Ray

04 December 2025 11:55

AI Native Computing Stack and Ray Summit 2025

Evolution of Computing eras:

Era	Compute Engine	Core stack/Technologies
Client server & Enterprise computing	Server (Windows/Unix)	Oracle databases etc
Internet and Web services	Virtual Machines (VMs)	LAMP stack (Linux | Apache | MySQL | PHP)
Cloud computing	Container	Microservices, Big data, Kubernetes, MongoDB
AI Era	Ray (Distributed Computer Engine)	AI native computing stack (PyTorch, VLLM, Kubernetes)

Core Drivers demanding the AI Native Computing Stack:

- Heterogeneous hardware
    - CPUs + Accelerators
    - High speed networking
    - Purpose built AI data centres
- Complex, Heterogeneous AI processing Pipelines
    - Data processing
    - Training
    - Serving/Inference
- Non deterministic systems
- Higher iteration velocity

Ray: The AI Compute Engine

- Originated from Reinforcement Learning Research
- Adoption: 5 times more download increase since last year
- Core workloads:
    - Data processing
    - Training
    - Serving
- Evolution stages
    - Stage 1: Classic Neural Network (Moderate Scale)
    - Stage 2: First GenAI (Pre-Training Scaling Law, Accelerated Adoption)
    - Stage 3: Second GenAI (Post training - Reinforcement Learning, Massive Complexity)

AI Infrastructure Stack Layers:

- Top Layer: Training and Inference framework (PyTorch, VLM(Visual - Language model, integrates computer vision and natural language processing))
- Middle Layer: Distributed Compute Engine (Ray)
- Bottom Layer: Container orchestration (Kubernetes)
- Ecosystem: PyTorch Foundation

Challenge of Distributed Inference and Cross stack collaboration:

- Scale of inference
    - As models grow larger, running inference requires a lot more GPU
- Cross stack optimisation
    - Requires optimization across multiple stack levels, from API down to hardware placement
- Fine Grained Placement
    - Optimizing the placement of VLLM (Virtual Large Language Models)
- Cross node parallelism strategies
    - Requires establishing coordination across PyTorch, Ray, Kubernetes and VLLM

Ray | Anyscale

- **The AI Compute Engine**
    - Supports any AI or ML workload
    - Support any data types and model architectures
    - Uses heterogenous GPUs and CPUs with fine grained, independent scaling
    - Fully utilizes every accelerator
    - Scale from your laptop to thousands of GPUs

- **Ray Features**
    - Parallel python code
    - Multi modal data processing
    - Model training
    - Model serving
    - Batch inference
    - Reinforcement Learning
    - Gen AI
    - LLM inference
    - LLM Fine Tuning

- **Ray integrations with python**
    - Tasks

```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]
```

- Actors
    - While tasks are stateless, Ray actors allows you to create stateful workers that maintain their internal state between method calls, when you instantiate a ray actor
        - Ray starts a dedicated worker process somewhere in your cluster
        - The Actors methods run on that specific worker and can access and modify its state
        - The actor executes method calls serially in the order it receives them, preserving consistency

```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
```

- Objects
    - Three main ways to work with objects in Ray
        - Implicit creation: When tasks and actors return values, they are automatically stored in ray's distributed object store
        - Explicit creation: Use ray.put() to directly place objects in the store
        - Passing references: Pass object references to other tasks and actors

```python
import numpy as np

# Define a task that sums the values in a matrix.
@ray.remote
def sum_matrix(matrix):
    return np.sum(matrix)

# Call the task with a literal argument value.
print(ray.get(sum_matrix.remote(np.ones((100, 100)))))
# -> 10000.0

# Put a large array into the object store.
matrix_ref = ray.put(np.ones((1000, 1000)))

# Call the task with the object reference as an argument.
print(ray.get(sum_matrix.remote(matrix_ref)))
# -> 1000000.0
```

- **Ray Libraries**
    - Data: Scalable, framework agnostic data loading and transformation across training, tuning and prediction
    - Train: Distributed multi node and multi core model training with fault tolerance that integrates with popular training libraries
    - Tune: Scalable hyper parameter tuning to optimize model performance
    - Serve: Scalable and programmable serving to deploy models for online interface

- RLLib: Scalable distributed reinforcement learning workloads

Anyscale Unified AI Platform:

- Goal: Productization, Security, Governance
- Three main layers
    - Developer central
        - Workspaces
        - Debugging
        - Tooling
    - Ray Runtime (Optimized Ray Turbo Evolution)
    - Cluster controller (Compute Life Cycle Management)
- Platform innovations
    - Lineage Tracking
    - Anyscale Runtime (2x Faster Data, 7x Higher Throughput Serving)
    - GKE(Google Kubernetes Engine)/AKS(Azure Kubernetes Engine) integration
    - Multi Resource Clouds
    - Global Resource Scheduler: Flexible GPU allocation