KubeCon | CloudNativeCon

North America 2024

KubeCon | CloudNativeCon

North America 2024

Advanced Model Serving Techniques with Ray & Kubernetes

Andrew Sy Kim (Google), Kai-Hsun Chen (Anyscale)

Andrew Sy Kim

Software Engineer at Google, working on Google Kubernetes Engine (GKE).

Kubernetes maintainer and active contributor since 2017.

More recently maintaining KubeRay and helping GKE customers be successful with Ray!

Kai-Hsun Chen

Software Engineer at Anyscale, working on Ray Core team.

Maintaining KubeRay and contributing to Ray Compiled Graphs and Ray Core.

AI is all around you

Uber Spotify samsara Pinterest OpenAI NIANTIC NETFLIX

LinkedIn instacart DOORDASH cohere ANT GROUP Canva reddit

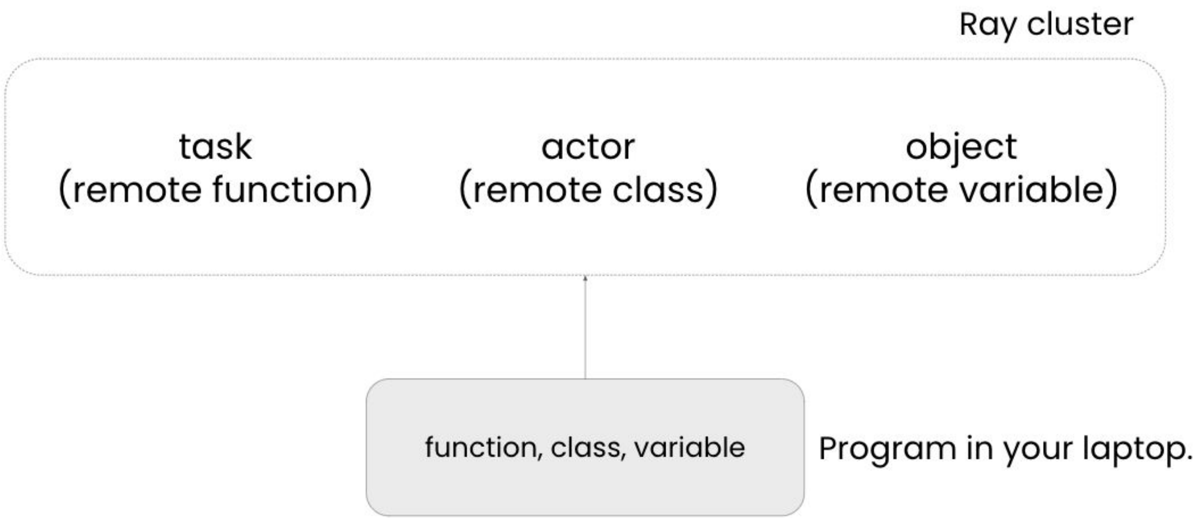
ROBLOX coinbase Adobe ByteDance shopify

https://raysummit.anyscale.com/flow/anyscale/raysummit2024/landing/page/eventsite

Ray Core: Infinite laptop

Ray Core enables users to program in a distributed system
as if they were working on their laptop!

RAY



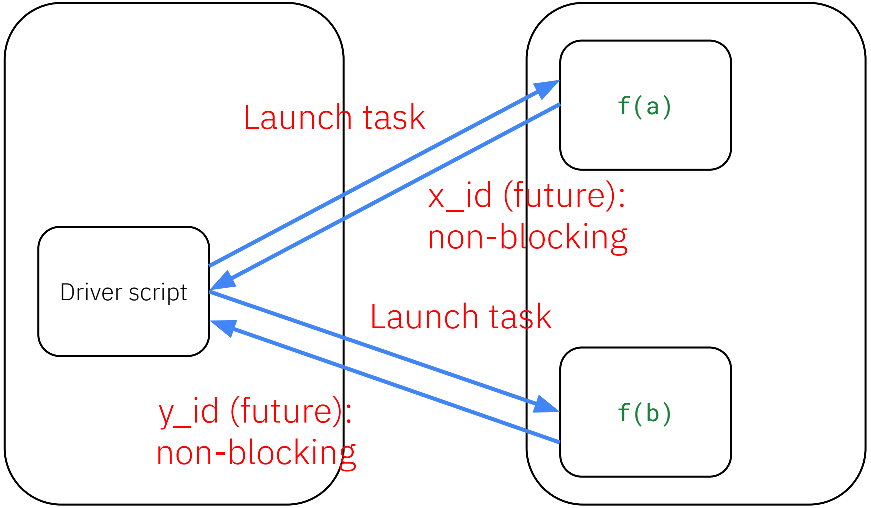
Ray Core example

Convert a function to a Ray task

```
@ray.remote
def f(x):
    # compute ... for 1s
    return r

x_id = f.remote(a)
y_id = f.remote(b)
ray.get([x_id, y_id])
```

1 sec

A synchronous operation to get remote objects

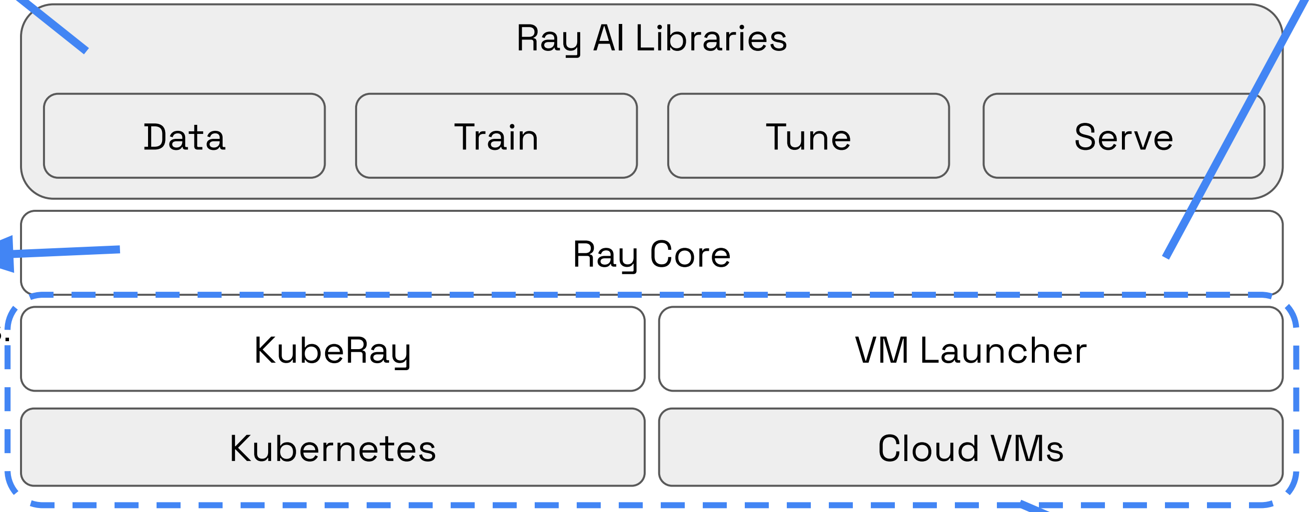


Ray: A “unified” open-source compute framework

Easy-to-use: parallelize your workloads with a few lines of code changes.

Versatile: Covers the end-to-end of the ML lifecycle.

Heterogeneous: Supports different hardware accelerators.

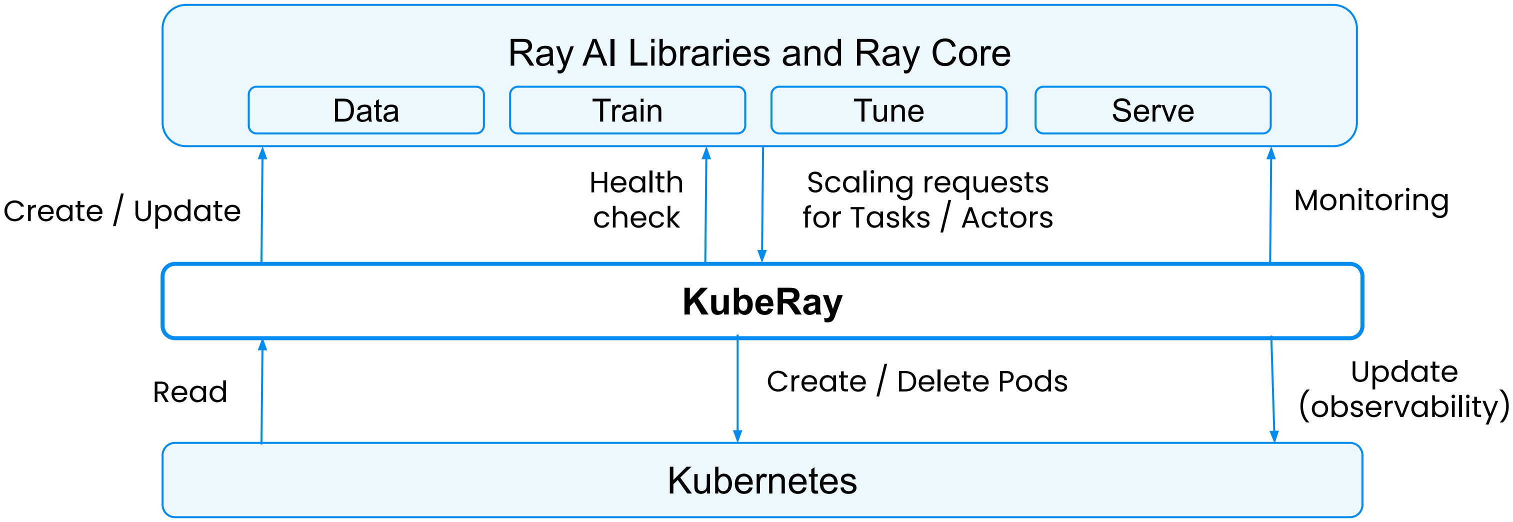


Portability: Deploy everywhere

KubeRay: The solution for open source Ray on Kubernetes

- KubeRay enables **data/ML scientists** to focus on computation while **infra engineers** concentrate on Kubernetes.

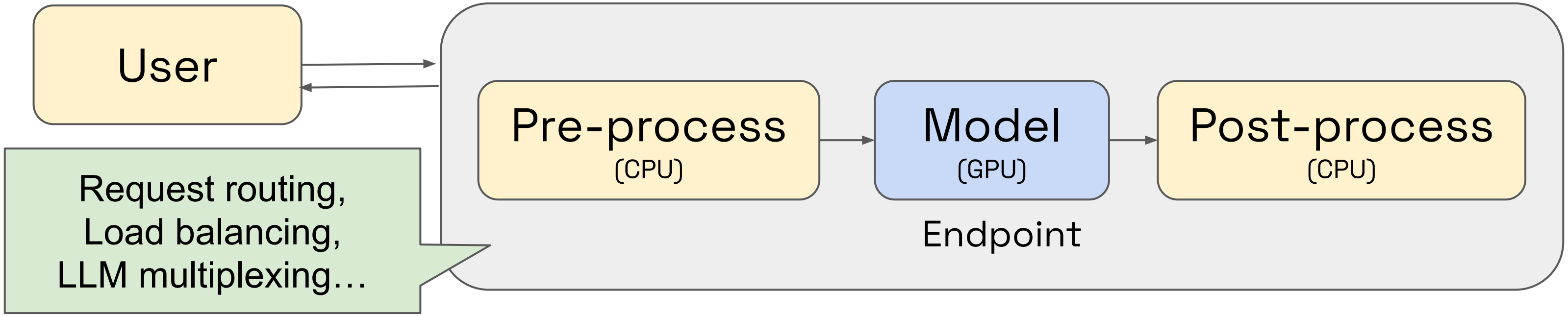
data/ML scientists: Develop Python scripts.



infra engineers: Integrate KubeRay with Kubernetes ecosystem tools, e.g. Prometheus, Grafana, and Nginx.

What is online inference?

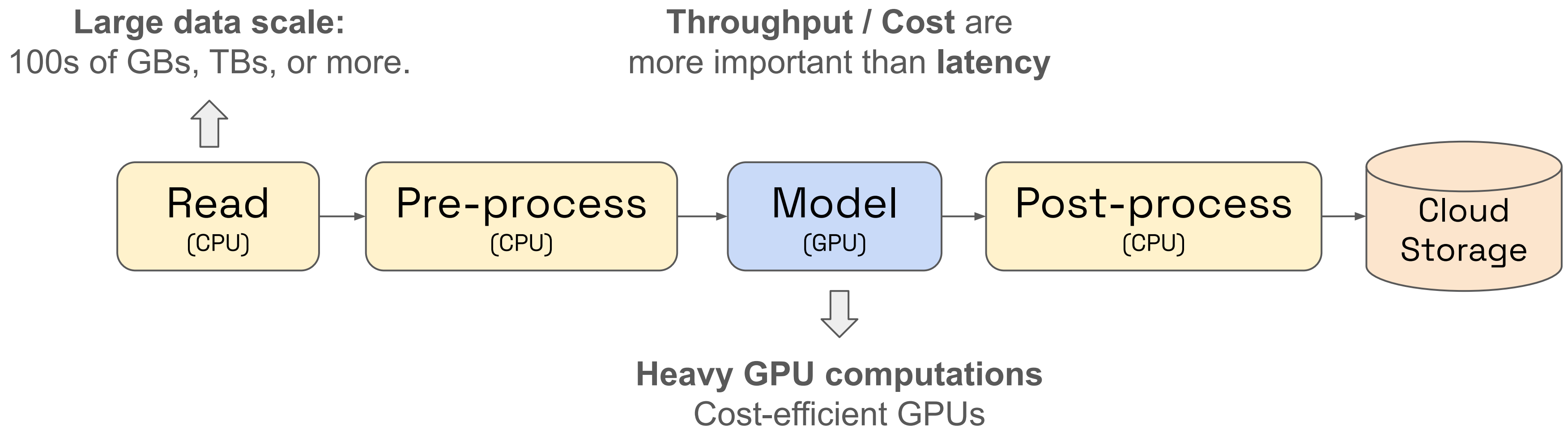
Maximize **reliability** and minimize **latency**



High-end GPUs

What is batch inference?

Batch inference workloads generate model outputs from a large input dataset



What is batch inference?

Workload / Modality	**Downstream Use Case**
Text Embedding Generation	RAG, vector DB uses
Image Embedding Generation	Training, Search/Retrieval
Image Model Batch Inference	Metadata Tagging, Classification, Segmentation, etc.
LLM Batch Inference	Summarization, Tagging, Sentiment Analysis, Keyword Extraction

Online Inference with Ray

Challenges w/ Online Inference

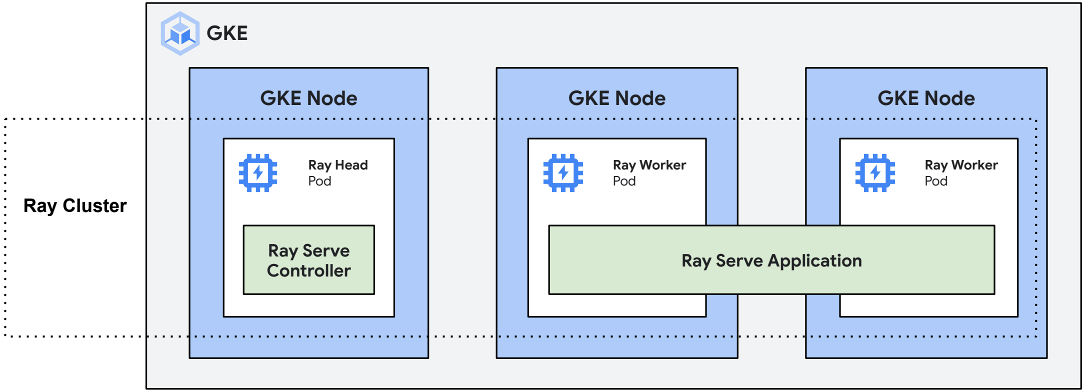
- **Deployment complexity:** Updating models with minimal downtime is challenging, especially with large model sizes. Fast deployment of models is important for prototyping.
- **Framework complexity:** There are many frameworks available for serving models (PyTorch, Tensorflow, etc).
- **GPU performance / utilization:** Price-to-performance and efficient utilization of GPUs is key to scaling online inference

Ray Serve

Ray Serve is a scalable model serving library for building online inference APIs.

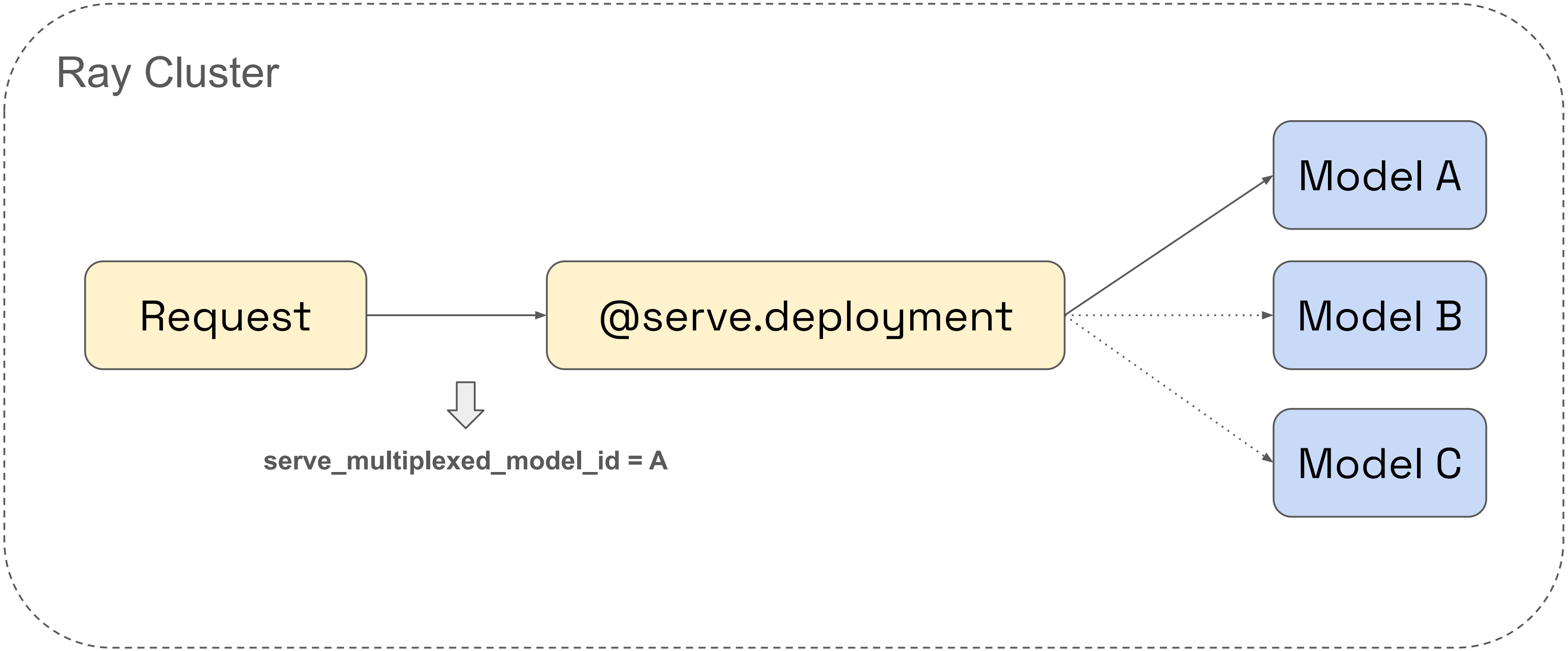
It's advantage over other frameworks include:

- **Fast prototyping** with easy on-ramp from laptop to distributed cluster
- **Simple Python APIs**, friendly to ML Engineers / Data Scientists, framework agnostic
- **Advanced capabilities:** model composition, model multiplexing, request batching, fractional GPU scheduling, etc.



```python
import requests
from starlette.requests import Request
from typing import Dict

from transformers import pipeline

from ray import serve

# 1: Wrap the pretrained sentiment analysis model in a Serve deployment.
@serve.deployment
class SentimentAnalysisDeployment:
    def __init__(self):
        self._model = pipeline("sentiment-analysis")

    def __call__(self, request: Request) -> Dict:
        return self._model(request.query_params["text"])[0]

# 2: Deploy the deployment.
serve.run(SentimentAnalysisDeployment.bind(), route_prefix="/")

# 3: Query the deployment and print the result.
print(
    requests.get(
        "http://localhost:8000/", params={"text": "Ray Serve is great!"}
    ).json()
)
# {'label': 'POSITIVE', 'score': 0.9998476505279541}
```

Model Multiplexing



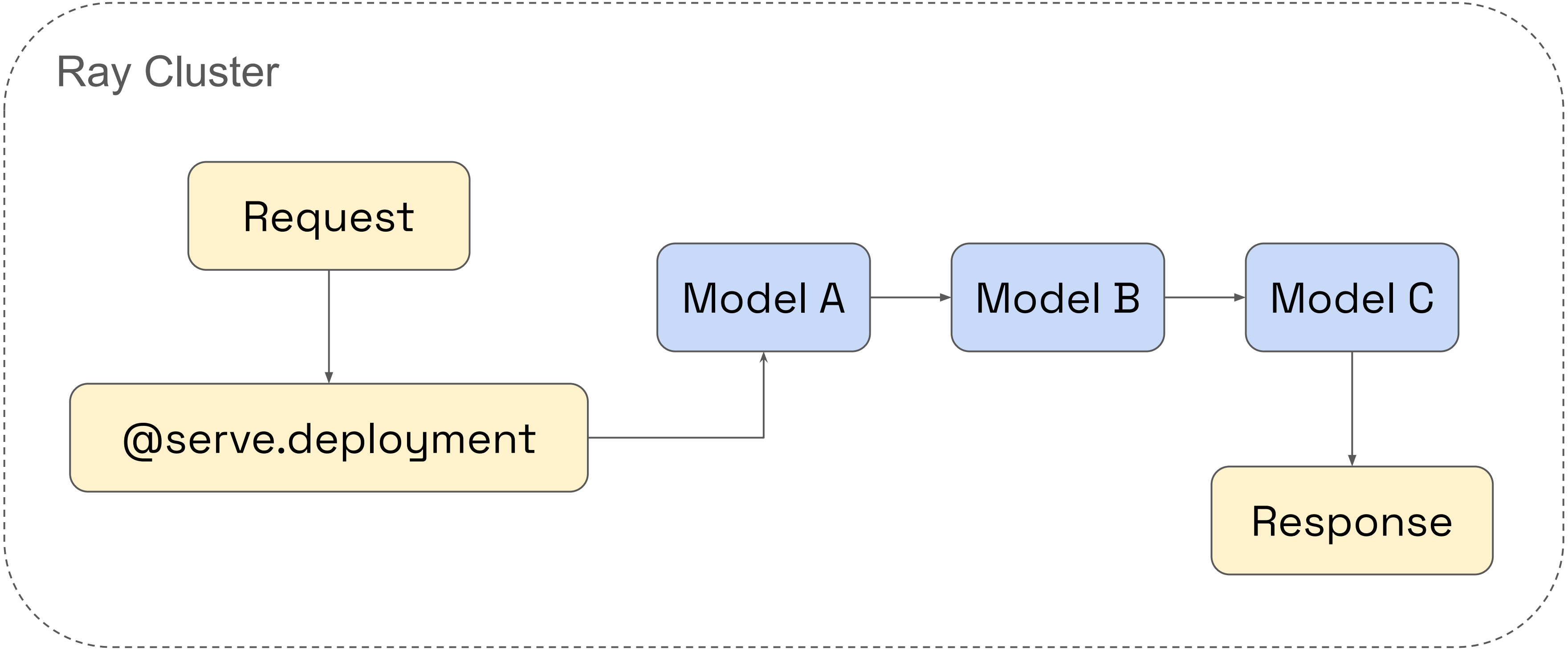
Model Multiplexing


```python
from ray import serve
import asyncio
from google.cloud import storage
import torch
import starlette


@serve.deployment
class ModelInferencer:
    def __init__(self, bucket_name: str):
        self.bucket_name = bucket_name
        self.storage_client = storage.Client()

    @serve.multiplexed(max_num_models_per_replica=3)
    async def get_model(self, model_id: str):
        bucket = self.storage_client.bucket(self.bucket_name)
        blob = bucket.blob(f"{model_id}/model.pt")

        # Download the model into memory
        model_bytes = await asyncio.to_thread(blob.download_as_bytes)
        return torch.load(model_bytes)


    async def __call__(self, request: starlette.requests.Request):
        model_id = serve.get_multiplexed_model_id()
        model = await self.get_model(model_id)
        return model.forward(torch.rand(64, 3, 512, 512))


entry = ModelInferencer.bind("your-bucket-name")
```

```
gs://my_bucket/1/model.pt
gs://my_bucket/2/model.pt
gs://my_bucket/3/model.pt
gs://my_bucket/4/model.pt
...
```

```python
import requests

resp = requests.get(
    "http://localhost:8000",
     headers={"serve_multiplexed_model_id": "1"}
)
```

Model Composition



Model Composition

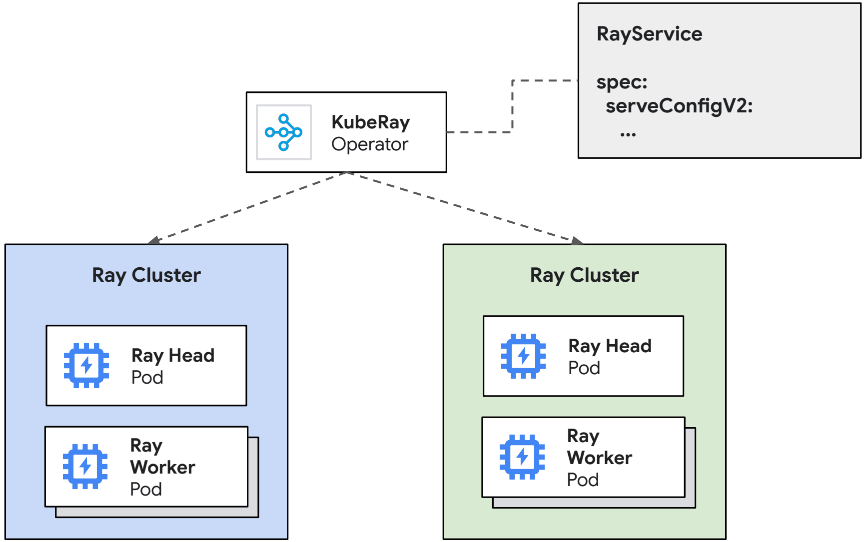

```python
from ray import serve
from ray.serve.handle import DeploymentHandle, DeploymentResponse

@serve.deployment
class Adder:
    def __init__(self, increment: int):
        self._increment = increment

    def __call__(self, val: int) -> int:
        return val + self._increment

@serve.deployment
class Multiplier:
    def __init__(self, multiple: int):
        self._multiple = multiple

    def __call__(self, val: int) -> int:
        return val * self._multiple
```

```python
@serve.deployment
class Ingress:
    def __init__(self, adder: DeploymentHandle, multiplier: DeploymentHandle):
        self._adder = adder
        self._multiplier = multiplier

    async def __call__(self, input: int) -> int:
        adder_response: DeploymentResponse = self._adder.remote(input)
        # Pass the adder response directly into the multipler (no `await` needed).
        multiplier_response: DeploymentResponse = self._multiplier.remote(
            adder_response
        )
        # `await` the final chained response.
        return await multiplier_response

app = Ingress.bind(
    Adder.bind(increment=1),
    Multiplier.bind(multiple=2),
)

handle: DeploymentHandle = serve.run(app)
response = handle.remote(5)
assert response.result() == 12, "(5 + 1) * 2 = 12"
```

KubeRay RayService CRD

RayService is a bundle of a RayCluster and Ray Serve applications.

```yaml
apiVersion: ray.io/v1
kind: RayService
metadata:
  name: llama-3-8b
spec:
  serveConfigV2: |
    applications:
    - name: llm
      route_prefix: /
      import_path:  ray-operator.config.samples.vllm.serve:model
      deployments:
      - name: VLLMDeployment
        num_replicas: 1
        ray_actor_options:
          num_cpus: 8
      runtime_env:
        working_dir: "https://github.com/ray-project/kuberay/archive/master.zip"
        pip: ["vllm==0.5.4"]
        env_vars:
          MODEL_ID: "meta-llama/Meta-Llama-3-8B-Instruct"
          TENSOR_PARALLELISM: "2"
          PIPELINE_PARALLELISM: "1"
  rayClusterConfig:
    ...
```



Online inference example with vLLM

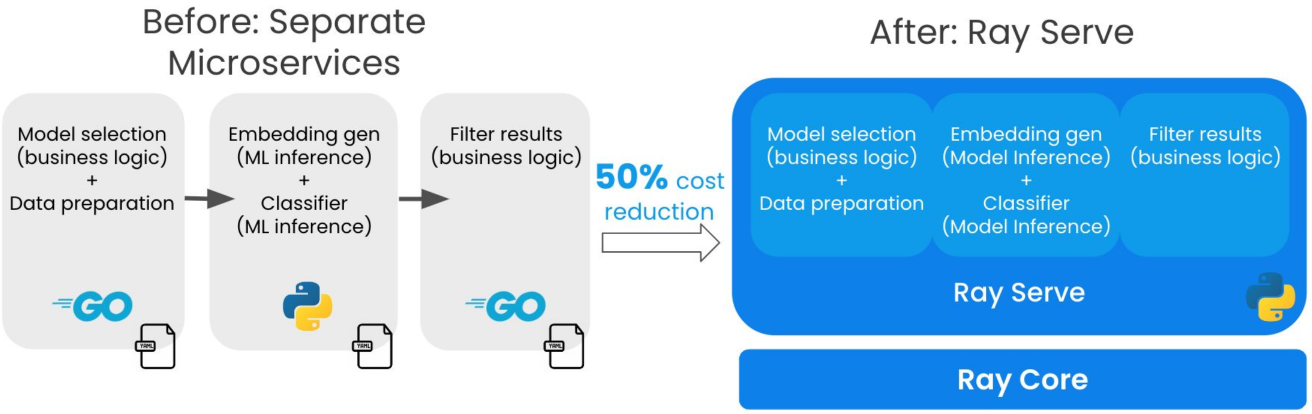

```python
@serve.deployment(name="VLLMDeployment")
@serve.ingress(app)
class VLLMDeployment:
    def __init__(
        self,
        engine_args: AsyncEngineArgs,
        response_role: str,
        lora_modules: Optional[List[LoRAModulePath]] = None,
        chat_template: Optional[str] = None,
    ):
        self.openai_serving_chat = None
        self.engine_args = engine_args
        self.response_role = response_role
        self.lora_modules = lora_modules
        self.chat_template = chat_template
        self.engine = AsyncLLMEngine.from_engine_args(engine_args)
```

```python
def build_app(cli_args: Dict[str, str]) ->
serve.Application:
    parsed_args = parse_vllm_args(cli_args)
    engine_args =
AsyncEngineArgs.from_cli_args(parsed_args)
    engine_args.worker_use_ray = True

    return VLLMDeployment.bind(
        engine_args,
        parsed_args.response_role,
        parsed_args.lora_modules,
        parsed_args.chat_template,
    )
```

```python
model = build_app(
    {"model": os.environ['MODEL_ID'], "tensor-parallel-size": os.environ['TENSOR_PARALLELISM'],
"pipeline-parallel-size": os.environ['PIPELINE_PARALLELISM']})
```

End-User Example

samsara



“Introducing Ray Serve dramatically improved our production ML pipeline performance, equating to a **~50% reduction in total ML inferencing cost per year** for the company.”

ref: https://www.samsara.com/blog/building-a-modern-machine-learning-platform-with-ray

KubeCon | CloudNativeCon

North America 2024

Offline Inference with Ray

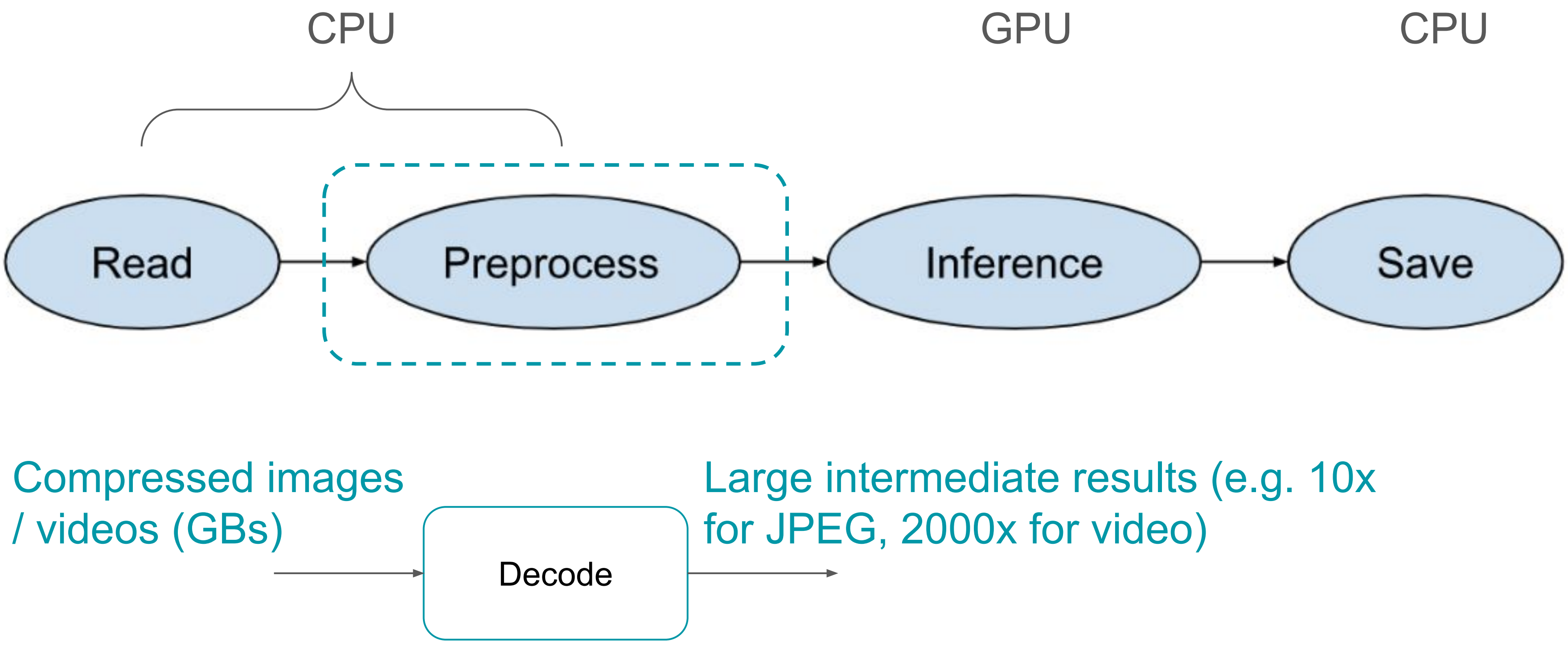
Challenges w/ Offline Inference: Part 1

- **Heterogeneous workloads:** It's common for a batch inference workload to consist of CPU tasks, e.g., preprocessing, and GPU tasks, e.g., model inference.

- **Significant memory to buffer intermediate results:** For example, decoding compressed images or videos generates large intermediate results.

- **Parallelism across multiple nodes:** Offloading CPU-intensive preprocessing to multiple CPU nodes instead of using GPU nodes to increase throughput.

A typical batch inference job



How to handle large intermediate results?

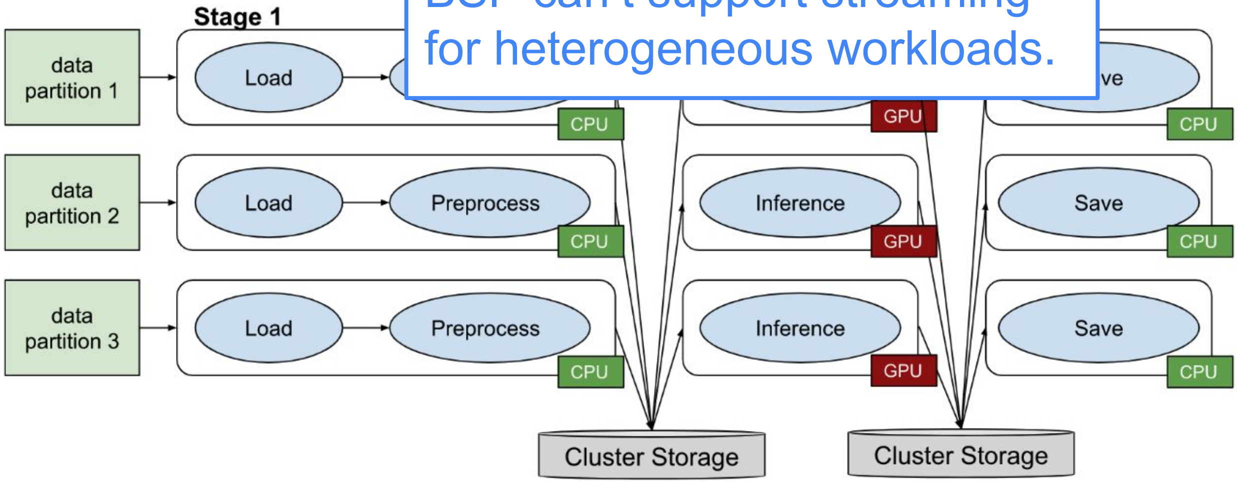
- **Naive solution:** Write intermediate results to disk or cloud storage; however, this could add several minutes of overhead.

- **Stream intermediate data** through cluster memory to avoid the overhead of writing to disk or cloud storage.

Bulk synchronous parallel (BSP) framework

* BSP (e.g. MapReduce, Spark)

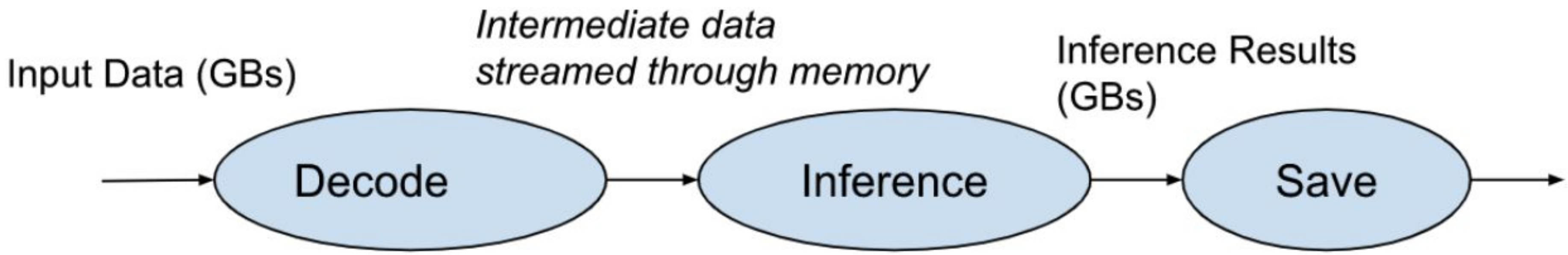
- A stage can't consist of both CPU and GPU workloads at the same time.
- A stage can only start once the previous stage has finished.



BSP can't support streaming for heterogeneous workloads.

Ray Data: Streaming Execution

Avoid the overhead of writing to disk or cloud storage!



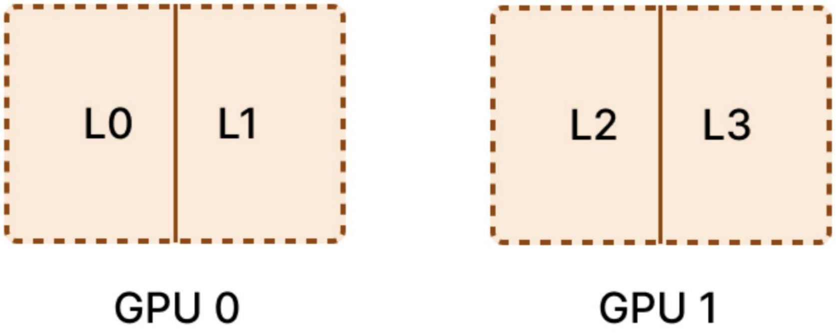
Challenges w/ Offline Inference: Part 2

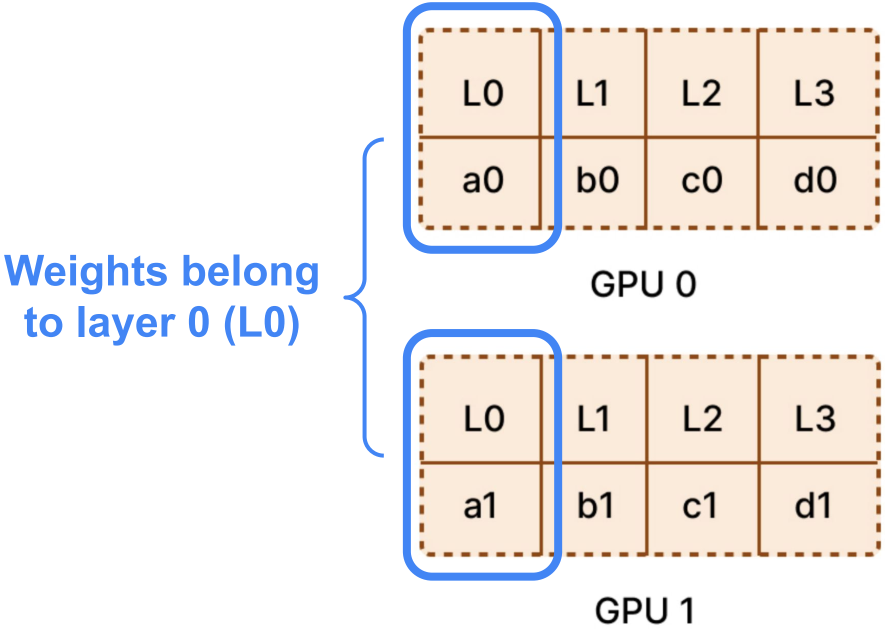
- **Model sharding** becomes increasingly important
  - **Models grow larger:** For example, it's hard for a single GPU node to serve Llama 3.1 405B. (fp16 → 810 GB).
  - **Cost:** For batch inference, cost is more important than latency. Model sharding enables the use of low-end GPUs to reduce costs.

Model Sharding: Pipeline Parallelism & Tensor Parallelism

Pipeline parallelism splits up a model **layer-wise** across multiple GPUs.

Tensor parallelism splits the weights from the **same layer** across different GPUs.

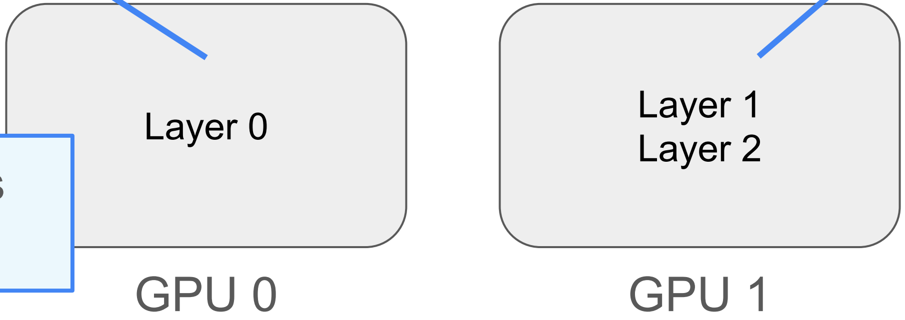




Challenges in Pipeline Parallelism

A larger layer occupies a single GPU.

Several smaller layers reside together on a GPU

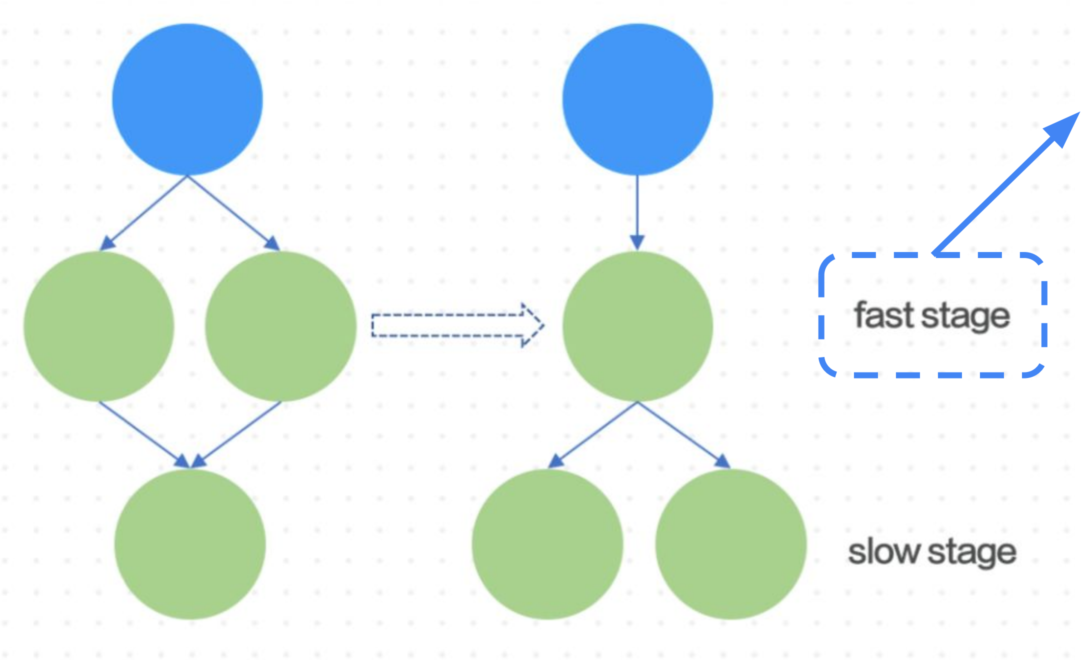


Elastic autoscaling is important!

Different stages have varying computation requirements, which are hard to predict!

Ray Autoscaling

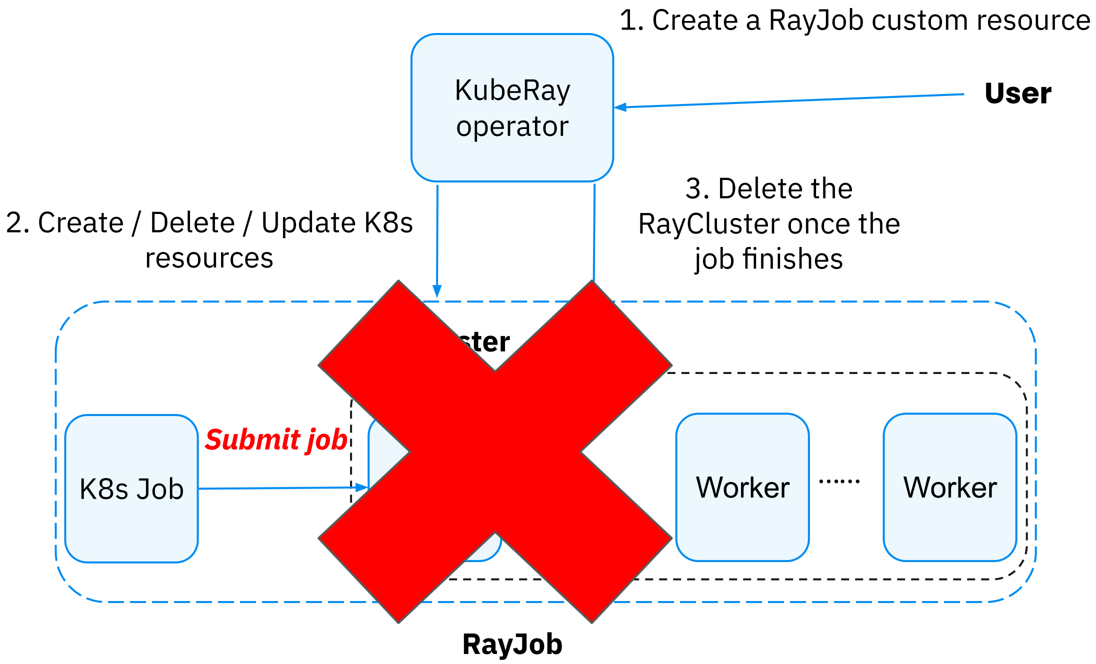
Ray Autoscaling is pretty flexible, allowing each stage to autoscale independently.



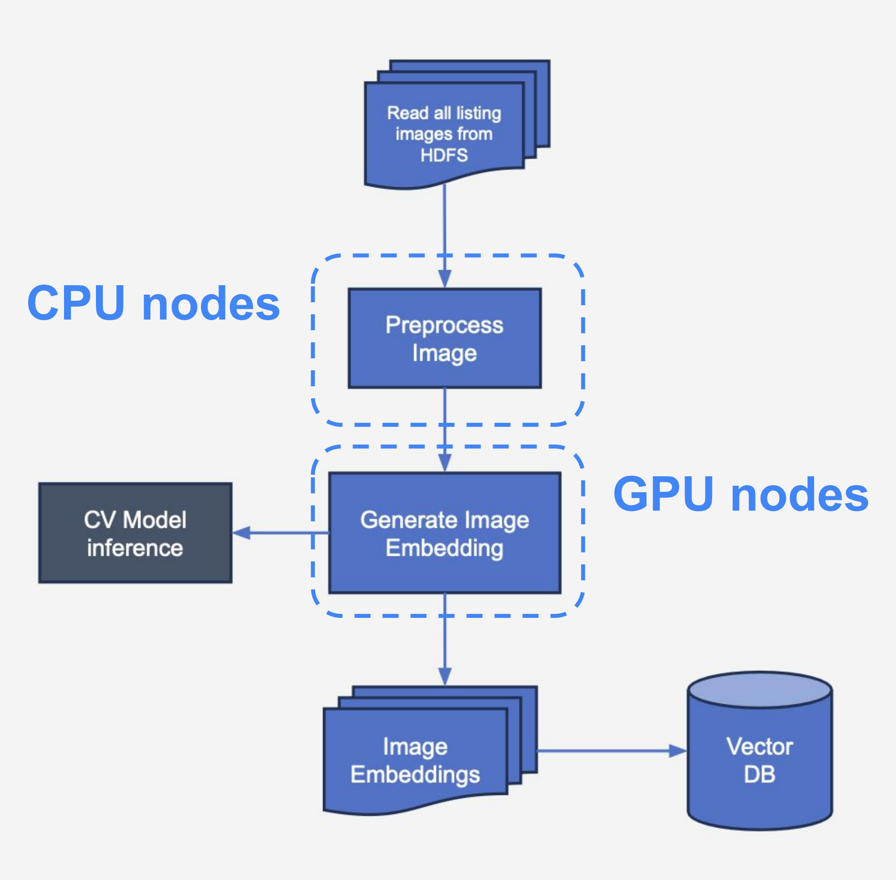
Faster stages can scale down and release resources for slower stages.

KubeRay RayJob CRD: Productionize batch workloads

- RayJob = RayCluster + K8s Job
- Automatic cleanup of compute resources after a job finishes.
- RayCluster Autoscaling
- Support advanced scheduling with Kueue / Volcano / Yunikorn



End-user example: eBay



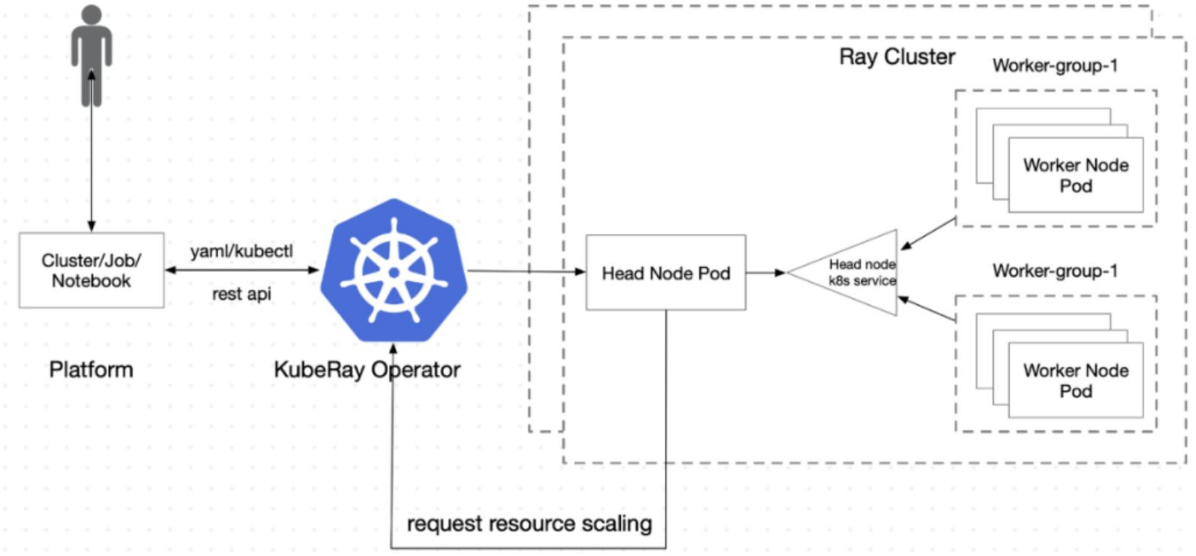
Increase GPU utilization by 4x with Ray Data streaming execution, Ray fractional GPUs, and Ray/KubeRay autoscaling.

ref: Yucai Yu, https://youtu.be/5KuTdRq9Zto?si=uI631EFxeTIbo32k

End-user example: ByteDance

KubeCon CloudNativeCon North America 2024

ByteDance scales offline inference with multi-modal LLMs to 200 TB data by Ray Data and KubeRay



ref: Wanxing Wang, https://mp.weixin.qq.com/s/R_N1AbQuMF3q186MQQLeBw

KubeCon CloudNativeCon

North America 2024

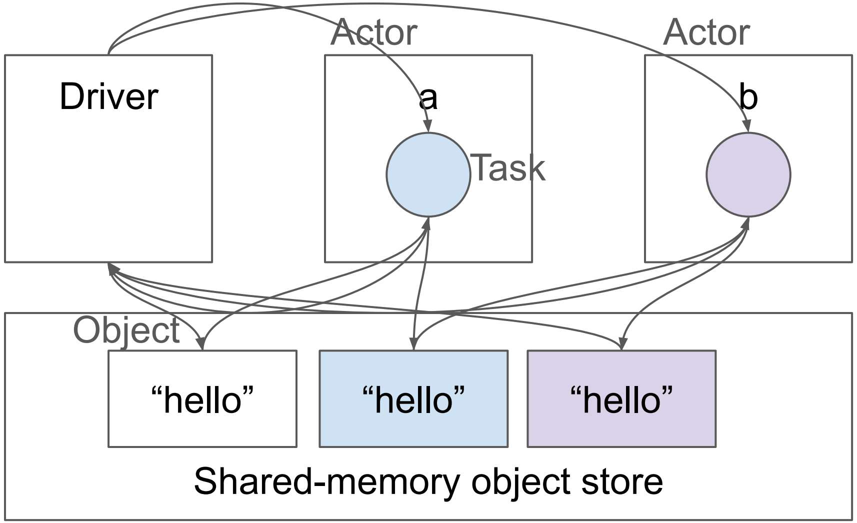
Ray Compiled Graphs

Ray Compiled Graphs are **10-20x faster** than Ray tasks* and support **GPU-native** communication (e.g., NCCL).

*for static task graphs

Default execution with Ray Core

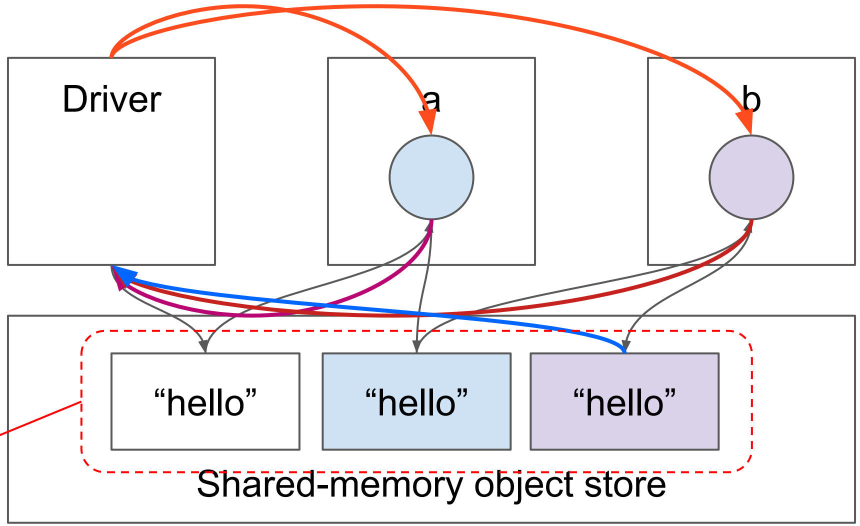
```
import ray

@ray.remote
class EchoActor:
  def echo(self, msg):
    return msg

a = EchoActor.remote()
b = EchoActor.remote()


msg_ref = a.echo.remote("hello")
msg_ref = b.echo.remote(msg_ref)
print(ray.get(msg_ref))
# hello
```



Driver RPC is a **bottleneck**

Default execution with Ray Core

Driver tells Ray tasks where is their input arguments



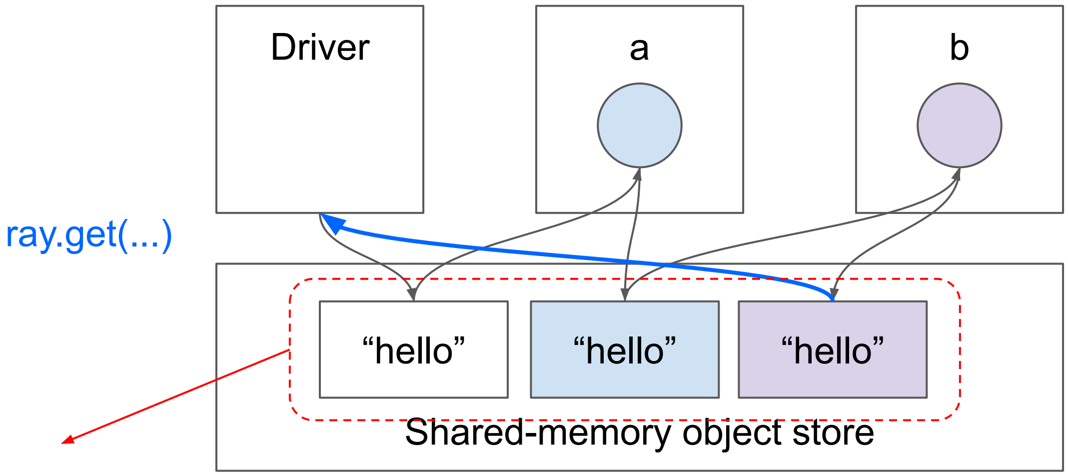
ray.get(...)

The Ray task tells the driver where the task output is.

Need to allocate memory for each execution.

Ideal execution with Ray Compiled Graphs

A Ray task reads inputs from the same memory address set during compilation. (**Reduce RPCs from driver to Ray tasks**)

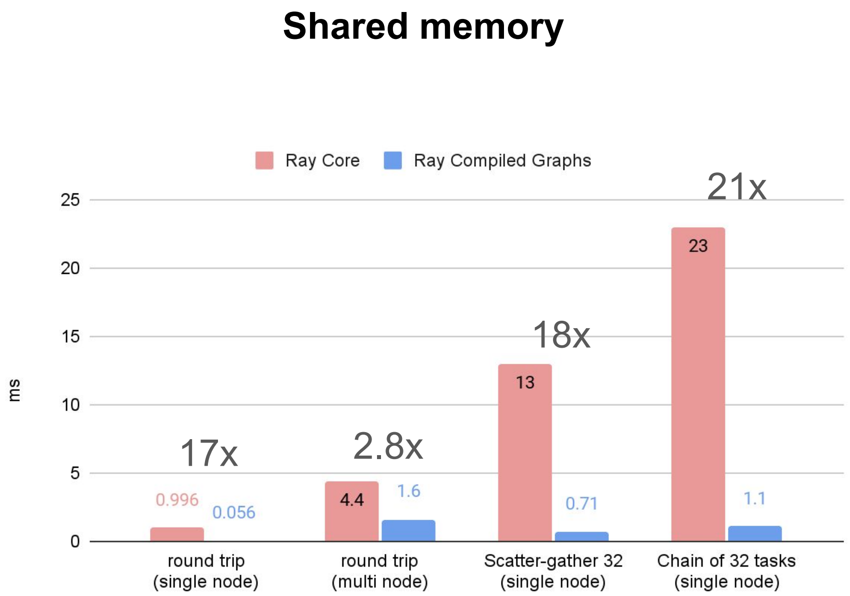


Allocate memory **once** during compilation, and then **reuse** it for multiple executions.

A Ray task writes outputs to the same memory address set during compilation. (**Reduce RPCs from Ray tasks to driver**)

Ray Core vs Ray Compiled Graphs

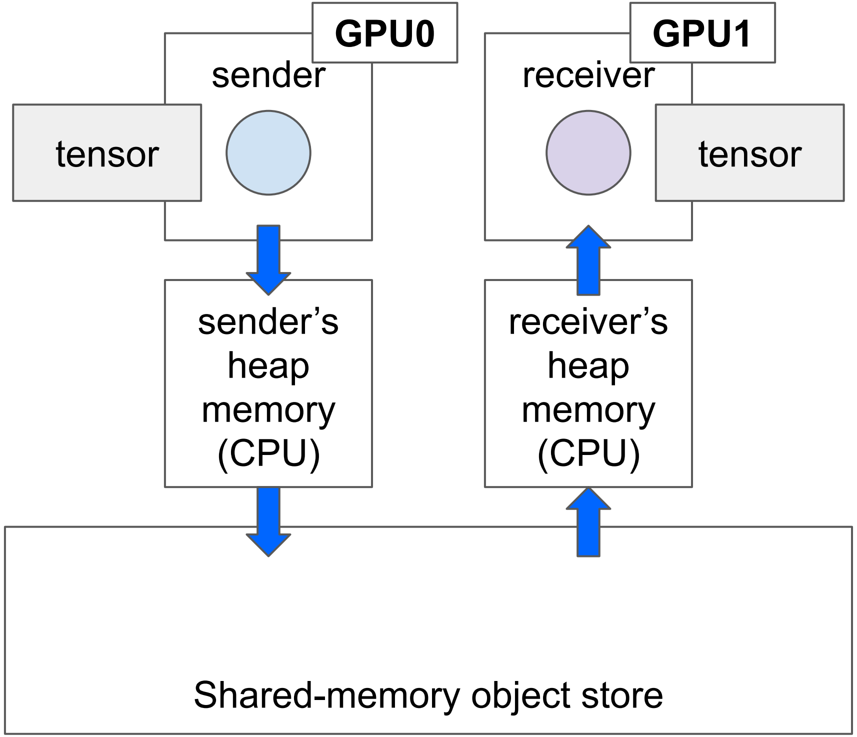
KubeCon CloudNativeCon North America 2024



GPU-GPU transfers with default Ray Core

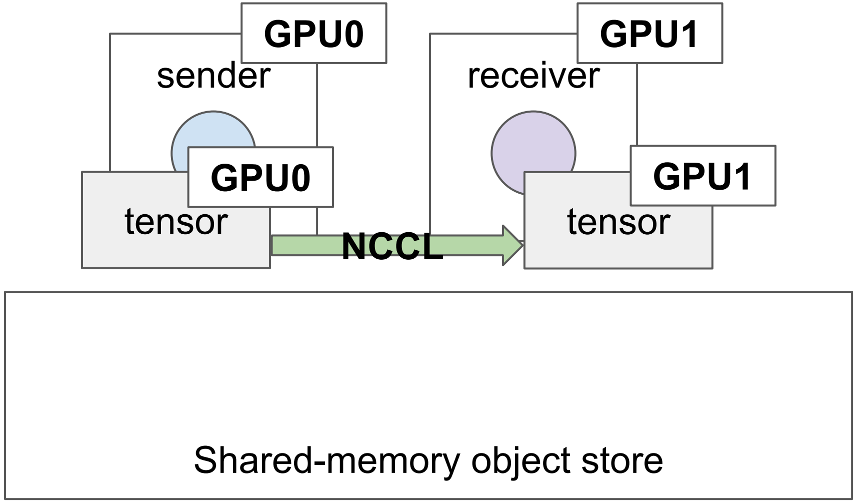
Transfer a GPU tensor from one actor to another actor.

Multiple data copies!

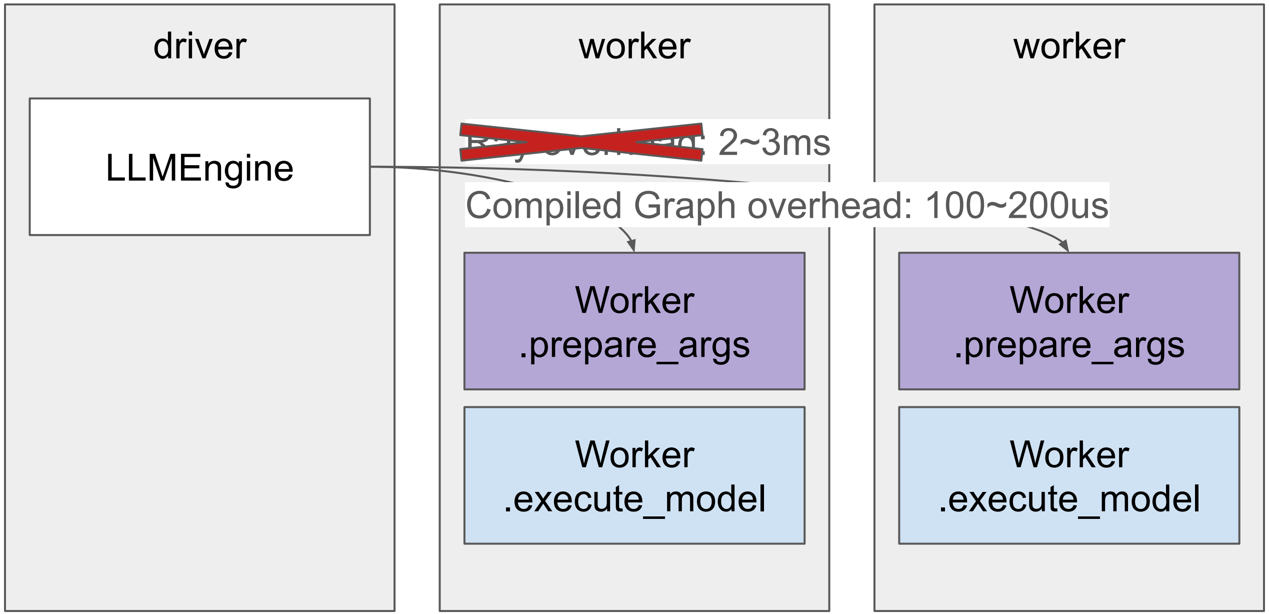


GPU-GPU transfers with Ray Compiled Graphs


```
from ray.experimental.channel.torch_tensor_type
  import TorchTensorType

with ray.dag.InputNode() as inp:
  data = sender.send.bind(inp)
  data = data.with_type_hint(TorchTensorType(
      transport="nccl"
  ))
  dag = receiver.recv.bind(data)

compiled_dag = dag.experimental_compile()
compiled_dag.execute((100, ))
```



Optimized vLLM control plane with Compiled Graphs



TP + PP with Compiled Graphs in vLLM


```python
with InputNode() as input:
    outputs = [input for _ in pp_tp_workers[0]]
    # pp_tp_workers is indexed first by PP stage, then TP rank.
    for pp_stage, tp_group in enumerate(pp_tp_workers):
        # Each PP worker takes in the output of the previous PP worker,
        # and the TP group executes in SPMD fashion.
        outputs = [
            worker.execute_model_spmd.bind(outputs[i])
            for i, worker in enumerate(tp_group)
        ]
        last_pp_stage = len(pp_tp_workers) - 1
        if pp_stage < last_pp_stage:
            outputs = [
                output.with_type_hint(
                    TorchTensorType(transport="nccl"))
                for output in outputs
            ]
    dag = MultiOutputNode(outputs)
compiled_dag = dag.compile()
```

Graph input

TP group

NCCL transfer to the next PP stage

vLLM performance: Compiled Graph vs original architecture

Metric	**Model**	**Parallelism**	**GPU**	**Result**
Latency (online)	Llama2 70B	PP=2, TP=4	A100	**-10~20%**
Throughput (offline)	Llama2 70B	PP=8, TP=1	L4	**+10%**

Performance parity in various other setups:

- A simple Compiled Graph implementation, v.s.
- The continuously improved vLLM implementation

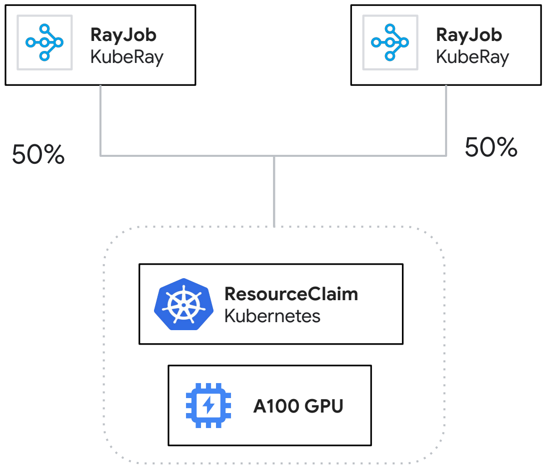
KubeCon CloudNativeCon
North America 2024

Dynamic Resource Allocation with KubeRay

GPU Time Slicing w/ DRA

GPU Time Slicing:

- Each Ray pod gets a slice of GPU time, exclusively
- Pods do not share any GPU resources simultaneously, each Pod gets full access during it’s time slice



GPU Time Slicing w/ DRA

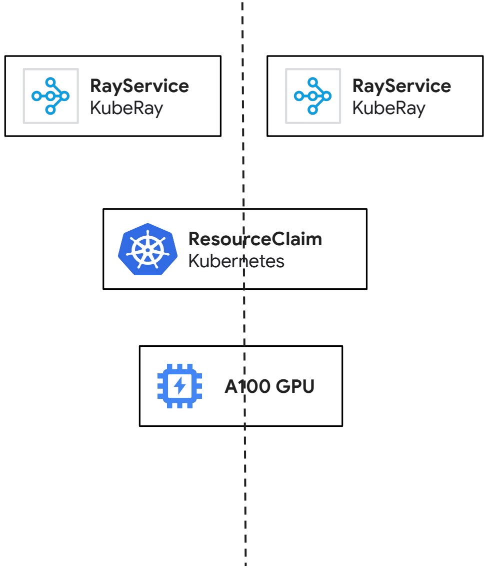
```yaml
---
apiVersion: resource.k8s.io/v1alpha3
kind: ResourceClaimTemplate
metadata:
  name: a100
spec:
  spec:
    devices:
      requests:
      - name: gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["gpu"]
        opaque:
          driver: gpu.nvidia.com
          parameters:
            apiVersion: gpu.nvidia.com/v1alpha1
            kind: GpuConfig
            sharing:
              strategy: TimeSlicing
              timeSlicingConfig:
                interval: Long
```

```yaml
---
apiVersion: ray.io/v1
kind: RayCluster
metadata:
  name: ray-cluster
spec:
  …
  …
  workerGroupSpecs:
  - replicas:1
    template:
      spec:
        containers:
          resources:
            claims:
            - name: a100
        resourceClaims:
        - name: a100
          resourceClaimTemplateName: a100
```

GPU Space Sharing w/ DRA

GPU Space Sharing:

- GPU resources (memory and compute time) are divided among Ray Pods
- Simultaneous consumption of GPU resources, no context switching required



GPU Space Sharing w/ DRA


```yaml
---
apiVersion: resource.k8s.io/v1alpha3
kind: ResourceClaimTemplate
metadata:
  name: a100
spec:
  spec:
    devices:
      requests:
      - name: gpu
        deviceClassName: gpu.nvidia.com
      config:
      - requests: ["gpu"]
        opaque:
          driver: gpu.example.com
          parameters:
            apiVersion: gpu.resource.example.com/v1alpha1
            kind: GpuConfig
            sharing:
              strategy: SpacePartitioning
              spacePartitioningConfig:
                partitionCount: 10
```

```yaml
---
apiVersion: ray.io/v1
kind: RayCluster
metadata:
  name: ray-cluster
spec:
  …
  …
  workerGroupSpecs:
  - replicas:1
    template:
      spec:
        containers:
          resources:
            claims:
            - name: a100
        resourceClaims:
        - name: a100
          resourceClaimTemplateName: a100
```

```python
import requests
from starlette.requests import Request
from typing import Dict

from transformers import pipeline

from ray import serve

# 1: Wrap the pretrained sentiment analysis model in a Serve deployment.
@serve.deployment(ray_actor_options={"num_gpus": 0.5})
class SentimentAnalysisDeployment:
    def __init__(self):
        self._model = pipeline("sentiment-analysis")

    def __call__(self, request: Request) -> Dict:
        return self._model(request.query_params["text"])[0]
```

Fractional GPU scheduling

Leverage DRA and Ray fractional GPU for fine-grain control of GPU consumption

```yaml
apiVersion: ray.io/v1
kind: RayService
metadata:
  name: llama-3-8b
spec:
  serveConfigV2: |
    applications:
    - name: llm
      route_prefix: /
      import_path:  ray-operator.config.samples.vllm.serve:model
      deployments:
      - name: VLLMDeployment
        num_replicas: 1
        ray_actor_options:
          num_cpus: 8
      runtime_env:
        working_dir:
"https://github.com/ray-project/kuberay/archive/master.zip"
        pip: ["vllm==0.5.4"]
        env_vars:
          MODEL_ID: "meta-llama/Meta-Llama-3-8B-Instruct"
          TENSOR_PARALLELISM: "2"
          PIPELINE_PARALLELISM: "1"
  rayClusterConfig:
    ...
```

vLLM Tensor / Pipeline Parallelism

Leverage DRA with Tensor / Pipeline Parallelism to optimize LLM performance, throughput and cost.

Summary

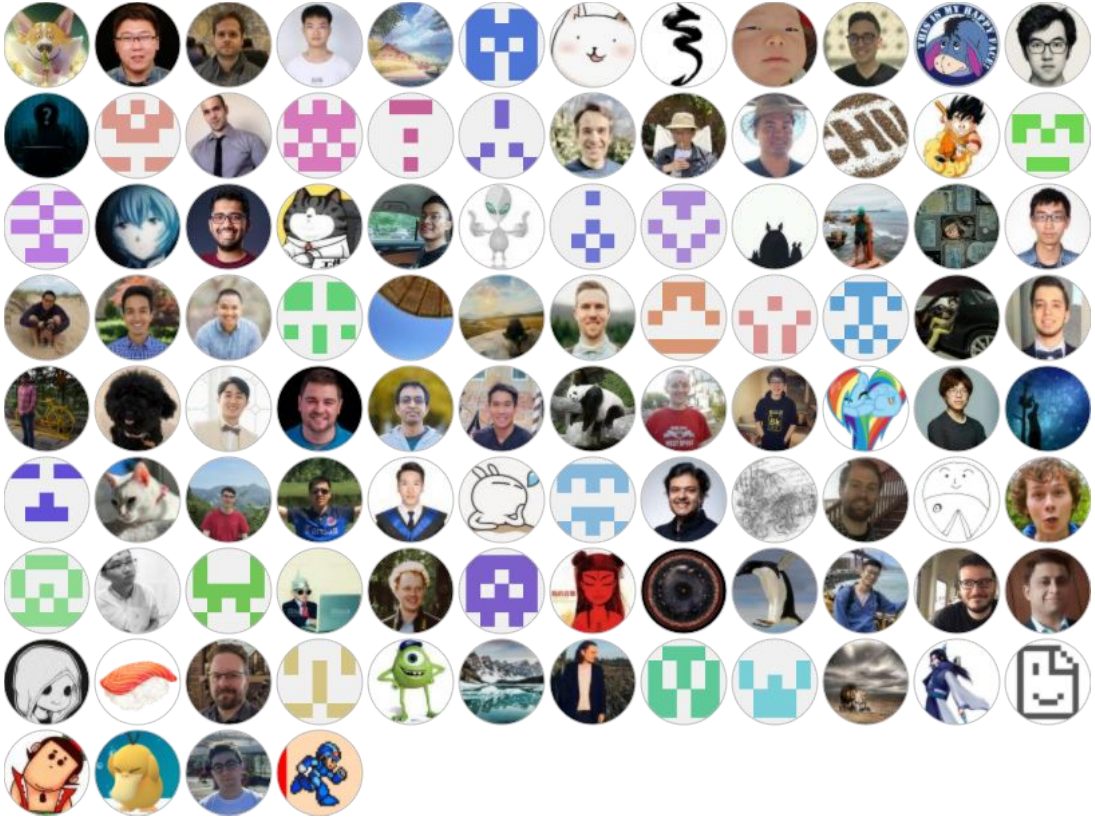
- **Ray** is a unified open-source compute framework for machine learning
- **Ray Serve** is a model serving framework in Ray for building online inference APIs
- **Ray Data** supports batch inference consisting of heterogeneous tasks.
- **Ray Compiled Graphs** are 10-20x faster than Ray tasks.
- **Kubernetes Dynamic Resource Allocation** provides flexibility to configure device specific parameters for optimal performance and utilization
- **KubeRay** enables all the of above in production!

Join the community!

GitHub repo:
ray-project/kuberay

Ray Slack channels:
#kuberay-questions
#kuberay-discuss

OSS community calendar:
https://shorturl.at/obu5G



Stop by the Anyscale/Ray Booth (S43)!

Anyscale

RAY

Booth S43

Speak to our team and learn how Anyscale and Ray can supercharge your AI Platforms!

KubeCon | CloudNativeCon

North America 2024

Q&A

Thank you!