Kueue

Unofficial slides to collate notes and learnings about kueue

Hannah DeFazio

Overview

Kueue

Definition

- **Kueue** is a kubernetes-native queueing system that manages quotas and how jobs consume them, especially in environments with scarce and expensive resources like GPUs. Kueue decides when a job should wait, when a job should be admitted to start (as in pods can be created) and when a job should be preempted (as in active pods should be deleted).
- Kueue is not a *replacement* for the main Kubernetes scheduler (`kube-scheduler`). Instead, it wraps around it, adding a queueing layer that gives control over ***when and how*** jobs are admitted to the cluster.
- The name **Kueue** is a stylized blend of **K** (for Kubernetes) and **queue**.
- A core design principle for Kueue is to **avoid duplicating mature functionality** in Kubernetes components and well-established third-party controllers. Autoscaling, pod-to-node scheduling and job lifecycle management are the responsibility of cluster-autoscaler, kube-scheduler and kube-controller-manager, respectively

A Simple Analogy

Imagine your expensive GPUs are a VIP lounge at a popular nightclub.

- **Without Kueue:** Anyone can rush the door at any time. It's chaos. Some people (jobs) might get in while others are crowded out, and the lounge might get dangerously full (cluster overload).
- **With Kueue:** Kueue is the bouncer at the door. Every job gets a ticket (`Workload`) and gets in a line (`LocalQueue`). The bouncer (`Kueue`) looks at the line and at the capacity of the VIP lounge (`ClusterQueue`). Only when there's space does the bouncer unclip the rope and let the next job in. The bouncer can also manage multiple lines, giving priority to some while ensuring everyone eventually gets a turn.

Kube-scheduler

Definition

- The `kube-scheduler` is the default scheduler for Kubernetes and a core component of its control plane. Its main job is to assign newly created or not yet scheduled (unscheduled) **pods** to the best possible **nodes** for them to run on.

`Kube-scheduler` selects a node for the pod in a 2-step operation:

1. Filtering: Finding the Candidates ✅

First, the scheduler finds a list of all possible nodes where the pod could run. It starts with all nodes in the cluster and then *filters* out any that don't meet the pod's specific requirements.

At the end of this step, the scheduler has a list of suitable candidate nodes. If the list is empty, that Pod isn't (yet) schedulable.

2. Scoring: Picking the Winner 🏆

Next, the scheduler takes the list of suitable nodes and gives each one a *score* based on a set of priority rules. The goal is to find the best node among the good candidates.

The node with the highest score wins, and the scheduler assigns the pod to it. If there is more than one node with equal scores, kube-scheduler selects one of these at random.This process is called *binding*

Kueue + `kube-scheduler`

Analogy (Enrolling in a university)

Kueue is the Registrar's Office. It decides **IF and WHEN** you are allowed to take a class. It looks at the big picture: Are you an enrolled student? Is there room in the department's quota? Do you have priority? Once it gives you the green light, you are "admitted."

The `kube-scheduler` is the seating algorithm for the classroom. Once you're admitted, the `kube-scheduler`'s only job is to find you an empty seat (`Node`) in the lecture hall. It handles the final, low-level placement.

	Kueue (The "When")	**Default `kube-scheduler` (The "Where")**
Primary Job	Decides **if and when** a batch job is allowed to start based on queues, quotas, and fairness.	Decides **where** an individual pod should run based on available node resources and placement rules.
Operates On	High-level `Workload` objects (representing a whole job).	Low-level `Pod` objects.
Manages	Queues, user/team quotas, priority, and resource fairness over time.	Node resources (CPU/memory), taints, tolerations, and affinity at a specific moment in time.
Role	**High-level gatekeeper.**	**Low-level placement engine.**

Problem Statement

What problem does kueue solve?

By default, Kubernetes is "first-come, first-served." If 100 ML training jobs requesting a GPU are submitted at once but there are only 10 GPUs available, 90 of those jobs will immediately fail. This leads to:

- **Resource Contention:** Jobs constantly fight for the same limited resources.
- **Lack of Fairness:** One team or user could hog all the GPUs, starving other important workloads.
- **Inefficient Utilization:** Expensive GPUs might be used for low-priority tasks while high-priority jobs wait.
- **No Quotas:** There's no easy way to say "Team A gets 60% of the GPUs, and Team B gets 40%."

Kueue solves this by stopping jobs *before* they create pods and placing them in a managed queue.

Kueue Objects

Workload

Definition

A `Workload` is an application that will run to completion. It can be composed by one or multiple Pods that, loosely or tightly coupled, as a whole, complete a task. A workload is the unit of admission in Kueue. Sometimes referred to as `job`.

This prevents resource contention and allows for fair, managed queueing.

The workload must have the `kueue.x-k8s.io/queue-name: <local-queue-name>` label. It tells Kueue which `LocalQueue` in the `Workload`'s namespace should handle this job.

The Role of the Workload

- **The Admission Ticket:** The `Workload` is the object that Kueue's control plane actually manages. It's what gets **admitted**, **suspended**, or **rejected**.
- **Resource Specification:** It calculates the resource requirements (CPU, memory, GPUs) of the entire job, aggregated from all its pods. This allows Kueue to make intelligent scheduling decisions based on quotas and fairness.
- **Decoupling:** It decouples the job's existence from its execution. The job can exist in the cluster as a `Workload` without consuming any active resources until it's admitted.

Workload

Lifecycle

A `Workload` moves through several distinct phases, managed by the Kueue controller.

- **Pending:** The `Workload` is created and is waiting in a `LocalQueue`. No pods are running.
- **Admitted:** The `ClusterQueue` has available resources and gives the `Workload` the green light. The original job controller is now allowed to create pods.
- **Running:** The job's pods are running on the cluster, consuming resources.
- **Finished:** The job completes successfully (or fails).
- **Suspended (Optional):** If preemption is enabled, a higher-priority `Workload` can cause this one to be paused and its pods deleted to free up resources.

LocalQueue

Definition

A `LocalQueue` is the **namespaced entry point** for all Kueue workloads. It's the specific "toll lane" a job enters within a user's project. Its primary role is to act as a **bridge**, funneling `Workloads` from a specific team or namespace into a shared, cluster-wide resource pool managed by a `ClusterQueue`.

The `LocalQueue` is the main object users interact with, simplifying job submission. The `LocalQueue` acts as a **necessary and secure** bridge between a user's **namespaced** world and the cluster's shared resources.

Enables Multi-Tenancy: Allows different teams to submit work without needing cluster-level permissions.

Simple Configuration: Its main job is to point to a `ClusterQueue`.

How It Works: The Workflow

Workloads (in Namespace) → LocalQueue (in Namespace) → ClusterQueue (Cluster-wide)

1. A user submits a job to a `LocalQueue` in their namespace.
2. The `LocalQueue` forwards the resource request to its designated `ClusterQueue`.
3. The `ClusterQueue` manages quotas and decides when to admit the job.

ClusterQueue

Definition

A `ClusterQueue` is a **cluster**-scoped object that governs a pool of resources such as pods, CPU, memory, and hardware accelerators, defining usage limits/quotas and Fair Sharing rules. Only batch administrators should create `ClusterQueue` objects.

While a `LocalQueue` is the entry lane, the `ClusterQueue` manages the entire highway.

A ClusterQueue defines:

- Quotas for the resource *flavors* that the ClusterQueue manages, with usage limits and order of consumption.
- Fair Sharing rules across the multiple `ClusterQueues` in the cluster.

How It Works: The Workflow

The `ClusterQueue` aggregates demand from multiple `LocalQueues` and manages a single resource pool.

LocalQueue (Team A) ↘️ LocalQueue (Team B) → ClusterQueue (Manages Quotas) → Admitted Workloads LocalQueue (Team C) ↗️

1. Multiple `LocalQueues` from different namespaces are configured to point to a single `ClusterQueue`.
2. The `ClusterQueue` aggregates all the pending `Workloads` from these queues.
3. Based on its defined quotas, `ResourceFlavors`, and queueing strategy, it decides which `Workload` to "admit" next.

ResourceFlavor

Definition

A `ResourceFlavor` defines available compute resources in a cluster and enables fine-grained resource management by associating workloads with specific node types. It allows you to differentiate and request **specific types of a generic resource**. It's the "type of vehicle" in our toll plaza analogy—it lets you request a high-performance sports car instead of just any car.

This is essential for heterogeneous clusters with different kinds of hardware, such as:

- NVIDIA A100 vs. T4 GPUs
- Nodes with high-memory vs. standard memory
- On-demand vs. spot instances

How It Works: The Workflow

`ResourceFlavors` use node labels to direct workloads to the correct hardware.

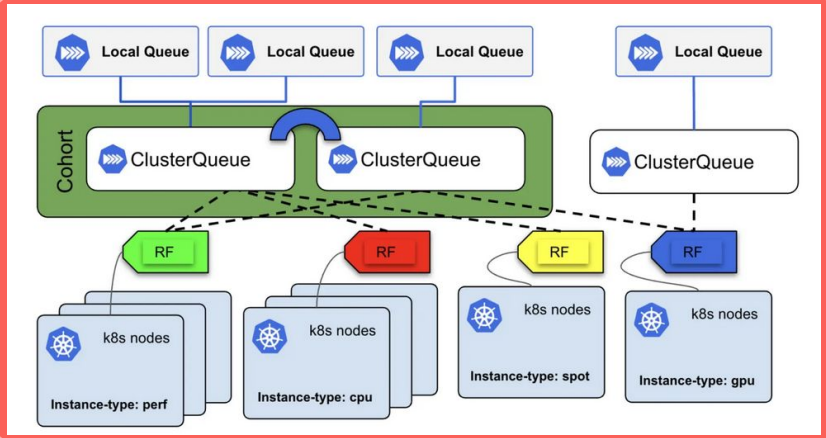
Admin Labels Nodes → Admin Creates `ResourceFlavor` → User's `Workload` Requests Flavor → Kueue Admits to Correct Node Type

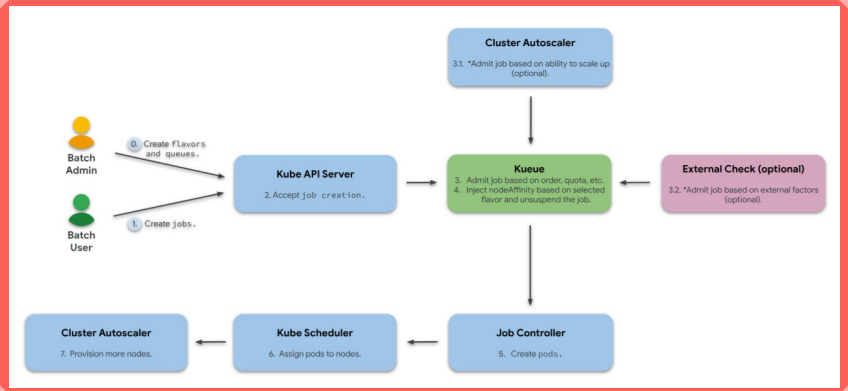
1. An administrator applies labels to different nodes (e.g., `gpu-type: a100`).
2. A `ResourceFlavor` is created that uses a `nodeSelector` to match those labels.
3. The `ResourceFlavor` is associated with a `ClusterQueue`.
4. A user's `Workload` can then request a specific flavor, ensuring it lands on the appropriate hardware.

Workflow

What does kueue do?

1. **Workload Creation:** A user submits a job. Kueue's webhook intercepts it, pauses it, and creates a `Workload` object to represent its resource request before any pods are created.
2. **Enters LocalQueue:** The `Workload` is placed in the `LocalQueue` specified by the user, entering a specific "toll lane" within their namespace.
3. **Forwards to ClusterQueue:** The `LocalQueue` forwards the `Workload`'s request to the central `ClusterQueue` it's configured to use.
4. **Admission Decision:** The `ClusterQueue` (the control tower) checks its quotas. If the required resources (and a specific `ResourceFlavor`, if requested) are available, it **admits** the `Workload`.
5. **Pods are Scheduled:** Only after admission are the job's actual pods created. The standard `kube-scheduler` then takes over to find the best node and run them.
6. **Monitoring:** Kueue's job doesn't end when it admits a workload. It continuously monitors the state of the cluster to know how many resources each queue is consuming.





Kueue + Kserve Overview

Red Hat Build of Kueue

The Red Hat Build of Kueue is an enterprise-supported version of the open-source Kueue project, packaged as an OpenShift Operator to simplify the installation, management, and integration of job queueing on a cluster.

Label Propagation

- The Red Hat Build of Kueue operator monitors `Pods` with the kueue labels
- This means that the kueue labels in the `ISVC`/`LLMISVC` need to be propagated to the `Deployment` and `Pods`

ISVC Workflow (Raw Deployment)

1. The predictor reconciler copies the labels from the `ISVC` (here)
2. The predictor reconciler copies and filters the annotations from the `ISVC` (here)
3. These are used to create the predictor's object metadata (here)
4. Which is then passed to the raw kubernetes resource reconciler (via here / here)
5. Which creates a deployment reconciler with the componentmeta created in (3) (here)
6. …… gets passed around through various functions
7. A few steps later, the default deployment is made with the componentmeta (here)

LLMISVC Workflow

TODO

Workflow

1. **Kueue is configured:** The RH Build of Kueue Operator is installed. A `Kueue` object is created and setup to monitor Deployments and Pods.

2. **`Namespace` is Created:** A `Namespace` is created with the *`kueue.openshift.io/managed: "true"`* label to explicitly tell `Kueue` that it should actively monitor and manage workloads created within that namespace.

3. **`InferenceService` is Created:** You create an `InferenceService`/`LLMInferenceService` in the `Namespace` and add Kueue-specific labels/annotations to its metadata section (e.g., *`kueue.x-k8s.io/queue-name: <local-queue-name>`*).

4. **`Deployment` is Created:** The `Deployment` inherits the kueue labels from the `InferenceService` for its own labels and its pod template spec

Workflow (cont.)

5. **Pods are Created:** The Kubernetes `ReplicaSet` controller creates the final `Pods` based on the `Deployment`'s template, so they are created with the propagated Kueue labels.

    a. The *`kueue.x-k8s.io/queue-name`* label on a running `Pod` tells Kueue's monitoring components, "This `Pod`'s CPU and GPU usage should be counted against the quota of the user-queue." Without this label, Kueue wouldn't be able to track which queues are using which resources, and it couldn't make accurate decisions about when to admit new jobs from the queue or evict (terminate) lower-priority `Pods`.

6. **Controller Creates a Workload:** An integration controller (which is designed to link KServe and Kueue) detects the new `InferenceService` . It pauses the `InferenceService` and creates a corresponding Kueue `Workload` object, copying the labels from the `InferenceService` to the `Workload`.

7. **Kueue Admits the `Workload`:** Kueue's scheduler sees the new `Workload` in its queue. When there are enough resources available in the cluster according to its rules, Kueue "admits" the `Workload`.

Kueue + Kserve Instructions

Kueue + kserve

Installation

Operator dependencies:

1. Serverless
2. Authorino
3. OSSM 2
4. cert-manager
5. ODH
6. Red Hat build of Kueue

Setup:

1. ODH operator:
   a. Create a `DSCI`
   b. Create a `DSC` with **everything** set to *removed* except kserve
2. RH Build of Kueue operator:
   a. Create `Kueue` in the
   b. Add `Pod`, `Deployment`, and `StatefulSet` to the `spec.config.integrations.frameworks` list in the Kueue

Please see Running the opendatahub-tests Locally: Specific Test Instructions: Kueue Tests for more detailed instructions

Kueue + kserve

Object Creation

Kueue:

1. Create a `ClusterQueue`
2. Create a `LocalQueue`
3. Create a `ResourceFlavor`

ISVC:

1. Create a namespace
2. Add the label `kueue.openshift.io/managed: “true”` to the namespace
3. Create an ISVC with label `kueue.x-k8s.io/queue-name=<localqueue>`

Links

- https://kueue.sigs.k8s.io/docs/overview/
- https://cloud.google.com/kubernetes-engine/docs/tutorials/kueue-intro
- https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/
- https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
- https://kubernetes.io/docs/reference/scheduling/config/#profiles
- https://github.com/kubernetes-sigs/kueue/blob/main/README.md
- https://developers.redhat.com/articles/2025/05/22/improve-gpu-utilization-kueue-openshift-ai?source=sso#enter_kueue__smarter_batch_workload_orchestration_for_kubernetes
- https://docs.redhat.com/en/documentation/openshift_container_platform/4.19/html/ai_workloads/red-hat-build-of-kueue