Dmitri Lerko

Personal blog where I cover my experiences and best practices with GCP, GKE, GitOps, Certifications and DevOps.

The shipwreck of GKE Cluster Upgrade

Mar 25, 2019
10 min read
gke gcp

GKE Rocks!

I am a huge fan of the Google Cloud Platform and especially GKE. It was fundamental in our migration of Lovehololays on-prem applications to Kubernetes. GKE's rich integrations with other GCP products allowed us to shift a lot of configuration from Ansible into Kubernetes yaml definitions which are now managed by GitOps. We are running GKE in Prod for 6+ months and will struggle to imagine any other way to run and deploy our applications now. The story I am about to tell you is a precaution on things that may go wrong during cluster/node upgrades. Its aim is to save you stress and worry next time you are in the same situation.

With the migration to GCP underway, we were too busy to focus on more BAU operations like cluster and node pools upgrades. We've got a little rusty and were two versions behind of what is available on GKE. 1.10.X vs 1.12.X. With the migration workload out of the way, we've finally got time to perform a long-overdue upgrade. Up to this point, we've tested same upgrades across multiple other clusters and environments. We've noticed that there were some changes that we had to keep in mind, but none of those changes had anything to do with a production application, cluster stability.

On the evening of the upgrade, I've estimated that 2-2.5 hours will be plenty to upgrade masters and NodePools. Cluster upgrade is a two-step process, first masters, then NodePools.

Master upgrade

During the master node upgrade you are unable to change the cluster's configuration for several minutes. I've chosen the latest available 1.12.5-gke.10 and went for it. Pretty quiet start.

NodePools upgrades

The NodePool upgrade is a more impactful process. Sequentially, for each node in the NodePool, nodes are stopped from scheduling node Pods, existing Pods are getting drained, and finally, node is getting upgraded. It is quite clear that the overall process is highly dependent on total number of nodes in the cluster.

We are running a lot of NodePools for various reasons. Not all of those have active nodes at all times. NodePools were stuck at version 1.10.X. I started a small and upgraded couple of 7 node NodePools which run our clustered applications. This went smoothly, clusters managed failover gracefully, and I've got a first taste of running on the latest GKE. Not much flavour to it at the moment as it just worked as well as usual. Beyond the first two NodePools it all went downhill.

Problem 1: It is taking too long to drain nodes

GKE Cluster with 14 nodes and 5 minutes per node upgrades was a breeze, yet it still took over an hour to complete. What would happen if you have 100 more nodes to go through with some of them taking over 15 minutes to drain fully. *You've got something wrong with your configuration* you'd say, and you won't be too far from the truth.

Configuration which may slow down node drain

emptyDir and other non-persistent storage

Kubernetes drain is unsure what to do when Pods have data stored in emptyDir. From the point of view of the drain, it is unable to make a call whether it is safe to evict a pod with local data.

```
- name: shared-data
  emptyDir: {}
```

Solution:

By adding safe-to-evict annotation, we are essentially allowing for pod eviction. This flag is also useful for improving utilisation of your cluster, as certain types of pods will never get rescheduled and may end up the only pod running on the node, preventing cluster autoscaler from the downscaling.

```
annotations:
  cluster-autoscaler.kubernetes.io/safe-to-evict: "true"
```

Long terminationGracePeriodSeconds

By default, Kubernetes sets termination grace period to be 30 seconds which is more than enough for lightweight, cloud native applications to terminate. This limit is too low for either heavyweight applications or applications with long, slow transactions. Both types got plenty of work to do to gracefully terminate.

```
terminationGracePeriodSeconds: 600
```

I've discovered apps with 5-10 minute grace periods, which dramatically prolongs the drain time. The termination grace period may also eat into your deployment times, however, with deployments and cluster autoscaler, it is at least possible to allow for 100% spike during the deployments, therefore hiding away slow termination grace periods.

Solution:

There could be no quick way out of the legacy. My main suggestion would be to re-evaluate existing grace periods as some of those may not have been well tuned, and an arbitrary large number was set by developers to minimise the risk.

Pod Disruption Budget

Pod Disruption Budget (PDB) is a mechanism where you allow for a number/percentage of pods to be terminated, despite it bringing your current replica count below desired. This may speed up the node drain by removing the need to wait for migrated pods to become fully operational. Instead, drain evicts pods from a node per configured PDB, leaving it to deployment to scale missing pods back up on some other available node.

```
# Sample PDB
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: your-app
  namespace: your-namespace
spec:
  selector:
    matchLabels:
      app: your-app
  maxUnavailable: 3
```

Last-resort trick for speeding up slow NodePool upgrades

At one point it was clear that sequential drains will be too slow to complete this maintenance at a reasonable hour. Quick and dirty trick was to list all the nodes in the NodePool and drain nodes preemptively. You should exercise extreme care with this approach as it is possible to evict more pods than your service can tolerate losing.

```
kubectl drain node-name-bb78e11e-6f84 --ignore-daemonsets --delete-local-data
```

Another alternative is available to IaaS users - spin up a new NodePool and migrate pods to it. It will be on the latest version from the start.

Problem 2: Confused Managed Instance Group

We've discovered that one of the older NodePools was left in the confused state. Essentially, there were already six nodes which were not running any pods, yet not getting terminated. While we could not track down the exact point of time when this has happened, we could not explain certain inconsistencies in the pod scheduling around these nodes. Essentially, there were pods in a pending state, never get scheduled on these zombie nodes, nor scale up the cluster.

Solution:

1. Map Managed Instance Groups (MIGs) to the NodePool
2. List instances per MIG

   ```
   # List instances for your particular MIG
   gcloud compute instance-groups managed list-instances $YOUR_MIG_NAME --project $YOUR_PROJECT_NAME --zone $YOUR_MIG_Z
   ```

3. Delete stuck instances

   ```
   # Individually
   gcloud compute instance-groups managed delete-instances $YOUR_MIG_NAME --instances=INSTANCE_1 --project $YOUR_PROJEC


   # In groups
   gcloud compute instance-groups managed delete-instances $YOUR_MIG_NAME --instances=INSTANCE_1,INSTANCE_2,INSTANCE_3
   ```

Problem 3: Stuck NodePool upgrade

Upgrade for one of the NodePools has simply got stuck. There were no signs of nodes getting drained. When you perform NodePool changes, you are blocked from making any further changes to the Cluster via UI. You can't even cancel an in-progress upgrade.

Solution:

Fallback on gcloud to cancel cluster operation. Please be aware, that NodePool upgrade can leave your NodePool in an inconsistent state where nodes which got already upgraded won't get rolled back to the previous version.

List operations across Clusters in your projects. You are looking for operation with `operationType: UPGRADE_NODES` and `status: RUNNING`

```
# This lists operations from ALL clusters in the project
gcloud container operations list --project $YOUR_PROJECT_NAME
```

Get more details about stuck operation.

```
gcloud container operations describe $YOUR_OPERATION_ID --project $YOUR_PROJECT_NAME --region $YOUR_CLUSTER_REGION

detail: 'Done with 1 out of 20 nodes (5.0%): 1 being processed, 1 succeeded'
name: operation-xxxxxxxxxxxxx-xxxxxxxxxx
operationType: UPGRADE_NODES
selfLink: https://container.googleapis.com/v1/projects/xxxxxxxxxxxxx
startTime: '2019-03-19T02:23:01.568097105Z'
status: RUNNING
targetLink: https://container.googleapis.com/v1/projects/xxxxxxxxxxxx
zone: xxxxxxxxxx
```

Cancel operation

```
# Terminate the offender
gcloud beta container operations cancel $YOUR_OPERATION_ID --project $YOUR_PROJECT_NAME --region $YOUR_CLUSTER_REGION
```

You'll be asked to confirm the cancellation. gcloud cancel operation also prints detailed information about the operation that you've cancelled. e.g. how many nodes got upgraded.

Problem 4: Ingress gone

One of our convenience Ingresses has stopped working after the cluster upgrade. Logs weren't helpful, and we've fallen back on Google support to learn more about the problem. Apparently, there was a known bug which only affects ingresses which have between 44 and 47 characters in the namespace + name. Likelihood of hitting something like this is low, and reproducibility of the issue in such a narrow range is also very odd. Nevertheless, this has taken down a couple of our internal convenience ingresses. We had to fall back to the port-forward for a little while.

Solution:

Whether its development, staging, UAT, QA, pre-prod - there is very little excuse to not run an almost identical replica of your production in a sandbox environment to discover these types of issues early. In our case, we do have early discovery environments. However, production ingress had an extra three characters in the name.

Lesson learned - keep environments as identical as possible, including the *naming*. This issue has now been fixed and available for GKE 1.12.8-gke.7.

Problem 5: Confused HPA

Next day after the upgrade we've noticed that our CPU based HorizontalPodAutoscaler started to get confused. *HPA would stop reporting current CPU load, therefore triggering more and more scale up events until it hits the max ceiling.* This has a number of consequences:

- Increased load on downstream systems. Even if requests count didn't change, maintaining connections with 100 pods vs 10 may pose significant performance impacts in terms of memory and load.
- Unjustified cost increase by running more nodes with low utilisation
- Slown down deployments. With more pods to schedule and terminate, deployment times have slipped.

Solution:

We could not pinpoint the specific root cause of this issue. However, at times we've noticed that a single deployment will have two active replicasets for hours. In this case, a gentle scale down of the older replicaset did the trick. In other instances in which there was a single active replicaset, but HPA was still confused, a dummy deployment (change dummy envvar) seemed to help.

To further protect ourselves from running an excessive amount of compute we've implemented following alerts using Alertmanager, Prometheus and Slack.

- Alert when HorizontalPodAutoscaler is at 70% and 100% of its capacity
- Alert when deployment has two active replicasets for 30 minutes and 60 minutes

Post(nearly)mortem

Per GKE release notes on 14th of March:

> GKE 1.12.5-gke.10 is no longer available for new clusters or master upgrades.
>
> We have received reports of master nodes experiencing elevated error rates when upgrading to version 1.12.5-gke.10 in all regions. Therefore, we have begun the process of making it unavailable for new clusters or upgrades.
>
> If you have already upgraded to 1.12.5-gke.10 and are experiencing elevated error rates, you can contact support.

It was 19th when I was still able to choose and upgrade the cluster to 1.12.5-gke.10. Therefore some of the above issues were easily preventable. As of 26th of March, there is 1.12.6-gke.7 available patches ingress and improves cluster autoscaling.

Lessons learned

- Always consult GKE release notes first before making upgrades to the cluster
- Use auto-upgrade features for non-production environments and discover impacts early
- Use a combination of manual cluster upgrades and auto node upgrades for production to have a great balance between predictability and stability of your production and effort needed to keep everything up-to-date.

While upgrading gcloud cli recently, I've noticed the following release notes:

```
240.0.0 (2019-03-26)
  Breaking Changes
      * ** Kubernetes Engine ** Enabled node auto-upgrade by default for
        clusters and node-pools created
      * ** Kubernetes Engine ** with gcloud beta container
        <clusters|node-pools> create. To disable manually,
      * ** Kubernetes Engine ** use the --no-enable-autoupgrade flag.
```

Seems like GKE is pushing towards auto-upgrades as the default option which you have to opt-out from. I can relate on this being my new preferred option too.

Find posts by tags

certifications dataflow flux gcp gcr gitops gke jekyll kubernetes label_replace prometheus recording-rules redis sensu