

Building scalable applications

When running an application efficiently, the ability to scale and upgrade your application is critical. Scaling allows your application to handle additional load. While upgrading, scaling is needed to keep your application up to date and to introduce new functionality.

Scaling on demand is one of the key benefits of using cloud-native applications. It also helps optimize resources for your application. If the front end component encounters heavy load, you can scale the front end alone, while keeping the same number of back end instances. You can increase or reduce the number of **virtual machines (VMs)** required depending on your workload and peak demand hours. This chapter will cover the scale dimensions of the application and its infrastructure in detail.

In this chapter, you will learn how to scale the sample guestbook application that was introduced in *Chapter 3, Application deployment on AKS*. You will first scale this application using manual commands, and afterward you'll learn how to autoscale it using the **Horizontal Pod Autoscaler (HPA)**. The goal is to make you comfortable with `kubectl`, which is an important tool for managing applications running on top of **Azure Kubernetes Service (AKS)**. After scaling the application, you will also scale the cluster. You will first scale the cluster manually, and then use the **cluster autoscaler** to automatically scale the cluster. In addition, you will get a brief introduction on how you can upgrade applications running on top of AKS.

In this chapter, we will cover the following topics:

- Scaling your application
- Scaling your cluster
- Upgrading your application

Let's begin this chapter by discussing the different dimensions of scaling applications on top of AKS.

Scaling your application

There are two scale dimensions for applications running on top of AKS. The first scale dimension is the number of pods a deployment has, while the second scale dimension in AKS is the number of nodes in the cluster.

By adding new pods to a deployment, also known as scaling out, you can add additional compute power to the deployed application. You can either scale out your applications manually or have Kubernetes take care of this automatically via HPA. HPA can monitor metrics such as the CPU to determine whether pods need to be added to your deployment.

The second scale dimension in AKS is the number of nodes in the cluster. The number of nodes in a cluster defines how much CPU and memory are available for all the applications running on that cluster. You can scale your cluster manually by changing the number of nodes, or you can use the cluster autoscaler to automatically scale out your cluster. The cluster autoscaler watches the cluster

for pods that cannot be scheduled due to resource constraints. If pods cannot be scheduled, it will add nodes to the cluster to ensure that your applications can run.

Both scale dimensions will be covered in this chapter. In this section, you will learn how you can scale your application. First, you will scale your application manually, and then later, you will scale your application automatically.

Manually scaling your application

To demonstrate manual scaling, let's use the guestbook example that we used in the previous chapter. Follow these steps to learn how to implement manual scaling:

Note

In the previous chapter, we cloned the example files in Cloud Shell. If you didn't do this back then, we recommend doing that now:

```
git clone https://github.com/PacktPublishing/Hands-On-Kubernetes-on-Azure-third-edition
```

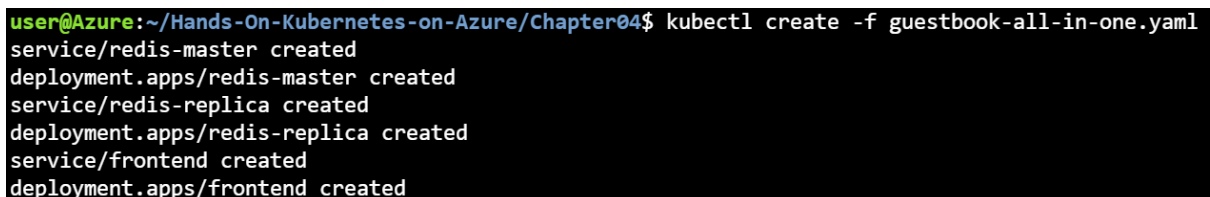
For this chapter, navigate to the Chapter04 directory:

```
cd Chapter04
```

1. Set up the guestbook by running the `kubectl create` command in the Azure command line:

```
kubectl create -f guestbook-all-in-one.yaml
```

2. After you have entered the preceding command, you should see something similar to what is shown in *Figure 4.1* in your command-line output:

A terminal window with a black background and white text. The prompt is 'user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04\$'. The command entered is 'kubectl create -f guestbook-all-in-one.yaml'. The output shows six resources created: 'service/redis-master created', 'deployment.apps/redis-master created', 'service/redis-replica created', 'deployment.apps/redis-replica created', 'service/frontend created', and 'deployment.apps/frontend created'.

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl create -f guestbook-all-in-one.yaml
service/redis-master created
deployment.apps/redis-master created
service/redis-replica created
deployment.apps/redis-replica created
service/frontend created
deployment.apps/frontend created
```

Figure 4.1: Launching the guestbook application

3. Right now, none of the services are publicly accessible. We can verify this by running the following command:

```
kubectl get service
```

4. As seen in *Figure 4.2*, none of the services have an external IP:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get service
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|---------------|-----------|--------------|-------------|----------|-------|
| frontend | ClusterIP | 10.0.118.101 | <none> | 80/TCP | 76s |
| kubernetes | ClusterIP | 10.0.0.1 | <none> | 443/TCP | 3d22h |
| redis-master | ClusterIP | 10.0.181.124 | <none> | 6379/TCP | 77s |
| redis-replica | ClusterIP | 10.0.138.136 | <none> | 6379/TCP | 77s |

Figure 4.2: Output confirming that none of the services have a public IP

5. To test the application, you will need to expose it publicly. For this, let's introduce a new command that will allow you to edit the service in Kubernetes without having to change the file on your file system. To start the edit, execute the following command:

```
kubectl edit service frontend
```

6. This will open a vi environment. Use the down arrow key to navigate to the line that says type: ClusterIP and change that to type: LoadBalancer, as shown in *Figure 4.3*. To make that change, hit the *I* button, change type to LoadBalancer, hit the *Esc* button, type *:wq!*, and then hit *Enter* to save the changes:

```
spec:
  clusterIP: 10.0.118.101
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: guestbook
    tier: frontend
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer: {}
```

Figure 4.3: Changing this line to type: LoadBalancer

- Once the changes are saved, you can watch the service object until the public IP becomes available. To do this, type the following:

```
kubectl get service -w
```

- It will take a couple of minutes to show you the updated IP. Once you see the correct public IP, you can exit the watch command by hitting `Ctrl + C`:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get service -w
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|---------------|--------------|--------------|---------------|--------------|-------|
| frontend | LoadBalancer | 10.0.118.101 | <pending> | 80:30009/TCP | 3m55s |
| kubernetes | ClusterIP | 10.0.0.1 | <none> | 443/TCP | 3d22h |
| redis-master | ClusterIP | 10.0.181.124 | <none> | 6379/TCP | 3m56s |
| redis-replica | ClusterIP | 10.0.138.136 | <none> | 6379/TCP | 3m56s |
| frontend | LoadBalancer | 10.0.118.101 | 52.149.17.246 | 80:30009/TCP | 4m7s |

Figure 4.4: Output showing the front-end service getting a public IP

- Type the IP address from the preceding output into your browser navigation bar as follows: `http://<EXTERNAL-IP>/`. The result of this is shown in *Figure 4.5*:

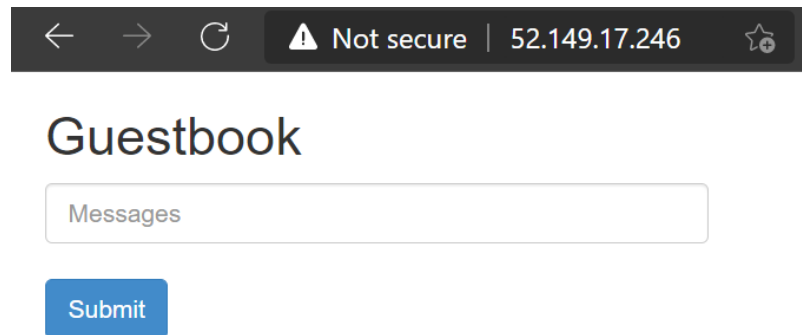


Figure 4.5: Browse to the guestbook application

The familiar guestbook sample should be visible. This shows that you have successfully publicly accessed the guestbook.

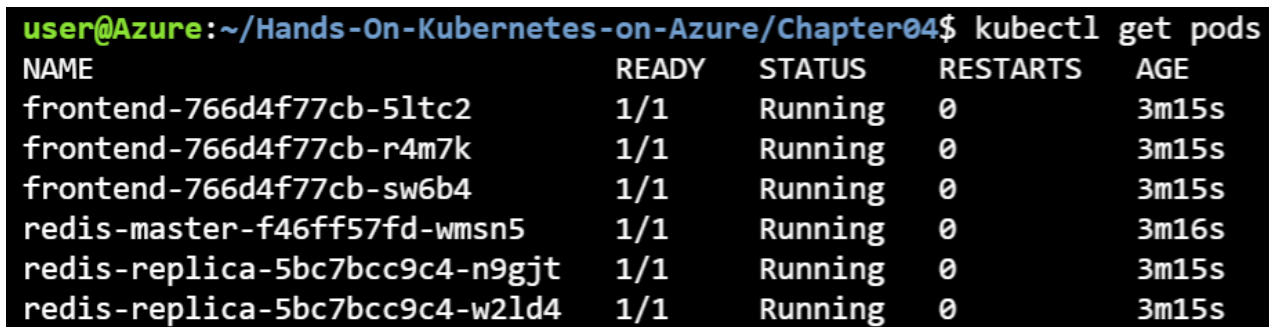
Now that you have the guestbook application deployed, you can start scaling the different components of the application.

Scaling the guestbook front-end component

Kubernetes gives us the ability to scale each component of an application dynamically. In this section, we will show you how to scale the front end of the guestbook application. Right now, the front-end deployment is deployed with three replicas. You can confirm by using the following command:

```
kubectl get pods
```

This should return an output as shown in *Figure 4.6*:



```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-5ltc2           1/1     Running   0          3m15s
frontend-766d4f77cb-r4m7k           1/1     Running   0          3m15s
frontend-766d4f77cb-sw6b4           1/1     Running   0          3m15s
redis-master-f46ff57fd-wmsn5        1/1     Running   0          3m16s
redis-replica-5bc7bcc9c4-n9gjt      1/1     Running   0          3m15s
redis-replica-5bc7bcc9c4-w2ld4      1/1     Running   0          3m15s
```

Figure 4.6: Confirming the three replicas in the front-end deployment

To scale the front-end deployment, you can execute the following command:

```
kubectl scale deployment/frontend --replicas=6
```

This will cause Kubernetes to add additional pods to the deployment. You can set the number of replicas you want, and Kubernetes takes care of the rest. You can even scale it down to zero (one of the tricks used to reload the configuration when the application doesn't support the dynamic reload of configuration). To verify that the overall scaling worked correctly, you can use the following command:

```
kubectl get pods
```


This should give you the output shown in Figure 4.7:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------------------------|-------|---------|----------|-------|
| frontend-766d4f77cb-5ltc2 | 1/1 | Running | 0 | 3m57s |
| frontend-766d4f77cb-6xwvz | 1/1 | Running | 0 | 5s |
| frontend-766d4f77cb-gmd5p | 1/1 | Running | 0 | 5s |
| frontend-766d4f77cb-r4m7k | 1/1 | Running | 0 | 3m57s |
| frontend-766d4f77cb-sw6b4 | 1/1 | Running | 0 | 3m57s |
| frontend-766d4f77cb-vz726 | 1/1 | Running | 0 | 5s |
| redis-master-f46ff57fd-wmsn5 | 1/1 | Running | 0 | 3m58s |
| redis-replica-5bc7bcc9c4-n9gjt | 1/1 | Running | 0 | 3m57s |
| redis-replica-5bc7bcc9c4-w2ld4 | 1/1 | Running | 0 | 3m57s |

Figure 4.7: Different pods running in the guestbook application after scaling out

As you can see, the front-end service scaled to six pods. Kubernetes also spread these pods across multiple nodes in the cluster. You can see the nodes that this is running on with the following command:

```
kubectl get pods -o wide
```

This will generate the following output:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods -o wide
```

| NAME | READY | STATUS | RESTARTS | AGE | IP | NODE |
|--------------------------------|-------|---------|----------|-------|-------------|-----------------------------------|
| frontend-766d4f77cb-5ltc2 | 1/1 | Running | 0 | 4m22s | 10.244.1.6 | aks-agentpool-39838025-vmss000001 |
| frontend-766d4f77cb-6xwvz | 1/1 | Running | 0 | 30s | 10.244.1.8 | aks-agentpool-39838025-vmss000001 |
| frontend-766d4f77cb-gmd5p | 1/1 | Running | 0 | 30s | 10.244.0.11 | aks-agentpool-39838025-vmss000000 |
| frontend-766d4f77cb-r4m7k | 1/1 | Running | 0 | 4m22s | 10.244.0.8 | aks-agentpool-39838025-vmss000000 |
| frontend-766d4f77cb-sw6b4 | 1/1 | Running | 0 | 4m22s | 10.244.1.7 | aks-agentpool-39838025-vmss000001 |
| frontend-766d4f77cb-vz726 | 1/1 | Running | 0 | 30s | 10.244.0.10 | aks-agentpool-39838025-vmss000000 |
| redis-master-f46ff57fd-wmsn5 | 1/1 | Running | 0 | 4m23s | 10.244.1.4 | aks-agentpool-39838025-vmss000001 |
| redis-replica-5bc7bcc9c4-n9gjt | 1/1 | Running | 0 | 4m22s | 10.244.1.5 | aks-agentpool-39838025-vmss000001 |
| redis-replica-5bc7bcc9c4-w2ld4 | 1/1 | Running | 0 | 4m22s | 10.244.0.9 | aks-agentpool-39838025-vmss000000 |

Figure 4.8: Showing which nodes the pods are running on

In this section, you have seen how easy it is to scale pods with Kubernetes. This capability provides a very powerful tool for you to not only dynamically adjust your application components but also provide resilient applications with failover capabilities enabled by running multiple instances of components at the same time. However, you won't always want to manually scale your application. In the next section, you will learn how you can automatically scale your application in and out by automatically adding and removing pods in a deployment.

Using the HPA

Scaling manually is useful when you're working on your cluster. For example, if you know your load is going to increase, you can manually scale out your application. In most cases, however, you will want some sort of autoscaling to happen on your application. In Kubernetes, you can configure autoscaling of your deployment using an object called the **Horizontal Pod Autoscaler (HPA)**.

HPA monitors Kubernetes metrics at regular intervals and, based on the rules you define, it automatically scales your deployment. For example, you can configure the HPA to add additional pods to your deployment once the CPU utilization of your application is above 50%.

In this section, you will configure the HPA to scale the front-end of the application automatically:

1. To start the configuration, let's first manually scale down our deployment to one instance:

```
kubectl scale deployment/frontend --replicas=1
```

2. Next up, we'll create an HPA. Open up the code editor in Cloud Shell by typing `code hpa.yaml` and enter the following code:

```
1  apiVersion: autoscaling/v1
2  kind: HorizontalPodAutoscaler
3  metadata:
4    name: frontend-scaler
5  spec:
6    scaleTargetRef:
7      apiVersion: apps/v1
8      kind: Deployment
9      name: frontend
10   minReplicas: 1
11   maxReplicas: 10
12   targetCPUUtilizationPercentage: 50
```

Let's investigate what is configured in this file:

- **Line 2:** Here, we define that we need HorizontalPodAutoscaler.
- **Lines 6-9:** These lines define the deployment that we want to autoscale.
- **Lines 10-11:** Here, we configure the minimum and maximum pods in our deployment.
- **Lines 12:** Here, we define the target CPU utilization percentage for our deployment.

3. Save this file, and create the HPA using the following command:

```
kubectl create -f hpa.yaml
```

This will create our autoscaler. You can see your autoscaler with the following command:

```
kubectl get hpa
```

This will initially output something as shown in *Figure 4.9*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get hpa
```

| NAME | REFERENCE | TARGETS | MINPODS | MAXPODS | REPLICAS | AGE |
|-----------------|---------------------|---------------|---------|---------|----------|-----|
| frontend-scaler | Deployment/frontend | <unknown>/50% | 1 | 10 | 0 | 2s |

Figure 4.9: The target unknown shows that the HPA isn't ready yet

It takes a couple of seconds for the HPA to read the metrics. Wait for the return from the HPA to look something similar to the output shown in *Figure 4.10*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get hpa -w
```

| NAME | REFERENCE | TARGETS | MINPODS | MAXPODS | REPLICAS | AGE |
|-----------------|---------------------|---------------|---------|---------|----------|-----|
| frontend-scaler | Deployment/frontend | <unknown>/50% | 1 | 10 | 0 | 1s |
| frontend-scaler | Deployment/frontend | <unknown>/50% | 1 | 10 | 1 | 15s |
| frontend-scaler | Deployment/frontend | 10%/50% | 1 | 10 | 1 | 31s |

Figure 4.10: Once the target shows a percentage, the HPA is ready

4. You will now go ahead and do two things: first, you will watch the pods to see whether new pods are created. Then, you will create a new shell, and create some load for the system. Let's start with the first task—watching our pods:

```
kubectl get pods -w
```

This will continuously monitor the pods that get created or terminated.

Let's now create some load in a new shell. In Cloud Shell, hit the **open new session** icon to open a new shell:



Figure 4.11: Use this button to open a new Cloud Shell

This will open a new tab in your browser with a new session in Cloud Shell. You will generate load for the application from this tab.

5. Next, you will use a program called `hey` to generate this load. `hey` is a tiny program that sends loads to a web application. You can install and run `hey` using the following commands:

```
export GOPATH=~/.go
export PATH=$GOPATH/bin:$PATH
go get -u github.com/rakyll/hey
hey -z 20m http://<external-ip>
```

The `hey` program will now try to create up to 20 million connections to the front-end. This will generate CPU loads on the system, which will trigger the HPA to start scaling the deployment. It will take a couple of minutes for this to trigger a scale action, but at a certain point, you should see multiple pods being created to handle the additional load, as shown in *Figure 4.12*:

```

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods -w
NAME                                READY   STATUS    RESTARTS   AGE
frontend-766d4f77cb-r4m7k          1/1     Running   0           5m40s
redis-master-f46ff57fd-wmsn5       1/1     Running   0           5m41s
redis-replica-5bc7bcc9c4-n9gjt     1/1     Running   0           5m40s
redis-replica-5bc7bcc9c4-w2ld4     1/1     Running   0           5m40s
frontend-766d4f77cb-kvd24          0/1     Pending   0           0s
frontend-766d4f77cb-kvd24          0/1     Pending   0           0s
frontend-766d4f77cb-25bjj          0/1     Pending   0           0s
frontend-766d4f77cb-z855p          0/1     Pending   0           0s
frontend-766d4f77cb-z855p          0/1     Pending   0           0s
frontend-766d4f77cb-25bjj          0/1     Pending   0           0s
frontend-766d4f77cb-z855p          0/1     ContainerCreating   0           0s
frontend-766d4f77cb-kvd24          0/1     ContainerCreating   0           0s
frontend-766d4f77cb-25bjj          0/1     ContainerCreating   0           0s
frontend-766d4f77cb-z855p          1/1     Running   0           1s
frontend-766d4f77cb-25bjj          1/1     Running   0           1s
frontend-766d4f77cb-kvd24          1/1     Running   0           2s

```

Figure 4.12: New pods get started by the HPA

At this point, you can go ahead and kill the hey program by hitting Ctrl + C.

- Let's have a closer look at what the HPA did by running the following command:

```
kubectl describe hpa
```

We can see a few interesting points in the describe operation, as shown in Figure 4.13:

```

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl describe hpa
Name:                                frontend-scaler
Namespace:                           default
Labels:                               <none>
Annotations:                          <none>
CreationTimestamp:                    Wed, 20 Jan 2021 01:10:58 +0000
Reference:                            Deployment/frontend
Metrics:                              ( current / target )
  resource cpu on pods  (as a percentage of request): 384% (38m) / 50% 1
Min replicas:                          1
Max replicas:                         10
Deployment pods:                       10 current / 10 desired
Conditions:
  Type            Status  Reason                        Message
  ----            -
  AbleToScale     True    ReadyForNewScale             recommended size matches current size
  ScalingActive   True    ValidMetricFound             the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
  ScalingLimited  True    TooManyReplicas 2           the desired replica count is more than the maximum replica count
Events:
  Type    Reason                      Age    From                      Message
  ----    -
  Warning  FailedGetResourceMetric      18m (x3 over 18m)  horizontal-pod-autoscaler  unable to get metrics for resource cpu: no metrics returned from resource metrics API
  Warning  FailedComputeMetricsReplicas 18m (x3 over 18m)  horizontal-pod-autoscaler  invalid metrics (1 invalid out of 1), first error is: failed to get cpu utilization:
  unable to get metrics for resource cpu: no metrics returned from resource metrics API
  Normal  SuccessfulRescale           13m                      horizontal-pod-autoscaler  New size: 1, 3 reason: All metrics below target
  Normal  SuccessfulRescale           11m                      horizontal-pod-autoscaler  New size: 4; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale           11m                      horizontal-pod-autoscaler  New size: 8; reason: cpu resource utilization (percentage of request) above target
  Normal  SuccessfulRescale           11m                      horizontal-pod-autoscaler  New size: 10; reason: cpu resource utilization (percentage of request) above target

```

Figure 4.13: Detailed view of the HPA

Scaling your cluster

In the previous section, you dealt with scaling the application running on top of a cluster. In this section, you'll learn how you can scale the actual cluster you are running. First, you will manually scale your cluster to one node. Then, you'll configure the cluster autoscaler. The cluster autoscaler will monitor your cluster and scale out when there are pods that cannot be scheduled on the cluster.

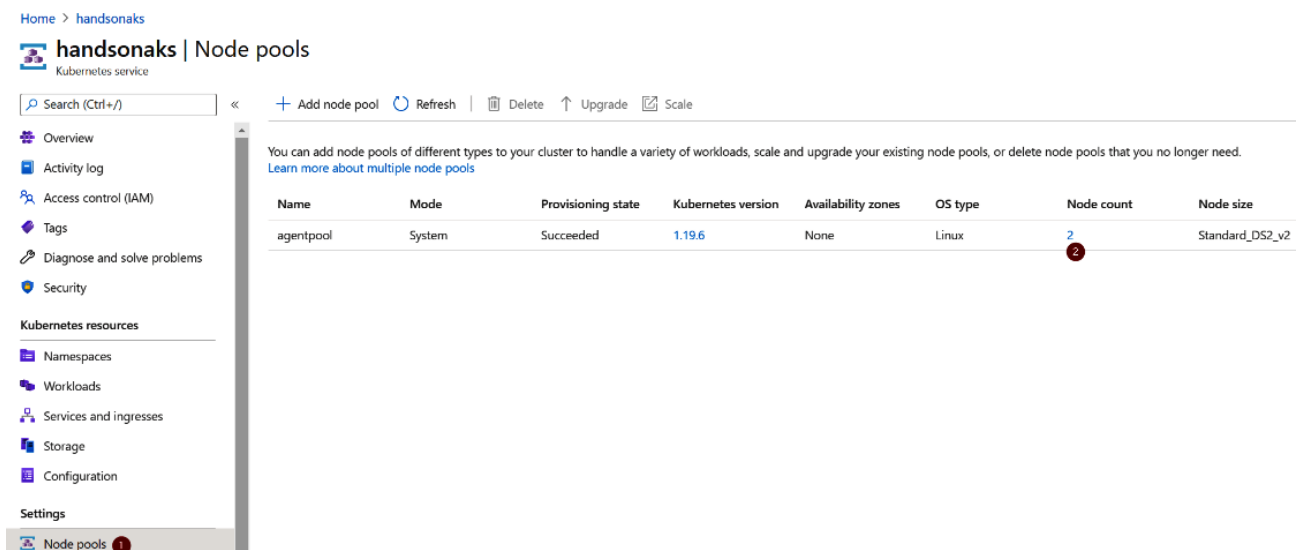
Manually scaling your cluster

You can manually scale your AKS cluster by setting a static number of nodes for the cluster. The scaling of your cluster can be done either via the Azure portal or the command line.

In this section, you'll learn how you can manually scale your cluster by scaling it down to one node. This will cause Azure to remove one of the nodes from your cluster. First, the workload on the node that is about to be removed will be rescheduled onto the other node. Once the workload is safely rescheduled, the node will be removed from your cluster, and then the VM will be deleted from Azure.

To scale your cluster, follow these steps:

1. Open the Azure portal and go to your cluster. Once there, go to **Node pools** and click on the number below **Node count**, as shown in *Figure 4.15*:



Home > handsonaks

handsonaks | Node pools
Kubernetes service

Search (Ctrl+J) << + Add node pool Refresh Delete Upgrade Scale

You can add node pools of different types to your cluster to handle a variety of workloads, scale and upgrade your existing node pools, or delete node pools that you no longer need.
[Learn more about multiple node pools](#)

| Name | Mode | Provisioning state | Kubernetes version | Availability zones | OS type | Node count | Node size |
|-----------|--------|--------------------|--------------------|--------------------|---------|------------|-----------------|
| agentpool | System | Succeeded | 1.19.6 | None | Linux | 2 | Standard_DS2_v2 |

Kubernetes resources

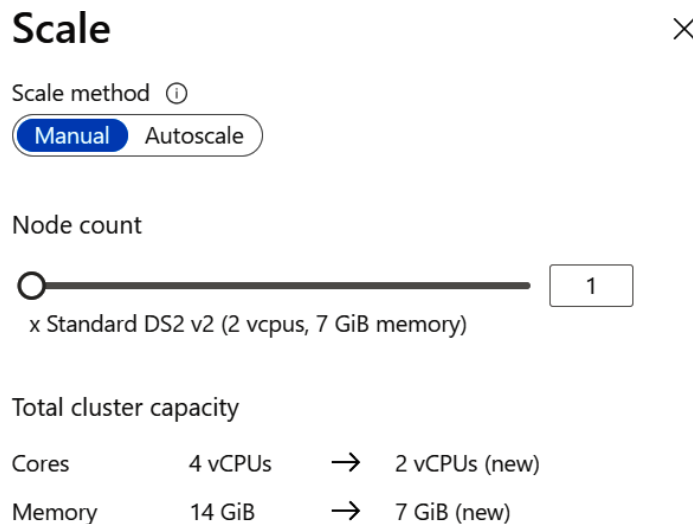
- Namespaces
- Workloads
- Services and ingresses
- Storage
- Configuration

Settings

- Node pools

Figure 4.15: Manually scaling the cluster

- This will open a pop-up window that will give the option to scale your cluster. For our example, we will scale down our cluster to one node, as shown in Figure 4.16:



The image shows a 'Scale' pop-up window with a close button (X) in the top right. It features a 'Scale method' section with 'Manual' and 'Autoscale' buttons, where 'Manual' is selected. Below this is a 'Node count' section with a slider and a text box showing '1'. The text 'x Standard DS2 v2 (2 vcpus, 7 GiB memory)' is displayed below the slider. At the bottom, there is a 'Total cluster capacity' section with a table showing the transition from 4 vCPUs to 2 vCPUs (new) and 14 GiB to 7 GiB (new).

| Total cluster capacity | | | |
|------------------------|---------|---|---------------|
| Cores | 4 vCPUs | → | 2 vCPUs (new) |
| Memory | 14 GiB | → | 7 GiB (new) |

Figure 4.16: Pop-up window confirming the new cluster size

- Hit the **Apply** button at the bottom of the screen to save these settings. This will cause Azure to remove a node from your cluster. This process will take about 5 minutes to complete. You can follow the progress by clicking on the notification icon at the top of the Azure portal as follows:

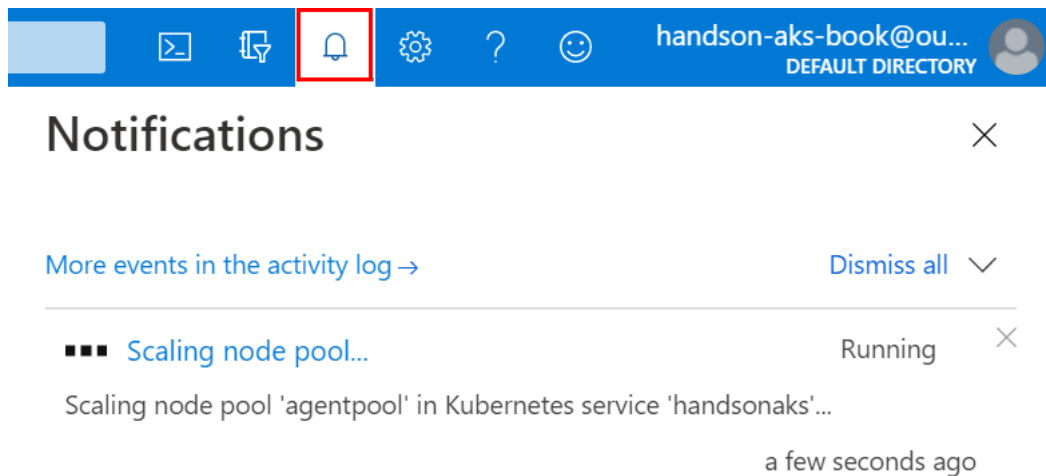


Figure 4.17: Cluster scaling can be followed using the notifications in the Azure portal

Once this scale-down operation has completed, relaunch the guestbook application on this small cluster:

```
kubectl create -f guestbook-all-in-one.yaml
```

In the next section, you will scale out the guestbook so that it can no longer run on this small cluster. You will then configure the cluster autoscaler to scale out the cluster.

Scaling your cluster using the cluster autoscaler

In this section, you will explore the cluster autoscaler. The cluster autoscaler will monitor the deployments in your cluster and scale your cluster to meet your application requirements. The cluster autoscaler watches the number of pods in your cluster that cannot be scheduled due to insufficient resources. You will first force your deployment to have pods that cannot be scheduled, and then configure the cluster autoscaler to automatically scale your cluster.

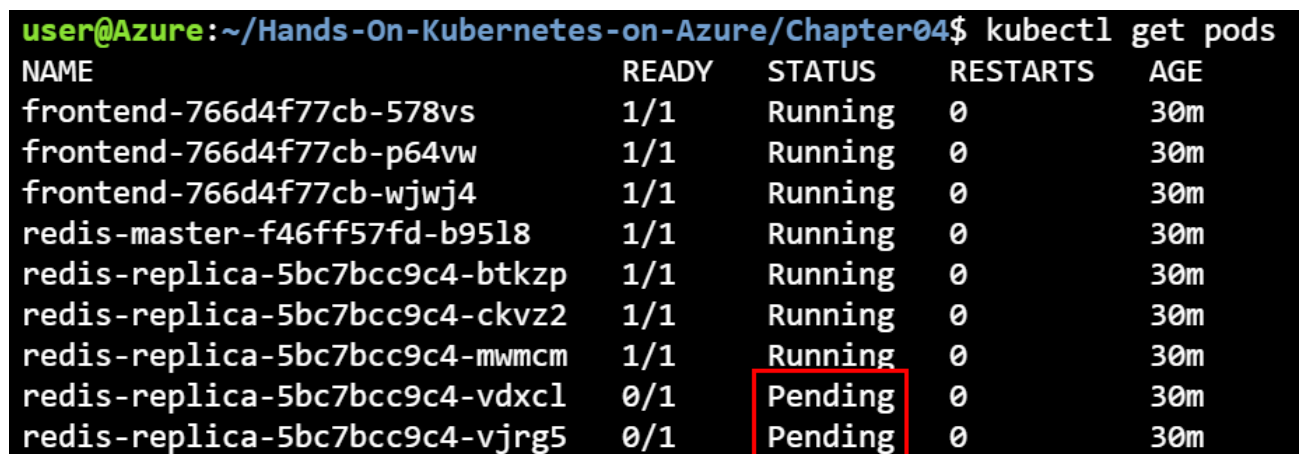
To force your cluster to be out of resources, you will—manually—scale out the `redis-replica` deployment. To do this, use the following command:

```
kubectl scale deployment redis-replica --replicas 5
```

You can verify that this command was successful by looking at the pods in our cluster:

```
kubectl get pods
```

This should show you something similar to the output shown in *Figure 4.18*:



| NAME | READY | STATUS | RESTARTS | AGE |
|--------------------------------|-------|---------|----------|-----|
| frontend-766d4f77cb-578vs | 1/1 | Running | 0 | 30m |
| frontend-766d4f77cb-p64vw | 1/1 | Running | 0 | 30m |
| frontend-766d4f77cb-wjwj4 | 1/1 | Running | 0 | 30m |
| redis-master-f46ff57fd-b95l8 | 1/1 | Running | 0 | 30m |
| redis-replica-5bc7bcc9c4-btkzp | 1/1 | Running | 0 | 30m |
| redis-replica-5bc7bcc9c4-ckvz2 | 1/1 | Running | 0 | 30m |
| redis-replica-5bc7bcc9c4-mwmcm | 1/1 | Running | 0 | 30m |
| redis-replica-5bc7bcc9c4-vdxc1 | 0/1 | Pending | 0 | 30m |
| redis-replica-5bc7bcc9c4-vjrg5 | 0/1 | Pending | 0 | 30m |

Figure 4.18: Four out of five pods are pending, meaning they cannot be scheduled

As you can see, you now have two pods in a Pending state. The Pending state in Kubernetes means that that pod cannot be scheduled onto a node. In this case, this is due to the cluster being out of resources.

Note

If your cluster is running on a larger VM size than the DS2v2, you might not notice pods in a Pending state now. In that case, increase the number of replicas to a higher number until you see pods in a pending state.

You will now configure the cluster autoscaler to automatically scale the cluster. Similar to manual scaling in the previous section, there are two ways you can configure the cluster autoscaler. You can configure it either via the Azure portal—similar to how we did the manual scaling—or you can configure it using the **command-line interface (CLI)**. In this example, you will use CLI to enable the cluster autoscaler. The following command will configure the cluster autoscaler for your cluster:

```
az aks nodepool update --enable-cluster-autoscaler \
  -g rg-handsonaks --cluster-name handsonaks \
  --name agentpool --min-count 1 --max-count 2
```

This command configures the cluster autoscaler on the node pool you have in the cluster. It configures it to have a minimum of one node and a maximum of two nodes. This will take a couple of minutes to configure.

Once the cluster autoscaler is configured, you can see it in action by using the following command to watch the number of nodes in the cluster:

```
kubectl get nodes -w
```

It will take about 5 minutes for the new node to show up and become Ready in the cluster. Once the new node is Ready, you can stop watching the nodes by hitting **Ctrl + C**. You should see an output similar to what you see in *Figure 4.19*:

```

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get nodes -w
NAME                                STATUS    ROLES    AGE   VERSION
aks-agentpool-39838025-vmss000000 Ready     agent    58m   v1.19.6
aks-agentpool-39838025-vmss000002 NotReady  <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002 NotReady  <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002 NotReady  <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002 NotReady  <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002 NotReady  <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002 NotReady  <none>   0s    v1.19.6
aks-agentpool-39838025-vmss000002 Ready     <none>   10s   v1.19.6
aks-agentpool-39838025-vmss000002 Ready     <none>   10s   v1.19.6
aks-agentpool-39838025-vmss000002 Ready     <none>   10s   v1.19.6
aks-agentpool-39838025-vmss000002 Ready     <none>   11s   v1.19.6
aks-agentpool-39838025-vmss000002 Ready     <none>   30s   v1.19.6
aks-agentpool-39838025-vmss000002 Ready     <none>   43s   v1.19.6
aks-agentpool-39838025-vmss000002 Ready     agent    46s   v1.19.6

```

Figure 4.19: The new node joins the cluster

The new node should ensure that your cluster has sufficient resources to schedule the scaled-out redis- replica deployment. To verify this, run the following command to check the status of the pods:

```
kubectl get pods
```

This should show you all the pods in a Running state as follows:

```

user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get pods
NAME                                READY    STATUS    RESTARTS   AGE
frontend-766d4f77cb-578vs           1/1      Running   0           37m
frontend-766d4f77cb-p64vw           1/1      Running   0           37m
frontend-766d4f77cb-wjwj4           1/1      Running   0           37m
redis-master-f46ff57fd-b95l8        1/1      Running   0           37m
redis-replica-5bc7bcc9c4-btkzp       1/1      Running   0           37m
redis-replica-5bc7bcc9c4-ckvz2       1/1      Running   0           37m
redis-replica-5bc7bcc9c4-mwmcm       1/1      Running   0           37m
redis-replica-5bc7bcc9c4-vdxc1       1/1      Running   0           37m
redis-replica-5bc7bcc9c4-vjrg5       1/1      Running   0           37m

```

Figure 4.20: All pods are now in a Running state

Now clean up the resources you created, disable the cluster autoscaler, and ensure that your cluster has two nodes for the next example. To do this, use the following commands:

```
kubectl delete -f guestbook-all-in-one.yaml
az aks nodepool update --disable-cluster-autoscaler \
  -g rg-handsonaks --cluster-name handsonaks --name agentpool
az aks nodepool scale --node-count 2 -g rg-handsonaks \
  --cluster-name handsonaks --name agentpool
```

Note

The last command from the previous example will show you an error message, The new node count is the same as the current node count ., if the cluster already has two nodes. You can safely ignore this error.

In this section, you first manually scaled down your cluster and then used the cluster autoscaler to scale out your cluster. You used the Azure portal to scale down the cluster manually and then used the Azure CLI to configure the cluster autoscaler. In the next section, you will look into how you can upgrade applications running on AKS.

Upgrading your application

Using deployments in Kubernetes makes upgrading an application a straightforward operation. As with any upgrade, you should have good fallbacks in case something goes wrong. Most of the issues you will run into will happen during upgrades. Cloud-native applications are supposed to make dealing with this relatively easy, which is possible if you have a very strong development team that embraces DevOps principles.

The State of DevOps report (<https://puppet.com/resources/report/2020-state-of-devops-report/>) has reported for multiple years that companies that have high software deployment frequency rates have higher availability and stability in their applications as well. This might seem counterintuitive, as doing software deployments heightens the risk of issues. However, by deploying more frequently and deploying using automated DevOps practices, you can limit the impact of software deployment.

There are multiple ways you can make updates to applications running in a Kubernetes cluster. In this section, you will explore the following ways to update Kubernetes resources:

- Upgrading by changing YAML files: This method is useful when you have access to the full YAML file required to make the update. This can be done either from your command line or from an automated system.
- Upgrading using `kubectl edit`: This method is mostly used for minor changes on a cluster. It is a quick way to update your configuration live on a cluster.
- Upgrading using `kubectl patch`: This method is useful when you need to script a particular small update to a Kubernetes but don't have access to the full YAML file. It can be done either from a command line or an automated system. If you have access to the original YAML files, it is typically better to edit the YAML file and use `kubectl apply` to apply the updates.
- Upgrading using Helm: This method is used when your application is deployed through Helm.

The methods described in the following sections work great if you have stateless applications. If you have a state stored anywhere, make sure to back up that state before you try upgrading your application.

Let's start this section by doing the first type of upgrade by changing YAML files.

Upgrading by changing YAML files

In order to upgrade a Kubernetes service or deployment, you can update the actual YAML definition file and apply that to the currently deployed application. Typically, we use `kubectl create` to create resources. Similarly, we can use `kubectl apply` to make changes to the resources.

The deployment detects the changes (if any) and matches the running state to the desired state. Let's see how this is done:

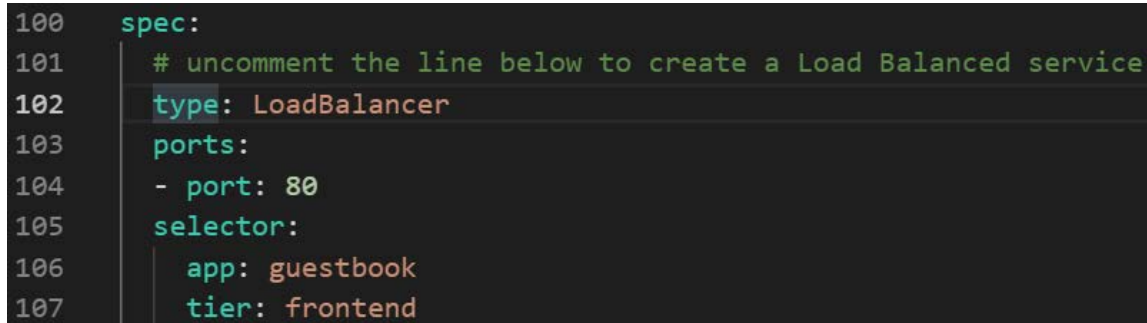
1. Start with our guestbook application to explore this example:

```
kubectl apply -f guestbook-all-in-one.yaml
```

2. After a few minutes, all the pods should be running. Let's perform the first upgrade by changing the service from ClusterIP to LoadBalancer, as you did earlier in the chapter. However, now you will edit the YAML file rather than using `kubectl edit`. Edit the YAML file using the following command:

```
code guestbook-all-in-one.yaml
```

Uncomment line 102 in this file to set the type to LoadBalancer, and save the file, as shown in *Figure 4.21*:



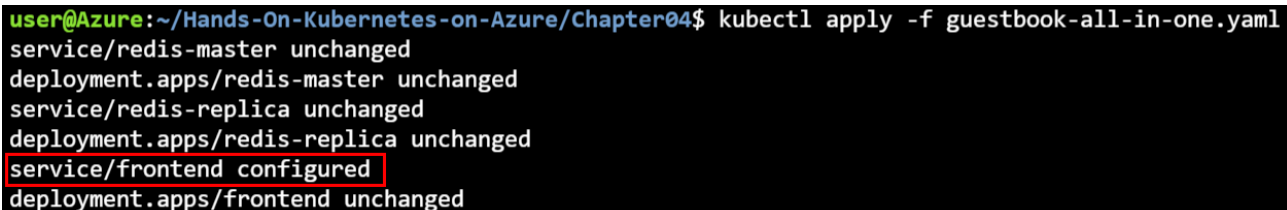
```
100 spec:
101   # uncomment the line below to create a Load Balanced service
102   type: LoadBalancer
103   ports:
104   - port: 80
105   selector:
106     app: guestbook
107     tier: frontend
```

Figure 4.21: Setting the type to LoadBalancer in the guestbook-all-in-one YAML file

3. Apply the change as shown in the following code:

```
kubectl apply -f guestbook-all-in-one.yaml
```

You should see an output similar to *Figure 4.22*:



```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl apply -f guestbook-all-in-one.yaml
service/redis-master unchanged
deployment.apps/redis-master unchanged
service/redis-replica unchanged
deployment.apps/redis-replica unchanged
service/frontend configured
deployment.apps/frontend unchanged
```

Figure 4.22: The service's front-end is updated

As you can see in *Figure 4.22*, only the object that was updated in the YAML file, which is the service in this case, was updated on Kubernetes, and the other objects remained unchanged.

4. You can now get the public IP of the service using the following command:

```
kubectl get service
```

Give it a few minutes, and you should be shown the IP, as displayed in *Figure 4.23*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get service
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|---------------|--------------|-------------|--------------|--------------|-------|
| frontend | LoadBalancer | 10.0.119.74 | 40.64.105.32 | 80:32287/TCP | 2m43s |
| kubernetes | ClusterIP | 10.0.0.1 | <none> | 443/TCP | 4d7h |
| redis-master | ClusterIP | 10.0.75.94 | <none> | 6379/TCP | 2m43s |
| redis-replica | ClusterIP | 10.0.1.20 | <none> | 6379/TCP | 2m43s |

Figure 4.23: Output displaying a public IP

5. You will now make another change. You'll downgrade the front-end image on line 127 from image: gcr.io/google-samples/gb-frontend:v4 to the following:

```
image: gcr.io/google-samples/gb-frontend:v3
```

This change can be made by opening the guestbook application in the editor by using this familiar command:

```
code guestbook-all-in-one.yaml
```

6. Run the following command to perform the update and watch the pods change:

```
kubectl apply -f guestbook-all-in-one.yaml && kubectl get pods -w
```

This will generate an output similar to *Figure 4.24*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl apply -f guestbook-all-in-one.yaml
&& kubectl get pods -w
service/redis-master unchanged
deployment.apps/redis-master unchanged
service/redis-replica unchanged
deployment.apps/redis-replica unchanged
service/frontend unchanged
deployment.apps/frontend configured
```

| NAME | READY | STATUS | RESTARTS | AGE |
|--------------------------------|-------|-------------------|----------|-------|
| frontend-666b4455f5-bv77x | 1/1 | Running | 0 | 29s |
| frontend-666b4455f5-mn4x7 | 1/1 | Running | 0 | 27s |
| frontend-666b4455f5-ws2mn | 1/1 | Running | 0 | 30s |
| frontend-74f5779d98-bb58v | 0/1 | ContainerCreating | 0 | 0s |
| redis-master-f46ff57fd-jg6zx | 1/1 | Running | 0 | 6m11s |
| redis-replica-5bc7bcc9c4-4f2tj | 1/1 | Running | 0 | 6m11s |
| redis-replica-5bc7bcc9c4-d9mhn | 1/1 | Running | 0 | 6m11s |
| frontend-74f5779d98-bb58v | 1/1 | Running | 0 | 1s |
| frontend-666b4455f5-mn4x7 | 1/1 | Terminating | 0 | 28s |
| frontend-74f5779d98-qllgn | 0/1 | Pending | 0 | 0s |
| frontend-74f5779d98-qllgn | 0/1 | Pending | 0 | 0s |
| frontend-74f5779d98-qllgn | 0/1 | ContainerCreating | 0 | 0s |
| frontend-666b4455f5-mn4x7 | 0/1 | Terminating | 0 | 29s |

Figure 4.24: Pods from a new ReplicaSet are created

What you can see here is that a new version of the pod gets created (based on a new ReplicaSet). Once the new pod is running and ready, one of the old pods is terminated. This create-terminate loop is repeated until only new pods are running. In *Chapter 5, Handling common failures in AKS*, you'll see an example of such an upgrade gone wrong and you'll see that Kubernetes will not continue with the upgrade process until the new pods are healthy.

7. Running `kubectl get events | grep ReplicaSet` will show the rolling update strategy that the deployment uses to update the front-end images:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get events | grep ReplicaSet
54m      Normal    ScalingReplicaSet   deployment/frontend    Scaled up replica set frontend-666b4455f5 to 3
5m1s     Normal    ScalingReplicaSet   deployment/frontend    Scaled up replica set frontend-666b4455f5 to 3
6m5s     Normal    ScalingReplicaSet   deployment/frontend    Scaled up replica set frontend-74f5779d98 to 1
4m33s    Normal    ScalingReplicaSet   deployment/frontend    Scaled down replica set frontend-666b4455f5 to 2
```

Figure 4.25: Monitoring Kubernetes events and filtering to only see ReplicaSet-related events

Note

In the preceding example, you are making use of a pipe—shown by the `|` sign—and the `grep` command. A pipe in Linux is used to send the output of one command to the input of another command. In this case, you sent the output of `kubectl get events` to the `grep` command. Linux uses the `grep` command to filter text. In this case, you used the `grep` command to only show lines that contain the word `ReplicaSet`.

You can see here that the new ReplicaSet gets scaled up, while the old one gets scaled down. You will also see two ReplicaSets for the front-end, the new one replacing the other one pod at a time:

```
kubectl get replicaset
```

This will display the output shown in *Figure 4.26*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get replicaset
NAME                                DESIRED   CURRENT   READY   AGE
frontend-666b4455f5                 0         0         0       12m
frontend-74f5779d98                 3         3         3       8m11s
redis-master-f46ff57fd              1         1         1       12m
redis-replica-5bc7bcc9c4            2         2         2       12m
```

Figure 4.26: Two different ReplicaSets

- Kubernetes will also keep a history of your rollout. You can see the rollout history using this command:

```
kubectl rollout history deployment frontend
```

This will generate the output shown in *Figure 4.27*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl rollout history deployment frontend
deployment.apps/frontend
REVISION  CHANGE-CAUSE
3          <none>
4          <none>
```

Figure 4.27: Deployment history of the application

- Since Kubernetes keeps a history of the rollout, this also enables rollback. Let's do a rollback of your deployment:

```
kubectl rollout undo deployment frontend
```

This will trigger a rollback. This means that the new ReplicaSet will be scaled down to zero instances, and the old one will be scaled up to three instances again. You can verify this using the following command:

```
kubectl get replicaset
```

The resultant output is as shown in *Figure 4.28*:

```
user@Azure:~/Hands-On-Kubernetes-on-Azure/Chapter04$ kubectl get replicaset
```

| NAME | DESIRED | CURRENT | READY | AGE |
|--------------------------|---------|---------|-------|-----|
| frontend-666b4455f5 | 3 | 3 | 3 | 14m |
| frontend-74f5779d98 | 0 | 0 | 0 | 10m |
| redis-master-f46ff57fd | 1 | 1 | 1 | 14m |
| redis-replica-5bc7bcc9c4 | 2 | 2 | 2 | 14m |

Figure 4.28: The old ReplicaSet now has three pods, and the new one is scaled down to zero

This shows you, as expected, that the old ReplicaSet is scaled back to three instances and the new one is scaled down to zero instances.

- Finally, let's clean up again by running the `kubectl delete` command:

```
kubectl delete -f guestbook-all-in-one.yaml
```

Congratulations! You have completed the upgrade of an application and a rollback to a previous version.

In this example, you have used `kubectl apply` to make changes to your application. You can similarly also use `kubectl edit` to make changes, which will be explored in the next section.

Upgrading an application using `kubectl edit`

You can also make changes to your application running on top of Kubernetes by using `kubectl edit`. You used this previously in this chapter, in the *Manually scaling your application* section. When running `kubectl edit`, the `vi` editor will be opened for you, which will allow you to make changes directly against the object in Kubernetes.

Let's redeploy the guestbook application without a public load balancer and use `kubectl` to create the load balancer:

1. Undo the changes you made in the previous step. You can do this by using the following command:

```
git reset --hard
```

2. You will then deploy the guestbook application:

```
kubectl create -f guestbook-all-in-one.yaml
```

3. To start the edit, execute the following command:

```
kubectl edit service frontend
```

4. This will open a `vi` environment. Navigate to the line that now says type: ClusterIP (line 27) and change that to type: LoadBalancer, as shown in *Figure 4.29*. To make that change, hit the `I` button, type your changes, hit the `Esc` button, type `:wq!`, and then hit `Enter` to save the changes:

```
spec:
  clusterIP: 10.0.118.101
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: guestbook
    tier: frontend
  sessionAffinity: None
  type: LoadBalancer
status:
  loadBalancer: {}
```

Figure 4.29: Changing this line to type: LoadBalancer

5. Once the changes are saved, you can watch the service object until the public IP becomes available. To do this, type the following:

```
kubectl get svc -w
```

6. It will take a couple of minutes to show you the updated IP. Once you see the right public IP, you can exit the watch command by hitting *Ctrl + C*.

This is an example of using `kubectl edit` to make changes to a Kubernetes object. This command will open up a text editor to interactively make changes. This means that you need to interact with the text editor to make the changes. This will not work in an automated environment. To make automated changes, you can use the `kubectl patch` command.

Upgrading an application using `kubectl patch`

In the previous example, you used a text editor to make the changes to Kubernetes. In this example, you will use the `kubectl patch` command to make changes to resources on Kubernetes. The patch command is particularly useful in automated systems when you don't have access to the original YAML file that is deployed on a cluster. It can be used, for example, in a script or in a continuous integration/continuous deployment system.

There are two main ways in which to use `kubectl patch`: either by creating a file containing your changes (called a patch file) or by providing the changes inline. Both approaches will be explained here. First, in this example, you'll change the image of the front-end from v4 to v3 using a patch file:

1. Start this example by creating a file called `frontend-image-patch.yaml`:

```
code frontend-image-patch.yaml
```

2. Use the following text as a patch in that file:

```
spec:
  template:
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v3
```

This patch file uses the same YAML layout as a typical YAML file. The main thing about a patch file is that it only has to contain the changes and doesn't have to be capable of deploying the whole resource.

3. To apply the patch, use the following command:

```
kubectl patch deployment frontend \
  --patch "$(cat frontend-image-patch.yaml)"
```

This command does two things: first, it reads the `frontend-image-patch.yaml` file using the `cat` command, and then it passes that to the `kubectl patch` command to execute the change.

4. You can verify the changes by describing the front-end deployment and looking for the Image section:

```
kubectl describe deployment frontend
```

This will display an output as follows:

```
Pod Template:
  Labels:  app=guestbook
           tier=frontend
  Containers:
    php-redis:
      Image:      gcr.io/google-samples/gb-frontend:v4
      Port:      80/TCP
      Host Port:  0/TCP
      Requests:
        cpu:      10m
        memory:   10Mi
```

Figure 4.30: After the patch, we are running the old image

This was an example of using the patch command using a patch file. You can also apply a patch directly on the command line without creating a YAML file. In this case, you would describe the change in JSON rather than in YAML.

Let's run through an example in which we will revert the image change to v4:

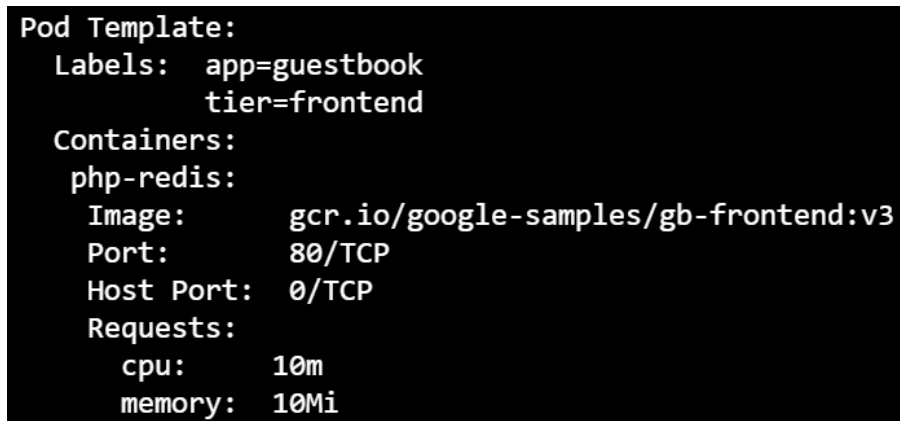
5. Run the following command to patch the image back to v4:

```
kubectl patch deployment frontend \
--patch='
{
  "spec": {
    "template": {
      "spec": {
        "containers": [{
          "name": "php-redis",
          "image": "gcr.io/google-samples/gb-frontend:v4"
        }]
      }
    }
  }
}
```

6. You can verify this change by describing the deployment and looking for the Image section:

```
kubectl describe deployment frontend
```

This will display the output shown in *Figure 4.31*:



```
Pod Template:
  Labels:  app=guestbook
           tier=frontend
  Containers:
    php-redis:
      Image:      gcr.io/google-samples/gb-frontend:v3
      Port:       80/TCP
      Host Port:  0/TCP
      Requests:
        cpu:      10m
        memory:   10Mi
```

Figure 4.31: After another patch, we are running the new version again

Before moving on to the next example, let's remove the guestbook application from the cluster:

```
kubectl delete -f guestbook-all-in-one.yaml
```

So far, you have explored three ways of upgrading Kubernetes applications. First, you made changes to the actual YAML file and applied them using `kubectl apply`. Afterward, you used `kubectl edit` and `kubectl patch` to make more changes. In the final section of this chapter, you will use Helm to upgrade an application.

Upgrading applications using Helm

This section will explain how to perform upgrades using Helm operators:

1. Run the following command:

```
helm install wp bitnami/wordpress
```

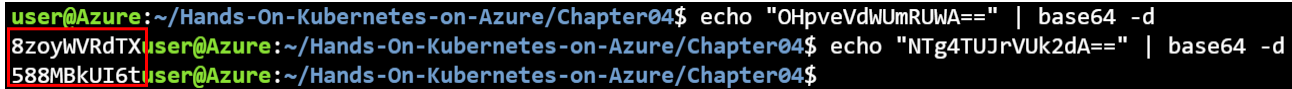
You will force an update of the image of the MariaDB container. Let's first check the version of the current image:

```
kubectl describe statefulset wp-mariadb | grep Image
```

In order to get the decoded password, use the following command:

```
echo "<password>" | base64 -d
```

This will show us the decoded root password and the decoded database password, as shown in *Figure 4.34*:

A terminal window with a black background and green text. The prompt is 'user@Azure: ~/Hands-On-Kubernetes-on-Azure/Chapter04\$'. The first command is 'echo "OHpveVdWUmRUWA==" | base64 -d' and the output is '8zoyWVRdTX'. The second command is 'echo "NTg4TUJrVUk2dA==" | base64 -d' and the output is '588MBkUI6t'.

```
user@Azure: ~/Hands-On-Kubernetes-on-Azure/Chapter04$ echo "OHpveVdWUmRUWA==" | base64 -d
8zoyWVRdTX
user@Azure: ~/Hands-On-Kubernetes-on-Azure/Chapter04$ echo "NTg4TUJrVUk2dA==" | base64 -d
588MBkUI6t
user@Azure: ~/Hands-On-Kubernetes-on-Azure/Chapter04$
```

Figure 4.34: The decoded root and database passwords

You also need the WordPress password. You can get that by getting the wp-wordpress secret and using the same decoding process:

```
kubectl get secret wp-wordpress -o yaml
echo "<WordPress password>" | base64 -d
```

2. You can update the image tag with Helm and then watch the pods change using the following command:

```
helm upgrade wp bitnami/wordpress \
--set mariadb.image.tag=10.5.8-debian-10-r44\
--set mariadb.auth.password="<decoded password>" \
--set mariadb.auth.rootPassword="<decoded password>" \
--set wordpressPassword="<decoded password>" \
&& kubectl get pods -w
```

This will update the image of MariaDB and make a new pod start. You should see an output similar to *Figure 4.35*, where you can see the previous version of the database pod being terminated, and a new one start:

| NAME | READY | STATUS | RESTARTS | AGE |
|-------------------------------|-------|-------------------|----------|-------|
| wp-mariadb-0 | 1/1 | Terminating | 0 | 3m37s |
| wp-wordpress-6f7c4f85b5-t9dlf | 1/1 | Running | 0 | 3m37s |
| wp-mariadb-0 | 0/1 | Terminating | 0 | 3m39s |
| wp-mariadb-0 | 0/1 | Terminating | 0 | 3m40s |
| wp-mariadb-0 | 0/1 | Terminating | 0 | 3m40s |
| wp-mariadb-0 | 0/1 | Pending | 0 | 0s |
| wp-mariadb-0 | 0/1 | Pending | 0 | 0s |
| wp-mariadb-0 | 0/1 | ContainerCreating | 0 | 0s |
| wp-mariadb-0 | 0/1 | Running | 0 | 27s |
| wp-wordpress-6f7c4f85b5-t9dlf | 0/1 | Running | 0 | 4m29s |
| wp-wordpress-6f7c4f85b5-t9dlf | 0/1 | Running | 1 | 4m39s |
| wp-mariadb-0 | 1/1 | Running | 0 | 63s |
| wp-wordpress-6f7c4f85b5-t9dlf | 1/1 | Running | 1 | 5m9s |

Figure 4.35: The previous MariaDB pod gets terminated and a new one starts

Running `describe` on the new pod and grepping for `Image` will show us the new image version:

```
kubectl describe pod wp-mariadb-0 | grep Image
```

This will generate an output as shown in Figure 4.36:

```
user@Azure:~$ kubectl describe pod wp-mariadb-0 | grep Image
Image:          docker.io/bitnami/mariadb:10.5.8-debian-10-r44
Image ID:       docker.io/bitnami/mariadb@sha256:02ea62312a3b05
```

Figure 4.36: Showing the new image

3. Finally, clean up by running the following command:

```
helm delete wp
kubectl delete pvc --all
kubectl delete pv --all
```

You have now learned how to upgrade an application using Helm. As you have seen in this example, upgrading using Helm can be done by using the `--set` operator. This makes performing upgrades and multiple deployments using Helm efficient.

Summary

This a chapter covered a plethora of information on building scalable applications. The goal was to show you how to scale deployments with Kubernetes, which was achieved by creating multiple instances of your application.

We started the chapter by looking at how to define the use of a load balancer and leverage the deployment scale feature in Kubernetes to achieve scalability. With this type of scalability, you can also achieve failover by using a load balancer and multiple instances of the software for stateless applications. We also looked into using the HPA to automatically scale your deployment based on load.

After that, we looked at how you can scale the cluster itself. First, we manually scaled the cluster, and afterward we used a cluster autoscaler to scale the cluster based on application demand.

We finished the chapter by looking into different ways to upgrade a deployed application: first, by exploring updating YAML files manually, and then by learning two additional `kubectl` commands (`edit` and `patch`) that can be used to make changes. Finally, we learned how Helm can be used to perform these upgrades.

In the next chapter, we will look at a couple of common failures that you may face while deploying applications to AKS and how to fix them.