**What are Resource Quotas and Limits?**

In Kubernetes, **Resource Quotas** and **Resource Limits** are ways to **control how much CPU, memory, or other resources your pods and namespaces can use**.

They help you:

* Prevent a single pod or team from consuming all cluster resources.
* Ensure fair resource distribution.
* Avoid out-of-memory (OOM) or CPU throttling issues.

**2️Key Concepts**

**a) Resource Limits (Pod/Container Level)**

* Set on **containers inside pods**.
* Specify **maximum** and **minimum guaranteed** resources for CPU and memory.

There are **two fields**:

| **Field** | **Description** |
| --- | --- |
| requests | Minimum resources the container is guaranteed to get. Scheduler uses this to place pods on nodes. |
| limits | Maximum resources the container can use. Container is throttled if it exceeds this. |

**Example:**

apiVersion: v1

kind: Pod

metadata:

name: nginx-pod

spec:

containers:

- name: nginx

image: nginx:latest

resources:

requests:

memory: "128Mi"

cpu: "250m"

limits:

memory: "256Mi"

cpu: "500m"

✅ Explanation:

* requests: Scheduler ensures **128Mi memory & 0.25 CPU** are available.
* limits: Pod **cannot use more than 256Mi memory & 0.5 CPU**.

**b) Resource Quotas (Namespace Level)**

* Set on a **namespace** to limit the **total resources** all pods in that namespace can consume.
* You can limit: CPU, memory, number of pods, number of services, configmaps, PVCs, etc.

**Example:**

apiVersion: v1

kind: ResourceQuota

metadata:

name: project-quota

namespace: dev

spec:

hard:

pods: "10" # Max 10 pods in this namespace

requests.cpu: "2" # Total CPU requests in this namespace

requests.memory: "4Gi" # Total memory requests in this namespace

limits.cpu: "4" # Total CPU limits

limits.memory: "8Gi" # Total memory limits

✅ Explanation:

* Any pod in the dev namespace **cannot exceed these quotas collectively**.
* For example, if 5 pods request 1 CPU each, the next pod cannot request 1 CPU because requests.cpu maxed out at 2 CPUs.

**3️How Resource Requests and Limits Work Together**

* **Requests:** Help **scheduler decide node placement**.
* **Limits:** **Enforce maximum usage**; container is throttled if exceeded.
* If a pod exceeds memory **limit**, it may be **killed (OOMKilled)**.

**CPU Example:**

* Pod requests 100m CPU and limits 200m CPU:
  + Scheduler ensures at least 100m CPU is available.
  + Pod can burst up to 200m CPU if node has spare capacity.

**4️Types of Resource Quotas**

| **Quota Type** | **Description** |
| --- | --- |
| hard | Absolute limit (max resources) |
| used | Shows current usage in namespace (read-only) |
| scopeSelector | Apply quota only to certain pods (e.g., BestEffort, HighPriority) |
| scopes | Restrict quotas based on pod type (e.g., Terminating, NotTerminating) |

**Example of Scope:**

apiVersion: v1

kind: ResourceQuota

metadata:

name: high-memory-pods

namespace: dev

spec:

hard:

requests.memory: "4Gi"

scopes:

- NotTerminating

* Applies quota **only to running pods**, ignores terminating pods.

**5️Best Practices**

1. Always define **requests & limits** for production workloads.
2. Use **ResourceQuotas per namespace** in multi-team clusters.
3. Monitor **OOMKilled events** in logs to adjust limits.
4. Use **Vertical Pod Autoscaler** for dynamic adjustments.
5. Avoid setting **limits too high** — can starve other pods.

**6️⃣ Commands to Check Resource Usage**

# Check ResourceQuota in a namespace

kubectl get quota -n <namespace>

# Describe quota to see details

kubectl describe quota -n <namespace>

# Check pod resource requests/limits

kubectl get pod <pod-name> -o yaml | grep -A 5 resources

**7️⃣ Example: Deployment with Requests & Limits**

apiVersion: apps/v1

kind: Deployment

metadata:

name: webapp

namespace: dev

spec:

replicas: 3

selector:

matchLabels:

app: webapp

template:

metadata:

labels:

app: webapp

spec:

containers:

- name: webapp

image: nginx

resources:

requests:

cpu: "100m"

memory: "128Mi"

limits:

cpu: "250m"

memory: "256Mi"

✅ This ensures each pod is guaranteed 100m CPU and 128Mi memory and cannot exceed 250m CPU or 256Mi memory.

**What Are Probes in Kubernetes?**

**Probes** are **checks Kubernetes performs on your containers** to see if they are healthy or ready to serve traffic.

They help Kubernetes:

* Restart containers that are not working correctly.
* Avoid sending traffic to containers that are not ready.
* Improve reliability and stability of applications.

There are **three types of probes**:

| **Probe Type** | **Purpose** |
| --- | --- |
| **Liveness Probe** | Checks if the container is **alive**. If it fails, the container is restarted. |
| **Readiness Probe** | Checks if the container is **ready to serve traffic**. If it fails, the pod is removed from the Service endpoints. |
| **Startup Probe** | Checks if the container **has started successfully**. Useful for slow-starting apps. |

**2️⃣ Liveness Probe**

* Detects if a container is **stuck or dead**.
* Kubernetes **restarts the container** if the probe fails.

**Example:**

livenessProbe:

httpGet:

path: /healthz

port: 8080

initialDelaySeconds: 10

periodSeconds: 5

failureThreshold: 3

✅ Explanation:

* httpGet: Calls the /healthz endpoint on port 8080.
* initialDelaySeconds: Wait 10s before starting checks.
* periodSeconds: Check every 5s.
* failureThreshold: Restart the container after 3 consecutive failures.

**3️ Readiness Probe**

* Determines if a container is **ready to accept traffic**.
* If it fails, the pod is **removed from the Service endpoints**.

**Example:**

readinessProbe:

tcpSocket:

port: 3306

initialDelaySeconds: 5

periodSeconds: 10

✅ Explanation:

* tcpSocket: Checks if the TCP port 3306 is open.
* Pod only receives traffic if probe succeeds.

**4️Startup Probe**

* Useful for **slow-starting applications**.
* If it fails repeatedly, Kubernetes restarts the container.
* Once it succeeds, **readiness and liveness probes are activated**.

**Example:**

startupProbe:

exec:

command:

- cat

- /tmp/healthy

initialDelaySeconds: 5

periodSeconds: 5

failureThreshold: 12

✅ Explanation:

* Checks if /tmp/healthy file exists using exec.
* Pod is considered started after successful probe.

**5️Types of Probe Handlers**

1. **HTTP GET:** Checks a URL in the container.
2. **TCP Socket:** Checks if a TCP port is open.
3. **Exec:** Runs a command inside the container.

**6️Example Deployment with Probes**

apiVersion: apps/v1

kind: Deployment

metadata:

name: myapp

namespace: dev

spec:

replicas: 2

selector:

matchLabels:

app: myapp

template:

metadata:

labels:

app: myapp

spec:

containers:

- name: myapp

image: myapp:latest

ports:

- containerPort: 8080

livenessProbe:

httpGet:

path: /healthz

port: 8080

initialDelaySeconds: 10

periodSeconds: 5

readinessProbe:

httpGet:

path: /ready

port: 8080

initialDelaySeconds: 5

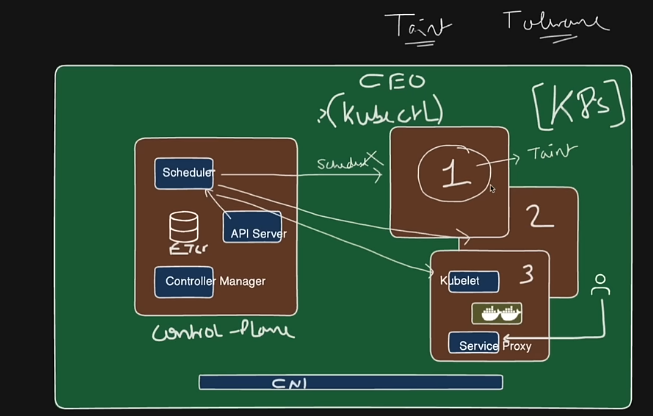
periodSeconds: 5

✅ Key Points:

* Liveness → restarts unhealthy pods.
* Readiness → controls traffic flow.
* Both work independently but complement each other.

**7️Why Probes Matter**

* Avoid sending traffic to a pod that’s **not ready yet**.
* Automatically **recover from crashes or deadlocks**.
* Improve **application reliability** and **cluster stability**.



**Taints and Tolerations**

**Purpose:**

* Control **which pods can be scheduled** on which nodes.
* Prevent pods from running on certain nodes unless they **tolerate the taint**.

**Taints:** Applied on **nodes**. Syntax:

kubectl taint nodes <node-name> key=value:effect

* **effect** can be:
  + NoSchedule → Pod won’t schedule unless it tolerates the taint
  + PreferNoSchedule → Scheduler tries not to schedule, but may if needed
  + NoExecute → Evict existing pods and prevent new pods from scheduling

**Example:**

kubectl taint nodes worker1 key=value:NoSchedule

**Tolerations:** Applied on **pods** to allow them to be scheduled on tainted nodes.

spec:

tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

**Use Case:**

* Dedicated nodes for high-memory or GPU workloads.
* Prevent normal workloads from running on special nodes.

 cordon = security guard at the gate saying *“No one new can enter.”*

 taint = security guard saying *“Only people with a VIP pass (toleration) can enter.”*

**Kubernetes Metrics Server – Setup & Usage**

**🔹 What is Metrics Server?**

* **Metrics Server** is a cluster-wide aggregator of resource usage data.
* It collects CPU and memory usage from **Kubelets** on each node and exposes it via the **Metrics API**.
* It is **not a long-term storage solution** (like Prometheus).
* Commonly used by:
  + kubectl top nodes and kubectl top pods
  + **Horizontal Pod Autoscaler (HPA)**
  + **Vertical Pod Autoscaler (VPA)**

**🔹 Step 1: Install Metrics Server**

Run the official YAML from Kubernetes SIGs:

kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml

✅ This creates:

* Deployment (metrics-server)
* Service
* RBAC (ClusterRoles & Bindings)
* APIService (metrics.k8s.io)

**🔹 Step 2: Verify Installation**

Check pods in kube-system:

kubectl get pods -n kube-system

Example output:

NAME READY STATUS RESTARTS AGE

metrics-server-7db4d65c7b-xh8wq 1/1 Running 0 1m

If STATUS is **CrashLoopBackOff** or **Error**, you may need to add arguments to allow insecure TLS (in Kind, Minikube, or private clusters).

**🔹 Step 3: (Optional) Edit Deployment for Insecure TLS**

Sometimes Metrics Server cannot talk to kubelets due to TLS or certificate issues.  
To fix, edit the Deployment:

kubectl -n kube-system edit deployment metrics-server

Add these args under the container:

containers:

- name: metrics-server

image: k8s.gcr.io/metrics-server/metrics-server:v0.6.4

args:

- --kubelet-insecure-tls

- --kubelet-preferred-address-types=InternalIP,Hostname,ExternalIP

Then restart it:

kubectl -n kube-system rollout restart deployment metrics-server

**🔹 Step 4: Test Metrics API**

Check node metrics:

kubectl top nodes

Example:

NAME CPU(cores) CPU% MEMORY(bytes) MEMORY%

himanshu-control-plane 559m 6% 812Mi 16%

himanshu-worker 82m 1% 223Mi 4%

himanshu-worker2 85m 1% 322Mi 6%

himanshu-worker3 54m 0% 193Mi 3%

Check pod metrics:

kubectl top pods -A

**Horizontal Pod Autoscaler (HPA)**

**📘 Concept**

HPA automatically scales the **number of pods** in a Deployment, ReplicaSet, or StatefulSet based on observed **CPU, memory, or custom metrics**.

**⚙️ Example**

Let’s say you have an Apache web server and want it to scale between 1 and 5 pods when CPU usage goes above 50%.

**HPA.yml**

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: apache-hpa

namespace: scaling

spec:

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: apache

minReplicas: 1

maxReplicas: 5

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 50

**🧠 How it works**

* HPA monitors CPU usage of each pod.
* If average CPU > 50%, it increases replicas.
* If usage < 50%, it reduces replicas.

**🔍 Commands**

kubectl apply -f HPA.yml

kubectl get hpa -n scaling

kubectl describe hpa apache-hpa -n scaling

**Vertical Pod Autoscaler (VPA)**

**📘 Concept**

VPA automatically adjusts **CPU and memory requests/limits** for your pods to match actual usage — **it doesn’t add more pods** like HPA does.

**⚙️ Example**

apiVersion: autoscaling.k8s.io/v1

kind: VerticalPodAutoscaler

metadata:

name: apache-vpa

namespace: scaling

spec:

targetRef:

apiVersion: "apps/v1"

kind: Deployment

name: apache

updatePolicy:

updateMode: "Auto" # Options: Off, Initial, Auto

**🧠 How it works**

* VPA Recommender monitors CPU/memory usage.
* If a pod is under-provisioned or over-provisioned, VPA updates its resource requests.
* “Auto” mode lets it **restart pods** with updated resources.

**🔍 Commands**

kubectl get vpa -n scaling

kubectl describe vpa apache-vpa -n scaling

**Node Affinity**

**📘 Concept**

Node Affinity controls **which nodes** a pod is scheduled on based on node labels.  
It’s like “run this pod only on certain types of nodes.”

**⚙️ Example**

Label your nodes first:

kubectl label nodes himanshu-worker type=frontend

kubectl label nodes himanshu-worker2 type=backend

Then create a pod with Node Affinity:

apiVersion: apps/v1

kind: Deployment

metadata:

name: apache

namespace: scaling

spec:

replicas: 2

selector:

matchLabels:

app: apache

template:

metadata:

labels:

app: apache

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: type

operator: In

values:

- frontend

containers:

- name: apache

image: httpd

**🧠 How it works**

* Pods are only scheduled on nodes with type=frontend.
* If no matching node exists, pods stay in **Pending** state.

**Role-Based Access Control (RBAC)**

**📘 Concept**

RBAC manages **who can do what** in the Kubernetes cluster — it assigns roles and permissions to users or service accounts.

**⚙️ Example**

Let’s create a **Role** that allows only pod viewing, and bind it to a ServiceAccount.

**rbac.yml**

apiVersion: v1

kind: ServiceAccount

metadata:

name: dev-user

namespace: scaling

---

apiVersion: rbac.authorization.k8s.io/v1

kind: Role

metadata:

namespace: scaling

name: pod-viewer

rules:

- apiGroups: [""]

resources: ["pods"]

verbs: ["get", "list", "watch"]

---

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: pod-viewer-binding

namespace: scaling

subjects:

- kind: ServiceAccount

name: dev-user

namespace: scaling

roleRef:

kind: Role

name: pod-viewer

apiGroup: rbac.authorization.k8s.io

**🧠 How it works**

* ServiceAccount → identity for automation or users.
* Role → defines allowed actions.
* RoleBinding → assigns the role to a user/service account.

**🔍 Commands**

kubectl apply -f rbac.yml

kubectl auth can-i get pods --as system:serviceaccount:scaling:dev-user -n scaling