# Exercise - Scale your application to meet demand

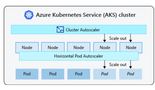
100 XP

5 minutes

Fruit Smoothies has shops worldwide with a large follower base and the expectation is that many users will use the ratings website to rate their favorite smoothy flavor. As the popularity of your application grows, the application needs to scale appropriately to manage demand changes. You have to ensure that your application remains responsive as the number of ratings increases.

In this exercise, you'll:

- Create an AKS horizontal pod autoscaler
- Run a load test with horizontal pod autoscaler enabled
- Autoscale the AKS cluster




## Create the horizontal pod autoscaler

With increased traffic, the `ratings-api` container is unable to cope with the number of requests coming through. To fix the bottleneck, you can deploy more instances of that container.

We have two options to choose from when you need to scale out container instances in AKS. You can either manually increase the number of replicas in the deployment or use the horizontal pod autoscaler.

### What is a horizontal pod autoscaler (HPA)?

The horizontal pod autoscaler (HPA) controller is Kubernetes control loop that allows the Kubernetes controller manager to query resource usage against the metrics specified in a *HorizontalPodAutoscaler* definition. The HPA controller calculates the ratio between a desired metric value specified in its definition file and the current metric value measured. The HPA automatically scales the number of pods up or down based on the calculated value.

HPA allows AKS to detect when your deployed pods need more resources based on metrics such as CPU. HPA can then schedule more pods onto the cluster to cope with the demand. You can configure HPA by using the `kubectl autoscale` command, or you can define the HPA object in a YAML file.

1. Create a file called `ratings-api-hpa.yaml` by using the integrated editor.

   bash
   ```
   code ratings-api-hpa.yaml
   ```

2. Paste the following text in the file.

   YAML
   ```
   apiVersion: autoscaling/v2beta2
   kind: HorizontalPodAutoscaler
   metadata:
     name: ratings-api
   spec:
     scaleTargetRef:
       apiVersion: apps/v1
       kind: Deployment
       name: ratings-api
     minReplicas: 1
     maxReplicas: 10
     metrics:
     - type: Resource
       resources:
         name: cpu
         target:
           type: Utilization
           averageUtilization: 30
   ```

3. Review the file, and note the following points:

   - **Scale target**

     The target for scaling is the **ratings-api** deployment.

   - **Min and max replicas**

     The minimum and maximum number of replicas to be deployed.

   - **Metrics**

     The autoscaling metric monitored is the CPU utilization, set at 30%. When the utilization goes above that level, the HPA creates more replicas.

4. To save the file, select `Ctrl+S`. To close the editor, select `Ctrl+Q`.

5. Apply the configuration by using the `kubectl apply` command. Deploy the HPA object in the `ratingsapp` namespace.

   bash
   ```
   kubectl apply \
       --namespace ratingsapp \
       -f ratings-api-hpa.yaml
   ```

   You'll see an output similar to this example.

   output
   ```
   horizontalpodautoscaler.autoscaling/ratings-api created
   ```

> **Important**
>
> For the horizontal pod autoscaler to work, you *must* remove any explicit replica count from your `ratings-api` deployment. Keep in mind that you need to redeploy your deployment when you make any changes.

## Run a load test with horizontal pod autoscaler enabled

To create the load test, you can use a prebuilt image called `acr/artillery` that's available on Docker hub. The image contains a tool called artillery that's used to send traffic to the API. Azure Container Instances can be used to run this image as a container.

When it runs as a container instance set, you don't want it to restart after it has finished. Use the `--restart-policy` parameter and set the value to `Never` to prevent the restart.

1. In Azure Cloud Shell, store the front-end API load test endpoint in a Bash variable and replace `<frontend hostname>` with your exposed ingress host name, for example, https://frontend.13-68-177-68.nip.io.

   bash
   ```
   LOADTEST_API_ENDPOINT=https://<frontend hostname>/api/loadtest
   ```

   Let's run a load test to see how the HPA scales your deployment.

2. Run the load test by using the following command, which sets the duration of the test to 120 seconds to simulate up to 500 requests per second.

   bash
   ```
   az container create \
       -g $RESOURCE_GROUP \
       -n loadtest \
       --cpu 4 \
       --memory 1 \
       --image acr/artillery \
       --restart-policy Never \
       --command-line "artillery quick -r 500 -d 120 $LOADTEST_API_ENDPOINT"
   ```

   You might need to run this command a few times.

3. Watch the horizontal pod autoscaler working.

   bash
   ```
   kubectl get hpa \
     -namespace ratingsapp -w
   ```

   In a few seconds, you'll see the HPA transition to deploying more replicas. It scales up from 1 to 10 to accommodate the load. Select `Ctrl+C` to stop watching.

   output
   ```
   NAME          REFERENCE                TARGETS    MINPODS   MAXPODS   REPLICAS   AGE
   ratings-api   Deployment/ratings-api   0%/30%     1         10        1          19m
   ratings-api   Deployment/ratings-api   46%/30%    1         10        1          20m
   ratings-api   Deployment/ratings-api   46%/30%    1         10        2          20m
   ratings-api   Deployment/ratings-api   120%/30%   1         10        2          21m
   ratings-api   Deployment/ratings-api   120%/30%   1         10        4          21m
   ratings-api   Deployment/ratings-api   93%/30%    1         10        4          22m
   ratings-api   Deployment/ratings-api   93%/30%    1         10        8          22m
   ratings-api   Deployment/ratings-api   93%/30%    1         10        10         22m
   ratings-api   Deployment/ratings-api   0%/30%     1         10        10         23m
   ```

## Autoscale the cluster

HPA scales out with new pods as required. Eventually, the cluster runs out of resources, and you'll see scheduled pods in a pending state.

### What is a cluster autoscaler?

The cluster autoscaler watches for pods that can't be scheduled on nodes because of resource constraints. The cluster then automatically increases the number of nodes in the cluster.

Let's introduce load to the cluster to force it to autoscale. We can simulate this by artificially increasing the resource `request` and `limit` for CPU in the `ratings-api` deployment to `cpu: "1000m"` and redeploy. This forces the pods to request more resources across the cluster than is actually available. We can then enable autoscaling, and increase the available nodes that are available to run pods.

1. Edit the file called `ratings-api-deployment.yaml` by using the integrated editor.

   bash
   ```
   code ratings-api-deployment.yaml
   ```

2. Change the `resources.requests` and `resources.limits` for the container to be 1000m, which means one core. The section should now look like this.

   YAML
   ```
   resources:
     requests: # minimum resources required
       cpu: 100m
       memory: 64Mi
     limits: # maximum resources allocated
       cpu: 100m
       memory: 256Mi
   ```

3. Apply the configuration by using the `kubectl apply` command. Deploy the resource update in the `ratingsapp` namespace.

   bash
   ```
   kubectl apply \
       --namespace ratingsapp \
       -f ratings-api-deployment.yaml
   ```

   You'll seen an output similar to this example.

   output
   ```
   deployment.apps/ratings-api configured
   ```

4. Review the new pods rolling out. Query for pods in the `ratingsapp` namespace, which are labeled with `app=ratings-api`.

   bash
   ```
   kubectl get pods \
       --namespace ratingsapp \
       -l app=ratings-api -w
   ```

   You'll now see multiple pods stuck in the `Pending` state because there isn't enough capacity on the cluster to schedule those new pods.

   output
   ```
   NAME                           READY   STATUS    RESTARTS   AGE
   ratings-api-7746bb6444-6k24p   0/1     Pending   0          5m42s
   ratings-api-7746bb6444-brkd8   0/1     Pending   0          5m42s
   ratings-api-7746bb6444-l7fdq   0/1     Pending   0          5m42s
   ratings-api-7746bb6444-nfb5d   0/1     Pending   0          5m42s
   ratings-api-7746bb6444-rmvb2   0/1     Pending   0          5m42s
   ratings-api-7cf598048-7wmnl    1/1     Running   0          35m
   ratings-api-7cf598048-98mwd    1/1     Running   0          12m
   ratings-api-7cf598048-c5bbq    1/1     Running   0          11m
   ratings-api-7cf598048-cmhk5    1/1     Running   0          10m
   ratings-api-7cf598048-t6ztk    1/1     Running   0          10m
   ratings-api-7cf598048-vs44s    1/1     Running   0          10m
   ratings-api-7cf598048-xxhxs    1/1     Running   0          11m
   ratings-api-7cf598048-z9klk    1/1     Running   0          10m
   ratings-mongodb-5c8757f58-k6qcd   1/1   Running   0          16d
   ratings-web-7bc649bccb-bwjfc   1/1     Running   0          99m
   ratings-web-7bc649bccb-gshn7   1/1     Running   0          99m
   ```

   To solve the pending pod problem, you can enable the cluster autoscaler to scale the cluster automatically.

5. Configure the cluster autoscaler. You should see it dynamically adding and removing nodes based on the cluster utilization. Use the `az aks update` command to enable the cluster autoscaler. Specify a minimum and maximum value for the number of nodes. Make sure to use the same resource group from earlier, for example, **aksworkshop**.

   The following example sets the `--min-count` to *3* and the `--max-count` to *5*.

   bash
   ```
   az aks update \
   --resource-group $RESOURCE_GROUP \
   --name $AKS_CLUSTER_NAME  \
   --enable-cluster-autoscaler \
   --min-count 3 \
   --max-count 5
   ```

   In a few minutes, the cluster should be configured with the cluster autoscaler. You'll see the number of nodes increase.

6. Verify the number of nodes has increased.

   bash
   ```
   kubectl get nodes -w
   ```

   In a few minutes, you'll see some new nodes popping up and transitioning to the `Ready` state. Select `Ctrl+C` to stop watching.

   output
   ```
   NAME                                STATUS   ROLES   AGE   VERSION
   aks-nodepool1-24503160-vmss000000   Ready    agent   50m   v1.15.7
   aks-nodepool1-24503160-vmss000001   Ready    agent   50m   v1.15.7
   aks-nodepool1-24503160-vmss000002   Ready    agent   50m   v1.15.7
   aks-nodepool1-24503160-vmss000003   Ready    agent   14s   v1.15.7
   aks-nodepool1-24503160-vmss000004   Ready    agent   21s   v1.15.7
   ```

## Summary

In this exercise you created a horizontal pod autoscaler and ran a load test to scale out the pods on your cluster. You then increased the compute capacity of your cluster through the cluster autoscaler, adding nodes to your AKS cluster. You now have the knowledge to ensure the Fruit Smoothies AKS environment can scale in response to fluctuations in user traffic.

Let's next wrap up what you've learned here.

**Next unit: Summary and cleanup**

Continue >