Autoscaling In Kubernetes

Muhammet Tayyip Yazıcı, Hüseyin Can Bölükbaş

Introduction

Kubernetes is a system for automating the deployment, management and scaling of containerized applications[1]. It provides native and third party tools for scaling applications. Horizontal Pod Autoscaling (HPA) is supported by the controller manager[2] and is present in every Kubernetes cluster. The Autoscaler project[3] allows users to scale their application with Vertical Pod Autoscaling (VPA) and Cluster Autoscaling (CA). They can be added to clusters manually or automatically by using some Kubernetes platform providers. We wrote a sample application that exposes a CPU intensive endpoint and examined its performance with no auto scaling, *HPA*, *VPA* and *CA*. In this document we assume the reader is familiar with the general concepts of Kubernetes pods, horizontal and vertical scaling.

Code and all configuration files for this project can be found in:
https://github.com/canbolukbas/scaling-in-kubernetes

Architecture Design

We chose Google Kubernetes Engine (GKE) as our Kubernetes platform provider. It has options for automatically deploying the controllers for *VPA* and *CA*. However, because we were using free tier we had quota restrictions on the resources we could use. We were able to create at most 4 nodes with 2 vCPU each. This limited our tests regarding *CA*, incidentally it also required us to reduce the default number of nodes to 2 since comparing the performance improvements from 3 nodes to 4 nodes wouldn't be very significant.

We created the Kubernetes clusters in the "europe-west-1b" zone. Our initial tests were conducted in the "us-east-1a" zone but we saw that the roundtrip time was too much, so we changed the zone to europe. Below is the architecture diagram.

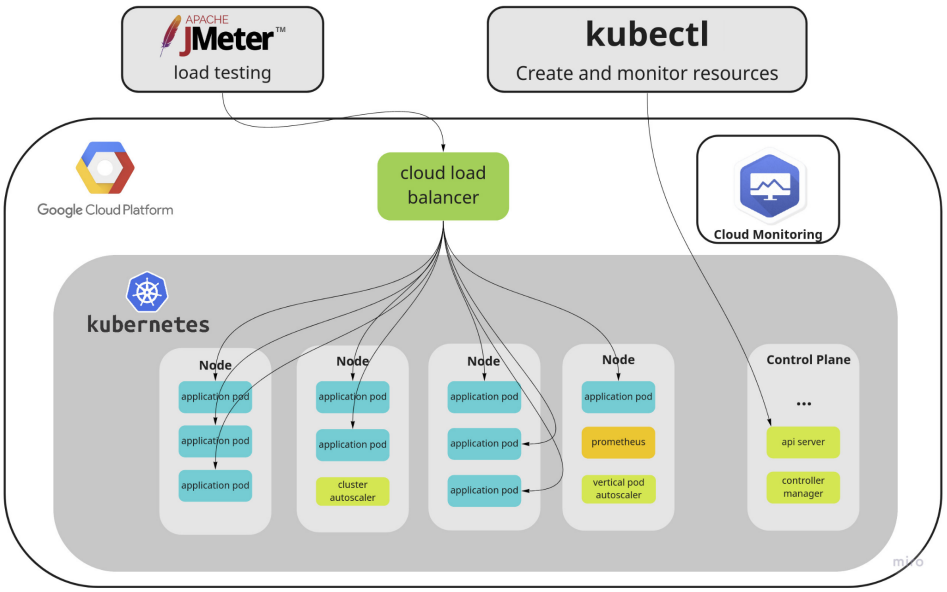

Figure 1 - Architecture Diagram

Services and Software

We developed a very basic API, written with Go. It has three endpoints which are:

- '/cpu' which starts a CPU-intensive task.
- '/memory' which starts a memory-intensive task.
- '/metrics' which provides custom metrics for Prometheus to scrape.

As our CPU-intensive task, we've looped over numbers starting from 0 up to 10^4. We omitted using the memory-intensive endpoint in our tests when we came to realize that it required more CPU resources than memory resources and we could not find a task in which memory would be the limiting factor.

We followed the pods' resource usage through both "kubectl top pod/node" commands and the Google Kubernetes Engine dashboards. However, we only used the data we gathered from the "kubectl" commands when the output was stabilized.

We used Prometheus[4] as the metrics collector to enable *HPA* with custom metrics. We deployed the Prometheus with the Prometheus Operator[5]. We also deployed a Prometheus Adapter[6] for enabling Kubernetes to collect metrics from Prometheus.

Performance Evaluation

Testing Tools

We've used JMeter as our load testing tool. JMeter provides load testing with several options such as concurrency (number of user threads), ramp up time (allowed time in seconds for creating user threads) and loop count (iterations). We've used 2000 as our concurrency, 1 second as our ramp up time and 300 as our loop count in all of our tests. This resulted in 600.000 requests to be sent for each scenario. We also added a 20 second connection timeout and a 30 second response timeout for HTTP requests.

Scenarios

We tested our application with 3 main scenarios: with no autoscaling, with *HPA* enabled and with *VPA* enabled. Both HPA and VPA have additional scenarios.

- *VPA* is tested with *CA* on and off.
- *HPA* is tested with two metric types: CPU usage percentage and average number of requests per second per pod. Both these scenarios are also tested with *CA* on and off. CPU percentage is also tested with CPU percentage limits at 50% and 75%.

Below is a diagram representing all the scenarios tested.

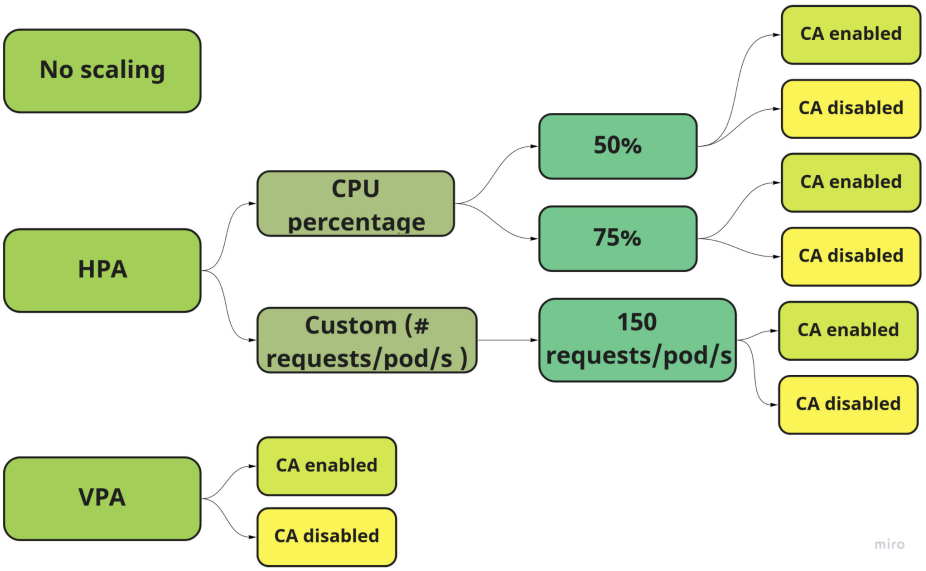

Figure 2 - Testing Scenarios

All scenarios started with exactly 3 pods that each guaranteed to use 100 millicores (10% of a vCPU) and limited to use at most 250 millicores.

In the next sections we will briefly explain how we conducted the scenarios and discuss the results.

HPA

As mentioned before, *HPA* is present in all Kubernetes clusters, it only needs the metric server to be deployed which is the case in default GKE clusters. *HPA* can scale applications using different metrics: CPU usage, memory usage and custom metrics. *HPA* collects the metrics about these resources for a span of time and calculates the average usage. If the average usage is greater than the given limit, it autoscales. Our initial thought was to test all of the cases. However, we were not able to find a task in which memory would be the limiting resource, so we dropped testing it.

For the custom metric we decided on using the number of requests coming per pod per second. Upon a few tries we decided to give the limit value as 150 requests per pod per second. The *HPA* controller cannot collect custom metrics from the metric server. For it to work properly we needed to deploy Prometheus and Prometheus adapter.

For testing *HPA* with CPU usage we decided to test two limit values to see what difference they would make, we set the limits as 50% and 75% CPU utilization.

Following are the results we gathered:

Requests	Pod information				
Label	**# running pods at end**	**# pending pods**	**avg cpu usage (millicores)**	**# nodes at end**	**total cpu usage**
No Autoscaling	3	0	248m	2	744m
HPA with CPU (50%)	7	24	210m	2	1470m
HPA with CPU (75%)	7	13	212m	2	1484m
HPA with CPU & CA (50%)	21	11	68m	4	1428m
HPA with CPU & CA (75%)	21	0	76m	4	1596m

Figure 3 - HPA with CPU test results on pod resource usages

The most apparent thing we see from *Figure 3* is that no matter the pod count the total CPU usage is quite close for all cases where auto scaling is present. This shows that our tests were not able to create enough load on the application for *CA* to make a difference in CPU usage. The difference between CPU limits 50% and 75% also is not significant. Only thing we see is the number of running and pending pods. The scenarios with 50% CPU limit seems to create more pods with no additional total CPU usage benefit.

Requests	Pod information						
Label	**# running pods at end**	**# pending pods**	**avg cpu usage**	**# nodes at end**	**requests/pod/s**	**total cpu usage**	**throughput**
No Autoscaling	3	0	248m	2	412	744m	1236
HPA with custom (150 requests/pod/s)	5	6	248m	2	276	1240m	1380
HPA with custom & CA (150 requests/pod/s)	12	0	130m	4	129	1560m	1548

Figure 4 - HPA with custom (requests/pod/s) test results on pod resource usages

Looking at *Figure 4,* we see that 12 pods seems to be enough for handling the limit of 150 requests per pod per second. Also there is a slight increase in throughput when we increase the auto scaling options. Comparing *Figure 3* and *Figure 4*, we do not quite see a difference in CPU usage between *HPA with CPU* and *HPA with custom*. However, it is not enough just to compare the CPU usages to decide which metric is better to use in auto scaling. Custom metrics can also have the advantage of adding business logic into scaling options which could be more valuable for the businesses.

We also see a difference in the number of running pods in *HPA with CPU* and *HPA with custom* scenarios. At first this seems odd since these applications run in the Kubernetes clusters with the same capabilities and resources. However, for custom metric collection to work we deployed Prometheus, Prometheus adapter and Prometheus operator. Since we have only 2 nodes, these deployments make a difference when we load test the application.

VPA

VPA is not supported by default. As mentioned before, the Autoscaler project maintains the *VPA* controller. However, GKE allows us to create the Kubernetes cluster with *VPA* enabled by deploying necessary resources by itself. Below are the results we gathered.

Requests	Pod information						
Label	# pods at end	# pending pods	% of scaling pod	avg cpu usage	new cpu requests	new cpu limit	# nodes at end
VPA	3	0	66%	453m	295m	885m	2
VPA & CA	3	0	100%	762m	805m	2415m	3

Figure 5 - VPA test results on pod resource usages

From *Figure 5* we see that for the *VPA & CA* scenario all pods were scaled up, this is possible because there is also an extra node created. However, for the *VPA* scenario we see that only 2 pods are scaled and the new CPU request is at 295m. Finally, comparing the average CPU usages we would expect the *VPA & CA* scenario to have a smaller response time since it is able to use more resources along with an extra node.

One other thing we want to mention is, because of our resource and quota limitations we chose 3 pods as the default starting point for all the scenarios. However, for *VPA* tests it would be more beneficial to start with extra pods so that the vertical scaling could be more effective with multiple pods spanning across multiple nodes.

Comparisons

The following comparisons include tables and graphs which are only part of the data we have collected. We include the results that we think worth sharing. For example, we did not take memory into account because memory was never a bottleneck, whereas CPU was.

Comparison #1 - HPA, CA and VPA

This comparison includes all the scenarios except *HPA with CPU percentage 75%* which will be inspected later.

To start with, looking at *Figure 6* we see that there is no significant difference in failures between *HPA* and *No Autoscaling* scenarios. However, there is a significant increase in failures with the inclusion of *VPA*. We suspect this is related to the down time of the server while scaling up. In other words, when pods get restarted with the new resource limits, their ongoing requests fail. However, we think the failures for *VPA* could be prevented if we used the *termination grace period* feature that Kubernetes provides for the pods. On the other hand, *HPA* looks exceptionally good independent of the inclusion of *CA*.

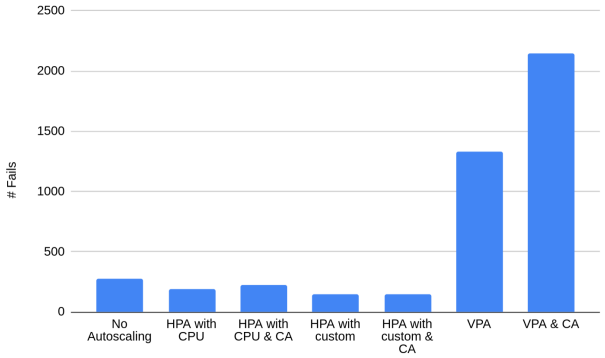

Figure 6 - Number of failures for each scenario

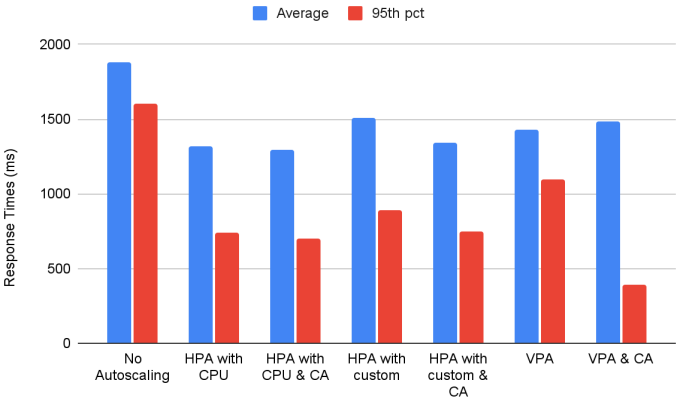

Figure 7 - Response times for each scenario

To continue with the response times, in *Figure 7* it is visible that each auto scaling option is better than the one with no auto scaling, with the 95th percentile being far better than average. This graph also shows that the average response times are very close to each other, whereas 95th percentile of response times are significantly better at *VPA & CA*. This result might motivate one to choose *VPA & CA* to achieve maximum user happiness for 95% of their users.

Comparison #2 - HPA Comparisons

In this comparison, we investigated the effects of CPU thresholds (50% and %75) for auto scaling. In *Figure 8* below, there is only a slight difference between the response times of different CPU thresholds with 75% having a smaller response time for 95th percentile.

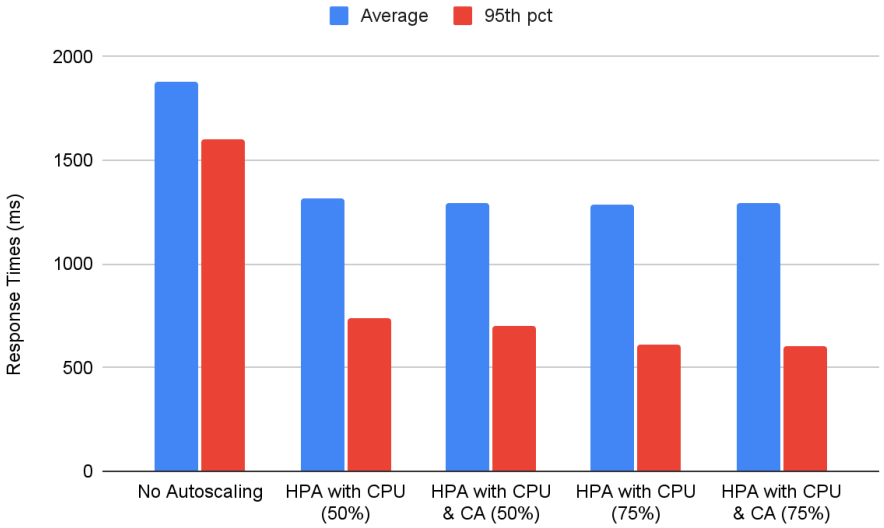

Figure 8 - Response times for HPA scenarios

As we mentioned in previous sections, we were not expecting a significant improvement for *HPA* when *CA* was enabled on response times because our load testing was not able to bring all pods into their limits in *HPA with CPU*. So additional nodes did not bring a significant value. However, this is the case because the load is not big enough to utilize the advantage of *CA*. If we were to use a more computationally complex endpoint

or increase the number of requests, the advantage of *CA* would be clearly visible. To justify this claim, please take a look at the table below.

Requests	Pod information				
Label	**# pods at end**	**# pending pods**	**avg cpu usage (millicores)**	**# nodes at end**	**total cpu usage (millicores)**
HPA with CPU (50%)	7	24	210	2	1470
HPA with CPU & CA (50%)	21	11	68	4	1428

Figure 9 - Pod information for HPA with CPU 50%

First scenario runs 7 pods, each using on average 210 millicores of CPU whereas the maximum limit is 250 millicores. Scaling up at this point doesn't affect the response times since the load on a server affects the response times when incoming requests can't find any resources on the fly. This is the reason why *CA* doesn't benefit *HPA with CPU* (75%), also.

Comparison #3 - CPU Usage vs Custom Metric

In *Figure 10* we can see that there is a slight performance difference between the CPU usage and custom metric based auto scaling. CPU usage based auto scaling slightly outperforms custom metric based auto scaling. Also, the effect of *CA* is more significant in the scenario with custom metric based auto scaling. These are all expected since we mentioned that for *HPA with custom* we deployed Prometheus related pods which take some of the CPU resources themselves. In spite of this effect, *HPA with custom & CA* performs similarly with its CPU counterpart due to additional 2 nodes.

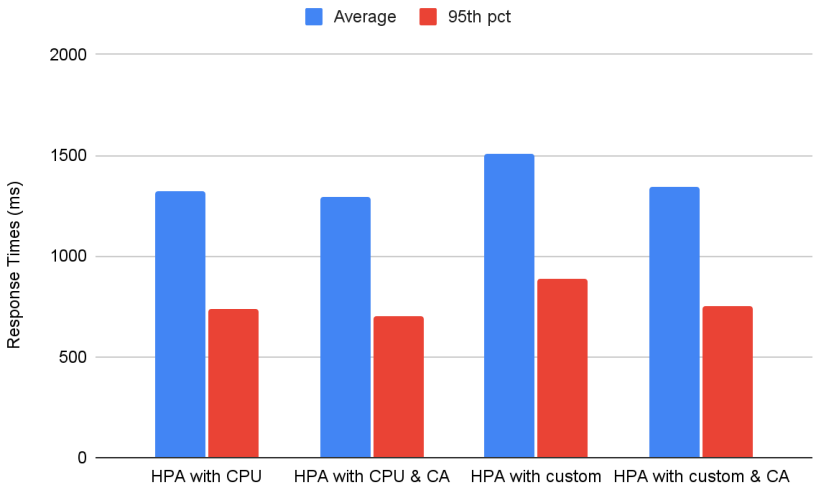

Figure 10 - Response times for HPA with CPU and custom metric

Comparison #4 - VPA with CA

VPA with CA scenario outperforms other scenarios in the 95th percentile of the response times as stated in Comparison #1. Also, the effect of *CA* is more visible in *VPA* than *HPA*. *VPA* with *CA* decreases almost 3 times in the response time. This might be caused by *VPA & CA* having an additional node just for the scaled up pod.

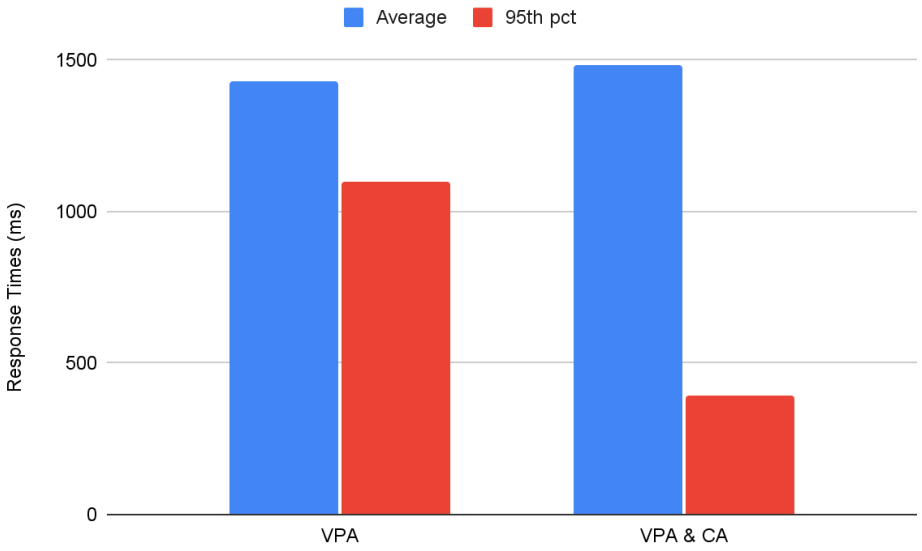

Figure 11 - Response times for VPA scenarios

Comparison #5 - Responsiveness of HPA and VPA

In *Figure 12* below, we see the response time graphs for *HPA* and *VPA* scenarios with no *CA* enabled. Looking at the *HPA* graph, we see that around 1 minute the response times get smaller indicating the pods have horizontally scaled. On the other hand looking at the graph at the right, we see that the response time decreases at around 3 minutes after the tests start, meaning the pods started vertically scaling. This shows that *HPA* could be a better suite for handling instantaneous loads where *VPA* would not have a chance to react.

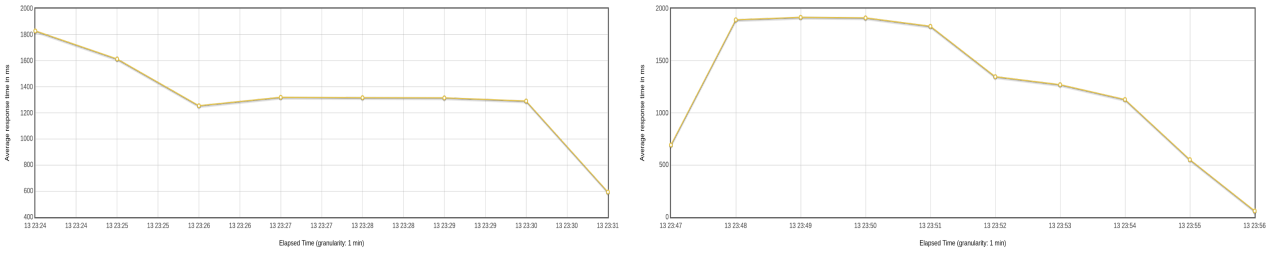

Figure 12 - Response times vs time, HPA left, VPA right

Conclusion

We have examined various auto scaling options in Kubernetes in terms of various metrics such as the number of failures, response times, average and total pod CPU usages. We performed this examination with JMeter, as our load tester, on different scenarios including *HPA*, *VPA* and *CA*. We came to realize that the scenarios

with *VPA* result with more failures compared to *HPA* and no auto scaling. No auto scaling performs the worst in the 95th percentile of response times whereas *VPA & CA* performs the best. Regarding the response times, inclusion of *CA* is more significant in *VPA* than it would be in *HPA*. However, this was partly caused by us not being able to provide a stronger load testing. On the other hand, *HPA* is better for handling instantaneous loads compared to *VPA*. Within *HPA* variations, CPU usage based auto scaling slightly outperforms custom metric based auto scaling. However, application developers might want to choose the custom metrics since they can include business logic into their application's scaling options.

The experiments we have done also inspired us about other scenarios that we could not do or did not think of testing. For example, we know that *HPA* and *VPA* do not work with each other if we are using CPU or memory resource usages to scale. However, if we use custom metrics for HPA, we can also use VPA to scale our applications. The combination of both could result in far better outcomes. We are also aware that we could not take full advantage of CA in Kubernetes. It would be interesting to see the results with a far stronger load testing in a Kubernetes cluster without resource quota.

Bibliography

1. https://kubernetes.io/
2. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/
3. https://github.com/kubernetes/autoscaler
4. https://github.com/prometheus/prometheus
5. https://github.com/prometheus-operator/prometheus-operator
6. https://github.com/kubernetes-sigs/prometheus-adapter