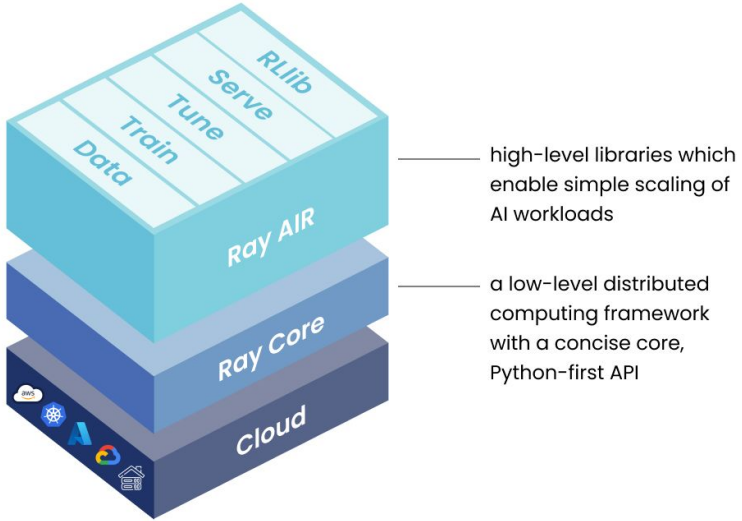
# All About Ray

– @kerthcet

***Ray** is an open-source **unified framework** for scaling AI and Python applications. It provides the compute layer for **parallel processing** so that you don't need to be a distributed systems expert.*



# Ray Framework

- **Ray Core:** An open-source, Python, general purpose, **distributed computing library** that enables ML engineers and Python developers to scale Python applications and accelerate machine learning workloads.
- **Ray AI Runtime:** An open-source, Python, domain-specific set of libraries that equip ML engineers, data scientists, and researchers with a scalable and unified **toolkit** for ML applications
- **Ray Cluster:** A set of worker nodes connected to a common Ray head node. Ray clusters can seamlessly **scale workloads** from a laptop to a large cluster for production

# Use Cases With Ray

- Batch Inference
- Model Training
  - Parallel training
  - Distributed training of large models
- Model Serving
- Hyperparameter Tuning
  - Number of hidden layers
  - Number of nodes/neurons per layer
  - Learning rate
  - Momentum
- Reinforcement Learning
- ML platform

# Ray Core

# Application Concepts

- **Task**: Represent the execution of a remote function on a stateless worker
- **Actor**: Represent a stateful computation, defined by a python class.
- **Objects**: An application value returned by a task. Objects are immutable.

```python
# Define the square task.
@ray.remote
def square(x):
    return x * x

# Launch four parallel square tasks.
futures = [square.remote(i) for i in range(4)]

# Retrieve results.
print(ray.get(futures))
# -> [0, 1, 4, 9]
```

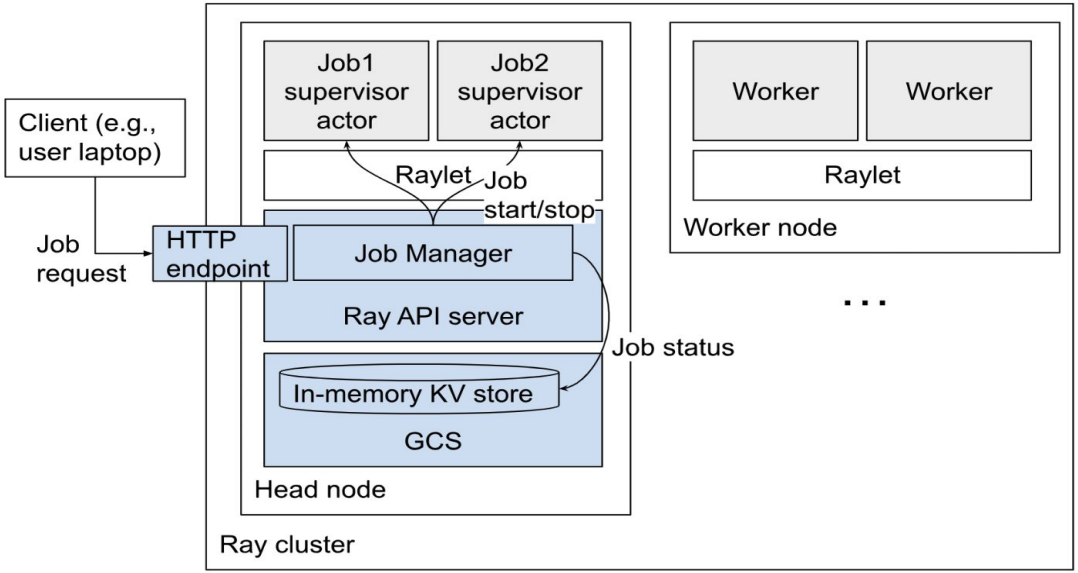
```python
# Define the Counter actor.
@ray.remote
class Counter:
    def __init__(self):
        self.i = 0

    def get(self):
        return self.i

    def incr(self, value):
        self.i += value

# Create a Counter actor.
c = Counter.remote()

# Submit calls to the actor. These calls run
asynchronously but in
# submission order on the remote actor
process.
for _ in range(10):
    c.incr.remote(1)

# Retrieve final actor state.
print(ray.get(c.get.remote()))
# -> 10
```

# Actors, Tasks and Workers

- Each Ray worker is a python process
  - Used to execute multiple Ray tasks
  - Started as a dedicated Ray actor
- Tasks: When Ray starts, a number of workers will be started automatically. They will be used to execute tasks, like a process pool.
- Actors: A Ray actor is also a Ray worker, but is instantiated at runtime. All of its methods will run on the same process, using the resources designed by the Actor. The python processes can't be reused, they will be terminated when the Actor is deleted.

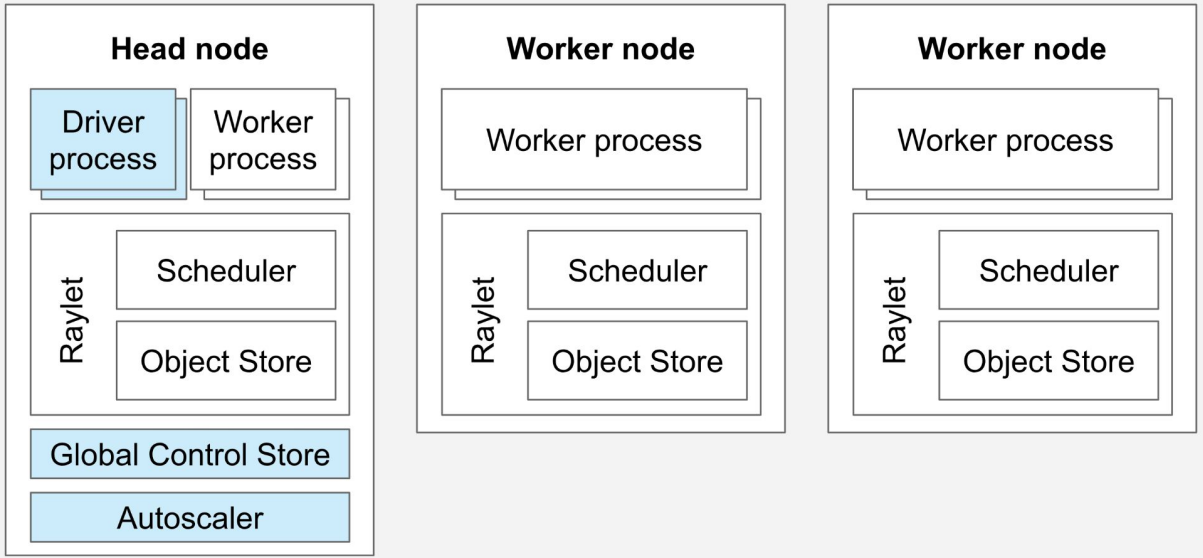
# Ray Jobs

- **Job**: A collection of Ray tasks, objects, and actors that originate from the same script.




*A diagram of the architecture of Ray Job Submission. Blue boxes indicate singleton services that manage job submission, among other cluster-level operations.*

# A Ray Cluster




Consist of:

- A single Head node
- A collection of Worker nodes

1. *The head node runs additional control processes (highlighted in blue).*
2. *Driver process is the program root, or the "main" program, it runs Ray jobs. Worker process runs user code in tasks and actors.*

# Components

- **Driver process** is the program root, or the “main” program, it runs Ray jobs.
- **Worker process** runs user code in tasks and actors.
- **Raylet** manages shared resources on each node.
  - **Scheduler** is responsible for resource management, task placement, and fulfilling task arguments that are stored in the distributed object store
  - **Object Store** is a shared memory object store, responsible for storing, transferring and spilling large objects.
- **GCS** manages cluster-level metadata, such as nodes and actors.
- **Autoscaler**

# Scheduler - scheduling policies

- **Default Hybrid Policy**:  Pack tasks onto the local node until the node's critical resource utilization exceeds a configured threshold (50% by default)
- **Spread Policy**: This policy spreads tasks among nodes with available resources using round-robin
- **Node Affinity Policy**: A user can explicitly specify a target node where the task or actor should run.
- **Data Locality Policy**: Ray supports data locality by having each task caller choose a preferred raylet based on the caller's local information about task argument locations
- **Placement Group Policy**: Reserve groups of resources across multiple nodes

# Global Control Service

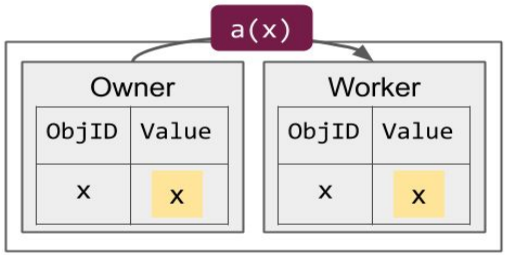
- **Node Management**: Manage addition and deletion of nodes to the cluster. It also broadcasts this information to all raylets so the raylets will be aware of the node changes.
- **Resource Management**: Broadcast the resource availability of each raylet to the whole cluster to make sure each raylet's view of the resource usage is updated.
- **Actor management**: Manage actor creation and deletion requests. It also monitors the actor liveness and triggers recreation (if configured) in case of an actor failure.
- **Placement group management**: Coordinate placement group creation and deletion in the Ray cluster.
- **Metadata store**: Provide a key-value store which can be accessed by any worker. Note that this is meant for small metadata only and is not meant to be a scalable storage system.
- **Worker manager**: Handle worker failure reported by raylet
- **Runtime env**: GCS is the place managing the runtime env package, including counting the number of usage of the plackages and the garbage collection.

# Autoscaler

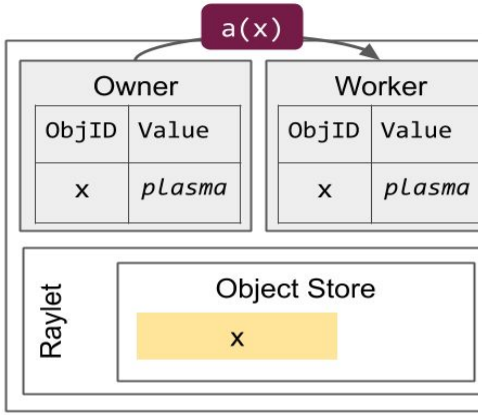
- A process that runs on the head node (or as a sidecar container in the head pod if using Kubernetes)
- Scale up/down worker nodes based on the resource demands, not the metrics or physical resource utilization, e.g. `ray.autoscaler.sdk.request_resources(num_cpus=4)`
- Constrain the number of replicas of an autoscaling workerGroup e.g. `max_workers[default_value=2, min_value=0], min_workers[default_value=0, min_value=0]`
- Upscaling and downscaling speed can also be controlled, e.g. `upscaling_speed[default_value=1.0, min_value=1.0], idle_timeout_minutes[default_value=5, min_value=0]`
- (Recommended) Start with non-autoscaling clusters if you're new to Ray.

# Object Management

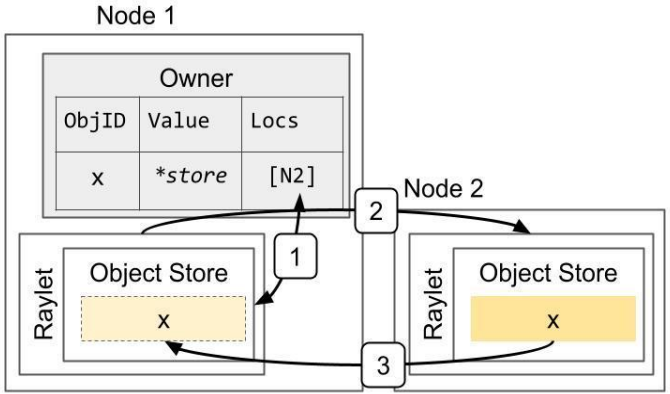
- Small Objects are stored in their owner's in-process store
- Large objects are stored in the distributed object store.
- A process to reference an object without having the object local




(a) Small objects are stored in the in-process store. They are copied directly to the worker through the task description.

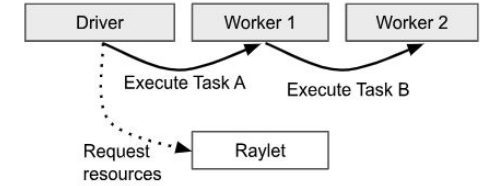



(b) Large objects are stored in the distributed object store. The worker retrieves the value through a protocol with the plasma store.

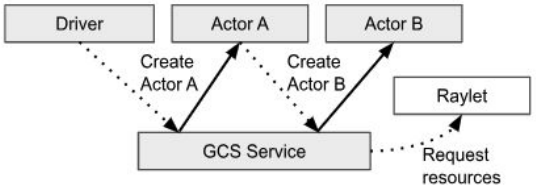


# Lifecycle Management

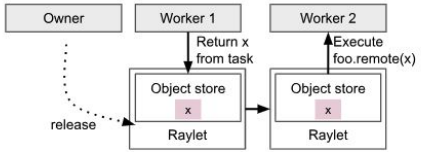
- Owner managed the lifecycle of a Task
- Owner managed the lifecycle of an Object
- GCS managed the lifecycle of an Actor




*The process that submits a task is considered to be the owner of the result and is responsible for acquiring resources from a raylet to execute the task. Here, the driver owns the result of `A`, and `Worker 1` owns the result of `B`.*



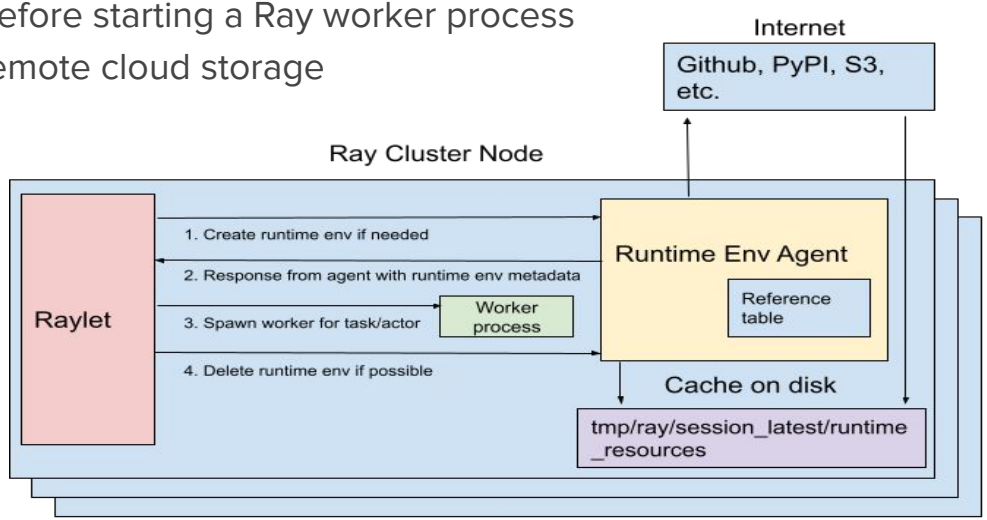

*Unlike task submission, which is fully decentralized and managed by the owner of the task, actor lifetimes are managed centrally by the GCS service.*




*Distributed memory management in Ray. Workers can create and get objects. The owner is responsible for determining when the object is safe to release.*

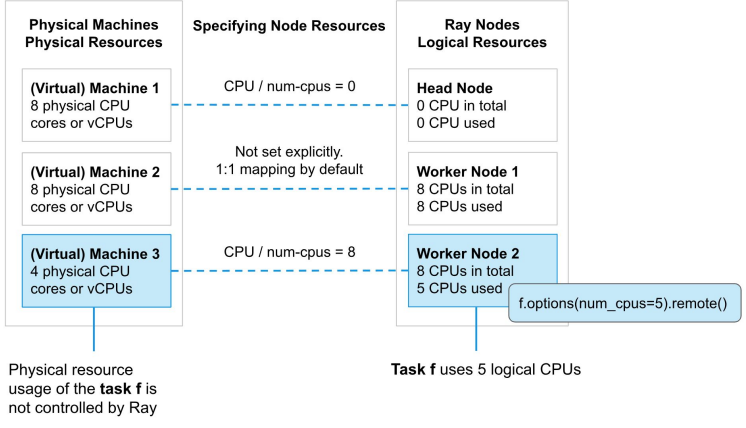
# Runtime Environment

- Create an environment includes:
  - Downloading and installing packages via `pip install`
  - Setting environment variables for a Ray worker process
  - Calling `conda activate` before starting a Ray worker process
  - Downloading files from remote cloud storage



# Resources

- Ray Resources are **logical** and don't need to have 1-to-1 mapping with physical resources (mainly for admission control and autoscaling)
- Specify node resources via **num_cpus**, **num_gpus ...** (set to physical quantities by default)
- Task or actor will only run on a Ray node with **sufficient resources**
- Support fractional resource, e.g. **num_cpus=0.5**



# GPU Support

- Support fractional GPUs, but it's the user's responsibility to make sure that the individual tasks don't use more than their share of the GPU memory
- Ray wil pack one GPU before moving on to the next one to avoid fragmentation
- Support specified accelerator types, e.g. @ray.remote(num_gpus=1, accelerator_type=NVIDIA_TESLA_V100)
- Ray will automatically set the CUDA_VISIBLE_DEVICES environment variable which most ML frameworks will respect for purposes of GPU assignment

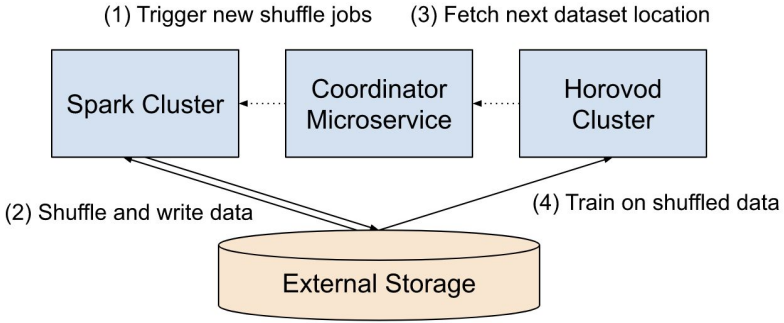
# In a nutshell

- Requirements
  - Fine-gained, heterogeneous computations, e.g. duration, hardware
  - Flexible computation model, both stateful and stateless computations
  - Dynamic execution, as the order in which computations finish is not always known in advance.
- Programming Model (Non-Blocking)
  - Task: Stateless, idempotence, parallelism
  - Actor: Stateful, serially by default
  - Future: The result of a remote function. Futures can be retrieved by ray.get() and can be passed as arguments to other remote functions without waiting for the result.
- Computation Model
  - Ray employs a dynamic task graph computation model, in which the execution of both remote functions and actor methods is automatically triggered by the system when their inputs become available

# Ray Air

# RayData - Changes with Big Data Training

- Distributed preprocessing
- Distributed shuffling
- Pipelining data processing and training

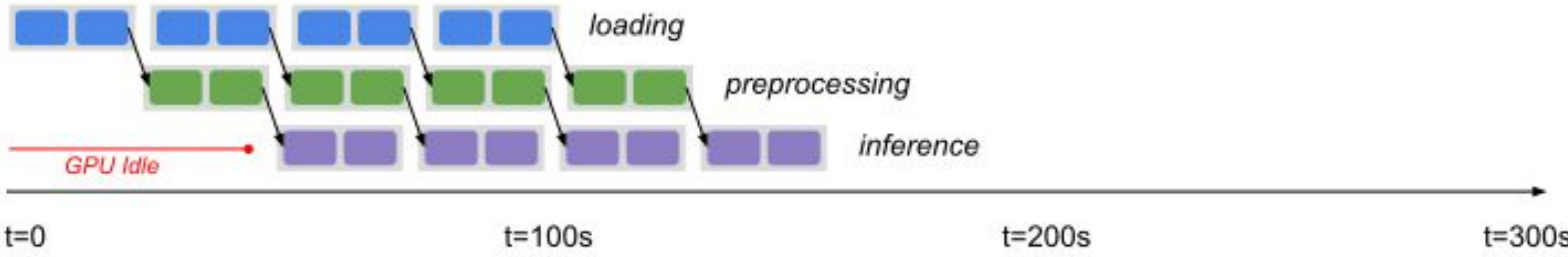



Lack of programmability, you need to manage 3+ systems.
Performance overhead, encode/decode data for multi times.

# RayData - How Ray Does

**Integration:**


**Pipeline:**

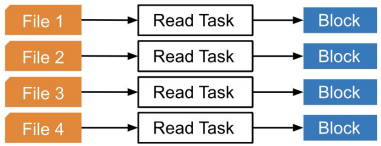


# Ray Data - How it looks like

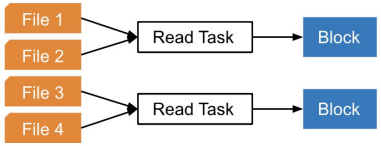
- See Jupyter Notebook

# Ray Data - Operations

- Reading Files
  - Ray Data uses Ray tasks to read files in parallel. Each read task reads one or more files and produces an output block:

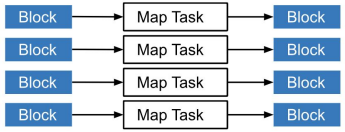



Using the default cluster parallelism:
`ray.data.read_csv(path)`

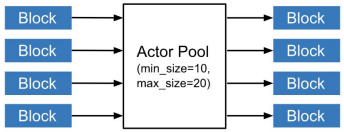



Manually specifying parallelism:
`ray.data.read_csv(path, parallelism=2)`

- Transforming data
  - Ray Data uses either Ray tasks or Ray actors to transform blocks. By default, it uses tasks.



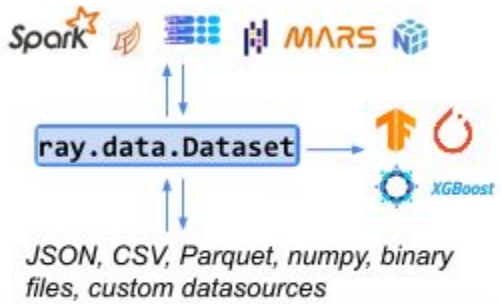

```
ds.map_batches(stateless_fn)
```



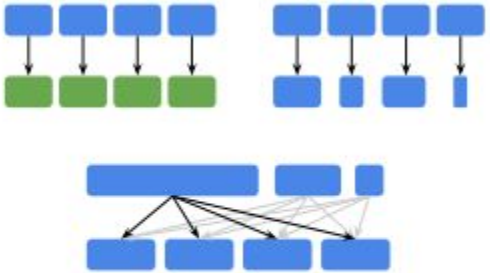

```
ds.map_batches(callable_cls,
   compute=ActorPoolStrategy(
      min_size=10, max_size=20,
   ))
```

# RayData - Why

- Faster and cheaper for modern deep learning applications
- Cloud, framework and data format agnostic
- Out of the box scaling
- Python first
- Suit for offline batch inference, data preprocessing and ingest for Training




Standard way to load and exchange data in Ray



Basic distributed ops: map, filter, and repartition

```
ds = ray.data.read_parquet(...)
# Pass datasets to tasks and actors
func.remote(ds)
# Split datasets into shards
shards = ds.split(xgboost.num_workers,
    locality_hints=xgboost.workers)
# Distributed training on datasets
for i, s in enumerate(shards):
   xgboost.workers[i].train.remote(s)
```

Seamless interop with Ray tasks, actors, and libraries

# Ray Train

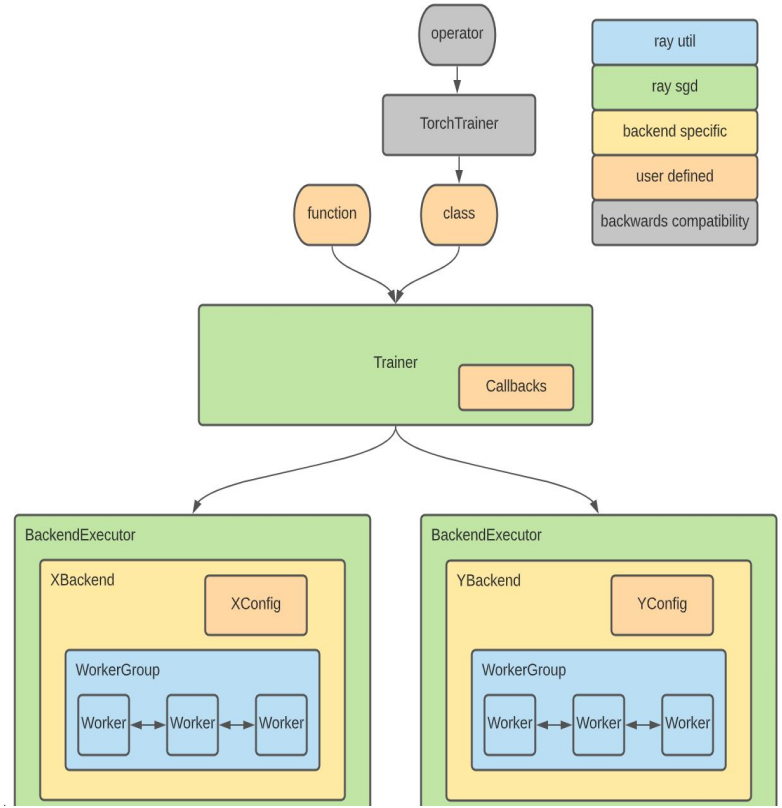
- Ray Train scales model training for popular ML frameworks such as Torch, XGBoost, TensorFlow, and more.
- Features
  - Framework support:
    - Deep learning trainers, pytorch, tensorflow, horovod
    - Tree-based trainers, XGBoost, LightGBM
    - Other ML frameworks, HuggingFace, RLlib
  - Build for ML practitioners:
    - Callbacks for early stopping
    - Checkpointing
    - Integration with TensorBoard, Weights and MLflow
    - Jupyter notebooks
  - Batteries included:
    - Work smoothly with Ray Data, Ray Tune, Autoscaling

# Ray Train - How it looks like

- See Jupyter Notebook

# Ray Train - Architecture

- **Train**: manage training process
  - create an Executor to run the distributed training
  - handle callbacks based on the results from the executor
- **Executor**: executes distributed training
  - Handles the creation of a group of workers
- **Backend**: manages framework-specific communication protocols
- **Workgroup**: manages a group of Ray Actors

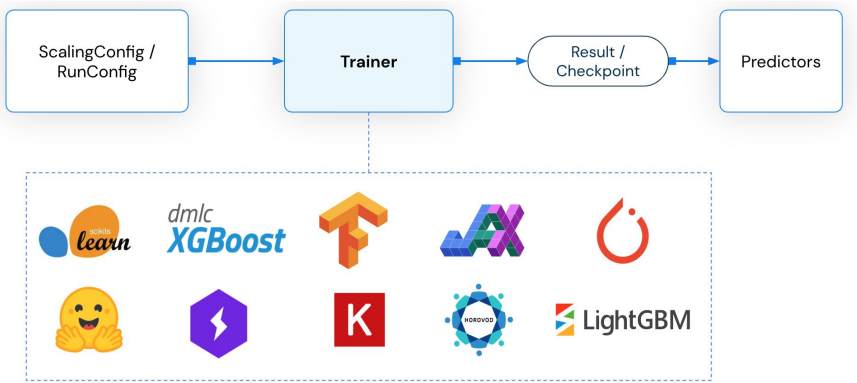


# Ray Train - Concepts

- **Trainers**: execute distributed training
- **Configuration**: used to config training
- **Checkpoints**: returned as the the result of training
- **Predictors**: used for inference and batch prediction

# Ray Train - Distributed Training

- Using DL frameworks as backends
  - PyTorch with DistributedDataParallel
  - TensorFlow with MultiWorkerMirroredStrategy
  - Horovod with DistributedOptimizer
- How fast is Ray Train compared to PyTorch, TensorFlow, etc.?
  - At its core, training speed should be the same



# Ray Train - Configuration

- ScalingConfig

```python
from ray.air import ScalingConfig

scaling_config = ScalingConfig(
    # Number of distributed workers.
    num_workers=2,
    # Turn on/off GPU.
    use_gpu=True,
    # Specify resources used for trainer.
    trainer_resources={"GPU": 1},
    # Try to schedule workers on different nodes.
    placement_strategy="SPREAD",
)
```

- RunConfig
  - FailureConfig
  - CheckpointConfig
  - SyncConfig

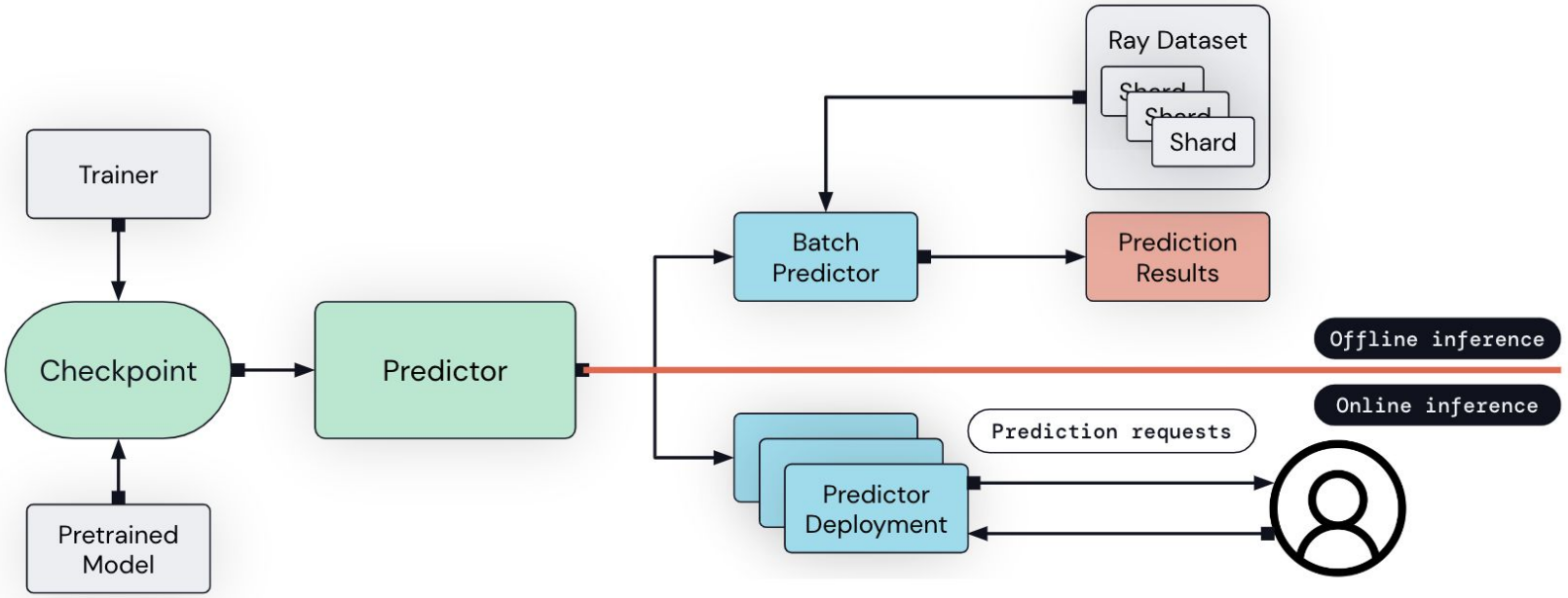
```python
run_config = RunConfig(
    # Name of the training run (directory name).
    name="my_train_run",
    # The experiment results will be saved to:
storage_path/name
    storage_path="~/ray_results",
    # storage_path="s3://my_bucket/tune_results",
    # Low training verbosity.
    verbose=1,
    # Custom and built-in callbacks
    callbacks=[WandbLoggerCallback()],
    # Stopping criteria
    stop={"training_iteration": 10},
)
```

# Ray Train - Checkpoint

- Calling Trainer.fit() returns a Result object, which includes information about the run such as the reported metrics and the saved checkpoints.
- Purposes for checkpoint
  - They can be passed to a Trainer to resume training from the given model state
  - They can be used to create a Predictor / BatchPredictor for scalable batch prediction.
  - They can be deployed with Ray Serve.

Examples see jupyter notebook.

# Ray Train - Predictor




Examples see jupyter notebook.

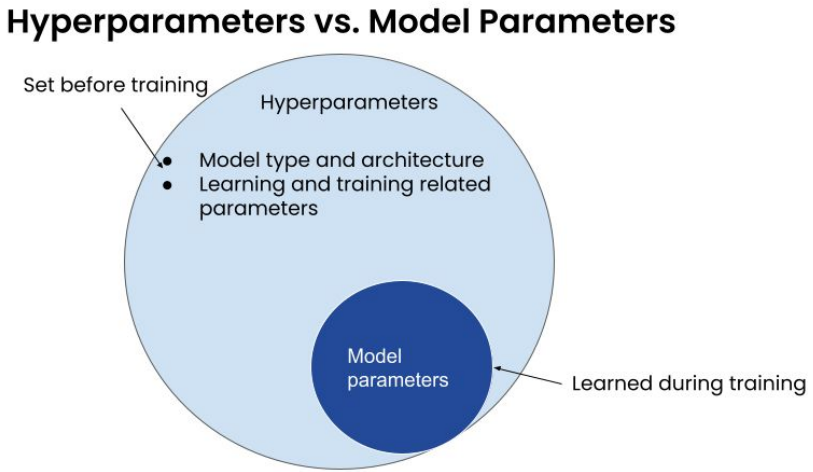
# Ray Train - Custom Predictor

- Implement your own Predictor:
  - An unsupported framework
  - Build-in predictors are not flexible enough
- To implement a custom predictor, subclass Predictor and implement:
  - __init__()
  - _predict_numpy() or _predict_pandas()
  - from_checkpoint()

In prediction, we don't necessarily care why something happens or how each variable effects each other. Let's look at an example, In inference, however, we care how the predictor variables affect the response, and may have to delve in to why that is.

# Ray Tune

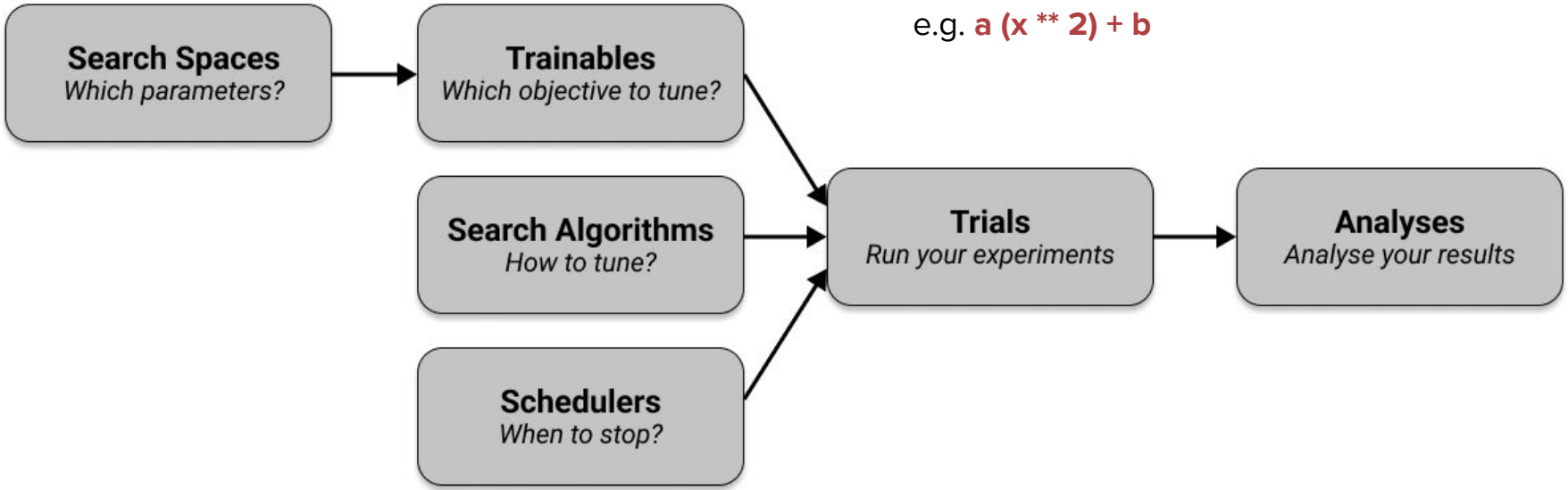
- Tune is a Python library for experiment execution and hyperparameter tuning at any scale



# Ray Tune: How it looks like

- See Jupyter Notebook

# Ray Tune: Architecture




e.g. **a (x ** 2) + b**

# Ray Tune: Why

- Cutting-edge Optimization Algorithms
  - Terminating bad runs early
  - Choosing better parameters to evaluate
  - Changing the hyperparameters during training to optimize schedules
- First-class developer productivity
  - A key problem with many hyperparameter optimization frameworks is the need to restructure your code to fit the framework.
- Multi-GPU & Distributed Training out-of-box support
- Seamlessly work with existing hyperparameter tools

# RayServe

- Ray Serve is a scalable model serving library for building online inference APIs
- Why RayServe
  - Building end-to-end ML-powered applications
    - You can combine multiple ML models, business logic and HTTP handlers.
  - Combine multiple models using a programmable API
  - Flexibly scale up and allocate resources
  - Avoid framework or vendor lock-in

# RayServe - How It Works

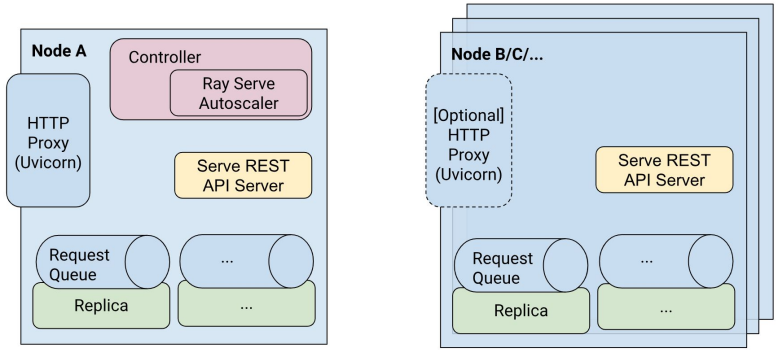
- See Jupyter Notebook

# RayServe - Concepts

- Deployment
  - The central concept in ray serve
  - It contains business logic or a ML model to handle incoming requests
  - Scale up to run across a Ray cluster
- Application
  - An application consists of one or more deployments
- ServeHandle
  - Composing deployments
- Ingress Deployment
  - The entrypoint for all the traffic to the application

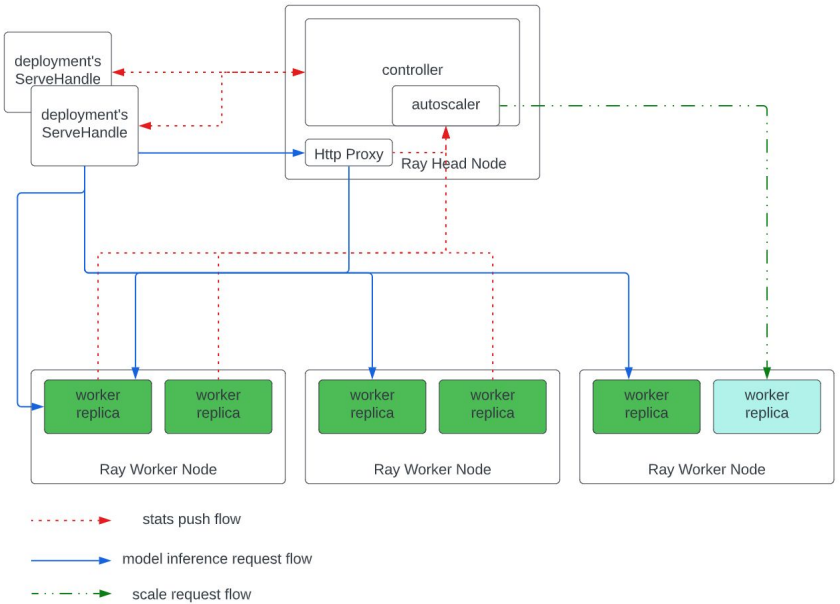
# RayServe - Architecture

- Made up by actors
  - Controller: unique to each Serve instance, responsible for CURD other actors
    - Serve API calls like creating or getting a deployment make remote calls to the Controller
  - HTTP Proxy: accept incoming requests, forward them to replicas and response
  - Replicas: Actors that actually execute the code in response to a request



# RayServe - Autoscaling

- Ray Serve's autoscaling feature automatically increases or decreases a deployment's number of replicas based on its load (ServeHandle queues vs in-flight queues on replicas).



# RayServe - Scaling & Resource Allocation

- Specify replicas by setting num_replicas
- Specify autoscaling by setting autoscaling_config = {"min_replicas": 1, "initial_replicas": 2, "max_replicas": 5}
- Specify resources by setting ray_actor_options={"num_gpus": 1}

```
@serve.deployment(
    num_replicas=1,
    autoscaling_config={
        "min_replicas": 1,
        "initial_replicas": 2,
        "max_replicas": 5,
        "target_num_ongoing_requests_per_replica": 10,
    },
    ray_actor_options={"num_gpus": 0.5}
)
def func(_):
    time.sleep(1)
    return ""


serve.run(
    func.bind()
)
```

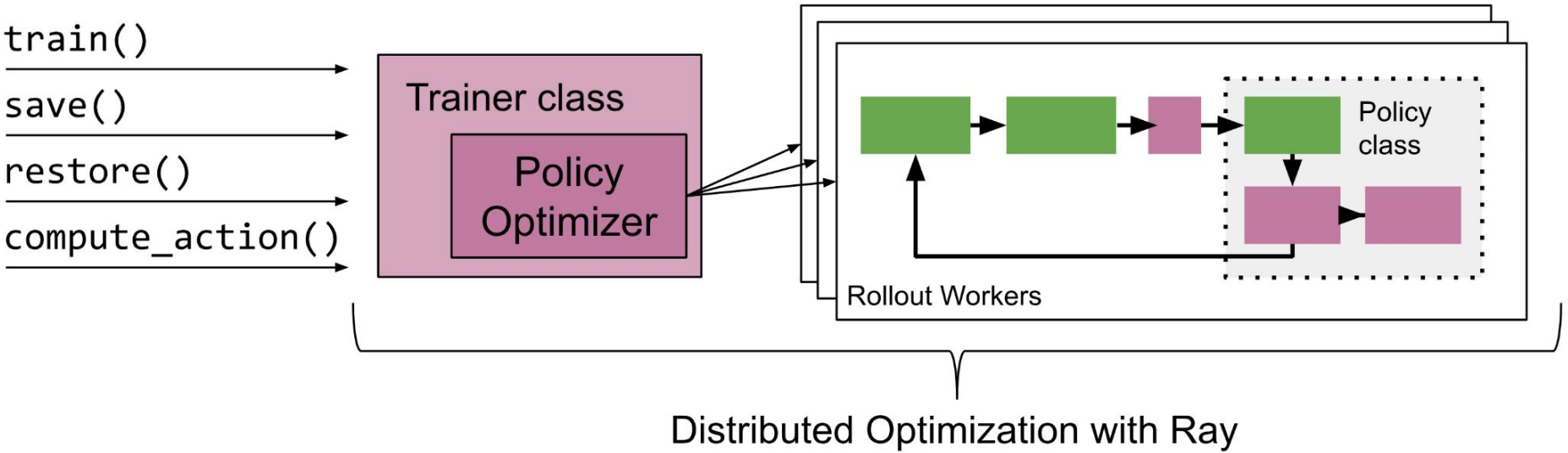
# RayServe - Batching Tutorial

- Serving online queries when your model can take advantage of batching
- How to use
  - @serve.batch(max_batch_size=8, batch_wait_timeout_s=0.1)
    - batch_wait_timeout_s: how long Serve should wait for a batch to process
    - max_batch_size: the batching decorator will wait for a full batch

# RayServe - Tips

- To enhance the reliability of your application, particularly when dealing with large dependencies that may require a significant amount of time to download, consider increasing the value of the `deploymentUnhealthySecondThreshold` to avoid a cluster restart.
- Alternatively, include the dependencies in your image's Dockerfile, so the dependencies are available as soon as the pods start.

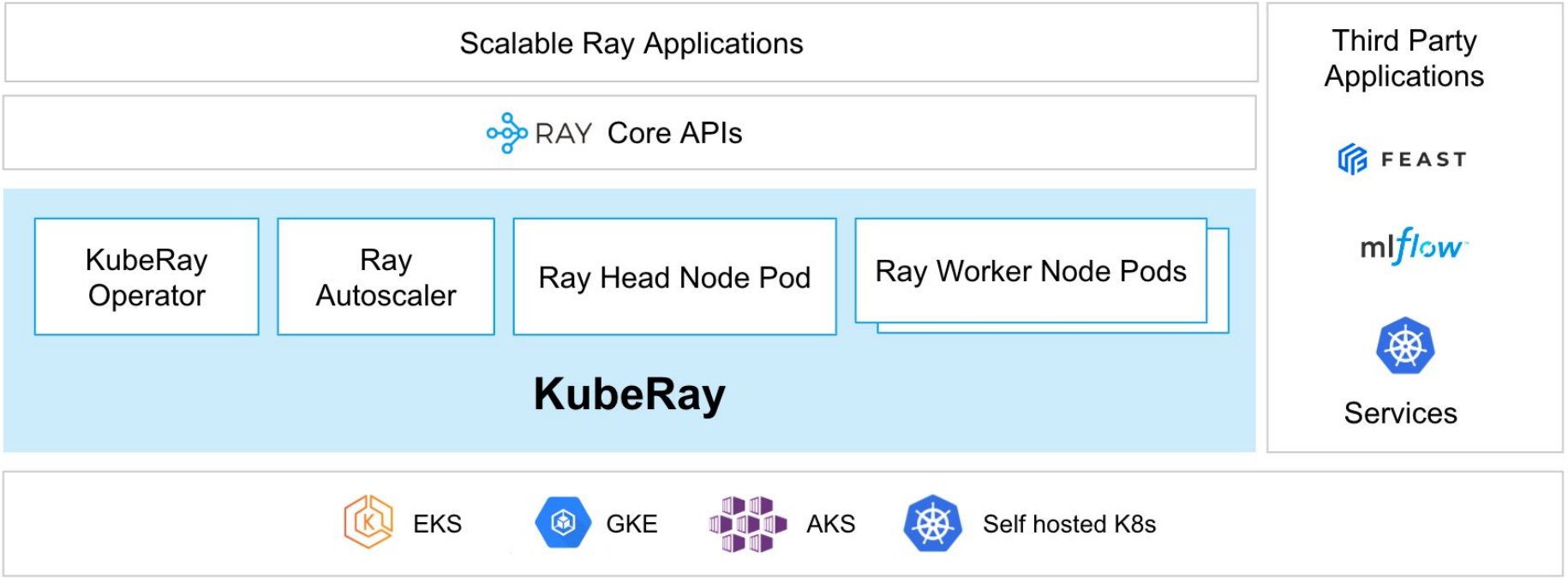
# Ray RLlib - Industry-Grade Reinforcement Learning



# Ray on Kubernetes

Although it's actively developed and maintained, KubeRay is still considered **alpha**, or experimental, so some APIs may be subject to change.

# Overview


Scalable Ray Applications
RAY Core APIs
KubeRay Operator
Ray Autoscaler
Ray Head Node Pod
Ray Worker Node Pods
KubeRay
Third Party Applications
FEAST
mlflow
Services
EKS
GKE
AKS
Self hosted K8s

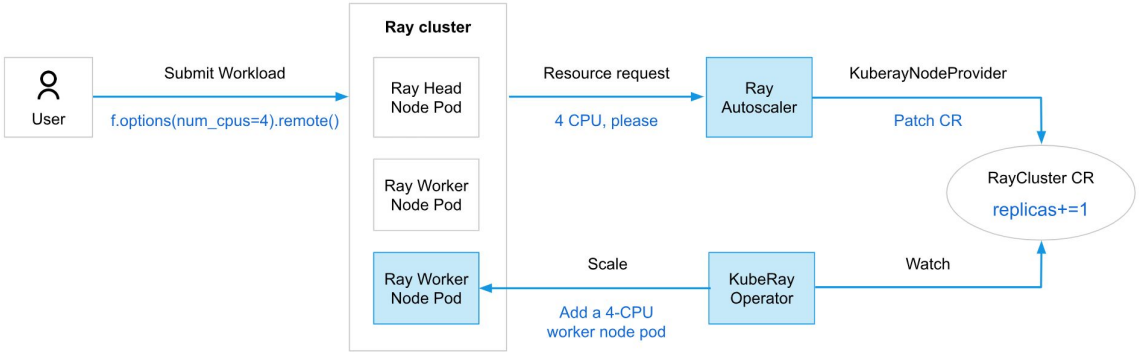
# Installation & Submission

- KubeRay Installation
  - helm repo add kuberay https://ray-project.github.io/kuberay-helm/
  - helm install kuberay-operator kuberay/kuberay-operator --version 0.5.0
- Ray Job submission
  - Submit via CLI
    - Submit: ray job submit --address=<xxx-head-svc>:8265 (you should setup a Ray cluster first)
    - List: ray job list –address=<xxx-head-svc>:8265
  - Submit via RayJob, it will create a Ray cluster automatically
    - Submit: kubectl apply -f
    - List: kubectl get rayjob

# RayJob (alpha)

- Consist of
  - A job
  - Configuration for a Ray cluster
- Lifecycle
  - Creation: RayJob controller will create a Ray cluster and run the job
  - Deletion: Ray cluster can be deleted when the job finishes (optional)
- Suspend Capacity (integrate with queueing system, e.g. Kueue)

# KubeRay AutoScaler




*The autoscalerOptions field provides options for configuring the autoscaler container.*

# Ray Autoscaler In Kubernetes

- Ray Autoscaler VS HPA - Ray scaling nodes as Pod
  - HPA makes decision based on metrics
  - Ray makes decision based on the logic resource demands expressed in tass/actor annotations, i.e. 20 tasks x @ray.remote(num_cpus=5) → 10 Ray pods with a limit of 10 CPUs
  - Fine-grained control of scale-down
- Ray Autoscaler VS Cluster Autoscaler - one Ray pod fits per Kubernetes node
  - Ray pod scaling events will one-to-one cluster autoscaler node scaling events
- Ray Autoscaler VS VPA
  - Not related
  - If Ray pod's load is too high, we can increase the resource requirements specified in the ray.remote annotation

# RayService

- Consist of
  - A KubeRay RayCluster config defining the cluster that the Serve application runs on
  - A Ray Serve config defining the Serve application to run on the cluster
- It will create a RayCluster, a Serve application and a Kubernetes Service
- Provide
  - Kubernetes native support for Ray cluster and Ray Serve application
  - In-place update for Ray Serve application
  - Zero downtime upgrade for Ray cluster(rolling update)
  - Service HA (RayService will monitor the Ray cluster and Serve deployments' health statuses)

# Ray Dashboard

- See Dashboard Instance

# Using GPUs in KubeRay

- Supply CUDA-based container images packaged with Ray and certain machine learning libraries, see Docker Hub
- Configuring Ray Pod requests with GPU
- Annotate tasks and actors with @ray.remote(num_gpus=1)
- Autoscaling Capacity
  - Triggered by the Ray worker group's GPU capacity

```python
import ray
ray.init()
@ray.remote(num_gpus=1)
class GPUActor:
    def say_hello(self):
        print("I live in a pod with GPU access.")
# Request actor placement.
gpu_actors = [GPUActor.remote() for _ in range(2)]
# The following command will block until two Ray pods with GPU
access are scaled
# up and the actors are placed.
ray.get([actor.say_hello.remote() for actor in gpu_actors])
```

  - Direct request, i.e. ray.autoscaler.sdk.request_resources(bundles=[{"GPU": 1}] * 2)
- Override Ray Pod GPU capacity, set the rayStartParams.num-gpus

# GCS Fault Tolerance(exp.)

- Intro: Ray cluster can tolerate GCS failures by using external storage backend (HA Redis) and recover from failures without affecting important services, the Ray head's GCS will recover its state from the external Redis instance
- Enable:

```
kind: RayCluster
metadata:
  annotations:
    ray.io/ft-enabled: "true"
    ray.io/external-storage-namespace: "my-raycluster-storage-namespace" # RayCluster UID by default
```

- Mechanism:
  - Newly created Ray cluster has Readiness Probe and Liveness Probe added to all the head/worker nodes
  - KubeRay Operator controller watches for Event object changes which can notify in case of readiness probe failures and mark them as Unhealthy
  - KubeRay Operator controller delete and recreate any Unhealthy Ray head/worker node
- External storage namespaces can be used to share a single storage backend among multiple Ray clusters

# Best Practice

- Ray container should be the first in the containers list
- Best to run one large Ray pod per Kubernetes node
  - more efficient use of each Ray pod's shared memory object store
  - reduced communication overhead between Ray pods
  - reduced redundancy of per-pod Ray control structures such as Raylets
- Set num-cpus:"0" for the Ray head pod to prevent Ray workloads from being scheduled on the head
- Every Ray container's configuration better to enable graceful shutdown

```
lifecycle:
  preStop:
    exec:
       command: ["/bin/sh","-c","ray stop"]
```

- CPU, GPU, and memory requests will be ignored by Ray, better to set resource requests equal to resource limits.

# Summary

# In short

Ray provides a unified API for the ML ecosystem. This diagram shows how Ray enables an ecosystem of libraries to be run at scale in just a few lines of code.