Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 20, 2026

Pod Admission and Resource Controls: Requests, Limits, LimitRange, and ResourceQuota

Understand the full Kubernetes resource-control path: size containers with requests and limits, apply namespace defaults and guardrails with LimitRange, and cap aggregate consumption with ResourceQuota.



This is Learn post 10 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Earlier posts built Pods and controllers, then supplied application configuration. This lesson adds the resource contract that every shared cluster needs:

how much compute a container needs, how much it may consume, and how a namespace prevents one workload from taking everything.

Kubernetes makes three separate decisions. Admission decides whether the Pod is allowed to exist. Scheduling decides which node can accommodate it. Runtime enforcement constrains the running containers. Requests, limits, LimitRange, and ResourceQuota connect those decisions, but each solves a different problem.

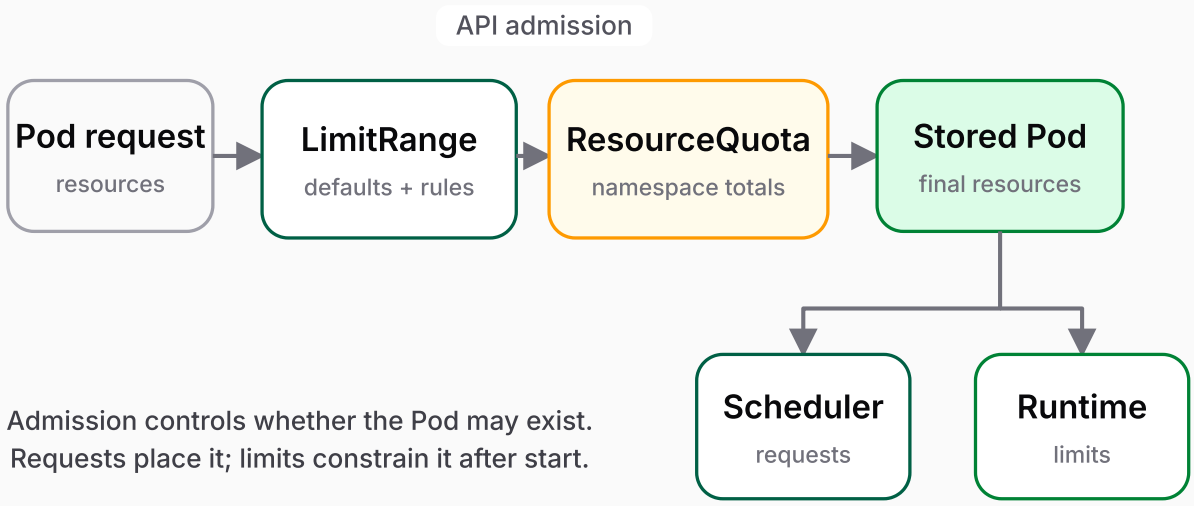
What you'll learn

- Read CPU and memory requests as placement inputs, and limits as runtime boundaries.
- Predict the different outcomes of exceeding a CPU limit and a memory limit.
- Use a LimitRange to default and constrain individual containers in a namespace.
- Use a ResourceQuota to cap aggregate requests, limits, and object counts across a namespace.
- Observe defaulting, quota usage, admission rejection, and controller behavior with kubectl.

The mental model: promise, ceiling, policy

> Request = scheduling promise. Limit = runtime ceiling. LimitRange = per-object policy. ResourceQuota = namespace budget.

A Pod creation request first passes API validation and admission. A LimitRange can add defaults and reject values outside its rules. A ResourceQuota can reject the request when accepting it would exceed namespace totals. Only an admitted Pod is stored, scheduled, and eventually started.



Resource controls act at different stages

Admission answers “may this Pod exist?” Requests later help the scheduler answer “where can it fit?” Limits guide runtime enforcement after the container starts.

These controls do not measure the same thing. Admission compares declared values with policy; the scheduler compares declared requests with node allocatable capacity; the operating system constrains actual usage. Keeping those stages separate prevents most resource-control confusion.

Requests describe the workload's scheduling footprint

A container request states the amount of a resource Kubernetes should plan for. For a normal multi-container Pod, think of the Pod request as the sum of its application containers' requests. The scheduler places the Pod only on a node where those requests fit within the node's remaining allocatable CPU, memory, and other requested resources.

The scheduler uses declared requests, not current utilization. A nearly idle node can still reject a Pod when existing Pods have already requested its allocatable capacity. Conversely, a container may use more than its request when spare resources and its limits allow it.

Regular init containers run sequentially, so Kubernetes does not add all of their requests together. For each resource, scheduling accounts for the larger of the application-container sum and the largest regular init-container request. This connects the init-container lifecycle from post 04 to the Pod's real scheduling footprint.

> A request is accounting, not an exclusive slice of hardware. It influences placement and contention behavior; it does not stop the container at that value.

On Linux, CPU requests also commonly influence relative CPU weight when containers compete. Memory requests are primarily scheduling inputs and also influence how Kubernetes treats Pods during node memory pressure. Detailed scheduling comes next in post 11; broader failure signals and resource usage arrive in post 36.

Limits describe the runtime boundary

When the kubelet starts a container, it passes resource settings to the container runtime. On Linux, the runtime normally configures control groups (cgroups), and the kernel enforces the effective limits. CPU and memory limits therefore produce different symptoms because CPU time can be delayed, while allocated memory cannot be reclaimed safely on demand from an arbitrary process.

- CPU limit: after the container consumes its allowed CPU time, the kernel throttles it until more CPU time becomes available. The process usually keeps running, but it may become slower. Exceeding a CPU limit does not normally terminate the container.
- Memory limit: if the cgroup cannot satisfy further memory allocation, the kernel's out-of-memory handling may terminate a process in the container. If the main process exits, Kubernetes reports a terminated container—commonly with reason OOMKilled—and applies the Pod's restart policy.

Memory enforcement is reactive: the kernel acts when pressure from allocations reaches the boundary. Do not expect a graceful pause analogous to CPU throttling.

Read resource quantities correctly

CPU is measured in CPU units. 1 means one logical CPU; 500m means 500 millicpu, or half a CPU. Memory is measured in bytes and is normally written with binary suffixes such as Mi and Gi. Suffix case matters: 400Mi is a sensible memory value, while 400m means 0.4 bytes.

Official resource-management documentation is the reference for supported resource fields, units, scheduling behavior, and runtime enforcement.

Declare a realistic resource contract

The following Deployment gives each web container a small scheduling footprint and room to burst. Because there are two replicas, their combined declared request is 200m CPU and 128Mi memory; their combined limit is 600m CPU and 256Mi memory.

web.yaml · yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: cka-resources
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: nginx
          image: nginx:1.27.5
          resources:
            requests:
              cpu: 100m
              memory: 64Mi
            limits:
              cpu: 300m
              memory: 128Mi
```

The limit must not be lower than the request for the same resource. Explicit values make the Pod template portable and reviewable. If you specify only a limit and no admission mechanism supplies a request, Kubernetes generally copies that limit as the request; declare both when they express different intentions.

LimitRange sets defaults and per-object guardrails

A LimitRange is namespaced policy. For type Container, it can inject default requests and limits into containers that omit them, then validate minimums, maximums, and optional limit-to-request ratios. It answers questions about each admitted object; it does not maintain an aggregate namespace total.

This policy gives omitted resources safe demonstration defaults and prevents any one container from declaring more than 1 CPU or 512Mi memory. The minimum prevents misleadingly tiny declarations.

namespace-policy.yaml · yaml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: cka-resources
---
apiVersion: v1
kind: LimitRange
metadata:
  name: container-guardrails
  namespace: cka-resources
spec:
  limits:
    - type: Container
      defaultRequest:
        cpu: 100m
        memory: 64Mi
      default:
        cpu: 500m
        memory: 256Mi
      min:
        cpu: 50m
        memory: 32Mi
      max:
        cpu: "1"
        memory: 512Mi
---
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-budget
  namespace: cka-resources
spec:
  hard:
    pods: "5"
    requests.cpu: 1200m
    requests.memory: 1Gi
    limits.cpu: "3"
    limits.memory: 2Gi
```

defaultRequest and default are mutations: the admitted Pod is stored with the injected values. min and max are validations: a violating creation or update is rejected. A new or changed LimitRange does not rewrite Pods that already exist, because its checks happen during admission.

Official LimitRange documentation covers the supported constraint types and admission behavior.

ResourceQuota caps the namespace total

A ResourceQuota is also namespaced, but it aggregates consumption across objects. In this example, all non-terminal Pods together may request at most 1200m CPU and 1Gi memory, declare at most 3 CPU and 2Gi memory in limits, and number at most five Pods.

Quota checks declarations, not live CPU or memory consumption. requests.cpu sums requests; limits.memory sums limits; pods counts non-terminal Pods. Object-count quotas can also protect API resources such as ConfigMaps, Secrets, Services, or Jobs, but the compute-and-Pod budget is the focus here.

> LimitRange asks “is this container declaration acceptable?” ResourceQuota asks “does this namespace still have budget for it?”

A quota does not reserve matching physical capacity and does not make a Pod schedulable. Namespace quotas can add up to more than cluster capacity; the scheduler still decides whether a particular node can fit an admitted Pod.

When a quota tracks requests or limits for a resource, new Pods need corresponding declarations. A LimitRange with defaultRequest and default complements that quota by filling omitted values before quota accounting, as the defaulted Pod will show.

Official ResourceQuota documentation lists quota keys for compute resources, storage, and object counts.

Watch admission default and account for resources

Apply the namespace policies before the workload. Then create one Pod without a resources section. This is a demonstration of defaulting, not a recommendation to hide resource intent in ordinary workload manifests.

```bash
kubectl apply -f namespace-policy.yaml
kubectl apply -f web.yaml
kubectl rollout status deployment/web -n cka-resources

kubectl run defaulted \
  --image=nginx:1.27.5 \
  --restart=Never \
  --namespace=cka-resources
```

Read the stored Pod rather than assuming admission changed it. The custom columns keep the important fields visible without printing the full object.

```bash
kubectl get pod defaulted -n cka-resources \
  -o custom-columns='POD:.metadata.name,CPU-REQUEST:.spec.containers[0].resources.requests.cpu,CPU-LIMIT:.spec.containers[0].resources.limits.cpu,MEMORY-REQUEST:.spec.containers[0].resources.requests.memory,MEMORY-LIMIT:.spec.containers[0].resources.limits.memory'
```

```plaintext
POD        CPU-REQUEST   CPU-LIMIT   MEMORY-REQUEST   MEMORY-LIMIT
defaulted  100m          500m        64Mi             256Mi
```

The submitted Pod omitted resources, but the stored Pod contains all four values. LimitRange supplied them during admission, so the scheduler and quota system now see concrete requests and limits.

Inspect hard limits and current use together

```bash
kubectl describe limitrange container-guardrails -n cka-resources
kubectl describe resourcequota team-budget -n cka-resources
```

```plaintext
Relevant quota rows after two Deployment Pods and one defaulted Pod · plaintext

Resource         Used    Hard
--------         ----    ----
limits.cpu       1100m   3
limits.memory    512Mi   2Gi
pods             3       5
requests.cpu     300m    1200m
requests.memory  192Mi   1Gi
```

The arithmetic matches the stored declarations. Two web Pods each contribute 100m/64Mi requests and 300m/128Mi limits. The defaulted Pod contributes 100m/64Mi requests and 500m/256Mi limits. Used changes as qualifying objects are admitted or deleted; it is not a live utilization reading.

See how controllers meet an admission boundary

Scale the Deployment to six replicas. The Deployment update itself is valid, but its ReplicaSet must create Pods one by one. With the standalone defaulted Pod already present, the namespace can admit only four web Pods before reaching the five-Pod quota.

```bash
kubectl scale deployment/web -n cka-resources --replicas=6

kubectl get deployment web -n cka-resources
kubectl get pods -n cka-resources
kubectl describe replicaset -n cka-resources -l app=web
```

```plaintext
NAME   READY   UP-TO-DATE   AVAILABLE   AGE
web    4/6     4            4           2m

Warning  FailedCreate  replicaset-controller
Error creating: pods is forbidden: exceeded quota: team-budget,
requested: pods=1, used: pods=5, limited: pods=5
```

The desired replica count remains six, so the controller continues trying to reconcile. Admission rejects only the Pod creations that exceed quota. This is why an administrator checks the owning controller's events when a Deployment has fewer Pods than desired; the Deployment object can exist even when some child Pods cannot be admitted.

Return the Deployment to two replicas so the namespace is back within its original demonstration state.

```bash
kubectl scale deployment/web -n cka-resources --replicas=2
kubectl rollout status deployment/web -n cka-resources
```

A per-container violation fails earlier

This Pod declares a 2-CPU request and limit, above the LimitRange maximum of 1 CPU per container. The API server rejects it during admission; no Pending Pod is stored for the scheduler to examine.

oversized.yaml · yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: oversized
  namespace: cka-resources
spec:
  containers:
    - name: nginx
      image: nginx:1.27.5
      resources:
        requests:
          cpu: "2"
          memory: 64Mi
        limits:
          cpu: "2"
          memory: 128Mi
```

bash

```bash
kubectl apply -f oversized.yaml
```

Representative rejection; exact wording can vary · plaintext

```
Error from server (Forbidden): maximum cpu usage per Container is 1, but limit is 2
```

The distinction is operationally useful: admission rejection means the object was not created; scheduling failure means an admitted Pod exists but remains Pending; runtime enforcement appears only after a container has started.

Clean up the demonstration

bash

```bash
kubectl delete namespace cka-resources
```

Deleting the namespace removes the Deployment, Pods, LimitRange, and ResourceQuota together. Both policy resources are namespace-scoped, so the demonstration never affects workloads elsewhere.