Best practices for Openshift

Importance of masters and ETCD

ETCD which is running on masters is key-value database that holds all data and objects of cluster. Any performance issues (timeouts, delays) can lead to unstable cluster.

Master count

On Openshift, only 3 masters are supported and it was confirmed that higher number of masters will not help with performance (as it makes synchronization and network latency even more slow).

If you plan to run extra large clusters with 100+ workers, you should make sure you use best performant storage for masters.

Storage

For Three-Node OpenShift Compact Clusters, make sure that you have dedicated NVMe or SSD drives for the control plane and separated Drives for application and another infrastructure stacks. Eventually, according to current workload in place, dedicated SSD drives for etcd (/var/lib/etcd) must be also taken in consideration.

ETCD network considerations

Network can handle enough IO and bandwidth between masters is fast enough. Having masters in different DCs is not recommended (as rather RHACM should be used in case of OCP), but if required, network latency should be ideally below 2ms.

Technologies like OCS, big data or even vMotion could mean extremely high IO at specific times (when moving VMs or huge data) and could seriously affect ETCD performance.

troubleshooting:

Apart from metrics you can check

```
$ oc debug node/<master_node>
[...]
sh-4.4# chroot /host bash
sh-4.4# ip -s link show
```

on each node to see if there is no excessive amount of dropped packets or RX/TX errors.

To see latency, you can run

```
curl -k https://api.<OCP URL>.com -w "%{time_connect}\n"
ping <node>
```

ETCD object count

ETCD hosted on average storage will usually have performance problems when there is more than ~8k of any of objects (like images, secrets, deployments, replicasets, etc..) and they have to be periodically cleaned up, unless you move to fastest storage (with low latency and high sequential IOPS).

For excessive number of events it's not enough to delete them, but rather it should be identified which pod/operator/pipeline is producing those events. Also, creating and deleting too many objects in short time can lead to compaction being triggered too often and this also could have effect on overall performance.

Namespaces

By default, there are actually three namespaces that Kubernetes ships with: default, kube-system (used for Kubernetes components), and kube-public (used for public resources). kube-public isn't really used for much right now, and it's usually a good idea to leave kube-system alone, especially in a managed system like Google Kubernetes Engine (GKE). On Openshift we have several openshift-namespaces used for cluster components.

This leaves the default Namespace as the place where your services and apps are created.

There is absolutely nothing special about this Namespace, except that the Kubernetes tooling is set up out of the box to use this namespace and you can't delete it. While it is great for getting started and for smaller production systems, I would recommend against using it in large production systems. This is because it is very easy for a team to accidentally overwrite or disrupt another service without even realizing it. Instead, create multiple namespaces and use them to segment your services into manageable chunks.

Creating many NS don't add a performance penalty, and in many cases can actually improve performance as the Kubernetes API will have a smaller set of objects to work with.

Do not overload namespaces with multiple workloads that perform unrelated tasks. Keep your namespaces precise and straightforward.

Namespaces should not be created at will by anyone in the organization. Allowing individuals to create namespaces without management will lead to many additional environments.

Cross Namespace communication

Namespaces are "hidden" from each other, but they are not fully isolated by default. A service in one Namespace can talk to a service in another Namespace. This can often be very useful, for example to have your team's service in your Namespace communicate with another team's service in another Namespace. When your app wants to access a Kubernetes sService, you can use the built-in DNS service discovery and just point your app at the Service's name. However, you can create a service with the same name in multiple Namespaces! Thankfully, it's easy to get around this by using the expanded form of the DNS address.

Services in Kubernetes expose their endpoint using a common DNS pattern. It looks like this:

..svc.cluster.local

Pods

Don't use naked Pods (that is, Pods not bound to a ReplicaSet or Deployment) if you can avoid it.

Readiness and Liveness Probes

Readiness and liveness probes are strongly recommended; it is almost always better to use them than to foreget them. These probes are essentially health checks.

Readiness probe ensures a given pod is up-and-running before allowing the load to get directed to that pod. If the pod is not ready, the requests are taken away from your service until the probe verifies the pod is up.

Liveness probe verifies if the application is still running or not. This probe tries to ping the pod for a response from it and then check its health. If there is no response, then the application is not running on the pod. The liveness probe launches a new pod and starts the application on it if the check fails.

Resource Requests and Limits

Resource limits are the operating parameters that you provide to kubernetes that tell it two critical things about your workload: what resources it requires to run properly; and the maximum resources it is allowed to consume. The first is a critical input to the scheduler that enables it to choose the right node on which to run the pod. The second is important to the kubelet, the daemon on each node that is responsible for pod health.

```
resources:
  requests:
    cpu: 50m
```

```
    memory: 50Mi
  limits:
    cpu: 100m
    memory: 100Mi
```

Resource requests specify the minimum amount of resources a container can use *Resource limits* specify the maximum amount of resources a container can use.

Limit Resource Usage for Specific Namespaces

You can solve this through the application of resource quotas. If you run the aforementioned command to see all namespaced API resources, you'll note that ResourceQuotas is among them. This means that each namespace can have its own quota that limits the amount of resources it can use from the node.

Images

Huge number of images could not have only effect on ETCD performance, but also registry performance. Searching for or listing images could take several seconds if there's more than 10-15k images.

Be aware that images are referenced also in Deployments and Replicasets and huge number of unused revisions of those (deployments and replicasets) could mean you will be left with huge number of images that won't be deleted when pruning.

When to use a ReplicaSet and Deployment

A ReplicaSet ensures that a specified number of pod replicas are running at any given time. However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features. Therefore, we recommend using Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

This actually means that you may never need to manipulate ReplicaSet objects use a Deployment instead, and define your application in the spec section.

Deployments and Replicasets

.spec.revisionHistoryLimit is an optional field that specifies the number of old ReplicaSets to retain to allow rollback. These old ReplicaSets consume resources in etcd and crowd the output of kubectl get rs. The configuration of each Deployment revision is stored in its ReplicaSets; therefore, once an old ReplicaSet is deleted, you lose the ability to rollback to that revision of Deployment. By default, 10 old ReplicaSets will be kept, however its ideal value depends on the frequency and stability of new Deployments.

This means, that with 2000 deployments you could create up to 20k replicasets and this could have huge performance impact on ETCD.

Operators

The Operator is a piece of software running in a Pod on the cluster, interacting with the Kubernetes API server.

What is Operator doing?

Main concern should be what is operator doing and what overhead it brings. Operators that do lot of API calling, ones that scan the files or create heavy IO/traffic could have big performance impact on the storage, CPU or network. Make sure you understand what Operator is doing, how it affects overall performance and how you can tweak it to avoid such issues. Some resource hungry operators might require that you add extra CPU and RAM to your master nodes.

Where does Operator run?

If Operator runs also on masters (virus or file scanner), it could have performance impact on master's storage (and therefor impact on ETCD).

How often Operator calls API?

Operator that is calling API server very often (collecting statistics or orchestrating CRDs) could have impact on apiserver CPU usage, which could in turn affect also ETCD.

Pipelines

Pipeline can be simple, but running complex commands that could create high IO or API activity could have impact on masters/ETCD. Another issue could be broken pipeline, that running infinitely in loop could be creating new objects or events.

3rd party software and services

TODO

Logging, networking and registry

Customer applications produce logs and you should consider add enough resources (CPU/RAM) to monitoring/infra nodes.

Another option is to move logging, networking and registry to infra nodes and offload the masters.

CLEANUP

It is very important to clean up unused resources that could be referencing other unneeded resources like images or secrets. If you run any pipeline that creates CRDs, make sure there is pipeline also to clean up those CRDs.

Be consistent Re-evaluate constantly