# KubeRay RayJob Refactor Design Doc

Authors: Archit Kulkarni Kai-Hsun Chen Praveen Gorthy
Date: Jun 6, 2023
Status: PUBLIC

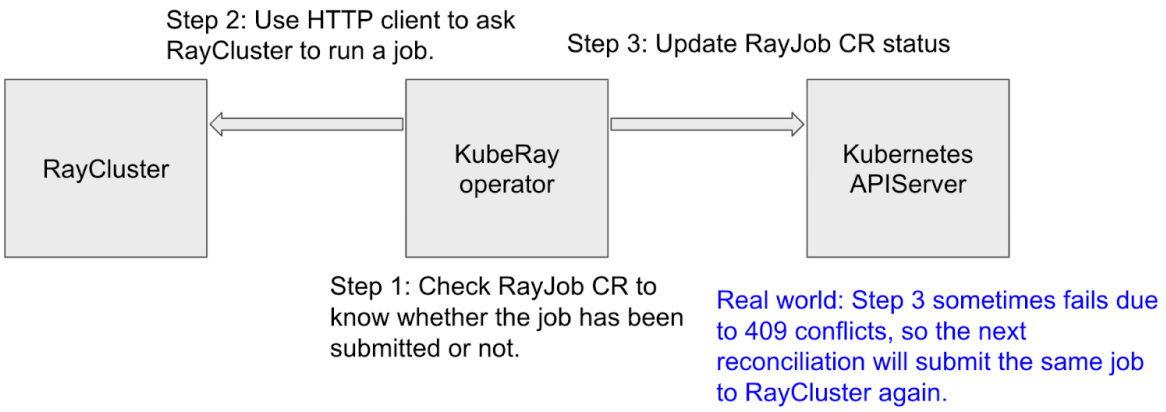
Related document: RayJob - KubeRay Docs

TL;DR At a high level, in the RayJob implementation we will remove the existing Dashboard HTTP client and instead submit a Kubernetes job that runs all the necessary Ray Job API commands (`ray job submit`, `ray job status`, etc.)

## Context

The current state of RayJob is unstable. Here is one major issue, arising from checking the RayJob CR "Status" field in each reconciliation loop to decide whether to submit the job: https://github.com/ray-project/kuberay/issues/756

- [Bug] 409 conflicts cause two jobs creations




(Source: Kai-Hsun Chen)

Furthermore, RayJob is missing a few components which are standard in Kubernetes Jobs:

1. Support retries
2. Support timeout
3. Have a good user experience for viewing Job logs and Ray logs after the job is finished

To fix this, we propose a refactor of RayJob which uses a K8s job under the hood to call `ray job submit`, instead of checking the RayJob CR status in each reconciliation loop and using

that to determine whether to submit the job. We made this decision to fix the issue above, which is fundamentally hard to solve without this refactor.

## Job Lifecycle

In the new refactor, when the RayJob CR is submitted, the Ray operator creates the following two resources:

1. A RayCluster
2. A Kubernetes Job
    a. The Kubernetes job `command` will be a script that does the following:
        i. Wait until the RayCluster Dashboard server is ready *
        ii. Submit the job via `ray job submit` which tails output to `stdout`
        iii. Periodically monitor `ray job status`. When the Ray job reaches a terminal state (SUCCEEDED, FAILED, STOPPED) , exit.

If `shutdownAfterJobFinishes` is set, the RayCluster will be deleted by the Ray Operator when the Kubernetes Job exits.

* Point for discussion: As an alternative, we could wait until all pods are ready before submitting the job. This is what's done currently. The downside is that it could cause flakiness if there's a pod that fails to come up. In this refactor, we propose to just wait for the head node and Dashboard server to be ready.

If at runtime some Ray actors or tasks require more pods than are currently available, those actors and tasks will just be pending until the Ray autoscaler brings up the necessary pods.

Since this is a breaking behavior change, we can have an option in the CRD to make it configurable (wait_for_all_pods_healthy).

### Submitting jobs to existing clusters

Currently the RayJob spec contains an undocumented field `cluster_selector` which allows the user to select an existing RayCluster to submit the job to. However, in our view this unnecessarily duplicates the existing Ray Job API (`ray job submit --address=<head_node_ip>:8265`). We should deprecate and remove this field. This will simplify the behavior of RayJob: it will always create a new cluster.

## Job Status

Currently we have two fields in the CR that are updated periodically by the Ray Operator:

- `RayJobDeploymentStatus`: The status of the Ray Operator deploying the job, including the lifecycle of the RayCluster. This is not a Ray-level status, this is an internal status for the Ray Operator.
    - Currently includes "Initializing", "FailedToGetOrCreateRayCluster", "WaitForDashboard", "FailedJobDeploy", "Running", "FailedToGetJobStatus", "Complete", "Suspended"
- `JobStatus`: The underlying `ray job status` output (SUCCEEDED, FAILED, etc)
- `Message`: The `message` field of `JobInfo` exposed by Ray, containing more detailed status information.

Changes:

- Due to the changes in the job lifecycle above, some fields in `RayJobDeploymentStatus` will change. For example, FailedToGetJobStatus will no longer be necessary, and we may have other statuses related to the lifecycle of the underlying K8s job.
- (Non-change) `JobInfo` exposed by Ray contains other useful information such as `runtime_env`, `metadata`, and so on. However these are specified in `spec` and do not change during runtime, so it does not make sense to add them to `status`.

### How the status will be updated

The `JobStatus` will be updated from within the K8s job. The `RayJobDeploymentStatus` will be updated from the Ray operator.

## Job Retries (GH Issue)

This is a basic feature of K8s jobs, so we will commit to supporting it. There are two situations in which we should retry:

1. The pods are unhealthy. In this case we must restart the RayCluster.
2. The cluster is healthy, but the Ray Job FAILED due to an application error.
    a. In this case we could consider restarting the job with `ray job submit` without restarting the cluster. However, this is a premature optimization. It is simpler and more reproducible to just restart the RayCluster.

We will go with the approach to **always restart the RayCluster**.

The existing K8s Job spec contains the field `spec.template.spec.restartPolicy` and `spec.backoffLimit`. The meaning of these is slightly different from the behavior described above, so we will add a new field `spec.maxRetries` in the RayJob CRD rather than reuse the names `restartPolicy` and `backoffLimit`. This will prevent confusing the user.

In the RayJob status, we can publish the number of retries so far as `status.numRetries`.

## Timeout (GH Issue)

This is a basic feature of K8s jobs, so we will commit to supporting it.
The implementation will be to simply set the timeout in the K8s job. When the Ray Operator detects that the K8s Job has completed, it should delete the cluster or leave it alive depending on `shutdownAfterJobFinishes`.

- To stop the Ray Job while keeping the cluster alive, we will call `ray job stop` in a PreStop hook set in the K8s job pod template spec.

We will add the field `spec.ActiveDeadlineSeconds` to the RayJob CRD because this is a well-known existing field for K8s Jobs, and the meaning is the same.
We will update the status with a new field `deadlineExceeded` for the timeout.

As in K8s jobs, the timeout will take precedence over the retries. The Job will not retry if the timeout has been exceeded.

## Viewing logs after the job is finished

There are two types of logs that are of interest:

1. The output from the user-provided Ray job `entrypoint` script
2. The collection of Ray system logs from the lifetime of the Ray Cluster

For (1), we can support the same API as K8s jobs, namely `kubectl logs rayjobs/my_job`. This simply gets the output from the underlying K8s job pod which is running `ray job submit`. Ray automatically tails the entrypoint script output to `stdout`.

For (2), we can support specifying a persistent volume for the `RayCluster`. We shouldn't need any additional implementation here, besides passing the volume name when starting the cluster:

```
ray start --temp-dir=/my_volume/tmp_path
```

When the Cluster goes down, the user can still read the Ray logs from disk at `/my_volume/tmp_path/logs`.

## New configuration fields

- Currently `RuntimeEnv` is passed in as a base64 encoded string. This is a bad user experience. We should pass it in YAML format, the same way RayService does.
- `Entrypoint_num_cpus` and `entrypoint_num_gpus` and `entrypoint_resources` need to be supported in the RayJob CRD `spec` field. See ray.job_submission.JobSubmissionClient.submit_job — Ray 2.4.0 for the definition.

## Backwards incompatibility

This refactor introduces the new fields `spec.maxRetries` and `spec.activeDeadlineSeconds`, as well as the changes to the `Status` field.

If we were only introducing new fields, we might not need to bump the version from the current version, `v1alpha1`. But since we are introducing changes to the `Status` field and changing the default behavior of the job to just wait for the head node to be ready instead of waiting for all nodes, we can consider bumping the version to `v1beta1`.

## CronJob

This is a standard K8s feature and we should support it. Similar to K8s, we will have a new CRD `RayCronJob` or `CronRayJob`. We can mimic the implementation of the K8s `CronJob`.