Designing a Kubernetes Operator for Machine Learning Applications

Ali Kanso, Edi Palencia,
Kinshuman Patra
Machine Teaching Group
Microsoft
Bellevue, Washington, USA
{ali.kanso, edilmo.palencia,
kinshu}@microsoft.com

Jiaxin Shan, Mengyuan
Chao, Xu Wei
Infrastructure Lab
Bytedance
Bellevue, Washington, USA
{jiaxin.shan, mengyuan.chao,
wei.xu}@bytedance.com

Tengwei Cai, Kang Chen,
Shuai Qiao
Computing Intelligence
Department
Ant Group
Hangzhou, Zhejiang, China
{tengwei.ctw, wuhua.ck,
qiaoshuai.qs}@antgroup.com

ABSTRACT

Machine Learning workloads such as deep learning and hyperparameter tuning are compute-intensive by nature. Parallel execution is key to reducing the learning time. The Ray Framework is a distributed middleware that provides primitives to seamlessly parallelize machine learning code execution across a cluster of compute node. Launching a Ray managed machine learning application requires a Ray cluster that is diligently configured, well connected and easily scalable. Kubernetes, the container management middleware, satisfies all the requirements to create and scale ray clusters. However, setting up a cluster within Kubernetes, is a tedious and error prone task when done manually. In this paper we present KubeRay, an Operator and suite of tools designed, and built to create Ray cluster in Kubernetes with minimum effort. We present our architectural choices, our open-source implementation, and we analyze the performance of our solution.

CCS CONCEPTS

• **Software and its engineering** → Software creation and management

KEYWORDS

Machine Learning, Ray Framework, Kubernetes, Operators, Distributed Systems, Clustering, Linux Containers.

1 Introduction

Machine learning workloads are compute-intensive by nature [1]. Especially with large datasets that need to be moved, processed, and stored. Having a single compute-instance (physical machine, virtual machine (VM) or a container) handle all the computation is not a feasible approach. Especially that some model training can last days or weeks. The classic solution for this problem is parallelization. We can significantly reduce the compute time by splitting the application code into smaller tasks that can be scheduled on different compute-instances and collect their computational results. However, this raises another set of challenges. Such as managing (storing and sharing) the tasks input and output, scheduling the tasks on compute-instances, and dynamically scaling those instances as the computation progresses. The Ray framework [2] is built specifically to target the above issues. It is developed for building distributed applications, allowing developers to leverage simple Ray APIs to make their code clustering enabled. For instance, developers can turn their code functions into tasks (or remote functions) by annotating them with a Ray decorator. Similarly, a class can be turned into an actor with another Ray decorator [3]. Ray can then schedule and execute those tasks/actors in parallel on multiple nodes and collect their results.

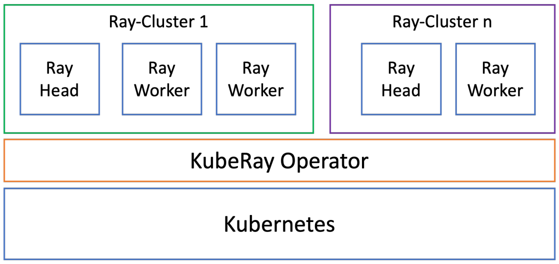


Figure 1: KubeRay Operator

A Ray cluster can be deployed on any compute-instance, such as a VM or a container. In this article we argue that Ray is best deployed in containers and managed by Kubernetes [4], mainly for the following reasons:

- Containers can scale easily and fast. As we will show in our experimental results, it takes less than 5 seconds to add a new container compute-instance.
- Containers can easily be ported from one environment to another. The Kubernetes API on Microsoft Azure AKS [5],

WoC '21, December 6, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-9171-9/21/12…$15.00
https://doi.org/10.1145/3493649.3493654

Amazon EKS [6], or Google GKE [7] is always the same. Therefore, a single Ray cluster can be replicated in multiple environments, or it can span across clouds.

While Ray and Kubernetes form an ideal match, many data-scientists developing ML applications lack the necessary Kubernetes expertise to setup a Ray cluster on Kubernetes and monitor, debug, and operate their applications running in such environment. In this paper we introduce the KubeRay Kubernetes Operator [8]. The term Operator in this context refers to a specialized software in which we embed the domain knowledge of an expert [9]. As a result, the operator maintains the Ray cluster with minimum intervention from the application developers.

Our objective in this paper is to provide the reader with the thought process that goes into creating an Operator. The different design alternatives that can be considered and present the advantages and shortcomings of each. And finally, the performance that can be expected from an Operator.

This paper is organized as follows: in Section 2 we discuss in more details the internals of Kubernetes and Ray. In Section 3 we present the alternative architectures that we debated (as a Ray open-source community) when designing our operator, and we present the advantages and limitations of each alternative. In Section 4 we present our experimental results. We survey the literature in Section 5. Finally, we conclude and discuss the future work in Section 6.

2 Background

There are two modes of using the Ray framework. Either one Ray cluster can host multiple applications, or each machine learning application can be encapsulated in its own Ray cluster, and hence multiple Ray clusters can co-exist in the same compute environment (such a Kubernetes cluster). We believe the latter case is the most prevalent when offering machine learning as a service on Kubernetes. In this Section we will present more details on the Ray framework and Kubernetes.

2.1 Ray Under the Hood

To better understand Ray, we will present (1) the overall architecture of Ray and (2) the start sequence and parameters for Ray nodes.

2.1.1 Ray Architecture. Ray defines the notion of a head node and worker nodes as shown in Figure 2. The head node manages the metadata that can be accessed by other nodes in the cluster such as the directory for locating objects. The driver is the root python process for the application and workers which are python processes that can execute tasks. Each node has a *Raylet* containing a scheduler that manages resources and an object store that handles memory and state. The worker nodes are identical to the head node without the global control store and the driver. It is important here to distinguish between a Ray node, and a Ray worker. A Ray node is a compute-instance that has the Ray components running inside. A Ray worker is a process (within a Ray node) that is assigned tasks to execute by the scheduler. Large objects are stored in the object store (which acts like shared storage across nodes). A worker can find a large objects by consulting the global control store.

Each worker node (compute-instance) can have different resources (CPU, GPU, Memory, Disk). The Ray scheduler takes the available resources into consideration when scheduling the tasks.

2.1.2 Ray Startup [1]. The Ray head node is expected to start first and spawn (internally) a Redis database that persists the data. The worker nodes are given as parameters the address (TCP/IP) information of the head node, as well as the credentials to connect. If the head node is not ready, this causes the worker node to crash when trying to connect. From this perspective, the order of instantiation is important.

2.1.3 Ray Scaling. The Ray tasks/actors can be spawned anytime during the application lifetime. Similarly, completed tasks can be removed. As such the needed resources for the cluster vary at runtime. Ray supports the notion of dynamic scaling where compute-instance can be added/removed at runtime. While any type of node can be added at runtime, not any random can be removed, only the nodes that are idle, and no longer hosting tasks are marked for removal.

2.1.4 Ray Upgrading. The Ray framework not only supports distributing machine learning applications, but it also offers serving the machine learning models to be consumed by other services. From that perspective it acts like a long running webserver. Upgrading a long running Ray cluster with minimum downtime is not currently supported natively, and hence there is a need for a management component (e.g., an Operator) that upgrades the Ray nodes in a proper sequence.

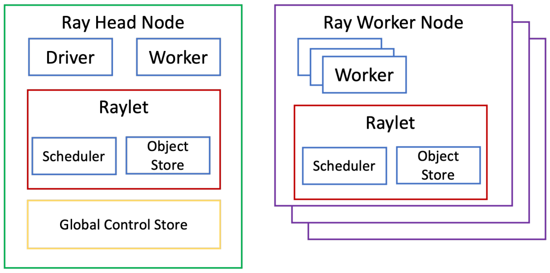


Figure 2: Ray internal architecture

2.2 Kubernetes

Kubernetes is the de-facto container management platform [4]. It defines a declarative model for specifying a desired system state

[1] As of the time of writing, Ray is at version 1.7.

and implements a controller logic that constantly strives to reconcile the actual system state with the desired state.

The Kubernetes API-server serves REST operations and provides the frontend to the cluster's shared state. The state is stored in a key/value store with a watch API that provides an event-based interface for asynchronously monitoring changes to keys. The keys' values represent the artefacts managed by Kubernetes such as Pods. Pods (which abstract containers) are the smallest deployable units of computing that you can create and manage in Kubernetes. Kubernetes manages the lifecycle (creation, deletion, update, monitoring, etc.) of Pods through controllers. The controller manager in Kubernetes supports dozens of controllers (such as the *deployment* and *replicaset* controllers). For instance, the deployment controller, constantly watches for the creation of a deployment manifest that defines how replicated pods should be managed, and accordingly makes sure the number of actual Pods matches the ones running in the cluster.

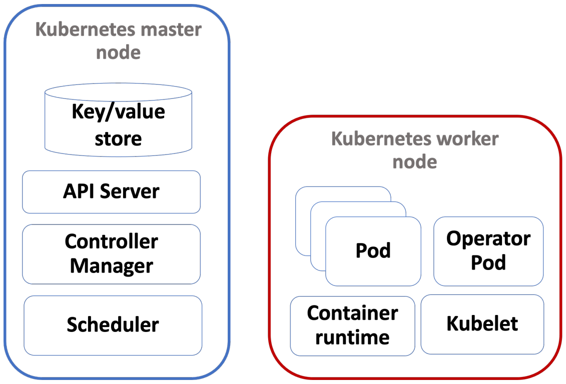


Figure 3: Kubernetes architecture

2.3 Kubernetes Operator

Kubernetes supports the notion of custom resources. A custom resource is an extension of the Kubernetes API. Operators are software extensions to Kubernetes that make use of custom resources to manage applications and their components.

3 KubeRay: the Ray Clustering Operator

We define at a high level the set of duties required for managing a Ray cluster. We split those duties into *lifecycle management* duties and *auto-scaling* duties.

Lifecycle management:

1. Create a Ray cluster.
2. Delete a Ray cluster and cleanup all the relevant resources.
3. Failure recovery (e.g., if a physical machine fails, create the Ray nodes on another machine(s))
4. Create additional Ray nodes and delete idle nodes.

Auto-scaling:

5. Metrics collection: determine the CPU, Mem, GPU, TPU utilization in the cluster (currently there is only one threshold supported). And determine the idle nodes.
6. Making a scaling up/down **decision** based on the metrics collected.

The clear separation of duties under two categories gives way to three main alternative designs for a Ray clustering operator, henceforth referred to as KubeRay [8].

3.1 Why Using a Kubernetes Operator?

As discussed in Section 2. Managing a Ray cluster requires an intricate set of actions and configuration parameters that must be performed and set with domain expertise for the cluster to be functional. This knowledge is embedded in the KubeRay Operator to alleviate the burden of cluster management from Ray users and shift it to the Operator itself.

Operators inherit major benefits by being closely integrated with Kubernetes:

- Added security by managing through Role Based Access Control (RBAC) who has permissions to create/modify/delete Ray clusters on the K8s cluster.
- Ability to enforce admission control on the Ray cluster configuration. E.g., reject configurations that are not semantically valid.
- Using the OpenAPI V3 validation mechanisms to enforce syntactical validation with minimum effort.
- Ability to enforce resource quota in the namespace of the Ray cluster.
- Leveraging the ubiquitous availability of Kubernetes as a service offering by cloud vendors. And as a result, allowing the Operator to manage a Ray cluster using kube-federation [10] across clusters.

3.2 Alternative Designs for the KubeRay Operator

The designs discussions with the open-source community [11] revolved mainly around the idea of integrating the lifecycle management duties and the auto-scaling duties in the same operator (Design 1) versus having the operator manage the lifecycle management and a separate auto-scaler making the auto-scaling decisions (Design 2.a and Design 2.b). All the designs presented benefit from the advantages of using a Kubernetes Operator discussed earlier.

3.2.1 Design 1: Operator with Auto-scaling Ability

The first design alternative is to embed the auto-scaling in the operator itself. This mixes concerns, having the same code base for the auto-scaler and lifecycle management.

Pros:

- Faster communication between the auto-scaler and the lifecycle manager.

Cons:

- Mixing of concerns. By having the auto-scaler as part of the Operator, we introduce tight coupling between the scaling and lifecycle management. Hence, if a novel auto-scaler with smarter (or domain specific algorithms) is to be used for the scaling decisions, then the entire Operator needs to be changed. We believe that this design pattern is not aligned the Kubernetes design principle of extensibility. For example, the Kubernetes scheduler, is not only easily replaceable, but also extensible with webhooks that can leverage other schedulers. As such, we see the Kubernetes horizontal pod auto-scaler (HPA) developed as a completely independent component from the scheduler. Similarly, we believe that a better design pattern is to use a modular design where the Ray auto-scaler is easily replaceable and extensible.

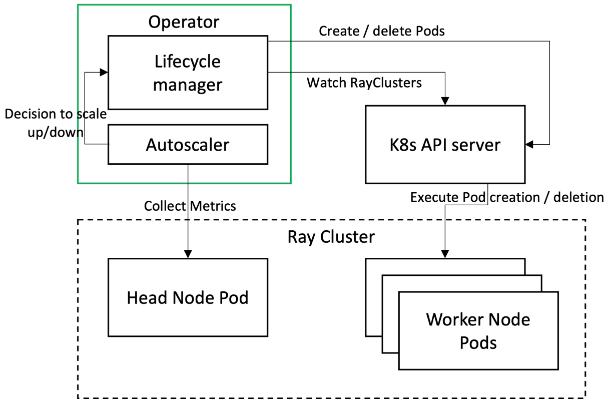


Figure 4: Design 1: auto-scaler as part of the Operator

3.2.2 Design 2.A: Operator Exposing Web-interface.

In this design, we extract the auto-scaling logic out of the operator, and instead, the Operator exposes a web-interface (HTTP/GRPC) through which the auto-scaling decisions are conveyed to the lifecycle manager.

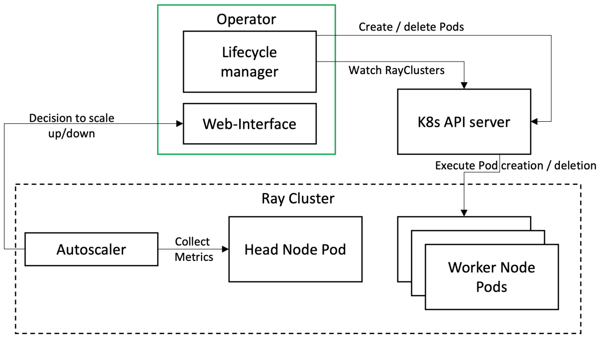


Figure 5: Design 2.A Operator with web-interface

Pros:

- Relatively faster communication between the auto-scaler and the lifecycle manager.

Cons:

- A complicated logic is needed to enforce authentication and authorization between the auto-scaler and the Operator. This logic is already implemented in the Kubernetes API-server. And hence, we lack the justification of this added complexity.

3.2.3 Design 2.B: Operator without Auto-scaling Ability

In this design, we have complete separation between the auto-scaler and the lifecycle manager.

Pros:

- Complete separation of concerns between the auto-scaler and the lifecycle manager.

Cons:

- Indirect communication between the auto-scaler and the lifecycle manager. The Kubernetes API-server is the hub through which the communication occurs. The auto-scaler modifies the custom resource representing the Ray cluster in Kubernetes, and this triggers an event in the Operator watching for those changes to take the appropriate actions and reconcile the actual state with the desired state. As we shall see in the experimental results section, this indirect communication while it brings lots of value it incurs a latency of roughly 600 milliseconds.

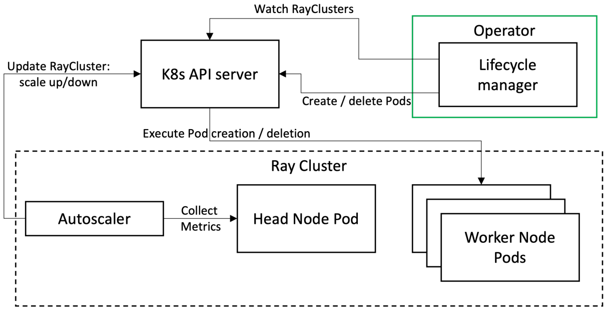


Figure 6: Design 2.B Operator with independent auto-scaler

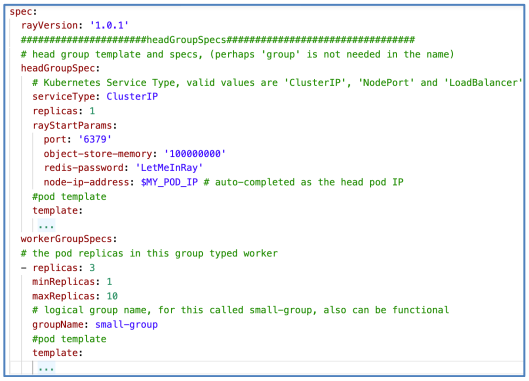
Among all the presented designs, we believe that Design 2.B is the optimal one in terms of making the most benefit from close integration with Kubernetes and at the same time having loose coupling with the auto-scaler.

3.3 The Internals of KubeRay

KubeRay reacts to the creation/deletion/modification of Kubernetes custom resources of type *raycluster*. Typically, a *raycluster* custom resource is created by the user, and it corresponds to an instance of a machine learning application.

3.3.1 The raycluster custom resource

Figure 7 presents a snippet of the *raycluster* custom resource. The *raycluster* includes (among others) the service type used to abstract the head pod, the Ray *start* parameters, as well as the information for one or many worker groups. The default Kubernetes Pod template is used to describe the pods to ensure a consistent Kubernetes user experience when using the Operator. One thing to note here, is that many parameters are calculated at runtime and embedded by the operator into the pods as we will show in Section 3.3.2.



```
spec:
  rayVersion: '1.0.1'
  ######################headGroupSpecs#################################
  # head group template and specs, (perhaps 'group' is not needed in the name)
  headGroupSpec:
    # Kubernetes Service Type, valid values are 'ClusterIP', 'NodePort' and 'LoadBalancer'
    serviceType: ClusterIP
    replicas: 1
    rayStartParams:
      port: '6379'
      object-store-memory: '100000000'
      redis-password: 'LetMeInRay'
      node-ip-address: $MY_POD_IP # auto-completed as the head pod IP
    #pod template
    template:
      ...
  workerGroupSpecs:
  # the pod replicas in this group typed worker
  - replicas: 3
    minReplicas: 1
    maxReplicas: 10
    # logical group name, for this called small-group, also can be functional
    groupName: small-group
    #pod template
    template:
      ...
```

Figure 7: Snippet of the *raycluster* custom resource

3.3.2 KubeRay in Action

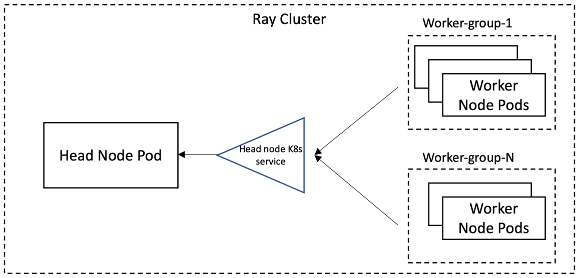


Figure 8: A Ray cluster in Kubernetes

KubeRay parses each *raycluster* custom resource instance and starts by creating the head pod. It assigns the head pod labels and environment variables. It also uses Kubernetes downward API [12] to set as environment variable the pod's IP address and ports. This information is set to be consumed by the machine learning application using the Ray library inside the head pod. Kuberay also mounts an *emptyDir* volume to the shared memory mount point /dev/shm to be used by the Ray object store. Once the head pod is created KubeRay creates a Kubernetes with the proper selectors to abstract the head pod. This is mainly to ensure that even if the head pod fails and is recreated with a new IP address, the workers can still connect/reconnect using the service IP address stored in the Kubernetes DNS server.

Next, KubeRay generates a configuration for an init-container that is injected in the Ray worker pods. The init-container is container that runs until a condition is met and then exists allowing the main container of the Pod to start. KubeRay injects a script in our init-container that periodically checks if the DNS lookup for the head service yields in a valid result (signaling that the head pod has been assigned a valid IP). This avoids workers trying to connect to an unavailable head and crashing. KubeRay sets up the proper parameters for the "*ray start*" command in the workers to make sure they can reach the head and the correct flag values including Redis authentication.

After the Ray cluster is up, it can be scaled up by increasing the number workers (in the *raycluster* custom resource instance) for a given worker group or creating a new worker group all together. As for scaling down, the scaling strategy field supports defining pods by name to remove a specific node. Otherwise, if the any worker can be removed, then we can simply decrement the number of workers.

4 Experimental Results

We tested KubeRay on an Azure cluster of three virtual machines each with 64 cores Intel(R) Xeon(R) Platinum 8272CL CPU @ 2.60GHz, with 256 GB or RAM and 10Gbit/s Ethernet interface. The experimental results shown below are the average of 50 experiments running a basic Python Ray application that creates 100 tasks to print the Ray worker node names.

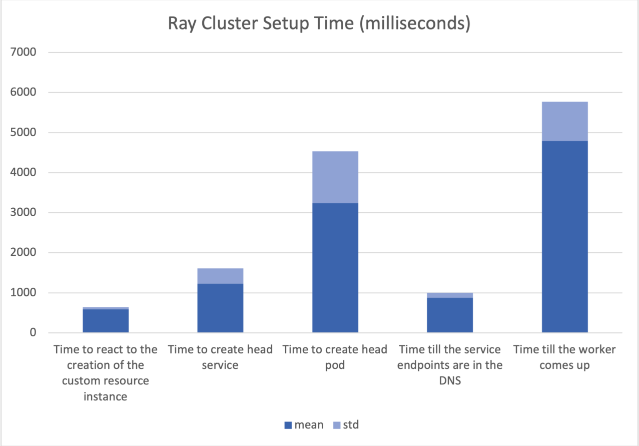


Figure 9: Experimental Results

For each raycluster custom resource instance, KubeRay starts by creating the head service, followed by the head Pod and finally the workers. The rationale behind this flow is that if the head Pod creation returns an error, there is no point in continuing with the workers. However, this does not mean that the workers are created after the head is up and ready, it only means once Kubernetes accepts the head Pod definition KubeRay sends the worker Pods definitions to be created. Therefore, the time shown for the Ray worker to come up, is considered the total time it takes to setup the cluster (which is roughly 5 seconds), because the worker is only up after the head Pod and the head service have been successfully created. The time from the user creates the raycluster custom resource instance in Kubernetes till the time KubeRay receives this request is less than 600 milliseconds. This includes Kubernetes authentication, authorization, validation, persisting the raycluster instance and sending the notification event to KubeRay. An interesting observation here is that the 5 seconds average time is takes to setup the Ray cluster coincides with the Kubernetes service level objective (SLO) of a startup latency of schedulable stateless pods, excluding time to pull images and run init-containers being less than 5 seconds. This latency is measured from pod creation till all its containers are reported as started and observed via watch, measured as 99th percentile over last 5 minutes [13].

5 Related Work

The Ray *Autoscaler* [23] can scale a Ray cluster in Kubernetes. We opted out of using it for the following reasons: at the time of the writing, this *Autoscaler* had mixed concerns for scaling and lifecycle management. Including scaling the underlying cluster in the cloud. We believe other more mature tools like Terraform or the cloud provider APIs are more suited for this task, Moreover, the API for the Autoscaler is not intuitive, for instance it does not use the Pod-Template to describe pods. It lacks several abilities like using init-containers, and automatically mounting needed volumes. Finally, it lacks the testing needed to be used in production. Partly because it is written in Python, and the Kubernetes testing support is still limited there, whereas in Golang, we have readily available in-memory Kubernetes API-servers for testing. We believe that both the design and implementation do not meet our requirements to be production ready. Therefore, in agreement with the Ray developers, we developed KubeRay.

Kubeflow [14] is an open-source framework for deploying machine learning workflows on Kubernetes. It allows the users to declaratively specify a sequence of creating Kubernetes objects. While KubeRay also enforces a sequence of events, the main purpose is to extract information from each step and use it in the next step, this is done in an imperative (procedural) way in the code which is not trivial to do in a declarative way with Kubeflow and may take more effort. In addition, reacting to failures, and scaling requests and managing the lifecycle of the cluster are not supported in Kubeflow.

Amazon *SageMaker* is a fully managed machine learning service. We see Ray as an alternative open-source solution that is cloud agnostic. However, it is worth noting that Amazon *SageMaker* Operators [15] (a lightweight wrapper to call *Sagemaker* API instead of managing resources natively on Kubernetes) enables the use of fully managed *SageMaker* machine learning tools natively from Kubernetes or Kubeflow.

Other solutions that focus on scheduling machine learning workloads in an efficient manner (such as *Run:AI* [16]) can improve the overall performance of the training process. The authors in [17], present a Kubernetes scheduler that leverage machine learning to optimize scheduling in Kubernetes. However, unlike KubeRay, both of these solutions are not intended to manage the lifecycle of the machine learning application.

Other works [18] [19] targeting machine learning platforms based on Kubernetes relate to KubeRay in the broader sense but they are not leveraging the Ray framework and hence are not specific to the domain we are targeting in KubeRay.

6 Conclusion and Future Work

In this paper we presented KubeRay, a Kubernetes Operator for managing distributed machine learning applications running on the Ray framework. We presented different design alternatives and discussed the advantages and shortcomings of each design. We then discussed the internals of KubeRay and showed our experimental results showing that KubeRay is a viable solution for managing Ray clusters in Kubernetes.

The work on improving and extending KubeRay is ongoing. As future work we will add a tool suit that enhances the user experience such as client tools that enable creating clusters using a command line interface.

ACKNOWLEDGMENTS

We would like to extend our gratitude to the engineers from *AnyScale* [20], *ByteDance* [21], And *AntGroup* [22] that contributed to the design discussions. And to all maintainers of the KubeRay Operator.

REFERENCES

[1] Thang Le Duc, Rafael García Leiva, Paolo Casari, and Per-Olov Östberg. 2019. Machine Learning Methods for Reliable Resource Provisioning in Edge-Cloud Computing: A Survey. ACM Comput. Surv. 52, 5, Article 94 (October 2019), 39 pages. DOI:https://doi.org/10.1145/3341145

[2] The Ray Framework. URL: https://www.ray.io/ last accessed in October 2021

[3] Ray Remote Functions URL: https://docs.ray.io/en/latest/walkthrough.html last accessed in October 2021

[4] Kubernetes: The Container Management Framework. URL: https://kubernetes.io/ last accessed in October 2021.

[5] Azure Kubernetes Service. URL: https://azure.microsoft.com/en-us/services/kubernetes-service/#overview last accessed in October 2021

[6] AWS Kubernetes Service. URL: https://aws.amazon.com/eks/ last accessed in October 2021.

[7] Google Kubernetes Engine. URL: https://cloud.google.com/kubernetes-engine last accessed in October 2021.

[8] KubeRay: The Ray Kubernetes Operator. URL: https://github.com/ray-project/kuberay/tree/master/ray-operator last accessed in October 2021

[9] The Kubernetes Operator Design Pattern. URL: https://kubernetes.io/docs/concepts/extend-kubernetes/operator/ last accessed in October 2021

[10] Kubernetes Cluster Federation. URL: https://github.com/kubernetes-sigs/kubefed last accessed in October 2021.

[11] KubeRay design document, URL: https://docs.google.com/document/d/1DPS-e34DkqQ4AeJpoBnSrUM8SnHnQVkiLlcmI4zWEWg/edit#heading=h.uvghkc1tgnfw, last accessed on October 2021.

[12] Kubernetes Downward API. URL: https://kubernetes.io/docs/tasks/inject-data-application/downward-api-volume-expose-pod-information/ last accessed in October 2021.

[13] Kubernetes Service Level Objective / Service Level Indicator SLO/SLI https://github.com/kubernetes/community/blob/master/sig-scalability/slos/slos.md last accessed in October 2021.

[14] KubeFlow https://www.kubeflow.org/ last accessed in October 2021.

[15] Amazon Sagemaker URL: https://aws.amazon.com/pm/sagemaker/ last accessed in October 2021.

[16] Run:AI https://www.run.ai/ last accessed in October 2021.

[17] L. Toka, G. Dobreff, B. Fodor and B. Sonkoly, "Machine Learning-Based Scaling Management for Kubernetes Edge Clusters," in *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 958-972, March 2021, doi: 10.1109/TNSM.2021.3052837.

[18] Chun-Hsiang Lee, Zhaofeng Li, Xu Lu, Tiyun Chen, Saisai Yang, and Chao Wu. 2020. Multi-Tenant Machine Learning Platform Based on Kubernetes. In Proceedings of the 2020 6th International Conference on Computing and Artificial Intelligence (ICCAI '20). Association for Computing Machinery, New York, NY, USA, 5–12. DOI:https://doi.org/10.1145/3404555.3404565

[19] Yuzhou Huang, Kaiyu cai, Ran Zong, and Yugang Mao. 2019. Design and implementation of an edge computing platform architecture using Docker and Kubernetes for machine learning. In Proceedings of the 3rd International Conference on High Performance Compilation, Computing and Communications (HP3C '19). Association for Computing Machinery, New York, NY, USA, 29–32. DOI:https://doi.org/10.1145/3318265.3318288

[20] AnyScale. URL: https://www.anyscale.com/ last accessed in October 2021.

[21] ByteDance. URL: https://www.bytedance.com/en/ last accessed in October 2021.

[22] AntGroup. URL: https://www.antgroup.com/en last accessed in October 2021.

[23] Ray Autoscaler Kubernetes-plugin URL: https://github.com/ray-project/ray/tree/ray-1.7.0/python/ray/autoscaler last accessed in October 2021.