stitch together a bunch of distributed systems

Challenge with stitching distributed systems

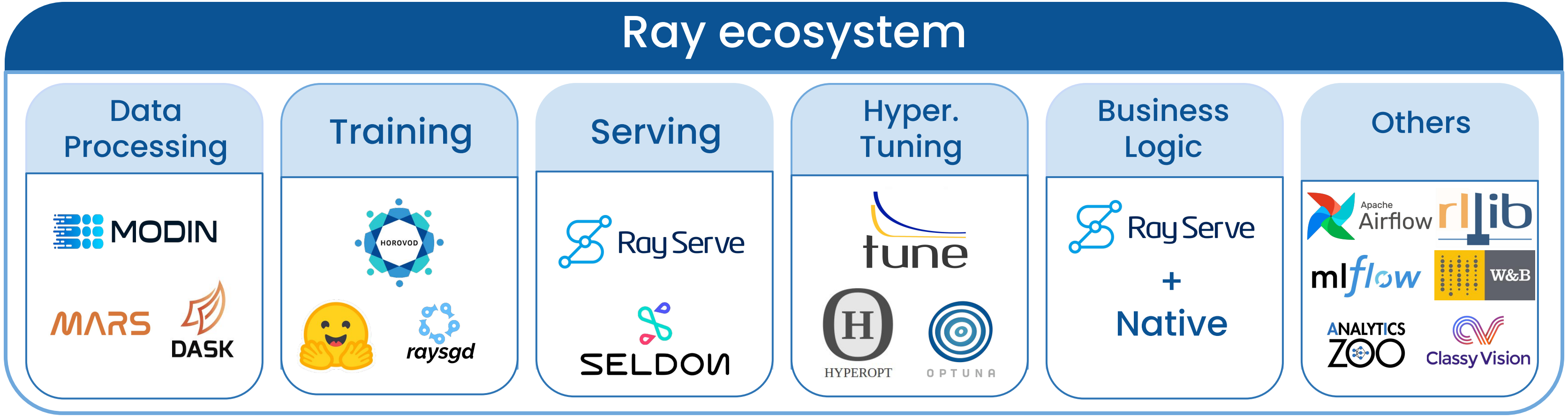
Data Processing	Training	Serving	Hyper. Tuning	Business Logic	Simulations
Apache Spark	Horovod	Seldon	Hyperopt	Go	MPI
Flink	PyTorch	KFServing	Optuna	dapr	Orleans
kafka		Serving	SigOpt	Vert.x	
DASK				spring	

Hard to develop: different APIs

Hard to deploy & manage

Slow: high overhead of moving data between different systems

Ray Unifies Distributed Workloads



Best ecosystem of distributed libraries

Instead of stitching systems, call libraries in **same** system

Easy to develop, manage, and deploy

History and Intro

1. Why was Ray created
2. **API and architecture basics**

Minimalist API

ray.init()	Initialize Ray context.
@ray.remote	Function or class decorator specifying that the function will be executed as a task or the class as an actor in a different process.
.remote	Postfix to every remote function, remote class declaration, or invocation of a remote class method. Remote operations are *asynchronous*.
ray.put()	Store object in object store, and return its ID. This ID can be used to pass object as an argument to any remote function or method call. This is a *synchronous* operation.
ray.get()	Return an object or list of objects from the object ID or list of object IDs. This is a *synchronous* (i.e., blocking) operation.

Ray in a nutshell

Functions → Tasks : stateless computations (essentially RPC)

Classes → Actors : stateful computations

Distributed futures : enables parallelism

In-memory object store: enables passing args/results by reference

Extension of existing languages, rather than a new language

Function

```
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a


def add(a, b):
    return np.add(a, b)

a = read_array(file1)
b = read_array(file2)
sum = add(a, b)
```

Class

```
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter()
c.inc()
c.inc()
```

RAY

Function → **Task**

```
@ray.remote
def read_array(file):
    # read ndarray “a”
    # from “file”
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

a = read_array(file1)
b = read_array(file2)
sum = add(a, b)
```

Class → **Actor**

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter()
c.inc()
c.inc()
```

Function → **Task**

```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```

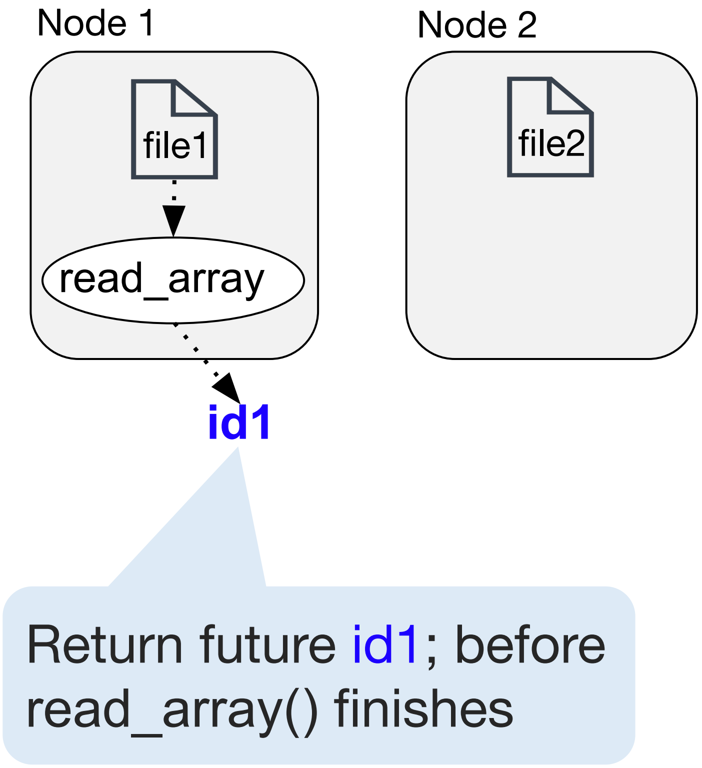
Object → **Actor**

```
@ray.remote
class Counter(object):
    def __init__(self):
        self.value = 0
    def inc(self):
        self.value += 1
        return self.value

c = Counter.remote()
id4 = c.inc.remote()
id5 = c.inc.remote()
```

Task API

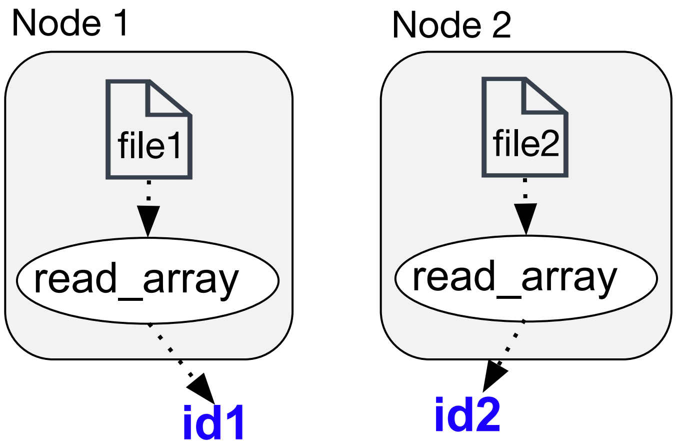
```
@ray.remote
def read_array(file):
    # read ndarray “a”
    # from “file”
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```

id1: distributed future (object id)



Task API

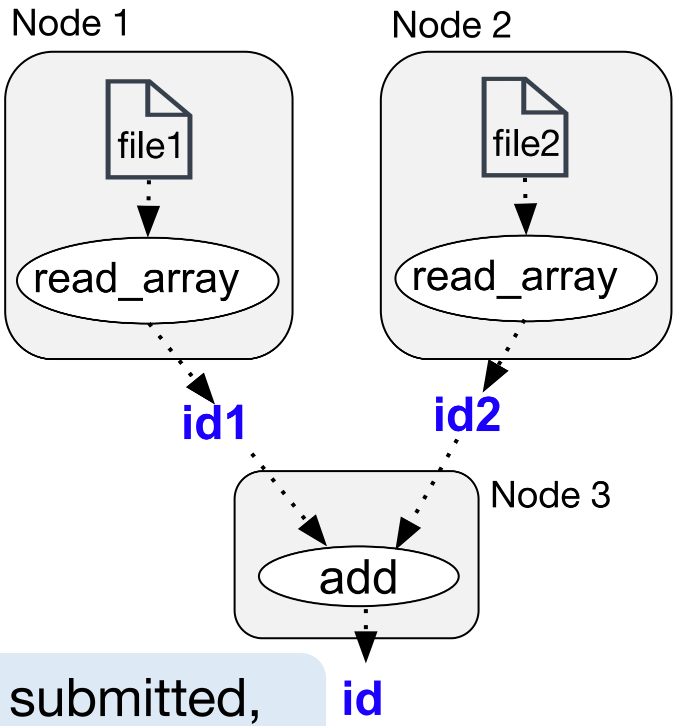
```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```



Dynamic task graph: build at runtime

Task API

```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```

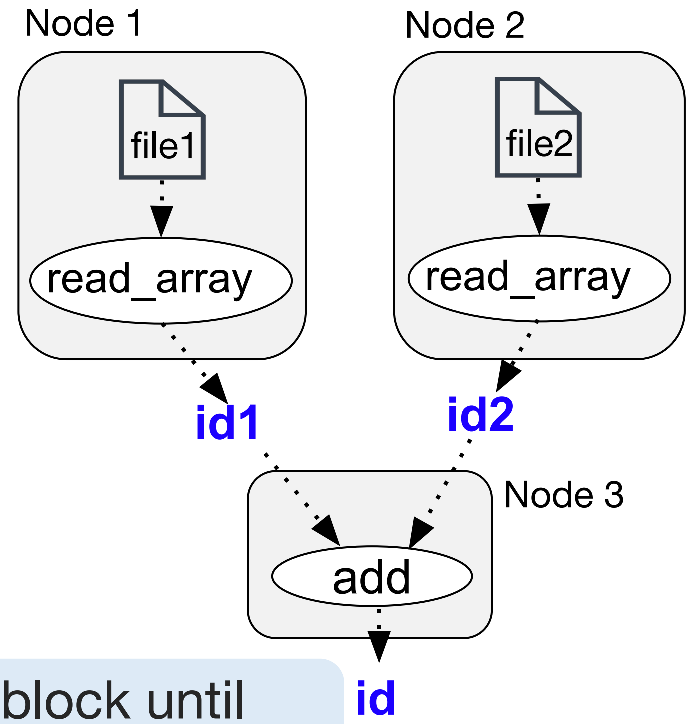


Every task submitted, but not finished yet

Task API

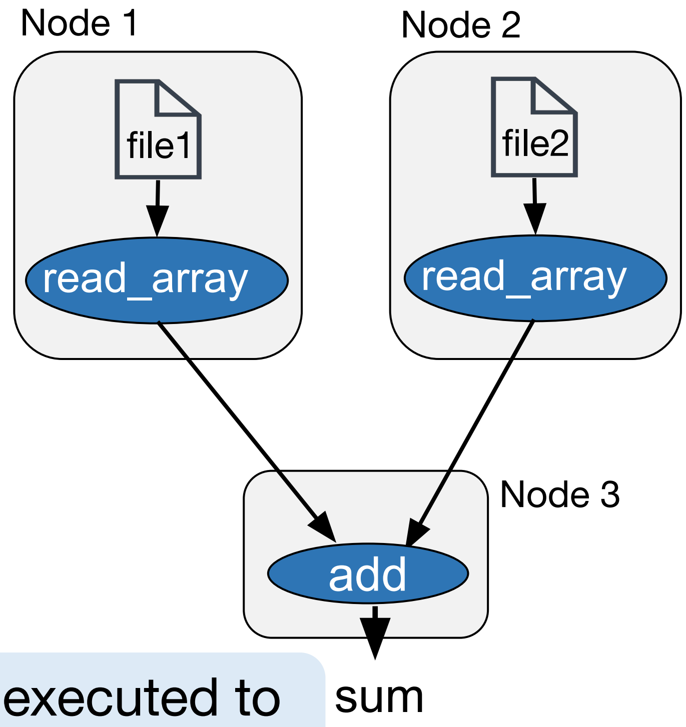
```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```

ray.get() block until result available



Task API

```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```



Task graph executed to compute sum

Function → **Task**

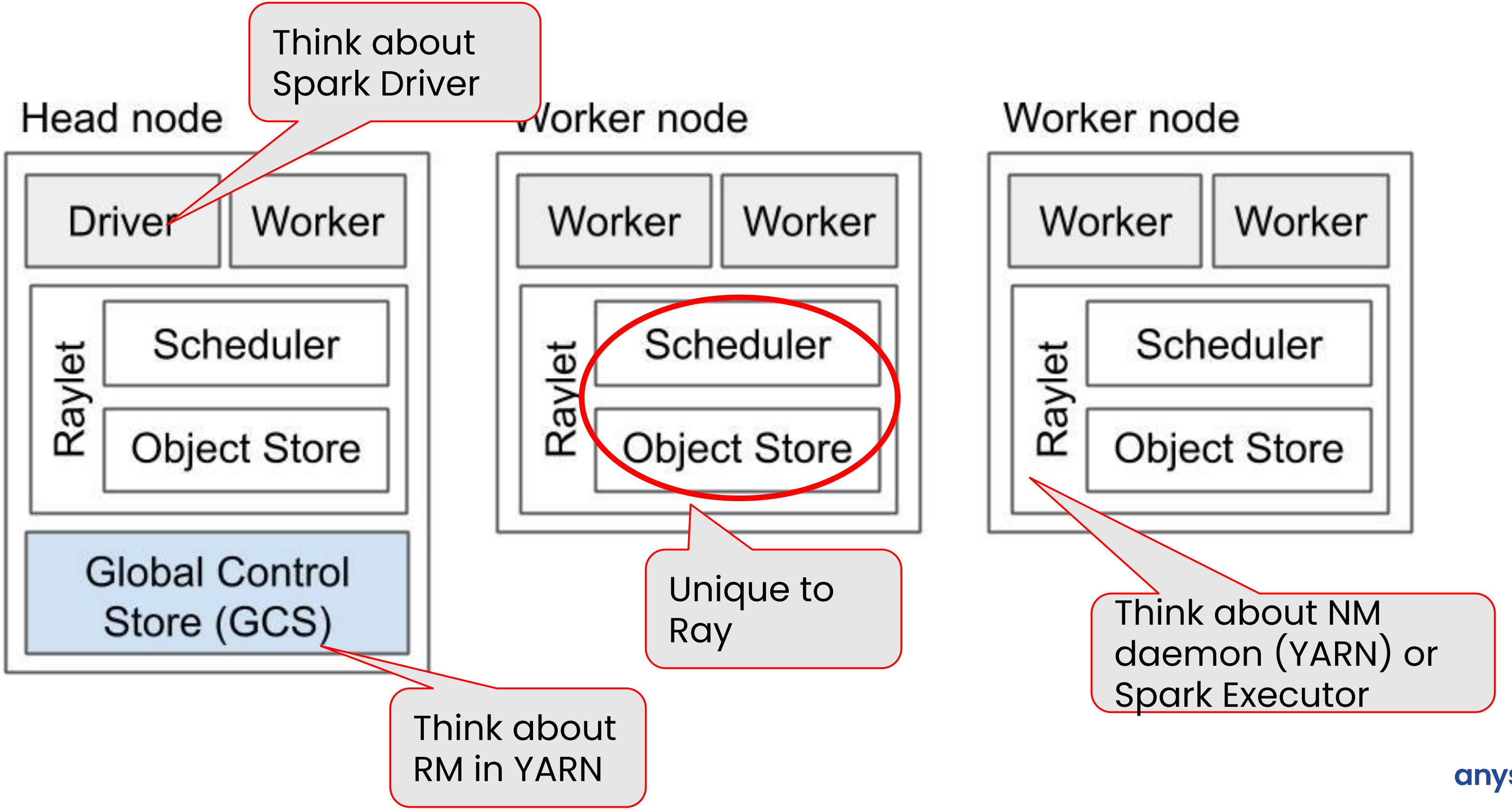
```
@ray.remote
def read_array(file):
    # read ndarray "a"
    # from "file"
    return a

@ray.remote
def add(a, b):
    return np.add(a, b)

id1 = read_array.remote(file1)
id2 = read_array.remote(file2)
id = add.remote(id1, id2)
sum = ray.get(id)
```

Class → **Actor**

```
@ray.remote(num_cpus=2, num_gpus=1)
class Counter(object):

        self.value += 1
        return self.value

c = Counter.remote()
id1 = c.inc.remote()
id2 = c.inc.remote()
val = ray.get(id2)
```

can specify
resource demands;
support heterogeneous hardware

What is Ray? A Cluster Looks Like…



This Talk

Adoption Highlights

Ray Intro

<u>Use Case Details</u>

Getting Hands Dirty

Trend: Ray as the “New AI Infra”

ML Platform

- Easy access to GPU
- Prototype -> production

Deep Learning

- High performance data ingest
- CPU / GPU heterogeneous cluster

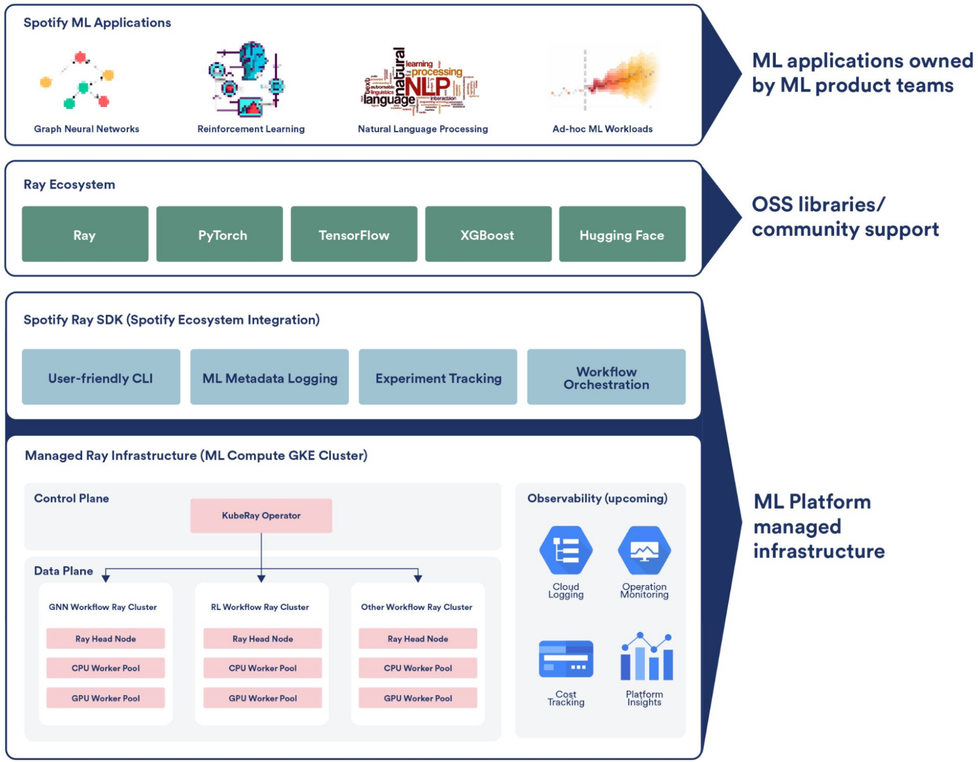
Batch Inference

- Efficiently “cache” the model
- High performance data pipeline

LLM / Generative AI

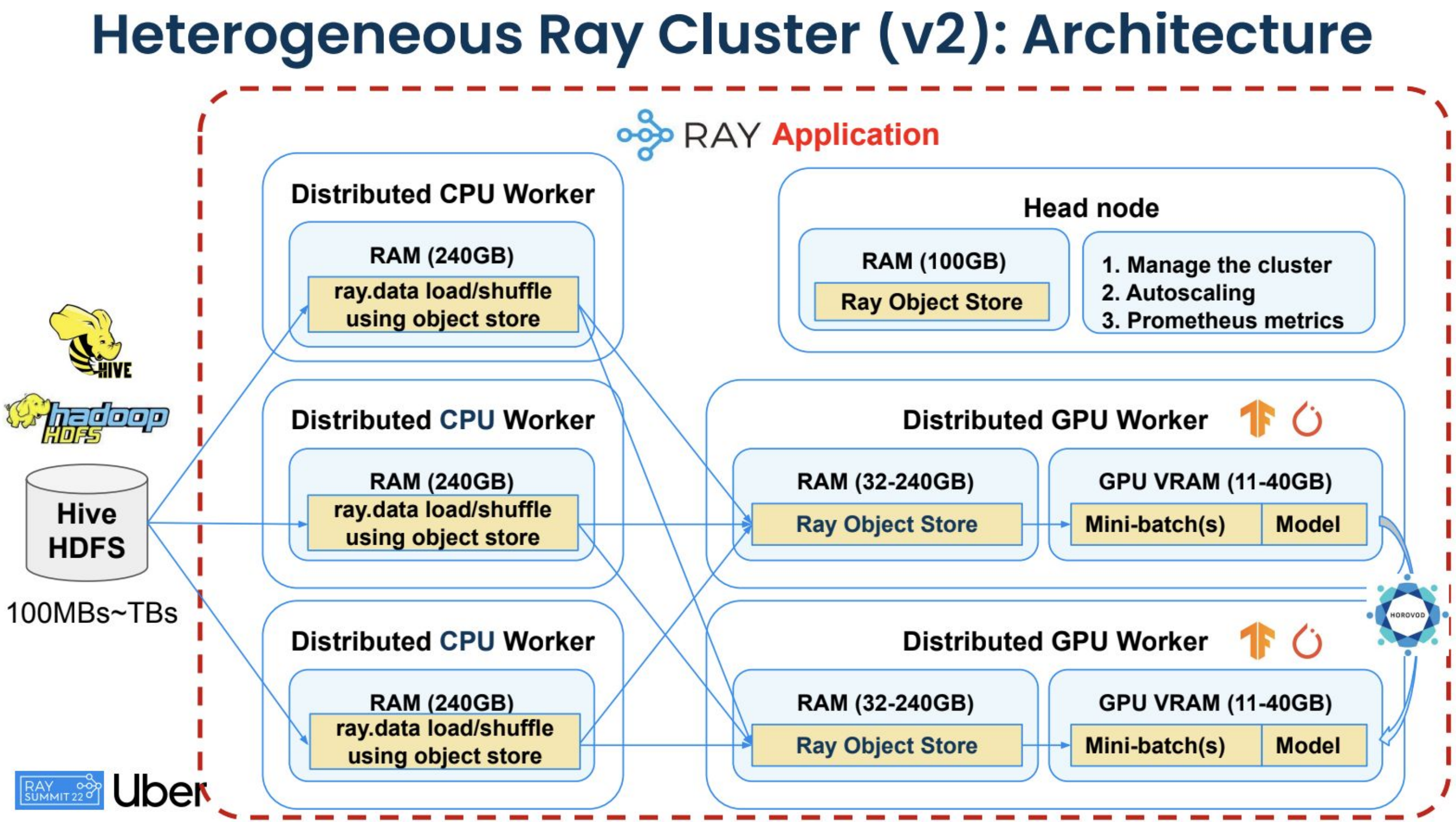
- Flexible model parallelism
- Utilizing smaller / cheaper GPUs

Unleashing ML **Innovation** at Spotify

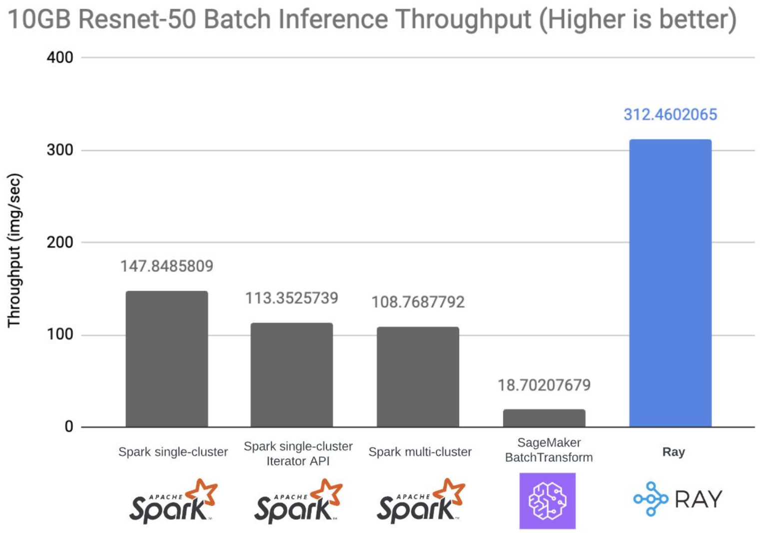


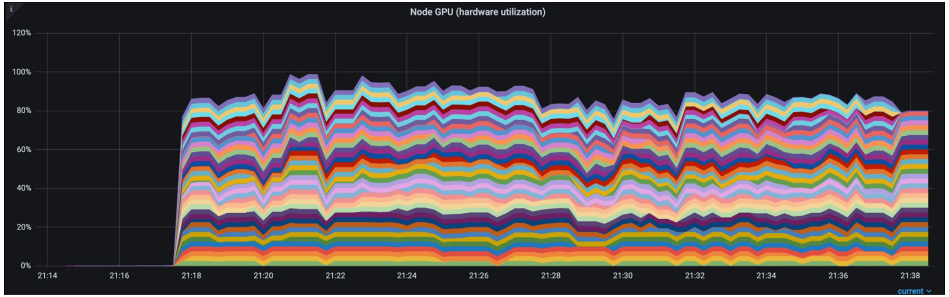
- Broadening production support for ML frameworks beyond TensorFlow to support nove ML solutions for Spotify
- Providing a more user-friendly way for users to access GPU and distributed compute
- Accelerating the user journey for ML research an prototyping
- Providing solutions to productionize more advanced ML paradigms, such as reinforcement learning (RL) and graph neural networks (GNN) workflows

Deep Learning Cluster at Uber



Batch Inference





[Ray Data vs Spark vs SageMaker](#)

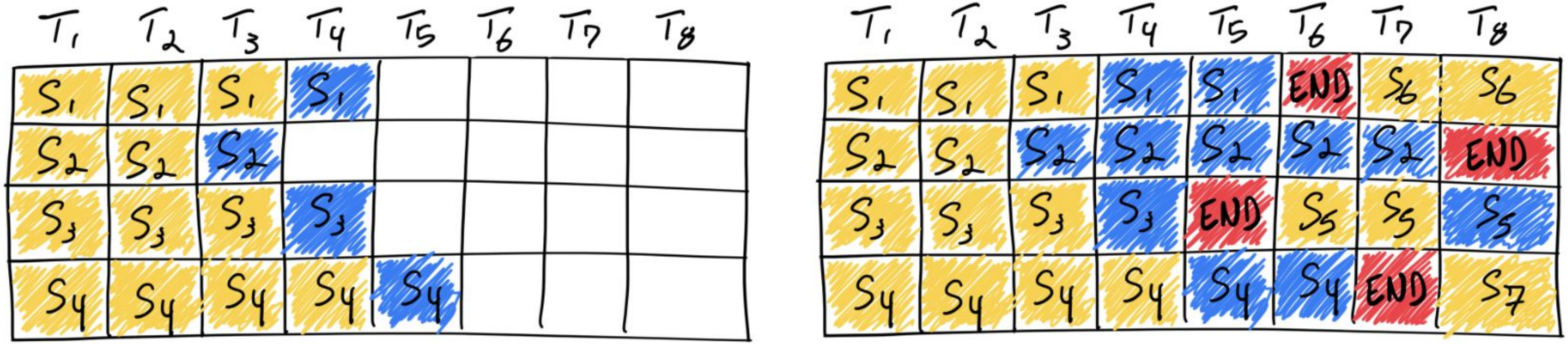
Batch Inference

```
import ray
from ray.data import ActorPoolStrategy

Model = ...
preprocess_fn = ...

# Define the model as a stateful class with cached setup.
class MyModelCallableCls:
    def __init__(self):
        self.model = Model(...)

    def __call__(self, batch: Dict[str, np.ndarray]) ->
        return {"results": self.model(batch)}

# Modify the pipeline to use GPU actors.
ray.data.read_parquet(...) \
    .map_batches(preprocess_fn) \
    .map_batches(
        MyModelCallableCls,
        num_gpus=1,
        compute=ActorPoolStrategy(size=N)) \
    .write_parquet(...)
```

Ray provides generic platform for LLMs

01	Simplify orchestration and scaling	• Spot instance support for data parallel training • Easily spin up and run distributed workloads on any cloud • Can run on a lot of cheap preemptible GPUs
02	Inference and serving	• Ability to support complex pipelines integrating business logic • Ability to support multiple node serving
03	Training	• Integrates distributed training with distributed hyperparameter tuning • Very fast interactive development model • Minimal code changes to enable training

High Performance Inference



Completing seven sequences using continuous batching. Left shows the batch after a single iteration, right shows the batch after several iterations. Once a sequence emits an end-of-sequence token, we insert a new sequence in its place (i.e. sequences S5, S6, and S7). This achieves higher GPU utilization since the GPU does not wait for all sequences to complete before starting a new one.

This Talk

Adoption Highlights

Ray Intro

Use Case Details

Getting Hands Dirty

Getting ✋✋ Dirty! – Simple Stuff

Linux `grep` on TBs of data?

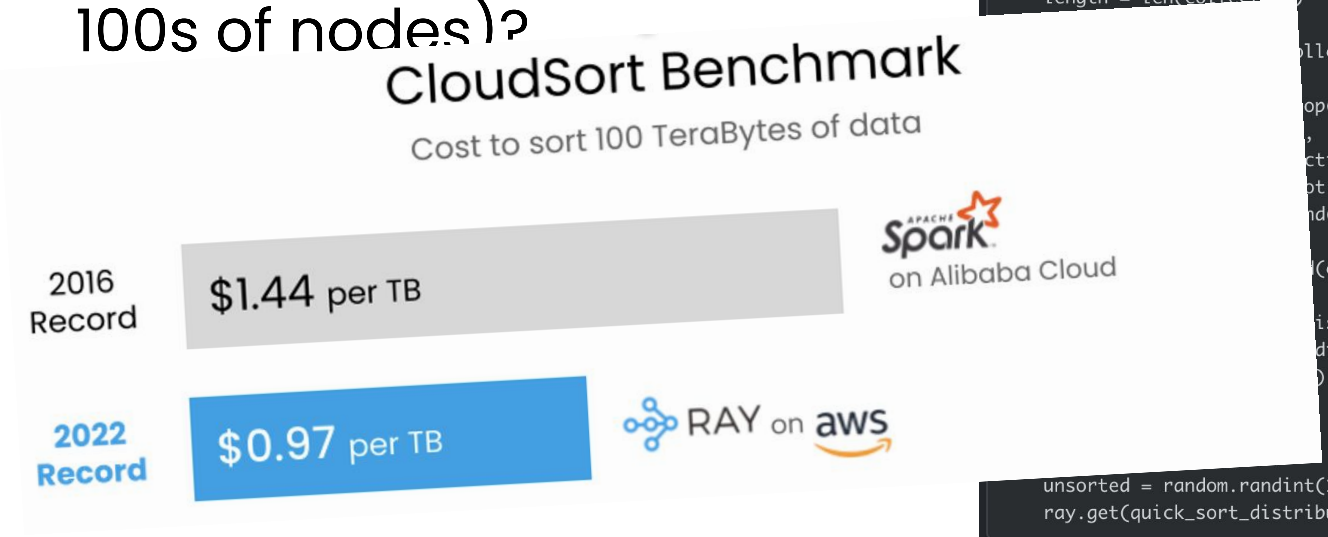
```
import ray
import sys
import re

regex_pattern = sys.argv[2]
pattern = re.compile(regex_pattern)

ds = ray.data.read_text(sys.argv[1])
result = ds.map_batches(lambda batch: [v if pattern.search(v) else
None for v in batch])
for r in result.iter_rows():
    if r != None:
        print(r)
```

Getting ✋✋ Dirty!

Back to interview time!

QuickSort (but on 100s of nodes)?

```
def quick_sort(collection):
    length = len(collection)
    if length <= 1:
        return collection
    else:
        pivot = collection.pop()
        greater, lesser = [], []
        for element in collection:
            if element > pivot:
                greater.append(element)
            else:
                lesser.append(element)
        return quick_sort(lesser) + [pivot] + quick_sort(greater)


@ray.remote
def quick_sort_distributed(collection):
    length = len(collection)
                                          llection)

                                          op()
                                          , []
                                          ction:
                                          ot:
                                          nd(element)

                                          (element)

                                          istributed.remote(lesser)
                                          distributed.remote(greater)
                                          ) + [pivot] + ray.get(greater)



    unsorted = random.randint(1000000, size=(4000000)).tolist()
    ray.get(quick_sort_distributed.remote(unsorted))
```



GPT-J-6B Fine-Tuning with Ray AIR and DeepSpeed

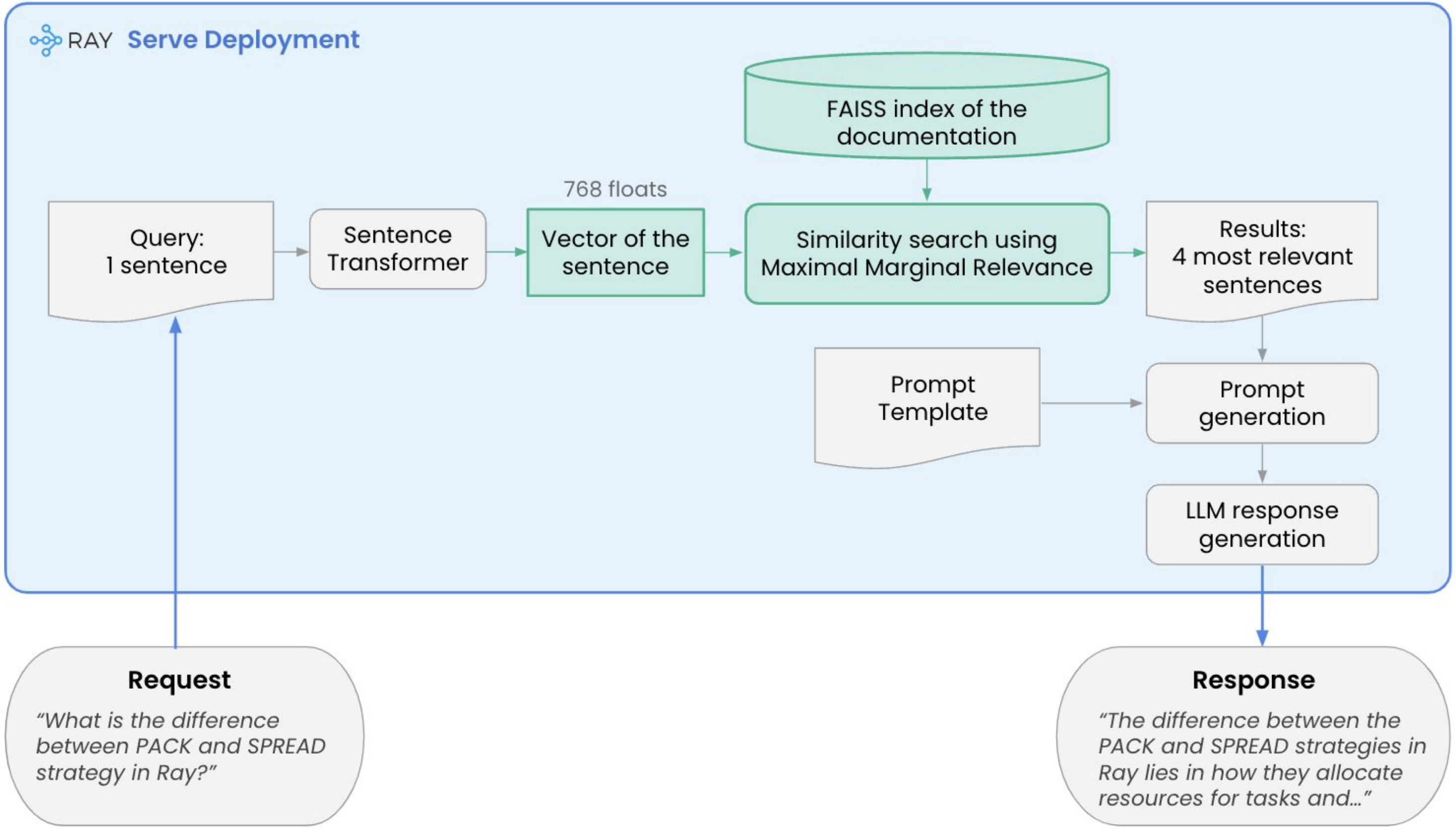
In this example, we will showcase how to use the Ray AIR for **GPT-J fine-tuning**. GPT-J is a GPT-2-like causal language model trained on the Pile dataset. This particular model has 6 billion parameters. For more information on GPT-J, click here.

```python
from ray.train.huggingface import HuggingFaceTrainer
from ray.air.config import ScalingConfig
from ray.data.preprocessors import Chain


trainer = HuggingFaceTrainer(
    trainer_init_per_worker=trainer_init_per_worker,
    trainer_init_config={
        "batch_size": 16,  # per device
        "epochs": 1,
    },
    scaling_config=ScalingConfig(
        num_workers=num_workers,
        use_gpu=use_gpu,
        resources_per_worker={"GPU": 1, "CPU": cpus_per_worker},
    ),
    datasets={"train": ray_datasets["train"], "evaluation":
ray_datasets["validation"]},
    preprocessor=Chain(splitter, tokenizer),
)
```

Finally, we call the `fit()` method to start training with Ray AIR. We will save the `Result` object to a variable so we can access metrics and checkpoints.

```python
results = trainer.fit()
```

Answer Questions with RAY + LangChain



Roadmap

- Performance
  - First, define clear benchmarks (need your input!)
  - Optimizations: GPU, scheduling, etc.
- Usability / Quality
  - Push Ray AI Runtime to GA
  - Simpler APIs, easier onboarding
- Better support for LLM
  - Integrations with new models (e.g. Falcon)
  - Integrations with new optimizations (e.g. vLLM)

Summary

Ray is positioned to be the **New AI Infra** in the large model / generative AI era. Let's collaborate and make it happen!

Call to action

- Get started at docs.ray.io
- Dive deeper on https://github.com/ray-project/ray/
- Ray Summit 2023 is coming up!