Controlling workloads

Namespaces and capabilities

Namespaces

Namespaces are well suited for carving out environments within the cluster like development, staging, and production

network policies, RBAC policies, and quotas

Namespaces - builtin

default: Adding an object to a cluster without providing a namespace will place it within the default namespace. It cannot be deleted.

The **kube-public** namespace is intended to be globally readable to all users with or without authentication.

The **kube-system** namespace is used for Kubernetes components managed by Kubernetes. It is intended to be managed directly by the system and as such, it has fairly permissive policies.

Create a namespace

```
kubectl create namespace demo-namespace
```

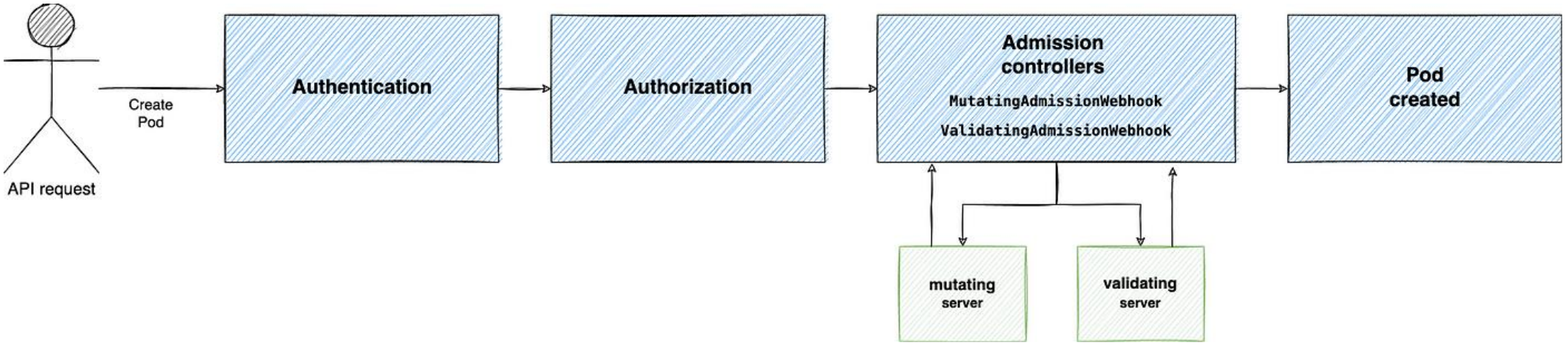
```
# demo-namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: demo-namespace
```

Admission Controllers

An admission controller is a piece of code that intercepts requests to the Kubernetes API server prior to persistence of the object, but after the request is authenticated and authorized.

Can be (one or both)

- Validating
- Mutating



Admission controllers

Kubernetes can be configured to use so admission plugins:
`kube-apiserver --enable-admission-plugins=…`

Runs after authorization but before storage of the object

Intercepts calls and fills default values, checks settings and possibly authorization

Default a set is enabled:
`kube-apiserver -h | grep enable-admission-plugins`

https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

Admission Controllers

K8S 1.28 HAS THESE BY DEFAULT :

CertificateApproval,
CertificateSigning,
CertificateSubjectRestriction,
DefaultIngressClass,
DefaultStorageClass,
DefaultTolerationSeconds,
LimitRanger,
MutatingAdmissionWebhook,
NamespaceLifecycle,
PersistentVolumeClaimResize,
PodSecurity,
Priority,
ResourceQuota,
RuntimeClass,
ServiceAccount,
StorageObjectInUseProtection,
TaintNodesByCondition,
ValidatingAdmissionPolicy,
ValidatingAdmissionWebhook

Dynamic Admission Control

Prerequisites:

- Corresponding controllers need to be enabled
- API admissionregistration.k8s.io/v1 should be enabled

Webhook service needs to run over valid https

Example

```yaml
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  rules:
  - apiGroups:   [""]
    apiVersions: ["v1"]
    operations:  ["CREATE"]
    resources:   ["pods"]
    scope:       "Namespaced"
  clientConfig:
    service:
      namespace: "example-namespace"
      name: "example-service"
    caBundle: <CA_BUNDLE>
  admissionReviewVersions: ["v1"]
  sideEffects: None
  timeoutSeconds: 5
```

Limit resources in a namespace

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
```

Set the security context for a Pod

You can change the userid & groupid of the user inside the containers.

Set the context to ‘privileged’ to allow the process almost the same rights as it would have running outside Kubernetes
(access to the host)

Break down unrestricted root access by adding or revoking linux capabilities

- Like binding to a port < 1024 as non root

More info:

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

Set the security context for a Container

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

Capabilities

Linux capabilities allow you to can grant certain privileges to a process without granting all the privileges of the root user (or take away some).

Change Linux capabilities by setting the capabilities field in securityContext

Capabilities

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

Kubernetes Privileged Containers

A **privileged container** in Kubernetes is a container that has elevated permissions

granting it **full access** to the host's kernel and resources.

Privileged containers are useful for:

- Running system-level tools like iptables or mount.
- Managing storage devices or network configurations.
- Debugging host-level issues.

Warning! Privileged containers impose serious security threats!

Alternative:
set capabilities

Privileged pod example

```
apiVersion: v1
kind: Pod
metadata:
 name: privileged-pod
spec:
 containers:
 - name: privileged-container
  image: centos
  command: ["sh", "-c", "sleep 999"]
  securityContext:
   privileged: true
```

Allow privileged containers

Add --allow-privileged=true to:

kubelet config
https://kubernetes.io/docs/tasks/administer-cluster/kubelet-config-file/

sudo vim /var/snap/microk8s/current/args/kubelet

#kube-apiserver config
https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/

sudo vim /var/snap/microk8s/current/args/kube-apiserver
Restart services:
sudo systemctl restart snap.microk8s.daemon-kubelet.service
sudo systemctl restart snap.microk8s.daemon-apiserver.service

Kubernetes POD Security Policy (PSP)

Implemented as Admission Controller

Restrict the containers allowed to run on the cluster

- Running of privileged containers
- Allowed hostpaths
- RunAs user
- Allowed (and default) capabilities
- Whitelist of Volume Types

https://kubernetes.io/docs/concepts/security/pod-security-policy/

Note: deprecated after v1.25!
Use https://kubernetes.io/docs/concepts/security/pod-security-admission/

Pod Security Admission Controller

You can set the Pod Security Level per namespace to one of these settings:

3 modes

- Audit
  Record in the audit log, continue
- Warn
  Display a warning when violation, but continue
- Enforce
  Causes the pod to be rejected

Protecting a namespace

```
apiVersion: v1
kind: Namespace
metadata:
  name: protected
  labels:
    pod-security.kubernetes.io/enforce: restricted
                                        (or baseline, privileged)
```

```yaml
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    # Defaults applied when a mode label is not set.
    #
    # Level label values must be one of:
    # - "privileged" (default)
    # - "baseline"
    # - "restricted"
    #
    # Version label values must be one of:
    # - "latest" (default)
    # - specific version like "v1.25"
```

```
18      defaults:
19        enforce: "privileged"
20        enforce-version: "latest"
21        audit: "privileged"
22        audit-version: "latest"
23        warn: "privileged"
24        warn-version: "latest"
25      exemptions:
26        # Array of authenticated usernames to exempt.
27        usernames: []
28        # Array of runtime class names to exempt.
29        runtimeClasses: []
30        # Array of namespaces to exempt.
31        namespaces: []
```

https://kubernetes.io/docs/concepts/security/pod-security-standards/

Kubernetes Networking Policies

Control networking at **POD level**

Like firewalling / iptables

Using Kubernetes context like pod labels, namespaces, etc.

Possible 3rd party plugins for pod overlay networks
https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/

K8s Networking Policies

Scope: They apply at **Layer 3/4** (IP + port level).

Targets: You select Pods using labels.

Rules: You define allowed **Ingress** (incoming) and/or **Egress** (outgoing) traffic.

Isolation: Once a Pod is selected by a NetworkPolicy, it becomes “isolated” — only traffic explicitly allowed is permitted.

Requirement: Your cluster must use a CNI plugin that supports NetworkPolicy (e.g., Calico, Cilium, Weave Net). Without that, the YAML has no effect

Network policy example

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-same-namespace
  namespace: my-app
spec:
  podSelector: {}   # applies to all Pods in this namespace
  policyTypes:
  - Ingress
  ingress:
  - from:
    - podSelector: {}   # allow from any Pod in the same namespace
```

Selectors

```
...
ingress:
- from:
  - namespaceSelector:
      matchLabels:
        user: alice
    podSelector:
      matchLabels:
        role: client
...
```

Ingress options

```yaml
ingress:
- from:
  - ipBlock:
      cidr: 172.17.0.0/16
      except:
      - 172.17.1.0/24
  - namespaceSelector:
      matchLabels:
        project: myproject
  - podSelector:
      matchLabels:
        role: frontend
  ports:
  - protocol: TCP
    port: 6379
```

Allows traffic in from:

IP addresses in the ranges 172.17.0.0–172.17.0.255 and 172.17.2.0–172.17.255.255

any pod in a namespace with the label project=myproject

any pod in the default namespace with the label role=frontend

Only traffic from sources above AND only on port 6379

Blocks all other incoming traffic

Egress options

```
egress:
- to:
  - ipBlock:
      cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978
```

Allows traffic out to:

IP addresses in the ranges 172.17.0.0–172.17.0.255 and 172.17.2.0–172.17.255.255

any pod in a namespace with the label project=myproject

any pod in the default namespace with the label role=frontend

Only traffic from sources above AND only on port 6379

Blocks all other incoming traffic

Default examples

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-ingress
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-ingress
spec:
  podSelector: {}
  ingress:
  - {}
  policyTypes:
  - Ingress
```

Port ranges

```yaml
ports:
  - protocol: TCP
    port: 32000
    endPort: 32768
```

Ports from 32000 - 32768

PID 1

PID 1 processes in Linux do not have any default signal handlers and as a result will not receive and propogate signals. They are also expected to take on certain responsibilities, such as adopting orphaned processes, and reaping zombie processes.

```
$ docker exec pedantic_matsumoto ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 {main} /bin/sh ./main &
    6 root      0:00 python ./main.py
   12 root      0:00 ps aux

$ docker exec pedantic_matsumoto pstree -p
main(1)---python(6)
```

PID 1 - workarounds

DIY Signal Handling and exec
The first way to get around this issue is to install custom signal handlers for SIGTERM and other signals you need directly in your application code, and then run exec in your wrapper shell script. This replaces the running process with your application.

dumb-init
is a simple init process which does everything an init process is supposed to do. If you install it in your Docker container and use it as your entrypoint, you'll be able to handle signals just fine.

Tini
If you run your Docker container with --init, Docker will automatically start its own init process as PID 1. The problem with using tini is that container orchestrators, such as Kubernetes, can't start your Docker container with the --init flag.If you want to use tini, you'll have to download and install it in your Dockerfile and pass along the -g option for signal forwarding.

Liveness probe

Simple process check is not enough

Kubernetes can check if your application inside the container is alive by doing a liveness check.

These need to be added manually into the manifest

```
livenessProbe:
  httpGet:
    path: /
    port: "80"
  initialDelaySeconds: 5
  timeoutSeconds: 1
  periodSeconds: 10
  failureTreshold: 3
  successThreshold: 1
```

Liveness probe

HTTP type:

Checks periodically by polling at the given path and portnumber

Result should be statuscode between 200 and 400

If it fails <failureTreshold> times the container will be restarted

Readiness probe

Describes when a container is ready to handle user requests

Configured just like liveness probes

Combination of readiness and liveness probes makes sure only healthy containers are running

Types

- HTTP check
- tcpSocket (tests if port can be opened)
- Exec (runs a command, 0 resultcode is success)

Setting Limits in Kubernetes

Resource limits

Default limits

Eviction

Resource management

In addition to simplified distributed application deployment we want to increase overall utilization of our compute nodes

Basic cost of operating a machine is constant, regardless of being idle or fully loaded

With Kubernetes managing resource packing we can go up well over 50%

To do this Kubernetes needs to know what can be expected from your application in terms of resource requirements, so packing can be optimal across the cluster

Resource management

Based on 2 different resource metrics

- Requested resources
  - Minimum required:
  - CPU (i.e. 500m is half a cpu)
  - Memory (i.e. 128Mi is 128MB of memory)
  - Expressed in container section(s) of manifest file
- Resource limits
  - Same settings, but maximizing/capping to the configured level

Resource management

```
containers:
 - name: my-pod
   image: nginx
   resources:
       requests:
         cpu: 100m
         memory: 128Mi
       limits:
         cpu: 500m
         memory: 256Mi
```

Setting quota on namespace

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-quota
  namespace: my-team
spec:
  hard:
    requests.cpu: "2"         # total CPU requests across namespace
    requests.memory: "4Gi"    # total memory requests
    limits.cpu: "4"           # total CPU limits
    limits.memory: "8Gi"      # total memory limits
```

Setting quota to object counts

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts
  namespace: my-team
spec:
  hard:
    pods: "20"           # max 20 Pods
    configmaps: "10"     # max 10 ConfigMaps
    persistentvolumeclaims: "5"
```

Set defaults with LimitRange

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
  - default: # this section defines default limits
      cpu: 500m
    defaultRequest: # this section defines default requests
      cpu: 500m
    max: # max and min define the limit range
      cpu: "1"
    min:
      cpu: 100m
    type: Container
```

Additional policy systems

- **Kyverno** = Kubernetes-native, easy, YAML-driven.
- **Gatekeeper** = Rego-powered, mature, governance-heavy.
- **Kubewarden** = Wasm-based, flexible, future-oriented.

Kyverno

- Cloud Native policy enging
- Kubernetes native syntax in YAML
- CNCF Incubating project
- Validate, Mutate, Generate or cleanup any Kubernetes resource

https://kyverno.io/#about-kyverno

OPA Gatekeeper

Policy Controller

Extensions for public cloud

YAML based ConstraintTemplate

Library with policies

Mature CNCF project, can be heavy on resources in complex environments

https://open-policy-agent.github.io/gatekeeper/website/docs/

Kubewarden

- Kubewarden is a Kubernetes Policy Engine
- CNCF Sandbox project
- Originally created by Rancher
- Security hardening, compliancy audits
- Use any programming language that generates WebAssembly binaries to write your policies

https://docs.kubewarden.io/

Feature / Aspect	**Kyverno**	**OPA Gatekeeper**3	**Kubewarden**
CNCF Status	Incubating project	OPA is CNCF Graduated; Gatekeeper is widely adopted	CNCF Sandbox project
Policy Language	YAML/JSON (Kubernetes-native syntax)	Rego (OPA's DSL)	Any language that compiles to WebAssembly (Rust, Go, C#, TS, etc.)
Learning Curve	Easy for Kubernetes users (no new language)	Steeper (Rego is powerful but complex)	Flexible, but requires dev skills in chosen language
Capabilities	Validate, Mutate, Generate, Cleanup, Verify images	Validate, Mutate, Audit; strong constraint framework	Admission control, continuous audit, flexible enforcement
Extensibility	Kubernetes-native CRDs, integrates with CI/CD	Highly extensible via Rego templates	Extremely extensible via Wasm modules
Use Cases	- Enforce labels/annotations - Auto-generate resources (e.g., NetworkPolicy) - Image signature verification - Pod Security replacement	- Governance & compliance - Enforce org-wide rules (naming, labels, security contexts) - Strong auditing	- Security hardening (privileges, registries) - Compliance auditing - Resource optimization
Community & Ecosystem	Growing fast, strong adoption in platform engineering	Large, mature community (OPA + Gatekeeper widely used)	Smaller but innovative, backed by SUSE Rancher
Performance	Lightweight, Kubernetes-native	Proven but Rego evaluation can be heavy in complex rules	Efficient (Wasm is portable and fast)
Distribution	Policies as CRDs, easy to apply with `kubectl`	ConstraintTemplates + Constraints CRDs	Policies packaged as OCI artifacts, distributed via registries

The downward API

DownwardAPI is a way to send Kubernetes information about a pod into a container.

Think of information as :

- the pod's name
- Namespace
- unique-id, labels
- Annotations
- host & pod ip
- Resource limit & request settings

The downward API

There are two ways to expose Pod and Container fields to a running Container:

Environment variables

DownwardAPIVolumeFiles

Through environment settings

```
env:
    - name: MY_NODE_NAME
      valueFrom:
        fieldRef:
          fieldPath: spec.nodeName
    - name: MY_POD_NAME
      valueFrom:
        fieldRef:
          fieldPath: metadata.name
    - name: MY_POD_NAMESPACE
      valueFrom:
        fieldRef:
          fieldPath: metadata.namespace
    - name: MY_POD_IP
      valueFrom:
        fieldRef:
          fieldPath: status.podIP
```

Via DownwardAPI volume

```
volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

Using hooks and initContainers

Specialized containers that run before app containers in a Pod.

Init containers can contain utilities or setup scripts not present in an app image.

Init containers are exactly like regular containers, except:

- Init containers always run to completion.
- Each init container must complete successfully before the next one starts.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for mysvc; sleep 2;
done;']
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

Inspecting the init containers

```
# Inspect the first init container
kubectl logs myapp-pod -c init-myservice

 # Inspect the second init container
kubectl logs myapp-pod -c init-mydb
```

Lifecycle Hooks

Kubernetes supports container lifecycle hooks which allows a handler code/script to be executed when Container lifecycle event occurs.

There are two lifecycle hooks. PostStart hook executes immediately after container is created. PreStop hook is called just before the container is terminated. Both the hooks take no parameters.

K8s support two types of handlers for the lifecycle hooks. Exec handler is used for calling shell command/script while HTTP handler is used for calling HTTP endpoint

Lifecycle hook example

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

Init Containers	Lifecycle Hooks
Configured at Pod Level. All init Containers execute before app Container	Separate hooks for each container in a Pod.
Don’t support post Stop lifecycle event for Pod	Supports both preStart and postStop hook
They are just like containers with own image, volumes and resources	They are executed inside parent container as script or HTTP command
Init Containers are guaranteed to execute before Pod is initialized.	There is no guarantee that the preStart hook will execute before the container ENTRYPOINT

Usage scenarios- Init Containers

Delay or Block the startup of app Containers until some set of preconditions are met. E.g . sleep 60

In some cases if one Pod has startup dependency on another (Avoid as much as possible)

For multi-container Pod, perform common task required at startup

Usage scenarios – lifecycle hooks

Perform cleanup before container is terminated with preStop hook

Save state of container before termination with preStop hook

Perform post configuration tasks on application startup inside container.

Stateful sets

Manages the deployment and scaling of a set of Pods

Provides guarantee about the ordering and uniqueness of these Pods.

Persistent Volume per pod (<pod>-01, <pod>-02 etc)

Pod priority and preemption

Pods can have priority

Relative to other pods

If a pod cannot be scheduled it preempts lower priority pods

Add one or more PriorityClasses

Use `priorityClassName` in Pod definition

PriorityClass example

```
apiVersion: scheduling.k8s.io/v1

kind: PriorityClass

metadata:

  name: high-priority

value: 1000000

globalDefault: false

description: "This priority class should be used for XYZ service pods only."
```

PriorityClass with disabled preemption

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority-nonpreempting
value: 1000000
preemptionPolicy: Never
globalDefault: false
description: "This priority class will not cause other pods to be preempted."
```

Using PriorityClass

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

Disable preemption

```
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider

...

disablePreemption: true
```

Eviction

When resources become low can stop Pods to free up memory or storage

Configure by Eviction Policy

- Eviction trigger signal
- Treshold value

Eviction setting

Either hard or softeviction

Set this in the kubelet configuration

```
--eviction-
hard="memory.available<100Mi,nodefs.available<1Gi,imagefs.available<1Gi"


--eviction-soft="memory.available<1.5Gi,…"

--eviction-soft-grace-period="memory.available=1m30s,…"
```

Affinity

```
#podspec
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - node1
```

Taints

Affinity rules set pods to requested nodes

Taints are the opposite: do not allow (certain) pods on a node

Example:
`kubectl taint nodes node1 key=value:NoSchedule`

Node 1 will not allow pods unless it has a matching toleration

`kubectl taint nodes node1 key=value:NoSchedule-` to remove taint

Toleration

```
# podspec
tolerations:
- key: key
  operator: Equal
  value: “value”
  effect: NoSchedule
```