Kubernetes

Observability

Resource usage metrics like container CPU and memory usage are available in Kubernetes through the Metrics API. These metrics can be either accessed directly by a user with `kubectl top`, or used by a controller in the cluster, such as the Horizontal Pod AutoScaler.

The Metrics API only reports the current resource utilization of a nodes and pods. Those wishing to view a history of resource utilization should use a monitoring tool and database such as Prometheus or InfluxDB/Grafana.

In this lab we'll install the Metrics Server and explore the metrics enabled features of Kubernetes. Delete all user defined deployments, services, and other resources before starting.

1. Deploy the Metrics Server

In our kubeadm clusters, the metrics server is not installed automatically. The Metrics Server runs as a stand alone deployment in the kube-system namespace rather than a module of the controller manager. The Metrics Server requires a ServiceAccount object with an appropriate RBAC role so that it can communicate with the API server and extend the K8s API.

The metrics server deployment consists of:

- ServiceAccount for the metrics server (metrics-server)
- Two cluster roles:
  - `aggregated-metrics-reader` (system:aggregated-metrics-reader) - provides access to the pod information within the metrics.k8s.io API group
  - `resource-reader` (system:metrics-server) - provides access to information methods for pods, nodes, nodes/stats and namespaces across all API groups and information access to deployments within the extensions API group
- ClusterRoleBindings bind the ClusterRoles to the metrics-server ServiceAccount created in the kube-system namespace.
- RoleBinding (metrics-server-auth-reader) binds the preexisting extension-apiserver-authentication-reader Role to the Metric Server's service account
- APIService manifest (v1beta1.metrics.k8s.io) defines the metric-server service API extension to the core Kubernetes API
- Metrics server deployment that actually scrapes kubelets for information
- Metrics server service that exposes the metrics server pods to the rest of the network

We can launch the Metrics Server by creating all of the resources in the deploy/kubernetes directory by providing `kubectl apply -f` with the URL to the latest release manifest of the metrics-server:

```
~$ kubectl apply -f https://raw.githubusercontent.com/RX-
M/classfiles/master/k8s-metrics-server.yaml

serviceaccount/metrics-server created
clusterrole.rbac.authorization.k8s.io/system:aggregated-metrics-reader
created
clusterrole.rbac.authorization.k8s.io/system:metrics-server created
rolebinding.rbac.authorization.k8s.io/metrics-server-auth-reader created
clusterrolebinding.rbac.authorization.k8s.io/metrics-server:system:auth-
delegator created
clusterrolebinding.rbac.authorization.k8s.io/system:metrics-server created
service/metrics-server created
deployment.apps/metrics-server created
apiservice.apiregistration.k8s.io/v1beta1.metrics.k8s.io created

~$
```

This metrics server deployment is a variation of the one served from the official Kubernetes metrics server repo. It configures the metrics server to include the `hostNetwork: true` key to enable the metrics server to use the host network of the node it is deployed on, and also includes the `--kubelet-preferred-address-types=InternalIP` and `--kubelet-insecure-tls` options on the metrics server to enable the metrics server to communicate with the Kubelets without TLS. These changes are only necessary in this lab environment - in production it is ideal to set up the metrics server to use certificates signed by the API Server.

Now check the status of the deployed resources:

```
~$ kubectl get all -n kube-system -l k8s-app=metrics-server

NAME                                  READY   STATUS    RESTARTS   AGE
pod/metrics-server-c5fbf4cb9-lr2kd    0/1     Running   0          9s

NAME                     TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
service/metrics-server   ClusterIP   10.107.229.22   <none>        443/TCP
9s

NAME                             READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/metrics-server   0/1     1            0           9s

NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/metrics-server-c5fbf4cb9    1         1         0       9s

~$
```

The metrics server resources are deployed to the kube-system namespace and bear the label `k8s-app=metrics-server`.

2. Test the Metrics Server

Metrics retrieved by the metrics server are accessible using `kubectl top`.

It may take about two or three minutes for the Metrics Server to collect data from the kubelets. Eventually you will see that top returns actual metrics. Use the Linux `watch` command (not to be confused with `kubectl –watch`) tool to continually run a `kubectl top node` to see this happen live. Once you see that the metrics are being reported, use `CTRL c` to stop watching:

```
~$ watch -t kubectl top node

NAME              CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
ip-172-31-4-161   163m         8%     1784Mi          47%

^C

~$
```

Great, our nodes are reporting metrics. Now try `kubectl top` for pods:

```
~$ kubectl top pod -n kube-system

NAME                                      CPU(cores)   MEMORY(bytes)
coredns-76f75df574-9k4tx                  2m           12Mi
coredns-76f75df574-9p29r                  2m           12Mi
etcd-ip-172-31-4-161                      20m          42Mi
kube-apiserver-ip-172-31-4-161            46m          281Mi
kube-controller-manager-ip-172-31-4-161   17m          50Mi
kube-proxy-kbdll                          1m           14Mi
kube-scheduler-ip-172-31-4-161            4m           20Mi
metrics-server-c5fbf4cb9-lr2kd            3m           15Mi
weave-net-x76wg                           1m           37Mi

~$
```

Awesome, metrics for our nodes and pods are now available!

2.1. CHALLENGE: kubectl top

Using the `kubectl top` command and its help text available from `-h` or `--help`, try the following operations:

- Retrieve the current CPU and Memory utilization of all control plane components in the `kube-system` namespace by their label
  - Sort the output by memory use
  - Which control plane component is currently using the most memory?
- Retrieve a listing of resource use of all nodes in your cluster sorted by CPU utilization without printing the column headers

3. Autoscaling

Pod auto-scaling controllers like the HorizontalPodAutoScaler use metrics collected by the metrics server. The base metrics used in most autoscaling scenarios are CPU and memory.

To test autoscaling we need:

- A target pod or container image that we can load
- A pod we can use to generate load on the target pod
- A Horizontal Pod Autoscalor (HPA) to scale the target pod's deployment in response to load

We'll be using a pre-made container that runs a simple php web server to serve as our target pod, `rxmllc/target`. This app's code computes a million square roots each time it receives a request on port 80:

```
<?php
  $x = 0.0001;
  for ($i = 0; $i <= 1000000; $i++) {
    $x += sqrt($x);
  }
  echo "OK!";
?>
```

First, create a deployment spec imperatively:

```
~$ mkdir ~/hpa && cd $_

~/hpa$ kubectl create deploy web1 --image=rxmllc/target --dry-run=client -
o yaml > target-deploy.yaml

~/hpa$
```

Then edit the deployment spec with the following flags:

- Add a `resources` section that `requests` for `200m` of `cpu`

```
~/hpa$ nano target-deploy.yaml && cat $_
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: web1
  name: web1
spec:
  replicas: 1
```

```
  selector:
    matchLabels:
      app: web1
  template:
    metadata:
      labels:
        app: web1
    spec:
      containers:
      - image: rxmllc/target
        name: target
        resources:            # Add this
          requests:           # Add this
            cpu: "200m"       # Add this
```

```
~/hpa$ kubectl apply -f target-deploy.yaml

deployment.apps/web1 created

~/hpa$
```

Now, create a service for the target deployment using `kubectl expose` so the target pods have an easily reachable DNS name:

```
~/hpa$ kubectl expose deploy web1 --port 80

service/web1 exposed

~/hpa$
```

After exposing the deployment, list your resources:

```
~/hpa$ kubectl get deploy,rs,po,svc

NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/web1   1/1     1            1           19s

NAME                              DESIRED   CURRENT   READY   AGE
replicaset.apps/web1-658f9667d6   1         1         1       19s

NAME                        READY   STATUS    RESTARTS   AGE
pod/web1-658f9667d6-w4c82   1/1     Running   0          19s

NAME                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
AGE
service/kubernetes   ClusterIP   10.96.0.1       <none>        443/TCP
103m
```

```
service/web1         ClusterIP   10.98.148.113   <none>        80/TCP
13s

~/hpa$
```

Our PHP server is now ready to generate some metrics.

Now let's create an HPA for our PHP server deployment:

```
~/hpa$ kubectl autoscale deployment web1 --cpu-percent=50 --min=1 --max=5

horizontalpodautoscaler.autoscaling/web1 autoscaled

~/hpa$
```

Our HPA will now scale up our deployment when one of its pods exceeds 50% of their desired CPU resources (100 mils). Right now we have one pod with a request for 20% of a CPU, or 200 mils.

Display the HPA with the `-w` flag, using `ctrl c` to end the watch when you see `TARGETS` reporting `0%/50%`:

```
~/hpa$ kubectl get hpa -w

NAME   REFERENCE         TARGETS         MINPODS   MAXPODS   REPLICAS
AGE
web1   Deployment/web1   <unknown>/50%   1         5         0          4s
web1   Deployment/web1   <unknown>/50%   1         5         1
15s
web1   Deployment/web1   0%/50%          1         5         1
30s

# takes a couple minutes
^C

~/hpa$
```

Now display the resources used by your pod:

```
~/hpa$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
web1-658f9667d6-w4c82   1m           9Mi

~/hpa$
```

Our pod is using 1 mil of maximum 200 mils and is below the scaling threshold defined in our HPA. It needs to get to 100 mils before the HPA will add another pod to the deployment. Let's create some load!

Run an interactive busybox pod to send requests to the PHP server:

```
~/hpa$ kubectl run -it driver --image=busybox:1.35

If you don't see a command prompt, try pressing enter.

/ #
```

Now start a loop that will make continuous requests of our PHP server:

```
/ # while true; do wget -q -O- http://web1.default.svc.cluster.local; done

OK!OK!OK!

...
```

While the requests are running, **open a new terminal** check the pod utilization with top:

```
@Laptop:~$ ssh -i key.pem 55.55.55.55

~$ cd ~/hpa

~/hpa$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
driver                  23m          0Mi
web1-658f9667d6-w4c82   778m         12Mi

~/hpa$
```

`kubectl top` shows that the web1 pod's CPU is above the threshold of 100 millicpu. But why don't we have more pods reporting metrics? The HPA will not respond to spikes because this would trash the cluster, creating and deleting pods wastefully. Rather the HPA waits for a configurable period (defined by Controller Manager settings, defaulting to about 3 minutes) and then begins to scale.

Run top several more times (or use `watch kubectl top pod`) until you see scaling activity.

```
~/hpa$ kubectl top pod

NAME                    CPU(cores)   MEMORY(bytes)
driver                  24m          0Mi
```

```
web1-658f9667d6-5dfsh   187m         12Mi
web1-658f9667d6-w4c82   182m         12Mi
web1-658f9667d6-wkm92   236m         12Mi
web1-658f9667d6-z4j24   310m         12Mi
# we are missing one!?!

~/hpa$ kubectl get hpa

NAME   REFERENCE         TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
web1   Deployment/web1   109%/50%   1         5         5          9m4s

~/hpa$
```

You will see that the HPA scaled the deployment out to the configured limit of 5, but they are not showing on `kubectl top pod`. The new pods need a few minutes to communicate with the metrics server before they appear on `kubectl top` output.

Let's confirm whether the scaling did occur in the meantime. Display your deployment:

```
~/hpa$ kubectl get deploy

NAME   READY   UP-TO-DATE   AVAILABLE   AGE
web1   4/5     5            4           10m

~/hpa$
```

The only thing the HPA does is change the replica count on the deployment. The deployment and replica set do the rest of the updates.

Depending on what your system is running, you may have all five running. Above we see 4 running. If you have issues, be sure to list the pods and describe them, checking events.

```
~/hpa$ kubectl get pods

NAME                    READY   STATUS    RESTARTS   AGE
driver                  1/1     Running   0          2m33s
web1-658f9667d6-5dfsh   1/1     Running   0          2m6s
web1-658f9667d6-msk6f   0/1     Pending   0          111s
web1-658f9667d6-w4c82   1/1     Running   0          10m
web1-658f9667d6-wkm92   1/1     Running   0          2m6s
web1-658f9667d6-z4j24   1/1     Running   0          2m6s
```

Lets grab the `Pending` pod and check its events:

```
~/hpa$ export PEND=$(kubectl get pods | grep Pending | awk '{print $1}')
&& echo $PEND
```

```
web1-75c44bd99c-qqkbq

~/hpa$ kubectl get event --field-selector involvedObject.name=$PEND

LAST SEEN   TYPE      REASON            OBJECT
MESSAGE
2m10s       Warning   FailedScheduling   pod/web1-658f9667d6-msk6f   0/1
nodes are available: 1 Insufficient cpu. preemption: 0/1 nodes are
available: 1 No preemption victims found for incoming pod.

~/hpa$
```

Not great, not terrible; yet its good to know!

Time to stop sending load to the `target` pod. Attach to the driver pod using `kubectl attach driver -c driver -i -t`, use `CTRL c` to stop the driver loop, then exit the pod:

```
~/hpa$ kubectl attach driver -c driver -i -t

If you don't see a command prompt, try pressing enter.
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!

^C

/ # exit

Session ended, resume using 'kubectl attach driver -c driver -i -t'
command when the pod is running

~/hpa$
```

Examine the metrics and HPA and you will see that activity is trending down, using `ctrl c` to end the watch:

```
~/hpa$ kubectl get hpa -w

NAME   REFERENCE         TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
web1   Deployment/web1   0%/50%    1         5         5          11m
web1   Deployment/web1   0%/50%    1         5         5          16m
web1   Deployment/web1   0%/50%    1         5         1          16m  #
takes 5 minutes to reach this point

^C

~/hpa$
```

Even though we stopped the load and resource use is trending downward, the deployment is still holding at the 5 replicas from the initial scaling. The HPA uses a setting on the kube-controller-manager called

`horizontal-pod-autoscaler-downscale-stabilization`. This argument prevents an HPA from scaling down for 5 minutes by default. After 5 minutes, the HPA will scale the deployment back down to one pod. This long tail is to reduce thrashing.

Review the HPA related events.

```
~/hpa$ kubectl get event --field-selector involvedObject.name=web1

LAST SEEN   TYPE      REASON                          OBJECT
MESSAGE
22m         Normal    ScalingReplicaSet               deployment/web1
Scaled up replica set web1-658f9667d6 to 1
22m         Warning   FailedGetResourceMetric
horizontalpodautoscaler/web1   failed to get cpu utilization: did not
receive metrics for targeted pods (pods might be unready)
22m         Warning   FailedComputeMetricsReplicas
horizontalpodautoscaler/web1   invalid metrics (1 invalid out of 1), first
error is: failed to get cpu resource metric value: failed to get cpu
utilization: did not receive metrics for targeted pods (pods might be
unready)
14m         Normal    SuccessfulRescale
horizontalpodautoscaler/web1   New size: 4; reason: cpu resource
utilization (percentage of request) above target
14m         Normal    ScalingReplicaSet               deployment/web1
Scaled up replica set web1-658f9667d6 to 4 from 1
13m         Normal    SuccessfulRescale
horizontalpodautoscaler/web1   New size: 5; reason: cpu resource
utilization (percentage of request) above target
13m         Normal    ScalingReplicaSet               deployment/web1
Scaled up replica set web1-658f9667d6 to 5 from 4
6m21s       Normal    SuccessfulRescale
horizontalpodautoscaler/web1   New size: 1; reason: All metrics below
target
6m21s       Normal    ScalingReplicaSet               deployment/web1
Scaled down replica set web1-658f9667d6 to 1 from 5

~/hpa$
```

With a working metrics server, your cluster can now dynamically create or remove additional resources based on metrics measured from incoming workloads.

3.1. CHALLENGE: HPA editing

Before you clean up, try the following operations:

- Adjust the deployment, HPA, or both so that the `web1` deployment scales when a pod reaches 50m of CPU
- Adjust the HPA so it can scale the `web1` deployment up to 3 replicas
- Adjust the HPA minimum count to 2 replicas
    - What happens to the deployment?

5. Clean up

Delete everything you created for this lab using `kubectl delete`:

```
~/hpa$ kubectl delete service/web1 deploy/web1 pod/driver hpa/web1

service "web1" deleted
deployment.apps "web1" deleted
pod "driver" deleted
horizontalpodautoscaler.autoscaling "web1" deleted

~/hpa$
```

Congratulations, you have completed the lab!