Research Document

ROOSH CHALLENGE 5

Date		**:**	**04/12/2023**
Version	**:**	**1.0**	
Status		**:**	**Open**
Author		**:**	**Group 1**

CONTENTS

RESEARCH

KUBERNETES - CLUSTER AUTOSCALING (CA)

Scalability is one of the core value propositions of Kubernetes. Alongside Vertical Pod Autoscaler (VPA) and Horizontal Pod Autoscaler (HPA), Cluster Autoscaler (CA) is one of the three autoscaling functionalities in Kubernetes. Therefore, understanding Cluster Autoscaler is an integral part of getting the most out of your Kubernetes platform.

CLUSTER AUTOSCALER VS OTHER TYPES OF AUTOSCALERS

Before we are going more into the details of CA, let's review the different types of autoscaling in Kubernetes.

1. Cluster Autoscaler (CA): Adjusts the number of nodes in the cluster when pods fail to schedule or when nodes are underutilized.

2. Horizontal Pod Autoscaler (HPA): Adjusts the number of replicas of an application.

3. Vertical Pod Autoscaler: adjusts the resource requests and limits of a container

A simple way to remember those Kubernetes autoscaling functionality is that HPA and VPA operate on the pod level whereas CA works at the cluster level.

WHAT IS CA?

The Cluster Autoscaler atomically adds or removes nodes in a cluster based on the resource requests from the pods. The cluster autoscaler doesn't directly measure CPU and memory usage values. It checks every 10 seconds to detect any pods in a pending state, suggesting that the scheduler could not assign them to a node due to insufficient cluster capacity.

HOW DOES CA WORK?

In the scaling up scenario, CA atomically kicks in when the number of pending (un-schedulable) pods increases duo to resource shortages and adds additional nodes to the cluster.

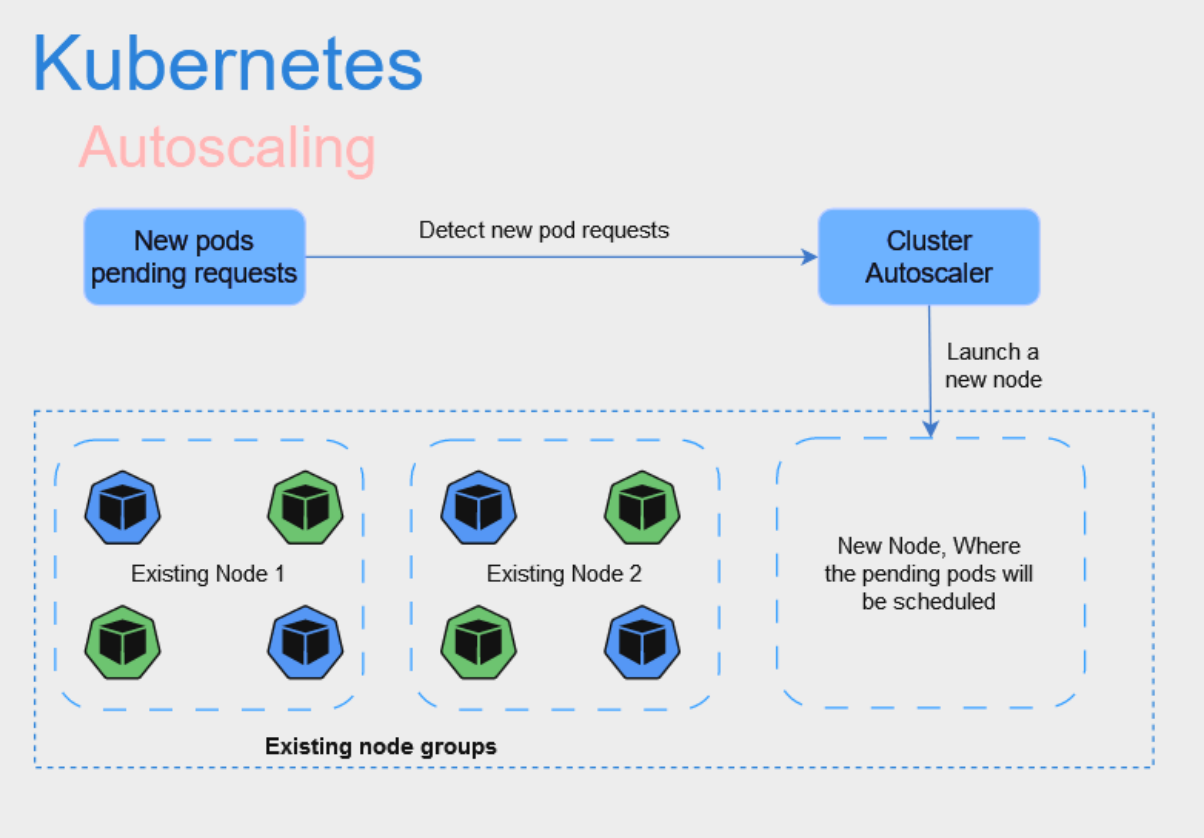

Figure 1, Cluster autoscaling.

The diagram above illustrates the Cluster Autoscaler decision-making process when there is a need to increase capacity. It uses a similar mechanism in a scale-down scenario where CA may consolidate pods onto fewer nodes to free up a node and terminate it.

The four steps for scaling up in CA are as follow:

1. When Cluster Autoscaler is active, it will check for pending pods. The default interval is 10 seconds, but this can be adjusted in the configuration.

2. When there are any pending pods and the cluster needs more resources, CA will extend the cluster by launching a new node as long as it is within the limits configured.

3. Kubernetes registers the newly provisioned node with the central plane to make it available to the Kubernetes scheduler that assigns pods.

4. Kubernetes scheduler allocates the pending pods to the new node.

LIMITATIONS OF CA

Cluster Autoscaler has a couple of limitations worth keeping in mind when planning an CA implementation:

1. CA does not make scaling decisions based on CPU or memory usage. It only checks a pod's requests and limits for CPU and memory resources. This limitation means that the unused computing resources requested by users will not be detected by CA, resulting in a cluster with waste of resources and low utilization efficiency.

2. Whenever there is a request to scale up the cluster, CA issues a scale-up request to the provider within 30-60 seconds. The actual time the provider takes to create a node can be several minutes. This means that there is a delay when scaling-up your node, this may impact the performance of your application for a while for the cluster to extend its capacity.

WHAT IS HPA?

Horizontal Pod Autoscaling, HPA is a form of autoscaling that increases and/or decreases the number of pods in a replication controller, deployment, replica set, or stateful set based on various metrics such as CPU utilization. The scaling is horizontal because it affects the number of instances, and pods rather than the resources allocated to a single container.

HPA can make scaling decisions based on custom or externally provided metrics and works automatically after the initial configuration. All you need is to define the minimal and maximum number of replicas.

After you have configured the Horizontal Pod Autoscaler Controller is in charge of checking the metrics and then scaling your replicas up or down accordingly. By default, HPA checks the metrics every 15 seconds.

To check metrics, HPA depends on another Kubernetes resource known as the Metrics Server. The metrics server provides standard resource usage measurement data by capturing data from the "Kubernetes.summary_api" such as the CPU and memory usage for nodes and pods. It can also provide access to custom metrics, that can be collected from an external source like the number of active sessions on a load balancer to indicate traffic volume.

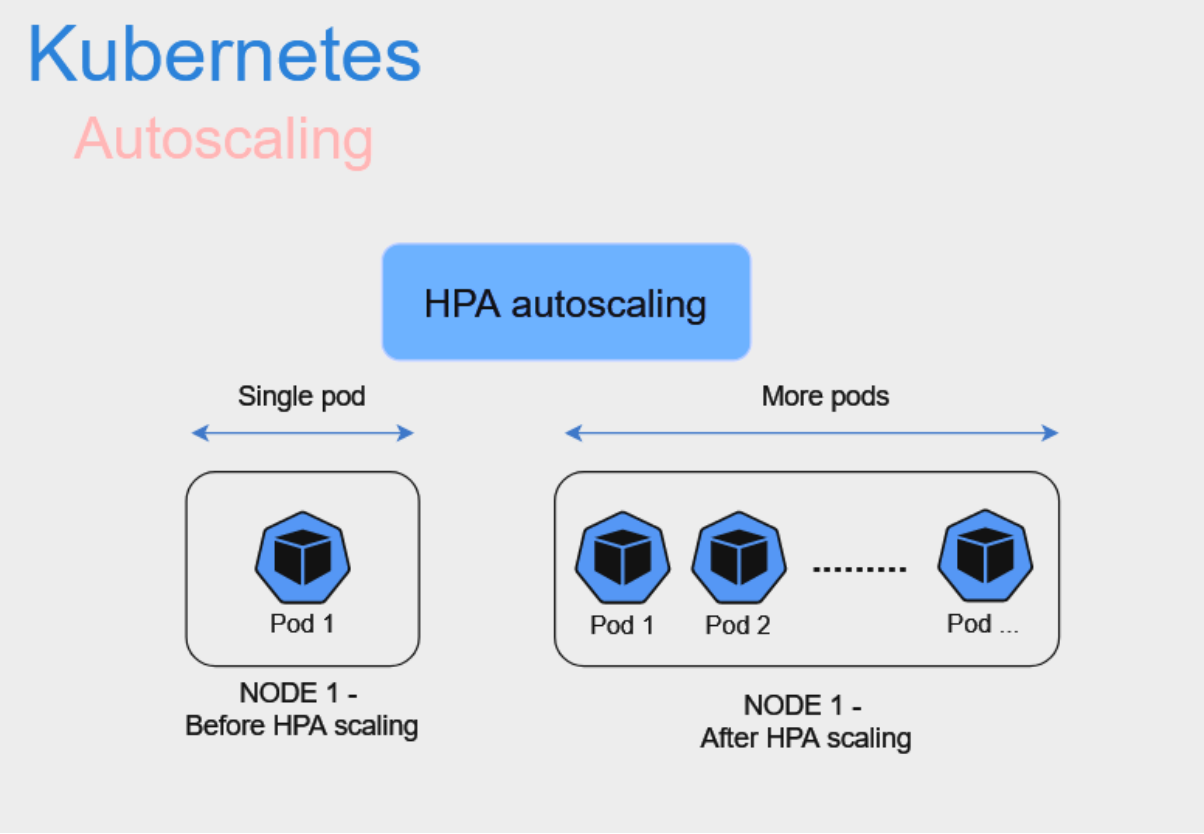

Figure 2, HPA.

While the HPA autoscaling process is automatic, you can also help account for predictable load fluctuant in some cases. A few examples:

- Adjust the replica count based on the time of day.
- Set different capacity requirements for weekends or off-peak hours.
- Implement an event-based replica capacity schedule.

HOW DOES HPA WORK?

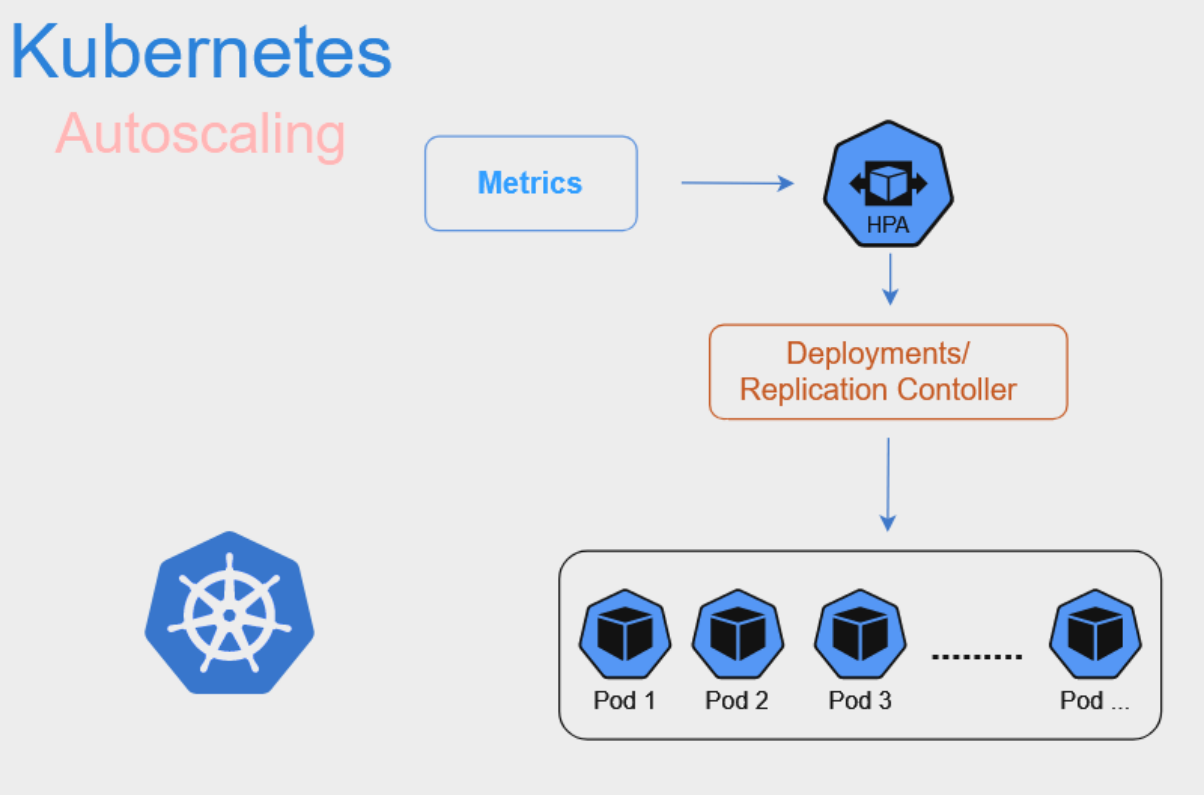

Figure 3, Horizontal pod autoscaling.

In simple words, HPA works in a "check, update, check" style loop. Here is how each of the steps in that loop works.

1. HPA continuously monitors the metrics server for resource usage.

2. Based on the collected resource usage, HPA will calculate the desired number of replicas required.

3. Then, HPA decides to scale up the application to the desired number of replicas.

4. Finally, HPA changes the desired number of replicas.

5. Since HPA is continuously monitored the process repeats from step 1.

LIMITATIONS OF HPA

While HPA is a powerful solution, it's not ideal for every use case and can't address every cluster resource issue. Here are a few examples:

- One of HPA's most well-known limitations is that it does not work with DaemonSets
- If you don't efficiently set CPU and memory limits on pods, your pods may terminate frequently or on the other end of the spectrum, waste your resources.
- If the cluster is out of capacity, HPA can't scale up until new nodes are added to the cluster. Cluster Autoscaler (CA) can automate this process.

Cluster Autoscaler (CA) atomically adds or removes nodes in a cluster based on resource requests from pods. Unlike HPA, cluster autoscaling doesn't look at memory or CPU available when it triggers the autoscaling. Instead, it reacts to events and checks for any unscheduled pods every 10 seconds.

KUBERNETES - VERTICAL POD AUTOSCALING (VPA)

WHAT IS VPA?

Kubernetes Vertical Pod Autoscaler (VPA) is a component you install in your cluster. It increases and decreases container CPU and memory resource configuration to align cluster resource allotment with actual usage.

KUBERNETES VPA RESOURCE CONFIGURATION TYPES

With VPA there are two different types of resource configurations that can be managed on each container of a pod:

- Requests
  - Requests define the minimum number of resources that a container needs. For example, an application can use more than 256MB of memory, but Kubernetes will guarantee a minimum of 256MB to the container if its request is 256 of Memory.
- Limits
  - Limits define the maximum number of resources that a given container can consume. The application might require at least 256MB of memory, but you might want to ensure that it doesn't consume more than 512MB of memory, to limit its memory consumption to 512MB.

KUBERNETES VPA VS HPA

Fundamentally, the difference between VPA and HPA lies in how they scale. HPA scales by adding or removing pods, using this solution it scales capacity horizontally. VPA, scales by increasing or decreasing CPU and memory resources within the existing pod containers, using this solution it scales the capacity vertically. The table below explains the differences between Kubernetes VPA and HPA.

VERTICAL SCALING (VPA)	VERTICAL SCALING (VPA)	VERTICAL SCALING (VPA)
More resources	Add more pods	Increase CPU or memory resources of existing pod containers
Less resources	Remove pods	Decrease CPU or memory resources of existing pod containers

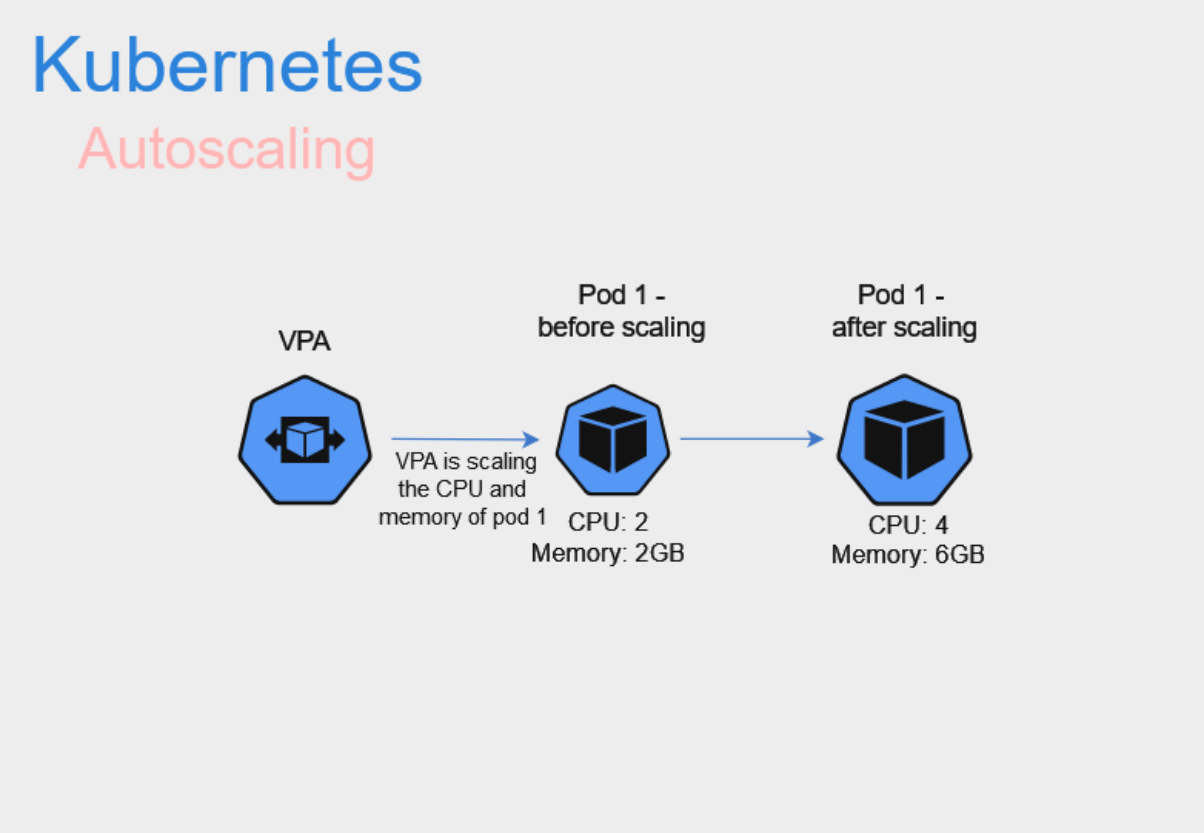

Figure 4, VPA.

COMPONENTS OF VPA

A VPA deployment has three main components: VPA recommender, VPA updater, and VPA admission controller.

THE VPA RECOMMENDER

- Monitors resource utilization and computes target values.
- Looks at the metric history, events and the VPA deployment spec and suggests fair requests. The limits are raised/lowered based on the _limits-requests_ proportion defined.

THE VPA UPDATER

- Evicts those pods that need the new resource limits.
- Implements whatever the Recommender recommends if “updateMode: Auto“is defined.

THE VPA ADMISSION CONTROLLER

- Changes the CPU and memory settings before a new pod starts whenever the VPA Updater evicts and restarts a pod.
- Evicts a pod if it needs to change the pod's resource requests when the Vertical Pod Autoscaler is set with an updateMode of "Auto.” Due to the design of Kubernetes, the only way to modify the resource requests of a running pod is to recreate the pod.

HOW DOES VPA WORK?

The diagram below provides a practical example of how Kubernetes VPA works and is followed by a numbered explanation of each step.

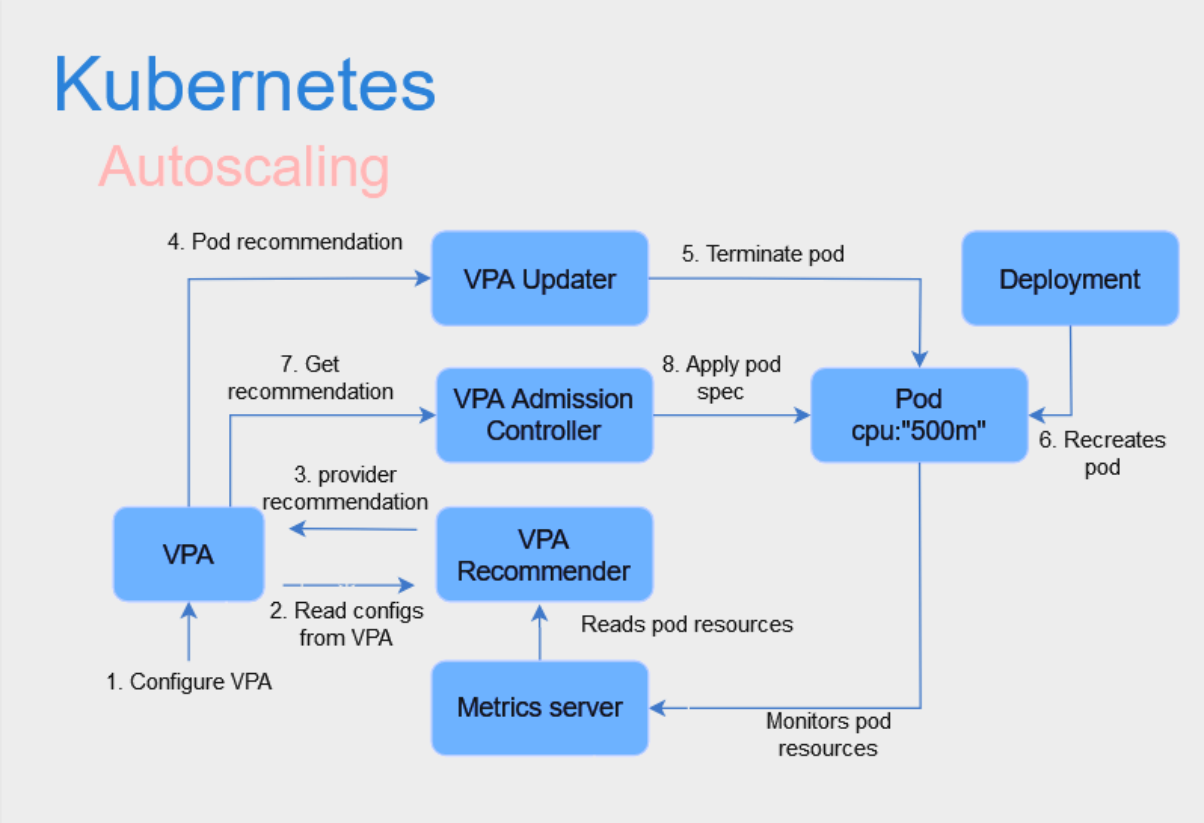

Figure 5, VPA Workflow.

1. The user configures VPA.

2. VPA Recommender reads the VPA configuration and the resource utilization metrics from the metric server.

3. VPA Recommender provides pod resource recommendations.

4. VPA Updater reads the pod resource recommendations.

5. VPA Updater initiates the pod termination.

6. The deployment realizes the pod was terminated and will recreate the pod to match its replica configuration.

7. When the pod is in the recreation process, the VPA Admission Controller gets the pod resource recommendation. Since Kubernetes does not support dynamically changing the resource limits of a running pod, VPA cannot update existing pods with new limits. It terminates pods that are using outdated limits. When the pod's controller requests the replacement from the Kubernetes API service, the VPA Admission Controller injects the updated resource request and limit values into the new pod's specification.

8. Finally, the VPA Admission Controller overwrites the recommendations to the pod. In our example, the VPA admission controller adds a "250m" CPU to the pod.

LIMITATIONS OF VPA

VPA is useful in many applications, but there are several important limitations to keep in mind.

- Do not use Vertical Pod Autoscaler with the Horizontal Pod Autoscaler, which scales based on the same resource metrics such as CPU and MEMORY usage. This is because when a metric (CPU/MEMORY) reaches its defined threshold, the scaling event will happen for both VPA and HPA at the same time, which may have unknown side effects and may lead to issues.
- VPA might recommend more resources than available in the cluster, thus causing the pod to not be assigned to a node (due to insufficient resources) and therefore never run. To overcome this limitation, it's a good idea to set the LimitRange to the maximum available resources. This will ensure that pods do not ask for more resources than the LimitRange defines.

KUBERNETES - MULTI-DIMENSIONAL POD AUTOSCALING (MPA)

WHAT IS MPA?

Multi-dimensional Pod Autoscaling (MPA) is a framework designed to combine horizontal and vertical autoscaling actions in a single operation. It addresses the challenge of independently managed Horizontal Pod Autoscaler (HPA) and Vertical Pod Autoscaler (VPA) controllers, ensuring synchronized and efficient scaling for containerized applications in Kubernetes.

HOW DOES MPA WORK?

MPA consists of three controllers (recommender, updater, and admission controller) and an MPA API, which connects autoscaling recommendations to actuation. The framework separates decision actuation from recommendations, allowing users to replace the default recommender with their customized algorithms. MPA integrates with existing HPA and VPA libraries to reuse their functionalities while providing a unified approach to multi-dimensional scaling.

PROS AND CONS OF MPA

Pros:

- Vertical scaling is handled by webhooks to avoid overloading etc.
- Horizontal scaling is handled through deployment to avoid extra overhead by webhooks.
- Authentication and authorization for vertical scaling are handled by admission webhooks.
- Recommendation and the actuation are completely separated.

Cons:

- Webhooks introduce extra overhead for vertical scaling operations (can be avoided after in-place resizing of pod is enabled without eviction)
- Vertical and horizontal scaling executions are separated (can be avoided after in-place resizing of pod is enabled without eviction)
- State changes in pod sizes are not persisted (too much to keep in etcd, could use Prometheus to store pod state changes)