TECHNICAL WHITEPAPER

YELLOWDOG

RAYDOG – A DEEP DIVE ON ITS CAPABILITIES AND BENEFITS

1 Introduction

With the increasing pace of change and competition, organisations are looking for tools and workflows that streamline the development and scaling of their machine learning (ML) models and simulations to accelerate time to results.

Ray is an open-source framework that does just that, enabling developers to build and test their models locally on their laptop, and then easily scale to a compute cluster in the cloud without needing to understand or rewrite their code for distributed systems.

RayDog amplifies this capability, utilising the power of YellowDog's fully-featured cloud orchestration platform to harness resources across multiple cloud regions to build clusters for Ray, and scale all the way up to ~10,000 nodes if needed.

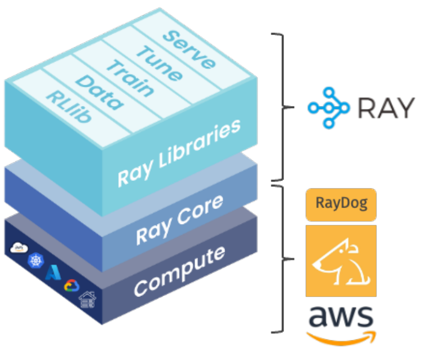
In addition, YellowDog is fast, able to rapidly build clusters to accelerate time to results, and then tear them down just as quickly to minimise cloud costs. Moreover, YellowDog is able to find and leverage Spot instances, and combine them with advanced pre-emption handling to deliver resilient compute within a set budget.

In short, RayDog combines the ease of use of Ray with the power of YellowDog's cloud orchestration Platform to deploy Ray anywhere, at scale, and at low cost.

This document takes a look at the compute requirements and topology options for supporting large scale Ray workloads, and the unique capabilities of RayDog and the YellowDog Platform for supporting them.

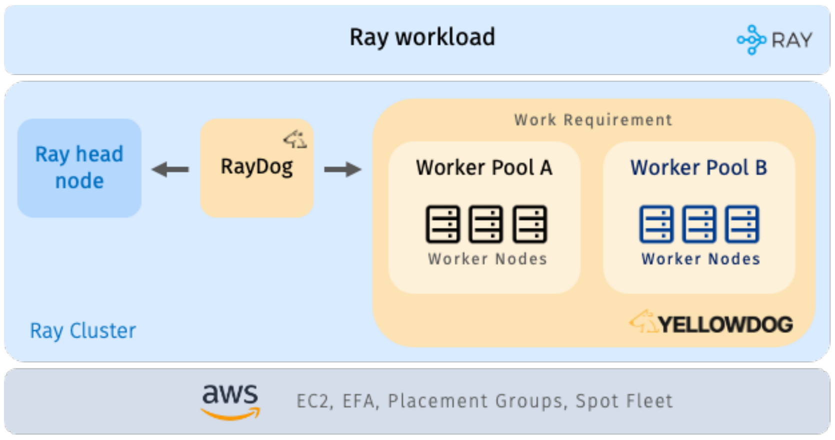
2 RayDog - what is it?

RayDog is an open source module that uses the YellowDog Platform to provision and scale Ray clusters; in effect, providing the infrastructure layer for Ray.



It enables developers to build their application in Ray, leveraging the full Ray ecosystem without modification, and then scale easily with YellowDog to seamlessly transition ideas from prototype to production.

It operates by modelling the Ray cluster as a YellowDog Work Requirement, and provisions YellowDog Worker Pools to provide the Ray cluster with Worker Nodes.



Within the Work Requirement, the Ray Head Node and each Worker Pool is defined as individual Task Groups, and Worker Tags are used to allocate Tasks from the Ray cluster to each specific Worker Pool (e.g., via a user-configured Worker Pool name or the Worker Pool ID).

Worker Pools can be defined and employed in many ways. For instance, grouping nodes across and within Availability Zones (AZs) and cloud regions, or for delineating by pricing model (Spot/on-demand/reserved), or by node type, such as supplying dedicated pools for CPU instances vs accelerated compute (GPU) instances to provide the optimal compute for different workload tasks.

RayDog offers two different usage models:

1. RayDog Builder
   - Enables the user to manually provision and scale one or more Worker Pools to provide Worker Nodes for the Ray cluster
   - Each Worker Pool can use different node types and source instances from different AZs or regions, and can be added and removed dynamically to expand and contract the Ray cluster during its lifetime

2. RayDog Autoscaler
   - Integrates with the in-built Ray Autoscaler, enabling it to dynamically adjust the number of Worker Nodes within the Ray cluster based on Ray workload demands
   - A YAML config file identifies RayDog as the cluster Provider, and the specific node types and Worker Groups[1] that the Ray Autoscaler can request in building the Ray cluster
   - Ray by default scales to 5 Worker Nodes at cluster launch and then ramps from there depending on workload demand, with the ramping rate prescribed within the YAML file

Where the compute requirement is well understood upfront, RayDog Builder is an efficient solution for provisioning the Ray cluster, and can be faster in operation than the RayDog Autoscaler by immediately provisioning Worker Pools of the requested sizes itself rather than waiting for Ray job submissions to drive the Ray Autoscaler into requesting the additional resources.

[1] A set of nodes of the same type that the Ray Autoscaler manages as a unit based on resource requirements. Ray Worker Groups map to YellowDog Worker Pools via the Compute Requirement Template (specifying Sources, Provisioning Strategy etc.)

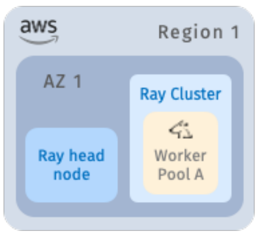
On the flip side, with RayDog Builder the cluster size must be balanced manually for the submitted jobs, and the cluster must also be downscaled manually. Hence where the Ray workload requirements are likely to be more volatile, or for users not wanting to manually tune the Ray cluster, the RayDog Autoscaler is the better option.

3 RayDog support for different Ray cluster topologies

A range of different cluster topologies could be used when deploying Ray, depending on the amount of compute required, the degree of coupling between tasks within the workload, latency/throughput tolerances, and where the data resides (e.g., central vs distributed).

3.1 Single Ray cluster; single AZ

For ML training, and latency-sensitive tightly coupled workloads more generally, tasks should be scheduled together on co-located resources (e.g. using gang scheduling). Provisioning a Ray cluster in a single Availability Zone (AZ) within a region is a good way of meeting these needs.



Single Ray cluster comprised of Worker Nodes in a single pool in a single AZ

First and foremost, it provides low latency - network latency intra-AZ is ~0.1-0.2ms for AWS EC2 instances, an order of magnitude less than the ~1–2 ms between AZs in the same region. A single AZ also avoids incurring inter-AZ traffic costs (typically ~$0.01 per GB in each direction).

The YellowDog Platform includes a number of measures for optimising Ray clusters for tightly-coupled, latency-sensitive workloads including the option of using AWS Placement Groups to utilise instances in close proximity, and fast networking fabrics (e.g., AWS EFA) to further improve interconnectivity between nodes, as well as configuration of Advanced Worker Pools to assign node types, and use of action groups to keep all the nodes in sync.

This optimisation at the infrastructure level then complements well the functionality within the Ray scheduler for controlling where tasks are placed in the cluster (e.g., using `STRICT_PACK` to place tasks as close as possible to minimise cross-node communication).

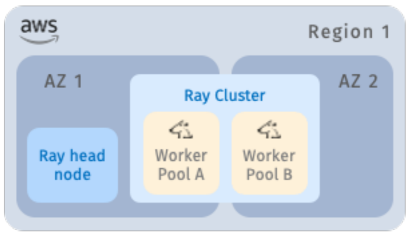
The Ray Autoscaler would typically be used to grow/shrink the cluster dynamically; however, in the case of latency-sensitive tightly coupled workloads, it's often better to keep the cluster size fixed for the job, hence manually configuring the cluster using RayDog Builder would be the recommended approach.

Whilst building a cluster from a single AZ brings many performance benefits, it can also introduce some fragility - if the AZ were to go down, the entire cluster would go offline.

Moreover, accelerated compute quotas and Spot capacity can be depleted quickly in a single AZ, and preemption rates are likely to be higher and with more limited scope for automatic failover.

3.2 Single Ray cluster; multiple AZs

For workloads requiring more fault-tolerance and/or simply those that are embarrassingly parallel and can be spread across a more distributed compute footprint (such as in video rendering, data preprocessing, or running multiple Monte Carlo simulations), a better approach is to deploy the cluster using resources from across multiple AZs within a given region.



Single Ray cluster comprised of Worker Nodes from two Worker Pools in adjacent AZs

Doing so improves instance type availability (especially important for getting the best Spot availability/pricing) and reduces the risk and impact of broad preemption, as well as providing an opportunity to spin up temporary Worker Pools to accommodate surge demand. And whilst this introduces a penalty in terms of network latency, it's relatively small and typically acceptable for most Ray workloads.

RayDog provides comprehensive support for the creation of Worker Pools across a range of compute sources to form a cluster of Worker Nodes for Ray, and can programmatically search for Spot instances to reduce cost which can be combined with a fallback strategy to on-demand instances to ensure sufficient compute is provided for the Ray cluster.

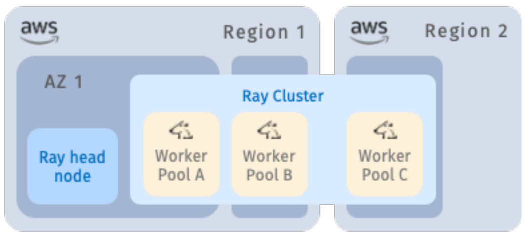
Furthermore, YellowDog is multi-region & multi-cloud aware by design, and by abstracting away cloud-specific CLI/API calls enables the same Ray config to be used across different environments without needing any manual tweaking, thereby simplifying operations.

It also brings powerful orchestration, being able to support Ray in dispatching tasks to individual Worker Pools based on: policy priorities (e.g., region preference, carbon footprint); Spot vs on-demand pricing; capacity / quota availability; data locality and transfer costs (Data Anywhere) etc.

A single Ray Cluster composed of multiple Worker Pools situated in different AZs should provide sufficient capacity for the majority of Ray workloads. For some workloads though, such as high-fidelity financial simulations, the large numbers of GPUs required may even exhaust the resources within a given region - accelerated compute with Spot pricing is in high demand right now, and for high-end P5 (H100) instances there may be <100 available in large regions and only a handful in mid-size regions.

3.3 Single Ray cluster; multiple regions

An obvious solution might be to expand the Ray cluster further and span it across additional regions to boost capacity.



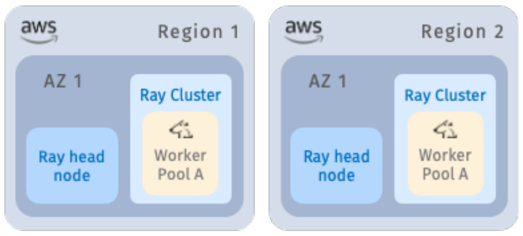
Single Ray cluster comprised of Worker Nodes from three Worker Pools using resources from multiple regions

For workloads that are comprised of embarrassingly parallel tasks with little/no interdependency (such as data preprocessing), this approach can potentially work, although spanning clusters across multiple regions is not natively supported within AWS and would typically require some form of manual scripting, or use of a 3rd party orchestrator such as YellowDog.

However, when it comes to Ray, spanning a Ray cluster across multiple regions can severely degrade performance - round trips between worker nodes and with the head node can jump from ~1ms (intra-AZ) to 50–200ms (inter-region) and throughput also suffers, even with inter-region private links. This latency and variability in network connectivity can destabilise Ray's control plane and kill performance. As cluster size increases, these issues snowball and can reduce scaling efficiency to below 20–30%, effectively thwarting any attempt at growing compute capacity through scaling cluster size alone.

3.4 Individual Ray clusters per region

An alternate approach would be to split the workload at the job level and deploy tasks to individual Ray clusters situated in different regions. For instance, retaining mission-critical and latency-sensitive tasks within a home region, and farming out batch tasks to cheaper resources in more remote regions.



Workload split across individual Ray clusters in each region

Ray itself cannot schedule tasks across clusters (as each Ray head node is located in and manages its own cluster) but with RayDog it's possible to provision Worker Pools sourced from different regions, or even different CSPs, in a similar fashion to sourcing Worker Pools from individual AZs.

3.5 Scaling to very large Ray clusters

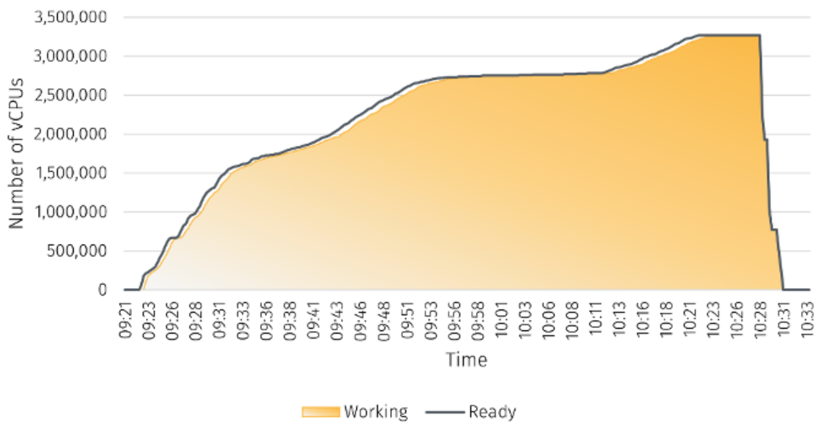
Ray works well running many small tasks with short runtimes across a cluster of a few hundred nodes. As cluster size increases though into the thousands, Ray's control plane can start becoming unstable, and this is exacerbated by the increased likelihood of node preemption/failure at large cluster sizes which further stresses the Ray head node when it comes to rescheduling tasks and managing fault recovery. Pushing beyond 4-5k nodes often takes both Ray and Kubernetes (KubeRay) out of their respective comfort zones.

Large organisations such as Google and Ant Group solve these intrinsic constraints by patching Ray internals and/or tuning the Kubernetes control plane, but very few companies will have the team and resources to do this.

RayDog is subject to the same constraints to some extent, but by abstracting Ray from needing to interface with the CSP provisioning APIs or handle preemptions, it reduces the loading on the Ray head node and thereby enables Ray to scale further to ~10k nodes per cluster.

For those use cases that need to scale even further to cluster sizes well into the tens of thousands of nodes and beyond, YellowDog provide an alternate path that moves away from Ray and refactors the Python workloads to run natively on YellowDog.

The YellowDog Platform in conjunction with AWS previously demonstrated an ability to rapidly scale to 3.2m vCPUs (46,733 Spot instances) across multiple regions within just 33mins, and with any reclaimed Spot instances being immediately replaced to avoid any disruption to workload execution. Since then, the YellowDog platform has been further enhanced and is now able to scale to an impressive 200,000 nodes (30m+ vCPUs) where needed.



3.6 Hybrid (on-prem & cloud)

Ray, in many respects, is agnostic as to whether the nodes are sourced from a single AZ, multiple AZs or multiple regions, and could therefore build Ray clusters from a mix of on-prem and cloud resources.

However, similar to the networking issues incurred by spanning clusters across multiple regions, combining on-prem and cloud resources in a single cluster can also be detrimental to performance depending on the network latency and stability between the disparate resources.

This issue can be ameliorated to some extent through use of AWS Direct Connect to avoid using the public Internet, but in general a better approach is to apportion tasks to dedicated on-prem or cloud Worker Pools using RayDog as previously discussed.

3.7 Summary

There are a variety of cluster topologies that could be pursued for Ray, depending on the level of compute required and workload shape - RayDog amply supports them all.

In broad terms, tightly-coupled workloads, and/or fine-grained tasks that are more latency-sensitive (such as ML training and iterative simulations) are best served by a cluster in a single region, or better still a single AZ, and deployed using RayDog Builder to obtain maximum performance.

Loosely-coupled and embarrassingly parallel workloads (e.g., Monte Carlo simulations, data preprocessing etc.) on the other hand are less constrained and the tasks could be split across RayDog Worker Pools in multiple AZs and potentially in different regions to improve preferred node availability, cost, and resilience, with autoscaling provided by RayDog Autoscaler.

4 RayDog benefits

RayDog's benefits can be summarised as follows:

Powerful orchestration

- YellowDog provides production-grade provisioning and autoscaling with an ability to automatically find, provision and manage compute based on a set of preferences and policies
- YellowDog Worker Pools allow heterogeneous node types to be sourced from different AZs and regions to form a cluster of Worker Nodes for Ray, and also facilitates workload-aware scheduling with tasks being allocated to the optimal Worker Pool
- YellowDog fully supports building clusters using AWS Placement Groups where required to ensure instances are in close proximity, and fast networking fabrics (e.g., AWS EFA) to further improve interconnectivity between nodes

Ultimate scale

- Worker Pools, and hence Ray clusters, can be scaled beyond the constraints typically experienced when using Ray (4-5k+ nodes) to as much as ~10k nodes
- Scaling further to clusters in the tens of thousands can be supported by refactoring the Python workloads to run natively on the YellowDog Platform with an ability to scale all the way up to 200k nodes if needed

Cost efficient

- Ray handles preemption via a mix of fault tolerance mechanisms and resource rebalancing but is reliant on the infrastructure layer to achieve full preemption handling with fallback
- YellowDog does just that through three key features: 1) ceases scheduling tasks to a node during a preemption notification window and remaps them to alternate nodes; 2) enables splitting and prioritising Spot instances across different regions and AZs and auto re-provisions preempted Spot instances; 3) supports Waterfall provisioning strategies to prioritise Spot where required, but with fallback to on-demand to ensure sufficient compute is provided for the Ray cluster
- RayDog is also cost-efficient, automatically spinning up and tearing down Worker Pools when jobs arrive or finish and maintaining high utilisation at the node level

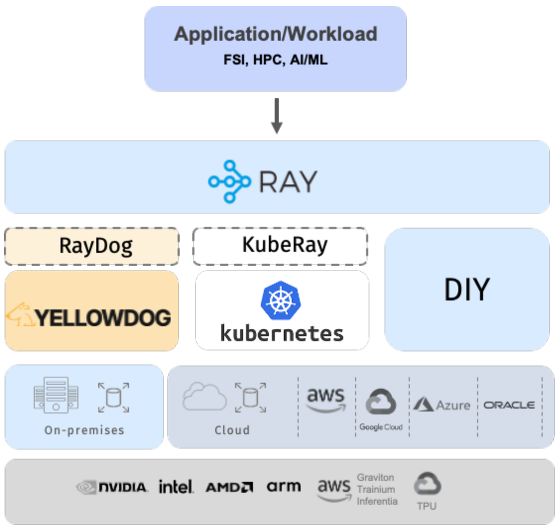
Simpler

- RayDog is simple to use, abstracting away the low-level operational complexity of running Ray at large scale whilst also enabling the same Ray config to be used across different environments without needing any manual tweaking
- YellowDog provides a comprehensive set of tools, SDKs and APIs for those needing fine-grained control, and full Enterprise-grade Observability, Governance, and Security

5 RayDog positioning

To-date, deployments of Ray have depended on either developing a complete framework in-house using the Ray open-source code with VMs, or combining it with an existing Kubernetes framework for the infrastructure layer (using KubeRay), or employing end-end bespoke solutions from companies such as Anyscale.

RayDog provides a compelling alternative, combining open-source Ray with a fully featured infrastructure layer from YellowDog and the breadth and depth of cloud resources from AWS EC2 to offer a solution that is very much greater than the sum of its parts.

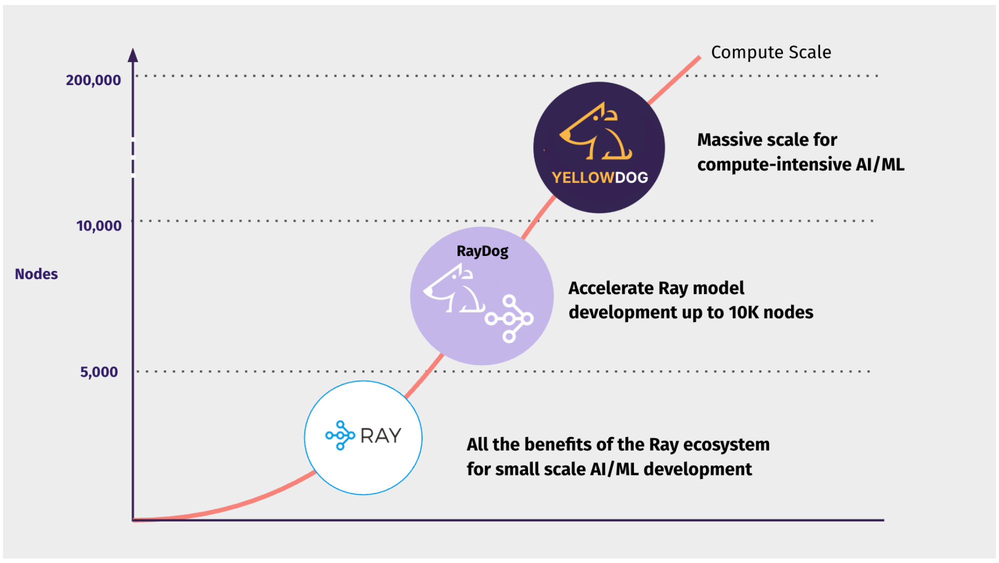


Fully flexible

- YellowDog can build Ray clusters using a variety of node types sourced from different regions, clouds and hybrid sources, with a flexible set of provisioning strategies (Single, Split, Waterfall) and an ability to delineate nodes using Worker Pools for optimal task allocation
- Whilst KubeRay may in theory be able to achieve something similar by running multiple Ray clusters within the same Kubernetes cluster, it's primarily focused on, and limited to, single-region clusters

Superior scale

- Kubernetes can work well for Ray clusters in the 100s-1k nodes. However, when deployed in larger clusters into the 1000s, Kubernetes can struggle to react fast enough to the rapid demands of the Ray head node due to a combination of control plane overheads and node startup latencies
- RayDog doesn't suffer from this issue, and can scale way beyond Kubernetes' limitations to 10k nodes and more in just a few mins
- Such a capability is vital within the Financial Services Industry and already in-use by some of YellowDog's global hedge fund customers
- Where even greater scale is needed, customers have the option of switching to YellowDog native and scaling their clusters to 200k nodes and beyond


Compute Scale
200,000
10,000
5,000
Nodes
YELLOWDOG
Massive scale for compute-intensive AI/ML
RayDog
Accelerate Ray model development up to 10K nodes
RAY
All the benefits of the Ray ecosystem for small scale AI/ML development

Cost efficient

- By utilising AWS' comprehensive Spot Fleet, YellowDog can bring substantial cost savings to Ray whilst still providing capacity and resiliency through advanced pre-emption handling and node utilisation
- It's also able to spin-up clusters much faster than Ray/Kubernetes to accelerate Ray workloads, and then tear-down cloud resources to minimise cloud costs

Enterprise-grade

- YellowDog is an Enterprise-grade platform, providing Observability, Governance, and Security plus a comprehensive set of tools, SDKs and APIs to ensure that your Ray cluster runs smoothly and efficiently and ROI is being optimised; a company building their own solution around open-source Ray would need to do all this themselves

In summary, RayDog combines the ease of use of Ray with the powerful orchestration of YellowDog in a complete Enterprise-grade package. It enables developers to build their application in Ray, leveraging the full Ray ecosystem without modification, and then scale easily with YellowDog to seamlessly transition ideas from prototype to production.

Experience the power of Ray today at scale, and without any of the operational headaches of managing your own infrastructure.

RayDog is publicly available on PyPi and GitHub:

- https://pypi.org/project/yellowdog-ray/#files
- https://github.com/yellowdog/raydog
- https://docs.yellowdog.ai/raydog/index.html
- https://yellowdog.ai/integrations/raydog