Available online www.ejaet.com

European Journal of Advances in Engineering and Technology, 2021, 8(12):109-114

Research Article ISSN: 2394 - 658X

Kubernetes Resource Planning to improve Application Performance and Stability

Pallavi Priya Patharlagadda

Pallavipriya527.p@gmail.com
United States of America

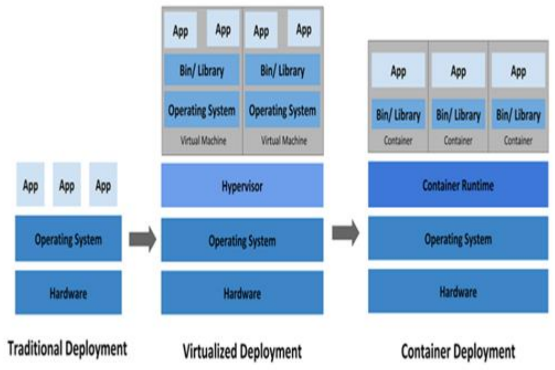
ABSTRACT
Deploying containerized applications on Kubernetes does not absolve us from resource management. Though we can scale our application much more simply as demand grows, our way of thinking may have altered because we now frequently have to take into account the possibility of resource conflicts between our containers. In a Kubernetes cluster, the "noisy neighbor" issue can be resolved with the usage of resource requests and limits. In this paper, we deep dive into the details and the importance of setting up resource limits on the Kubernetes cluster.

Keywords: Kubernetes, resource management, noisy neighbor

INTRODUCTION

It is indisputable that Kubernetes offers performance gains. Let's only focus on the efficacy that Kubernetes offers, which comes with inexpensive infrastructure expenses because of its capacity to tightly schedule containers onto the underlying machines. With the tools available to separate apps from one another, avoiding the possibility that a runaway container might affect the operation of a crucial service. It's not unexpected, given the higher adoption rate, that Kubernetes expenses are rising, as indicated by the Kubernetes FinOps Report (June 2021) survey; however, one would wonder if this also partially reflects the acknowledged difficulties in making Kubernetes energy-efficient. Yet not all in Kubernetes Land is rosy. To be honest, even the most seasoned performance Engineers and SREs find it challenging to maintain application speed, stability, and efficiency on Kubernetes. Many stories about teams facing Kubernetes application performance and stability problems—like unanticipated CPU slowdowns and even abrupt container terminations—can be found on the Kubernetes Failure Stories website, which was established to facilitate the sharing of Kubernetes incident reports and teach best practices.
In the sections that follow, we will describe Kubernetes and the major reasons on how Kubernetes manages resources, which necessitates careful application configuration to ensure cost effectiveness and performance improvement.

WHAT IS KUBERNETES?

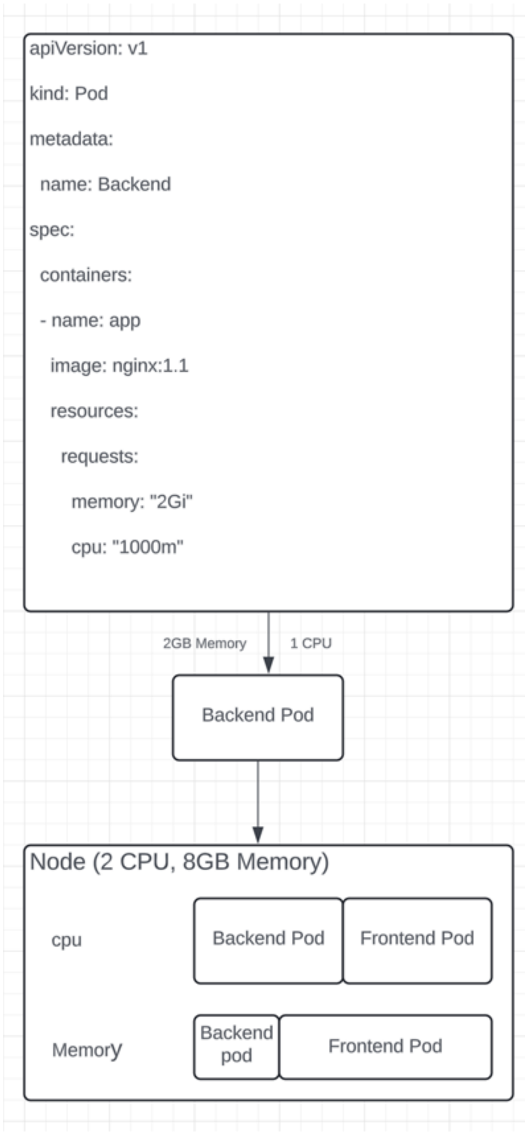


As defined on the kubernetes.io website, "Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.". The term Kubernetes comes from Greek and means helmsman or pilot. K8s is an abbreviation that comes from counting the eight letters between the "K" and the "s". In 2014, Kubernetes was made available to the public by Google. Kubernetes blends Google's 15-year experience running production workloads at scale with community-generated best-of-breed ideas and practices. The below image gives us an understanding of how the application deployment evolved over time.

THE NEED OF KUBERNETES AND ITS CAPABILITIES

Bundling and running your apps in containers is a smart idea. In a production setting, you are responsible for maintaining uptime and managing the containers that execute the apps. For instance, another container needs to start in case the first one fails. If a system oversaw this conduct, wouldn't it make things easier?
That's where Kubernetes comes into play! You may operate distributed systems with resilience by using the framework that Kubernetes offers. It offers deployment patterns, handles scaling and failover for your application, and much more. For instance: Kubernetes can effortlessly oversee a system's canary deployment.

KUBERNETES OFFERINGS



```
apiVersion: v1
kind: Pod
metadata:
  name: Backend
spec:
  containers:
  - name: app
    image: nginx:1.1
    resources:
      requests:
        memory: "2Gi"
        cpu: "1000m"
```

Kubernetes offers a wide range of services. Below are some of the widely used offerings

● **Service discovery and load balancing:** Kubernetes can expose a container using its DNS name or its own IP address. To ensure a stable deployment, Kubernetes may load balance and distribute network traffic to a container in cases of significant traffic.

● **Storage orchestration**: With Kubernetes, you can set up any storage system to be mounted automatically, including local drives, public cloud storage, and more.

● **Automated rollouts and rollbacks:** With Kubernetes, you can define the ideal state for your deployed containers, and it will gradually transform the current state to the desired state. To build new containers for your deployment, for instance, you can automate Kubernetes to delete old containers and transfer all of their resources to the new container.
● **Automatic bin packing**: For the purpose of executing containerized jobs, you give Kubernetes a cluster of nodes. You specifyinspecify in Bytes are the units of measurement for memory. However, memory can be expressed using a variety of suffixes (E, P, T, G, M, K, and Ei, Pi, Ti, Gi, Mi, Ki) ranging from mebibytes (Mi) to petabytes. Most people simply utilize Mi Pods, like CPUs, will never be scheduled if their resource requirements exceed a node's capacity. Memory, unlike CPUs, is not compressible. Memory cannot be slowed or accelerated in the same way that CPUs or networks can. If a pod hits its memory limit, it will be terminated. The picture below depicts the same. The yml file specifies the memory requests of "2Gi" and 1 CPU. If the pod is deployed on a node with 2 CPUs and 8GB of RAM, then the backend pod would take half of the CPU and 25% of RAM.

Now, let's look at an example with multiple containers.
Since Kubernetes has no defaults, we need to define the resources in YAML format. A configuration may look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: Backend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "512Mi"
        cpu: "1000m"
  name: simpleApp
    image: sampleImg
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "256Mi"
        cpu: "500m"
```

Using a MySQL database and the sample image, this is an example of a sample application. Provisioned with 64 Mi (megabyte) of RAM AND 250 millicores (one-fourth of a core) of CPU power are the two sampleApp and database containers. The request block contains definitions for these parameters.
Regarding limitations, the sampleApp image has a 500m CPU limit and memory set to 256Mi. The database has a full core with a 1000m setting and doubles the limit at 512Mi. As database applications often use more resources, we can test using a larger estimate.
You can raise the restrictions to 1 GB of memory and 2 CPU cores if you believe the database needs additional resources.
We can determine whether to raise or decrease the resources based on testing. Adjust the settings based on your needs. Certain workloads, like Node.JS and React, will use a lot less resources than legacy Java monolith apps, which could need a lot more RAM.
Here are a few key points to keep in mind: The request cannot be less than the limit at any point. If you try to do this, Kubernetes will error out. Kubernetes will never schedule a container whose request exceeds the capacity of a node. Your program won't ever be deployed, for instance, if it is designed to employ 3.5 cores on a 2-core node.
Namespaces for Kubernetes:
In addition to the resources of each individual container, you might want to look at limiting namespaces. But first, what is a namespace? A cluster of apps, departments, or environments can be defined using namespaces. A Kubernetes cluster's Namespace is essentially the scope or grouping of its items. A development namespace can be defined, and it has tougher limitations than a production namespace. Consequently, this exemplifies the use of

namespaces to denote deployment environments. A few organizations divide up teams or groups into various namespaces. In an ideal world, smaller deployments could function flawlessly with the aforementioned container resource settings, but in larger, multi-silo businesses with various teams, certain apps might use more cluster resources than they should. You can use any namespace that you want it to be.
You can configure Limit Ranges and Resource Quotas at the Namespace level.

RESOURCES QUOTAS

Limit Ranges apply to specific pods inside a resource, whereas Resource Quotas are the maximum limits for all pods within a namespace. Resource Quotas is the tool for managing resource quotas, as the name implies. It gives administrators the option to specify and enforce the total number of resources within a namespace.
Here is an illustration of a resource quote. There is a 3Gi memory restriction for this namespace.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: app
spec:
  hard:
    cpu: "10"
    memory: 3Gi
    pods: "5"
    replicationcontrollers: "10"
    resourcequotas: "1"
    services: "2"
```

Limit Ranges:
Limit Ranges impact specific pods and containers as opposed to Resource Quotas, which span the entire namespace.
A sample configuration can be found below. Limit ranges will stop users from over-provisioning tiny or extremely broad applications into a node inside a namespace.

```
limits:
  - default:
      memory: 1Gi
    defaultRequest:
      memory: 1Gi
    max:
      memory: 1Gi
    min:
      memory: 500Mi
    type: Container
```

There are 4 parts: default, default Request, max, and min.

SECTION	APPLIES TO
Default	Default Limits
Default Request	Default Requests
Max	Max Limits
Min	Min Requests

If a pod doesn't have a resource definition, the namespace defaults will apply. A pod will inherit the provided values if it does not specifically declare a particular resource. For instance, it will inherit the default limitations in the absence of a default value.

When you have settings that are contradictory or competitive, things get complicated. A few instances are: If there is no default option or if the containers do not establish limitations explicitly, namespace max values take precedence. It shall assume the minimum value in the absence of a default Request assignment. Additionally, neither the pod's individual request settings nor the default Request setting may be lower than the minimum.
If you try to apply conflicting settings, Kubernetes will throw an exception. Simultaneously, Kubernetes is intelligent enough to detect omissions and reassign values according to the previously stated guidelines.

Benefits of the Kubernetes Limit Range

The main purpose of limit ranges is to reduce the number of apps that might use more resources than they should. The advantages extend beyond limiting the number of resources used by applications.

Multiple workloads, including development and production, can be supported by a cluster. While development may only require 1 GB of RAM, a production workload may require 16 GB. Strict quotas are required by best practices for lower-tier settings, such as staging and QA. It is best to lock down development environments.

A further benefit is homogeneity in terms of deployment. Suppose one program uses up 80% of the resources, leaving too little for other applications to use. Thus, in order to offer consistent scheduling, administrators might need a minimum of 25% of RAM and CPU.

COMMON CHALLENGES

Although resource allocation and restriction are possible (by object type, count, or namespaces), these configurations may still result in overcommitments and slack, as K8s administrators refer to wasted spending.

For instance, the application would be Out Of Memory (OOM) and crash in the event that not enough memory resources were allocated. Inadequate planning may result in a production application outage while provisioning a new pod for testing.

The use of pod priority and preemption is one approach to solving this issue. Pod priority indicates a pod's significance in relation to other pods. Preempting (evicting) a lower-priority pod is the scheduler's attempt to schedule a higher-priority pod when it cannot.

```
apiVersion: v1
 kind: PriorityClass
 metadata:
   name: critical
 value: 9999999
 globalDefault: false
 description: "Use this class for critical service pods only."
```

By changing the preemption Policy to "Never," you can also prevent preemption of a high Priority Class.

```
apiVersion: v1
 kind: PriorityClass
 metadata:
   name: high-priority
 value: 7777777
 globalDefault: false
 preemptionPolicy: Never
 description: "Use this class for critical service pods only."
```

Pod priorities are associated with a pod using the priorityCassName field.

```
apiVersion: v1
 kind: Pod
 metadata:
   name: a-pod
 spec:
   containers:
   name: a1-container
image: nginx
     imagePullPolicy: IfNotPresent
     resources:
        limits:
          memory: “1Gi”
          cpu: “500Mi”
 name: a2-container
     image: nginx
     imagePullPolicy: IfNotPresent
     resources:
        limits:
          memory: “1Gi”
          cpu: “500Mi”
   priorityClassName: critical
```

You can schedule your most important tasks with confidence and not worry about over-provisioning your clusters thanks to pod priority and preemption.

QOS CLASSES

When Kubernetes creates a pod, it assigns one of these three QoS classes:

- Guaranteed
  These need stringent regulation. The limits and requests are the same for both CPU and RAM. To put it simply, requests and limits have the same value. They operate until they surpass the limitations and are regarded as high priorities.
- Burstable
  These classes fall into two categories. They either fall short of the guaranteed QoS requirements or atleast one memory or CPU request is present.
- BestEffort
  This is a container that has no requests, memory limits, or CPU restrictions. If you haven't defined any resources, your pod is operating in BestEffort by default.

QOS BEST PRACTICES

BestEffort is not recommended for workloads related to production. Remember that these are the containers that will be killed first. For most general workloads, burstable works well. Applying a QoS assured class is advised for sensitive applications that could experience spikes or anything that operates in a stateful set, such as databases.

CONCLUSION

Microservice-based applications work well on Kubernetes, but applications must be properly designed to guarantee good performance and low costs. Resource scheduling is becoming more and more important as container clusters get bigger and more complicated. Container limits can be used to monitor application resources. Namespace settings allow distinct groups and classes to have different limitations when there are additional teams working on different projects in a cluster. One namespace's resources are not visible to other namespaces. Additionally, these namespaces can be used in various tier environments across the whole CI/CD pipeline, from workflows in staging to production. In addition, mission-critical operations may need a higher degree of uptime for important applications like databases. Those apps are given precedence through careful resource allocation and QoS classes.
As we've covered, even the most seasoned performance specialists find it difficult to tune these apps because of the intricacy of Kubernetes' resource management. The intended application performance, stability, and cost-efficiency cannot always be guaranteed by traditional methods that mostly rely on human tuning.

REFERENCES

[1]. https://www.akamas.io/resources/kubernetes-optimization-costs-slo/
[2]. https://kubernetes.io/docs/concepts/overview/
[3]. https://www.densify.com/kubernetes-tools/kubernetes-resource-limits/
[4]. https://goteleport.com/blog/kubernetes-resource-planning/