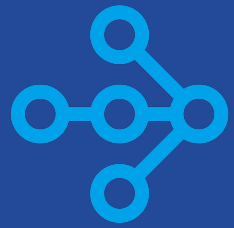
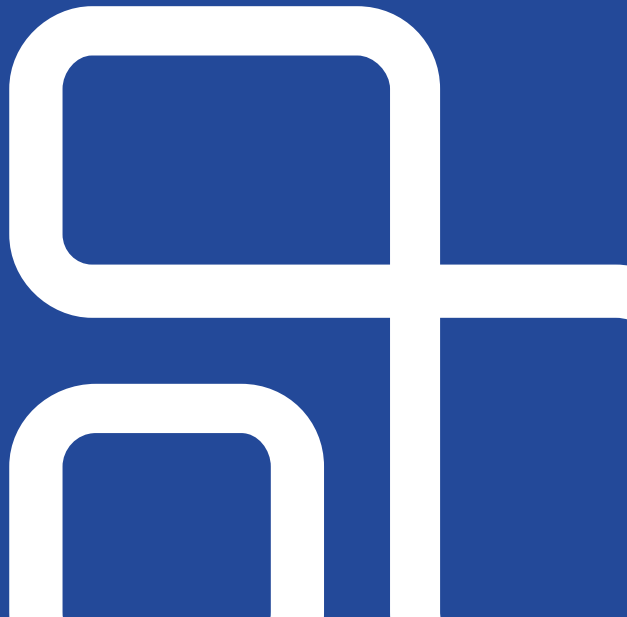
Introduction to Model Deployment with Ray Serve



Jules S. Damji, Lead Developer Advocate, Ray Team

Archit Kulkarni, Software Engineer, Ray Serve Team



anyscale

$whoami (Jules)

- Lead Developer Advocate @Anyscale
- Senior Developer Advocate @Databricks
- Led Developer Advocacy @Hortonworks
- Held software engineering positions:
  - Sun Microsystems
  - Netscape
  - @Home
  - LoudCloud/Opsware
  - Verisign
  - ProQuest/Ebrary

$whoami (Archit)

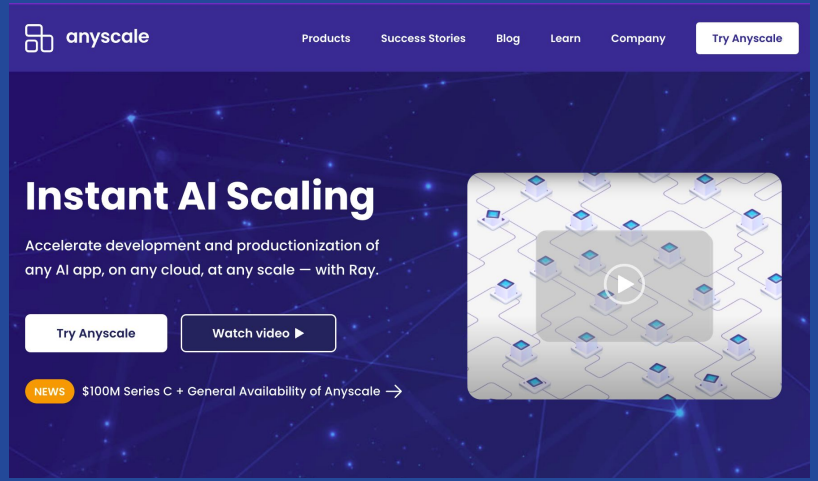
- Software Engineer at Anyscale working on OSS
  - Ray Serve
  - Ray Job Submission
  - Ray Runtime Environments
- Previously: Math + CS at UC Berkeley and Carnegie Mellon

Anyscale

Who we are: Original creators of Ray

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

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



anyscale

Agenda

- Why & What's Ray & Ray Ecosystem
- Ray Architecture & Components
- Ray Core Design Patterns & APIs
- Module -1 Ray Core Tutorial
- Break
- Ray Serve and Deployments
- Module -2 Ray Serve Tutorial

https://bit.ly/ray-serve-tutorial-mlops

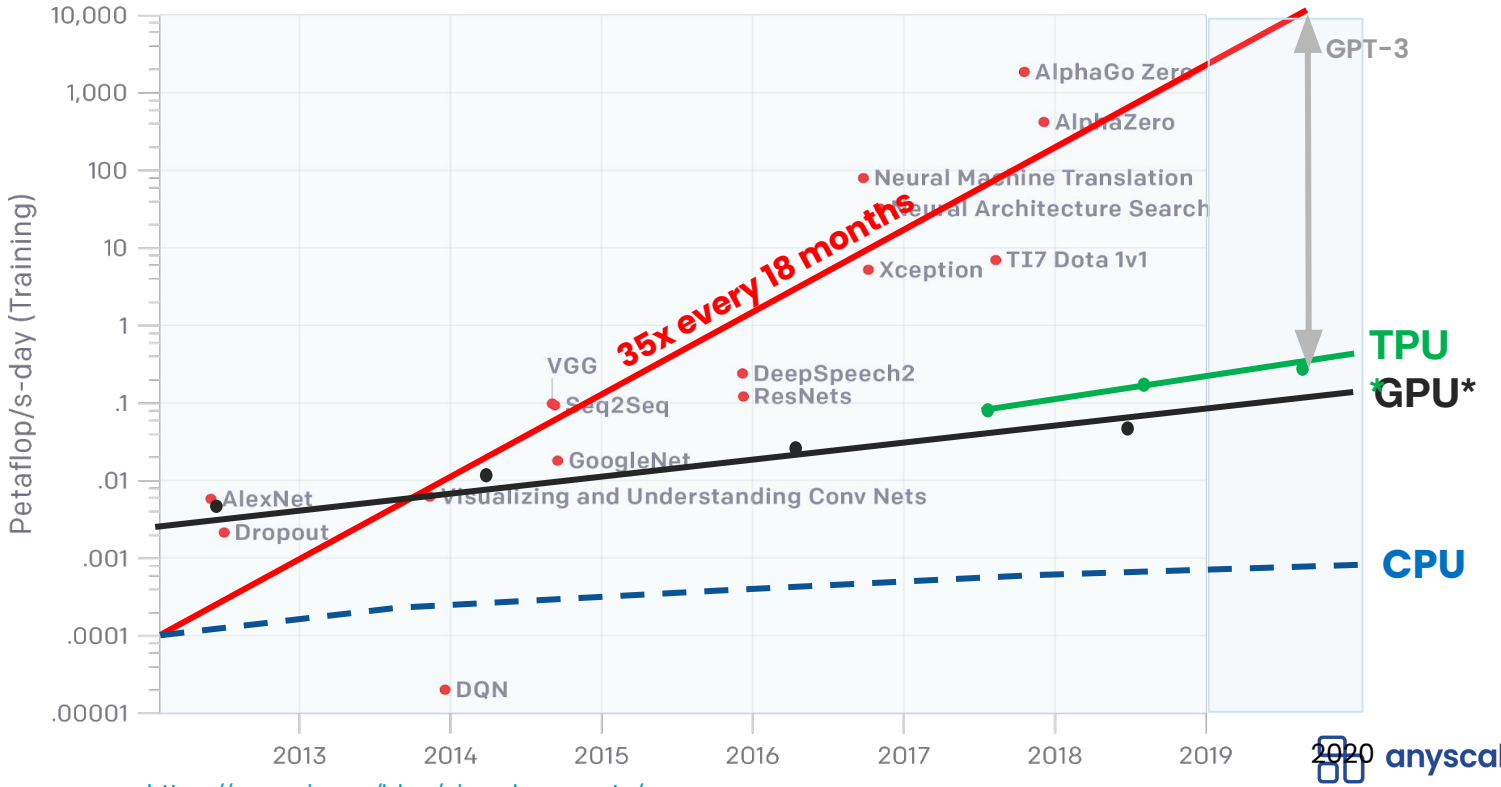
Why Ray  ?

-  Machine learning is pervasive in every domain

-  Distributed machine learning is a necessity

-  Python is the default language for DS & ML

Compute Demand - supply problem

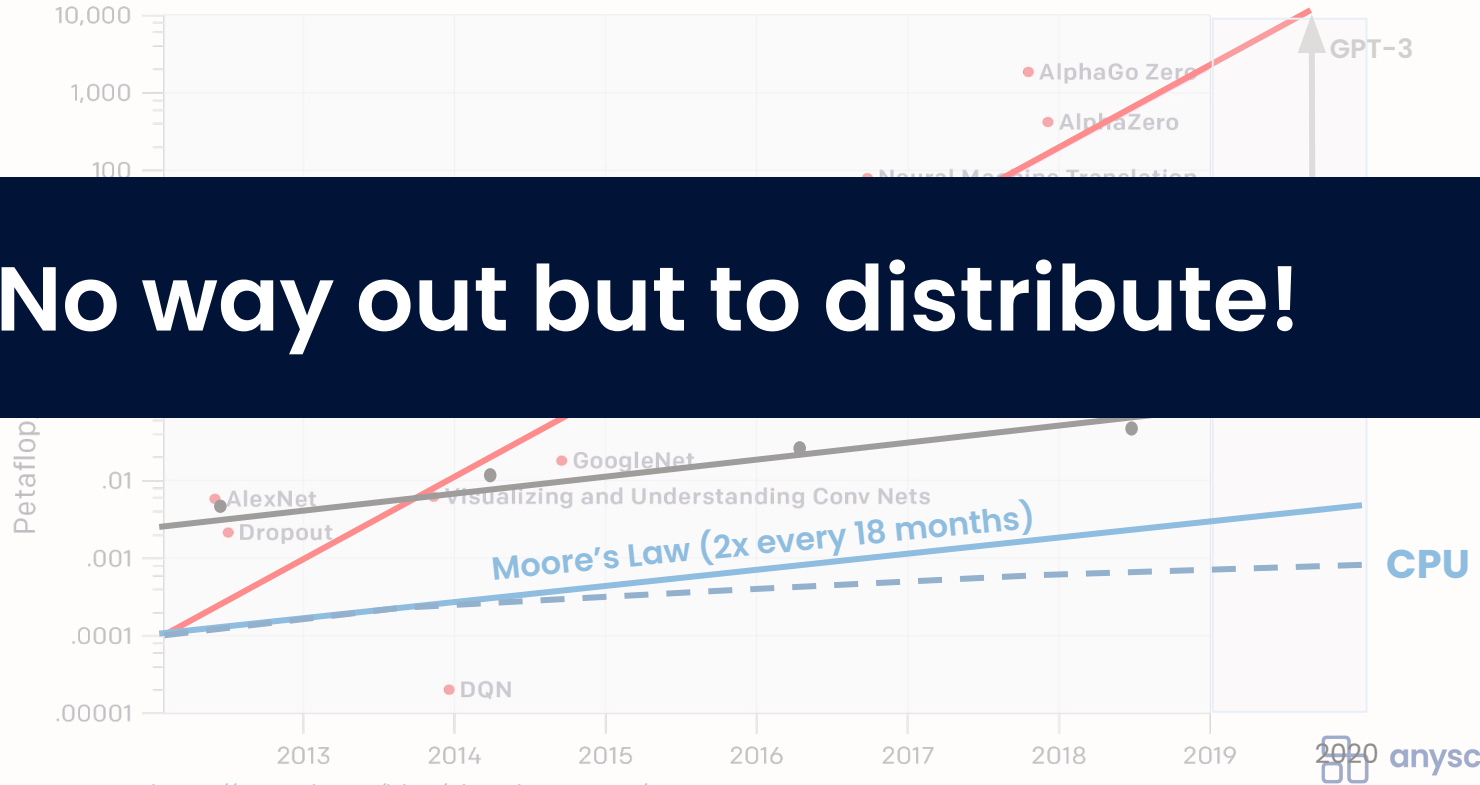


https://openai.com/blog/ai-and-compute/

RAY

anyscale

Specialized hardware is also not enough



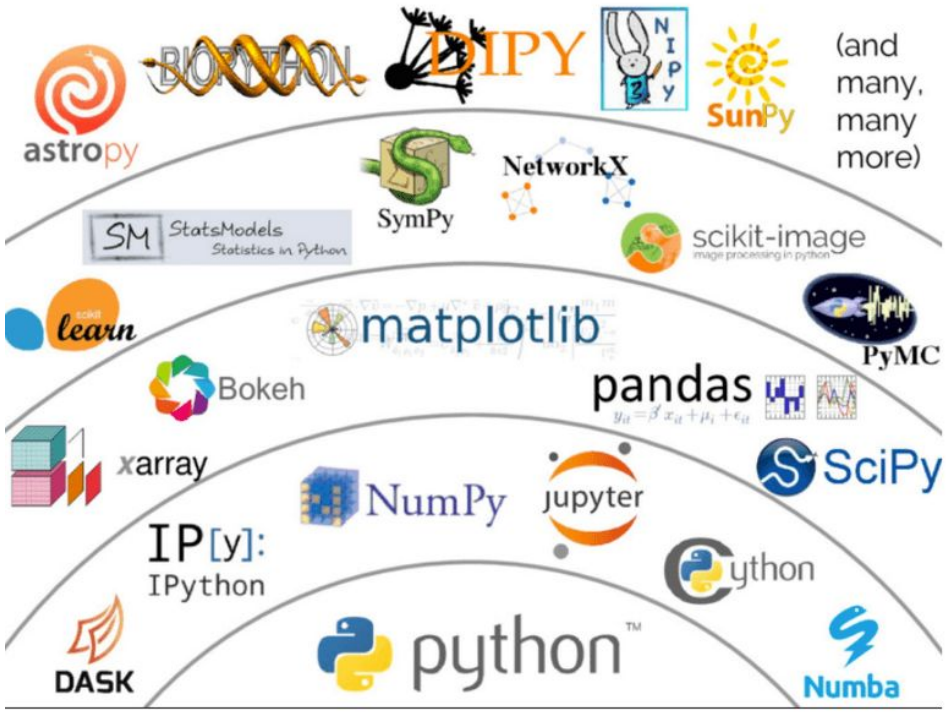
No way out but to distribute!

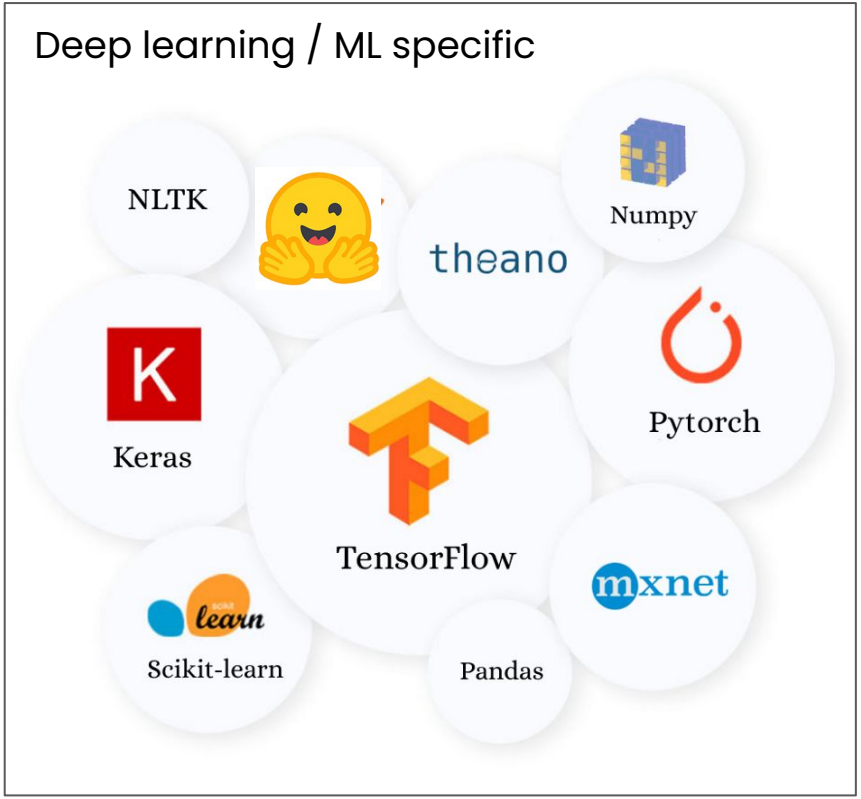
https://openai.com/blog/ai-and-compute/

RAY

anyscale

🐍 Python data science ecosystem dominating



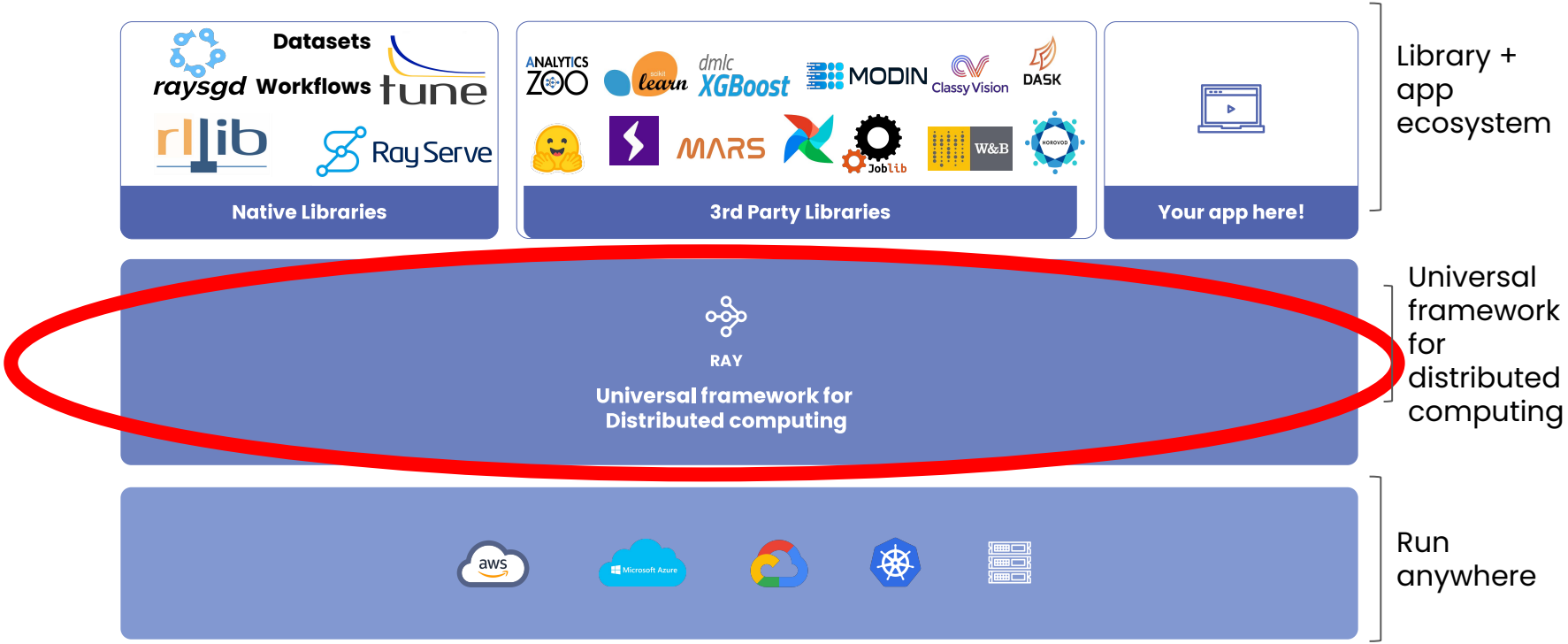


What is Ray? 

- 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

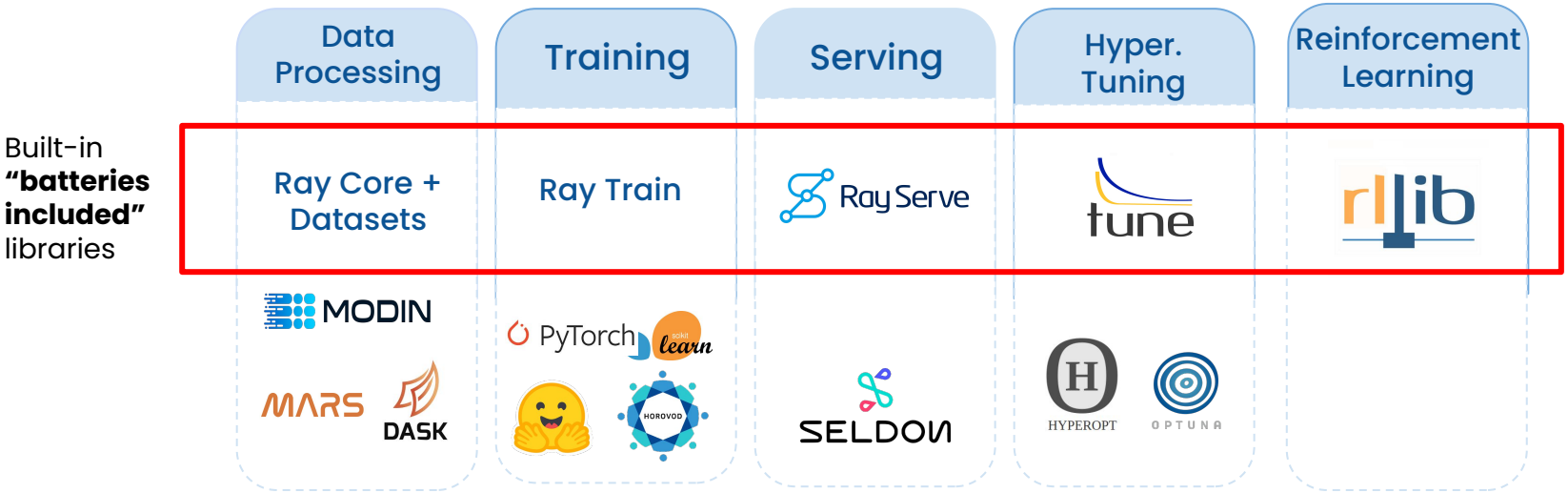
The Ray Layered Cake and Ecosystem



RAY

anyscale

Rich ecosystem for scaling ML workloads

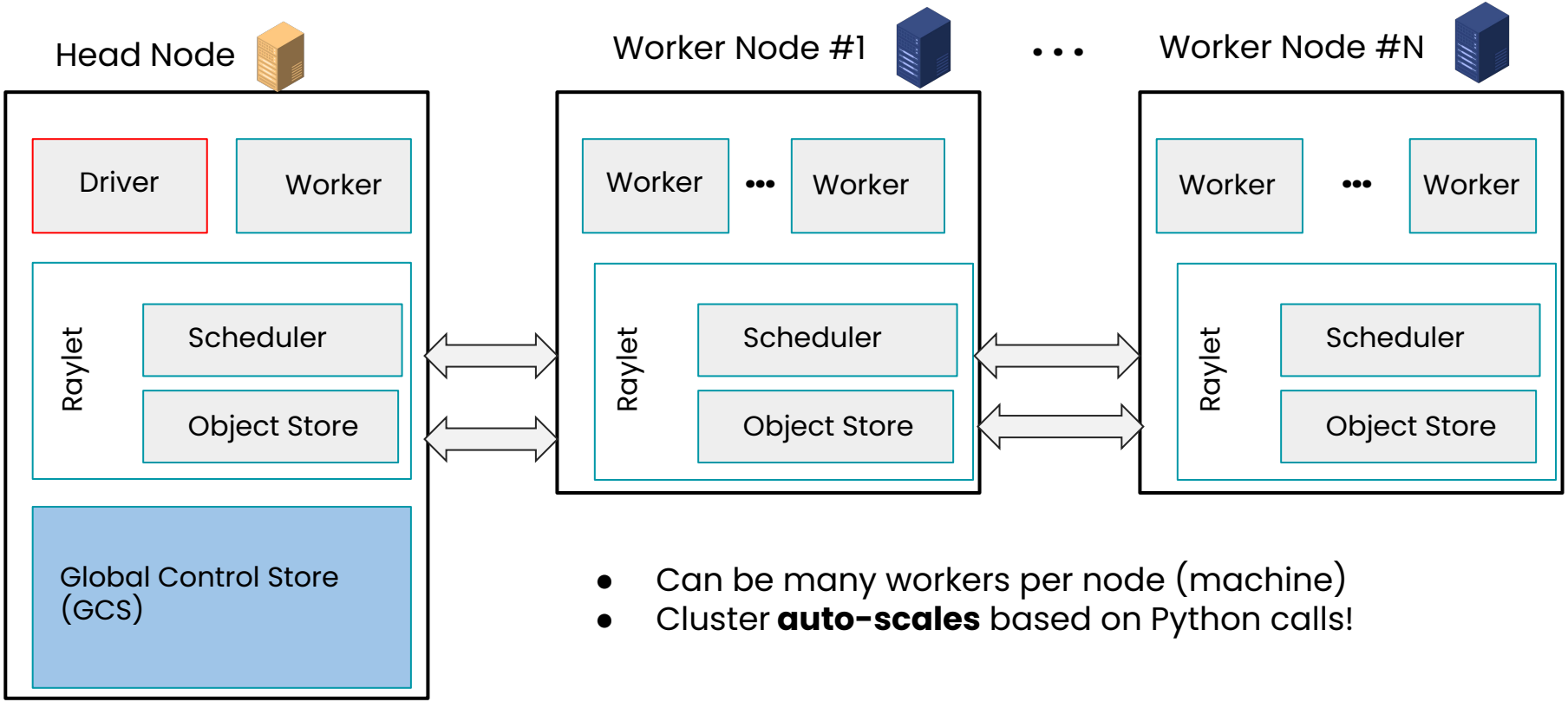


Only use the libraries you need!

Some companies scaling ML with Ray

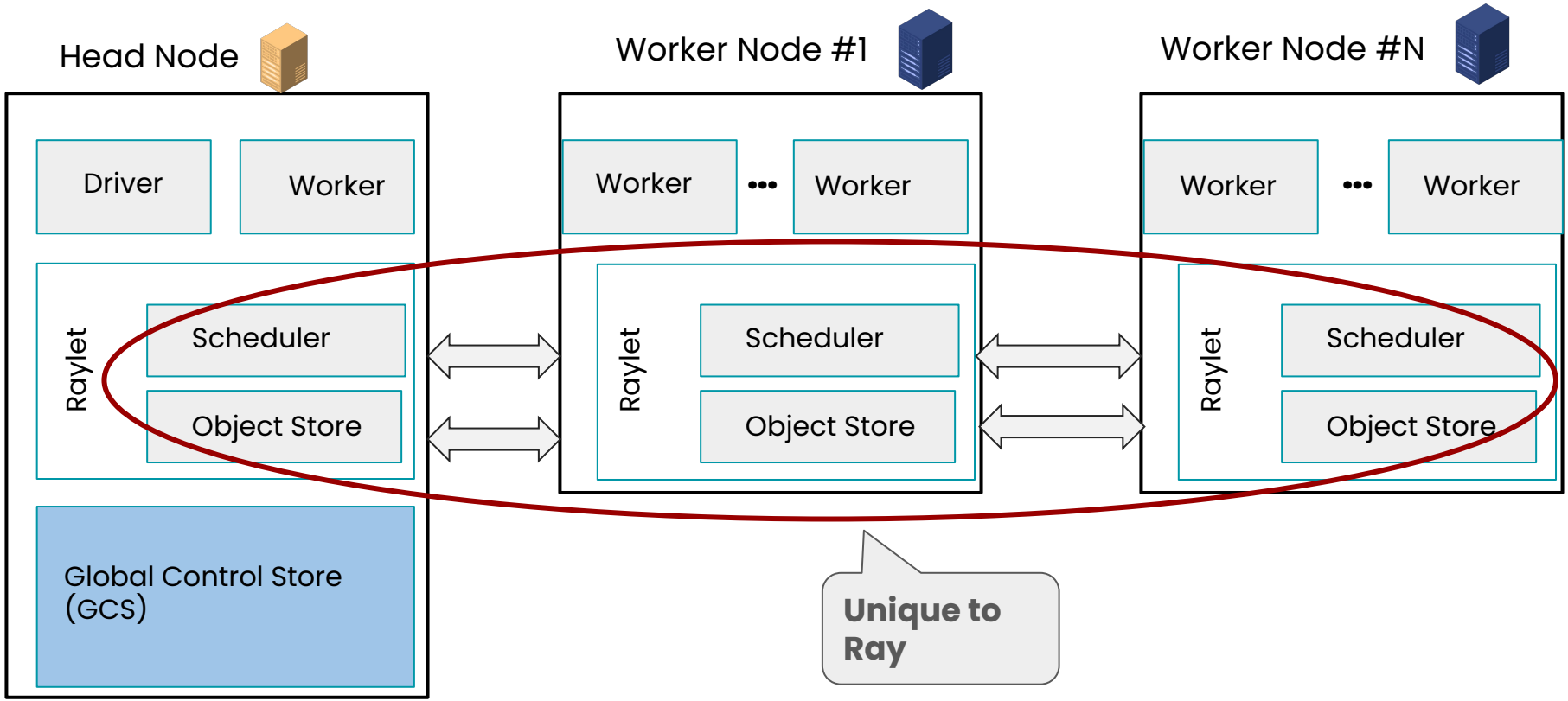
Ray Architecture & Components

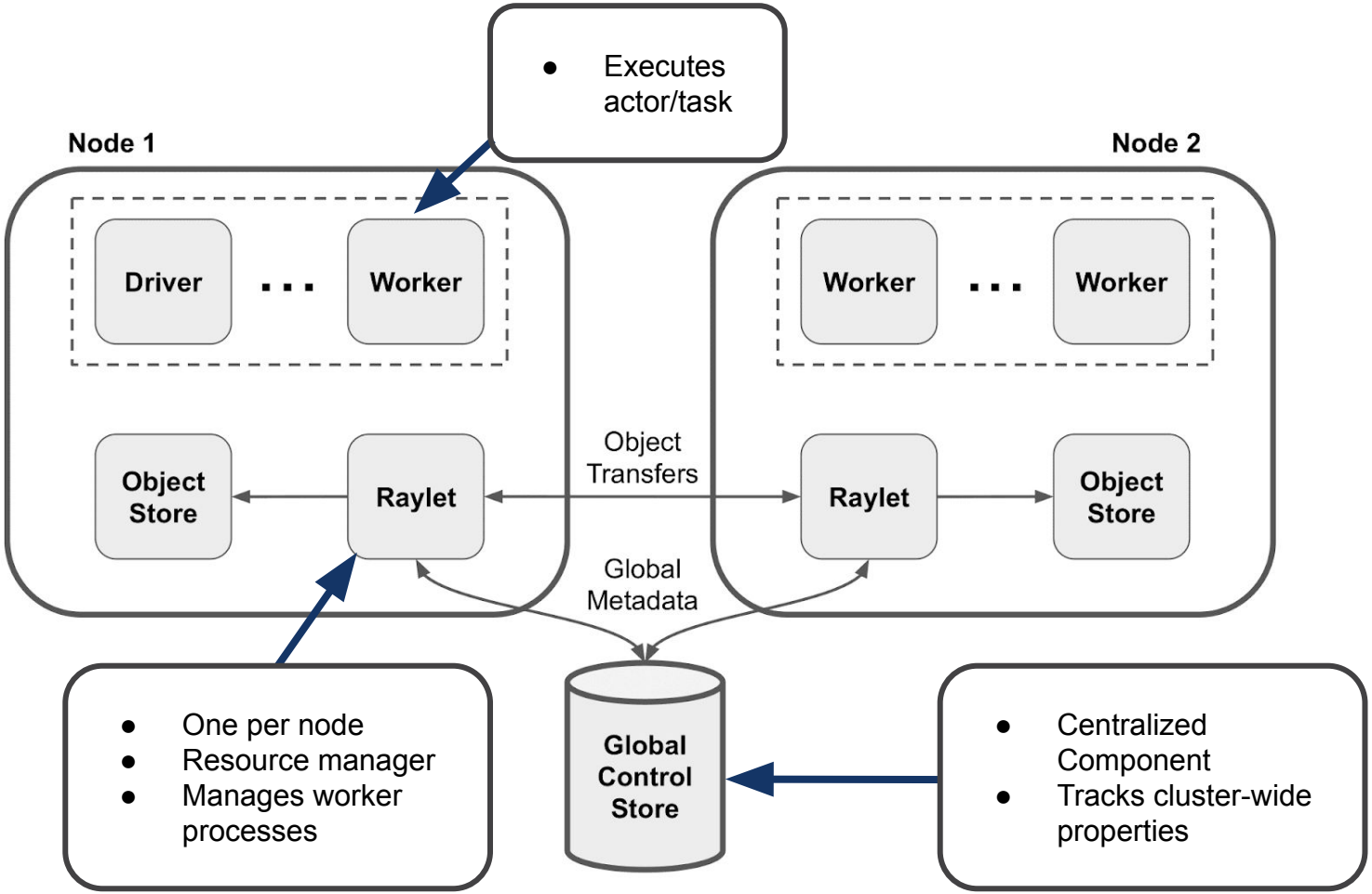
Anatomy of a Ray cluster



- Can be many workers per node (machine)
- Cluster **auto-scales** based on Python calls!

Anatomy of a Ray cluster

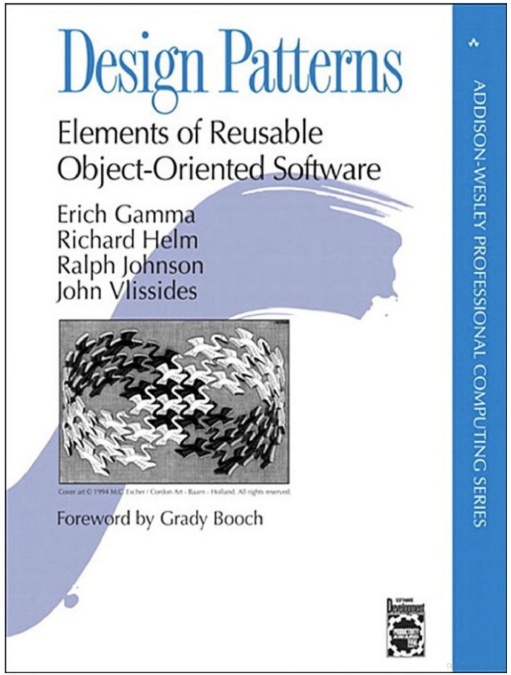



Executes actor/task
Node 1
Node 2
Driver
Worker
Worker
Worker
Object Store
Raylet
Object Transfers
Raylet
Object Store
Global Metadata
One per node
Resource manager
Manages worker processes
Global Control Store
Centralized Component
Tracks cluster-wide properties

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 of
- 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

```
@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()
```

anyscale

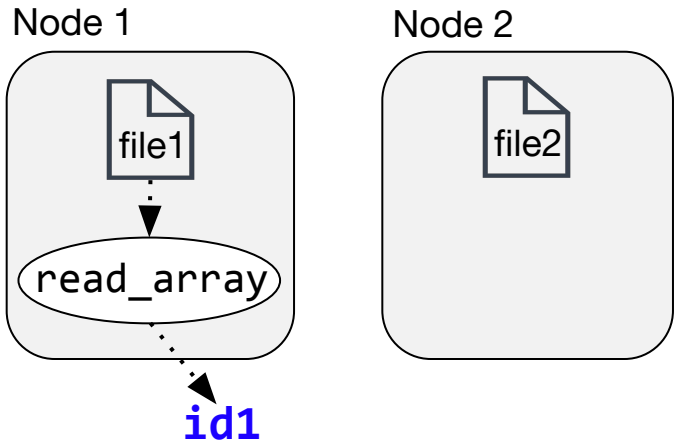
Task API

Blue variables are ObjectRef 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)
```



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

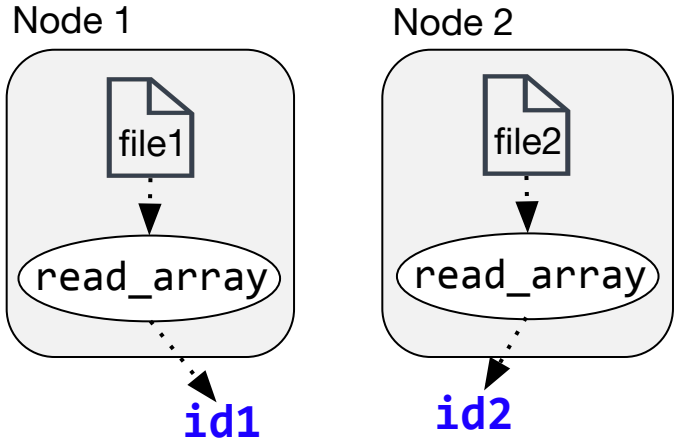
Task API

```
@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)
```

Blue variables are Object IDs (similar to futures)



Dynamic task graph: build at runtime

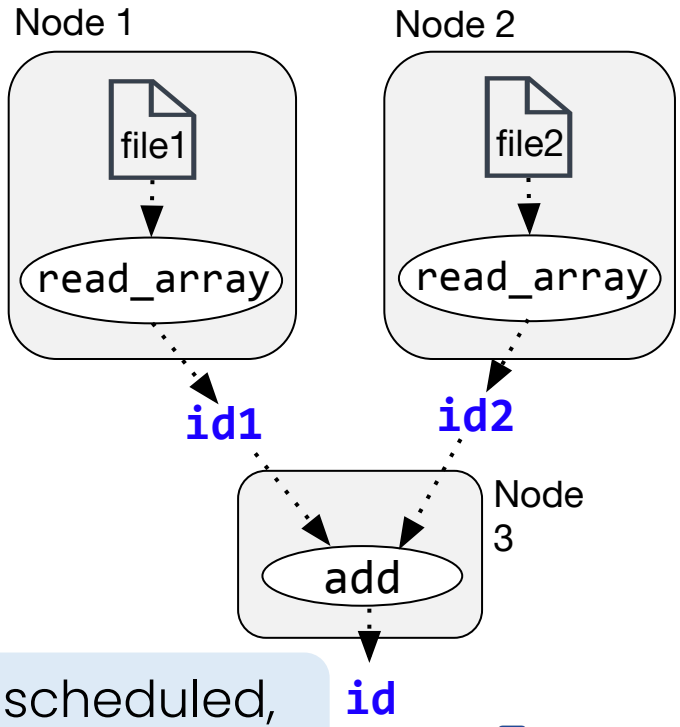
Task API

```
@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)
```

Blue variables are Object IDs (similar to futures)



Every task scheduled, but not finished yet

anyscale

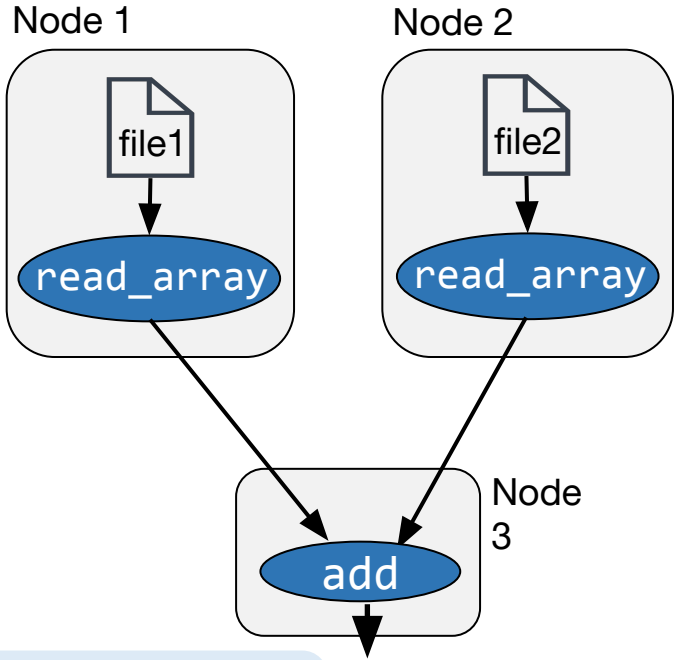
Task API

```
@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)
```

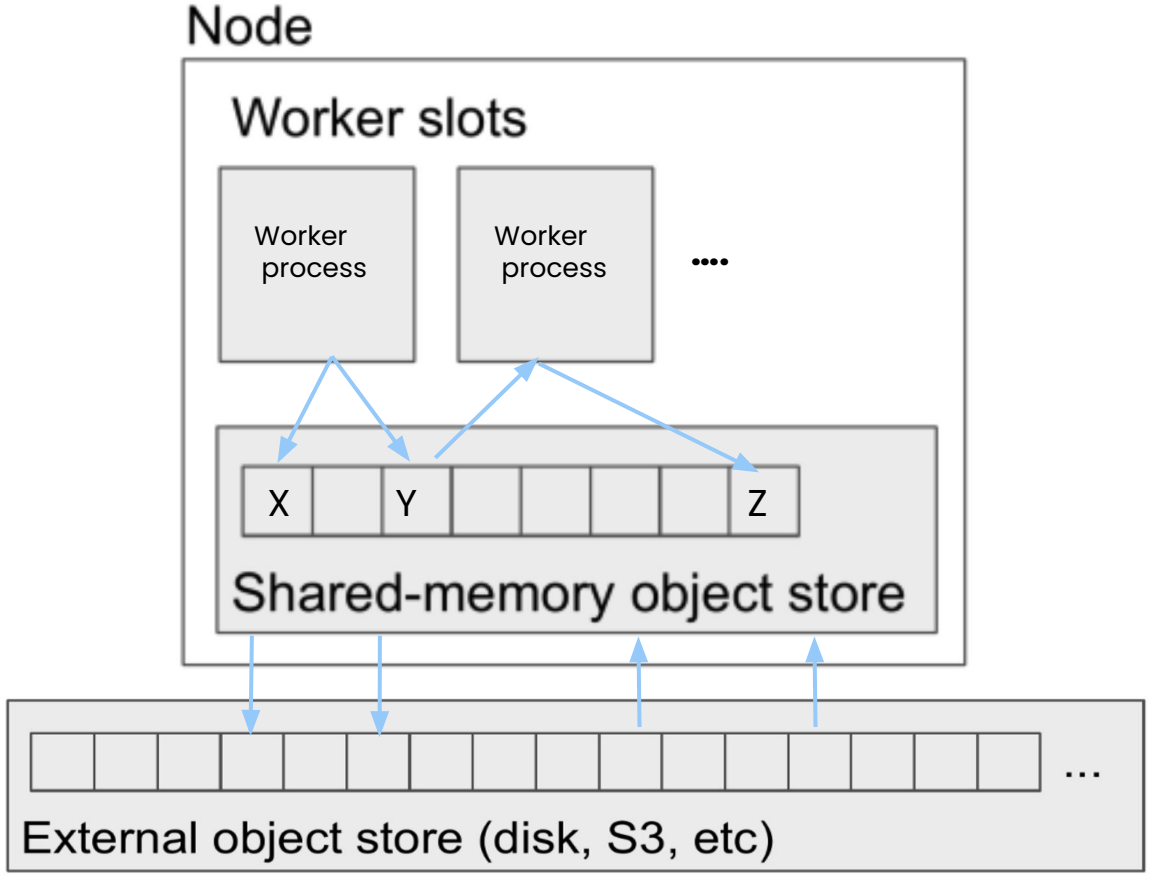
Blue variables are Object IDs (similar to futures)



Task graph executed to compute `sum`

anyscale

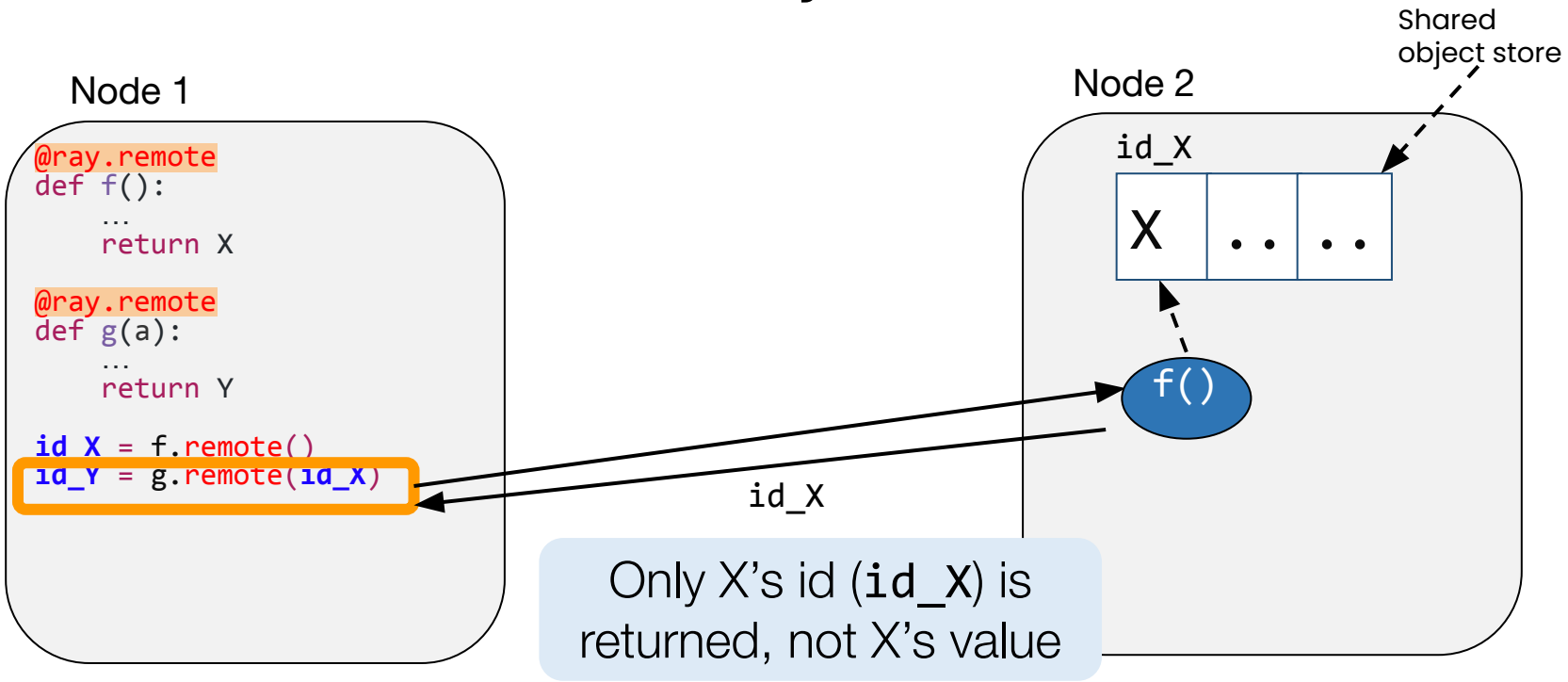
Distributed Immutable object store



Spill over to external storage

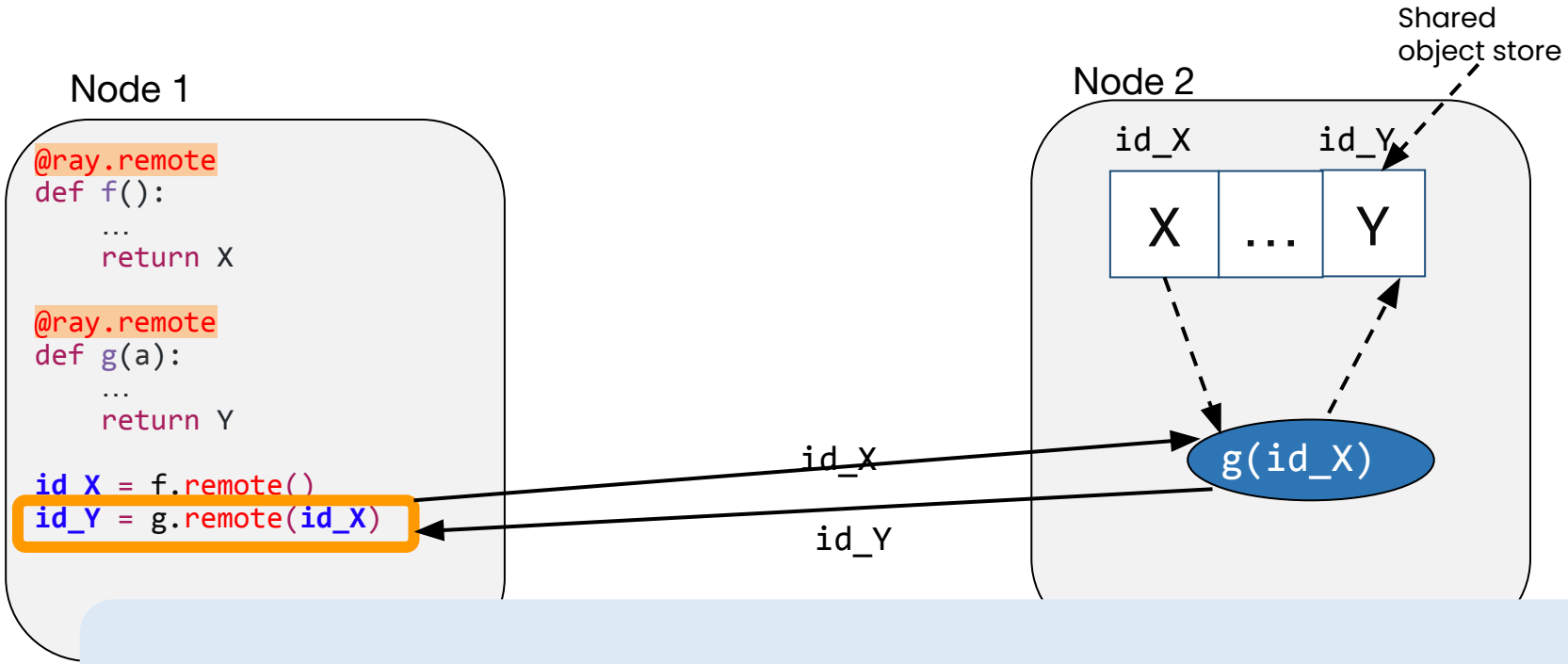
anyscale

Distributed Immutable object store



anyscale

Distributed object store



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

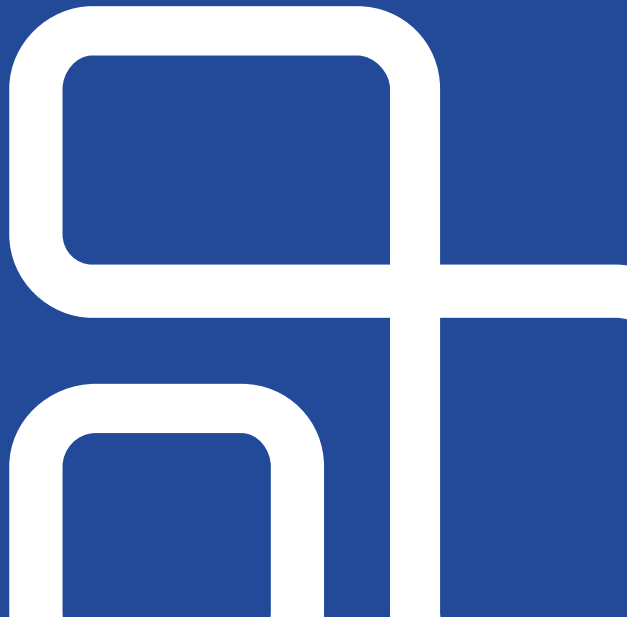
Module -1 Tutorial

https://bit.ly/ray-serve-tutorial-mlops

https://docs.ray.io/en/latest/

anyscale

Introduction to Ray Serve



What is Ray Serve?

☑️Scalable and Programmable Serving Framework on Ray

☑️Framework Agnostic, Python First, and Easy to Use

☑️Helps you **Scale** in Production

Ray Serve as a Web Framework

```python
from ray import serve
serve.start()

@serve.deployment
def hello(request):
    name = request.query_params["name"]
    return f"Hello {name}!"

hello.deploy()
```

Simple to Deploy Web Services on Ray

```python
import requests
response = requests.get(
  "http://127.0.0.1:8000/hello?name=serve"
)
assert response.text == "Hello serve!"
```

Ray Serve for Model Serving

```python
@serve.deployment(
    num_replicas=10,
    ray_actor_options={"num_gpus": 1}
)
class MyModel:
    def __init__(self):
        self.model = load_model_from_registry()

    @serve.batch
    async def __call__(self, requests_batch):
        return self.model.compute(requests_batch)
```

Specialized for ML Model Serving

GPUs
Batching
Scale-out
Model Composition

Building ML Service

Ease of development

Ready for Production

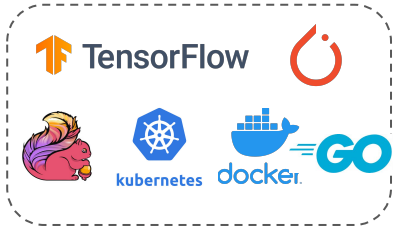
Web Frameworks

Can't achieve
- high performance
- low cost

Pythonic API
Native FastAPI

Ray Serve

Custom Tooling



Hard to
- develop
- deploy
- manage

High Performance
Scalability

Specialized Systems

Lost
- flexibility
- ease of use

What makes Serve Different?

Many Tools Run 1 Model Well

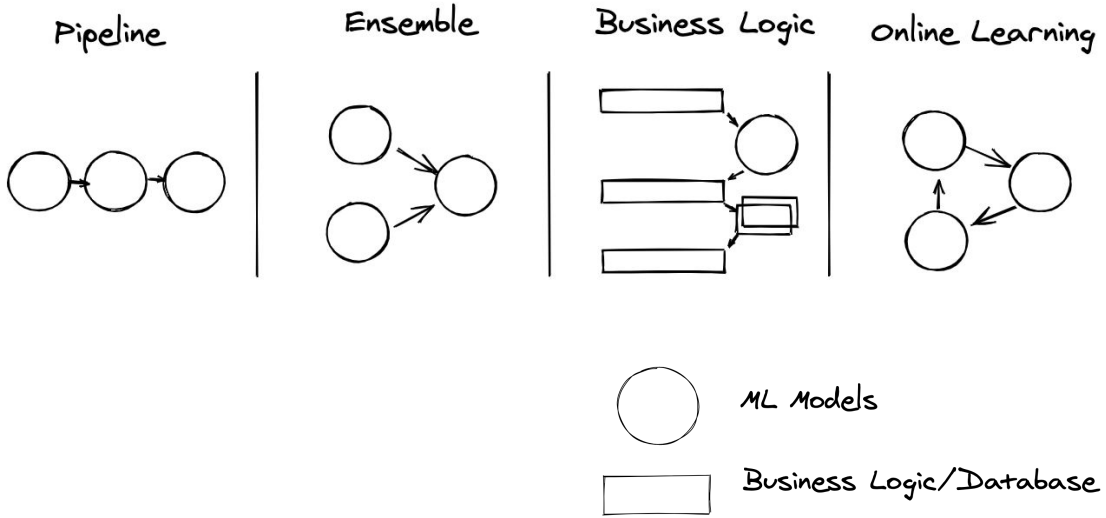
With 1+ copies of the model
-> Impossible?
-> Complex YAML
-> Scalability issue
-> $$$

Reality:

- New models are developed over time
- Scale out a single model
- Compose multiple models together for real world use cases

Patterns of ML Models in Production

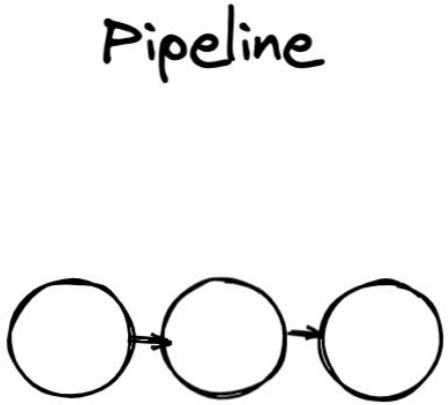
- Pipeline
- Ensemble
- Business Logic
- Online Learning



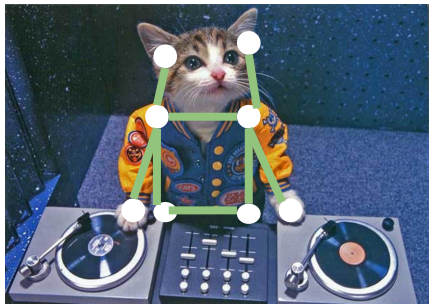
anyscale

Patterns

- Pipeline
- Ensemble
- Business Logic
- Online Learning



A Typical Computer Vision Pipeline



Standing Cat

Image Decoding
Augmentation
Clipping

→ Detection
Classifier

→ Keypoint
Detection

→ NLP
Synthesis

Pipeline Implementation

Wrap models in web server

```python
@app.route("/predict")
def prediction_handler(raw_input):
    models = load_models()
    output = raw_input
    for model in models:
        output = model(output)
    return output
```

Simple but not performant

Many specialized microservices

Complex and hard to operate

anyscale

Pipeline in Ray Serve

```python
@serve.deployment
class Featurizer: ...

@serve.deployment
class Predictor: ...

@serve.deployment
class Orchestrator:
    def __init__(self):
        self.featurizer = Featurizer.get_handle()
        self.predictor = Predictor.get_handle()

    async def __call__(self, inp):
        feat = await self.featurizer.remote(inp)
        predicted = await self.predictor.remote(feat)
        return predicted

Featurizer.deploy()
Predictor.deploy()
Orchestrator.deploy()
```

Deployments call other deployments with handles.

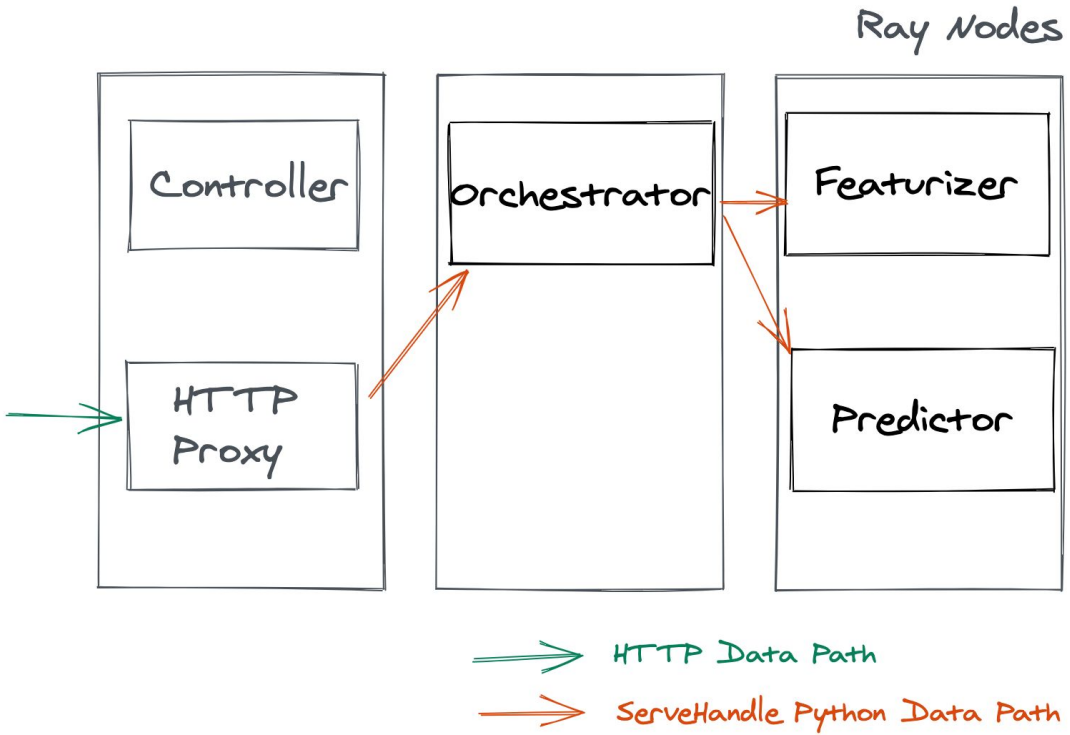
Ray Serve: Architecture

```python
@serve.deployment
class Featurizer: ...

@serve.deployment
class Predictor: ...

@serve.deployment
class Orchestrator:
  def __init__(self):
    self.featurizer = Featurizer.get_handle()
    self.predictor = Predictor.get_handle()
  async def __call__(self, inp):
    feat = (
      await self.featurizer.remote(inp))
    predicted = (
      await self.predictor.remote(feat))
    return predicted

Featurizer.deploy()
Predictor.deploy()
Orchestrator.deploy()
```



Ray Serve Enables Seamless Model Composition

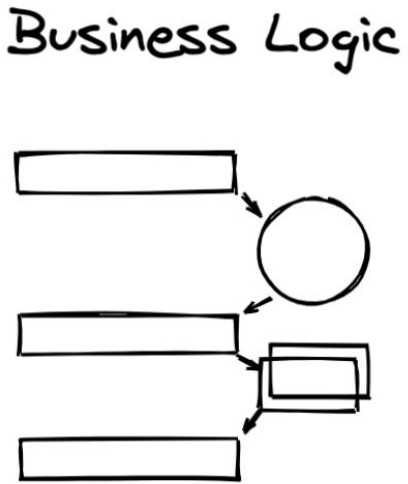
Pythonic API

High Performance Calls (No HTTP)

1 line to scale to 100 machines (`num_replicas=100`)

Patterns

- Pipeline
- Ensemble
- Business Logic
- Online Learning



Business Logic

- Database Lookup
- Web API Calls
- Feature Store Lookup
- Feature Transformations

Business Logic: Upgrading to Ray Serve

```
1 - @app.route("/predict")
2 + @serve.deployment(route_prefix="/predict")
3 def prediction_handler(raw_input):
4 -    model = load_model()
5 +    model = ModelDeployement.get_handle()
6      inputs = [
7          db.validate(raw_input),
8          feature_store.retrieve(raw_input)
9      ]
10 -   output = model(inputs)
11 +   output = ray.get(model.remote(inputs))
12     return output
```

Just Python

Just Python

Network Bound

I/O Heavy

Compute-bound, memory hungry → Offloaded to another Deployment

anyscale

Ray Serve Enables Arbitrary Business Logic

Separate I/O and Compute Heavy Work

Native FastAPI Ingress

Scale-out Web Serving to Replicas

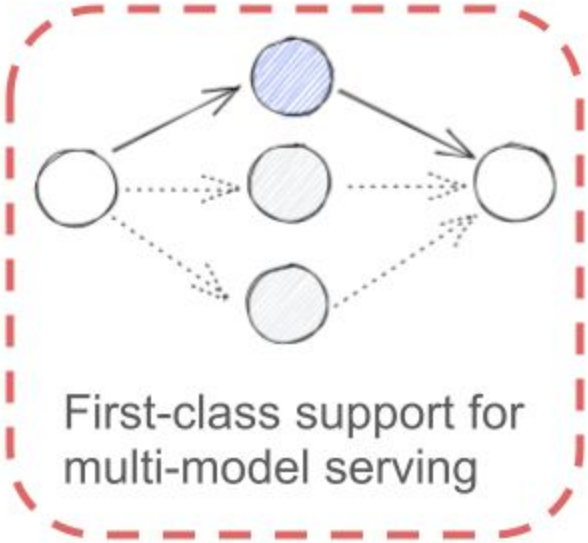
Ray Serve TL;DR

Flexible, scalable compute for model serving

Ray Serve

RAY

1. Scalable
2. Low latency
3. Efficient



First-class support for multi-model serving

python™

Python-native: mix business logic & ML

Module -2 Tutorial

https://bit.ly/ray-serve-tutorial-mlops

https://docs.ray.io/en/latest/

Takeaways

- Distributed computing is a necessity & norm
- Ray's vision: make distributed programming simple
  - Don't have to be distributed systems expert. Just use @ray.remote :)
- Ray Serve is simple and scalable. Just use @serve.deployment :)

RAY Libraries Documentation Learn Ecosystem Community 20,541

Get involved and stay informed

Live and on-demand. Virtual and in-person. Ask questions and share learnings. File bugs and submit code. Whatever your preference, there are lots of ways to connect with the global Ray community, get involved with the project, and stay informed on the latest and greatest.

ray.io/community

GitHub

Follow the project, track issues, file bugs and feature requests, or contribute code.

Follow the project →

Discussion forum

Join the forum to get technical help and share best practices and tips with the Ray community.

Join the forum →

Slack

Connect with other users and project maintainers on the official Ray Slack channel.

Chat with other users →

Newsletter

Subscribe to the monthly Ray newsletter to get curated updates delivered to your inbox.

Sign up for updates →

Twitter

Follow @raydistributed on Twitter to stay informed on the latest news and updates.

Follow us on Twitter →

Ray Summit 2022

2 Day Ray Experience | August 23-24 | San Francisco

Industry Luminaries

- Anyscale
- Uber
- Riot Games
- Meta AI
- PyTorch
- Netflix
- Microsoft
- Open.AI

Sessions

- Ray Trainings
- Ray Deep Dives
- Ray Use Case Talks
- AI/ML Platform & Applications Talks
- Lightning Talks
- Ray Partner Talks

Experience

- Ray Community Networking
- Ray Partner Expo
- RAYdiate Bar
- Ray Community Celebration

Register: https://www.anyscale.com/ray-summit-2022

Exclusive Ray community conference pass discount code: : **RayMeetUp25**

Thank you!

Let's stay in touch:

jules@anyscale.com  @2twitme
https://www.linkedin.com/in/dmatrix/

archit@anyscale.com  @ArchitKulkarni7
https://www.linkedin.com/in/archit-kulkarni-a013