Limit Ranges

By default, containers run with unbounded compute resources on a Kubernetes cluster. Using Kubernetes resource quotas, administrators (also termed *cluster operators*) can restrict consumption and creation of cluster resources (such as CPU time, memory, and persistent storage) within a specified namespace. Within a namespace, a Pod can consume as much CPU and memory as is allowed by the ResourceQuotas that apply to that namespace. As a cluster operator, or as a namespace-level administrator, you might also be concerned about making sure that a single object cannot monopolize all available resources within a namespace.

A LimitRange is a policy to constrain the resource allocations (limits and requests) that you can specify for each applicable object kind (such as Pod or PersistentVolumeClaim) in a namespace.

A *LimitRange* provides constraints that can:

- Enforce minimum and maximum compute resources usage per Pod or Container in a namespace.
- Enforce minimum and maximum storage request per PersistentVolumeClaim in a namespace.
- Enforce a ratio between request and limit for a resource in a namespace.
- Set default request/limit for compute resources in a namespace and automatically inject them to Containers at runtime.

Kubernetes constrains resource allocations to Pods in a particular namespace whenever there is at least one LimitRange object in that namespace.

The name of a LimitRange object must be a valid DNS subdomain name.

Constraints on resource limits and requests

- The administrator creates a LimitRange in a namespace.
- Users create (or try to create) objects in that namespace, such as Pods or PersistentVolumeClaims.
- First, the LimitRange admission controller applies default request and limit values for all Pods (and their containers) that do not set compute resource requirements.
- Second, the LimitRange tracks usage to ensure it does not exceed resource minimum, maximum and ratio defined in any LimitRange present in the namespace.
- If you attempt to create or update an object (Pod or PersistentVolumeClaim) that violates a LimitRange constraint, your request to the API server will fail with anHTTP status code `403 Forbidden` and a message explaining the constraint that has been violated.
- If you add a LimitRange in a namespace that applies to compute-related resources such as `cpu` and `memory`, you must specify requests or limits for those values. Otherwise, the system may reject Pod creation.
- LimitRange validations occur only at Pod admission stage, not on running Pods. If you add or modify a LimitRange, the Pods that already exist in that namespace continue unchanged.
- If two or more LimitRange objects exist in the namespace, it is not deterministic which default value will be applied.

LimitRange and admission checks for Pods

A LimitRange does **not** check the consistency of the default values it applies. This means that a default value for the *limit* that is set by LimitRange may be less than the *request* value specified for the container in the spec that a client submits to the API server. If that happens, the final Pod will not be schedulable.

For example, you define a LimitRange with below manifest:

Note:

The following examples operate within the default namespace of your cluster, as the namespace parameter is undefined and the LimitRange scope is limited to the namespace level. This implies that any references or operations within these examples will interact with elements within the default namespace of your cluster. You can override the operating namespace by configuring namespace in the `metadata.namespace` field.

concepts/policy/limit-range/problematic-limit-range.yaml Copy concepts/policy/limit-range/problematic-limit-range.yaml to clipboard

```
apiVersion: v1
kind: LimitRangemetadata:  name: cpu-resource-constraintspec:  limits:  - default: # this section defines default limits      cpu:
```

along with a Pod that declares a CPU resource request of `700m`, but not a limit:

concepts/policy/limit-range/example-conflict-with-limitrange-cpu.yaml
Copy concepts/policy/limit-range/example-conflict-with-limitrange-cpu.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: example-conflict-with-limitrange-cpuspec:  containers:  - name: demo    image: registry.k8s.io/pause:3.8
```

then that Pod will not be scheduled, failing with an error similar to:

```
Pod "example-conflict-with-limitrange-cpu" is invalid: spec.containers[0].resources.requests: Invalid value: "700m": must be less t
```

If you set both `request` and `limit`, then that new Pod will be scheduled successfully even with the same LimitRange in place:

concepts/policy/limit-range/example-no-conflict-with-limitrange-cpu.yaml
Copy concepts/policy/limit-range/example-no-conflict-with-limitrange-cpu.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: example-no-conflict-with-limitrange-cpuspec:  containers:  - name: demo    image: registry.k8s.io/pause:
```

Example resource constraints

Examples of policies that could be created using LimitRange are:

- In a 2 node cluster with a capacity of 8 GiB RAM and 16 cores, constrain Pods in a namespace to request 100m of CPU with a max limit of 500m for CPU and request 200Mi for Memory with a max limit of 600Mi for Memory.
- Define default CPU limit and request to 150m and memory default request to 300Mi for Containers started with no cpu and memory requests in their specs.

In the case where the total limits of the namespace is less than the sum of the limits of the Pods/Containers, there may be contention for resources. In this case, the Containers or Pods will not be created.

Neither contention nor changes to a LimitRange will affect already created resources.

What's next

For examples on using limits, see:

- how to configure minimum and maximum CPU constraints per namespace.
- how to configure minimum and maximum Memory constraints per namespace.
- how to configure default CPU Requests and Limits per namespace.
- how to configure default Memory Requests and Limits per namespace.
- how to configure minimum and maximum Storage consumption per namespace.
- a detailed example on configuring quota per namespace.

Refer to the LimitRanger design document for context and historical information.

Process ID Limits And Reservations

FEATURE STATE: `Kubernetes v1.20 [stable]`

Kubernetes allow you to limit the number of process IDs (PIDs) that a Pod can use. You can also reserve a number of allocatable PIDs for each node for use by the operating system and daemons (rather than by Pods).

Process IDs (PIDs) are a fundamental resource on nodes. It is trivial to hit the task limit without hitting any other resource limits, which can then cause instability to a host machine.

Cluster administrators require mechanisms to ensure that Pods running in the cluster cannot induce PID exhaustion that prevents host daemons (such as the kubelet or kube-proxy, and potentially also the container runtime) from running. In addition, it is important to ensure that PIDs are limited among Pods in order to ensure they have limited impact on other workloads on the same node.

Note:

On certain Linux installations, the operating system sets the PIDs limit to a low default, such as `32768`. Consider raising the value of `/proc/sys/kernel/pid_max`.

You can configure a kubelet to limit the number of PIDs a given Pod can consume. For example, if your node's host OS is set to use a maximum of `262144` PIDs and expect to host less than `250` Pods, one can give each Pod a budget of `1000` PIDs to prevent using up that node's overall number of available PIDs. If the admin wants to overcommit PIDs similar to CPU or memory, they may do so as well with some additional risks. Either way, a single Pod will not be able to bring the whole machine down. This kind of resource limiting helps to prevent simple fork bombs from affecting operation of an entire cluster.

Per-Pod PID limiting allows administrators to protect one Pod from another, but does not ensure that all Pods scheduled onto that host are unable to impact the node overall. Per-Pod limiting also does not protect the node agents themselves from PID exhaustion.

You can also reserve an amount of PIDs for node overhead, separate from the allocation to Pods. This is similar to how you can reserve CPU, memory, or other resources for use by the operating system and other facilities outside of Pods and their containers.

PID limiting is an important sibling to compute resource requests and limits. However, you specify it in a different way: rather than defining a Pod's resource limit in the `.spec` for a Pod, you configure the limit as a setting on the kubelet. Pod-defined PID limits are not currently supported.

Caution:

This means that the limit that applies to a Pod may be different depending on where the Pod is scheduled. To make things simple, it's easiest if all Nodes use the same PID resource limits and reservations.

Node PID limits

Kubernetes allows you to reserve a number of process IDs for the system use. To configure the reservation, use the parameter `pid=<number>` in the `--system-reserved` and `--kube-reserved` command line options to the kubelet. The value you specified declares that the specified number of process IDs will be reserved for the system as a whole and for Kubernetes system daemons respectively.

Pod PID limits

Kubernetes allows you to limit the number of processes running in a Pod. You specify this limit at the node level, rather than configuring it as a resource limit for a particular Pod. Each Node can have a different PID limit.
To configure the limit, you can specify the command line parameter `--pod-max-pids` to the kubelet, or set `PodPidsLimit` in the kubelet configuration file.

PID based eviction

You can configure kubelet to start terminating a Pod when it is misbehaving and consuming abnormal amount of resources. This feature is called eviction. You can Configure Out of Resource Handling for various eviction signals. Use `pid.available` eviction signal to configure the threshold for number of PIDs used by Pod. You can set soft and hard eviction policies. However, even with the hard eviction policy, if the number of PIDs growing very fast, node can still get into unstable state by hitting the node PIDs limit. Eviction signal value is calculated periodically and does NOT enforce the limit.

PID limiting - per Pod and per Node sets the hard limit. Once the limit is hit, workload will start experiencing failures when trying to get a new PID. It may or may not lead to rescheduling of a Pod, depending on how workload reacts on these failures and how liveness and readiness probes are configured for the Pod. However, if limits were set correctly, you can guarantee that other Pods workload and system processes will not run out of PIDs when one Pod is misbehaving.

What's next

- Refer to the PID Limiting enhancement document for more information.
- For historical context, read Process ID Limiting for Stability Improvements in Kubernetes 1.14.
- Read Managing Resources for Containers.
- Learn how to Configure Out of Resource Handling.

Node Resource Managers

In order to support latency-critical and high-throughput workloads, Kubernetes offers a suite of Resource Managers. The managers aim to co-ordinate and optimise the alignment of node's resources for pods configured with a specific requirement for CPUs, devices, and memory (hugepages) resources.

Hardware topology alignment policies

Topology Manager is a kubelet component that aims to coordinate the set of components that are responsible for these optimizations. The overall resource management process is governed using the policy you specify. To learn more, read Control Topology Management Policies on a Node.

Policies for assigning CPUs to Pods

FEATURE STATE: `Kubernetes v1.26 [stable]` (enabled by default: true)

Once a Pod is bound to a Node, the kubelet on that node may need to either multiplex the existing hardware (for example, sharing CPUs across multiple Pods) or allocate hardware by dedicating some resource (for example, assigning one of more CPUs for a Pod's exclusive use).

By default, the kubelet uses CFS quota to enforce pod CPU limits. When the node runs many CPU-bound pods, the workload can move to different CPU cores depending on whether the pod is throttled and which CPU cores are available at scheduling time. Many workloads are not sensitive to this migration and thus work fine without any intervention.

However, in workloads where CPU cache affinity and scheduling latency significantly affect workload performance, the kubelet allows alternative CPU management policies to determine some placement preferences on the node. This is implemented using the *CPU Manager* and its policy. There are two available policies:

- `none`: the `none` policy explicitly enables the existing default CPU affinity scheme, providing no affinity beyond what the OS scheduler does automatically. Limits on CPU usage for Guaranteed pods and Burstable pods are enforced using CFS quota.
- `static`: the `static` policy allows containers in `Guaranteed` pods with integer CPU `requests` access to exclusive CPUs on the node. This exclusivity is enforced using the cpuset cgroup controller.

Note:

System services such as the container runtime and the kubelet itself can continue to run on these exclusive CPUs. The exclusivity only extends to other pods.

CPU Manager doesn't support offlining and onlining of CPUs at runtime.

Static policy

The static policy enables finer-grained CPU management and exclusive CPU assignment. This policy manages a shared pool of CPUs that initially contains all CPUs in the node. The amount of exclusively allocatable CPUs is equal to the total number of CPUs in the node minus any CPU reservations set by the kubelet configuration. CPUs reserved by these options are taken, in integer quantity, from the initial shared pool in ascending order by physical core ID. This shared pool is the set of CPUs on which any containers in `BestEffort` and `Burstable` pods run. Containers in `Guaranteed` pods with fractional CPU `requests` also run on CPUs in the shared pool. Only containers that are part of a `Guaranteed` pod and have integer CPU `requests` are assigned exclusive CPUs.

Note:

The kubelet requires a CPU reservation greater than zero when the static policy is enabled. This is because a zero CPU reservation would allow the shared pool to become empty.

As `Guaranteed` pods whose containers fit the requirements for being statically assigned are scheduled to the node, CPUs are removed from the shared pool and placed in the cpuset for the container. CFS quota is not used to bound the CPU usage of these containers as their usage is bound by the scheduling domain itself. In others words, the number of CPUs in the container cpuset is equal to the integer CPU `limit` specified in the pod spec. This static assignment increases CPU affinity and decreases context switches due to throttling for the CPU-bound workload.

Consider the containers in the following pod specs:

```
spec:
  containers:
```

```
  - name: nginx
    image: nginx
```

The pod above runs in the `BestEffort` QoS class because no resource `requests` or `limits` are specified. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
      requests:
        memory: "100Mi"
```

The pod above runs in the `Burstable` QoS class because resource `requests` do not equal `limits` and the `cpu` quantity is not specified. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
      requests:
        memory: "100Mi"
        cpu: "1"
```

The pod above runs in the `Burstable` QoS class because resource `requests` do not equal `limits`. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
      requests:
        memory: "200Mi"
        cpu: "2"
```

The pod above runs in the `Guaranteed` QoS class because `requests` are equal to `limits`. And the container's resource limit for the CPU resource is an integer greater than or equal to one. The `nginx` container is granted 2 exclusive CPUs.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "1.5"
      requests:
        memory: "200Mi"
        cpu: "1.5"
```

The pod above runs in the `Guaranteed` QoS class because `requests` are equal to `limits`. But the container's resource limit for the CPU resource is a fraction. It runs in the shared pool.

```
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      limits:
        memory: "200Mi"
        cpu: "2"
```

The pod above runs in the `Guaranteed` QoS class because only `limits` are specified and `requests` are set equal to `limits` when not explicitly specified. And the container's resource limit for the CPU resource is an integer greater than or equal to one. The `nginx` container is granted 2 exclusive CPUs.

Static policy options

Here are the available policy options for the static CPU management policy, listed in alphabetical order:

`align-by-socket` (alpha, hidden by default)
: Align CPUs by physical package / socket boundary, rather than logical NUMA boundaries (available since Kubernetes v1.25)

`distribute-cpus-across-cores` (alpha, hidden by default)
: Allocate virtual cores, sometimes called hardware threads, across different physical cores (available since Kubernetes v1.31)

`distribute-cpus-across-numa` (beta, visible by default)
: Spread CPUs across different NUMA domains, aiming for an even balance between the selected domains (available since Kubernetes v1.23)

`full-pcpus-only` (GA, visible by default)
: Always allocate full physical cores (available since Kubernetes v1.22, GA since Kubernetes v1.33)

`strict-cpu-reservation` (beta, visible by default)
: Prevent all the pods regardless of their Quality of Service class to run on reserved CPUs (available since Kubernetes v1.32)

`prefer-align-cpus-by-uncorecache` (beta, visible by default)
: Align CPUs by uncore (Last-Level) cache boundary on a best-effort way (available since Kubernetes v1.32)

You can toggle groups of options on and off based upon their maturity level using the following feature gates:

- `CPUManagerPolicyBetaOptions` (default enabled). Disable to hide beta-level options.
- `CPUManagerPolicyAlphaOptions` (default disabled). Enable to show alpha-level options.

You will still have to enable each option using the `cpuManagerPolicyOptions` field in the kubelet configuration file.

For more detail about the individual options you can configure, read on.

`full-pcpus-only`

If the `full-pcpus-only` policy option is specified, the static policy will always allocate full physical cores. By default, without this option, the static policy allocates CPUs using a topology-aware best-fit allocation. On SMT enabled systems, the policy can allocate individual virtual cores, which correspond to hardware threads. This can lead to different containers sharing the same physical cores; this behaviour in turn contributes to the noisy neighbours problem. With the option enabled, the pod will be admitted by the kubelet only if the CPU request of all its containers can be fulfilled by allocating full physical cores. If the pod does not pass the admission, it will be put in Failed state with the message `SMTAlignmentError`.

`distribute-cpus-across-numa`

If the `distribute-cpus-across-numa`policy option is specified, the static policy will evenly distribute CPUs across NUMA nodes in cases where more than one NUMA node is required to satisfy the allocation. By default, the `CPUManager` will pack CPUs onto one NUMA node until it is filled, with any remaining CPUs simply spilling over to the next NUMA node. This can cause undesired bottlenecks in parallel code relying on barriers (and similar synchronization primitives), as this type of code tends to run only as fast as its slowest worker (which is slowed down by the fact that fewer CPUs are available on at least one NUMA node). By distributing CPUs evenly across NUMA nodes, application developers can more easily ensure that no single worker suffers from NUMA effects more than any other, improving the overall performance of these types of applications.

`align-by-socket`

If the `align-by-socket` policy option is specified, CPUs will be considered aligned at the socket boundary when deciding how to allocate CPUs to a container. By default, the `CPUManager` aligns CPU allocations at the NUMA boundary, which could result in performance degradation if CPUs need to be pulled from more than one NUMA node to satisfy the allocation. Although it tries to ensure that all CPUs are allocated from the *minimum* number of NUMA nodes, there is no guarantee that those NUMA nodes will be on the same socket. By directing the `CPUManager` to explicitly align CPUs at the socket boundary rather than the NUMA boundary, we are able to avoid such issues. Note, this policy option is not compatible with `TopologyManager` `single-numa-node` policy and does not apply to hardware where the number of sockets is greater than number of NUMA nodes.

`distribute-cpus-across-cores`

If the `distribute-cpus-across-cores` policy option is specified, the static policy will attempt to allocate virtual cores (hardware threads) across different physical cores. By default, the `CPUManager` tends to pack CPUs onto as few physical cores as possible, which can lead to contention among CPUs on the same physical core and result in performance bottlenecks. By enabling the `distribute-cpus-across-cores` policy, the static policy ensures that CPUs are distributed across as many physical cores as possible, reducing the contention on the same physical core and thereby improving overall performance. However, it is important to note that this strategy might be less effective when the system is heavily loaded. Under such conditions, the benefit of reducing contention diminishes. Conversely, default behavior can help in reducing inter-core communication overhead, potentially providing better performance under high load conditions.

`strict-cpu-reservation`

The `reservedSystemCPUs` parameter in KubeletConfiguration, or the deprecated kubelet command line option `--reserved-cpus`, defines an explicit CPU set for OS system daemons and kubernetes system daemons. More details of this parameter can be found on the Explicitly Reserved CPU List page. By default, this isolation is implemented only for guaranteed pods with integer CPU requests not for burstable and best-effort pods (and guaranteed pods with fractional CPU requests). Admission is only comparing the CPU requests against the allocatable CPUs. Since the CPU limit is higher than the request, the default behaviour allows burstable and best-effort pods to use up the capacity of `reservedSystemCPUs` and cause host OS services to starve in real life deployments. If the `strict-cpu-reservation` policy option is enabled, the static policy will not allow any workload to use the CPU cores specified in `reservedSystemCPUs`.

`prefer-align-cpus-by-uncorecache`

If the `prefer-align-cpus-by-uncorecache` policy is specified, the static policy will allocate CPU resources for individual containers such that all CPUs assigned to a container share the same uncore cache block (also known as the Last-Level Cache or LLC). By default, the `CPUManager` will tightly pack CPU assignments which can result in containers being assigned CPUs from multiple uncore caches. This option enables the `CPUManager` to allocate CPUs in a way that maximizes the efficient use of the uncore cache. Allocation is performed on a best-effort basis, aiming to affine as many CPUs as possible within the same uncore cache. If the container's CPU requirement exceeds the CPU capacity of a single uncore cache, the `CPUManager` minimizes the number of uncore caches used in order to maintain optimal uncore cache alignment. Specific workloads can benefit in performance from the reduction of inter-cache latency and noisy neighbors at the cache level. If the `CPUManager` cannot align optimally while the node has sufficient resources, the container will still be admitted using the default packed behavior.

Memory Management Policies

FEATURE STATE: `Kubernetes v1.32 [stable]` (enabled by default: true)

The Kubernetes *Memory Manager* enables the feature of guaranteed memory (and hugepages) allocation for pods in the `Guaranteed` QoS class.

The Memory Manager employs hint generation protocol to yield the most suitable NUMA affinity for a pod. The Memory Manager feeds the central manager (*Topology Manager*) with these affinity hints. Based on both the hints and Topology Manager policy, the pod is rejected or admitted to the node.

Moreover, the Memory Manager ensures that the memory which a pod requests is allocated from a minimum number of NUMA nodes.

Other resource managers

The configuration of individual managers is elaborated in dedicated documents:

- Device Manager

Policies

Manage security and best-practices with policies.

Kubernetes policies are configurations that manage other configurations or runtime behaviors. Kubernetes offers various forms of policies, described below:

Apply policies using API objects

Some API objects act as policies. Here are some examples:

- NetworkPolicies can be used to restrict ingress and egress traffic for a workload.
- LimitRanges manage resource allocation constraints across different object kinds.
- ResourceQuotas limit resource consumption for a namespace.

Apply policies using admission controllers

An admission controller runs in the API server and can validate or mutate API requests. Some admission controllers act to apply policies. For example, the AlwaysPullImages admission controller modifies a new Pod to set the image pull policy to `Always`.

Kubernetes has several built-in admission controllers that are configurable via the API server `--enable-admission-plugins` flag.

Details on admission controllers, with the complete list of available admission controllers, are documented in a dedicated section:

- Admission Controllers

Apply policies using ValidatingAdmissionPolicy

Validating admission policies allow configurable validation checks to be executed in the API server using the Common Expression Language (CEL). For example, a `ValidatingAdmissionPolicy` can be used to disallow use of the `latest` image tag.

A `ValidatingAdmissionPolicy` operates on an API request and can be used to block, audit, and warn users about non-compliant configurations.

Details on the `ValidatingAdmissionPolicy` API, with examples, are documented in a dedicated section:

- Validating Admission Policy

Apply policies using dynamic admission control

Dynamic admission controllers (or admission webhooks) run outside the API server as separate applications that register to receive webhooks requests to perform validation or mutation of API requests.

Dynamic admission controllers can be used to apply policies on API requests and trigger other policy-based workflows. A dynamic admission controller can perform complex checks including those that require retrieval of other cluster resources and external data. For example, an image verification check can lookup data from OCI registries to validate the container image signatures and attestations.

Details on dynamic admission control are documented in a dedicated section:

- Dynamic Admission Control

Implementations

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

Dynamic Admission Controllers that act as flexible policy engines are being developed in the Kubernetes ecosystem, such as:

- Kubewarden
- Kyverno
- OPA Gatekeeper
- Polaris

Apply policies using Kubelet configurations

Kubernetes allows configuring the Kubelet on each worker node. Some Kubelet configurations act as policies:

- Process ID limits and reservations are used to limit and reserve allocatable PIDs.
- Node Resource Managers can manage compute, memory, and device resources for latency-critical and high-throughput workloads.

Resource Quotas

When several users or teams share a cluster with a fixed number of nodes, there is a concern that one team could use more than its fair share of resources.

Resource quotas are a tool for administrators to address this concern.

A resource quota, defined by a ResourceQuota object, provides constraints that limit aggregate resource consumption per namespace. A ResourceQuota can also limit the quantity of objects that can be created in a namespace by API kind, as well as the total amount of infrastructure resources that may be consumed by API objects found in that namespace.

Caution:

Neither contention nor changes to quota will affect already created resources.

How Kubernetes ResourceQuotas work

ResourceQuotas work like this:

- Different teams work in different namespaces. This separation can be enforced with RBAC or any other authorization mechanism.
- A cluster administrator creates at least one ResourceQuota for each namespace.
  - To make sure the enforcement stays enforced, the cluster administrator should also restrict access to delete or update that ResourceQuota; for example, by defining a ValidatingAdmissionPolicy.
- Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a ResourceQuota.

  You can apply a scope to a ResourceQuota to limit where it applies,
- If creating or updating a resource violates a quota constraint, the control plane rejects that request with HTTP status code `403 Forbidden`. The error includes a message explaining the constraint that would have been violated.
- If quotas are enabled in a namespace for resource such as `cpu` and `memory`, users must specify requests or limits for those values when they define a Pod; otherwise, the quota system may reject pod creation.

  The resource quota walkthrough shows an example of how to avoid this problem.

Note:

- You can define a LimitRange to force defaults on pods that make no compute resource requirements (so that users don't have to remember to do that).

You often do not create Pods directly; for example, you more usually create a workload management object such as a Deployment. If you create a Deployment that tries to use more resources than are available, the creation of the Deployment (or other workload management object) **succeeds**, but the Deployment may not be able to get all of the Pods it manages to exist. In that case you can check the status of the Deployment, for example with `kubectl describe`, to see what has happened.

- For `cpu` and `memory` resources, ResourceQuotas enforce that **every** (new) pod in that namespace sets a limit for that resource. If you enforce a resource quota in a namespace for either `cpu` or `memory`, you and other clients, **must** specify either `requests` or `limits` for that resource, for every new Pod you submit. If you don't, the control plane may reject admission for that Pod.
- For other resources: ResourceQuota works and will ignore pods in the namespace without setting a limit or request for that resource. It means that you can create a new pod without limit/request for ephemeral storage if the resource quota limits the ephemeral storage of this namespace.

You can use a LimitRange to automatically set a default request for these resources.

The name of a ResourceQuota object must be a valid DNS subdomain name.

Examples of policies that could be created using namespaces and quotas are:

- In a cluster with a capacity of 32 GiB RAM, and 16 cores, let team A use 20 GiB and 10 cores, let B use 10GiB and 4 cores, and hold 2GiB and 2 cores in reserve for future allocation.
- Limit the "testing" namespace to using 1 core and 1GiB RAM. Let the "production" namespace use any amount.

In the case where the total capacity of the cluster is less than the sum of the quotas of the namespaces, there may be contention for resources. This is handled on a first-come-first-served basis.

Enabling Resource Quota

ResourceQuota support is enabled by default for many Kubernetes distributions. It is enabled when the API server `--enable-admission-plugins=` flag has `ResourceQuota` as one of its arguments.

A resource quota is enforced in a particular namespace when there is a ResourceQuota in that namespace.

Types of resource quota

The ResourceQuota mechanism lets you enforce different kinds of limits. This section describes the types of limit that you can enforce.

Quota for infrastructure resources

You can limit the total sum of compute resources that can be requested in a given namespace.

The following resource types are supported:

Resource Name	Description
`limits.cpu`	Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value.
`limits.memory`	Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value.
`requests.cpu`	Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value.
`requests.memory`	Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value.
`hugepages-<size>`	Across all pods in a non-terminal state, the number of huge page requests of the specified size cannot exceed this value.
`cpu`	Same as `requests.cpu`
`memory`	Same as `requests.memory`

Quota for extended resources

In addition to the resources mentioned above, in release 1.10, quota support for extended resources is added.

As overcommit is not allowed for extended resources, it makes no sense to specify both `requests` and `limits` for the same extended resource in a quota. So for extended resources, only quota items with prefix `requests.` are allowed.

Take the GPU resource as an example, if the resource name is `nvidia.com/gpu`, and you want to limit the total number of GPUs requested in a namespace to 4, you can define a quota as follows:

- `requests.nvidia.com/gpu: 4`

See Viewing and Setting Quotas for more details.

Quota for storage

You can limit the total sum of storage for volumes that can be requested in a given namespace.

In addition, you can limit consumption of storage resources based on associated StorageClass.

Resource Name	Description
`requests.storage`	Across all persistent volume claims, the sum of storage requests cannot exceed this value.
`persistentvolumeclaims`	The total number of PersistentVolumeClaims that can exist in the namespace.
`<storage-class-name>.storageclass.storage.k8s.io/requests.storage`	Across all persistent volume claims associated with the `<storage-class-name>`, the sum of storage requests cannot exceed this value.
`<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims`	Across all persistent volume claims associated with the `<storage-class-name>`, the total number of persistent volume claims that can exist in the namespace.

For example, if you want to quota storage with `gold` StorageClass separate from a `bronze` StorageClass, you can define a quota as follows:

- `gold.storageclass.storage.k8s.io/requests.storage: 500Gi`
- `bronze.storageclass.storage.k8s.io/requests.storage: 100Gi`

Quota for local ephemeral storage

FEATURE STATE: `Kubernetes v1.8 [alpha]`

Resource Name	Description
`requests.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage requests cannot exceed this value.
`limits.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage limits cannot exceed this value.
`ephemeral-storage`	Same as `requests.ephemeral-storage`.

Note:

When using a CRI container runtime, container logs will count against the ephemeral storage quota. This can result in the unexpected eviction of pods that have exhausted their storage quotas.

Refer to Logging Architecture for details.

Quota on object count

You can set quota for *the total number of one particular resource kind* in the Kubernetes API, using the following syntax:

- `count/<resource>.<group>` for resources from non-core API groups
- `count/<resource>` for resources from the core API group

For example, the PodTemplate API is in the core API group and so if you want to limit the number of PodTemplate objects in a namespace, you use `count/podtemplates`.

These types of quotas are useful to protect against exhaustion of control plane storage. For example, you may want to limit the number of Secrets in a server given their large size. Too many Secrets in a cluster can actually prevent servers and controllers from starting. You can set a quota for Jobs to protect against a poorly configured CronJob. CronJobs that create too many Jobs in a namespace can lead to a denial of service.

If you define a quota this way, it applies to Kubernetes' APIs that are part of the API server, and to any custom resources backed by a CustomResourceDefinition. For example, to create a quota on a `widgets` custom resource in the `example.com` API group, use `count/widgets.example.com`.

If you use API aggregation to add additional, custom APIs that are not defined as CustomResourceDefinitions, the core Kubernetes control plane does not enforce quota for the aggregated API. The extension API server is expected to provide quota enforcement if that's appropriate for the custom API.

Generic syntax

This is a list of common examples of object kinds that you may want to put under object count quota, listed by the configuration string that you would use.

- `count/pods`
- `count/persistentvolumeclaims`
- `count/services`
- `count/secrets`
- `count/configmaps`
- `count/deployments.apps`
- `count/replicasets.apps`
- `count/statefulsets.apps`
- `count/jobs.batch`
- `count/cronjobs.batch`

Specialized syntax

There is another syntax only to set the same type of quota, that only works for certain API kinds. The following types are supported:

Resource Name	Description
`configmaps`	The total number of ConfigMaps that can exist in the namespace.
`persistentvolumeclaims`	The total number of PersistentVolumeClaims that can exist in the namespace.
`pods`	The total number of Pods in a non-terminal state that can exist in the namespace. A pod is in a terminal state if `.status.phase in (Failed, Succeeded)` is true.
`replicationcontrollers`	The total number of ReplicationControllers that can exist in the namespace.
`resourcequotas`	The total number of ResourceQuotas that can exist in the namespace.
`services`	The total number of Services that can exist in the namespace.
`services.loadbalancers`	The total number of Services of type `LoadBalancer` that can exist in the namespace.
`services.nodeports`	The total number of `NodePorts` allocated to Services of type `NodePort` or `LoadBalancer` that can exist in the namespace.
`secrets`	The total number of Secrets that can exist in the namespace.

For example, `pods` quota counts and enforces a maximum on the number of `pods` created in a single namespace that are not terminal. You might want to set a `pods` quota on a namespace to avoid the case where a user creates many small pods and exhausts the cluster's supply of Pod IPs.

You can find more examples on Viewing and Setting Quotas.

Viewing and Setting Quotas

kubectl supports creating, updating, and viewing quotas:

```
kubectl create namespace myspace
```

```
cat <<EOF > compute-resources.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    requests.cpu: "1"
    requests.memory: "1Gi"
    limits.cpu: "2"
    limits.memory: "2Gi"
    requests.nvidia.com/gpu: 4
EOF
```

```
kubectl create -f ./compute-resources.yaml --namespace=myspace
```

```
cat <<EOF > object-counts.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
spec:
  hard:
    configmaps: "10"
    persistentvolumeclaims: "4"
    pods: "4"
    replicationcontrollers: "20"
    secrets: "10"
    services: "10"
    services.loadbalancers: "2"
EOF
```

```
kubectl create -f ./object-counts.yaml --namespace=myspace
```

```
kubectl get quota --namespace=myspace
```

```
NAME                    AGE
compute-resources       30s
object-counts           32s
```

```
kubectl describe quota compute-resources --namespace=myspace
```

```
Name:                   compute-resources
Namespace:              myspace
Resource                Used  Hard
--------                ----  ----
limits.cpu              0     2
limits.memory           0     2Gi
requests.cpu            0     1
requests.memory         0     1Gi
requests.nvidia.com/gpu 0     4

kubectl describe quota object-counts --namespace=myspace

Name:                   object-counts
Namespace:              myspace
Resource                Used    Hard
--------                ----    ----
configmaps              0       10
persistentvolumeclaims  0       4
pods                    0       4
replicationcontrollers  0       20
secrets                 1       10
services                0       10
services.loadbalancers  0       2
```

kubectl also supports object count quota for all standard namespaced resources using the syntax `count/<resource>.<group>`:

```
kubectl create namespace myspace

kubectl create quota test --hard=count/deployments.apps=2,count/replicasets.apps=4,count/pods=3,count/secrets=4 --namespace=myspac

kubectl create deployment nginx --image=nginx --namespace=myspace --replicas=2

kubectl describe quota --namespace=myspace

Name:                         test
Namespace:                    myspace
Resource                      Used  Hard
--------                      ----  ----
count/deployments.apps        1     2
count/pods                    2     3
count/replicasets.apps        1     4
count/secrets                 1     4
```

Quota and Cluster Capacity

ResourceQuotas are independent of the cluster capacity. They are expressed in absolute units. So, if you add nodes to your cluster, this does *not* automatically give each namespace the ability to consume more resources.

Sometimes more complex policies may be desired, such as:

- Proportionally divide total cluster resources among several teams.
- Allow each tenant to grow resource usage as needed, but have a generous limit to prevent accidental resource exhaustion.
- Detect demand from one namespace, add nodes, and increase quota.

Such policies could be implemented using `ResourceQuotas` as building blocks, by writing a "controller" that watches the quota usage and adjusts the quota hard limits of each namespace according to other signals.

Note that resource quota divides up aggregate cluster resources, but it creates no restrictions around nodes: pods from several namespaces may run on the same node.

Quota scopes

Each quota can have an associated set of `scopes`. A quota will only measure usage for a resource if it matches the intersection of enumerated scopes.

When a scope is added to the quota, it limits the number of resources it supports to those that pertain to the scope. Resources specified on the quota outside of the allowed set results in a validation error.

Kubernetes 1.34 supports the following scopes:

Scope	Description
`BestEffort`	Match pods that have best effort quality of service.
`CrossNamespacePodAffinity`	Match pods that have cross-namespace pod (anti)affinity terms.
`NotBestEffort`	Match pods that do not have best effort quality of service.
`NotTerminating`	Match pods where `.spec.activeDeadlineSeconds`]() is `nil`]()
`PriorityClass`	Match pods that references the specified priority class.
`Terminating`	Match pods where `.spec.activeDeadlineSeconds`]() >= `0`]()
`VolumeAttributesClass`	Match PersistentVolumeClaims that reference the specified volume attributes class.

ResourceQuotas with a scope set can also have a optional `scopeSelector` field. You define one or more *match expressions* that specify an `operators` and, if relevant, a set of `values` to match. For example:

```yaml
  scopeSelector:
    matchExpressions:
      - scopeName: BestEffort # Match pods that have best effort quality of service
        operator: Exists # optional; "Exists" is implied for BestEffort scope
```

The `scopeSelector` supports the following values in the `operator` field:

- `In`
- `NotIn`
- `Exists`
- `DoesNotExist`

If the `operator` is `In` or `NotIn`, the `values` field must have at least one value. For example:

```yaml
scopeSelector:
  matchExpressions:
    - scopeName: PriorityClass
      operator: In
      values:
        - middle
```

If the `operator` is `Exists` or `DoesNotExist`, the `values` field must *NOT* be specified.

Best effort Pods scope

This scope only tracks quota consumed by Pods. It only matches pods that have the best effort QoS class.

The `operator` for a `scopeSelector` must be `Exists`.

Not-best-effort Pods scope

This scope only tracks quota consumed by Pods. It only matches pods that have the Guaranteed or Burstable QoS class.

The `operator` for a `scopeSelector` must be `Exists`.

Non-terminating Pods scope

This scope only tracks quota consumed by Pods that are not terminating. The `operator` for a `scopeSelector` must be `Exists`.

A Pod is not terminating if the `.spec.activeDeadlineSeconds` field is unset.

You can use a ResourceQuota with this scope to manage the following resources:

- `count.pods`
- `pods`
- `cpu`
- `memory`
- `requests.cpu`
- `requests.memory`
- `limits.cpu`
- `limits.memory`

Terminating Pods scope

This scope only tracks quota consumed by Pods that are terminating. The `operator` for a `scopeSelector` must be `Exists`.

A Pod is considered as *terminating* if the `.spec.activeDeadlineSeconds` field is set to any number.

You can use a ResourceQuota with this scope to manage the following resources:

- `count.pods`
- `pods`
- `cpu`
- `memory`
- `requests.cpu`
- `requests.memory`
- `limits.cpu`
- `limits.memory`

Cross-namespace pod affinity scope

FEATURE STATE: `Kubernetes v1.24 [stable]`

You can use `CrossNamespacePodAffinity` quota scope to limit which namespaces are allowed to have pods with affinity terms that cross namespaces. Specifically, it controls which pods are allowed to set `namespaces` or `namespaceSelector` fields in pod (anti)affinity terms.

Preventing users from using cross-namespace affinity terms might be desired since a pod with anti-affinity constraints can block pods from all other namespaces from getting scheduled in a failure domain.

Using this scope, you (as a cluster administrator) can prevent certain namespaces - such as `foo-ns` in the example below - from having pods that use cross-namespace pod affinity. You configure this creating a ResourceQuota object in that namespace with `CrossNamespacePodAffinity` scope and hard limit of 0:

```yaml
apiVersion: v1
kind: ResourceQuotametadata:  name: disable-cross-namespace-affinity  namespace: foo-nsspec:  hard:    pods: "0"  scopeSelector:
```

If you want to disallow using `namespaces` and `namespaceSelector` by default, and only allow it for specific namespaces, you could configure `CrossNamespacePodAffinity` as a limited resource by setting the kube-apiserver flag `--admission-control-config-file` to the path of the following

configuration file:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfigurationplugins:- name: "ResourceQuota"  configuration:    apiVersion: apiserver.config.k8s.io/v1    kind: Res
```

With the above configuration, pods can use `namespaces` and `namespaceSelector` in pod affinity only if the namespace where they are created have a resource quota object with `CrossNamespacePodAffinity` scope and a hard limit greater than or equal to the number of pods using those fields.

PriorityClass scope

FEATURE STATE: `Kubernetes v1.17 [stable]`

A ResourceQuota with a PriorityClass scope only matches Pods that have a particular priority class, and only if any `scopeSelector` in the quota spec selects a particular Pod.

Pods can be created at a specific priority. You can control a pod's consumption of system resources based on a pod's priority, by using the `scopeSelector` field in the quota spec.

When quota is scoped for PriorityClass using the `scopeSelector` field, the ResourceQuota can only track (and limit) the following resources:

- `pods`
- `cpu`
- `memory`
- `ephemeral-storage`
- `limits.cpu`
- `limits.memory`
- `limits.ephemeral-storage`
- `requests.cpu`
- `requests.memory`
- `requests.ephemeral-storage`

Example

This example creates a ResourceQuota matches it with pods at specific priorities. The example works as follows:

- Pods in the cluster have one of the three PriorityClasses, "low", "medium", "high".
  - If you want to try this out, use a testing cluster and set up those three PriorityClasses before you continue.
- One quota object is created for each priority.

Inspect this set of ResourceQuotas:

policy/quota.yaml Copy policy/quota.yaml to clipboard

```
apiVersion: v1
kind: ResourceQuotametadata:  name: pods-highspec:  hard:    cpu: "1000"    memory: "200Gi"    pods: "10"  scopeSelector:    matchl
```

Apply the YAML using `kubectl create`.

```
kubectl create -f https://k8s.io/examples/policy/quota.yaml
```

```
resourcequota/pods-high created
resourcequota/pods-medium created
resourcequota/pods-low created
```

Verify that `Used` quota is `0` using `kubectl describe quota`.

```
kubectl describe quota
```

```
Name:       pods-high
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     1k
memory      0     200Gi
pods        0     10


Name:       pods-low
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     5
memory      0     10Gi
pods        0     10


Name:       pods-medium
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     10
memory      0     20Gi
pods        0     10
```

Create a pod with priority "high".

policy/high-priority-pod.yaml Copy policy/high-priority-pod.yaml to clipboard

```
apiVersion: v1
```

```
kind: Podmetadata:  name: high-priorityspec:  containers:  - name: high-priority    image: ubuntu    command: ["/bin/sh"]    args:
```

To create the Pod:

```
kubectl create -f https://k8s.io/examples/policy/high-priority-pod.yaml
```

Verify that "Used" stats for "high" priority quota, `pods-high`, has changed and that the other two quotas are unchanged.

```
kubectl describe quota
```

```
Name:       pods-high
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         500m  1k
memory      10Gi  200Gi
pods        1     10


Name:       pods-low
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     5
memory      0     10Gi
pods        0     10


Name:       pods-medium
Namespace:  default
Resource    Used  Hard
--------    ----  ----
cpu         0     10
memory      0     20Gi
pods        0     10
```

Limiting PriorityClass consumption by default

It may be desired that pods at a particular priority, such as "cluster-services", should be allowed in a namespace, if and only if, a matching quota object exists.

With this mechanism, operators are able to restrict usage of certain high priority classes to a limited number of namespaces and not every namespace will be able to consume these priority classes by default.

To enforce this, `kube-apiserver` flag `--admission-control-config-file` should be used to pass path to the following configuration file:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfigurationplugins:- name: "ResourceQuota"  configuration:    apiVersion: apiserver.config.k8s.io/v1    kind: Res
```

Then, create a resource quota object in the `kube-system` namespace:

policy/priority-class-resourcequota.yaml Copy policy/priority-class-resourcequota.yaml to clipboard

```
apiVersion: v1
kind: ResourceQuotametadata:  name: pods-cluster-servicesspec:  scopeSelector:    matchExpressions:      - operator : In        sc
```

```
kubectl apply -f https://k8s.io/examples/policy/priority-class-resourcequota.yaml -n kube-system
```

```
resourcequota/pods-cluster-services created
```

In this case, a pod creation will be allowed if:

1. the Pod's `priorityClassName` is not specified.
2. the Pod's `priorityClassName` is specified to a value other than `cluster-services`.
3. the Pod's `priorityClassName` is set to `cluster-services`, it is to be created in the `kube-system` namespace, and it has passed the resource quota check.

A Pod creation request is rejected if its `priorityClassName` is set to `cluster-services` and it is to be created in a namespace other than `kube-system`.

VolumeAttributesClass scope

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

This scope only tracks quota consumed by PersistentVolumeClaims.

PersistentVolumeClaims can be created with a specific VolumeAttributesClass, and might be modified after creation. You can control a PVC's consumption of storage resources based on the associated VolumeAttributesClasses, by using the `scopeSelector` field in the quota spec.

The PVC references the associated VolumeAttributesClass by the following fields:

- `spec.volumeAttributesClassName`
- `status.currentVolumeAttributesClassName`
- `status.modifyVolumeStatus.targetVolumeAttributesClassName`

A relevant ResourceQuota is matched and consumed only if the ResourceQuota has a `scopeSelector` that selects the PVC.

When the quota is scoped for the volume attributes class using the `scopeSelector` field, the quota object is restricted to track only the following resources:

- `persistentvolumeclaims`

- `requests.storage`

Read Limit Storage Consumption to learn more about this.

What's next

- See a detailed example for how to use resource quota.
- Read the ResourceQuota API reference
- Learn about LimitRanges
- You can read the historical ResourceQuota design document for more information.
- You can also read the Quota support for priority class design document.