

# Distributed Python with Ray: Hands-on with the Ray Core APIs

---

**Stephanie Wang**

Software Engineer, Ray Core @ Anyscale  
PhD student @ UC Berkeley  
ODSC East 2022



\$whoami

- Software Engineer @ Anyscale
- Lead committer @ the Ray project
- PhD student @ UC Berkeley

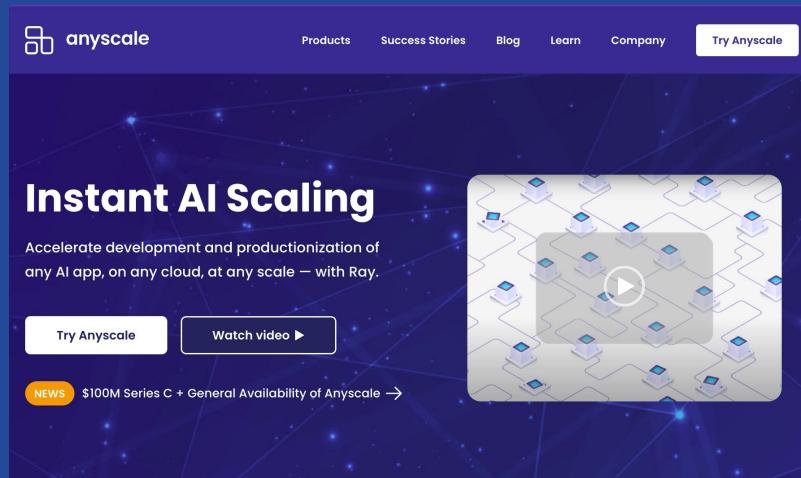


# Anyscale

**Who we are:** Original creators of Ray

**What we do:** Provide managed service for Ray on public cloud to scale AI workloads

**Why do it:** Make distributed computing easy and simple for everyone



# Overview

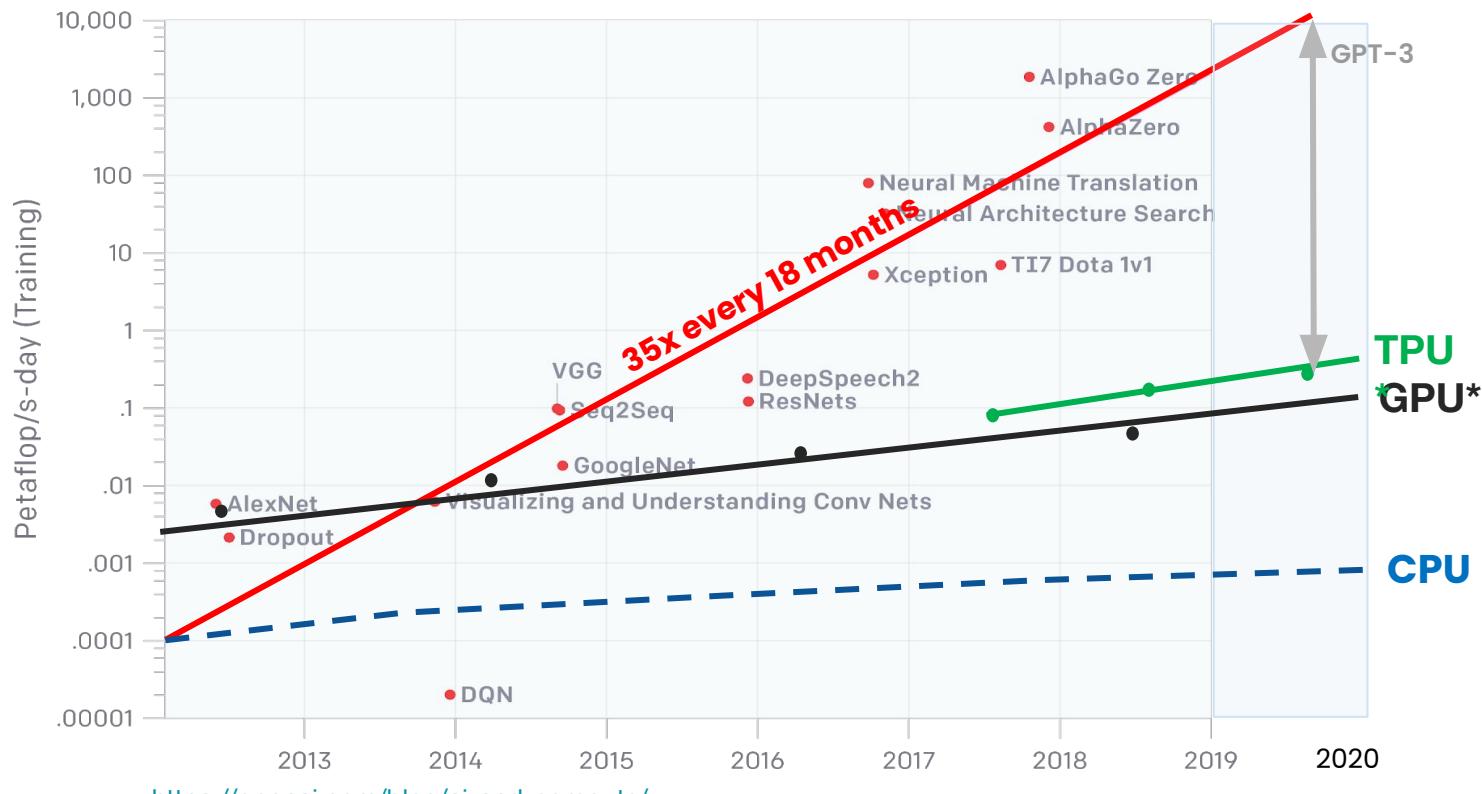
- **Why & What's Ray & Ray Ecosystem**
- **Ray Architecture & Components**
- **Ray Core Design Patterns & APIs**
- **Hands-on Tutorials**

<https://tinyurl.com/pycon-ray-tutorial>

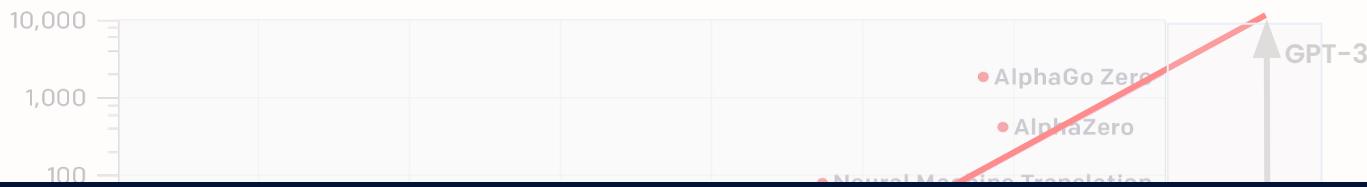
# Why Ray ?

-  Machine learning is pervasive in every domain
-  Distributed machine learning is becoming a necessity
-  Python is the default language for DS & ML

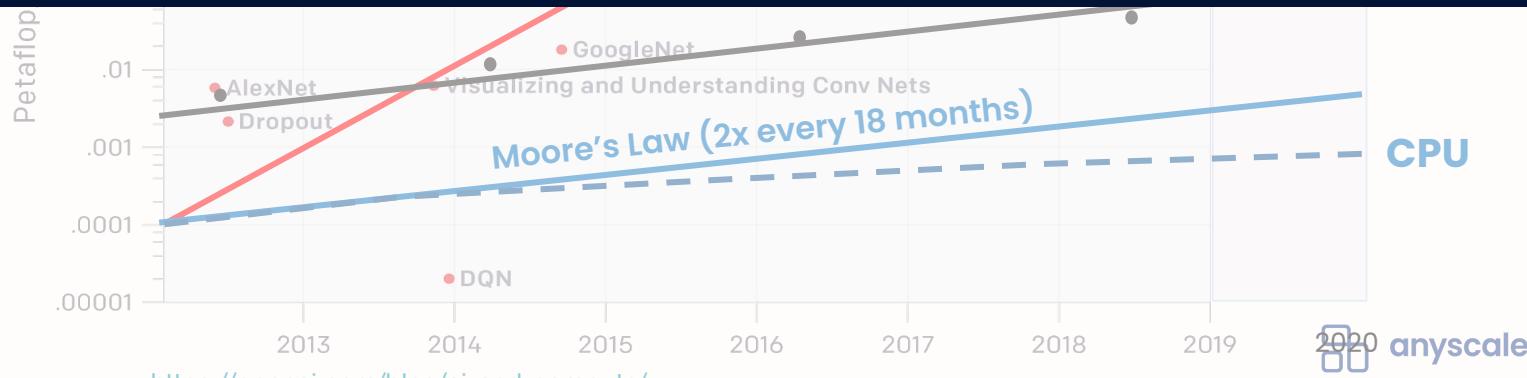
# Compute Demand - supply problem



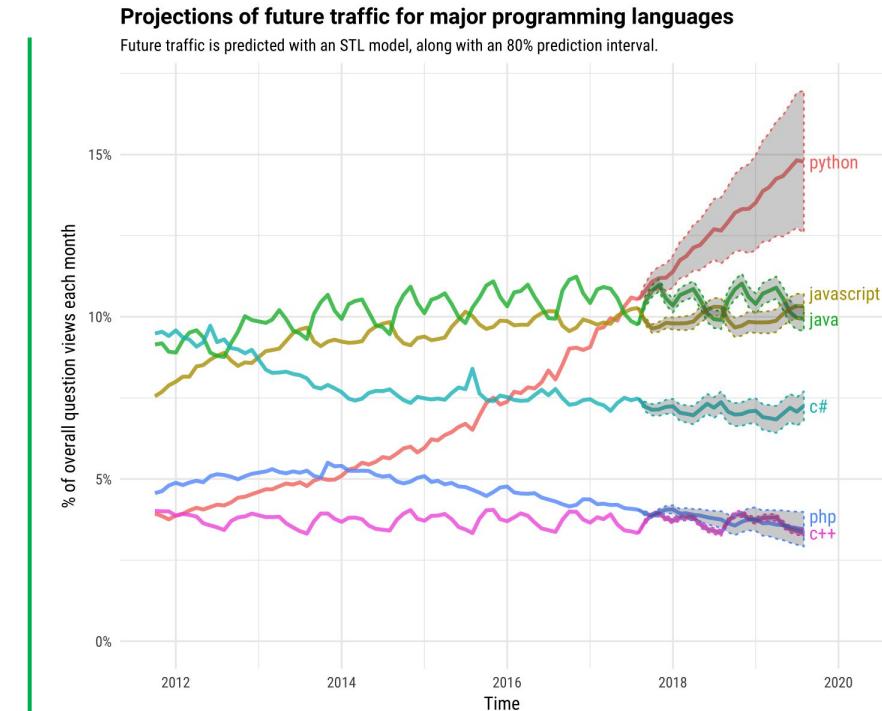
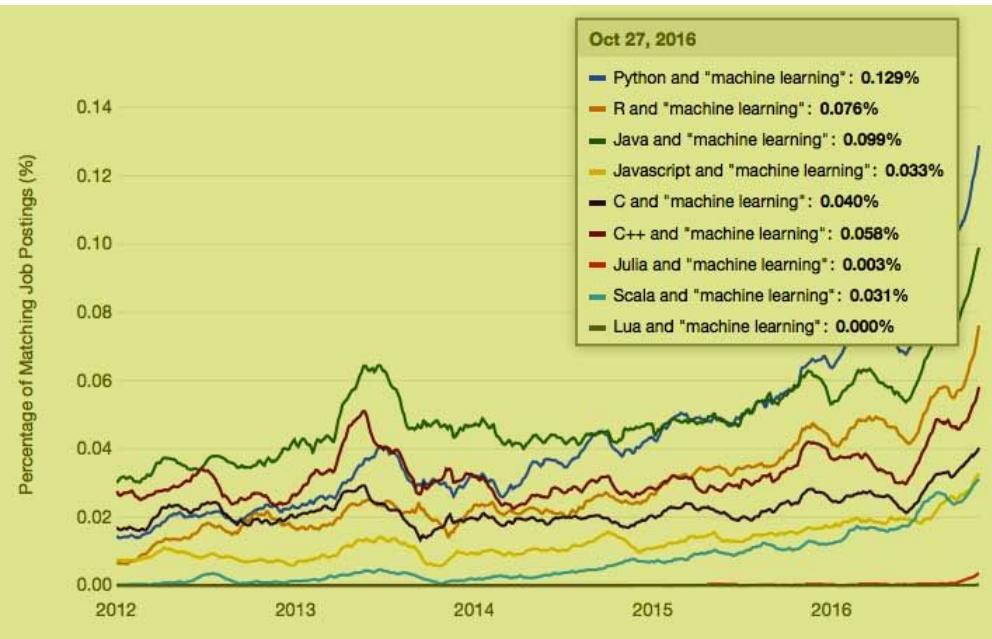
# Specialized hardware is also not enough



No way out but to distribute!

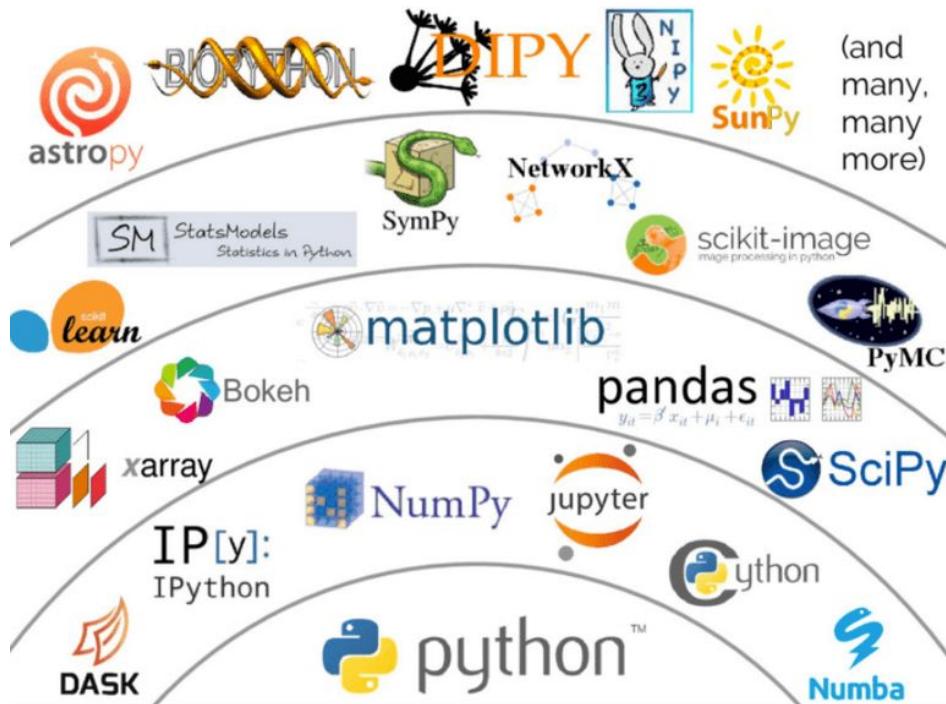


# Python's growth as programming language

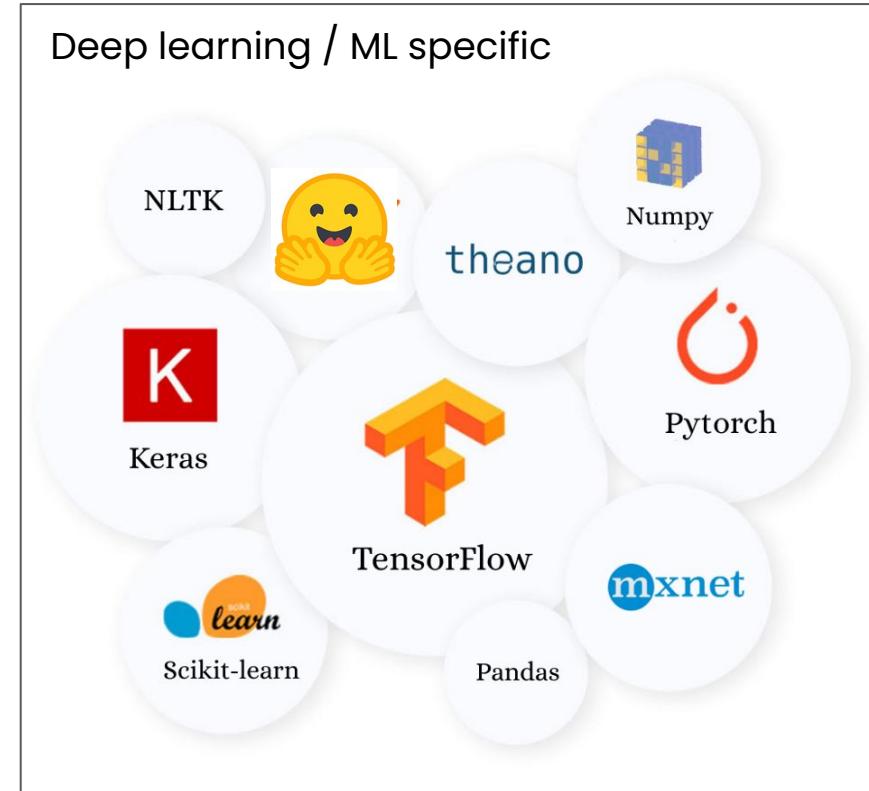




# Python data science ecosystem dominating



Deep learning / ML specific



# Origins of Ray

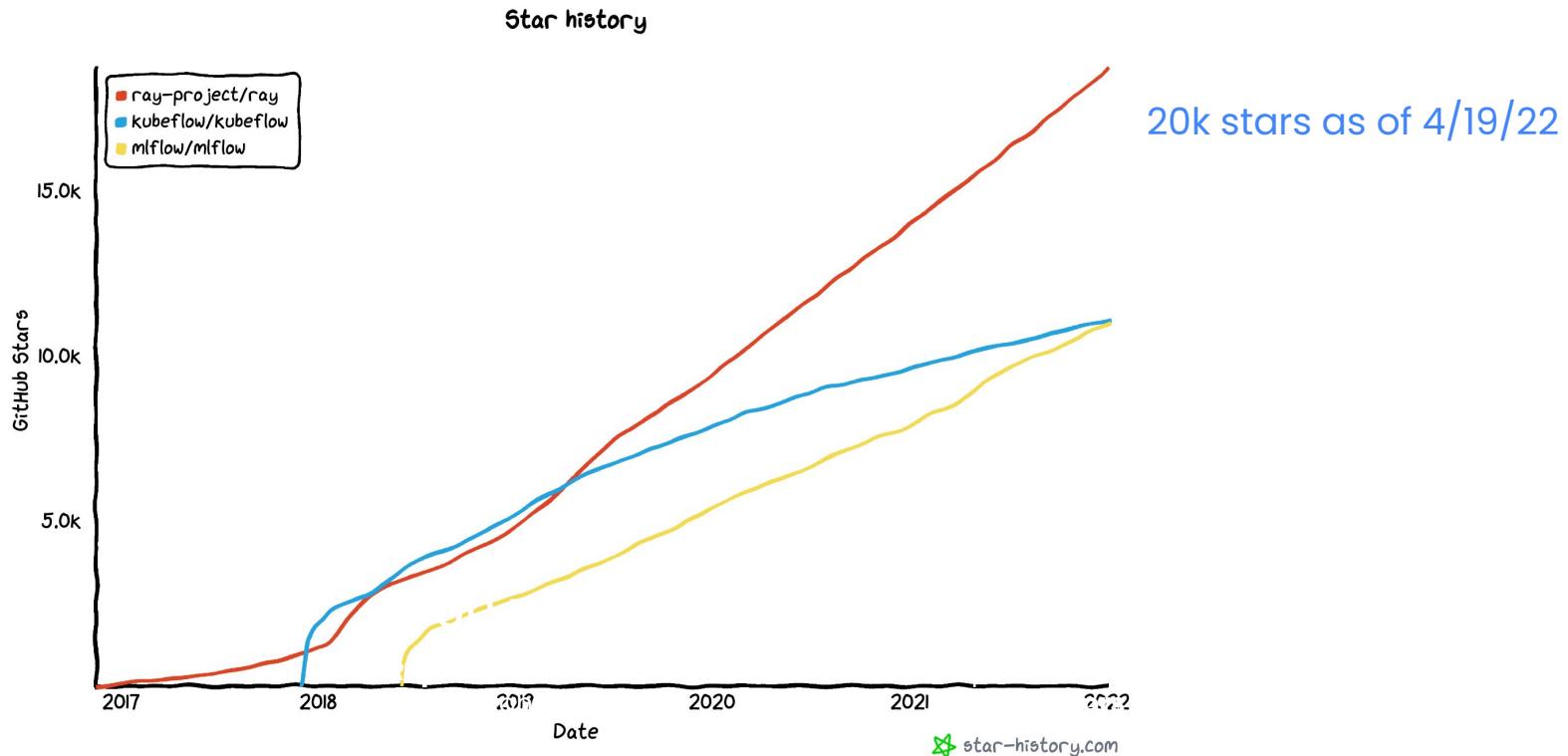


# Caffe



# RAY

# Growth of open-source Ray adoption

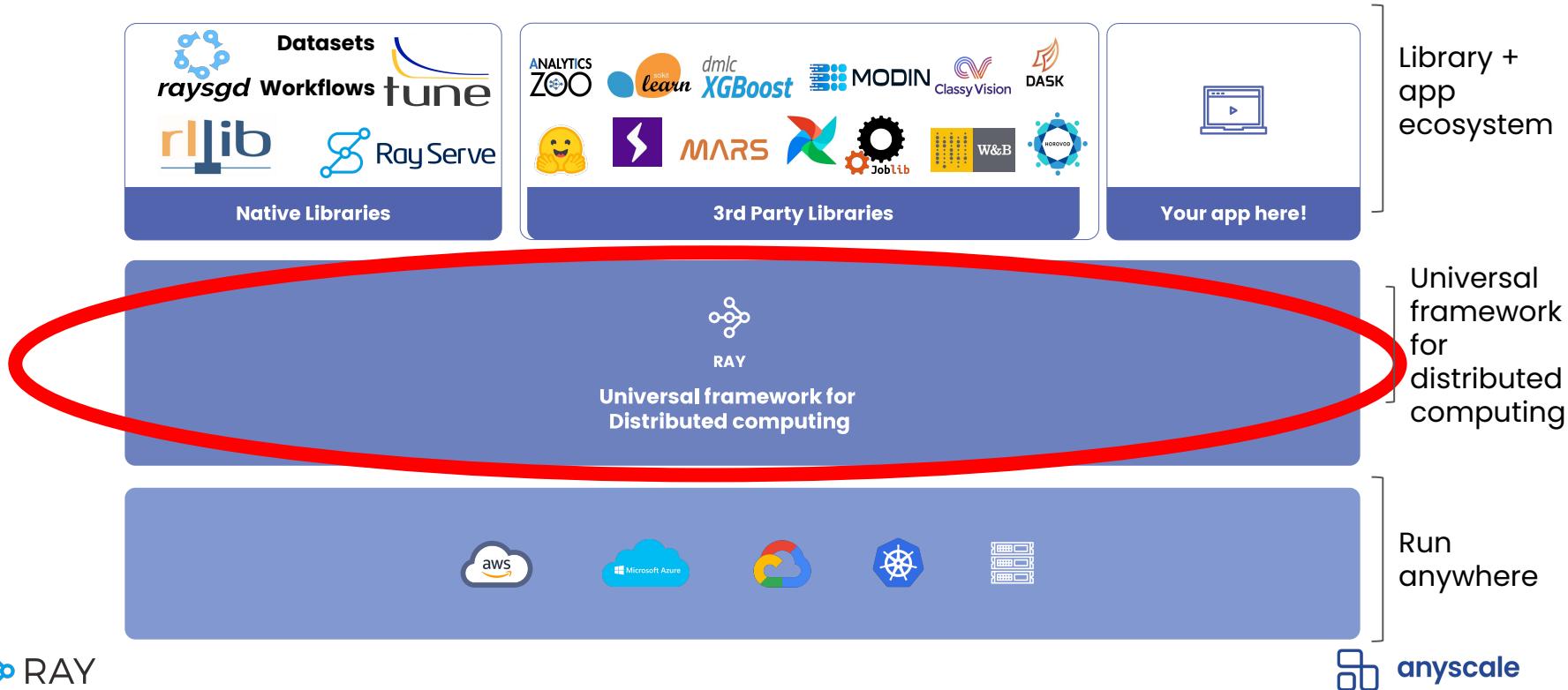


# What is Ray?

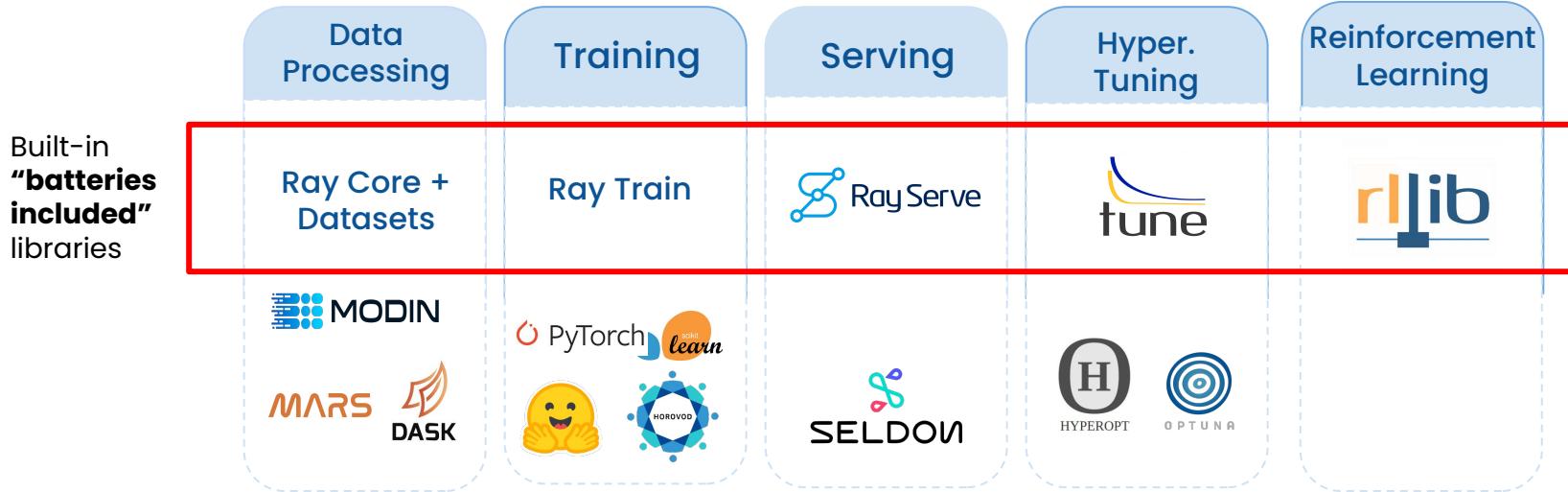
- A simple/general library for distributed computing
- An ecosystem of Python libraries (for scaling ML and more)
- Runs on laptop, public cloud, K8s, on-premise

A layered cake of functionality and capabilities for scaling ML workloads

# The Ray Layered Cake and Ecosystem



# Rich ecosystem for scaling ML workloads



Only use the libraries you need!

# Companies scaling ML with Ray

amazon

intel

dendra  
SYSTEMS

Uber

Microsoft

VISA

McKinsey  
& Company

ANT  
GROUP

rbi  
restaurant  
brands  
international

WILD  
LIFE

robinhood

NVIDIA.

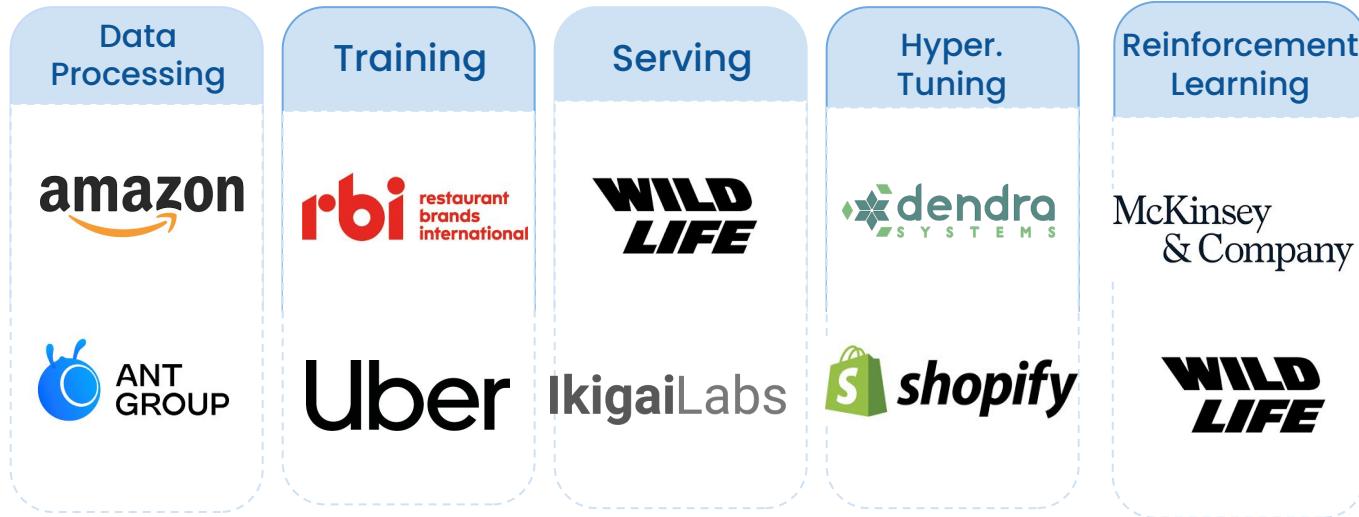
shopify

IBM

2 $\sigma$  TWO SIGMA

Alibaba.com

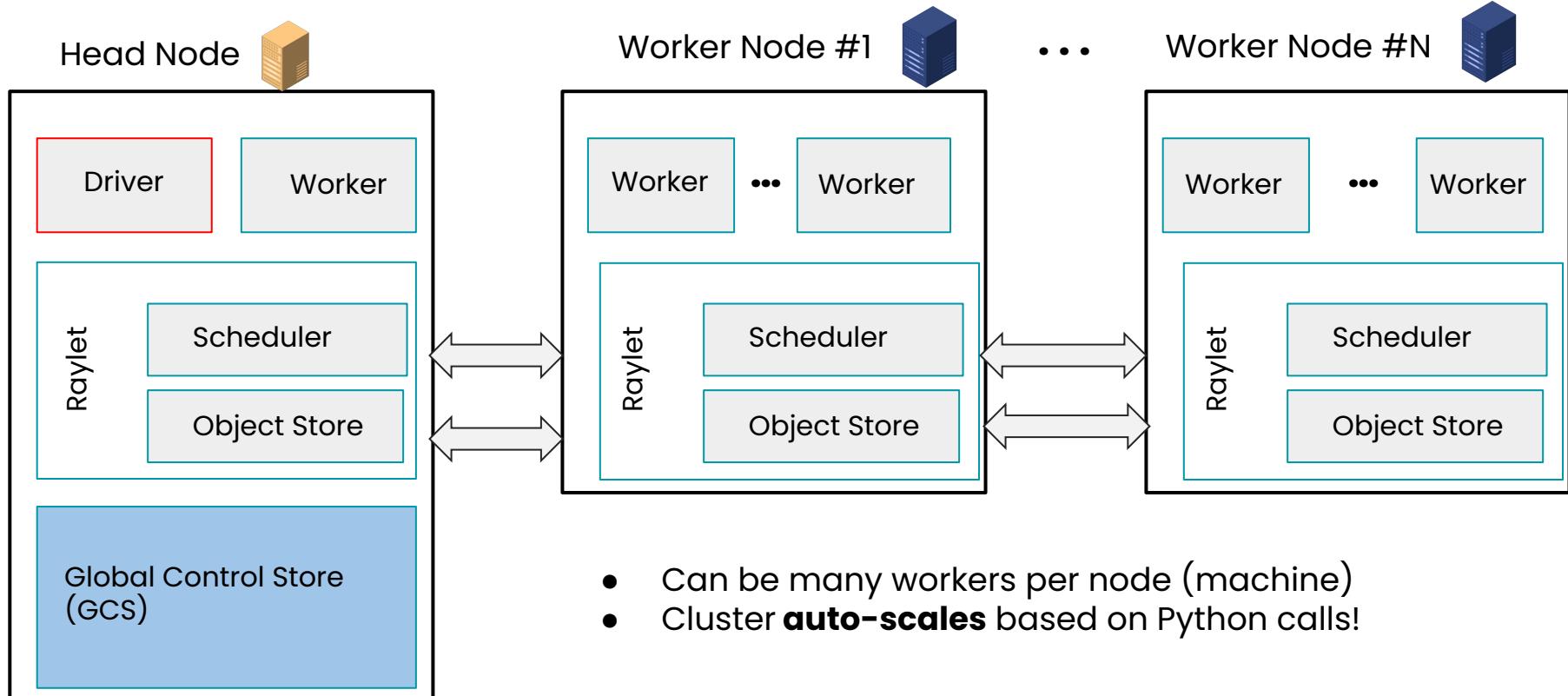
# Companies scaling ML with Ray



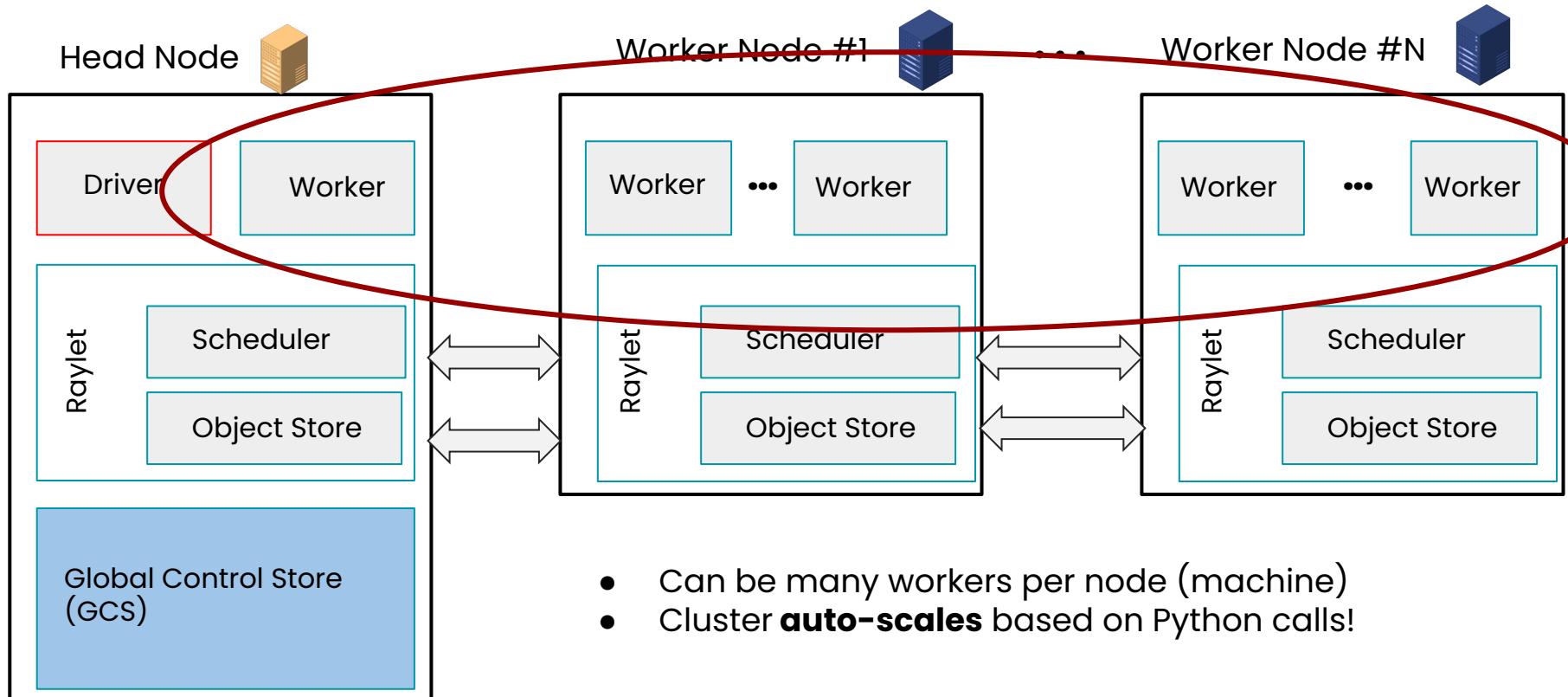
- <https://eng.uber.com/horovod-ray/>
- <https://www.anyscale.com/blog/wildlife-studios-serves-in-game-offers-3x-faster-at-1-10th-the-cost-with-ray>
- <https://www.ikigailabs.com/blog/how-ikigai-labs-serves-interactive-ai-workflows-at-scale-using-ray-serve>

# Ray Architecture & Components

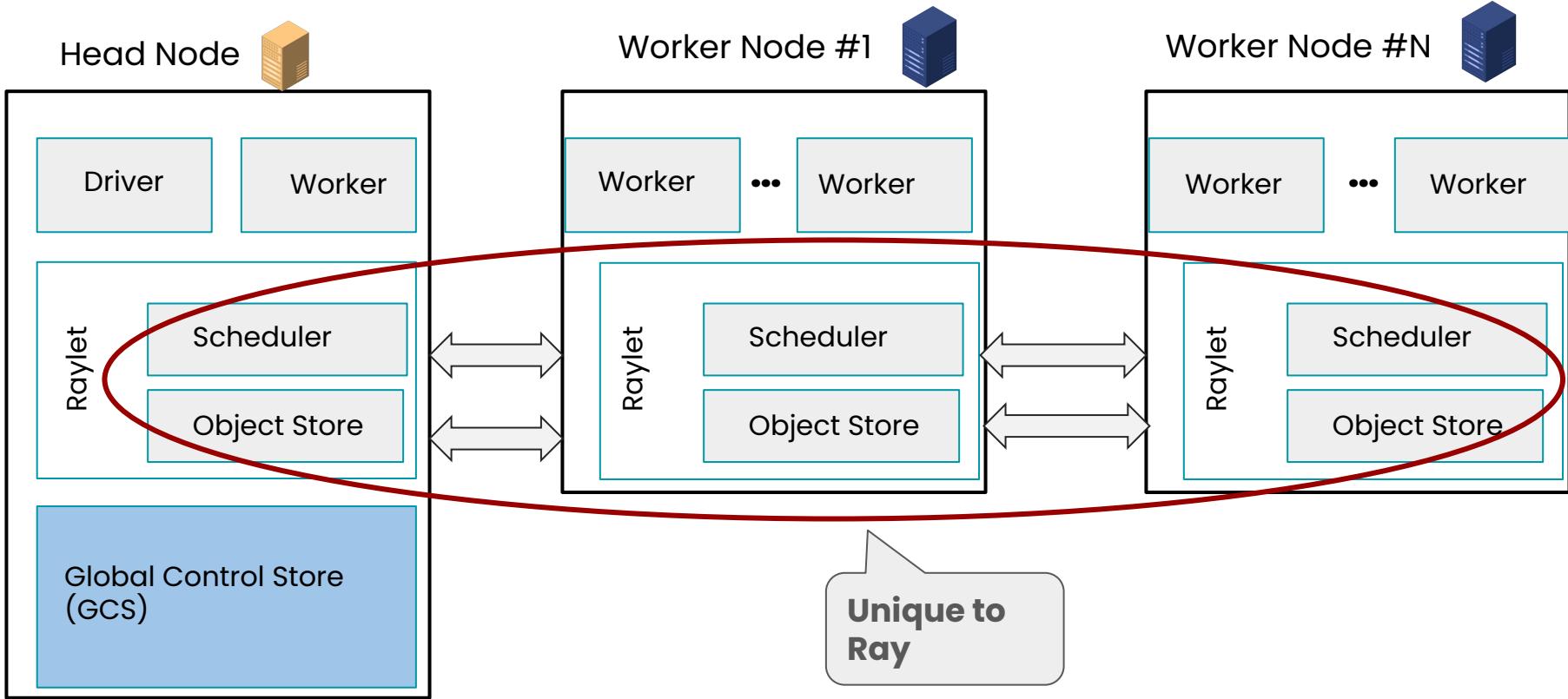
# Anatomy of a Ray cluster



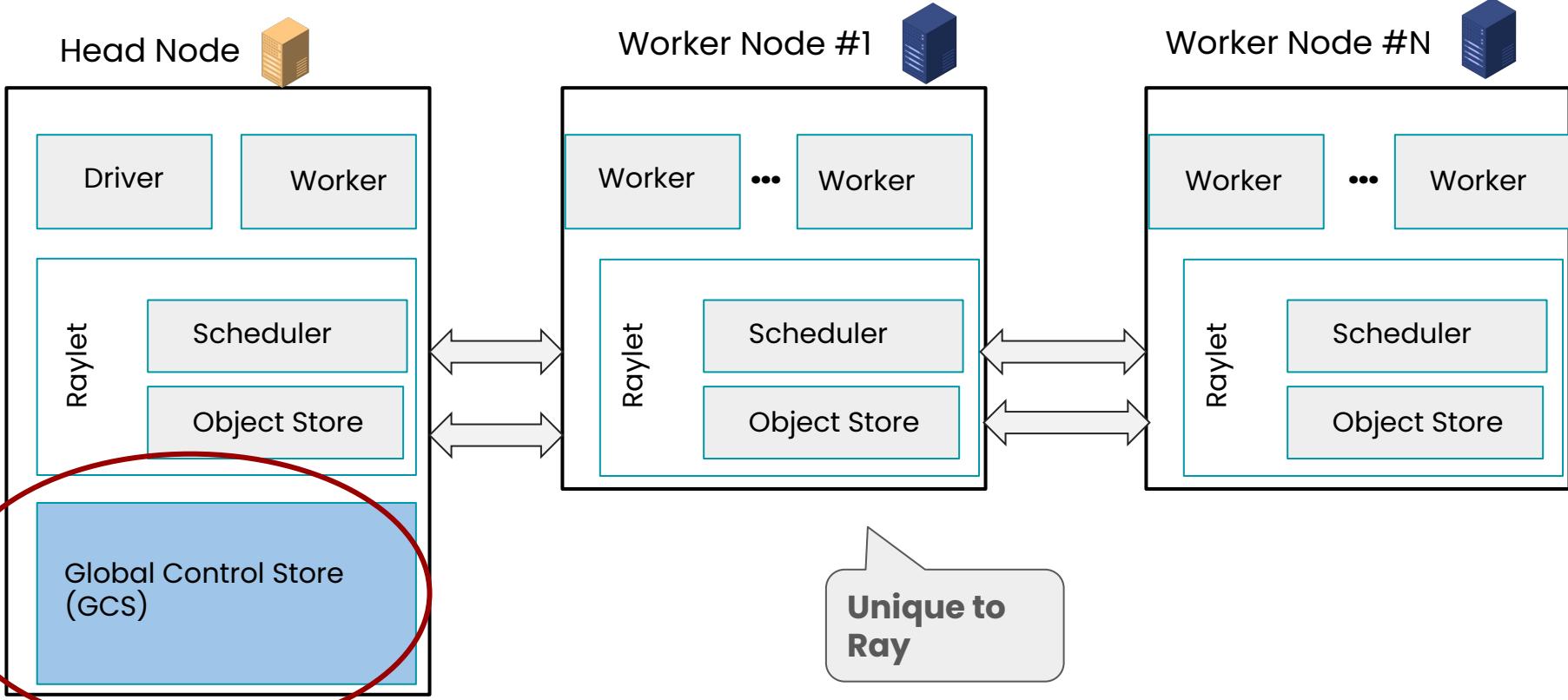
# Anatomy of a Ray cluster



# Anatomy of a Ray cluster



# Anatomy of a Ray cluster



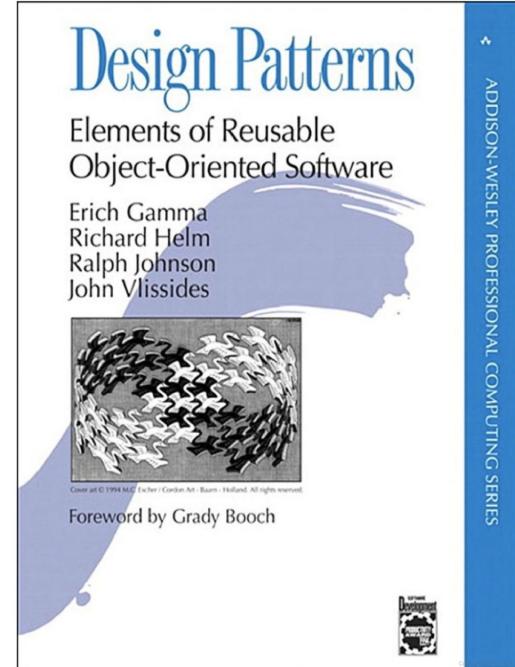
# Ray Distributed Design Patterns & APIs



# Ray Basic Design Patterns

- Ray Parallel Tasks
  - Functions as stateless units of execution
  - Functions distributed across a clusters as tasks
- Ray Objects or Futures
  - Distributed (immutable) Object stored in cluster
  - Retrievable when available
  - Enable asynchronous execution
- Ray Actors
  - Stateful service on a cluster
  - Message passing and maintains state

1. [Patterns for Parallel Programming](#)
2. [Ray Design Patterns](#)
3. [Ray Distributed Library Integration Patterns](#)



# Function → Task

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

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

array1 = read_array(file1)
array2 = read_array(file2)
sum = add(array1, array2)
```

# 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)

array1 = read_array(file1)
array2 = read_array(file2)
sum = add(array1, array2)
```

# 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)
```

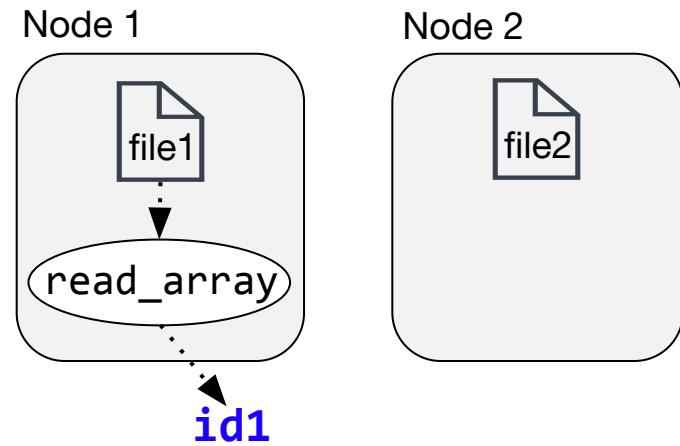
# Task API

Blue variables are ObjectRefs  
(similar to futures)

```
@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)
```



Return `id1` (future) immediately,  
before `read_array()` finishes

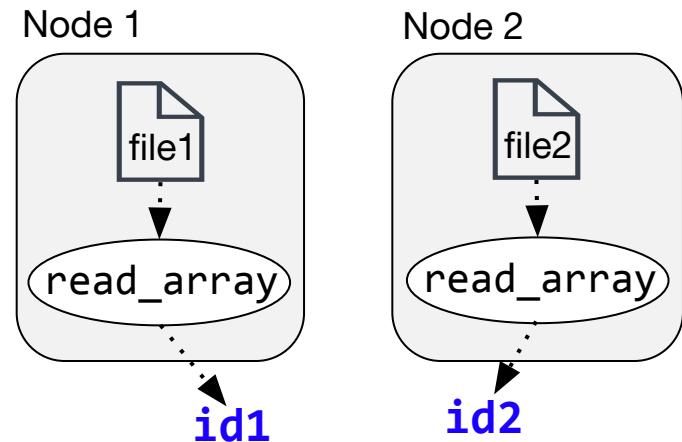
# Task API

Blue variables are Object IDs  
(similar to futures)

```
@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

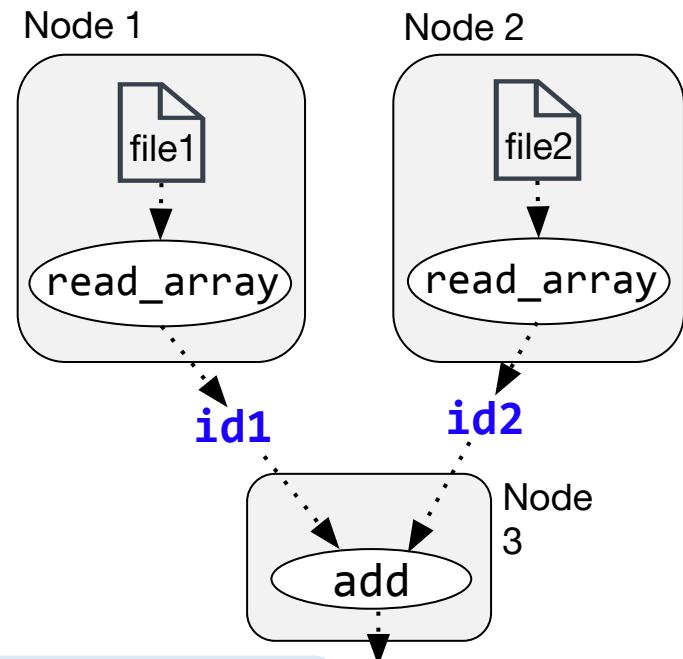
# Task API

Blue variables are Object IDs  
(similar to futures)

```
@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 scheduled,  
but not finished yet

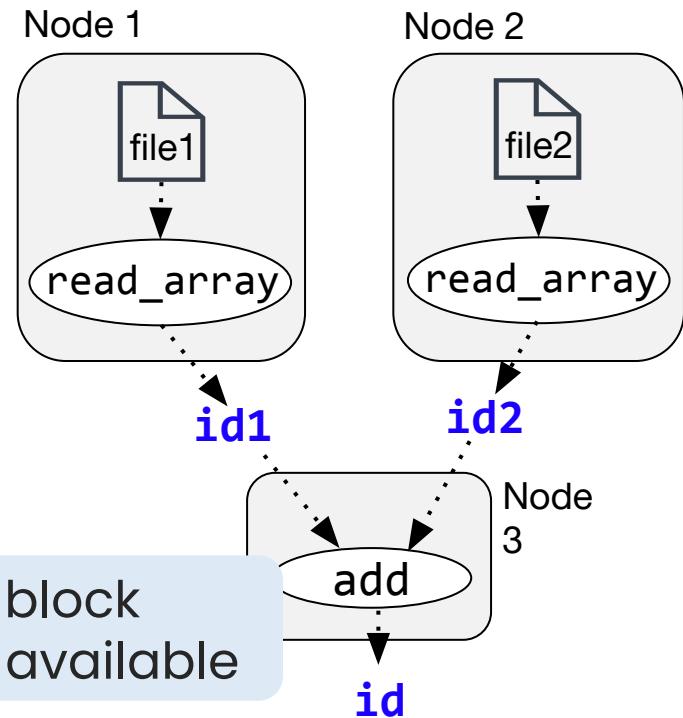
# Task API

Blue variables are Object IDs  
(similar to futures)

```
@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)
```



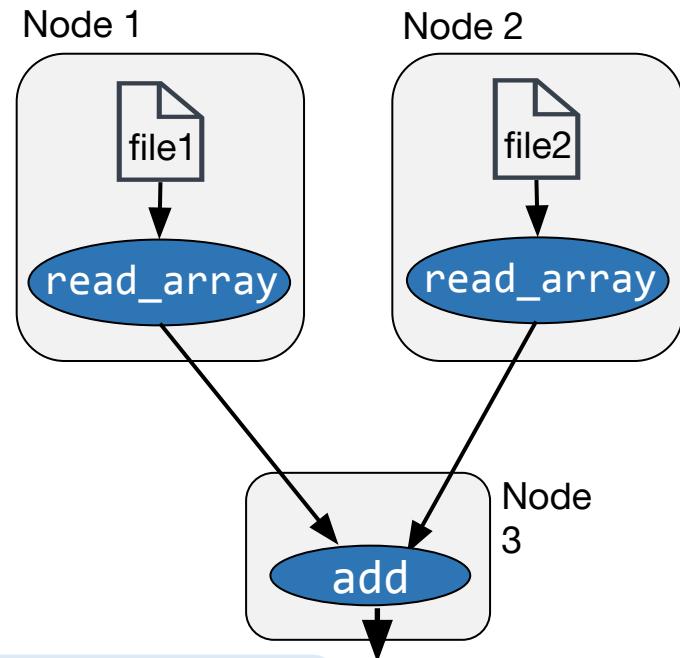
# Task API

Blue variables are Object IDs  
(similar to futures)

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

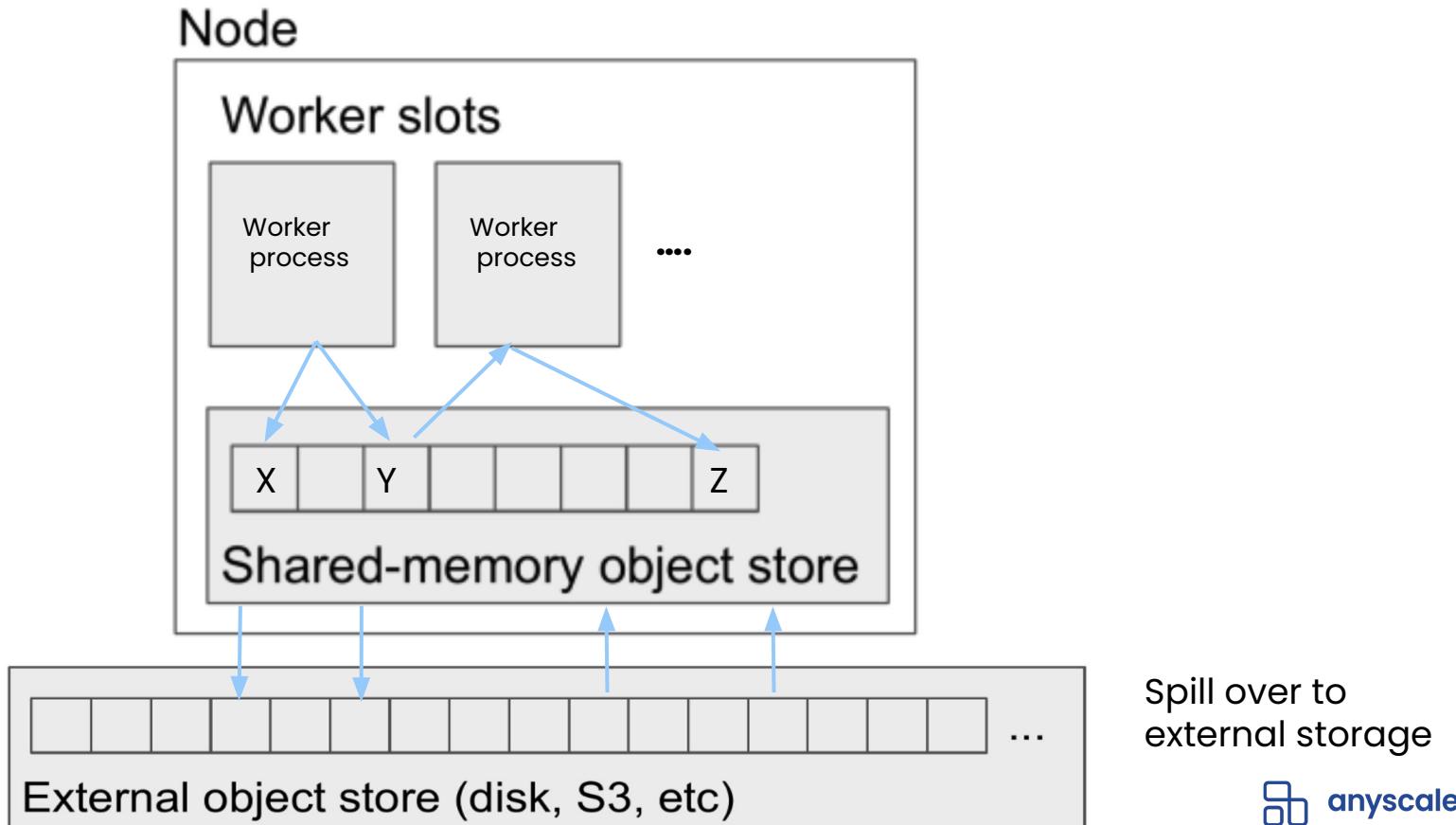
@ray.remote
def add(a, b):
    return np.add(a, b)

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

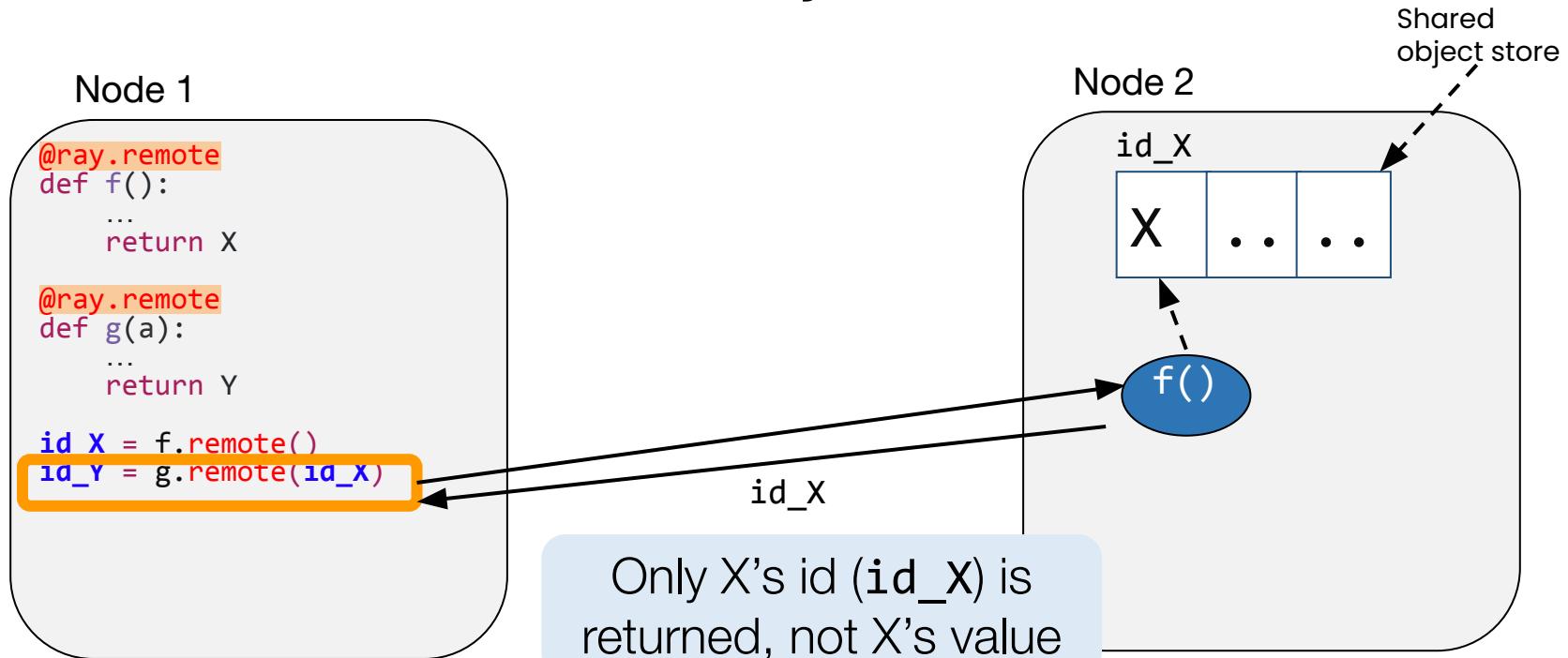


Task graph executed to sum  
compute sum

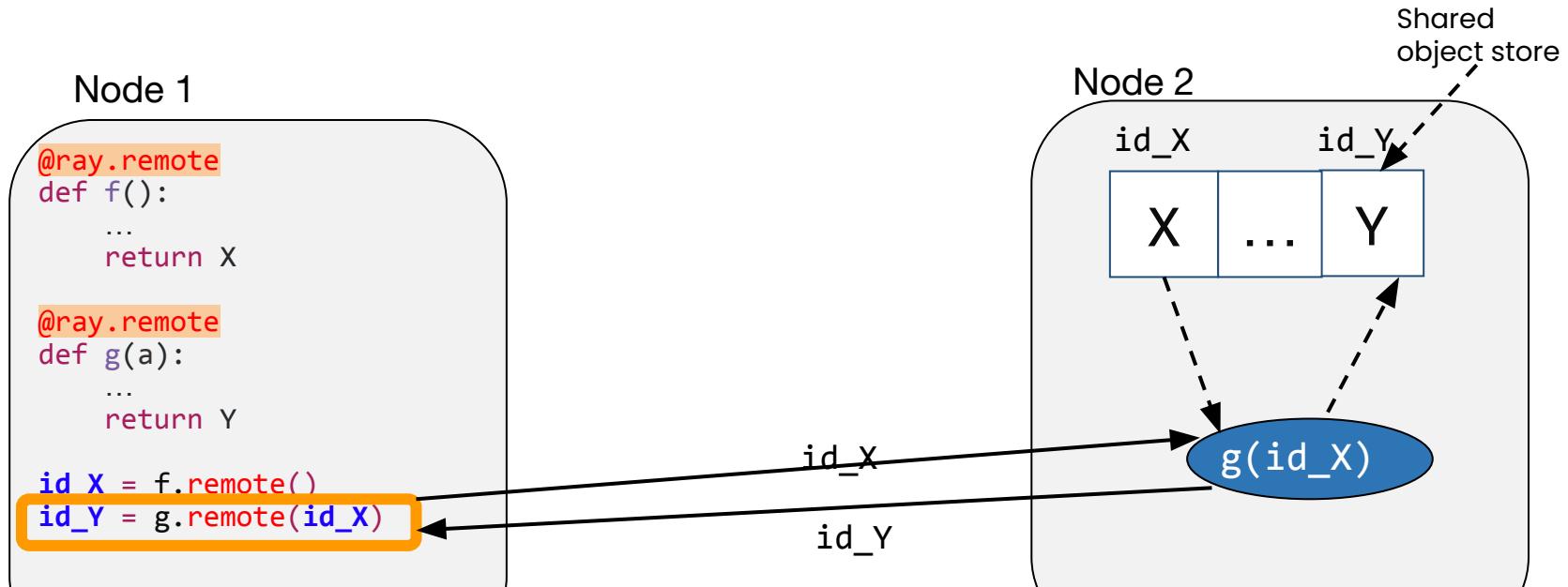
# Distributed Immutable object store



# Distributed Immutable object store



# Distributed object store

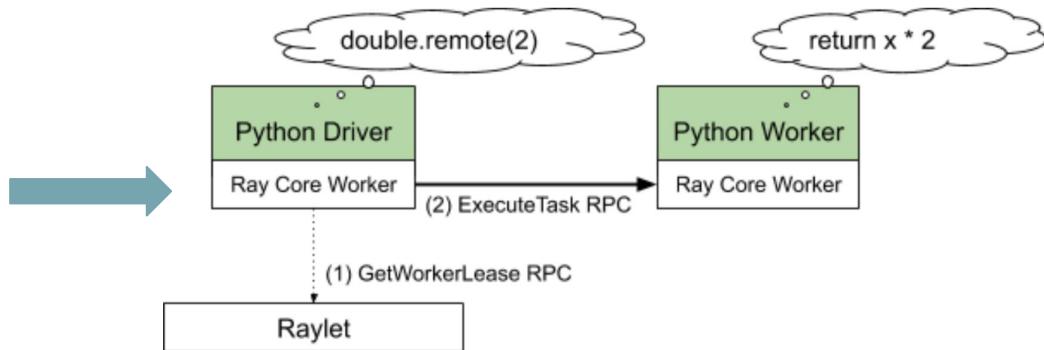


`g(id_X)` is scheduled on same node, so X is never transferred

# How Raylet Schedules Tasks

## Basic Ray Task Call

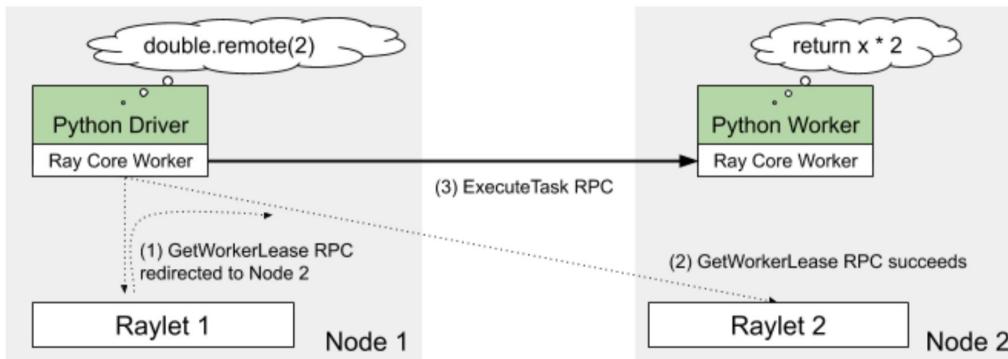
```
@ray.remote  
def double(x):  
    return x * 2  
  
fut1 = double.remote(2)  
assert ray.get(fut1) == 4
```



Components in green boxes represent Python code. Components in white boxes are part of the Ray common runtime written in C++. Joined boxes represent a process. Any Python driver or worker can call into the Ray C++ core worker library to execute further tasks. In this figure, all processes are running on the same machine. Ray uses gRPC as a unified communication layer for both local and remote procedure calls.

# Scaling to Multiple Nodes

1. The driver asks Raylet 1 for a worker to execute `double`. It has no free workers, but Raylet 1 knows Raylet 2 has free resources, and redirects the request to Raylet 2.
2. The driver sends `ExecuteTask` to the remote Python worker leased from Raylet 2 over gRPC.

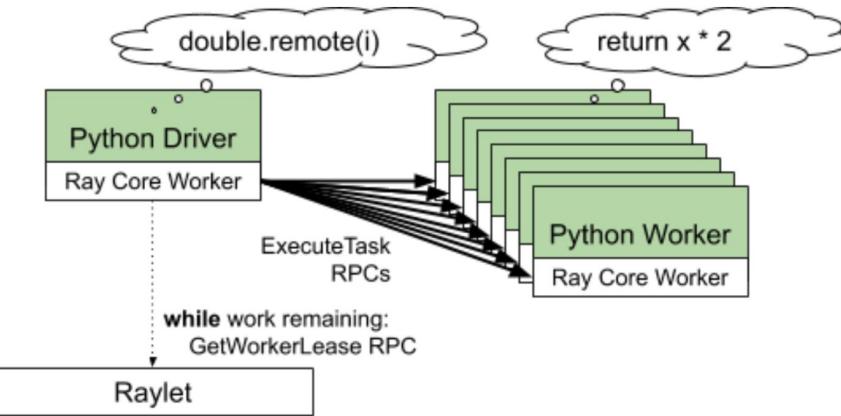


Tasks are sent to remote workers if there are no local resources available, transparently scaling Ray applications out to multiple nodes.

# Caching Scheduling Decisions

```
futures = [double.remote(i)
for i in range(10000)]
ray.get(futures)

# [0, 2, 4, 6 ...]
```



Once a scheduling decision is made by the Raylet, the worker returned can be reused for other tasks with the same resource requirements and input dependencies. This amortizes scheduling RPC overhead when executing many similar tasks. To avoid unfair monopolization of workers when there are multiple processes trying to submit tasks, callers are only allowed to reuse workers within a few hundred milliseconds of initial grant.

# 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

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

```
c = Counter()
val1 = c.inc()
val2 = 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)
```

# Class → Actor

```
@ray.remote(num_gpus=1)  
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()
```

Scheduled the same way as tasks,  
via direct RPC to the actor process

# Takeaways

- **Distributed computing is a necessity & norm**
- **Ray's vision: make distributed programming simple**
  - **Don't have to be a distributed systems expert. Just use `@ray.remote` :)**
- **Scale your ML workloads with Ray Libraries**

# Start learning Ray and contributing ...

**Getting Started:** pip install ray

**Documentation (docs.ray.io)**

*Quick start example, reference guides, etc*

**Join Ray Meetup**

*Meetup each month and publish recording to the members*

<https://www.meetup.com/Bay-Area-Ray-Meetup/>

**Forums (discuss.ray.io)**

*Learn / share with broader Ray community, including core team*

**Social Media (@raydistrributed, @anyscalecompute)**

*Follow us on Twitter and linkedIn*

**GitHub**

*Check out sources, file an issue, become a contributor, give us a Star :)*

<https://github.com/ray-project/ray>

# Tutorials

<https://tinyurl.com/odsc-ray-core>

# Thank you!

---

Let's stay in touch:

[swang@anyscale.com](mailto:swang@anyscale.com)

GitHub: [@stephanie-wang](https://github.com/@stephanie-wang)

<https://www.linkedin.com/in/swang93/>