Kubernetes Patterns : Capacity Planning

An excellent cloud-native application design should declare any specific resources that it needs to operate correctly. Kubernetes uses those requirements to make the most efficient decisions to ensure maximum performance and availability of the application.

Additionally, knowing the application requirements firsthand allows you to make cost-effective decisions regarding the hardware specifications of the cluster nodes. We will explore in this article best practices to declare storage, CPU, and memory resources needs. We will also discuss how Kubernetes behaves if you don't specify some of these dependencies.

Storage Dependency

Let’s explore the most common runtime requirement of an application: persistent storage. By default, any modifications made to the filesystem of a running container are lost when the container is restarted. Kubernetes provides two solutions to ensure that changes persist: emptyDir and Persistent Volumes.

Using Persistent Volumes, you can store data that does not get deleted even if the whole Pod was terminated or restarted. There are several methods by which you can provision a backend storage to the cluster. It depends on the environment where the cluster is hosted (on-prem or in the cloud and the cloud provider). In the following lab, we use the host’s disk as the persistent volume backend storage. Provisioning storage using Persistent Volumes involves two steps:

1. Creating the Persistent Volume: this is the disk on which Pods *claim* space. This step differs depending on the hosting environment.
2. Creating a Persistent Volume Claim: this is where you actually provision the storage for the Pod by claiming space on the Persistent Volume.

In the following lab, we create a Persistent Volume using the host’s local disk. Create a new YAML definition file, PV.yaml, and add the following lines:

apiVersion: v1

kind: PersistentVolume

metadata:

name: hostpath-vol

spec:

storageClassName: local

capacity:

storage: 1Gi

accessModes:

- ReadWriteOnce

hostPath:

path: "/tmp/data"

Copy

This definition creates a Persistent Volume (PV for short) that uses the host disk as the backend storage. The volume is mounted on /tmp/data directory on the host. We need to create this directory before applying the configuration:

mkdir /tmp/data

Copy

$ kubectl apply -f PV.yaml

persistentvolume/hostpath-vol created

Copy

Now, can create a Persistent Volume Claim (PVC for short) and avail it to our Pod to store data through a mount point. The following definition file creates both a PVC and a Pod that uses it:

apiVersion: v1

kind: PersistentVolumeClaim

metadata:

name: my-pvc

spec:

storageClassName: local

accessModes:

- ReadWriteOnce

resources:

requests:

storage: 100Mi

---

apiVersion: v1

kind: Pod

metadata:

name: pvc-example

spec:

containers:

- image: alpine

name: pvc-example

command: ['sh', '-c', 'sleep 10000']

volumeMounts:

- mountPath: "/data"

name: my-vol

volumes:

- name: my-vol

persistentVolumeClaim:

claimName: my-pvc

Applying this definition file creates the PVC followed by the Pod:

$ kubectl apply -f pvc\_pod.yaml

persistentvolumeclaim/my-pvc created

pod/pvc-example created

Any data that gets created or modified on /data inside the container will be persisted to the host’s disk. You can check that by logging into the container, creating a file under /data, restarting the Pod and then ensuring that the file still exists on the Pod. You can also notice that files created in /tmp/data are immediately available to the Pod and its containers.

Notice that in the preceding lab, we used one node only so there shouldn’t be any problems when we need to schedule a Pod that requires a PVC to function correctly. However, if we were in a multi-node environment (which is often the case when using Kubernetes), and if a given node is incapable of provisioning the persistent volume, the Pod will never get scheduled to this node. A worse scenario may occur if all the nodes in the cluster cannot provide the requested volume. In such a case, the Pod will not get scheduled at all.

[Learn how to continuously optimize your k8s cluster](https://www.magalix.com/cs/c/?cta_guid=18dc390a-3729-410b-a0ec-41819020a462&placement_guid=963b2ba4-df32-4a3b-8781-f8fa4ff38750&portal_id=3487587&canon=https%3A%2F%2Fwww.magalix.com%2Fblog%2Fkubernetes-patterns-capacity-planning&redirect_url=APefjpGio0xZb_WkBWw6JFG8u8xh3BXtpMvLQeuEYq2cCdzD6HxLMiUg5Qet4mgcOYYNP5V-mIItQ-bNMoLKnmMyKPnMvfSpwHS6UsiJu6GGlytnpr_Po6aCJMVAJTgG83WhKpJqaBeizrHomB_SVgPALf7rmYHtfzuRM18OACm72c1oa3pfpfyBMBH4L0hpsySJblU-MD-8nHWdww3bUb7fwpaIeTtv-H5f507Ymo6i6SagL4NBqm8&click=4f70ad8c-1bb8-4277-98ce-6714af11a305&hsutk=b55ef47ea0732f7b6cf56a7be715fe66&signature=AAH58kFievvf7XfZwiRAQMc28A8Dyt1nrg&utm_referrer=https%3A%2F%2Fwww.google.com%2F&pageId=13387084209&__hstc=197029363.b55ef47ea0732f7b6cf56a7be715fe66.1568178953911.1570301143166.1570405422564.8&__hssc=197029363.1.1570405422564&__hsfp=3494084449&contentType=blog-post)

The hostPort Dependency

If you are using the [hostPort](http://alesnosek.com/blog/2017/02/14/accessing-kubernetes-pods-from-outside-of-the-cluster/) option, you are explicitly allowing the internal container port to be accessible from outside the host. A Pod that uses hostPort cannot have more than one replica on the same host because of port conflicts. If no node can provide the required the port (let’s say it is a standard port number like port 80 or 443), then the Pod using in the hostPort option will never get scheduled. Additionally, this creates a one-to-one relationship between the Pod and its hosting node. So, in a cluster with four nodes, you can only have a maximum of four Pods that use the hostPort option (provided that the port is available on each node).

Configuration Dependency

Almost all applications are designed so that they can be customized through variables. For example, MySQL needs at least the initial root credentials; WordPress needs the database host and name, and so on. Kubernetes provides configMaps for injecting variables to containers inside Pods and Secrets for supplying confidential variables like account credentials. Let’s have a quick example on how to use configMaps to provision variables to a Pod:

The following definition file creates both a configMap and Pod that uses the variables defined in this configMap:

kind: ConfigMap

apiVersion: v1

metadata:

name: myconfigmap

data:

# Configuration values can be set as key-value properties

dbhost: db.example.com

dbname: mydb

---

kind: Pod

apiVersion: v1

metadata:

name: mypod

spec:

containers:

- name: mycontainer

image: nginx

envFrom:

- configMapRef:

name: myconfigmap

Copy

Now, let’s apply this configuration and ensure that we can use the environment variables inside our container:

$ kubectl apply -f pod.yml

configmap/myconfigmap created

pod/mypod created

$ kubectl exec -it mypod -- bash

root@mypod:/# echo $dbhost

db.example.com

root@mypod:/# echo $dbname

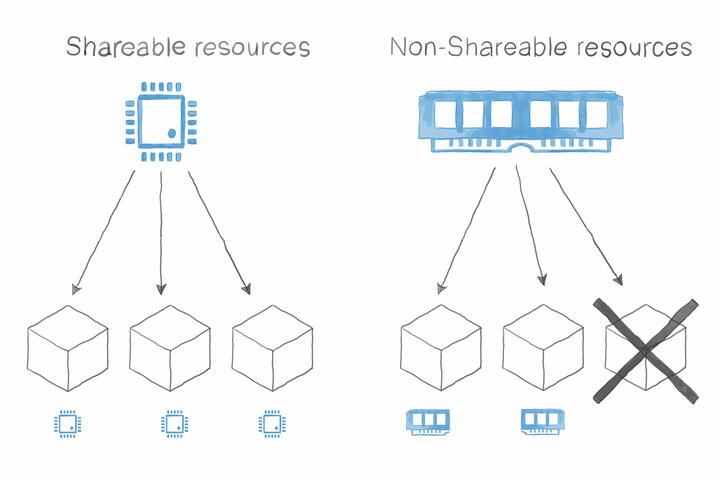
mydb

Copy

However, this creates a dependency of its own: if the configMap was not available, the container might not work as expected. In our example, if this container an application that needs a constant database connection to work, then if it failed to obtain the database name and host, it may not work at all.

The same thing holds for Secrets, which must be available firsthand before any client containers can get spawned.

Resource Dependencies



So far we discussed the different runtime dependencies that affect which node will the Pod get scheduled (if at all) and the various prerequisites that must be availed for the Pod to function correctly. However, you must also take into consideration that capacity requirement of the containers.

**Controllable and Uncontrollable Resources**

When designing an application, we need to be aware of the type of resources that this application may consume. Generally, resources can be classified into two main categories:

* **Shareable:** those are the resources that can be shared among different consumers and, thus, limited when required. Examples of this are CPU and network bandwidth.
* **Non-shareable:** resources that cannot be shared by nature. For example, memory. If a container tries to use more memory than its allocation, it will get killed.

**Declaring Pods Resource Requirements**

The distinction between both resource types is crucial for a good design. Kubernetes allows you to declare the amount of CPU and memory the Pod requires to function. There are two parameters that you can use for this declaration:

* **requests:** this is the minimum amount of resources that the Pod needs. For example, you may already have the knowledge that the hosted application will fail to start if it does not have access to at least 512 MB of memory.
* **limits:** the limits define the maximum amount of resources that you need to supply for a given Pod.

Let’s have a quick example for a hypothetical application that needs at least 512 MB and 0.25% of a CPU core to run. The definition file for such a Pod may look like this:

kind: Pod

apiVersion: v1

metadata:

name: mypod

spec:

containers:

- name: mycontainer

image: myapp

resources:

requests:

cpu: 250m

memory: 512Mi

limits:

cpu: 500m

memory: 750Mi

Copy

When the scheduler manages to deploy this Pod, it will search for a node that has at least 512 MB of memory free. If a suitable node was found, the Pod gets scheduled on it. Otherwise, the Pod will never get deployed. Notice that *only* the requests field is considered by the scheduler when determining where to deploy the Pod.

**How Are the Resource Requests and Limits Calculated?**

Memory is calculated in bytes, but you are allowed to use units like Mi and Gi to specify the requested amount. Notice that you should not specify a memory limit that is higher than the amount of memory on your nodes. If you did, the Pod would never get scheduled. Additionally, since memory is a non-shareable resource as we discussed, if a container tried to request more memory than the limit, it will get killed. Pods that are created through a higher controller like a ReplicaSet or a Deployment have their containers restarted automatically when they crash or get terminated. Hence, it is always recommended that you create Pods through a controller.

CPU is calculated through *millicores*. 1 core = 1000 millicores. So, if you expect your container needs at least half a core to operate, you set the request to 500m. However, since CPU belongs to shareable resources when the container requests more CPU than the limit, it will not get terminated. Rather, the Kubelet *throttles* the container, which may negatively affect its performance. It is advised here that you use liveness and readiness probes to ensure that your application’s latency does not affect your business requirements.

**What Happens When You (not) Specify Requests and Limits?**

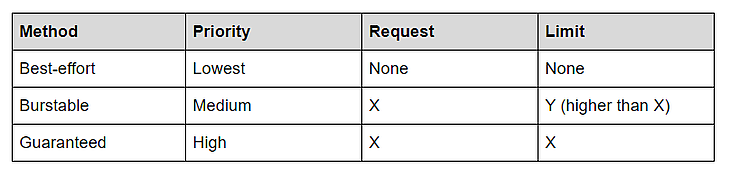
Most of the Pod definitions examples ignore the requests and limits parameters. You are not strictly required to include them when designing your cluster. Adding or ignoring requests and limits affects the Quality of Service (QoS) that the Pod receives as follows:

**Lowest priority pods:** when you do not specify requests and limits, the Kubelet will deal with your Pod in a *best-effort* manner. The Pod, in this case, has the lowest priority. If the node runs out of non-shareable resources, best-effort Pods are the first to get killed.

**Medium priority pods:** if you define both parameters and set the requests to be less than the limit, then Kubernetes manages your Pod in the *Burstable* manner. When the node runs out of non-shareable resources, the Burstable Pods will get killed only when there are not more best-effort Pods running.

**Highest priority pods:** your Pod will be deemed as of the most top priority when you set the requests and the limits to equal values. It’s as if you’re saying, “I need this Pod to consume no less and no more than x memory and y CPU.” In this case, and in the event of the node running out of shareable resources, Kubernetes does not terminate those Pods until the best-effort, and the burstable Pods are terminated. Those are the highest priority Pods.

We can summarize how the Kubelet deals with Pod priority as follows:



Pod Priority and Preemption

Sometimes you may need to have more fine-grained control over which of your Pods get evicted first in the event of resource starvation. You can guarantee that a given Pod get evicted last if you set the request and limit to equal values. However, consider a scenario when you have two Pods, one hosting your core application and another hosting its database. You need those Pods to have the highest priority among other Pods that coexist with them. But you have an additional requirement: you want the application Pods to get evicted *before* the database ones do. Fortunately, Kubernetes has a feature that addresses this need: Pod Priority and preemption. Pod Priority and preemption is stable as of Kubernetes 1.14 or higher. The feature is enabled by default since Kubernetes version 1.11 (beta release). If your cluster version is less than 1.11, you will need to enable this feature explicitly.

So, back to our example scenario, we need two high priority Pods, yet one of them is more important than the other. We start by creating a PriorityClass then a Pod that uses this PriorityClass:

apiVersion: scheduling.k8s.io/v1

kind: PriorityClass

metadata:

name: high-priority

value: 1000

---

apiVersion: v1

kind: Pod

metadata:

name: mypod

spec:

containers:

- image: redis

name: mycontainer

priorityClassName: high-priority

Copy

The definition file creates two objects: the PriorityClass and a Pod. Let’s have a closer look at the PriorityClass object:

Line 1: the API version. As mentioned, PriorityClass is stable as of Kubernetes 1.14.

Line 2: the object type

Line 3 and 4: the metadata where we define the object name.

Line 5: we specify the value on which the priority is calculated relative to other Pods in the cluster. A higher value indicates a higher priority.

Next, we define the Pod that uses this PriorityClass by referring its name.

**How Pods Get scheduled Given their PriorityClass Value?**

When we have multiple Pods with different PriorityClass values, the admission controller starts by sorting Pods according to their priority. Highest priority Pods (those having the highest PriorityClass numbers) get scheduled first as long as no other constraints are preventing their scheduling.

Now, what happens if there are no nodes with available resources to schedule a high-priority pod? The scheduler will evict (preempt) lower priority Pods from the node to give enough room for the higher priority ones. The scheduler will continue lower-priority Pods until there is enough room to accommodate the more upper Pods.

This feature helps you when you design the cluster so that you ensure that the highest priority Pods (for example, the core application and database) are never evicted unless no other option is possible. At the same time, they also get scheduled first.

**Things to Consider in Your Design When Using QoS and Pod Priority**

You may be asking what happens when you use resources and limits (QoS) combined with the PriorityClass parameter. Do they overlap or override each other? In the following lines, we show you some of the essential things to note when influencing the schedule decisions:

* The Kubelet uses QoS to control and manage the node’s limited resources among the Pods. QoS eviction happens only when the node starts to run out of shareable resources (see the earlier discussion in this article). The Kubelet considers QoS before considering Preemption priorities.
* The scheduler considers the PriorityClass of the Pod before the QoS. It does not attempt to evict Pods unless higher-priority Pods need to be scheduled and the node does not have enough room for them.
* When the scheduler decides to preempt lower-priority pods, it attempts a clean shutdown and respects the grace period (thirty seconds by default). However, it does not honor[PodDisruptionBudget](https://kubernetes.io/docs/concepts/workloads/pods/disruptions/), which may lead to disrupting the cluster quorum of several low priority Pods.
* On a single node environment, you can think of a container as a way of packaging and isolating applications. But when your environment spans across multiple nodes, you can also use containers as a means of efficient capacity planning.
* Determining the resources that your applications require at design time will save you a lot of time and effort afterwards. An application may have the following dependencies:
  + Volumes
  + Ports
  + Node resources like CPU and memory
* Part of the dependency fulfillment is handled by the cluster administrator (volumes, configMaps, Secrets,etc.) while the scheduler and the kubelet control the other part. You can still influence the decisions of both.
* You can design your Pods without defining the minimum and maximum resources they may need. However, doing so forces Kubernetes to use “best-effort” when scheduling your Pod, which may result in evicting more important Pods than others. You should be careful when setting the resource limits (requests and limits) because setting an incorrect value may lead to unnecessary eviction (when using non-shareable resources) or degraded performance (when using shareable resources).
* Setting the requests and limits parameters influences the kubelet decision as to which Pod to evict first in the event of resource starvation.
* Defining and setting the PriorityClass for a Pod influences the scheduler decision as to which Pods get scheduled first and which Pods get evicted if no room is available on the nodes.