


# Our First Kubernetes Outage



Adam Hawkins

Follow

Apr 26, 2017 · 5 min read



This is public postmortem for an a complete shutdown of our internal Kubernetes cluster. It's shared with you all so everyone may learn.

## Background

Saltside has multiple Kubernetes clusters for different use cases. This postmortem covers an outage on `np.k8s.saltside.io`. This is our nonproduction cluster used for general developer experiments and as a target for CI builds. I lead the SRE team at Saltside. We're responsible for provisioning and maintaining these clusters.

## Impact

- Internal Kubernetes services unavailable inside `np.k8s.saltside.io`
- Tiller unavailable (`helm` commands fail, no `helm installs`)
- Unable to create new pods
- Existing pods in `Pending` phase become `Unknown`
- Flaky delete behavior

## Timeline

I had completed a round of work on our application's Helm chart and needed to test the CI process on my machine before making some commits. The CI process builds charts for all our markets (Efritin.com, Tonaton.com, Ikman.lk, and Bikroy.com), installs them on a cluster (`np.k8s.saltside.io`), and runs `helm test` against each release. I fired off the command and monitored pods with `watch kubectl get pods`. All pods came to the `Running` phase after sometime. I continued watching the pods incase of failed probes triggering extra restarts. I noticed some pods entered the `Pending` phase after being `Running`. I shrugged this off as a transient issue that would resolve itself. Nothing was broken or critical at this point, plus I had a meeting to attend. I expected things to resolve during the meeting.

I checked on things two hours later. The issue had not resolved itself. It became larger. I noticed pods in an `Unknown` state. This was new to me and really piqued my interest and made me thing something had definitely gone wrong internally.

I used `kubectl describe pod` to find more information. The pod event logs were full of image pulling errors from the container runtime (Docker in our case). Again, I largely wrote this problem off because:

- This is the nonproduction cluster
- Node use `t2.medium`
- Historically we've had a lot of instability from the official Docker registry, so failed pulled are nothing new

My estimation at his point was things has gotten overwhelmed pulling all the images. I decided it would be easier to start deleting things given the uncertainty around the root cause and low business impact of the current conditions.

I attempted to start with deleting Helm releases. This would delete everything (`Deployment`, `Service`, `Pod`, `ConfigMap`, `Secret`) without needing to delete individual resources. Unfortunately the `tiller` pods in `kube-system` had started trashing. They were in a crash loop for an unknown reason. Now the only option was to delete things outside of `helm` and worry about `helm` afterwards.

Initial `kubectl delete` commands *seemed* to work successfully but nothing actually was removed. I started to assume there was some strange internal inconsistency that was being exacerbated by CPU/Memory/Disk consumption. A quick google search (naturally leading to Stack overflow) revealed a way to forcibly delete resources. I added `--force --grace-period=0` and attempted to delete pods. The pods where removed from Kubernetes (not known in `kubectl get`). However I did not verify at Docker level (which *should* have been done). Now I needed turned my attention to restoring the `tiller` service.

I ran `kubectl get pods -n kube-system` to see the internal state there. Pods were in a variety of undesired states. Notably some where in `DeadNode`. This was another I had not seen before — another indicator of something going seriously wrong. This prompted me to inspect the nodes with `kubectl describe nodes`. Some nodes reported the container runtime was done and others reported the kubelet was done for various failed checks. I attempted to SSH into the instances but my connections repeatedly timeout. I decided it would be easier to simply terminate these instances and let the ASG sort it out. I terminated the three worker nodes and waited for new instances to join the cluster. Sure enough everything was back to normal after about 10 minutes.

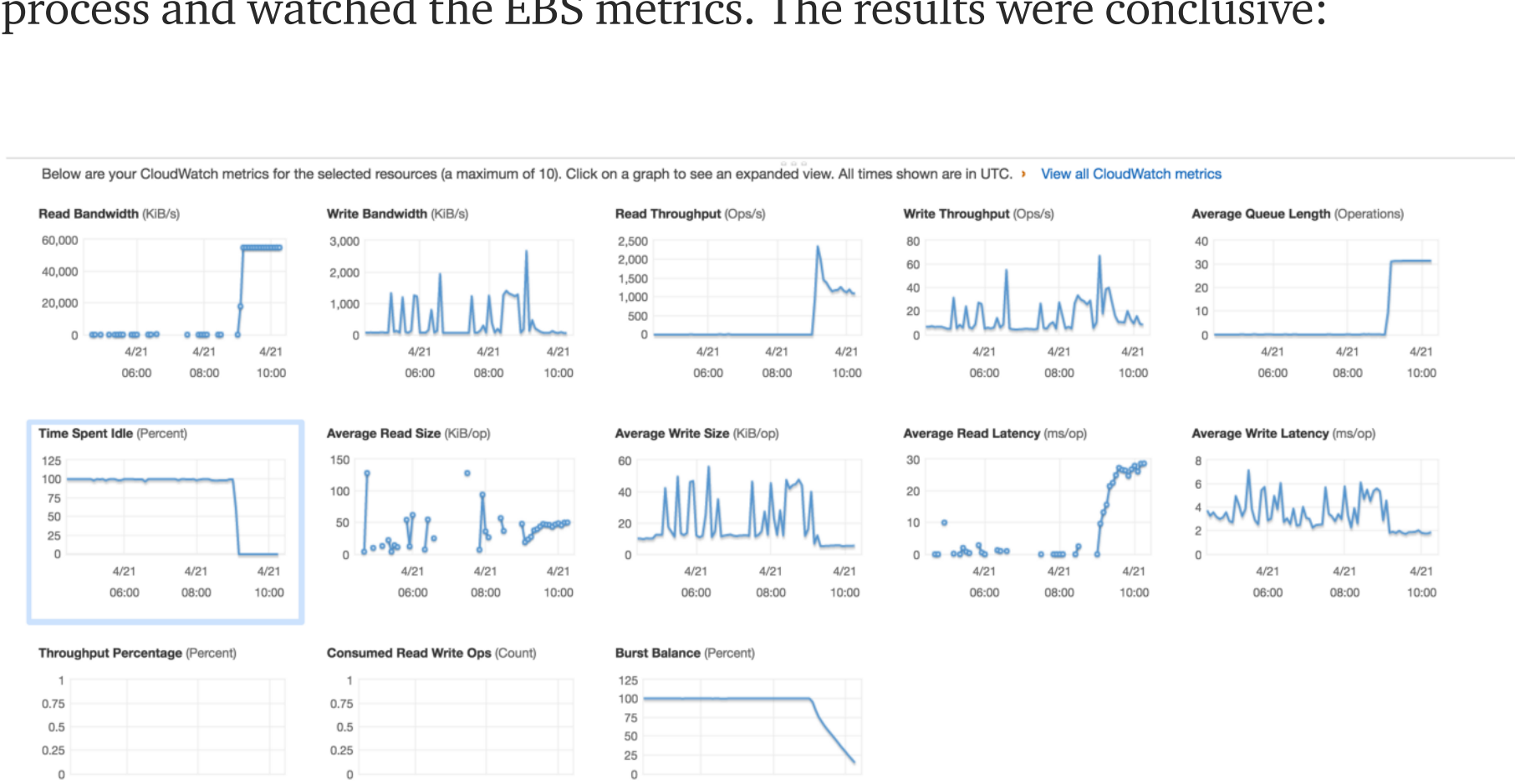
## Observations

This sequence of events was quite concerning. The normal activity of deploying our application could collapse the cluster and leave things in disarray. Terminating worker nodes is undesirable solution because pods are not drained to other nodes which would like create unavailability for our customers. It works, but it cannot be standard procedure for these kind of outages. Beyond that there were other important observations about our current setup:

- Node unavailability was not picked up any monitoring. `kubectl get nodes` reported that all three worker nodes were not ready, but nothing was reported.
- We had no pods/phase metrics (i.e. Number of pods in `Unknown` or `Pending`).
- Deploying a small number of a containers in parallel completely overloaded the cluster
- Tiller pods failed to reschedule because of CPU limits. This is curious because there were no CPU request issues at the node level. This warrant future investigation.
- `helm init` runs tiller with a single replica. This is not HA. It's also uncertain what the HA story is with tiller.
- We had no node (CPU/Memory/Disk) metrics
- We did not understand what the Kubelet checks test under the covers (e.g. What is “Disk Pressure?”)
- We had no way to throttle the number of newly created pods. This would not solve the problem, but it would mitigate risk in future scenarios. There's no need to DoS our own cluster in case something goes wrong.
- We had no CloudWatch metrics in DataDog for this AWS account.

## Outcomes

There were some short term actions to take based on the observations. First, I wanted to repeat the process with more telemetry to better diagnose the system bottleneck. I fixed the issue with the DataDog integration and our nonproduction AWS account so there was *some* data. I suspected bottleneck was either the disk or the Docker daemon itself. I repeated the process and watched the EBS metrics. The results were conclusive:



The graphs show the EBS queue length hockey sicking. It turns out that Kops used 20GB `gp2` EBS volumes by default. `gp2` IOPs are tied to disk size. We needed more IOPs and more disk space (`df` reported 90% usage). This also revealed `kops` does not support EBS optimization. Hopefully this if fixed ([GitHub issue](#)) in future release. This was may not solve the problem, but it will increase overall performance. This is also a problem for large instance types that require EBS optimization. I scaled up the root EBS volume to 120GB and repeated the exercise. I also deployed `DaemonSet` that ran a sleep loop inside common base images (e.g. `ruby:2.4` or `alpine:3.2`). This hack work around “pre-pulled” images on nodes until [Kops supports user supported hooks](#). This time charts installed successfully without killing the Kubelet. However other things broke so the cluster still cannot support this amount of parallel container operations.

I also spend time investigating the other observations.

- The DataDog agent does not yet support the full suite of `kube-state-metrics` (notably the pod/phase metrics). This planned for a [future release](#).
- We need to configure [different allocation limits](#) for different layers in Kubernetes.

There are two things that keep me up at night right now.

- HA tiller. If tiller enters a crash loop this blocks our ability to deploy fixes (via chart installs/upgrades). HA must be discussed with the Helm team.
- Pod throttling. Our product comprises ~35 different containers deployed 4 different times. We can scale the cluster to handle this but it's an unending battle. It would be nice though to have *something* in this area.


Top highlight

Kubernetes

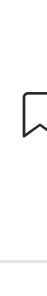



Docker

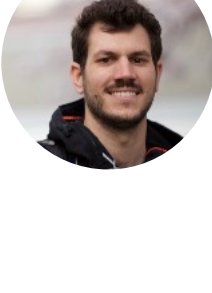
DevOps

Containers



302 claps






WRITTEN BY

Adam Hawkins

Code DJ! Tech: <https://hawkins.io>. Podcast: <https://smallbatches.fm>.



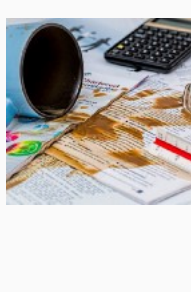
Saltside Engineering

Field notes from the team behind Tonaton.com, Bikroy.com, and Ikman.lk

See responses (9)

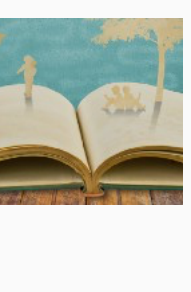
## More From Medium

Custom Error Page for Nginx Ingress Controller



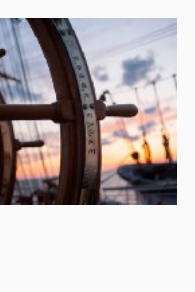
Zhimin Wen in ITNEXT

How to Create a Kubernetes Cluster Locally — Simple Tutorial




Srikant Vavilapalli in Capital One Tech

Helm 3: Fun With the New Beta



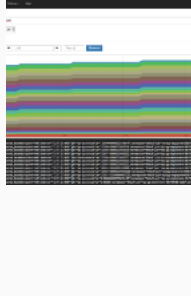
Chris Cooney in Better Programming

Cheapskate's Journey to On-Demand Load Tests with Locust



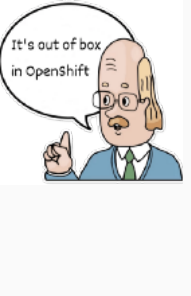
Michael Bogan in Level Up Coding

Simple Management of Prometheus Monitoring Pipeline with the Prometheus Operator




Kirill Goltzman in Supergiant.io

Kubernetes authentication via GitHub OAuth and Dex




Arnet Umerov in Preply Engineering Blog

Garbage Collection in Kubernetes



Abhishek Sharma

5 Things We Overlooked When Putting Our First App on Kubernetes



Julian Gindi in gumgum-tech

Discover Medium

Make Medium yours

Become a member

Welcome to a place where words matter. On Medium, smart voices and original ideas take center stage - with no ads in sight. [Watch](#)

Follow all the topics you care about, and we'll deliver the best stories for you to your homepage and inbox. [Explore](#)

Get unlimited access to the best stories on Medium — and support writers while you're at it. Just \$5/month. [Upgrade](#)

Medium

About Help Legal