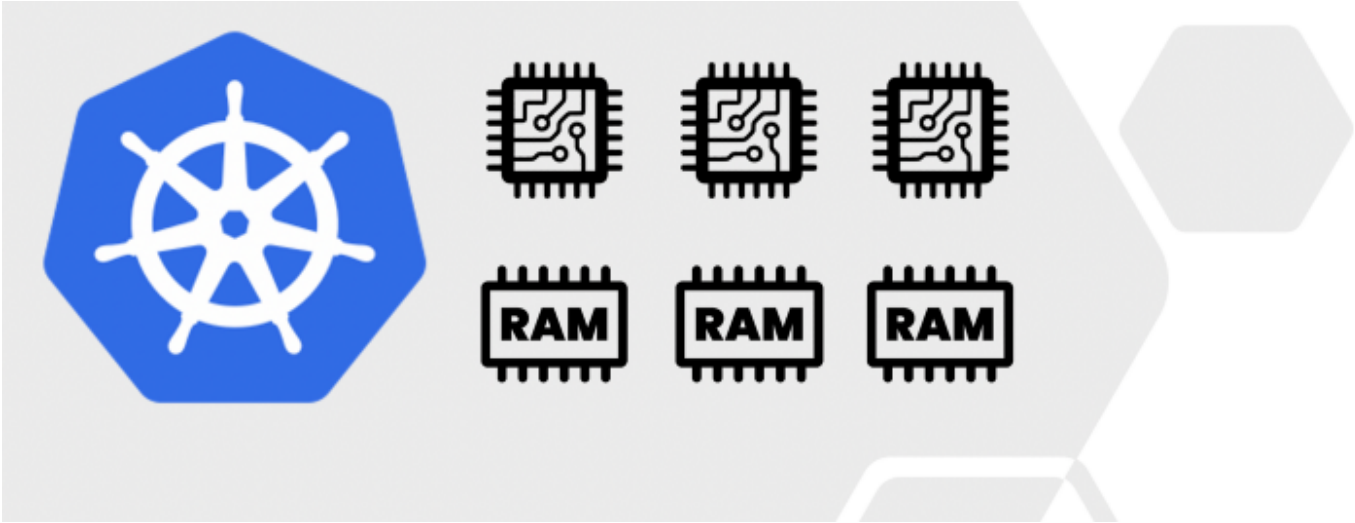
Ultimate Kubernetes Resource Planning Guide

Yitaek Hwang Follow
Jul 19 · 7 min read ★

Understanding allocatable CPU/memory on Kubernetes nodes and optimizing resource usage.



Capacity planning for Kubernetes is a critical step to running production workloads on clusters optimized for performance and cost. Given too little resources, Kubernetes may start to throttle CPU or kill pods with out-of-memory (OOM) error. On the other hand, if the pods demand too much, Kubernetes will struggle to allocate new workloads and waste idle resources.

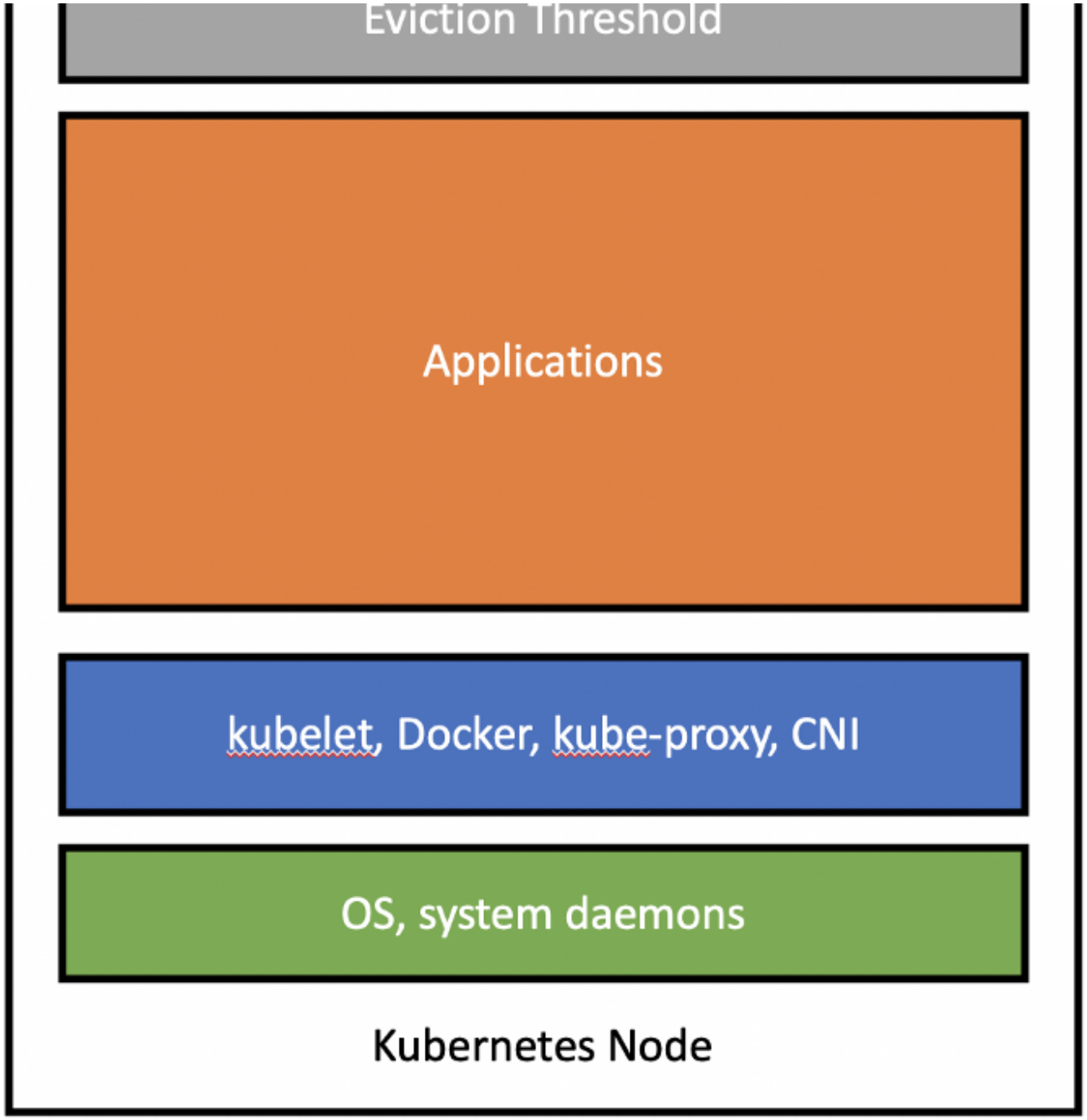
Unfortunately, capacity planning for Kubernetes is not simple. Allocatable resources depend on underlying node type as well as reserved system and Kubernetes components (e.g. OS, kubelet, monitoring agents). Also, the pods require some fine-tuning of resource requests and limits for optimal performance. In this guide, we will review some Kubernetes resource

allocation concepts and optimization strategies to help estimate capacity usage and modify the cluster accordingly.

Allocatable CPU & Memory

One of the first things to understand is that **not all the CPU and memory on the Kubernetes nodes can be used for your application**. The available resources in each node are divided in the following way:

1. Resources reserved for the underlying VM (e.g. operating system, system daemons like sshd, udev)
2. Resources need to run Kubernetes (e.g. kubelet, container runtime, kube-proxy)
3. Resources for other Kubernetes-related add-ons (e.g. monitoring agents, node problem detector, CNI plugins)
4. Resources available for my applications
5. Capacity determined by the eviction threshold to prevent system OOMs



Node Allocation

For self-administered clusters (e.g. kubeadm), each of these resources can be configured via system-reserved, kube-reserved, and eviction-threshold flags. For managed Kubernetes clusters, cloud providers detail node resource allocation per VM type (GKE and AKS explicitly state usage, whereas EKS values are estimated from EKS AMI or EKS bootstrap comments).

Let's take GKE as an example. First, GKE reserves 100 MiB of memory on each node for the eviction threshold.

For CPU, GKE reserves:

- 6% of the first core
- 1% of the next core (up to 2 cores)
- 0.5% of the next 2 cores (up to 4 cores)
- 0.25% of any cores above 4 cores

For memory, GKE reserves:

- 255 MiB of memory for machines with less than 1 GB of memory
- 25% of the first 4GB of memory

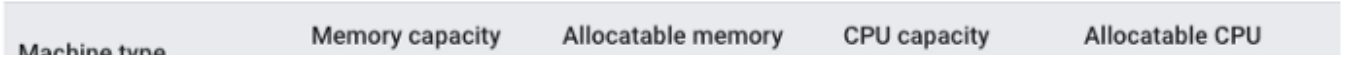
- 20% of the next 4GB of memory (up to 8GB)
- 10% of the next 8GB of memory (up to 16GB)
- 6% of the next 112GB of memory (up to 128GB)
- 2% of any memory above 128GB

Using the general-purpose n1-standard-1 VM type (1 vCPU, 3.75GB memory), we are then left with:

- Allocatable CPU = 1vCPU - (0.06 * 1vCPU) = 0.94 vCPU
- Allocatable memory = 3.75GB - (100MiB - 0.25 * 3.75GB) =2.71GB

Before we run any applications, we can see that we only have ~75% of the underlying node's memory and ~95% of the CPU. On the other hand, bigger nodes are less impacted by the system and Kubernetes overhead. As shown below, an n1-standard-96 node leaves 99% of the CPU and 96% of the memory for your applications.

(*The full list of allocatable memory and CPU resources for each machine type can be found here with caveats for Windows Server nodes*.)



Machine type	(GB)	(GB)	(cores)	(cores)
e2-micro	1	0.62	2	0.94[1]
e2-small	2	1.35	2	0.94[1]
e2-medium	4	2.76	2	0.94[1]
g1-small	1.7	1.2	1	0.94
n1-standard-1 (default)	3.75	2.7	1	0.94
n1-standard-2	7.5	5.7	2	1.93
n1-standard-4	15	12.3	4	3.92
n1-standard-8	30	26.6	8	7.91
n1-standard-16	60	54.7	16	15.89
n1-standard-32	120	111.2	32	31.85
n1-standard-64	240	228.4	64	63.77
n1-standard-96	360	346.4	96	95.69

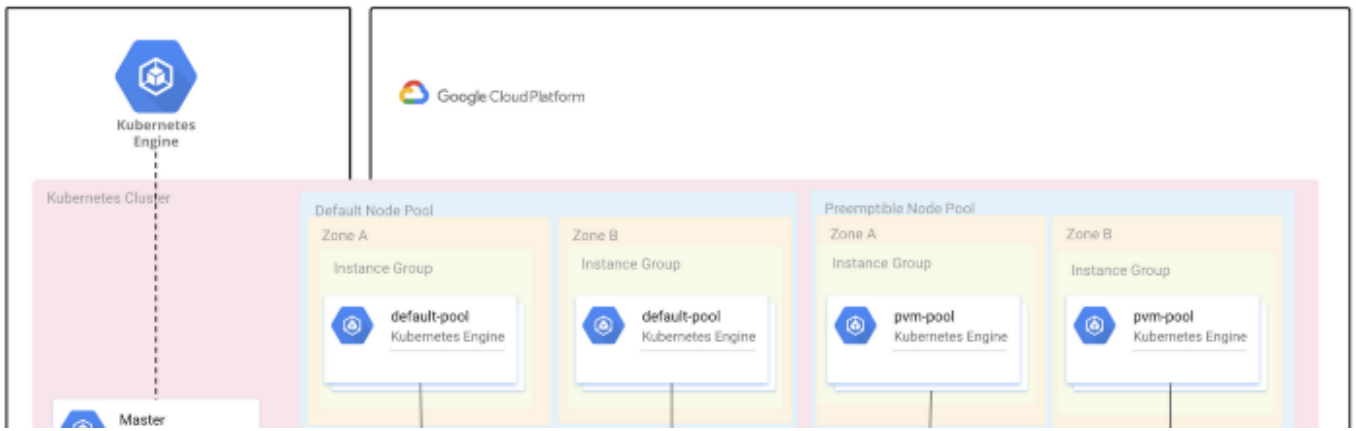
GKE Allocatable CPU & Memory Resources — Image Credit: GKE Documentation

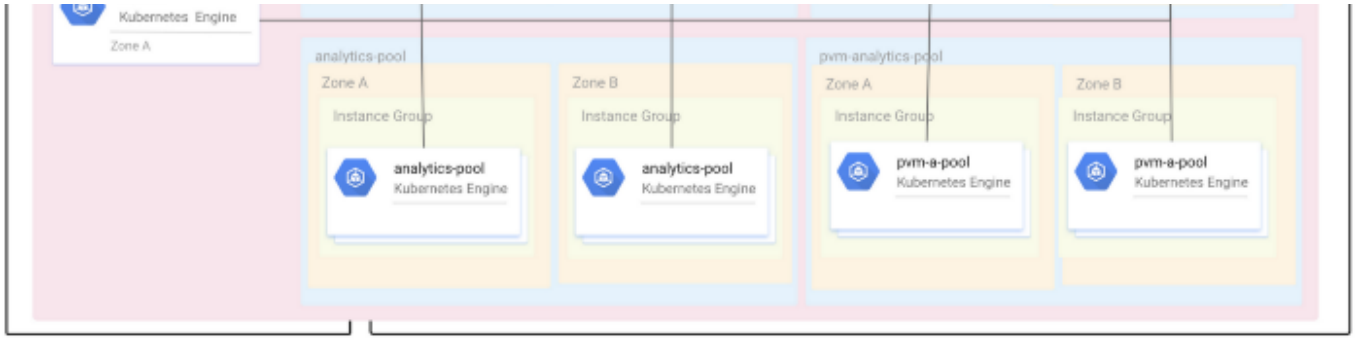
Resource Asymmetry

Now that we understand allocatable resources, the next challenge is dealing with resource asymmetry. Some applications may be CPU-intensive (e.g. machine learning workloads, video streaming), whereas some may be memory-intensive (e.g. Redis). Given such resource asymmetry, kube-scheduler will try its best to schedule each workload to the most optimal

node given the resource constraints. Kube-scheduler's decision to schedule pods on different nodes is guided by a scoring algorithm influenced by resource requirements, anti-/affinity rules, data locality, and inter-workload interference. Although it is possible to tune the scheduler's performance in terms of latency (i.e. time to schedule a new pod) and the node scoring threshold for scheduling decisions, selecting the correct node type is critical to avoid unnecessary scaling and unused resources on each node.

One method to deal with resource asymmetry is to create multiple node pools for different application types. For example, stateless applications may run on general-purpose, preemptible nodes, whereas databases may be scheduled to run on CPU or memory-optimized nodes. This can be controlled by node taints and affinity rules to schedule specific workloads to tainted nodes only.





GKE Example of Multiple Node Pools — Image Credit: GCP Blog

Even with multiple node pools and affinity rules set, Kubernetes resource usage may become suboptimal over time given its dynamic nature:

- New nodes may be added to the cluster to deal with a higher load.
- Nodes may fail or be recreated for cluster upgrades.
- Taints or pod/node affinity rules may change to deal with new application requirements.
- Some nodes may become under- or over-utilized after applications are deleted or added.

To rebalance the pods across the nodes, run descheduler as a Job or CronJob inside the Kubernetes cluster. Descheduler is a kubernetes-sig project that includes seven strategies (`RemoveDuplicates` , `LowNodeUtilization` , `RemovePodsViolatingInterPodAntiAffinity` ,

`RemovePodsViolatingNodeAffinity`, `RemovePodsiolatingNodeTaints`, `RemovePodsHavingTooManyRestarts`, and `PodLifeTime`) to optimize node resource usage automatically.

Resource Ranges & Quotas

Finally, we must understand and define resource ranges for our application. Kubernetes provides two basic configurable parameters for resource management:

- **Requests**: lower bound on resource usage per workload
- **Limits**: upper bound on resource usage per workload