Introduction to Distributed workload with Ray on Kubernetes

Mofi Rahman

@mofi.dev

Abdelfettah Sghiouar

@boredabdel.bsky.social

Distributed Computing

Why use one computer to solve a problem when you can use thousands?

Distributed Computing

Python is the “lingua franca” of AI

With GenAI distributed compute is no longer optional, it is **required**

Why Distributed Computing?

Scalability

Availability

Efficiency

Flexibility

Challenges

Consistency

Fault Tolerance

Concurrency Control

Load Balancing

Security Concerns

Complexity of Management

CAP Theorem

Consistency

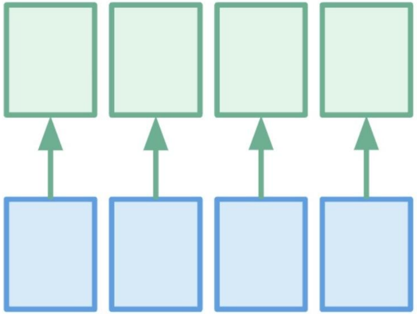
Availability

Partition Tolerance

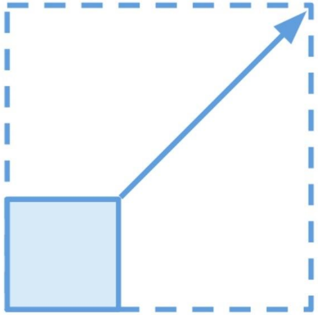
Ray

Ray - Key Characteristics

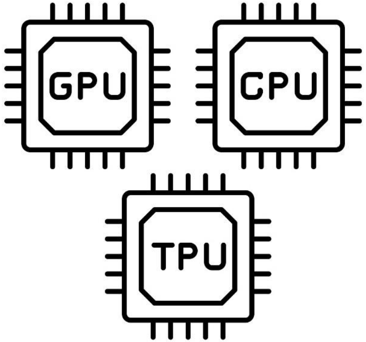
Python first approach, open source



Simple and flexible API

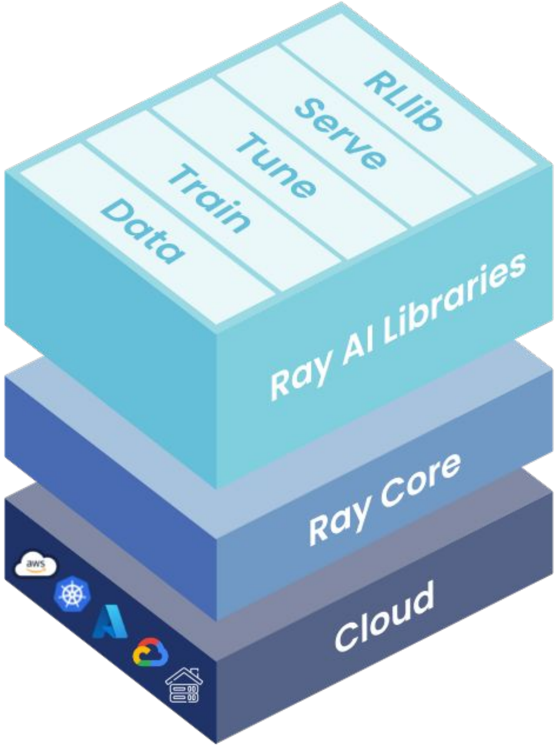


Scalability



Support for bleeding edge hardware

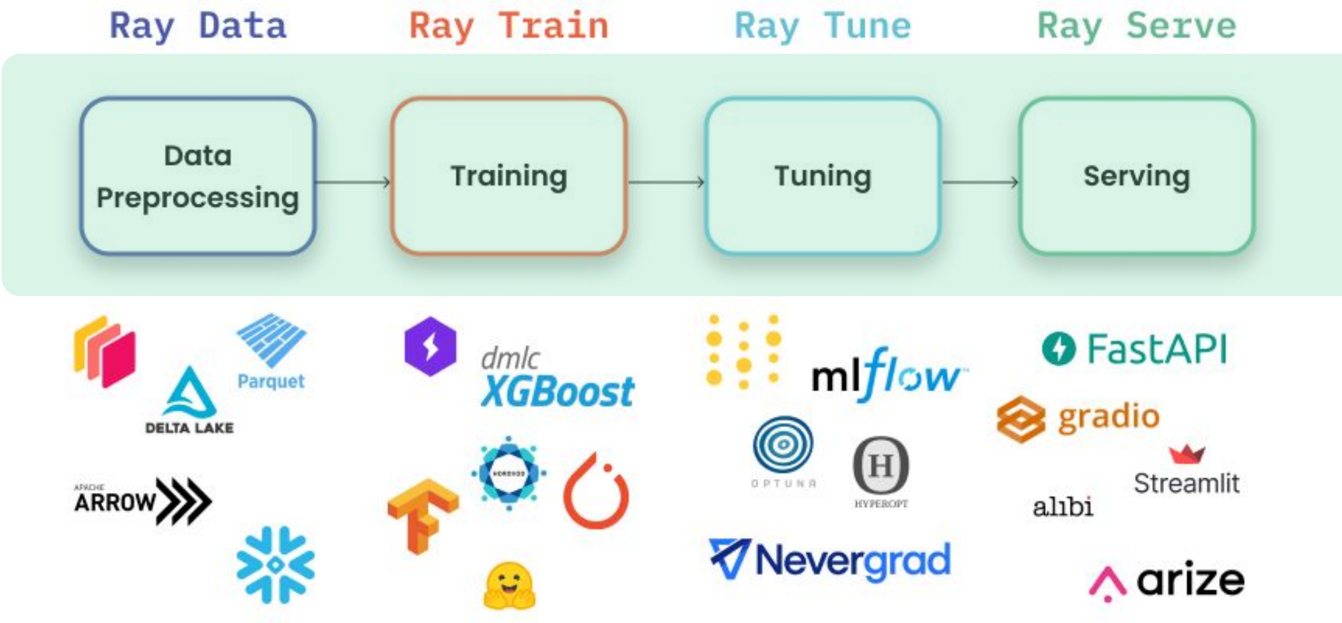
Ray - Components



high-level libraries that enable simple scaling of AI workloads

a low-level distributed computing framework with a concise core and Python-first API

Ray AI Libraries



High-level libraries that make scaling easy for both data scientists and ML engineers.

Key Concepts

Tasks

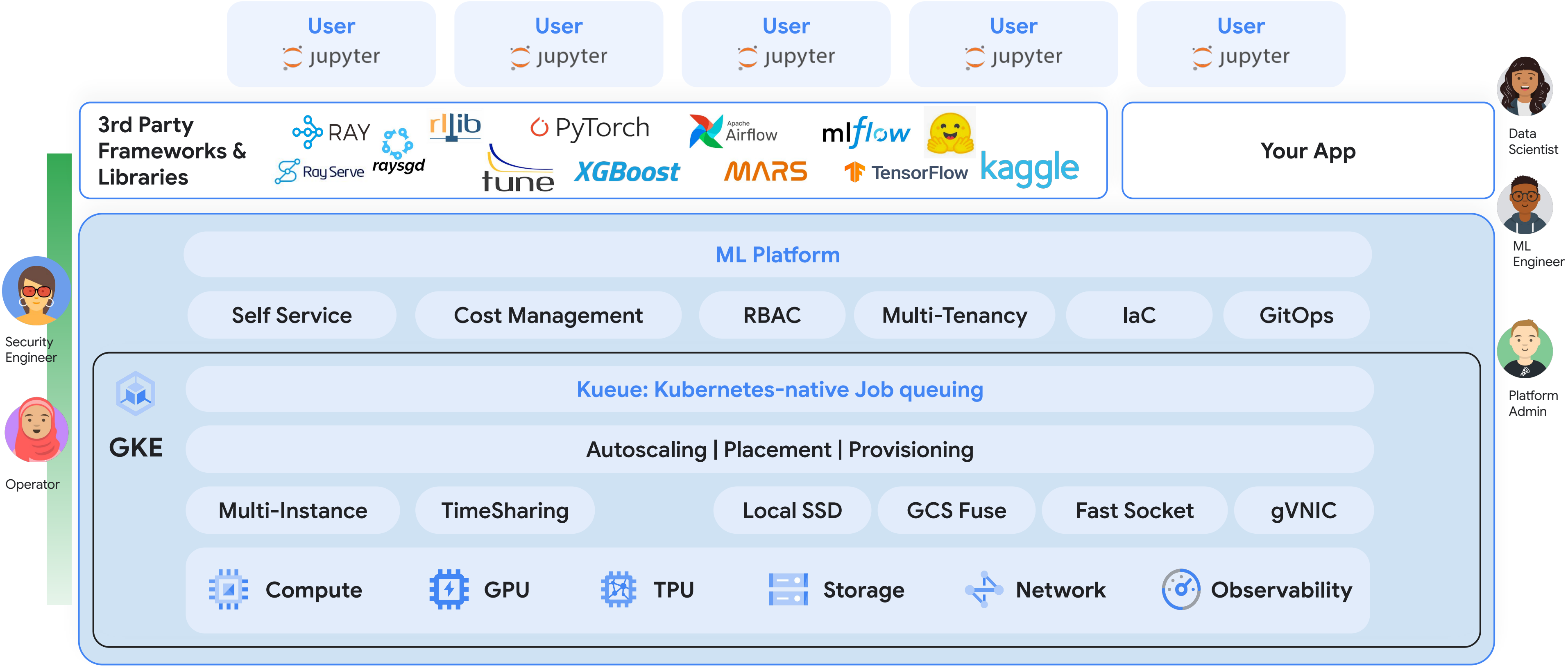
Actors

Objects

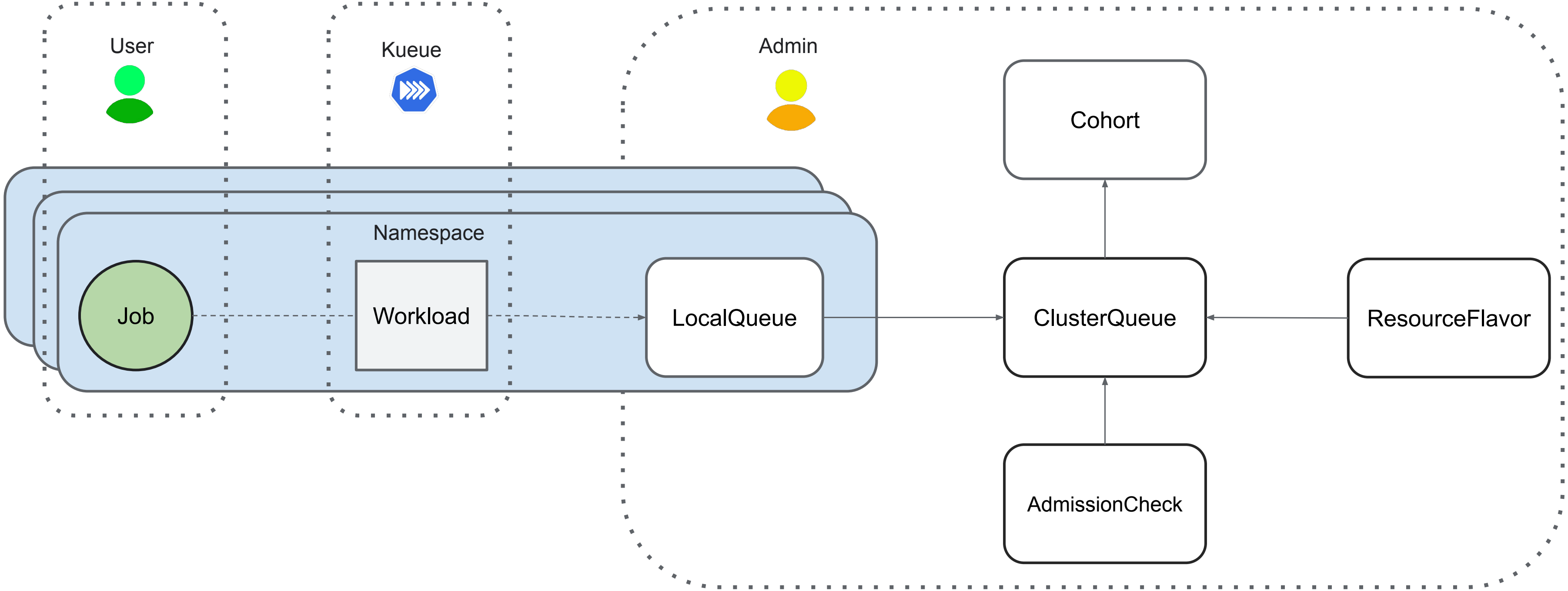
Demo

ML Platform

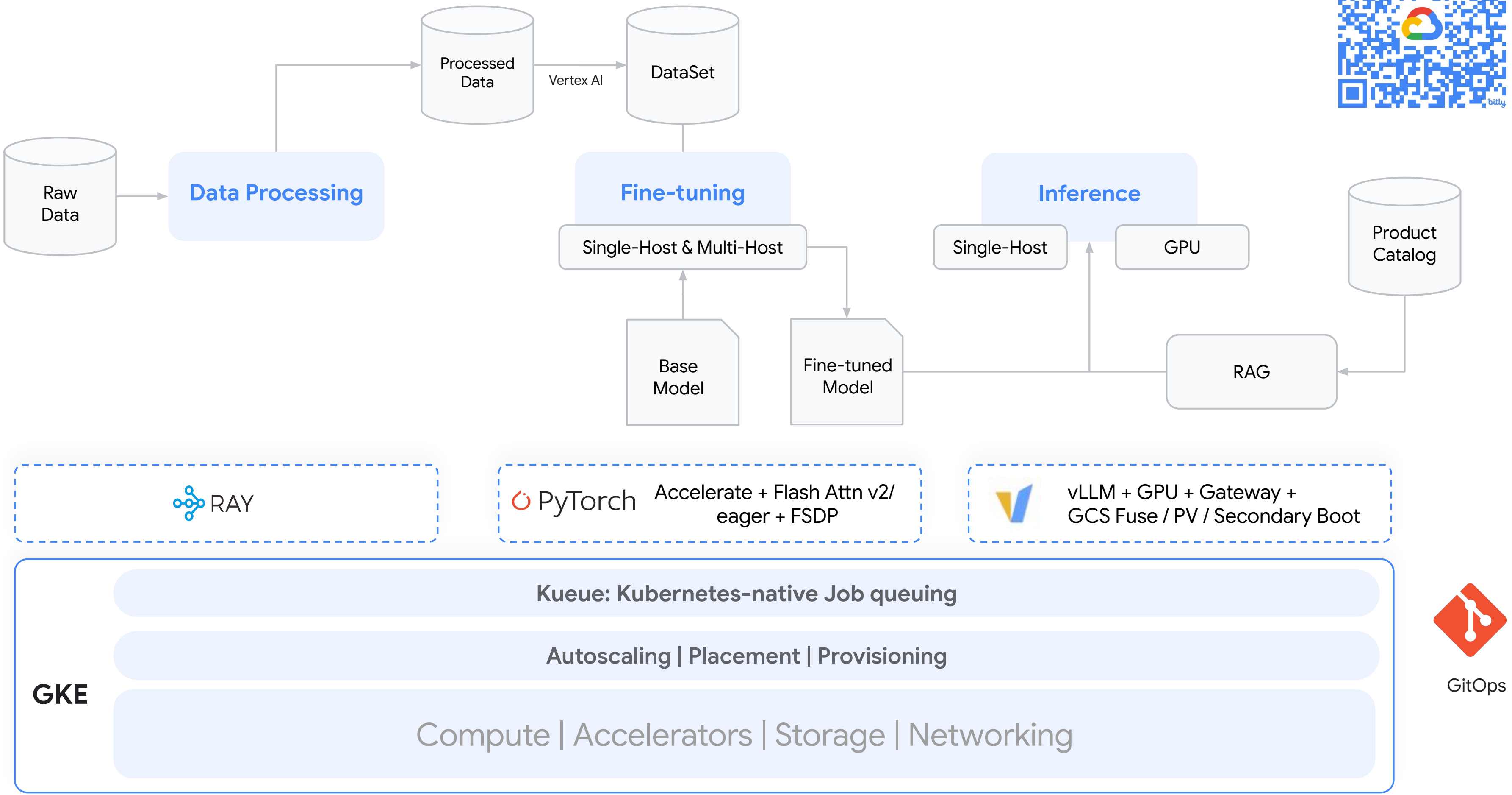
The ever growing AI/ML ecosystem



Kueue


User
Kueue
Admin
Cohort
Namespace
Job
Workload
LocalQueue
ClusterQueue
ResourceFlavor
AdmissionCheck

Data Preprocessing, LLM Fine-Tuning, Inference at Scale



Raw Data

Data Processing

Processed Data

Vertex AI

DataSet

Fine-tuning

Single-Host & Multi-Host

Base Model

Fine-tuned Model

Inference

Single-Host

GPU

Product Catalog

RAG

RAY

PyTorch

Accelerate + Flash Attn v2/ eager + FSDP

vLLM + GPU + Gateway + GCS Fuse / PV / Secondary Boot

GKE

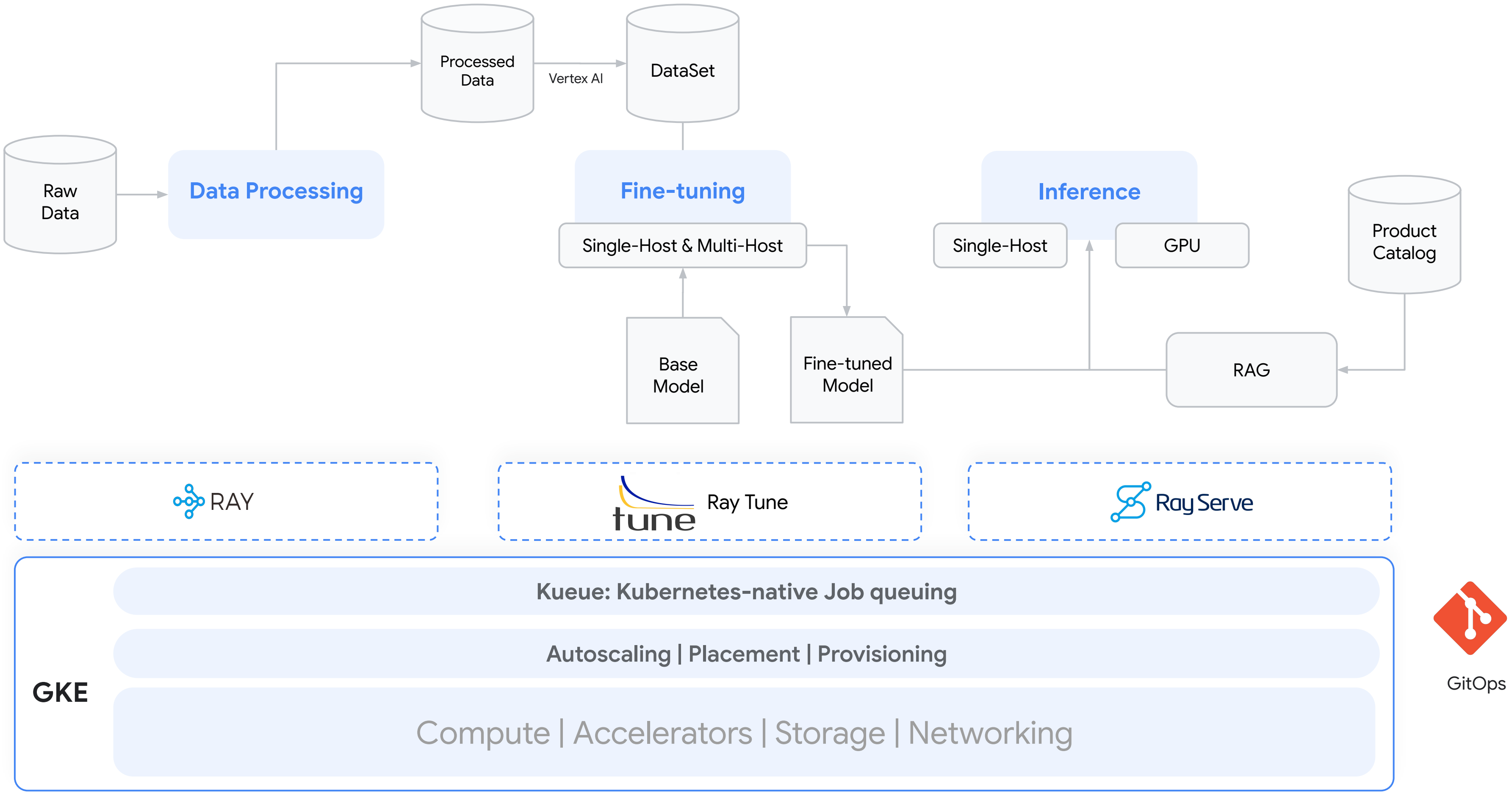
Kueue: Kubernetes-native Job queuing

Autoscaling | Placement | Provisioning

Compute | Accelerators | Storage | Networking

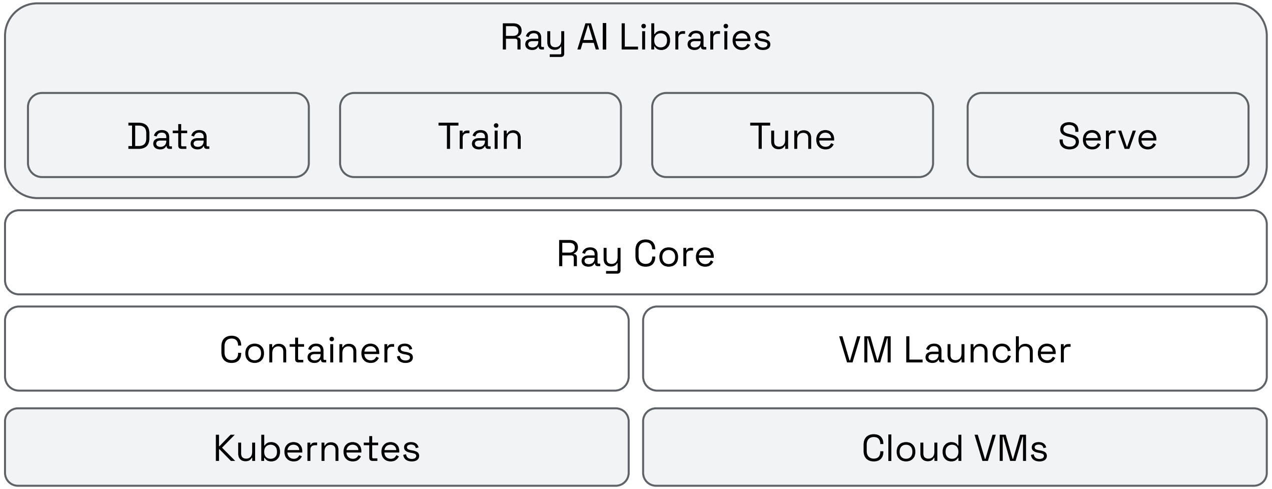
GitOps

Data Preprocessing, LLM Fine-Tuning, Inference at Scale

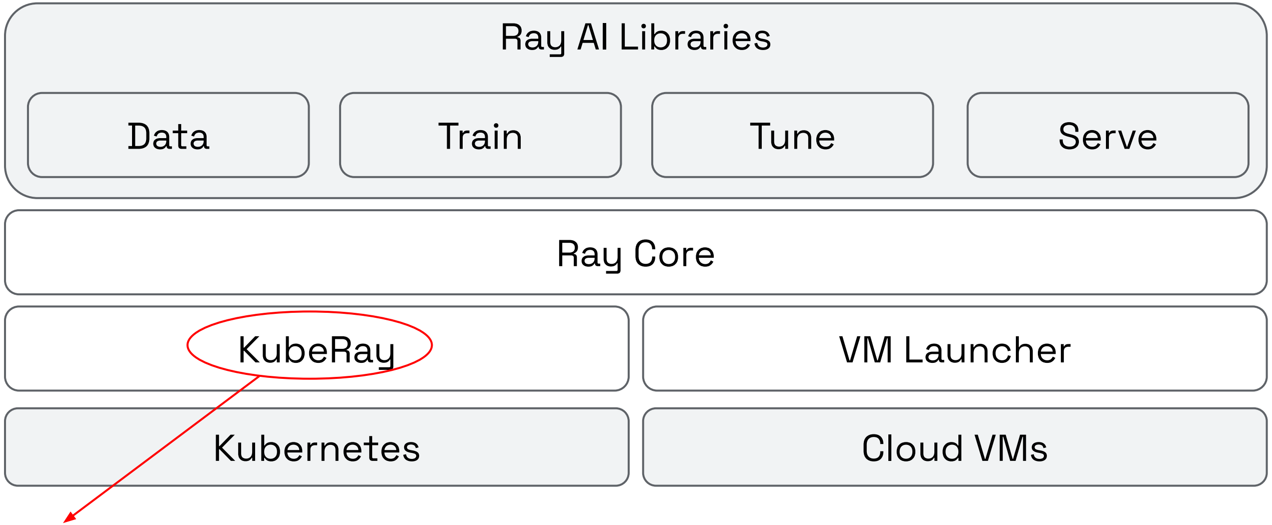


So how do we run Ray?

Ray can be run anywhere!

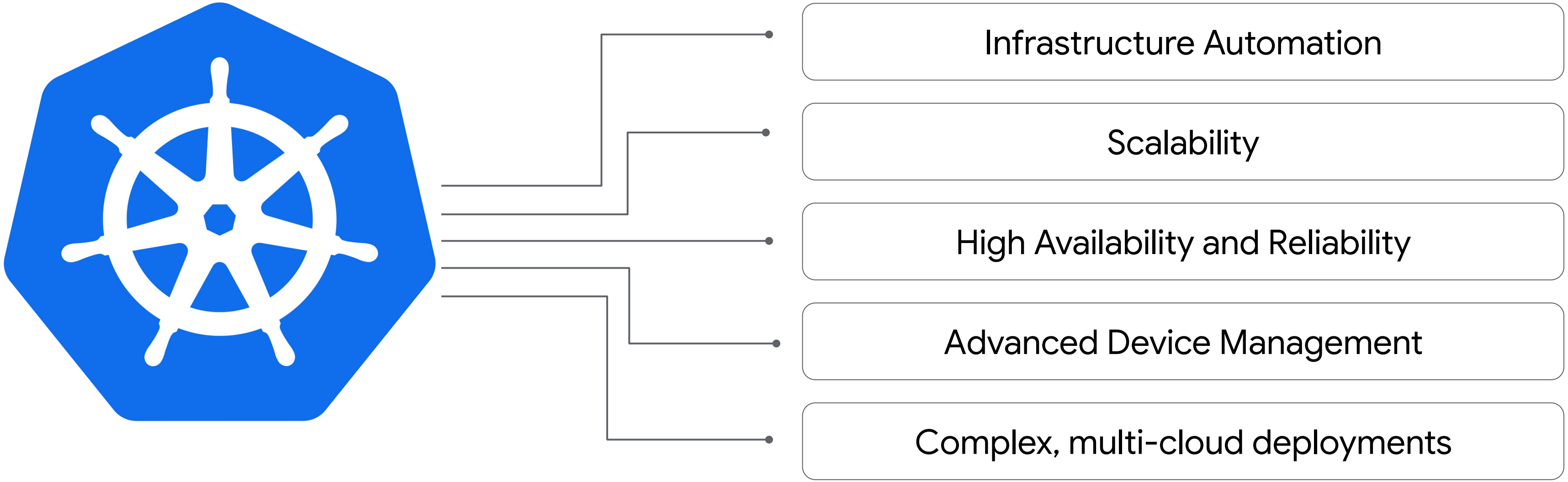


Enter Kuberay



Manages the lifecycle of Ray clusters and associated applications on Kubernetes.

Why Ray on Kubernetes?



Infrastructure Automation

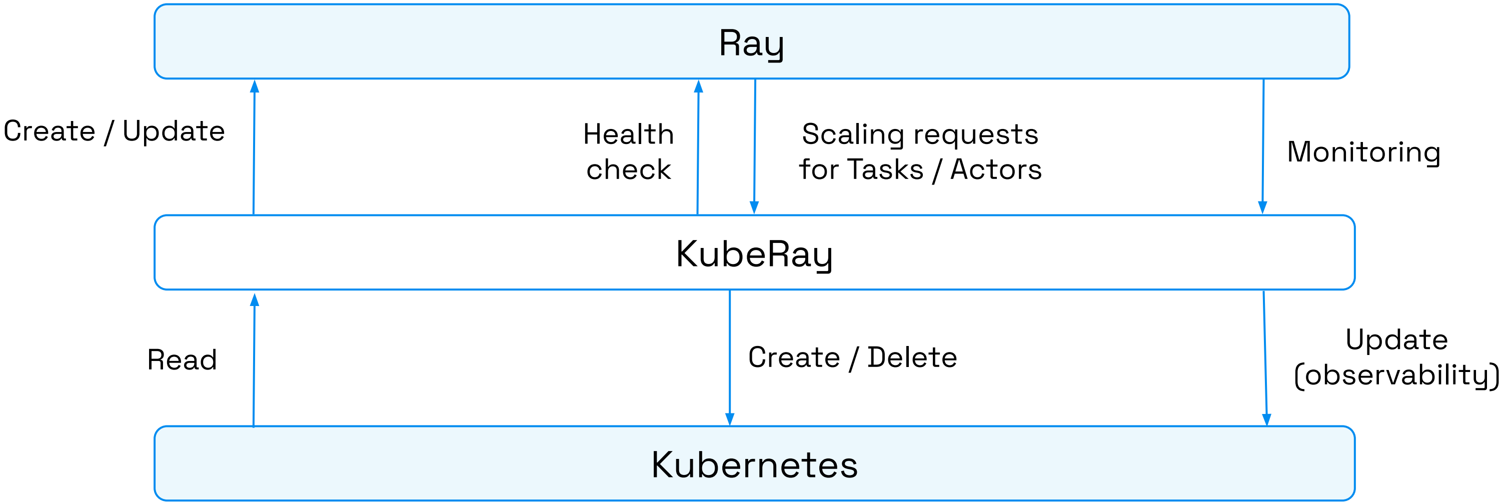
Scalability

High Availability and Reliability

Advanced Device Management

Complex, multi-cloud deployments

KubeRay: The best solution for Ray on Kubernetes



KubeRay APIs

RayCluster

Manage and scale Ray clusters

Ideal for prototyping / development

RayJob

Execute a Ray job with ephemeral Ray clusters

Ideal for productionizing Ray batch workloads

RayService

Deploy a Ray Serve application with zero-downtime upgrades

Ideal for inference in production

Demo

Ephemeral vs Persistent Ray Cluster

Prons

- Reproduciblity
- No need for maintenance since a fresh cluster is started for each job
- Better observability for single distributed jobs metrics

Cons

- Startup latency can be long
- Ray dashboard lasts as long as the workload
- Logs and Metrics has to be stored outside the cluster

Ephemeral vs Persistent Ray Cluster

Prons

- Startup latency for workload is small
- Minimal packaging required if the clusters already has the dependencies
- Ray dashboard can be used to track history

Cons

- New dependencies are tricky
- If cluster is brought down for maintenance the behavior can be unpredictable

Security

- Ray endpoints are not locked down by default (not Authn or Authz)
    - ➩ Leverage Cloud Providers tools to secure the endpoints (LBs, proxies…)

- A lot of layers to secure (Kubernetes, Ray pods, Workloads…)
    - ➩ Kubernetes tools can be used to secure the clusters and the pods (RBAC, namespaces, Quotas, Pod Security Policies…)

KubeRay is growing!

Community

1000+ commits

140+ contributors

Adoption

100s organizations

50+ blogs & talks

Scale

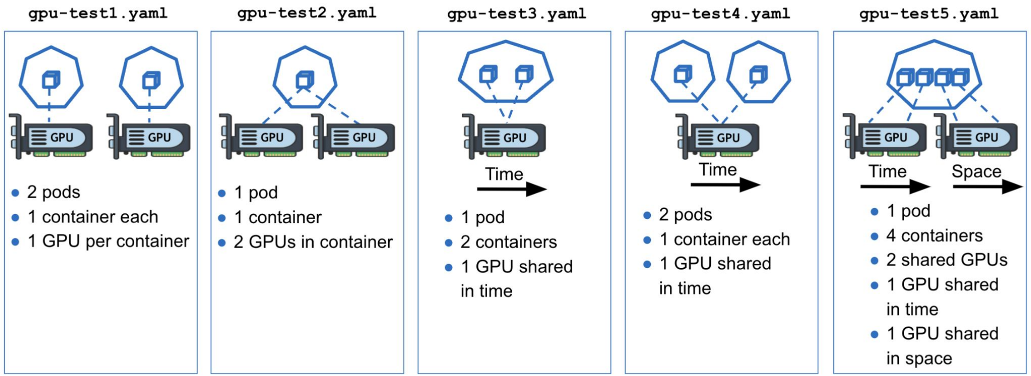
10K Ray clusters

40K Pods

DRA: Optimizing Resource Allocation

DRA enhances the Kubernetes scheduler with awareness of Ray's needs and the dynamic nature of certain workloads:

- Optimized resource utilization
- Improved cluster efficiency



`gpu-test1.yaml`

- 2 pods
- 1 container each
- 1 GPU per container

`gpu-test2.yaml`

- 1 pod
- 1 container
- 2 GPUs in container

`gpu-test3.yaml`

- 1 pod
- 2 containers
- 1 GPU shared in time

`gpu-test4.yaml`

- 2 pods
- 1 container each
- 1 GPU shared in time

`gpu-test5.yaml`

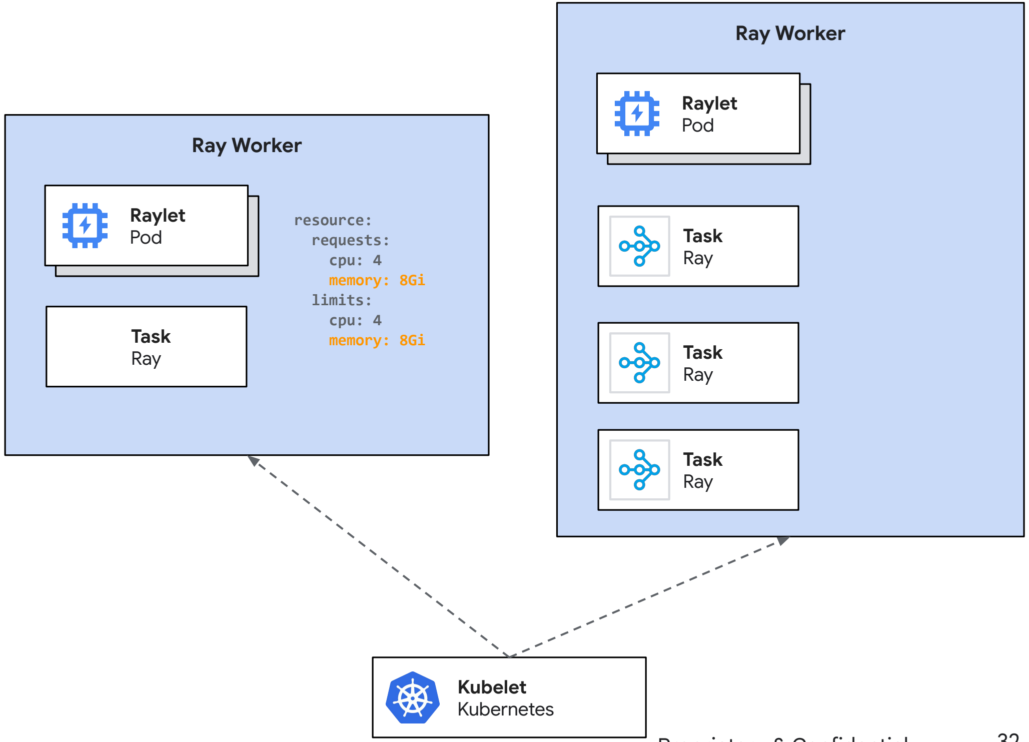
- 1 pod
- 4 containers
- 2 shared GPUs
- 1 GPU shared in time
- 1 GPU shared in space

Kubernetes v1.31 introduced new Device Resource Assignment (DRA) APIs.

In-place VPA: Minimizing Disruptions

In-place Vertical Pod Autoscaling enables elastic memory consumption for Ray containers without requiring restarts.

Prevent performance degradation and risk of OOM-kill with resizable Pod memory



```
resource:
  requests:
    cpu: 4
    memory: 8Gi
  limits:
    cpu: 4
    memory: 8Gi
```

Ray Operator on GKE

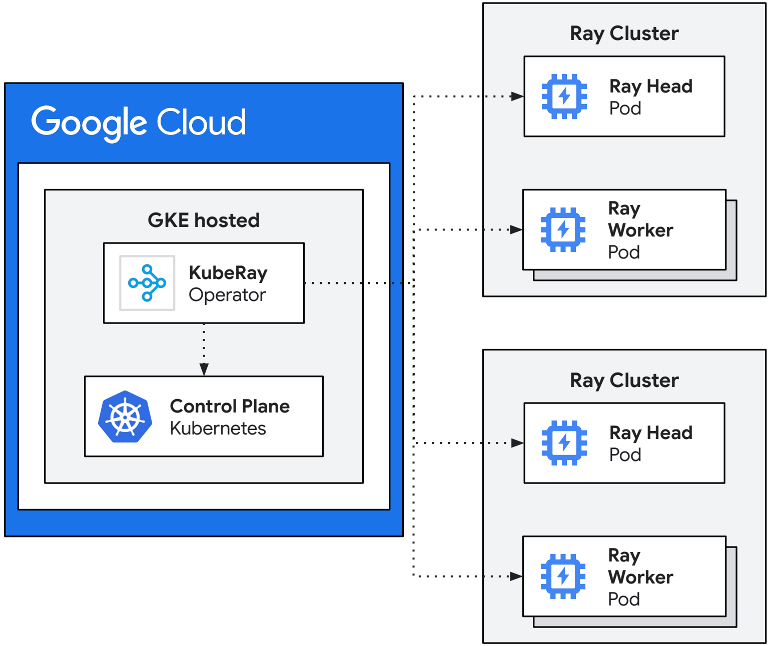
GKE hosts the KubeRay operator on your behalf.

Optimally configured for high performance and scalability.

Get started with Ray on GKE with a single option using:

- gcloud CLI
- Google Cloud Console
- Terraform

See Enable the Ray operator on Google Kubernetes Engine for more details.



Feedback 🙏

Thank you

Google Cloud