Kubernetes Rightsizing Guide

sysdig

Table of Contents

01

About This Guide

Kubernetes is the most used container orchestration system. Since its release in 2014, it has driven a complete revolution in the cloud applications environment.

It's done this by using powerful abstractions — like Pods — and a high-availability design to incentivize the creation of cloud-native applications and, more importantly, their deployment into production environments.

Additionally, thanks to the collaborations from its thriving community and major adoption from cloud vendors, Kubernetes now represents a key element in the DevOps methodology.

In order to accomplish the best possible performance, companies try to find the best capacity and resource allocation for their cloud solutions. But this is a complex process and sometimes it's neglected.

Think of the following scenarios:

- A company has started a successful business, but now needs to extend operations to the whole country.
- Specific product releases make traffic multiply by 10x in sudden bursts.
- A cloud application is mostly idle in certain periods of time or certain availability zones, but the cloud bill skyrockets because the company doesn't want to take measures.

As you can see, choosing the right resource allocation is a delicate matter. This is where rightsizing and capacity planning comes in.

In this guide, you will learn:

- Why scaling is important
- What the most important resources in Kubernetes are
- How CPU and memory work internally
- How to take into account scaling for control plane elements, like etcd or CoreDNS
- Kubernetes Limits and Requests
- How to measure Kubernetes metrics for sizing with Prometheus
- How to build autoscaling with custom metrics
- How to rightsize Kubernetes requests
- How to rightsize Kubernetes limits
- Best practices for Capacity Planning

02

The Importance of Kubernetes Rightsizing

You are either starting your Kubernetes journey or have many flight hours. At some point, you may want to review your infrastructure and analyze whether a rightsizing exercise would be beneficial for you. Sometimes, companies fail to design and properly size their applications, leading to availability issues if they are short when assigning resources, or wasting tons of CPU, memory, disk, etc. if they go too far with requests and limits. Kubernetes rightsizing helps practitioners accommodate their applications by adjusting them to a fair number of resources. This approach lets companies ensure desired quality of service and avoid wasted resources.

Elastic scale

Scalability is the process where resources are increased or decreased to meet the demand of the system.

Elastic scale is a pattern where that scalability is done to meet the current demand of the system. That scaling can be done in several ways:

- **Horizontally scaling:** Modify the number of replicas
- **Vertically scaling:** Change the resources of the Pods
- **Cluster scaling:** Change the number of Nodes in the cluster

Overprovisioning leads to waste

Companies usually want to be very conservative about resources in their cloud applications, and reserve a cluster big enough so there's a minimum downtime and they don't have to worry about it in the future.

The Sysdig 2023 Cloud-native Security and Usage Report found that, on average, Kubernetes clusters have 69% unused reserved CPU, which is unnecessarily costing big companies running applications in the cloud millions of dollars.

High availability

Over the years, more and more processing in applications is done in the cloud. Therefore, cloud applications need to stay up all the time and minimize the downtime in case of problems because of their business importance or criticality. This is the reason why cloud vendors provide high-availability (HA) solutions that are built specifically to cover these cases.

These high-availability solutions are based on four pillars:

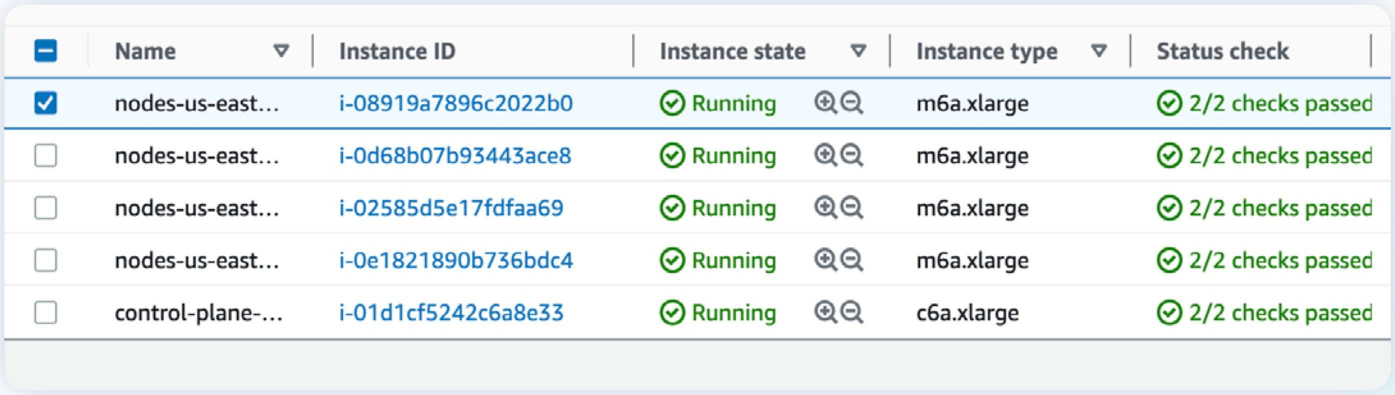
- **Redundancy:** Additional replicas for the application or storage to guarantee that if one has a disruption, the rest can make the system function as normal.
- **Monitoring:** Get full observability on the metrics that define your system.
- **Recovery:** Techniques to guarantee that once there is a disruption, the system can go back to normal in a seamless and quick manner.
- **Checkpointing:** The ability to persist application state and data for a long period of time, regardless of any disruption.

Initial sizing and deployment of HA applications and services, like Kubernetes deployments, needs to be very well planned. Multiple instances guaranteeing redundancy means more resources, which may lead companies to increase their costs. Rightsizing HA deployments may help to significantly reduce companies' wasted resources and spending.

Key Resources to Monitor and Size Properly

When it comes to Cloud instances, some of the most important resources are:

- CPU
- Memory
- Storage size
- Network bandwidth



	Name	Instance ID	Instance state	Instance type	Status check
☑	nodes-us-east...	i-08919a7896c2022b0	✓ Running	m6a.xlarge	✓ 2/2 checks passed
☐	nodes-us-east...	i-0d68b07b93443ace8	✓ Running	m6a.xlarge	✓ 2/2 checks passed
☐	nodes-us-east...	i-02585d5e17fdfaa69	✓ Running	m6a.xlarge	✓ 2/2 checks passed
☐	nodes-us-east...	i-0e1821890b736bdc4	✓ Running	m6a.xlarge	✓ 2/2 checks passed
☐	control-plane-...	i-01d1cf5242c6a8e33	✓ Running	c6a.xlarge	✓ 2/2 checks passed

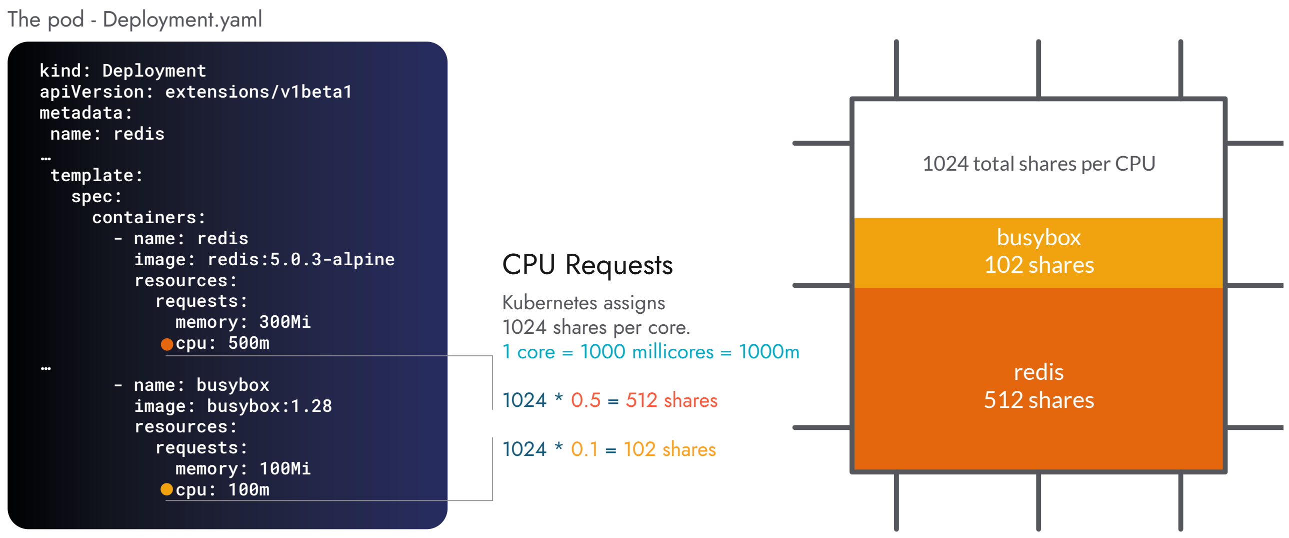
All these resources are not only mandatory to have Cloud instances up and running, but crucial when designing architectures and during the whole application lifecycle, especially when a rightsizing might be required. Let's see each of these in detail in the following section.

CPU

CPU represents **computing processing time**, measured in cores.

CPU is a shared and finite resource that tries to satisfy all the demands that are requested. In the event that the processes request too much CPU, more than its actual limit, which can be either a physical CPU capacity limit or a software limit, some of these processes will be throttled.

- You can use millicores (m) to represent smaller amounts than a core (e.g., 500m would be half a core).
- The minimum amount is 1m.
- A Node might have more than one core available, so requesting CPU > 1 is possible.



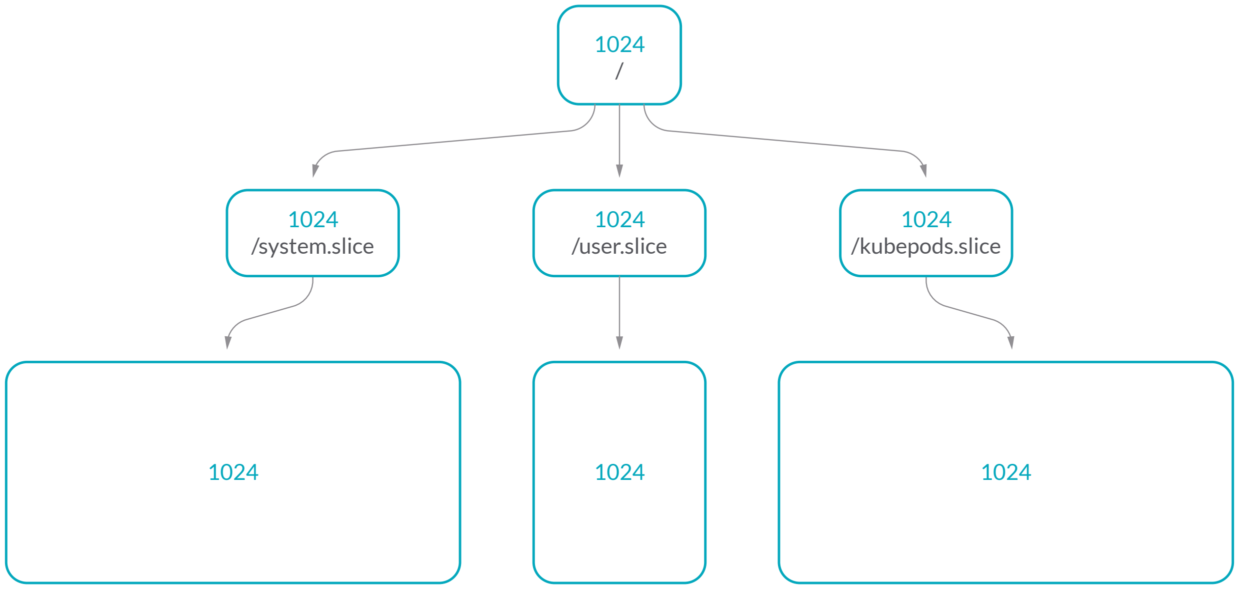
The pod - Deployment.yaml

```
kind: Deployment
apiVersion: extensions/v1beta1
metadata:
 name: redis
…
 template:
   spec:
     containers:
       - name: redis
         image: redis:5.0.3-alpine
         resources:
           requests:
             memory: 300Mi
            ●cpu: 500m
…
       - name: busybox
         image: busybox:1.28
         resources:
           requests:
             memory: 100Mi
            ●cpu: 100m
```

CPU shares

Once it has determined what Node to run it on (one that first survived the Filter and then had the highest Score), Kubernetes sets up the Linux CPU shares to roughly align with the mCPU metrics. CPU shares (cpu.share) are a Linux Control Groups (cgroups) feature.

Usually, CPU shares can be any number, and Linux uses the ratio between each cgroups' shares number to the total of all the shares to prioritize what processes get scheduled on the CPU. For example, if a process has 1,000 shares and the sum of all the shares is 3,000, it will get a third of the total time.



The example above belongs to a typical Kubernetes environment, where you'll find three cgroups under root cgroup: /system.slyce, /user.slice, and /kubepods.slice. For this example, we set every slice with a 33.3% of CPU (1024 from 3072 shares in total) if all the groups demand shares simultaneously.

In summary, Kubernetes uses cpu.share to allocate the CPU resources to pods placed under the / kubepod.slice cgroup. The Completely Fair Scheduler (CFS) in Linux leverages the value assigned to each cgroup (1024 for every cgroup in the diagram above) to allocate the CPU resources proportionally.

Multi-thread and multiple cores

If this is a single-threaded app (which can only ever run on one CPU core at a time), then this makes intuitive sense. If you set a limit to 1 CPU, you get 100 milliseconds (ms) of CPU time every 100ms, or all of it. The issue is when we have a multi-threaded app that can run across multiple CPUs at once.

If you have two threads, you can consume one CPU period (100ms) in as little as 50ms. And 10 threads can consume 1 CPU period in 10ms, leaving us throttled for 90ms every 100ms — 90% of the time! That usually leads to worse performance than if you had a single thread that is unthrottled.

Another critical consideration is that some apps or languages/runtimes will see the number of cores in the Node and assume it can use all of them, regardless of its requests or limits. Suppose our Kubernetes Nodes/Clusters/Environments are inconsistent regarding how many cores we have. In that case, the same Limit can lead to different behaviors between Nodes, given they'll be running different numbers of threads to correspond to the different quantities of Cores.

So, you either need to:

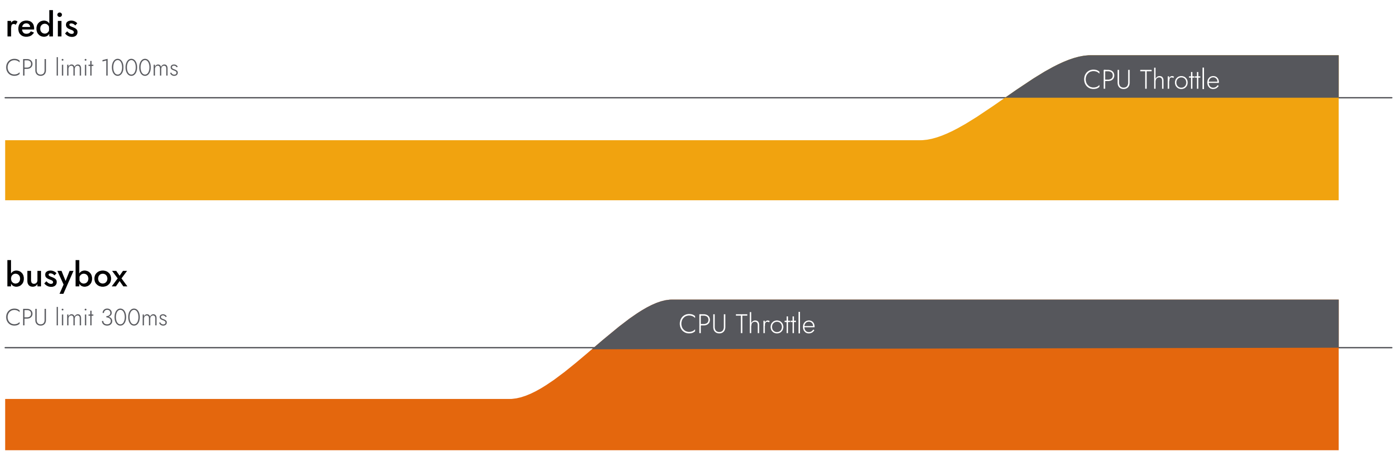
- Set the Limit to accommodate all of your threads.
- Lower the thread count to align with the Limit.

CPU Throttling

CPU Throttling in Kubernetes is a Kernel mechanism to implement CPU limits, where processes are slowed when they are about to reach some resource limits. It's common to see this phenomenon in Kubernetes environments where limits were not accurately set.

These limits could be:

- A Kubernetes Limit set on the container.
- A Kubernetes ResourceQuota set on the namespace.
- The node's actual CPU size.



CPU usage below limits
Everything works fine

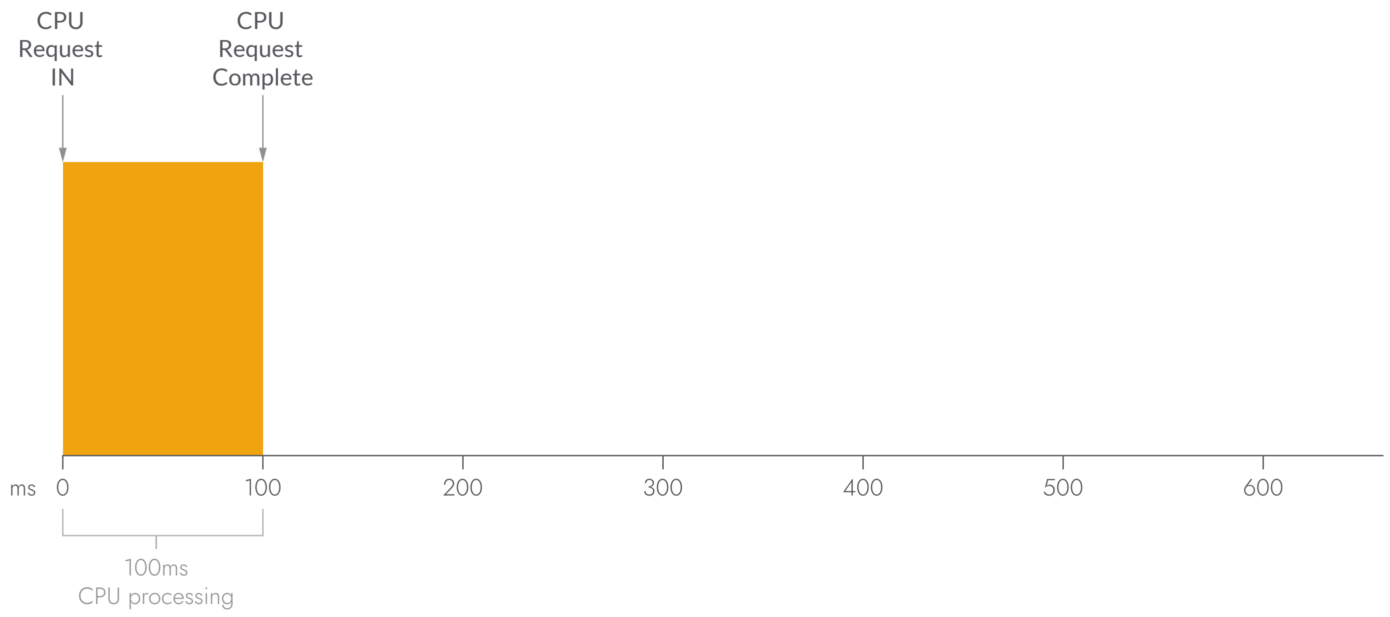
One container above limit
CPU Throttle kicks in
The other containers are "unaffected"

All containers above limit
No container is killed

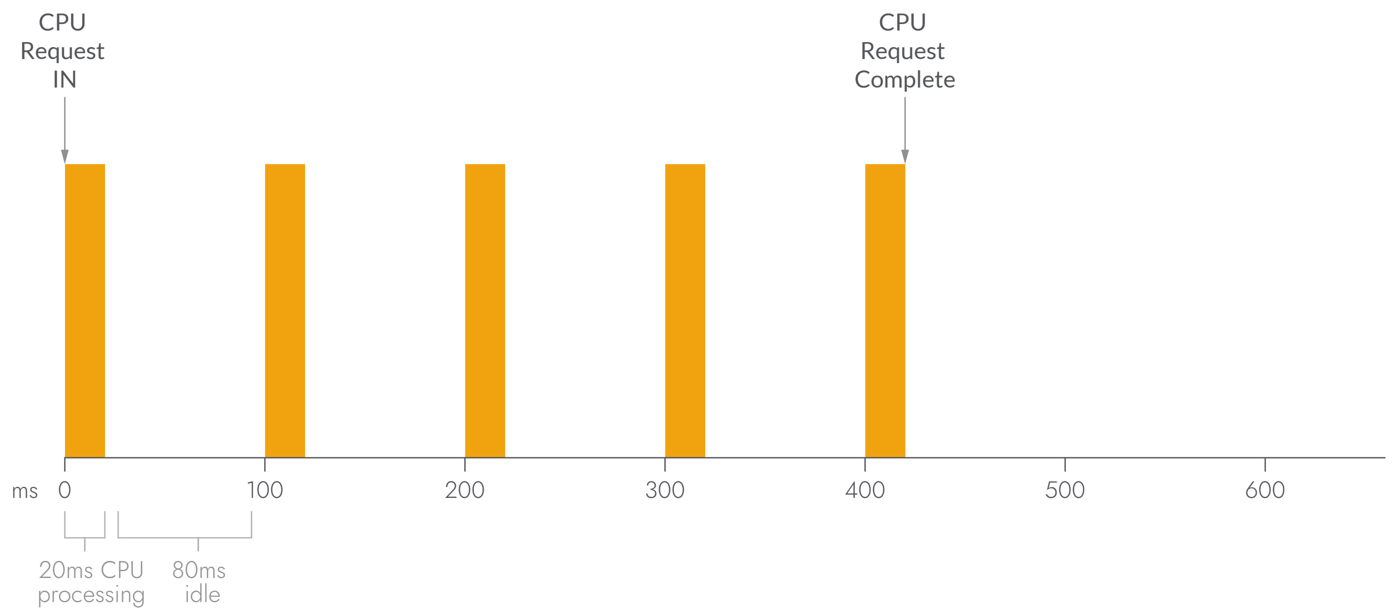
If a process or application goes above the limit, it will be throttled. In other words, it will get fewer CPU cycles, so the process or application may significantly slow down.

CPU considerations for scaling

Consider a single-thread process which runs for 100ms on a CPU. As can be seen in the next diagram, when there is no limit the process runs steadily for 100ms, then the CPU request is completed.



Now, let's set a 0.2 CPU limit, which means that a single-thread process will get about 20ms for each cycle of 100ms.



In this scenario, the same process will remain alive for a few 100ms cycles. It'll get CPU time for 20ms and the other 80ms will remain idle. Instead of completing this request in 100ms, it will need around 420ms in total.

When the CPU is throttled for an application or process, it may cause a lot of trouble, like performance degradation or even service outage. Consider double checking whether those CPU limits are appropriate and according to the application requirements.

Exercise

In this exercise, you will explore the concepts of CPU limits and throttling in Kubernetes.

Step 1: Deploy the sample application

Create two deployments, dep1 and dep2. The first one is a stress test with no limits, requests, or constraints. The second is a stress test that will consume up to two cores and allocate memory in chunks of 100 Mebibytes (Mi).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dep1
  labels:
    app: nginx
spec:
    containers:
    - name: stress
      image: vish/stress
–
apiVersion: apps/v1
kind: Deployment
metadata:
  name: dep2
  labels:
    app: stress2
spec:
  selector:
    matchLabels:
      app: stress2
  template:
    metadata:
      labels:
        app: stress2
    spec:
      containers:
      - name: stress
        image: vish/stress
        resources:
          limits:
            cpu: “1”
        args:
        - -cpus
        - “2”
        - -mem-alloc-size
        - “100Mi”
```

Step 2: Compare CPU consumption for both Pods:

Let's use the `kubectl top` command to retrieve the resource consumption for our Pods.

```
$ kubectl top pods
NAME                     CPU(cores)    MEMORY(bytes)
dep1-664d7ccfd4-h2wgm    1m            0Mi
dep2-67d48489dc-6sz42    1002m         1Mi
```

Step 3: Review the logs for memory

The stress container we deployed in our Pods is writing several log entries for its memory consumption. Let's check them with `kubectl logs`, which will return the entries written by a container in that Pod.

```
$ kubectl logs dep1-664d7ccfd4-h2wgm
I0602 11:21:19.819966       1 main.go:26] Allocating “0” memory, in “4Ki”
chunks, with a 1ms sleep between allocations
I0602 11:21:19.820005       1 main.go:29] Allocated “0” memory


$ kubectl logs dep2-67d48489dc-6sz42
I0602 11:23:13.806926       1 main.go:26] Allocating “0” memory, in “100Mi”
chunks, with a 1ms sleep between allocations
I0602 11:23:13.806995       1 main.go:39] Spawning a thread to consume CPU
I0602 11:23:13.807005       1 main.go:39] Spawning a thread to consume CPU
I0602 11:23:13.807009       1 main.go:29] Allocated “0” memory
```

Step 4: Try to redeploy dep2 without the CPU limit

You will see that two CPU cores are being used for the Pod in the second deployment.

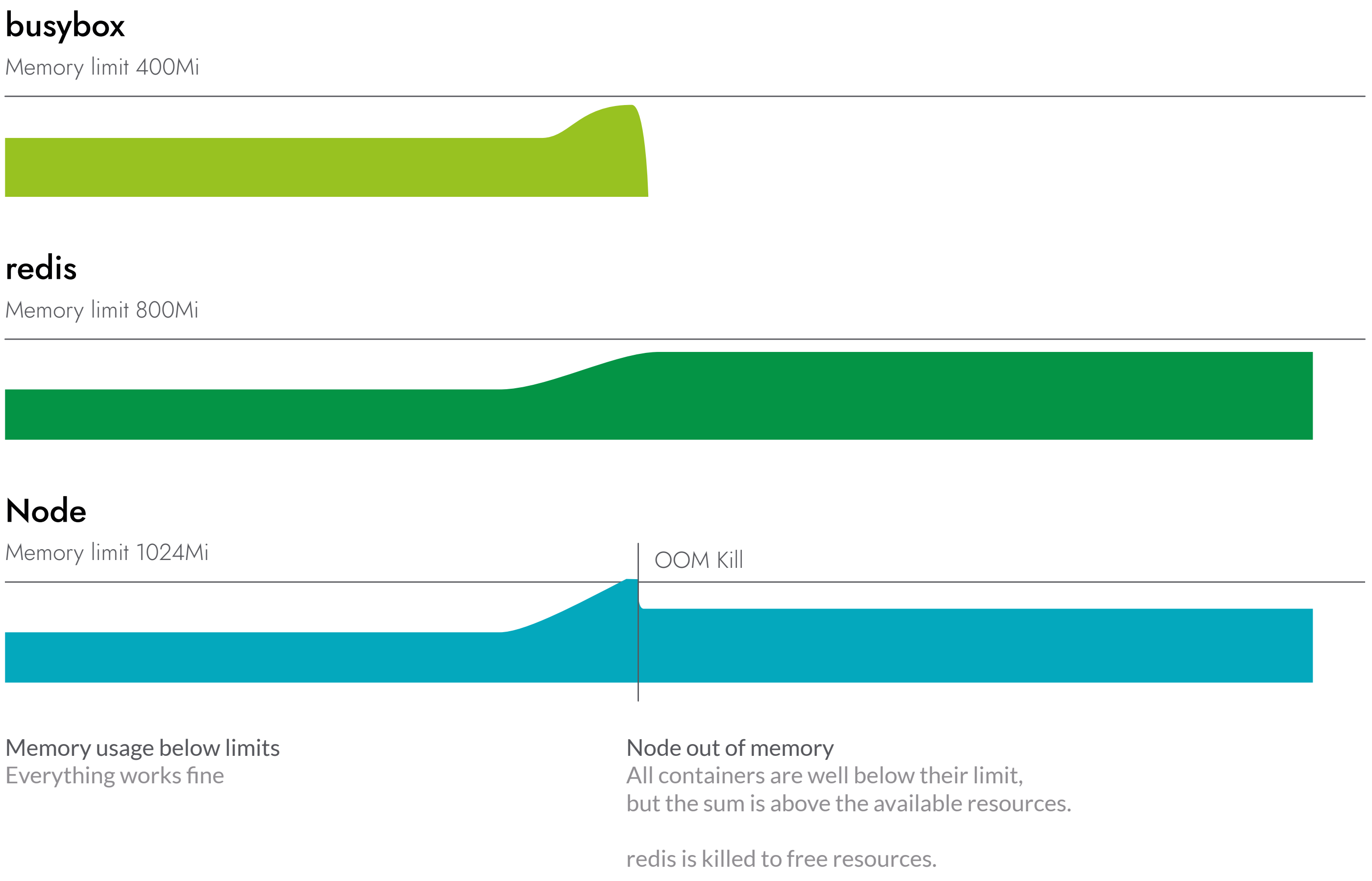
```
$ kubectl top pods
NAME                     CPU(cores)    MEMORY(bytes)
dep1-664d7ccfd4-h2wgm    0m            0Mi
dep2-78bdc7b6ff-6wzjn    2001m         1Mi
```

Memory

Memory is also a shared and finite resource. The way the Linux Kernel manages memory access and sharing is different from CPU, however. If a process tries to allocate more memory than is allowed to work, the Linux Kernel memory subsystem may intervene, killing the process.

Memory is measured in Kubernetes in bytes.

- You can use E, P, T, G, M, and k to represent Exabyte, Petabyte, Terabyte, Gigabyte, Megabyte, and kilobyte, although only the last four are commonly used (e.g., 500M, 4G).
- Warning: Don't use lowercase m for memory (this represents Millibytes, which is ridiculously low).
- You can define Mebibytes using Mi, as well as the rest as Ei, Pi, Ti (e.g., 500Mi).


busybox
Memory limit 400Mi
redis
Memory limit 800Mi
Node
Memory limit 1024Mi
OOM Kill
Memory usage below limits
Everything works fine
Node out of memory
All containers are well below their limit,
but the sum is above the available resources.
redis is killed to free resources.

As you can see in the diagram above, when a node runs out of memory (OOM), the Kubelet can proactively terminate Pods to reclaim resources on nodes. If the node experiences an OOM prior to the Kubelet being able to reclaim memory by terminating Pods, the node has to rely on the Kernel `oom_killer`, which will kill containers based on its own oom_score calculations for each container.

Memory considerations for scaling

One of the main characteristics of Kubernetes regarding memory is that there are some scenarios where a Pod might get evicted due to consumption:

- Node-pressure eviction
- Preemption
- Out of Memory (OOM) killed due to limits: If a container in a Pod consumes more memory than the actual limit, the container is killed with the message OOMKilled.

Node-pressure eviction

In cases where memory is critical for the functioning of the Node, `kubelet` might evict some Pods in order to retrieve memory. This is a constant evaluation from kubelet that takes into account Quality of Service to decide on which Pod to evict first. Quality of Service is further explained in the 'Kubernetes scaling at application level' chapter.

```
Reason:          Evicted
Message:         Pod The node had condition: [MemoryPressure].
```

Preemption

Every time a Pod is created in Kubernetes, it goes to a queue and waits to be scheduled. When the Kubernetes scheduler is not able to find an appropriate Node to satisfy Pod requests, the preemption logic is automatically activated. In order to schedule this new Pod in a Node with limited resources, another Pod (or multiple) with lower priority needs to be terminated to leave resources to the first one.

OOM killed due to limits

When certain containers try to get more resources than the actual limit in the container, the Linux OOM Killer terminates the process that tried to allocate memory. If that process is the container's PID 1, then Kubernetes will restart that container. Note that, unlike the evictions above, this happens at container level and not at Pod level.

When checking a Pod where a container has been OOMKilled, you should see an entry like this in the `kubectl describe` command:

```
   State:          Waiting
     Reason:       CrashLoopBackOff
   Last State:     Terminated
     Reason:       OOMKilled
```

Check this article for more information about OOM and Pod termination.

Exercise: Memory handling

In this exercise, you will simulate how a container gets over the limit in resource consumption and it's OOMKilled.

Step 1: Deploy the sample application

Create a yaml file under sample-deployment.yaml with the following deployment specification:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: flaviostutz/web-stress-simulator:latest
```

Apply the deployment to your Kubernetes cluster. It will create a web infrastructure under stress:

```
$ kubectl apply -f sample-deployment.yaml
```

Step 2: Verify the deployment

Ensure that the deployment is running and that the pod is ready:

```
$ kubectl get deployments
$ kubectl get pods
```

Confirm that the pod of the sample-app deployment is in the “Running” and “Ready” state.

Step 3: Analyze Memory usage

Retrieve the memory usage metrics for the pod:

```
$ kubectl top pods
NAME                              CPU(cores)    MEMORY(bytes)
my-deployment-797dcd58f5-9rvmh    2m            117Mi
```

Observe the memory utilization of the pod. Take note of the current memory usage and any fluctuations.

Step 4: Configure Memory requests and limits

Update the sample-deployment.yaml file to include memory limits for the container:

```
      containers:
        - name: sample-app
          image: flaviostutz/web-stress-simulator:latest
          resources:
            limits:
              memory: “100Mi”
```

Notice how the container is now OOMKilled:

```
$ kubectl get pods
my-deployment-75b88fff7d-l4pvk   0/1     OOMKilled          1 (19s ago)
```

Disk storage

While CPU and memory are often the centerpiece of resource handling in Kubernetes, disk storage management is an element that could be overlooked.

Chances are that your Kubernetes cluster will use Persistent Volumes for long-term storage. They are user-provisioned volumes that run independently of the Pods.

To allocate a certain storage size, Kubernetes includes an object called Persistent Volume Claim (PVC for short).

Like CPU and Memory, disk space in Kubernetes can be handled through limits and requests, but always using a PersistentVolumeClaim.

In Kubernetes, you can allocate a certain storage size for a `Persistent Volume` using `PersistentVolumeClaims`.

Since Kubernetes 1.24, you have the ability to use `PersistentVolumeClaimResize` in order to alter the amount of storage allocated at any given time. Note, however, that for it to work, you need to set `allowVolumeExpansion`: true in the StorageClass definition.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Gi
  storageClassName: <your-storage-class-name>
```

In order to scale a disk storage in a Kubernetes node, you can either:

1. If you are looking for vertical Pod storage scaling, you can dynamically resize your Pod volume storage (PV) size. Check the next section to learn more about that.
2. If you want to horizontally scale your PV, you can create a StatefulSet to horizontally scale the application and add a volumeClaimTemplate to the StatefulSet. That way, any time a new Pod is created, a new unique volume will also be provisioned.

Exercise: Disk storage scaling

In this exercise, you will practice scaling disk storage in Kubernetes by dynamically provisioning a Persistent Volume Claim (PVC) and expanding its volume size using dynamic volume expansion afterwards. You will gain hands-on experience in managing storage resources and observing the effects of volume expansion on your application.

Step 1: Create a new StorageClass

Create a file named my-storageclass.yaml and define a new dynamic StorageClass. In this example, we use AWS EBS as a provisioner. You can choose the StorageClass provider that best fits your needs.

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: my-storageclass
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: “10”
  fsType: ext4
allowVolumeExpansion: true
```

Step 2: Create a Persistent Volume Claim (PVC)

Create a file named pvc.yaml and define a PVC with a specific storage size. Bear in mind there is no need to create a PV first; using dynamic provisioning, Kubernetes will automatically provision that space every time it’s requested.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: my-storageclass
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Step 3: Check that PVC is provisioned

```
$ kubectl get pvc
```

Step 4: Expand the volume

Now, let's see how a volume can be expanded thanks to the Kubernetes volume expansion feature. Just edit the volume created in previous steps by increasing the `.spec.resources.requests.storage` field.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: my-storageclassmanual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi # specify the new size here
```

Control plane

When talking about scaling, the first instinct would be to focus on the worker nodes. It makes sense obviously, since your containers are running there and the need for scaling would be tied to the specifics of your application. But this is tricky, as scaling is also much needed for the control plane elements. Ignoring this might cause performance to drop over time.

etcd

etcd is a control plane component that acts as a high-availability key-value store, storing all cluster states from Kubernetes components, like Pods, containers, services, etc.

It works as a leader-based quorum system, where, if etcd goes down, the rest of Pods and workloads will keep running. In the meantime, a new leader will be elected by consensus using the RAFT algorithm. This consensus election in case of failure means that by running an odd number of nodes, you can guarantee that there always will be a majority of nodes winning, and split-brain scenarios will be avoided.

For that reason, the general suggestion is to create a cluster of an odd number of etcd members (3, 5, etc.), while in most cases three members would be enough. As a general rule, avoid autoscaling for etcd, and consider scaling from a three-member cluster to a five-member one if more reliability is required.

Please refer to this guide for more information on etcd monitoring.

CoreDNS

CoreDNS is a CNCF graduated project to create a DNS add-on for Kubernetes. It's aimed to be scalable and configurable via plugins to expand its functionality.

Note that, by default, many Kubernetes clusters have a functional CoreDNS running in their Control Plane, but it's not going to scale accordingly unless it's explicitly configured.

CoreDNS autoscaler

The CoreDNS autoscaler is a horizontal scaling tool provided in the CoreDNS helm chart.

After adding CoreDNS to your Kubernetes cluster, you can see the following deployments ready:

```
kube-system   coredns              2/2    2           2           20h
kube-system   coredns-autoscaler   1/1    1           1           20h
kube-system   coredns-coredns      1/1    1           1           99s
```

CoreDNS autoscaler has the following parameters:

- **coresPerReplica:** How many CoreDNS replicas should be created based on the number of cores in the cluster.
- **nodesPerReplica:** How many CoreDNS replicas should be created based on the number of worker nodes.
- **preventSinglePointFailure:** Ensure at least two CoreDNS replicas are running even if the above conditions are not met.

You can edit the coredns-autoscaler values by editing its configMap:

```
kubectl edit configmap coredns-autoscaler -n kube-system
```

```
apiVersion: v1
data:
   linear:
'{"coresPerReplica":256,"nodesPerReplica":16,"preventSinglePointFailure":true}'
kind: ConfigMap
metadata:
   creationTimestamp:"2023-05-2213:31:24Z"
   name: coredns-autoscaler
   namespace: kube-system
   resourceVersion: "937"

   uid: 349bc163-287-49aa-b278-60683a757ab3
```

Exercise: CoreDNS autoscaler

In this exercise, you will learn how to configure the CoreDNS horizontal autoscaler in Kubernetes to automatically scale the number of CoreDNS replicas based on the number of nodes and cores in the cluster. You will work with a sample cluster and configure the autoscaler to ensure efficient DNS resolution.

Prerequisites:

- Kubernetes cluster set up and properly configured.
- kubectl command-line tool installed and configured.

Step 1: Verify CoreDNS deployment

Ensure that CoreDNS is deployed and running in your cluster:

Confirm that the CoreDNS deployment is present and that the replicas are running and ready.

```
$ kubectl get deployment -l k8s-app=kube-dns --namespace=kube-system
NAME       READY   UP-TO-DATE   AVAILABLE   AGE
coredns    2/2     2            2           278d
```

Step 2: Create the CoreDNS autoscaler objects

In order to deploy the CoreDNS horizontal autoscaler, it is necessary to create a few Kubernetes objects. Create a new file named coredns-horizontal-autoscaler.yaml with the following content to create the required Kubernetes resources.

```
kind: ServiceAccount
apiVersion: v1
metadata:
  name: kube-dns-autoscaler
  namespace: kube-system
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: system:kube-dns-autoscaler
rules:
  - apiGroups: [“”]
    resources: [“nodes”]
    verbs: [“list”, “watch”]
  - apiGroups: [“”]
    resources: [“replicationcontrollers/scale”]
    verbs: [“get”, “update”]
  - apiGroups: [“apps”]
    resources: [“deployments/scale”, “replicasets/scale”]
    verbs: [“get”, “update”]
# Remove the configmaps rule once below issue is fixed:
# kubernetes-incubator/cluster-proportional-autoscaler#16
  - apiGroups: [“”]
    resources: [“configmaps”]
    verbs: [“get”, “create”]
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: system:kube-dns-autoscaler
subjects:
  - kind: ServiceAccount
    name: kube-dns-autoscaler
    namespace: kube-system
roleRef:
  kind: ClusterRole
  name: system:kube-dns-autoscaler
  apiGroup: rbac.authorization.k8s.io

---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: kube-dns-autoscaler
```

```
    namespace: kube-system
    labels:
      k8s-app: kube-dns-autoscaler
      kubernetes.io/cluster-service: “true”
  spec:
    selector:
      matchLabels:
        k8s-app: kube-dns-autoscaler
    template:
      metadata:
        labels:
          k8s-app: kube-dns-autoscaler
      spec:
        priorityClassName: system-cluster-critical
        securityContext:
          seccompProfile:
            type: RuntimeDefault
          supplementalGroups: [ 65534 ]
          fsGroup: 65534
        nodeSelector:
          kubernetes.io/os: linux
        containers:
        - name: autoscaler
          image: registry.k8s.io/cpa/cluster-proportional-autoscaler:1.8.4
          resources:
              requests:
                  cpu: “20m”
                  memory: “10Mi”
          command:
            - /cluster-proportional-autoscaler
            - --namespace=kube-system
            - --configmap=kube-dns-autoscaler
            # Should keep target in sync with cluster/addons/dns/kube-dns.yaml.
  base
            - --target=Deployment/coredns
            # When cluster is using large nodes(with more cores),
  “coresPerReplica” should dominate.
            # If using small nodes, “nodesPerReplica” should dominate.
            - --default-params={“linear”:{“coresPerReplica”:256,”nodesPerReplica”
  :16,”preventSinglePointFailure”:true,”includeUnschedulableNodes”:true}}
            - --logtostderr=true
            - --v=2
        tolerations:
        - key: “CriticalAddonsOnly”
          operator: “Exists”
        serviceAccountName: kube-dns-autoscaler
```

Create the Kubernetes objects with kubectl:

```
$ kubectl apply -f coredns-horizontal-autoscaler.yaml
serviceaccount/kube-dns-autoscaler created
clusterrole.rbac.authorization.k8s.io/system:kube-dns-autoscaler created
clusterrolebinding.rbac.authorization.k8s.io/system:kube-dns-autoscaler created
deployment.apps/kube-dns-autoscaler created
```

Step 3: Edit CoreDNS horizontal autoscaler based on your preferences

Edit the existing kube-dns-autoscaler ConfigMap and set parameters of your choice:

```
$ kubectl edit configmap kube-dns-autoscaler --namespace=kube-system
```

In this example, "min" has been modified to require a minimum of 3 CoreDNS replicas running:

```
data:
  linear: '{"min":3, "coresPerReplica":256, "includeUnschedulableNodes":true,
"nodesPerReplica":16, "preventSinglePointFailure":true}'
```

Step 4: Cleanup

Delete the CoreDNS autoscaler:

```
$ kubectl delete deployment kube-dns-autoscaler --namespace=kube-system
deployment.apps "kube-dns-autoscaler" deleted
```

Network

Container Network Interface plugins

In a Kubernetes cluster, the network infrastructure plays a crucial role in enabling communication between pods, services, and external clients. As your cluster grows and workloads increase, it becomes essential to properly scale the network to ensure optimal performance, reliability, and security.

Kubernetes allows users to deploy Container Network Interface (CNI) plugins for cluster networking. There are several CNIs available out there, such as Calico, OVN-Kubernetes, and Cisco ACI, among others. Every CNI may come with its own specific features and may or may not fit your needs. When choosing a CNI for your Kubernetes clusters, there are a lot of things to consider which may impact the performance, availability, and scalability of your Kubernetes cluster.

For example, CNI dataplane technology may vary from eBPF, standard Linux dataplane, to Vector Packet Processor (VPP), among others. Network model is also important as you will see following.

Overlay networks using Virtual Extensive LAN (VXLAN) for encapsulation are widely adopted. This model allows creating abstractions on top of existing networks for separating and securing logical networks. The overlay encapsulation process generates an IP header, which introduces a minimal overhead cost. When it comes to scaling the Kubernetes network, VXLAN offers enough flexibility to help Kubernetes and cloud deployments scale by encapsulating L2 Ethernet frames.

Border Gateway Protocol (BGP) can also be used for an unencapsulated model. This protocol is responsible for routing packets between Pods on top of a L3 network. Instead of using IP headers for encapsulation, BGP gets information on how to reach pods by using a kind of extended router network between Kubernetes nodes. Routes are dynamically updated at the OS level, which helps to reduce the latency.

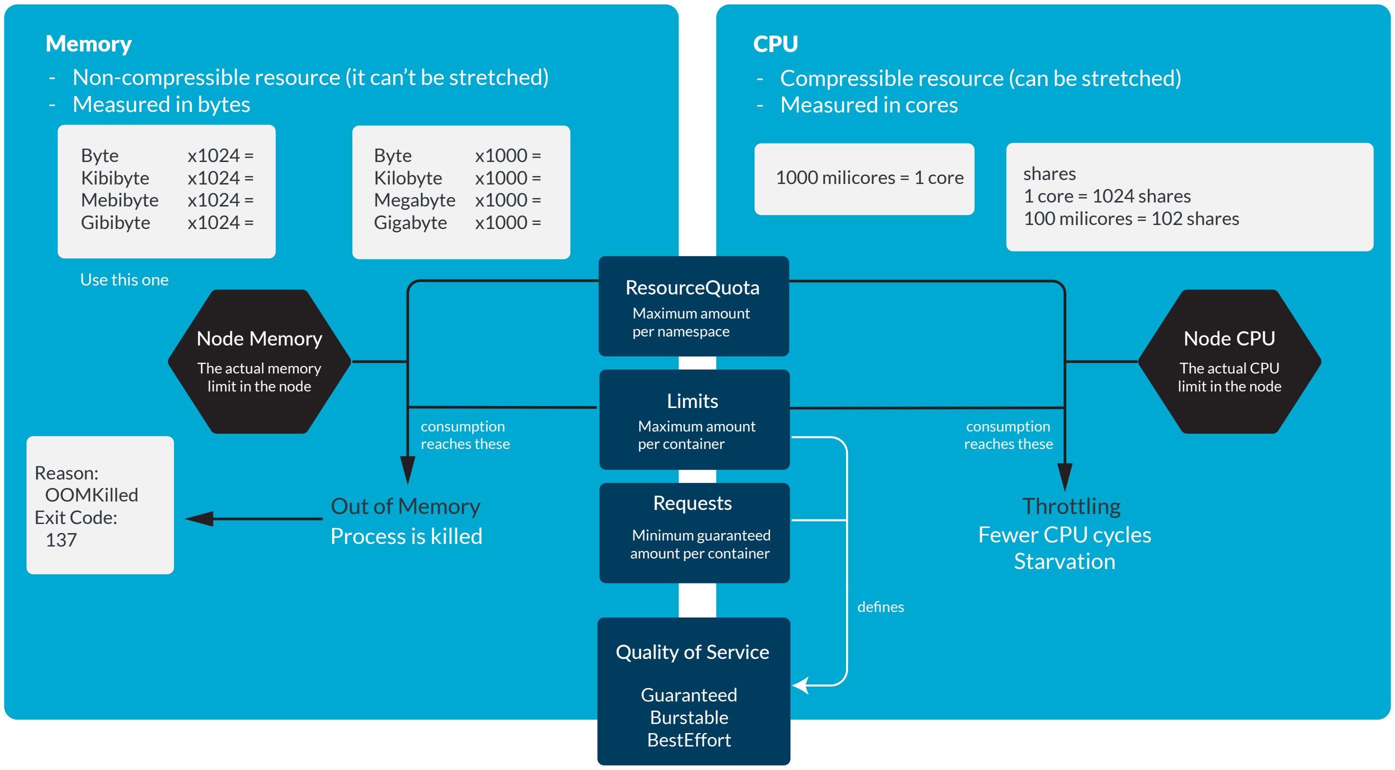
Load balancers in Kubernetes

One of the first considerations is whether you should be using a Load Balancer to guarantee that all traffic is equally distributed among your Pods. This can be achieved by using an external one by deploying the Kubernetes LoadBalancer object, which exposes the service externally using either an external one, or integrating some of your cloud providers' load balancers. By enabling that component, you will ensure the requests are distributed between the Pods in your Kubernetes cluster.

Another way to load balance traffic in a Kubernetes cluster is by using Ingress with one of the ingress controllers available. Ingress can provide externally reachable URLs, load balance traffic, and SSL/TLS termination among others.

Lessons learned

- Memory and CPU are the most important resources for containers in Kubernetes. Workloads need both resources to run.
- Going beyond memory can cause a lot of trouble, even process termination. In the end, if an application doesn't have enough memory available, it will be killed.
- CPU limits won't kill your application but will significantly slow down processes. If an application doesn't have enough CPU, it will be throttled and will wait to get CPU cycles.
- Control plane components, such as CoreDNS and etcd, are key for the proper operation of a Kubernetes cluster. DNS performance is critical, so ensure you have enough replicas to guarantee good response times. Etcd deployment must consist of an odd number of instances. Otherwise, you may face split brain related issues, losing quorum and synchronization of the cluster objects status.
- When it comes to the Kubernetes network, wisely choose the CNI plugin that best fits your needs. This needs to be done at very early stages, and is key to ensure your Kubernetes cluster can scale properly.
- Load balancing may be tackled from different angles. While an external cloud load balancer can be the common choice when running Kubernetes on your cloud provider, Kubernetes ingress may be a good choice when running on-prem clusters. Choose the load balancing model that best fits your needs.

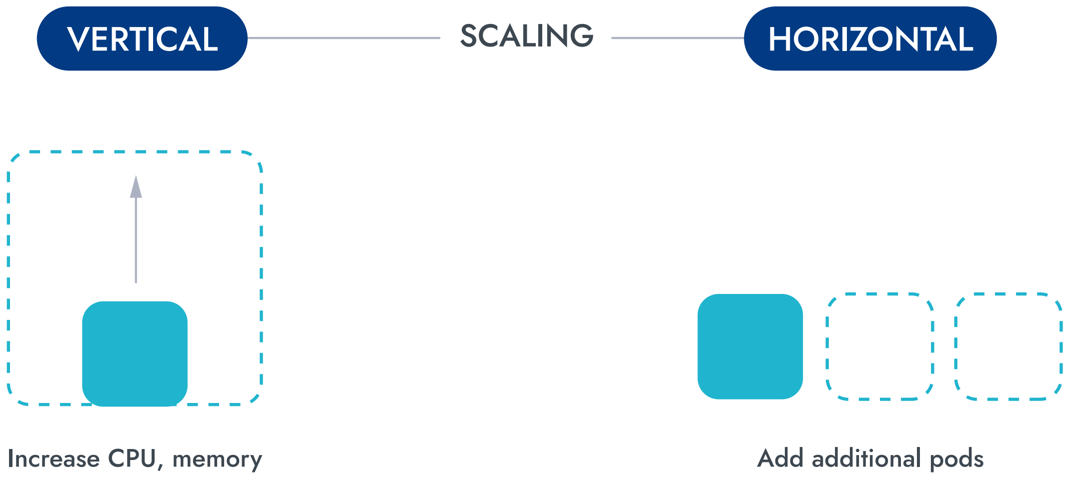


04

Sizing a Kubernetes cluster

Vertical scaling

Vertical scaling is the process of increasing or decreasing the resources of a Kubernetes node, like CPU or memory. In most of the cases, this will imply switching to a new node with more resources so the system's uptime might be impacted.



Vertical Pod autoscaler

Kubernetes includes a component called Vertical Pod Autoscaler that can be installed in the cluster. It's aimed to dynamically adjust the limits and requests of your containers based on the demand.

Horizontal scaling

Horizontal scaling (also known as application-level scaling) is the process of increasing or decreasing the number of replicas in a Kubernetes cluster, so resources are increased to match the current system usage.

Usually, horizontal scaling is done in Kubernetes by modifying the amount of Pods in a Node, although it's possible to scale horizontally by increasing the number of nodes. Many cloud vendors provide node autoscaler solutions in this manner.

HorizontalPodAutoscaler

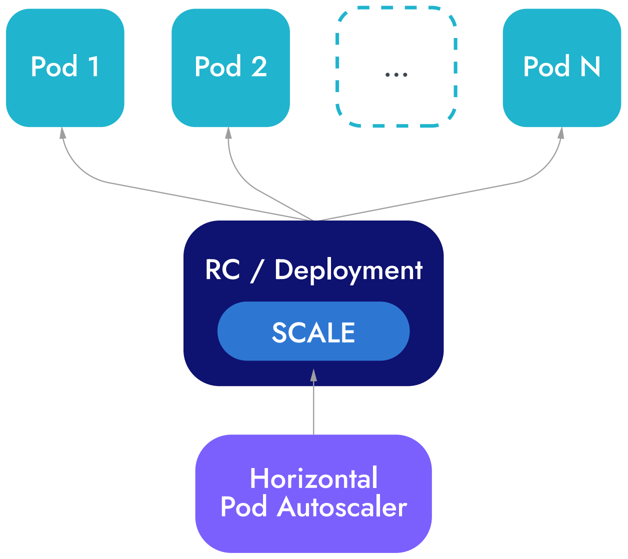
Kubernetes provides an object called HorizontalPodAutoscaler (HPA) for Pod autoscaling purposes. It will evaluate a metric and, if certain thresholds are passed, will scale down or scale up the workload.

The basic parameters that you will need for any HPA are:

- **Scale target:** The controller that this HPA will interact with.
- **minReplicas:** Minimum number of pods — the HPA cannot go below this value.
- **maxRepicas:** Maximum number of pods — the HPA cannot go above this value.
- **Target metric(s):** Metric (or metrics) used to evaluate current load and take scaling decisions.
- **targetValue:** Threshold value for the metric. If the metric readings are above this value, and currentReplicas < maxReplicas, HPA will scale up. Unfortunately, Kubernetes HPA only allows a single metric to be used for evaluation, which represents a limitation in the configuration for horizontal scaling.

The following formula is applied in any HPA:

$$\text{newReplicas}=[\text{currentReplicas } x \; \frac{\text{currentMetricValue}}{\text{targetValue}} \;]$$



Exercise:

Create an HPA with Kubectl using:

```
$ kubectl autoscale deployment nginx --min=2 --max=4 --cpu-percent=80
```

You will see that a new entry has been created as a HPA Kubernetes object:

```
$ kubectl get hpa
NAME    REFERENCE          TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
nginx   Deployment/nginx   0%/80%    2         5         4          114s
```

As you can see, the cpu-percent metric is used to control the amount of replicas where another Pod is scaled up if the threshold is reached.

Alternatively, HPA can be defined using a YAML definition:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-scaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 80
status:
  observedGeneration: 1
  lastScaleTime: <some-time>
  currentReplicas: 1
  desiredReplicas: 1
  currentMetrics:
  - type: Resource
    resource:
      name: cpu
      current:
        averageUtilization: 0
        averageValue: 0
```

Additionally, Kubernetes provides a parameter, called `stabilizationWindowSeconds`, which acts as a grace period to restrict the flapping of replica count when metrics used for scaling keeps fluctuating.

```
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
```

05

Kubernetes Scaling at Application Level

Limits and requests

Requests

Kubernetes defines requests as a guaranteed minimum amount of a resource to be used by a container. As seen in previous sections of this guide, the Linux Kernel provides such capability by using cgroups.

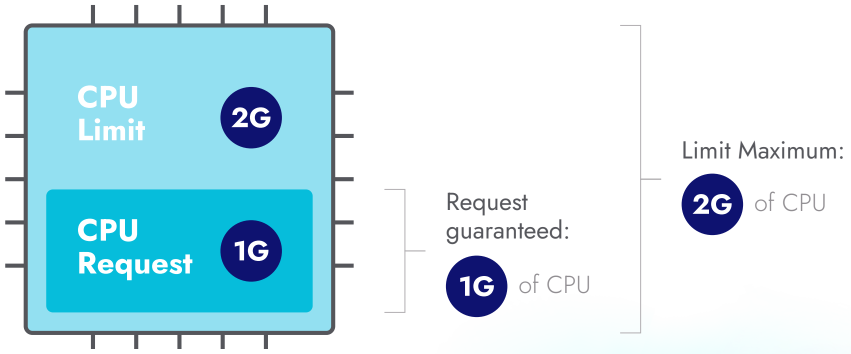
Basically, it will set the minimum amount of the resource for the container to consume.

When a Pod is scheduled, kube-scheduler will check the Kubernetes requests in order to allocate it to a particular Node that can satisfy at least that amount for all containers in the Pod. If the requested amount is higher than the available resource, the Pod will not be scheduled and remain in Pending status.

Limits

Kubernetes defines limits as a maximum amount of a resource to be used by a container.

This means that the container can never consume more than the memory amount or CPU amount indicated. When a container tries to surpass its limits, not only will it not be able to get such resources, but it will be throttled when requesting CPU or even killed when it comes to memory.



Quality of Service

In Kubernetes, Pods are giving one of three QoS Classes which will define how likely they are going to be evicted in case of lack of resources, from less likely to more likely:

- Guaranteed
- Burstable
- BestEffort

How are these QoS Classes assigned to Pods? This is based on limits and requests for CPU and memory.

Guaranteed

A Pod is assigned with a QoS Class of **Guaranteed** if:

- All containers in the Pod have both Limits and Requests set for CPU and memory.
- All containers in the Pod have the same value for CPU Limit and CPU Request.
- All containers in the Pod have the same value for memory Limit and memory Request.

A Guaranteed Pod won't be evicted in normal circumstances to allocate another Pod in the node.

Burstable

A Pod is assigned with a QoS Class of **Burstable** if:

- It doesn't have QoS Class of Guaranteed.
- Either Limits or Requests have been set for a container in the Pod.

A Burstable Pod can be evicted, but is less likely than the next category.

BestEffort

A Pod will be assigned with a QoS Class of **BestEffort** if:

- No Limits and Requests are set for any container in the Pod.

BestEffort Pods have the highest chance of eviction in case of a node-pressure process happening in the node.

In case of node-pressure eviction, Kubelet will start evicting Pods in order. First, it will try to evict Pods with Quality of Service equal to BestEffort. Then, Burstable Pods and, finally, Guaranteed.

PodDisruptionBudget

Disruptions are events (voluntary or involuntary) that cause unstable or problematic effects in a cloud environment.

PodDisruptionBudget (PDB) is a specific Kubernetes object that sets a limit to specific disruptions that might happen in your system. A PDB specifies the number of replicas that an application can tolerate having, relative to how many it's intended to have.

Parameters for PodDisruptionBudget:

- `minAvailable`: Minimum number of replicas that need to be available at all times. You can also set a percentage of the total (e.g., 25%).
- `maxUnavailable`: Maximum number of unavailable replicas.

Tip: If no replicas should ever be evicted due to criticality, you can achieve this by setting `minavailable: 100%` or `maxUnavailable: 0`.

Exercise: PodDisruptionBudget

In this exercise, you will learn how to apply PDBs in Kubernetes to ensure high availability of your applications during maintenance or node disruptions. You will work with a sample application and configure a PDB to limit the number of simultaneous pod disruptions.

Prerequisites:

- Kubernetes cluster set up and properly configured
- kubectl command-line tool installed and configured

Step 1: Deploy the sample application

Create a file named sample-deployment.yaml and define a Deployment for your sample application:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample-app
spec:
  replicas: 5
  selector:
    matchLabels:
      app: sample-app
  template:
    metadata:
      labels:
        app: sample-app
    spec:
      containers:
        - name: sample-app
          image: your-sample-app-image:latest
```

Apply the deployment to your Kubernetes cluster:

```
$ kubectl apply -f sample-deployment.yaml
```

Step 2: Verify the deployment

Ensure that the deployment is running and that all pods are ready:

```
$ kubectl get pods
```

Confirm that all pods of the sample-app deployment are in the “Running” and “Ready” state.

Step 3: Create a PodDisruptionBudget

Create a file named pdb.yaml and define a PodDisruptionBudget for your sample-app deployment:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: sample-app-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: sample-app
```

Apply the PodDisruptionBudget to your Kubernetes cluster:

```
$ kubectl apply -f pdb.yaml
```

Step 4: Test Pod disruptions

Verify the PodDisruptionBudget:

```
$ kubectl describe pdb sample-app-pdb
```

The output should show that the minimum number of available pods is set to 3.

Trigger a disruption to test the PodDisruptionBudget. You can scale down the deployment temporarily:

```
$ kubectl scale deployment sample-app --replicas=2
```

Monitor the pods to ensure that the number of available pods doesn't drop below the minimum specified in the PDB.

Once the deployment stabilizes, scale it back up to the original number of replicas:

```
$ kubectl scale deployment sample-app --replicas=5
```

Step 5: Cleanup

Delete the PodDisruptionBudget and deployment:

```
$ kubectl delete pdb sample-app-pdb
$ kubectl delete deployment sample-app
```

06

LimitRanges

LimitRanges are a Kubernetes policy that restricts the resource settings for each entity in a namespace.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-resource-constraint
spec:
  limits:
  - default:
      cpu: 500m
    defaultRequest:
      cpu: 500m
    min:
      cpu: 100m
    max:
      cpu: "1"
    type: Container
```

Where the values indicate the following:

- **default:** Containers created will have this value if none is specified.
- **min:** Containers created can't have limits or requests smaller than this.
- **max:** Containers created can't have limits or requests bigger than this.

Note that you can also define limitranges to control the disk space by affecting the PersistentVolumeClaim definitions in your cluster:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: storagelimits
```

```
spec:
  limits:
  - type: PersistentVolumeClaim
    max:
      storage: 2Gi
    min:
      storage: 1Gi
```

Scaling velocity

One of the most important aspects of scaling is how it should happen to adapt to the customer needs, which are often tied to the actual consumer use case:

- An e-commerce application might receive a sudden spike of visits when a product is online.
- A critical application should never go down, regardless of resource consumption.

Because of that, Kubernetes created a specific configurable velocity for autoscaling, which tries to address these issues.

Example of behaviors that you can configure with it:

- Scale up as fast as possible
- Scale down gradually
- Scale up gradually
- Never scale down

Or, any combination of the modes above.

Horizontal scaling with Keda

Keda is an event-driven, open source autoscaler for Kubernetes that can be used along Prometheus metrics.

Its usage is currently out of scope for this guide, but you can check the official repo for Keda and a step-by-step explanation on how to create an HPA scaling config based on Prometheus metrics.

Horizontal scaling pros and sons

- ✓ Pods in Kubernetes are abstractions made to allow seamless horizontal scaling
- ✓ Can be part of an automatic evaluation process
- ✓ No downtime
- ✗ Higher learning curve needed

Vertical scaling pros and cons

- ✅ Lower learning curve needed
- ❎ May imply downtime in case changes in the node are needed
- ❎ Meant as a one-time solution rather than a continuous evaluation process

Cluster scaling pros and cons

- ❎ Costs can skyrocket as the vendors costs are multiplicative
- ❎ Scaling down might not be possible or may have negative side effects, as it implies draining a node

Lessons learned

- Horizontal scaling is the process of increasing or decreasing the number of replicas in a Kubernetes cluster, so resources are increased to match the current system usage. The same principle can also be applied to Kubernetes nodes horizontal scaling.
- Vertical scaling is the process of increasing or decreasing the resources of the Kubernetes nodes or Pods.
- Keda is an open source solution that can be used to autoscale your cluster based on specific metrics.

07

Kubernetes Capacity Planning

What is capacity planning?

Historically, capacity planning has been defined as the process to determine the future needs of your project.

More specifically, Kubernetes capacity planning is the process of evaluating and forecasting the resource needs of a Kubernetes cluster.

Ideally, this will eventually become a loop where you're constantly:

- Improving observability of your metrics
- Adding additional visualization for your resources
- Changing the values for resources and limits

Steps for capacity planning a K8s cluster

Capacity planning should be more of a continuous process where it's always striving to achieve the best balance between consumption and costs.

We can summarize the process for capacity planning under these steps:

1. Monitor
2. Calculate average utilization
3. Resize
4. Monitor
5. Tweak

Monitor

As a first step, you need to get observability in place on your system. Get the important metrics from your system, especially those involving CPU, memory, and other resources you want to rightsize.

Calculate average utilization

Here's a sample PromQL query to retrieve the average utilization of containers in your cluster, grouped by namespace:

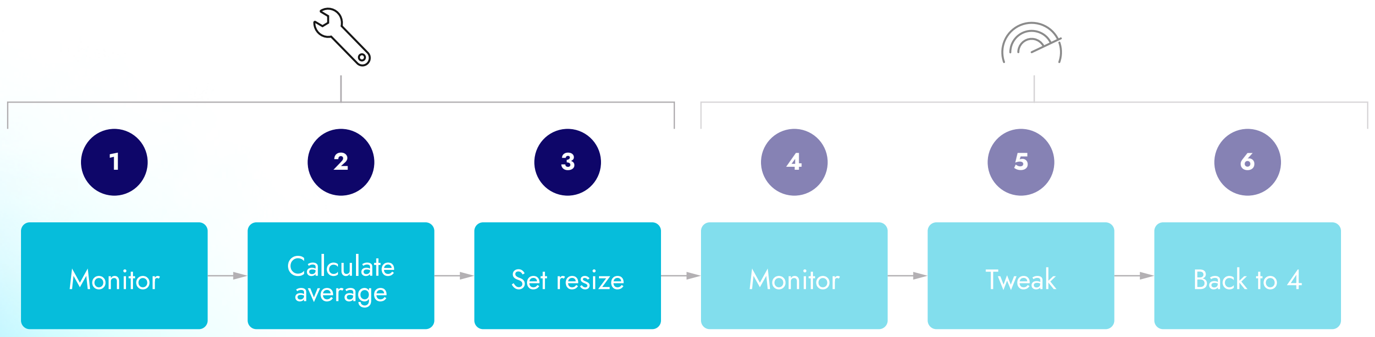
```
avg by (namespace,owner_name,container)((rate(container_cpu_usage_seconds_
total{container!="POD",container!=""}[5m])) * on(namespace,pod) group_
left(owner_name) avg by (namespace,pod,owner_name)(kube_pod_owner{owner_
kind=~"DaemonSet|StatefulSet|Deployment"}))
```

You will also need to analyze if the average is a good measurement for your application, as chances are that your traffic and usage has spikes or deviations from the average.

Resize

As a rule of thumb, you can set the requests of containers in your system with a value between 85% and 115% of the average.

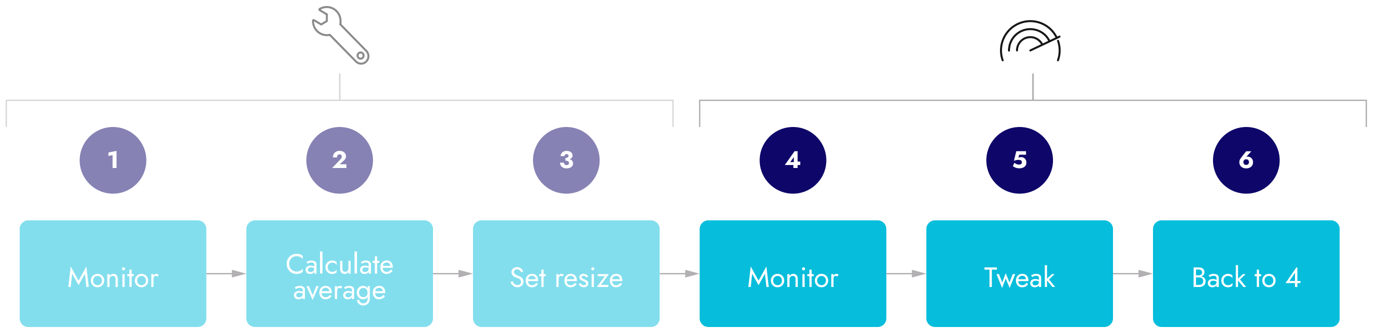
Rightsizing the workloads



Tweak

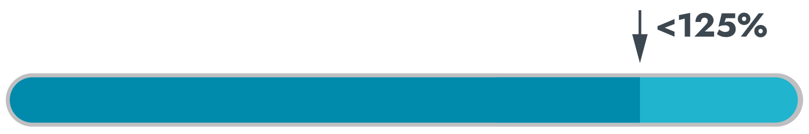
Over time, you will need to constantly review both the metrics you are using and the consumption for resources in order to adjust the value for your resource limits.

Checking the impact



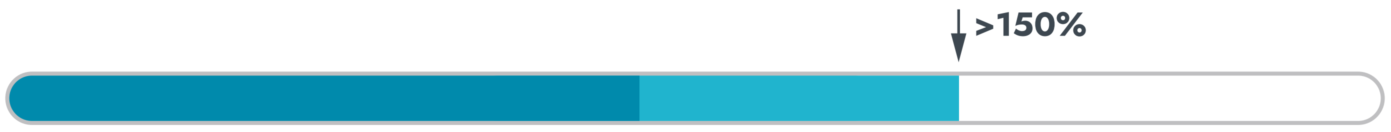
Limit overcommit

Conservative



Limited risk of resource starvation
More chances of resource wasting

Aggresive



Increase resource exploitation
More chances of resource starvation in the node

08

Considerations for multi-tenant

In small companies or organizations, normally, a cluster is dedicated exclusively to a single team or application. But, in reality, companies are more complex. They usually want to share their clusters between multiple services and departments while granting a fair share for each of them.

A multi-tenant cluster implies that the following elements might be shared:

- Compute power
- Network resources
- Workloads

Usually, this sharing is achieved by using namespaces to effectively assign these resources to specific Pods and containers within the same node.

While multi-tenancy has many implications, like security isolations and network policies, this guide will focus only on the sizing limitations.

Resource Quotas

Resource Quotas are a key Kubernetes object for multi-tenant, as they provide a way to assign a maximum limit for a resource per namespace.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resource-quota
spec:
  hard:
    pods: “4”
    requests.cpu: “1”
    requests.memory: 1Gi
    limits.cpu: “2”
    limits.memory: 2Gi
```

This means that for the whole namespace, the sum of all limits in the namespace can’t be higher than 2 cores of CPU and 2 Gibibytes of memory, while the sum of all requests in the namespace can’t be higher than 1 core of CPU and 1 Gibibyte of memory.

09

Rightsizing with Prometheus and KSM

In order to correctly size the resources of our Kubernetes cluster, we need to have visibility on what's happening inside.

These are the essential elements needed:

- Prometheus: The popular OSS for cloud monitoring.
- Kube state metrics (KSM): A service that provides metrics on Kubernetes objects.
- cAdvisor (container advisor): A daemon that retrieves information about running containers.

In this section, we will review different Prometheus metrics and their related PromQL queries in order to get visibility on the current usage of Kubernetes resources.

Detecting idle cores

Using the information given by container_cpu_usage_seconds_total and kube_pod_container_resource_requests, you can detect how many CPU cores are underutilized.

Basically, this PromQL query subtracts the current usage by the requests assigned to containers in your Pods. It will only make sense to use it if you're actively using Kubernetes requests.

```
sum((rate(container_cpu_usage_seconds_total{container!="POD",container!=""}
[30m]) - on (namespace,pod,container) group_left avg by
(namespace,pod,container)(kube_pod_container_resource_requests{resource="cpu"}))
* -1 >0)
```

Finding the top 10 containers that are CPU oversized

As we covered in our PromQL getting started guide, you can use the **topk** function to easily get the top n results for a PromQL query — just like this:

```
topk(10,sum by (namespace,pod,container)((rate(container_cpu_usage_seconds_
total{container!="POD",container!=""}[30m]) - on (namespace,pod,container)
group_left avg by (namespace,pod,container)(kube_pod_container_resource_
requests{resource="cpu"})) * -1 >0))
```

Detecting unused memory

You can use the information from container_memory_usage_bytes and kube_pod_container_resource_requests to see how much memory you are wasting.

```
sum((container_memory_usage_bytes{container!="POD",container!=""} - on
(namespace,pod,container) avg by (namespace,pod,container)(kube_pod_container_
resource_requests{resource="memory"})) * -1 >0 ) / (1024*1024*1024)
```

Percentage of memory overcommitted

In case you're using limits to cap the amount of memory that a container might consume, you can check in which cases this value is way higher than the actual usage.

```
100 * sum(kube_pod_container_resource_limits{container!="",resource="memory"} )
/ sum(kube_node_status_capacity_memory_bytes)
```

Percentage of CPU overcommitted

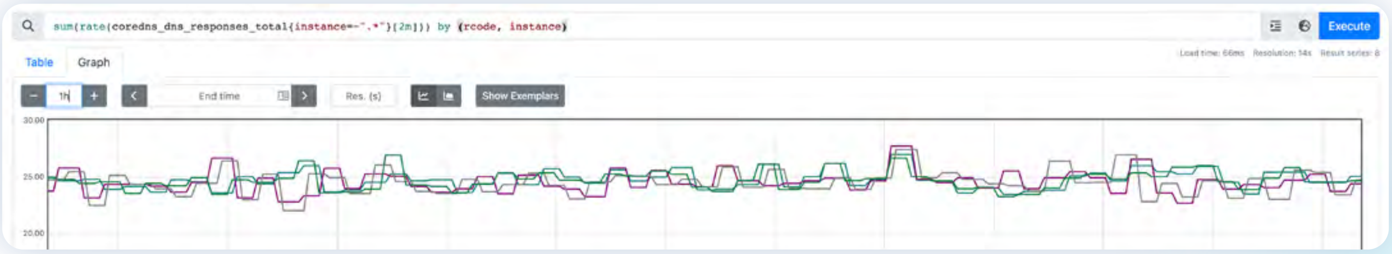
Alternatively, you might want to compare the current limits set to containers with the actual usage in order to detect limits that are way above the values that CPU might reach.

```
100 * sum(kube_pod_container_resource_limits{container!=””,resource=”cpu”} ) /
sum(kube_node_status_capacity_cpu_cores)
```

CoreDNS scaling

As we mentioned in the CoreDNS section, as your cluster scales, it’s key to check how well control plane components are growing to accept this higher demand.

```
(sum(rate(coredns_dns_requests_total{instance=~”.*”}[2m])) by (type,instance))
```



```
sum(rate(coredns_dns_responses_total{instance=~".*"}[2m])) by (rcode, instance)
```

Load time: 66ms Resolution: 14s Result series: 8

10

Rightsizing with Sysdig Monitor

Sysdig Monitor offers more comprehensive Kubernetes and cloud monitoring in a simple SaaS offering that is 100% Prometheus-compatible.

It also provides out-of-the-box dashboards with the essential metrics to track regarding Kubernetes sizing.

Underutilization of allocated resources Dashboard

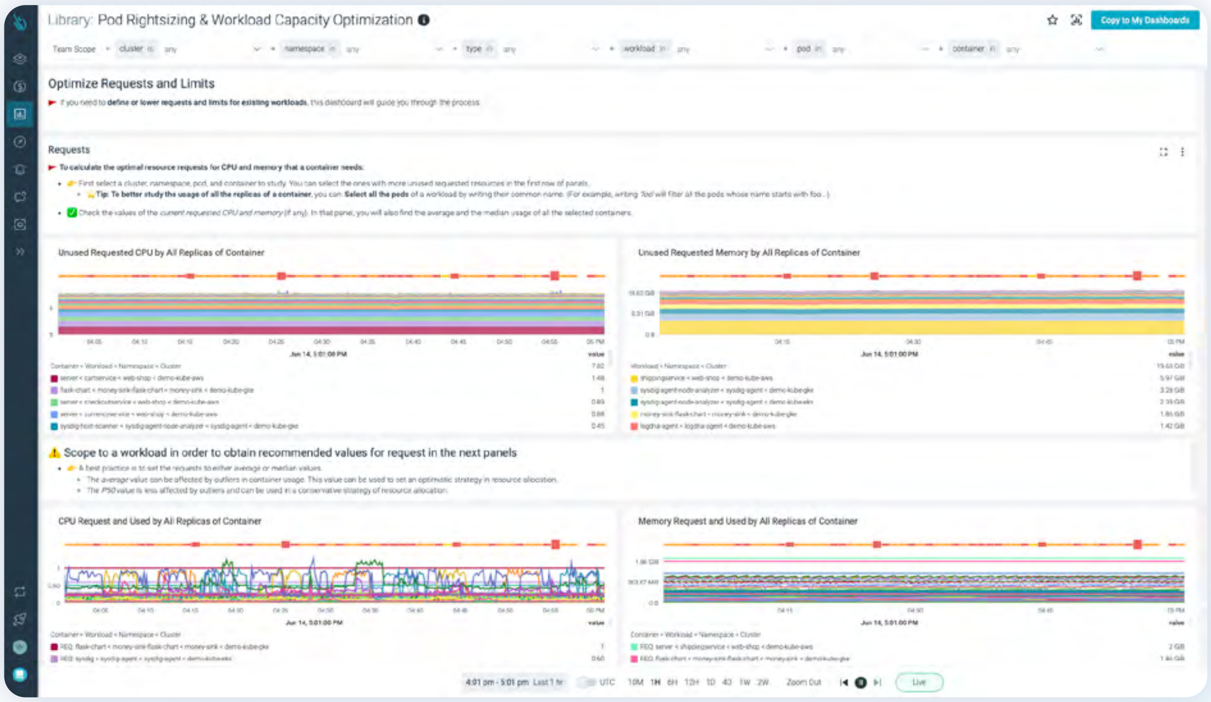
Allocation is defined as the combination of limits and requests to ensure that the normal usage of a container in a Kubernetes Pod receives both a fair share of the resource and can't go beyond a certain threshold.

With this dashboard, we can compare current allocation with the usage, aggregated per namespace and cluster.

Note, this dashboard shows both CPU and memory information.

Instructions to add the dashboard in Sysdig Monitor:

1. Go to Dashboards
2. Select Dashboard Manager
3. Find "Pod Rightsizing & Workload Capacity Optimization"


Library: Pod Rightsizing & Workload Capacity Optimization
Copy to My Dashboards
Team Scope
cluster in any
namespace in any
type in any
workload in any
pod in any
container in any
Unused Requested CPU by All Replicas of Container
Unused Requested Memory by All Replicas of Container
CPU Request and Used by All Replicas of Container
Memory Request and Used by All Replicas of Container
Live

Optimize Requests and Limits

If you need to **define or lower requests and limits for existing workloads**, this dashboard will guide you through the process

Requests

To calculate the optimal resource requests for CPU and memory that a container needs:

- First select a cluster, namespace, pod, and container to study. You can select the ones with more unused requested resources in the first row of panels.
  - **Tip: To better study the usage of all the replicas of a container**, you can **Select all the pods** of a workload by writing their common name. (For example, writing 'foo' will filter all the pods whose name starts with foo...)
- ✅ Check the values of the *current requested CPU and memory* (if any). In that panel, you will also find the average and the median usage of all the selected containers.

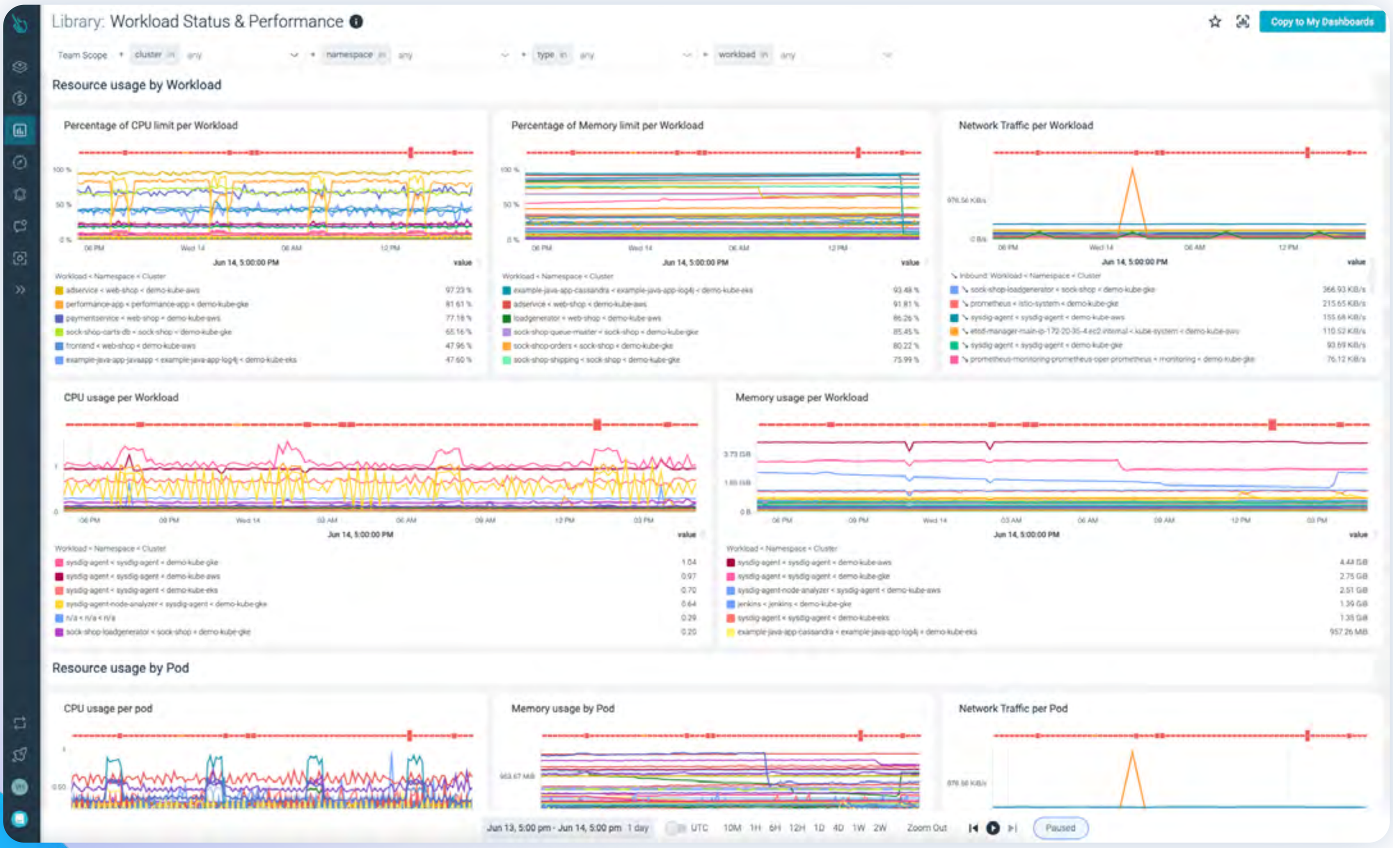
⚠ Scope to a workload in order to obtain recommended values for request in the next panels

- A best practice is to set the requests to either *average* or *median* values.
  - The *average* value can be affected by outliers in container usage. This value can be used to set an optimistic strategy in resource allocation.
  - The *P50* value is less affected by outliers and can be used in a conservative strategy of resource allocation.

Optimize CPU and memory allocation

Whether the CPU or memory is under-utilized or over-utilized, this Sysdig Monitor dashboard gives you the possibility of drilling down through the real usage of resources per Pod or even per workload running in a Pod. Thanks to this invaluable information, Sysdig Monitor users can learn more about their resource usage trends and optimize accordingly.

1. Go to Dashboards
2. Select Dashboard Manager
3. Find "Workload Status & Performance"


Library: Workload Status & Performance
Copy to My Dashboards
Team Scope
cluster in any
namespace in any
type in any
workload in any
Resource usage by Workload
Percentage of CPU limit per Workload
Percentage of Memory limit per Workload
Network Traffic per Workload
CPU usage per Workload
Memory usage per Workload
Resource usage by Pod
CPU usage per pod
Memory usage by Pod
Network Traffic per Pod
Jun 13, 5:00 pm - Jun 14, 5:00 pm 1 day
UTC
10M 1H 6H 12H 1D 4D 1W 2W
Zoom Out
Paused

Percentage of CPU limit per Workload

Workload < Namespace < Cluster	value
🟨 adservice < web-shop < demo-kube-aws	97.23 %
🟧 performance-app < performance-app < demo-kube-gke	81.61 %
🟦 paymentservice < web-shop < demo-kube-aws	77.18 %
🟩 sock-shop-carts-db < sock-shop < demo-kube-gke	65.16 %
🟦 frontend < web-shop < demo-kube-aws	47.96 %
🟦 example-java-app-javaapp < example-java-app-log4j < demo-kube-eks	47.60 %

Percentage of Memory limit per Workload

Workload < Namespace < Cluster	value
🟦 example-java-app-cassandra < example-java-app-log4j < demo-kube-eks	93.48 %
🟥 adservice < web-shop < demo-kube-aws	91.81 %
🟩 loadgenerator < web-shop < demo-kube-aws	86.26 %
🟪 sock-shop-queue-master < sock-shop < demo-kube-gke	85.45 %
🟧 sock-shop-orders < sock-shop < demo-kube-gke	80.22 %
🟩 sock-shop-shipping < sock-shop < demo-kube-gke	75.99 %

Network Traffic per Workload

↘ Inbound: Workload < Namespace < Cluster	value
🟦 ↘ sock-shop-loadgenerator < sock-shop < demo-kube-gke	366.93 KiB/s
🟥 ↘ prometheus < istio-system < demo-kube-gke	215.65 KiB/s
🟦 ↘ sysdig-agent < sysdig-agent < demo-kube-aws	155.68 KiB/s
🟧 ↘ etcd-manager-main-ip-172-20-35-4.ec2.internal < kube-system < demo-kube-aws	110.52 KiB/s
🟩 ↘ sysdig-agent < sysdig-agent < demo-kube-gke	93.69 KiB/s
🟥 ↘ prometheus-monitoring-prometheus-oper-prometheus < monitoring < demo-kube-gke	76.12 KiB/s

CPU usage per Workload

Workload < Namespace < Cluster	value
🟥 sysdig-agent < sysdig-agent < demo-kube-gke	1.04
🟥 sysdig-agent < sysdig-agent < demo-kube-aws	0.97
🟥 sysdig-agent < sysdig-agent < demo-kube-eks	0.70
🟨 sysdig-agent-node-analyzer < sysdig-agent < demo-kube-gke	0.64
🟦 n/a < n/a < n/a	0.29
🟪 sock-shop-loadgenerator < sock-shop < demo-kube-gke	0.20

Memory usage per Workload

Workload < Namespace < Cluster	value
🟥 sysdig-agent < sysdig-agent < demo-kube-aws	4.44 GB
🟥 sysdig-agent < sysdig-agent < demo-kube-gke	2.75 GB
🟦 sysdig-agent-node-analyzer < sysdig-agent < demo-kube-aws	2.51 GB
🟦 jenkins < jenkins < demo-kube-gke	1.39 GB
🟥 sysdig-agent < sysdig-agent < demo-kube-eks	1.35 GB
🟨 example-java-app-cassandra < example-java-app-log4j < demo-kube-eks	957.26 MB

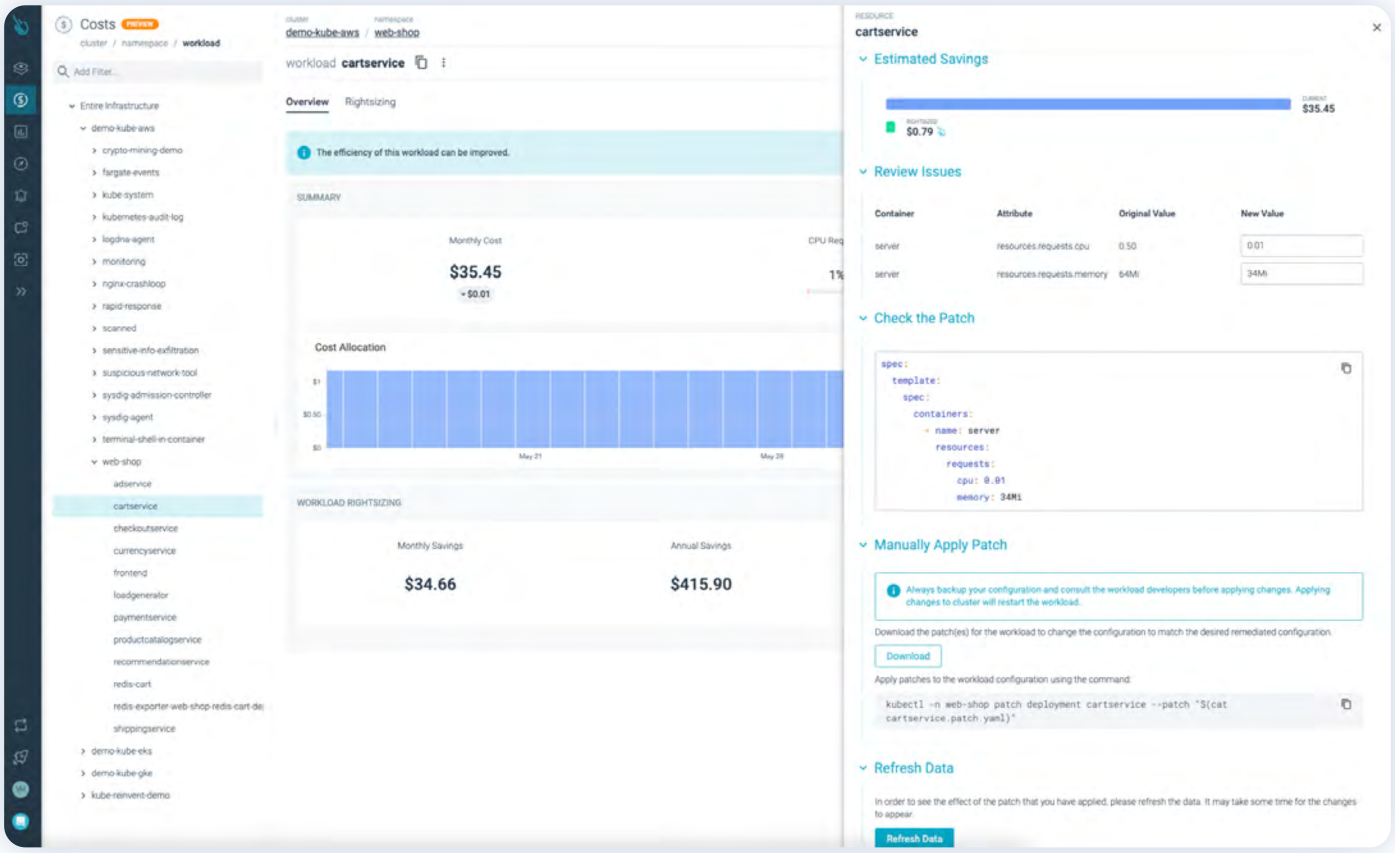
Cost Advisor

Cost Advisor is a tool inside Sysdig Monitor that pulls cost data from cloud providers.

Cost Advisor automatically pulls cost data from cloud providers, and this is enriched with Kubernetes context to show a line-by-line item of costs by clusters and workloads. The unified view of utilization, performance, and costs insights gives essential data to associate costs with teams for performing chargebacks, and helps drive accountability.

Cost Advisor intelligently identifies workloads that can be optimized, and users can prioritize where to focus efforts with estimated savings. By looking at the historical utilization data Sysdig collects, a baseline is established and recommendations are made on how to size workloads. This allows teams to confidently rightsize workloads without sacrificing application performance or availability. That includes a one-liner to apply a change from the command line, as well as the corrections to make at source within Git using infrastructure as code.

With Cost Advisor, organizations can establish FinOps best practices, including a culture of cost discipline where all teams review and optimize costs.



The efficiency of this workload can be improved.

Container	Attribute	Original Value	New Value
server	resources.requests.cpu	0.50	0.01
server	resources.requests.memory	64Mi	34Mi

```
spec:
  template:
    spec:
      containers:
        - name: server
          resources:
            requests:
              cpu: 0.01
              memory: 34Mi
```

Always backup your configuration and consult the workload developers before applying changes. Applying changes to cluster will restart the workload.

Download the patch(es) for the workload to change the configuration to match the desired remediated configuration.

Apply patches to the workload configuration using the command:

```
kubectl -n web-shop patch deployment cartservice --patch "$(cat
cartservice.patch.yaml)"
```

In order to see the effect of the patch that you have applied, please refresh the data. It may take some time for the changes to appear.

11

Conclusion

In this guide, you have learned more about the importance of rightsizing your Kubernetes applications. If you are new to this, adopting this process into your workflow and application lifecycle is highly recommended. That way you'll be able to anticipate performance and availability issues if you run short on resources. This approach will help you avoid wasting resources, therefore reducing your wasted spending.

Capacity planning and proper application monitoring is key to get performance and resource usage insights. If you are interested in expanding your knowledge on Kubernetes monitoring, check out this Kubernetes Monitoring Guide. Get tips and tricks to monitor your Kubernetes control plane and cloud-native applications more efficiently.

Along with that performance and resource usage information, it is also necessary to rely on a Cost Advisor tool that can leverage that data to provide remediation steps and reduce your wasted spending right away.

If you want to learn more about how to reduce your cloud-native application costs, check out the 5 Keys to Optimizing Costs of Running Cloud-Native Apps.

How Sysdig Monitor Helps

Sysdig provides observability into your cloud and Kubernetes workloads to simplify monitoring and lower costs. Get immediate granular insights and remediation steps to troubleshoot rapidly changing container environments. Sysdig Monitor helps you observe more and spend less. You can request a 30 days trial account and try it for free.

sysdig.com/start-free