Why do Kubernetes clusters in AWS cost more than they should?

Dmytro Dyachuk Follow
May 24, 2018 · 7 min read

Congratulations, if you are reading this article this means that you made it to the point where the efficiency and cost your Kubernetes clusters became or about to become of some interest. Your journey probably started from a few fleet instances running in the cloud and now probably the costs for running the cluster amount to some significant number. Perhaps, you have already looked at your AWS bills and have already wondered if it can cost less. The truth is: yes it can. I spoke to a bunch of different companies running Kubernetes in the cloud and summarized the “tribal” knowledge about cost reduction. So, here is a list of the five issues which can contribute to cost overruns and easy remedies for fixing them.

Over-sized pods

It is common practice to use a standard template for limits and requests for pod provisioning. If requests describe the minimal requirement for the CPU and memory for a pod to be scheduled on a node, the limits describe the max amount of CPU and memory the pod can consume on that node. Very often engineers set the initial limits by “guesstimating”, doubling it just to be on the safe side, and then promising change it later, once more information is known. Usually, “later” never

happens… As a result, the footprint of the entire cluster starts inflating over time, exceeding the demand for the actual services running inside the cluster. Just think about it, if every pod is using only 50% of the allocated capacities and the cluster is always is 80% full, that means that 40% of the cluster capacity is allocated but not used, or simply put —wasted. There are a few different ways of dealing with such scenarios.

The first and the most simple thing you can do is set up monitoring with Grafana and Prometheus and periodically check the CPU and memory utilization by each pod. Then you can use the 99.9th percentile of the each to determine the new more optimal limits.

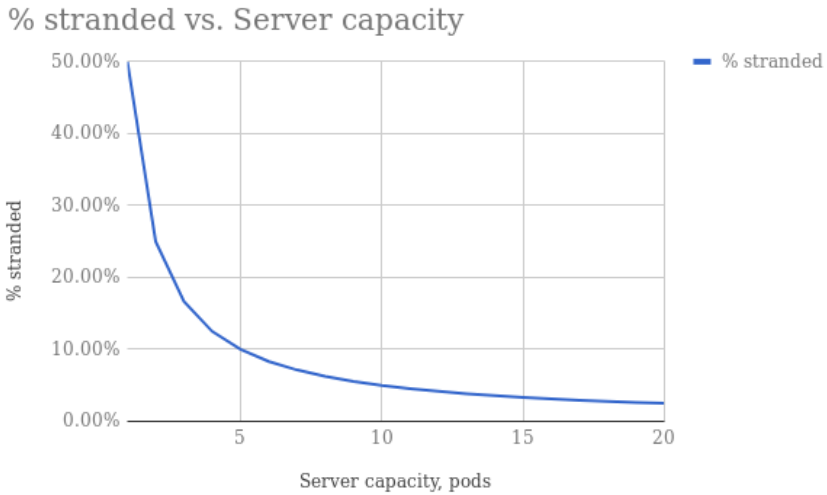
Employing the vertical pod auto-scaler (VPA) is another way of addressing this problem. The vertical pod auto-scaler effectively automates the promise the engineer made in the previous step. It monitors the CPU and memory utilization by the pods and adjusts the limits accordingly by terminating the pods and restarting them with the new limits. It works in both directions—if a pod is over-sized, the VPA reduces its size according to the observed utilization. If the pod is getting killed by OOM killer due the shortage of the allocated memory, it will be restarted with a larger memory limit. VPA handles CPU under-provisioning in a similar fashion and respects a disruption budget to avoid constant restarting.

Another way of dealing with the situation without resizing the pods is by using the horizontal auto-scaler (HPA). If the demand for CPU and memory correlates with the traffic hitting those pods, rather than resizing the pods, it might be easier to increase their number. That's

exactly what HPA does. This can be triggered either by the CPU or memory utilization, or using a custom application metric. For specific information on setting up HPA with custom metrics it's worth checking out this blog post by Mark Luksa.

Wrong node size and type

Smaller nodes have higher relative OS footprint and increase management overhead. The smaller the node, the higher the amount of stranded resources. Stranded resources are CPU or memory which are idle, yet cannot be allocated to any of the pods, because the pods which are to be scheduled are too big to claim it. In my next post, I might dive a bit deeper into the subject of the maximum theoretical cluster utilization, but for now here is a simple graph showing the % of stranded resources as a function of node capacity. The server capacity is measured by the number of pods it can accommodate. As you can see, if the pods' sizes are close to the size of the node (server) the percentage of the resources which are stranded gets higher. As a rule of thumb, if the node can accommodate 10 pods, the loss due to stranded resources is only 5%.

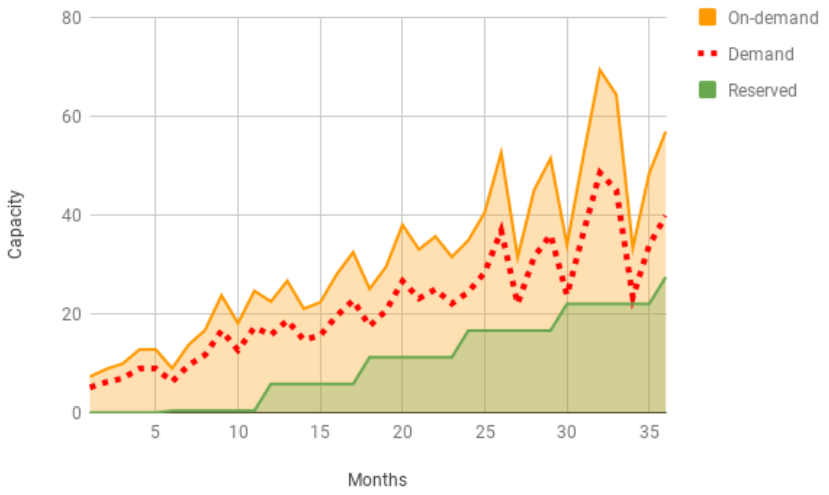


Most of the companies I've spoken with mentioned that they run their workloads on M-type instances. Alternatively you can use C-type instances which have a higher ratio of CPU to memory. Which instance type is the right one to use? Well, it is a simple dynamic programming problem from a formal standpoint. I've put together a Google Sheet calculator—please check the tab "Instance types". It factors in the percentage of stranded resources and OS overhead, and estimates the monthly EC2 bill based on the pod size and the number of pods. You can clone the document, update the number of pods you plan to run and the average size of each pod, and it will find the cheapest instance type/size.

	A	B	C	D	E	F
1	**Inputs**					
2	Pod size, GB	4				
3	Pods size, cores	2				
4	Containers	200				
5	Node OS footprint, GB	2				
6						
7	Total vCPUs demand	400.00				
8	Total memory demand, GB	800.00				
9						
10	**Type**	**vCPUs**	**Memory, GB**	**Storage**	**Hosting costs, $/month**	
11	c5.9xlarge	36	72	EBS Only	$13,395.46	
12	c5.18xlarge	72	144	EBS Only	$13,395.46	
13	c5.4xlarge	16	32	EBS Only	$14,387.71	
14	m5.4xlarge	16	64	EBS Only	$15,128.99	
15	m5.12xlarge	48	192	EBS Only	$15,128.99	
16	c5d.9xlarge	36	72	1 x 900	$15,128.99	
17	c5d.18xlarge	72	144	1 x 1800	$15,128.99	
18	m4.4xlarge	16	64	EBS Only	$15,759.36	
19	m4.10xlarge	40	160	EBS Only	$16,051.20	
20	m5.2xlarge	8	32	EBS Only	$16,249.65	
21	c5d.4xlarge	16	32	1 x 400	$16,249.65	
22	m4.16xlarge	64	256	EBS Only	$16,343.04	

Incorrect instance class

On-demand, reserved, or spot instances—which ones are the best? On-demand is the most expensive option which permits the most flexibility and reliability. Reserved instances provide a good trade-off between pricing and flexibility. Spot instances are the trade-off between the reliability of the nodes and pricing. The math for the optimal selection of the instance type gets complicated pretty fast. As an alternative, the more straightforward rule is to periodically (e.g. quarterly) revise the instance allocation and switch from on-demand to reserved instances if your fleet size has not changed for some time.

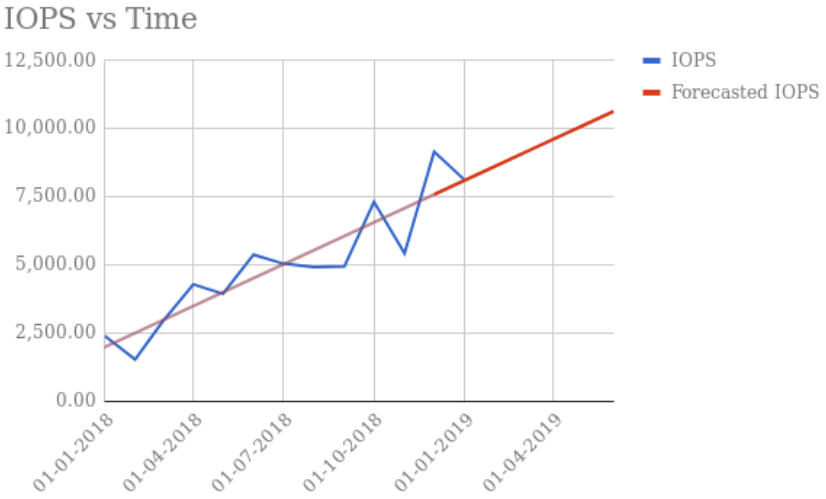


In the chart above, I've put together a visualization of this process with an example of a gradually growing yet fluctuating workload. This algorithm allows you to enjoy the savings from using reserved instances, yet preserves the flexibility of using on-demand instances.

Over-provisioned storage

Often little thought is given to the sizing of volumes attached to nodes. AWS offers two main types of storage: local, and EBS. The latter has different classes: GP, IO1, ST1 and SC1. While local storage comes for free with a selected list of instances, EBS storage can constitute 20–40% of the final bill. An audit of the allocated EBS volumes vs utilization often reveals that up to 50% of the volumes were substantially (>60%) over-provisioned. The simplest remedy is as simple as checking the volume size, comparing it to the allocated size

and then resizing it. The important step is to factor in the growth for the volumes and IOPS demand. I've put together a simple spreadsheet which shows how to do that [Google doc].



Increasing the volume size can be done via the AWS API, yet downsizing it is not that straightforward. Recently, I came across this post where Andrew Trott shows simple steps for downsizing EBS volumes. With a bit of scripting, this process can be fully automated.

If you want to eliminate the EBS costs completely, you might consider using Rook. Just pick some instances with which have local SSD storage and Rook will roll out Ceph. Ceph is a distributed storage system. It unites the local storage from multiple servers and makes it available to pods in the form of provisioned volumes.

Lost and abandoned resources

Bugs in provisioning scripts, misconfigurations and human errors can result in a significant amount of lost resources. This can include pretty much any type of resources used, e.g. instances which are not a part of any cluster, pods and deployments which are not part of any service, public IPs, ELBs, EBS volumes which are not attached, backups, and so on. Dealing with such cases is pretty straightforward, but it might require introducing some organizational rules. Here is an example of what such rules might look like.

1. Each resource should have an owner; the owner can be declared using a tag. Some organizations really struggle with establishing the ownership of their cloud resources. My next suggestion might come across as a somewhat draconian rule, but you can introduce a script which deletes all the resources which do not have owners. Usually, if you implement such a rule, you will discover the owners of those resources right away.

2. Periodically perform utilization audits. The resources which have seen zero traffic automatically get on the eviction list.

3. Contact each of the owners which have items on the eviction list and clarify the business context of that resource. Sometimes, those resources might be a part of a critical system that is still in development.

From my professional experience, the eviction lists can be as large as 30–40% of the total resources used, and 95% of the items on those lists

do not bring any business value. So even though it might sound like a bit work, it is usually quite worth it in the end.

Conclusion

Kubernetes permits co-allocating the applications on the same nodes, which can dramatically reduce the AWS bill. Yet, incorrectly sized instances, volumes, instances and lost resources can lead to the inflation of the cost of Kubernetes clusters. Simple due diligence of collecting metrics and cloud bills, and some spreadsheet magic can help you to trim the total bill by 30–50%. I've shared a simple calculator which can help with finding the best fitting instance size, stranded resource overhead, and a template for doing IOPS and size projections using linear regression. For more complicated use cases of full automation of processes I described this post, you might want to check out Untab. Here is a short video explaining what you can do with it.

Dmytro Dyachuk is a co-founder of Untab.io which audits Kubernetes clusters for inefficiencies, estimates the true cost of services and implements team billing for the use of infrastructure.