Deferred ResourceQuota Validation and Usage Tracking

Authors: Illya Chekrygin
Status: Proposal
Last revision: Sep 2, 2025

Introduction

In Kubernetes, `ResourceQuota` objects are applied at the **namespace level**, but their effects are most visible in how pods and other resources are admitted and managed within that namespace. A quota defines aggregate limits on resources such as CPU, memory, storage, or even object counts (for example, the number of pods) that can be consumed across all objects in the namespace. When a pod is created, the API server evaluates its requests and limits against the namespace's quota. If the new pod would cause the namespace to exceed the quota, the creation request is rejected at admission. This mechanism ensures that no single namespace can over-consume cluster resources. In practice, quotas also drive pod compliance by requiring resource requests and limits for CPU and memory, and may work in conjunction with `LimitRange` defaults. While quotas are defined at the namespace scope, their enforcement directly governs the lifecycle of pods, shaping both what can be created and how resources are distributed across the cluster.

Related Work

An earlier version of this proposal was scoped specifically to support pods with `SchedulingReadinessGates`. That narrower effort has since been superseded by the broader scope of this proposal, which generalizes deferred `ResourceQuota` validation and tracking to all pods.

Problem Statement

The current model of `ResourceQuota` enforcement is tightly coupled to pod admission, where requests that exceed quota are rejected outright. While this ensures strict fairness, it introduces several challenges.

First, it diverges from the broader Kubernetes paradigm of optimistic allocation, where unschedulable workloads are permitted to remain in the Pending state until resources become available.

Second, it conflates **potential usage**, the sum of resource requests declared by pods at admission, with **actual usage**, which only materializes when pods are successfully bound to nodes and those resources are exclusively claimed. This lack of distinction results in inefficient capacity utilization, as resources may appear exhausted even when no pods are actively consuming them.

Furthermore, admission-time enforcement forces premature failures that are opaque to users and inconsistent across workload controllers, especially in dynamic scenarios such as elastic jobs, schedule-gated pods, or workloads that gradually scale. Without deferred validation, the scheduler lacks the ability to coordinate quota with placement, leading to unfairness, resource starvation, and, in some cases, deadlocks when multiple workloads compete for constrained capacity.

```
⎇> kubectl get pods
NAME             READY   STATUS            RESTARTS   AGE
node-affinity    0/1     Pending           0          76s
schedule-gated   0/1     SchedulingGated   0          70s

⎇> kubectl got resourcequota
NAME  AGE  HARD            USED            REMAINING
p1    40s  count/pods=20   count/pods=2    count/pods=18
           cpu=2           cpu=2           cpu=0 # ← No remaining CPU!
           memory=200.0Gi  memory=200.0Mi  memory=199.8Gi

⎇>kubectl create -f test-pod-1.yaml
Error from server (Forbidden): error when creating "test-pod-1.yaml": pods
"test-1" is forbidden: exceeded quota: p1, requested: cpu=1, used: cpu=2,
limited: cpu=2
```

The consequences of this model can be grouped into the following areas:

- **Wasted capacity**: Quota may appear fully consumed even when no pods are bound, leaving cluster resources underutilized.
- **Poor user experience**: Pods fail immediately at admission instead of being visible in a Pending state, providing little opportunity for troubleshooting or recovery.
- **Inconsistent workload behavior**: Workload controllers (e.g., Jobs, Deployments) must handle quota rejections differently, leading to fragmented semantics and operational overhead.
- **Increased risk of starvation and deadlocks**: Multiple workloads competing for quota without scheduler coordination can block each other indefinitely.

Purpose

This document frames the problem with current resource quota enforcement and proposes a design to:

- Change pod admission behaviour to allow newly created pods to be admitted irrespective of current quota usage (pod with assigned `nodeName` being a notable exception; see design section).
- Defer pod resource quota validation until the scheduling cycle.
- Track pod resource quota usage for assigned pods only.

Goals

- **Scope – Pod compute resources**: Focus on CPU and memory resources reserved on Kubernetes nodes.
- **Deferred validation**: Perform `ResourceQuota` validation during scheduling rather than at admission.
- **Deferred usage tracking**: Record `ResourceQuota` usage only once a pod is assigned or bound to a node.
- **Alignment with scheduling**: Allow pods to remain Pending until both resources and quota are available, ensuring consistent behavior with other Kubernetes paradigms.

Non-Goals

- **Quota object schema changes**: The shape of the `ResourceQuota` API will remain unchanged; no new fields or resource types are introduced.
- **Non-compute resources**: This effort focuses on CPU and memory requests and limits. Quotas for storage, ephemeral storage, or object counts (e.g., number of pods, PVCs, Services) remain outside the scope.
- **Eviction and preemption policies**: This proposal does not redefine how evictions or preemptions are triggered or handled, though deferred validation may provide a foundation for future improvements.
- **Autoscaler integration**: While deferred validation may benefit autoscaler behavior indirectly, this proposal does not require changes to Cluster Autoscaler or Horizontal Pod Autoscaler.
- **Controller-specific logic**: Higher-level workload controllers (e.g., Jobs, Deployments, StatefulSets) are not modified. Their interactions with ResourceQuota remain unchanged apart from experiencing deferred validation semantics.

Use Cases

Pending Pods Consuming Quota
In the current model, Pending pods count fully against namespace quota even though they are not bound to nodes and may never run. This can deplete the namespace's quota and prevent other workloads from being created, even if those workloads could be scheduled immediately. Deferred quota validation avoids this deadlock by charging quota only when pods are placed, ensuring that runnable workloads are not blocked by Pending ones.

User Burden of Re-Creation After Rejection
When pods are rejected at admission due to exceeding quota, the responsibility to retry lies with the user or the workload controller. This introduces operational complexity: users must monitor failures, manually re-attempt pod creation, or rely on controllers with inconsistent retry logic. Deferred quota validation removes this burden by keeping pods in the Pending state until quota is available, ensuring consistent behavior across workloads without requiring user intervention.

Proposed Design

Deferred ResourceQuota shifts enforcement for pod compute resources (CPU, memory, etc.) from admission to scheduling. Validation occurs as part of the scheduler pipeline. ResourceQuota accounting moves to the point a pod is actually placed on a node. The model introduces a clear distinction between potential usage, intent at admission, and actual usage, exclusive claim at bind time. The ResourceQuota API shape remains unchanged, behavior is gated and opt-in.

> *Unassigned pod resources are not counted towards ResourceQuota usage. Since most (typica) pods are created by default "un-assigned" - ResourceQuota enforcement is deferred to the Scheduling phase.*

Scope

The scope of this proposal is strictly limited to **pod compute resource quotas** as defined in the Kubernetes documentation. Specifically, it covers enforcement and tracking of:

- `requests.cpu`
- `limits.cpu`
- `requests.memory`
- `limits.memory`

Other types of `ResourceQuota` constraints — such as object count quotas (e.g., maximum number of pods, ConfigMaps, or PersistentVolumeClaims) and storage-related quotas (e.g.,

`requests.storage`, `ephemeral-storage`) — remain unaffected. These will continue to be validated and enforced at admission as they are today.

By narrowing the focus to CPU and memory compute resources, this proposal ensures targeted improvements to scheduling alignment and usage tracking, while maintaining compatibility with existing quota mechanisms for all other resource types.

Admission

At admission time, pods **without a node assignment** (i.e., unassigned pods) are not validated against current compute resource quotas. This defers enforcement to the scheduling phase, where quota can be validated in the context of actual placement.

Pods **with an assigned node name** continue to be validated using the existing `ResourceQuota` admission process, preserving current enforcement semantics for explicitly scheduled or pre-bound pods.

Scheduling

To enforce quota at scheduling time, we will introduce a new ResourceQuota scheduler extension plugin (RQ plugin) dedicated to validating pod resource quota during Pod Scheduling Context cycles:

PreFilter

> *[PreFilter is] used to pre-process info about the Pod, or to check certain conditions that the cluster or the Pod must meet. If a PreFilter plugin returns an error, the scheduling cycle is aborted.*

The `ResourceQuota` scheduler plugin will enforce quota validation during the **`PreFilter`** extension point. At this stage, if a pod's resource requests would cause the namespace to exceed its compute resource quota, the pod is skipped from scheduling consideration.

In this case, the scheduler records the failure in the pod's status using a condition consistent with other unschedulable scenarios. An example condition is shown below:

```
conditions:
  - lastProbeTime: null
    lastTransitionTime: "2025-09-03T04:40:52Z"
    message: '0/1 nodes are available: quota exceeded: p1, requested: cpu=1, used:
cpu=2, limited: cpu=2. preemption: 0/1 nodes are available: 1 Preemption is not
helpful for scheduling.'
    reason: Unschedulable
    status: "False"
    type: PodScheduled
```

This behavior ensures that pods which cannot be scheduled due to quota constraints remain Pending, with clear visibility into the cause, rather than being rejected outright at admission.

Alternative: PreEnqueue

The **`PreEnqueue`** extension is invoked before pods are added to the scheduler's internal active queue, where they are marked as ready for scheduling. In theory, this extension could also be used to enforce `ResourceQuota` validation earlier in the scheduling pipeline.

However, `PreEnqueue` has a significant limitation: it lacks a mechanism to surface **failure visibility** back to the user. If quota checks fail at this stage, the pod is simply withheld from the active queue without a clear condition or message recorded on the pod object. This limitation has been documented as a major usability issue (see related issue as an example).

As a result, while `PreEnqueue` could reduce scheduling churn by filtering pods earlier, it does not provide the transparency and user feedback required for diagnosing quota-related scheduling failures.

ResourceQuota Usage Tracking

Currently, quota usage tracking occurs alongside quota validation at admission time. Since this proposal removes quota validation for unassigned pods at admission, it also removes quota usage tracking at that point. To properly account for quota usage under deferred validation, the quota controller's **update filter** must be extended to trigger recalculation when pods transition into a **bound** state.

At present, the update filter triggers quota recalculation in the following cases:

- **Delete** events, to release quota.
- **Update** events, specifically when:
    - Pods transition into a terminal phase (`Failed` or `Succeeded`).
    - Pods are deleted.

The controller's eligibility check is implemented by `QuotaV1Pod`, which determines whether a pod should count against quota:

```
// QuotaV1Pod returns true if the pod is eligible to track against a quota
// if it's not in a terminal state according to its phase.
func QuotaV1Pod(pod *corev1.Pod, clock clock.Clock) bool {
   // if pod is terminal, ignore it for quota
   if corev1.PodFailed == pod.Status.Phase || corev1.PodSucceeded ==
```

```
pod.Status.Phase {
      return false
   }
   // if pods are stuck terminating (for example, a node is lost), we do not want
   // to charge the user for that pod in quota because it could prevent them from
   // scaling up new pods to service their application.
   if pod.DeletionTimestamp != nil && pod.DeletionGracePeriodSeconds != nil {
      now := clock.Now()
      deletionTime := pod.DeletionTimestamp.Time
      gracePeriod := time.Duration(*pod.DeletionGracePeriodSeconds) * time.Second
      if now.After(deletionTime.Add(gracePeriod)) {
         return false
      }
   }
   return true
}
```

Change for Deferred Quota

With deferred validation, unassigned pods must not count toward quota usage. Therefore, the following condition is added:

```
// QuotaV1Pod returns true if the pod is eligible to track against a quota
// if it's not in a terminal state according to its phase.
func QuotaV1Pod(pod *corev1.Pod, clock clock.Clock) bool {
   // Do not track quota of unassigned pods.
   if pod.Spec.NodeName == "" {
      return false
   }

   // if pod is terminal, ignore it for quota
   if corev1.PodFailed == pod.Status.Phase || corev1.PodSucceeded ==
pod.Status.Phase {
      return false
   }
   // if pods are stuck terminating (for example, a node is lost), we do not want
   // to charge the user for that pod in quota because it could prevent them from
   // scaling up new pods to service their application.
   if pod.DeletionTimestamp != nil && pod.DeletionGracePeriodSeconds != nil {
      now := clock.Now()
      deletionTime := pod.DeletionTimestamp.Time
      gracePeriod := time.Duration(*pod.DeletionGracePeriodSeconds) * time.Second
      if now.After(deletionTime.Add(gracePeriod)) {
         return false
      }
```

```
    }
    return true
}
```

Update Filter Adjustment

The update filter must now capture transitions where pods move from **unassigned** → **assigned** (i.e., when `Spec.NodeName` changes). This ensures quota usage is recalculated when pods begin consuming actual node resources.

```
// DefaultUpdateFilter returns the default update filter for resource update events
for consideration for quota.
func DefaultUpdateFilter() func(resource schema.GroupVersionResource, oldObj,
newObj interface{}) bool {
    return func(resource schema.GroupVersionResource, oldObj, newObj interface{})
bool {
        switch resource.GroupResource() {
        case schema.GroupResource{Resource: "pods"}:
            oldPod := oldObj.(*v1.Pod)
            newPod := newObj.(*v1.Pod)
            return core.QuotaV1Pod(oldPod, clock.RealClock{}) !=
core.QuotaV1Pod(newPod, clock.RealClock{})
```

Pod Binding

Pods may be bound to nodes without passing through the scheduler's validation path by using the `pods/binding`subresource directly. In such cases, deferred `ResourceQuota` checks performed by the scheduler are skipped, creating a risk of quota overcommit.

Mitigations:

- **RBAC restrictions**: Limit `create` access on the `pods/binding` subresource so that only the scheduler and trusted system controllers can perform direct binding. Regular users and untrusted controllers must not be able to bypass the scheduler. This ensures the scheduler remains the authoritative enforcement point for deferred quota validation.
- **Validation at BindingREST**: As an additional safeguard, resource quota validation logic could be introduced into the pod storage `BindingREST` path to calculate and validate quota on `pods/binding` calls. This provides a secondary enforcement point in case direct binding bypasses the scheduler.

Component Change Summary

API Server – Admission

Currently, all pod compute resource requests are validated against `ResourceQuota` at admission, and quota usage is updated immediately. Under this proposal, compute quota validation will be skipped for pods without a `spec.nodeName`(unassigned pods). Pods with a node name will continue to be validated exactly as they are today. Non-compute quotas, such as object counts and storage quotas, remain enforced at admission. This ensures backward compatibility while deferring compute quota enforcement to scheduling.

Scheduler

Today, the scheduler has no role in `ResourceQuota` validation. With this proposal, a new **ResourceQuota scheduler plugin** is introduced. During the `PreFilter` phase, the plugin validates the pod's compute resource requests against the namespace quota. If the quota would be exceeded, the pod is marked `Unschedulable` and skipped from scheduling consideration. The failure surfaces in pod conditions, providing clear visibility. An alternative considered was `PreEnqueue`, but that extension lacks mechanisms for user feedback and would leave pods invisible in the active queue without explanation.

ResourceQuota Controller:

The controller tracks quota usage at admission, releases quota on delete, and updates when pods transition to terminal phases. With deferred validation, this behavior must be revised: unassigned pods are ignored for quota accounting, and quota is only charged once a pod is bound. The `QuotaV1Pod` helper function is extended to exclude pods without a `spec.nodeName`.

Update Filter

The update filter currently triggers recalculation on Delete events and on pod phase transitions (Failed or Succeeded). With deferred validation, it must also detect transitions where pods move from **unassigned** → **assigned**. This ensures recalculation occurs as soon as pods begin to consume actual node resources.

Pod Binding

Direct binding of pods via the `pods/binding` subresource presents a bypass risk, as it allows pods to be bound without scheduler involvement and therefore without quota validation. To mitigate this, RBAC should restrict `create` access on `pods/binding` so only the scheduler and trusted system controllers can perform direct binding. As an additional safeguard, quota

validation could be added to the `BindingREST` path, ensuring validation occurs even if binding bypasses the scheduler.

Risks

Race Condition Between Scheduler and Quota Controller

A race condition between quota validation in the scheduler and quota usage tracking in the ResourceQuota controller can result in temporary overcommitment of resources. For example, multiple pods may pass scheduler-side validation before the quota controller updates usage, causing the cluster to admit workloads that collectively exceed the namespace quota.

Mitigation

Introduce a ResourceQuota cache in the scheduler plugin to track near-term quota utilization. The cache is reconciled against authoritative usage from `ResourceQuota.status` and updated on pod binding and relevant pod updates.

Because the cache can be temporarily stale, it should bias toward **overestimating** usage, which increases validation rejections, reducing the risk of over-commitment. While this may reject some pods conservatively, it is preferable to **under-estimate** usage, which could admit pods that collectively exceed quota.

- **Bias and reconciliation.** Maintain a per-namespace compute-quota view that increments on pod assumption and decrements on assumption rollbacks or terminal phases, then reconcile to `ResourceQuota.status` on bind and periodic resync.
- **Safety over liveness.** Prefer false negatives, conservative rejections, over false positives that lead to quota over-commitment.

This problem is not unique to the proposed ResourceQuota scheduler plugin. A similar challenge exists in the node utilization space and is addressed through maintaining a node utilization cache. The same approach, maintaining a ResourceQuota cache with conservative bias, can mitigate temporary inconsistencies between scheduler validation and controller reconciliation.

Scheduler Complexity and Latency

Integrating quota validation into the scheduler introduces additional logic into the scheduling cycle. This raises the risk of increased scheduling latency, especially under high churn when many pods are Pending. If quota checks are computationally expensive or require frequent quota controller updates, the scheduling path could become a new scalability bottleneck.

User Experience and Semantics Shift

Today, quota enforcement is deterministic and immediate: pods that exceed quota are rejected outright. Under deferred validation, such pods remain Pending, which changes the semantics visible to end users and workload controllers. This could create confusion in monitoring, alerting, or autoscaling systems that assume Pending pods indicate only node-level resource shortages, not quota constraints.

Operational Complexity

Deferring quota enforcement spreads responsibility across multiple components: API server, scheduler, and quota controller. This distributed enforcement model introduces new coordination risks and makes troubleshooting more complex. Operators may need additional tooling and observability to understand why pods are Pending, whether due to quota, node capacity, or other constraints.

Next Steps

- Gather community feedback on goals and scope.
- Prototype scheduler plugin for deferred validation.
- Extend the ResourceQuota controller for dual accounting.