Ray
On GKE

Niklas Voss
Google Cloud

Berlin
Oct 18, 2023

Agenda

01 Why is ML hard?

02 How does Ray help?

03 KubeRay

04 Learnings running KubeRay on GKE

Why is ML hard?

01

Google Cloud

Simple ML training program by data scientist

```python
import pandas
from sklearn import compose, impute, model, pipeline, preprocessor
import my_pkg

housing_df = pandas.read_parquet(...)  # Raw training data I/O

# Preprocessing definitions
num_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.StandardScaler())
cat_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.OneHotEncoder(...))

my_preprocessor = ColumnTransformer([("bedrooms", my_pkg.ratio_pipeline(...), [...]),
                                     ("log", my_pkg.log_pipeline, [...]),
                                     ("geo", my_pkg.similarity_pipeline, [...],
                                     ("cat", cat_pipeline, compose.make_column_selector(...))],
                                    remainder=num_pipeline)

# Create the dataset split
strat_train_set, strat_test_set = model.train_test_split(housing_df, test_size=0.2,
stratify=...)

# Model training loop
model = pipeline.make_pipeline(my_preprocessor, sklearn.linear_modeol.LinearRegression())
model.fit(strat_train_set, strat_train_set["label_col”].copy())

# Batch prediction and offline evaluation
X_test, y_test =  strat_test_set.drop("label_col”, axis=1), strat_test_set["label_col”].copy()
predictions = model.predict(X_test)
rmse = mean_squared_error(y_test, predictions, squared=False)
```

Google Cloud

Why is a simple ML training program difficult?

```
import pandas
from sklearn import compose, impute, model, pipeline, preprocessor
import my_pkg

housing_df = pandas.read_parquet(...)  # Raw training data I/O

# Preprocessing definitions
num_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.StandardScaler())
cat_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.OneHotEncoder(...))

my_preprocessor = ColumnTransformer([("bedrooms", my_pkg.ratio_pipeline(...), [...]),
                                     ("log", my_pkg.log_pipeline, [...]),
                                     ("geo", my_pkg.similarity_pipeline, [...],
                                     ("cat", cat_pipeline, compose.make_column_selector(...))],
                                    remainder=num_pipeline)

# Create the dataset split
strat_train_set, strat_test_set = model.train_test_split(housing_df, test_size=0.2,
stratify=...)

# Model training loop
model = pipeline.make_pipeline(my_preprocessor, sklearn.linear_modeol.LinearRegression())
model.fit(strat_train_set, strat_train_set["label_col”].copy())

# Batch prediction and offline evaluation
X_test, y_test =  strat_test_set.drop("label_col”, axis=1), strat_test_set["label_col”].copy()
predictions = model.predict(X_test)
rmse = mean_squared_error(y_test, predictions, squared=False)
```

Google Cloud

- **No clear scope delineation** of “data processing” code and “modeling” code.

```
import pandas
from sklearn import compose, impute, model, pipeline, preprocessor
import my_pkg

housing_df = pandas.read_parquet(...)  # Raw training data I/O

# Preprocessing definitions
num_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.StandardScaler())
cat_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.OneHotEncoder(...))

my_preprocessor = ColumnTransformer([("bedrooms", my_pkg.ratio_pipeline(...), [...]),
                                     ("log", my_pkg.log_pipeline, [...]),
                                     ("geo", my_pkg.similarity_pipeline, [...],
                                     ("cat", cat_pipeline, compose.make_column_selector(...))],
                                    remainder=num_pipeline)

# Create the dataset split
strat_train_set, strat_test_set = model.train_test_split(housing_df, test_size=0.2,
stratify=...)

# Model training loop
model = pipeline.make_pipeline(my_preprocessor, sklearn.linear_modeol.LinearRegression())
model.fit(strat_train_set, strat_train_set["label_col”].copy())

# Batch prediction and offline evaluation
X_test, y_test =  strat_test_set.drop("label_col”, axis=1), strat_test_set["label_col”].copy()
predictions = model.predict(X_test)
rmse = mean_squared_error(y_test, predictions, squared=False)
```

data code

model code

Google Cloud

- **No clear scope delineation** of "data processing" code and "modeling" code.
- Different kinds of computation yields **different distribution characteristics**

```
import pandas
from sklearn import compose, impute, model, pipeline, preprocessor
import my_pkg
```

likely big, sharded file loaded from disk

```
housing_df = pandas.read_parquet(...) # Raw training data I/O
```

stateful processing req'ed

```
# Preprocessing definitions
num_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.StandardScaler())
cat_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.OneHotEncoder(...))
```

(mostly) stateless

mix of stateful and stateless

```
my_preprocessor = ColumnTransformer([("bedrooms", my_pkg.ratio_pipeline(...), [...]),
                                     ("log", my_pkg.log_pipeline, [...]),
                                     ("geo", my_pkg.similarity_pipeline, [...],
                                     ("cat", cat_pipeline, compose.make_column_selector(...))],
                                    remainder=num_pipeline)
```

stateless

```
# Create the dataset split
strat_train_set, strat_test_set = model.train_test_split(housing_df, test_size=0.2,
stratify=...)
```

mix of stateless and stateful (potentially big) data-parallel computation

```
# Model training loop
model = pipeline.make_pipeline(my_preprocessor, sklearn.linear_modeol.LinearRegression())
model.fit(strat_train_set, strat_train_set["label_col"].copy())
```

stateful (potentially big) model-parallel computation

mostly stateless

```
# Batch prediction and offline evaluation
X_test, y_test =  strat_test_set.drop("label_col", axis=1), strat_test_set["label_col"].copy()
predictions = model.predict(X_test)
rmse = mean_squared_error(y_test, predictions, squared=False)
```

Google Cloud

- **No clear scope delineation** of "data processing" code and "modeling" code.

- Different kinds of computation yields **different distribution characteristics**

- **Try and error is essential**: rapid iterations and quick response-to-result is critical.

```
import pandas
from sklearn import compose, impute, model, pipeline, preprocessor
import my_pkg

housing_df = pandas.read_parquet(...)  # Raw training data I/O

# Preprocessing definitions
num_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.StandardScaler())
cat_pipeline = pipeline.make_pipeline(SimpleImputer(...), preprocessing.OneHotEncoder(...))

my_preprocessor = ColumnTransformer([("bedrooms", my_pkg.ratio_pipeline(...), [...]),
                                     ("log", my_pkg.log_pipeline, [...]),
                                     ("geo", my_pkg.similarity_pipeline, [...],
                                     ("cat", cat_pipeline, compose.make_column_selector(...))],
                                    remainder=num_pipeline)


# Create the dataset split
strat_train_set, strat_test_set = model.train_test_split(housing_df, test_size=0.2,
stratify=...)

# Model training loop
model = pipeline.make_pipeline(my_preprocessor, sklearn.linear_modeol.LinearRegression())
model.fit(strat_train_set, strat_train_set["label_col"].copy())

# Batch prediction and offline evaluation
X_test, y_test =  strat_test_set.drop("label_col", axis=1), strat_test_set["label_col"].copy()
predictions = model.predict(X_test)
rmse = mean_squared_error(y_test, predictions, squared=False)
```

Google Cloud

How does Ray help?

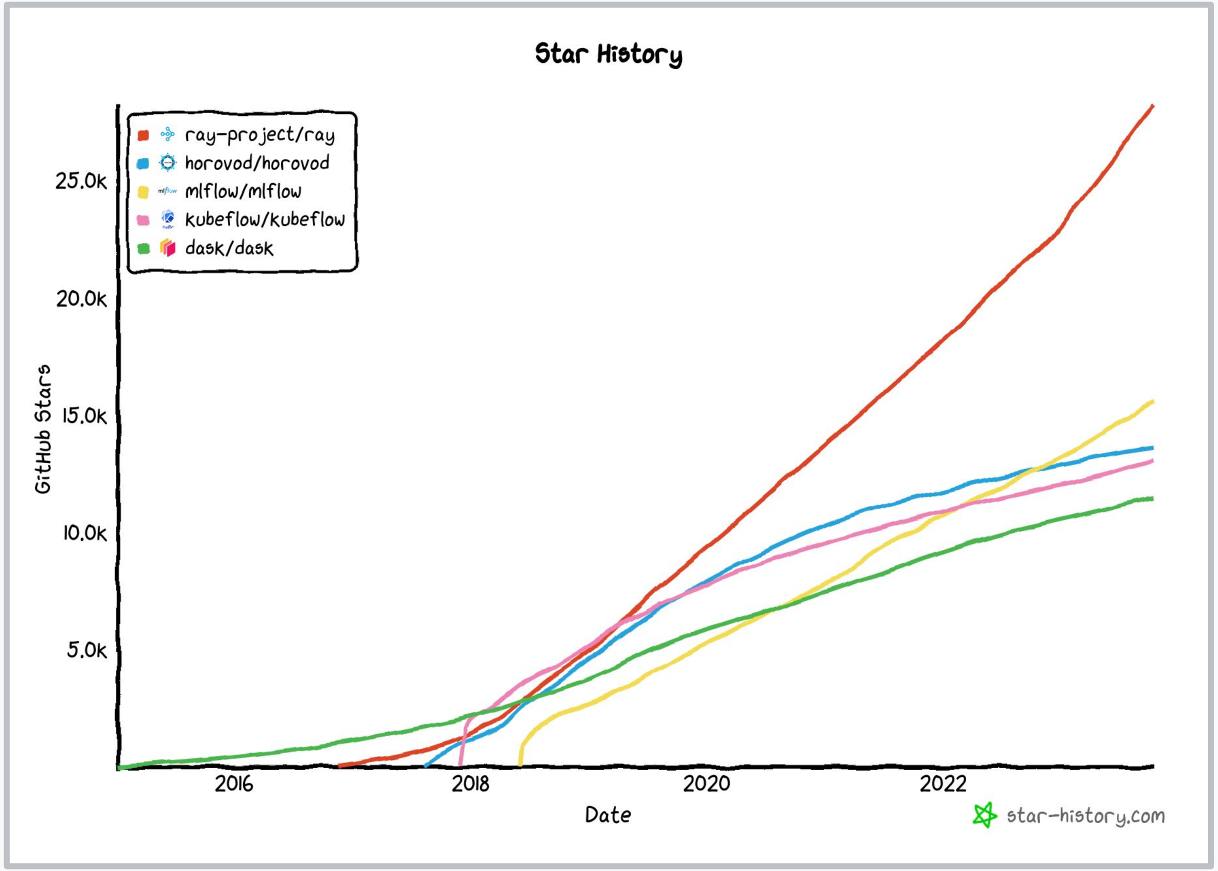
02

Google Cloud

Ray

"**Ray is an open-source unified compute framework that makes it easy to scale AI and Python workloads** — from reinforcement learning to deep learning to tuning, and model serving. Learn more about Ray's rich set of libraries and integrations."

– ray.io



Google Cloud

Ray

Ray is an **open source distributed programming framework for Python**

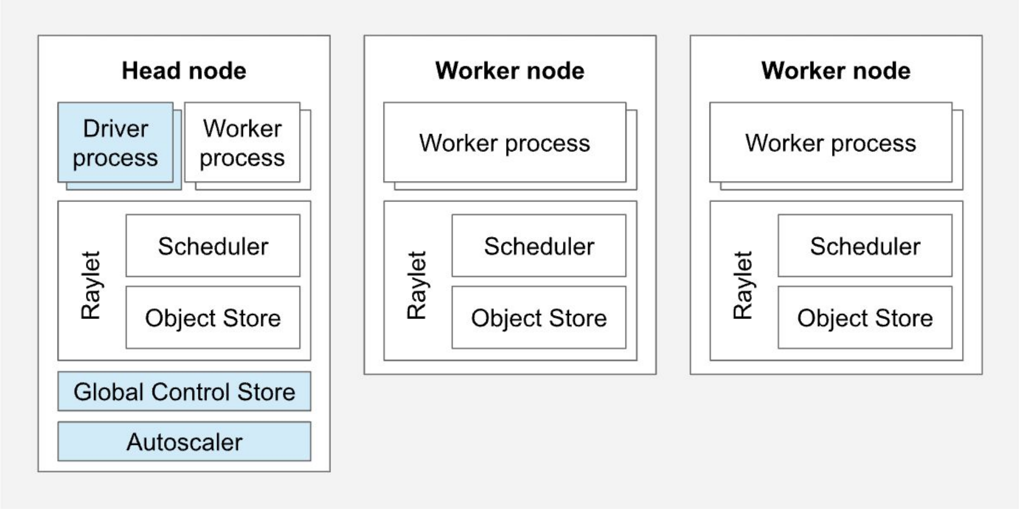
```python
# Define the square task
@ray.remote
def square(x):
    return x * x

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

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

Ray

Ray is an open source distributed programming framework for Python **and its distributed computation platform for Python**

Key Concepts

- **Cluster**: 1 **Head** nodes, >= 0 **Worker** nodes
- **Driver** (Driver process)
- **Raylet** (Scheduler and Object Store)
- **Ray Task** and **Actor**
- **Global Control Store**
- **Autoscaler**



A Ray cluster with two worker nodes. Each node runs Ray helper processes to facilitate distributed scheduling and memory management. The head node runs additional control processes (highlighted in blue).

github.com/ray-project/ray

Ray

Driver program:
Notebook code cells, or
my_ray_app.py

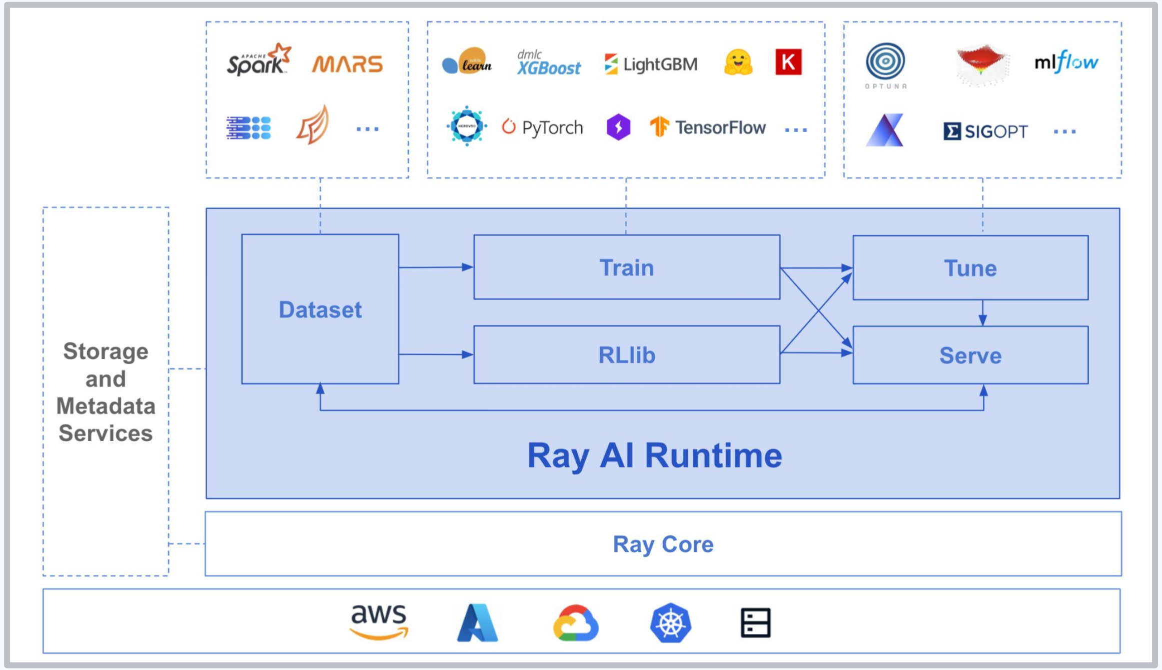
```python
# Define the square task
@ray.remote
def square(x):
    return x * x

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

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

Ray AIR

5 key components:

- **Data processing** (Ray Data)
- **Model Training** (Ray Train)
- **Reinforcement Learning** (Ray RLlib)
- **Hyperparameter Tuning** (Ray Tune)
- **Model Serving** (Ray Serve)

⇒ **Ergonomic distributed Data + ML APIs** using the `@ray.remote` primitive under the hood



github.com/ray-project/ray

Google Cloud

Ray Data + Ray Train

- All code in a **single Python program** (the **Driver** program).
- **Objects and computations distributed** accordingly in the Ray Cluster nodes.

```
import ray
# Initialize the Driver process (i.e. connection to the Cluster Head node)
ray.init(..)

# Create distributed dataset and preprocessing pipeline.
# Internally, Ray Dataset is an iterable of Arrow rows with pyarrow schema.
train_dataset = ray.data.read_parquet("gcs://bucket/training")
preprocessor = ray.data.preprocessors.StandardScaler()

# Distributed last-mile data processing pipeline can be defined.
# In practice, it'd better be a custom preprocessor to mitigate train-serve skew.
train_dataset = train_dataset.map(fn, ...).filter(fn, ...).groupby(fn,
...).shuffle().split()
```

```
trainer =
ray.train.xgboost.XGBoostTrainer(
    scaling_config={"num_workers": 4},
    datasets={"train": train_dataset},
    preprocessor=preprocessor,
    **xgboost_params,
)
```

```
trainer =
ray.train.torch.TorchTrainer(
    torch_train_loop_fn,
    scaling_config={"num_workers": 4},
    datasets={"train": train_dataset},
    preprocessor=preprocessor,
    **torch_params,
)
```

```
# Fit the given Trainer on the distributed dataset.
result = trainer.fit()
```

Ray Train + Ray Tune

- **Tuning loop** would still belong to the **same Driver program**

```python
trainer =
ray.train.xgboost.XGBoostTrainer(
    scaling_config={"num_workers": 4},
    datasets={"train": train_dataset},
    preprocessor=preprocessor,
    **xgboost_params,
)
```

```python
trainer =
ray.train.torch.TorchTrainer(
    torch_train_loop_fn,
    scaling_config={"num_workers": 4},
    datasets={"train": train_dataset},
    preprocessor=preprocessor,
    **torch_params,
)
```

```python
# Fit the given Trainer on the distributed dataset.
result = trainer.fit()

# Optionally, use Tune to optimize in a range of hyper-params for training.
tuner = ray.tune.Tuner(
    trainer,
    param_space={"params": {"model_size": tune.randint(1, 9)}},
    tune_config=TuneConfig(num_samples=5, metric="logloss", search_alg=...),
)
result_grid = tuner.fit()
```

Google Cloud

Batch Prediction (and offline evaluation) = Ray Train + Ray Data

- Batch prediction is a Ray Data processing w/ Predictor
- Implementation can happen in the **same Driver program**

```
trainer =
ray.train.xgboost.XGBoostTrainer(
    scaling_config={"num_workers": 4},
    datasets={"train": train_dataset},
    preprocessor=preprocessor,
    **xgboost_params,
)
```

```
trainer =
ray.train.torch.TorchTrainer(
    torch_train_loop_fn,
    scaling_config={"num_workers": 4},
    datasets={"train": train_dataset},
    preprocessor=preprocessor,
    **torch_params,
)
```

```
# Fit the given Trainer on the distributed dataset.
result = trainer.fit()
```

```
bp = BatchPredictor.from_checkpoint(
    result.checkpoint, XGBoostPredictor)
```

```
bp = BatchPredictor.from_checkpoint(
    result.checkpoint, TorchPredictor)
```

```
# Load historical data, run batch prediction, and write out results.
historical_dataset = ray.data.read_parquet("s3://bucket/historical")
predict_dataset = bp.predict(historical_dataset)
predict_dataset.write_csv("s3://bucket/predict_out")

# You may continue on offline evaluation as Ray Data operations
offline_eval_metrics = predict_dataset.aggregate(eval_metric_fn) ...
```

Google Cloud

Deployment to online serving = Ray Train + Ray Serve

- Ray Serve creates **online serving endpoints** from Predictor in the Ray Cluster
- Implementation can happen in the **same Driver program**

```
trainer =
ray.train.xgboost.XGBoostTrainer(
    scaling_config={"num_workers": 4},
    datasets={"train": train_dataset},
    preprocessor=preprocessor,
    **xgboost_params,
)
```

```
trainer =
ray.train.torch.TorchTrainer(
    torch_train_loop_fn,
    scaling_config={"num_workers": 4},
    datasets={"train": train_dataset},
    preprocessor=preprocessor,
    **torch_params,
)
```

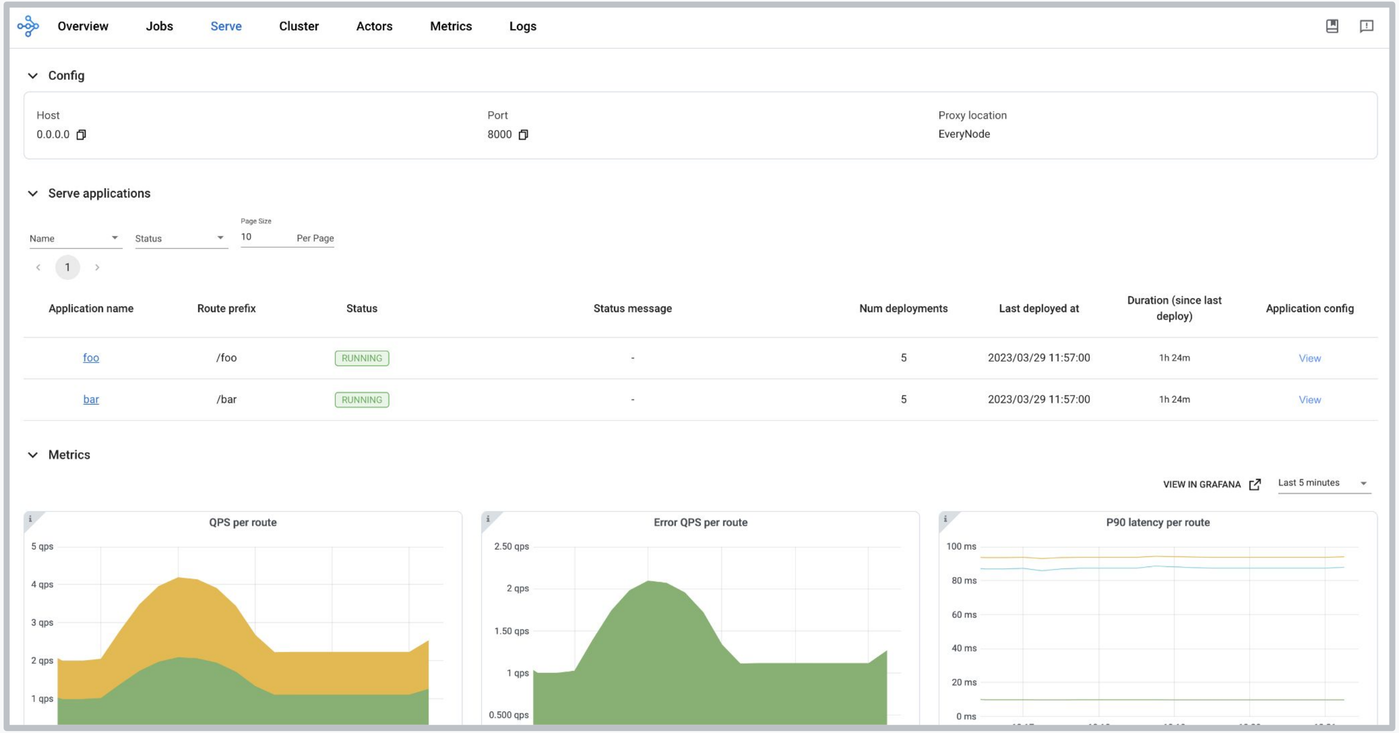
```
# Fit the given Trainer on the distributed dataset.
result = trainer.fit()
```

```
deployment =
ray.serve.PredictorDeployment.deploy(
    XGBoostPredictor, result.checkpoint)
```

```
deployment =
ray.serve.PredictorDeployment.deploy(
    TorchPredictor, result.checkpoint)
```

```
# Inspect the HTTP endpoint of the Serve deployment.
print(deployment.url)
```

Ray Dashboard



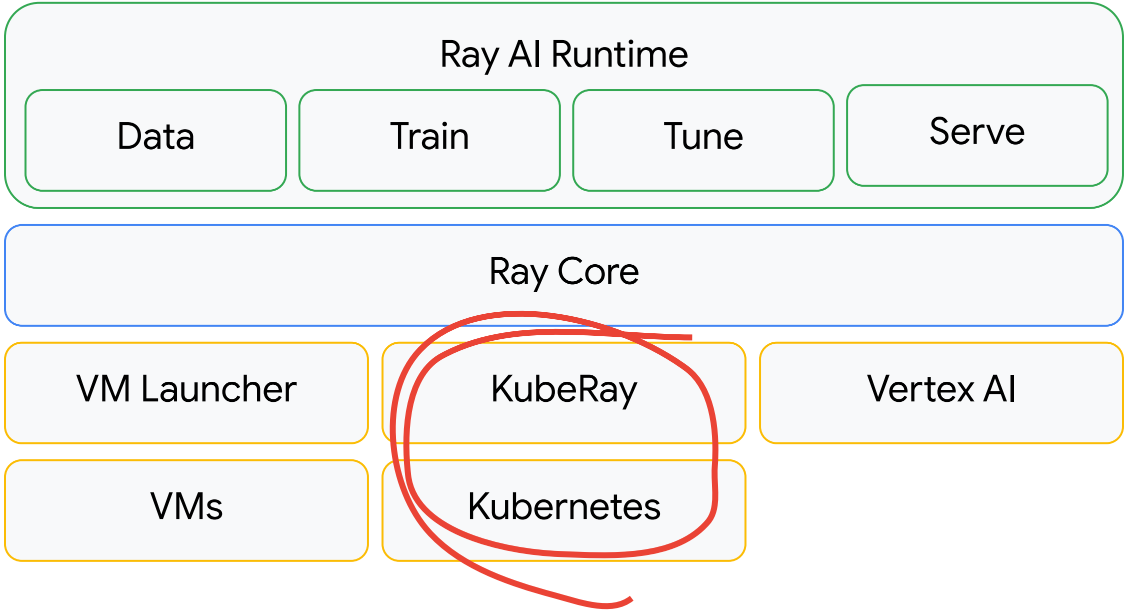
Google Cloud

KubeRay

03

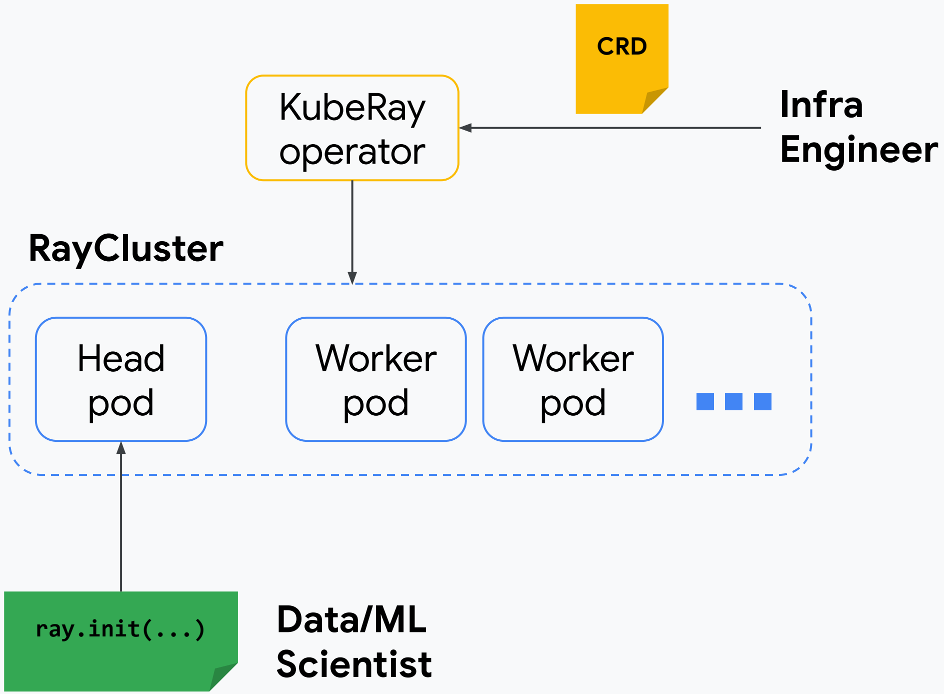
Google Cloud

Integration options with Ray on Google Cloud



KubeRay

- Follows **Kubernetes operator** pattern
- Integrates Ray into Kubernetes ecosystem
- **Seamless autoscaling**



Google Cloud

KubeRay introduces 3 custom resources

RayCluster

- Manage lifecycle of Ray cluster
- **Autoscaling**
- **GCS fault tolerance**

RayJob = RayCluster + Job

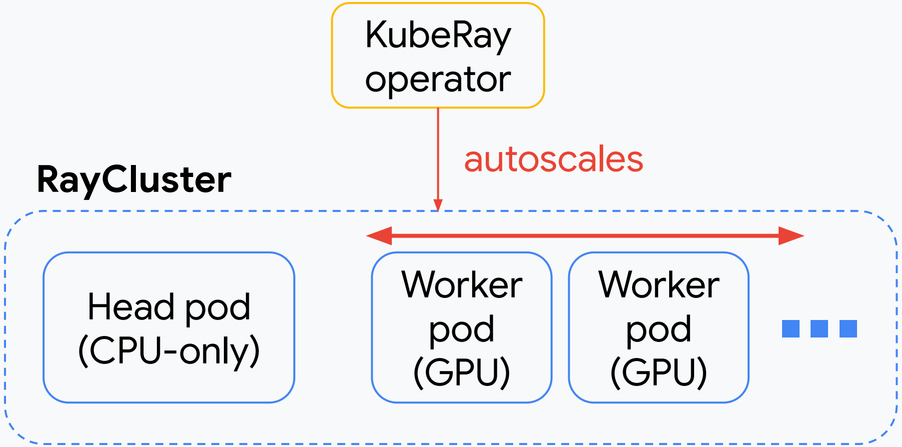
- Submits job on cluster
- RayCluster can be **recycled**

RayService = RayCluster + Serve

- Delpoys Ray Serve on cluster
- Supports **in-place updates**
- **Zero downtime** upgrades
- **High availability**

Ray Autoscaler on KubeRay

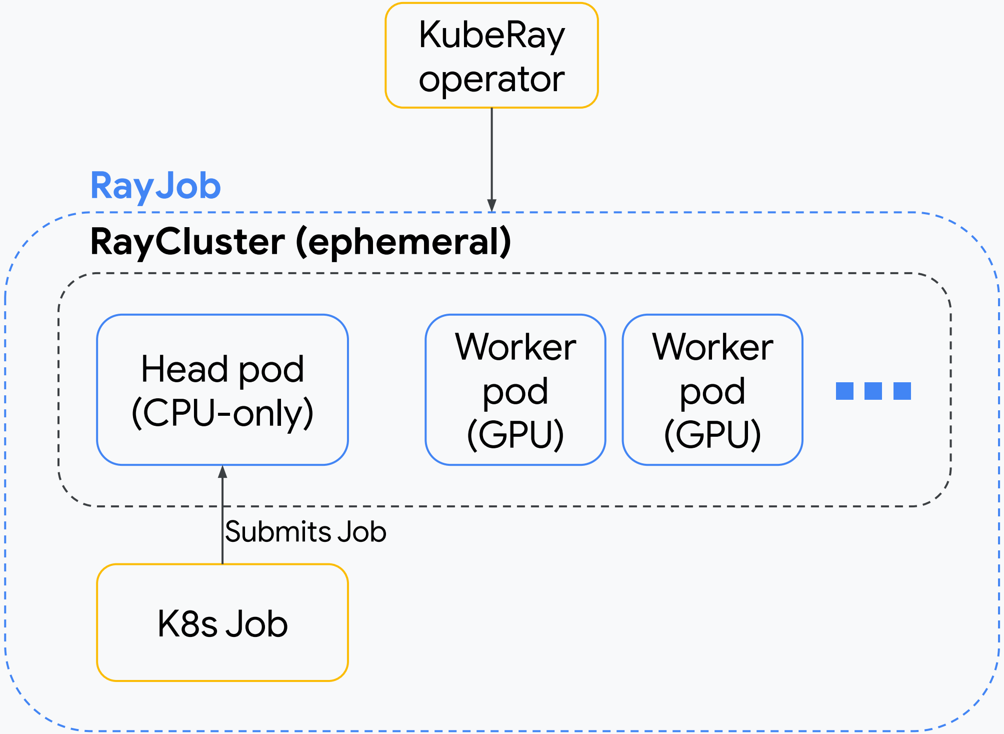
- Integrated with Ray's autoscaler to **scale based on workload demand**
- **Significant cost saver** ($$$)



Google Cloud

RayJob-CRD

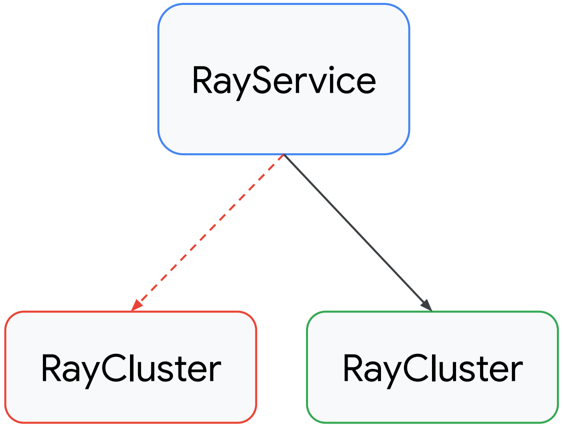
- Cluster **only exists when it is needed** by the Job
- Conserves resources and **reduces cost** ($$$), e.g. for batch jobs



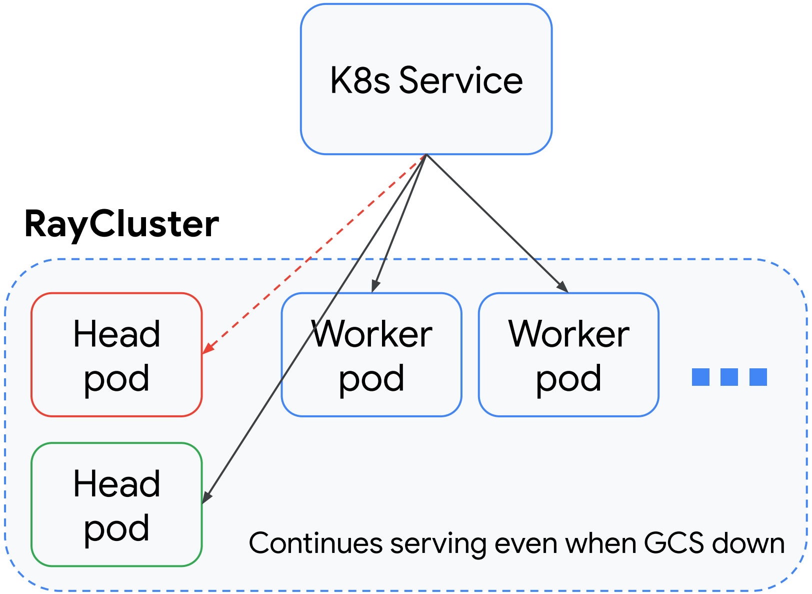
Google Cloud

RayService stability features

Zero-downtime



High-Availability + GCS fault tolerance

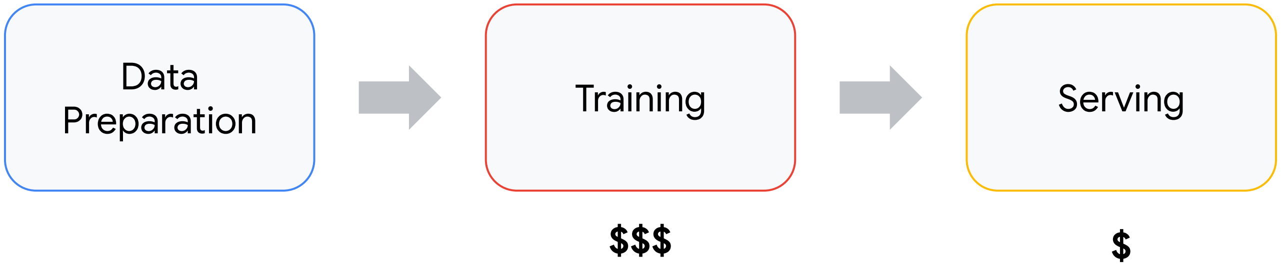


Learnings running KubeRay on GKE

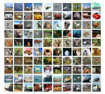
04

Google Cloud

Traditional ML model lifecycle



Traditional ML model lifecycle

Training	Serving/Inference
 → Data Ingestion and Preprocessing → Model Training	 → Model Serving → "cat"
Infrequent	Continuous
Batch	Real-time
Millions/Billions of samples in parallel	A few sample at a time,
Can Saturate a GPU (higher utilization)	Cannot saturate a GPU (lower utilization)
Cost-optimization: **Spot VMs with GPUs**	Cost-optimization: Either **Multi-Instance GPUs or Time-Sharing GPUs**

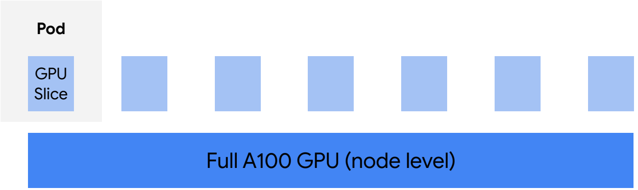
Google Cloud

[Using Spot VMs with GPU node pools]() and [Running multi-instance GPUs on GKE]()

GPU sharing

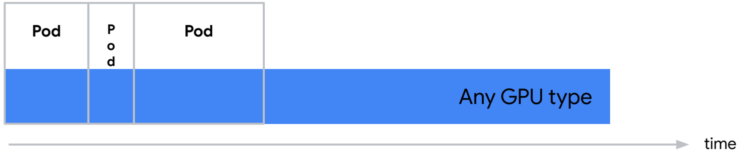
Multi-instance GPUs

- Partition one A100 GPUs into up to 7 slices
- Pod configuration keeps the same
- Use cases
  - Hardware isolation from other containers on the same physical GPU
  - **Predictable throughput** and latency for parallel workloads



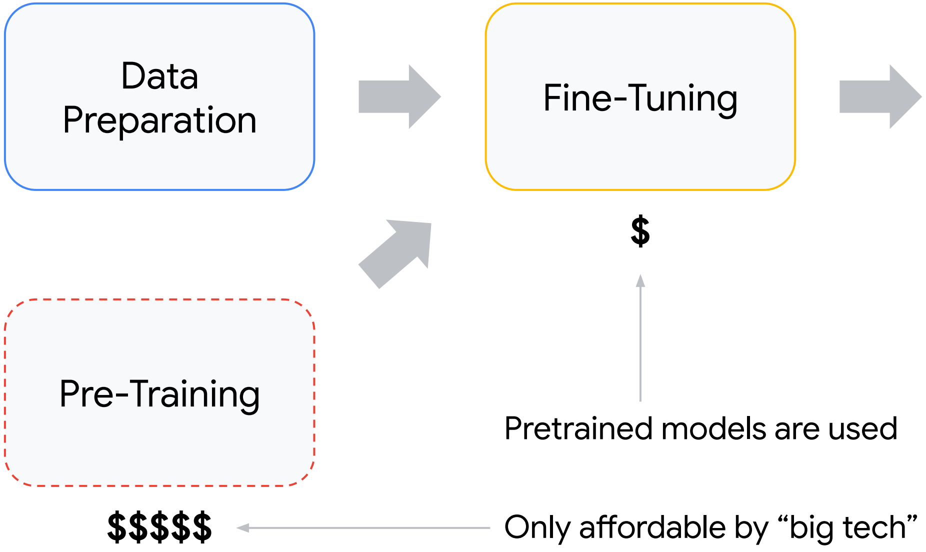
Time-sharing GPUs

- Allow to share any GPU between workloads
- Request fractional GPU units. Setup in nodepool
- Pod configuration keeps the same
- Use cases
  - Workloads with low GPU requests
  - **Burstable GPU workloads**
  - Rendering
  - Inference
  - Small-scale machine learning model training



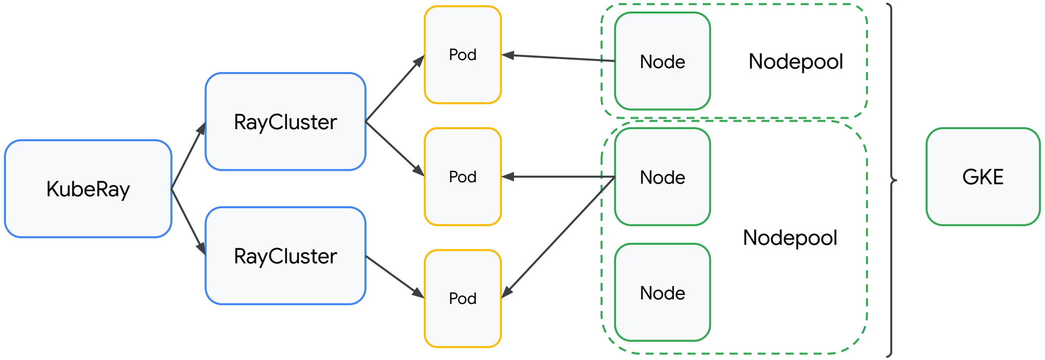
[Time-sharing GPUs on GKE](#) and [Request limits for time-shared GPUs](#)

LLM model lifecycle



KubeRay Autoscaling

- Adjust based on **dynamic load**
- **Multi-tenancy** to increase utilization
- Allows **heterogeneous compute**, e.g. different GPUs, Spot, ...
- Nodepools can have **different autoscaling profiles** and **location policies**



KubeRay Autoscaling Latency

- **GPU nodes take longer** to be ready for workloads
- Ray has **huge container images** (~10GB)

Google Cloud

KubeRay Autoscaling Latency

- **GPU nodes take longer** to be ready for workloads
- Ray has **huge container images** (~10GB)

- **Overprovision**, use [pause/balloon pods]
- Duplicate images to Google's Artifact Registry to **leverage image streaming**

Google Cloud

Day 2 Operations

- Ray does **not log to stdout** ⇒ **add sidecar**, e.g. fluent-bit
- **Monitor via Prometheus**, e.g. Managed Prometheus
- Mark pods as “safe to evict”

Try it out!

Demo of Ray incl. Monitoring on GKE:
https://github.com/GoogleCloudPlatform/ai-on-gke/tree/main/ray-on-gke
KubeRay serving StableDiffusion on GKE (simple example):
https://github.com/trevex/kuberay-example
Great blog post:
https://cloud.google.com/blog/products/containers-kubernetes/use-ray-on-kubernetes-with-kuberay

Ray on Vertex AI in Preview:
https://cloud.google.com/vertex-ai/docs/open-source/ray-on-vertex-ai/overview

Thank you!
Questions?

Niklas Voss
Google Cloud

Google Cloud