

# Contents

## Azure Kubernetes Service (AKS)

### Overview

#### About AKS

### Quickstarts

#### Create an AKS Cluster

##### Use the Azure CLI

##### Use the Azure portal

##### Use a Resource Manager template

### Develop applications

#### Use Draft

#### Azure Dev Spaces

##### Use Azure Dev Spaces for team development

##### Use Visual Studio Code

##### Use Visual Studio

##### Use the CLI

##### Use Azure Dev Spaces with Java

##### Use Azure Dev Spaces Node.js

### Tutorials

#### 1 - Prepare application for AKS

#### 2 - Create container registry

#### 3 - Create Kubernetes cluster

#### 4 - Run application

#### 5 - Scale application

#### 6 - Update application

#### 7 - Upgrade cluster

### Concepts

#### Clusters and workloads

#### Access and identity

#### Security

[Networking](#)

[Storage](#)

[Scale](#)

[Best practices](#)

[Overview](#)

[For cluster operators](#)

[Multi-tenancy and cluster isolation](#)

[Basic scheduler features](#)

[Advanced scheduler features](#)

[Authentication and authorization](#)

[Cluster security](#)

[Container image management](#)

[Networking](#)

[Storage](#)

[Business continuity \(BC\) and disaster recovery \(DR\)](#)

[For application developers](#)

[Resource management](#)

[Pod security](#)

[Quotas and regional limits](#)

[Migrate to AKS](#)

[Supported Kubernetes version](#)

[Security Hardening in host OS](#)

[Azure Kubernetes Service Diagnostics overview](#)

[How-to guides](#)

[Cluster operations](#)

[Create an AKS cluster](#)

[Scale an AKS cluster](#)

[Upgrade an AKS cluster](#)

[Process node OS updates](#)

[Delete an AKS cluster](#)

[Integrate ACR with an AKS cluster](#)

[Create virtual nodes](#)

- [Use the Azure CLI](#)
- [Use the Azure portal](#)
- [Use Cluster Autoscaler](#)
- [Use Availability Zones](#)
- [Use multiple node pools](#)
- [Use spot node pools \(preview\)](#)
- [Deploy AKS with Terraform](#)
- [Use the Kubernetes dashboard](#)
- [Configure data volumes](#)
  - [Azure Disk - Dynamic](#)
  - [Azure Disk - Static](#)
  - [Azure Files - Dynamic](#)
  - [Azure Files - Static](#)
  - [NFS Server - Static](#)
  - [Azure NetApp Files](#)
- [Configure networking](#)
  - [Create or use existing virtual network](#)
    - [Use kubenet](#)
    - [Use Azure-CNI](#)
  - [Create an internal load balancer](#)
  - [Use a Standard Load Balancer](#)
  - [Use a user defined route for egress](#)
  - [Use a static IP address](#)
- [Ingress](#)
  - [Create a basic controller](#)
  - [Use HTTP application routing](#)
  - [Use internal network](#)
  - [Use TLS with your own certificates](#)
  - [Use TLS with Let's Encrypt](#)
    - [Use a dynamic public IP address](#)
    - [Use a static public IP address](#)
- [Egress traffic](#)

## Customize CoreDNS

### Security and authentication

Create service principal

Use managed identities (preview)

Limit access to cluster configuration file

Secure pod traffic with network policies

Use pod security policies (preview)

Define API server authorized IP ranges

Control deployments with Azure Policy (preview)

Update cluster service principal credentials

Restrict and control cluster egress traffic

Enable Azure Active Directory integration

    Use the Azure CLI

    Use the Azure portal

Use Kubernetes RBAC with Azure AD integration

Rotate certificates

Create a private cluster

BYOK for disks

### Monitoring and logging

Azure Monitor for containers

View the master component logs

View the kubelet logs

View container data real-time

### Use Windows Server containers (preview)

Create an AKS cluster

Connect remotely

Known limitations

Use the Kubernetes dashboard

Create Dockerfiles for Windows Server containers

Optimize Dockerfiles for Windows Server containers

### Develop and run applications

Develop with Dev Spaces

## [Java \(VS Code & CLI\)](#)

- [1 - Get started](#)
- [2 - Multi-service development](#)
- [3 - Team development](#)

## [.NET Core \(VS Code & CLI\)](#)

- [1 - Get started](#)
- [2 - Multi-service development](#)
- [3 - Team development](#)

## [.NET Core \(Visual Studio 2017\)](#)

- [1 - Get started](#)
- [2 - Multi-service development](#)
- [3 - Team development](#)

## [Node.js \(VS Code & CLI\)](#)

- [1 - Get started](#)
- [2 - Multi-service development](#)
- [3 - Team development](#)

## [Run applications with Helm](#)

### [Use Open Service Broker for Azure](#)

### [Use Cosmos DB API for MongoDB with OSBA](#)

### [Use OpenFaaS](#)

### [Run Spark jobs](#)

### [Use GPUs](#)

### [Use Azure Database for PostgreSQL](#)

### [Use Azure API Management](#)

## [Select and deploy a service mesh](#)

### [About Service Meshes](#)

### [Use Istio](#)

#### [About Istio](#)

#### [Install and configure](#)

#### [Scenario - Intelligent routing and canary releases](#)

### [Use Linkerd](#)

#### [About Linkerd](#)

[Install and configure](#)

[Use Consul](#)

[About Consul](#)

[Install and configure](#)

[DevOps](#)

[Use Ansible to create AKS clusters](#)

[Jenkins continuous deployment](#)

[Azure DevOps Project](#)

[Deployment Center Launcher](#)

[GitHub Actions for Kubernetes](#)

[Troubleshoot](#)

[Common issues](#)

[Checking for best practices](#)

[SSH node access](#)

[Linux performance tools](#)

[Reference](#)

[Azure CLI](#)

[REST](#)

[PowerShell](#)

[.NET](#)

[Python](#)

[Java](#)

[Node.js](#)

[Resource Manager template](#)

[Resources](#)

[Build your skill with Microsoft Learn](#)

[Region availability](#)

[Pricing](#)

[Support policies](#)

[Roadmap](#)

[Provide product feedback](#)

[Stack Overflow](#)

[Videos](#)

[FAQ](#)

# Azure Kubernetes Service (AKS)

2/25/2020 • 5 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) makes it simple to deploy a managed Kubernetes cluster in Azure. AKS reduces the complexity and operational overhead of managing Kubernetes by offloading much of that responsibility to Azure. As a hosted Kubernetes service, Azure handles critical tasks like health monitoring and maintenance for you. The Kubernetes masters are managed by Azure. You only manage and maintain the agent nodes. As a managed Kubernetes service, AKS is free - you only pay for the agent nodes within your clusters, not for the masters.

You can create an AKS cluster in the Azure portal, with the Azure CLI, or template driven deployment options such as Resource Manager templates and Terraform. When you deploy an AKS cluster, the Kubernetes master and all nodes are deployed and configured for you. Additional features such as advanced networking, Azure Active Directory integration, and monitoring can also be configured during the deployment process. Windows Server containers support is currently in preview in AKS.

For more information on Kubernetes basics, see [Kubernetes core concepts for AKS](#).

To get started, complete the AKS quickstart [in the Azure portal](#) or [with the Azure CLI](#).

## NOTE

This service supports [Azure delegated resource management](#), which lets service providers sign in to their own tenant to manage subscriptions and resource groups that customers have delegated. For more info, see [Azure Lighthouse](#).

## Access, security, and monitoring

For improved security and management, AKS lets you integrate with Azure Active Directory and use Kubernetes role-based access controls. You can also monitor the health of your cluster and resources.

### Identity and security management

To limit access to cluster resources, AKS supports [Kubernetes role-based access control \(RBAC\)](#). RBAC lets you control access to Kubernetes resources and namespaces, and permissions to those resources. You can also configure an AKS cluster to integrate with Azure Active Directory (AD). With Azure AD integration, Kubernetes access can be configured based on existing identity and group membership. Your existing Azure AD users and groups can be provided access to AKS resources and with an integrated sign-on experience.

For more information on identity, see [Access and identity options for AKS](#).

To secure your AKS clusters, see [Integrate Azure Active Directory with AKS](#).

### Integrated logging and monitoring

To understand how your AKS cluster and deployed applications are performing, Azure Monitor for container health collects memory and processor metrics from containers, nodes, and controllers. Container logs are available, and you can also [review the Kubernetes master logs](#). This monitoring data is stored in an Azure Log Analytics workspace, and is available through the Azure portal, Azure CLI, or a REST endpoint.

For more information, see [Monitor Azure Kubernetes Service container health](#).

## Clusters and nodes

AKS nodes run on Azure virtual machines. You can connect storage to nodes and pods, upgrade cluster

components, and use GPUs. AKS supports Kubernetes clusters that run multiple node pools to support mixed operating systems and Windows Server containers (currently in preview). Linux nodes run a customized Ubuntu OS image, and Windows Server nodes run a customized Windows Server 2019 OS image.

### Cluster node and pod scaling

As demand for resources change, the number of cluster nodes or pods that run your services can automatically scale up or down. You can use both the horizontal pod autoscaler or the cluster autoscaler. This approach to scaling lets the AKS cluster automatically adjust to demands and only run the resources needed.

For more information, see [Scale an Azure Kubernetes Service \(AKS\) cluster](#).

### Cluster node upgrades

Azure Kubernetes Service offers multiple Kubernetes versions. As new versions become available in AKS, your cluster can be upgraded using the Azure portal or Azure CLI. During the upgrade process, nodes are carefully cordoned and drained to minimize disruption to running applications.

To learn more about lifecycle versions, see [Supported Kubernetes versions in AKS](#). For steps on how to upgrade, see [Upgrade an Azure Kubernetes Service \(AKS\) cluster](#).

### GPU enabled nodes

AKS supports the creation of GPU enabled node pools. Azure currently provides single or multiple GPU enabled VMs. GPU enabled VMs are designed for compute-intensive, graphics-intensive, and visualization workloads.

For more information, see [Using GPUs on AKS](#).

### Storage volume support

To support application workloads, you can mount storage volumes for persistent data. Both static and dynamic volumes can be used. Depending on how many connected pods are to share the storage, you can use storage backed by either Azure Disks for single pod access, or Azure Files for multiple concurrent pod access.

For more information, see [Storage options for applications in AKS](#).

Get started with dynamic persistent volumes using [Azure Disks](#) or [Azure Files](#).

## Virtual networks and ingress

An AKS cluster can be deployed into an existing virtual network. In this configuration, every pod in the cluster is assigned an IP address in the virtual network, and can directly communicate with other pods in the cluster, and other nodes in the virtual network. Pods can connect also to other services in a peered virtual network, and to on-premises networks over ExpressRoute or site-to-site (S2S) VPN connections.

For more information, see the [Network concepts for applications in AKS](#).

To get started with ingress traffic, see [HTTP application routing](#).

### Ingress with HTTP application routing

The HTTP application routing add-on makes it easy to access applications deployed to your AKS cluster. When enabled, the HTTP application routing solution configures an ingress controller in your AKS cluster. As applications are deployed, publicly accessible DNS names are auto configured. The HTTP application routing configures a DNS zone and integrates it with the AKS cluster. You can then deploy Kubernetes ingress resources as normal.

To get started with ingress traffic, see [HTTP application routing](#).

## Development tooling integration

Kubernetes has a rich ecosystem of development and management tools such as Helm, Draft, and the Kubernetes extension for Visual Studio Code. These tools work seamlessly with AKS.

Additionally, Azure Dev Spaces provides a rapid, iterative Kubernetes development experience for teams. With minimal configuration, you can run and debug containers directly in AKS. To get started, see [Azure Dev Spaces](#).

The Azure DevOps project provides a simple solution for bringing existing code and Git repository into Azure. The DevOps project automatically creates Azure resources such as AKS, a release pipeline in Azure DevOps Services that includes a build pipeline for CI, sets up a release pipeline for CD, and then creates an Azure Application Insights resource for monitoring.

For more information, see [Azure DevOps project](#).

## Docker image support and private container registry

AKS supports the Docker image format. For private storage of your Docker images, you can integrate AKS with Azure Container Registry (ACR).

To create private image store, see [Azure Container Registry](#).

## Kubernetes certification

Azure Kubernetes Service (AKS) has been CNCF certified as Kubernetes conformant.

## Regulatory compliance

Azure Kubernetes Service (AKS) is compliant with SOC, ISO, PCI DSS, and HIPAA. For more information, see [Overview of Microsoft Azure compliance](#).

## Next steps

Learn more about deploying and managing AKS with the Azure CLI quickstart.

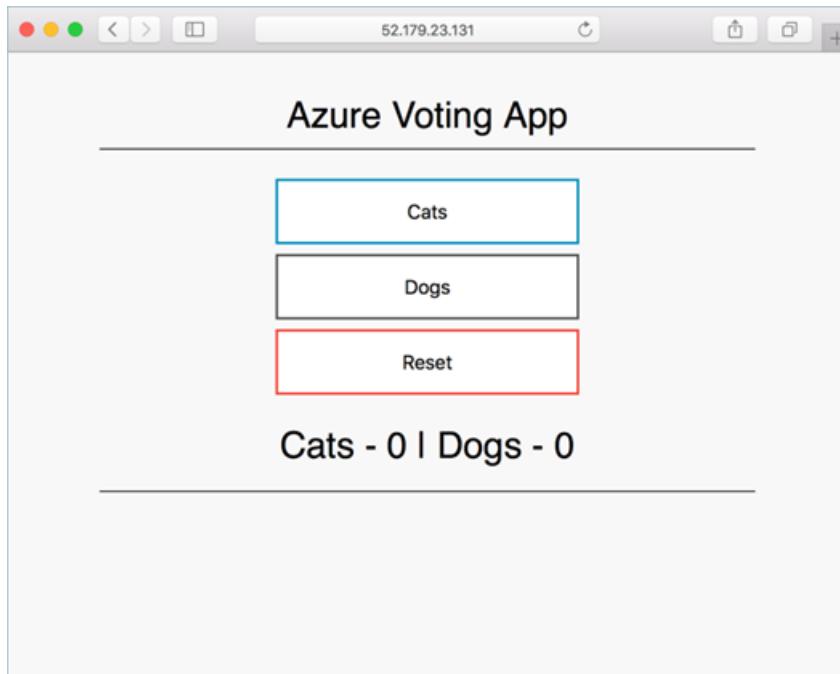
[AKS quickstart](#)

# Quickstart: Deploy an Azure Kubernetes Service cluster using the Azure CLI

2/25/2020 • 6 minutes to read • [Edit Online](#)

In this quickstart, you deploy an Azure Kubernetes Service (AKS) cluster using the Azure CLI. AKS is a managed Kubernetes service that lets you quickly deploy and manage clusters. A multi-container application that includes a web front end and a Redis instance is run in the cluster. You then see how to monitor the health of the cluster and pods that run your application.

If you want to use Windows Server containers (currently in preview in AKS), see [Create an AKS cluster that supports Windows Server containers](#).



This quickstart assumes a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	<a href="#">Azure CLI</a> (The Try It button is highlighted with a red box.)

OPTION	EXAMPLE/LINK
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

If you choose to install and use the CLI locally, this quickstart requires that you are running the Azure CLI version 2.0.64 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

#### NOTE

If running the commands in this quickstart locally (instead of Azure Cloud Shell), ensure you run the commands as administrator.

## Create a resource group

An Azure resource group is a logical group in which Azure resources are deployed and managed. When you create a resource group, you are asked to specify a location. This location is where resource group metadata is stored, it is also where your resources run in Azure if you don't specify another region during resource creation. Create a resource group using the [az group create](#) command.

The following example creates a resource group named *myResourceGroup* in the *eastus* location.

```
az group create --name myResourceGroup --location eastus
```

The following example output shows the resource group created successfully:

```
{
  "id": "/subscriptions/<guid>/resourceGroups/myResourceGroup",
  "location": "eastus",
  "managedBy": null,
  "name": "myResourceGroup",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
}
```

## Create AKS cluster

Use the [az aks create](#) command to create an AKS cluster. The following example creates a cluster named

*myAKSCluster* with one node. Azure Monitor for containers is also enabled using the `--enable-addons monitoring` parameter. This will take several minutes to complete.

#### NOTE

When creating an AKS cluster a second resource group is automatically created to store the AKS resources. For more information see [Why are two resource groups created with AKS?](#)

```
az aks create --resource-group myResourceGroup --name myAKSCluster --node-count 1 --enable-addons monitoring --generate-ssh-keys
```

After a few minutes, the command completes and returns JSON-formatted information about the cluster.

## Connect to the cluster

To manage a Kubernetes cluster, you use [kubectl](#), the Kubernetes command-line client. If you use Azure Cloud Shell, `kubectl` is already installed. To install `kubectl` locally, use the [az aks install-cli](#) command:

```
az aks install-cli
```

To configure `kubectl` to connect to your Kubernetes cluster, use the [az aks get-credentials](#) command. This command downloads credentials and configures the Kubernetes CLI to use them.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, use the [kubectl get](#) command to return a list of the cluster nodes.

```
kubectl get nodes
```

The following example output shows the single node created in the previous steps. Make sure that the status of the node is *Ready*:

NAME	STATUS	ROLES	AGE	VERSION
aks-nodepool1-31718369-0	Ready	agent	6m44s	v1.12.8

## Run the application

A Kubernetes manifest file defines a desired state for the cluster, such as what container images to run. In this quickstart, a manifest is used to create all objects needed to run the Azure Vote application. This manifest includes two [Kubernetes deployments](#) - one for the sample Azure Vote Python applications, and the other for a Redis instance. Two [Kubernetes Services](#) are also created - an internal service for the Redis instance, and an external service to access the Azure Vote application from the internet.

#### TIP

In this quickstart, you manually create and deploy your application manifests to the AKS cluster. In more real-world scenarios, you can use [Azure Dev Spaces](#) to rapidly iterate and debug your code directly in the AKS cluster. You can use Dev Spaces across OS platforms and development environments, and work together with others on your team.

Create a file named `azure-vote.yaml` and copy in the following YAML definition. If you use the Azure Cloud

Shell, this file can be created using `vi` or `nano` as if working on a virtual or physical system:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-back
  template:
    metadata:
      labels:
        app: azure-vote-back
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": linux
      containers:
        - name: azure-vote-back
          image: redis
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 250m
              memory: 256Mi
          ports:
            - containerPort: 6379
              name: redis
      ---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
    - port: 6379
  selector:
    app: azure-vote-back
  ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-front
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": linux
      containers:
        - name: azure-vote-front
          image: microsoft/azure-vote-front:v1
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 250m
              memory: 256Mi
```

```
  ports:
    - containerPort: 80
  env:
    - name: REDIS
      value: "azure-vote-back"
  ...
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
```

Deploy the application using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f azure-vote.yaml
```

The following example output shows the Deployments and Services created successfully:

```
deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

## Test the application

When the application runs, a Kubernetes service exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the [kubectl get service](#) command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

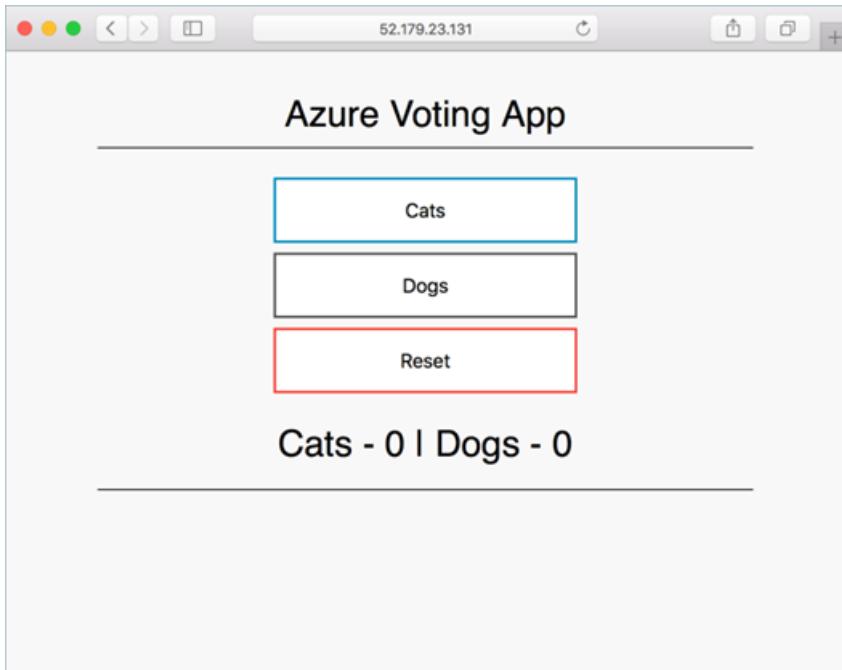
Initially the *EXTERNAL-IP* for the *azure-vote-front* service is shown as *pending*.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
azure-vote-front	LoadBalancer	10.0.37.27	<pending>	80:30572/TCP	6s

When the *EXTERNAL-IP* address changes from *pending* to an actual public IP address, use `CTRL-C` to stop the [kubectl](#) watch process. The following example output shows a valid public IP address assigned to the service:

```
azure-vote-front  LoadBalancer  10.0.37.27  52.179.23.131  80:30572/TCP  2m
```

To see the Azure Vote app in action, open a web browser to the external IP address of your service.



When the AKS cluster was created, [Azure Monitor for containers](#) was enabled to capture health metrics for both the cluster nodes and pods. These health metrics are available in the Azure portal.

## Delete the cluster

To avoid Azure charges, you should clean up unneeded resources. When the cluster is no longer needed, use the `az group delete` command to remove the resource group, container service, and all related resources.

```
az group delete --name myResourceGroup --yes --no-wait
```

### NOTE

When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see [AKS service principal considerations and deletion](#).

## Get the code

In this quickstart, pre-created container images were used to create a Kubernetes deployment. The related application code, Dockerfile, and Kubernetes manifest file are available on GitHub.

<https://github.com/Azure-Samples/azure-voting-app-redis>

## Next steps

In this quickstart, you deployed a Kubernetes cluster and deployed a multi-container application to it. You can also [access the Kubernetes web dashboard](#) for your AKS cluster.

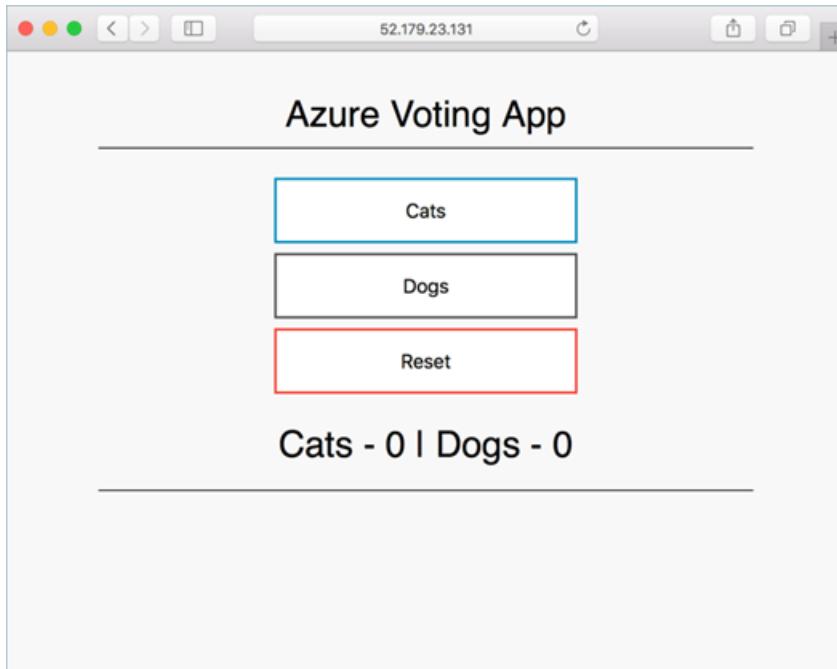
To learn more about AKS, and walk through a complete code to deployment example, continue to the Kubernetes cluster tutorial.

[AKS tutorial](#)

# Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using the Azure portal

2/25/2020 • 7 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) is a managed Kubernetes service that lets you quickly deploy and manage clusters. In this quickstart, you deploy an AKS cluster using the Azure portal. A multi-container application that includes a web front end and a Redis instance is run in the cluster. You then see how to monitor the health of the cluster and pods that run your application.



This quickstart assumes a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

## Sign in to Azure

Sign in to the Azure portal at <https://portal.azure.com>.

## Create an AKS cluster

To create an AKS cluster, complete the following steps:

1. On the Azure portal menu or from the **Home** page, select **Create a resource**.
2. Select **Containers > Kubernetes Service**.
3. On the **Basics** page, configure the following options:
  - **Project details:** Select an Azure **Subscription**, then select or create an Azure **Resource group**, such as *myResourceGroup*.
  - **Cluster details:** Enter a **Kubernetes cluster name**, such as *myAKSCluster*. Select a **Region**, **Kubernetes version**, and **DNS name prefix** for the AKS cluster.
  - **Primary node pool:** Select a VM **Node size** for the AKS nodes. The VM size *can't* be changed once

an AKS cluster has been deployed. - Select the number of nodes to deploy into the cluster. For this quickstart, set **Node count** to 1. Node count *can* be adjusted after the cluster has been deployed.

### Create Kubernetes cluster

**Basics** Scale Authentication Networking Monitoring Tags Review + create

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. [Learn more about Azure Kubernetes Service](#)

#### Project details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription \* ⓘ My Subscription Name

Resource group \* ⓘ myResourceGroup  Create new

#### Cluster details

Kubernetes cluster name \* ⓘ myAKSCluster

Region \* ⓘ (US) East US

Kubernetes version \* ⓘ 1.13.11 (default)

DNS name prefix \* ⓘ myAKSCluster-dns

#### Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. You will not be able to change the node size after cluster creation, but you will be able to change the number of nodes in your cluster after creation. If you would like additional node pools, you will need to enable the "X" feature on the "Scale" tab which will allow you to add more node pools after creating the cluster. [Learn more about node pools in Azure Kubernetes Service](#)

Node size \* ⓘ Standard B2s  
2 vcpus, 4 GiB memory

Node count \* ⓘ

**Review + create** < Previous **Next : Scale >**

Select **Next: Scale** when complete.

- On the **Scale** page, keep the default options. At the bottom of the screen, click **Next: Authentication**.

#### Caution

Creating new AAD Service Principals may take multiple minutes to propagate and become available causing Service Principal not found errors and validation failures in Azure portal. If you hit this please visit [here](#) for mitigations.

- On the **Authentication** page, configure the following options:

- Create a new service principal by leaving the **Service Principal** field with **(new) default service principal**. Or you can choose *Configure service principal* to use an existing one. If you use an existing one, you will need to provide the SPN client ID and secret.
- Enable the option for Kubernetes role-based access controls (RBAC). This will provide more fine-grained control over access to the Kubernetes resources deployed in your AKS cluster.

By default, **Basic** networking is used, and Azure Monitor for containers is enabled. Click **Review + create** and then **Create** when validation completes.

It takes a few minutes to create the AKS cluster. When your deployment is complete, click **Go to resource**, or browse to the AKS cluster resource group, such as *myResourceGroup*, and select the AKS resource, such as *myAKSCluster*. The AKS cluster dashboard is shown, as in this example:

The screenshot shows the AKS cluster dashboard for 'myAKSCluster'. The left sidebar contains navigation links: Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Node pools, Upgrade, Scale, Networking, Dev Spaces, Deployment center (preview), Policies (preview), Properties, Locks), and a search bar. The main content area displays cluster details: Resource group (myResourceGroup), Kubernetes version (1.13.11), Status (Succeeded), Location (East US), API server address (myakscluster-dns-429509e5.hcp.eastus.azmk8s.io), HTTP application routing domain (N/A), Subscription (My Subscription Name), Node pools (1 node pools), Subscription ID (00000000-0000-0000-0000-000000000000), and Tags (Click here to add tags). Below these details are two cards: 'Monitor containers' (Get health and performance insights, Go to Azure Monitor insights) and 'View logs' (Search and analyze logs using ad-hoc queries, Go to Azure Monitor logs).

## Connect to the cluster

To manage a Kubernetes cluster, you use [kubectl](#), the Kubernetes command-line client. The `kubectl` client is pre-installed in the Azure Cloud Shell.

Open Cloud Shell using the `>_` button on the top of the Azure portal.

**myAKSCluster**  
Kubernetes service

Search (Ctrl+/  
)

Overview Activity log Access control (IAM) Tags Diagnose and solve problems

Resource group (change)  
myResourceGroup Status Succeeded Location East US Subscription (change)  
My Subscription Name Subscription ID 00000000-0000-0000-0000-000000000000 Tags (change)  
Click here to add tags

Kubernetes version 1.13.11 API server address myakscluster-dns-429509e5... HTTP application routing N/A Node pools 1 node pools

**Monitor containers**  
Get health and performance insights  
Go to Azure Monitor insights

**View logs**  
Search and analyze logs  
Go to Azure Monitor logs

Bash Requesting a Cloud Shell. **Succeeded.** Connecting terminal... Welcome to Azure Cloud Shell Type "az" to use Azure CLI Type "help" to learn about Cloud Shell user@Azure:~\$

To configure `kubectl` to connect to your Kubernetes cluster, use the `az aks get-credentials` command. This command downloads credentials and configures the Kubernetes CLI to use them. The following example gets credentials for the cluster name *myAKSCluster* in the resource group named *myResourceGroup*:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, use the `kubectl get` command to return a list of the cluster nodes.

```
kubectl get nodes
```

The following example output shows the single node created in the previous steps. Make sure that the status of the node is *Ready*:

NAME	STATUS	ROLES	AGE	VERSION
aks-agentpool-14693408-0	Ready	agent	15m	v1.11.5

## Run the application

A Kubernetes manifest file defines a desired state for the cluster, such as what container images to run. In this quickstart, a manifest is used to create all objects needed to run the Azure Vote application. This manifest includes two [Kubernetes deployments](#) - one for the sample Azure Vote Python applications, and the other for a Redis instance. Two [Kubernetes Services](#) are also created - an internal service for the Redis instance, and an external service to access the Azure Vote application from the internet.

### TIP

In this quickstart, you manually create and deploy your application manifests to the AKS cluster. In more real-world scenarios, you can use [Azure Dev Spaces](#) to rapidly iterate and debug your code directly in the AKS cluster. You can use Dev Spaces across OS platforms and development environments, and work together with others on your team.

In the cloud shell, use either the `nano azure-vote.yaml` or `vi azure-vote.yaml` command to create a file named `azure-vote.yaml`. Then copy in the following YAML definition:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-back
  template:
    metadata:
      labels:
        app: azure-vote-back
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": linux
      containers:
        - name: azure-vote-back
          image: redis
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 250m
              memory: 256Mi
          ports:
            - containerPort: 6379
              name: redis
      ---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
    - port: 6379
  selector:
    app: azure-vote-back
  ---
apiVersion: apps/v1
```

```

kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-front
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": linux
      containers:
        - name: azure-vote-front
          image: microsoft/azure-vote-front:v1
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 250m
              memory: 256Mi
          ports:
            - containerPort: 80
          env:
            - name: REDIS
              value: "azure-vote-back"
      ---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
    - port: 80
  selector:
    app: azure-vote-front

```

Deploy the application using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f azure-vote.yaml
```

The following example output shows the Deployments and Services created successfully:

```

deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created

```

## Test the application

When the application runs, a Kubernetes service exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the [kubectl get service](#) command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

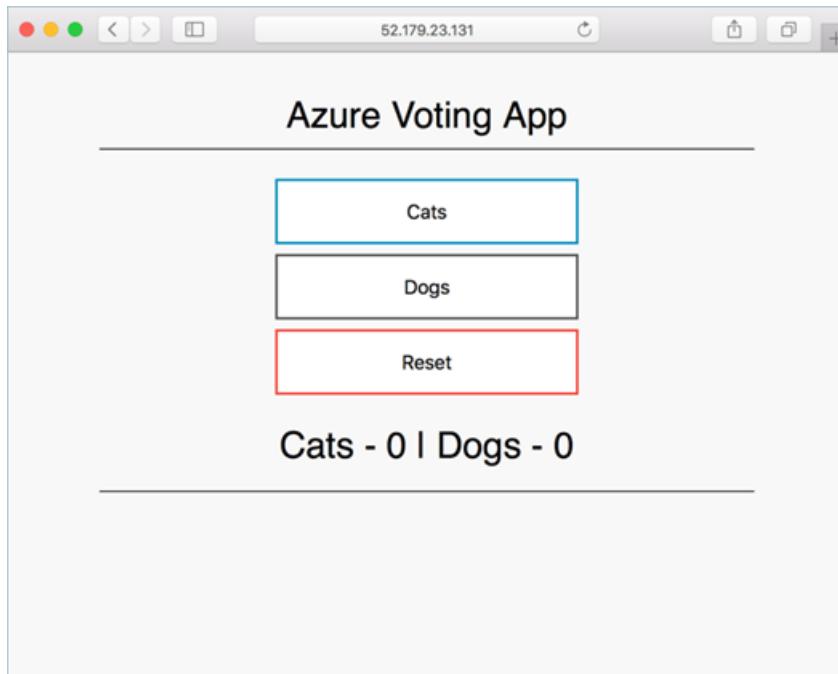
Initially the *EXTERNAL-IP* for the *azure-vote-front* service is shown as *pending*.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
azure-vote-front	LoadBalancer	10.0.37.27	<pending>	80:30572/TCP	6s

When the *EXTERNAL-IP* address changes from *pending* to an actual public IP address, use **CTRL-C** to stop the `kubectl` watch process. The following example output shows a valid public IP address assigned to the service:

```
azure-vote-front LoadBalancer 10.0.37.27 52.179.23.131 80:30572/TCP 2m
```

To see the Azure Vote app in action, open a web browser to the external IP address of your service.



## Monitor health and logs

When you created the cluster, Azure Monitor for containers was enabled. This monitoring feature provides health metrics for both the AKS cluster and pods running on the cluster.

It may take a few minutes for this data to populate in the Azure portal. To see current status, uptime, and resource usage for the Azure Vote pods, browse back to the AKS resource in the Azure portal, such as *myAKSCluster*. You can then access the health status as follows:

1. Under **Monitoring** on the left-hand side, choose **Insights**
2. Across the top, choose to **+ Add Filter**
3. Select *Namespace* as the property, then choose *<All but kube-system>*
4. Choose to view the **Containers**.

The *azure-vote-back* and *azure-vote-front* containers are displayed, as shown in the following example:

The screenshot shows the AKS Insights blade for the 'myAKSCluster - Insights' resource. On the left, the navigation menu includes 'Overview', 'Activity log', 'Access control (IAM)', 'Tags', 'Settings' (with 'Upgrade', 'Scale', 'Dev Spaces', 'Properties', 'Locks', 'Automation script'), 'Monitoring' (selected), 'Logs' (under Insights), 'Metrics (preview)', and 'Support + troubleshooting'. The main area displays container metrics for 'Cluster', 'Nodes', 'Controllers', and 'Containers'. For the 'Containers' tab, the 'az-vote-back' container is selected, showing CPU usage (millicores) over time. A detailed view on the right shows the container's status as 'running', image as 'redis', and tag as 'latest'. It also lists creation time (12/18/2018, 10:54:08 AM), start time (12/18/2018, 10:54:08 AM), and finish time (empty). Resource limits and requests are also listed.

To see logs for the `azure-vote-front` pod, select the **View container logs** from the drop down of the containers list. These logs include the `stdout` and `stderr` streams from the container.

The screenshot shows the Log Analytics workspace for the 'defaultworkspace-19da35d3-9a1a-4fb0-9b9c-3c56e1408565-eus' workspace. The 'Logs' table is selected. A query editor at the top shows a complex SQL-like query for filtering logs by container ID and name. Below the query, a 'Completed' table view displays log entries for the 'stdout' stream of the 'azure-vote-front' pod. The table has columns: LogEntrySource, LogEntry, TimeGenerated [UTC], Computer, Image. The log entries show initial boot messages, connection readiness, and Redis startup information. The bottom of the screen shows pagination controls and a note indicating 1 - 8 of 8 items.

## Delete cluster

When the cluster is no longer needed, delete the cluster resource, which deletes all associated resources. This operation can be completed in the Azure portal by selecting the **Delete** button on the AKS cluster dashboard. Alternatively, the `az aks delete` command can be used in the Cloud Shell:

```
az aks delete --resource-group myResourceGroup --name myAKSCluster --no-wait
```

**NOTE**

When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see [AKS service principal considerations and deletion](#).

## Get the code

In this quickstart, pre-created container images were used to create a Kubernetes deployment. The related application code, Dockerfile, and Kubernetes manifest file are available on GitHub.

<https://github.com/Azure-Samples/azure-voting-app-redis>

## Next steps

In this quickstart, you deployed a Kubernetes cluster and deployed a multi-container application to it.

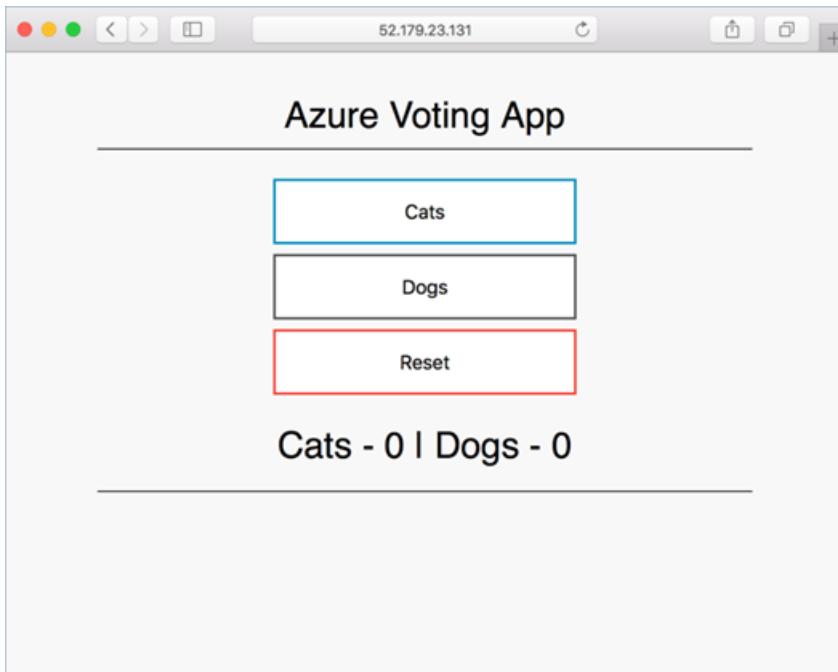
To learn more about AKS, and walk through a complete code to deployment example, continue to the Kubernetes cluster tutorial.

[AKS tutorial](#)

# Quickstart: Deploy an Azure Kubernetes Service (AKS) cluster using an Azure Resource Manager template

2/25/2020 • 7 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) is a managed Kubernetes service that lets you quickly deploy and manage clusters. In this quickstart, you deploy an AKS cluster using an Azure Resource Manager template. A multi-container application that includes a web front end and a Redis instance is run in the cluster.



This quickstart assumes a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	A screenshot of a code block. It shows the text "Azure CLI" followed by a "Copy" button and a "Try It" button. The "Try It" button is highlighted with a red border.
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	A screenshot of a blue button labeled "Launch Cloud Shell" with a white triangle icon.

OPTION	EXAMPLE/LINK
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

If you choose to install and use the CLI locally, this quickstart requires that you are running the Azure CLI version 2.0.61 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Prerequisites

To create an AKS cluster using a Resource Manager template, you provide an SSH public key and Azure Active Directory service principal. If you need either of these resources, see the following section; otherwise skip to the [Create an AKS cluster](#) section.

### Create an SSH key pair

To access AKS nodes, you connect using an SSH key pair. Use the `ssh-keygen` command to generate SSH public and private key files. By default, these files are created in the `~/.ssh` directory. If an SSH key pair with the same name exists in the given location, those files are overwritten.

The following command creates an SSH key pair using RSA encryption and a bit length of 2048:

```
ssh-keygen -t rsa -b 2048
```

For more information about creating SSH keys, see [Create and manage SSH keys for authentication in Azure](#).

### Create a service principal

To allow an AKS cluster to interact with other Azure resources, an Azure Active Directory service principal is used. Create a service principal using the `az ad sp create-for-rbac` command. The `--skip-assignment` parameter limits any additional permissions from being assigned. By default, this service principal is valid for one year.

```
az ad sp create-for-rbac --skip-assignment
```

The output is similar to the following example:

```
{
  "appId": "8b1ede42-d407-46c2-a1bc-6b213b04295f",
  "displayName": "azure-cli-2019-04-19-21-42-11",
  "name": "http://azure-cli-2019-04-19-21-42-11",
  "password": "27e5ac58-81b0-46c1-bd87-85b4ef622682",
  "tenant": "73f978cf-87f2-41bf-92ab-2e7ce012db57"
}
```

Make a note of the `appId` and `password`. These values are used in the following steps.

# Create an AKS cluster

The template used in this quickstart is to [deploy an Azure Kubernetes Service cluster](#). For more AKS samples, see the [AKS quickstart templates](#) site.

1. Select the following image to sign in to Azure and open a template.



2. Select or enter the following values.

For this quickstart, leave the default values for the *OS Disk Size GB*, *Agent Count*, *Agent VM Size*, *OS Type*, and *Kubernetes Version*. Provide your own values for the following template parameters:

- **Subscription:** Select an Azure subscription.
- **Resource group:** Select **Create new**. Enter a unique name for the resource group, such as *myResourceGroup*, then choose **OK**.
- **Location:** Select a location, such as **East US**.
- **Cluster name:** Enter a unique name for the AKS cluster, such as *myAKScluster*.
- **DNS prefix:** Enter a unique DNS prefix for your cluster, such as *myakscluster*.
- **Linux Admin Username:** Enter a username to connect using SSH, such as *azureuser*.
- **SSH RSA Public Key:** Copy and paste the *public* part of your SSH key pair (by default, the contents of `~/.ssh/id_rsa.pub`).
- **Service Principal Client Id:** Copy and paste the *appId* of your service principal from the `az ad sp create-for-rbac` command.
- **Service Principal Client Secret:** Copy and paste the *password* of your service principal from the `az ad sp create-for-rbac` command.

- **I agree to the terms and conditions state above:** Check this box to agree.

The screenshot shows the Azure Container Service (AKS) creation interface. On the left is the Azure portal sidebar with various service icons. The main area has a title 'Azure Container Service (AKS)' and a sub-section 'Azure quickstart template'. A list shows '1D1-aks' with '1 resource'. Below this are two sections: 'BASICS' and 'SETTINGS'. In 'BASICS', fields include 'Subscription' (set to 'Azure'), 'Resource group' (set to '(New) myResourceGroup' with a 'Create new' link), and 'Location' (set to 'East US'). In 'SETTINGS', fields include 'Cluster Name' (set to 'myAKScluster'), 'Location' (set to '[resourceGroup].location'), 'Dns Prefix' (set to 'myakscluster'), 'Os Disk Size GB' (set to '0'), 'Agent Count' (set to '3'), 'Agent VM Size' (set to 'Standard\_DS2\_v2'), 'Linux Admin Username' (set to 'azureuser'), 'SSH RSA Public Key' (set to a long string of characters), 'Service Principal Client Id' (set to '\*\*\*\*\*'), 'Service Principal Client Secret' (set to '\*\*\*\*\*'), 'Os Type' (set to 'Linux'), and 'Kubernetes Version' (set to '1.12.6'). At the bottom is a blue 'Purchase' button.

3. Select **Purchase**.

It takes a few minutes to create the AKS cluster. Wait for the cluster to be successfully deployed before you move

on to the next step.

## Connect to the cluster

To manage a Kubernetes cluster, you use `kubectl`, the Kubernetes command-line client. If you use Azure Cloud Shell, `kubectl` is already installed. To install `kubectl` locally, use the `az aks install-cli` command:

```
az aks install-cli
```

To configure `kubectl` to connect to your Kubernetes cluster, use the `az aks get-credentials` command. This command downloads credentials and configures the Kubernetes CLI to use them.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, use the `kubectl get` command to return a list of the cluster nodes.

```
kubectl get nodes
```

The following example output shows the nodes created in the previous steps. Make sure that the status for all the nodes is *Ready*:

NAME	STATUS	ROLES	AGE	VERSION
aks-agentpool-41324942-0	Ready	agent	6m44s	v1.12.6
aks-agentpool-41324942-1	Ready	agent	6m46s	v1.12.6
aks-agentpool-41324942-2	Ready	agent	6m45s	v1.12.6

## Run the application

A Kubernetes manifest file defines a desired state for the cluster, such as what container images to run. In this quickstart, a manifest is used to create all objects needed to run the Azure Vote application. This manifest includes two [Kubernetes deployments](#) - one for the sample Azure Vote Python applications, and the other for a Redis instance. Two [Kubernetes Services](#) are also created - an internal service for the Redis instance, and an external service to access the Azure Vote application from the internet.

### TIP

In this quickstart, you manually create and deploy your application manifests to the AKS cluster. In more real-world scenarios, you can use [Azure Dev Spaces](#) to rapidly iterate and debug your code directly in the AKS cluster. You can use Dev Spaces across OS platforms and development environments, and work together with others on your team.

Create a file named `azure-vote.yaml` and copy in the following YAML definition. If you use the Azure Cloud Shell, this file can be created using `vi` or `nano` as if working on a virtual or physical system:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-back
  template:
```

```

metadata:
  labels:
    app: azure-vote-back
spec:
  nodeSelector:
    "beta.kubernetes.io/os": linux
  containers:
    - name: azure-vote-back
      image: redis
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      ports:
        - containerPort: 6379
          name: redis
  ---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
    - port: 6379
  selector:
    app: azure-vote-back
  ---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-front
  template:
    metadata:
      labels:
        app: azure-vote-front
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": linux
      containers:
        - name: azure-vote-front
          image: microsoft/azure-vote-front:v1
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 250m
              memory: 256Mi
          ports:
            - containerPort: 80
          env:
            - name: REDIS
              value: "azure-vote-back"
  ---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:

```

```
- port: 80
  selector:
    app: azure-vote-front
```

Deploy the application using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f azure-vote.yaml
```

The following example output shows the Deployments and Services created successfully:

```
deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

## Test the application

When the application runs, a Kubernetes service exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the [kubectl get service](#) command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

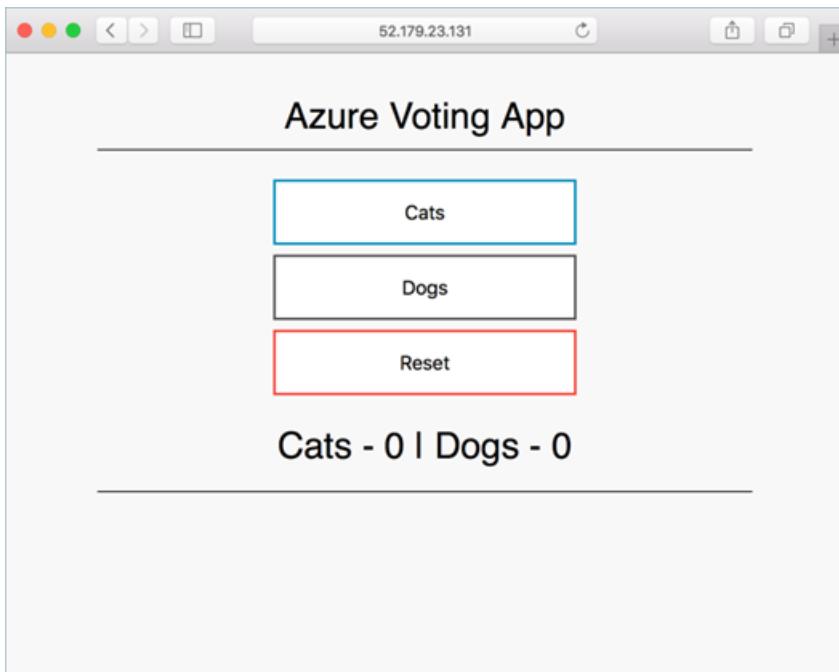
Initially the *EXTERNAL-IP* for the *azure-vote-front* service is shown as *pending*.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
azure-vote-front	LoadBalancer	10.0.37.27	<pending>	80:30572/TCP	6s

When the *EXTERNAL-IP* address changes from *pending* to an actual public IP address, use `CTRL-C` to stop the [kubectl](#) watch process. The following example output shows a valid public IP address assigned to the service:

```
azure-vote-front  LoadBalancer  10.0.37.27  52.179.23.131  80:30572/TCP  2m
```

To see the Azure Vote app in action, open a web browser to the external IP address of your service.



## Delete cluster

When the cluster is no longer needed, use the [az group delete](#) command to remove the resource group, container service, and all related resources.

```
az group delete --name myResourceGroup --yes --no-wait
```

### NOTE

When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see [AKS service principal considerations and deletion](#).

## Get the code

In this quickstart, pre-created container images were used to create a Kubernetes deployment. The related application code, Dockerfile, and Kubernetes manifest file are available on GitHub.

<https://github.com/Azure-Samples/azure-voting-app-redis>

## Next steps

In this quickstart, you deployed a Kubernetes cluster and deployed a multi-container application to it. [Access the Kubernetes web dashboard](#) for the cluster you created.

To learn more about AKS, and walk through a complete code to deployment example, continue to the Kubernetes cluster tutorial.

[AKS tutorial](#)

# Quickstart: Develop on Azure Kubernetes Service (AKS) with Draft

2/25/2020 • 7 minutes to read • [Edit Online](#)

Draft is an open-source tool that helps package and run application containers in a Kubernetes cluster. With Draft, you can quickly redeploy an application to Kubernetes as code changes occur without having to commit your changes to version control. For more information on Draft, see the [Draft documentation on GitHub](#).

This article shows you how to use Draft to package and run an application on AKS.

## Prerequisites

- An Azure subscription. If you don't have an Azure subscription, you can create a [free account](#).
- [Azure CLI installed](#).
- Docker installed and configured. Docker provides packages that configure Docker on a [Mac](#), [Windows](#), or [Linux](#) system.
- [Helm v2 installed](#).
- [Draft installed](#).

## Create an Azure Kubernetes Service cluster

Create an AKS cluster. The below commands create a resource group called MyResourceGroup and an AKS cluster called MyAKS.

```
az group create --name MyResourceGroup --location eastus
az aks create -g MyResourceGroup -n MyAKS --location eastus --node-vm-size Standard_DS2_v2 --node-count 1 --
generate-ssh-keys
```

## Create an Azure Container Registry

To use Draft to run your application in your AKS cluster, you need an Azure Container Registry to store your container images. The below example uses [az acr create](#) to create an ACR named *MyDraftACR* in the *MyResourceGroup* resource group with the *Basic* SKU. You should provide your own unique registry name. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. The *Basic* SKU is a cost-optimized entry point for development purposes that provides a balance of storage and throughput.

```
az acr create --resource-group MyResourceGroup --name MyDraftACR --sku Basic
```

The output is similar to the following example. Make a note of the *loginServer* value for your ACR since it will be used in a later step. In the below example, *mydraftacr.azurecr.io* is the *loginServer* for *MyDraftACR*.

```
{  
    "adminUserEnabled": false,  
    "creationDate": "2019-06-11T13:35:17.998425+00:00",  
    "id":  
        "/subscriptions/<ID>/resourceGroups/MyResourceGroup/providers/Microsoft.ContainerRegistry/registries/MyDraftACR",  
    "location": "eastus",  
    "loginServer": "mydraftacr.azurecr.io",  
    "name": "MyDraftACR",  
    "networkRuleSet": null,  
    "provisioningState": "Succeeded",  
    "resourceGroup": "MyResourceGroup",  
    "sku": {  
        "name": "Basic",  
        "tier": "Basic"  
    },  
    "status": null,  
    "storageAccount": null,  
    "tags": {},  
    "type": "Microsoft.ContainerRegistry/registries"  
}
```

For Draft to use the ACR instance, you must first sign in. Use the [az acr login](#) command to sign in. The below example will sign in to an ACR named *MyDraftACR*.

```
az acr login --name MyDraftACR
```

The command returns a *Login Succeeded* message once completed.

## Create trust between AKS cluster and ACR

Your AKS cluster also needs access to your ACR to pull the container images and run them. You allow access to the ACR from AKS by establishing a trust. To establish trust between an AKS cluster and an ACR registry, grant permissions for the Azure Active Directory service principal used by the AKS cluster to access the ACR registry. The following commands grant permissions to the service principal of the *MyAKS* cluster in the *MyResourceGroup* to the *MyDraftACR* ACR in the *MyResourceGroup*.

```
# Get the service principal ID of your AKS cluster  
AKS_SP_ID=$(az aks show --resource-group MyResourceGroup --name MyAKS --query  
"servicePrincipalProfile.clientId" -o tsv)  
  
# Get the resource ID of your ACR instance  
ACR_RESOURCE_ID=$(az acr show --resource-group MyResourceGroup --name MyDraftACR --query "id" -o tsv)  
  
# Create a role assignment for your AKS cluster to access the ACR instance  
az role assignment create --assignee $AKS_SP_ID --scope $ACR_RESOURCE_ID --role contributor
```

## Connect to your AKS cluster

To connect to the Kubernetes cluster from your local computer, you use [kubectl](#), the Kubernetes command-line client.

If you use the Azure Cloud Shell, `kubectl` is already installed. You can also install it locally using the [az aks install-cli](#) command:

```
az aks install-cli
```

To configure `kubectl` to connect to your Kubernetes cluster, use the `az aks get-credentials` command. The following example gets credentials for the AKS cluster named *MyAKS* in the *MyResourceGroup*:

```
az aks get-credentials --resource-group MyResourceGroup --name MyAKS
```

## Create a service account for Helm

Before you can deploy Helm in an RBAC-enabled AKS cluster, you need a service account and role binding for the Tiller service. For more information on securing Helm / Tiller in an RBAC enabled cluster, see [Tiller, Namespaces, and RBAC](#). If your AKS cluster isn't RBAC enabled, skip this step.

Create a file named `helm-rbac.yaml` and copy in the following YAML:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
```

Create the service account and role binding with the `kubectl apply` command:

```
kubectl apply -f helm-rbac.yaml
```

## Configure Helm

To deploy a basic Tiller into an AKS cluster, use the `helm init` command. If your cluster isn't RBAC enabled, remove the `--service-account` argument and value.

```
helm init --service-account tiller --node-selectors "beta.kubernetes.io/os"="linux"
```

## Configure Draft

If you haven't configured Draft on your local machine, run `draft init`:

```
$ draft init
Installing default plugins...
Installation of default plugins complete
Installing default pack repositories...
...
Happy Sailing!
```

You also need to configure Draft to use the *loginServer* of your ACR. The following command uses `draft config set` to use `mydraftacr.azurecr.io` as a registry.

```
draft config set registry mydraftacr.azurecr.io
```

You've configured Draft to use your ACR, and Draft can push container images to your ACR. When Draft runs your application in your AKS cluster, no passwords or secrets are required to push to or pull from the ACR registry. Since a trust was created between your AKS cluster and your ACR, authentication happens at the Azure Resource Manager level, using Azure Active Directory.

## Download the sample application

This quickstart uses [an example Java application from the Draft GitHub repository](#). Clone the application from GitHub and navigate to the `draft/examples/example-java/` directory.

```
git clone https://github.com/Azure/draft
cd draft/examples/example-java/
```

## Run the sample application with Draft

Use the `draft create` command to prepare the application.

```
draft create
```

This command creates the artifacts that are used to run the application in a Kubernetes cluster. These items include a Dockerfile, a Helm chart, and a *draft.toml* file, which is the Draft configuration file.

```
$ draft create
--> Draft detected Java (92.205567%)
--> Ready to sail
```

To run the sample application in your AKS cluster, use the `draft up` command.

```
draft up
```

This command builds the Dockerfile to create a container image, pushes the image to your ACR, and installs the Helm chart to start the application in AKS. The first time you run this command, pushing and pulling the container image may take some time. Once the base layers are cached, the time taken to deploy the application is dramatically reduced.

```
$ draft up
Draft Up Started: 'example-java': 01CMZAR1F4T1TJZ8SWJQ70HCNH
example-java: Building Docker Image: SUCCESS (73.0720s)
example-java: Pushing Docker Image: SUCCESS (19.5727s)
example-java: Releasing Application: SUCCESS (4.6979s)
Inspect the logs with `draft logs 01CMZAR1F4T1TJZ8SWJQ70HCNH`
```

## Connect to the running sample application from your local machine

To test the application, use the `draft connect` command.

```
draft connect
```

This command proxies a secure connection to the Kubernetes pod. When complete, the application can be accessed on the provided URL.

```
$ draft connect

Connect to java:4567 on localhost:49804
[java]: SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
[java]: SLF4J: Defaulting to no-operation (NOP) logger implementation
[java]: SLF4J: See https://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
[java]: == Spark has ignited ...
[java]: >> Listening on 0.0.0.0:4567
```

Navigate to the application in a browser using the `localhost` URL to see the sample application. In the above example, the URL is `http://localhost:49804`. Stop the connection using `Ctrl+c`.

## Access the application on the internet

The previous step created a proxy connection to the application pod in your AKS cluster. As you develop and test your application, you may want to make the application available on the internet. To expose an application on the internet, you can create a Kubernetes service with a type of [LoadBalancer](#).

Update `charts/example-java/values.yaml` to create a *LoadBalancer* service. Change the value of `service.type` from `ClusterIP` to `LoadBalancer`.

```
...
service:
  name: java
  type: LoadBalancer
  externalPort: 80
  internalPort: 4567
...
```

Save your changes, close the file, and run `draft up` to rerun the application.

```
draft up
```

It takes a few minutes for the service to return a public IP address. To monitor the progress, use the `kubectl get service` command with the `watch` parameter:

```
$ kubectl get service --watch

NAME           TYPE      CLUSTER-IP   EXTERNAL-IP     PORT(S)        AGE
example-java-j  LoadBalanc  10.0.141.72  <pending>    80:32150/TCP  2m
...
example-java-j  LoadBalanc  10.0.141.72  52.175.224.118  80:32150/TCP  7m
```

Navigate to the load balancer of your application in a browser using the `EXTERNAL-IP` to see the sample application. In the above example, the IP is `52.175.224.118`.

## Iterate on the application

You can iterate your application by making changes locally and rerunning `draft up`.

Update the message returned on [line 7 of src/main/java/helloworld/Hello.java](#)

```
public static void main(String[] args) {
    get("/", (req, res) -> "Hello World, I'm Java in AKS!");
}
```

Run the `draft up` command to redeploy the application:

```
$ draft up

Draft Up Started: 'example-java': 01CMZC9RF0T7T7XPWGFCJE15X4
example-java: Building Docker Image: SUCCESS (25.0202s)
example-java: Pushing Docker Image: SUCCESS (7.1457s)
example-java: Releasing Application: SUCCESS (3.5773s)
Inspect the logs with `draft logs 01CMZC9RF0T7T7XPWGFCJE15X4`
```

To see the updated application, navigate to the IP address of your load balancer again and verify your changes appear.

## Delete the cluster

When the cluster is no longer needed, use the [az group delete](#) command to remove the resource group, the AKS cluster, the container registry, the container images stored there, and all related resources.

```
az group delete --name MyResourceGroup --yes --no-wait
```

### NOTE

When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see [AKS service principal considerations and deletion](#).

## Next steps

For more information about using Draft, see the Draft documentation on GitHub.

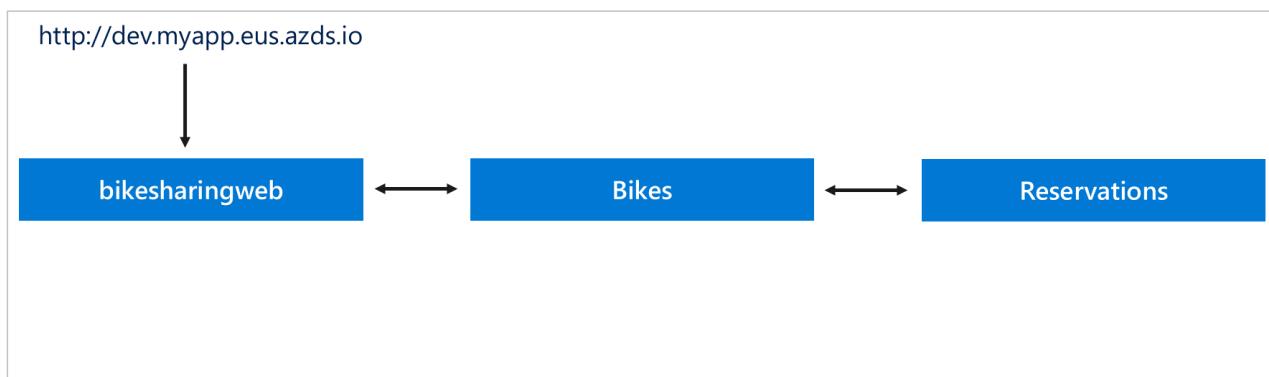
[Draft documentation](#)

# Quickstart: Team development on Kubernetes - Azure Dev Spaces

2/25/2020 • 6 minutes to read • [Edit Online](#)

In this guide, you will learn how to:

- Set up Azure Dev Spaces on a managed Kubernetes cluster in Azure.
- Deploy a large application with multiple microservices to a dev space.
- Test a single microservice in an isolated dev space within the context of the full application.



## Prerequisites

- An Azure subscription. If you don't have an Azure subscription, you can create a [free account](#).
- [Azure CLI installed](#).
- [Helm 3 installed](#).

## Create an Azure Kubernetes Service cluster

You must create an AKS cluster in a [supported region](#). The below commands create a resource group called *MyResourceGroup* and an AKS cluster called *MyAKS*.

```
az group create --name MyResourceGroup --location eastus
az aks create -g MyResourceGroup -n MyAKS --location eastus --generate-ssh-keys
```

## Enable Azure Dev Spaces on your AKS cluster

Use the `use-dev-spaces` command to enable Dev Spaces on your AKS cluster and follow the prompts. The below command enables Dev Spaces on the *MyAKS* cluster in the *MyResourceGroup* group and creates a dev space called *dev*.

### NOTE

The `use-dev-spaces` command will also install the Azure Dev Spaces CLI if its not already installed. You cannot install the Azure Dev Spaces CLI in the Azure Cloud Shell.

```
az aks use-dev-spaces -g MyResourceGroup -n MyAKS --space dev --yes
```

# Get sample application code

In this article, you use the [Azure Dev Spaces Bike Sharing sample application](#) to demonstrate using Azure Dev Spaces.

Clone the application from GitHub and navigate into its directory:

```
git clone https://github.com/Azure/dev-spaces  
cd dev-spaces/samples/BikeSharingApp/
```

## Retrieve the HostSuffix for *dev*

Use the `azds show-context` command to show the HostSuffix for *dev*.

```
$ azds show-context  
  
Name          ResourceGroup     DevSpace  HostSuffix  
-----  
MyAKS        MyResourceGroup   dev       fedcab0987.eus.azds.io
```

## Update the Helm chart with your HostSuffix

Open [charts/values.yaml](#) and replace all instances of `<REPLACE_ME_WITH_HOST_SUFFIX>` with the HostSuffix value you retrieved earlier. Save your changes and close the file.

## Run the sample application in Kubernetes

The commands for running the sample application on Kubernetes are part of an existing process and have no dependency on Azure Dev Spaces tooling. In this case, Helm is the tooling used to run this sample application but other tooling could be used to run your entire application in a namespace within a cluster. The Helm commands are targeting the *dev* space named *dev* you created earlier, but this *dev* space is also a Kubernetes namespace. As a result, *dev* spaces can be targeted by other tooling the same as other namespaces.

You can use Azure Dev Spaces for team development after an application is running in a cluster regardless of the tooling used to deploy it.

Use the `helm install` command to set up and install the sample application on your cluster.

```
cd charts/  
helm install bikesharingsampleappsampleapp . --dependency-update --namespace dev --atomic
```

The `helm install` command may take several minutes to complete. After the sample application is installed on your cluster and since you have Dev Spaces enabled on your cluster, use the `azds list-uris` command to display the URLs for the sample application in *dev* that is currently selected.

```
$ azds list-uris  
Uri                      Status  
-----  
http://dev.bikesharingweb.fedcab0987.eus.azds.io/ Available  
http://dev.gateway.fedcab0987.eus.azds.io/      Available
```

Navigate to the *bikesharingweb* service by opening the public URL from the `azds list-uris` command. In the above example, the public URL for the *bikesharingweb* service is

<http://dev.bikesharingweb.fedcab0987.eus.azds.io/>. Select *Aurelia Briggs (customer)* as the user. Verify you see the text *Hi Aurelia Briggs | Sign Out* at the top.

The screenshot shows a web browser window with the title "Adventure Works Cycles". The URL in the address bar is "http://azureuser2.s.dev.bikesharingweb.fedcab0987.eus.azds.io/". The page content is as follows:

**Bikes available in Seattle area**  
A selection of bikes that are best suited for your preferences.

Bike Type	Location	Price (\$/hour)
Women's Cruiser	1907 18th Ave S, Seattle, WA 98144	\$1/hour
Men's Cruiser	283 NW Market St, Seattle, WA 98107	\$1/hour
Women's Cruiser	1302 Market St, Kirkland, WA 98033	\$1/hour
Men's Cruiser	8638 22nd Ave SW, Seattle, WA 98106	\$1/hour
Men's Comfort	3401 California Ave SW, Seattle, WA 98116	\$1.5/hour
Women's Racer	14505 NE 91st St, Redmond, WA 98052	\$2/hour
Men's Cruiser	8431 SE 39th St, Mercer Island, WA 98040	\$1/hour
Girl's Cruiser	500 17th Ave, Seattle, WA 98122	\$1/hour
Women's Cruiser	8049 18th Ave NW, Seattle, WA 98117	\$1/hour
Men's Comfort	2100 Queen Anne Ave N, Seattle, WA 98109	\$1.5/hour
Men's Racing	7516 135th Ave SE, Newcastle, WA 98059	\$2/hour

## Create child dev spaces

Use the `azds space select` command to create two child spaces under *dev*:

```
azds space select -n dev/azureuser1 -y  
azds space select -n dev/azureuser2 -y
```

The above commands create two child spaces under *dev* named *azureuser1* and *azureuser2*. These two child spaces represent distinct dev spaces for developers *azureuser1* and *azureuser2* to use for making changes to the sample application.

Use the `azds space list` command to list all the dev spaces and confirm *dev/azureuser2* is selected.

```
$ azds space list  
  Name          DevSpacesEnabled  
  -----  
  default      False  
  dev          True  
  dev/azureuser1 True  
 * dev/azureuser2 True
```

Use the `azds list-uris` to display the URLs for the sample application in the currently selected space that is *dev/azureuser2*.

```
$ azds list-uris
Uri                                         Status
-----
http://azureuser2.s.dev.bikesharingweb.fedcab0987.eus.azds.io/ Available
http://azureuser2.s.dev.gateway.fedcab0987.eus.azds.io/     Available
```

Confirm that the URLs displayed by the `azds list-uris` command have the `azureuser2.s.dev` prefix. This prefix confirms that the current space selected is `azureuser2`, which is a child of `dev`.

Navigate to the `bikesharingweb` service for the `dev/azureuser2` dev space by opening the public URL from the `azds list-uris` command. In the above example, the public URL for the `bikesharingweb` service is `http://azureuser2.s.dev.bikesharingweb.fedcab0987.eus.azds.io/`. Select *Aurelia Briggs (customer)* as the user. Verify you see the text *Hi Aurelia Briggs | Sign out* at the top.

## Update code

Open `BikeSharingWeb/components/Header.js` with a text editor and change the text in the `span` element with the `userSignOut` `className`.

```
<span className="userSignOut">
  <Link href="/devsignin"><span tabIndex="0">Welcome {props.userName} | Sign out</span></Link>
</span>
```

Save your changes and close the file.

## Build and run the updated bikesharingweb service in the `dev/azureuser2` dev space

Navigate to the `BikeSharingWeb/` directory and run the `azds up` command.

```
$ cd ../BikeSharingWeb/
$ azds up

Using dev space 'dev/azureuser2' with target 'MyAKS'
Synchronizing files...2s
...
Service 'bikesharingweb' port 'http' is available at
http://azureuser2.s.dev.bikesharingweb.fedcab0987.eus.azds.io/
Service 'bikesharingweb' port 80 (http) is available at http://localhost:54256
...
```

This command builds and runs the `bikesharingweb` service in the `dev/azureuser2` dev space. This service runs in addition to the `bikesharingweb` service running in `dev` and is only used for requests with the `azureuser2.s` URL prefix. For more information on how routing works between parent and child dev spaces, see [How Azure Dev Spaces works and is configured](#).

Navigate to the `bikesharingweb` service for the `dev/azureuser2` dev space by opening the public URL displayed in the output of the `azds up` command. Select *Aurelia Briggs (customer)* as the user. Verify you see the updated text in the upper right corner. You may need to refresh the page or clear your browser's cache if you do not immediately see this change.



Welcome Aurelia Briggs | Sign out

#### NOTE

When you navigate to your service while running `azds up`, the HTTP request traces are also displayed in the output of the `azds up` command. These traces can help you troubleshoot and debug your service. You can disable these traces using `--disable-http-traces` when running `azds up`.

## Verify other Dev Spaces are unchanged

If the `azds up` command is still running, press *Ctrl+c*.

```
$ azds list-uris --all
Uri                                         Status
-----
http://azureuser1.s.dev.bikesharingweb.fedcab0987.eus.azds.io/ Available
http://azureuser1.s.dev.gateway.fedcab0987.eus.azds.io/     Available
http://azureuser2.s.dev.bikesharingweb.fedcab0987.eus.azds.io/ Available
http://azureuser2.s.dev.gateway.fedcab0987.eus.azds.io/     Available
http://dev.bikesharingweb.fedcab0987.eus.azds.io/          Available
http://dev.gateway.fedcab0987.eus.azds.io/                 Available
```

Navigate to the *dev* version of *bikesharingweb* in your browser, choose *Aurelia Briggs (customer)* as the user, and verify you see the original text in the upper right corner. Repeat these steps with the *dev/azureuser1* URL. Notice the changes are only applied to the *dev/azureuser2* version of *bikesharingweb*. This isolation of changes to *dev/azureuser2* allows for *azureuser2* to make changes without affecting *azureuser1*.

To have these changes reflected in *dev* and *dev/azureuser1*, you should follow your team's existing workflow or CI/CD pipeline. For example, this workflow may involve committing your change to your version control system and deploying the update using a CI/CD pipeline or tooling such as Helm.

## Clean up your Azure resources

```
az group delete --name MyResourceGroup --yes --no-wait
```

## Next steps

Learn how Azure Dev Spaces helps you develop more complex apps across multiple containers, and how you can simplify collaborative development by working with different versions or branches of your code in different spaces.

[Working with multiple containers and team development](#)

# Quickstart: Debug and iterate on Kubernetes: Visual Studio Code and .NET Core - Azure Dev Spaces

2/25/2020 • 5 minutes to read • [Edit Online](#)

In this guide, you will learn how to:

- Set up Azure Dev Spaces with a managed Kubernetes cluster in Azure.
- Iteratively develop code in containers using Visual Studio Code.
- Debug the code in your dev space from Visual Studio Code.

Azure Dev Spaces also allows you to debug and iterate using:

- [Java and Visual Studio Code](#)
- [Node.js and Visual Studio Code](#)
- [.NET Core and Visual Studio](#)

## Prerequisites

- An Azure subscription. If you don't have one, you can create a [free account](#).
- [Visual Studio Code installed](#).
- The [Azure Dev Spaces](#) and [C#](#) extensions for Visual Studio Code installed.
- [Azure CLI installed](#).

## Create an Azure Kubernetes Service cluster

You need to create an AKS cluster in a [supported region](#). The below commands create a resource group called *MyResourceGroup* and an AKS cluster called *MyAKS*.

```
az group create --name MyResourceGroup --location eastus
az aks create -g MyResourceGroup -n MyAKS --location eastus --generate-ssh-keys
```

## Enable Azure Dev Spaces on your AKS cluster

Use the `use-dev-spaces` command to enable Dev Spaces on your AKS cluster and follow the prompts. The below command enables Dev Spaces on the *MyAKS* cluster in the *MyResourceGroup* group and creates a *default* dev space.

### NOTE

The `use-dev-spaces` command will also install the Azure Dev Spaces CLI if its not already installed. You cannot install the Azure Dev Spaces CLI in the Azure Cloud Shell.

```
$ az aks use-dev-spaces -g MyResourceGroup -n MyAKS

'An Azure Dev Spaces Controller' will be created that targets resource 'MyAKS' in resource group
'MyResourceGroup'. Continue? (y/N): y

Creating and selecting Azure Dev Spaces Controller 'MyAKS' in resource group 'MyResourceGroup' that targets
resource 'MyAKS' in resource group 'MyResourceGroup'...2m 24s

Select a dev space or Kubernetes namespace to use as a dev space.
[1] default
Type a number or a new name: 1

Kubernetes namespace 'default' will be configured as a dev space. This will enable Azure Dev Spaces
instrumentation for new workloads in the namespace. Continue? (Y/n): Y

Configuring and selecting dev space 'default'...3s

Managed Kubernetes cluster 'MyAKS' in resource group 'MyResourceGroup' is ready for development in dev space
'default'. Type `azds prep` to prepare a source directory for use with Azure Dev Spaces and `azds up` to run.
```

## Get sample application code

In this article, you use the [Azure Dev Spaces sample application](#) to demonstrate using Azure Dev Spaces.

Clone the application from GitHub.

```
git clone https://github.com/Azure/dev-spaces
```

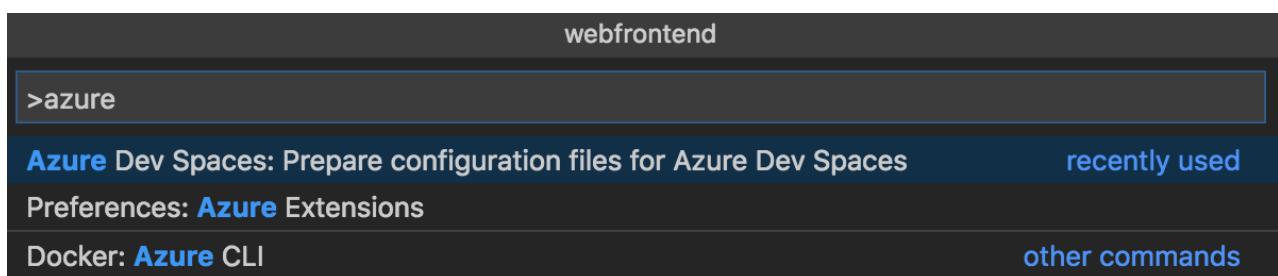
## Prepare the sample application in Visual Studio Code

Open Visual Studio Code, click *File* then *Open...*, navigate to the *dev-spaces/samples/dotnetcore/getting-started/webfrontend* directory, and click *Open*.

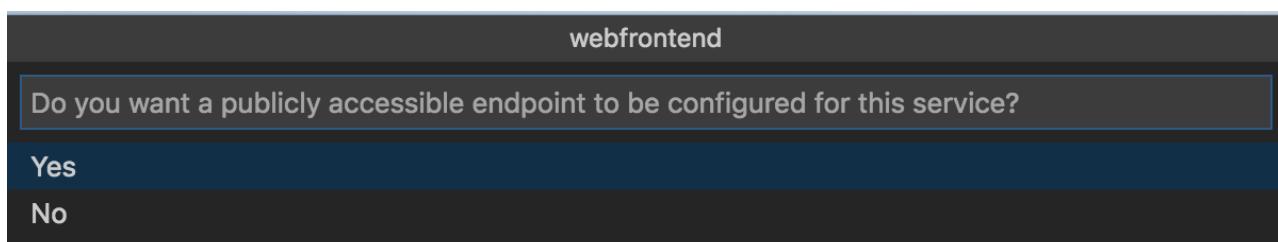
You now have the *webfrontend* project open in Visual Studio Code. To run the application in your dev space, generate the Docker and Helm chart assets using the Azure Dev Spaces extension in the Command Palette.

To open the Command Palette in Visual Studio Code, click *View* then *Command Palette*. Begin typing

`Azure Dev Spaces` and click on `Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces`.



When Visual Studio Code also prompts you to configure your public endpoint, choose `Yes` to enable a public endpoint.



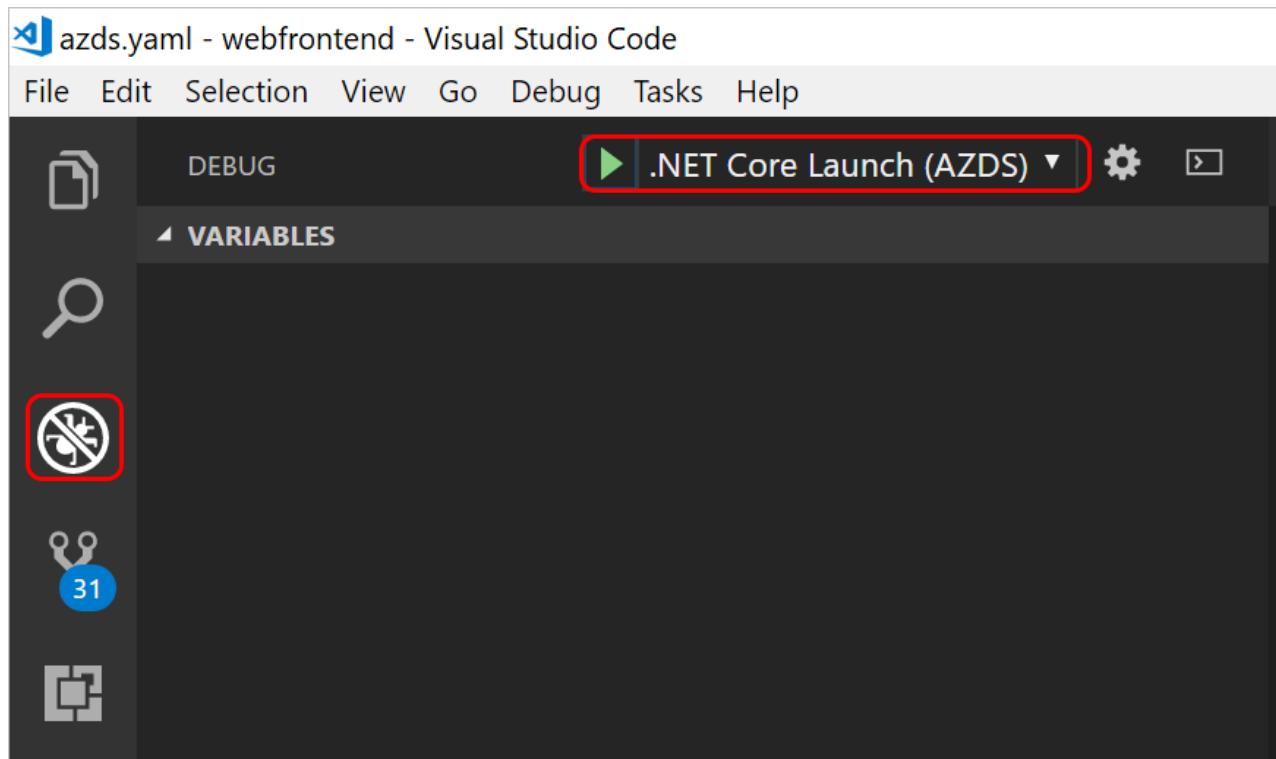
This command prepares your project to run in Azure Dev Spaces by generating a Dockerfile and Helm chart. It also generates a .vscode directory with debugging configuration at the root of your project.

#### TIP

The [Dockerfile and Helm chart](#) for your project is used by Azure Dev Spaces to build and run your code, but you can modify these files if you want to change how the project is built and ran.

## Build and run code in Kubernetes from Visual Studio Code

Click on the *Debug* icon on the left and click *.NET Core Launch (AZDS)* at the top.



This command builds and runs your service in Azure Dev Spaces in debugging mode. The *Terminal* window at the bottom shows the build output and URLs for your service running in Azure Dev Spaces. The *Debug Console* shows the log output.

#### NOTE

If you don't see any Azure Dev Spaces commands in the *Command Palette*, make sure you have installed the [Visual Studio Code extension for Azure Dev Spaces](#). Also verify you opened the `dev-spaces/samples/dotnetcore/getting-started/webfrontend` directory in Visual Studio Code.

You can see the service running by opening the public URL.

Click *Debug* then *Stop Debugging* to stop the debugger.

## Update code

To deploy an updated version of your service, you can update any file in your project and rerun *.NET Core Launch (AZDS)*. For example:

1. If your application is still running, click *Debug* then *Stop Debugging* to stop it.
2. Update line 22 in `Controllers/HomeController.cs` to:

```
ViewData["Message"] = "Your application description page in Azure.";
```

3. Save your changes.
4. Rerun .NET Core Launch (AZDS).
5. Navigate to your running service and click *About*.
6. Observe your changes.
7. Click *Debug* then *Stop Debugging* to stop your application.

## Setting and using breakpoints for debugging

Start your service in debugging mode using *.NET Core Launch (AZDS)*.

Navigate back to the *Explorer* view by clicking *View* then *Explorer*. Open `Controllers/HomeController.cs` and click somewhere on line 22 to put your cursor there. To set a breakpoint hit *F9* or click *Debug* then *Toggle Breakpoint*.

Open your service in a browser and notice no message is displayed. Return to Visual Studio Code and observe line 20 is highlighted. The breakpoint you set has paused the service at line 20. To resume the service, hit *F5* or click *Debug* then *Continue*. Return to your browser and notice the message is now displayed.

While running your service in Kubernetes with a debugger attached, you have full access to debug information such as the call stack, local variables, and exception information.

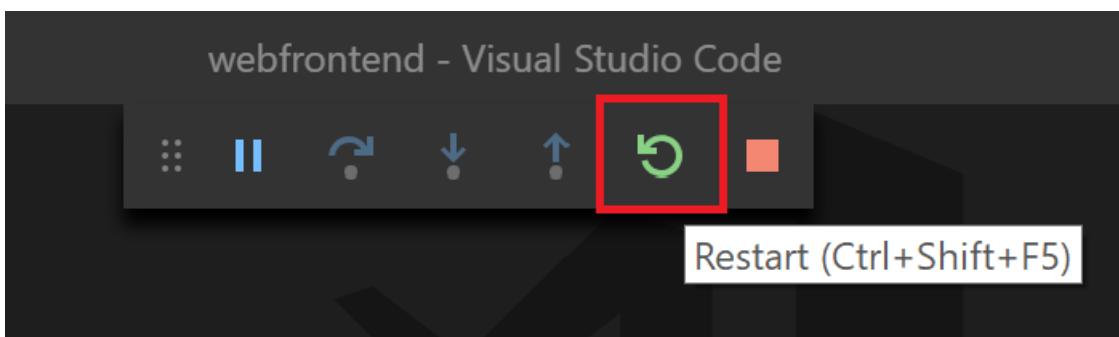
Remove the breakpoint by putting your cursor on line 22 in `Controllers/HomeController.cs` and hitting *F9*.

## Update code from Visual Studio Code

While the service is running in debugging mode, update line 22 in `Controllers/HomeController.cs`. For example:

```
ViewData["Message"] = "Your application description page in Azure while debugging!";
```

Save the file. Click *Debug* then *Restart Debugging* or in the *Debug toolbar*, click the *Restart Debugging* button.



Open your service in a browser and notice your updated message is displayed.

Instead of rebuilding and redeploying a new container image each time code edits are made, Azure Dev Spaces incrementally recompiles code within the existing container to provide a faster edit/debug loop.

## Clean up your Azure resources

```
az group delete --name MyResourceGroup --yes --no-wait
```

## Next steps

Learn how Azure Dev Spaces helps you develop more complex applications across multiple containers, and how you can simplify collaborative development by working with different versions or branches of your code in different spaces.

[Working with multiple containers and team development](#)

# Quickstart: Debug and iterate on Kubernetes: Visual Studio & .NET Core - Azure Dev Spaces

1/8/2020 • 3 minutes to read • [Edit Online](#)

In this guide, you will learn how to:

- Set up Azure Dev Spaces with a managed Kubernetes cluster in Azure.
- Iteratively develop code in containers using Visual Studio.
- Debug code running in your cluster using Visual Studio.

Azure Dev Spaces also allows you debug and iterate using:

- [Java and Visual Studio Code](#)
- [Node.js and Visual Studio Code](#)
- [.NET Core and Visual Studio Code](#)

## Prerequisites

- An Azure subscription. If you don't have one, you can create a [free account](#).
- Visual Studio 2019 on Windows with the Azure Development workload installed. You can also use Visual Studio 2017 on Windows with the Web Development workload and [Visual Studio Tools for Kubernetes](#) installed. If you don't have Visual Studio installed, download it [here](#).

## Create an Azure Kubernetes Service cluster

You must create an AKS cluster in a [supported region](#). To create a cluster:

1. Sign in to the [Azure portal](#)
2. Select + *Create a resource > Kubernetes Service*.
3. Enter the *Subscription, Resource Group, Kubernetes cluster name, Region, Kubernetes version, and DNS name prefix*.

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. [Learn more about Azure Kubernetes Service](#)

**Project details**

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

**Subscription** \* ⓘ

**Resource group** \* ⓘ  (New) MyResourceGroup [Create new](#)

**Cluster details**

**Kubernetes cluster name** \* ⓘ  MyAKS

**Region** \* ⓘ  (US) East US

**Kubernetes version** \* ⓘ  1.13.12 (default)

**DNS name prefix** \* ⓘ  MyAKS-dns

**Primary node pool**

**Review + create** [< Previous](#) [Next : Scale >](#)

4. Click *Review + create*.

5. Click *Create*.

## Enable Azure Dev Spaces on your AKS cluster

Navigate to your AKS cluster in the Azure portal and click *Dev Spaces*. Change *Use Dev Spaces* to *Yes* and click *Save*.

Home > MyAKS - Dev Spaces

## MyAKS - Dev Spaces

Kubernetes service

Search (Ctrl+ /)

- [Overview](#)
- [Activity log](#)
- [Access control \(IAM\)](#)
- [Tags](#)
- [Diagnose and solve problems](#)

---

### Settings

- [Node pools](#)
- [Upgrade](#)
- [Scale](#)
- [Networking](#)
- [Dev Spaces](#) (selected)
- [Deployment center \(preview\)](#)
- [Policies \(preview\)](#)
- [Properties](#)
- [Locks](#)
- [Export template](#)

---

### Monitoring

Save Discard

## Use Dev Spaces

No Yes

Azure Dev Spaces is a free capability for your AKS cluster that allows you to build and debug services in a cloud-native application without needing to replicate or mock their dependencies. It also reduces the complexity of collaborating with your team in a shared AKS cluster and provides tools for running and debugging containers directly in AKS.

### Dev Spaces enables

- ✓ Dev Spaces Connect (Preview)
- ✓ Deploying review apps with GitHub Actions pull request flow (Preview)
- ✓ Remote debugging
- ✓ Team collaboration
- ✓ Configuring your project to run in the cloud
- ✓ Quickly running your code in AKS

### Resources

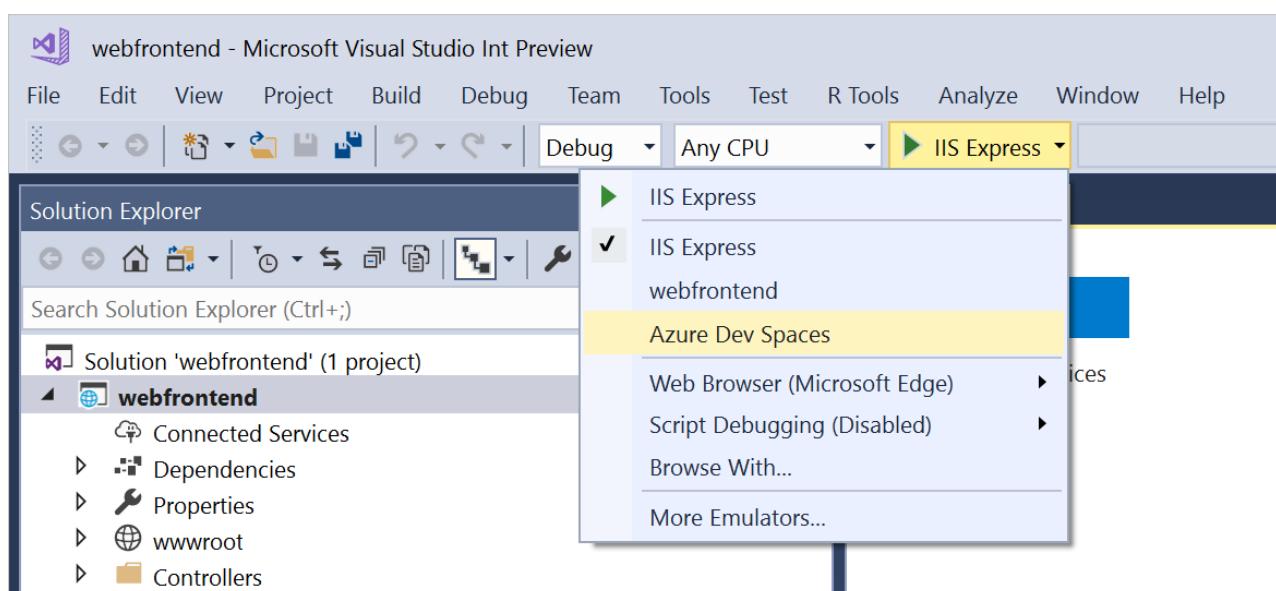
- [Dev Spaces documentation, including Quickstarts](#)
- [Install Azure Dev Spaces clients](#)
- [Issues in the Dev Spaces GitHub repository](#)
- [Dev Spaces troubleshooting page](#)

## Create a new ASP.NET web app

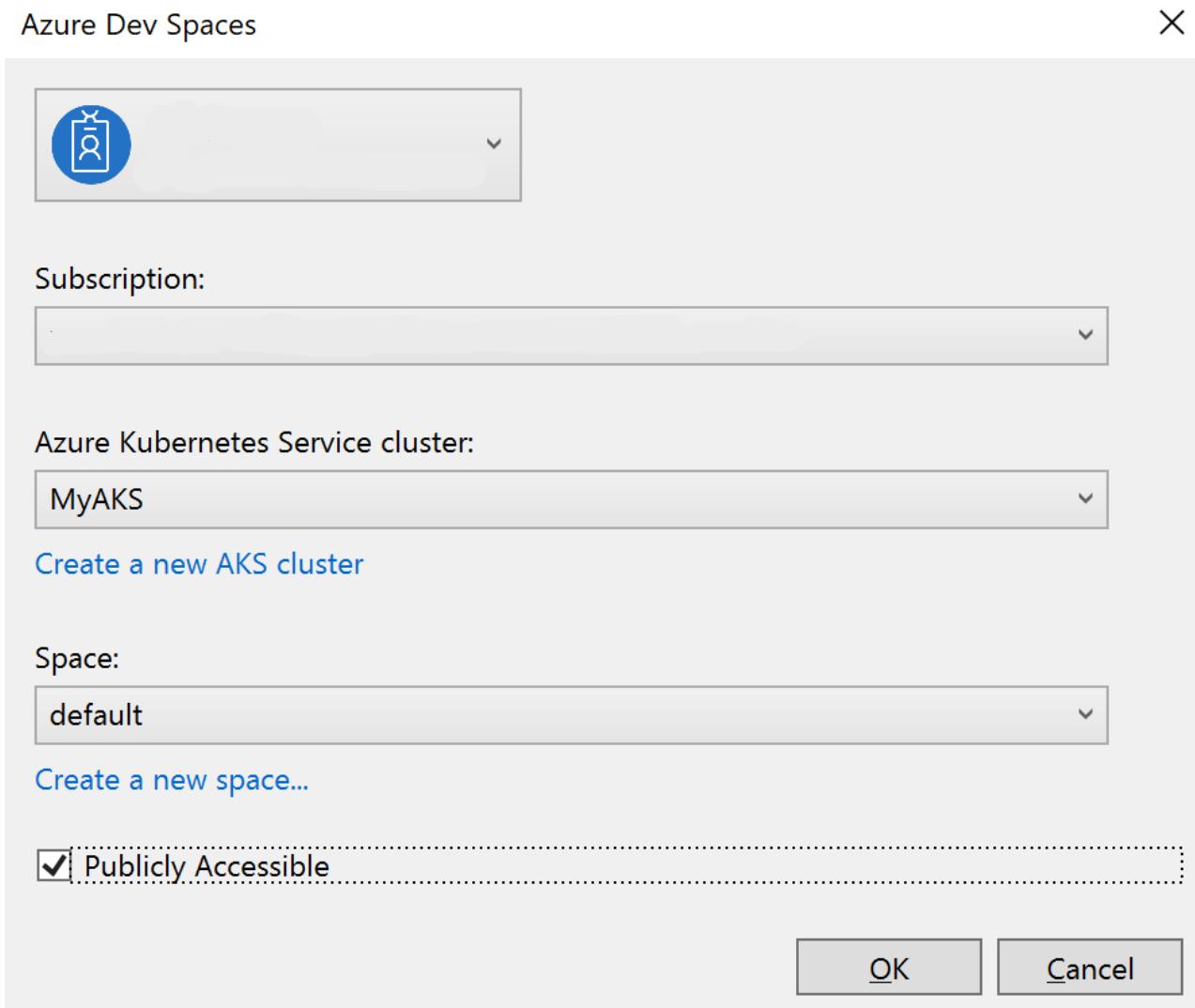
1. Open Visual Studio.
2. Create a new project.
3. Choose *ASP.NET Core Web Application* and click *Next*.
4. Name your project *webfrontend* and click *Create*.
5. When prompted, choose *Web Application (Model-View-Controller)* for the template.
6. Select *.NET Core* and *ASP.NET Core 2.1* at the top.
7. Click *Create*.

## Connect your project to your dev space

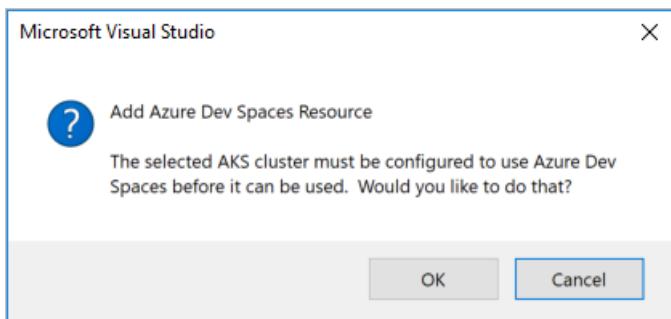
In your project, select **Azure Dev Spaces** from the launch settings dropdown as shown below.



In the Azure Dev Spaces dialog, select your *Subscription* and *Azure Kubernetes Cluster*. Leave *Space* set to *default* and enable the *Publicly Accessible* checkbox. Click *OK*.



This process deploys your service to the *default* dev space with a publicly accessible URL. If you choose a cluster that hasn't been configured to work with Azure Dev Spaces, you'll see a message asking if you want to configure it. Click *OK*.



The public URL for the service running in the *default* dev space is displayed in the *Output* window:

```
Starting warmup for project 'webfrontend'.
Waiting for namespace to be provisioned.
Using dev space 'default' with target 'MyAKS'
...
Successfully built 1234567890ab
Successfully tagged webfrontend:devspaces-11122233344455566
Built container image in 39s
Waiting for container...
36s

Service 'webfrontend' port 'http' is available at http://default.webfrontend.1234567890abcdef1234.eus.azds.io/
Service 'webfrontend' port 80 (http) is available at http://localhost:62266
Completed warmup for project 'webfrontend' in 125 seconds.
```

In the above example, the public URL is <http://default.webfrontend.1234567890abcdef1234.eus.azds.io/>. Navigate to your service's public URL and interact with the service running in your dev space.

This process may have disabled public access to your service. To enable public access, you can update the [ingress value in the values.yaml](#).

## Update code

If Visual Studio is still connected to your dev space, click the stop button. Change line 20 in `Controllers/HomeController.cs` to:

```
 ViewData["Message"] = "Your application description page in Azure.";
```

Save your changes and start your service using **Azure Dev Spaces** from the launch settings dropdown. Open the public URL of your service in a browser and click *About*. Observe that your updated message appears.

Instead of rebuilding and redeploying a new container image each time code edits are made, Azure Dev Spaces incrementally recompiles code within the existing container to provide a faster edit/debug loop.

## Setting and using breakpoints for debugging

If Visual Studio is still connected to your dev space, click the stop button. Open `Controllers/HomeController.cs` and click somewhere on line 20 to put your cursor there. To set a breakpoint hit *F9* or click *Debug* then *Toggle Breakpoint*. To start your service in debugging mode in your dev space, hit *F5* or click *Debug* then *Start Debugging*.

Open your service in a browser and notice no message is displayed. Return to Visual Studio and observe line 20 is highlighted. The breakpoint you set has paused the service at line 20. To resume the service, hit *F5* or click *Debug* then *Continue*. Return to your browser and notice the message is now displayed.

While running your service in Kubernetes with a debugger attached, you have full access to debug information such as the call stack, local variables, and exception information.

Remove the breakpoint by putting your cursor on line 20 in `Controllers/HomeController.cs` and hitting *F9*.

## Clean up your Azure resources

Navigate to your resource group in the Azure portal and click *Delete resource group*. Alternatively, you can use the [az aks delete](#) command:

```
az group delete --name MyResourceGroup --yes --no-wait
```

## Next steps

[Working with multiple containers and team development](#)

# Quickstart: Develop an application on Kubernetes - Azure Dev Spaces

2/25/2020 • 4 minutes to read • [Edit Online](#)

In this guide, you will learn how to:

- Set up Azure Dev Spaces with a managed Kubernetes cluster in Azure.
- Develop and run code in containers using the command line.

## Prerequisites

- An Azure subscription. If you don't have an Azure subscription, you can create a [free account](#).
- [Azure CLI installed](#).

## Create an Azure Kubernetes Service cluster

You need to create an AKS cluster in a [supported region](#). The below commands create a resource group called *MyResourceGroup* and an AKS cluster called *MyAKS*.

```
az group create --name MyResourceGroup --location eastus
az aks create -g MyResourceGroup -n MyAKS --location eastus --generate-ssh-keys
```

## Enable Azure Dev Spaces on your AKS cluster

Use the `use-dev-spaces` command to enable Dev Spaces on your AKS cluster and follow the prompts. The below command enables Dev Spaces on the *MyAKS* cluster in the *MyResourceGroup* group and creates a *default* dev space.

### NOTE

The `use-dev-spaces` command will also install the Azure Dev Spaces CLI if its not already installed. You cannot install the Azure Dev Spaces CLI in the Azure Cloud Shell.

```
$ az aks use-dev-spaces -g MyResourceGroup -n MyAKS

'An Azure Dev Spaces Controller' will be created that targets resource 'MyAKS' in resource group
'MyResourceGroup'. Continue? (y/N): y

Creating and selecting Azure Dev Spaces Controller 'MyAKS' in resource group 'MyResourceGroup' that targets
resource 'MyAKS' in resource group 'MyResourceGroup'...2m 24s

Select a dev space or Kubernetes namespace to use as a dev space.
[1] default
Type a number or a new name: 1

Kubernetes namespace 'default' will be configured as a dev space. This will enable Azure Dev Spaces
instrumentation for new workloads in the namespace. Continue? (Y/n): Y

Configuring and selecting dev space 'default'...3s

Managed Kubernetes cluster 'MyAKS' in resource group 'MyResourceGroup' is ready for development in dev space
'default'. Type `azds prep` to prepare a source directory for use with Azure Dev Spaces and `azds up` to run.
```

## Get sample application code

In this article, you use the [Azure Dev Spaces sample application](#) to demonstrate using Azure Dev Spaces.

Clone the application from GitHub and navigate into the `dev-spaces/samples/nodejs/getting-started/webfrontend` directory:

```
git clone https://github.com/Azure/dev-spaces
cd dev-spaces/samples/nodejs/getting-started/webfrontend
```

## Prepare the application

In order to run your application on Azure Dev Spaces, you need a Dockerfile and Helm chart. For some languages, such as [Java](#), [.NET core](#), and [Node.js](#), the Azure Dev Spaces client tooling can generate all the assets you need. For many other languages, such as Go, PHP, and Python, the client tooling can generate the Helm chart as long as you can provide a valid Dockerfile.

Generate the Docker and Helm chart assets for running the application in Kubernetes using the `azds prep` command:

```
azds prep --enable-ingress
```

You must run the `prep` command from the `dev-spaces/samples/nodejs/getting-started/webfrontend` directory to correctly generate the Docker and Helm chart assets.

### TIP

The `prep` command attempts to generate a [Dockerfile](#) and [Helm chart](#) for your project. Azure Dev Spaces uses these files to build and run your code, but you can modify these files if you want to change how the project is built and ran.

## Build and run code in Kubernetes

Build and run your code in AKS using the `azds up` command:

```
$ azds up
Using dev space 'default' with target 'MyAKS'
Synchronizing files...2s
Installing Helm chart...2s
Waiting for container image build...2m 25s
Building container image...
Step 1/8 : FROM node
Step 2/8 : ENV PORT 80
Step 3/8 : EXPOSE 80
Step 4/8 : WORKDIR /app
Step 5/8 : COPY package.json .
Step 6/8 : RUN npm install
Step 7/8 : COPY . .
Step 8/8 : CMD ["npm", "start"]
Built container image in 6m 17s
Waiting for container...13s
Service 'webfrontend' port 'http' is available at http://webfrontend.1234567890abcdef1234.eus.azds.io/
Service 'webfrontend' port 80 (http) is available at http://localhost:54256
...
```

You can see the service running by opening the public URL, which is displayed in the output from the `azds up` command. In this example, the public URL is <http://webfrontend.1234567890abcdef1234.eus.azds.io/>.

#### NOTE

When you navigate to your service while running `azds up`, the HTTP request traces are also displayed in the output of the `azds up` command. These traces can help you troubleshoot and debug your service. You can disable these traces using `--disable-http-traces` when running `azds up`.

If you stop the `azds up` command using *Ctrl+c*, the service will continue to run in AKS, and the public URL will remain available.

## Update code

To deploy an updated version of your service, you can update any file in your project and rerun the `azds up` command. For example:

1. If `azds up` is still running, press *Ctrl+c*.

2. Update [line 13 in `server.js`](#) to:

```
res.send('Hello from webfrontend in Azure');
```

3. Save your changes.

4. Rerun the `azds up` command:

```
$ azds up
Using dev space 'default' with target 'MyAKS'
Synchronizing files...1s
Installing Helm chart...3s
Waiting for container image build...
...
```

5. Navigate to your running service and observe your changes.

6. Press *Ctrl+c* to stop the `azds up` command.

# Clean up your Azure resources

```
az group delete --name MyResourceGroup --yes --no-wait
```

## Next steps

Learn how Azure Dev Spaces helps you develop more complex applications across multiple containers, and how you can simplify collaborative development by working with different versions or branches of your code in different spaces.

[Team development in Azure Dev Spaces](#)

# Quickstart: Debug and iterate on Kubernetes with Visual Studio Code and Java - Azure Dev Spaces

2/25/2020 • 5 minutes to read • [Edit Online](#)

In this quickstart, you set up Azure Dev Spaces with a managed Kubernetes cluster, and use a Java app in Visual Studio Code to iteratively develop and debug code in containers. Azure Dev Spaces lets you debug and test all the components of your application in Azure Kubernetes Service (AKS) with minimal development machine setup.

## Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Java Development Kit \(JDK\) 1.8.0+](#).
- [Maven 3.5.0+](#).
- [Visual Studio Code](#).
- The [Azure Dev Spaces](#) and [Java Debugger for Azure Dev Spaces](#) extensions for Visual Studio Code.
- [Azure CLI](#).
- [Git](#).

## Create an Azure Kubernetes Service cluster

You need to create an AKS cluster in a [supported region](#). The following commands create a resource group called *MyResourceGroup* and an AKS cluster called *MyAKS*.

```
az group create --name MyResourceGroup --location eastus
az aks create -g MyResourceGroup -n MyAKS --location eastus --generate-ssh-keys
```

## Enable Azure Dev Spaces on your AKS cluster

Use the `use-dev-spaces` command to enable Dev Spaces on your AKS cluster and follow the prompts. The following command enables Dev Spaces on the *MyAKS* cluster in the *MyResourceGroup* group and creates a *default* dev space.

### NOTE

The `use-dev-spaces` command will also install the Azure Dev Spaces CLI if its not already installed. You can't install the Azure Dev Spaces CLI in the Azure Cloud Shell.

```
$ az aks use-dev-spaces -g MyResourceGroup -n MyAKS

'An Azure Dev Spaces Controller' will be created that targets resource 'MyAKS' in resource group
'MyResourceGroup'. Continue? (y/N): y

Creating and selecting Azure Dev Spaces Controller 'MyAKS' in resource group 'MyResourceGroup' that targets
resource 'MyAKS' in resource group 'MyResourceGroup'...2m 24s

Select a dev space or Kubernetes namespace to use as a dev space.
[1] default
Type a number or a new name: 1

Kubernetes namespace 'default' will be configured as a dev space. This will enable Azure Dev Spaces
instrumentation for new workloads in the namespace. Continue? (Y/n): Y

Configuring and selecting dev space 'default'...3s

Managed Kubernetes cluster 'MyAKS' in resource group 'MyResourceGroup' is ready for development in dev space
'default'. Type `azds prep` to prepare a source directory for use with Azure Dev Spaces and `azds up` to run.
```

## Get sample application code

In this article, you use the [Azure Dev Spaces sample application](#) to demonstrate using Azure Dev Spaces.

Clone the application from GitHub.

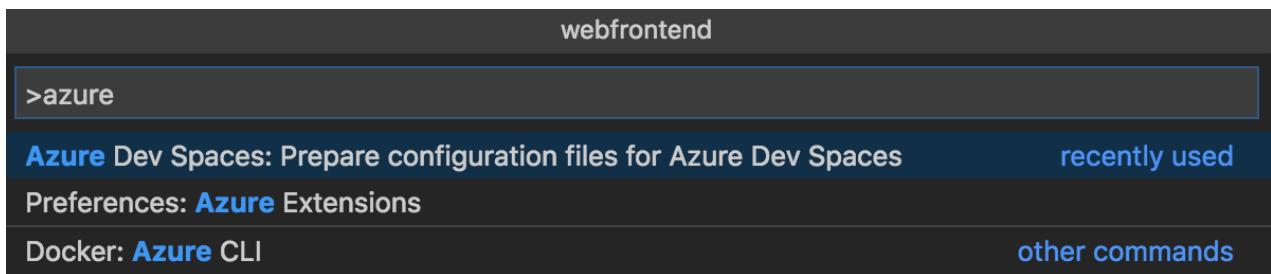
```
git clone https://github.com/Azure/dev-spaces
```

## Prepare the sample application in Visual Studio Code

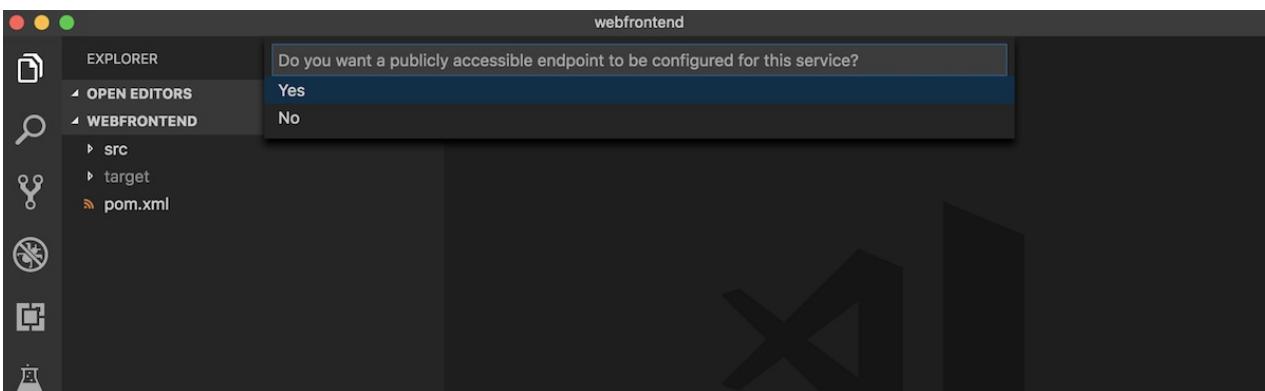
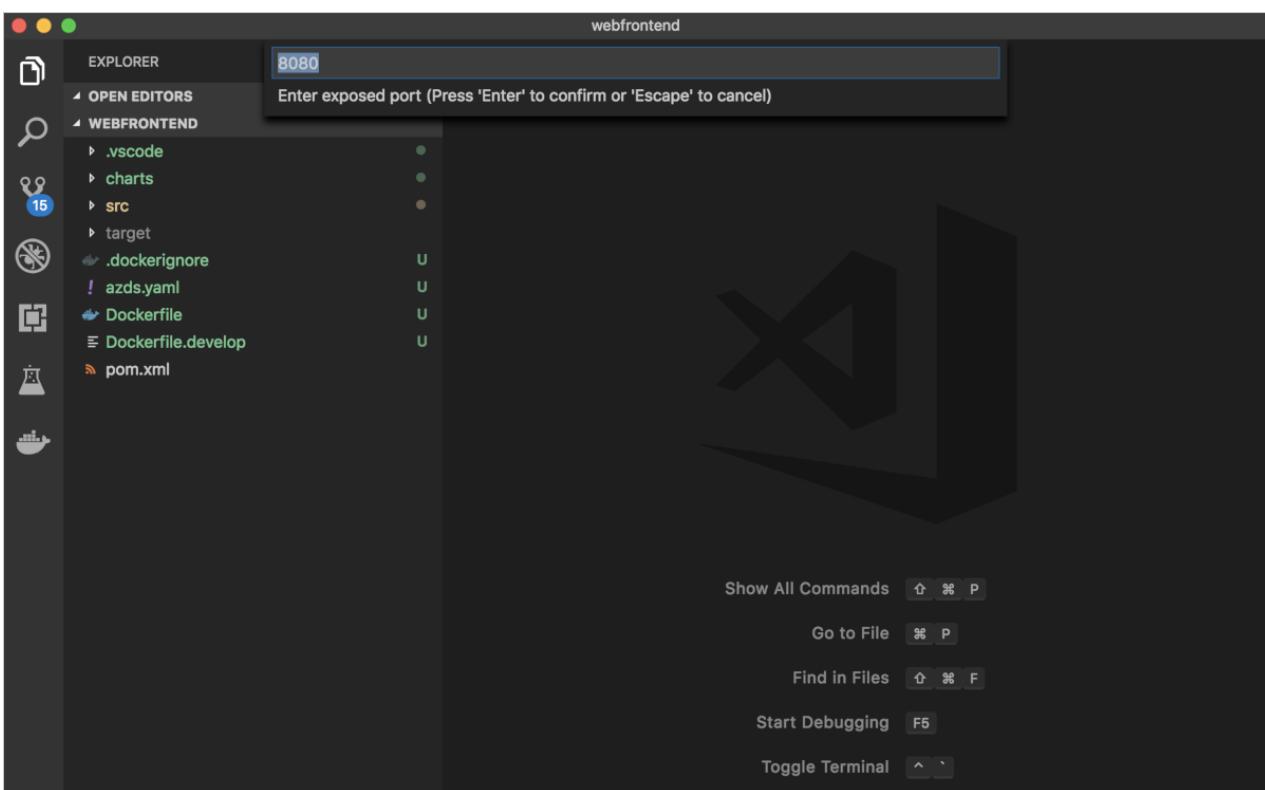
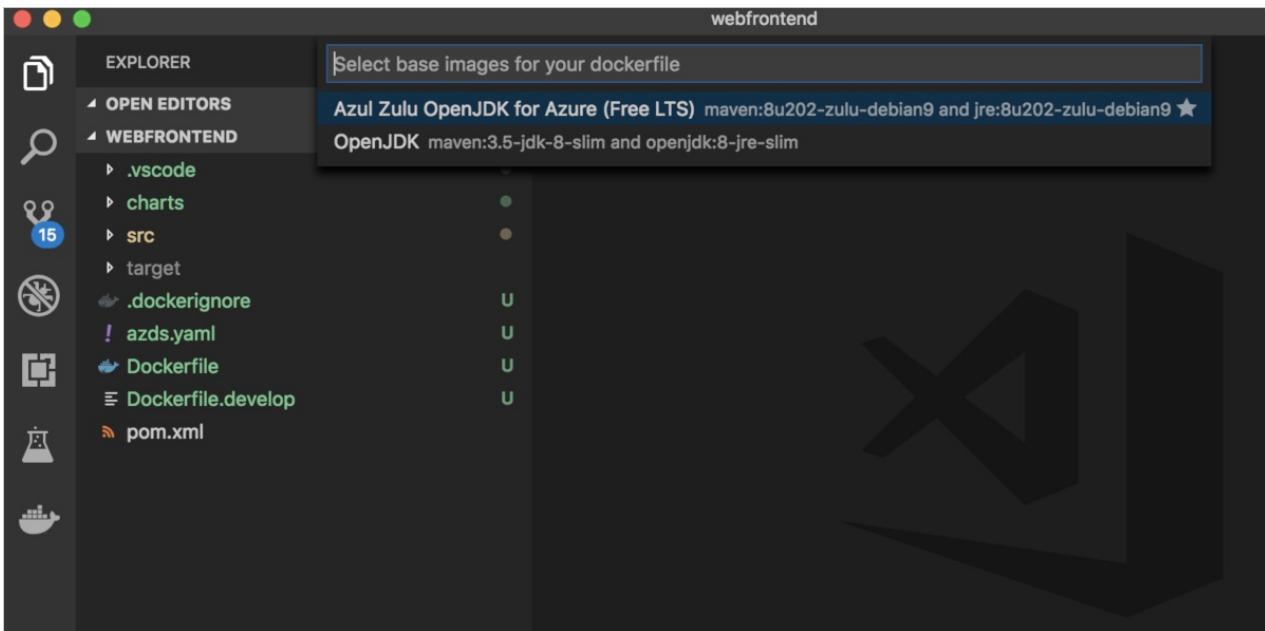
Open Visual Studio Code, select **File** then **Open**, navigate to the *dev-spaces/samples/java/getting-started/webfrontend* directory, and select **Open**.

You now have the *webfrontend* project open in Visual Studio Code. To run the application in your dev space, generate the Docker and Helm chart assets using the Azure Dev Spaces extension in the Command Palette.

To open the Command Palette in Visual Studio Code, select **View** then **Command Palette**. Begin typing **Azure Dev Spaces** and select **Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces**.



When Visual Studio Code also prompts you to configure your base images, exposed port and public endpoint, choose **Azul Zulu OpenJDK for Azure (Free LTS)** for the base image, **8080** for the exposed port, and **Yes** to enable a public endpoint.



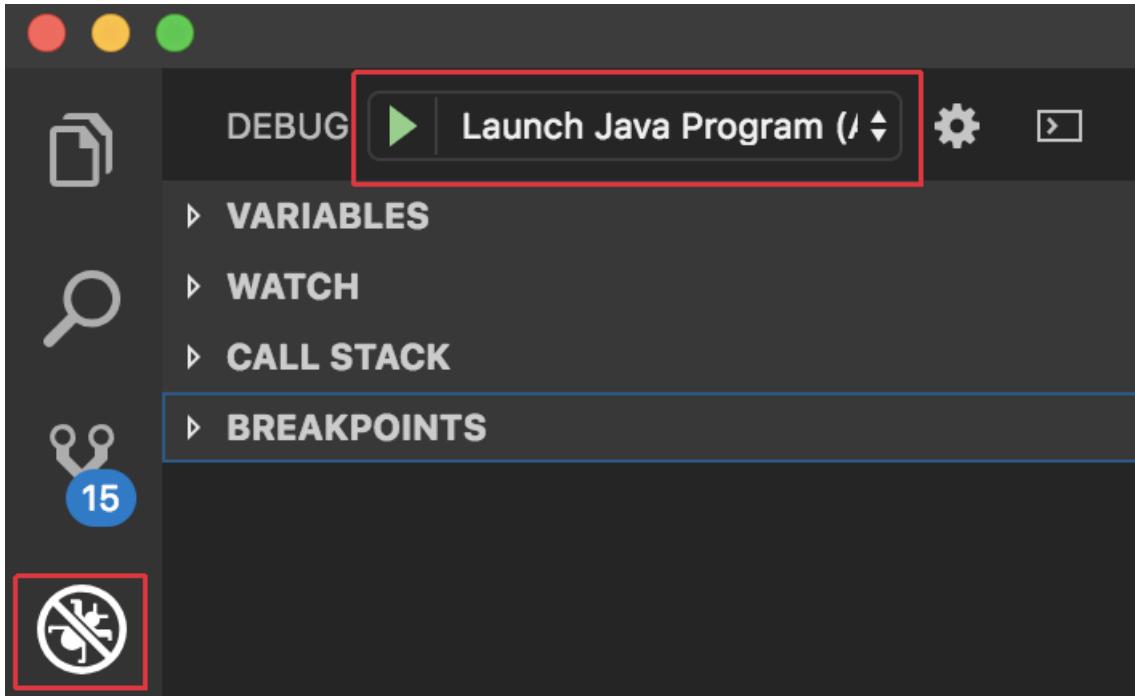
This command prepares your project to run in Azure Dev Spaces by generating a Dockerfile and Helm chart. It also generates a .vscode directory with debugging configuration at the root of your project.

#### TIP

The [Dockerfile](#) and [Helm chart](#) for your project is used by Azure Dev Spaces to build and run your code, but you can modify these files if you want to change how the project is built and run.

## Build and run code in Kubernetes from Visual Studio Code

Select the **Debug** icon on the left and select **Launch Java Program (AZDS)** at the top.



This command builds and runs your service in Azure Dev Spaces. The **Terminal** window at the bottom shows the build output and URLs for your service running Azure Dev Spaces. The **Debug Console** shows the log output.

#### NOTE

If you don't see any Azure Dev Spaces commands in the **Command Palette**, make sure you have installed the [Visual Studio Code extension for Azure Dev Spaces](#). Also verify you opened the `dev-spaces/samples/java/getting-started/webfrontend` directory in Visual Studio Code.

You can see the service running by opening the public URL.

Select **Debug** then **Stop Debugging** to stop the debugger.

## Update code

To deploy an updated version of your service, you can update any file in your project and rerun **Launch Java Program (AZDS)**. For example:

1. If your application is still running, select **Debug** then **Stop Debugging** to stop it.
2. Update line 19 in `src/main/java/com/ms/sample/webfrontend/Application.java` to:

```
return "Hello from webfrontend in Azure!";
```

3. Save your changes.

4. Rerun **Launch Java Program (AZDS)**.
5. Navigate to your running service and observe your changes.
6. Select **Debug** then **Stop Debugging** to stop your application.

## Setting and using breakpoints for debugging

Start your service using **Launch Java Program (AZDS)**. This also runs your service in debugging mode.

Navigate back to the **Explorer** view by selecting **View** then **Explorer**. Open `src/main/java/com/ms/sample/webfrontend/Application.java` and click somewhere on line 19 to put your cursor there. To set a breakpoint press **F9** or select **Debug** then **Toggle Breakpoint**.

Open your service in a browser and notice no message is displayed. Return to Visual Studio Code and observe line 19 is highlighted. The breakpoint you set has paused the service at line 19. To resume the service, press **F5** or select **Debug** then **Continue**. Return to your browser and notice the message is now displayed.

While running your service in Kubernetes with a debugger attached, you have full access to debug information such as the call stack, local variables, and exception information.

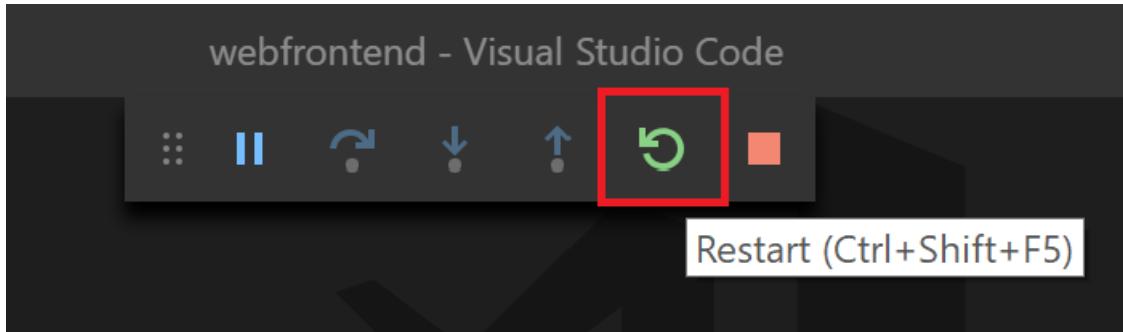
Remove the breakpoint by putting your cursor on line 19 in `src/main/java/com/ms/sample/webfrontend/Application.java` and pressing **F9**.

## Update code from Visual Studio Code

While the service is running in debugging mode, update line 19 in `src/main/java/com/ms/sample/webfrontend/Application.java`. For example:

```
return "Hello from webfrontend in Azure while debugging!";
```

Save the file. Select **Debug** then **Restart Debugging** or in the **Debug toolbar**, select the **Restart Debugging** button.



Open your service in a browser and notice your updated message is displayed.

Instead of rebuilding and redeploying a new container image each time code edits are made, Azure Dev Spaces incrementally recompiles code within the existing container to provide a faster edit/debug loop.

## Clean up your Azure resources

```
az group delete --name MyResourceGroup --yes --no-wait
```

## Next steps

Learn how Azure Dev Spaces helps you develop more complex applications across multiple containers, and how you can simplify collaborative development by working with different versions or branches of your code in different spaces.

[Working with multiple containers and team development](#)

# Quickstart: Debug and iterate on Kubernetes with Visual Studio Code and Node.js - Azure Dev Spaces

2/25/2020 • 5 minutes to read • [Edit Online](#)

In this quickstart, you set up Azure Dev Spaces with a managed Kubernetes cluster, and use a Node.js app in Visual Studio Code to iteratively develop and debug code in containers. Azure Dev Spaces lets you debug and test all the components of your application in Azure Kubernetes Service (AKS) with minimal development machine setup.

## Prerequisites

- An Azure account with an active subscription. [Create an account for free](#).
- [Latest version of Node.js](#).
- [Visual Studio Code](#).
- The [Azure Dev Spaces](#) extension for Visual Studio Code.
- [Azure CLI](#).
- [Git](#).

## Create an Azure Kubernetes Service cluster

You need to create an AKS cluster in a [supported region](#). The following commands create a resource group called *MyResourceGroup* and an AKS cluster called *MyAKS*.

```
az group create --name MyResourceGroup --location eastus  
az aks create -g MyResourceGroup -n MyAKS --location eastus --generate-ssh-keys
```

## Enable Azure Dev Spaces on your AKS cluster

Use the `use-dev-spaces` command to enable Dev Spaces on your AKS cluster and follow the prompts. The following command enables Dev Spaces on the *MyAKS* cluster in the *MyResourceGroup* group and creates a *default* dev space.

### NOTE

The `use-dev-spaces` command will also install the Azure Dev Spaces CLI if its not already installed. You can't install the Azure Dev Spaces CLI in the Azure Cloud Shell.

```
$ az aks use-dev-spaces -g MyResourceGroup -n MyAKS

'An Azure Dev Spaces Controller' will be created that targets resource 'MyAKS' in resource group
'MyResourceGroup'. Continue? (y/N): y

Creating and selecting Azure Dev Spaces Controller 'MyAKS' in resource group 'MyResourceGroup' that targets
resource 'MyAKS' in resource group 'MyResourceGroup'...2m 24s

Select a dev space or Kubernetes namespace to use as a dev space.
[1] default
Type a number or a new name: 1

Kubernetes namespace 'default' will be configured as a dev space. This will enable Azure Dev Spaces
instrumentation for new workloads in the namespace. Continue? (Y/n): Y

Configuring and selecting dev space 'default'...3s

Managed Kubernetes cluster 'MyAKS' in resource group 'MyResourceGroup' is ready for development in dev space
'default'. Type `azds prep` to prepare a source directory for use with Azure Dev Spaces and `azds up` to run.
```

## Get sample application code

In this article, you use the [Azure Dev Spaces sample application](#) to demonstrate using Azure Dev Spaces.

Clone the application from GitHub.

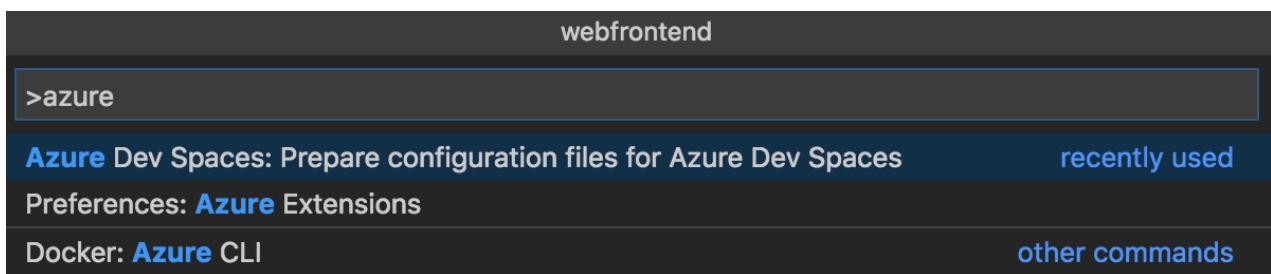
```
git clone https://github.com/Azure/dev-spaces
```

## Prepare the sample application in Visual Studio Code

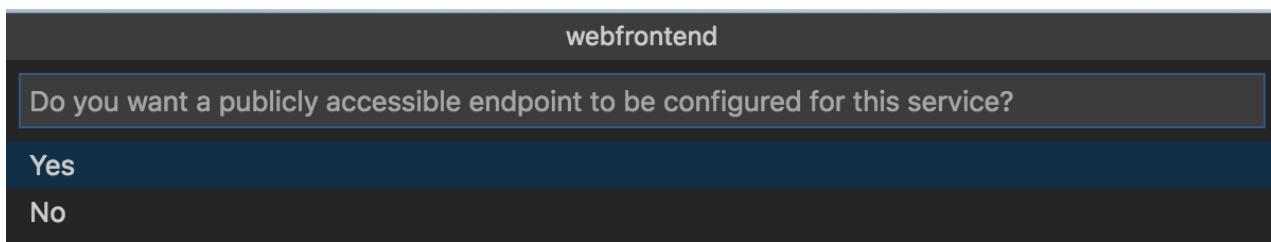
Open Visual Studio Code, select **File** then **Open**, navigate to the *dev-spaces/samples/nodejs/getting-started/webfrontend* directory, and select **Open**.

You now have the *webfrontend* project open in Visual Studio Code. To run the application in your dev space, generate the Docker and Helm chart assets using the Azure Dev Spaces extension in the Command Palette.

To open the Command Palette in Visual Studio Code, select **View** then **Command Palette**. Begin typing **Azure Dev Spaces** and select **Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces**.



When Visual Studio Code also prompts you to configure your public endpoint, choose **Yes** to enable a public endpoint.



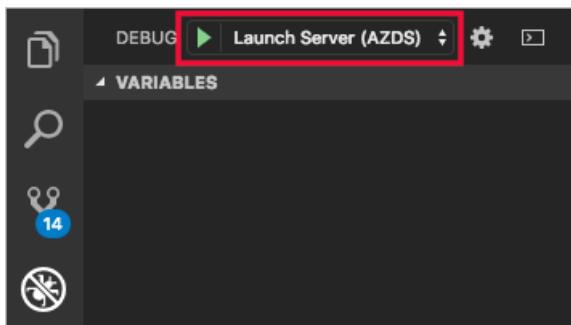
This command prepares your project to run in Azure Dev Spaces by generating a Dockerfile and Helm chart. It also generates a `.vscode` directory with debugging configuration at the root of your project.

#### TIP

The [Dockerfile and Helm chart](#) for your project is used by Azure Dev Spaces to build and run your code, but you can modify these files if you want to change how the project is built and run.

## Build and run code in Kubernetes from Visual Studio Code

Select the **Debug** icon on the left and select **Launch Server (AZDS)** at the top.



This command builds and runs your service in Azure Dev Spaces. The **Terminal** window at the bottom shows the build output and URLs for your service running Azure Dev Spaces. The **Debug Console** shows the log output.

#### NOTE

If you don't see any Azure Dev Spaces commands in the **Command Palette**, make sure you have installed the [Visual Studio Code extension for Azure Dev Spaces](#). Also verify you opened the `dev-spaces/samples/nodejs/getting-started/webfrontend` directory in Visual Studio Code.

You can see the service running by opening the public URL.

Select **Debug** then **Stop Debugging** to stop the debugger.

## Update code

To deploy an updated version of your service, you can update any file in your project and rerun **Launch Server**. For example:

1. If your application is still running, select **Debug** then **Stop Debugging** to stop it.
2. Update `line 13 in server.js` to:

```
res.send('Hello from webfrontend in Azure');
```

3. Save your changes.
4. Rerun **Launch Server**.
5. Navigate to your running service and observe your changes.
6. Select **Debug** then **Stop Debugging** to stop your application.

## Setting and using breakpoints for debugging

Start your service using **Launch Server (AZDS)**.

Navigate back to the Explorer view by selecting **View** then **Explorer**. Open *server.js* and click somewhere on line 13 to put your cursor there. To set a breakpoint press **F9** or select **Debug** then **Toggle Breakpoint**.

Open your service in a browser and notice no message is displayed. Return to Visual Studio Code and observe line 13 is highlighted. The breakpoint you set has paused the service at line 13. To resume the service, press **F5** or select **Debug** then **Continue**. Return to your browser and notice the message is now displayed.

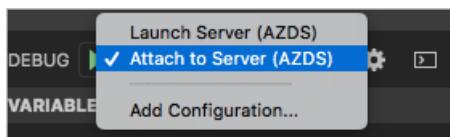
While running your service in Kubernetes with a debugger attached, you have full access to debug information such as the call stack, local variables, and exception information.

Remove the breakpoint by putting your cursor on line 13 in *server.js* and pressing **F9**.

Select **Debug** then **Stop Debugging** to stop the debugger.

## Update code from Visual Studio Code

Change the debug mode to **Attach to a Server (AZDS)** and start the service:



This command builds and runs your service in Azure Dev Spaces. It also starts a *nodemon* process in your service's container and attaches VS Code to it. The *nodemon* process allows for automatic restarts when source code changes are made, enabling faster inner loop development similar to developing on your local machine.

After the service starts, navigate to it using your browser and interact with it.

While the service is running, return to VS Code and update line 13 in *server.js*. For example:

```
res.send('Hello from webfrontend in Azure while debugging!');
```

Save the file and return to your service in a browser. Interact with the service and notice your updated message is displayed.

While running *nodemon*, the Node process is automatically restarted as soon as any code changes are detected. This automatic restart process is similar to the experience of editing and restarting your service on your local machine, providing an inner loop development experience.

## Clean up your Azure resources

```
az group delete --name MyResourceGroup --yes --no-wait
```

## Next steps

Learn how Azure Dev Spaces helps you develop more complex applications across multiple containers, and how you can simplify collaborative development by working with different versions or branches of your code in different spaces.

[Working with multiple containers and team development](#)

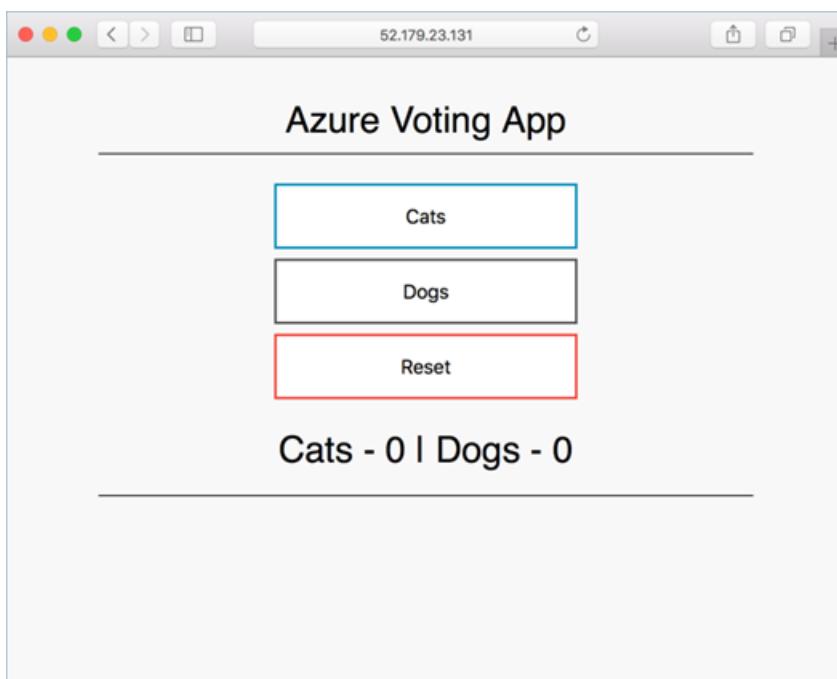
# Tutorial: Prepare an application for Azure Kubernetes Service (AKS)

2/25/2020 • 3 minutes to read • [Edit Online](#)

In this tutorial, part one of seven, a multi-container application is prepared for use in Kubernetes. Existing development tools such as Docker Compose are used to locally build and test an application. You learn how to:

- Clone a sample application source from GitHub
- Create a container image from the sample application source
- Test the multi-container application in a local Docker environment

Once completed, the following application runs in your local development environment:



In additional tutorials, the container image is uploaded to an Azure Container Registry, and then deployed into an AKS cluster.

## Before you begin

This tutorial assumes a basic understanding of core Docker concepts such as containers, container images, and `docker` commands. For a primer on container basics, see [Get started with Docker](#).

To complete this tutorial, you need a local Docker development environment running Linux containers. Docker provides packages that configure Docker on a [Mac](#), [Windows](#), or [Linux](#) system.

Azure Cloud Shell does not include the Docker components required to complete every step in these tutorials. Therefore, we recommend using a full Docker development environment.

## Get application code

The sample application used in this tutorial is a basic voting app. The application consists of a front-end web component and a back-end Redis instance. The web component is packaged into a custom container image. The Redis instance uses an unmodified image from Docker Hub.

Use [git](#) to clone the sample application to your development environment:

```
git clone https://github.com/Azure-Samples/azure-voting-app-redis.git
```

Change into the cloned directory.

```
cd azure-voting-app-redis
```

Inside the directory is the application source code, a pre-created Docker compose file, and a Kubernetes manifest file. These files are used throughout the tutorial set.

## Create container images

[Docker Compose](#) can be used to automate building container images and the deployment of multi-container applications.

Use the sample `docker-compose.yaml` file to create the container image, download the Redis image, and start the application:

```
docker-compose up -d
```

When completed, use the [docker images](#) command to see the created images. Three images have been downloaded or created. The `azure-vote-front` image contains the front-end application and uses the `nginx-flask` image as a base. The `redis` image is used to start a Redis instance.

```
$ docker images

REPOSITORY          TAG      IMAGE ID      CREATED        SIZE
azure-vote-front    latest   9cc914e25834  40 seconds ago  694MB
redis               latest   a1b99da73d05  7 days ago    106MB
tiangolo/uwsgi-nginx-flask  flask   788ca94b2313  9 months ago   694MB
```

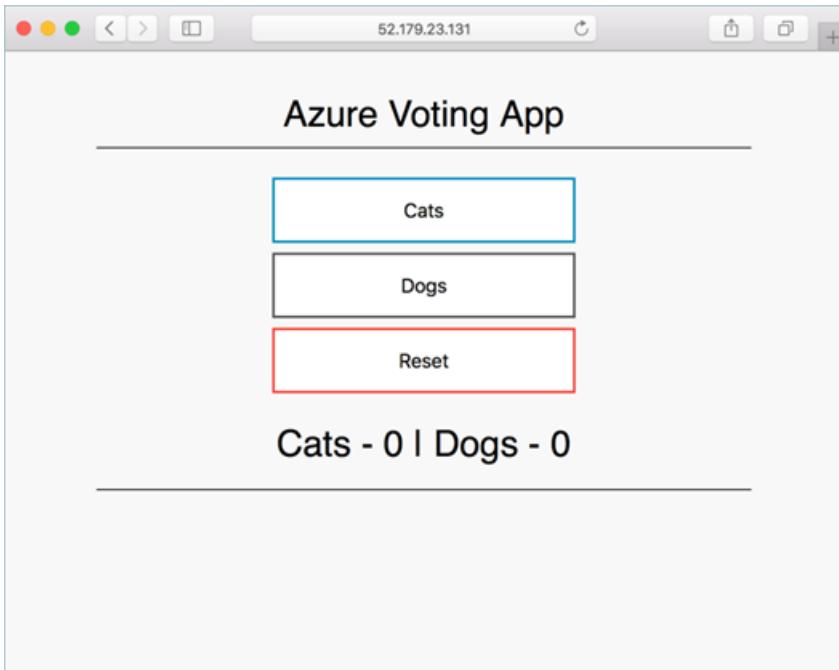
Run the [docker ps](#) command to see the running containers:

```
$ docker ps

CONTAINER ID        IMAGE             COMMAND            CREATED           STATUS            PORTS
NAMES
82411933e8f9        azure-vote-front   "/usr/bin/supervisord"   57 seconds ago   Up 30 seconds
443/tcp, 0.0.0.0:8080->80/tcp   azure-vote-front
b68fed4b66b6        redis              "docker-entrypoint..."  57 seconds ago   Up 30 seconds
0.0.0.0:6379->6379/tcp       azure-vote-back
```

## Test application locally

To see the running application, enter <http://localhost:8080> in a local web browser. The sample application loads, as shown in the following example:



## Clean up resources

Now that the application's functionality has been validated, the running containers can be stopped and removed. Do not delete the container images – in the next tutorial, the *azure-vote-front* image is uploaded to an Azure Container Registry instance.

Stop and remove the container instances and resources with the [docker-compose down](#) command:

```
docker-compose down
```

When the local application has been removed, you have a Docker image that contains the Azure Vote application, *azure-vote-front*, for use with the next tutorial.

## Next steps

In this tutorial, an application was tested and container images created for the application. You learned how to:

- Clone a sample application source from GitHub
- Create a container image from the sample application source
- Test the multi-container application in a local Docker environment

Advance to the next tutorial to learn how to store container images in Azure Container Registry.

[Push images to Azure Container Registry](#)

# Tutorial: Deploy and use Azure Container Registry

2/25/2020 • 4 minutes to read • [Edit Online](#)

Azure Container Registry (ACR) is a private registry for container images. A private container registry lets you securely build and deploy your applications and custom code. In this tutorial, part two of seven, you deploy an ACR instance and push a container image to it. You learn how to:

- Create an Azure Container Registry (ACR) instance
- Tag a container image for ACR
- Upload the image to ACR
- View images in your registry

In additional tutorials, this ACR instance is integrated with a Kubernetes cluster in AKS, and an application is deployed from the image.

## Before you begin

In the [previous tutorial](#), a container image was created for a simple Azure Voting application. If you have not created the Azure Voting app image, return to [Tutorial 1 – Create container images](#).

This tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an Azure Container Registry

To create an Azure Container Registry, you first need a resource group. An Azure resource group is a logical container into which Azure resources are deployed and managed.

Create a resource group with the `az group create` command. In the following example, a resource group named *myResourceGroup* is created in the *eastus* region:

```
az group create --name myResourceGroup --location eastus
```

Create an Azure Container Registry instance with the `az acr create` command and provide your own registry name. The registry name must be unique within Azure, and contain 5-50 alphanumeric characters. In the rest of this tutorial, `<acrName>` is used as a placeholder for the container registry name. Provide your own unique registry name. The *Basic* SKU is a cost-optimized entry point for development purposes that provides a balance of storage and throughput.

```
az acr create --resource-group myResourceGroup --name <acrName> --sku Basic
```

## Log in to the container registry

To use the ACR instance, you must first log in. Use the `az acr login` command and provide the unique name given to the container registry in the previous step.

```
az acr login --name <acrName>
```

The command returns a *Login Succeeded* message once completed.

## Tag a container image

To see a list of your current local images, use the [docker images](#) command:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
azure-vote-front	latest	4675398c9172	13 minutes ago	694MB
redis	latest	a1b99da73d05	7 days ago	106MB
tiangolo/uwsgi-nginx-flask	flask	788ca94b2313	9 months ago	694MB

To use the *azure-vote-front* container image with ACR, the image needs to be tagged with the login server address of your registry. This tag is used for routing when pushing container images to an image registry.

To get the login server address, use the [az acr list](#) command and query for the *loginServer* as follows:

```
az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table
```

Now, tag your local *azure-vote-front* image with the *acrLoginServer* address of the container registry. To indicate the image version, add :v1 to the end of the image name:

```
docker tag azure-vote-front <acrLoginServer>/azure-vote-front:v1
```

To verify the tags are applied, run [docker images](#) again. An image is tagged with the ACR instance address and a version number.

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
azure-vote-front	latest	eaf2b9c57e5e	8 minutes ago	716 MB
mycontainerregistry.azurecr.io/azure-vote-front	v1	eaf2b9c57e5e	8 minutes ago	716 MB
redis	latest	a1b99da73d05	7 days ago	106MB
tiangolo/uwsgi-nginx-flask	flask	788ca94b2313	8 months ago	694 MB

## Push images to registry

With your image built and tagged, push the *azure-vote-front* image to your ACR instance. Use [docker push](#) and provide your own *acrLoginServer* address for the image name as follows:

```
docker push <acrLoginServer>/azure-vote-front:v1
```

It may take a few minutes to complete the image push to ACR.

## List images in registry

To return a list of images that have been pushed to your ACR instance, use the [az acr repository list](#) command. Provide your own `<acrName>` as follows:

```
az acr repository list --name <acrName> --output table
```

The following example output lists the *azure-vote-front* image as available in the registry:

```
Result
-----
azure-vote-front
```

To see the tags for a specific image, use the [az acr repository show-tags](#) command as follows:

```
az acr repository show-tags --name <acrName> --repository azure-vote-front --output table
```

The following example output shows the *v1* image tagged in a previous step:

```
Result
-----
v1
```

You now have a container image that is stored in a private Azure Container Registry instance. This image is deployed from ACR to a Kubernetes cluster in the next tutorial.

## Next steps

In this tutorial, you created an Azure Container Registry and pushed an image for use in an AKS cluster. You learned how to:

- Create an Azure Container Registry (ACR) instance
- Tag a container image for ACR
- Upload the image to ACR
- View images in your registry

Advance to the next tutorial to learn how to deploy a Kubernetes cluster in Azure.

[Deploy Kubernetes cluster](#)

# Tutorial: Deploy an Azure Kubernetes Service (AKS) cluster

2/26/2020 • 2 minutes to read • [Edit Online](#)

Kubernetes provides a distributed platform for containerized applications. With AKS, you can quickly create a production ready Kubernetes cluster. In this tutorial, part three of seven, a Kubernetes cluster is deployed in AKS. You learn how to:

- Deploy a Kubernetes AKS cluster that can authenticate to an Azure container registry
- Install the Kubernetes CLI (kubectl)
- Configure kubectl to connect to your AKS cluster

In additional tutorials, the Azure Vote application is deployed to the cluster, scaled, and updated.

## Before you begin

In previous tutorials, a container image was created and uploaded to an Azure Container Registry instance. If you haven't done these steps, and would like to follow along, start at [Tutorial 1 – Create container images](#).

This tutorial requires that you're running the Azure CLI version 2.0.75 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create a Kubernetes cluster

AKS clusters can use Kubernetes role-based access controls (RBAC). These controls let you define access to resources based on roles assigned to users. Permissions are combined if a user is assigned multiple roles, and permissions can be scoped to either a single namespace or across the whole cluster. By default, the Azure CLI automatically enables RBAC when you create an AKS cluster.

Create an AKS cluster using `az aks create`. The following example creates a cluster named *myAKSCluster* in the resource group named *myResourceGroup*. This resource group was created in the [previous tutorial](#). To allow an AKS cluster to interact with other Azure resources, an Azure Active Directory service principal is automatically created, since you did not specify one. Here, this service principal is granted the right to pull images from the Azure Container Registry (ACR) instance you created in the previous tutorial.

```
az aks create \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --node-count 2 \
    --generate-ssh-keys \
    --attach-acr <acrName>
```

After a few minutes, the deployment completes, and returns JSON-formatted information about the AKS deployment.

### NOTE

To ensure your cluster to operate reliably, you should run at least 2 (two) nodes.

# Install the Kubernetes CLI

To connect to the Kubernetes cluster from your local computer, you use [kubectl](#), the Kubernetes command-line client.

If you use the Azure Cloud Shell, `kubectl` is already installed. You can also install it locally using the [az aks install-cli](#) command:

```
az aks install-cli
```

## Connect to cluster using kubectl

To configure `kubectl` to connect to your Kubernetes cluster, use the [az aks get-credentials](#) command. The following example gets credentials for the AKS cluster named *myAKSCluster* in the *myResourceGroup*:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, run the [kubectl get nodes](#) command to return a list of the cluster nodes:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
aks-nodepool1-12345678-0	Ready	agent	32m	v1.14.8

## Next steps

In this tutorial, a Kubernetes cluster was deployed in AKS, and you configured `kubectl` to connect to it. You learned how to:

- Deploy a Kubernetes AKS cluster that can authenticate to an Azure container registry
- Install the Kubernetes CLI (`kubectl`)
- Configure `kubectl` to connect to your AKS cluster

Advance to the next tutorial to learn how to deploy an application to the cluster.

[Deploy application in Kubernetes](#)

# Tutorial: Run applications in Azure Kubernetes Service (AKS)

2/25/2020 • 3 minutes to read • [Edit Online](#)

Kubernetes provides a distributed platform for containerized applications. You build and deploy your own applications and services into a Kubernetes cluster, and let the cluster manage the availability and connectivity. In this tutorial, part four of seven, a sample application is deployed into a Kubernetes cluster. You learn how to:

- Update a Kubernetes manifest file
- Run an application in Kubernetes
- Test the application

In additional tutorials, this application is scaled out and updated.

This quickstart assumes a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#).

## Before you begin

In previous tutorials, an application was packaged into a container image, this image was uploaded to Azure Container Registry, and a Kubernetes cluster was created.

To complete this tutorial, you need the pre-created `azure-vote-all-in-one-redis.yaml` Kubernetes manifest file.

This file was downloaded with the application source code in a previous tutorial. Verify that you've cloned the repo, and that you have changed directories into the cloned repo. If you haven't done these steps, and would like to follow along, start with [Tutorial 1 – Create container images](#).

This tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Update the manifest file

In these tutorials, an Azure Container Registry (ACR) instance stores the container image for the sample application. To deploy the application, you must update the image name in the Kubernetes manifest file to include the ACR login server name.

Get the ACR login server name using the `az acr list` command as follows:

```
az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table
```

The sample manifest file from the git repo cloned in the first tutorial uses the login server name of *microsoft*. Make sure that you're in the cloned *azure-voting-app-redis* directory, then open the manifest file with a text editor, such as `vi`:

```
vi azure-vote-all-in-one-redis.yaml
```

Replace *microsoft* with your ACR login server name. The image name is found on line 51 of the manifest file. The following example shows the default image name:

```
containers:
- name: azure-vote-front
  image: microsoft/azure-vote-front:v1
```

Provide your own ACR login server name so that your manifest file looks like the following example:

```
containers:
- name: azure-vote-front
  image: <acrName>.azurecr.io/azure-vote-front:v1
```

Save and close the file. In `vi`, use `:wq`.

## Deploy the application

To deploy your application, use the [kubectl apply](#) command. This command parses the manifest file and creates the defined Kubernetes objects. Specify the sample manifest file, as shown in the following example:

```
kubectl apply -f azure-vote-all-in-one-redis.yaml
```

The following example output shows the resources successfully created in the AKS cluster:

```
$ kubectl apply -f azure-vote-all-in-one-redis.yaml

deployment "azure-vote-back" created
service "azure-vote-back" created
deployment "azure-vote-front" created
service "azure-vote-front" created
```

## Test the application

When the application runs, a Kubernetes service exposes the application front end to the internet. This process can take a few minutes to complete.

To monitor progress, use the [kubectl get service](#) command with the `--watch` argument.

```
kubectl get service azure-vote-front --watch
```

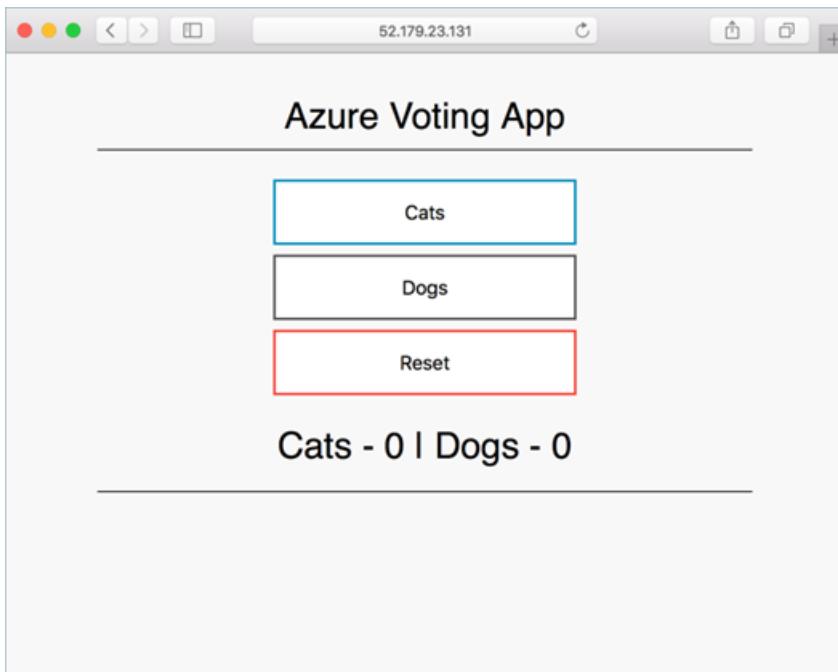
Initially the *EXTERNAL-IP* for the *azure-vote-front* service is shown as *pending*:

```
azure-vote-front  LoadBalancer  10.0.34.242  <pending>      80:30676/TCP  5s
```

When the *EXTERNAL-IP* address changes from *pending* to an actual public IP address, use `CTRL-C` to stop the `kubectl` watch process. The following example output shows a valid public IP address assigned to the service:

```
azure-vote-front  LoadBalancer  10.0.34.242  52.179.23.131  80:30676/TCP  67s
```

To see the application in action, open a web browser to the external IP address of your service:



If the application didn't load, it might be due to an authorization problem with your image registry. To view the status of your containers, use the `kubectl get pods` command. If the container images can't be pulled, see [Authenticate with Azure Container Registry from Azure Kubernetes Service](#).

## Next steps

In this tutorial, a sample Azure vote application was deployed to a Kubernetes cluster in AKS. You learned how to:

- Update a Kubernetes manifest files
- Run an application in Kubernetes
- Test the application

Advance to the next tutorial to learn how to scale a Kubernetes application and the underlying Kubernetes infrastructure.

[Scale Kubernetes application and infrastructure](#)

# Tutorial: Scale applications in Azure Kubernetes Service (AKS)

2/26/2020 • 4 minutes to read • [Edit Online](#)

If you've followed the tutorials, you have a working Kubernetes cluster in AKS and you deployed the sample Azure Voting app. In this tutorial, part five of seven, you scale out the pods in the app and try pod autoscaling. You also learn how to scale the number of Azure VM nodes to change the cluster's capacity for hosting workloads. You learn how to:

- Scale the Kubernetes nodes
- Manually scale Kubernetes pods that run your application
- Configure autoscaling pods that run the app front-end

In additional tutorials, the Azure Vote application is updated to a new version.

## Before you begin

In previous tutorials, an application was packaged into a container image. This image was uploaded to Azure Container Registry, and you created an AKS cluster. The application was then deployed to the AKS cluster. If you haven't done these steps, and would like to follow along, start with [Tutorial 1 – Create container images](#).

This tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Manually scale pods

When the Azure Vote front-end and Redis instance were deployed in previous tutorials, a single replica was created. To see the number and state of pods in your cluster, use the [kubectl get](#) command as follows:

```
kubectl get pods
```

The following example output shows one front-end pod and one back-end pod:

NAME	READY	STATUS	RESTARTS	AGE
azure-vote-back-2549686872-4d2r5	1/1	Running	0	31m
azure-vote-front-848767080-tf34m	1/1	Running	0	31m

To manually change the number of pods in the *azure-vote-front* deployment, use the [kubectl scale](#) command. The following example increases the number of front-end pods to 5:

```
kubectl scale --replicas=5 deployment/azure-vote-front
```

Run [kubectl get pods](#) again to verify that AKS creates the additional pods. After a minute or so, the additional pods are available in your cluster:

```
kubectl get pods
```

	READY	STATUS	RESTARTS	AGE
azure-vote-back-2606967446-nmpcf	1/1	Running	0	15m
azure-vote-front-3309479140-2hfh0	1/1	Running	0	3m
azure-vote-front-3309479140-bzt05	1/1	Running	0	3m
azure-vote-front-3309479140-fvcvm	1/1	Running	0	3m
azure-vote-front-3309479140-hrbf2	1/1	Running	0	15m
azure-vote-front-3309479140-qphz8	1/1	Running	0	3m

## Autoscale pods

Kubernetes supports [horizontal pod autoscaling](#) to adjust the number of pods in a deployment depending on CPU utilization or other select metrics. The [Metrics Server](#) is used to provide resource utilization to Kubernetes, and is automatically deployed in AKS clusters versions 1.10 and higher. To see the version of your AKS cluster, use the [az aks show](#) command, as shown in the following example:

```
az aks show --resource-group myResourceGroup --name myAKSCluster --query kubernetesVersion --output table
```

### NOTE

If your AKS cluster is less than 1.10, the Metrics Server is not automatically installed. To install, clone the [metrics-server](#) GitHub repo and install the example resource definitions. To view the contents of these YAML definitions, see [Metrics Server for Kuberentes 1.8+](#).

```
git clone https://github.com/kubernetes-incubator/metrics-server.git
kubectl create -f metrics-server/deploy/1.8+/-
```

To use the autoscaler, all containers in your pods and your pods must have CPU requests and limits defined. In the [azure-vote-front](#) deployment, the front-end container already requests 0.25 CPU, with a limit of 0.5 CPU. These resource requests and limits are defined as shown in the following example snippet:

```
resources:
  requests:
    cpu: 250m
  limits:
    cpu: 500m
```

The following example uses the [kubectl autoscale](#) command to autoscale the number of pods in the [azure-vote-front](#) deployment. If average CPU utilization across all pods exceeds 50% of their requested usage, the autoscaler increases the pods up to a maximum of 10 instances. A minimum of 3 instances is then defined for the deployment:

```
kubectl autoscale deployment azure-vote-front --cpu-percent=50 --min=3 --max=10
```

Alternatively, you can create a manifest file to define the autoscaler behavior and resource limits. The following is an example of a manifest file named [azure-vote-hpa.yaml](#).

```

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: azure-vote-back-hpa
spec:
  maxReplicas: 10 # define max replica count
  minReplicas: 3 # define min replica count
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: azure-vote-back
  targetCPUUtilizationPercentage: 50 # target CPU utilization

apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: azure-vote-front-hpa
spec:
  maxReplicas: 10 # define max replica count
  minReplicas: 3 # define min replica count
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: azure-vote-front
  targetCPUUtilizationPercentage: 50 # target CPU utilization

```

Use `kubectl apply` to apply the autoscaler defined in the `azure-vote-hpa.yaml` manifest file.

```
kubectl apply -f azure-vote-hpa.yaml
```

To see the status of the autoscaler, use the `kubectl get hpa` command as follows:

```

kubectl get hpa

```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
azure-vote-front	Deployment/azure-vote-front	0% / 50%	3	10	3	2m

After a few minutes, with minimal load on the Azure Vote app, the number of pod replicas decreases automatically to three. You can use `kubectl get pods` again to see the unneeded pods being removed.

## Manually scale AKS nodes

If you created your Kubernetes cluster using the commands in the previous tutorial, it has two nodes. You can adjust the number of nodes manually if you plan more or fewer container workloads on your cluster.

The following example increases the number of nodes to three in the Kubernetes cluster named *myAKSCluster*. The command takes a couple of minutes to complete.

```
az aks scale --resource-group myResourceGroup --name myAKSCluster --node-count 3
```

When the cluster has successfully scaled, the output is similar to following example:

```
"agentPoolProfiles": [  
  {  
    "count": 3,  
    "dnsPrefix": null,  
    "fqdn": null,  
    "name": "myAKSCluster",  
    "osDiskSizeGb": null,  
    "osType": "Linux",  
    "ports": null,  
    "storageProfile": "ManagedDisks",  
    "vmSize": "Standard_D2_v2",  
    "vnetSubnetId": null  
  }  
]
```

## Next steps

In this tutorial, you used different scaling features in your Kubernetes cluster. You learned how to:

- Manually scale Kubernetes pods that run your application
- Configure autoscaling pods that run the app front-end
- Manually scale the Kubernetes nodes

Advance to the next tutorial to learn how to update application in Kubernetes.

[Update an application in Kubernetes](#)

# Tutorial: Update an application in Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

After an application has been deployed in Kubernetes, it can be updated by specifying a new container image or image version. An update is staged so that only a portion of the deployment is updated at the same time. This staged update enables the application to keep running during the update. It also provides a rollback mechanism if a deployment failure occurs.

In this tutorial, part six of seven, the sample Azure Vote app is updated. You learn how to:

- Update the front-end application code
- Create an updated container image
- Push the container image to Azure Container Registry
- Deploy the updated container image

## Before you begin

In previous tutorials, an application was packaged into a container image. This image was uploaded to Azure Container Registry, and you created an AKS cluster. The application was then deployed to the AKS cluster.

An application repository was also cloned that includes the application source code, and a pre-created Docker Compose file used in this tutorial. Verify that you've created a clone of the repo, and have changed directories into the cloned directory. If you haven't completed these steps, and want to follow along, start with [Tutorial 1 – Create container images](#).

This tutorial requires that you're running the Azure CLI version 2.0.53 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Update an application

Let's make a change to the sample application, then update the version already deployed to your AKS cluster. Make sure that you're in the cloned *azure-voting-app-redis* directory. The sample application source code can then be found inside the *azure-vote* directory. Open the *config\_file.cfg* file with an editor, such as `vi`:

```
vi azure-vote/azure-vote/config_file.cfg
```

Change the values for *VOTE1VALUE* and *VOTE2VALUE* to different values, such as colors. The following example shows the updated values:

```
# UI Configurations
TITLE = 'Azure Voting App'
VOTE1VALUE = 'Blue'
VOTE2VALUE = 'Purple'
SHOWHOST = 'false'
```

Save and close the file. In `vi`, use `:wq`.

## Update the container image

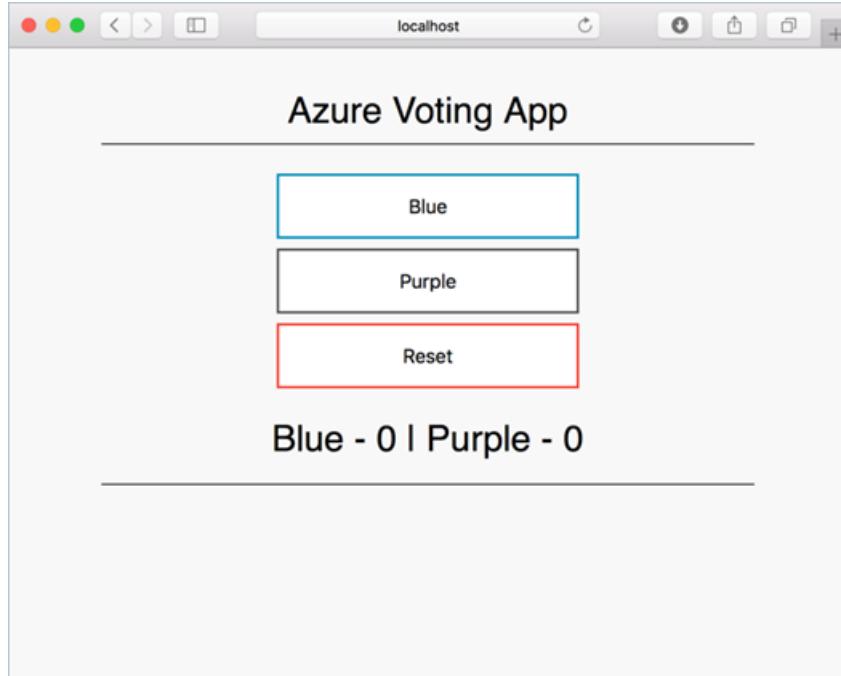
To re-create the front-end image and test the updated application, use `docker-compose`. The `--build` argument is used to instruct Docker Compose to re-create the application image:

```
docker-compose up --build -d
```

## Test the application locally

To verify that the updated container image shows your changes, open a local web browser to

```
http://localhost:8080 .
```



The updated values provided in the `config_file.cfg` file are displayed in your running application.

## Tag and push the image

To correctly use the updated image, tag the `azure-vote-front` image with the login server name of your ACR registry. Get the login server name with the `az acr list` command:

```
az acr list --resource-group myResourceGroup --query "[].{acrLoginServer:loginServer}" --output table
```

Use `docker tag` to tag the image. Replace `<acrLoginServer>` with your ACR login server name or public registry hostname, and update the image version to `:v2` as follows:

```
docker tag azure-vote-front <acrLoginServer>/azure-vote-front:v2
```

Now use `docker push` to upload the image to your registry. Replace `<acrLoginServer>` with your ACR login server name.

### NOTE

If you experience issues pushing to your ACR registry, make sure that you are still logged in. Run the `az acr login` command using the name of your Azure Container Registry that you created in the [Create an Azure Container Registry](#) step. For example, `az acr login --name <azure container registry name>`.

```
docker push <acrLoginServer>/azure-vote-front:v2
```

## Deploy the updated application

To provide maximum uptime, multiple instances of the application pod must be running. Verify the number of running front-end instances with the [kubectl get pods](#) command:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
azure-vote-back-217588096-5w632	1/1	Running	0	10m
azure-vote-front-233282510-b5pkz	1/1	Running	0	10m
azure-vote-front-233282510-dhrtr	1/1	Running	0	10m
azure-vote-front-233282510-pqbfk	1/1	Running	0	10m

If you don't have multiple front-end pods, scale the *azure-vote-front* deployment as follows:

```
kubectl scale --replicas=3 deployment/azure-vote-front
```

To update the application, use the [kubectl set](#) command. Update `<acrLoginServer>` with the login server or host name of your container registry, and specify the v2 application version:

```
kubectl set image deployment azure-vote-front azure-vote-front=<acrLoginServer>/azure-vote-front:v2
```

To monitor the deployment, use the [kubectl get pod](#) command. As the updated application is deployed, your pods are terminated and re-created with the new container image.

```
kubectl get pods
```

The following example output shows pods terminating and new instances running as the deployment progresses:

```
$ kubectl get pods
```

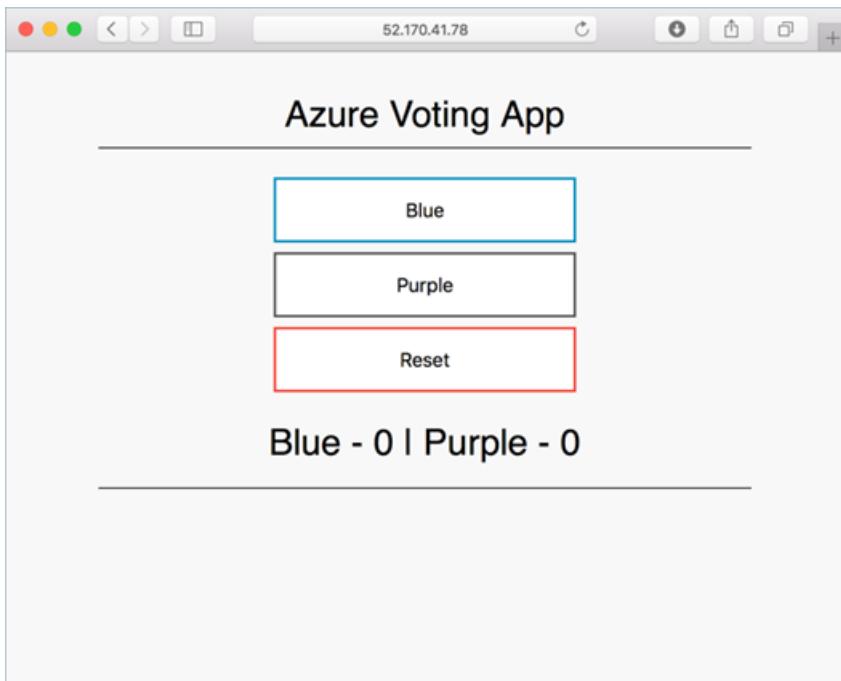
NAME	READY	STATUS	RESTARTS	AGE
azure-vote-back-2978095810-gq9g0	1/1	Running	0	5m
azure-vote-front-1297194256-tpjlg	1/1	Running	0	1m
azure-vote-front-1297194256-tptnx	1/1	Running	0	5m
azure-vote-front-1297194256-zktw9	1/1	Terminating	0	1m

## Test the updated application

To view the update application, first get the external IP address of the `azure-vote-front` service:

```
kubectl get service azure-vote-front
```

Now open a local web browser to the IP address of your service:



## Next steps

In this tutorial, you updated an application and rolled out this update to your AKS cluster. You learned how to:

- Update the front-end application code
- Create an updated container image
- Push the container image to Azure Container Registry
- Deploy the updated container image

Advance to the next tutorial to learn how to upgrade an AKS cluster to a new version of Kubernetes.

[Upgrade Kubernetes](#)

# Tutorial: Upgrade Kubernetes in Azure Kubernetes Service (AKS)

2/26/2020 • 3 minutes to read • [Edit Online](#)

As part of the application and cluster lifecycle, you may wish to upgrade to the latest available version of Kubernetes and use new features. An Azure Kubernetes Service (AKS) cluster can be upgraded using the Azure CLI.

In this tutorial, part seven of seven, a Kubernetes cluster is upgraded. You learn how to:

- Identify current and available Kubernetes versions
- Upgrade the Kubernetes nodes
- Validate a successful upgrade

## Before you begin

In previous tutorials, an application was packaged into a container image. This image was uploaded to Azure Container Registry, and you created an AKS cluster. The application was then deployed to the AKS cluster. If you have not done these steps, and would like to follow along, start with [Tutorial 1 – Create container images](#).

This tutorial requires that you are running the Azure CLI version 2.0.53 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Get available cluster versions

Before you upgrade a cluster, use the `az aks get-upgrades` command to check which Kubernetes releases are available for upgrade:

```
az aks get-upgrades --resource-group myResourceGroup --name myAKSCluster --output table
```

In the following example, the current version is 1.14.8, and the available versions are shown under the *Upgrades* column.

Name	ResourceGroup	MasterVersion	NodePoolVersion	Upgrades
default	myResourceGroup	1.14.8	1.14.8	1.15.5, 1.15.7

## Upgrade a cluster

To minimize disruption to running applications, AKS nodes are carefully cordoned and drained. In this process, the following steps are performed:

1. The Kubernetes scheduler prevents additional pods being scheduled on a node that is to be upgraded.
2. Running pods on the node are scheduled on other nodes in the cluster.
3. A node is created that runs the latest Kubernetes components.
4. When the new node is ready and joined to the cluster, the Kubernetes scheduler begins to run pods on it.
5. The old node is deleted, and the next node in the cluster begins the cordon and drain process.

Use the `az aks upgrade` command to upgrade the AKS cluster. The following example upgrades the cluster to

Kubernetes version 1.14.6.

#### NOTE

You can only upgrade one minor version at a time. For example, you can upgrade from 1.14.x to 1.15.x, but cannot upgrade from 1.14.x to 1.16.x directly. To upgrade from 1.14.x to 1.16.x, first upgrade from 1.14.x to 1.15.x, then perform another upgrade from 1.15.x to 1.16.x.

```
az aks upgrade --resource-group myResourceGroup --name myAKSCluster --kubernetes-version 1.15.5
```

The following condensed example output shows the *kubernetesVersion* now reports 1.15.5:

```
{
  "agentPoolProfiles": [
    {
      "count": 3,
      "maxPods": 110,
      "name": "nodepool1",
      "osType": "Linux",
      "storageProfile": "ManagedDisks",
      "vmSize": "Standard_DS1_v2",
    }
  ],
  "dnsPrefix": "myAKSClust-myResourceGroup-19da35",
  "enableRbac": false,
  "fqdn": "myaksclust-myresourcegroup-19da35-bd54a4be.hcp.eastus.azmk8s.io",
  "id": "/subscriptions/<Subscription
ID>/resourcegroups/myResourceGroup/providers/Microsoft.ContainerService/managedClusters/myAKSCluster",
  "kubernetesVersion": "1.15.5",
  "location": "eastus",
  "name": "myAKSCluster",
  "type": "Microsoft.ContainerService/ManagedClusters"
}
```

## Validate an upgrade

Confirm that the upgrade was successful using the [az aks show](#) command as follows:

```
az aks show --resource-group myResourceGroup --name myAKSCluster --output table
```

The following example output shows the AKS cluster runs *KubernetesVersion* 1.15.5:

Name	Location	ResourceGroup	KubernetesVersion	ProvisioningState	Fqdn
myAKSCluster	eastus	myResourceGroup	1.15.5	Succeeded	myaksclust- myresourcegroup-19da35-bd54a4be.hcp.eastus.azmk8s.io

## Delete the cluster

As this tutorial is the last part of the series, you may want to delete the AKS cluster. As the Kubernetes nodes run on Azure virtual machines (VMs), they continue to incur charges even if you don't use the cluster. Use the [az group delete](#) command to remove the resource group, container service, and all related resources.

```
az group delete --name myResourceGroup --yes --no-wait
```

#### NOTE

When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see [AKS service principal considerations and deletion](#).

## Next steps

In this tutorial, you upgraded Kubernetes in an AKS cluster. You learned how to:

- Identify current and available Kubernetes versions
- Upgrade the Kubernetes nodes
- Validate a successful upgrade

Follow this link to learn more about AKS.

[AKS overview](#)

# Kubernetes core concepts for Azure Kubernetes Service (AKS)

2/25/2020 • 15 minutes to read • [Edit Online](#)

As application development moves towards a container-based approach, the need to orchestrate and manage resources is important. Kubernetes is the leading platform that provides the ability to provide reliable scheduling of fault-tolerant application workloads. Azure Kubernetes Service (AKS) is a managed Kubernetes offering that further simplifies container-based application deployment and management.

This article introduces the core Kubernetes infrastructure components such as the *control plane*, *nodes*, and *node pools*. Workload resources such as *pods*, *deployments*, and *sets* are also introduced, along with how to group resources into *namespaces*.

## What is Kubernetes?

Kubernetes is a rapidly evolving platform that manages container-based applications and their associated networking and storage components. The focus is on the application workloads, not the underlying infrastructure components. Kubernetes provides a declarative approach to deployments, backed by a robust set of APIs for management operations.

You can build and run modern, portable, microservices-based applications that benefit from Kubernetes orchestrating and managing the availability of those application components. Kubernetes supports both stateless and stateful applications as teams progress through the adoption of microservices-based applications.

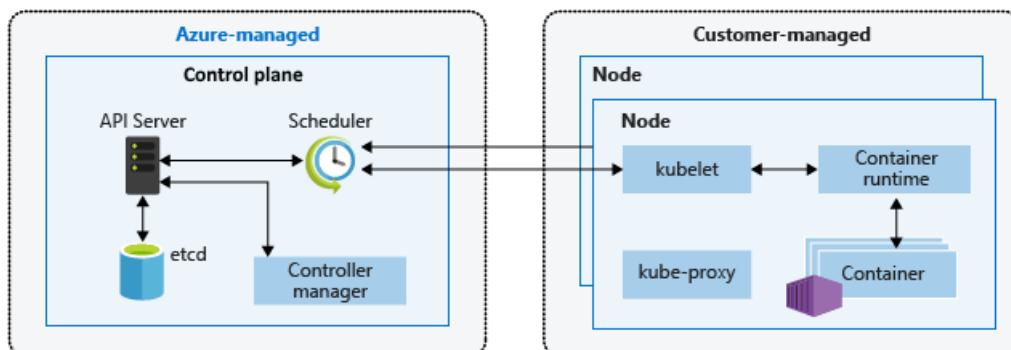
As an open platform, Kubernetes allows you to build your applications with your preferred programming language, OS, libraries, or messaging bus. Existing continuous integration and continuous delivery (CI/CD) tools can integrate with Kubernetes to schedule and deploy releases.

Azure Kubernetes Service (AKS) provides a managed Kubernetes service that reduces the complexity for deployment and core management tasks, including coordinating upgrades. The AKS control plane is managed by the Azure platform, and you only pay for the AKS nodes that run your applications. AKS is built on top of the open-source Azure Kubernetes Service Engine ([aks-engine](#)).

## Kubernetes cluster architecture

A Kubernetes cluster is divided into two components:

- *Control plane* nodes provide the core Kubernetes services and orchestration of application workloads.
- *Nodes* run your application workloads.



# Control plane

When you create an AKS cluster, a control plane is automatically created and configured. This control plane is provided as a managed Azure resource abstracted from the user. There's no cost for the control plane, only the nodes that are part of the AKS cluster.

The control plane includes the following core Kubernetes components:

- *kube-apiserver* - The API server is how the underlying Kubernetes APIs are exposed. This component provides the interaction for management tools, such as `kubectl` or the Kubernetes dashboard.
- *etcd* - To maintain the state of your Kubernetes cluster and configuration, the highly available *etcd* is a key value store within Kubernetes.
- *kube-scheduler* - When you create or scale applications, the Scheduler determines what nodes can run the workload and starts them.
- *kube-controller-manager* - The Controller Manager oversees a number of smaller Controllers that perform actions such as replicating pods and handling node operations.

AKS provides a single-tenant control plane, with a dedicated API server, Scheduler, etc. You define the number and size of the nodes, and the Azure platform configures the secure communication between the control plane and nodes. Interaction with the control plane occurs through Kubernetes APIs, such as `kubectl` or the Kubernetes dashboard.

This managed control plane means that you don't need to configure components like a highly available *etcd* store, but it also means that you can't access the control plane directly. Upgrades to Kubernetes are orchestrated through the Azure CLI or Azure portal, which upgrades the control plane and then the nodes. To troubleshoot possible issues, you can review the control plane logs through Azure Monitor logs.

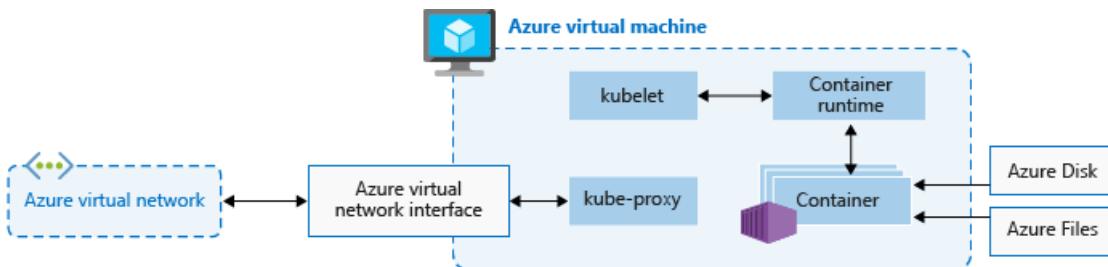
If you need to configure the control plane in a particular way or need direct access to it, you can deploy your own Kubernetes cluster using [aks-engine](#).

For associated best practices, see [Best practices for cluster security and upgrades in AKS](#).

## Nodes and node pools

To run your applications and supporting services, you need a Kubernetes *node*. An AKS cluster has one or more nodes, which is an Azure virtual machine (VM) that runs the Kubernetes node components and container runtime:

- The `kubelet` is the Kubernetes agent that processes the orchestration requests from the control plane and scheduling of running the requested containers.
- Virtual networking is handled by the *kube-proxy* on each node. The proxy routes network traffic and manages IP addressing for services and pods.
- The *container runtime* is the component that allows containerized applications to run and interact with additional resources such as the virtual network and storage. In AKS, Moby is used as the container runtime.



The Azure VM size for your nodes defines how many CPUs, how much memory, and the size and type of storage available (such as high-performance SSD or regular HDD). If you anticipate a need for applications that require large amounts of CPU and memory or high-performance storage, plan the node size accordingly. You can also

scale up the number of nodes in your AKS cluster to meet demand.

In AKS, the VM image for the nodes in your cluster is currently based on Ubuntu Linux or Windows Server 2019. When you create an AKS cluster or scale up the number of nodes, the Azure platform creates the requested number of VMs and configures them. There's no manual configuration for you to perform. Agent nodes are billed as standard virtual machines, so any discounts you have on the VM size you're using (including [Azure reservations](#)) are automatically applied.

If you need to use a different host OS, container runtime, or include custom packages, you can deploy your own Kubernetes cluster using [aks-engine](#). The upstream [aks-engine](#) releases features and provides configuration options before they are officially supported in AKS clusters. For example, if you wish to use a container runtime other than Moby, you can use [aks-engine](#) to configure and deploy a Kubernetes cluster that meets your current needs.

## Resource reservations

Node resources are utilized by AKS to make the node function as part of your cluster. This can create a discrepancy between your node's total resources and the resources allocatable when used in AKS. This is important to note when setting requests and limits for user deployed pods.

To find a node's allocatable resources run:

```
kubectl describe node [NODE_NAME]
```

To maintain node performance and functionality, resources are reserved on each node by AKS. As a node grows larger in resources, the resource reservation grows due to a higher amount of user deployed pods needing management.

### NOTE

Using AKS add-ons such as Container Insights (OMS) will consume additional node resources.

- **CPU** - reserved CPU is dependent on node type and cluster configuration which may cause less allocatable CPU due to running additional features

CPU CORES ON HOST	1	2	4	8	16	32	64
Kube-reserved (millicores)	60	100	140	180	260	420	740

- **Memory** - memory utilized by AKS includes the sum of two values.

1. The kubelet daemon is installed on all Kubernetes agent nodes to manage container creation and termination. By default on AKS, this daemon has the following eviction rule: `memory.available < 750Mi`, which means a node must always have at least 750 Mi allocatable at all times. When a host is below that threshold of available memory, the kubelet will terminate one of the running pods to free memory on the host machine and protect it. This is a reactive action once available memory decreases beyond the 750Mi threshold.
2. The second value is a progressive rate of memory reservations for the kubelet daemon to properly function (kube-reserved).
  - 25% of the first 4 GB of memory

- 20% of the next 4 GB of memory (up to 8 GB)
- 10% of the next 8 GB of memory (up to 16 GB)
- 6% of the next 112 GB of memory (up to 128 GB)
- 2% of any memory above 128 GB

The above rules for memory and CPU allocation are used to keep agent nodes healthy, including some hosting system pods that are critical to cluster health. These allocation rules also cause the node to report less allocatable memory and CPU than it would if it were not part of a Kubernetes cluster. The above resource reservations can't be changed.

For example, if a node offers 7 GB, it will report 34% of memory not allocatable on top of the 750Mi hard eviction threshold.

$$(0.25^*4) + (0.20^*3) = + 1 \text{ GB} + 0.6\text{GB} = 1.6\text{GB} / 7\text{GB} = 22.86\% \text{ reserved}$$

In addition to reservations for Kubernetes itself, the underlying node OS also reserves an amount of CPU and memory resources to maintain OS functions.

For associated best practices, see [Best practices for basic scheduler features in AKS](#).

## Node pools

Nodes of the same configuration are grouped together into *node pools*. A Kubernetes cluster contains one or more node pools. The initial number of nodes and size are defined when you create an AKS cluster, which creates a *default node pool*. This default node pool in AKS contains the underlying VMs that run your agent nodes.

### NOTE

To ensure your cluster to operate reliably, you should run at least 2 (two) nodes in the default node pool.

When you scale or upgrade an AKS cluster, the action is performed against the default node pool. You can also choose to scale or upgrade a specific node pool. For upgrade operations, running containers are scheduled on other nodes in the node pool until all the nodes are successfully upgraded.

For more information about how to use multiple node pools in AKS, see [Create and manage multiple node pools for a cluster in AKS](#).

## Node selectors

In an AKS cluster that contains multiple node pools, you may need to tell the Kubernetes Scheduler which node pool to use for a given resource. For example, ingress controllers shouldn't run on Windows Server nodes (currently in preview in AKS). Node selectors let you define various parameters, such as the node OS, to control where a pod should be scheduled.

The following basic example schedules an NGINX instance on a Linux node using the node selector "beta.kubernetes.io/os": linux:

```
kind: Pod
apiVersion: v1
metadata:
  name: nginx
spec:
  containers:
    - name: myfrontend
      image: nginx:1.15.12
  nodeSelector:
    "beta.kubernetes.io/os": linux
```

For more information on how to control where pods are scheduled, see [Best practices for advanced scheduler](#)

features in AKS.

## Pods

Kubernetes uses *pods* to run an instance of your application. A pod represents a single instance of your application. Pods typically have a 1:1 mapping with a container, although there are advanced scenarios where a pod may contain multiple containers. These multi-container pods are scheduled together on the same node, and allow containers to share related resources.

When you create a pod, you can define *resource requests* to request a certain amount of CPU or memory resources. The Kubernetes Scheduler tries to schedule the pods to run on a node with available resources to meet the request. You can also specify maximum resource limits that prevent a given pod from consuming too much compute resource from the underlying node. A best practice is to include resource limits for all pods to help the Kubernetes Scheduler understand which resources are needed and permitted.

For more information, see [Kubernetes pods](#) and [Kubernetes pod lifecycle](#).

A pod is a logical resource, but the container(s) are where the application workloads run. Pods are typically ephemeral, disposable resources, and individually scheduled pods miss some of the high availability and redundancy features Kubernetes provides. Instead, pods are usually deployed and managed by Kubernetes *Controllers*, such as the Deployment Controller.

## Deployments and YAML manifests

A *deployment* represents one or more identical pods, managed by the Kubernetes Deployment Controller. A deployment defines the number of *replicas* (pods) to create, and the Kubernetes Scheduler ensures that if pods or nodes encounter problems, additional pods are scheduled on healthy nodes.

You can update deployments to change the configuration of pods, container image used, or attached storage. The Deployment Controller drains and terminates a given number of replicas, creates replicas from the new deployment definition, and continues the process until all replicas in the deployment are updated.

Most stateless applications in AKS should use the deployment model rather than scheduling individual pods. Kubernetes can monitor the health and status of deployments to ensure that the required number of replicas run within the cluster. When you only schedule individual pods, the pods aren't restarted if they encounter a problem, and aren't rescheduled on healthy nodes if their current node encounters a problem.

If an application requires a quorum of instances to always be available for management decisions to be made, you don't want an update process to disrupt that ability. *Pod Disruption Budgets* can be used to define how many replicas in a deployment can be taken down during an update or node upgrade. For example, if you have 5 replicas in your deployment, you can define a pod disruption of 4 to only permit one replica from being deleted/rescheduled at a time. As with pod resource limits, a best practice is to define pod disruption budgets on applications that require a minimum number of replicas to always be present.

Deployments are typically created and managed with `kubectl create` or `kubectl apply`. To create a deployment, you define a manifest file in the YAML (YAML Ain't Markup Language) format. The following example creates a basic deployment of the NGINX web server. The deployment specifies 3 replicas to be created, and that port 80 be open on the container. Resource requests and limits are also defined for CPU and memory.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.2
        ports:
        - containerPort: 80
      resources:
        requests:
          cpu: 250m
          memory: 64Mi
        limits:
          cpu: 500m
          memory: 256Mi

```

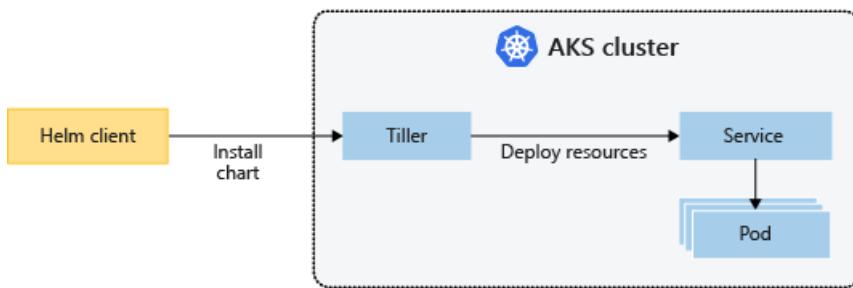
More complex applications can be created by also including services such as load balancers within the YAML manifest.

For more information, see [Kubernetes deployments](#).

### Package management with Helm

A common approach to managing applications in Kubernetes is with [Helm](#). You can build and use existing public Helm *charts* that contain a packaged version of application code and Kubernetes YAML manifests to deploy resources. These Helm charts can be stored locally, or often in a remote repository, such as an [Azure Container Registry Helm chart repo](#).

To use Helm, a server component called *Tiller* is installed in your Kubernetes cluster. The Tiller manages the installation of charts within the cluster. The Helm client itself is installed locally on your computer, or can be used within the [Azure Cloud Shell](#). You can search for or create Helm charts with the client, and then install them to your Kubernetes cluster.



For more information, see [Install applications with Helm in Azure Kubernetes Service \(AKS\)](#).

## StatefulSets and DaemonSets

The Deployment Controller uses the Kubernetes Scheduler to run a given number of replicas on any available node with available resources. This approach of using deployments may be sufficient for stateless applications, but not for applications that require a persistent naming convention or storage. For applications that require a replica to exist on each node, or selected nodes, within a cluster, the Deployment Controller doesn't look at how

replicas are distributed across the nodes.

There are two Kubernetes resources that let you manage these types of applications:

- *StatefulSets* - Maintain the state of applications beyond an individual pod lifecycle, such as storage.
- *DaemonSets* - Ensure a running instance on each node, early in the Kubernetes bootstrap process.

## StatefulSets

Modern application development often aims for stateless applications, but *StatefulSets* can be used for stateful applications, such as applications that include database components. A StatefulSet is similar to a deployment in that one or more identical pods are created and managed. Replicas in a StatefulSet follow a graceful, sequential approach to deployment, scale, upgrades, and terminations. With a StatefulSet (as replicas are rescheduled) the naming convention, network names, and storage persist.

You define the application in YAML format using `kind: StatefulSet`, and the StatefulSet Controller then handles the deployment and management of the required replicas. Data is written to persistent storage, provided by Azure Managed Disks or Azure Files. With StatefulSets, the underlying persistent storage remains even when the StatefulSet is deleted.

For more information, see [Kubernetes StatefulSets](#).

Replicas in a StatefulSet are scheduled and run across any available node in an AKS cluster. If you need to ensure that at least one pod in your Set runs on a node, you can instead use a DaemonSet.

## DaemonSets

For specific log collection or monitoring needs, you may need to run a given pod on all, or selected, nodes. A *DaemonSet* is again used to deploy one or more identical pods, but the DaemonSet Controller ensures that each node specified runs an instance of the pod.

The DaemonSet Controller can schedule pods on nodes early in the cluster boot process, before the default Kubernetes scheduler has started. This ability ensures that the pods in a DaemonSet are started before traditional pods in a Deployment or StatefulSet are scheduled.

Like StatefulSets, a DaemonSet is defined as part of a YAML definition using `kind: DaemonSet`.

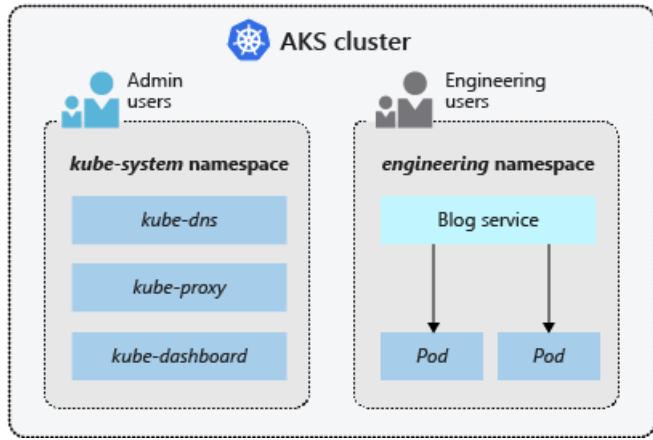
For more information, see [Kubernetes DaemonSets](#).

### NOTE

If using the [Virtual Nodes add-on](#), DaemonSets will not create pods on the virtual node.

## Namespaces

Kubernetes resources, such as pods and Deployments, are logically grouped into a *namespace*. These groupings provide a way to logically divide an AKS cluster and restrict access to create, view, or manage resources. You can create namespaces to separate business groups, for example. Users can only interact with resources within their assigned namespaces.



When you create an AKS cluster, the following namespaces are available:

- *default* - This namespace is where pods and deployments are created by default when none is provided. In smaller environments, you can deploy applications directly into the default namespace without creating additional logical separations. When you interact with the Kubernetes API, such as with `kubectl get pods`, the default namespace is used when none is specified.
- *kube-system* - This namespace is where core resources exist, such as network features like DNS and proxy, or the Kubernetes dashboard. You typically don't deploy your own applications into this namespace.
- *kube-public* - This namespace is typically not used, but can be used for resources to be visible across the whole cluster, and can be viewed by any user.

For more information, see [Kubernetes namespaces](#).

## Next steps

This article covers some of the core Kubernetes components and how they apply to AKS clusters. For additional information on core Kubernetes and AKS concepts, see the following articles:

- [Kubernetes / AKS access and identity](#)
- [Kubernetes / AKS security](#)
- [Kubernetes / AKS virtual networks](#)
- [Kubernetes / AKS storage](#)
- [Kubernetes / AKS scale](#)

# Access and identity options for Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

There are different ways to authenticate with and secure Kubernetes clusters. Using role-based access controls (RBAC), you can grant users or groups access to only the resources they need. With Azure Kubernetes Service (AKS), you can further enhance the security and permissions structure by using Azure Active Directory. These approaches help you secure your application workloads and customer data.

This article introduces the core concepts that help you authenticate and assign permissions in AKS:

- [Kubernetes service accounts](#)
- [Azure Active Directory integration](#)
- [Role-based access controls \(RBAC\)](#)
- [Roles and ClusterRoles](#)
- [RoleBindings and ClusterRoleBindings](#)

## Kubernetes service accounts

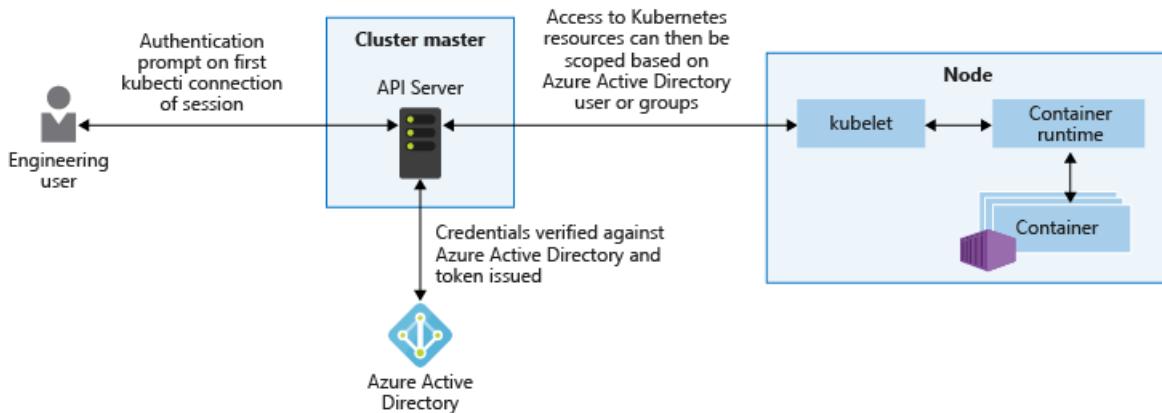
One of the primary user types in Kubernetes is a *service account*. A service account exists in, and is managed by, the Kubernetes API. The credentials for service accounts are stored as Kubernetes secrets, which allows them to be used by authorized pods to communicate with the API Server. Most API requests provide an authentication token for a service account or a normal user account.

Normal user accounts allow more traditional access for human administrators or developers, not just services and processes. Kubernetes itself doesn't provide an identity management solution where regular user accounts and passwords are stored. Instead, external identity solutions can be integrated into Kubernetes. For AKS clusters, this integrated identity solution is Azure Active Directory.

For more information on the identity options in Kubernetes, see [Kubernetes authentication](#).

## Azure Active Directory integration

The security of AKS clusters can be enhanced with the integration of Azure Active Directory (AD). Built on decades of enterprise identity management, Azure AD is a multi-tenant, cloud-based directory, and identity management service that combines core directory services, application access management, and identity protection. With Azure AD, you can integrate on-premises identities into AKS clusters to provide a single source for account management and security.



With Azure AD-integrated AKS clusters, you can grant users or groups access to Kubernetes resources within a namespace or across the cluster. To obtain a `kubectl` configuration context, a user can run the [az aks get-credentials](#) command. When a user then interacts with the AKS cluster with `kubectl`, they are prompted to sign in with their Azure AD credentials. This approach provides a single source for user account management and password credentials. The user can only access the resources as defined by the cluster administrator.

Azure AD authentication in AKS clusters uses OpenID Connect, an identity layer built on top of the OAuth 2.0 protocol. OAuth 2.0 defines mechanisms to obtain and use access tokens to access protected resources, and OpenID Connect implements authentication as an extension to the OAuth 2.0 authorization process. For more information on OpenID Connect, see the [Open ID Connect documentation](#). To verify the authentication tokens obtained from Azure AD through OpenID Connect, AKS clusters use Kubernetes Webhook Token Authentication. For more information, see the [Webhook Token Authentication documentation](#).

## Role-based access controls (RBAC)

To provide granular filtering of the actions that users can perform, Kubernetes uses role-based access controls (RBAC). This control mechanism lets you assign users, or groups of users, permission to do things like create or modify resources, or view logs from running application workloads. These permissions can be scoped to a single namespace, or granted across the entire AKS cluster. With Kubernetes RBAC, you create *roles* to define permissions, and then assign those roles to users with *role bindings*.

For more information, see [Using RBAC authorization](#).

### Azure role-based access controls (RBAC)

One additional mechanism for controlling access to resources is Azure role-based access controls (RBAC). Kubernetes RBAC is designed to work on resources within your AKS cluster, and Azure RBAC is designed to work on resources within your Azure subscription. With Azure RBAC, you create a *role definition* that outlines the permissions to be applied. A user or group is then assigned this role definition for a particular *scope*, which could be an individual resource, a resource group, or across the subscription.

For more information, see [What is Azure RBAC?](#)

## Roles and ClusterRoles

Before you assign permissions to users with Kubernetes RBAC, you first define those permissions as a *Role*. Kubernetes roles *grant* permissions. There is no concept of a *deny* permission.

Roles are used to grant permissions within a namespace. If you need to grant permissions across the entire cluster, or to cluster resources outside a given namespace, you can instead use *ClusterRoles*.

A ClusterRole works in the same way to grant permissions to resources, but can be applied to resources across the entire cluster, not a specific namespace.

## RoleBindings and ClusterRoleBindings

Once roles are defined to grant permissions to resources, you assign those Kubernetes RBAC permissions with a *RoleBinding*. If your AKS cluster integrates with Azure Active Directory, bindings are how those Azure AD users are granted permissions to perform actions within the cluster.

Role bindings are used to assign roles for a given namespace. This approach lets you logically segregate a single AKS cluster, with users only able to access the application resources in their assigned namespace. If you need to bind roles across the entire cluster, or to cluster resources outside a given namespace, you can instead use *ClusterRoleBindings*.

A ClusterRoleBinding works in the same way to bind roles to users, but can be applied to resources across the entire cluster, not a specific namespace. This approach lets you grant administrators or support engineers access

to all resources in the AKS cluster.

## Next steps

To get started with Azure AD and Kubernetes RBAC, see [Integrate Azure Active Directory with AKS](#).

For associated best practices, see [Best practices for authentication and authorization in AKS](#).

For additional information on core Kubernetes and AKS concepts, see the following articles:

- [Kubernetes / AKS clusters and workloads](#)
- [Kubernetes / AKS security](#)
- [Kubernetes / AKS virtual networks](#)
- [Kubernetes / AKS storage](#)
- [Kubernetes / AKS scale](#)

# Security concepts for applications and clusters in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

To protect your customer data as you run application workloads in Azure Kubernetes Service (AKS), the security of your cluster is a key consideration. Kubernetes includes security components such as *network policies* and *Secrets*. Azure then adds in components such as network security groups and orchestrated cluster upgrades. These security components are combined to keep your AKS cluster running the latest OS security updates and Kubernetes releases, and with secure pod traffic and access to sensitive credentials.

This article introduces the core concepts that secure your applications in AKS:

- [Master components security](#)
- [Node security](#)
- [Cluster upgrades](#)
- [Network security](#)
- [Kubernetes Secrets](#)

## Master security

In AKS, the Kubernetes master components are part of the managed service provided by Microsoft. Each AKS cluster has its own single-tenanted, dedicated Kubernetes master to provide the API Server, Scheduler, etc. This master is managed and maintained by Microsoft.

By default, the Kubernetes API server uses a public IP address and a fully qualified domain name (FQDN). You can control access to the API server using Kubernetes role-based access controls and Azure Active Directory. For more information, see [Azure AD integration with AKS](#).

## Node security

AKS nodes are Azure virtual machines that you manage and maintain. Linux nodes run an optimized Ubuntu distribution using the Moby container runtime. Windows Server nodes (currently in preview in AKS) run an optimized Windows Server 2019 release and also use the Moby container runtime. When an AKS cluster is created or scaled up, the nodes are automatically deployed with the latest OS security updates and configurations.

The Azure platform automatically applies OS security patches to Linux nodes on a nightly basis. If a Linux OS security update requires a host reboot, that reboot is not automatically performed. You can manually reboot the Linux nodes, or a common approach is to use [Kured](#), an open-source reboot daemon for Kubernetes. Kured runs as a [DaemonSet](#) and monitors each node for the presence of a file indicating that a reboot is required. Reboots are managed across the cluster using the same [cordon and drain process](#) as a cluster upgrade.

For Windows Server nodes (currently in preview in AKS), Windows Update does not automatically run and apply the latest updates. On a regular schedule around the Windows Update release cycle and your own validation process, you should perform an upgrade on the Windows Server node pool(s) in your AKS cluster. This upgrade process creates nodes that run the latest Windows Server image and patches, then removes the older nodes. For more information on this process, see [Upgrade a node pool in AKS](#).

Nodes are deployed into a private virtual network subnet, with no public IP addresses assigned. For troubleshooting and management purposes, SSH is enabled by default. This SSH access is only available using the internal IP address.

To provide storage, the nodes use Azure Managed Disks. For most VM node sizes, these are Premium disks backed by high-performance SSDs. The data stored on managed disks is automatically encrypted at rest within the Azure platform. To improve redundancy, these disks are also securely replicated within the Azure datacenter.

Kubernetes environments, in AKS or elsewhere, currently aren't completely safe for hostile multi-tenant usage. Additional security features such as *Pod Security Policies* or more fine-grained role-based access controls (RBAC) for nodes make exploits more difficult. However, for true security when running hostile multi-tenant workloads, a hypervisor is the only level of security that you should trust. The security domain for Kubernetes becomes the entire cluster, not an individual node. For these types of hostile multi-tenant workloads, you should use physically isolated clusters. For more information on ways to isolate workloads, see [Best practices for cluster isolation in AKS](#),

## Cluster upgrades

For security and compliance, or to use the latest features, Azure provides tools to orchestrate the upgrade of an AKS cluster and components. This upgrade orchestration includes both the Kubernetes master and agent components. You can view a [list of available Kubernetes versions](#) for your AKS cluster. To start the upgrade process, you specify one of these available versions. Azure then safely cordons and drains each AKS node and performs the upgrade.

### Cordon and drain

During the upgrade process, AKS nodes are individually cordoned from the cluster so new pods aren't scheduled on them. The nodes are then drained and upgraded as follows:

- A new node is deployed into the node pool. This node runs the latest OS image and patches.
- One of the existing nodes is identified for upgrade. Pods on this node are gracefully terminated and scheduled on the other nodes in the node pool.
- This existing node is deleted from the AKS cluster.
- The next node in the cluster is cordoned and drained using the same process until all nodes are successfully replaced as part of the upgrade process.

For more information, see [Upgrade an AKS cluster](#).

## Network security

For connectivity and security with on-premises networks, you can deploy your AKS cluster into existing Azure virtual network subnets. These virtual networks may have an Azure Site-to-Site VPN or Express Route connection back to your on-premises network. Kubernetes ingress controllers can be defined with private, internal IP addresses so services are only accessible over this internal network connection.

### Azure network security groups

To filter the flow of traffic in virtual networks, Azure uses network security group rules. These rules define the source and destination IP ranges, ports, and protocols that are allowed or denied access to resources. Default rules are created to allow TLS traffic to the Kubernetes API server. As you create services with load balancers, port mappings, or ingress routes, AKS automatically modifies the network security group for traffic to flow appropriately.

## Kubernetes Secrets

A Kubernetes *Secret* is used to inject sensitive data into pods, such as access credentials or keys. You first create a Secret using the Kubernetes API. When you define your pod or deployment, a specific Secret can be requested. Secrets are only provided to nodes that have a scheduled pod that requires it, and the Secret is stored in *tmpfs*, not written to disk. When the last pod on a node that requires a Secret is deleted, the Secret is deleted from the node's *tmpfs*. Secrets are stored within a given namespace and can only be accessed by pods within the same

namespace.

The use of Secrets reduces the sensitive information that is defined in the pod or service YAML manifest. Instead, you request the Secret stored in Kubernetes API Server as part of your YAML manifest. This approach only provides the specific pod access to the Secret. Please note: the raw secret manifest files contains the secret data in base64 format (see the [official documentation](#) for more details). Therefore, this file should be treated as sensitive information, and never committed to source control.

## Next steps

To get started with securing your AKS clusters, see [Upgrade an AKS cluster](#).

For associated best practices, see [Best practices for cluster security and upgrades in AKS](#) and [Best practices for pod security in AKS](#).

For additional information on core Kubernetes and AKS concepts, see the following articles:

- [Kubernetes / AKS clusters and workloads](#)
- [Kubernetes / AKS identity](#)
- [Kubernetes / AKS virtual networks](#)
- [Kubernetes / AKS storage](#)
- [Kubernetes / AKS scale](#)

# Network concepts for applications in Azure Kubernetes Service (AKS)

2/25/2020 • 9 minutes to read • [Edit Online](#)

In a container-based microservices approach to application development, application components must work together to process their tasks. Kubernetes provides various resources that enable this application communication. You can connect to and expose applications internally or externally. To build highly available applications, you can load balance your applications. More complex applications may require configuration of ingress traffic for SSL/TLS termination or routing of multiple components. For security reasons, you may also need to restrict the flow of network traffic into or between pods and nodes.

This article introduces the core concepts that provide networking to your applications in AKS:

- [Services](#)
- [Azure virtual networks](#)
- [Ingress controllers](#)
- [Network policies](#)

## Kubernetes basics

To allow access to your applications, or for application components to communicate with each other, Kubernetes provides an abstraction layer to virtual networking. Kubernetes nodes are connected to a virtual network, and can provide inbound and outbound connectivity for pods. The *kube-proxy* component runs on each node to provide these network features.

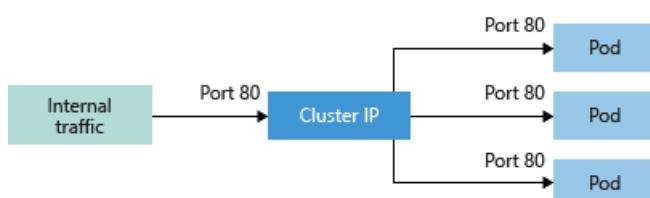
In Kubernetes, *Services* logically group pods to allow for direct access via an IP address or DNS name and on a specific port. You can also distribute traffic using a *load balancer*. More complex routing of application traffic can also be achieved with *Ingress Controllers*. Security and filtering of the network traffic for pods is possible with Kubernetes *network policies*.

The Azure platform also helps to simplify virtual networking for AKS clusters. When you create a Kubernetes load balancer, the underlying Azure load balancer resource is created and configured. As you open network ports to pods, the corresponding Azure network security group rules are configured. For HTTP application routing, Azure can also configure *external DNS* as new ingress routes are configured.

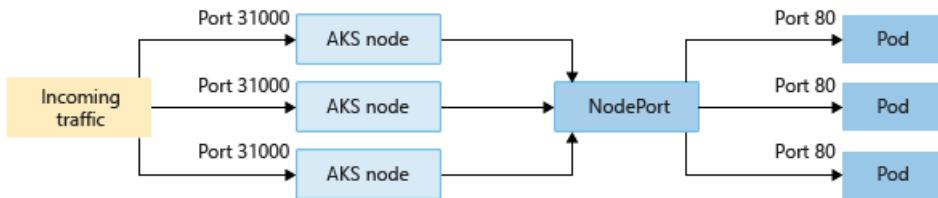
## Services

To simplify the network configuration for application workloads, Kubernetes uses *Services* to logically group a set of pods together and provide network connectivity. The following Service types are available:

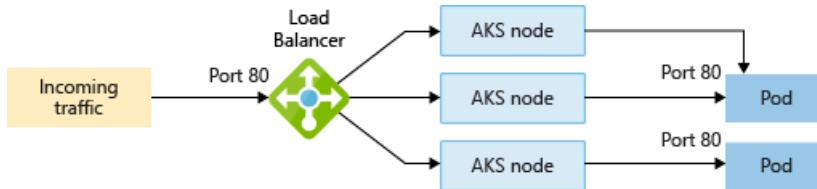
- **Cluster IP** - Creates an internal IP address for use within the AKS cluster. Good for internal-only applications that support other workloads within the cluster.



- **NodePort** - Creates a port mapping on the underlying node that allows the application to be accessed directly with the node IP address and port.



- **LoadBalancer** - Creates an Azure load balancer resource, configures an external IP address, and connects the requested pods to the load balancer backend pool. To allow customers' traffic to reach the application, load balancing rules are created on the desired ports.



For additional control and routing of the inbound traffic, you may instead use an [Ingress controller](#).

- **ExternalName** - Creates a specific DNS entry for easier application access.

The IP address for load balancers and services can be dynamically assigned, or you can specify an existing static IP address to use. Both internal and external static IP addresses can be assigned. This existing static IP address is often tied to a DNS entry.

Both *internal* and *external* load balancers can be created. Internal load balancers are only assigned a private IP address, so they can't be accessed from the Internet.

## Azure virtual networks

In AKS, you can deploy a cluster that uses one of the following two network models:

- *Kubenet* networking - The network resources are typically created and configured as the AKS cluster is deployed.
- *Azure Container Networking Interface (CNI)* networking - The AKS cluster is connected to existing virtual network resources and configurations.

### Kubenet (basic) networking

The *kubenet* networking option is the default configuration for AKS cluster creation. With *kubenet*, nodes get an IP address from the Azure virtual network subnet. Pods receive an IP address from a logically different address space to the Azure virtual network subnet of the nodes. Network address translation (NAT) is then configured so that the pods can reach resources on the Azure virtual network. The source IP address of the traffic is NAT'd to the node's primary IP address.

Nodes use the [kubenet](#) Kubernetes plugin. You can let the Azure platform create and configure the virtual networks for you, or choose to deploy your AKS cluster into an existing virtual network subnet. Again, only the nodes receive a routable IP address, and the pods use NAT to communicate with other resources outside the AKS cluster. This approach greatly reduces the number of IP addresses that you need to reserve in your network space for pods to use.

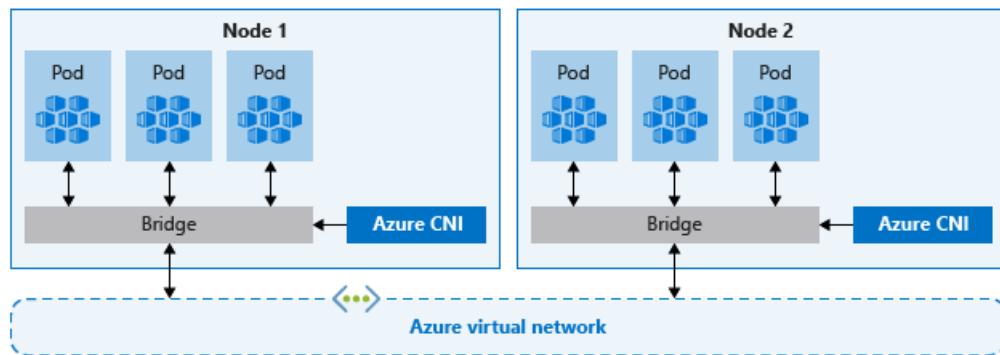
For more information, see [Configure kubenet networking for an AKS cluster](#).

### Azure CNI (advanced) networking

With Azure CNI, every pod gets an IP address from the subnet and can be accessed directly. These IP addresses must be unique across your network space, and must be planned in advance. Each node has a configuration parameter for the maximum number of pods that it supports. The equivalent number of IP addresses per node are then reserved up front for that node. This approach requires more planning, as can otherwise lead to IP

address exhaustion or the need to rebuild clusters in a larger subnet as your application demands grow.

Nodes use the [Azure Container Networking Interface \(CNI\)](#) Kubernetes plugin.



For more information, see [Configure Azure CNI for an AKS cluster](#).

### Compare network models

Both kubenet and Azure CNI provide network connectivity for your AKS clusters. However, there are advantages and disadvantages to each. At a high level, the following considerations apply:

- **kubenet**

- Conserves IP address space.
- Uses Kubernetes internal or external load balancer to reach pods from outside of the cluster.
- You must manually manage and maintain user-defined routes (UDRs).
- Maximum of 400 nodes per cluster.

- **Azure CNI**

- Pods get full virtual network connectivity and can be directly reached via their private IP address from connected networks.
- Requires more IP address space.

The following behavior differences exist between kubenet and Azure CNI:

CAPABILITY	KUBENET	AZURE CNI
Deploy cluster in existing or new virtual network	Supported - UDRs manually applied	Supported
Pod-pod connectivity	Supported	Supported
Pod-VM connectivity; VM in the same virtual network	Works when initiated by pod	Works both ways
Pod-VM connectivity; VM in peered virtual network	Works when initiated by pod	Works both ways
On-premises access using VPN or Express Route	Works when initiated by pod	Works both ways
Access to resources secured by service endpoints	Supported	Supported
Expose Kubernetes services using a load balancer service, App Gateway, or ingress controller	Supported	Supported

CAPABILITY	KUBENET	AZURE CNI
Default Azure DNS and Private Zones	Supported	Supported

Regarding DNS, with both kubenet and Azure CNI plugins DNS is offered by CoreDNS, a daemon set running in AKS. For more information on CoreDNS on Kubernetes see [Customizing DNS Service](#). CoreDNS is configured per default to forward unknown domains to the node DNS servers, in other words, to the DNS functionality of the Azure Virtual Network where the AKS cluster is deployed. Hence, Azure DNS and Private Zones will work for pods running in AKS.

### Support scope between network models

Regardless of the network model you use, both kubenet and Azure CNI can be deployed in one of the following ways:

- The Azure platform can automatically create and configure the virtual network resources when you create an AKS cluster.
- You can manually create and configure the virtual network resources and attach to those resources when you create your AKS cluster.

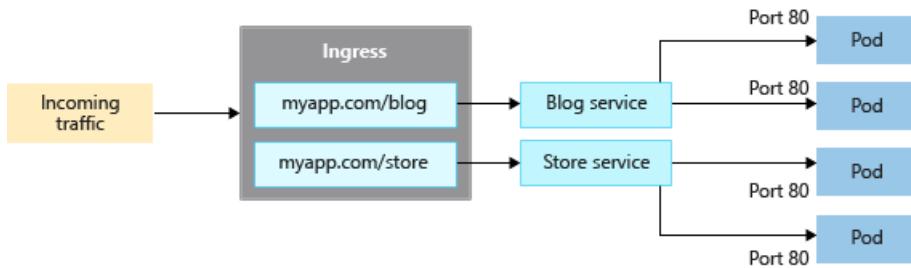
Although capabilities like service endpoints or UDRs are supported with both kubenet and Azure CNI, the [support policies for AKS](#) define what changes you can make. For example:

- If you manually create the virtual network resources for an AKS cluster, you're supported when configuring your own UDRs or service endpoints.
- If the Azure platform automatically creates the virtual network resources for your AKS cluster, it isn't supported to manually change those AKS-managed resources to configure your own UDRs or service endpoints.

## Ingress controllers

When you create a LoadBalancer type Service, an underlying Azure load balancer resource is created. The load balancer is configured to distribute traffic to the pods in your Service on a given port. The LoadBalancer only works at layer 4 - the Service is unaware of the actual applications, and can't make any additional routing considerations.

*Ingress controllers* work at layer 7, and can use more intelligent rules to distribute application traffic. A common use of an Ingress controller is to route HTTP traffic to different applications based on the inbound URL.



In AKS, you can create an Ingress resource using something like NGINX, or use the AKS HTTP application routing feature. When you enable HTTP application routing for an AKS cluster, the Azure platform creates the Ingress controller and an *External-DNS* controller. As new Ingress resources are created in Kubernetes, the required DNS A records are created in a cluster-specific DNS zone. For more information, see [deploy HTTP application routing](#).

Another common feature of Ingress is SSL/TLS termination. On large web applications accessed via HTTPS, the TLS termination can be handled by the Ingress resource rather than within the application itself. To provide automatic TLS certification generation and configuration, you can configure the Ingress resource to use

providers such as Let's Encrypt. For more information on configuring an NGINX Ingress controller with Let's Encrypt, see [Ingress and TLS](#).

You can also configure your ingress controller to preserve the client source IP on requests to containers in your AKS cluster. When a client's request is routed to a container in your AKS cluster via your ingress controller, the original source IP of that request won't be available to the target container. When you enable *client source IP preservation*, the source IP for the client is available in the request header under *X-Forwarded-For*. If you're using client source IP preservation on your ingress controller, you can't use SSL pass-through. Client source IP preservation and SSL pass-through can be used with other services, such as the *LoadBalancer* type.

## Network security groups

A network security group filters traffic for VMs, such as the AKS nodes. As you create Services, such as a LoadBalancer, the Azure platform automatically configures any network security group rules that are needed. Don't manually configure network security group rules to filter traffic for pods in an AKS cluster. Define any required ports and forwarding as part of your Kubernetes Service manifests, and let the Azure platform create or update the appropriate rules. You can also use network policies, as discussed in the next section, to automatically apply traffic filter rules to pods.

## Network policies

By default, all pods in an AKS cluster can send and receive traffic without limitations. For improved security, you may want to define rules that control the flow of traffic. Backend applications are often only exposed to required frontend services, or database components are only accessible to the application tiers that connect to them.

Network policy is a Kubernetes feature available in AKS that lets you control the traffic flow between pods. You can choose to allow or deny traffic based on settings such as assigned labels, namespace, or traffic port. Network security groups are more for the AKS nodes, not pods. The use of network policies is a more suitable, cloud-native way to control the flow of traffic. As pods are dynamically created in an AKS cluster, the required network policies can be automatically applied.

For more information, see [Secure traffic between pods using network policies in Azure Kubernetes Service \(AKS\)](#).

## Next steps

To get started with AKS networking, create and configure an AKS cluster with your own IP address ranges using [kubenet](#) or [Azure CNI](#).

For associated best practices, see [Best practices for network connectivity and security in AKS](#).

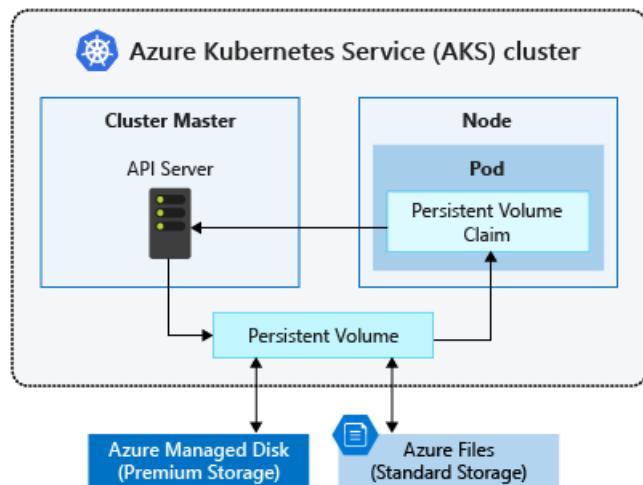
For additional information on core Kubernetes and AKS concepts, see the following articles:

- [Kubernetes / AKS clusters and workloads](#)
- [Kubernetes / AKS access and identity](#)
- [Kubernetes / AKS security](#)
- [Kubernetes / AKS storage](#)
- [Kubernetes / AKS scale](#)

# Storage options for applications in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

Applications that run in Azure Kubernetes Service (AKS) may need to store and retrieve data. For some application workloads, this data storage can use local, fast storage on the node that is no longer needed when the pods are deleted. Other application workloads may require storage that persists on more regular data volumes within the Azure platform. Multiple pods may need to share the same data volumes, or reattach data volumes if the pod is rescheduled on a different node. Finally, you may need to inject sensitive data or application configuration information into pods.



This article introduces the core concepts that provide storage to your applications in AKS:

- [Volumes](#)
- [Persistent volumes](#)
- [Storage classes](#)
- [Persistent volume claims](#)

## Volumes

Applications often need to be able to store and retrieve data. As Kubernetes typically treats individual pods as ephemeral, disposable resources, different approaches are available for applications to use and persist data as necessary. A *volume* represents a way to store, retrieve, and persist data across pods and through the application lifecycle.

Traditional volumes to store and retrieve data are created as Kubernetes resources backed by Azure Storage. You can manually create these data volumes to be assigned to pods directly, or have Kubernetes automatically create them. These data volumes can use Azure Disks or Azure Files:

- *Azure Disks* can be used to create a Kubernetes *DataDisk* resource. Disks can use Azure Premium storage, backed by high-performance SSDs, or Azure Standard storage, backed by regular HDDs. For most production and development workloads, use Premium storage. Azure Disks are mounted as *ReadWriteOnce*, so are only available to a single pod. For storage volumes that can be accessed by multiple pods simultaneously, use Azure Files.
- *Azure Files* can be used to mount an SMB 3.0 share backed by an Azure Storage account to pods. Files let you share data across multiple nodes and pods. Files can use Azure Standard storage backed by regular HDDs, or

Azure Premium storage, backed by high-performance SSDs.

#### NOTE

Azure Files support premium storage in AKS clusters that run Kubernetes 1.13 or higher.

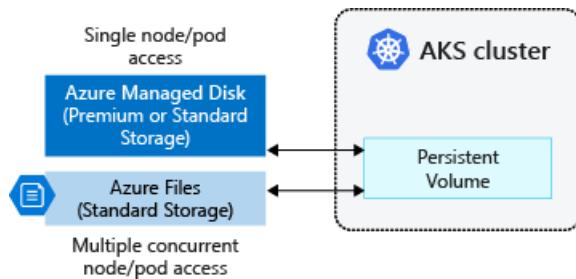
In Kubernetes, volumes can represent more than just a traditional disk where information can be stored and retrieved. Kubernetes volumes can also be used as a way to inject data into a pod for use by the containers. Common additional volume types in Kubernetes include:

- *emptyDir* - This volume is commonly used as temporary space for a pod. All containers within a pod can access the data on the volume. Data written to this volume type persists only for the lifespan of the pod - when the pod is deleted, the volume is deleted. This volume typically uses the underlying local node disk storage, though it can also exist only in the node's memory.
- *secret* - This volume is used to inject sensitive data into pods, such as passwords. You first create a Secret using the Kubernetes API. When you define your pod or deployment, a specific Secret can be requested. Secrets are only provided to nodes that have a scheduled pod that requires it, and the Secret is stored in *tmpfs*, not written to disk. When the last pod on a node that requires a Secret is deleted, the Secret is deleted from the node's *tmpfs*. Secrets are stored within a given namespace and can only be accessed by pods within the same namespace.
- *configMap* - This volume type is used to inject key-value pair properties into pods, such as application configuration information. Rather than defining application configuration information within a container image, you can define it as a Kubernetes resource that can be easily updated and applied to new instances of pods as they are deployed. Like using a Secret, you first create a ConfigMap using the Kubernetes API. This ConfigMap can then be requested when you define a pod or deployment. ConfigMaps are stored within a given namespace and can only be accessed by pods within the same namespace.

## Persistent volumes

Volumes that are defined and created as part of the pod lifecycle only exist until the pod is deleted. Pods often expect their storage to remain if a pod is rescheduled on a different host during a maintenance event, especially in StatefulSets. A *persistent volume* (PV) is a storage resource created and managed by the Kubernetes API that can exist beyond the lifetime of an individual pod.

Azure Disks or Files are used to provide the PersistentVolume. As noted in the previous section on Volumes, the choice of Disks or Files is often determined by the need for concurrent access to the data or the performance tier.



A PersistentVolume can be *statically* created by a cluster administrator, or *dynamically* created by the Kubernetes API server. If a pod is scheduled and requests storage that is not currently available, Kubernetes can create the underlying Azure Disk or Files storage and attach it to the pod. Dynamic provisioning uses a *StorageClass* to identify what type of Azure storage needs to be created.

## Storage classes

To define different tiers of storage, such as Premium and Standard, you can create a *StorageClass*. The *StorageClass* also defines the *reclaimPolicy*. This reclaimPolicy controls the behavior of the underlying Azure

storage resource when the pod is deleted and the persistent volume may no longer be required. The underlying storage resource can be deleted, or retained for use with a future pod.

In AKS, two initial StorageClasses are created:

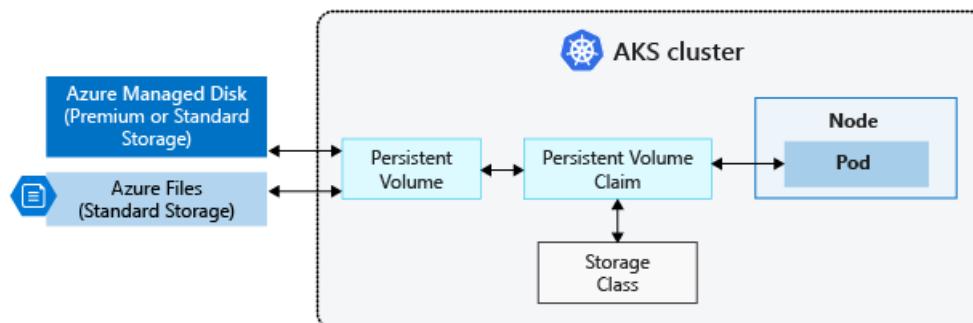
- *default* - Uses Azure Standard storage to create a Managed Disk. The reclaim policy indicates that the underlying Azure Disk is deleted when the persistent volume that used it is deleted.
- *managed-premium* - Uses Azure Premium storage to create Managed Disk. The reclaim policy again indicates that the underlying Azure Disk is deleted when the persistent volume that used it is deleted.

If no StorageClass is specified for a persistent volume, the default StorageClass is used. Take care when requesting persistent volumes so that they use the appropriate storage you need. You can create a StorageClass for additional needs using `kubectl`. The following example uses Premium Managed Disks and specifies that the underlying Azure Disk should be *retained* when the pod is deleted:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: managed-premium-retain
provisioner: kubernetes.io/azure-disk
reclaimPolicy: Retain
parameters:
  storageaccounttype: Premium_LRS
  kind: Managed
```

## Persistent volume claims

A PersistentVolumeClaim requests either Disk or File storage of a particular StorageClass, access mode, and size. The Kubernetes API server can dynamically provision the underlying storage resource in Azure if there is no existing resource to fulfill the claim based on the defined StorageClass. The pod definition includes the volume mount once the volume has been connected to the pod.



A PersistentVolume is *bound* to a PersistentVolumeClaim once an available storage resource has been assigned to the pod requesting it. There is a 1:1 mapping of persistent volumes to claims.

The following example YAML manifest shows a persistent volume claim that uses the *managed-premium* StorageClass and requests a Disk 5Gi in size:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: managed-premium
  resources:
    requests:
      storage: 5Gi
```

When you create a pod definition, the persistent volume claim is specified to request the desired storage. You also then specify the *volumeMount* for your applications to read and write data. The following example YAML manifest shows how the previous persistent volume claim can be used to mount a volume at */mnt/azure*:

```
kind: Pod
apiVersion: v1
metadata:
  name: nginx
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
        - mountPath: "/mnt/azure"
          name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: azure-managed-disk
```

## Next steps

For associated best practices, see [Best practices for storage and backups in AKS](#).

To see how to create dynamic and static volumes that use Azure Disks or Azure Files, see the following how-to articles:

- [Create a static volume using Azure Disks](#)
- [Create a static volume using Azure Files](#)
- [Create a dynamic volume using Azure Disks](#)
- [Create a dynamic volume using Azure Files](#)

For additional information on core Kubernetes and AKS concepts, see the following articles:

- [Kubernetes / AKS clusters and workloads](#)
- [Kubernetes / AKS identity](#)
- [Kubernetes / AKS security](#)
- [Kubernetes / AKS virtual networks](#)
- [Kubernetes / AKS scale](#)

# Scaling options for applications in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

As you run applications in Azure Kubernetes Service (AKS), you may need to increase or decrease the amount of compute resources. As the number of application instances you need change, the number of underlying Kubernetes nodes may also need to change. You also might need to quickly provision a large number of additional application instances.

This article introduces the core concepts that help you scale applications in AKS:

- [Manually scale](#)
- [Horizontal pod autoscaler \(HPA\)](#)
- [Cluster autoscaler](#)
- [Azure Container Instance \(ACI\) integration with AKS](#)

## Manually scale pods or nodes

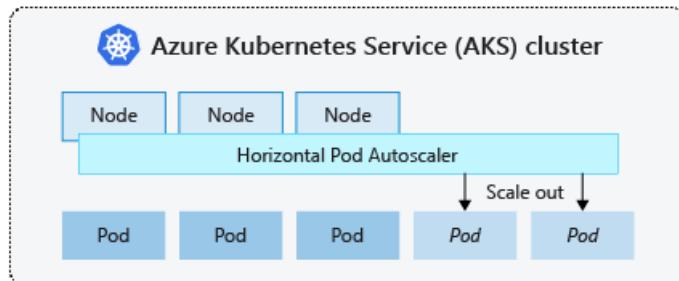
You can manually scale replicas (pods) and nodes to test how your application responds to a change in available resources and state. Manually scaling resources also lets you define a set amount of resources to use to maintain a fixed cost, such as the number of nodes. To manually scale, you define the replica or node count. The Kubernetes API then schedules creating additional pods or draining nodes based on that replica or node count.

When scaling down nodes, the Kubernetes API calls the relevant Azure Compute API tied to the compute type used by your cluster. For example, for clusters built on VM Scale Sets the logic for selecting which nodes to remove is determined by the VM Scale Sets API. To learn more about how nodes are selected for removal on scale down, see the [VMSS FAQ](#).

To get started with manually scaling pods and nodes see [Scale applications in AKS](#).

## Horizontal pod autoscaler

Kubernetes uses the horizontal pod autoscaler (HPA) to monitor the resource demand and automatically scale the number of replicas. By default, the horizontal pod autoscaler checks the Metrics API every 30 seconds for any required changes in replica count. When changes are required, the number of replicas is increased or decreased accordingly. Horizontal pod autoscaler works with AKS clusters that have deployed the Metrics Server for Kubernetes 1.8+.



When you configure the horizontal pod autoscaler for a given deployment, you define the minimum and maximum number of replicas that can run. You also define the metric to monitor and base any scaling decisions on, such as CPU usage.

To get started with the horizontal pod autoscaler in AKS, see [Autoscale pods in AKS](#).

### Cooldown of scaling events

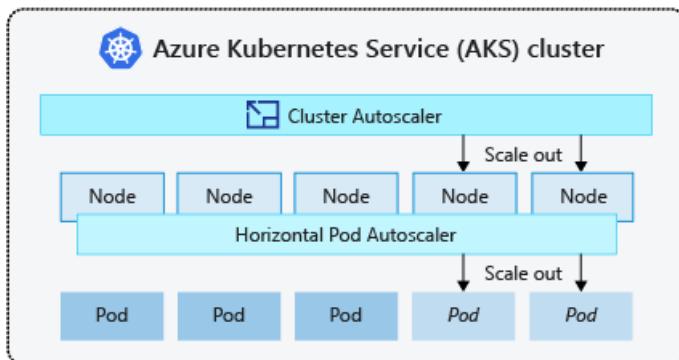
As the horizontal pod autoscaler checks the Metrics API every 30 seconds, previous scale events may not have successfully completed before another check is made. This behavior could cause the horizontal pod autoscaler to change the number of replicas before the previous scale event could receive application workload and the resource demands to adjust accordingly.

To minimize these race events, cooldown or delay values are set. These values define how long the horizontal pod autoscaler must wait after a scale event before another scale event can be triggered. This behavior allows the new replica count to take effect and the Metrics API to reflect the distributed workload. By default, the delay on scale up events is 3 minutes, and the delay on scale down events is 5 minutes

Currently, you can't tune these cooldown values from the default.

## Cluster autoscaler

To respond to changing pod demands, Kubernetes has a cluster autoscaler, that adjusts the number of nodes based on the requested compute resources in the node pool. By default, the cluster autoscaler checks the Metrics API server every 10 seconds for any required changes in node count. If the cluster autoscale determines that a change is required, the number of nodes in your AKS cluster is increased or decreased accordingly. The cluster autoscaler works with RBAC-enabled AKS clusters that run Kubernetes 1.10.x or higher.



Cluster autoscaler is typically used alongside the horizontal pod autoscaler. When combined, the horizontal pod autoscaler increases or decreases the number of pods based on application demand, and the cluster autoscaler adjusts the number of nodes as needed to run those additional pods accordingly.

To get started with the cluster autoscaler in AKS, see [Cluster Autoscaler on AKS](#).

### Scale up events

If a node doesn't have sufficient compute resources to run a requested pod, that pod can't progress through the scheduling process. The pod can't start unless additional compute resources are available within the node pool.

When the cluster autoscaler notices pods that can't be scheduled because of node pool resource constraints, the number of nodes within the node pool is increased to provide the additional compute resources. When those additional nodes are successfully deployed and available for use within the node pool, the pods are then scheduled to run on them.

If your application needs to scale rapidly, some pods may remain in a state waiting to be scheduled until the additional nodes deployed by the cluster autoscaler can accept the scheduled pods. For applications that have high burst demands, you can scale with virtual nodes and Azure Container Instances.

### Scale down events

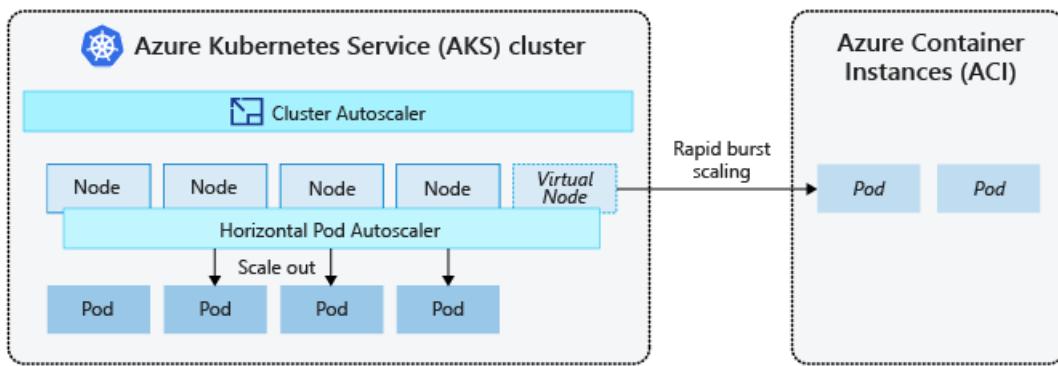
The cluster autoscaler also monitors the pod scheduling status for nodes that haven't recently received new scheduling requests. This scenario indicates the node pool has more compute resources than are required, and the number of nodes can be decreased.

A node that passes a threshold for no longer being needed for 10 minutes by default is scheduled for deletion. When this situation occurs, pods are scheduled to run on other nodes within the node pool, and the cluster autoscaler decreases the number of nodes.

Your applications may experience some disruption as pods are scheduled on different nodes when the cluster autoscaler decreases the number of nodes. To minimize disruption, avoid applications that use a single pod instance.

## Burst to Azure Container Instances

To rapidly scale your AKS cluster, you can integrate with Azure Container Instances (ACI). Kubernetes has built-in components to scale the replica and node count. However, if your application needs to rapidly scale, the horizontal pod autoscaler may schedule more pods than can be provided by the existing compute resources in the node pool. If configured, this scenario would then trigger the cluster autoscaler to deploy additional nodes in the node pool, but it may take a few minutes for those nodes to successfully provision and allow the Kubernetes scheduler to run pods on them.



ACI lets you quickly deploy container instances without additional infrastructure overhead. When you connect with AKS, ACI becomes a secured, logical extension of your AKS cluster. The [virtual nodes](#) component, which is based on [Virtual Kubelet](#), is installed in your AKS cluster that presents ACI as a virtual Kubernetes node. Kubernetes can then schedule pods that run as ACI instances through virtual nodes, not as pods on VM nodes directly in your AKS cluster. Virtual nodes are currently in preview in AKS.

Your application requires no modification to use virtual nodes. Deployments can scale across AKS and ACI and with no delay as cluster autoscaler deploys new nodes in your AKS cluster.

Virtual nodes are deployed to an additional subnet in the same virtual network as your AKS cluster. This virtual network configuration allows the traffic between ACI and AKS to be secured. Like an AKS cluster, an ACI instance is a secure, logical compute resource that is isolated from other users.

## Next steps

To get started with scaling applications, first follow the [quickstart to create an AKS cluster with the Azure CLI](#). You can then start to manually or automatically scale applications in your AKS cluster:

- Manually scale [pods](#) or [nodes](#)
- Use the [horizontal pod autoscaler](#)
- Use the [cluster autoscaler](#)

For more information on core Kubernetes and AKS concepts, see the following articles:

- [Kubernetes / AKS clusters and workloads](#)
- [Kubernetes / AKS access and identity](#)
- [Kubernetes / AKS security](#)
- [Kubernetes / AKS virtual networks](#)

- Kubernetes / AKS storage

# Cluster operator and developer best practices to build and manage applications on Azure Kubernetes Service (AKS)

2/25/2020 • 2 minutes to read • [Edit Online](#)

To build and run applications successfully in Azure Kubernetes Service (AKS), there are some key considerations to understand and implement. These areas include multi-tenancy and scheduler features, cluster and pod security, or business continuity and disaster recovery. The following best practices are grouped to help cluster operators and developers understand the considerations for each of these areas, and implement the appropriate features.

These best practices and conceptual articles have been written in conjunction with the AKS product group, engineering teams, and field teams including global black belts (GBBs).

## Cluster operator best practices

As a cluster operator, work together with application owners and developers to understand their needs. You can then use the following best practices to configure your AKS clusters as needed.

### Multi-tenancy

- [Best practices for cluster isolation](#)
  - Includes multi-tenancy core components and logical isolation with namespaces.
- [Best practices for basic scheduler features](#)
  - Includes using resource quotas and pod disruption budgets.
- [Best practices for advanced scheduler features](#)
  - Includes using taints and tolerations, node selectors and affinity, and inter-pod affinity and anti-affinity.
- [Best practices for authentication and authorization](#)
  - Includes integration with Azure Active Directory, using role-based access controls (RBAC), and pod identities.

### Security

- [Best practices for cluster security and upgrades](#)
  - Includes securing access to the API server, limiting container access, and managing upgrades and node reboots.
- [Best practices for container image management and security](#)
  - Includes securing the image and runtimes and automated builds on base image updates.
- [Best practices for pod security](#)
  - Includes securing access to resources, limiting credential exposure, and using pod identities and digital key vaults.

### Network and storage

- [Best practices for network connectivity](#)
  - Includes different network models, using ingress and web application firewalls (WAF), and securing node SSH access.
- [Best practices for storage and backups](#)
  - Includes choosing the appropriate storage type and node size, dynamically provisioning volumes, and

data backups.

## Running enterprise-ready workloads

- [Best practices for business continuity and disaster recovery](#)
  - Includes using region pairs, multiple clusters with Azure Traffic Manager, and geo-replication of container images.

## Developer best practices

As a developer or application owner, you can simplify your development experience and define requirements for application performance needs.

- [Best practices for application developers to manage resources](#)
  - Includes defining pod resource requests and limits, configuring development tools, and checking for application issues.
- [Best practices for pod security](#)
  - Includes securing access to resources, limiting credential exposure, and using pod identities and digital key vaults.

## Kubernetes / AKS concepts

To help understand some of the features and components of these best practices, you can also see the following conceptual articles for clusters in Azure Kubernetes Service (AKS):

- [Kubernetes core concepts](#)
- [Access and identity](#)
- [Security concepts](#)
- [Network concepts](#)
- [Storage options](#)
- [Scaling options](#)

## Next steps

If you need to get started with AKS, follow one of the quickstarts to deploy an Azure Kubernetes Service (AKS) cluster using the [Azure CLI](#) or [Azure portal](#).

# Best practices for cluster isolation in Azure Kubernetes Service (AKS)

2/25/2020 • 3 minutes to read • [Edit Online](#)

As you manage clusters in Azure Kubernetes Service (AKS), you often need to isolate teams and workloads. AKS provides flexibility in how you can run multi-tenant clusters and isolate resources. To maximize your investment in Kubernetes, these multi-tenancy and isolation features should be understood and implemented.

This best practices article focuses on isolation for cluster operators. In this article, you learn how to:

- Plan for multi-tenant clusters and separation of resources
- Use logical or physical isolation in your AKS clusters

## Design clusters for multi-tenancy

Kubernetes provides features that let you logically isolate teams and workloads in the same cluster. The goal should be to provide the least number of privileges, scoped to the resources each team needs. A [Namespace](#) in Kubernetes creates a logical isolation boundary. Additional Kubernetes features and considerations for isolation and multi-tenancy include the following areas:

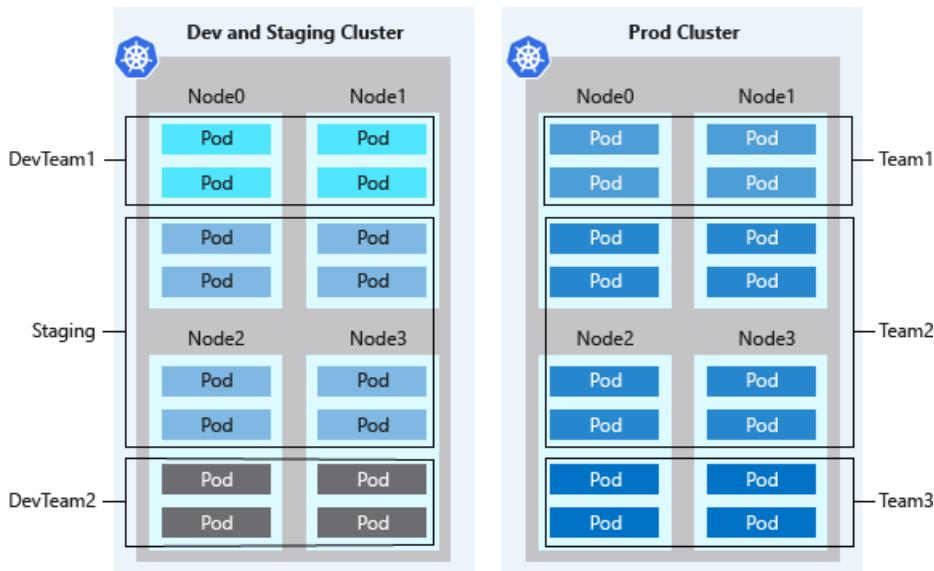
- **Scheduling** includes the use of basic features such as resource quotas and pod disruption budgets. For more information about these features, see [Best practices for basic scheduler features in AKS](#).
  - More advanced scheduler features include taints and tolerations, node selectors, and node and pod affinity or anti-affinity. For more information about these features, see [Best practices for advanced scheduler features in AKS](#).
- **Networking** includes the use of network policies to control the flow of traffic in and out of pods.
- **Authentication and authorization** include the use of role-based access control (RBAC) and Azure Active Directory (AD) integration, pod identities, and secrets in Azure Key Vault. For more information about these features, see [Best practices for authentication and authorization in AKS](#).
- **Containers** include pod security policies, pod security contexts, scanning images and runtimes for vulnerabilities. Also involves using App Armor or Seccomp (Secure Computing) to restrict container access to the underlying node.

## Logically isolate clusters

**Best practice guidance** - Use logical isolation to separate teams and projects. Try to minimize the number of physical AKS clusters you deploy to isolate teams or applications.

With logical isolation, a single AKS cluster can be used for multiple workloads, teams, or environments.

Kubernetes [Namespaces](#) form the logical isolation boundary for workloads and resources.



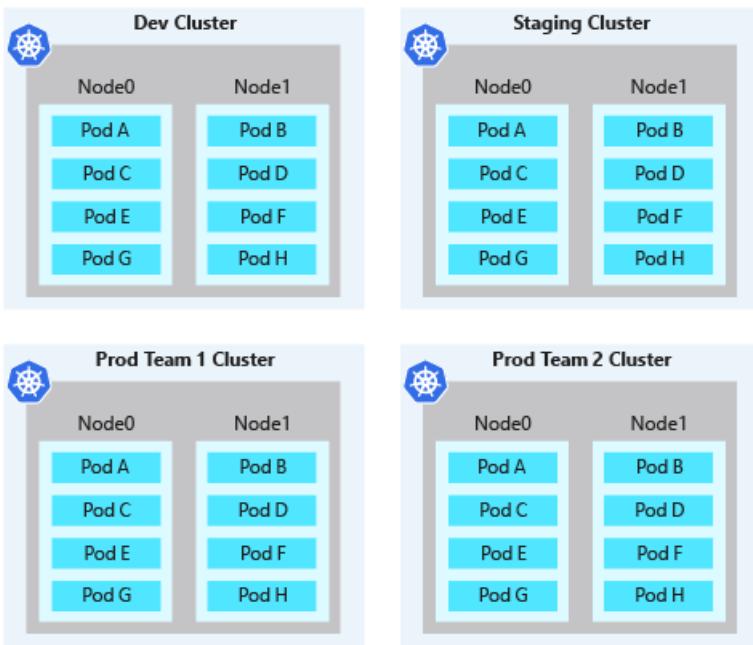
Logical separation of clusters usually provides a higher pod density than physically isolated clusters. There's less excess compute capacity that sits idle in the cluster. When combined with the Kubernetes cluster autoscaler, you can scale the number of nodes up or down to meet demands. This best practice approach to autoscaling lets you run only the number of nodes required and minimizes costs.

Kubernetes environments, in AKS or elsewhere, aren't completely safe for hostile multi-tenant usage. In a multi-tenant environment multiple tenants are working on a common, shared infrastructure. As a result if all tenants cannot be trusted, you need to do additional planning to avoid one tenant impacting the security and service of another. Additional security features such as *Pod Security Policy* and more fine-grained role-based access controls (RBAC) for nodes make exploits more difficult. However, for true security when running hostile multi-tenant workloads, a hypervisor is the only level of security that you should trust. The security domain for Kubernetes becomes the entire cluster, not an individual node. For these types of hostile multi-tenant workloads, you should use physically isolated clusters.

## Physically isolate clusters

**Best practice guidance** - Minimize the use of physical isolation for each separate team or application deployment. Instead, use *logical* isolation, as discussed in the previous section.

A common approach to cluster isolation is to use physically separate AKS clusters. In this isolation model, teams or workloads are assigned their own AKS cluster. This approach often looks like the easiest way to isolate workloads or teams, but adds additional management and financial overhead. You now have to maintain these multiple clusters, and have to individually provide access and assign permissions. You're also billed for all the individual nodes.



Physically separate clusters usually have a low pod density. As each team or workload has their own AKS cluster, the cluster is often over-provisioned with compute resources. Often, a small number of pods are scheduled on those nodes. Unused capacity on the nodes can't be used for applications or services in development by other teams. These excess resources contribute to the additional costs in physically separate clusters.

## Next steps

This article focused on cluster isolation. For more information about cluster operations in AKS, see the following best practices:

- [Basic Kubernetes scheduler features](#)
- [Advanced Kubernetes scheduler features](#)
- [Authentication and authorization](#)

# Best practices for basic scheduler features in Azure Kubernetes Service (AKS)

2/25/2020 • 5 minutes to read • [Edit Online](#)

As you manage clusters in Azure Kubernetes Service (AKS), you often need to isolate teams and workloads. The Kubernetes scheduler provides features that let you control the distribution of compute resources, or limit the impact of maintenance events.

This best practices article focuses on basic Kubernetes scheduling features for cluster operators. In this article, you learn how to:

- Use resource quotas to provide a fixed amount of resources to teams or workloads
- Limit the impact of scheduled maintenance using pod disruption budgets
- Check for missing pod resource requests and limits using the `kube-advisor` tool

## Enforce resource quotas

**Best practice guidance** - Plan and apply resource quotas at the namespace level. If pods don't define resource requests and limits, reject the deployment. Monitor resource usage and adjust quotas as needed.

Resource requests and limits are placed in the pod specification. These limits are used by the Kubernetes scheduler at deployment time to find an available node in the cluster. These limits and requests work at the individual pod level. For more information about how to define these values, see [Define pod resource requests and limits](#)

To provide a way to reserve and limit resources across a development team or project, you should use *resource quotas*. These quotas are defined on a namespace, and can be used to set quotas on the following basis:

- **Compute resources**, such as CPU and memory, or GPUs.
- **Storage resources**, includes the total number of volumes or amount of disk space for a given storage class.
- **Object count**, such as maximum number of secrets, services, or jobs can be created.

Kubernetes doesn't overcommit resources. Once the cumulative total of resource requests or limits passes the assigned quota, no further deployments are successful.

When you define resource quotas, all pods created in the namespace must provide limits or requests in their pod specifications. If they don't provide these values, you can reject the deployment. Instead, you can [configure default requests and limits for a namespace](#).

The following example YAML manifest named *dev-app-team-quotas.yaml* sets a hard limit of a total of 10 CPUs, 20Gi of memory, and 10 pods:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-app-team
spec:
  hard:
    cpu: "10"
    memory: 20Gi
    pods: "10"
```

This resource quota can be applied by specifying the namespace, such as `dev-apps`:

```
kubectl apply -f dev-app-team-quotas.yaml --namespace dev-apps
```

Work with your application developers and owners to understand their needs and apply the appropriate resource quotas.

For more information about available resource objects, scopes, and priorities, see [Resource quotas in Kubernetes](#).

## Plan for availability using pod disruption budgets

**Best practice guidance** - To maintain the availability of applications, define Pod Disruption Budgets (PDBs) to make sure that a minimum number of pods are available in the cluster.

There are two disruptive events that cause pods to be removed:

- *Involuntary disruptions* are events beyond the typical control of the cluster operator or application owner.
  - These involuntary disruptions include a hardware failure on the physical machine, a kernel panic, or the deletion of a node VM
- *Voluntary disruptions* are events requested by the cluster operator or application owner.
  - These voluntary disruptions include cluster upgrades, an updated deployment template, or accidentally deleting a pod.

The involuntary disruptions can be mitigated by using multiple replicas of your pods in a deployment. Running multiple nodes in the AKS cluster also helps with these involuntary disruptions. For voluntary disruptions, Kubernetes provides *pod disruption budgets* that let the cluster operator define a minimum available or maximum unavailable resource count. These pod disruption budgets let you plan for how deployments or replica sets respond when a voluntary disruption event occurs.

If a cluster is to be upgraded or a deployment template updated, the Kubernetes scheduler makes sure additional pods are scheduled on other nodes before the voluntary disruption events can continue. The scheduler waits before a node is rebooted until the defined number of pods are successfully scheduled on other nodes in the cluster.

Let's look at an example of a replica set with five pods that run NGINX. The pods in the replica set are assigned the label `app: nginx-frontend`. During a voluntary disruption event, such as a cluster upgrade, you want to make sure at least three pods continue to run. The following YAML manifest for a *PodDisruptionBudget* object defines these requirements:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: nginx-pdb
spec:
  minAvailable: 3
  selector:
    matchLabels:
      app: nginx-frontend
```

You can also define a percentage, such as `60%`, which allows you to automatically compensate for the replica set scaling up the number of pods.

You can define a maximum number of unavailable instances in a replica set. Again, a percentage for the maximum unavailable pods can also be defined. The following pod disruption budget YAML manifest defines that no more than two pods in the replica set be unavailable:

```
apiVersion: policy/v1beta1
kind: PodDisruptionBudget
metadata:
  name: nginx-pdb
spec:
  maxUnavailable: 2
  selector:
    matchLabels:
      app: nginx-frontend
```

Once your pod disruption budget is defined, you create it in your AKS cluster as with any other Kubernetes object:

```
kubectl apply -f nginx-pdb.yaml
```

Work with your application developers and owners to understand their needs and apply the appropriate pod disruption budgets.

For more information about using pod disruption budgets, see [Specify a disruption budget for your application](#).

## Regularly check for cluster issues with kube-advisor

**Best practice guidance** - Regularly run the latest version of `kube-advisor` open source tool to detect issues in your cluster. If you apply resource quotas on an existing AKS cluster, run `kube-advisor` first to find pods that don't have resource requests and limits defined.

The `kube-advisor` tool is an associated AKS open source project that scans a Kubernetes cluster and reports on issues that it finds. One useful check is to identify pods that don't have resource requests and limits in place.

The `kube-advisor` tool can report on resource request and limits missing in PodSpecs for Windows applications as well as Linux applications, but the `kube-advisor` tool itself must be scheduled on a Linux pod. You can schedule a pod to run on a node pool with a specific OS using a `node selector` in the pod's configuration.

In an AKS cluster that hosts multiple development teams and applications, it can be hard to track pods without these resource requests and limits set. As a best practice, regularly run `kube-advisor` on your AKS clusters, especially if you don't assign resource quotas to namespaces.

## Next steps

This article focused on basic Kubernetes scheduler features. For more information about cluster operations in AKS, see the following best practices:

- [Multi-tenancy and cluster isolation](#)
- [Advanced Kubernetes scheduler features](#)
- [Authentication and authorization](#)

# Best practices for advanced scheduler features in Azure Kubernetes Service (AKS)

2/25/2020 • 7 minutes to read • [Edit Online](#)

As you manage clusters in Azure Kubernetes Service (AKS), you often need to isolate teams and workloads. The Kubernetes scheduler provides advanced features that let you control which pods can be scheduled on certain nodes, or how multi-pod applications can be appropriately distributed across the cluster.

This best practices article focuses on advanced Kubernetes scheduling features for cluster operators. In this article, you learn how to:

- Use taints and tolerations to limit what pods can be scheduled on nodes
- Give preference to pods to run on certain nodes with node selectors or node affinity
- Split apart or group together pods with inter-pod affinity or anti-affinity

## Provide dedicated nodes using taints and tolerations

**Best practice guidance** - Limit access for resource-intensive applications, such as ingress controllers, to specific nodes. Keep node resources available for workloads that require them, and don't allow scheduling of other workloads on the nodes.

When you create your AKS cluster, you can deploy nodes with GPU support or a large number of powerful CPUs. These nodes are often used for large data processing workloads such as machine learning (ML) or artificial intelligence (AI). As this type of hardware is typically an expensive node resource to deploy, limit the workloads that can be scheduled on these nodes. You may instead wish to dedicate some nodes in the cluster to run ingress services, and prevent other workloads.

This support for different nodes is provided by using multiple node pools. An AKS cluster provides one or more node pools.

The Kubernetes scheduler can use taints and tolerations to restrict what workloads can run on nodes.

- A **taint** is applied to a node that indicates only specific pods can be scheduled on them.
- A **toleration** is then applied to a pod that allows them to *tolerate* a node's taint.

When you deploy a pod to an AKS cluster, Kubernetes only schedules pods on nodes where a toleration is aligned with the taint. As an example, assume you have a node pool in your AKS cluster for nodes with GPU support. You define name, such as `gpu`, then a value for scheduling. If you set this value to `NoSchedule`, the Kubernetes scheduler can't schedule pods on the node if the pod doesn't define the appropriate toleration.

```
kubectl taint node aks-nodepool1 sku=gpu:NoSchedule
```

With a taint applied to nodes, you then define a toleration in the pod specification that allows scheduling on the nodes. The following example defines the `sku: gpu` and `effect: NoSchedule` to tolerate the taint applied to the node in the previous step:

```

kind: Pod
apiVersion: v1
metadata:
  name: tf-mnist
spec:
  containers:
    - name: tf-mnist
      image: microsoft/samples-tf-mnist-demo:gpu
      resources:
        requests:
          cpu: 0.5
          memory: 2Gi
        limits:
          cpu: 4.0
          memory: 16Gi
      tolerations:
        - key: "sku"
          operator: "Equal"
          value: "gpu"
          effect: "NoSchedule"

```

When this pod is deployed, such as using `kubectl apply -f gpu-toleration.yaml`, Kubernetes can successfully schedule the pod on the nodes with the taint applied. This logical isolation lets you control access to resources within a cluster.

When you apply taints, work with your application developers and owners to allow them to define the required tolerations in their deployments.

For more information about taints and tolerations, see [applying taints and tolerations](#).

For more information about how to use multiple node pools in AKS, see [Create and manage multiple node pools for a cluster in AKS](#).

### Behavior of taints and tolerations in AKS

When you upgrade a node pool in AKS, taints and tolerations follow a set pattern as they're applied to new nodes:

- **Default clusters that use virtual machine scale sets**

- Let's assume you have a two-node cluster - *node1* and *node2*. You upgrade the node pool.
- Two additional nodes are created, *node3* and *node4*, and the taints are passed on respectively.
- The original *node1* and *node2* are deleted.

- **Clusters without virtual machine scale set support**

- Again, let's assume you have a two-node cluster - *node1* and *node2*. When you upgrade, an additional node (*node3*) is created.
- The taints from *node1* are applied to *node3*, then *node1* is then deleted.
- Another new node is created (named *node1*, since the previous *node1* was deleted), and the *node2* taints are applied to the new *node1*. Then, *node2* is deleted.
- In essence *node1* becomes *node3*, and *node2* becomes *node1*.

When you scale a node pool in AKS, taints and tolerations do not carry over by design.

## Control pod scheduling using node selectors and affinity

**Best practice guidance** - Control the scheduling of pods on nodes using node selectors, node affinity, or inter-pod affinity. These settings allow the Kubernetes scheduler to logically isolate workloads, such as by hardware in the node.

Taints and tolerations are used to logically isolate resources with a hard cut-off - if the pod doesn't tolerate a node's taint, it isn't scheduled on the node. An alternate approach is to use node selectors. You label nodes, such as to indicate locally attached SSD storage or a large amount of memory, and then define in the pod specification a node selector. Kubernetes then schedules those pods on a matching node. Unlike tolerations, pods without a matching node selector can be scheduled on labeled nodes. This behavior allows unused resources on the nodes to consume, but gives priority to pods that define the matching node selector.

Let's look at an example of nodes with a high amount of memory. These nodes can give preference to pods that request a high amount of memory. To make sure that the resources don't sit idle, they also allow other pods to run.

```
kubectl label node aks-nodepool1 hardware:highmem
```

A pod specification then adds the `nodeSelector` property to define a node selector that matches the label set on a node:

```
kind: Pod
apiVersion: v1
metadata:
  name: tf-mnist
spec:
  containers:
    - name: tf-mnist
      image: microsoft/samples-tf-mnist-demo:gpu
      resources:
        requests:
          cpu: 0.5
          memory: 2Gi
        limits:
          cpu: 4.0
          memory: 16Gi
      nodeSelector:
        hardware: highmem
```

When you use these scheduler options, work with your application developers and owners to allow them to correctly define their pod specifications.

For more information about using node selectors, see [Assigning Pods to Nodes](#).

## Node affinity

A node selector is a basic way to assign pods to a given node. More flexibility is available using *node affinity*. With node affinity, you define what happens if the pod can't be matched with a node. You can *require* that Kubernetes scheduler matches a pod with a labeled host. Or, you can *prefer* a match but allow the pod to be scheduled on a different host if no match is available.

The following example sets the node affinity to *requiredDuringSchedulingIgnoredDuringExecution*. This affinity requires the Kubernetes schedule to use a node with a matching label. If no node is available, the pod has to wait for scheduling to continue. To allow the pod to be scheduled on a different node, you can instead set the value to *preferredDuringScheduledIgnoreDuringExecution*:

```

kind: Pod
apiVersion: v1
metadata:
  name: tf-mnist
spec:
  containers:
    - name: tf-mnist
      image: microsoft/samples-tf-mnist-demo:gpu
      resources:
        requests:
          cpu: 0.5
          memory: 2Gi
        limits:
          cpu: 4.0
          memory: 16Gi
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: hardware
                    operator: In
                    values: highmem

```

The *IgnoredDuringExecution* part of the setting indicates that if the node labels change, the pod shouldn't be evicted from the node. The Kubernetes scheduler only uses the updated node labels for new pods being scheduled, not pods already scheduled on the nodes.

For more information, see [Affinity and anti-affinity](#).

### Inter-pod affinity and anti-affinity

One final approach for the Kubernetes scheduler to logically isolate workloads is using inter-pod affinity or anti-affinity. The settings define that pods *shouldn't* be scheduled on a node that has an existing matching pod, or that they *should* be scheduled. By default, the Kubernetes scheduler tries to schedule multiple pods in a replica set across nodes. You can define more specific rules around this behavior.

A good example is a web application that also uses an Azure Cache for Redis. You can use pod anti-affinity rules to request that the Kubernetes scheduler distributes replicas across nodes. You can then use affinity rules to make sure that each web app component is scheduled on the same host as a corresponding cache. The distribution of pods across nodes looks like the following example:

NODE 1	NODE 2	NODE 3
webapp-1	webapp-2	webapp-3
cache-1	cache-2	cache-3

This example is a more complex deployment than the use of node selectors or node affinity. The deployment gives you control over how Kubernetes schedules pods on nodes and can logically isolate resources. For a complete example of this web application with Azure Cache for Redis example, see [Co-locate pods on the same node](#).

## Next steps

This article focused on advanced Kubernetes scheduler features. For more information about cluster operations in AKS, see the following best practices:

- [Multi-tenancy and cluster isolation](#)

- Basic Kubernetes scheduler features
- Authentication and authorization

# Best practices for authentication and authorization in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

As you deploy and maintain clusters in Azure Kubernetes Service (AKS), you need to implement ways to manage access to resources and services. Without these controls, accounts may have access to resources and services they don't need. It can also be hard to track which set of credentials were used to make changes.

This best practices article focuses on how a cluster operator can manage access and identity for AKS clusters. In this article, you learn how to:

- Authenticate AKS cluster users with Azure Active Directory
- Control access to resources with role-based access controls (RBAC)
- Use a managed identity to authenticate themselves with other services

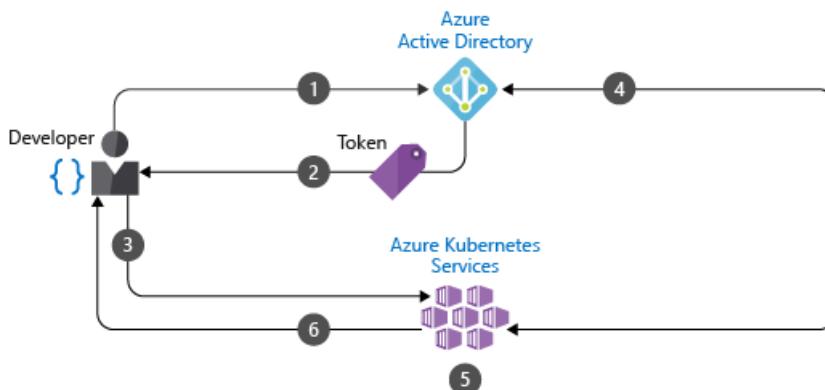
## Use Azure Active Directory

**Best practice guidance** - Deploy AKS clusters with Azure AD integration. Using Azure AD centralizes the identity management component. Any change in user account or group status is automatically updated in access to the AKS cluster. Use Roles or ClusterRoles and Bindings, as discussed in the next section, to scope users or groups to least amount of permissions needed.

The developers and application owners of your Kubernetes cluster need access to different resources. Kubernetes doesn't provide an identity management solution to control which users can interact with what resources.

Instead, you typically integrate your cluster with an existing identity solution. Azure Active Directory (AD) provides an enterprise-ready identity management solution, and can integrate with AKS clusters.

With Azure AD-integrated clusters in AKS, you create *Roles* or *ClusterRoles* that define access permissions to resources. You then *bind* the roles to users or groups from Azure AD. These Kubernetes role-based access controls (RBAC) are discussed in the next section. The integration of Azure AD and how you control access to resources can be seen in the following diagram:



1. Developer authenticates with Azure AD.
2. The Azure AD token issuance endpoint issues the access token.
3. The developer performs an action using the Azure AD token, such as `kubectl create pod`
4. Kubernetes validates the token with Azure Active Directory and fetches the developer's group memberships.
5. Kubernetes role-based access control (RBAC) and cluster policies are applied.
6. Developer's request is successful or not based on previous validation of Azure AD group membership and Kubernetes RBAC and policies.

To create an AKS cluster that uses Azure AD, see [Integrate Azure Active Directory with AKS](#).

## Use role-based access controls (RBAC)

**Best practice guidance** - Use Kubernetes RBAC to define the permissions that users or groups have to resources in the cluster. Create roles and bindings that assign the least amount of permissions required. Integrate with Azure AD so any change in user status or group membership is automatically updated and access to cluster resources is current.

In Kubernetes, you can provide granular control of access to resources in the cluster. Permissions can be defined at the cluster level, or to specific namespaces. You can define what resources can be managed, and with what permissions. These roles are then applied to users or groups with a binding. For more information about *Roles*, *ClusterRoles*, and *Bindings*, see [Access and identity options for Azure Kubernetes Service \(AKS\)](#).

As an example, you can create a Role that grants full access to resources in the namespace named *finance-app*, as shown in the following example YAML manifest:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: finance-app-full-access-role
  namespace: finance-app
rules:
- apiGroups: [""]
  resources: ["*"]
  verbs: ["*"]
```

A RoleBinding is then created that binds the Azure AD user *developer1@contoso.com* to the RoleBinding, as shown in the following YAML manifest:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: finance-app-full-access-role-binding
  namespace: finance-app
subjects:
- kind: User
  name: developer1@contoso.com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: finance-app-full-access-role
  apiGroup: rbac.authorization.k8s.io
```

When *developer1@contoso.com* is authenticated against the AKS cluster, they have full permissions to resources in the *finance-app* namespace. In this way, you logically separate and control access to resources. Kubernetes RBAC should be used in conjunction with Azure AD-integration, as discussed in the previous section.

To see how to use Azure AD groups to control access to Kubernetes resources using RBAC, see [Control access to cluster resources using role-based access controls and Azure Active Directory identities in AKS](#).

## Use pod identities

**Best practice guidance** - Don't use fixed credentials within pods or container images, as they are at risk of exposure or abuse. Instead, use pod identities to automatically request access using a central Azure AD identity solution. Pod identities is intended for use with Linux pods and container images only.

When pods need access to other Azure services, such as Cosmos DB, Key Vault, or Blob Storage, the pod needs

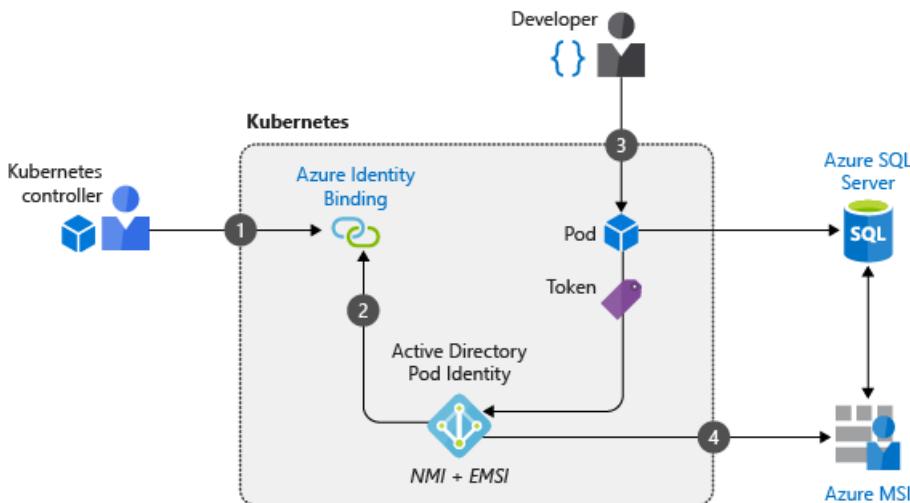
access credentials. These access credentials could be defined with the container image or injected as a Kubernetes secret, but need to be manually created and assigned. Often, the credentials are reused across pods, and aren't regularly rotated.

Managed identities for Azure resources (currently implemented as an associated AKS open source project) let you automatically request access to services through Azure AD. You don't manually define credentials for pods, instead they request an access token in real time, and can use it to access only their assigned services. In AKS, two components are deployed by the cluster operator to allow pods to use managed identities:

- **The Node Management Identity (NMI) server** is a pod that runs as a DaemonSet on each node in the AKS cluster. The NMI server listens for pod requests to Azure services.
- **The Managed Identity Controller (MIC)** is a central pod with permissions to query the Kubernetes API server and checks for an Azure identity mapping that corresponds to a pod.

When pods request access to an Azure service, network rules redirect the traffic to the Node Management Identity (NMI) server. The NMI server identifies pods that request access to Azure services based on their remote address, and queries the Managed Identity Controller (MIC). The MIC checks for Azure identity mappings in the AKS cluster, and the NMI server then requests an access token from Azure Active Directory (AD) based on the pod's identity mapping. Azure AD provides access to the NMI server, which is returned to the pod. This access token can be used by the pod to then request access to services in Azure.

In the following example, a developer creates a pod that uses a managed identity to request access to an Azure SQL Server instance:



1. Cluster operator first creates a service account that can be used to map identities when pods request access to services.
2. The NMI server and MIC are deployed to relay any pod requests for access tokens to Azure AD.
3. A developer deploys a pod with a managed identity that requests an access token through the NMI server.
4. The token is returned to the pod and used to access an Azure SQL Server instance.

#### NOTE

Managed pod identities is an open source project, and is not supported by Azure technical support.

To use pod identities, see [Azure Active Directory identities for Kubernetes applications](#).

## Next steps

This best practices article focused on authentication and authorization for your cluster and resources. To implement some of these best practices, see the following articles:

- [Integrate Azure Active Directory with AKS](#)
- [Use managed identities for Azure resources with AKS](#)

For more information about cluster operations in AKS, see the following best practices:

- [Multi-tenancy and cluster isolation](#)
- [Basic Kubernetes scheduler features](#)
- [Advanced Kubernetes scheduler features](#)

# Best practices for cluster security and upgrades in Azure Kubernetes Service (AKS)

2/25/2020 • 9 minutes to read • [Edit Online](#)

As you manage clusters in Azure Kubernetes Service (AKS), the security of your workloads and data is a key consideration. Especially when you run multi-tenant clusters using logical isolation, you need to secure access to resources and workloads. To minimize the risk of attack, you also need to make sure you apply the latest Kubernetes and node OS security updates.

This article focuses on how to secure your AKS cluster. You learn how to:

- Use Azure Active Directory and role-based access controls to secure API server access
- Secure container access to node resources
- Upgrade an AKS cluster to the latest Kubernetes version
- Keep nodes up to date and automatically apply security patches

You can also read the best practices for [container image management](#) and for [pod security](#).

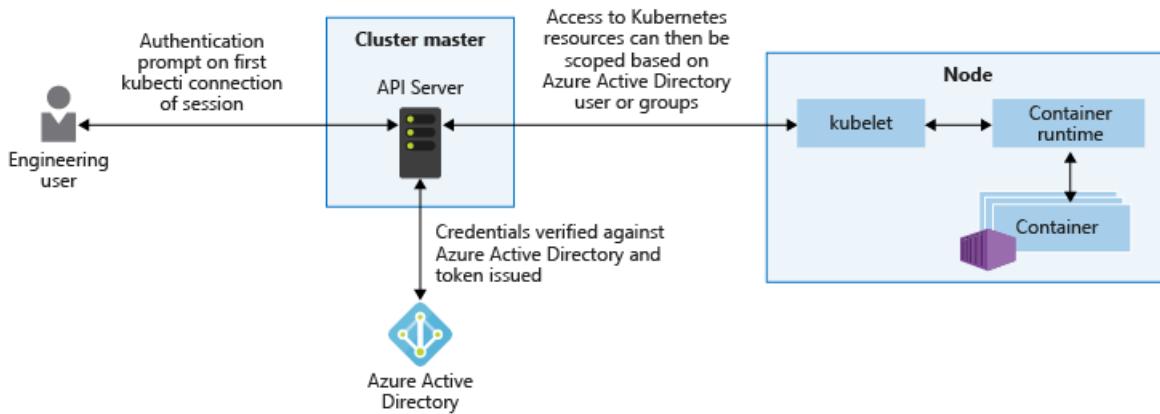
You can also use [Azure Kubernetes Services integration with Security Center](#) to help detect threats and view recommendations for securing your AKS clusters.

## Secure access to the API server and cluster nodes

**Best practice guidance** - Securing access to the Kubernetes API-Server is one of the most important things you can do to secure your cluster. Integrate Kubernetes role-based access control (RBAC) with Azure Active Directory to control access to the API server. These controls let you secure AKS the same way that you secure access to your Azure subscriptions.

The Kubernetes API server provides a single connection point for requests to perform actions within a cluster. To secure and audit access to the API server, limit access and provide the least privileged access permissions required. This approach isn't unique to Kubernetes, but is especially important when the AKS cluster is logically isolated for multi-tenant use.

Azure Active Directory (AD) provides an enterprise-ready identity management solution that integrates with AKS clusters. As Kubernetes doesn't provide an identity management solution, it can otherwise be hard to provide a granular way to restrict access to the API server. With Azure AD-integrated clusters in AKS, you use your existing user and group accounts to authenticate users to the API server.



Use Kubernetes RBAC and Azure AD-integration to secure the API server and provide the least number of

permissions required to a scoped set of resources, such as a single namespace. Different users or groups in Azure AD can be granted different RBAC roles. These granular permissions let you restrict access to the API server, and provide a clear audit trail of actions performed.

The recommended best practice is to use groups to provide access to files and folders versus individual identities, use Azure AD *group* membership to bind users to RBAC roles rather than individual *users*. As a user's group membership changes, their access permissions on the AKS cluster would change accordingly. If you bind the user directly to a role, their job function may change. The Azure AD group memberships would update, but permissions on the AKS cluster would not reflect that. In this scenario, the user ends up being granted more permissions than a user requires.

For more information about Azure AD integration and RBAC, see [Best practices for authentication and authorization in AKS](#).

## Secure container access to resources

**Best practice guidance** - Limit access to actions that containers can perform. Provide the least number of permissions, and avoid the use of root / privileged escalation.

In the same way that you should grant users or groups the least number of privileges required, containers should also be limited to only the actions and processes that they need. To minimize the risk of attack, don't configure applications and containers that require escalated privileges or root access. For example, set

`allowPrivilegeEscalation: false` in the pod manifest. These *pod security contexts* are built in to Kubernetes and let you define additional permissions such as the user or group to run as, or what Linux capabilities to expose. For more best practices, see [Secure pod access to resources](#).

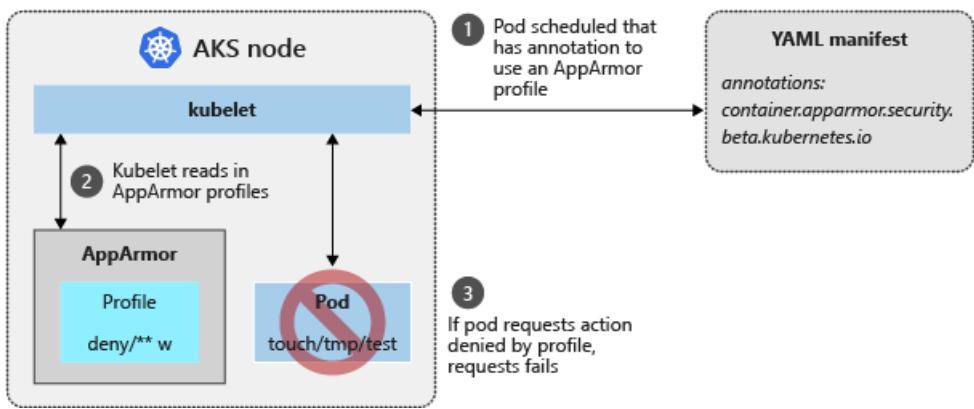
For more granular control of container actions, you can also use built-in Linux security features such as *AppArmor* and *seccomp*. These features are defined at the node level, and then implemented through a pod manifest. Built-in Linux security features are only available on Linux nodes and pods.

### NOTE

Kubernetes environments, in AKS or elsewhere, aren't completely safe for hostile multi-tenant usage. Additional security features such as *AppArmor*, *seccomp*, *Pod Security Policies*, or more fine-grained role-based access controls (RBAC) for nodes make exploits more difficult. However, for true security when running hostile multi-tenant workloads, a hypervisor is the only level of security that you should trust. The security domain for Kubernetes becomes the entire cluster, not an individual node. For these types of hostile multi-tenant workloads, you should use physically isolated clusters.

## App Armor

To limit the actions that containers can perform, you can use the [AppArmor](#) Linux kernel security module. AppArmor is available as part of the underlying AKS node OS, and is enabled by default. You create AppArmor profiles that restrict actions such as read, write, or execute, or system functions such as mounting filesystems. Default AppArmor profiles restrict access to various `/proc` and `/sys` locations, and provide a means to logically isolate containers from the underlying node. AppArmor works for any application that runs on Linux, not just Kubernetes pods.



To see AppArmor in action, the following example creates a profile that prevents writing to files. [SSH](#) to an AKS node, then create a file named `deny-write.profile` and paste the following content:

```
#include <tunables/global>
profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
    #include <abstractions/base>

    file,
    # Deny all file writes.
    deny /** w,
}
```

AppArmor profiles are added using the `apparmor_parser` command. Add the profile to AppArmor and specify the name of the profile created in the previous step:

```
sudo apparmor_parser deny-write.profile
```

There's no output returned if the profile is correctly parsed and applied to AppArmor. You're returned to the command prompt.

From your local machine, now create a pod manifest named `aks-apparmor.yaml` and paste the following content. This manifest defines an annotation for `container.apparmor.security.beta.kubernetes.io` add references the `deny-write` profile created in the previous steps:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-deny-write
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
```

Deploy the sample pod using the `kubectl apply` command:

```
kubectl apply -f aks-apparmor.yaml
```

With the pod deployed, use the `kubectl exec` command to write to a file. The command can't be executed, as shown in the following example output:

```
$ kubectl exec hello-apparmor touch /tmp/test  
  
touch: /tmp/test: Permission denied  
command terminated with exit code 1
```

For more information about AppArmor, see [AppArmor profiles in Kubernetes](#).

## Secure computing

While AppArmor works for any Linux application, [seccomp \(secure computing\)](#) works at the process level. Seccomp is also a Linux kernel security module, and is natively supported by the Docker runtime used by AKS nodes. With seccomp, the process calls that containers can perform are limited. You create filters that define what actions to allow or deny, and then use annotations within a pod YAML manifest to associate with the seccomp filter. This aligns to the best practice of only granting the container the minimal permissions that are needed to run, and no more.

To see seccomp in action, create a filter that prevents changing permissions on a file. [SSH](#) to an AKS node, then create a seccomp filter named `/var/lib/kubelet/seccomp/prevent-chmod` and paste the following content:

```
{  
  "defaultAction": "SCMP_ACT_ALLOW",  
  "syscalls": [  
    {  
      "name": "chmod",  
      "action": "SCMP_ACT_ERRNO"  
    }  
  ]  
}
```

From your local machine, now create a pod manifest named `aks-seccomp.yaml` and paste the following content. This manifest defines an annotation for `seccomp.security.alpha.kubernetes.io` and references the `prevent-chmod` filter created in the previous step:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: chmod-prevented  
  annotations:  

```

Deploy the sample pod using the [kubectl apply](#) command:

```
kubectl apply -f ./aks-seccomp.yaml
```

View the status of the pods using the [kubectl get pods](#) command. The pod reports an error. The `chmod` command is prevented from running by the seccomp filter, as shown in the following example output:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
chmod-prevented	0/1	Error	0	7s

For more information about available filters, see [Seccomp security profiles for Docker](#).

## Regularly update to the latest version of Kubernetes

**Best practice guidance** - To stay current on new features and bug fixes, regularly upgrade to the Kubernetes version in your AKS cluster.

Kubernetes releases new features at a quicker pace than more traditional infrastructure platforms. Kubernetes updates include new features, and bug or security fixes. New features typically move through an *alpha* and then *beta* status before they become *stable* and are generally available and recommended for production use. This release cycle should allow you to update Kubernetes without regularly encountering breaking changes or adjusting your deployments and templates.

AKS supports four minor versions of Kubernetes. This means that when a new minor patch version is introduced, the oldest minor version and patch releases supported are retired. Minor updates to Kubernetes happen on a periodic basis. Make sure that you have a governance process to check and upgrade as needed so you don't fall out of support. For more information, see [Supported Kubernetes versions AKS](#)

To check the versions that are available for your cluster, use the `az aks get-upgrades` command as shown in the following example:

```
az aks get-upgrades --resource-group myResourceGroup --name myAKSCluster
```

You can then upgrade your AKS cluster using the `az aks upgrade` command. The upgrade process safely cordons and drains one node at a time, schedules pods on remaining nodes, and then deploys a new node running the latest OS and Kubernetes versions.

```
az aks upgrade --resource-group myResourceGroup --name myAKSCluster --kubernetes-version KUBERNETES_VERSION
```

For more information about upgrades in AKS, see [Supported Kubernetes versions in AKS](#) and [Upgrade an AKS cluster](#).

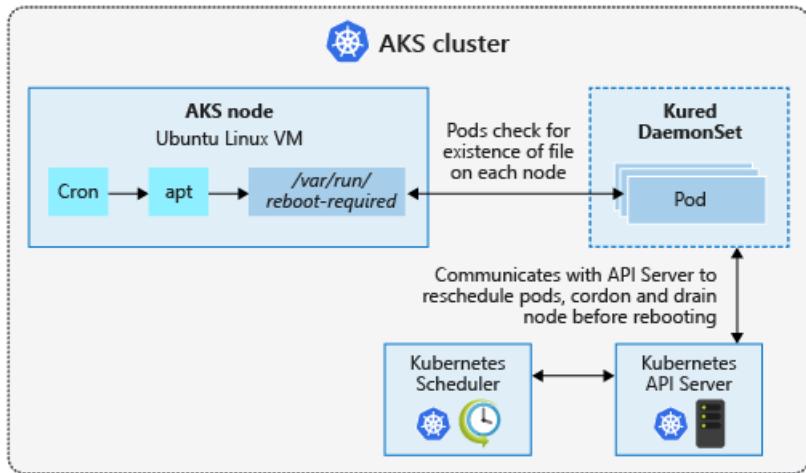
## Process Linux node updates and reboots using kured

**Best practice guidance** - AKS automatically downloads and installs security fixes on each Linux nodes, but does not automatically reboot if necessary. Use `kured` to watch for pending reboots, then safely cordon and drain the node to allow the node to reboot, apply the updates and be as secure as possible with respect to the OS. For Windows Server nodes (currently in preview in AKS), regularly perform an AKS upgrade operation to safely cordon and drain pods and deploy updated nodes.

Each evening, Linux nodes in AKS get security patches available through their distro update channel. This behavior is configured automatically as the nodes are deployed in an AKS cluster. To minimize disruption and potential impact to running workloads, nodes are not automatically rebooted if a security patch or kernel update requires it.

The open-source [kured \(KUBernetes REboot Daemon\)](#) project by Weaveworks watches for pending node reboots. When a Linux node applies updates that require a reboot, the node is safely cordoned and drained to move and schedule the pods on other nodes in the cluster. Once the node is rebooted, it is added back into the cluster and Kubernetes resumes scheduling pods on it. To minimize disruption, only one node at a time is permitted to be

rebooted by `kured`.



If you want finer grain control over when reboots happen, `kured` can integrate with Prometheus to prevent reboots if there are other maintenance events or cluster issues in progress. This integration minimizes additional complications by rebooting nodes while you are actively troubleshooting other issues.

For more information about how to handle node reboots, see [Apply security and kernel updates to nodes in AKS](#).

## Next steps

This article focused on how to secure your AKS cluster. To implement some of these areas, see the following articles:

- [Integrate Azure Active Directory with AKS](#)
- [Upgrade an AKS cluster to the latest version of Kubernetes](#)
- [Process security updates and node reboots with kured](#)

# Best practices for container image management and security in Azure Kubernetes Service (AKS)

2/25/2020 • 2 minutes to read • [Edit Online](#)

As you develop and run applications in Azure Kubernetes Service (AKS), the security of your containers and container images is a key consideration. Containers that include out of date base images or unpatched application runtimes introduce a security risk and possible attack vector. To minimize these risks, you should integrate tools that scan for and remediate issues in your containers at build time as well as runtime. The earlier in the process the vulnerability or out of date base image is caught, the more secure the cluster. In this article, *containers* means both the container images stored in a container registry, and the running containers.

This article focuses on how to secure your containers in AKS. You learn how to:

- Scan for and remediate image vulnerabilities
- Automatically trigger and redeploy container images when a base image is updated

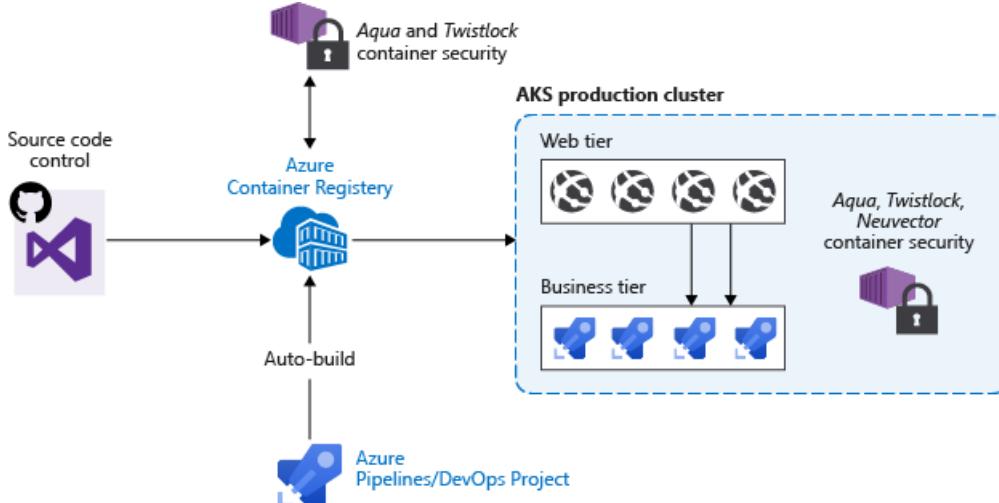
You can also read the best practices for [cluster security](#) and for [pod security](#).

You can also use [Container security in Security Center](#) to help scan your containers for vulnerabilities. There is also [Azure Container Registry integration](#) with Security Center to help protect your images and registry from vulnerabilities.

## Secure the images and run time

**Best practice guidance** - Scan your container images for vulnerabilities, and only deploy images that have passed validation. Regularly update the base images and application runtime, then redeploy workloads in the AKS cluster.

One concern with the adoption of container-based workloads is verifying the security of images and runtime used to build your own applications. How do you make sure that you don't introduce security vulnerabilities into your deployments? Your deployment workflow should include a process to scan container images using tools such as [Twistlock](#) or [Aqua](#), and then only allow verified images to be deployed.



In a real-world example, you can use a continuous integration and continuous deployment (CI/CD) pipeline to automate the image scans, verification, and deployments. Azure Container Registry includes these vulnerabilities scanning capabilities.

## Automatically build new images on base image update

**Best practice guidance** - As you use base images for application images, use automation to build new images when the base image is updated. As those base images typically include security fixes, update any downstream application container images.

Each time a base image is updated, any downstream container images should also be updated. This build process should be integrated into validation and deployment pipelines such as [Azure Pipelines](#) or Jenkins. These pipelines make sure that your applications continue to run on the updated based images. Once your application container images are validated, the AKS deployments can then be updated to run the latest, secure images.

Azure Container Registry Tasks can also automatically update container images when the base image is updated. This feature allows you to build a small number of base images, and regularly keep them updated with bug and security fixes.

For more information about base image updates, see [Automate image builds on base image update with Azure Container Registry Tasks](#).

## Next steps

This article focused on how to secure your containers. To implement some of these areas, see the following articles:

- [Automate image builds on base image update with Azure Container Registry Tasks](#)

# Best practices for network connectivity and security in Azure Kubernetes Service (AKS)

2/25/2020 • 9 minutes to read • [Edit Online](#)

As you create and manage clusters in Azure Kubernetes Service (AKS), you provide network connectivity for your nodes and applications. These network resources include IP address ranges, load balancers, and ingress controllers. To maintain a high quality of service for your applications, you need to plan for and then configure these resources.

This best practices article focuses on network connectivity and security for cluster operators. In this article, you learn how to:

- Compare the kubenet and Azure CNI network modes in AKS
- Plan for required IP addressing and connectivity
- Distribute traffic using load balancers, ingress controllers, or a web application firewall (WAF)
- Securely connect to cluster nodes

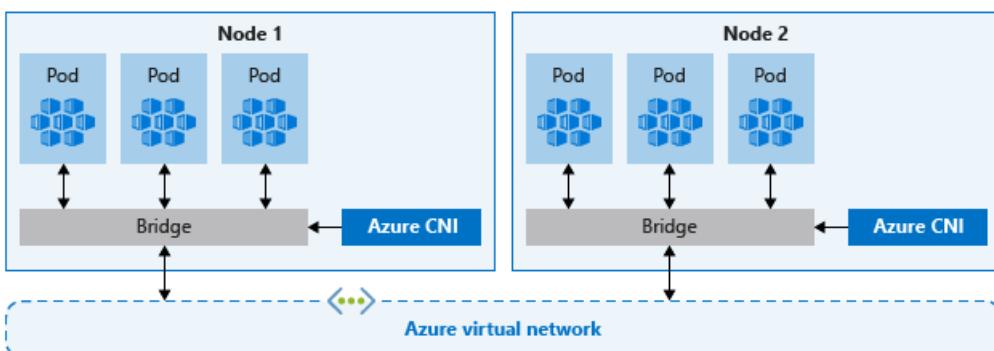
## Choose the appropriate network model

**Best practice guidance** - For integration with existing virtual networks or on-premises networks, use Azure CNI networking in AKS. This network model also allows greater separation of resources and controls in an enterprise environment.

Virtual networks provide the basic connectivity for AKS nodes and customers to access your applications. There are two different ways to deploy AKS clusters into virtual networks:

- **Kubenet networking** - Azure manages the virtual network resources as the cluster is deployed and uses the [kubenet](#) Kubernetes plugin.
- **Azure CNI networking** - Deploys into an existing virtual network, and uses the [Azure Container Networking Interface \(CNI\)](#) Kubernetes plugin. Pods receive individual IPs that can route to other network services or on-premises resources.

The Container Networking Interface (CNI) is a vendor-neutral protocol that lets the container runtime make requests to a network provider. The Azure CNI assigns IP addresses to pods and nodes, and provides IP address management (IPAM) features as you connect to existing Azure virtual networks. Each node and pod resource receives an IP address in the Azure virtual network, and no additional routing is needed to communicate with other resources or services.



For most production deployments, you should use Azure CNI networking. This network model allows for separation of control and management of resources. From a security perspective, you often want different teams

to manage and secure those resources. Azure CNI networking lets you connect to existing Azure resources, on-premises resources, or other services directly via IP addresses assigned to each pod.

When you use Azure CNI networking, the virtual network resource is in a separate resource group to the AKS cluster. Delegate permissions for the AKS service principal to access and manage these resources. The service principal used by the AKS cluster must have at least [Network Contributor](#) permissions on the subnet within your virtual network. If you wish to define a [custom role](#) instead of using the built-in Network Contributor role, the following permissions are required:

- `Microsoft.Network/virtualNetworks/subnets/join/action`
- `Microsoft.Network/virtualNetworks/subnets/read`

For more information about AKS service principal delegation, see [Delegate access to other Azure resources](#).

As each node and pod receive its own IP address, plan out the address ranges for the AKS subnets. The subnet must be large enough to provide IP addresses for every node, pods, and network resources that you deploy. Each AKS cluster must be placed in its own subnet. To allow connectivity to on-premises or peered networks in Azure, don't use IP address ranges that overlap with existing network resources. There are default limits to the number of pods that each node runs with both kubenet and Azure CNI networking. To handle scale out events or cluster upgrades, you also need additional IP addresses available for use in the assigned subnet. This additional address space is especially important if you use Windows Server containers (currently in preview in AKS), as those node pools require an upgrade to apply the latest security patches. For more information on Windows Server nodes, see [Upgrade a node pool in AKS](#).

To calculate the IP address required, see [Configure Azure CNI networking in AKS](#).

### Kubenet networking

Although kubenet doesn't require you to set up the virtual networks before the cluster is deployed, there are disadvantages:

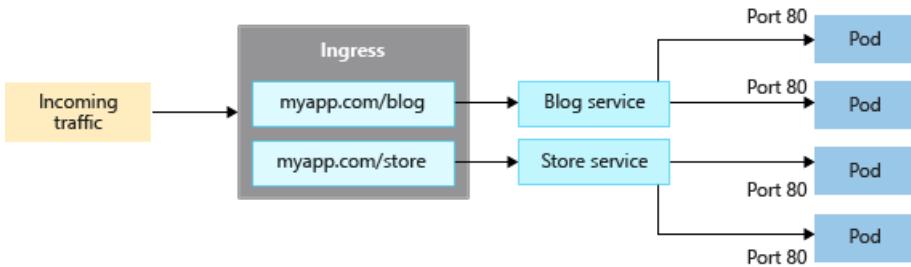
- Nodes and pods are placed on different IP subnets. User Defined Routing (UDR) and IP forwarding is used to route traffic between pods and nodes. This additional routing may reduce network performance.
- Connections to existing on-premises networks or peering to other Azure virtual networks can be complex.

Kubenet is suitable for small development or test workloads, as you don't have to create the virtual network and subnets separately from the AKS cluster. Simple websites with low traffic, or to lift and shift workloads into containers, can also benefit from the simplicity of AKS clusters deployed with kubenet networking. For most production deployments, you should plan for and use Azure CNI networking. You can also [configure your own IP address ranges and virtual networks using kubenet](#).

## Distribute ingress traffic

**Best practice guidance** - To distribute HTTP or HTTPS traffic to your applications, use ingress resources and controllers. Ingress controllers provide additional features over a regular Azure load balancer, and can be managed as native Kubernetes resources.

An Azure load balancer can distribute customer traffic to applications in your AKS cluster, but it's limited in what it understands about that traffic. A load balancer resource works at layer 4, and distributes traffic based on protocol or ports. Most web applications that use HTTP or HTTPS should use Kuberentes ingress resources and controllers, which work at layer 7. Ingress can distribute traffic based on the URL of the application and handle TLS/SSL termination. This ability also reduces the number of IP addresses you expose and map. With a load balancer, each application typically needs a public IP address assigned and mapped to the service in the AKS cluster. With an ingress resource, a single IP address can distribute traffic to multiple applications.



There are two components for ingress:

- An ingress *resource*, and
- An ingress *controller*

The ingress resource is a YAML manifest of `kind: Ingress` that defines the host, certificates, and rules to route traffic to services that run in your AKS cluster. The following example YAML manifest would distribute traffic for `myapp.com` to one of two services, `blogservice` or `storeservice`. The customer is directed to one service or the other based on the URL they access.

```

kind: Ingress
metadata:
  name: myapp-ingress
  annotations: kubernetes.io/ingress.class: "PublicIngress"
spec:
  tls:
  - hosts:
    - myapp.com
    secretName: myapp-secret
  rules:
  - host: myapp.com
    http:
      paths:
      - path: /blog
        backend:
          serviceName: blogservice
          servicePort: 80
      - path: /store
        backend:
          serviceName: storeservice
          servicePort: 80
  
```

An ingress controller is a daemon that runs on an AKS node and watches for incoming requests. Traffic is then distributed based on the rules defined in the ingress resource. The most common ingress controller is based on [NGINX](#). AKS doesn't restrict you to a specific controller, so you can use other controllers such as [Contour](#), [HAProxy](#), or [Traefik](#).

Ingress controllers must be scheduled on a Linux node. Windows Server nodes (currently in preview in AKS) shouldn't run the ingress controller. Use a node selector in your YAML manifest or Helm chart deployment to indicate that the resource should run on a Linux-based node. For more information, see [Use node selectors to control where pods are scheduled in AKS](#).

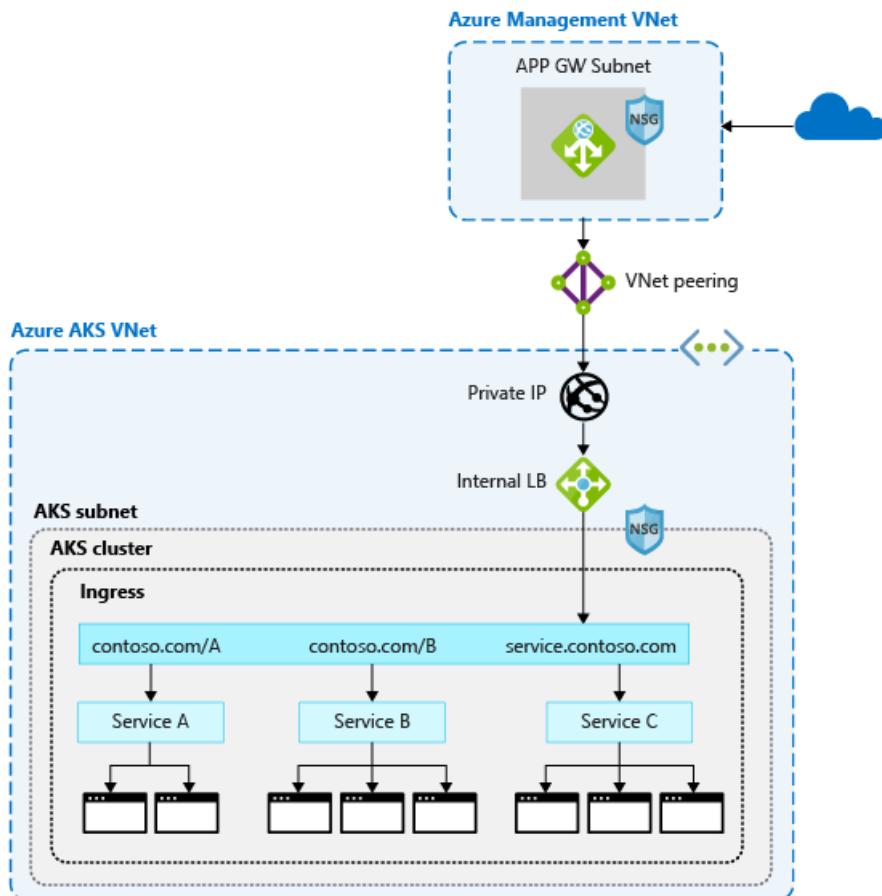
There are many scenarios for ingress, including the following how-to guides:

- [Create a basic ingress controller with external network connectivity](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller that uses your own TLS certificates](#)
- [Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates with a dynamic public IP address or with a static public IP address](#)

# Secure traffic with a web application firewall (WAF)

**Best practice guidance** - To scan incoming traffic for potential attacks, use a web application firewall (WAF) such as [Barracuda WAF for Azure](#) or Azure Application Gateway. These more advanced network resources can also route traffic beyond just HTTP and HTTPS connections or basic SSL termination.

An ingress controller that distributes traffic to services and applications is typically a Kubernetes resource in your AKS cluster. The controller runs as a daemon on an AKS node, and consumes some of the node's resources such as CPU, memory, and network bandwidth. In larger environments, you often want to offload some of this traffic routing or TLS termination to a network resource outside of the AKS cluster. You also want to scan incoming traffic for potential attacks.



A web application firewall (WAF) provides an additional layer of security by filtering the incoming traffic. The Open Web Application Security Project (OWASP) provides a set of rules to watch for attacks like cross site scripting or cookie poisoning. [Azure Application Gateway](#) (currently in preview in AKS) is a WAF that can integrate with AKS clusters to provide these security features, before the traffic reaches your AKS cluster and applications. Other third-party solutions also perform these functions, so you can continue to use existing investments or expertise in a given product.

Load balancer or ingress resources continue to run in your AKS cluster to further refine the traffic distribution. App Gateway can be centrally managed as an ingress controller with a resource definition. To get started, [create an Application Gateway Ingress controller](#).

## Control traffic flow with network policies

**Best practice guidance** - Use network policies to allow or deny traffic to pods. By default, all traffic is allowed between pods within a cluster. For improved security, define rules that limit pod communication.

Network policy is a Kubernetes feature that lets you control the traffic flow between pods. You can choose to allow or deny traffic based on settings such as assigned labels, namespace, or traffic port. The use of network policies gives a cloud-native way to control the flow of traffic. As pods are dynamically created in an AKS cluster, the

required network policies can be automatically applied. Don't use Azure network security groups to control pod-to-pod traffic, use network policies.

To use network policy, the feature must be enabled when you create an AKS cluster. You can't enable network policy on an existing AKS cluster. Plan ahead to make sure that you enable network policy on clusters and can use them as needed. Network policy should only be used for Linux-based nodes and pods in AKS.

A network policy is created as a Kubernetes resource using a YAML manifest. The policies are applied to defined pods, then ingress or egress rules define how the traffic can flow. The following example applies a network policy to pods with the `app: backend` label applied to them. The ingress rule then only allows traffic from pods with the `app: frontend` label:

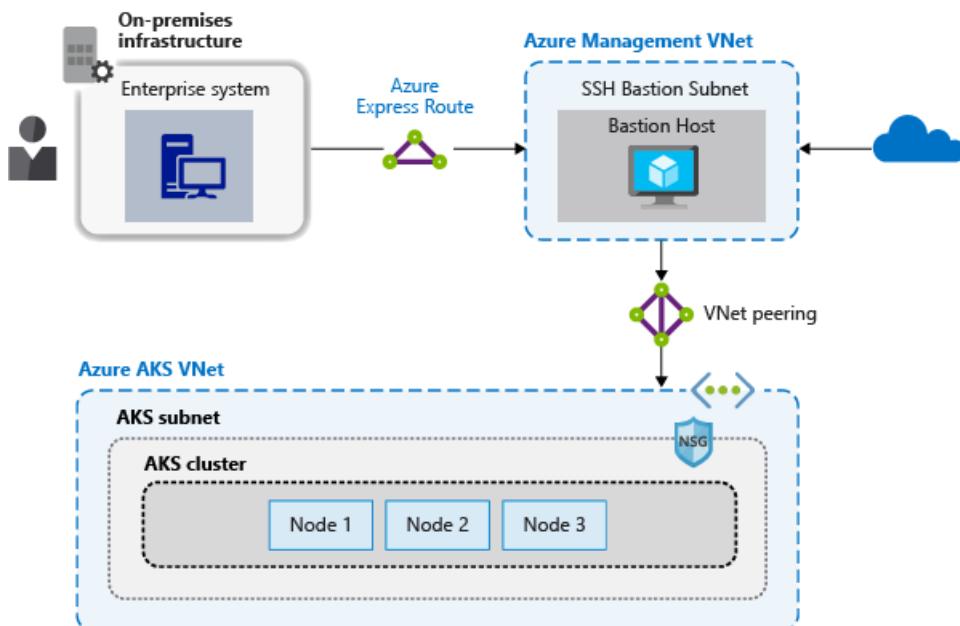
```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: backend-policy
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
```

To get started with policies, see [Secure traffic between pods using network policies in Azure Kubernetes Service \(AKS\)](#).

## Securely connect to nodes through a bastion host

**Best practice guidance** - Don't expose remote connectivity to your AKS nodes. Create a bastion host, or jump box, in a management virtual network. Use the bastion host to securely route traffic into your AKS cluster to remote management tasks.

Most operations in AKS can be completed using the Azure management tools or through the Kubernetes API server. AKS nodes aren't connected to the public internet, and are only available on a private network. To connect to nodes and perform maintenance or troubleshoot issues, route your connections through a bastion host, or jump box. This host should be in a separate management virtual network that is securely peered to the AKS cluster virtual network.



The management network for the bastion host should be secured, too. Use an [Azure ExpressRoute](#) or [VPN gateway](#) to connect to an on-premises network, and control access using network security groups.

## Next steps

This article focused on network connectivity and security. For more information about network basics in Kubernetes, see [Network concepts for applications in Azure Kubernetes Service \(AKS\)](#)

# Best practices for storage and backups in Azure Kubernetes Service (AKS)

2/26/2020 • 6 minutes to read • [Edit Online](#)

As you create and manage clusters in Azure Kubernetes Service (AKS), your applications often need storage. It's important to understand the performance needs and access methods for pods so that you can provide the appropriate storage to applications. The AKS node size may impact these storage choices. You should also plan for ways to back up and test the restore process for attached storage.

This best practices article focuses on storage considerations for cluster operators. In this article, you learn:

- What types of storage are available
- How to correctly size AKS nodes for storage performance
- Differences between dynamic and static provisioning of volumes
- Ways to back up and secure your data volumes

## Choose the appropriate storage type

**Best practice guidance** - Understand the needs of your application to pick the right storage. Use high performance, SSD-backed storage for production workloads. Plan for network-based storage when there is a need for multiple concurrent connections.

Applications often require different types and speeds of storage. Do your applications need storage that connects to individual pods, or shared across multiple pods? Is the storage for read-only access to data, or to write large amounts of structured data? These storage needs determine the most appropriate type of storage to use.

The following table outlines the available storage types and their capabilities:

USE CASE	VOLUME PLUGIN	READ/WRITE ONCE	READ-ONLY MANY	READ/WRITE MANY	WINDOWS SERVER CONTAINER SUPPORT
Shared configuration	Azure Files	Yes	Yes	Yes	Yes
Structured app data	Azure Disks	Yes	No	No	Yes
Unstructured data, file system operations	<a href="#">BlobFuse</a>	Yes	Yes	Yes	No

The two primary types of storage provided for volumes in AKS are backed by Azure Disks or Azure Files. To improve security, both types of storage use Azure Storage Service Encryption (SSE) by default that encrypts data at rest. Disks cannot currently be encrypted using Azure Disk Encryption at the AKS node level.

Azure Files are currently available in the Standard performance tier. Azure Disks are available in Standard and Premium performance tiers:

- *Premium* disks are backed by high-performance solid-state disks (SSDs). Premium disks are recommended for all production workloads.
- *Standard* disks are backed by regular spinning disks (HDDs), and are good for archival or infrequently

accessed data.

Understand the application performance needs and access patterns to choose the appropriate storage tier. For more information about Managed Disks sizes and performance tiers, see [Azure Managed Disks overview](#)

### Create and use storage classes to define application needs

The type of storage you use is defined using Kubernetes *storage classes*. The storage class is then referenced in the pod or deployment specification. These definitions work together to create the appropriate storage and connect it to pods. For more information, see [Storage classes in AKS](#).

## Size the nodes for storage needs

**Best practice guidance** - Each node size supports a maximum number of disks. Different node sizes also provide different amounts of local storage and network bandwidth. Plan for your application demands to deploy the appropriate size of nodes.

AKS nodes run as Azure VMs. Different types and sizes of VM are available. Each VM size provides a different amount of core resources such as CPU and memory. These VM sizes have a maximum number of disks that can be attached. Storage performance also varies between VM sizes for the maximum local and attached disk IOPS (input/output operations per second).

If your applications require Azure Disks as their storage solution, plan for and choose an appropriate node VM size. The amount of CPU and memory isn't the only factor when you choose a VM size. The storage capabilities are also important. For example, both the *Standard\_B2ms* and *Standard\_DS2\_v2* VM sizes include a similar amount of CPU and memory resources. Their potential storage performance is different, as shown in the following table:

NODE TYPE AND SIZE	VCPU	MEMORY (GiB)	MAX DATA DISKS	MAX UNCACHED DISK IOPS	MAX UNCACHED THROUGHPUT (Mbps)
Standard_B2ms	2	8	4	1,920	22.5
Standard_DS2_v2	2	7	8	6,400	96

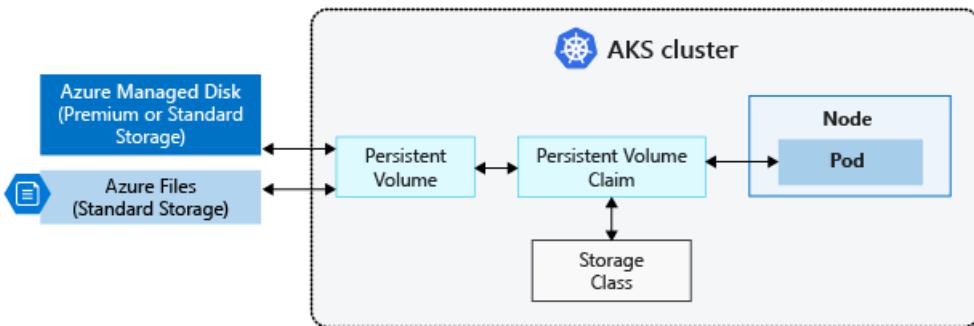
Here, the *Standard\_DS2\_v2* allows double the number of attached disks, and provides three to four times the amount of IOPS and disk throughput. If you only looked at the core compute resources and compared costs, you may choose the *Standard\_B2ms* VM size and have poor storage performance and limitations. Work with your application development team to understand their storage capacity and performance needs. Choose the appropriate VM size for the AKS nodes to meet or exceed their performance needs. Regularly baseline applications to adjust VM size as needed.

For more information about available VM sizes, see [Sizes for Linux virtual machines in Azure](#).

## Dynamically provision volumes

**Best practice guidance** - To reduce management overhead and let you scale, don't statically create and assign persistent volumes. Use dynamic provisioning. In your storage classes, define the appropriate reclaim policy to minimize unneeded storage costs once pods are deleted.

When you need to attach storage to pods, you use persistent volumes. These persistent volumes can be created manually or dynamically. Manual creation of persistent volumes adds management overhead, and limits your ability to scale. Use dynamic persistent volume provisioning to simplify storage management and allow your applications to grow and scale as needed.



A persistent volume claim (PVC) lets you dynamically create storage as needed. The underlying Azure disks are created as pods request them. In the pod definition, you request a volume to be created and attached to a designed mount path

For the concepts on how to dynamically create and use volumes, see [Persistent Volumes Claims](#).

To see these volumes in action, see how to dynamically create and use a persistent volume with [Azure Disks](#) or [Azure Files](#).

As part of your storage class definitions, set the appropriate *reclaimPolicy*. This reclaimPolicy controls the behavior of the underlying Azure storage resource when the pod is deleted and the persistent volume may no longer be required. The underlying storage resource can be deleted, or retained for use with a future pod. The reclaimPolicy can set to *retain* or *delete*. Understand your application needs, and implement regular checks for storage that is retained to minimize the amount of un-used storage that is used and billed.

For more information about storage class options, see [storage reclaim policies](#).

## Secure and back up your data

**Best practice guidance** - Back up your data using an appropriate tool for your storage type, such as Velero or Azure Site Recovery. Verify the integrity, and security, of those backups.

When your applications store and consume data persisted on disks or in files, you need to take regular backups or snapshots of that data. Azure Disks can use built-in snapshot technologies. You may need to look for your applications to flush writes to disk before you perform the snapshot operation. [Velero](#) can back up persistent volumes along with additional cluster resources and configurations. If you can't [remove state from your applications](#), back up the data from persistent volumes and regularly test the restore operations to verify data integrity and the processes required.

Understand the limitations of the different approaches to data backups and if you need to quiesce your data prior to snapshot. Data backups don't necessarily let you restore your application environment of cluster deployment. For more information about those scenarios, see [Best practices for business continuity and disaster recovery in AKS](#).

## Next steps

This article focused on storage best practices in AKS. For more information about storage basics in Kubernetes, see [Storage concepts for applications in AKS](#).

# Best practices for business continuity and disaster recovery in Azure Kubernetes Service (AKS)

2/25/2020 • 7 minutes to read • [Edit Online](#)

As you manage clusters in Azure Kubernetes Service (AKS), application uptime becomes important. AKS provides high availability by using multiple nodes in an availability set. But these multiple nodes don't protect your system from a region failure. To maximize your uptime, plan ahead to maintain business continuity and prepare for disaster recovery.

This article focuses on how to plan for business continuity and disaster recovery in AKS. You learn how to:

- Plan for AKS clusters in multiple regions.
- Route traffic across multiple clusters by using Azure Traffic Manager.
- Use geo-replication for your container image registries.
- Plan for application state across multiple clusters.
- Replicate storage across multiple regions.

## Plan for multiregion deployment

**Best practice:** When you deploy multiple AKS clusters, choose regions where AKS is available, and use paired regions.

An AKS cluster is deployed into a single region. To protect your system from region failure, deploy your application into multiple AKS clusters across different regions. When you plan where to deploy your AKS cluster, consider:

- **AKS region availability:** Choose regions close to your users. AKS continually expands into new regions.
- **Azure paired regions:** For your geographic area, choose two regions that are paired with each other. Paired regions coordinate platform updates and prioritize recovery efforts where needed.
- **Service availability:** Decide whether your paired regions should be hot/hot, hot/warm, or hot/cold. Do you want to run both regions at the same time, with one region *ready* to start serving traffic? Or do you want one region to have time to get ready to serve traffic?

AKS region availability and paired regions are a joint consideration. Deploy your AKS clusters into paired regions that are designed to manage region disaster recovery together. For example, AKS is available in East US and West US. These regions are paired. Choose these two regions when you're creating an AKS BC/DR strategy.

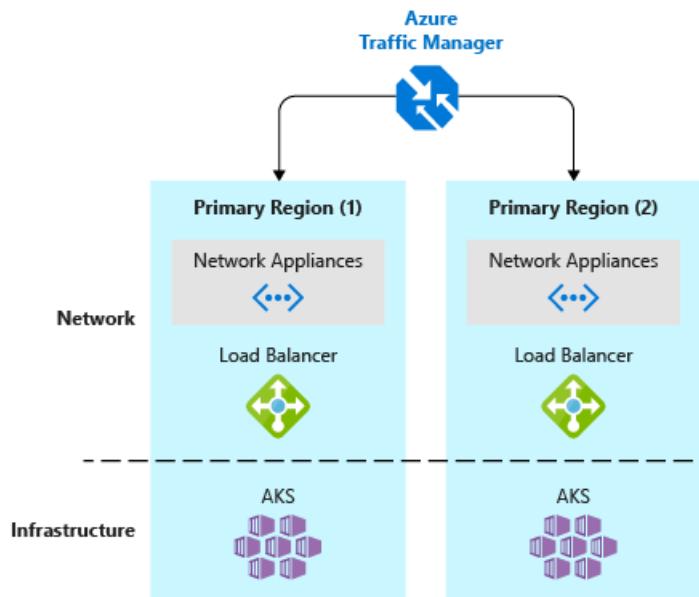
When you deploy your application, add another step to your CI/CD pipeline to deploy to these multiple AKS clusters. If you don't update your deployment pipelines, applications might be deployed into only one of your regions and AKS clusters. Customer traffic that's directed to a secondary region won't receive the latest code updates.

## Use Azure Traffic Manager to route traffic

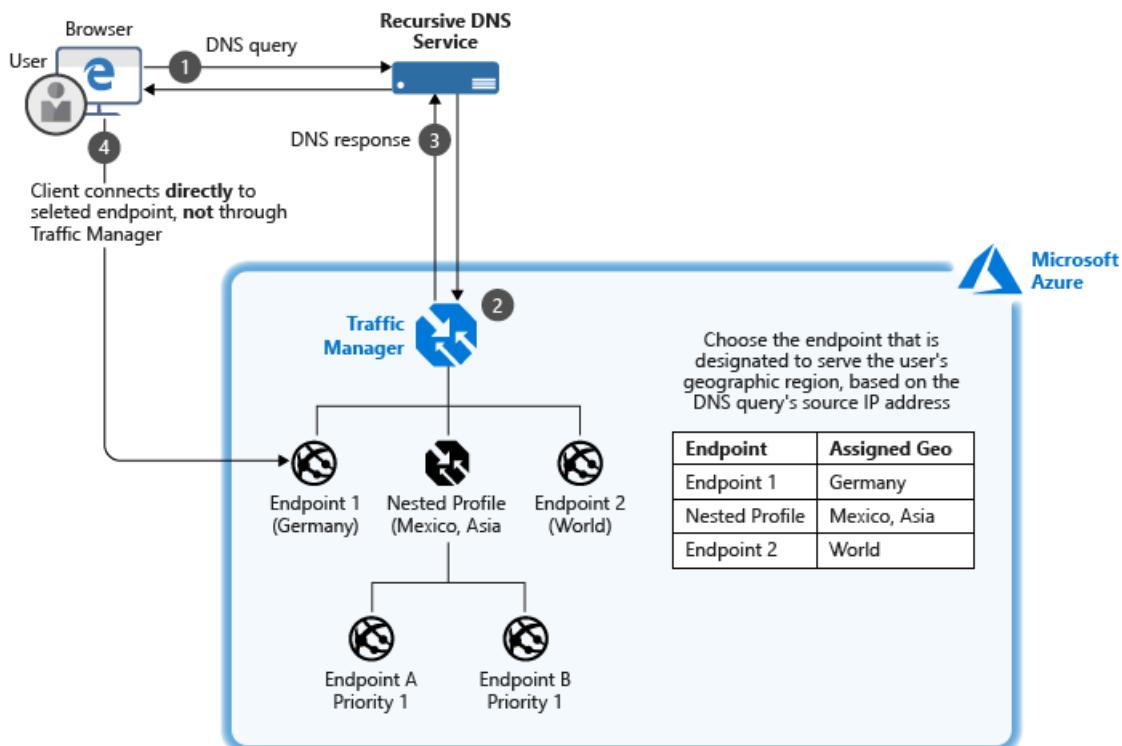
**Best practice:** Azure Traffic Manager can direct customers to their closest AKS cluster and application instance. For the best performance and redundancy, direct all application traffic through Traffic Manager before it goes to your AKS cluster.

If you have multiple AKS clusters in different regions, use Traffic Manager to control how traffic flows to the applications that run in each cluster. [Azure Traffic Manager](#) is a DNS-based traffic load balancer that can distribute

network traffic across regions. Use Traffic Manager to route users based on cluster response time or based on geography.



Customers who have a single AKS cluster typically connect to the service IP or DNS name of a given application. In a multicloud deployment, customers should connect to a Traffic Manager DNS name that points to the services on each AKS cluster. Define these services by using Traffic Manager endpoints. Each endpoint is the *service load balancer IP*. Use this configuration to direct network traffic from the Traffic Manager endpoint in one region to the endpoint in a different region.



Traffic Manager performs DNS lookups and returns a user's most appropriate endpoint. Nested profiles can prioritize a primary location. For example, users should generally connect to their closest geographic region. If that region has a problem, Traffic Manager instead directs the users to a secondary region. This approach ensures that customers can connect to an application instance even if their closest geographic region is unavailable.

For information on how to set up endpoints and routing, see [Configure the geographic traffic routing method by using Traffic Manager](#).

#### Layer 7 application routing with Azure Front Door Service

Traffic Manager uses DNS (layer 3) to shape traffic. [Azure Front Door Service](#) provides an HTTP/HTTPS (layer 7)

routing option. Additional features of Azure Front Door Service include SSL termination, custom domain, web application firewall, URL Rewrite, and session affinity. Review the needs of your application traffic to understand which solution is the most suitable.

### Interconnect regions with global virtual network peering

If the clusters need to talk to each other, connecting both virtual networks to each other can be achieved through [virtual network peering](#). This technology interconnects virtual networks to each other providing high bandwidth across Microsoft's backbone network, even across different geographic regions.

A prerequisite to peer the virtual networks where AKS clusters are running is to use the standard Load Balancer in your AKS cluster, so that Kubernetes services are reachable across the virtual network peering.

## Enable geo-replication for container images

**Best practice:** Store your container images in Azure Container Registry and geo-replicate the registry to each AKS region.

To deploy and run your applications in AKS, you need a way to store and pull the container images. Container Registry integrates with AKS, so it can securely store your container images or Helm charts. Container Registry supports multimaster geo-replication to automatically replicate your images to Azure regions around the world.

To improve performance and availability, use Container Registry geo-replication to create a registry in each region where you have an AKS cluster. Each AKS cluster then pulls container images from the local container registry in the same region:



When you use Container Registry geo-replication to pull images from the same region, the results are:

- **Faster:** You pull images from high-speed, low-latency network connections within the same Azure region.
- **More reliable:** If a region is unavailable, your AKS cluster pulls the images from an available container registry.
- **Cheaper:** There's no network egress charge between datacenters.

Geo-replication is a feature of *Premium* SKU container registries. For information on how to configure geo-replication, see [Container Registry geo-replication](#).

## Remove service state from inside containers

**Best practice:** Where possible, don't store service state inside the container. Instead, use an Azure platform as a service (PaaS) that supports multiregion replication.

*Service state* refers to the in-memory or on-disk data that a service requires to function. State includes the data structures and member variables that the service reads and writes. Depending on how the service is architected, the state might also include files or other resources that are stored on the disk. For example, the state might include the files a database uses to store data and transaction logs.

State can be either externalized or colocated with the code that manipulates the state. Typically, you externalize state by using a database or other data store that runs on different machines over the network or that runs out of process on the same machine.

Containers and microservices are most resilient when the processes that run inside them don't retain state. Because applications almost always contain some state, use a PaaS solution such as Azure Database for MySQL, Azure Database for PostgreSQL, or Azure SQL Database.

To build portable applications, see the following guidelines:

- [The 12-factor app methodology](#)
- [Run a web application in multiple Azure regions](#)

## Create a storage migration plan

**Best practice:** If you use Azure Storage, prepare and test how to migrate your storage from the primary region to the backup region.

Your applications might use Azure Storage for their data. Because your applications are spread across multiple AKS clusters in different regions, you need to keep the storage synchronized. Here are two common ways to replicate storage:

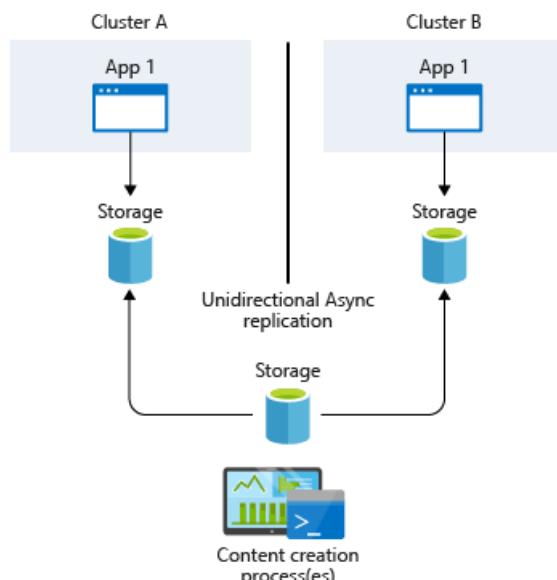
- Infrastructure-based asynchronous replication
- Application-based asynchronous replication

### Infrastructure-based asynchronous replication

Your applications might require persistent storage even after a pod is deleted. In Kubernetes, you can use persistent volumes to persist data storage. Persistent volumes are mounted to a node VM and then exposed to the pods. Persistent volumes follow pods even if the pods are moved to a different node inside the same cluster.

The replication strategy you use depends on your storage solution. Common storage solutions such as [Gluster](#), [Ceph](#), [Rook](#), and [Portworx](#) provide their own guidance about disaster recovery and replication.

The typical strategy is to provide a common storage point where applications can write their data. This data is then replicated across regions and then accessed locally.

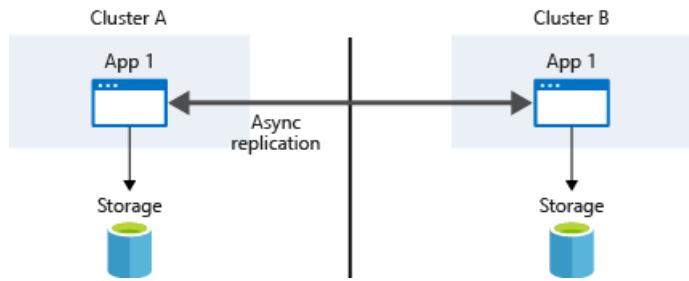


If you use Azure Managed Disks, you can choose replication and DR solutions such as these:

- [Velero on Azure](#)
- [Azure Site Recovery](#)

## Application-based asynchronous replication

Kubernetes currently provides no native implementation for application-based asynchronous replication. Because containers and Kubernetes are loosely coupled, any traditional application or language approach should work. Typically, the applications themselves replicate the storage requests, which are then written to each cluster's underlying data storage.



## Next steps

This article focuses on business continuity and disaster recovery considerations for AKS clusters. For more information about cluster operations in AKS, see these articles about best practices:

- [Multitenancy and cluster isolation](#)
- [Basic Kubernetes scheduler features](#)

# Best practices for application developers to manage resources in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

As you develop and run applications in Azure Kubernetes Service (AKS), there are a few key areas to consider. How you manage your application deployments can negatively impact the end-user experience of services that you provide. To help you succeed, keep in mind some best practices you can follow as you develop and run applications in AKS.

This best practices article focuses on how to run your cluster and workloads from an application developer perspective. For information about administrative best practices, see [Cluster operator best practices for isolation and resource management in Azure Kubernetes Service \(AKS\)](#). In this article, you learn:

- What are pod resource requests and limits
- Ways to develop and deploy applications with Dev Spaces and Visual Studio Code
- How to use the `kube-advisor` tool to check for issues with deployments

## Define pod resource requests and limits

**Best practice guidance** - Set pod requests and limits on all pods in your YAML manifests. If the AKS cluster uses *resource quotas*, your deployment may be rejected if you don't define these values.

A primary way to manage the compute resources within an AKS cluster is to use pod requests and limits. These requests and limits let the Kubernetes scheduler know what compute resources a pod should be assigned.

- **Pod CPU/Memory requests** define a set amount of CPU and memory that the pod needs on a regular basis.
  - When the Kubernetes scheduler tries to place a pod on a node, the pod requests are used to determine which node has sufficient resources available for scheduling.
  - Not setting a pod request will default it to the limit defined.
  - It is very important to monitor the performance of your application to adjust these requests. If insufficient requests are made, your application may receive degraded performance due to over scheduling a node. If requests are overestimated, your application may have increased difficulty getting scheduled.
- **Pod CPU/Memory limits** are the maximum amount of CPU and memory that a pod can use. These limits help define which pods should be killed in the event of node instability due to insufficient resources. Without proper limits set pods will be killed until resource pressure is lifted.
  - Pod limits help define when a pod has lost control of resource consumption. When a limit is exceeded, the pod is prioritized for killing to maintain node health and minimize impact to pods sharing the node.
  - Not setting a pod limit defaults it to the highest available value on a given node.
  - Don't set a pod limit higher than your nodes can support. Each AKS node reserves a set amount of CPU and memory for the core Kubernetes components. Your application may try to consume too many resources on the node for other pods to successfully run.
  - Again, it is very important to monitor the performance of your application at different times during the day or week. Determine when the peak demand is, and align the pod limits to the resources required to meet the application's max needs.

In your pod specifications, it's **best practice and very important** to define these requests and limits based on the above information. If you don't include these values, the Kubernetes scheduler cannot take into account the resources your applications require to aid in scheduling decisions.

If the scheduler places a pod on a node with insufficient resources, application performance will be degraded. It is highly recommended for cluster administrators to set *resource quotas* on a namespace that requires you to set resource requests and limits. For more information, see [resource quotas on AKS clusters](#).

When you define a CPU request or limit, the value is measured in CPU units.

- 1.0 CPU equates to one underlying virtual CPU core on the node.
- The same measurement is used for GPUs.
- You can define fractions measured in millicores. For example, *100m* is 0.1 of an underlying vCPU core.

In the following basic example for a single NGINX pod, the pod requests *100m* of CPU time, and *128Mi* of memory. The resource limits for the pod are set to *250m* CPU and *256Mi* memory:

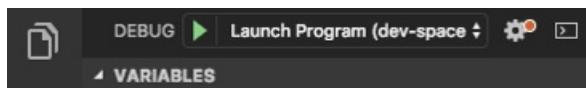
```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: nginx:1.15.5
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
```

For more information about resource measurements and assignments, see [Managing compute resources for containers](#).

## Develop and debug applications against an AKS cluster

**Best practice guidance** - Development teams should deploy and debug against an AKS cluster using Dev Spaces. This development model makes sure that role-based access controls, network, or storage needs are implemented before the app is deployed to production.

With Azure Dev Spaces, you develop, debug, and test applications directly against an AKS cluster. Developers within a team work together to build and test throughout the application lifecycle. You can continue to use existing tools such as Visual Studio or Visual Studio Code. An extension is installed for Dev Spaces that gives an option to run and debug the application in an AKS cluster:



This integrated development and test process with Dev Spaces reduces the need for local test environments, such as [minikube](#). Instead, you develop and test against an AKS cluster. This cluster can be secured and isolated as noted in previous section on the use of namespaces to logically isolate a cluster. When your apps are ready to deploy to production, you can confidently deploy as your development was all done against a real AKS cluster.

Azure Dev Spaces is intended for use with applications that run on Linux pods and nodes.

## Use the Visual Studio Code extension for Kubernetes

**Best practice guidance** - Install and use the VS Code extension for Kubernetes when you write YAML manifests. You can also use the extension for integrated deployment solution, which may help application owners that infrequently interact with the AKS cluster.

The [Visual Studio Code extension for Kubernetes](#) helps you develop and deploy applications to AKS. The extension provides intellisense for Kubernetes resources, and Helm charts and templates. You can also browse, deploy, and edit Kubernetes resources from within VS Code. The extension also provides an intellisense check for resource requests or limits being set in the pod specifications:

```
1 kind: Pod
2 apiVersion: v1
3 metadata:
4   name: mypod
5 spec:
6   containers:
7     - name: mypod
8       image: nginx:1.15.5
9       resources:
10         requests:
11           cpu: 100m
12           No memory limit specified for this container - this could starve other processes
13
14         limits:
15           cpu: 250m
```

## Regularly check for application issues with kube-advisor

**Best practice guidance** - Regularly run the latest version of [kube-advisor](#) open source tool to detect issues in your cluster. If you apply resource quotas on an existing AKS cluster, run [kube-advisor](#) first to find pods that don't have resource requests and limits defined.

The [kube-advisor](#) tool is an associated AKS open source project that scans a Kubernetes cluster and reports on issues that it finds. One useful check is to identify pods that don't have resource requests and limits in place.

The [kube-advisor](#) tool can report on resource request and limits missing in PodSpecs for Windows applications as well as Linux applications, but the [kube-advisor](#) tool itself must be scheduled on a Linux pod. You can schedule a pod to run on a node pool with a specific OS using a [node selector](#) in the pod's configuration.

In an AKS cluster that hosts many development teams and applications, it can be hard to track pods without these resource requests and limits set. As a best practice, regularly run [kube-advisor](#) on your AKS clusters.

## Next steps

This best practices article focused on how to run your cluster and workloads from a cluster operator perspective. For information about administrative best practices, see [Cluster operator best practices for isolation and resource management in Azure Kubernetes Service \(AKS\)](#).

To implement some of these best practices, see the following articles:

- [Develop with Dev Spaces](#)
- [Check for issues with kube-advisor](#)

# Best practices for pod security in Azure Kubernetes Service (AKS)

2/25/2020 • 5 minutes to read • [Edit Online](#)

As you develop and run applications in Azure Kubernetes Service (AKS), the security of your pods is a key consideration. Your applications should be designed for the principle of least number of privileges required. Keeping private data secure is top of mind for customers. You don't want credentials like database connection strings, keys, or secrets and certificates exposed to the outside world where an attacker could take advantage of those secrets for malicious purposes. Don't add them to your code or embed them in your container images. This approach would create a risk for exposure and limit the ability to rotate those credentials as the container images will need to be rebuilt.

This best practices article focuses on how to secure pods in AKS. You learn how to:

- Use pod security context to limit access to processes and services or privilege escalation
- Authenticate with other Azure resources using pod managed identities
- Request and retrieve credentials from a digital vault such as Azure Key Vault

You can also read the best practices for [cluster security](#) and for [container image management](#).

## Secure pod access to resources

**Best practice guidance** - To run as a different user or group and limit access to the underlying node processes and services, define pod security context settings. Assign the least number of privileges required.

For your applications to run correctly, pods should run as a defined user or group and not as *root*. The `securityContext` for a pod or container lets you define settings such as *runAsUser* or *fsGroup* to assume the appropriate permissions. Only assign the required user or group permissions, and don't use the security context as a means to assume additional permissions. The *runAsUser*, privilege escalation, and other Linux capabilities settings are only available on Linux nodes and pods.

When you run as a non-root user, containers cannot bind to the privileged ports under 1024. In this scenario, Kubernetes Services can be used to disguise the fact that an app is running on a particular port.

A pod security context can also define additional capabilities or permissions for accessing processes and services. The following common security context definitions can be set:

- **allowPrivilegeEscalation** defines if the pod can assume *root* privileges. Design your applications so this setting is always set to *false*.
- **Linux capabilities** let the pod access underlying node processes. Take care with assigning these capabilities. Assign the least number of privileges needed. For more information, see [Linux capabilities](#).
- **SELinux labels** is a Linux kernel security module that lets you define access policies for services, processes, and filesystem access. Again, assign the least number of privileges needed. For more information, see [SELinux options in Kubernetes](#)

The following example pod YAML manifest sets security context settings to define:

- Pod runs as user ID *1000* and part of group ID *2000*
- Can't escalate privileges to use `root`
- Allows Linux capabilities to access network interfaces and the host's real-time (hardware) clock

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  containers:
    - name: security-context-demo
      image: nginx:1.15.5
    securityContext:
      runAsUser: 1000
      fsGroup: 2000
      allowPrivilegeEscalation: false
    capabilities:
      add: ["NET_ADMIN", "SYS_TIME"]
```

Work with your cluster operator to determine what security context settings you need. Try to design your applications to minimize additional permissions and access the pod requires. There are additional security features to limit access using AppArmor and seccomp (secure computing) that can be implemented by cluster operators. For more information, see [Secure container access to resources](#).

## Limit credential exposure

**Best practice guidance** - Don't define credentials in your application code. Use managed identities for Azure resources to let your pod request access to other resources. A digital vault, such as Azure Key Vault, should also be used to store and retrieve digital keys and credentials. Pod managed identities is intended for use with Linux pods and container images only.

To limit the risk of credentials being exposed in your application code, avoid the use of fixed or shared credentials. Credentials or keys shouldn't be included directly in your code. If these credentials are exposed, the application needs to be updated and redeployed. A better approach is to give pods their own identity and way to authenticate themselves, or automatically retrieve credentials from a digital vault.

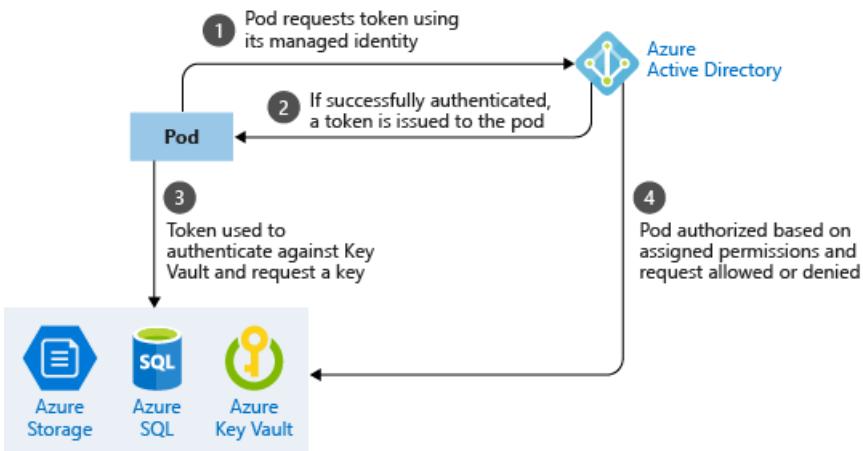
The following [associated AKS open source projects](#) let you automatically authenticate pods or request credentials and keys from a digital vault:

- Managed identities for Azure resources, and
- Azure Key Vault FlexVol driver

Associated AKS open source projects are not supported by Azure technical support. They are provided to gather feedback and bugs from our community. These projects are not recommended for production use.

### Use pod managed identities

A managed identity for Azure resources lets a pod authenticate itself against Azure services that support it, such as Storage or SQL. The pod is assigned an Azure Identity that lets them authenticate to Azure Active Directory and receive a digital token. This digital token can be presented to other Azure services that check if the pod is authorized to access the service and perform the required actions. This approach means that no secrets are required for database connection strings, for example. The simplified workflow for pod managed identity is shown in the following diagram:



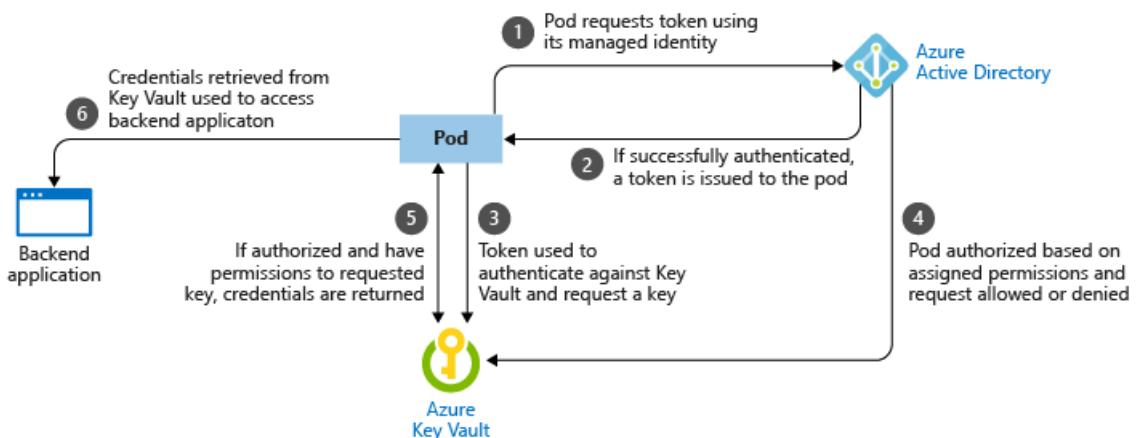
With a managed identity, your application code doesn't need to include credentials to access a service, such as Azure Storage. As each pod authenticates with its own identity, so you can audit and review access. If your application connects with other Azure services, use managed identities to limit credential reuse and risk of exposure.

For more information about pod identities, see [Configure an AKS cluster to use pod managed identities and with your applications](#)

### Use Azure Key Vault with FlexVol

Managed pod identities work great to authenticate against supporting Azure services. For your own services or applications without managed identities for Azure resources, you still authenticate using credentials or keys. A digital vault can be used to store these credentials.

When applications need a credential, they communicate with the digital vault, retrieve the latest credentials, and then connect to the required service. Azure Key Vault can be this digital vault. The simplified workflow for retrieving a credential from Azure Key Vault using pod managed identities is shown in the following diagram:



With Key Vault, you store and regularly rotate secrets such as credentials, storage account keys, or certificates. You can integrate Azure Key Vault with an AKS cluster using a FlexVolume. The FlexVolume driver lets the AKS cluster natively retrieve credentials from Key Vault and securely provide them only to the requesting pod. Work with your cluster operator to deploy the Key Vault FlexVol driver onto the AKS nodes. You can use a pod managed identity to request access to Key Vault and retrieve the credentials you need through the FlexVolume driver.

Azure Key Vault with FlexVol is intended for use with applications and services running on Linux pods and nodes.

## Next steps

This article focused on how to secure your pods. To implement some of these areas, see the following articles:

- [Use managed identities for Azure resources with AKS](#)
- [Integrate Azure Key Vault with AKS](#)



# Quotas, virtual machine size restrictions, and region availability in Azure Kubernetes Service (AKS)

2/25/2020 • 2 minutes to read • [Edit Online](#)

All Azure services set default limits and quotas for resources and features. Certain virtual machine (VM) SKUs are also restricted for use.

This article details the default resource limits for Azure Kubernetes Service (AKS) resources and the availability of AKS in Azure regions.

## Service quotas and limits

RESOURCE	DEFAULT LIMIT
Maximum clusters per subscription	100
Maximum nodes per cluster with Virtual Machine Availability Sets and Basic Load Balancer SKU	100
Maximum nodes per cluster with Virtual Machine Scale Sets and <a href="#">Standard Load Balancer SKU</a>	1000 (100 nodes per <a href="#">node pool</a> )
Maximum pods per node: <a href="#">Basic networking</a> with Kubenet	110
Maximum pods per node: <a href="#">Advanced networking</a> with Azure Container Networking Interface	Azure CLI deployment: 30 <sup>1</sup> Azure Resource Manager template: 30 <sup>1</sup> Portal deployment: 30

<sup>1</sup>When you deploy an Azure Kubernetes Service (AKS) cluster with the Azure CLI or a Resource Manager template, this value is configurable up to 250 pods per node. You can't configure maximum pods per node after you've already deployed an AKS cluster, or if you deploy a cluster by using the Azure portal.

## Provisioned infrastructure

All other network, compute, and storage limitations apply to the provisioned infrastructure. For the relevant limits, see [Azure subscription and service limits](#).

### IMPORTANT

When you upgrade an AKS cluster, additional resources are temporarily consumed. These resources include available IP addresses in a virtual network subnet, or virtual machine vCPU quota. If you use Windows Server containers (currently in preview in AKS), the only endorsed approach to apply the latest updates to the nodes is to perform an upgrade operation. A failed cluster upgrade process may indicate that you don't have the available IP address space or vCPU quota to handle these temporary resources. For more information on the Windows Server node upgrade process, see [Upgrade a node pool in AKS](#).

## Restricted VM sizes

Each node in an AKS cluster contains a fixed amount of compute resources such as vCPU and memory. If an AKS node contains insufficient compute resources, pods might fail to run correctly. To ensure that the required *kube-*

*system* pods and your applications can reliably be scheduled, **don't use the following VM SKUs in AKS:**

- Standard\_A0
- Standard\_A1
- Standard\_A1\_v2
- Standard\_B1s
- Standard\_B1ms
- Standard\_F1
- Standard\_F1s

For more information on VM types and their compute resources, see [Sizes for virtual machines in Azure](#).

## Region availability

For the latest list of where you can deploy and run clusters, see [AKS region availability](#).

## Next steps

Certain default limits and quotas can be increased. If your resource supports an increase, request the increase through an [Azure support request](#) (for **Issue type**, select **Quota**).

# Migrate to Azure Kubernetes Service (AKS)

2/26/2020 • 7 minutes to read • [Edit Online](#)

This article helps you plan and execute a successful migration to Azure Kubernetes Service (AKS). To help you make key decisions, this guide provides details for the current recommended configuration for AKS. This article doesn't cover every scenario, and where appropriate, the article contains links to more detailed information for planning a successful migration.

This document can be used to help support the following scenarios:

- Migrating an AKS Cluster backed by [Availability Sets](#) to [Virtual Machine Scale Sets](#)
- Migrating an AKS cluster to use a [Standard SKU](#) load balancer
- Migrating from [Azure Container Service \(ACS\) - retiring January 31, 2020](#) to AKS
- Migrating from [AKS engine](#) to AKS
- Migrating from non-Azure based Kubernetes clusters to AKS

When migrating, ensure your target Kubernetes version is within the supported window for AKS. If using an older version, it may not be within the supported range and require upgrading versions to be supported by AKS. See [AKS supported Kubernetes versions](#) for more information.

If you're migrating to a newer version of Kubernetes, review [Kubernetes version and version skew support policy](#).

Several open-source tools can help with your migration, depending on your scenario:

- [Velero](#) (Requires Kubernetes 1.7+)
- [Azure Kube CLI extension](#)
- [ReShifter](#)

In this article we will summarize migration details for:

- AKS with Standard Load Balancer and Virtual Machine Scale Sets
- Existing attached Azure Services
- Ensure valid quotas
- High Availability and business continuity
- Considerations for stateless applications
- Considerations for stateful applications
- Deployment of your cluster configuration

## AKS with Standard Load Balancer and Virtual Machine Scale Sets

AKS is a managed service offering unique capabilities with lower management overhead. As a result of being a managed service, you must select from a set of [regions](#) which AKS supports. The transition from your existing cluster to AKS may require modifying your existing applications so they remain healthy on the AKS managed control plane.

We recommend using AKS clusters backed by [Virtual Machine Scale Sets](#) and the [Azure Standard Load Balancer](#) to ensure you get features such as [multiple node pools](#), [Availability Zones](#), [Authorized IP ranges](#), [Cluster Autoscaler](#), [Azure Policy for AKS](#), and other new features as they are released.

AKS clusters backed by [Virtual Machine Availability Sets](#) lack support for many of these features.

The following example creates an AKS cluster with single node pool backed by a virtual machine scale set. It uses a

standard load balancer. It also enables the cluster autoscaler on the node pool for the cluster and sets a minimum of 1 and maximum of 3 nodes:

```
# First create a resource group
az group create --name myResourceGroup --location eastus

# Now create the AKS cluster and enable the cluster autoscaler
az aks create \
  --resource-group myResourceGroup \
  --name myAKSCluster \
  --node-count 1 \
  --vm-set-type VirtualMachineScaleSets \
  --load-balancer-sku standard \
  --enable-cluster-autoscaler \
  --min-count 1 \
  --max-count 3
```

## Existing attached Azure Services

When migrating clusters you may have attached external Azure services. These do not require resource recreation, but they will require updating connections from previous to new clusters to maintain functionality.

- Azure Container Registry
- Log Analytics
- Application Insights
- Traffic Manager
- Storage Account
- External Databases

## Ensure valid quotas

Because additional virtual machines will be deployed into your subscription during migration, you should verify that your quotas and limits are sufficient for these resources. You may need to request an increase in [vCPU quota](#).

You may need to request an increase for [Network quotas](#) to ensure you don't exhaust IPs. See [networking and IP ranges for AKS](#) for additional information.

For more information, see [Azure subscription and service limits](#). To check your current quotas, in the Azure portal, go to the [subscriptions blade](#), select your subscription, and then select **Usage + quotas**.

## High Availability and Business Continuity

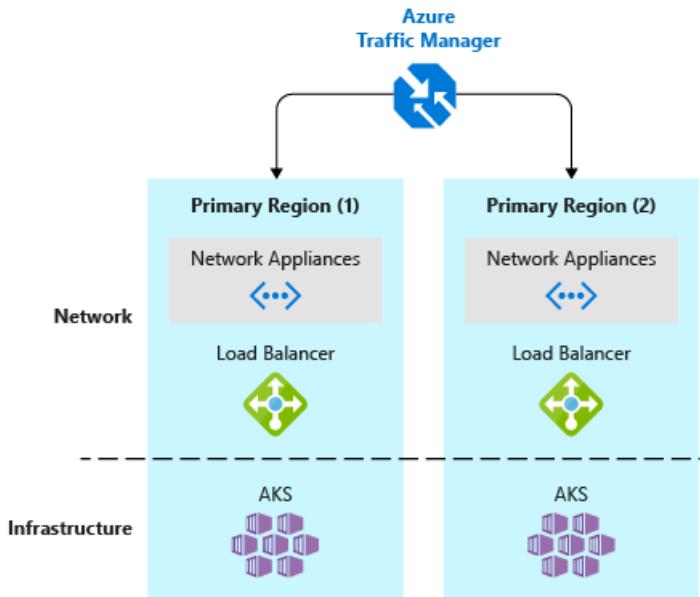
If your application cannot handle downtime, you will need to follow best practices for high availability migration scenarios. Best practices for complex business continuity planning, disaster recovery, and maximizing uptime are beyond the scope of this document. Read more about [Best practices for business continuity and disaster recovery in Azure Kubernetes Service \(AKS\)](#) to learn more.

For complex applications, you'll typically migrate over time rather than all at once. That means that the old and new environments might need to communicate over the network. Applications that previously used `ClusterIP` services to communicate might need to be exposed as type `LoadBalancer` and be secured appropriately.

To complete the migration, you'll want to point clients to the new services that are running on AKS. We recommend that you redirect traffic by updating DNS to point to the Load Balancer that sits in front of your AKS cluster.

[Azure Traffic Manager](#) can direct customers to the desired Kubernetes cluster and application instance. Traffic Manager is a DNS-based traffic load balancer that can distribute network traffic across regions. For the best

performance and redundancy, direct all application traffic through Traffic Manager before it goes to your AKS cluster. In a multicloud deployment, customers should connect to a Traffic Manager DNS name that points to the services on each AKS cluster. Define these services by using Traffic Manager endpoints. Each endpoint is the *service load balancer IP*. Use this configuration to direct network traffic from the Traffic Manager endpoint in one region to the endpoint in a different region.



[Azure Front Door Service](#) is another option for routing traffic for AKS clusters. Azure Front Door Service enables you to define, manage, and monitor the global routing for your web traffic by optimizing for best performance and instant global failover for high availability.

### Considerations for stateless applications

Stateless application migration is the most straightforward case. Apply your resource definitions (YAML or Helm) to the new cluster, make sure everything works as expected, and redirect traffic to activate your new cluster.

### Considerations for stateful applications

Carefully plan your migration of stateful applications to avoid data loss or unexpected downtime.

If you use Azure Files, you can mount the file share as a volume into the new cluster:

- [Mount Static Azure Files as a Volume](#)

If you use Azure Managed Disks, you can only mount the disk if unattached to any VM:

- [Mount Static Azure Disk as a Volume](#)

If neither of those approaches work, you can use a backup and restore options:

- [Velero on Azure](#)

#### Azure Files

Unlike disks, Azure Files can be mounted to multiple hosts concurrently. In your AKS cluster, Azure and Kubernetes don't prevent you from creating a pod that your ACS cluster still uses. To prevent data loss and unexpected behavior, ensure that the clusters don't write to the same files at the same time.

If your application can host multiple replicas that point to the same file share, follow the stateless migration steps and deploy your YAML definitions to your new cluster. If not, one possible migration approach involves the following steps:

- Validate your application is working correctly.
- Point your live traffic to your new AKS cluster.
- Disconnect the old cluster.

If you want to start with an empty share and make a copy of the source data, you can use the `az storage file copy` commands to migrate your data.

### Migrating persistent volumes

If you're migrating existing persistent volumes to AKS, you'll generally follow these steps:

- Quiesce writes to the application. (This step is optional and requires downtime.)
- Take snapshots of the disks.
- Create new managed disks from the snapshots.
- Create persistent volumes in AKS.
- Update pod specifications to [use existing volumes](#) rather than PersistentVolumeClaims (static provisioning).
- Deploy your application to AKS.
- Validate your application is working correctly.
- Point your live traffic to your new AKS cluster.

#### IMPORTANT

If you choose not to quiesce writes, you'll need to replicate data to the new deployment. Otherwise you'll miss the data that was written after you took the disk snapshots.

Some open-source tools can help you create managed disks and migrate volumes between Kubernetes clusters:

- [Azure CLI Disk Copy extension](#) copies and converts disks across resource groups and Azure regions.
- [Azure Kube CLI extension](#) enumerates ACS Kubernetes volumes and migrates them to an AKS cluster.

### Deployment of your cluster configuration

We recommend that you use your existing Continuous Integration (CI) and Continuous Deliver (CD) pipeline to deploy a known-good configuration to AKS. You can use Azure Pipelines to [build and deploy your applications to AKS](#). Clone your existing deployment tasks and ensure that `kubeconfig` points to the new AKS cluster.

If that's not possible, export resource definitions from your existing Kubernetes cluster and then apply them to AKS. You can use `kubectl` to export objects.

```
kubectl get deployment -o=yaml --export > deployments.yaml
```

### Moving existing resources to another region

You may want to move your AKS cluster to a [different region supported by AKS](#). We recommend that you create a new cluster in the other region then deploy your resources and applications to your new cluster. In addition, if you have any services such as [Azure Dev Spaces](#) running on your AKS cluster, you will also need to install and configure those services on your cluster in the new region.

In this article we summarized migration details for:

- AKS with Standard Load Balancer and Virtual Machine Scale Sets
- Existing attached Azure Services
- Ensure valid quotas
- High Availability and business continuity
- Considerations for stateless applications
- Considerations for stateful applications
- Deployment of your cluster configuration

# Supported Kubernetes versions in Azure Kubernetes Service (AKS)

2/25/2020 • 5 minutes to read • [Edit Online](#)

The Kubernetes community releases minor versions roughly every three months. These releases include new features and improvements. Patch releases are more frequent (sometimes weekly) and are only intended for critical bug fixes in a minor version. These patch releases include fixes for security vulnerabilities or major bugs impacting a large number of customers and products running in production based on Kubernetes.

AKS aims to certify and release new Kubernetes versions within 30 days of an upstream release, subject to the stability of the release.

## Kubernetes versions

Kubernetes uses the standard [Semantic Versioning](#) versioning scheme. This means that each version of Kubernetes follows this numbering scheme:

[major].[minor].[patch]

Example:

1.12.14  
1.12.15

Each number in the version indicates general compatibility with the previous version:

- Major versions change when incompatible API changes or backwards compatibility may be broken.
- Minor versions change when functionality changes are made that are backwards compatible to the other minor releases.
- Patch versions change when backwards-compatible bug fixes are made.

Users should aim to run the latest patch release of the minor version they are running, for example if your production cluster is on 1.12.14 and 1.12.15 is the latest available patch version available for the 1.12 series, you should upgrade to 1.12.15 as soon as you are able to ensure your cluster is fully patched and supported.

## Kubernetes version support policy

AKS supports three minor versions of Kubernetes:

- The current minor version that is released in AKS (N)
- Two previous minor versions. Each supported minor version also supports two stable patches.

This is known as "N-2": (N (Latest release) - 2 (minor versions)).

For example, if AKS introduces 1.15.a today, support is provided for the following versions:

NEW MINOR VERSION	SUPPORTED VERSION LIST
1.15.a	1.15.a, 1.15.b, 1.14.c, 1.14.d, 1.13.e, 1.13.f

Where ".letter" is representative of patch versions.

For details on communications regarding version changes and expectations, see "Communications" below.

When a new minor version is introduced, the oldest minor version and patch releases supported are deprecated and removed. For example, if the current supported version list is:

```
1.15.a  
1.15.b  
1.14.c  
1.14.d  
1.13.e  
1.13.f
```

And AKS releases 1.16, *this means that the 1.13 versions (all 1.13 versions) will be removed and are out of support.*

#### NOTE

Please note, that if customers are running an unsupported Kubernetes version, they will be asked to upgrade when requesting support for the cluster. Clusters running unsupported Kubernetes releases are not covered by the [AKS support policies](#).

In addition to the above on minor versions, AKS supports the two latest **patch** releases of a given minor version. For example, given the following supported versions:

```
Current Supported Version List  
-----  
1.15.2, 1.15.1, 1.14.5, 1.14.4
```

If upstream Kubernetes released 1.15.3 and 1.14.6 and AKS releases those patch versions, the oldest patch versions are deprecated and removed, and the supported version list becomes:

```
New Supported Version List  
-----  
1.15.*3*, 1.15.*2*, 1.14.*6*, 1.14.*5*
```

## Communications

- For new **minor** versions of Kubernetes
  - All users are notified publicly of the new version and what version will be removed.
  - When a new patch version is released, the oldest patch release is removed at the same time.
  - Customers have **30 days** from the public notification date to upgrade to a supported minor version release.
- For new **patch** versions of Kubernetes
  - All users are notified of the new patch version being released and to upgrade to the latest patch release.
  - Users have **30 days** to upgrade to a newer, supported patch release before the oldest is removed.

AKS defines a "released version" as the generally available versions, enabled in all SLO / Quality of Service measurements and available in all regions. AKS may also support preview versions which are explicitly labeled and subject to Preview terms and conditions.

#### Notification channels for AKS changes

AKS publishes regular service updates which summarize new Kubernetes versions, service changes, and component updates that have been released on the service on [GitHub](#).

These changes are rolled to all customers as part of regular maintenance that is offered as part of the managed

service, some require explicit upgrades while others require no action.

Notifications are also sent via:

- [AKS Release notes](#)
- Azure portal notifications
- [Azure update channel](#)

## Supported Versions Policy Exceptions

AKS reserves the right to add or remove new/existing versions that have been identified to have one or more critical production impacting bugs or security issues without advance notice.

Specific patch releases may be skipped, or rollout accelerated depending on the severity of the bug or security issue.

### Azure portal and CLI default versions

When you deploy an AKS cluster in the portal or with the Azure CLI, the cluster is defaulted to the N-1 minor version and latest patch. For example, if AKS supports *1.15.a*, *1.15.b*, *1.14.c*, *1.14.d*, *1.13.e*, and *1.13.f*, the default version selected is *1.14.c*.

AKS chooses the default of N-1 to provide customers a known, stable, and patched version by default.

## List currently supported versions

To find out what versions are currently available for your subscription and region, use the [az aks get-versions](#) command. The following example lists the available Kubernetes versions for the *EastUS* region:

```
az aks get-versions --location eastus --output table
```

## FAQ

### What happens when a customer upgrades a Kubernetes cluster with a minor version that is not supported?

If you are on the *n-3* version, you are outside of support and will be asked to upgrade. If your upgrade from version *n-3* to *n-2* succeeds, you are now within our support policies. For example:

- If the oldest supported AKS version is *1.13.a* and you are on *1.12.b* or older, you are outside of support.
- If the upgrade from *1.12.b* to *1.13.a* or higher succeeds, you are back within our support policies.

Upgrades to versions older than the supported window of *N-2* are not supported. In such cases, we recommend customers create new AKS clusters and redeploy their workloads with versions in the supported window.

### What does 'Outside of Support' mean

'Outside of Support' means that the version you are running is outside of the supported versions list, and you will be asked to upgrade the cluster to a supported version when requesting support. Additionally, AKS does not make any runtime or other guarantees for clusters outside of the supported versions list.

### What happens when a customer scales a Kubernetes cluster with a minor version that is not supported?

For minor versions not supported by AKS, scaling in or out should continue to work but it is highly recommended to upgrade to bring your cluster back into support.

### Can a customer stay on a Kubernetes version forever?

Yes. However, if the cluster is not on one of the versions supported by AKS, the cluster is out of the AKS support

policies. Azure does not automatically upgrade your cluster or delete it.

## **What version does the control plane support if the node pool is not in one of the supported AKS versions?**

The control plane must be within a window of versions from all node pools. For details on upgrading the control plane or node pools, visit documentation on [upgrading node pools](#).

## Next steps

For information on how to upgrade your cluster, see [Upgrade an Azure Kubernetes Service \(AKS\) cluster](#).

# Security hardening in AKS virtual machine hosts

2/25/2020 • 3 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) is a secure service compliant with SOC, ISO, PCI DSS, and HIPAA standards. This article covers the security hardening applied to AKS virtual machine hosts. For more information about AKS security, see [Security concepts for applications and clusters in Azure Kubernetes Service \(AKS\)](#).

AKS clusters are deployed on host virtual machines, which run a security optimized OS. This host OS is currently based on an Ubuntu 16.04.LTS image with a set of additional security hardening steps applied (see Security hardening details).

The goal of the security hardened host OS is to reduce the surface area of attack and allow the deployment of containers in a secure fashion.

## IMPORTANT

The security hardened OS is NOT CIS benchmarked. While there are overlaps with CIS benchmarks, the goal is not to be CIS-compliant. The goal for host OS hardening is to converge on a level of security consistent with Microsoft's own internal host security standards.

## Security hardening features

- AKS provides a security optimized host OS by default. There is no current option to select an alternate operating system.
- Azure applies daily patches (including security patches) to AKS virtual machine hosts. Some of these patches will require a reboot, while others will not. You are responsible for scheduling AKS VM host reboots as needed. For guidance on how to automate AKS patching see [patching AKS nodes](#).

Below is a summary of image hardening work that is implemented in AKS-Engine to produce the security optimized host OS. The work was implemented in [this GitHub project](#).

AKS-Engine does not promote or adhere to any specific security standard at this time, but CIS (Center for Internet Security) audit IDs are provided for convenience where applicable.

## What's configured?

CIS	Audit Description
1.1.1.1	Ensure mounting of cramfs filesystems is disabled
1.1.1.2	Ensure mounting of freevxf filesystems is disabled
1.1.1.3	Ensure mounting of jffs2 filesystems is disabled
1.1.1.4	Ensure mounting of HFS filesystems is disabled
1.1.1.5	Ensure mounting of HFS Plus filesystems is disabled
1.4.3	Ensure authentication required for single user mode

CIS	AUDIT DESCRIPTION
1.7.1.2	Ensure local login warning banner is configured properly
1.7.1.3	Ensure remote login warning banner is configured properly
1.7.1.5	Ensure permissions on /etc/issue are configured
1.7.1.6	Ensure permissions on /etc/issue.net are configured
2.1.5	Ensure that --streaming-connection-idle-timeout is not set to 0
3.1.2	Ensure packet redirect sending is disabled
3.2.1	Ensure source routed packages are not accepted
3.2.2	Ensure ICMP redirects are not accepted
3.2.3	Ensure secure ICMP redirects are not accepted
3.2.4	Ensure suspicious packets are logged
3.3.1	Ensure IPv6 router advertisements are not accepted
3.5.1	Ensure DCCP is disabled
3.5.2	Ensure SCTP is disabled
3.5.3	Ensure RDS is disabled
3.5.4	Ensure TIPC is disabled
4.2.1.2	Ensure logging is configured
5.1.2	Ensure permissions on /etc/crontab are configured
5.2.4	Ensure SSH X11 forwarding is disabled
5.2.5	Ensure SSH MaxAuthTries is set to 4 or less
5.2.8	Ensure SSH root login is disabled
5.2.10	Ensure SSH PermitUserEnvironment is disabled
5.2.11	Ensure only approved MAX algorithms are used
5.2.12	Ensure SSH Idle Timeout Interval is configured
5.2.13	Ensure SSH LoginGraceTime is set to one minute or less
5.2.15	Ensure SSH warning banner is configured

CIS	AUDIT DESCRIPTION
5.3.1	Ensure password creation requirements are configured
5.4.1.1	Ensure password expiration is 90 days or less
5.4.1.4	Ensure inactive password lock is 30 days or less
5.4.4	Ensure default user umask is 027 or more restrictive
5.6	Ensure access to the su command is restricted

## Additional notes

- To further reduce the attack surface area, some unnecessary kernel module drivers have been disabled in the OS.
- The security hardened OS is NOT supported outside of the AKS platform.

## Next steps

See the following articles for more information about AKS security:

[Azure Kubernetes Service \(AKS\)](#)

[AKS security considerations](#)

[AKS best practices](#)

# Azure Kubernetes Service (AKS) Diagnostics overview

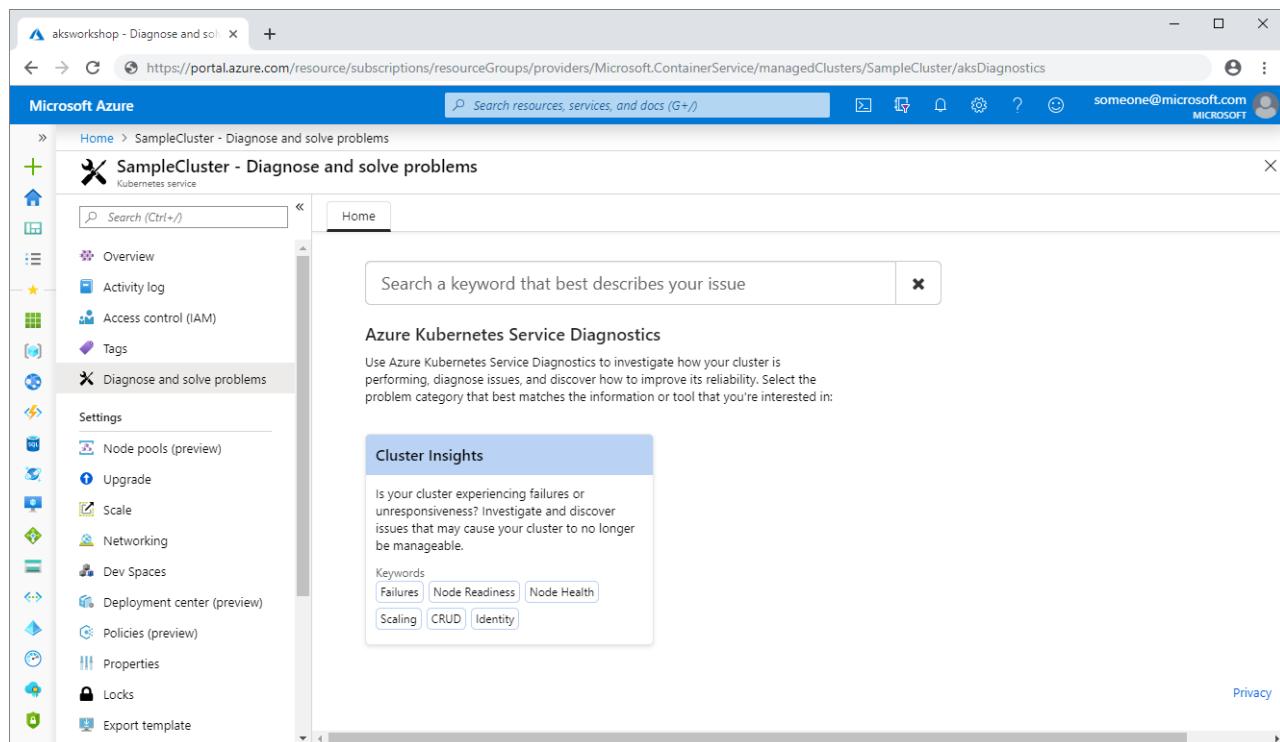
2/25/2020 • 2 minutes to read • [Edit Online](#)

Troubleshooting Azure Kubernetes Service (AKS) cluster issues is an important part of maintaining your cluster, especially if your cluster is running mission-critical workloads. AKS Diagnostics is an intelligent, self-diagnostic experience that helps you identify and resolve problems in your cluster. AKS Diagnostics is cloud-native, and you can use it with no extra configuration or billing cost.

## Open AKS Diagnostics

To access AKS Diagnostics:

- Navigate to your Kubernetes cluster in the [Azure portal](#).
- Click on **Diagnose and solve problems** in the left navigation, which opens AKS Diagnostics.
- Choose a category that best describes the issue of your cluster by using the keywords in the homepage tile, or type a keyword that best describes your issue in the search bar, for example *Cluster Node Issues*.



## View a diagnostic report

After you click on a category, you can view a diagnostic report specific to your cluster. Diagnostic report intelligently calls out if there is any issue in your cluster with status icons. You can drill down on each topic by clicking on **More Info** to see detailed description of the issue, recommended actions, links to helpful docs, related-metrics, and logging data. Diagnostic reports are intelligently generated based on the current state of your cluster after running a variety of checks. Diagnostic reports can be a useful tool for pinpointing the problem of your cluster and finding the next steps to resolve the issue.

1h 6h 1d 2019-10-21 23:23 2019-10-22 23:07 UTC

## Cluster Insights

Identifies scenarios that may cause a cluster to no longer be manageable.

[Send Feedback](#) [Copy Report](#)

### Observations

Events observed during this time period

Issues	Description	Link
Cluster Node Issues	Node Issues Detected One or more <a href="#">Node based issues</a> have been detected.	<a href="#">More Info</a>

### Successful Checks

Tests that succeeded for your app

Checks	Description	Link
Identity and Security Management	No Identity or Security failures detected.	<a href="#">More Info</a>
Create, Read, Update & Delete Operations	There were no Create, Update, Read or Delete operations that we found to have issues.	<a href="#">More Info</a>

### Cluster Node Issues [Back to Observations](#)

#### Node Issues Detected

**Description** One or more [Node based issues](#) have been detected.

#### Node Insufficient Resources Detected

**Description** AKS monitoring has detected at least one node has insufficient resources. You can run the following to better understand your resource consumption: kubectl get no -o wide && kubectl top no

**Recommended Action** It is recommended you attempt to either:

- 1) Horizontally Scale your Cluster [Click Here](#)
  - 2) Resize your VMs (temporary fix: [Click Here](#))
  - 3) Migrate your workload to an AKS cluster with appropriate VM Sizes
- If the error still persists, please engage Microsoft Support.

#### Logging Data

NodeName	ConstrainedResource
aks-agentpool-00000000-0	memoryPressure

## Cluster Insights

The following diagnostic checks are available in **Cluster Insights**.

### Cluster Node Issues

Cluster Node Issues checks for node-related issues that may cause your cluster to behave unexpectedly.

- Node readiness issues
- Node failures
- Insufficient resources
- Node missing IP configuration
- Node CNI failures
- Node not found
- Node power off
- Node authentication failure

- Node kube-proxy stale

## Create, read, update & delete operations

CRUD Operations checks for any CRUD operations that may cause issues in your cluster.

- In-use subnet delete operation error
- Network security group delete operation error
- In-use route table delete operation error
- Referenced resource provisioning error
- Public IP address delete operation error
- Deployment failure due to deployment quota
- Operation error due to organization policy
- Missing subscription registration
- VM extension provisioning error
- Subnet capacity
- Quota exceeded error

## Identity and security management

Identity and Security Management detects authentication and authorization errors that may prevent communication to your cluster.

- Node authorization failures
- 401 errors
- 403 errors

## Next steps

Collect logs to help you further troubleshoot your cluster issues by using [AKS Periscope](#).

Post your questions or feedback at [UserVoice](#) by adding "[Diag]" in the title.

# Scale the node count in an Azure Kubernetes Service (AKS) cluster

2/25/2020 • 2 minutes to read • [Edit Online](#)

If the resource needs of your applications change, you can manually scale an AKS cluster to run a different number of nodes. When you scale down, nodes are carefully [cordoned and drained](#) to minimize disruption to running applications. When you scale up, AKS waits until nodes are marked Ready by the Kubernetes cluster before pods are scheduled on them.

## Scale the cluster nodes

First, get the *name* of your node pool using the [az aks show](#) command. The following example gets the node pool name for the cluster named *myAKSCluster* in the *myResourceGroup* resource group:

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query agentPoolProfiles
```

The following example output shows that the *name* is *nodepool1*:

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query agentPoolProfiles
[
  {
    "count": 1,
    "maxPods": 110,
    "name": "nodepool1",
    "osDiskSizeGb": 30,
    "osType": "Linux",
    "storageProfile": "ManagedDisks",
    "vmSize": "Standard_DS2_v2"
  }
]
```

Use the [az aks scale](#) command to scale the cluster nodes. The following example scales a cluster named *myAKSCluster* to a single node. Provide your own *--nodepool-name* from the previous command, such as *nodepool1*:

```
az aks scale --resource-group myResourceGroup --name myAKSCluster --node-count 1 --nodepool-name <your node pool name>
```

The following example output shows the cluster has successfully scaled to one node, as shown in the *agentPoolProfiles* section:

```
{  
  "aadProfile": null,  
  "addonProfiles": null,  
  "agentPoolProfiles": [  
    {  
      "count": 1,  
      "maxPods": 110,  
      "name": "nodepool1",  
      "osDiskSizeGb": 30,  
      "osType": "Linux",  
      "storageProfile": "ManagedDisks",  
      "vmSize": "Standard_DS2_v2",  
      "vnetSubnetId": null  
    }  
  ],  
  [...]  
}
```

## Next steps

In this article, you manually scaled an AKS cluster to increase or decrease the number of nodes. You can also use the [cluster autoscaler](#) to automatically scale your cluster.

# Upgrade an Azure Kubernetes Service (AKS) cluster

2/26/2020 • 3 minutes to read • [Edit Online](#)

As part of the lifecycle of an AKS cluster, you often need to upgrade to the latest Kubernetes version. It is important you apply the latest Kubernetes security releases, or upgrade to get the latest features. This article shows you how to upgrade the master components or a single, default node pool in an AKS cluster.

For AKS clusters that use multiple node pools or Windows Server nodes (currently in preview in AKS), see [Upgrade a node pool in AKS](#).

## Before you begin

This article requires that you are running the Azure CLI version 2.0.65 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

### WARNING

An AKS cluster upgrade triggers a cordon and drain of your nodes. If you have a low compute quota available, the upgrade may fail. See [increase quotas](#) for more information. If you are running your own cluster autoscaler deployment please disable it (you can scale it to zero replicas) during the upgrade as there is a chance it will interfere with the upgrade process. Managed autoscaler automatically handles this.

## Check for available AKS cluster upgrades

To check which Kubernetes releases are available for your cluster, use the `az aks get-upgrades` command. The following example checks for available upgrades to the cluster named `myAKSCluster` in the resource group named `myResourceGroup`:

```
az aks get-upgrades --resource-group myResourceGroup --name myAKSCluster --output table
```

### NOTE

When you upgrade an AKS cluster, Kubernetes minor versions cannot be skipped. For example, upgrades between `1.12.x -> 1.13.x` or `1.13.x -> 1.14.x` are allowed, however `1.12.x -> 1.14.x` is not.

To upgrade, from `1.12.x -> 1.14.x`, first upgrade from `1.12.x -> 1.13.x`, then upgrade from `1.13.x -> 1.14.x`.

The following example output shows that the cluster can be upgraded to versions `1.13.9` and `1.13.10`:

Name	ResourceGroup	MasterVersion	NodePoolVersion	Upgrades
default	myResourceGroup	1.12.8	1.12.8	1.13.9, 1.13.10

If no upgrade is available, you will get:

```
ERROR: Table output unavailable. Use the --query option to specify an appropriate query. Use --debug for more info.
```

# Upgrade an AKS cluster

With a list of available versions for your AKS cluster, use the [az aks upgrade](#) command to upgrade. During the upgrade process, AKS adds a new node to the cluster that runs the specified Kubernetes version, then carefully [cordon and drains](#) one of the old nodes to minimize disruption to running applications. When the new node is confirmed as running application pods, the old node is deleted. This process repeats until all nodes in the cluster have been upgraded.

The following example upgrades a cluster to version *1.13.10*:

```
az aks upgrade --resource-group myResourceGroup --name myAKSCluster --kubernetes-version 1.13.10
```

It takes a few minutes to upgrade the cluster, depending on how many nodes you have.

## NOTE

There is a total allowed time for a cluster upgrade to complete. This time is calculated by taking the product of `10 minutes * total number of nodes in the cluster`. For example in a 20 node cluster, upgrade operations must succeed in 200 minutes or AKS will fail the operation to avoid an unrecoverable cluster state. To recover on upgrade failure, retry the upgrade operation after the timeout has been hit.

To confirm that the upgrade was successful, use the [az aks show](#) command:

```
az aks show --resource-group myResourceGroup --name myAKSCluster --output table
```

The following example output shows that the cluster now runs *1.13.10*:

Name	Location	ResourceGroup	KubernetesVersion	ProvisioningState	Fqdn
myAKSCluster	eastus	myResourceGroup	1.13.10	Succeeded	myaksclust-myresourcegroup-19da35-90efab95.hcp.eastus.azmk8s.io

## Next steps

This article showed you how to upgrade an existing AKS cluster. To learn more about deploying and managing AKS clusters, see the set of tutorials.

[AKS tutorials](#)

# Apply security and kernel updates to Linux nodes in Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

To protect your clusters, security updates are automatically applied to Linux nodes in AKS. These updates include OS security fixes or kernel updates. Some of these updates require a node reboot to complete the process. AKS doesn't automatically reboot these Linux nodes to complete the update process.

The process to keep Windows Server nodes (currently in preview in AKS) up to date is a little different. Windows Server nodes don't receive daily updates. Instead, you perform an AKS upgrade that deploys new nodes with the latest base Window Server image and patches. For AKS clusters that use Windows Server nodes, see [Upgrade a node pool in AKS](#).

This article shows you how to use the open-source [kured \(KUBernetes REboot Daemon\)](#) to watch for Linux nodes that require a reboot, then automatically handle the rescheduling of running pods and node reboot process.

## NOTE

Kured is an open-source project by Weaveworks. Support for this project in AKS is provided on a best-effort basis. Additional support can be found in the #weave-community Slack channel.

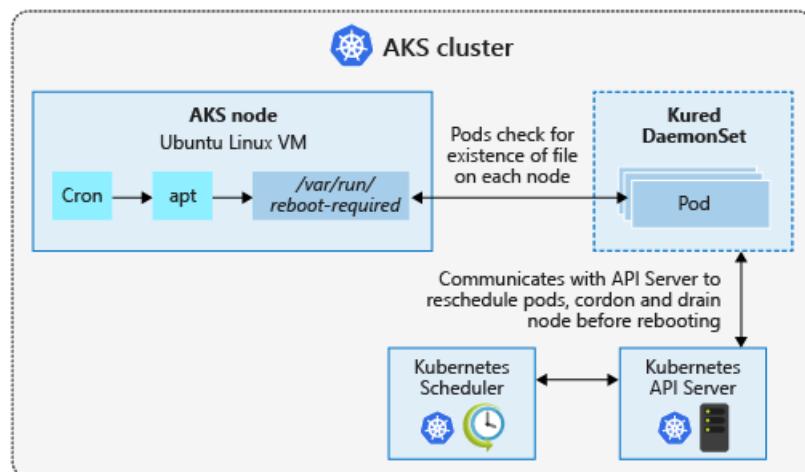
## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the [Azure CLI](#) or [using the Azure portal](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Understand the AKS node update experience

In an AKS cluster, your Kubernetes nodes run as Azure virtual machines (VMs). These Linux-based VMs use an Ubuntu image, with the OS configured to automatically check for updates every night. If security or kernel updates are available, they are automatically downloaded and installed.



Some security updates, such as kernel updates, require a node reboot to finalize the process. A Linux node that requires a reboot creates a file named `/var/run/reboot-required`. This reboot process doesn't happen

automatically.

You can use your own workflows and processes to handle node reboots, or use `kured` to orchestrate the process. With `kured`, a [DaemonSet](#) is deployed that runs a pod on each Linux node in the cluster. These pods in the DaemonSet watch for existence of the `/var/run/reboot-required` file, and then initiate a process to reboot the nodes.

## Node upgrades

There is an additional process in AKS that lets you *upgrade* a cluster. An upgrade is typically to move to a newer version of Kubernetes, not just apply node security updates. An AKS upgrade performs the following actions:

- A new node is deployed with the latest security updates and Kubernetes version applied.
- An old node is cordoned and drained.
- Pods are scheduled on the new node.
- The old node is deleted.

You can't remain on the same Kubernetes version during an upgrade event. You must specify a newer version of Kubernetes. To upgrade to the latest version of Kubernetes, you can [upgrade your AKS cluster](#).

## Deploy kured in an AKS cluster

To deploy the `kured` DaemonSet, apply the following sample YAML manifest from their GitHub project page. This manifest creates a role and cluster role, bindings, and a service account, then deploys the DaemonSet using `kured` version 1.1.0 that supports AKS clusters 1.9 or later.

```
kubectl apply -f https://github.com/weaveworks/kured/releases/download/1.2.0/kured-1.2.0-dockerhub.yaml
```

You can also configure additional parameters for `kured`, such as integration with Prometheus or Slack. For more information about additional configuration parameters, see the [kured installation docs](#).

## Update cluster nodes

By default, Linux nodes in AKS check for updates every evening. If you don't want to wait, you can manually perform an update to check that `kured` runs correctly. First, follow the steps to [SSH to one of your AKS nodes](#). Once you have an SSH connection to the Linux node, check for updates and apply them as follows:

```
sudo apt-get update && sudo apt-get upgrade -y
```

If updates were applied that require a node reboot, a file is written to `/var/run/reboot-required`. `Kured` checks for nodes that require a reboot every 60 minutes by default.

## Monitor and review reboot process

When one of the replicas in the DaemonSet has detected that a node reboot is required, a lock is placed on the node through the Kubernetes API. This lock prevents additional pods being scheduled on the node. The lock also indicates that only one node should be rebooted at a time. With the node cordoned off, running pods are drained from the node, and the node is rebooted.

You can monitor the status of the nodes using the `kubectl get nodes` command. The following example output shows a node with a status of *SchedulingDisabled* as the node prepares for the reboot process:

NAME	STATUS	ROLES	AGE	VERSION
aks-nodepool1-28993262-0	Ready, SchedulingDisabled	agent	1h	v1.11.7

Once the update process is complete, you can view the status of the nodes using the `kubectl get nodes` command with the `--output wide` parameter. This additional output lets you see a difference in `KERNEL-VERSION` of the underlying nodes, as shown in the following example output. The `aks-nodepool1-28993262-0` was updated in a previous step and shows kernel version `4.15.0-1039-azure`. The node `aks-nodepool1-28993262-1` that hasn't been updated shows kernel version `4.15.0-1037-azure`.

NAME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE
KERNEL-VERSION	CONTAINER-RUNTIME						
aks-nodepool1-28993262-0	Ready	agent	1h	v1.11.7	10.240.0.4	<none>	Ubuntu 16.04.6 LTS 4.15.0-1039-azure docker://3.0.4
aks-nodepool1-28993262-1	Ready	agent	1h	v1.11.7	10.240.0.5	<none>	Ubuntu 16.04.6 LTS 4.15.0-1037-azure docker://3.0.4

## Next steps

This article detailed how to use `kured` to reboot Linux nodes automatically as part of the security update process. To upgrade to the latest version of Kubernetes, you can [upgrade your AKS cluster](#).

For AKS clusters that use Windows Server nodes, see [Upgrade a node pool in AKS](#).

# Authenticate with Azure Container Registry from Azure Kubernetes Service

2/25/2020 • 2 minutes to read • [Edit Online](#)

When you're using Azure Container Registry (ACR) with Azure Kubernetes Service (AKS), an authentication mechanism needs to be established. This article provides examples for configuring authentication between these two Azure services.

You can set up the AKS to ACR integration in a few simple commands with the Azure CLI.

## Before you begin

These examples require:

- **Owner** or **Azure account administrator** role on the **Azure subscription**
- Azure CLI version 2.0.73 or later

## Create a new AKS cluster with ACR integration

You can set up AKS and ACR integration during the initial creation of your AKS cluster. To allow an AKS cluster to interact with ACR, an Azure Active Directory **service principal** is used. The following CLI command allows you to authorize an existing ACR in your subscription and configures the appropriate **ACRPull** role for the service principal. Supply valid values for your parameters below.

```
# set this to the name of your Azure Container Registry. It must be globally unique
MYACR=myContainerRegistry

# Run the following line to create an Azure Container Registry if you do not already have one
az acr create -n $MYACR -g myContainerRegistryResourceGroup --sku basic

# Create an AKS cluster with ACR integration
az aks create -n myAKScluster -g myResourceGroup --generate-ssh-keys --attach-acr $MYACR
```

Alternatively, you can specify the ACR name using an ACR resource ID, which has the following format:

/subscriptions/<subscription-id>/resourceGroups/<resource-group-name>/providers/Microsoft.ContainerRegistry/registries/<name>

```
az aks create -n myAKScluster -g myResourceGroup --generate-ssh-keys --attach-acr
/subscriptions/<subscription-
id>/resourceGroups/myContainerRegistryResourceGroup/providers/Microsoft.ContainerRegistry/registries/myContain
erRegistry
```

This step may take several minutes to complete.

## Configure ACR integration for existing AKS clusters

Integrate an existing ACR with existing AKS clusters by supplying valid values for **acr-name** or **acr-resource-id** as below.

```
az aks update -n myAKScluster -g myResourceGroup --attach-acr <acrName>
```

or,

```
az aks update -n myAKScluster -g myResourceGroup --attach-acr <acr-resource-id>
```

You can also remove the integration between an ACR and an AKS cluster with the following

```
az aks update -n myAKScluster -g myResourceGroup --detach-acr <acrName>
```

or

```
az aks update -n myAKScluster -g myResourceGroup --detach-acr <acr-resource-id>
```

## Working with ACR & AKS

### Import an image into your ACR

Import an image from docker hub into your ACR by running the following:

```
az acr import -n <myContainerRegistry> --source docker.io/library/nginx:latest --image nginx:v1
```

### Deploy the sample image from ACR to AKS

Ensure you have the proper AKS credentials

```
az aks get-credentials -g myResourceGroup -n myAKScluster
```

Create a file called **acr-nginx.yaml** that contains the following:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx0-deployment
  labels:
    app: nginx0-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx0
  template:
    metadata:
      labels:
        app: nginx0
    spec:
      containers:
        - name: nginx
          image: <replace this image property with your acr login server, image and tag>
          ports:
            - containerPort: 80
```

Next, run this deployment in your AKS cluster:

```
kubectl apply -f acr-nginx.yaml
```

You can monitor the deployment by running:

```
kubectl get pods
```

You should have two running pods.

NAME	READY	STATUS	RESTARTS	AGE
nginx0-deployment-669dfc4d4b-x74kr	1/1	Running	0	20s
nginx0-deployment-669dfc4d4b-xdpd6	1/1	Running	0	20s

# Create and configure an Azure Kubernetes Services (AKS) cluster to use virtual nodes using the Azure CLI

2/25/2020 • 8 minutes to read • [Edit Online](#)

To rapidly scale application workloads in an Azure Kubernetes Service (AKS) cluster, you can use virtual nodes. With virtual nodes, you have quick provisioning of pods, and only pay per second for their execution time. You don't need to wait for Kubernetes cluster autoscaler to deploy VM compute nodes to run the additional pods. Virtual nodes are only supported with Linux pods and nodes.

This article shows you how to create and configure the virtual network resources and AKS cluster, then enable virtual nodes.

## Before you begin

Virtual nodes enable network communication between pods that run in ACI and the AKS cluster. To provide this communication, a virtual network subnet is created and delegated permissions are assigned. Virtual nodes only work with AKS clusters created using *advanced* networking. By default, AKS clusters are created with *basic* networking. This article shows you how to create a virtual network and subnets, then deploy an AKS cluster that uses advanced networking.

If you have not previously used ACI, register the service provider with your subscription. You can check the status of the ACI provider registration using the [az provider list](#) command, as shown in the following example:

```
az provider list --query "[?contains(namespace,'Microsoft.ContainerInstance')]" -o table
```

The *Microsoft.ContainerInstance* provider should report as *Registered*, as shown in the following example output:

Namespace	RegistrationState
Microsoft.ContainerInstance	Registered

If the provider shows as *NotRegistered*, register the provider using the [az provider register](#) as shown in the following example:

```
az provider register --namespace Microsoft.ContainerInstance
```

## Regional availability

The following regions are supported for virtual node deployments:

- Australia East (australiaeast)
- Central US (centralus)
- East US (eastus)
- East US 2 (eastus2)
- Japan East (japaneast)
- North Europe (northeurope)
- Southeast Asia (southeastasia)

- West Central US (westcentralus)
- West Europe (westeurope)
- West US (westus)
- West US 2 (westus2)

## Known limitations

Virtual Nodes functionality is heavily dependent on ACI's feature set. The following scenarios are not yet supported with Virtual Nodes

- Using service principal to pull ACR images. [Workaround](#) is to use [Kubernetes secrets](#)
- [Virtual Network Limitations](#) including VNet peering, Kubernetes network policies, and outbound traffic to the internet with network security groups.
- Init containers
- [Host aliases](#)
- [Arguments](#) for exec in ACI
- [DaemonSets](#) will not deploy pods to the virtual node
- [Windows Server nodes \(currently in preview in AKS\)](#) are not supported alongside virtual nodes. You can use virtual nodes to schedule Windows Server containers without the need for Windows Server nodes in an AKS cluster.

## Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account.

To open the Cloud Shell, select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com/bash>. Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press enter to run it.

If you prefer to install and use the CLI locally, this article requires Azure CLI version 2.0.49 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create a resource group

An Azure resource group is a logical group in which Azure resources are deployed and managed. Create a resource group with the [az group create](#) command. The following example creates a resource group named `myResourceGroup` in the `westus` location.

```
az group create --name myResourceGroup --location westus
```

## Create a virtual network

Create a virtual network using the [az network vnet create](#) command. The following example creates a virtual network name `myVnet` with an address prefix of `10.0.0.0/8`, and a subnet named `myAKSSubnet`. The address prefix of this subnet defaults to `10.240.0.0/16`:

```
az network vnet create \
--resource-group myResourceGroup \
--name myVnet \
--address-prefixes 10.0.0.0/8 \
--subnet-name myAKSSubnet \
--subnet-prefix 10.240.0.0/16
```

Now create an additional subnet for virtual nodes using the [az network vnet subnet create](#) command. The following example creates a subnet named *myVirtualNodeSubnet* with the address prefix of *10.241.0.0/16*.

```
az network vnet subnet create \
--resource-group myResourceGroup \
--vnet-name myVnet \
--name myVirtualNodeSubnet \
--address-prefixes 10.241.0.0/16
```

## Create a service principal

To allow an AKS cluster to interact with other Azure resources, an Azure Active Directory service principal is used. This service principal can be automatically created by the Azure CLI or portal, or you can pre-create one and assign additional permissions.

Create a service principal using the [az ad sp create-for-rbac](#) command. The `--skip-assignment` parameter limits any additional permissions from being assigned.

```
az ad sp create-for-rbac --skip-assignment
```

The output is similar to the following example:

```
{
  "appId": "bef76eb3-d743-4a97-9534-03e9388811fc",
  "displayName": "azure-cli-2018-11-21-18-42-00",
  "name": "http://azure-cli-2018-11-21-18-42-00",
  "password": "1d257915-8714-4ce7-a7fb-0e5a5411df7f",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db48"
}
```

Make a note of the *appId* and *password*. These values are used in the following steps.

## Assign permissions to the virtual network

To allow your cluster to use and manage the virtual network, you must grant the AKS service principal the correct rights to use the network resources.

First, get the virtual network resource ID using [az network vnet show](#):

```
az network vnet show --resource-group myResourceGroup --name myVnet --query id -o tsv
```

To grant the correct access for the AKS cluster to use the virtual network, create a role assignment using the [az role assignment create](#) command. Replace `<appId>` and `<vnetId>` with the values gathered in the previous two steps.

```
az role assignment create --assignee <appId> --scope <vnetId> --role Contributor
```

# Create an AKS cluster

You deploy an AKS cluster into the AKS subnet created in a previous step. Get the ID of this subnet using [az network vnet subnet show](#):

```
az network vnet subnet show --resource-group myResourceGroup --vnet-name myVnet --name myAKSSubnet --query id  
-o tsv
```

Use the [az aks create](#) command to create an AKS cluster. The following example creates a cluster named *myAKSCluster* with one node. Replace `<subnetId>` with the ID obtained in the previous step, and then `<appId>` and `<password>` with the

```
az aks create \  
--resource-group myResourceGroup \  
--name myAKSCluster \  
--node-count 1 \  
--network-plugin azure \  
--service-cidr 10.0.0.0/16 \  
--dns-service-ip 10.0.0.10 \  
--docker-bridge-address 172.17.0.1/16 \  
--vnet-subnet-id <subnetId> \  
--service-principal <appId> \  
--client-secret <password>
```

After several minutes, the command completes and returns JSON-formatted information about the cluster.

## Enable virtual nodes addon

To enable virtual nodes, now use the [az aks enable-addons](#) command. The following example uses the subnet named *myVirtualNodeSubnet* created in a previous step:

```
az aks enable-addons \  
--resource-group myResourceGroup \  
--name myAKSCluster \  
--addons virtual-node \  
--subnet-name myVirtualNodeSubnet
```

## Connect to the cluster

To configure `kubectl` to connect to your Kubernetes cluster, use the [az aks get-credentials](#) command. This step downloads credentials and configures the Kubernetes CLI to use them.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, use the `kubectl get` command to return a list of the cluster nodes.

```
kubectl get nodes
```

The following example output shows the single VM node created and then the virtual node for Linux, *virtual-node-aci-linux*:

```
$ kubectl get nodes

NAME                  STATUS  ROLES   AGE      VERSION
virtual-node-aci-linux  Ready   agent   28m     v1.11.2
aks-agentpool-14693408-0  Ready   agent   32m     v1.11.2
```

## Deploy a sample app

Create a file named `virtual-node.yaml` and copy in the following YAML. To schedule the container on the node, a `nodeSelector` and `toleration` are defined.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aci-helloworld
spec:
  replicas: 1
  selector:
    matchLabels:
      app: aci-helloworld
  template:
    metadata:
      labels:
        app: aci-helloworld
    spec:
      containers:
        - name: aci-helloworld
          image: microsoft/aci-helloworld
          ports:
            - containerPort: 80
      nodeSelector:
        kubernetes.io/role: agent
        beta.kubernetes.io/os: linux
        type: virtual-kubelet
      tolerations:
        - key: virtual-kubelet.io/provider
          operator: Exists
        - key: azure.com/aci
          effect: NoSchedule
```

Run the application with the `kubectl apply` command.

```
kubectl apply -f virtual-node.yaml
```

Use the `kubectl get pods` command with the `-o wide` argument to output a list of pods and the scheduled node. Notice that the `aci-helloworld` pod has been scheduled on the `virtual-node-aci-linux` node.

```
$ kubectl get pods -o wide

NAME                  READY   STATUS    RESTARTS   AGE      IP           NODE
aci-helloworld-9b55975f-bnmfl  1/1     Running   0          4m      10.241.0.4   virtual-node-aci-linux
```

The pod is assigned an internal IP address from the Azure virtual network subnet delegated for use with virtual nodes.

## NOTE

If you use images stored in Azure Container Registry, [configure and use a Kubernetes secret](#). A current limitation of virtual nodes is that you can't use integrated Azure AD service principal authentication. If you don't use a secret, pods scheduled on virtual nodes fail to start and report the error `HTTP response status code 400 error code "InaccessibleImage"`.

## Test the virtual node pod

To test the pod running on the virtual node, browse to the demo application with a web client. As the pod is assigned an internal IP address, you can quickly test this connectivity from another pod on the AKS cluster. Create a test pod and attach a terminal session to it:

```
kubectl run --generator=run-pod/v1 -it --rm testvk --image=debian
```

Install `curl` in the pod using `apt-get`:

```
apt-get update && apt-get install -y curl
```

Now access the address of your pod using `curl`, such as <http://10.241.0.4>. Provide your own internal IP address shown in the previous `kubectl get pods` command:

```
curl -L http://10.241.0.4
```

The demo application is displayed, as shown in the following condensed example output:

```
$ curl -L 10.241.0.4

<html>
<head>
  <title>Welcome to Azure Container Instances!</title>
</head>
[...]
```

Close the terminal session to your test pod with `exit`. When your session is ended, the pod is deleted.

## Remove virtual nodes

If you no longer wish to use virtual nodes, you can disable them using the `az aks disable-addons` command.

First, delete the helloworld pod running on the virtual node:

```
kubectl delete -f virtual-node.yaml
```

The following example command disables the Linux virtual nodes:

```
az aks disable-addons --resource-group myResourceGroup --name myAKSCluster --addons virtual-node
```

Now, remove the virtual network resources and resource group:

```

# Change the name of your resource group, cluster and network resources as needed
RES_GROUP=myResourceGroup
AKS_CLUSTER=myAKScluster
AKS_VNET=myVnet
AKS_SUBNET=myVirtualNodeSubnet

# Get AKS node resource group
NODE_RES_GROUP=$(az aks show --resource-group $RES_GROUP --name $AKS_CLUSTER --query nodeResourceGroup --output tsv)

# Get network profile ID
NETWORK_PROFILE_ID=$(az network profile list --resource-group $NODE_RES_GROUP --query [0].id --output tsv)

# Delete the network profile
az network profile delete --id $NETWORK_PROFILE_ID -y

# Delete the subnet delegation to Azure Container Instances
az network vnet subnet update --resource-group $RES_GROUP --vnet-name $AKS_VNET --name $AKS_SUBNET --remove delegations 0

```

## Next steps

In this article, a pod was scheduled on the virtual node and assigned a private, internal IP address. You could instead create a service deployment and route traffic to your pod through a load balancer or ingress controller. For more information, see [Create a basic ingress controller in AKS](#).

Virtual nodes are often one component of a scaling solution in AKS. For more information on scaling solutions, see the following articles:

- [Use the Kubernetes horizontal pod autoscaler](#)
- [Use the Kubernetes cluster autoscaler](#)
- [Check out the Autoscale sample for Virtual Nodes](#)
- [Read more about the Virtual Kubelet open source library](#)

# Create and configure an Azure Kubernetes Services (AKS) cluster to use virtual nodes in the Azure portal

2/25/2020 • 6 minutes to read • [Edit Online](#)

To quickly deploy workloads in an Azure Kubernetes Service (AKS) cluster, you can use virtual nodes. With virtual nodes, you have fast provisioning of pods, and only pay per second for their execution time. In a scaling scenario, you don't need to wait for the Kubernetes cluster autoscaler to deploy VM compute nodes to run the additional pods. Virtual nodes are only supported with Linux pods and nodes.

This article shows you how to create and configure the virtual network resources and an AKS cluster with virtual nodes enabled.

## Before you begin

Virtual nodes enable network communication between pods that run in Azure Container Instances (ACI) and the AKS cluster. To provide this communication, a virtual network subnet is created and delegated permissions are assigned. Virtual nodes only work with AKS clusters created using *advanced* networking. By default, AKS clusters are created with *basic* networking. This article shows you how to create a virtual network and subnets, then deploy an AKS cluster that uses advanced networking.

If you have not previously used ACI, register the service provider with your subscription. You can check the status of the ACI provider registration using the [az provider list](#) command, as shown in the following example:

```
az provider list --query "[?contains(namespace,'Microsoft.ContainerInstance')]" -o table
```

The *Microsoft.ContainerInstance* provider should report as *Registered*, as shown in the following example output:

Namespace	RegistrationState
Microsoft.ContainerInstance	Registered

If the provider shows as *NotRegistered*, register the provider using the [az provider register](#) as shown in the following example:

```
az provider register --namespace Microsoft.ContainerInstance
```

## Regional availability

The following regions are supported for virtual node deployments:

- Australia East (australiaeast)
- Central US (centralus)
- East US (eastus)
- East US 2 (eastus2)
- Japan East (japaneast)
- North Europe (northeurope)
- Southeast Asia (southeastasia)

- West Central US (westcentralus)
- West Europe (westeurope)
- West US (westus)
- West US 2 (westus2)

## Known limitations

Virtual Nodes functionality is heavily dependent on ACI's feature set. The following scenarios are not yet supported with Virtual Nodes

- Using service principal to pull ACR images. [Workaround](#) is to use [Kubernetes secrets](#)
- [Virtual Network Limitations](#) including VNet peering, Kubernetes network policies, and outbound traffic to the internet with network security groups.
- Init containers
- [Host aliases](#)
- [Arguments](#) for exec in ACI
- [DaemonSets](#) will not deploy pods to the virtual node
- [Windows Server nodes \(currently in preview in AKS\)](#) are not supported alongside virtual nodes. You can use virtual nodes to schedule Windows Server containers without the need for Windows Server nodes in an AKS cluster.

## Sign in to Azure

Sign in to the Azure portal at <https://portal.azure.com>.

## Create an AKS cluster

In the top left-hand corner of the Azure portal, select **Create a resource > Kubernetes Service**.

On the **Basics** page, configure the following options:

- **PROJECT DETAILS:** Select an Azure subscription, then select or create an Azure resource group, such as *myResourceGroup*. Enter a **Kubernetes cluster name**, such as *myAKSCluster*.
- **CLUSTER DETAILS:** Select a region, Kubernetes version, and DNS name prefix for the AKS cluster.
- **PRIMARY NODE POOL:** Select a VM size for the AKS nodes. The VM size **cannot** be changed once an AKS cluster has been deployed.
  - Select the number of nodes to deploy into the cluster. For this article, set **Node count** to 1. Node count **can** be adjusted after the cluster has been deployed.

Click **Next: Scale**.

On the **Scale** page, select *Enabled* under **Virtual nodes**.

## Create Kubernetes cluster



Basics **Scale** Authentication Networking Monitoring Tags Review + create

Enable scaling features to allow flexible capacity and burstable scaling options within your cluster.

- **Virtual nodes** allow burstable scaling backed by serverless Azure Container Instances. [Learn more about virtual nodes](#)
- **VM scale sets** are required for a variety of scenarios including autoscaling and multiple node pools [Learn more about VM scale sets in AKS](#)

**Virtual nodes** ⓘ Disabled Enabled

**VM scale sets** ⓘ Disabled Enabled

**i** VM scale sets are required for the following scenarios:  
 \* Autoscaling  
 \* Multiple node pools

**Review + create**

< Previous

Next : Authentication >

By default, an Azure Active Directory service principal is created. This service principal is used for cluster communication and integration with other Azure services.

The cluster is also configured for advanced networking. The virtual nodes are configured to use their own Azure virtual network subnet. This subnet has delegated permissions to connect Azure resources between the AKS cluster. If you don't already have delegated subnet, the Azure portal creates and configures the Azure virtual network and subnet for use with the virtual nodes.

Select **Review + create**. After the validation is complete, select **Create**.

It takes a few minutes to create the AKS cluster and to be ready for use.

## Connect to the cluster

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account. To manage a Kubernetes cluster, use `kubectl`, the Kubernetes command-line client. The `kubectl` client is pre-installed in the Azure Cloud Shell.

To open the Cloud Shell, select **Try it** from the upper right corner of a code block. You can also launch Cloud Shell in a separate browser tab by going to <https://shell.azure.com/bash>. Select **Copy** to copy the blocks of code, paste it into the Cloud Shell, and press enter to run it.

Use the `az aks get-credentials` command to configure `kubectl` to connect to your Kubernetes cluster. The following example gets credentials for the cluster name `myAKSCluster` in the resource group named `myResourceGroup`:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, use the `kubectl get` command to return a list of the cluster nodes.

```
kubectl get nodes
```

The following example output shows the single VM node created and then the virtual node for Linux, `virtual-node-aci-linux`:

```
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
virtual-node-aci-linux	Ready	agent	28m	v1.11.2
aks-agentpool-14693408-0	Ready	agent	32m	v1.11.2

## Deploy a sample app

In the Azure Cloud Shell, create a file named `virtual-node.yaml` and copy in the following YAML. To schedule the container on the node, a `nodeSelector` and `toleration` are defined. These settings allow the pod to be scheduled on the virtual node and confirm that the feature is successfully enabled.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: aci-helloworld
spec:
  replicas: 1
  selector:
    matchLabels:
      app: aci-helloworld
  template:
    metadata:
      labels:
        app: aci-helloworld
    spec:
      containers:
        - name: aci-helloworld
          image: microsoft/aci-helloworld
          ports:
            - containerPort: 80
      nodeSelector:
        kubernetes.io/role: agent
        beta.kubernetes.io/os: linux
        type: virtual-kubelet
      tolerations:
        - key: virtual-kubelet.io/provider
          operator: Exists
        - key: azure.com/aci
          effect: NoSchedule
```

Run the application with the `kubectl apply` command.

```
kubectl apply -f virtual-node.yaml
```

Use the `kubectl get pods` command with the `-o wide` argument to output a list of pods and the scheduled node. Notice that the `virtual-node-helloworld` pod has been scheduled on the `virtual-node-linux` node.

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

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
virtual-node-helloworld-9b55975f-brnmfl	1/1	Running	0	4m	10.241.0.4	virtual-node-aci-linux

The pod is assigned an internal IP address from the Azure virtual network subnet delegated for use with virtual nodes.

#### NOTE

If you use images stored in Azure Container Registry, [configure and use a Kubernetes secret](#). A current limitation of virtual nodes is that you can't use integrated Azure AD service principal authentication. If you don't use a secret, pods scheduled on virtual nodes fail to start and report the error `HTTP response status code 400 error code "InaccessibleImage"`.

## Test the virtual node pod

To test the pod running on the virtual node, browse to the demo application with a web client. As the pod is assigned an internal IP address, you can quickly test this connectivity from another pod on the AKS cluster. Create a test pod and attach a terminal session to it:

```
kubectl run -it --rm virtual-node-test --image=debian
```

Install `curl` in the pod using `apt-get`:

```
apt-get update && apt-get install -y curl
```

Now access the address of your pod using `curl`, such as <http://10.241.0.4>. Provide your own internal IP address shown in the previous `kubectl get pods` command:

```
curl -L http://10.241.0.4
```

The demo application is displayed, as shown in the following condensed example output:

```
$ curl -L 10.241.0.4

<html>
<head>
  <title>Welcome to Azure Container Instances!</title>
</head>
[...]
```

Close the terminal session to your test pod with `exit`. When your session is ended, the pod is deleted.

## Next steps

In this article, a pod was scheduled on the virtual node and assigned a private, internal IP address. You could instead create a service deployment and route traffic to your pod through a load balancer or ingress controller. For more information, see [Create a basic ingress controller in AKS](#).

Virtual nodes are one component of a scaling solution in AKS. For more information on scaling solutions, see the following articles:

- [Use the Kubernetes horizontal pod autoscaler](#)
- [Use the Kubernetes cluster autoscaler](#)
- [Check out the Autoscale sample for Virtual Nodes](#)
- [Read more about the Virtual Kubelet open source library](#)

# Automatically scale a cluster to meet application demands on Azure Kubernetes Service (AKS)

2/25/2020 • 10 minutes to read • [Edit Online](#)

To keep up with application demands in Azure Kubernetes Service (AKS), you may need to adjust the number of nodes that run your workloads. The cluster autoscaler component can watch for pods in your cluster that can't be scheduled because of resource constraints. When issues are detected, the number of nodes in a node pool is increased to meet the application demand. Nodes are also regularly checked for a lack of running pods, with the number of nodes then decreased as needed. This ability to automatically scale up or down the number of nodes in your AKS cluster lets you run an efficient, cost-effective cluster.

This article shows you how to enable and manage the cluster autoscaler in an AKS cluster.

## Before you begin

This article requires that you are running the Azure CLI version 2.0.76 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Limitations

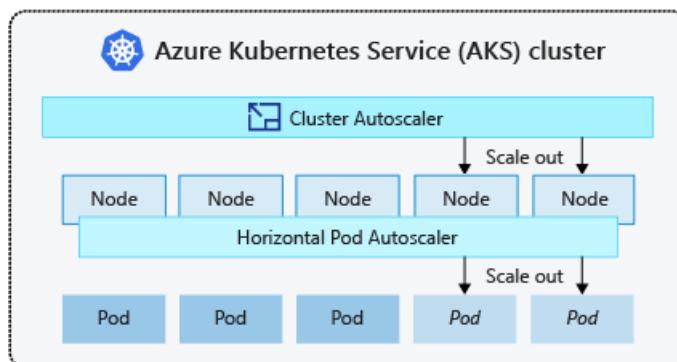
The following limitations apply when you create and manage AKS clusters that use the cluster autoscaler:

- The HTTP application routing add-on can't be used.

## About the cluster autoscaler

To adjust to changing application demands, such as between the workday and evening or on a weekend, clusters often need a way to automatically scale. AKS clusters can scale in one of two ways:

- 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.
- The **horizontal pod autoscaler** uses the Metrics Server in a Kubernetes cluster to monitor the resource demand of pods. If an application needs more resources, the number of pods is automatically increased to meet the demand.



Both the horizontal pod autoscaler and cluster autoscaler can also decrease the number of pods and nodes as needed. The cluster autoscaler decreases the number of nodes when there has been unused capacity for a period of time. Pods on a node to be removed by the cluster autoscaler are safely scheduled elsewhere in the cluster. The cluster autoscaler may be unable to scale down if pods can't move, such as in the following situations:

- A pod is directly created and isn't backed by a controller object, such as a deployment or replica set.
- A pod disruption budget (PDB) is too restrictive and doesn't allow the number of pods to be fall below a certain threshold.
- A pod uses node selectors or anti-affinity that can't be honored if scheduled on a different node.

For more information about how the cluster autoscaler may be unable to scale down, see [What types of pods can prevent the cluster autoscaler from removing a node?](#)

The cluster autoscaler uses startup parameters for things like time intervals between scale events and resource thresholds. These parameters are defined by the Azure platform, and aren't currently exposed for you to adjust. For more information on what parameters the cluster autoscaler uses, see [What are the cluster autoscaler parameters?](#).

The cluster and horizontal pod autoscalers can work together, and are often both deployed in a cluster. When combined, the horizontal pod autoscaler is focused on running the number of pods required to meet application demand. The cluster autoscaler is focused on running the number of nodes required to support the scheduled pods.

#### **NOTE**

Manual scaling is disabled when you use the cluster autoscaler. Let the cluster autoscaler determine the required number of nodes. If you want to manually scale your cluster, [disable the cluster autoscaler](#).

## Create an AKS cluster and enable the cluster autoscaler

If you need to create an AKS cluster, use the `az aks create` command. To enable and configure the cluster autoscaler on the node pool for the cluster, use the `--enable-cluster-autoscaler` parameter, and specify a node `--min-count` and `--max-count`.

#### **IMPORTANT**

The cluster autoscaler is a Kubernetes component. Although the AKS cluster uses a virtual machine scale set for the nodes, don't manually enable or edit settings for scale set autoscale in the Azure portal or using the Azure CLI. Let the Kubernetes cluster autoscaler manage the required scale settings. For more information, see [Can I modify the AKS resources in the node resource group?](#)

The following example creates an AKS cluster with a single node pool backed by a virtual machine scale set. It also enables the cluster autoscaler on the node pool for the cluster and sets a minimum of 1 and maximum of 3 nodes:

```
# First create a resource group
az group create --name myResourceGroup --location eastus

# Now create the AKS cluster and enable the cluster autoscaler
az aks create \
  --resource-group myResourceGroup \
  --name myAKSCluster \
  --node-count 1 \
  --vm-set-type VirtualMachineScaleSets \
  --load-balancer-sku standard \
  --enable-cluster-autoscaler \
  --min-count 1 \
  --max-count 3
```

It takes a few minutes to create the cluster and configure the cluster autoscaler settings.

# Change the cluster autoscaler settings

## IMPORTANT

If you have multiple node pools in your AKS cluster, skip to the [autoscale with multiple agent pools section](#). Clusters with multiple agent pools require use of the `az aks nodepool` command set to change node pool specific properties instead of `az aks`.

In the previous step to create an AKS cluster or update an existing node pool, the cluster autoscaler minimum node count was set to 1, and the maximum node count was set to 3. As your application demands change, you may need to adjust the cluster autoscaler node count.

To change the node count, use the [az aks update](#) command.

```
az aks update \
--resource-group myResourceGroup \
--name myAKScluster \
--update-cluster-autoscaler \
--min-count 1 \
--max-count 5
```

The above example updates cluster autoscaler on the single node pool in *myAKScluster* to a minimum of 1 and maximum of 5 nodes.

## NOTE

You can't set a higher minimum node count than is currently set for the node pool. For example, if you currently have min count set to 1, you can't update the min count to 3.

Monitor the performance of your applications and services, and adjust the cluster autoscaler node counts to match the required performance.

## Using the autoscaler profile

You can also configure more granular details of the cluster autoscaler by changing the default values in the cluster-wide autoscaler profile. For example, a scale down event happens after nodes are under-utilized after 10 minutes. If you had workloads that ran every 15 minutes, you may want to change the autoscaler profile to scale down under utilized nodes after 15 or 20 minutes. When you enable the cluster autoscaler, a default profile is used unless you specify different settings. The cluster autoscaler profile has the following settings that you can update:

SETTING	DESCRIPTION	DEFAULT VALUE
scan-interval	How often cluster is reevaluated for scale up or down	10 seconds
scale-down-delay-after-add	How long after scale up that scale down evaluation resumes	10 minutes
scale-down-delay-after-delete	How long after node deletion that scale down evaluation resumes	scan-interval
scale-down-delay-after-failure	How long after scale down failure that scale down evaluation resumes	3 minutes

Setting	Description	Default Value
scale-down-unneeded-time	How long a node should be unneeded before it is eligible for scale down	10 minutes
scale-down-unready-time	How long an unready node should be unneeded before it is eligible for scale down	20 minutes
scale-down-utilization-threshold	Node utilization level, defined as sum of requested resources divided by capacity, below which a node can be considered for scale down	0.5
max-graceful-termination-sec	Maximum number of seconds the cluster autoscaler waits for pod termination when trying to scale down a node.	600 seconds

#### IMPORTANT

The cluster autoscaler profile affects all node pools that use the cluster autoscaler. You can't set an autoscaler profile per node pool.

### Install `aks-preview` CLI extension

To set the cluster autoscaler settings profile, you need the `aks-preview` CLI extension version 0.4.30 or higher. Install the `aks-preview` Azure CLI extension using the [az extension add](#) command, then check for any available updates using the [az extension update](#) command:

```
# Install the aks-preview extension
az extension add --name aks-preview

# Update the extension to make sure you have the latest version installed
az extension update --name aks-preview
```

### Set the cluster autoscaler profile on an existing AKS cluster

Use the [az aks update](#) command with the `cluster-autoscaler-profile` parameter to set the cluster autoscaler profile on your cluster. The following example configures the scan interval setting as 30s in the profile.

```
az aks update \
--resource-group myResourceGroup \
--name myAKSCluster \
--cluster-autoscaler-profile scan-interval=30s
```

When you enable the cluster autoscaler on node pools in the cluster, those clusters will also use the cluster autoscaler profile. For example:

```
az aks nodepool update \
--resource-group myResourceGroup \
--cluster-name myAKSCluster \
--name mynodepool \
--enable-cluster-autoscaler \
--min-count 1 \
--max-count 3
```

## IMPORTANT

When you set the cluster autoscaler profile, any existing node pools with the cluster autoscaler enabled will start using the profile immediately.

## Set the cluster autoscaler profile when creating an AKS cluster

You can also use the *cluster-autoscaler-profile* parameter when you create your cluster. For example:

```
az aks create \  
  --resource-group myResourceGroup \  
  --name myAKSCluster \  
  --node-count 1 \  
  --enable-cluster-autoscaler \  
  --min-count 1 \  
  --max-count 3 \  
  --cluster-autoscaler-profile scan-interval=30s
```

The above command creates an AKS cluster and defines the scan interval as 30 seconds for the cluster-wide autoscaler profile. The command also enables the cluster autoscaler on the initial node pool, sets the minimum node count to 1 and the maximum node count to 3.

## Reset cluster autoscaler profile to default values

Use the [az aks update](#) command to reset the cluster autoscaler profile on your cluster.

```
az aks update \  
  --resource-group myResourceGroup \  
  --name myAKSCluster \  
  --cluster-autoscaler-profile ""
```

## Disable the cluster autoscaler

If you no longer wish to use the cluster autoscaler, you can disable it using the [az aks update](#) command, specifying the *--disable-cluster-autoscaler* parameter. Nodes aren't removed when the cluster autoscaler is disabled.

```
az aks update \  
  --resource-group myResourceGroup \  
  --name myAKSCluster \  
  --disable-cluster-autoscaler
```

You can manually scale your cluster after disabling the cluster autoscaler by using the [az aks scale](#) command. If you use the horizontal pod autoscaler, that feature continues to run with the cluster autoscaler disabled, but pods may end up unable to be scheduled if all node resources are in use.

## Re-enable a disabled cluster autoscaler

If you wish to re-enable the cluster autoscaler on an existing cluster, you can re-enable it using the [az aks update](#) command, specifying the *--enable-cluster-autoscaler*, *--min-count*, and *--max-count* parameters.

## Retrieve cluster autoscaler logs and status

To diagnose and debug autoscaler events, logs and status can be retrieved from the autoscaler add-on.

AKS manages the cluster autoscaler on your behalf and runs it in the managed control plane. Master node logs must be configured to be viewed as a result.

To configure logs to be pushed from the cluster autoscaler into Log Analytics, follow these steps.

1. Set up a rule for diagnostic logs to push cluster-autoscaler logs to Log Analytics. [Instructions are detailed here](#), ensure you check the box for `cluster-autoscaler` when selecting options for "Logs".
2. Click on the "Logs" section on your cluster via the Azure portal.
3. Input the following example query into Log Analytics:

```
AzureDiagnostics  
| where Category == "cluster-autoscaler"
```

You should see logs similar to the following example as long as there are logs to retrieve.

TimeGenerated [UTC]	OperationName	Category	ccpNamespace_s
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8
11/7/2019, 9:35:00.000 PM	Microsoft.ContainerService/managedClusters/diagnosticLogs/Read	cluster-autoscaler	5d9540c357069d00016b13d8

The cluster autoscaler will also write out health status to a configmap named `cluster-autoscaler-status`. To retrieve these logs, execute the following `kubectl` command. A health status will be reported for each node pool configured with the cluster autoscaler.

```
kubectl get configmap -n kube-system cluster-autoscaler-status -o yaml
```

To learn more about what is logged from the autoscaler, read the FAQ on the [Kubernetes/autoscaler GitHub project](#).

## Use the cluster autoscaler with multiple node pools enabled

The cluster autoscaler can be used together with [multiple node pools](#) enabled. Follow that document to learn how to enable multiple node pools and add additional node pools to an existing cluster. When using both features together, you enable the cluster autoscaler on each individual node pool in the cluster and can pass unique autoscaling rules to each.

The below command assumes you followed the [initial instructions](#) earlier in this document and you want to update an existing node pool's max-count from 3 to 5. Use the `az aks nodepool update` command to update an existing node pool's settings.

```
az aks nodepool update \
--resource-group myResourceGroup \
--cluster-name myAKSCluster \
--name nodepool1 \
--update-cluster-autoscaler \
--min-count 1 \
--max-count 5
```

The cluster autoscaler can be disabled with `az aks nodepool update` and passing the `--disable-cluster-autoscaler` parameter.

```
az aks nodepool update \
--resource-group myResourceGroup \
--cluster-name myAKSCluster \
--name nodepool1 \
--disable-cluster-autoscaler
```

If you wish to re-enable the cluster autoscaler on an existing cluster, you can re-enable it using the `az aks nodepool update` command, specifying the `--enable-cluster-autoscaler`, `--min-count`, and `--max-count` parameters.

## Next steps

This article showed you how to automatically scale the number of AKS nodes. You can also use the horizontal pod autoscaler to automatically adjust the number of pods that run your application. For steps on using the horizontal pod autoscaler, see [Scale applications in AKS](#).

# Create an Azure Kubernetes Service (AKS) cluster that uses availability zones

2/25/2020 • 6 minutes to read • [Edit Online](#)

An Azure Kubernetes Service (AKS) cluster distributes resources such as the nodes and storage across logical sections of the underlying Azure compute infrastructure. This deployment model makes sure that the nodes run across separate update and fault domains in a single Azure datacenter. AKS clusters deployed with this default behavior provide a high level of availability to protect against a hardware failure or planned maintenance event.

To provide a higher level of availability to your applications, AKS clusters can be distributed across availability zones. These zones are physically separate datacenters within a given region. When the cluster components are distributed across multiple zones, your AKS cluster is able to tolerate a failure in one of those zones. Your applications and management operations continue to be available even if one entire datacenter has a problem.

This article shows you how to create an AKS cluster and distribute the node components across availability zones.

## Before you begin

You need the Azure CLI version 2.0.76 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Limitations and region availability

AKS clusters can currently be created using availability zones in the following regions:

- Central US
- East US 2
- East US
- France Central
- Japan East
- North Europe
- Southeast Asia
- UK South
- West Europe
- West US 2

The following limitations apply when you create an AKS cluster using availability zones:

- You can only enable availability zones when the cluster is created.
- Availability zone settings can't be updated after the cluster is created. You also can't update an existing, non-availability zone cluster to use availability zones.
- You can't disable availability zones for an AKS cluster once it has been created.
- The node size (VM SKU) selected must be available across all availability zones.
- Clusters with availability zones enabled require use of Azure Standard Load Balancers for distribution across zones.
- You must use Kubernetes version 1.13.5 or greater in order to deploy Standard Load Balancers.

AKS clusters that use availability zones must use the Azure load balancer *standard* SKU, which is the default value for the load balancer type. This load balancer type can only be defined at cluster create time. For more information

and the limitations of the standard load balancer, see [Azure load balancer standard SKU limitations](#).

### Azure disks limitations

Volumes that use Azure managed disks are currently not zonal resources. Pods rescheduled in a different zone from their original zone can't reattach their previous disk(s). It's recommended to run stateless workloads that don't require persistent storage that may come across zonal issues.

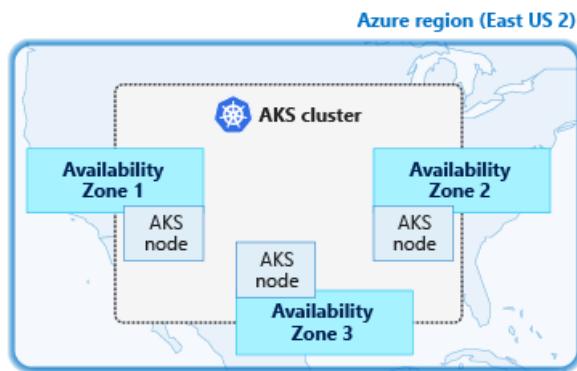
If you must run stateful workloads, use taints and tolerations in your pod specs to tell the Kubernetes scheduler to create pods in the same zone as your disks. Alternatively, use network-based storage such as Azure Files that can attach to pods as they're scheduled between zones.

## Overview of availability zones for AKS clusters

Availability zones is a high-availability offering that protects your applications and data from datacenter failures. Zones are unique physical locations within an Azure region. Each zone is made up of one or more datacenters equipped with independent power, cooling, and networking. To ensure resiliency, there's a minimum of three separate zones in all enabled regions. The physical separation of availability zones within a region protects applications and data from datacenter failures. Zone-redundant services replicate your applications and data across availability zones to protect from single-points-of-failure.

For more information, see [What are availability zones in Azure?](#).

AKS clusters that are deployed using availability zones can distribute nodes across multiple zones within a region. For example, a cluster in the *East US 2* region can create nodes in all three availability zones in *East US 2*. This distribution of AKS cluster resources improves cluster availability as they're resilient to failure of a specific zone.



In a zone outage, the nodes can be rebalanced manually or using the `cluster autoscaler`. If a single zone becomes unavailable, your applications continue to run.

## Create an AKS cluster across availability zones

When you create a cluster using the `az aks create` command, the `--zones` parameter defines which zones agent nodes are deployed into. The AKS control plane components for your cluster are also spread across zones in the highest available configuration when you define the `--zones` parameter at cluster creation time.

If you don't define any zones for the default agent pool when you create an AKS cluster, the AKS control plane components for your cluster will not use availability zones. You can add additional node pools using the `az aks nodepool add` command and specify `--zones` for those new nodes, however the control plane components remain without availability zone awareness. You can't change the zone awareness for a node pool or the AKS control plane components once they're deployed.

The following example creates an AKS cluster named *myAKSCluster* in the resource group named *myResourceGroup*. A total of 3 nodes are created - one agent in zone 1, one in 2, and then one in 3. The AKS control plane components are also distributed across zones in the highest available configuration since they're

defined as part of the cluster create process.

```
az group create --name myResourceGroup --location eastus2

az aks create \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --generate-ssh-keys \
    --vm-set-type VirtualMachineScaleSets \
    --load-balancer-sku standard \
    --node-count 3 \
    --zones 1 2 3
```

It takes a few minutes to create the AKS cluster.

## Verify node distribution across zones

When the cluster is ready, list the agent nodes in the scale set to see what availability zone they're deployed in.

First, get the AKS cluster credentials using the [az aks get-credentials](#) command:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

Next, use the [kubectl describe](#) command to list the nodes in the cluster. Filter on the *failure-domain.beta.kubernetes.io/zone* value as shown in the following example:

```
kubectl describe nodes | grep -e "Name:" -e "failure-domain.beta.kubernetes.io/zone"
```

The following example output shows the three nodes distributed across the specified region and availability zones, such as *eastus2-1* for the first availability zone and *eastus2-2* for the second availability zone:

```
Name:      aks-nodepool1-28993262-vmss000000
           failure-domain.beta.kubernetes.io/zone=eastus2-1
Name:      aks-nodepool1-28993262-vmss000001
           failure-domain.beta.kubernetes.io/zone=eastus2-2
Name:      aks-nodepool1-28993262-vmss000002
           failure-domain.beta.kubernetes.io/zone=eastus2-3
```

As you add additional nodes to an agent pool, the Azure platform automatically distributes the underlying VMs across the specified availability zones.

Note that in newer Kubernetes versions (1.17.0 and later), AKS is using the newer label

`topology.kubernetes.io/zone` in addition to the deprecated `failure-domain.beta.kubernetes.io/zone`.

## Verify pod distribution across zones

As documented in [Well-Known Labels, Annotations and Taints](#), Kubernetes uses the

`failure-domain.beta.kubernetes.io/zone` label to automatically distribute pods in a replication controller or service across the different zones available. In order to test this, you can scale up your cluster from 3 to 5 nodes, to verify correct pod spreading:

```
az aks scale \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --node-count 5
```

When the scale operation completes after a few minutes, the command

```
kubectl describe nodes | grep -e "Name:" -e "failure-domain.beta.kubernetes.io/zone"
```

 should give an output similar to this sample:

```
Name: aks-nodepool1-28993262-vmss000000  
failure-domain.beta.kubernetes.io/zone=eastus2-1  
Name: aks-nodepool1-28993262-vmss000001  
failure-domain.beta.kubernetes.io/zone=eastus2-2  
Name: aks-nodepool1-28993262-vmss000002  
failure-domain.beta.kubernetes.io/zone=eastus2-3  
Name: aks-nodepool1-28993262-vmss000003  
failure-domain.beta.kubernetes.io/zone=eastus2-1  
Name: aks-nodepool1-28993262-vmss000004  
failure-domain.beta.kubernetes.io/zone=eastus2-2
```

As you can see, we now have two additional nodes in zones 1 and 2. You can deploy an application consisting of three replicas. We will use NGINX as example:

```
kubectl run nginx --image=nginx --replicas=3
```

If you verify that nodes where your pods are running, you will see that the pods are running on the pods corresponding to three different availability zones. For example with the command

```
kubectl describe pod | grep -e "Name:" -e "Node:"
```

 you would get an output similar to this:

```
Name: nginx-6db489d4b7-ktdwg  
Node: aks-nodepool1-28993262-vmss000000/10.240.0.4  
Name: nginx-6db489d4b7-v7zvj  
Node: aks-nodepool1-28993262-vmss000002/10.240.0.6  
Name: nginx-6db489d4b7-xz6wj  
Node: aks-nodepool1-28993262-vmss000004/10.240.0.8
```

As you can see from the previous output, the first pod is running on node 0, which is located in the availability zone `eastus2-1`. The second pod is running on node 2, which corresponds to `eastus2-3`, and the third one in node 4, in `eastus2-2`. Without any additional configuration, Kubernetes is spreading the pods correctly across all three availability zones.

## Next steps

This article detailed how to create an AKS cluster that uses availability zones. For more considerations on highly available clusters, see [Best practices for business continuity and disaster recovery in AKS](#).

# Create and manage multiple node pools for a cluster in Azure Kubernetes Service (AKS)

2/25/2020 • 19 minutes to read • [Edit Online](#)

In Azure Kubernetes Service (AKS), nodes of the same configuration are grouped together into *node pools*. These node pools contain the underlying VMs that run your applications. The initial number of nodes and their size (SKU) are defined when you create an AKS cluster, which creates a *default node pool*. To support applications that have different compute or storage demands, you can create additional node pools. For example, use these additional node pools to provide GPUs for compute-intensive applications, or access to high-performance SSD storage.

## NOTE

This feature enables higher control over how to create and manage multiple node pools. As a result, separate commands are required for create/update/delete. Previously cluster operations through `az aks create` or `az aks update` used the `managedCluster` API and were the only option to change your control plane and a single node pool. This feature exposes a separate operation set for agent pools through the `agentPool` API and require use of the `az aks nodepool` command set to execute operations on an individual node pool.

This article shows you how to create and manage multiple node pools in an AKS cluster.

## Before you begin

You need the Azure CLI version 2.0.76 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Limitations

The following limitations apply when you create and manage AKS clusters that support multiple node pools:

- See [Quotas, virtual machine size restrictions, and region availability in Azure Kubernetes Service \(AKS\)](#).
- You can't delete the system node pool, by default the first node pool.
- The AKS cluster must use the Standard SKU load balancer to use multiple node pools, the feature is not supported with Basic SKU load balancers.
- The AKS cluster must use virtual machine scale sets for the nodes.
- The name of a node pool may only contain lowercase alphanumeric characters and must begin with a lowercase letter. For Linux node pools the length must be between 1 and 12 characters, for Windows node pools the length must be between 1 and 6 characters.
- All node pools must reside in the same vnet and subnet.
- When creating multiple node pools at cluster create time, all Kubernetes versions used by node pools must match the version set for the control plane. This can be updated after the cluster has been provisioned by using per node pool operations.

## Create an AKS cluster

To get started, create an AKS cluster with a single node pool. The following example uses the `az group create` command to create a resource group named *myResourceGroup* in the *eastus* region. An AKS cluster named *myAKSCluster* is then created using the `az aks create` command. A `--kubernetes-version` of *1.15.7* is used to show

how to update a node pool in a following step. You can specify any [supported Kubernetes version](#).

#### NOTE

The *Basic* load balancer SKU is **not supported** when using multiple node pools. By default, AKS clusters are created with the *Standard* load balancer SKU from Azure CLI and Azure portal.

```
# Create a resource group in East US
az group create --name myResourceGroup --location eastus

# Create a basic single-node AKS cluster
az aks create \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --vm-set-type VirtualMachineScaleSets \
    --node-count 2 \
    --generate-ssh-keys \
    --kubernetes-version 1.15.7 \
    --load-balancer-sku standard
```

It takes a few minutes to create the cluster.

#### NOTE

To ensure your cluster operates reliably, you should run at least 2 (two) nodes in the default node pool, as essential system services are running across this node pool.

When the cluster is ready, use the [az aks get-credentials](#) command to get the cluster credentials for use with

```
kubectl :
```

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

## Add a node pool

The cluster created in the previous step has a single node pool. Let's add a second node pool using the [az aks nodepool add](#) command. The following example creates a node pool named *mynodepool* that runs 3 nodes:

```
az aks nodepool add \
    --resource-group myResourceGroup \
    --cluster-name myAKSCluster \
    --name mynodepool \
    --node-count 3 \
    --kubernetes-version 1.15.5
```

#### NOTE

The name of a node pool must start with a lowercase letter and can only contain alphanumeric characters. For Linux node pools the length must be between 1 and 12 characters, for Windows node pools the length must be between 1 and 6 characters.

To see the status of your node pools, use the [az aks node pool list](#) command and specify your resource group and cluster name:

```
az aks nodepool list --resource-group myResourceGroup --cluster-name myAKSCluster
```

The following example output shows that *mynodepool* has been successfully created with three nodes in the node pool. When the AKS cluster was created in the previous step, a default *nodepool1* was created with a node count of 2.

```
$ az aks nodepool list --resource-group myResourceGroup --cluster-name myAKSCluster

[
  {
    ...
    "count": 3,
    ...
    "name": "mynodepool",
    "orchestratorVersion": "1.15.5",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  },
  {
    ...
    "count": 2,
    ...
    "name": "nodepool1",
    "orchestratorVersion": "1.15.7",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  }
]
```

#### TIP

If no *VmSize* is specified when you add a node pool, the default size is *Standard\_DS2\_v3* for Windows node pools and *Standard\_DS2\_v2* for Linux node pools. If no *OrchestratorVersion* is specified, it defaults to the same version as the control plane.

## Upgrade a node pool

#### NOTE

Upgrade and scale operations on a cluster or node pool cannot occur simultaneously, if attempted an error is returned. Instead, each operation type must complete on the target resource prior to the next request on that same resource. Read more about this on our [troubleshooting guide](#).

When your AKS cluster was initially created in the first step, a `--kubernetes-version` of 1.15.7 was specified. This set the Kubernetes version for both the control plane and the default node pool. The commands in this section explain how to upgrade a single specific node pool.

The relationship between upgrading the Kubernetes version of the control plane and the node pool are explained in the [section below](#).

#### NOTE

The node pool OS image version is tied to the Kubernetes version of the cluster. You will only get OS image upgrades, following a cluster upgrade.

Since there are two node pools in this example, we must use [az aks nodepool upgrade](#) to upgrade a node pool. Let's upgrade the *mynodepool* to Kubernetes 1.15.7. Use the [az aks nodepool upgrade](#) command to upgrade the node pool, as shown in the following example:

```
az aks nodepool upgrade \
    --resource-group myResourceGroup \
    --cluster-name myAKSCluster \
    --name mynodepool \
    --kubernetes-version 1.15.7 \
    --no-wait
```

List the status of your node pools again using the [az aks node pool list](#) command. The following example shows that *mynodepool* is in the *Upgrading* state to 1.15.7:

```
$ az aks nodepool list -g myResourceGroup --cluster-name myAKSCluster

[
  {
    ...
    "count": 3,
    ...
    "name": "mynodepool",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Upgrading",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  },
  {
    ...
    "count": 2,
    ...
    "name": "nodepool1",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Succeeded",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  }
]
```

It takes a few minutes to upgrade the nodes to the specified version.

As a best practice, you should upgrade all node pools in an AKS cluster to the same Kubernetes version. The default behavior of [az aks upgrade](#) is to upgrade all node pools together with the control plane to achieve this alignment. The ability to upgrade individual node pools lets you perform a rolling upgrade and schedule pods between node pools to maintain application uptime within the above constraints mentioned.

## Upgrade a cluster control plane with multiple node pools

#### NOTE

Kubernetes uses the standard [Semantic Versioning](#) versioning scheme. The version number is expressed as x.y.z, where x is the major version, y is the minor version, and z is the patch version. For example, in version 1.12.6, 1 is the major version, 12 is the minor version, and 6 is the patch version. The Kubernetes version of the control plane and the initial node pool are set during cluster creation. All additional node pools have their Kubernetes version set when they are added to the cluster. The Kubernetes versions may differ between node pools as well as between a node pool and the control plane.

An AKS cluster has two cluster resource objects with Kubernetes versions associated.

1. A cluster control plane Kubernetes version.
2. A node pool with a Kubernetes version.

A control plane maps to one or many node pools. The behavior of an upgrade operation depends on which Azure CLI command is used.

Upgrading an AKS control plane requires using `az aks upgrade`. This upgrades the control plane version and all node pools in the cluster.

Issuing the `az aks upgrade` command with the `--control-plane-only` flag upgrades only the cluster control plane. None of the associated node pools in the cluster are changed.

Upgrading individual node pools requires using `az aks nodepool upgrade`. This upgrades only the target node pool with the specified Kubernetes version.

#### Validation rules for upgrades

The valid Kubernetes upgrades for a cluster's control plane and node pools are validated by the following sets of rules.

- Rules for valid versions to upgrade node pools:
  - The node pool version must have the same *major* version as the control plane.
  - The node pool *minor* version must be within two *minor* versions of the control plane version.
  - The node pool version cannot be greater than the control `major.minor.patch` version.
- Rules for submitting an upgrade operation:
  - You cannot downgrade the control plane or a node pool Kubernetes version.
  - If a node pool Kubernetes version is not specified, behavior depends on the client being used.  
Declaration in Resource Manager templates fall back to the existing version defined for the node pool if used, if none is set the control plane version is used to fall back on.
  - You can either upgrade or scale a control plane or a node pool at a given time, you cannot submit multiple operations on a single control plane or node pool resource simultaneously.

## Scale a node pool manually

As your application workload demands change, you may need to scale the number of nodes in a node pool. The number of nodes can be scaled up or down.

To scale the number of nodes in a node pool, use the `az aks node pool scale` command. The following example scales the number of nodes in *mynodepool* to 5:

```
az aks nodepool scale \
    --resource-group myResourceGroup \
    --cluster-name myAKSCluster \
    --name mynodepool \
    --node-count 5 \
    --no-wait
```

List the status of your node pools again using the [az aks node pool list](#) command. The following example shows that *mynodepool* is in the *Scaling* state with a new count of 5 nodes:

```
$ az aks nodepool list -g myResourceGroup --cluster-name myAKSCluster

[
  {
    ...
    "count": 5,
    ...
    "name": "mynodepool",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Scaling",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  },
  {
    ...
    "count": 2,
    ...
    "name": "nodepool1",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Succeeded",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  }
]
```

It takes a few minutes for the scale operation to complete.

## Scale a specific node pool automatically by enabling the cluster autoscaler

AKS offers a separate feature to automatically scale node pools with a feature called the [cluster autoscaler](#). This feature can be enabled per node pool with unique minimum and maximum scale counts per node pool. Learn how to [use the cluster autoscaler per node pool](#).

## Delete a node pool

If you no longer need a pool, you can delete it and remove the underlying VM nodes. To delete a node pool, use the [az aks node pool delete](#) command and specify the node pool name. The following example deletes the *mynodepool* created in the previous steps:

#### Caution

There are no recovery options for data loss that may occur when you delete a node pool. If pods can't be scheduled on other node pools, those applications are unavailable. Make sure you don't delete a node pool when in-use applications don't have data backups or the ability to run on other node pools in your cluster.

```
az aks nodepool delete -g myResourceGroup --cluster-name myAKSCluster --name mynodepool --no-wait
```

The following example output from the [az aks node pool list](#) command shows that *mynodepool* is in the *Deleting* state:

```
$ az aks nodepool list -g myResourceGroup --cluster-name myAKSCluster

[
  {
    ...
    "count": 5,
    ...
    "name": "mynodepool",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Deleting",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  },
  {
    ...
    "count": 2,
    ...
    "name": "nodepool1",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Succeeded",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  }
]
```

It takes a few minutes to delete the nodes and the node pool.

## Specify a VM size for a node pool

In the previous examples to create a node pool, a default VM size was used for the nodes created in the cluster. A more common scenario is for you to create node pools with different VM sizes and capabilities. For example, you may create a node pool that contains nodes with large amounts of CPU or memory, or a node pool that provides GPU support. In the next step, you [use taints and tolerations](#) to tell the Kubernetes scheduler how to limit access to pods that can run on these nodes.

In the following example, create a GPU-based node pool that uses the *Standard\_NC6* VM size. These VMs are powered by the NVIDIA Tesla K80 card. For information on available VM sizes, see [Sizes for Linux virtual machines in Azure](#).

Create a node pool using the [az aks node pool add](#) command again. This time, specify the name *gpunodepool*, and use the `--node-vm-size` parameter to specify the *Standard\_NC6* size:

```
az aks nodepool add \
  --resource-group myResourceGroup \
  --cluster-name myAKSCluster \
  --name gpunodepool \
  --node-count 1 \
  --node-vm-size Standard_NC6 \
  --no-wait
```

The following example output from the `az aks node pool list` command shows that `gpunodepool` is *Creating* nodes with the specified `VmSize`:

```
$ az aks nodepool list -g myResourceGroup --cluster-name myAKScluster

[
  {
    ...
    "count": 1,
    ...
    "name": "gpunodepool",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Creating",
    ...
    "vmSize": "Standard_NC6",
    ...
  },
  {
    ...
    "count": 2,
    ...
    "name": "nodepool1",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Succeeded",
    ...
    "vmSize": "Standard_DS2_v2",
    ...
  }
]
```

It takes a few minutes for the `gpunodepool` to be successfully created.

## Schedule pods using taints and tolerations

You now have two node pools in your cluster - the default node pool initially created, and the GPU-based node pool. Use the `kubectl get nodes` command to view the nodes in your cluster. The following example output shows the nodes:

```
$ kubectl get nodes

NAME                      STATUS   ROLES      AGE      VERSION
aks-gpunodepool-28993262-vmss000000  Ready    agent     4m22s   v1.15.7
aks-nodepool1-28993262-vmss000000    Ready    agent     115m    v1.15.7
```

The Kubernetes scheduler can use taints and tolerations to restrict what workloads can run on nodes.

- A **taint** is applied to a node that indicates only specific pods can be scheduled on them.
- A **toleration** is then applied to a pod that allows them to *tolerate* a node's taint.

For more information on how to use advanced Kubernetes scheduled features, see [Best practices for advanced scheduler features in AKS](#)

In this example, apply a taint to your GPU-based node using the `--node-taints` command. Specify the name of your GPU-based node from the output of the previous `kubectl get nodes` command. The taint is applied as a `key:value` and then a scheduling option. The following example uses the `sku=gpu` pair and defines pods otherwise have the `NoSchedule` ability:

```
az aks nodepool add --node-taints aks-gpunodepool-28993262-vmss000000 sku=gpu:NoSchedule
```

The following basic example YAML manifest uses a toleration to allow the Kubernetes scheduler to run an NGINX pod on the GPU-based node. For a more appropriate, but time-intensive example to run a Tensorflow job against the MNIST dataset, see [Use GPUs for compute-intensive workloads on AKS](#).

Create a file named `gpu-toleration.yaml` and copy in the following example YAML:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - image: nginx:1.15.9
    name: mypod
    resources:
      requests:
        cpu: 100m
        memory: 128Mi
      limits:
        cpu: 1
        memory: 2G
  tolerations:
  - key: "sku"
    operator: "Equal"
    value: "gpu"
    effect: "NoSchedule"
```

Schedule the pod using the `kubectl apply -f gpu-toleration.yaml` command:

```
kubectl apply -f gpu-toleration.yaml
```

It takes a few seconds to schedule the pod and pull the NGINX image. Use the `kubectl describe pod` command to view the pod status. The following condensed example output shows the `sku=gpu:NoSchedule` toleration is applied. In the events section, the scheduler has assigned the pod to the `aks-gpunodepool-28993262-vmss000000` GPU-based node:

```
$ kubectl describe pod mypod

[...]
Tolerations:    node.kubernetes.io/not-ready:NoExecute for 300s
                node.kubernetes.io/unreachable:NoExecute for 300s
                sku=gpu:NoSchedule
Events:
  Type      Reason     Age      From               Message
  ----      ----     --      ----               -----
  Normal    Scheduled  4m48s   default-scheduler   successfully assigned default/mypod
  to aks-gpunodepool-28993262-vmss000000
  Normal    Pulling    4m47s   kubelet, aks-gpunodepool-28993262-vmss000000  pulling image "nginx:1.15.9"
  Normal    Pulled     4m43s   kubelet, aks-gpunodepool-28993262-vmss000000  successfully pulled image
  "nginx:1.15.9"
  Normal    Created    4m40s   kubelet, aks-gpunodepool-28993262-vmss000000  created container
  Normal    Started    4m40s   kubelet, aks-gpunodepool-28993262-vmss000000  started container
```

Only pods that have this taint applied can be scheduled on nodes in `gpunodepool`. Any other pod would be scheduled in the `nodepool1` node pool. If you create additional node pools, you can use additional taints and tolerations to limit what pods can be scheduled on those node resources.

# Specify a tag for a node pool

You can apply an Azure tag to node pools in your AKS cluster. Tags applied to a node pool are applied to each node within the node pool and are persisted through upgrades. Tags are also applied to new nodes added to a node pool during scale out operations. Adding a tag can help with tasks such as policy tracking or cost estimation.

## IMPORTANT

To use node pool tags, you need the `aks-preview` CLI extension version 0.4.29 or higher. Install the `aks-preview` Azure CLI extension using the [az extension add](#) command, then check for any available updates using the [az extension update](#) command:

```
# Install the aks-preview extension
az extension add --name aks-preview

# Update the extension to make sure you have the latest version installed
az extension update --name aks-preview
```

Create a node pool using the [az aks node pool add](#). Specify the name `tagnodepool` and use the `--tag` parameter to specify `dept=IT` and `costcenter=9999` for tags.

```
az aks nodepool add \
--resource-group myResourceGroup \
--cluster-name myAKSCluster \
--name tagnodepool \
--node-count 1 \
--tags dept=IT costcenter=9999 \
--no-wait
```

## NOTE

You can also use the `--tags` parameter when using [az aks nodepool update](#) command as well as during cluster creation. During cluster creation, the `--tags` parameter applies the tag to the initial node pool created with the cluster. All tag names must adhere to the limitations in [Use tags to organize your Azure resources](#). Updating a node pool with the `--tags` parameter updates any existing tag values and appends any new tags. For example, if your node pool had `dept=IT` and `costcenter=9999` for tags and you updated it with `team=dev` and `costcenter=111` for tags, your nodepool would have `dept=IT`, `costcenter=111`, and `team=dev` for tags.

The following example output from the [az aks nodepool list](#) command shows that `tagnodepool` is *Creating* nodes with the specified `tag`:

```
$ az aks nodepool list -g myResourceGroup --cluster-name myAKScluster

[
  {
    ...
    "count": 1,
    ...
    "name": "tagnodepool",
    "orchestratorVersion": "1.15.7",
    ...
    "provisioningState": "Creating",
    ...
    "tags": {
      "dept": "IT",
      "costcenter": "9999"
    },
    ...
  },
  ...
]
```

## Manage node pools using a Resource Manager template

When you use an Azure Resource Manager template to create and managed resources, you can typically update the settings in your template and redeploy to update the resource. With node pools in AKS, the initial node pool profile can't be updated once the AKS cluster has been created. This behavior means that you can't update an existing Resource Manager template, make a change to the node pools, and redeploy. Instead, you must create a separate Resource Manager template that updates only the node pools for an existing AKS cluster.

Create a template such as `aks-agentpools.json` and paste the following example manifest. This example template configures the following settings:

- Updates the *Linux* node pool named *myagentpool* to run three nodes.
- Sets the nodes in the node pool to run Kubernetes version *1.15.7*.
- Defines the node size as *Standard\_DS2\_v2*.

Edit these values as need to update, add, or delete node pools as needed:

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "clusterName": {
            "type": "string",
            "metadata": {
                "description": "The name of your existing AKS cluster."
            }
        },
        "location": {
            "type": "string",
            "metadata": {
                "description": "The location of your existing AKS cluster."
            }
        },
        "agentPoolName": {
            "type": "string",
            "defaultValue": "myagentpool",
            "metadata": {
                "description": "The name of the agent pool to create or update."
            }
        },
        "vnetSubnetId": {
            "type": "string",
            "defaultValue": "",
            "metadata": {
                "description": "The Vnet subnet resource ID for your existing AKS cluster."
            }
        }
    },
    "variables": {
        "apiVersion": {
            "aks": "2020-01-01"
        },
        "agentPoolProfiles": {
            "maxPods": 30,
            "osDiskSizeGB": 0,
            "agentCount": 3,
            "agentVmSize": "Standard_DS2_v2",
            "osType": "Linux",
            "vnetSubnetId": "[parameters('vnetSubnetId')]"
        }
    },
    "resources": [
        {
            "apiVersion": "2020-01-01",
            "type": "Microsoft.ContainerService/managedClusters/agentPools",
            "name": "[concat(parameters('clusterName'), '/', parameters('agentPoolName'))]",
            "location": "[parameters('location')]",
            "properties": {
                "maxPods": "[variables('agentPoolProfiles').maxPods]",
                "osDiskSizeGB": "[variables('agentPoolProfiles').osDiskSizeGB]",
                "count": "[variables('agentPoolProfiles').agentCount]",
                "vmSize": "[variables('agentPoolProfiles').agentVmSize]",
                "osType": "[variables('agentPoolProfiles').osType]",
                "storageProfile": "ManagedDisks",
                "type": "VirtualMachineScaleSets",
                "vnetSubnetID": "[variables('agentPoolProfiles').vnetSubnetId]",
                "orchestratorVersion": "1.15.7"
            }
        }
    ]
}
```

Deploy this template using the [az group deployment create](#) command, as shown in the following example. You

are prompted for the existing AKS cluster name and location:

```
az group deployment create \
--resource-group myResourceGroup \
--template-file aks-agentpools.json
```

#### TIP

You can add a tag to your node pool by adding the `tag` property in the template, as shown in the following example.

```
...
"resources": [
{
  ...
  "properties": {
    ...
    "tags": {
      "name1": "val1"
    },
    ...
  }
}
...
```

It may take a few minutes to update your AKS cluster depending on the node pool settings and operations you define in your Resource Manager template.

## Assign a public IP per node in a node pool

#### WARNING

During the preview of assigning a public IP per node, it cannot be used with the *Standard Load Balancer SKU* in AKS due to possible load balancer rules conflicting with VM provisioning. As a result of this limitation, Windows agent pools are not supported with this preview feature. While in preview you must use the *Basic Load Balancer SKU* if you need to assign a public IP per node.

AKS nodes do not require their own public IP addresses for communication. However, some scenarios may require nodes in a node pool to have their own public IP addresses. An example is gaming, where a console needs to make a direct connection to a cloud virtual machine to minimize hops. This can be achieved by registering for a separate preview feature, Node Public IP (preview).

```
az feature register --name NodePublicIPPreview --namespace Microsoft.ContainerService
```

After successful registration, deploy an Azure Resource Manager template following the same instructions as [above](#) and add the boolean value property `enableNodePublicIP` to `agentPoolProfiles`. Set the value to `true` as by default it is set as `false` if not specified. This is a create-time only property and requires a minimum API version of 2019-06-01. This can be applied to both Linux and Windows node pools.

## Clean up resources

In this article, you created an AKS cluster that includes GPU-based nodes. To reduce unnecessary cost, you may want to delete the `gpunodepool`, or the whole AKS cluster.

To delete the GPU-based node pool, use the [az aks nodepool delete](#) command as shown in following example:

```
az aks nodepool delete -g myResourceGroup --cluster-name myAKSCluster --name gpunodepool
```

To delete the cluster itself, use the [az group delete](#) command to delete the AKS resource group:

```
az group delete --name myResourceGroup --yes --no-wait
```

## Next steps

In this article, you learned how to create and manage multiple node pools in an AKS cluster. For more information about how to control pods across node pools, see [Best practices for advanced scheduler features in AKS](#).

To create and use Windows Server container node pools, see [Create a Windows Server container in AKS](#).

# Preview - Add a spot node pool to an Azure Kubernetes Service (AKS) cluster

2/26/2020 • 7 minutes to read • [Edit Online](#)

A spot node pool is a node pool backed by a [spot virtual machine scale set](#). Using spot VMs for nodes with your AKS cluster allows you to take advantage of unutilized capacity in Azure at a significant cost savings. The amount of available unutilized capacity will vary based on many factors, including node size, region, and time of day.

When deploying a spot node pool, Azure will allocate the spot nodes if there's capacity available. But there's no SLA for the spot nodes. A spot scale set that backs the spot node pool is deployed in a single fault domain and offers no high availability guarantees. At any time when Azure needs the capacity back, the Azure infrastructure will evict spot nodes.

Spot nodes are great for workloads that can handle interruptions, early terminations, or evictions. For example, workloads such as batch processing jobs, development and testing environments, and large compute workloads may be good candidates to be scheduled on a spot node pool.

In this article, you add a secondary spot node pool to an existing Azure Kubernetes Service (AKS) cluster.

This article assumes a basic understanding of Kubernetes and Azure Load Balancer concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#).

This feature is currently in preview.

If you don't have an Azure subscription, create a [free account](#) before you begin.

## Before you begin

When you create a cluster to use a spot node pool, that cluster must also use Virtual Machine Scale Sets for node pools and the *Standard* SKU load balancer. You must also add an additional node pool after you create your cluster to use a spot node pool. Adding an additional node pool is covered in a later step, but you first need to enable a preview feature.

### IMPORTANT

AKS preview features are self-service, opt-in. They are provided to gather feedback and bugs from our community. In preview, these features aren't meant for production use. Features in public preview fall under 'best effort' support. Assistance from the AKS technical support teams is available during business hours Pacific timezone (PST) only. For additional information, please see the following support articles:

- [AKS Support Policies](#)
- [Azure Support FAQ](#)

### Register `spotpoolpreview` preview feature

To create an AKS cluster that uses a spot node pool, you must enable the `spotpoolpreview` feature flag on your subscription. This feature provides the latest set of service enhancements when configuring a cluster.

#### Caution

When you register a feature on a subscription, you can't currently un-register that feature. After you enable some preview features, defaults may be used for all AKS clusters then created in the subscription. Don't enable preview features on production subscriptions. Use a separate subscription to test preview features and gather feedback.

Register the `spotpoolpreview` feature flag using the [az feature register](#) command as shown in the following example:

```
az feature register --namespace "Microsoft.ContainerService" --name "spotpoolpreview"
```

It takes a few minutes for the status to show *Registered*. You can check on the registration status using the [az feature list](#) command:

```
az feature list -o table --query "[?contains(name, 'Microsoft.ContainerService/spotpoolpreview')].{Name:name,State:properties.state}"
```

When ready, refresh the registration of the *Microsoft.ContainerService* resource provider using the [az provider register](#) command:

```
az provider register --namespace Microsoft.ContainerService
```

### Install aks-preview CLI extension

To create an AKS cluster that uses a spot node pool, you need the *aks-preview* CLI extension version 0.4.32 or higher. Install the *aks-preview* Azure CLI extension using the [az extension add](#) command, then check for any available updates using the [az extension update](#) command:

```
# Install the aks-preview extension
az extension add --name aks-preview

# Update the extension to make sure you have the latest version installed
az extension update --name aks-preview
```

### Limitations

The following limitations apply when you create and manage AKS clusters with a spot node pool:

- A spot node pool can't be the cluster's default node pool. A spot node pool can only be used for a secondary pool.
- You can't upgrade a spot node pool since spot node pools can't guarantee cordon and drain. You must replace your existing spot node pool with a new one to do operations such as upgrading the Kubernetes version. To replace a spot node pool, create a new spot node pool with a different version of Kubernetes, wait until its status is *Ready*, then remove the old node pool.
- The control plane and node pools cannot be upgraded at the same time. You must upgrade them separately or remove the spot node pool to upgrade the control plane and remaining node pools at the same time.
- A spot node pool must use Virtual Machine Scale Sets.
- You cannot change ScaleSetPriority or SpotMaxPrice after creation.
- When setting SpotMaxPrice, the value must be -1 or a positive value with up to five decimal places.
- A spot node pool will have the label `kubernetes.azure.com/scalesetpriority:spot`, the taint `kubernetes.azure.com/scalesetpriority=spot:NoSchedule`, and system pods will have anti-affinity.
- You must add a [corresponding toleration](#) to schedule workloads on a spot node pool.

## Add a spot node pool to an AKS cluster

You must add a spot node pool to an existing cluster that has multiple node pools enabled. More details on creating an AKS cluster with multiple node pools are available [here](#).

Create a node pool using the [az aks nodepool add](#).

```
az aks nodepool add \
--resource-group myResourceGroup \
--cluster-name myAKSCluster \
--name spotnodepool \
--priority Spot \
--eviction-policy Delete \
--spot-max-price -1 \
--enable-cluster-autoscaler \
--min-count 1 \
--max-count 3 \
--no-wait
```

By default, you create a node pool with a *priority* of *Regular* in your AKS cluster when you create a cluster with multiple node pools. The above command adds an auxiliary node pool to an existing AKS cluster with a *priority* of *Spot*. The *priority* of *Spot* makes the node pool a spot node pool. The *eviction-policy* parameter is set to *Delete* in the above example, which is the default value. When you set the [eviction policy](#) to *Delete*, nodes in the underlying scale set of the node pool are deleted when they're evicted. You can also set the eviction policy to *Deallocate*. When you set the eviction policy to *Deallocate*, nodes in the underlying scale set are set to the stopped-deallocated state upon eviction. Nodes in the stopped-deallocated state count against your compute quota and can cause issues with cluster scaling or upgrading. The *priority* and *eviction-policy* values can only be set during node pool creation. Those values can't be updated later.

The command also enables the [cluster autoscaler](#), which is recommended to use with spot node pools. Based on the workloads running in your cluster, the cluster autoscaler scales up and scales down the number of nodes in the node pool. For spot node pools, the cluster autoscaler will scale up the number of nodes after an eviction if additional nodes are still needed. If you change the maximum number of nodes a node pool can have, you also need to adjust the `maxCount` value associated with the cluster autoscaler. If you do not use a cluster autoscaler, upon eviction, the spot pool will eventually decrease to zero and require a manual operation to receive any additional spot nodes.

#### IMPORTANT

Only schedule workloads on spot node pools that can handle interruptions, such as batch processing jobs and testing environments. It is recommended that you set up [taints and tolerations](#) on your spot node pool to ensure that only workloads that can handle node evictions are scheduled on a spot node pool. For example, the above command by default adds a taint of `kubernetes.azure.com/scalesetpriority=spot:NoSchedule` so only pods with a corresponding toleration are scheduled on this node.

## Verify the spot node pool

To verify your node pool has been added as a spot node pool:

```
az aks nodepool show --resource-group myResourceGroup --cluster-name myAKSCluster --name spotnodepool
```

Confirm `scaleSetPriority` is *Spot*.

To schedule a pod to run on a spot node, add a toleration that corresponds to the taint applied to your spot node. The following example shows a portion of a yaml file that defines a toleration that corresponds to a `kubernetes.azure.com/scalesetpriority=spot:NoSchedule` taint used in the previous step.

```
spec:  
  containers:  
    - name: spot-example  
  tolerations:  
    - key: "kubernetes.azure.com/scalesetpriority"  
      operator: "Equal"  
      value: "spot"  
      effect: "NoSchedule"  
    ...
```

When a pod with this toleration is deployed, Kubernetes can successfully schedule the pod on the nodes with the taint applied.

## Max price for a spot pool

Pricing for spot instances is variable, based on region and SKU. For more information, see pricing for [Linux](#) and [Windows](#).

With variable pricing, you have option to set a max price, in US dollars (USD), using up to 5 decimal places. For example, the value 0.98765 would be a max price of \$0.98765 USD per hour. If you set the max price to -1, the instance won't be evicted based on price. The price for the instance will be the current price for Spot or the price for a standard instance, whichever is less, as long as there is capacity and quota available.

## Next steps

In this article, you learned how to add a spot node pool to an AKS cluster. For more information about how to control pods across node pools, see [Best practices for advanced scheduler features in AKS](#).

# Tutorial: Create a Kubernetes cluster with Azure Kubernetes Service using Terraform

2/19/2020 • 7 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment. AKS allows you to deploy and manage containerized applications without container orchestration expertise. AKS also enables you to do many common maintenance operations without taking your app offline. These operations include provisioning, upgrading, and scaling resources on demand.

In this tutorial, you learn how to do the following tasks:

- Use HCL (HashiCorp Language) to define a Kubernetes cluster
- Use Terraform and AKS to create a Kubernetes cluster
- Use the kubectl tool to test the availability of a Kubernetes cluster

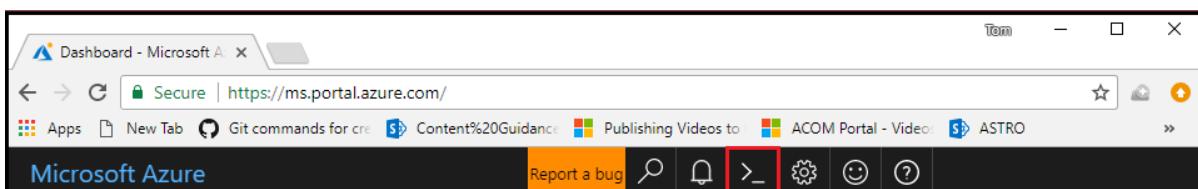
## Prerequisites

- **Azure subscription:** If you don't have an Azure subscription, create a [free account](#) before you begin.
- **Configure Terraform:** Follow the directions in the article, [Terraform and configure access to Azure](#)
- **Azure service principal:** Follow the directions in the section of the **Create the service principal** section in the article, [Create an Azure service principal with Azure CLI](#). Take note of the values for the appId, displayName, password, and tenant.

## Create the directory structure

The first step is to create the directory that holds your Terraform configuration files for the exercise.

1. Browse to the [Azure portal](#).
2. Open [Azure Cloud Shell](#). If you didn't select an environment previously, select **Bash** as your environment.



3. Change directories to the `clouddrive` directory.

```
cd clouddrive
```

4. Create a directory named `terraform-aks-k8s`.

```
mkdir terraform-aks-k8s
```

5. Change directories to the new directory:

```
cd terraform-aks-k8s
```

# Declare the Azure provider

Create the Terraform configuration file that declares the Azure provider.

1. In Cloud Shell, create a file named `main.tf`.

```
code main.tf
```

2. Paste the following code into the editor:

```
provider "azurerm" {  
    version = "~>1.5"  
}  
  
terraform {  
    backend "azurerm" {}  
}
```

3. Save the file (**<Ctrl>S**) and exit the editor (**<Ctrl>Q**).

# Define a Kubernetes cluster

Create the Terraform configuration file that declares the resources for the Kubernetes cluster.

1. In Cloud Shell, create a file named `k8s.tf`.

```
code k8s.tf
```

2. Paste the following code into the editor:

```
resource "azurerm_resource_group" "k8s" {  
    name      = var.resource_group_name  
    location = var.location  
}  
  
resource "random_id" "log_analytics_workspace_name_suffix" {  
    byte_length = 8  
}  
  
resource "azurerm_log_analytics_workspace" "test" {  
    # The WorkSpace name has to be unique across the whole of azure, not just the current  
    # subscription/tenant.  
    name          = "${var.log_analytics_workspace_name}-  
    ${random_id.log_analytics_workspace_name_suffix.dec}"  
    location      = var.log_analytics_workspace_location  
    resource_group_name = azurerm_resource_group.k8s.name  
    sku           = var.log_analytics_workspace_sku  
}  
  
resource "azurerm_log_analytics_solution" "test" {  
    solution_name      = "ContainerInsights"  
    location          = azurerm_log_analytics_workspace.test.location  
    resource_group_name = azurerm_resource_group.k8s.name  
    workspace_resource_id = azurerm_log_analytics_workspace.test.id  
    workspace_name     = azurerm_log_analytics_workspace.test.name  
  
    plan {  
        publisher = "Microsoft"  
        product   = "OMSGallery/ContainerInsights"  
    }  
}
```

```

resource "azurerm_kubernetes_cluster" "k8s" {
    name          = var.cluster_name
    location      = azurerm_resource_group.k8s.location
    resource_group_name = azurerm_resource_group.k8s.name
    dns_prefix     = var.dns_prefix

    linux_profile {
        admin_username = "ubuntu"

        ssh_key {
            key_data = file(var.ssh_public_key)
        }
    }

    default_node_pool {
        name          = "agentpool"
        node_count    = var.agent_count
        vm_size       = "Standard_DS1_v2"
    }

    service_principal {
        client_id      = var.client_id
        client_secret  = var.client_secret
    }

    addon_profile {
        oms_agent {
            enabled           = true
            log_analytics_workspace_id = azurerm_log_analytics_workspace.test.id
        }
    }

    tags = {
        Environment = "Development"
    }
}

```

The preceding code sets the name of the cluster, location, and the resource group name. The prefix for the fully qualified domain name (FQDN) is also set. The FQDN is used to access the cluster.

The `linux_profile` record allows you to configure the settings that enable signing into the worker nodes using SSH.

With AKS, you pay only for the worker nodes. The `default_node_pool` record configures the details for these worker nodes. The `default_node_pool` record includes the number of worker nodes to create and the type of worker nodes. If you need to scale up or scale down the cluster in the future, you modify the `count` value in this record.

- Save the file (**<Ctrl>S**) and exit the editor (**<Ctrl>Q**).

## Declare the variables

- In Cloud Shell, create a file named `variables.tf`.

```
code variables.tf
```

- Paste the following code into the editor:

```

variable "client_id" {}
variable "client_secret" {}

variable "agent_count" {
    default = 3
}

variable "ssh_public_key" {
    default = "~/.ssh/id_rsa.pub"
}

variable "dns_prefix" {
    default = "k8stest"
}

variable cluster_name {
    default = "k8stest"
}

variable resource_group_name {
    default = "azure-k8stest"
}

variable location {
    default = "Central US"
}

variable log_analytics_workspace_name {
    default = "testLogAnalyticsWorkspaceName"
}

# refer https://azure.microsoft.com/global-infrastructure/services/?products=monitor for log analytics
# available regions
variable log_analytics_workspace_location {
    default = "eastus"
}

# refer https://azure.microsoft.com/pricing/details/monitor/ for log analytics pricing
variable log_analytics_workspace_sku {
    default = "PerGB2018"
}

```

3. Save the file (**<Ctrl>S**) and exit the editor (**<Ctrl>Q**).

## Create a Terraform output file

Terraform outputs allow you to define values that will be highlighted to the user when Terraform applies a plan, and can be queried using the `terraform output` command. In this section, you create an output file that allows access to the cluster with `kubectl`.

1. In Cloud Shell, create a file named `output.tf`.

```
code output.tf
```

2. Paste the following code into the editor:

```

output "client_key" {
    value = azurerm_kubernetes_cluster.k8s.kube_config.0.client_key
}

output "client_certificate" {
    value = azurerm_kubernetes_cluster.k8s.kube_config.0.client_certificate
}

output "cluster_ca_certificate" {
    value = azurerm_kubernetes_cluster.k8s.kube_config.0.cluster_ca_certificate
}

output "cluster_username" {
    value = azurerm_kubernetes_cluster.k8s.kube_config.0.username
}

output "cluster_password" {
    value = azurerm_kubernetes_cluster.k8s.kube_config.0.password
}

output "kube_config" {
    value = azurerm_kubernetes_cluster.k8s.kube_config_raw
}

output "host" {
    value = azurerm_kubernetes_cluster.k8s.kube_config.0.host
}

```

- Save the file (**<Ctrl>S**) and exit the editor (**<Ctrl>Q**).

## Set up Azure storage to store Terraform state

Terraform tracks state locally via the `terraform.tfstate` file. This pattern works well in a single-person environment. In a multi-person environment, [Azure storage](#) is used to track state.

In this section, you see how to do the following tasks:

- Retrieve storage account information (account name and account key)
- Create a storage container into which Terraform state information will be stored.

- In the Azure portal, select **All services** in the left menu.
- Select **Storage accounts**.
- On the **Storage accounts** tab, select the name of the storage account into which Terraform is to store state. For example, you can use the storage account created when you opened Cloud Shell the first time. The storage account name created by Cloud Shell typically starts with `cs` followed by a random string of numbers and letters. Take note of the storage account you select. This value is needed later.
- On the storage account tab, select **Access keys**.

The screenshot shows the Azure Storage account settings page for a specific storage account. The left sidebar lists various management options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, and Storage Explorer (preview). Below these are sections for SETTINGS, including Access keys, Configuration, and Encryption. The 'Access keys' section is highlighted with a red box. The main content area displays details about the storage account, such as its resource group, status (Primary: Available), location (West US), subscription, and tags. It also lists services like Blobs and Files, each with a 'Configure CORS rules' link. The 'Access keys' section contains two key entries: 'key1' and 'key2', each with a 'Key' input field and a copy icon.

5. Make note of the **key1 key** value. (Selecting the icon to the right of the key copies the value to the clipboard.)

This screenshot is identical to the one above, but the 'key1' key value in the 'Key' input field is highlighted with a red box, indicating it is the value to be copied.

6. In Cloud Shell, create a container in your Azure storage account. Replace the placeholders with appropriate values for your environment.

```
az storage container create -n tfstate --account-name <YourAzureStorageAccountName> --account-key
<YourAzureStorageAccountKey>
```

## Create the Kubernetes cluster

In this section, you see how to use the `terraform init` command to create the resources defined in the configuration files you created in the previous sections.

1. In Cloud Shell, initialize Terraform. Replace the placeholders with appropriate values for your environment.

```
terraform init -backend-config="storage_account_name=<YourAzureStorageAccountName>" -backend-
config="container_name=tfstate" -backend-config="access_key=<YourStorageAccountAccessKey>" -backend-
config="key=codelab.microsoft.tfstate"
```

The `terraform init` command displays the success of initializing the backend and provider plug-in:

```
Initializing the backend...
```

```
Initializing provider plugins...
```

```
Terraform has been successfully initialized!
```

```
You may now begin working with Terraform. Try running "terraform plan" to see  
any changes that are required for your infrastructure. All Terraform commands  
should now work.
```

```
If you ever set or change modules or backend configuration for Terraform,  
rerun this command to reinitialize your working directory. If you forget, other  
commands will detect it and remind you to do so if necessary.
```

2. Export your service principal credentials. Replace the placeholders with appropriate values from your service principal.

```
export TF_VAR_client_id=<service-principal-appid>  
export TF_VAR_client_secret=<service-principal-password>
```

3. Run the `terraform plan` command to create the Terraform plan that defines the infrastructure elements.

```
terraform plan -out out.plan
```

The `terraform plan` command displays the resources that will be created when you run the `terraform apply` command:

```
  id: <computed>  
  location: "centralus"  
  name: "azure-k8stest"  
  tags.%: <computed>
```

```
Plan: 2 to add, 0 to change, 0 to destroy.
```

```
-----  
This plan was saved to: out.plan
```

```
To perform exactly these actions, run the following command to apply:  
  terraform apply "out.plan"
```

4. Run the `terraform apply` command to apply the plan to create the Kubernetes cluster. The process to create a Kubernetes cluster can take several minutes, resulting in the Cloud Shell session timing out. If the Cloud Shell session times out, you can follow the steps in the section "Recover from a Cloud Shell timeout" to enable you to complete the tutorial.

```
terraform apply out.plan
```

The `terraform apply` command displays the results of creating the resources defined in your configuration files:

```

tags.%:   "" => "<computed>"
azurerm_resource_group.k8s: Creation complete after 1s (ID: /subscriptions/ad7af7
azurerm_kubernetes_cluster.k8s: Creating...
  agent_pool_profile.#:                               "" => "1"
  agent_pool_profile.0.count:                         "" => "3"
  agent_pool_profile.0.dns_prefix:                   "" => "<computed>"
  agent_pool_profile.0.fqdn:                          "" => "<computed>"
  agent_pool_profile.0.name:                          "" => "default"
  agent_pool_profile.0.os_disk_size_gb:              "" => "30"
  agent_pool_profile.0.os_type:                      "" => "Linux"
  agent_pool_profile.0.vm_size:                     "" => "Standard_D2"
  dns_prefix:                                       "" => "k8stest"
  fqdn:                                            "" => "<computed>"
  kube_config.#:                                    "" => "<computed>"
  kube_config_raw:                                  "<sensitive>" => "<sensitive>"
  kubernetes_version:                             "" => "<computed>"
  linux_profile.#:                                "" => "1"
  linux_profile.0.admin_username:                  "" => "ubuntu"
  linux_profile.0.ssh_key.#:                      "" => "1"
  linux_profile.0.ssh_key.0.key_data:              "" => "ssh-rsa"
  location:                                         "" => "centralus"
  name:                                             "" => "k8stest"
  resource_group_name:                            "" => "azure-k8stest"
  service_principal.#:                           "" => "1"
  service_principal.2782116410.client_id:        "" => ""
  service_principal.2782116410.client_secret:     "<sensitive>" => "<sensitive>"
  tags.%:                                         "" => "1"
  tags.Environment:                             "" => "Development"
azurerm_kubernetes_cluster.k8s: Still creating... (10s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (20s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (30s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (40s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (50s elapsed)
azurerm_kubernetes_cluster.k8s: Still creating... (1m0s elapsed)

```

5. In the Azure portal, select **All resources** in the left menu to see the resources created for your new Kubernetes cluster.

Home > All resources		
All resources Microsoft		
<input type="button" value="Add"/> <input type="button" value="Edit columns"/> <input type="button" value="Refresh"/> <input type="button" value="Assign tags"/> <input type="button" value="Delete"/>		
Subscriptions: 1 of 6 selected		
<input type="text" value="Filter by name..."/> <input type="button" value="Azure Subscription"/> <input type="button" value="All resource groups"/> <input type="button" value="All types"/> <input type="button" value="All locations"/>		
48 items <input type="checkbox"/> Show hidden types <small>①</small>		
<input type="checkbox"/> NAME <small>④</small>	RESOURCE GROUP <small>④</small>	
<input type="checkbox"/> aks-agentpool-21324540-nsg	MC_nic-k8stest_k8stest_centralus	
<input type="checkbox"/> aks-agentpool-21324540-routetable	MC_nic-k8stest_k8stest_centralus	
<input type="checkbox"/> aks-default-21324540-0	MC_nic-k8stest_k8stest_centralus	
<input type="checkbox"/> aks-default-21324540-0_OsDisk_1_afcbe8a12cff4ed78ec32813cea6204a	MC_NIC-K8STEST_K8STEST_CENTRALUS	
<input type="checkbox"/> aks-default-21324540-1	MC_nic-k8stest_k8stest_centralus	
<input type="checkbox"/> aks-default-21324540-1_OsDisk_1_95c4a19caec8404f974868c84fa04523	MC_NIC-K8STEST_K8STEST_CENTRALUS	
<input type="checkbox"/> aks-default-21324540-2	MC_nic-k8stest_k8stest_centralus	
<input type="checkbox"/> aks-default-21324540-2_OsDisk_1_fb9917ab0b8f4ca0b901465819360f75	MC_NIC-K8STEST_K8STEST_CENTRALUS	
<input type="checkbox"/> aks-default-21324540-nic-0	MC_nic-k8stest_k8stest_centralus	
<input type="checkbox"/> aks-default-21324540-nic-1	MC_nic-k8stest_k8stest_centralus	
<input type="checkbox"/> aks-default-21324540-nic-2	MC_nic-k8stest_k8stest_centralus	

## Recover from a Cloud Shell timeout

If the Cloud Shell session times out, you can do the following steps to recover:

1. Start a Cloud Shell session.
2. Change to the directory containing your Terraform configuration files.

```
cd /clouddrive/terraform-aks-k8s
```

3. Run the following command:

```
export KUBECONFIG=./azurek8s
```

## Test the Kubernetes cluster

The Kubernetes tools can be used to verify the newly created cluster.

1. Get the Kubernetes configuration from the Terraform state and store it in a file that kubectl can read.

```
echo "$(terraform output kube_config)" > ./azurek8s
```

2. Set an environment variable so that kubectl picks up the correct config.

```
export KUBECONFIG=./azurek8s
```

3. Verify the health of the cluster.

```
kubectl get nodes
```

You should see the details of your worker nodes, and they should all have a status **Ready**, as shown in the following image:

```
$ kubectl get nodes
  NAME           STATUS    ROLES      AGE     VERSION
  aks-default-27881813-0   Ready    agent     48m    v1.9.6
  aks-default-27881813-1   Ready    agent     48m    v1.9.6
  aks-default-27881813-2   Ready    agent     48m    v1.9.6
```

## Monitor health and logs

When the AKS cluster was created, monitoring was enabled to capture health metrics for both the cluster nodes and pods. These health metrics are available in the Azure portal. For more information on container health monitoring, see [Monitor Azure Kubernetes Service health](#).

## Next steps

[Learn more about using Terraform in Azure](#)

# Access the Kubernetes web dashboard in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

Kubernetes includes a web dashboard that can be used for basic management operations. This dashboard lets you view basic health status and metrics for your applications, create and deploy services, and edit existing applications. This article shows you how to access the Kubernetes dashboard using the Azure CLI, then guides you through some basic dashboard operations.

For more information on the Kubernetes dashboard, see [Kubernetes Web UI Dashboard](#).

## Before you begin

The steps detailed in this document assume that you have created an AKS cluster and have established a `kubectl` connection with the cluster. If you need to create an AKS cluster, see the [AKS quickstart](#).

You also need the Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Start the Kubernetes dashboard

To start the Kubernetes dashboard, use the `az aks browse` command. The following example opens the dashboard for the cluster named *myAKSCluster* in the resource group named *myResourceGroup*:

```
az aks browse --resource-group myResourceGroup --name myAKSCluster
```

This command creates a proxy between your development system and the Kubernetes API, and opens a web browser to the Kubernetes dashboard. If a web browser doesn't open to the Kubernetes dashboard, copy and paste the URL address noted in the Azure CLI, typically `http://127.0.0.1:8001`.

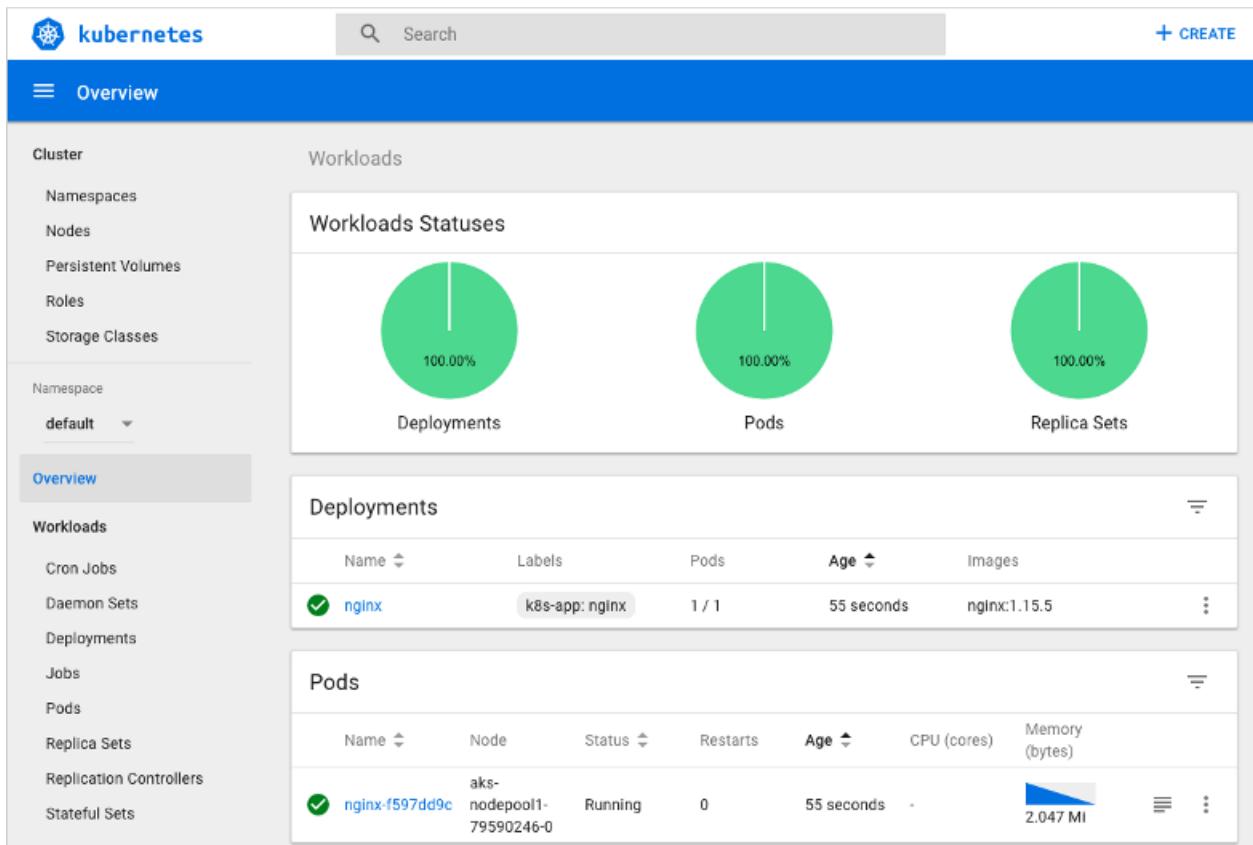
### IMPORTANT

If your AKS cluster uses RBAC, a *ClusterRoleBinding* must be created before you can correctly access the dashboard. By default, the Kubernetes dashboard is deployed with minimal read access and displays RBAC access errors. The Kubernetes dashboard does not currently support user-provided credentials to determine the level of access, rather it uses the roles granted to the service account. A cluster administrator can choose to grant additional access to the *kubernetes-dashboard* service account, however this can be a vector for privilege escalation. You can also integrate Azure Active Directory authentication to provide a more granular level of access.

To create a binding, use the `kubectl create clusterrolebinding` command. The following example shows how to create a sample binding, however, this sample binding does not apply any additional authentication components and may lead to insecure use. The Kubernetes dashboard is open to anyone with access to the URL. Do not expose the Kubernetes dashboard publicly.

```
kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-admin --serviceaccount=kube-system:kubernetes-dashboard
```

For more information on using the different authentication methods, see the Kubernetes dashboard wiki on [access controls](#).



## Create an application

To see how the Kubernetes dashboard can reduce the complexity of management tasks, let's create an application. You can create an application from the Kubernetes dashboard by providing text input, a YAML file, or through a graphical wizard.

To create an application, complete the following steps:

1. Select the **Create** button in the upper right window.
2. To use the graphical wizard, choose to **Create an app**.
3. Provide a name for the deployment, such as *nginx*
4. Enter the name for the container image to use, such as *nginx:1.15.5*
5. To expose port 80 for web traffic, you create a Kubernetes service. Under **Service**, select **External**, then enter **80** for both the port and target port.
6. When ready, select **Deploy** to create the app.

The screenshot shows the Kubernetes dashboard under the 'Resource creation' section. On the left sidebar, 'Discovery and Load Balancing' is selected. In the main area, the 'CREATE AN APP' tab is active. The form fields are as follows:

- App name\***: nginx
- Container image\***: nginx:1.15.5
- Number of pods\***: 1
- Service\***: External
- Port\***: 80
- Target port\***: 80
- Protocol\***: TCP

Below the form, there are 'SHOW ADVANCED OPTIONS' and 'DEPLOY' buttons.

It takes a minute or two for a public external IP address to be assigned to the Kubernetes service. On the left-hand side, under **Discovery and Load Balancing** select **Services**. Your application's service is listed, including the *External endpoints*, as shown in the following example:

The screenshot shows the Kubernetes dashboard under the 'Discovery and load balancing > Services' section. On the left sidebar, 'Services' is selected. The table displays the following data:

Name	Labels	Cluster IP	Internal endpoints	External endpoints	Age
nginx	k8s-app: nginx	10.0.180.34	nginx:80 TCP nginx:30344 TCP	23.96.99.120:80	41 minutes
kubernetes	component: ap... provider: kuber...	10.0.0.1	kubernetes:443 T kubernetes:0 TCP	-	7 days

Select the endpoint address to open a web browser window to the default NGINX page:

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

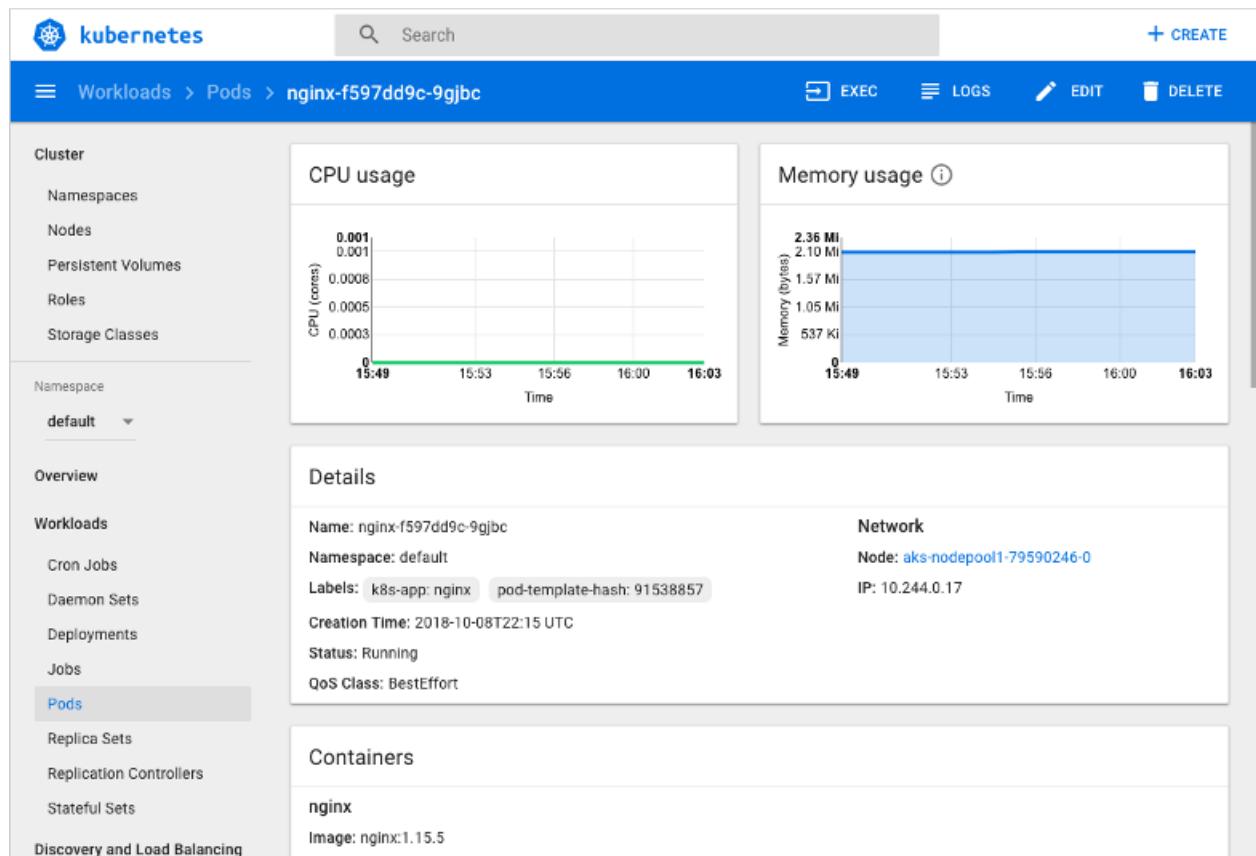
For online documentation and support please refer to [nginx.org](#). Commercial support is available at [nginx.com](#).

Thank you for using nginx.

## View pod information

The Kubernetes dashboard can provide basic monitoring metrics and troubleshooting information such as logs.

To see more information about your application pods, select **Pods** in the left-hand menu. The list of available pods is shown. Choose your *nginx* pod to view information, such as resource consumption:



## Edit the application

In addition to creating and viewing applications, the Kubernetes dashboard can be used to edit and update application deployments. To provide additional redundancy for the application, let's increase the number of NGINX replicas.

To edit a deployment:

1. Select **Deployments** in the left-hand menu, and then choose your *nginx* deployment.
2. Select **Edit** in the upper right-hand navigation bar.
3. Locate the `spec.replicas` value, at around line 20. To increase the number of replicas for the application, change this value from 1 to 3.
4. Select **Update** when ready.

```

15  "annotations": {
16    "deployment.kubernetes.io/revision": "1"
17  }
18  },
19  "spec": {
20    "replicas": 3,
21    "selector": {
22      "matchLabels": {
23        "k8s-app": "nginx"
24      }
25    },
26    "template": {
27      "metadata": {
28        "name": "nginx",
29        "creationTimestamp": null,
30        "labels": {

```

**CANCEL    COPY    UPDATE**

It takes a few moments for the new pods to be created inside a replica set. On the left-hand menu, choose **Replica Sets**, and then choose your *nginx* replica set. The list of pods now reflects the updated replica count, as shown in the following example output:

Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
nginx-f597dd9c	aks-nodepool1-79590246-0	Running	0	56 minutes	0	2.098 Mi
nginx-f597dd9c	aks-nodepool1-79590246-0	Running	0	10 seconds	-	2.059 Mi
nginx-f597dd9c	aks-nodepool1-79590246-0	Running	0	10 seconds	-	1.980 Mi

## Next steps

For more information about the Kubernetes dashboard, see the [Kubernetes Web UI Dashboard](#).

# Dynamically create and use a persistent volume with Azure disks in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

A persistent volume represents a piece of storage that has been provisioned for use with Kubernetes pods. A persistent volume can be used by one or many pods, and can be dynamically or statically provisioned. This article shows you how to dynamically create persistent volumes with Azure disks for use by a single pod in an Azure Kubernetes Service (AKS) cluster.

## NOTE

An Azure disk can only be mounted with *Access mode* type *ReadWriteOnce*, which makes it available to only a single pod in AKS. If you need to share a persistent volume across multiple pods, use [Azure Files](#).

For more information on Kubernetes volumes, see [Storage options for applications in AKS](#).

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the [Azure CLI](#) or [using the Azure portal](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Built in storage classes

A storage class is used to define how a unit of storage is dynamically created with a persistent volume. For more information on Kubernetes storage classes, see [Kubernetes Storage Classes](#).

Each AKS cluster includes two pre-created storage classes, both configured to work with Azure disks:

- The *default* storage class provisions a standard Azure disk.
  - Standard storage is backed by HDDs, and delivers cost-effective storage while still being performant. Standard disks are ideal for a cost effective dev and test workload.
- The *managed-premium* storage class provisions a premium Azure disk.
  - Premium disks are backed by SSD-based high-performance, low-latency disk. Perfect for VMs running production workload. If the AKS nodes in your cluster use premium storage, select the *managed-premium* class.

These default storage classes don't allow you to update the volume size once created. To enable this ability, add the `allowVolumeExpansion: true` line to one of the default storage classes, or create your own custom storage class. You can edit an existing storage class using the `kubectl edit sc` command. For more information on storage classes and creating your own, see [Storage options for applications in AKS](#).

Use the `kubectl get sc` command to see the pre-created storage classes. The following example shows the pre-create storage classes available within an AKS cluster:

```
$ kubectl get sc
```

NAME	PROVISIONER	AGE
default (default)	kubernetes.io/azure-disk	1h
managed-premium	kubernetes.io/azure-disk	1h

#### NOTE

Persistent volume claims are specified in GiB but Azure managed disks are billed by SKU for a specific size. These SKUs range from 32GiB for S4 or P4 disks to 32TiB for S80 or P80 disks (in preview). The throughput and IOPS performance of a Premium managed disk depends on the both the SKU and the instance size of the nodes in the AKS cluster. For more information, see [Pricing and Performance of Managed Disks](#).

## Create a persistent volume claim

A persistent volume claim (PVC) is used to automatically provision storage based on a storage class. In this case, a PVC can use one of the pre-created storage classes to create a standard or premium Azure managed disk.

Create a file named `azure-premium.yaml`, and copy in the following manifest. The claim requests a disk named `azure-managed-disk` that is 5GB in size with *ReadWriteOnce* access. The *managed-premium* storage class is specified as the storage class.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
  - ReadWriteOnce
  storageClassName: managed-premium
  resources:
    requests:
      storage: 5Gi
```

#### TIP

To create a disk that uses standard storage, use `storageClassName: default` rather than *managed-premium*.

Create the persistent volume claim with the `kubectl apply` command and specify your `azure-premium.yaml` file:

```
$ kubectl apply -f azure-premium.yaml
persistentvolumeclaim/azure-managed-disk created
```

## Use the persistent volume

Once the persistent volume claim has been created and the disk successfully provisioned, a pod can be created with access to the disk. The following manifest creates a basic NGINX pod that uses the persistent volume claim named `azure-managed-disk` to mount the Azure disk at the path `/mnt/azure`. For Windows Server containers (currently in preview in AKS), specify a *mountPath* using the Windows path convention, such as 'D:'.

Create a file named `azure-pvc-disk.yaml`, and copy in the following manifest.

```

kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: nginx:1.15.5
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      volumeMounts:
        - mountPath: "/mnt/azure"
          name: volume
  volumes:
    - name: volume
      persistentVolumeClaim:
        claimName: azure-managed-disk

```

Create the pod with the [kubectl apply](#) command, as shown in the following example:

```

$ kubectl apply -f azure-pvc-disk.yaml

pod/mypod created

```

You now have a running pod with your Azure disk mounted in the `/mnt/azure` directory. This configuration can be seen when inspecting your pod via [kubectl describe pod mypod](#), as shown in the following condensed example:

```

$ kubectl describe pod mypod

[...]
Volumes:
  volume:
    Type:      PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:  azure-managed-disk
    ReadOnly:   false
  default-token-smm2n:
    Type:      Secret (a volume populated by a Secret)
    SecretName: default-token-smm2n
    Optional:  false
[...]
Events:
  Type  Reason          Age    From           Message
  ----  -----          ----  --  -----
  Normal Scheduled      2m    default-scheduler  Successfully assigned mypod to aks-nodepool1-79590246-0
  Normal SuccessfulMountVolume 2m    kubelet, aks-nodepool1-79590246-0  MountVolume.SetUp succeeded for volume "default-token-smm2n"
  Normal SuccessfulMountVolume 1m    kubelet, aks-nodepool1-79590246-0  MountVolume.SetUp succeeded for volume "pvc-faf0f176-8b8d-11e8-923b-deb28c58d242"
[...]

```

## Back up a persistent volume

To back up the data in your persistent volume, take a snapshot of the managed disk for the volume. You can then use this snapshot to create a restored disk and attach to pods as a means of restoring the data.

First, get the volume name with the `kubectl get pvc` command, such as for the PVC named *azure-managed-disk*:

\$ kubectl get pvc azure-managed-disk						
NAME	STORAGECLASS	STATUS	VOLUME	CAPACITY	ACCESS MODES	
		AGE				
azure-managed-disk	premium	Bound 3m	pvc-faf0f176-8b8d-11e8-923b-deb28c58d242	5Gi	RWO	managed-

This volume name forms the underlying Azure disk name. Query for the disk ID with [az disk list](#) and provide your PVC volume name, as shown in the following example:

```
$ az disk list --query '[].id | [?contains(@, `pvc-faf0f176-8b8d-11e8-923b-deb28c58d242`)]' -o tsv
/subscriptions/<guid>/resourceGroups/MC_MYRESOURCEGROUP_MYAKSCLUSTER_EASTUS/providers/MicrosoftCompute/disks/k
ubernetes-dynamic-pvc-faf0f176-8b8d-11e8-923b-deb28c58d242
```

Use the disk ID to create a snapshot disk with [az snapshot create](#). The following example creates a snapshot named *pvcSnapshot* in the same resource group as the AKS cluster (*MC\_myResourceGroup\_myAKSCluster\_eastus*). You may encounter permission issues if you create snapshots and restore disks in resource groups that the AKS cluster does not have access to.

```
$ az snapshot create \
--resource-group MC_myResourceGroup_myAKSCluster_eastus \
--name pvcSnapshot \
--source
/subscriptions/<guid>/resourceGroups/MC_myResourceGroup_myAKSCluster_eastus/providers/MicrosoftCompute/disks/k
ubernetes-dynamic-pvc-faf0f176-8b8d-11e8-923b-deb28c58d242
```

Depending on the amount of data on your disk, it may take a few minutes to create the snapshot.

## Restore and use a snapshot

To restore the disk and use it with a Kubernetes pod, use the snapshot as a source when you create a disk with [az disk create](#). This operation preserves the original resource if you then need to access the original data snapshot. The following example creates a disk named *pvcRestored* from the snapshot named *pvcSnapshot*:

```
az disk create --resource-group MC_myResourceGroup_myAKSCluster_eastus --name pvcRestored --source pvcSnapshot
```

To use the restored disk with a pod, specify the ID of the disk in the manifest. Get the disk ID with the [az disk show](#) command. The following example gets the disk ID for *pvcRestored* created in the previous step:

```
az disk show --resource-group MC_myResourceGroup_myAKSCluster_eastus --name pvcRestored --query id -o tsv
```

Create a pod manifest named `azure-restored.yaml` and specify the disk URI obtained in the previous step. The following example creates a basic NGINX web server, with the restored disk mounted as a volume at `/mnt/azure`:

```
kind: Pod
apiVersion: v1
metadata:
  name: mypodrestored
spec:
  containers:
    - name: mypodrestored
      image: nginx:1.15.5
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      volumeMounts:
        - mountPath: "/mnt/azure"
          name: volume
  volumes:
    - name: volume
      azureDisk:
        kind: Managed
        diskName: pvcRestored
        diskURI:
          /subscriptions/<guid>/resourceGroups/MC_myResourceGroupAKS_myAKSCluster_eastus/providers/Microsoft.Compute/disks/pvcRestored
```

Create the pod with the [kubectl apply](#) command, as shown in the following example:

```
$ kubectl apply -f azure-restored.yaml
pod/mypodrestored created
```

You can use [kubectl describe pod mypodrestored](#) to view details of the pod, such as the following condensed example that shows the volume information:

```
$ kubectl describe pod mypodrestored
[...]
Volumes:
  volume:
    Type:      AzureDisk (an Azure Data Disk mount on the host and bind mount to the pod)
    DiskName:  pvcRestored
    DiskURI:   /subscriptions/19da35d3-9a1a-4f3b-9b9c-
3c56ef409565/resourceGroups/MC_myResourceGroupAKS_myAKSCluster_eastus/providers/Microsoft.Compute/disks/pvcRestored
    Kind:      Managed
    FSType:    ext4
    CachingMode:  ReadWrite
    ReadOnly:   false
[...]
```

## Next steps

For associated best practices, see [Best practices for storage and backups in AKS](#).

Learn more about Kubernetes persistent volumes using Azure disks.

[Kubernetes plugin for Azure disks](#)

# Manually create and use a volume with Azure disks in Azure Kubernetes Service (AKS)

2/25/2020 • 3 minutes to read • [Edit Online](#)

Container-based applications often need to access and persist data in an external data volume. If a single pod needs access to storage, you can use Azure disks to present a native volume for application use. This article shows you how to manually create an Azure disk and attach it to a pod in AKS.

## NOTE

An Azure disk can only be mounted to a single pod at a time. If you need to share a persistent volume across multiple pods, use [Azure Files](#).

For more information on Kubernetes volumes, see [Storage options for applications in AKS](#).

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the [Azure CLI](#) or [using the Azure portal](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an Azure disk

When you create an Azure disk for use with AKS, you can create the disk resource in the **node** resource group. This approach allows the AKS cluster to access and manage the disk resource. If you instead create the disk in a separate resource group, you must grant the Azure Kubernetes Service (AKS) service principal for your cluster the `Contributor` role to the disk's resource group.

For this article, create the disk in the node resource group. First, get the resource group name with the `az aks show` command and add the `--query nodeResourceGroup` query parameter. The following example gets the node resource group for the AKS cluster name *myAKSCluster* in the resource group name *myResourceGroup*:

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv  
MC_myResourceGroup_myAKSCluster_eastus
```

Now create a disk using the `az disk create` command. Specify the node resource group name obtained in the previous command, and then a name for the disk resource, such as *myAKSDisk*. The following example creates a 20GiB disk, and outputs the ID of the disk once created. If you need to create a disk for use with Windows Server containers (currently in preview in AKS), add the `--os-type windows` parameter to correctly format the disk.

```
az disk create \  
  --resource-group MC_myResourceGroup_myAKSCluster_eastus \  
  --name myAKSDisk \  
  --size-gb 20 \  
  --query id --output tsv
```

#### NOTE

Azure disks are billed by SKU for a specific size. These SKUs range from 32GiB for S4 or P4 disks to 32TiB for S80 or P80 disks (in preview). The throughput and IOPS performance of a Premium managed disk depends on both the SKU and the instance size of the nodes in the AKS cluster. See [Pricing and Performance of Managed Disks](#).

The disk resource ID is displayed once the command has successfully completed, as shown in the following example output. This disk ID is used to mount the disk in the next step.

```
/subscriptions/<subscriptionID>/resourceGroups/MC_myAKSCluster_myAKSCluster_eastus/providers/Microsoft.Compute
/disks/myAKSDisk
```

## Mount disk as volume

To mount the Azure disk into your pod, configure the volume in the container spec. Create a new file named `azure-disk-pod.yaml` with the following contents. Update `diskName` with the name of the disk created in the previous step, and `diskURI` with the disk ID shown in output of the disk create command. If desired, update the `mountPath`, which is the path where the Azure disk is mounted in the pod. For Windows Server containers (currently in preview in AKS), specify a `mountPath` using the Windows path convention, such as 'D:'.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - image: nginx:1.15.5
      name: mypod
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      volumeMounts:
        - name: azure
          mountPath: /mnt/azure
  volumes:
    - name: azure
      azureDisk:
        kind: Managed
        diskName: myAKSDisk
        diskURI:
/subscriptions/<subscriptionID>/resourceGroups/MC_myAKSCluster_myAKSCluster_eastus/providers/Microsoft.Compute
/disks/myAKSDisk
```

Use the `kubectl` command to create the pod.

```
kubectl apply -f azure-disk-pod.yaml
```

You now have a running pod with an Azure disk mounted at `/mnt/azure`. You can use `kubectl describe pod mypod` to verify the disk is mounted successfully. The following condensed example output shows the volume mounted in the container:

```
[...]
Volumes:
  azure:
    Type:        AzureDisk (an Azure Data Disk mount on the host and bind mount to the pod)
    DiskName:    myAKSDisk
    DiskURI:
/subscriptions/<subscriptionID/resourceGroups/MC_myResourceGroupAKS_myAKScluster_eastus/providers/Microsoft.Co
mpute/disks/myAKSDisk
      Kind:       Managed
      FSType:     ext4
      CachingMode: ReadWrite
      ReadOnly:   false
  default-token-z5sd7:
    Type:        Secret (a volume populated by a Secret)
    SecretName: default-token-z5sd7
    Optional:   false
[...]
Events:
  Type  Reason          Age   From            Message
  ----  -----          --   --              -----
  Normal Scheduled      1m    default-scheduler  Successfully assigned mypod to aks-
nodepool1-79590246-0
  Normal SuccessfulMountVolume 1m    kubelet, aks-nodepool1-79590246-0  MountVolume.SetUp succeeded for
volume "default-token-z5sd7"
  Normal SuccessfulMountVolume 41s   kubelet, aks-nodepool1-79590246-0  MountVolume.SetUp succeeded for
volume "azure"
[...]
```

## Next steps

For associated best practices, see [Best practices for storage and backups in AKS](#).

For more information about AKS clusters interact with Azure disks, see the [Kubernetes plugin for Azure Disks](#).

# Dynamically create and use a persistent volume with Azure Files in Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

A persistent volume represents a piece of storage that has been provisioned for use with Kubernetes pods. A persistent volume can be used by one or many pods, and can be dynamically or statically provisioned. If multiple pods need concurrent access to the same storage volume, you can use Azure Files to connect using the [Server Message Block \(SMB\) protocol](#). This article shows you how to dynamically create an Azure Files share for use by multiple pods in an Azure Kubernetes Service (AKS) cluster.

For more information on Kubernetes volumes, see [Storage options for applications in AKS](#).

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the [Azure CLI](#) or [using the Azure portal](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create a storage class

A storage class is used to define how an Azure file share is created. A storage account is automatically created in the [node resource group](#) for use with the storage class to hold the Azure file shares. Choose of the following [Azure storage redundancy](#) for *skuName*:

- *Standard\_LRS* - standard locally redundant storage (LRS)
- *Standard\_GRS* - standard geo-redundant storage (GRS)
- *Standard\_RAGRS* - standard read-access geo-redundant storage (RA-GRS)
- *Premium\_LRS* - premium locally redundant storage (LRS)

### NOTE

Azure Files support premium storage in AKS clusters that run Kubernetes 1.13 or higher.

For more information on Kubernetes storage classes for Azure Files, see [Kubernetes Storage Classes](#).

Create a file named `azure-file-sc.yaml` and copy in the following example manifest. For more information on *mountOptions*, see the [Mount options](#) section.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=1000
  - gid=1000
  - mfsymlinks
  - nobrl
  - cache=none
parameters:
  skuName: Standard_LRS
```

Create the storage class with the [kubectl apply](#) command:

```
kubectl apply -f azure-file-sc.yaml
```

## Create a persistent volume claim

A persistent volume claim (PVC) uses the storage class object to dynamically provision an Azure file share. The following YAML can be used to create a persistent volume claim 5 GB in size with *ReadWriteMany* access. For more information on access modes, see the [Kubernetes persistent volume](#) documentation.

Now create a file named `azure-file-pvc.yaml` and copy in the following YAML. Make sure that the `storageClassName` matches the storage class created in the last step:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azurefile
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: azurefile
  resources:
    requests:
      storage: 5Gi
```

### NOTE

If using the *Premium\_LRS* sku for your storage class, the minimum value for `storage` must be *100Gi*.

Create the persistent volume claim with the [kubectl apply](#) command:

```
kubectl apply -f azure-file-pvc.yaml
```

Once completed, the file share will be created. A Kubernetes secret is also created that includes connection information and credentials. You can use the [kubectl get](#) command to view the status of the PVC:

```
$ kubectl get pvc azurefile
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
azurefile	Bound	pvc-8436e62e-a0d9-11e5-8521-5a8664dc0477	5Gi	RWX	azurefile	5m

## Use the persistent volume

The following YAML creates a pod that uses the persistent volume claim *azurefile* to mount the Azure file share at the */mnt/azure* path. For Windows Server containers (currently in preview in AKS), specify a *mountPath* using the Windows path convention, such as '*D:*'.

Create a file named `azure-pvc-files.yaml`, and copy in the following YAML. Make sure that the *claimName* matches the PVC created in the last step.

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: nginx:1.15.5
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      volumeMounts:
        - mountPath: "/mnt/azure"
          name: volume
    volumes:
      - name: volume
        persistentVolumeClaim:
          claimName: azurefile
```

Create the pod with the [kubectl apply](#) command.

```
kubectl apply -f azure-pvc-files.yaml
```

You now have a running pod with your Azure Files share mounted in the */mnt/azure* directory. This configuration can be seen when inspecting your pod via `kubectl describe pod mypod`. The following condensed example output shows the volume mounted in the container:

```
Containers:
  mypod:
    Container ID:  docker://053bc9c0df72232d755aa040bfba8b533fa696b123876108dec400e364d2523e
    Image:         nginx:1.15.5
    Image ID:      docker-
    pullable://nginx@sha256:d85914d547a6c92faa39ce7058bd7529baacab7e0cd4255442b04577c4d1f424
    State:        Running
    Started:     Fri, 01 Mar 2019 23:56:16 +0000
    Ready:       True
    Mounts:
      /mnt/azure from volume (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-8rv4z (ro)
  [...]
Volumes:
  volume:
    Type:          PersistentVolumeClaim (a reference to a PersistentVolumeClaim in the same namespace)
    ClaimName:    azurefile
    ReadOnly:     false
  [...]
```

## Mount options

The default value for *fileMode* and *dirMode* is *0755* for Kubernetes version 1.9.1 and above. If using a cluster with Kubernetes version 1.8.5 or greater and dynamically creating the persistent volume with a storage class, mount options can be specified on the storage class object. The following example sets *0777*:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=1000
  - gid=1000
  - mfsymlinks
  - nobrl
  - cache=none
parameters:
  skuName: Standard_LRS
```

If using a cluster of version 1.8.0 - 1.8.4, a security context can be specified with the *runAsUser* value set to *0*. For more information on Pod security context, see [Configure a Security Context](#).

## Next steps

For associated best practices, see [Best practices for storage and backups in AKS](#).

Learn more about Kubernetes persistent volumes using Azure Files.

[Kubernetes plugin for Azure Files](#)

# Manually create and use a volume with Azure Files share in Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

Container-based applications often need to access and persist data in an external data volume. If multiple pods need concurrent access to the same storage volume, you can use Azure Files to connect using the [Server Message Block \(SMB\) protocol](#). This article shows you how to manually create an Azure Files share and attach it to a pod in AKS.

For more information on Kubernetes volumes, see [Storage options for applications in AKS](#).

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the [AKS quickstart using the Azure CLI](#) or [using the Azure portal](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an Azure file share

Before you can use Azure Files as a Kubernetes volume, you must create an Azure Storage account and the file share. The following commands create a resource group named *myAKSShare*, a storage account, and a Files share named *aksshare*:

```
# Change these four parameters as needed for your own environment
$AKS_PERS_STORAGE_ACCOUNT_NAME=mystorageaccount$RANDOM
$AKS_PERS_RESOURCE_GROUP=myAKSShare
$AKS_PERS_LOCATION=eastus
$AKS_PERS_SHARE_NAME=aksshare

# Create a resource group
az group create --name $AKS_PERS_RESOURCE_GROUP --location $AKS_PERS_LOCATION

# Create a storage account
az storage account create -n $AKS_PERS_STORAGE_ACCOUNT_NAME -g $AKS_PERS_RESOURCE_GROUP -l $AKS_PERS_LOCATION
--sku Standard_LRS

# Export the connection string as an environment variable, this is used when creating the Azure file share
export AZURE_STORAGE_CONNECTION_STRING=$(az storage account show-connection-string -n
$AKS_PERS_STORAGE_ACCOUNT_NAME -g $AKS_PERS_RESOURCE_GROUP -o tsv)

# Create the file share
az storage share create -n $AKS_PERS_SHARE_NAME --connection-string $AZURE_STORAGE_CONNECTION_STRING

# Get storage account key
$STORAGE_KEY=$(az storage account keys list --resource-group $AKS_PERS_RESOURCE_GROUP --account-name
$AKS_PERS_STORAGE_ACCOUNT_NAME --query "[0].value" -o tsv)

# Echo storage account name and key
echo Storage account name: $AKS_PERS_STORAGE_ACCOUNT_NAME
echo Storage account key: $STORAGE_KEY
```

Make a note of the storage account name and key shown at the end of the script output. These values are needed when you create the Kubernetes volume in one of the following steps.

## Create a Kubernetes secret

Kubernetes needs credentials to access the file share created in the previous step. These credentials are stored in a [Kubernetes secret](#), which is referenced when you create a Kubernetes pod.

Use the `kubectl create secret` command to create the secret. The following example creates a shared named `azure-secret` and populates the `azurerestorageaccountname` and `azurerestorageaccountkey` from the previous step. To use an existing Azure storage account, provide the account name and key.

```
kubectl create secret generic azure-secret --from-literal=azurerestorageaccountname=$AKS_PERS_STORAGE_ACCOUNT_NAME --from-literal=azurerestorageaccountkey=$STORAGE_KEY
```

## Mount the file share as a volume

To mount the Azure Files share into your pod, configure the volume in the container spec. Create a new file named `azure-files-pod.yaml` with the following contents. If you changed the name of the Files share or secret name, update the `shareName` and `secretName`. If desired, update the `mountPath`, which is the path where the Files share is mounted in the pod. For Windows Server containers (currently in preview in AKS), specify a `mountPath` using the Windows path convention, such as '`D:`'.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - image: nginx:1.15.5
      name: mypod
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      volumeMounts:
        - name: azure
          mountPath: /mnt/azure
  volumes:
    - name: azure
      azureFile:
        secretName: azure-secret
        shareName: aksshare
        readOnly: false
```

Use the `kubectl` command to create the pod.

```
kubectl apply -f azure-files-pod.yaml
```

You now have a running pod with an Azure Files share mounted at `/mnt/azure`. You can use `kubectl describe pod mypod` to verify the share is mounted successfully. The following condensed example output shows the volume mounted in the container:

```

Containers:
  mypod:
    Container ID:  docker://86d244cf7c4822401e88f55fd75217d213aa9c3c6a3df169e76e8e25ed28166
    Image:         nginx:1.15.5
    Image ID:      docker-
    pullable://nginx@sha256:9ad0746d8f2ea6df3a17ba89eca40b48c47066dfab55a75e08e2b70fc80d929e
    State:        Running
    Started:     Sat, 02 Mar 2019 00:05:47 +0000
    Ready:       True
    Mounts:
      /mnt/azure from azure (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-z5sd7 (ro)
  [...]
Volumes:
  azure:
    Type:     AzureFile (an Azure File Service mount on the host and bind mount to the pod)
    SecretName:  azure-secret
    ShareName:   aksshare
    ReadOnly:    false
  default-token-z5sd7:
    Type:     Secret (a volume populated by a Secret)
    SecretName: default-token-z5sd7
  [...]

```

## Mount options

The default value for *fileMode* and *dirMode* is 0755 for Kubernetes version 1.9.1 and above. If using a cluster with Kubernetes version 1.8.5 or greater and statically creating the persistent volume object, mount options need to be specified on the *PersistentVolume* object. The following example sets 0777:

```

apiVersion: v1
kind: PersistentVolume
metadata:
  name: azurefile
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  storageClassName: azurefile
  azureFile:
    secretName: azure-secret
    shareName: aksshare
    readOnly: false
  mountOptions:
    - dir_mode=0777
    - file_mode=0777
    - uid=1000
    - gid=1000
    - mfsymlinks
    - nobrl

```

If using a cluster of version 1.8.0 - 1.8.4, a security context can be specified with the *runAsUser* value set to 0. For more information on Pod security context, see [Configure a Security Context](#).

To update your mount options, create a *azurefile-mount-options-pv.yaml* file with a *PersistentVolume*. For example:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: azurefile
spec:
  capacity:
    storage: 5Gi
  accessModes:
    - ReadWriteMany
  storageClassName: azurefile
  azureFile:
    secretName: azure-secret
    shareName: aksshare
    readOnly: false
  mountOptions:
    - dir_mode=0777
    - file_mode=0777
    - uid=1000
    - gid=1000
    - mfsymlinks
    - nobrl
```

Create a `azurefile-mount-options-pvc.yaml` file with a `PersistentVolumeClaim` that uses the `PersistentVolume`. For example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azurefile
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: azurefile
  resources:
    requests:
      storage: 5Gi
```

Use the `kubectl` commands to create the `PersistentVolume` and `PersistentVolumeClaim`.

```
kubectl apply -f azurefile-mount-options-pv.yaml
kubectl apply -f azurefile-mount-options-pvc.yaml
```

Verify your `PersistentVolumeClaim` is created and bound to the `PersistentVolume`.

```
$ kubectl get pvc azurefile
NAME      STATUS  VOLUME      CAPACITY  ACCESS MODES  STORAGECLASS  AGE
azurefile  Bound   azurefile   5Gi       RWX          azurefile    5s
```

Update your container spec to reference your `PersistentVolumeClaim` and update your pod. For example:

```
...
volumes:
- name: azure
  persistentVolumeClaim:
    claimName: azurefile
```

## Next steps

For associated best practices, see [Best practices for storage and backups in AKS](#).

For more information about AKS clusters interact with Azure Files, see the [Kubernetes plugin for Azure Files](#).

# Manually create and use an NFS (Network File System) Linux Server volume with Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

Sharing data between containers is often a necessary component of container-based services and applications. You usually have various pods that need access to the same information on an external persistent volume. While Azure files are an option, creating an NFS Server on an Azure VM is another form of persistent shared storage.

This article will show you how to create an NFS Server on an Ubuntu virtual machine. And also give your AKS containers access to this shared file system.

## Before you begin

This article assumes that you have an existing AKS Cluster. If you need an AKS Cluster, see the AKS quickstart [using the Azure CLI](#) or [using the Azure portal](#).

Your AKS Cluster will need to live in the same or peered virtual networks as the NFS Server. The cluster must be created in an existing VNET, which can be the same VNET as your VM.

The steps for configuring with an existing VNET are described in the documentation: [creating AKS Cluster in existing VNET](#) and [connecting virtual networks with VNET peering](#)

It also assumes you've created an Ubuntu Linux Virtual Machine (for example, 18.04 LTS). Settings and size can be to your liking and can be deployed through Azure. For Linux quickstart, see [Linux VM management](#).

If you deploy your AKS Cluster first, Azure will automatically populate the virtual network field when deploying your Ubuntu machine, making them live within the same VNET. But if you want to work with peered networks instead, consult the documentation above.

## Deploying the NFS Server onto a Virtual Machine

Here is the script to set up an NFS Server within your Ubuntu virtual machine:

```

#!/bin/bash

# This script should be executed on Linux Ubuntu Virtual Machine

EXPORT_DIRECTORY=${1:-/export/data}
DATA_DIRECTORY=${2:-/data}
AKS_SUBNET=${3:-*}

echo "Updating packages"
apt-get -y update

echo "Installing NFS kernel server"

apt-get -y install nfs-kernel-server

echo "Making data directory ${DATA_DIRECTORY}"
mkdir -p ${DATA_DIRECTORY}

echo "Making new directory to be exported and linked to data directory: ${EXPORT_DIRECTORY}"
mkdir -p ${EXPORT_DIRECTORY}

echo "Mount binding ${DATA_DIRECTORY} to ${EXPORT_DIRECTORY}"
mount --bind ${DATA_DIRECTORY} ${EXPORT_DIRECTORY}

echo "Giving 777 permissions to ${EXPORT_DIRECTORY} directory"
chmod 777 ${EXPORT_DIRECTORY}

parentdir=$(dirname "${EXPORT_DIRECTORY}")
echo "Giving 777 permissions to parent: ${parentdir} directory"
chmod 777 $parentdir

echo "Appending bound directories into fstab"
echo "${DATA_DIRECTORY}    ${EXPORT_DIRECTORY}    none    bind  0  0" >> /etc/fstab

echo "Appending localhost and Kubernetes subnet address ${AKS_SUBNET} to exports configuration file"
echo "/export      ${AKS_SUBNET}(rw,async,insecure,fsid=0,crossmnt,no_subtree_check)" >> /etc/exports
echo "/export      localhost(rw,async,insecure,fsid=0,crossmnt,no_subtree_check)" >> /etc/exports

nohup service nfs-kernel-server restart

```

The server will restart (because of the script) and you can mount the NFS Server to AKS.

#### IMPORTANT

Make sure to replace the **AKS\_SUBNET** with the correct one from your cluster or else "\*" will open your NFS Server to all ports and connections.

After you've created your VM, copy the script above into a file. Then, you can move it from your local machine, or wherever the script is, into the VM using:

```
scp /path/to/script_file username@vm-ip-address:/home/{username}
```

Once your script is in your VM, you can ssh into the VM and execute it via the command:

```
sudo ./nfs-server-setup.sh
```

If its execution fails because of a permission denied error, set execution permission via the command:

```
chmod +x ~/nfs-server-setup.sh
```

## Connecting AKS Cluster to NFS Server

We can connect the NFS Server to our cluster by provisioning a persistent volume and persistent volume claim that specifies how to access the volume.

Connecting the two services in the same or peered virtual networks is necessary. Instructions for setting up the cluster in the same VNET are here: [Creating AKS Cluster in existing VNET](#)

Once they are in the same virtual network (or peered), you need to provision a persistent volume and a persistent volume claim in your AKS Cluster. The containers can then mount the NFS drive to their local directory.

Here is an example Kubernetes definition for the persistent volume (This definition assumes your cluster and VM are in the same VNET):

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: <NFS_NAME>
  labels:
    type: nfs
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: <NFS_INTERNAL_IP>
    path: <NFS_EXPORT_FILE_PATH>
```

Replace **NFS\_INTERNAL\_IP**, **NFS\_NAME** and **NFS\_EXPORT\_FILE\_PATH** with NFS Server information.

You'll also need a persistent volume claim file. Here is an example of what to include:

### IMPORTANT

"**storageClassName**" needs to remain an empty string or the claim won't work.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: <NFS_NAME>
spec:
  accessModes:
    - ReadWriteMany
  storageClassName: ""
  resources:
    requests:
      storage: 1Gi
  selector:
    matchLabels:
      type: nfs
```

## Troubleshooting

If you can't connect to the server from a cluster, an issue might be the exported directory, or its parent, doesn't have

sufficient permissions to access the server.

Check that both your export directory and its parent directory have 777 permissions.

You can check permissions by running the command below and the directories should have '`drwxrwxrwx`' permissions:

```
ls -l
```

## More information

To get a full walkthrough or to help you debug your NFS Server setup, here is an in-depth tutorial:

- [NFS Tutorial](#)

## Next steps

For associated best practices, see [Best practices for storage and backups in AKS](#).

# Integrate Azure NetApp Files with Azure Kubernetes Service

2/25/2020 • 5 minutes to read • [Edit Online](#)

Azure NetApp Files is an enterprise-class, high-performance, metered file storage service running on Azure. This article shows you how to integrate Azure NetApp Files with Azure Kubernetes Service (AKS).

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart [using the Azure CLI](#) or [using the Azure portal](#).

### IMPORTANT

Your AKS cluster must also be [in a region that supports Azure NetApp Files](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

### Limitations

The following limitations apply when you use Azure NetApp Files:

- Azure NetApp Files is only available [in selected Azure regions](#).
- Before you can use Azure NetApp Files, you must be granted access to the Azure NetApp Files service. To apply for access, you can use the [Azure NetApp Files waitlist submission form](#). You can't access the Azure NetApp Files service until you receive the official confirmation email from the Azure NetApp Files team.
- Your Azure NetApp Files service must be created in the same virtual network as your AKS cluster.
- After the initial deployment of an AKS cluster, only static provisioning for Azure NetApp Files is supported.
- To use dynamic provisioning with Azure NetApp Files, install and configure [NetApp Trident](#) version 19.07 or later.

## Configure Azure NetApp Files

### IMPORTANT

Before you can register the *Microsoft.NetApp* resource provider, you must complete the [Azure NetApp Files waitlist submission form](#) for your subscription. You can't register the resource provider until you receive the official confirmation email from the Azure NetApp Files team.

Register the *Microsoft.NetApp* resource provider:

```
az provider register --namespace Microsoft.NetApp --wait
```

### NOTE

This can take some time to complete.

When you create an Azure NetApp account for use with AKS, you need to create the account in the **node** resource group. First, get the resource group name with the [az aks show](#) command and add the `--query nodeResourceGroup` query parameter. The following example gets the node resource group for the AKS cluster named *myAKSCluster* in the resource group name *myResourceGroup*:

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv  
MC_myResourceGroup_myAKSCluster_eastus
```

Create an Azure NetApp Files account in the **node** resource group and same region as your AKS cluster using [az netappfiles account create](#). The following example creates an account named *myaccount1* in the *MC\_myResourceGroup\_myAKSCluster\_eastus* resource group and *eastus* region:

```
az netappfiles account create \  
--resource-group MC_myResourceGroup_myAKSCluster_eastus \  
--location eastus \  
--account-name myaccount1
```

Create a new capacity pool by using [az netappfiles pool create](#). The following example creates a new capacity pool named *mypool1* with 4 TB in size and *Premium* service level:

```
az netappfiles pool create \  
--resource-group MC_myResourceGroup_myAKSCluster_eastus \  
--location eastus \  
--account-name myaccount1 \  
--pool-name mypool1 \  
--size 4 \  
--service-level Premium
```

Create a subnet to [delegate to Azure NetApp Files](#) using [az network vnet subnet create](#). *This subnet must be in the same virtual network as your AKS cluster.*

```
RESOURCE_GROUP=MC_myResourceGroup_myAKSCluster_eastus  
VNET_NAME=$(az network vnet list --resource-group $RESOURCE_GROUP --query [].name -o tsv)  
VNET_ID=$(az network vnet show --resource-group $RESOURCE_GROUP --name $VNET_NAME --query "id" -o tsv)  
SUBNET_NAME=MyNetAppSubnet  
az network vnet subnet create \  
--resource-group $RESOURCE_GROUP \  
--vnet-name $VNET_NAME \  
--name $SUBNET_NAME \  
--delegations "Microsoft.NetApp/volumes" \  
--address-prefixes 10.0.0.0/28
```

Create a volume by using [az netappfiles volume create](#).

```

RESOURCE_GROUP=MC_myResourceGroup_myAKSCluster_eastus
LOCATION=eastus
ANF_ACCOUNT_NAME=myaccount1
POOL_NAME=mypool1
SERVICE_LEVEL=Premium
VNET_NAME=$(az network vnet list --resource-group $RESOURCE_GROUP --query [].name -o tsv)
VNET_ID=$(az network vnet show --resource-group $RESOURCE_GROUP --name $VNET_NAME --query "id" -o tsv)
SUBNET_NAME=MyNetAppSubnet
SUBNET_ID=$(az network vnet subnet show --resource-group $RESOURCE_GROUP --vnet-name $VNET_NAME --name $SUBNET_NAME --query "id" -o tsv)
VOLUME_SIZE_GiB=100 # 100 GiB
UNIQUE_FILE_PATH="myfilepath2" # Please note that creation token needs to be unique within all ANF Accounts

az netappfiles volume create \
    --resource-group $RESOURCE_GROUP \
    --location $LOCATION \
    --account-name $ANF_ACCOUNT_NAME \
    --pool-name $POOL_NAME \
    --name "myvol1" \
    --service-level $SERVICE_LEVEL \
    --vnet $VNET_ID \
    --subnet $SUBNET_ID \
    --usage-threshold $VOLUME_SIZE_GiB \
    --creation-token $UNIQUE_FILE_PATH \
    --protocol-types "NFSv3"

```

## Create the PersistentVolume

List the details of your volume using [az netappfiles volume show](#)

```

$ az netappfiles volume show --resource-group $RESOURCE_GROUP --account-name $ANF_ACCOUNT_NAME --pool-name
$POOL_NAME --volume-name "myvol1"

{
  ...
  "creationToken": "myfilepath2",
  ...
  "mountTargets": [
    {
      ...
      "ipAddress": "10.0.0.4",
      ...
    }
  ],
  ...
}

```

Create a `pv-nfs.yaml` defining a PersistentVolume. Replace `path` with the `creationToken` and `server` with `ipAddress` from the previous command. For example:

```
---  
apiVersion: v1  
kind: PersistentVolume  
metadata:  
  name: pv-nfs  
spec:  
  capacity:  
    storage: 100Gi  
  accessModes:  
    - ReadWriteMany  
  nfs:  
    server: 10.0.0.4  
    path: /myfilepath2
```

Update the *server* and *path* to the values of your NFS (Network File System) volume you created in the previous step. Create the PersistentVolume with the [kubectl apply](#) command:

```
kubectl apply -f pv-nfs.yaml
```

Verify the *Status* of the PersistentVolume is *Available* using the [kubectl describe](#) command:

```
kubectl describe pv pv-nfs
```

## Create the PersistentVolumeClaim

Create a `pvc-nfs.yaml` defining a PersistentVolume. For example:

```
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: pvc-nfs  
spec:  
  accessModes:  
    - ReadWriteMany  
  storageClassName: ""  
  resources:  
    requests:  
      storage: 1Gi
```

Create the PersistentVolumeClaim with the [kubectl apply](#) command:

```
kubectl apply -f pvc-nfs.yaml
```

Verify the *Status* of the PersistentVolumeClaim is *Bound* using the [kubectl describe](#) command:

```
kubectl describe pvc pvc-nfs
```

## Mount with a pod

Create a `nginx-nfs.yaml` defining a pod that uses the PersistentVolumeClaim. For example:

```
kind: Pod
apiVersion: v1
metadata:
  name: nginx-nfs
spec:
  containers:
  - image: nginx
    name: nginx-nfs
    command:
    - "/bin/sh"
    - "-c"
    - "while true; do echo $(date) >> /mnt/azure/outfile; sleep 1; done"
  volumeMounts:
  - name: disk01
    mountPath: /mnt/azure
  volumes:
  - name: disk01
    persistentVolumeClaim:
      claimName: pvc-nfs
```

Create the pod with the [kubectl apply](#) command:

```
kubectl apply -f nginx-nfs.yaml
```

Verify the pod is *Running* using the [kubectl describe](#) command:

```
kubectl describe pod nginx-nfs
```

Verify your volume has been mounted in the pod by using [kubectl exec](#) to connect to the pod then `df -h` to check if the volume is mounted.

```
$ kubectl exec -it nginx-nfs -- bash
root@nginx-nfs:/# df -h
Filesystem           Size   Used  Avail Use% Mounted on
...
10.0.0.4:/myfilepath2  100T   384K  100T   1% /mnt/azure
...
```

## Next steps

For more information on Azure NetApp Files, see [What is Azure NetApp Files](#). For more information on using NFS with AKS, see [Manually create and use an NFS \(Network File System\) Linux Server volume with Azure Kubernetes Service \(AKS\)](#).

# Use kubenet networking with your own IP address ranges in Azure Kubernetes Service (AKS)

2/25/2020 • 10 minutes to read • [Edit Online](#)

By default, AKS clusters use [kubenet](#), and an Azure virtual network and subnet are created for you. With *kubenet*, nodes get an IP address from the Azure virtual network subnet. Pods receive an IP address from a logically different address space to the Azure virtual network subnet of the nodes. Network address translation (NAT) is then configured so that the pods can reach resources on the Azure virtual network. The source IP address of the traffic is NAT'd to the node's primary IP address. This approach greatly reduces the number of IP addresses that you need to reserve in your network space for pods to use.

With [Azure Container Networking Interface \(CNI\)](#), every pod gets an IP address from the subnet and can be accessed directly. These IP addresses must be unique across your network space, and must be planned in advance. Each node has a configuration parameter for the maximum number of pods that it supports. The equivalent number of IP addresses per node are then reserved up front for that node. This approach requires more planning, and often leads to IP address exhaustion or the need to rebuild clusters in a larger subnet as your application demands grow.

This article shows you how to use *kubenet* networking to create and use a virtual network subnet for an AKS cluster. For more information on network options and considerations, see [Network concepts for Kubernetes and AKS](#).

## Prerequisites

- The virtual network for the AKS cluster must allow outbound internet connectivity.
- Don't create more than one AKS cluster in the same subnet.
- AKS clusters may not use `169.254.0.0/16`, `172.30.0.0/16`, `172.31.0.0/16`, or `192.0.2.0/24` for the Kubernetes service address range.
- The service principal used by the AKS cluster must have at least [Network Contributor](#) permissions on the subnet within your virtual network. If you wish to define a [custom role](#) instead of using the built-in Network Contributor role, the following permissions are required:
  - `Microsoft.Network/virtualNetworks/subnets/join/action`
  - `Microsoft.Network/virtualNetworks/subnets/read`

### WARNING

To use Windows Server node pools (currently in preview in AKS), you must use Azure CNI. The use of kubenet as the network model is not available for Windows Server containers.

## Before you begin

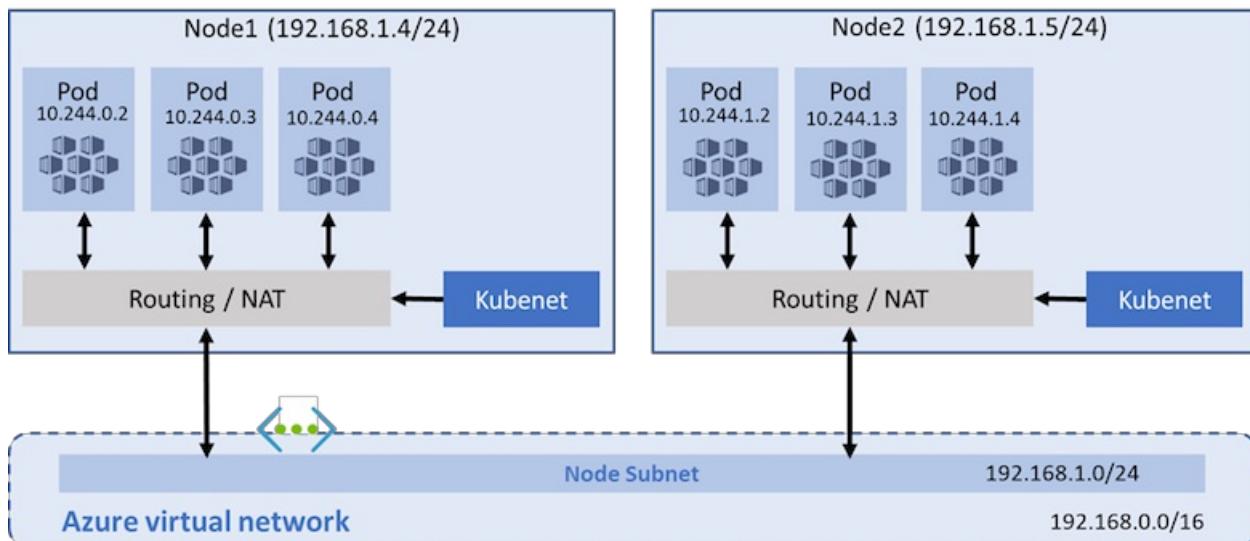
You need the Azure CLI version 2.0.65 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Overview of kubenet networking with your own subnet

In many environments, you have defined virtual networks and subnets with allocated IP address ranges. These virtual network resources are used to support multiple services and applications. To provide network connectivity,

AKS clusters can use *kubenet* (basic networking) or Azure CNI (advanced networking).

With *kubenet*, only the nodes receive an IP address in the virtual network subnet. Pods can't communicate directly with each other. Instead, User Defined Routing (UDR) and IP forwarding is used for connectivity between pods across nodes. You could also deploy pods behind a service that receives an assigned IP address and load balances traffic for the application. The following diagram shows how the AKS nodes receive an IP address in the virtual network subnet, but not the pods:



Azure supports a maximum of 400 routes in a UDR, so you can't have an AKS cluster larger than 400 nodes. AKS [Virtual Nodes](#) and Azure Network Policies aren't supported with *kubenet*. You can use [Calico Network Policies](#), as they are supported with *kubenet*.

With *Azure CNI*, each pod receives an IP address in the IP subnet, and can directly communicate with other pods and services. Your clusters can be as large as the IP address range you specify. However, the IP address range must be planned in advance, and all of the IP addresses are consumed by the AKS nodes based on the maximum number of pods that they can support. Advanced network features and scenarios such as [Virtual Nodes](#) or Network Policies (either Azure or Calico) are supported with *Azure CNI*.

#### IP address availability and exhaustion

With *Azure CNI*, a common issue is the assigned IP address range is too small to then add additional nodes when you scale or upgrade a cluster. The network team may also not be able to issue a large enough IP address range to support your expected application demands.

As a compromise, you can create an AKS cluster that uses *kubenet* and connect to an existing virtual network subnet. This approach lets the nodes receive defined IP addresses, without the need to reserve a large number of IP addresses up front for all of the potential pods that could run in the cluster.

With *kubenet*, you can use a much smaller IP address range and be able to support large clusters and application demands. For example, even with a /27 IP address range, you could run a 20-25 node cluster with enough room to scale or upgrade. This cluster size would support up to 2,200-2,750 pods (with a default maximum of 110 pods per node). The maximum number of pods per node that you can configure with *kubenet* in AKS is 110.

The following basic calculations compare the difference in network models:

- **kubenet** - a simple /24 IP address range can support up to 251 nodes in the cluster (each Azure virtual network subnet reserves the first three IP addresses for management operations)
  - This node count could support up to 27,610 pods (with a default maximum of 110 pods per node with *kubenet*)
- **Azure CNI** - that same basic /24 subnet range could only support a maximum of 8 nodes in the cluster
  - This node count could only support up to 240 pods (with a default maximum of 30 pods per node with *Azure CNI*)

**NOTE**

These maximums don't take into account upgrade or scale operations. In practice, you can't run the maximum number of nodes that the subnet IP address range supports. You must leave some IP addresses available for use during scale or upgrade operations.

## Virtual network peering and ExpressRoute connections

To provide on-premises connectivity, both *kubenet* and *Azure-CNI* network approaches can use [Azure virtual network peering](#) or [ExpressRoute connections](#). Plan your IP address ranges carefully to prevent overlap and incorrect traffic routing. For example, many on-premises networks use a *10.0.0.0/8* address range that is advertised over the ExpressRoute connection. It's recommended to create your AKS clusters into Azure virtual network subnets outside of this address range, such as *172.16.0.0/16*.

## Choose a network model to use

The choice of which network plugin to use for your AKS cluster is usually a balance between flexibility and advanced configuration needs. The following considerations help outline when each network model may be the most appropriate.

Use *kubenet* when:

- You have limited IP address space.
- Most of the pod communication is within the cluster.
- You don't need advanced AKS features such as virtual nodes or Azure Network Policy. Use [Calico network policies](#).

Use *Azure CNI* when:

- You have available IP address space.
- Most of the pod communication is to resources outside of the cluster.
- You don't want to manage the UDRs.
- You need AKS advanced features such as virtual nodes or Azure Network Policy. Use [Calico network policies](#).

For more information to help you decide which network model to use, see [Compare network models and their support scope](#).

## Create a virtual network and subnet

To get started with using *kubenet* and your own virtual network subnet, first create a resource group using the [az group create](#) command. The following example creates a resource group named *myResourceGroup* in the *eastus* location:

```
az group create --name myResourceGroup --location eastus
```

If you don't have an existing virtual network and subnet to use, create these network resources using the [az network vnet create](#) command. In the following example, the virtual network is named *myVnet* with the address prefix of *192.168.0.0/16*. A subnet is created named *myAKSSubnet* with the address prefix *192.168.1.0/24*.

```
az network vnet create \
--resource-group myResourceGroup \
--name myAKSVnet \
--address-prefixes 192.168.0.0/16 \
--subnet-name myAKSSubnet \
--subnet-prefix 192.168.1.0/24
```

## Create a service principal and assign permissions

To allow an AKS cluster to interact with other Azure resources, an Azure Active Directory service principal is used. The service principal needs to have permissions to manage the virtual network and subnet that the AKS nodes use. To create a service principal, use the [az ad sp create-for-rbac](#) command:

```
az ad sp create-for-rbac --skip-assignment
```

The following example output shows the application ID and password for your service principal. These values are used in additional steps to assign a role to the service principal and then create the AKS cluster:

```
$ az ad sp create-for-rbac --skip-assignment

{
  "appId": "476b3636-5eda-4c0e-9751-849e70b5cfad",
  "displayName": "azure-cli-2019-01-09-22-29-24",
  "name": "http://azure-cli-2019-01-09-22-29-24",
  "password": "a1024cd7-af7b-469f-8fd7-b293ecbb174e",
  "tenant": "72f998bf-85f1-41cf-92ab-2e7cd014db46"
}
```

To assign the correct delegations in the remaining steps, use the [az network vnet show](#) and [az network vnet subnet show](#) commands to get the required resource IDs. These resource IDs are stored as variables and referenced in the remaining steps:

```
VNET_ID=$(az network vnet show --resource-group myResourceGroup --name myAKSVnet --query id -o tsv)
SUBNET_ID=$(az network vnet subnet show --resource-group myResourceGroup --vnet-name myAKSVnet --name myAKSSubnet --query id -o tsv)
```

Now assign the service principal for your AKS cluster *Contributor* permissions on the virtual network using the [az role assignment create](#) command. Provide your own <appId> as shown in the output from the previous command to create the service principal:

```
az role assignment create --assignee <appId> --scope $VNET_ID --role Contributor
```

## Create an AKS cluster in the virtual network

You've now created a virtual network and subnet, and created and assigned permissions for a service principal to use those network resources. Now create an AKS cluster in your virtual network and subnet using the [az aks create](#) command. Define your own service principal <appId> and <password>, as shown in the output from the previous command to create the service principal.

The following IP address ranges are also defined as part of the cluster create process:

- The `--service-cidr` is used to assign internal services in the AKS cluster an IP address. This IP address range should be an address space that isn't in use elsewhere in your network environment. This range includes

any on-premises network ranges if you connect, or plan to connect, your Azure virtual networks using Express Route or a Site-to-Site VPN connection.

- The `--dns-service-ip` address should be the `.10` address of your service IP address range.
- The `--pod-cidr` should be a large address space that isn't in use elsewhere in your network environment. This range includes any on-premises network ranges if you connect, or plan to connect, your Azure virtual networks using Express Route or a Site-to-Site VPN connection.
  - This address range must be large enough to accommodate the number of nodes that you expect to scale up to. You can't change this address range once the cluster is deployed if you need more addresses for additional nodes.
  - The pod IP address range is used to assign a /24 address space to each node in the cluster. In the following example, the `--pod-cidr` of `10.244.0.0/16` assigns the first node `10.244.0.0/24`, the second node `10.244.1.0/24`, and the third node `10.244.2.0/24`.
  - As the cluster scales or upgrades, the Azure platform continues to assign a pod IP address range to each new node.
- The `--docker-bridge-address` lets the AKS nodes communicate with the underlying management platform. This IP address must not be within the virtual network IP address range of your cluster, and shouldn't overlap with other address ranges in use on your network.

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--node-count 3 \
--network-plugin kubenet \
--service-cidr 10.0.0.0/16 \
--dns-service-ip 10.0.0.10 \
--pod-cidr 10.244.0.0/16 \
--docker-bridge-address 172.17.0.1/16 \
--vnet-subnet-id $SUBNET_ID \
--service-principal <appId> \
--client-secret <password>
```

#### NOTE

If you wish to enable an AKS cluster to include a [Calico network policy](#) you can use the following command.

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--node-count 3 \
--network-plugin kubenet --network-policy calico \
--service-cidr 10.0.0.0/16 \
--dns-service-ip 10.0.0.10 \
--pod-cidr 10.244.0.0/16 \
--docker-bridge-address 172.17.0.1/16 \
--vnet-subnet-id $SUBNET_ID \
--service-principal <appId> \
--client-secret <password>
```

When you create an AKS cluster, a network security group and route table are created. These network resources are managed by the AKS control plane. The network security group is automatically associated with the virtual NICs on your nodes. The route table is automatically associated with the virtual network subnet. Network security group rules and route tables are automatically updated as you create and expose services.

## Next steps

With an AKS cluster deployed into your existing virtual network subnet, you can now use the cluster as normal. Get started with [building apps using Azure Dev Spaces](#) or [using Draft](#), or [deploy apps using Helm](#).

# Configure Azure CNI networking in Azure Kubernetes Service (AKS)

2/25/2020 • 12 minutes to read • [Edit Online](#)

By default, AKS clusters use [kubenet](#), and a virtual network and subnet are created for you. With *kubenet*, nodes get an IP address from a virtual network subnet. Network address translation (NAT) is then configured on the nodes, and pods receive an IP address "hidden" behind the node IP. This approach reduces the number of IP addresses that you need to reserve in your network space for pods to use.

With [Azure Container Networking Interface \(CNI\)](#), every pod gets an IP address from the subnet and can be accessed directly. These IP addresses must be unique across your network space, and must be planned in advance. Each node has a configuration parameter for the maximum number of pods that it supports. The equivalent number of IP addresses per node are then reserved up front for that node. This approach requires more planning, and often leads to IP address exhaustion or the need to rebuild clusters in a larger subnet as your application demands grow.

This article shows you how to use *Azure CNI* networking to create and use a virtual network subnet for an AKS cluster. For more information on network options and considerations, see [Network concepts for Kubernetes and AKS](#).

## Prerequisites

- The virtual network for the AKS cluster must allow outbound internet connectivity.
- AKS clusters may not use `169.254.0.0/16`, `172.30.0.0/16`, `172.31.0.0/16`, or `192.0.2.0/24` for the Kubernetes service address range.
- The service principal used by the AKS cluster must have at least [Network Contributor](#) permissions on the subnet within your virtual network. If you wish to define a [custom role](#) instead of using the built-in Network Contributor role, the following permissions are required:
  - `Microsoft.Network/virtualNetworks/subnets/join/action`
  - `Microsoft.Network/virtualNetworks/subnets/read`

## Plan IP addressing for your cluster

Clusters configured with Azure CNI networking require additional planning. The size of your virtual network and its subnet must accommodate the number of pods you plan to run and the number of nodes for the cluster.

IP addresses for the pods and the cluster's nodes are assigned from the specified subnet within the virtual network. Each node is configured with a primary IP address. By default, 30 additional IP addresses are pre-configured by Azure CNI that are assigned to pods scheduled on the node. When you scale out your cluster, each node is similarly configured with IP addresses from the subnet. You can also view the [maximum pods per node](#).

## IMPORTANT

The number of IP addresses required should include considerations for upgrade and scaling operations. If you set the IP address range to only support a fixed number of nodes, you cannot upgrade or scale your cluster.

- When you **upgrade** your AKS cluster, a new node is deployed into the cluster. Services and workloads begin to run on the new node, and an older node is removed from the cluster. This rolling upgrade process requires a minimum of one additional block of IP addresses to be available. Your node count is then  $n + 1$ .
  - This consideration is particularly important when you use Windows Server node pools (currently in preview in AKS). Windows Server nodes in AKS do not automatically apply Windows Updates, instead you perform an upgrade on the node pool. This upgrade deploys new nodes with the latest Window Server 2019 base node image and security patches. For more information on upgrading a Windows Server node pool, see [Upgrade a node pool in AKS](#).
- When you **scale** an AKS cluster, a new node is deployed into the cluster. Services and workloads begin to run on the new node. Your IP address range needs to take into considerations how you may want to scale up the number of nodes and pods your cluster can support. One additional node for upgrade operations should also be included. Your node count is then  $n + \text{number-of-additional-scaled-nodes-you-anticipate} + 1$ .

If you expect your nodes to run the maximum number of pods, and regularly destroy and deploy pods, you should also factor in some additional IP addresses per node. These additional IP addresses take into consideration it may take a few seconds for a service to be deleted and the IP address released for a new service to be deployed and acquire the address.

The IP address plan for an AKS cluster consists of a virtual network, at least one subnet for nodes and pods, and a Kubernetes service address range.

ADDRESS RANGE / AZURE RESOURCE	LIMITS AND SIZING
Virtual network	The Azure virtual network can be as large as /8, but is limited to 65,536 configured IP addresses.
Subnet	<p>Must be large enough to accommodate the nodes, pods, and all Kubernetes and Azure resources that might be provisioned in your cluster. For example, if you deploy an internal Azure Load Balancer, its front-end IPs are allocated from the cluster subnet, not public IPs. The subnet size should also take into account upgrade operations or future scaling needs.</p> <p>To calculate the <i>minimum</i> subnet size including an additional node for upgrade operations:</p> $(\text{number of nodes} + 1) + ((\text{number of nodes} + 1) * \text{maximum pods per node that you configure})$ <p>Example for a 50 node cluster: <math>(51) + (51 * 30 \text{ (default)}) = 1,581</math> (/21 or larger)</p> <p>Example for a 50 node cluster that also includes provision to scale up an additional 10 nodes: <math>(61) + (61 * 30 \text{ (default)}) = 1,891</math> (/21 or larger)</p> <p>If you don't specify a maximum number of pods per node when you create your cluster, the maximum number of pods per node is set to 30. The minimum number of IP addresses required is based on that value. If you calculate your minimum IP address requirements on a different maximum value, see <a href="#">how to configure the maximum number of pods per node</a> to set this value when you deploy your cluster.</p>

ADDRESS RANGE / AZURE RESOURCE	LIMITS AND SIZING
Kubernetes service address range	This range should not be used by any network element on or connected to this virtual network. Service address CIDR must be smaller than /12. You can reuse this range across different AKS clusters.
Kubernetes DNS service IP address	IP address within the Kubernetes service address range that will be used by cluster service discovery (kube-dns). Don't use the first IP address in your address range, such as .1. The first address in your subnet range is used for the <code>kubernetes.default.svc.cluster.local</code> address.
Docker bridge address	The Docker bridge network address represents the default <code>docker0</code> bridge network address present in all Docker installations. While <code>docker0</code> bridge is not used by AKS clusters or the pods themselves, you must set this address to continue to support scenarios such as <code>docker build</code> within the AKS cluster. It is required to select a CIDR for the Docker bridge network address because otherwise Docker will pick a subnet automatically which could conflict with other CIDRs. You must pick an address space that does not collide with the rest of the CIDRs on your networks, including the cluster's service CIDR and pod CIDR. Default of 172.17.0.1/16. You can reuse this range across different AKS clusters.

## Maximum pods per node

The maximum number of pods per node in an AKS cluster is 250. The *default* maximum number of pods per node varies between *kubenet* and *Azure CNI* networking, and the method of cluster deployment.

DEPLOYMENT METHOD	KUBENET DEFAULT	AZURE CNI DEFAULT	CONFIGURABLE AT DEPLOYMENT
Azure CLI	110	30	Yes (up to 250)
Resource Manager template	110	30	Yes (up to 250)
Portal	110	30	No

### Configure maximum - new clusters

You're able to configure the maximum number of pods per node *only at cluster deployment time*. If you deploy with the Azure CLI or with a Resource Manager template, you can set the maximum pods per node value as high as 250.

A minimum value for maximum pods per node is enforced to guarantee space for system pods critical to cluster health. The minimum value that can be set for maximum pods per node is 10 if and only if the configuration of each node pool has space for a minimum of 30 pods. For example, setting the maximum pods per node to the minimum of 10 requires each individual node pool to have a minimum of 3 nodes. This requirement applies for each new node pool created as well, so if 10 is defined as maximum pods per node each subsequent node pool added must have at least 3 nodes.

NETWORKING	MINIMUM	MAXIMUM
Azure CNI	10	250

Networking	Minimum	Maximum
Kubenet	10	110

#### NOTE

The minimum value in the table above is strictly enforced by the AKS service. You can not set a maxPods value lower than the minimum shown as doing so can prevent the cluster from starting.

- **Azure CLI:** Specify the `--max-pods` argument when you deploy a cluster with the `az aks create` command. The maximum value is 250.
- **Resource Manager template:** Specify the `maxPods` property in the `ManagedClusterAgentPoolProfile` object when you deploy a cluster with a Resource Manager template. The maximum value is 250.
- **Azure portal:** You can't change the maximum number of pods per node when you deploy a cluster with the Azure portal. Azure CNI networking clusters are limited to 30 pods per node when you deploy using the Azure portal.

#### Configure maximum - existing clusters

You can't change the maximum pods per node on an existing AKS cluster. You can adjust the number only when you initially deploy the cluster.

## Deployment parameters

When you create an AKS cluster, the following parameters are configurable for Azure CNI networking:

**Virtual network:** The virtual network into which you want to deploy the Kubernetes cluster. If you want to create a new virtual network for your cluster, select *Create new* and follow the steps in the *Create virtual network* section. For information about the limits and quotas for an Azure virtual network, see [Azure subscription and service limits, quotas, and constraints](#).

**Subnet:** The subnet within the virtual network where you want to deploy the cluster. If you want to create a new subnet in the virtual network for your cluster, select *Create new* and follow the steps in the *Create subnet* section. For hybrid connectivity, the address range shouldn't overlap with any other virtual networks in your environment.

**Kubernetes service address range:** This is the set of virtual IPs that Kubernetes assigns to internal `services` in your cluster. You can use any private address range that satisfies the following requirements:

- Must not be within the virtual network IP address range of your cluster
- Must not overlap with any other virtual networks with which the cluster virtual network peers
- Must not overlap with any on-premises IPs
- Must not be within the ranges `169.254.0.0/16`, `172.30.0.0/16`, `172.31.0.0/16`, or `192.0.2.0/24`

Although it's technically possible to specify a service address range within the same virtual network as your cluster, doing so is not recommended. Unpredictable behavior can result if overlapping IP ranges are used. For more information, see the [FAQ](#) section of this article. For more information on Kubernetes services, see [Services](#) in the Kubernetes documentation.

**Kubernetes DNS service IP address:** The IP address for the cluster's DNS service. This address must be within the *Kubernetes service address range*. Don't use the first IP address in your address range, such as `.1`. The first address in your subnet range is used for the `kubernetes.default.svc.cluster.local` address.

**Docker Bridge address:** The Docker bridge network address represents the default `docker0` bridge network address present in all Docker installations. While `docker0` bridge is not used by AKS clusters or the pods themselves, you must set this address to continue to support scenarios such as `docker build` within the AKS

cluster. It is required to select a CIDR for the Docker bridge network address because otherwise Docker will pick a subnet automatically which could conflict with other CIDRs. You must pick an address space that does not collide with the rest of the CIDRs on your networks, including the cluster's service CIDR and pod CIDR.

## Configure networking - CLI

When you create an AKS cluster with the Azure CLI, you can also configure Azure CNI networking. Use the following commands to create a new AKS cluster with Azure CNI networking enabled.

First, get the subnet resource ID for the existing subnet into which the AKS cluster will be joined:

```
$ az network vnet subnet list \
--resource-group myVnet \
--vnet-name myVnet \
--query "[0].id" --output tsv

/subscriptions/<guid>/resourceGroups/myVnet/providers/Microsoft.Network/virtualNetworks/myVnet/subnets/default
```

Use the `az aks create` command with the `--network-plugin azure` argument to create a cluster with advanced networking. Update the `--vnet-subnet-id` value with the subnet ID collected in the previous step:

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--network-plugin azure \
--vnet-subnet-id <subnet-id> \
--docker-bridge-address 172.17.0.1/16 \
--dns-service-ip 10.2.0.10 \
--service-cidr 10.2.0.0/24 \
--generate-ssh-keys
```

## Configure networking - portal

The following screenshot from the Azure portal shows an example of configuring these settings during AKS cluster creation:

## Create Kubernetes cluster

[Basics](#) [Networking](#) [Monitoring](#) [Tags](#) [Review + create](#)

You can enable Http ingress routing and choose between two networking options for Azure Kubernetes Services - "Basic" and "Advanced".

- "Basic" networking sets up a simple default config with a VNet and internal IP addresses.
- "Advanced" networking provides you the ability to configure your own VNet, providing pods automatic connectivity to VNet resources and full integration with VNet features.

[Learn more about networking in Azure Kubernetes Service](#)

Http application routing <small>i</small>	<input checked="" type="radio"/> No <input type="radio"/> Yes
Network configuration <small>i</small>	<input type="radio"/> Basic <input checked="" type="radio"/> Advanced
* Virtual network <small>i</small>	aks-vnet-16033614
<a href="#">Create new</a>	
* Subnet <small>i</small>	aks-subnet
<a href="#">Create new</a>	
* Kubernetes service address range <small>i</small>	10.0.0.0/16 ✓
* Kubernetes DNS service IP address <small>i</small>	10.0.0.10
* Docker Bridge address <small>i</small>	172.17.0.1/16 ✓

## Frequently asked questions

The following questions and answers apply to the **Azure CNI** networking configuration.

- *Can I deploy VMs in my cluster subnet?*

No. Deploying VMs in the subnet used by your Kubernetes cluster is not supported. VMs may be deployed in the same virtual network, but in a different subnet.

- *Can I configure per-pod network policies?*

Yes, Kubernetes network policy is available in AKS. To get started, see [Secure traffic between pods by using network policies in AKS](#).

- *Is the maximum number of pods deployable to a node configurable?*

Yes, when you deploy a cluster with the Azure CLI or a Resource Manager template. See [Maximum pods per node](#).

You can't change the maximum number of pods per node on an existing cluster.

- *How do I configure additional properties for the subnet that I created during AKS cluster creation? For example, service endpoints.*

The complete list of properties for the virtual network and subnets that you create during AKS cluster creation can be configured in the standard virtual network configuration page in the Azure portal.

- *Can I use a different subnet within my cluster virtual network for the Kubernetes service address range?*

It's not recommended, but this configuration is possible. The service address range is a set of virtual IPs (VIPs) that Kubernetes assigns to internal services in your cluster. Azure Networking has no visibility into the service IP range of the Kubernetes cluster. Because of the lack of visibility into the cluster's service

address range, it's possible to later create a new subnet in the cluster virtual network that overlaps with the service address range. If such an overlap occurs, Kubernetes could assign a service an IP that's already in use by another resource in the subnet, causing unpredictable behavior or failures. By ensuring you use an address range outside the cluster's virtual network, you can avoid this overlap risk.

## Next steps

Learn more about networking in AKS in the following articles:

- [Use a static IP address with the Azure Kubernetes Service \(AKS\) load balancer](#)
- [Use an internal load balancer with Azure Container Service \(AKS\)](#)
- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates](#)
- [Create an ingress controller with a static public IP and configure Let's Encrypt to automatically generate TLS certificates](#)

### AKS Engine

[Azure Kubernetes Service Engine \(AKS Engine\)](#) is an open-source project that generates Azure Resource Manager templates you can use for deploying Kubernetes clusters on Azure.

Kubernetes clusters created with AKS Engine support both the [kubenet](#) and [Azure CNI](#) plugins. As such, both networking scenarios are supported by AKS Engine.

# Use an internal load balancer with Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

To restrict access to your applications in Azure Kubernetes Service (AKS), you can create and use an internal load balancer. An internal load balancer makes a Kubernetes service accessible only to applications running in the same virtual network as the Kubernetes cluster. This article shows you how to create and use an internal load balancer with Azure Kubernetes Service (AKS).

## NOTE

Azure Load Balancer is available in two SKUs - *Basic* and *Standard*. By default, the Standard SKU is used when you create an AKS cluster. When creating a Service with type as LoadBalancer, you will get the same LB type as when you provision the cluster. For more information, see [Azure load balancer SKU comparison](#).

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart [using the Azure CLI](#) or [using the Azure portal](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

The AKS cluster service principal needs permission to manage network resources if you use an existing subnet or resource group. In general, assign the *Network contributor* role to your service principal on the delegated resources. For more information on permissions, see [Delegate AKS access to other Azure resources](#).

## Create an internal load balancer

To create an internal load balancer, create a service manifest named `internal-lb.yaml` with the service type *LoadBalancer* and the *azure-load-balancer-internal* annotation as shown in the following example:

```
apiVersion: v1
kind: Service
metadata:
  name: internal-app
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: internal-app
```

Deploy the internal load balancer using the [kubectl apply](#) and specify the name of your YAML manifest:

```
kubectl apply -f internal-lb.yaml
```

An Azure load balancer is created in the node resource group and connected to the same virtual network as the

AKS cluster.

When you view the service details, the IP address of the internal load balancer is shown in the *EXTERNAL-IP* column. In this context, *External* is in relation to the external interface of the load balancer, not that it receives a public, external IP address. It may take a minute or two for the IP address to change from <pending> to an actual internal IP address, as shown in the following example:

```
$ kubectl get service internal-app
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
internal-app	LoadBalancer	10.0.248.59	10.240.0.7	80:30555/TCP	2m

## Specify an IP address

If you would like to use a specific IP address with the internal load balancer, add the *loadBalancerIP* property to the load balancer YAML manifest. The specified IP address must reside in the same subnet as the AKS cluster and must not already be assigned to a resource.

```
apiVersion: v1
kind: Service
metadata:
  name: internal-app
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
spec:
  type: LoadBalancer
  loadBalancerIP: 10.240.0.25
  ports:
  - port: 80
  selector:
    app: internal-app
```

When deployed and you view the service details, the IP address in the *EXTERNAL-IP* column reflects your specified IP address:

```
$ kubectl get service internal-app
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
internal-app	LoadBalancer	10.0.184.168	10.240.0.25	80:30225/TCP	4m

## Use private networks

When you create your AKS cluster, you can specify advanced networking settings. This approach lets you deploy the cluster into an existing Azure virtual network and subnets. One scenario is to deploy your AKS cluster into a private network connected to your on-premises environment and run services only accessible internally. For more information, see [configure your own virtual network subnets with Kubenet or Azure CNI](#).

No changes to the previous steps are needed to deploy an internal load balancer in an AKS cluster that uses a private network. The load balancer is created in the same resource group as your AKS cluster but connected to your private virtual network and subnet, as shown in the following example:

```
$ kubectl get service internal-app
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
internal-app	LoadBalancer	10.1.15.188	10.0.0.35	80:31669/TCP	1m

#### NOTE

You may need to grant the service principal for your AKS cluster the *Network Contributor* role to the resource group where your Azure virtual network resources are deployed. View the service principal with `az aks show`, such as  
`az aks show --resource-group myResourceGroup --name myAKSCluster --query "servicePrincipalProfile.clientId"`.  
To create a role assignment, use the [az role assignment create](#) command.

## Specify a different subnet

To specify a subnet for your load balancer, add the *azure-load-balancer-internal-subnet* annotation to your service. The subnet specified must be in the same virtual network as your AKS cluster. When deployed, the load balancer *EXTERNAL-IP* address is part of the specified subnet.

```
apiVersion: v1
kind: Service
metadata:
  name: internal-app
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
    service.beta.kubernetes.io/azure-load-balancer-internal-subnet: "apps-subnet"
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: internal-app
```

## Delete the load balancer

When all services that use the internal load balancer are deleted, the load balancer itself is also deleted.

You can also directly delete a service as with any Kubernetes resource, such as

```
kubectl delete service internal-app
```

, which also then deletes the underlying Azure load balancer.

## Next steps

Learn more about Kubernetes services at the [Kubernetes services documentation](#).

# Use a Standard SKU load balancer in Azure Kubernetes Service (AKS)

2/26/2020 • 12 minutes to read • [Edit Online](#)

To provide access to applications via Kubernetes services of type `LoadBalancer` in Azure Kubernetes Service (AKS), you can use an Azure Load Balancer. A load balancer running on AKS can be used as an internal or an external load balancer. An internal load balancer makes a Kubernetes service accessible only to applications running in the same virtual network as the AKS cluster. An external load balancer receives one or more public IPs for ingress and makes a Kubernetes service accessible externally using the public IPs.

Azure Load Balancer is available in two SKUs - *Basic* and *Standard*. By default, the *Standard* SKU is used when you create an AKS cluster. Using a *Standard* SKU load balancer provides additional features and functionality, such as a larger backend pool size and Availability Zones. It's important that you understand the differences between *Standard* and *Basic* load balancers before choosing which to use. Once you create an AKS cluster, you cannot change the load balancer SKU for that cluster. For more information on the *Basic* and *Standard* SKUs, see [Azure load balancer SKU comparison](#).

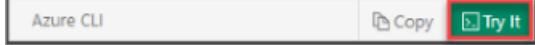
This article assumes a basic understanding of Kubernetes and Azure Load Balancer concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#) and [What is Azure Load Balancer?](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

If you choose to install and use the CLI locally, this article requires that you are running the Azure CLI version 2.0.81 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Before you begin

This article assumes you have an AKS cluster with the *Standard* SKU Azure Load Balancer. If you need an AKS cluster, see the AKS quickstart [using the Azure CLI](#) or [using the Azure portal](#).

The AKS cluster service principal needs also permission to manage network resources if you use an existing subnet or resource group. In general, assign the *Network contributor* role to your service principal on the delegated resources. For more information on permissions, see [Delegate AKS access to other Azure resources](#).

### Moving from a Basic SKU Load Balancer to Standard SKU

If you have an existing cluster with the Basic SKU Load Balancer, there are important behavioral differences to note when migrating to use a cluster with the Standard SKU Load Balancer.

For example, making blue/green deployments to migrate clusters is a common practice given the `load-balancer-sku` type of a cluster can only be defined at cluster create time. However, *Basic SKU* Load Balancers use *Basic SKU* IP Addresses which are not compatible with *Standard SKU* Load Balancers as they require *Standard SKU* IP Addresses. When migrating clusters to upgrade Load Balancer SKUs, a new IP address with a compatible IP Address SKU will be required.

For more considerations on how to migrate clusters, visit [our documentation on migration considerations](#) to view a list of important topics to consider when migrating. The below limitations are also important behavioral differences to note when using Standard SKU Load Balancers in AKS.

### Limitations

The following limitations apply when you create and manage AKS clusters that support a load balancer with the *Standard* SKU:

- At least one public IP or IP prefix is required for allowing egress traffic from the AKS cluster. The public IP or IP prefix is also required to maintain connectivity between the control plane and agent nodes as well as to maintain compatibility with previous versions of AKS. You have the following options for specifying public IPs or IP prefixes with a *Standard* SKU load balancer:
  - Provide your own public IPs.
  - Provide your own public IP prefixes.
  - Specify a number up to 100 to allow the AKS cluster to create that many *Standard* SKU public IPs in the same resource group created as the AKS cluster, which is usually named with `MC_` at the beginning. AKS assigns the public IP to the *Standard* SKU load balancer. By default, one public IP will automatically be created in the same resource group as the AKS cluster, if no public IP, public IP prefix, or number of IPs is specified. You also must allow public addresses and avoid creating any Azure Policy that bans IP creation.
- When using the *Standard* SKU for a load balancer, you must use Kubernetes version *1.13 or greater*.
- Defining the load balancer SKU can only be done when you create an AKS cluster. You cannot change the load balancer SKU after an AKS cluster has been created.
- You can only use one type of load balancer SKU (Basic or Standard) in a single cluster.
- *Standard* SKU Load Balancers only support *Standard* SKU IP Addresses.

## Use the *Standard* SKU load balancer

When you create an AKS cluster, by default, the *Standard* SKU load balancer is used when you run services in that cluster. For example, [the quickstart using the Azure CLI](#) deploys a sample application that uses the *Standard* SKU load balancer.

#### IMPORTANT

Public IP addresses can be avoided by customizing a user-defined route (UDR). Specifying an AKS cluster's outbound type as UDR can skip IP provisioning and backend pool setup for the AKS created Azure load balancer. See [setting a cluster's `outboundType` to 'userDefinedRouting'](#).

## Configure the load balancer to be internal

You can also configure the load balancer to be internal and not expose a public IP. To configure the load balancer as internal, add `service.beta.kubernetes.io/azure-load-balancer-internal: "true"` as an annotation to the *LoadBalancer* service. You can see an example yaml manifest as well as more details about an internal load balancer [here](#).

## Scale the number of managed public IPs

When using a *Standard* SKU load balancer with managed outbound public IPs, which are created by default, you can scale the number of managed outbound public IPs using the *load-balancer-managed-ip-count* parameter.

To update an existing cluster, run the following command. This parameter can also be set at cluster create-time to have multiple managed outbound public IPs.

```
az aks update \
--resource-group myResourceGroup \
--name myAKScluster \
--load-balancer-managed-outbound-ip-count 2
```

The above example sets the number of managed outbound public IPs to 2 for the *myAKScluster* cluster in *myResourceGroup*.

You can also use the *load-balancer-managed-ip-count* parameter to set the initial number of managed outbound public IPs when creating your cluster by appending the `--load-balancer-managed-outbound-ip-count` parameter and setting it to your desired value. The default number of managed outbound public IPs is 1.

## Provide your own public IPs or prefixes for egress

When using a *Standard* SKU load balancer, the AKS cluster automatically creates a public IP in same resource group created for the AKS cluster and assigns the public IP to the *Standard* SKU load balancer. Alternatively, you can assign your own public IP at cluster creation time or you can update an existing cluster's load balancer properties.

By bringing multiple IP addresses or prefixes, you are able to define multiple backing services when defining the IP address behind a single load balancer object. The egress endpoint of specific nodes will depend on what service they are associated with.

#### IMPORTANT

You must use *Standard* SKU public IPs for egress with your *Standard* SKU load balancer. You can verify the SKU of your public IPs using the [az network public-ip show](#) command:

```
az network public-ip show --resource-group myResourceGroup --name myPublicIP --query sku.name -o tsv
```

Use the [az network public-ip show](#) command to list the IDs of your public IPs.

```
az network public-ip show --resource-group myResourceGroup --name myPublicIP --query id -o tsv
```

The above command shows the ID for the *myPublicIP* public IP in the *myResourceGroup* resource group.

Use the *az aks update* command with the *load-balancer-outbound-ips* parameter to update your cluster with your public IPs.

The following example uses the *load-balancer-outbound-ips* parameter with the IDs from the previous command.

```
az aks update \
--resource-group myResourceGroup \
--name myAKSCluster \
--load-balancer-outbound-ips <publicIpId1>,<publicIpId2>
```

You can also use public IP prefixes for egress with your *Standard* SKU load balancer. The following example uses the *az network public-ip prefix show* command to list the IDs of your public IP prefixes:

```
az network public-ip prefix show --resource-group myResourceGroup --name myPublicIPPrefix --query id -o tsv
```

The above command shows the ID for the *myPublicIPPrefix* public IP prefix in the *myResourceGroup* resource group.

The following example uses the *load-balancer-outbound-ip-prefixes* parameter with the IDs from the previous command.

```
az aks update \
--resource-group myResourceGroup \
--name myAKSCluster \
--load-balancer-outbound-ip-prefixes <publicIpPrefixId1>,<publicIpPrefixId2>
```

#### IMPORTANT

The public IPs and IP prefixes must be in the same region and part of the same subscription as your AKS cluster.

#### Define your own public IP or prefixes at cluster create time

You may wish to bring your own IP addresses or IP prefixes for egress at cluster creation time to support scenarios like whitelisting egress endpoints. Append the same parameters shown above to your cluster creation step to define your own public IPs and IP prefixes at the start of a cluster's lifecycle.

Use the *az aks create* command with the *load-balancer-outbound-ips* parameter to create a new cluster with your public IPs at the start.

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--vm-set-type VirtualMachineScaleSets \
--node-count 1 \
--load-balancer-sku standard \
--generate-ssh-keys \
--load-balancer-outbound-ips <publicIpId1>,<publicIpId2>
```

Use the *az aks create* command with the *load-balancer-outbound-ip-prefixes* parameter to create a new cluster with your public IP prefixes at the start.

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--vm-set-type VirtualMachineScaleSets \
--node-count 1 \
--load-balancer-sku standard \
--generate-ssh-keys \
--load-balancer-outbound-ip-prefixes <publicIpPrefixId1>,<publicIpPrefixId2>
```

## Configure outbound ports and idle timeout

### WARNING

The following section is intended for advanced scenarios of larger scale networking or for addressing SNAT exhaustion issues with the default configurations. You must have an accurate inventory of available quota for VMs and IP addresses before changing *AllocatedOutboundPorts* or *IdleTimeoutInMinutes* from their default value in order to maintain healthy clusters.

Altering the values for *AllocatedOutboundPorts* and *IdleTimeoutInMinutes* may significantly change the behavior of the outbound rule for your load balancer. Review the [Load Balancer outbound rules](#), [load Balancer outbound rules](#), and [outbound connections in Azure](#) before updating these values to fully understand the impact of your changes.

Outbound allocated ports and their idle timeouts are used for [SNAT](#). By default, the *Standard* SKU load balancer uses [automatic assignment for the number of outbound ports based on backend pool size](#) and a 30-minute idle timeout for each port. To see these values, use [az network lb outbound-rule list](#) to show the outbound rule for the load balancer:

```
NODE_RG=$(az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv)
az network lb outbound-rule list --resource-group $NODE_RG --lb-name kubernetes -o table
```

The previous commands will list the outbound rule for your load balancer, for example:

AllocatedOutboundPorts	EnableTcpReset	IdleTimeoutInMinutes	Name	Protocol
ProvisioningState	ResourceGroup			
0	True	30	aksOutboundRule	All
MC_myResourceGroup_myAKSCluster_eastus				Succeeded

The example output shows the default value for *AllocatedOutboundPorts* and *IdleTimeoutInMinutes*. A value of 0 for *AllocatedOutboundPorts* sets the number of outbound ports using automatic assignment for the number of outbound ports based on backend pool size. For example, if the cluster has 50 or less nodes, 1024 ports for each node are allocated.

Consider changing the setting of *AllocatedOutboundPorts* or *IdleTimeoutInMinutes* if you expect to face SNAT exhaustion based on the above default configuration. Each additional IP address enables 64,000 additional ports for allocation, however the Azure Standard Load Balancer does not automatically increase the ports per node when more IP addresses are added. You can change these values by setting the *load-balancer-outbound-ports* and *load-balancer-idle-timeout* parameters. For example:

```
az aks update \
--resource-group myResourceGroup \
--name myAKSCluster \
--load-balancer-outbound-ports 0 \
--load-balancer-idle-timeout 30
```

## IMPORTANT

You must [calculate your required quota](#) before customizing *allocatedOutboundPorts* to avoid connectivity or scaling issues. The value you specify for *allocatedOutboundPorts* must also be a multiple of 8.

You can also use the *load-balancer-outbound-ports* and *load-balancer-idle-timeout* parameters when creating a cluster, but you must also specify either *load-balancer-managed-outbound-ip-count*, *load-balancer-outbound-ips*, or *load-balancer-outbound-ip-prefixes* as well. For example:

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--vm-set-type VirtualMachineScaleSets \
--node-count 1 \
--load-balancer-sku standard \
--generate-ssh-keys \
--load-balancer-managed-outbound-ip-count 2 \
--load-balancer-outbound-ports 0 \
--load-balancer-idle-timeout 30
```

When altering the *load-balancer-outbound-ports* and *load-balancer-idle-timeout* parameters from their default, it affects the behavior of the load balancer profile, which impacts the entire cluster.

### Required quota for customizing *allocatedOutboundPorts*

You must have enough outbound IP capacity based on the number of your node VMs and desired allocated outbound ports. To validate you have enough outbound IP capacity, use the following formula:

*outboundIPs \* 64,000 > nodeVMs \* desiredAllocatedOutboundPorts.*

For example, if you have 3 *nodeVMs*, and 50,000 *desiredAllocatedOutboundPorts*, you need to have at least 3 *outboundIPs*. It is recommended that you incorporate additional outbound IP capacity beyond what you need. Additionally, you must account for the cluster autoscaler and the possibility of node pool upgrades when calculating outbound IP capacity. For the cluster autoscaler, review the current node count and the maximum node count and use the higher value. For upgrading, account for an additional node VM for every node pool that allows upgrading.

When setting *IdleTimeoutInMinutes* to a different value than the default of 30 minutes, consider how long your workloads will need an outbound connection. Also consider the default timeout value for a *Standard* SKU load balancer used outside of AKS is 4 minutes. An *IdleTimeoutInMinutes* value that more accurately reflects your specific AKS workload can help decrease SNAT exhaustion caused by tying up connections no longer being used.

## Restrict access to specific IP ranges

The Network Security Group (NSG) associated with the virtual network for the load balancer, by default, has a rule to allow all inbound external traffic. You can update this rule to only allow specific IP ranges for inbound traffic. The following manifest uses *loadBalancerSourceRanges* to specify a new IP range for inbound external traffic:

```
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front
  loadBalancerSourceRanges:
  - MY_EXTERNAL_IP_RANGE
```

The above example updates the rule to only allow inbound external traffic from the *MY\_EXTERNAL\_IP\_RANGE* range. More information about using this method to restrict access to the load balancer service is available in the [Kubernetes documentation](#).

## Next steps

Learn more about Kubernetes services at the [Kubernetes services documentation](#).

# Customize cluster egress with a User-Defined Route (Preview)

2/25/2020 • 14 minutes to read • [Edit Online](#)

Egress from an AKS cluster can be customized to fit specific scenarios. By default, AKS will provision a Standard SKU Load Balancer to be setup and used for egress. However, the default setup may not meet the requirements of all scenarios if public IPs are disallowed or additional hops are required for egress.

This article walks through how to customize a cluster's egress route to support custom network scenarios, such as those which disallows public IPs and requires the cluster to sit behind a network virtual appliance (NVA).

## IMPORTANT

AKS preview features are self-service and are offered on an opt-in basis. Previews are provided *as is* and *as available* and are excluded from the service-level agreement (SLA) and limited warranty. AKS previews are partially covered by customer support on a *best effort* basis. Therefore, the features aren't meant for production use. For more information, see the following support articles:

- [AKS Support Policies](#)
- [Azure Support FAQ](#)

## Prerequisites

- Azure CLI version 2.0.81 or greater
- Azure CLI Preview extension version 0.4.28 or greater
- API version of `2020-01-01` or greater

## Install the latest Azure CLI AKS Preview extension

To set the outbound type of a cluster, you need the Azure CLI AKS Preview extension version 0.4.18 or later. Install the Azure CLI AKS Preview extension by using the `az extension add` command, and then check for any available updates by using the following `az extension update` command:

```
# Install the aks-preview extension
az extension add --name aks-preview

# Update the extension to make sure you have the latest version installed
az extension update --name aks-preview
```

## Limitations

- During preview, `outboundType` can only be defined at cluster create time and cannot be updated afterward.
- During preview, `outboundType` AKS clusters should use Azure CNI. Kubenet is configurable, usage requires manual associations of the route table to the AKS subnet.
- Setting `outboundType` requires AKS clusters with a `vm-set-type` of `VirtualMachineScaleSets` and `load-balancer-sku` of `Standard`.
- Setting `outboundType` to a value of `UDR` requires a user-defined route with valid outbound connectivity for the cluster.

- Setting `outboundType` to a value of `UDR` implies the ingress source IP routed to the load-balancer may **not** **match** the cluster's outgoing egress destination address.

## Overview of outbound types in AKS

An AKS cluster can be customized with a unique `outboundType` of type load balancer or user-defined routing.

### IMPORTANT

Outbound type impacts only the egress traffic of your cluster. See [setting up ingress controllers](#) for more information.

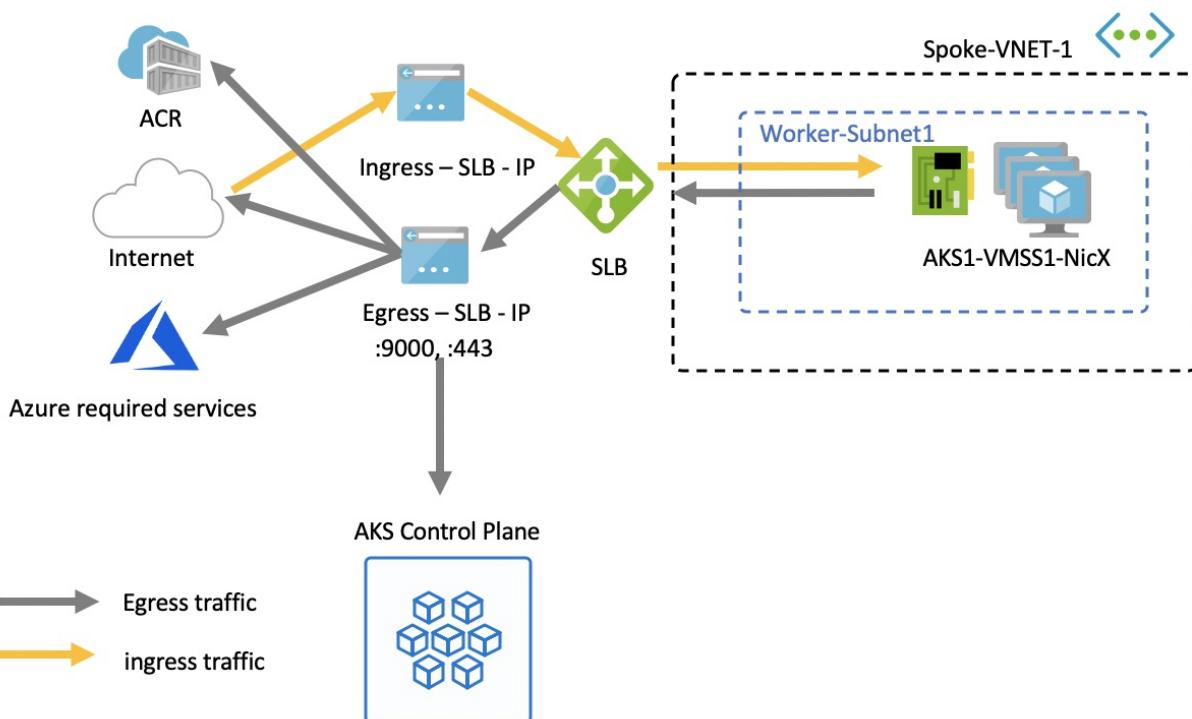
### Outbound type of loadBalancer

If `loadBalancer` is set, AKS completes the following setup automatically. The load balancer is used for egress through an AKS assigned public IP. An outbound type of `loadBalancer` supports Kubernetes services of type `loadBalancer`, which expect egress out of the load balancer created by the AKS resource provider.

The following setup is done by AKS.

- A public IP address is provisioned for cluster egress.
- The public IP address is assigned to the load balancer resource.
- Backend pools for the load balancer are setup for agent nodes in the cluster.

Below is a network topology deployed in AKS clusters by default, which use an `outboundType` of `loadBalancer`.



### Outbound type of userDefinedRouting

#### NOTE

Using outbound type is an advanced networking scenario and requires proper network configuration.

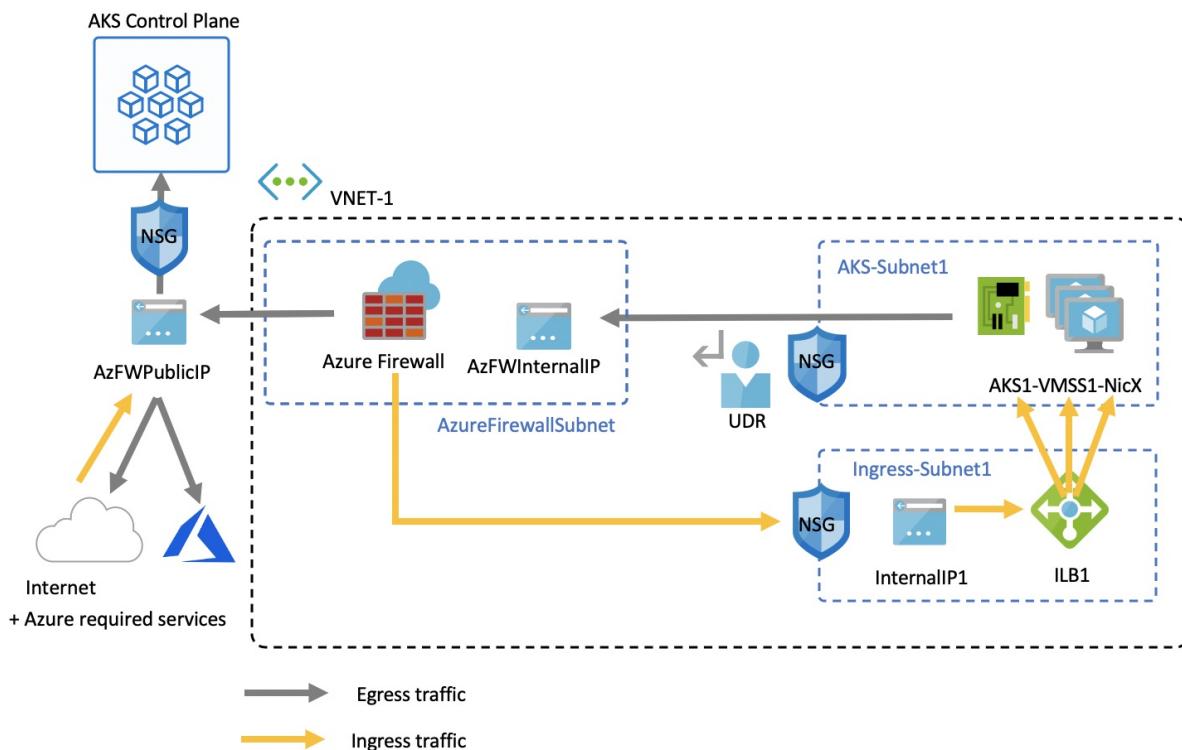
If `userDefinedRouting` is set, AKS will not automatically configure egress paths. The following is expected to be done by **the user**.

Cluster must be deployed into an existing virtual network with a subnet that has been configured. A valid user-defined route (UDR) must exist on the subnet with outbound connectivity.

The AKS resource provider will deploy a standard load balancer (SLB). The load balancer is not configured with any rules and [does not incur a charge until a rule is placed](#). AKS will **not** automatically provision a public IP address for the SLB frontend. AKS will **not** automatically configure the load balancer backend pool.

## Deploy a cluster with outbound type of UDR and Azure Firewall

To illustrate the application of a cluster with outbound type using a user-defined route, a cluster can be configured on a virtual network peered with an Azure Firewall.



- Ingress is forced to flow through firewall filters
  - An isolated subnet holds an internal load balancer for routing into agent nodes
  - Agent nodes are isolated in a dedicated subnet
- Outbound requests start from agent nodes to the Azure Firewall internal IP using a user-defined route
  - Requests from AKS agent nodes follow a UDR that has been placed on the subnet the AKS cluster was deployed into.
  - Azure Firewall egresses out of the virtual network from a public IP frontend
  - Access to the AKS control plane is protected by an NSG, which has enabled the firewall frontend IP address
  - Access to the public internet or other Azure services flows to and from the firewall frontend IP address

### Set configuration via environment variables

Define a set of environment variables to be used in resource creations.

```

PREFIX="contosofin"
RG="${PREFIX}-rg"
LOC="eastus"
NAME="${PREFIX}outboundudr"
AKS_NAME="${PREFIX}aks"
VNET_NAME="${PREFIX}vnet"
AKSSUBNET_NAME="${PREFIX}akssubnet"
SVCSUBNET_NAME="${PREFIX}svcsubnet"
# DO NOT CHANGE FWSUBNET_NAME - This is currently a requirement for Azure Firewall.
FWSUBNET_NAME="AzureFirewallSubnet"
FWNAME="${PREFIX}fw"
FWPUBLICIP_NAME="${PREFIX}fwpublicip"
FWIPCONFIG_NAME="${PREFIX}fwconfig"
FWROUTE_TABLE_NAME="${PREFIX}fwrt"
FWROUTE_NAME="${PREFIX}fwrn"
FWROUTE_NAME_INTERNET="${PREFIX}fwinternet"
DEVSUBNET_NAME="${PREFIX}dev"

```

Next, set subscription IDs.

```

# Get ARM Access Token and Subscription ID - This will be used for AuthN later.

ACCESS_TOKEN=$(az account get-access-token -o tsv --query 'accessToken')

# NOTE: Update Subscription Name
# Set Default Azure Subscription to be Used via Subscription ID

az account set -s <SUBSCRIPTION_ID_Goes_Here>

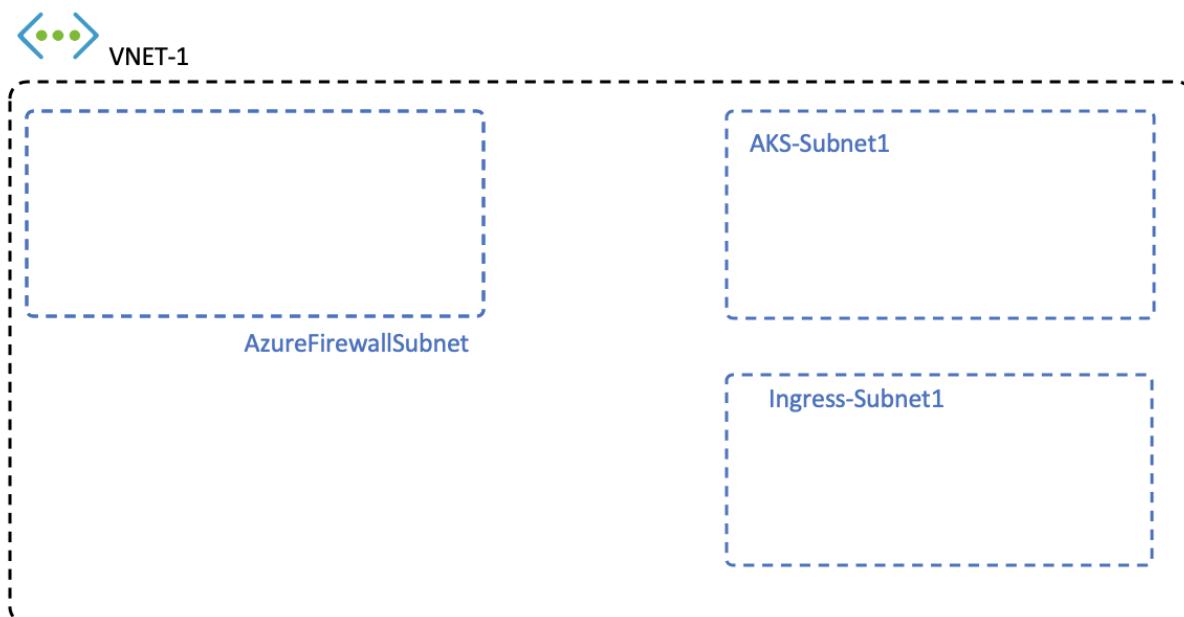
# NOTE: Update Subscription Name for setting SUBID

SUBID=$(az account show -s '<SUBSCRIPTION_NAME_Goes_Here>' -o tsv --query 'id')

```

## Create a virtual network with multiple subnets

Provision a virtual network with three separate subnets, one for the cluster, one for the firewall, and one for service ingress.



Create a resource group to hold all of the resources.

```
# Create Resource Group

az group create --name $RG --location $LOC
```

Create a two virtual networks to host the AKS cluster and the Azure Firewall. Each will have their own subnet. Let's start with the AKS network.

```
# Dedicated virtual network with AKS subnet

az network vnet create \
--resource-group $RG \
--name $VNET_NAME \
--address-prefixes 100.64.0.0/16 \
--subnet-name $AKSSUBNET_NAME \
--subnet-prefix 100.64.1.0/24

# Dedicated subnet for K8s services

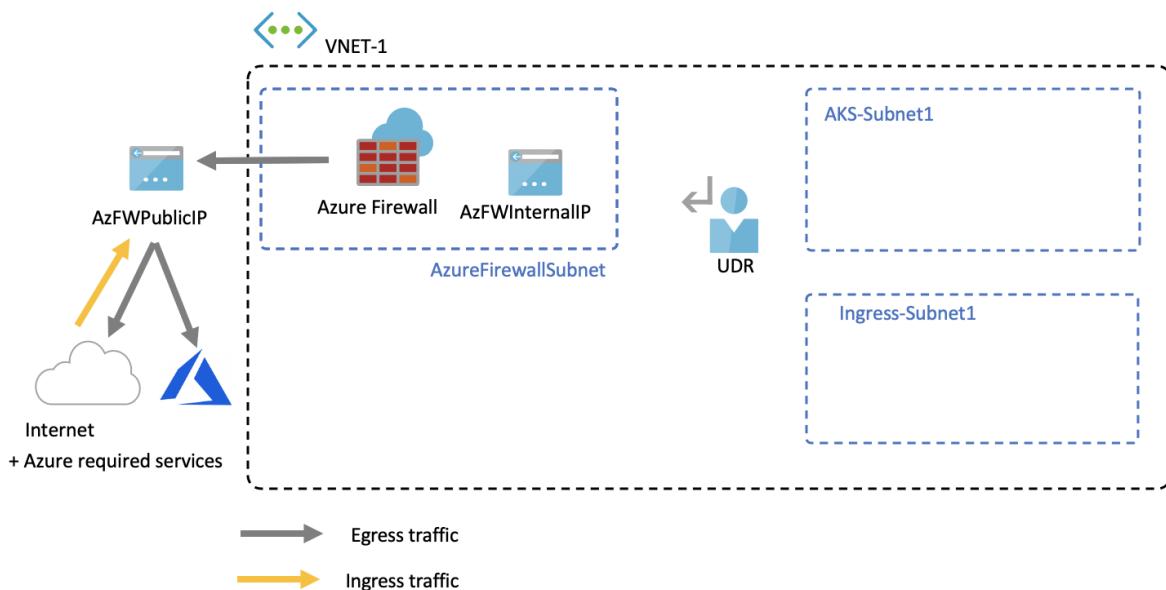
az network vnet subnet create \
--resource-group $RG \
--vnet-name $VNET_NAME \
--name $SVCSUBNET_NAME \
--address-prefix 100.64.2.0/24

# Dedicated subnet for Azure Firewall (Firewall name cannot be changed)

az network vnet subnet create \
--resource-group $RG \
--vnet-name $VNET_NAME \
--name $FWSUBNET_NAME \
--address-prefix 100.64.3.0/24
```

## Create and setup an Azure Firewall with a UDR

Azure Firewall inbound and outbound rules must be configured. The main purpose of the firewall is to enable organizations to setup granular ingress and egress traffic rules into and out of the AKS Cluster.



Create a standard SKU public IP resource which will be used as the Azure Firewall frontend address.

```
az network public-ip create -g $RG -n $FWPUBLICIP_NAME -l $LOC --sku "Standard"
```

Register the preview cli-extension to create an Azure Firewall.

```
# Install Azure Firewall preview CLI extension  
az extension add --name azure-firewall  
  
# Deploy Azure Firewall  
az network firewall create -g $RG -n $FWNAME -l $LOC
```

The IP address created earlier can now be assigned to the firewall frontend.

#### NOTE

Setup of the public IP address to the Azure Firewall may take a few minutes.

If errors are repeatedly received on the below command, delete the existing firewall and public IP and provision the Public IP and Azure Firewall through the portal at the same time.

```
# Configure Firewall IP Config  
  
az network firewall ip-config create -g $RG -f $FWNAME -n $FWIPCONFIG_NAME --public-ip-address  
$FWPUBLICIP_NAME --vnet-name $VNET_NAME
```

When the previous command has succeeded, save the firewall frontend IP address for configuration later.

```
# Capture Firewall IP Address for Later Use  
  
FWPUBLIC_IP=$(az network public-ip show -g $RG -n $FWPUBLICIP_NAME --query "ipAddress" -o tsv)  
FWPRIVATE_IP=$(az network firewall show -g $RG -n $FWNAME --query "ipConfigurations[0].privateIpAddress" -o tsv)
```

### Create a UDR with a hop to Azure Firewall

Azure automatically routes traffic between Azure subnets, virtual networks, and on-premises networks. If you want to change any of Azure's default routing, you do so by creating a route table.

Create an empty route table to be associated with a given subnet. The route table will define the next hop as the Azure Firewall created above. Each subnet can have zero or one route table associated to it.

```
# Create UDR and add a route for Azure Firewall  
  
az network route-table create -g $RG --name $FWRROUTE_TABLE_NAME  
az network route-table route create -g $RG --name $FWRROUTE_NAME --route-table-name $FWRROUTE_TABLE_NAME --  
address-prefix 0.0.0.0/0 --next-hop-type VirtualAppliance --next-hop-ip-address $FWPRIVATE_IP --subscription  
$SUBID  
az network route-table route create -g $RG --name $FWRROUTE_NAME_INTERNET --route-table-name  
$FWRROUTE_TABLE_NAME --address-prefix $FWPUBLIC_IP/32 --next-hop-type Internet
```

See [virtual network route table documentation](#) about how you can override Azure's default system routes or add additional routes to a subnet's route table.

## Adding network firewall rules

## WARNING

Below shows one example of adding a firewall rule. All egress endpoints defined in the [required egress endpoints](#) must be enabled by application firewall rules for AKS clusters to function. Without these endpoints enabled, your cluster cannot operate.

Below is an example of a network and application rule. We add a network rule which allows any protocol, source-address, destination-address, and destination-ports. We also add an application rule for **some** of the endpoints required by AKS.

In a production scenario, you should only enable access to required endpoints for your application and those defined in [AKS required egress](#).

```
# Add Network FW Rules

az network firewall network-rule create -g $RG -f $FWNAME --collection-name 'aksfwnr' -n 'netrules' --
protocols 'Any' --source-addresses '*' --destination-addresses '*' --destination-ports '*' --action allow --
priority 100

# Add Application FW Rules
# IMPORTANT: Add AKS required egress endpoints

az network firewall application-rule create -g $RG -f $FWNAME \
--collection-name 'AKS_Global_Required' \
--action allow \
--priority 100 \
-n 'required' \
--source-addresses '*' \
--protocols 'http=80' 'https=443' \
--target-fqdns \
'aksrepos.azurecr.io' \
'*blob.core.windows.net' \
'mcr.microsoft.com' \
'*cdn.msccr.io' \
'*data.mcr.microsoft.com' \
'management.azure.com' \
'login.microsoftonline.com' \
'ntp.ubuntu.com' \
'packages.microsoft.com' \
'acs-mirror.azureedge.net'
```

See [Azure Firewall documentation](#) to learn more about the Azure Firewall service.

## Associate the route table to AKS

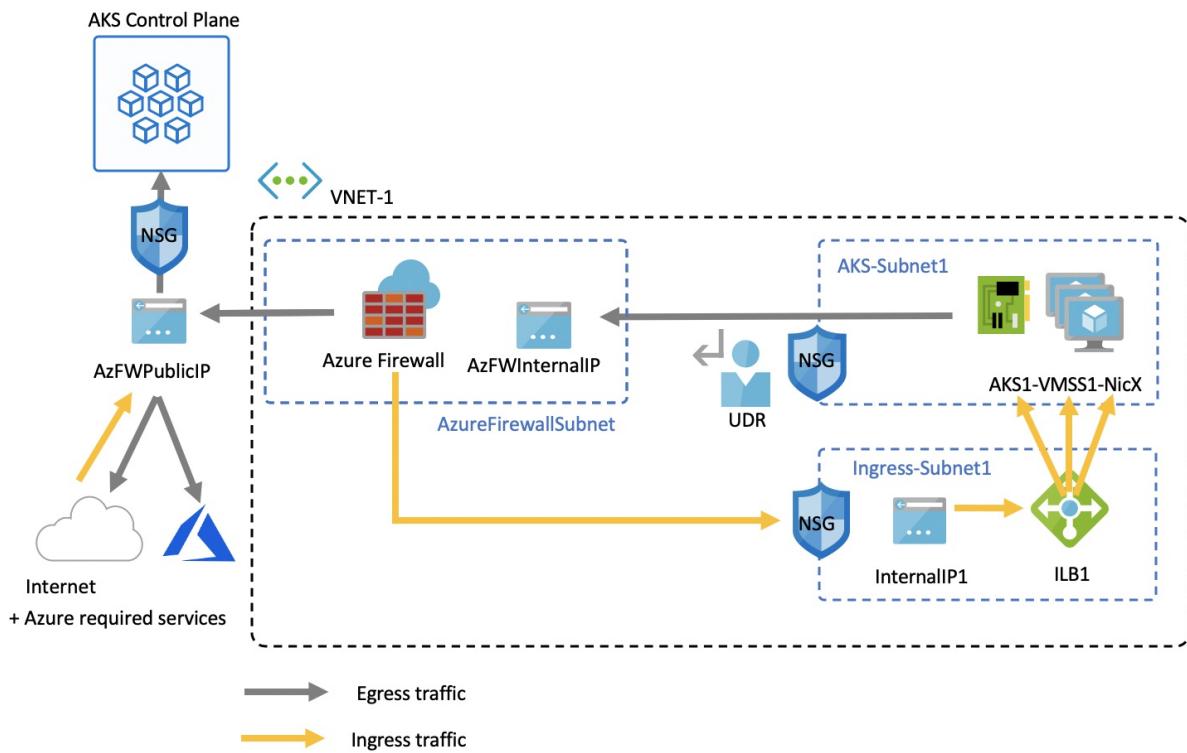
To associate the cluster with the firewall, the dedicated subnet for the cluster's subnet must reference the route table created above. Association can be done by issuing a command to the virtual network holding both the cluster and firewall to update the route table of the cluster's subnet.

```
# Associate route table with next hop to Firewall to the AKS subnet

az network vnet subnet update -g $RG --vnet-name $VNET_NAME --name $AKSSUBNET_NAME --route-table
$FWROUTE_TABLE_NAME
```

## Deploy AKS with outbound type of UDR to the existing network

Now an AKS cluster can be deployed into the existing virtual network setup. In order to set a cluster outbound type to user-defined routing, an existing subnet must be provided to AKS.



### Create a service principal with access to provision inside the existing virtual network

A service principal is used by AKS to create cluster resources. The service principal passed at create time is used to create underlying AKS resources such as VMs, Storage, and Load Balancers used by AKS. If granted too few permissions, it will not be able to provision an AKS Cluster.

```
# Create SP and Assign Permission to Virtual Network
az ad sp create-for-rbac -n "${PREFIX}sp" --skip-assignment
```

Now replace the `APPID` and `PASSWORD` below with the service principal appid and service principal password autogenerated by the previous command output. We will reference the VNET resource ID to grant the permissions to the service principal so AKS can deploy resources into it.

```
APPID=<SERVICE_PRINCIPAL_APPID_Goes_Here>
PASSWORD=<SERVICEPRINCIPAL_PASSWORD_Goes_Here>
VNETID=$(az network vnet show -g $RG --name $VNET_NAME --query id -o tsv)

# Assign SP Permission to VNET
az role assignment create --assignee $APPID --scope $VNETID --role Contributor

# View Role Assignment
az role assignment list --assignee $APPID --all -o table
```

### Deploy AKS

Finally, the AKS cluster can be deployed into the existing subnet we have dedicated for the cluster. The target subnet to be deployed into is defined with the environment variable, `$SUBNETID`.

We will define the outbound type to follow the UDR which exists on the subnet, enabling AKS to skip setup and IP provisioning for the load balancer which can now be strictly internal.

The AKS feature for [API server authorized IP ranges](#) can be added to limit API server access to only the firewall's public endpoint. The authorized IP ranges feature is denoted in the diagram as the NSG which must be passed to access the control plane. When enabling the authorized IP range feature to limit API server access, your developer

tools must use a jumpbox from the firewall's virtual network or you must add all developer endpoints to the authorized IP range.

#### TIP

Additional features can be added to the cluster deployment such as (Private Cluster)[]. When using authorized IP ranges, a jumpbox will be required inside of the cluster network to access the API server.

```
az aks create -g $RG -n $AKS_NAME -l $LOC \
--node-count 3 \
--network-plugin azure --generate-ssh-keys \
--service-cidr 192.168.0.0/16 \
--dns-service-ip 192.168.0.10 \
--docker-bridge-address 172.22.0.1/29 \
--vnet-subnet-id $SUBNETID \
--service-principal $APPID \
--client-secret $PASSWORD \
--load-balancer-sku standard \
--outbound-type userDefinedRouting \
--api-server-authorized-ip-ranges $FWPUBLIC_IP
```

### Enable developer access to the API server

Due to the authorized IP ranges setup for the cluster, you must add your developer tooling IP addresses to the AKS cluster list of approved IP ranges to access the API server. Another option is to configure a jumpbox with the needed tooling inside a separate subnet in the Firewall's virtual network.

Add another IP address to the approved ranges with the following command

```
# Retrieve your IP address
CURRENT_IP=$(dig @resolver1.opendns.com ANY myip.opendns.com +short)

# Add to AKS approved list
az aks update -g $RG -n $AKS_NAME --api-server-authorized-ip-ranges $CURRENT_IP/32
```

### Setup the internal load balancer

AKS has deployed a load balancer with the cluster which can be setup as an [internal load balancer](#).

To create an internal load balancer, create a service manifest named internal-lb.yaml with the service type LoadBalancer and the azure-load-balancer-internal annotation as shown in the following example:

```
apiVersion: v1
kind: Service
metadata:
  name: internal-app
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
    service.beta.kubernetes.io/azure-load-balancer-internal-subnet: "contosofinsvcsubnet"
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: internal-app
```

Deploy the internal load balancer using the kubectl apply and specify the name of your YAML manifest:

```
kubectl apply -f internal-lb.yaml
```

## Deploy a Kubernetes service

Since the cluster outbound type is set as UDR, associating the agent nodes as the backend pool for the load balancer is not completed automatically by AKS at cluster create time. However, backend pool association is handled by the Kubernetes Azure cloud provider when the Kubernetes service is deployed.

Deploy the Azure voting app application by copying the yaml below to a file named `example.yaml`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-back
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-back
  template:
    metadata:
      labels:
        app: azure-vote-back
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": linux
      containers:
        - name: azure-vote-back
          image: redis
          resources:
            requests:
              cpu: 100m
              memory: 128Mi
            limits:
              cpu: 250m
              memory: 256Mi
          ports:
            - containerPort: 6379
              name: redis
---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
    - port: 6379
  selector:
    app: azure-vote-back
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: azure-vote-front
spec:
  replicas: 1
  selector:
    matchLabels:
      app: azure-vote-front
  template:
    metadata:
      labels:
        app: azure-vote-front
      spec:
```

```

nodeSelector:
  "beta.kubernetes.io/os": linux
containers:
- name: azure-vote-front
  image: microsoft/azure-vote-front:v1
  resources:
    requests:
      cpu: 100m
      memory: 128Mi
    limits:
      cpu: 250m
      memory: 256Mi
  ports:
  - containerPort: 80
  env:
  - name: REDIS
    value: "azure-vote-back"
---
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-front
  annotations:
    service.beta.kubernetes.io/azure-load-balancer-internal: "true"
    service.beta.kubernetes.io/azure-load-balancer-internal-subnet: "contosofinsvcsubnet"
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-vote-front

```

Deploy the service by running:

```
kubectl apply -f example.yaml
```

## Add a DNAT rule to Azure Firewall

To configure inbound connectivity, a DNAT rule must be written to the Azure Firewall. To test connectivity to our cluster, a rule is defined for the firewall frontend public IP address to route to the internal IP exposed by the internal service.

The destination address can be customized as it is the port on the firewall to be accessed. The translated address must be the IP address of the internal load balancer. The translated port must be the exposed port for your Kubernetes service.

You will need to specify the internal IP address assigned to the load balancer created by the Kubernetes service. Retrieve the address by running:

```
kubectl get services
```

The IP address needed will be listed in the EXTERNAL-IP column, similar to the following.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
azure-vote-back	ClusterIP	192.168.92.209	<none>	6379/TCP	23m
azure-vote-front	LoadBalancer	192.168.19.183	100.64.2.5	80:32106/TCP	23m
kubernetes	ClusterIP	192.168.0.1	<none>	443/TCP	4d3h

```
az network firewall nat-rule create --collection-name exampleset --destination-addresses $FWPUBLIC_IP --destination-ports 80 --firewall-name $FWNAME --name inboundrule --protocols Any --resource-group $RG --source-addresses '*' --translated-port 80 --action Dnat --priority 100 --translated-address <INSERT IP OF K8s SERVICE>
```

## Clean up resources

### NOTE

When deleting the Kubernetes internal service, if the internal load balancer is no longer in use by any service, the Azure cloud provider will delete the internal load balancer. On the next service deployment, a load balancer will be deployed if none can be found with the configuration requested.

To clean up Azure resources, delete the AKS resource group.

```
az group delete -g $RG
```

## Validate connectivity

Navigate to the Azure Firewall frontend IP address in a browser to validate connectivity.

You should see an image of the Azure voting app.

## Next steps

See [Azure networking UDR overview](#).

See [how to create, change, or delete a route table](#).

# Use a static public IP address and DNS label with the Azure Kubernetes Service (AKS) load balancer

2/25/2020 • 4 minutes to read • [Edit Online](#)

By default, the public IP address assigned to a load balancer resource created by an AKS cluster is only valid for the lifespan of that resource. If you delete the Kubernetes service, the associated load balancer and IP address are also deleted. If you want to assign a specific IP address or retain an IP address for redeployed Kubernetes services, you can create and use a static public IP address.

This article shows you how to create a static public IP address and assign it to your Kubernetes service.

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart [using the Azure CLI](#) or [using the Azure portal](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

This article covers using a *Standard* SKU IP with a *Standard* SKU load balancer. For more information, see [IP address types and allocation methods in Azure](#).

## Create a static IP address

Create a static public IP address with the [az network public ip create](#) command. The following creates a static IP resource named *myAKSPublicIP* in the *myResourceGroup* resource group:

```
az network public-ip create \
    --resource-group myResourceGroup \
    --name myAKSPublicIP \
    --sku Standard \
    --allocation-method static
```

### NOTE

If you are using a *Basic* SKU load balancer in your AKS cluster, use *Basic* for the *sku* parameter when defining a public IP. Only *Basic* SKU IPs work with the *Basic* SKU load balancer and only *Standard* SKU IPs work with *Standard* SKU load balancers.

The IP address is displayed, as shown in the following condensed example output:

```
{
  "publicIp": {
    ...
    "ipAddress": "40.121.183.52",
    ...
  }
}
```

You can later get the public IP address using the [az network public-ip list](#) command. Specify the name of the node

resource group and public IP address you created, and query for the *ipAddress* as shown in the following example:

```
$ az network public-ip show --resource-group myResourceGroup --name myAKSPublicIP --query ipAddress --output tsv  
40.121.183.52
```

## Create a service using the static IP address

Before creating a service, ensure the service principal used by the AKS cluster has delegated permissions to the other resource group. For example:

```
az role assignment create \  
  --assignee <SP Client ID> \  
  --role "Contributor" \  
  --scope /subscriptions/<subscription id>/resourceGroups/<resource group name>
```

To create a *LoadBalancer* service with the static public IP address, add the `loadBalancerIP` property and the value of the static public IP address to the YAML manifest. Create a file named `load-balancer-service.yaml` and copy in the following YAML. Provide your own public IP address created in the previous step. The following example also sets the annotation to the resource group named *myResourceGroup*. Provide your own resource group name.

```
apiVersion: v1  
kind: Service  
metadata:  
  annotations:  
    service.beta.kubernetes.io/azure-load-balancer-resource-group: myResourceGroup  
  name: azure-load-balancer  
spec:  
  loadBalancerIP: 40.121.183.52  
  type: LoadBalancer  
  ports:  
  - port: 80  
  selector:  
    app: azure-load-balancer
```

Create the service and deployment with the `kubectl apply` command.

```
kubectl apply -f load-balancer-service.yaml
```

## Apply a DNS label to the service

If your service is using a dynamic or static public IP address, you can use the service annotation

`service.beta.kubernetes.io/azure-dns-label-name` to set a public-facing DNS label. This publishes a fully qualified domain name for your service using Azure's public DNS servers and top-level domain. The annotation value must be unique within the Azure location, so its recommended to use a sufficiently qualified label.

Azure will then automatically append a default subnet, such as `<location>.cloudapp.azure.com` (where location is the region you selected), to the name you provide, to create the fully qualified DNS name. For example:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    service.beta.kubernetes.io/azure-dns-label-name: myserviceuniquelabel
  name: azure-load-balancer
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    app: azure-load-balancer
```

#### NOTE

To publish the service on your own domain, see [Azure DNS](#) and the [external-dns](#) project.

## Troubleshoot

If the static IP address defined in the *loadBalancerIP* property of the Kubernetes service manifest does not exist, or has not been created in the node resource group and no additional delegations configured, the load balancer service creation fails. To troubleshoot, review the service creation events with the [kubectl describe](#) command. Provide the name of the service as specified in the YAML manifest, as shown in the following example:

```
kubectl describe service azure-load-balancer
```

Information about the Kubernetes service resource is displayed. The *Events* at the end of the following example output indicate that the *user supplied IP Address was not found*. In these scenarios, verify that you have created the static public IP address in the node resource group and that the IP address specified in the Kubernetes service manifest is correct.

```
Name:                   azure-load-balancer
Namespace:              default
Labels:                 <none>
Annotations:            <none>
Selector:               app=azure-load-balancer
Type:                  LoadBalancer
IP:                    10.0.18.125
IP:                    40.121.183.52
Port:                  <unset>  80/TCP
TargetPort:             80/TCP
NodePort:               <unset>  32582/TCP
Endpoints:              <none>
Session Affinity:      None
External Traffic Policy: Cluster
Events:
  Type     Reason          Age           From            Message
  ----     ----          ----          ----          -----
  Normal   CreatingLoadBalancer  7s (x2 over 22s)  service-controller  Creating load balancer
  Warning  CreatingLoadBalancerFailed 6s (x2 over 12s)  service-controller  Error creating load balancer
(will retry): Failed to create load balancer for service default/azure-load-balancer: user supplied IP Address
40.121.183.52 was not found
```

## Next steps

For additional control over the network traffic to your applications, you may want to instead [create an ingress controller](#). You can also [create an ingress controller with a static public IP address](#).



# Create an ingress controller in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the [NGINX ingress controller](#) in an Azure Kubernetes Service (AKS) cluster. Two applications are then run in the AKS cluster, each of which is accessible over the single IP address.

You can also:

- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller that uses your own TLS certificates](#)
- [Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates \*\*with a dynamic public IP address\*\* or \*\*with a static public IP address\*\*](#)

## Before you begin

This article uses Helm to install the NGINX ingress controller and a sample web app.

This article also requires that you are running the Azure CLI version 2.0.64 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an ingress controller

To create the ingress controller, use Helm to install `nginx-ingress`. For added redundancy, two replicas of the NGINX ingress controllers are deployed with the `--set controller.replicaCount` parameter. To fully benefit from running replicas of the ingress controller, make sure there's more than one node in your AKS cluster.

The ingress controller also needs to be scheduled on a Linux node. Windows Server nodes (currently in preview in AKS) shouldn't run the ingress controller. A node selector is specified using the `--set nodeSelector` parameter to tell the Kubernetes scheduler to run the NGINX ingress controller on a Linux-based node.

### TIP

The following example creates a Kubernetes namespace for the ingress resources named `ingress-basic`. Specify a namespace for your own environment as needed.

### TIP

If you would like to enable [client source IP preservation](#) for requests to containers in your cluster, add `--set controller.service.externalTrafficPolicy=Local` to the Helm install command. The client source IP is stored in the request header under `X-Forwarded-For`. When using an ingress controller with client source IP preservation enabled, SSL pass-through will not work.

```

# Create a namespace for your ingress resources
kubectl create namespace ingress-basic

# Add the official stable repository
helm repo add stable https://kubernetes-charts.storage.googleapis.com/

# Use Helm to deploy an NGINX ingress controller
helm install nginx-ingress stable/nginx-ingress \
  --namespace ingress-basic \
  --set controller.replicaCount=2 \
  --set controller.nodeSelector."beta\\.kubernetes\\.io/os"=linux \
  --set defaultBackend.nodeSelector."beta\\.kubernetes\\.io/os"=linux

```

When the Kubernetes load balancer service is created for the NGINX ingress controller, a dynamic public IP address is assigned, as shown in the following example output:

```

$ kubectl get service -l app=nginx-ingress --namespace ingress-basic

NAME                   TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)
AGE
nginx-ingress-controller   LoadBalancer   10.0.61.144   EXTERNAL_IP   80:30386/TCP,443:32276/TCP
6m2s
nginx-ingress-default-backend   ClusterIP   10.0.192.145   <none>       80/TCP
6m2s

```

No ingress rules have been created yet, so the NGINX ingress controller's default 404 page is displayed if you browse to the internal IP address. Ingress rules are configured in the following steps.

## Run demo applications

To see the ingress controller in action, let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple *Hello world* application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment as follows:

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create the first demo application from a Helm chart with the following command:

```
helm install aks-helloworld azure-samples/aks-helloworld --namespace ingress-basic
```

Now install a second instance of the demo application. For the second instance, you specify a new title so that the two applications are visually distinct. You also specify a unique service name:

```

helm install aks-helloworld-two azure-samples/aks-helloworld \
  --namespace ingress-basic \
  --set title="AKS Ingress Demo" \
  --set serviceName="aks-helloworld-two"

```

## Create an ingress route

Both applications are now running on your Kubernetes cluster. To route traffic to each application, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to `EXTERNAL_IP` is routed to the service named `aks-helloworld`. Traffic to `EXTERNAL_IP/hello-world-two` is routed to the `aks-helloworld-two` service. Traffic to `EXTERNAL_IP/static` is routed to the service named `aks-helloworld` for static assets.

Create a file named `hello-world-ingress.yaml` and copy in the following example YAML.

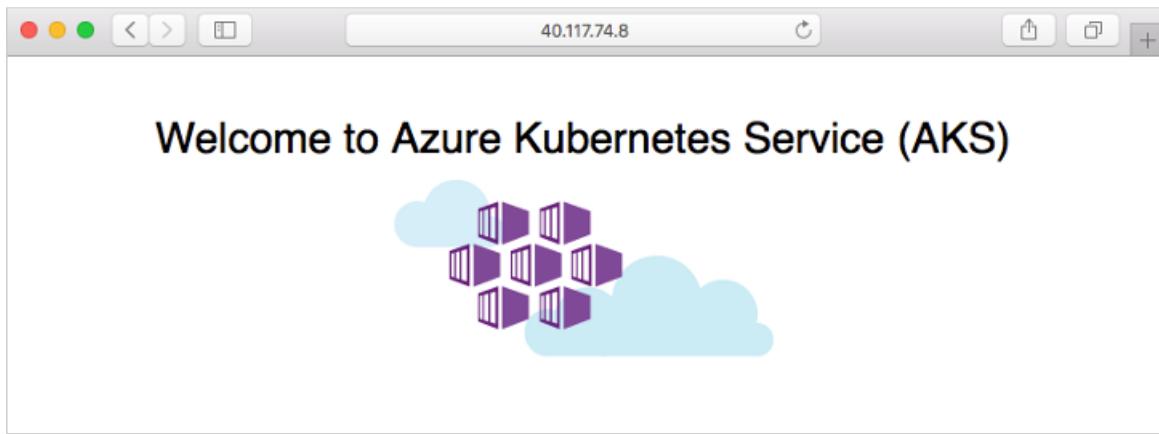
```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  namespace: ingress-basic
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  rules:
  - http:
    paths:
      - backend:
          serviceName: aks-helloworld
          servicePort: 80
          path: /(.*)
      - backend:
          serviceName: aks-helloworld-two
          servicePort: 80
          path: /hello-world-two(/|$(.).*)
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress-static
  namespace: ingress-basic
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /static/$2
spec:
  rules:
  - http:
    paths:
      - backend:
          serviceName: aks-helloworld
          servicePort: 80
          path: /static(/|$(.).*)
```

Create the ingress resource using the `kubectl apply -f hello-world-ingress.yaml` command.

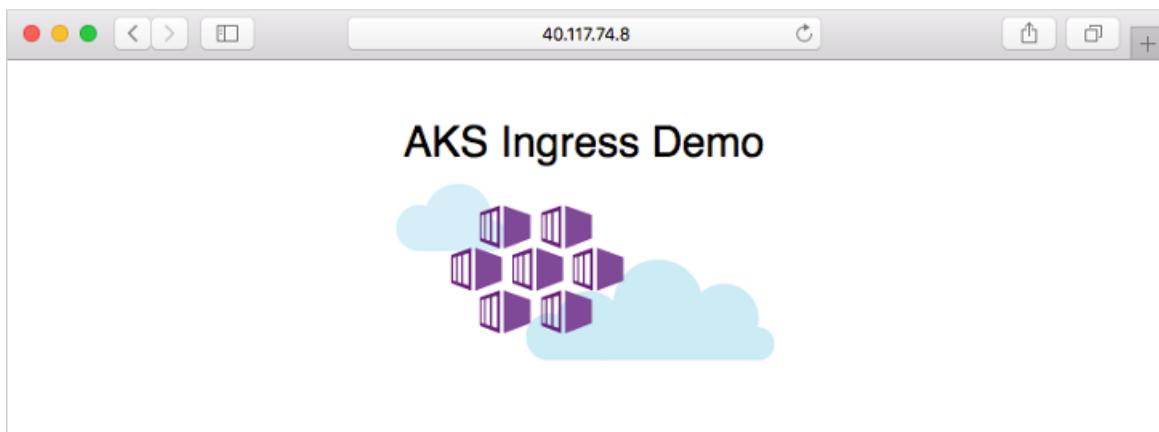
```
$ kubectl apply -f hello-world-ingress.yaml
ingress.extensions/hello-world-ingress created
ingress.extensions/hello-world-ingress-static created
```

## Test the ingress controller

To test the routes for the ingress controller, browse to the two applications. Open a web browser to the IP address of your NGINX ingress controller, such as `EXTERNAL_IP`. The first demo application is displayed in the web browser, as shown in the follow example:



Now add the `/hello-world-two` path to the IP address, such as `EXTERNAL_IP/hello-world-two`. The second demo application with the custom title is displayed:



## Clean up resources

This article used Helm to install the ingress components and sample apps. When you deploy a Helm chart, a number of Kubernetes resources are created. These resources includes pods, deployments, and services. To clean up these resources, you can either delete the entire sample namespace, or the individual resources.

### Delete the sample namespace and all resources

To delete the entire sample namespace, use the `kubectl delete` command and specify your namespace name. All the resources in the namespace are deleted.

```
kubectl delete namespace ingress-basic
```

Then, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

### Delete resources individually

Alternatively, a more granular approach is to delete the individual resources created. List the Helm releases with the `helm list` command. Look for charts named `nginx-ingress` and `aks-helloworld`, as shown in the following example output:

\$ helm list --namespace ingress-basic					
NAME	NAMESPACE	REVISION	UPDATED	STATUS	
CHART	APP VERSION				
aks-helloworld	ingress-basic	1	2020-01-06 19:57:00.131576 -0600 CST	deployed	
aks-helloworld-0.1.0					
aks-helloworld-two	ingress-basic	1	2020-01-06 19:57:10.971365 -0600 CST	deployed	
aks-helloworld-0.1.0					
nginx-ingress	ingress-basic	1	2020-01-06 19:55:46.358275 -0600 CST	deployed	
nginx-ingress-1.27.1	0.26.1				

Delete the releases with the `helm delete` command. The following example deletes the NGINX ingress deployment, and the two sample AKS hello world apps.

```
$ helm delete aks-helloworld aks-helloworld-two nginx-ingress --namespace ingress-basic  
release "aks-helloworld" uninstalled  
release "aks-helloworld-two" uninstalled  
release "nginx-ingress" uninstalled
```

Next, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

Remove the ingress route that directed traffic to the sample apps:

```
kubectl delete -f hello-world-ingress.yaml
```

Finally, you can delete the itself namespace. Use the `kubectl delete` command and specify your namespace name:

```
kubectl delete namespace ingress-basic
```

## Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- [Helm CLI](#)
- [NGINX ingress controller](#)

You can also:

- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller that uses your own TLS certificates](#)
- Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates [with a dynamic public IP address](#) or [with a static public IP address](#)

# HTTP application routing

2/25/2020 • 6 minutes to read • [Edit Online](#)

The HTTP application routing solution makes it easy to access applications that are deployed to your Azure Kubernetes Service (AKS) cluster. When the solution's enabled, it configures an [Ingress controller](#) in your AKS cluster. As applications are deployed, the solution also creates publicly accessible DNS names for application endpoints.

When the add-on is enabled, it creates a DNS Zone in your subscription. For more information about DNS cost, see [DNS pricing](#).

**Caution**

The HTTP application routing add-on is designed to let you quickly create an ingress controller and access your applications. This add-on is not recommended for production use. For production-ready ingress deployments that include multiple replicas and TLS support, see [Create an HTTPS ingress controller](#).

## HTTP routing solution overview

The add-on deploys two components: a [Kubernetes Ingress controller](#) and an [External-DNS](#) controller.

- **Ingress controller:** The Ingress controller is exposed to the internet by using a Kubernetes service of type LoadBalancer. The Ingress controller watches and implements [Kubernetes Ingress resources](#), which creates routes to application endpoints.
- **External-DNS controller:** Watches for Kubernetes Ingress resources and creates DNS A records in the cluster-specific DNS zone.

## Deploy HTTP routing: CLI

The HTTP application routing add-on can be enabled with the Azure CLI when deploying an AKS cluster. To do so, use the `az aks create` command with the `--enable-addons` argument.

```
az aks create --resource-group myResourceGroup --name myAKSCluster --enable-addons http_application_routing
```

**TIP**

If you want to enable multiple add-ons, provide them as a comma-separated list. For example, to enable HTTP application routing and monitoring, use the format `--enable-addons http_application_routing,monitoring`.

You can also enable HTTP routing on an existing AKS cluster using the `az aks enable-addons` command. To enable HTTP routing on an existing cluster, add the `--addons` parameter and specify `http_application_routing` as shown in the following example:

```
az aks enable-addons --resource-group myResourceGroup --name myAKSCluster --addons http_application_routing
```

After the cluster is deployed or updated, use the `az aks show` command to retrieve the DNS zone name. This name is needed to deploy applications to the AKS cluster.

```
az aks show --resource-group myResourceGroup --name myAKSCluster --query addonProfiles.httpApplicationRouting.config.HTTPApplicationRoutingZoneName -o table
```

Result

9f9c1fe7-21a1-416d-99cd-3543bb92e4c3.eastus.aksapp.io

## Deploy HTTP routing: Portal

The HTTP application routing add-on can be enabled through the Azure portal when deploying an AKS cluster.

You can change networking settings for your cluster, including enabling HTTP application routing and configuring your network using either the 'Basic' or 'Advanced' options:

- 'Basic' networking creates a new VNet for your cluster using default values.
- 'Advanced' networking allows clusters to use a new or existing VNet with customizable addresses. Application pods are connected directly to the VNet, which allows for native integration with VNet features.

Learn more about networking in Azure Kubernetes Service

HTTP application routing ⓘ  Yes  No

Load balancer ⓘ Standard

Network configuration ⓘ  Basic  Advanced

[Review + create](#) [< Previous](#) [Next : Monitoring >](#)

After the cluster is deployed, browse to the auto-created AKS resource group and select the DNS zone. Take note of the DNS zone name. This name is needed to deploy applications to the AKS cluster.

Home > Resource groups > MC\_myAKSCluster\_myAKSCluster\_westeurope > 8d61f730-e2a7-4e57-93ab-48a331a3ee54.westeurope.aksapp.io

DNS zone

8d61f730-e2a7-4e57-93ab-48a331a3ee54.westeurope.aksapp.io

Record set Move Delete zone Refresh

Resource group (change)  
mc\_myakscluster\_myakscluster\_westeurope

Subscription (change)  
Microsoft Internal - Billable

Subscription ID

Name server 1  
ns1-01.azure-dns.com.  
Name server 2  
ns2-01.azure-dns.net.  
Name server 3  
ns3-01.azure-dns.org.  
Name server 4  
ns4-01.azure-dns.info.

Tags (change)  
Click here to add tags

NAME	TYPE	TTL	VALUE
@	NS	172800	ns1-01.azure-dns.com. ns2-01.azure-dns.net. ns3-01.azure-dns.org. ns4-01.azure-dns.info.
@	SOA	3600	Email: azuredns-hostmaster.microsoft.com Host: ns1-01.azure-dns.com. Refresh: 3600 Retry: 300 Expire: 2419200 Minimum TTL: 300 Serial number: 1

## Use HTTP routing

The HTTP application routing solution may only be triggered on Ingress resources that are annotated as follows:

```
annotations:  
  kubernetes.io/ingress.class: addon-http-application-routing
```

Create a file named **samples-http-application-routing.yaml** and copy in the following YAML. On line 43, update `<CLUSTER_SPECIFIC_DNS_ZONE>` with the DNS zone name collected in the previous step of this article.

```
apiVersion: extensions/v1beta1  
kind: Deployment  
metadata:  
  name: party-clippy  
spec:  
  template:  
    metadata:  
      labels:  
        app: party-clippy  
    spec:  
      containers:  
      - image: r.j3ss.co/party-clippy  
        name: party-clippy  
        resources:  
          requests:  
            cpu: 100m  
            memory: 128Mi  
          limits:  
            cpu: 250m  
            memory: 256Mi  
        tty: true  
        command: ["party-clippy"]  
        ports:  
        - containerPort: 8080  
---  
apiVersion: v1  
kind: Service  
metadata:  
  name: party-clippy  
spec:  
  ports:  
  - port: 80  
    protocol: TCP  
    targetPort: 8080  
  selector:  
    app: party-clippy  
  type: ClusterIP  
---  
apiVersion: extensions/v1beta1  
kind: Ingress  
metadata:  
  name: party-clippy  
  annotations:  
    kubernetes.io/ingress.class: addon-http-application-routing  
spec:  
  rules:  
  - host: party-clippy.<CLUSTER_SPECIFIC_DNS_ZONE>  
    http:  
      paths:  
      - backend:  
          serviceName: party-clippy  
          servicePort: 80  
        path: /
```

Use the [kubectl apply](#) command to create the resources.

```
$ kubectl apply -f samples-http-application-routing.yaml  
  
deployment "party-clippy" created  
service "party-clippy" created  
ingress "party-clippy" created
```

Use cURL or a browser to navigate to the hostname specified in the host section of the samples-`http-application-routing.yaml` file. The application can take up to one minute before it's available via the internet.

```
$ curl party-clippy.471756a6-e744-4aa0-aa01-89c4d162a7a7.canadaeast.aksapp.io

/ It looks like you're building a \
\ microservice.                  /
-----
\
\
  / \
  |   |
  @   @
  |   |
  ||  ||
  ||  ||
  |\_|
  \_\_/

```

# Remove HTTP routing

The HTTP routing solution can be removed using the Azure CLI. To do so run the following command, substituting your AKS cluster and resource group name.

```
az aks disable-addons --addons http_application_routing --name myAKSCluster --resource-group myResourceGroup  
--no-wait
```

When the HTTP application routing add-on is disabled, some Kubernetes resources may remain in the cluster. These resources include *configMaps* and *secrets*, and are created in the *kube-system* namespace. To maintain a clean cluster you may want to remove these resources.

Look for `addon-http-application-routing` resources using the following `kubectl get` commands:

```
kubectl get deployments --namespace kube-system  
kubectl get services --namespace kube-system  
kubectl get configmaps --namespace kube-system  
kubectl get secrets --namespace kube-system
```

The following example output shows configMaps that should be deleted:

```
$ kubectl get configmaps --namespace kube-system
```

NAMESPACE	NAME	DATA	AGE
kube-system	addon-http-application-routing-nginx-configuration	0	9m7s
kube-system	addon-http-application-routing-tcp-services	0	9m7s
kube-system	addon-http-application-routing-udp-services	0	9m7s

To delete resources, use the `kubectl delete` command. Specify the resource type, resource name, and namespace.

The following example deletes one of the previous configmaps:

```
kubectl delete configmaps addon-http-application-routing-nginx-configuration --namespace kube-system
```

Repeat the previous `kubectl delete` step for all *addon-http-application-routing* resources that remained in your cluster.

## Troubleshoot

Use the [kubectl logs](#) command to view the application logs for the External-DNS application. The logs should confirm that an A and TXT DNS record were created successfully.

```
$ kubectl logs -f deploy/addon-http-application-routing-external-dns -n kube-system

time="2018-04-26T20:36:19Z" level=info msg="Updating A record named 'party-clippy' to '52.242.28.189' for
Azure DNS zone '471756a6-e744-4aa0-aa01-89c4d162a7a7.canadaeast.aksapp.io'."
time="2018-04-26T20:36:21Z" level=info msg="Updating TXT record named 'party-clippy' to '"heritage=external-
dns,external-dns/owner=default"' for Azure DNS zone '471756a6-e744-4aa0-aa01-
89c4d162a7a7.canadaeast.aksapp.io'."
```

These records can also be seen on the DNS zone resource in the Azure portal.

NAME	TYPE	TTL	VALUE
@	NS	172800	ns1-01.azure-dns.com. ns2-01.azure-dns.net. ns3-01.azure-dns.org. ns4-01.azure-dns.info.
@	SOA	3600	Email: azuredns-hostmaster.microsoft.com Host: ns1-01.azure-dns.com. Refresh: 3600 Retry: 300 Expire: 2419200 Minimum TTL: 300 Serial number: 1
party-clippy	A	300	40.91.216.190
party-clippy	TXT	300	"heritage=external-dns,external-dns/owner=default"

Use the [kubectl logs](#) command to view the application logs for the Nginx Ingress controller. The logs should confirm the `CREATE` of an Ingress resource and the reload of the controller. All HTTP activity is logged.

```
$ kubectl logs -f deploy/addon-http-application-routing-nginx-ingress-controller -n kube-system

-----
NGINX Ingress controller
  Release:    0.13.0
  Build:      git-4bc943a
  Repository: https://github.com/kubernetes/ingress-nginx
-----

I0426 20:30:12.212936      9 flags.go:162] Watching for ingress class: addon-http-application-routing
W0426 20:30:12.213041      9 flags.go:165] only Ingress with class "addon-http-application-routing" will be
processed by this ingress controller
W0426 20:30:12.213505      9 client_config.go:533] Neither --kubeconfig nor --master was specified. Using
the inClusterConfig. This might not work.
I0426 20:30:12.213752      9 main.go:181] Creating API client for https://10.0.0.1:443
I0426 20:30:12.287928      9 main.go:225] Running in Kubernetes Cluster version v1.8 (v1.8.11) - git (clean)
commit 1df6a8381669a6c753f79cb31ca2e3d57ee7c8a3 - platform linux/amd64
I0426 20:30:12.290988      9 main.go:84] validated kube-system/addon-http-application-routing-default-http-
backend as the default backend
I0426 20:30:12.294314      9 main.go:105] service kube-system/addon-http-application-routing-nginx-ingress
validated as source of Ingress status
I0426 20:30:12.426443      9 stat_collector.go:77] starting new nginx stats collector for Ingress controller
running in namespace (class addon-http-application-routing)
I0426 20:30:12.426509      9 stat_collector.go:78] collector extracting information from port 18080
I0426 20:30:12.448779      9 nginx.go:281] starting Ingress controller
I0426 20:30:12.463585      9 event.go:218] Event(v1.ObjectReference{Kind:"ConfigMap", Namespace:"kube-
system", Name:"addon-http-application-routing-nginx-configuration", UID:"2588536c-4990-11e8-a5e1-
0a58ac1f0ef2", APIVersion:"v1", ResourceVersion:"559", FieldPath:""}): type: 'Normal' reason: 'CREATE'
ConfigMap kube-system/addon-http-application-routing-nginx-configuration
I0426 20:30:12.466945      9 event.go:218] Event(v1.ObjectReference{Kind:"ConfigMap", Namespace:"kube-
system", Name:"addon-http-application-routing-tcp-services", UID:"258ca065-4990-11e8-a5e1-0a58ac1f0ef2",
APIVersion:"v1", ResourceVersion:"561", FieldPath:""}): type: 'Normal' reason: 'CREATE' ConfigMap kube-
system/addon-http-application-routing-tcp-services
I0426 20:30:12.467053      9 event.go:218] Event(v1.ObjectReference{Kind:"ConfigMap", Namespace:"kube-
system", Name:"addon-http-application-routing-udp-services", UID:"259023bc-4990-11e8-a5e1-0a58ac1f0ef2",
APIVersion:"v1", ResourceVersion:"562", FieldPath:""}): type: 'Normal' reason: 'CREATE' ConfigMap kube-
system/addon-http-application-routing-udp-services
I0426 20:30:13.649195      9 nginx.go:302] starting NGINX process...
I0426 20:30:13.649347      9 leaderelection.go:175] attempting to acquire leader lease  kube-system/ingress-
controller-leader-addon-http-application-routing...
I0426 20:30:13.649776      9 controller.go:170] backend reload required
I0426 20:30:13.649800      9 stat_collector.go:34] changing prometheus collector from  to default
I0426 20:30:13.662191      9 leaderelection.go:184] successfully acquired lease  kube-system/ingress-
controller-leader-addon-http-application-routing
I0426 20:30:13.662292      9 status.go:196] new leader elected: addon-http-application-routing-nginx-
ingress-controller-5cxntd6
I0426 20:30:13.763362      9 controller.go:179] ingress backend successfully reloaded...
I0426 21:51:55.249327      9 event.go:218] Event(v1.ObjectReference{Kind:"Ingress", Namespace:"default",
Name:"party-clippy", UID:"092c9599-499c-11e8-a5e1-0a58ac1f0ef2", APIVersion:"extensions",
ResourceVersion:"7346", FieldPath:""}): type: 'Normal' reason: 'CREATE' Ingress default/party-clippy
W0426 21:51:57.908771      9 controller.go:775] service default/party-clippy does not have any active
endpoints
I0426 21:51:57.908951      9 controller.go:170] backend reload required
I0426 21:51:58.042932      9 controller.go:179] ingress backend successfully reloaded...
167.220.24.46 - [167.220.24.46] - - [26/Apr/2018:21:53:20 +0000] "GET / HTTP/1.1" 200 234 "" "Mozilla/5.0
(compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0)" 197 0.001 [default-party-clippy-80] 10.244.0.13:8080 234
0.004 200
```

## Clean up

Remove the associated Kubernetes objects created in this article.

```
$ kubectl delete -f samples-http-application-routing.yaml

deployment "party-clippy" deleted
service "party-clippy" deleted
ingress "party-clippy" deleted
```

## Next steps

For information on how to install an HTTPS-secured Ingress controller in AKS, see [HTTPS Ingress on Azure Kubernetes Service \(AKS\)](#).

# Create an ingress controller to an internal virtual network in Azure Kubernetes Service (AKS)

2/25/2020 • 7 minutes to read • [Edit Online](#)

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the [NGINX ingress controller](#) in an Azure Kubernetes Service (AKS) cluster. The ingress controller is configured on an internal, private virtual network and IP address. No external access is allowed. Two applications are then run in the AKS cluster, each of which is accessible over the single IP address.

You can also:

- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses your own TLS certificates](#)
- Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates [with a dynamic public IP address](#) or [with a static public IP address](#)

## Before you begin

This article uses Helm to install the NGINX ingress controller, cert-manager, and a sample web app. You need to have Helm initialized within your AKS cluster and using a service account for Tiller. For more information on configuring and using Helm, see [Install applications with Helm in Azure Kubernetes Service \(AKS\)](#).

This article also requires that you are running the Azure CLI version 2.0.64 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an ingress controller

By default, an NGINX ingress controller is created with a dynamic public IP address assignment. A common configuration requirement is to use an internal, private network and IP address. This approach allows you to restrict access to your services to internal users, with no external access.

Create a file named *internal-ingress.yaml* using the following example manifest file. This example assigns 10.240.0.42 to the *loadBalancerIP* resource. Provide your own internal IP address for use with the ingress controller. Make sure that this IP address is not already in use within your virtual network.

```
controller:  
  service:  
    loadBalancerIP: 10.240.0.42  
    annotations:  
      service.beta.kubernetes.io/azure-load-balancer-internal: "true"
```

Now deploy the *nginx-ingress* chart with Helm. To use the manifest file created in the previous step, add the `-f internal-ingress.yaml` parameter. For added redundancy, two replicas of the NGINX ingress controllers are deployed with the `--set controller.replicaCount` parameter. To fully benefit from running replicas of the ingress controller, make sure there's more than one node in your AKS cluster.

The ingress controller also needs to be scheduled on a Linux node. Windows Server nodes (currently in preview in AKS) shouldn't run the ingress controller. A node selector is specified using the `--set nodeSelector` parameter to tell the Kubernetes scheduler to run the NGINX ingress controller on a Linux-based node.

#### TIP

The following example creates a Kubernetes namespace for the ingress resources named *ingress-basic*. Specify a namespace for your own environment as needed. If your AKS cluster is not RBAC enabled, add `--set rbac.create=false` to the Helm commands.

#### TIP

If you would like to enable [client source IP preservation](#) for requests to containers in your cluster, add `--set controller.service.externalTrafficPolicy=Local` to the Helm install command. The client source IP is stored in the request header under *X-Forwarded-For*. When using an ingress controller with client source IP preservation enabled, SSL pass-through will not work.

```
# Create a namespace for your ingress resources
kubectl create namespace ingress-basic

# Use Helm to deploy an NGINX ingress controller
helm install stable/nginx-ingress \
  --namespace ingress-basic \
  -f internal-ingress.yaml \
  --set controller.replicaCount=2 \
  --set controller.nodeSelector."beta\\.kubernetes\\.io/os"=linux \
  --set defaultBackend.nodeSelector."beta\\.kubernetes\\.io/os"=linux
```

When the Kubernetes load balancer service is created for the NGINX ingress controller, your internal IP address is assigned, as shown in the following example output:

```
$ kubectl get service -l app=nginx-ingress --namespace ingress-basic

NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP     PORT(S)
AGE
alternating-coral-nginx-ingress-controller   LoadBalancer  10.0.97.109  10.240.0.42
80:31507/TCP,443:30707/TCP    1m
alternating-coral-nginx-ingress-default-backend ClusterIP    10.0.134.66  <none>        80/TCP
1m
```

No ingress rules have been created yet, so the NGINX ingress controller's default 404 page is displayed if you browse to the internal IP address. Ingress rules are configured in the following steps.

## Run demo applications

To see the ingress controller in action, let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple 'Hello world' application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment as follows:

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create the first demo application from a Helm chart with the following command:

```
helm install azure-samples/aks-helloworld --namespace ingress-basic
```

Now install a second instance of the demo application. For the second instance, you specify a new title so that the two applications are visually distinct. You also specify a unique service name:

```
helm install azure-samples/aks-helloworld \
--namespace ingress-basic \
--set title="AKS Ingress Demo" \
--set serviceName="ingress-demo"
```

## Create an ingress route

Both applications are now running on your Kubernetes cluster. To route traffic to each application, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to the address `http://10.240.0.42/` is routed to the service named `aks-helloworld`. Traffic to the address `http://10.240.0.42/hello-world-two` is routed to the `ingress-demo` service.

Create a file named `hello-world-ingress.yaml` and copy in the following example YAML.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  namespace: ingress-basic
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
  - http:
    paths:
    - backend:
        serviceName: aks-helloworld
        servicePort: 80
        path: /(.*)
    - backend:
        serviceName: ingress-demo
        servicePort: 80
        path: /hello-world-two(/|$(.).*)
```

Create the ingress resource using the `kubectl apply -f hello-world-ingress.yaml` command.

```
$ kubectl apply -f hello-world-ingress.yaml
ingress.extensions/hello-world-ingress created
```

## Test the ingress controller

To test the routes for the ingress controller, browse to the two applications with a web client. If needed, you can quickly test this internal-only functionality from a pod on the AKS cluster. Create a test pod and attach a terminal session to it:

```
kubectl run -it --rm aks-ingress-test --image=debian --namespace ingress-basic
```

Install `curl` in the pod using `apt-get`:

```
apt-get update && apt-get install -y curl
```

Now access the address of your Kubernetes ingress controller using `curl`, such as <http://10.240.0.42>. Provide your own internal IP address specified when you deployed the ingress controller in the first step of this article.

```
curl -L http://10.240.0.42
```

No additional path was provided with the address, so the ingress controller defaults to the / route. The first demo application is returned, as shown in the following condensed example output:

```
$ curl -L 10.240.0.42

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <link rel="stylesheet" type="text/css" href="/static/default.css">
  <title>Welcome to Azure Kubernetes Service (AKS)</title>
[...]
```

Now add `/hello-world-two` path to the address, such as <http://10.240.0.42/hello-world-two>. The second demo application with the custom title is returned, as shown in the following condensed example output:

```
$ curl -L -k http://10.240.0.42/hello-world-two

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <link rel="stylesheet" type="text/css" href="/static/default.css">
  <title>AKS Ingress Demo</title>
[...]
```

## Clean up resources

This article used Helm to install the ingress components and sample apps. When you deploy a Helm chart, a number of Kubernetes resources are created. These resources includes pods, deployments, and services. To clean up these resources, you can either delete the entire sample namespace, or the individual resources.

### Delete the sample namespace and all resources

To delete the entire sample namespace, use the `kubectl delete` command and specify your namespace name. All the resources in the namespace are deleted.

```
kubectl delete namespace ingress-basic
```

Then, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

### Delete resources individually

Alternatively, a more granular approach is to delete the individual resources created. List the Helm releases with the `helm list` command. Look for charts named *nginx-ingress* and *aks-helloworld*, as shown in the following example output:

```
$ helm list
```

NAME	REVISION	UPDATED	STATUS	CHART	APP VERSION	NAMESPACE
kissing-ferret	1	Tue Oct 16 17:13:39 2018	DEPLOYED	nginx-ingress-0.22.1	0.15.0	kube-system
intended-lemur	1	Tue Oct 16 17:20:59 2018	DEPLOYED	aks-helloworld-0.1.0		default
pioneering-wombat	1	Tue Oct 16 17:21:05 2018	DEPLOYED	aks-helloworld-0.1.0		default

Delete the releases with the `helm delete` command. The following example deletes the NGINX ingress deployment, and the two sample AKS hello world apps.

```
$ helm delete kissing-ferret intended-lemur pioneering-wombat
```

```
release "kissing-ferret" deleted
release "intended-lemur" deleted
release "pioneering-wombat" deleted
```

Next, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

Remove the ingress route that directed traffic to the sample apps:

```
kubectl delete -f hello-world-ingress.yaml
```

Finally, you can delete the itself namespace. Use the `kubectl delete` command and specify your namespace name:

```
kubectl delete namespace ingress-basic
```

## Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- [Helm CLI](#)
- [NGINX ingress controller](#)

You can also:

- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates](#)
- [Create an ingress controller with a static public IP address and configure Let's Encrypt to automatically generate TLS certificates](#)

# Create an HTTPS ingress controller and use your own TLS certificates on Azure Kubernetes Service (AKS)

2/25/2020 • 9 minutes to read • [Edit Online](#)

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the [NGINX ingress controller](#) in an Azure Kubernetes Service (AKS) cluster. You generate your own certificates, and create a Kubernetes secret for use with the ingress route. Finally, two applications are run in the AKS cluster, each of which is accessible over a single IP address.

You can also:

- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates with a dynamic public IP address or with a static public IP address](#)

## Before you begin

This article uses Helm to install the NGINX ingress controller and a sample web app. You need to have Helm initialized within your AKS cluster and using a service account for Tiller. Make sure that you are using the latest release of Helm. For upgrade instructions, see the [Helm install docs](#). For more information on configuring and using Helm, see [Install applications with Helm in Azure Kubernetes Service \(AKS\)](#).

This article also requires that you are running the Azure CLI version 2.0.64 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an ingress controller

To create the ingress controller, use `Helm` to install `nginx-ingress`. For added redundancy, two replicas of the NGINX ingress controllers are deployed with the `--set controller.replicaCount` parameter. To fully benefit from running replicas of the ingress controller, make sure there's more than one node in your AKS cluster.

The ingress controller also needs to be scheduled on a Linux node. Windows Server nodes (currently in preview in AKS) shouldn't run the ingress controller. A node selector is specified using the `--set nodeSelector` parameter to tell the Kubernetes scheduler to run the NGINX ingress controller on a Linux-based node.

### TIP

The following example creates a Kubernetes namespace for the ingress resources named `ingress-basic`. Specify a namespace for your own environment as needed. If your AKS cluster is not RBAC enabled, add `--set rbac.create=false` to the Helm commands.

## TIP

If you would like to enable [client source IP preservation](#) for requests to containers in your cluster, add

```
--set controller.service.externalTrafficPolicy=Local
```

 to the Helm install command. The client source IP is stored in the request header under *X-Forwarded-For*. When using an ingress controller with client source IP preservation enabled, SSL pass-through will not work.

```
# Create a namespace for your ingress resources
kubectl create namespace ingress-basic

# Use Helm to deploy an NGINX ingress controller
helm install stable/nginx-ingress \
--namespace ingress-basic \
--set controller.replicaCount=2 \
--set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \
--set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux
```

During the installation, an Azure public IP address is created for the ingress controller. This public IP address is static for the life-span of the ingress controller. If you delete the ingress controller, the public IP address assignment is lost. If you then create an additional ingress controller, a new public IP address is assigned. If you wish to retain the use of the public IP address, you can instead [create an ingress controller with a static public IP address](#).

To get the public IP address, use the `kubectl get service` command. It takes a few minutes for the IP address to be assigned to the service.

```
$ kubectl get service -l app=nginx-ingress --namespace ingress-basic

NAME           TYPE        CLUSTER-IP      EXTERNAL-IP     PORT(S)
AGE
virulent-seal-nginx-ingress-controller   LoadBalancer   10.0.48.240   40.87.46.190
80:31159/TCP,443:30657/TCP    7m
virulent-seal-nginx-ingress-default-backend ClusterIP     10.0.50.5     <none>          80/TCP
7m
```

Make a note of this public IP address, as it's used in the last step to test the deployment.

No ingress rules have been created yet. If you browse to the public IP address, the NGINX ingress controller's default 404 page is displayed.

## Generate TLS certificates

For this article, let's generate a self-signed certificate with `openssl`. For production use, you should request a trusted, signed certificate through a provider or your own certificate authority (CA). In the next step, you generate a Kubernetes *Secret* using the TLS certificate and private key generated by OpenSSL.

The following example generates a 2048-bit RSA X509 certificate valid for 365 days named *aks-ingress-tls.crt*. The private key file is named *aks-ingress-tls.key*. A Kubernetes TLS secret requires both of these files.

This article works with the *demo.azure.com* subject common name and doesn't need to be changed. For production use, specify your own organizational values for the `-subj` parameter:

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
-out aks-ingress-tls.crt \
-keyout aks-ingress-tls.key \
-subj "/CN=demo.azure.com/O=aks-ingress-tls"
```

## Create Kubernetes secret for the TLS certificate

To allow Kubernetes to use the TLS certificate and private key for the ingress controller, you create and use a Secret. The secret is defined once, and uses the certificate and key file created in the previous step. You then reference this secret when you define ingress routes.

The following example creates a Secret name *aks-ingress-tls*:

```
kubectl create secret tls aks-ingress-tls \
--namespace ingress-basic \
--key aks-ingress-tls.key \
--cert aks-ingress-tls.crt
```

## Run demo applications

An ingress controller and a Secret with your certificate have been configured. Now let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple 'Hello world' application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment as follows:

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create the first demo application from a Helm chart with the following command:

```
helm install azure-samples/aks-helloworld --namespace ingress-basic
```

Now install a second instance of the demo application. For the second instance, you specify a new title so that the two applications are visually distinct. You also specify a unique service name:

```
helm install azure-samples/aks-helloworld \
--namespace ingress-basic \
--set title="AKS Ingress Demo" \
--set serviceName="ingress-demo"
```

## Create an ingress route

Both applications are now running on your Kubernetes cluster, however they're configured with a service of type `ClusterIP`. As such, the applications aren't accessible from the internet. To make them publicly available, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to the address `https://demo.azure.com/` is routed to the service named `aks-helloworld`. Traffic to the address `https://demo.azure.com/hello-world-two` is routed to the `ingress-demo` service. For this article, you don't need to change those demo host names. For production use, provide the names specified as part of the certificate request and generation process.

## TIP

If the host name specified during the certificate request process, the CN name, doesn't match the host defined in your ingress route, your ingress controller displays a *Kubernetes Ingress Controller Fake Certificate* warning. Make sure your certificate and ingress route host names match.

The *tls* section tells the ingress route to use the Secret named *aks-ingress-tls* for the host *demo.azure.com*. Again, for production use, specify your own host address.

Create a file named `hello-world-ingress.yaml` and copy in the following example YAML.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  namespace: ingress-basic
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  tls:
  - hosts:
    - demo.azure.com
    secretName: aks-ingress-tls
  rules:
  - host: demo.azure.com
    http:
      paths:
      - backend:
          serviceName: aks-helloworld
          servicePort: 80
          path: /(.*)
      - backend:
          serviceName: ingress-demo
          servicePort: 80
          path: /hello-world-two(/|$(.).*)
```

Create the ingress resource using the `kubectl apply -f hello-world-ingress.yaml` command.

```
$ kubectl apply -f hello-world-ingress.yaml
ingress.extensions/hello-world-ingress created
```

## Test the ingress configuration

To test the certificates with our fake *demo.azure.com* host, use `curl` and specify the *--resolve* parameter. This parameter lets you map the *demo.azure.com* name to the public IP address of your ingress controller. Specify the public IP address of your own ingress controller, as shown in the following example:

```
curl -v -k --resolve demo.azure.com:443:40.87.46.190 https://demo.azure.com
```

No additional path was provided with the address, so the ingress controller defaults to the */* route. The first demo application is returned, as shown in the following condensed example output:

```
$ curl -v -k --resolve demo.azure.com:443:40.87.46.190 https://demo.azure.com

[...]
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <link rel="stylesheet" type="text/css" href="/static/default.css">
  <title>Welcome to Azure Kubernetes Service (AKS)</title>
[...]
```

The `-v` parameter in our `curl` command outputs verbose information, including the TLS certificate received. Half-way through your curl output, you can verify that your own TLS certificate was used. The `-k` parameter continues loading the page even though we're using a self-signed certificate. The following example shows that the `issuer: CN=demo.azure.com; O=aks-ingress-tls` certificate was used:

```
[...]
* Server certificate:
*   subject: CN=demo.azure.com; O=aks-ingress-tls
*   start date: Oct 22 22:13:54 2018 GMT
*   expire date: Oct 22 22:13:54 2019 GMT
*   issuer: CN=demo.azure.com; O=aks-ingress-tls
*   SSL certificate verify result: self signed certificate (18), continuing anyway.
[...]
```

Now add `/hello-world-two` path to the address, such as `https://demo.azure.com/hello-world-two`. The second demo application with the custom title is returned, as shown in the following condensed example output:

```
$ curl -v -k --resolve demo.azure.com:443:137.117.36.18 https://demo.azure.com/hello-world-two

[...]
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <link rel="stylesheet" type="text/css" href="/static/default.css">
  <title>AKS Ingress Demo</title>
[...]
```

## Clean up resources

This article used Helm to install the ingress components and sample apps. When you deploy a Helm chart, a number of Kubernetes resources are created. These resources include pods, deployments, and services. To clean up these resources, you can either delete the entire sample namespace, or the individual resources.

### Delete the sample namespace and all resources

To delete the entire sample namespace, use the `kubectl delete` command and specify your namespace name. All the resources in the namespace are deleted.

```
kubectl delete namespace ingress-basic
```

Then, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

### Delete resources individually

Alternatively, a more granular approach is to delete the individual resources created. List the Helm releases with

the `helm list` command. Look for charts named *nginx-ingress* and *aks-helloworld*, as shown in the following example output:

```
$ helm list

NAME      REVISION UPDATED      STATUS      CHART          APP VERSION NAMESPACE
virulent-seal  1      Tue Oct 23 16:37:24 2018 DEPLOYED nginx-ingress-0.22.1 0.15.0      kube-system
billowing-guppy 1      Tue Oct 23 16:41:38 2018 DEPLOYED aks-helloworld-0.1.0      default
listless-quokka 1      Tue Oct 23 16:41:30 2018 DEPLOYED aks-helloworld-0.1.0      default
```

Delete the releases with the `helm delete` command. The following example deletes the NGINX ingress deployment and the two sample AKS hello world apps.

```
$ helm delete virulent-seal billowing-guppy listless-quokka

release "virulent-seal" deleted
release "billowing-guppy" deleted
release "listless-quokka" deleted
```

Next, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

Remove the ingress route that directed traffic to the sample apps:

```
kubectl delete -f hello-world-ingress.yaml
```

Delete the certificate Secret:

```
kubectl delete secret aks-ingress-tls
```

Finally, you can delete the itself namespace. Use the `kubectl delete` command and specify your namespace name:

```
kubectl delete namespace ingress-basic
```

## Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- [Helm CLI](#)
- [NGINX ingress controller](#)

You can also:

- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates with a dynamic public IP address or with a static public IP address](#)

# Create an HTTPS ingress controller on Azure Kubernetes Service (AKS)

2/25/2020 • 9 minutes to read • [Edit Online](#)

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the [NGINX ingress controller](#) in an Azure Kubernetes Service (AKS) cluster. The [cert-manager](#) project is used to automatically generate and configure [Let's Encrypt](#) certificates. Finally, two applications are run in the AKS cluster, each of which is accessible over a single IP address.

You can also:

- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller that uses your own TLS certificates](#)
- [Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates with a static public IP address](#)

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the [AKS quickstart using the Azure CLI](#) or [using the Azure portal](#).

This article also assumes you have [a custom domain](#) with a [DNS Zone](#) in the same resource group as your AKS cluster.

This article uses Helm to install the NGINX ingress controller, cert-manager, and a sample web app. Make sure that you are using the latest release of Helm. For upgrade instructions, see the [Helm install docs](#). For more information on configuring and using Helm, see [Install applications with Helm in Azure Kubernetes Service \(AKS\)](#).

This article also requires that you are running the Azure CLI version 2.0.64 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an ingress controller

To create the ingress controller, use the `helm` command to install `nginx-ingress`. For added redundancy, two replicas of the NGINX ingress controllers are deployed with the `--set controller.replicaCount` parameter. To fully benefit from running replicas of the ingress controller, make sure there's more than one node in your AKS cluster.

The ingress controller also needs to be scheduled on a Linux node. Windows Server nodes (currently in preview in AKS) shouldn't run the ingress controller. A node selector is specified using the `--set nodeSelector` parameter to tell the Kubernetes scheduler to run the NGINX ingress controller on a Linux-based node.

**TIP**

The following example creates a Kubernetes namespace for the ingress resources named *ingress-basic*. Specify a namespace for your own environment as needed.

**TIP**

If you would like to enable [client source IP preservation](#) for requests to containers in your cluster, add

```
--set controller.service.externalTrafficPolicy=Local
```

 to the Helm install command. The client source IP is stored in the request header under *X-Forwarded-For*. When using an ingress controller with client source IP preservation enabled, SSL pass-through will not work.

```
# Create a namespace for your ingress resources
kubectl create namespace ingress-basic

# Add the official stable repo
helm repo add stable https://kubernetes-charts.storage.googleapis.com/

# Use Helm to deploy an NGINX ingress controller
helm install nginx stable/nginx-ingress \
  --namespace ingress-basic \
  --set controller.replicaCount=2 \
  --set controller.nodeSelector."beta\\.kubernetes\\.io/os"=linux \
  --set defaultBackend.nodeSelector."beta\\.kubernetes\\.io/os"=linux
```

During the installation, an Azure public IP address is created for the ingress controller. This public IP address is static for the life-span of the ingress controller. If you delete the ingress controller, the public IP address assignment is lost. If you then create an additional ingress controller, a new public IP address is assigned. If you wish to retain the use of the public IP address, you can instead [create an ingress controller with a static public IP address](#).

To get the public IP address, use the `kubectl get service` command. It takes a few minutes for the IP address to be assigned to the service.

```
$ kubectl get service -l app=nginx-ingress --namespace ingress-basic

NAME                TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
billowing-kitten-nginx-ingress-controller   LoadBalancer   10.0.182.160  MY_EXTERNAL_IP
80:30920/TCP,443:30426/TCP    20m
billowing-kitten-nginx-ingress-default-backend ClusterIP     10.0.255.77   <none>
20m
```

No ingress rules have been created yet. If you browse to the public IP address, the NGINX ingress controller's default 404 page is displayed.

## Add an A record to your DNS zone

Add an A record to your DNS zone with the external IP address of the NGINX service using [az network dns record-set a add-record](#).

```
az network dns record-set a add-record \
--resource-group myResourceGroup \
--zone-name MY_CUSTOM_DOMAIN \
--record-set-name * \
--ipv4-address MY_EXTERNAL_IP
```

#### NOTE

Optionally, you can configure an FQDN for the ingress controller IP address instead of a custom domain. Note that this sample is for a Bash shell.

```
# Public IP address of your ingress controller
IP="MY_EXTERNAL_IP"

# Name to associate with public IP address
DNSNAME="demo-aks-ingress"

# Get the resource-id of the public ip
PUBLICIPID=$(az network public-ip list --query "[?ipAddress!=null] | [?contains(ipAddress, '$IP')].[id]" -o tsv)

# Update public ip address with DNS name
az network public-ip update --ids $PUBLICIPID --dns-name $DNSNAME

# Display the FQDN
az network public-ip show --ids $PUBLICIPID --query "[dnsSettings.fqdn]" --output tsv
```

## Install cert-manager

The NGINX ingress controller supports TLS termination. There are several ways to retrieve and configure certificates for HTTPS. This article demonstrates using [cert-manager](#), which provides automatic [Lets Encrypt](#) certificate generation and management functionality.

To install the cert-manager controller:

```
# Install the CustomResourceDefinition resources separately
kubectl apply --validate=false -f https://raw.githubusercontent.com/jetstack/cert-manager/release-0.12/deploy/manifests/00-crd.yaml --namespace ingress-basic

# Label the ingress-basic namespace to disable resource validation
kubectl label namespace ingress-basic certmanager.k8s.io/disable-validation=true

# Add the Jetstack Helm repository
helm repo add jetstack https://charts.jetstack.io

# Update your local Helm chart repository cache
helm repo update

# Install the cert-manager Helm chart
helm install cert-manager --namespace ingress-basic --version v0.12.0 jetstack/cert-manager --set ingressShim.defaultIssuerName=letsencrypt --set ingressShim.defaultIssuerKind=ClusterIssuer
```

For more information on cert-manager configuration, see the [cert-manager project](#).

## Create a CA cluster issuer

Before certificates can be issued, cert-manager requires an [Issuer](#) or [ClusterIssuer](#) resource. These Kubernetes resources are identical in functionality, however [Issuer](#) works in a single namespace, and [ClusterIssuer](#) works across all namespaces. For more information, see the [cert-manager issuer](#) documentation.

Create a cluster issuer, such as `cluster-issuer.yaml`, using the following example manifest. Update the email address with a valid address from your organization:

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt
spec:
  acme:
    server: https://acme-v02.api.letsencrypt.org/directory
    email: MY_EMAIL_ADDRESS
    privateKeySecretRef:
      name: letsencrypt
    solvers:
      - http01:
          ingress:
            class: nginx
```

To create the issuer, use the `kubectl apply` command.

```
kubectl apply -f cluster-issuer.yaml --namespace ingress-basic
```

## Run demo applications

An ingress controller and a certificate management solution have been configured. Now let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple *Hello world* application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment.

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create a demo application named *aks-helloworld* using the *azure-samples/aks-helloworld* Helm chart.

```
helm install aks-helloworld azure-samples/aks-helloworld --namespace ingress-basic
```

Create a second instance of the demo application named *aks-helloworld-two*. Specify a new title and unique service name so that the two applications are visually distinct using `--set`.

```
helm install aks-helloworld-two azure-samples/aks-helloworld \
  --namespace ingress-basic \
  --set title="AKS Ingress Demo" \
  --set serviceName="aks-helloworld-two"
```

## Create an ingress route

Both applications are now running on your Kubernetes cluster. However they're configured with a service of type `ClusterIP` and aren't accessible from the internet. To make them publicly available, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to the address *hello-world-ingress.MY\_CUSTOM\_DOMAIN* is routed to the *aks-helloworld* service. Traffic to the address *hello-world-ingress.MY\_CUSTOM\_DOMAIN/hello-world-two* is routed to the *aks-helloworld-two* service. Traffic to *hello-world-ingress.MY\_CUSTOM\_DOMAIN/static* is routed to the service named *aks-helloworld* for static assets.

Create a file named `hello-world-ingress.yaml` using below example YAML. Update the `hosts` and `host` to the DNS name you created in a previous step.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /$2
    cert-manager.io/cluster-issuer: letsencrypt
spec:
  tls:
    - hosts:
        - hello-world-ingress.MY_CUSTOM_DOMAIN
      secretName: tls-secret
  rules:
    - host: hello-world-ingress.MY_CUSTOM_DOMAIN
      http:
        paths:
          - backend:
              serviceName: aks-helloworld
              servicePort: 80
              path: /(.*)
          - backend:
              serviceName: aks-helloworld-two
              servicePort: 80
              path: /hello-world-two(/|$(.).*)
---
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress-static
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/rewrite-target: /static/$2
    cert-manager.io/cluster-issuer: letsencrypt
spec:
  tls:
    - hosts:
        - hello-world-ingress.MY_CUSTOM_DOMAIN
      secretName: tls-secret
  rules:
    - host: hello-world-ingress.MY_CUSTOM_DOMAIN
      http:
        paths:
          - backend:
              serviceName: aks-helloworld
              servicePort: 80
              path: /static(/|$(.).*)
```

Create the ingress resource using the `kubectl apply` command.

```
kubectl apply -f hello-world-ingress.yaml --namespace ingress-basic
```

## Verify a certificate object has been created

Next, a certificate resource must be created. The certificate resource defines the desired X.509 certificate. For more information, see [cert-manager certificates](#). Cert-manager has automatically created a certificate object for you using ingress-shim, which is automatically deployed with cert-manager since v0.2.2. For more information, see the [ingress-shim documentation](#).

To verify that the certificate was created successfully, use the `kubectl get certificate --namespace ingress-basic` command and verify *READY* is *True*, which may take several minutes.

```
$ kubectl get certificate --namespace ingress-basic

NAME      READY   SECRET      AGE
tls-secret  True    tls-secret  11m
```

## Test the ingress configuration

Open a web browser to *hello-world-ingress.MY\_CUSTOM\_DOMAIN* of your Kubernetes ingress controller. Notice you are redirect to use HTTPS and the certificate is trusted and the demo application is shown in the web browser. Add the */hello-world-two* path and notice the second demo application with the custom title is shown.

## Clean up resources

This article used Helm to install the ingress components, certificates, and sample apps. When you deploy a Helm chart, a number of Kubernetes resources are created. These resources includes pods, deployments, and services. To clean up these resources, you can either delete the entire sample namespace, or the individual resources.

### Delete the sample namespace and all resources

To delete the entire sample namespace, use the `kubectl delete` command and specify your namespace name. All the resources in the namespace are deleted.

```
kubectl delete namespace ingress-basic
```

Then, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

### Delete resources individually

Alternatively, a more granular approach is to delete the individual resources created. First, remove the cluster issuer resources:

```
kubectl delete -f cluster-issuer.yaml --namespace ingress-basic
```

List the Helm releases with the `helm list` command. Look for charts named *nginx-ingress* and *aks-helloworld*, as shown in the following example output:

```
$ helm list --namespace ingress-basic

NAME        NAMESPACE      REVISION      UPDATED           STATUS
CHART          APP VERSION
aks-helloworld  ingress-basic  1            2020-01-15 10:24:32.054871 -0600 CST  deployed
aks-helloworld-0.1.0
aks-helloworld-two  ingress-basic  1            2020-01-15 10:24:37.671667 -0600 CST  deployed
aks-helloworld-0.1.0
cert-manager    ingress-basic  1            2020-01-15 10:23:36.515514 -0600 CST  deployed
cert-manager-v0.12.0
nginx          ingress-basic  1            2020-01-15 10:09:45.982693 -0600 CST  deployed
nginx-ingress-1.29.1
                           0.27.0
```

Delete the releases with the `helm delete` command. The following example deletes the NGINX ingress

deployment, and the two sample AKS hello world apps.

```
$ helm delete aks-helloworld aks-helloworld-two cert-manager nginx --namespace ingress-basic  
release "aks-helloworld" uninstalled  
release "aks-helloworld-two" uninstalled  
release "cert-manager" uninstalled  
release "nginx" uninstalled
```

Next, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

Remove the ingress route that directed traffic to the sample apps:

```
kubectl delete -f hello-world-ingress.yaml --namespace ingress-basic
```

Finally, you can delete the itself namespace. Use the `kubectl delete` command and specify your namespace name:

```
kubectl delete namespace ingress-basic
```

## Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- [Helm CLI](#)
- [NGINX ingress controller](#)
- [cert-manager](#)

You can also:

- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller that uses your own TLS certificates](#)
- [Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates with a static public IP address](#)

# Create an ingress controller with a static public IP address in Azure Kubernetes Service (AKS)

2/25/2020 • 11 minutes to read • [Edit Online](#)

An ingress controller is a piece of software that provides reverse proxy, configurable traffic routing, and TLS termination for Kubernetes services. Kubernetes ingress resources are used to configure the ingress rules and routes for individual Kubernetes services. Using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

This article shows you how to deploy the [NGINX ingress controller](#) in an Azure Kubernetes Service (AKS) cluster. The ingress controller is configured with a static public IP address. The [cert-manager](#) project is used to automatically generate and configure [Let's Encrypt](#) certificates. Finally, two applications are run in the AKS cluster, each of which is accessible over a single IP address.

You can also:

- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses your own TLS certificates](#)
- [Create an ingress controller that uses Let's Encrypt to automatically generate TLS certificates with a dynamic public IP address](#)

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the [Azure CLI](#) or [using the Azure portal](#).

This article uses Helm to install the NGINX ingress controller, cert-manager, and a sample web app. You need to have Helm initialized within your AKS cluster and using a service account for Tiller. Make sure that you are using the latest release of Helm 3. For upgrade instructions, see the [Helm install docs](#). For more information on configuring and using Helm, see [Install applications with Helm in Azure Kubernetes Service \(AKS\)](#).

This article also requires that you are running the Azure CLI version 2.0.64 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an ingress controller

By default, an NGINX ingress controller is created with a new public IP address assignment. This public IP address is only static for the life-span of the ingress controller, and is lost if the controller is deleted and re-created. A common configuration requirement is to provide the NGINX ingress controller an existing static public IP address. The static public IP address remains if the ingress controller is deleted. This approach allows you to use existing DNS records and network configurations in a consistent manner throughout the lifecycle of your applications.

If you need to create a static public IP address, first get the resource group name of the AKS cluster with the `az aks show` command:

```
az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv
```

Next, create a public IP address with the *static* allocation method using the `az network public-ip create`

command. The following example creates a public IP address named *myAKSPublicIP* in the AKS cluster resource group obtained in the previous step:

```
az network public-ip create --resource-group MC_myResourceGroup_myAKScluster_eastus --name myAKSPublicIP --sku Standard --allocation-method static --query publicIp.ipAddress -o tsv
```

Now deploy the *nginx-ingress* chart with Helm. Add the `--set controller.service.loadBalancerIP` parameter, and specify your own public IP address created in the previous step. For added redundancy, two replicas of the NGINX ingress controllers are deployed with the `--set controller.replicaCount` parameter. To fully benefit from running replicas of the ingress controller, make sure there's more than one node in your AKS cluster.

The ingress controller also needs to be scheduled on a Linux node. Windows Server nodes (currently in preview in AKS) shouldn't run the ingress controller. A node selector is specified using the `--set nodeSelector` parameter to tell the Kubernetes scheduler to run the NGINX ingress controller on a Linux-based node.

#### TIP

The following example creates a Kubernetes namespace for the ingress resources named *ingress-basic*. Specify a namespace for your own environment as needed. If your AKS cluster is not RBAC enabled, add `--set rbac.create=false` to the Helm commands.

#### TIP

If you would like to enable [client source IP preservation](#) for requests to containers in your cluster, add `--set controller.service.externalTrafficPolicy=Local` to the Helm install command. The client source IP is stored in the request header under *X-Forwarded-For*. When using an ingress controller with client source IP preservation enabled, SSL pass-through will not work.

```
# Create a namespace for your ingress resources
kubectl create namespace ingress-basic

# Use Helm to deploy an NGINX ingress controller
helm install nginx-ingress stable/nginx-ingress \
  --namespace ingress-basic \
  --set controller.replicaCount=2 \
  --set controller.nodeSelector."beta\\.kubernetes\\.io/os"=linux \
  --set defaultBackend.nodeSelector."beta\\.kubernetes\\.io/os"=linux \
  --set controller.service.loadBalancerIP="40.121.63.72"
```

When the Kubernetes load balancer service is created for the NGINX ingress controller, your static IP address is assigned, as shown in the following example output:

```
$ kubectl get service -l app=nginx-ingress --namespace ingress-basic

NAME                   TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)
AGE
nginx-ingress-controller   LoadBalancer  10.0.232.56  40.121.63.72
80:31978/TCP,443:32037/TCP  3m
nginx-ingress-default-backend ClusterIP  10.0.95.248  <none>          80/TCP
3m
```

No ingress rules have been created yet, so the NGINX ingress controller's default 404 page is displayed if you browse to the public IP address. Ingress rules are configured in the following steps.

## Configure a DNS name

For the HTTPS certificates to work correctly, configure an FQDN for the ingress controller IP address. Update the following script with the IP address of your ingress controller and a unique name that you would like to use for the FQDN:

```
#!/bin/bash

# Public IP address of your ingress controller
IP="40.121.63.72"

# Name to associate with public IP address
DNSNAME="demo-aks-ingress"

# Get the resource-id of the public ip
PUBLICIPID=$(az network public-ip list --query "[?ipAddress!=null] | [?contains(ipAddress, '$IP')].[id]" --output tsv)

# Update public ip address with DNS name
az network public-ip update --ids $PUBLICIPID --dns-name $DNSNAME
```

The ingress controller is now accessible through the FQDN.

## Install cert-manager

The NGINX ingress controller supports TLS termination. There are several ways to retrieve and configure certificates for HTTPS. This article demonstrates using [cert-manager](#), which provides automatic [Lets Encrypt](#) certificate generation and management functionality.

### NOTE

This article uses the `staging` environment for Let's Encrypt. In production deployments, use `letsencrypt-prod` and <https://acme-v02.api.letsencrypt.org/directory> in the resource definitions and when installing the Helm chart.

To install the cert-manager controller in an RBAC-enabled cluster, use the following `helm install` command:

```
# Install the CustomResourceDefinition resources separately
kubectl apply --validate=false -f https://raw.githubusercontent.com/jetstack/cert-manager/release-0.12/deploy/manifests/00-crds.yaml

# Create the namespace for cert-manager
kubectl create namespace cert-manager

# Label the cert-manager namespace to disable resource validation
kubectl label namespace cert-manager cert-manager.io/disable-validation=true

# Add the Jetstack Helm repository
helm repo add jetstack https://charts.jetstack.io

# Update your local Helm chart repository cache
helm repo update

# Install the cert-manager Helm chart
helm install \
  cert-manager \
  --namespace cert-manager \
  --version v0.12.0 \
  jetstack/cert-manager
```

For more information on cert-manager configuration, see the [cert-manager project](#).

## Create a CA cluster issuer

Before certificates can be issued, cert-manager requires an [Issuer](#) or [ClusterIssuer](#) resource. These Kubernetes resources are identical in functionality, however [Issuer](#) works in a single namespace, and [ClusterIssuer](#) works across all namespaces. For more information, see the [cert-manager issuer](#) documentation.

Create a cluster issuer, such as `cluster-issuer.yaml`, using the following example manifest. Update the email address with a valid address from your organization:

```
apiVersion: cert-manager.io/v1alpha2
kind: ClusterIssuer
metadata:
  name: letsencrypt-staging
  namespace: ingress-basic
spec:
  acme:
    server: https://acme-staging-v02.api.letsencrypt.org/directory
    email: user@contoso.com
    privateKeySecretRef:
      name: letsencrypt-staging
    solvers:
      - http01:
          ingress:
            class: nginx
```

To create the issuer, use the `kubectl apply -f cluster-issuer.yaml` command.

```
$ kubectl apply -f cluster-issuer.yaml
clusterissuer.cert-manager.io/letsencrypt-staging created
```

## Run demo applications

An ingress controller and a certificate management solution have been configured. Now let's run two demo applications in your AKS cluster. In this example, Helm is used to deploy two instances of a simple 'Hello world' application.

Before you can install the sample Helm charts, add the Azure samples repository to your Helm environment as follows:

```
helm repo add azure-samples https://azure-samples.github.io/helm-charts/
```

Create the first demo application from a Helm chart with the following command:

```
helm install aks-helloworld azure-samples/aks-helloworld --namespace ingress-basic
```

Now install a second instance of the demo application. For the second instance, you specify a new title so that the two applications are visually distinct. You also specify a unique service name:

```
helm install aks-helloworld-2 azure-samples/aks-helloworld \
--namespace ingress-basic \
--set title="AKS Ingress Demo" \
--set serviceName="ingress-demo"
```

## Create an ingress route

Both applications are now running on your Kubernetes cluster, however they're configured with a service of type `ClusterIP`. As such, the applications aren't accessible from the internet. To make them publicly available, create a Kubernetes ingress resource. The ingress resource configures the rules that route traffic to one of the two applications.

In the following example, traffic to the address `https://demo-aks-ingress.eastus.cloudapp.azure.com/` is routed to the service named `aks-helloworld`. Traffic to the address

`https://demo-aks-ingress.eastus.cloudapp.azure.com/hello-world-two` is routed to the `ingress-demo` service.

Update the `hosts` and `host` to the DNS name you created in a previous step.

Create a file named `hello-world-ingress.yaml` and copy in the following example YAML.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-world-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
    cert-manager.io/cluster-issuer: letsencrypt-staging
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  tls:
  - hosts:
    - demo-aks-ingress.eastus.cloudapp.azure.com
    secretName: tls-secret
  rules:
  - host: demo-aks-ingress.eastus.cloudapp.azure.com
    http:
      paths:
      - backend:
          serviceName: aks-helloworld
          servicePort: 80
          path: /(.*)
      - backend:
          serviceName: ingress-demo
          servicePort: 80
          path: /hello-world-two(/|$(.).*)
```

Create the ingress resource using the `kubectl apply -f hello-world-ingress.yaml --namespace ingress-basic` command.

```
$ kubectl apply -f hello-world-ingress.yaml --namespace ingress-basic
ingress.extensions/hello-world-ingress created
```

## Create a certificate object

Next, a certificate resource must be created. The certificate resource defines the desired X.509 certificate. For more information, see [cert-manager certificates](#).

Cert-manager has likely automatically created a certificate object for you using ingress-shim, which is automatically deployed with cert-manager since v0.2.2. For more information, see the [ingress-shim documentation](#).

To verify that the certificate was created successfully, use the

`kubectl describe certificate tls-secret --namespace ingress-basic` command.

If the certificate was issued, you will see output similar to the following:

Type	Reason	Age	From	Message
Normal	CreateOrder	11m	cert-manager	Created new ACME order, attempting validation...
Normal	DomainVerified	10m	cert-manager	Domain "demo-aks-ingress.eastus.cloudapp.azure.com" verified with "http-01" validation
Normal	IssueCert	10m	cert-manager	Issuing certificate...
Normal	CertObtained	10m	cert-manager	Obtained certificate from ACME server
Normal	CertIssued	10m	cert-manager	Certificate issued successfully

If you need to create an additional certificate resource, you can do so with the following example manifest. Update the *dnsNames* and *domains* to the DNS name you created in a previous step. If you use an internal-only ingress controller, specify the internal DNS name for your service.

```
apiVersion: cert-manager.io/v1alpha2
kind: Certificate
metadata:
  name: tls-secret
  namespace: ingress-basic
spec:
  secretName: tls-secret
  dnsNames:
    - demo-aks-ingress.eastus.cloudapp.azure.com
  acme:
    config:
      - http01:
          ingressClass: nginx
          domains:
            - demo-aks-ingress.eastus.cloudapp.azure.com
  issuerRef:
    name: letsencrypt-staging
    kind: ClusterIssuer
```

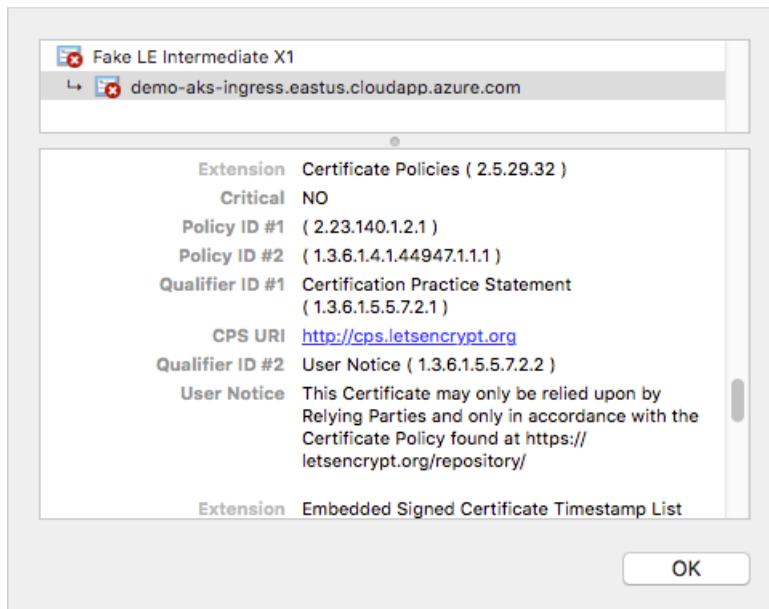
To create the certificate resource, use the `kubectl apply -f certificates.yaml` command.

```
$ kubectl apply -f certificates.yaml
certificate.cert-manager.io/tls-secret created
```

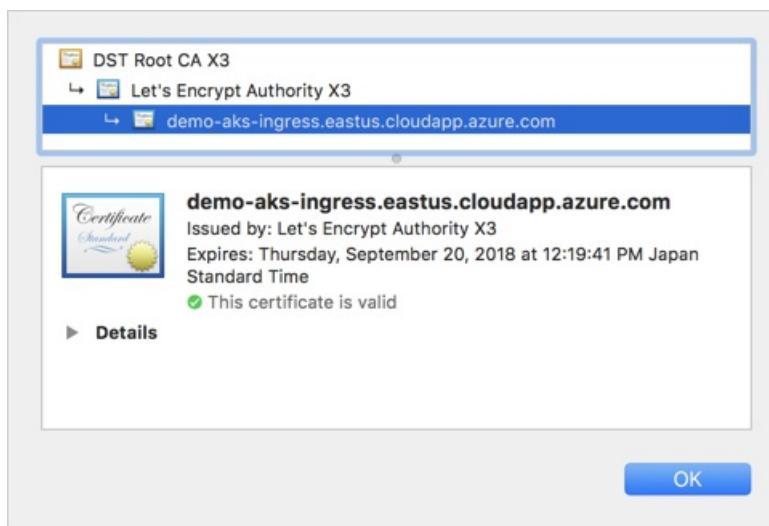
## Test the ingress configuration

Open a web browser to the FQDN of your Kubernetes ingress controller, such as <https://demo-aks-ingress.eastus.cloudapp.azure.com>.

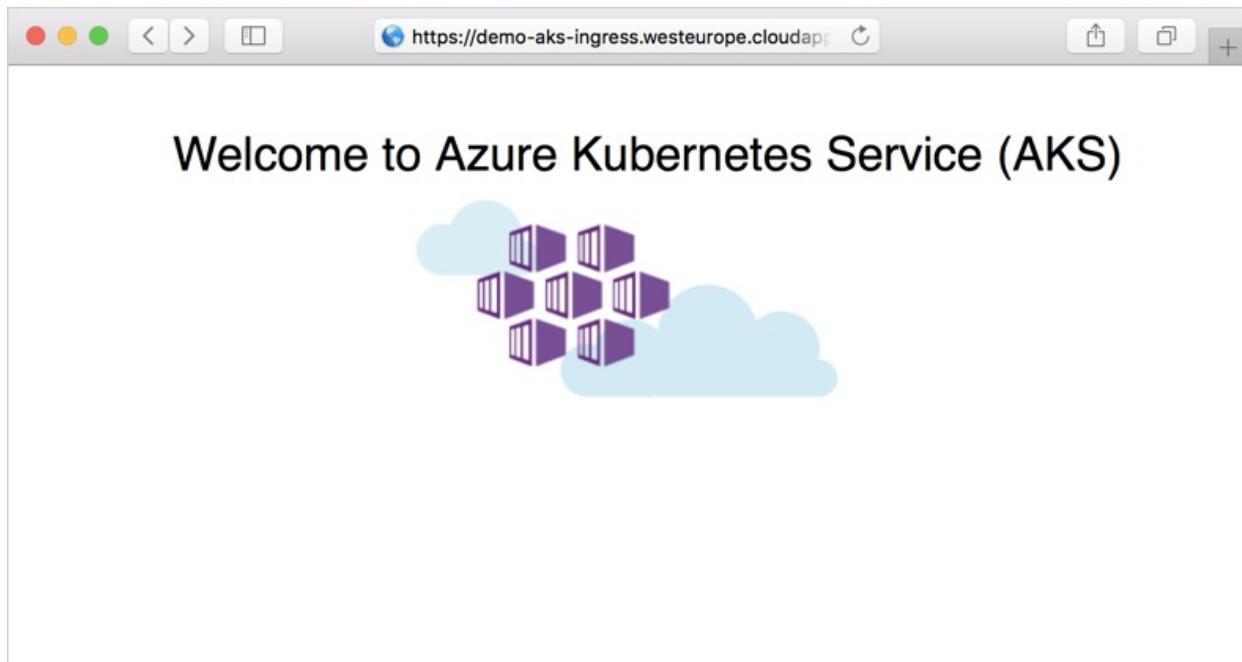
As these examples use `letsencrypt-staging`, the issued SSL certificate is not trusted by the browser. Accept the warning prompt to continue to your application. The certificate information shows this *Fake LE Intermediate X1* certificate is issued by Let's Encrypt. This fake certificate indicates `cert-manager` processed the request correctly and received a certificate from the provider:



When you change Let's Encrypt to use `prod` rather than `staging`, a trusted certificate issued by Let's Encrypt is used, as shown in the following example:

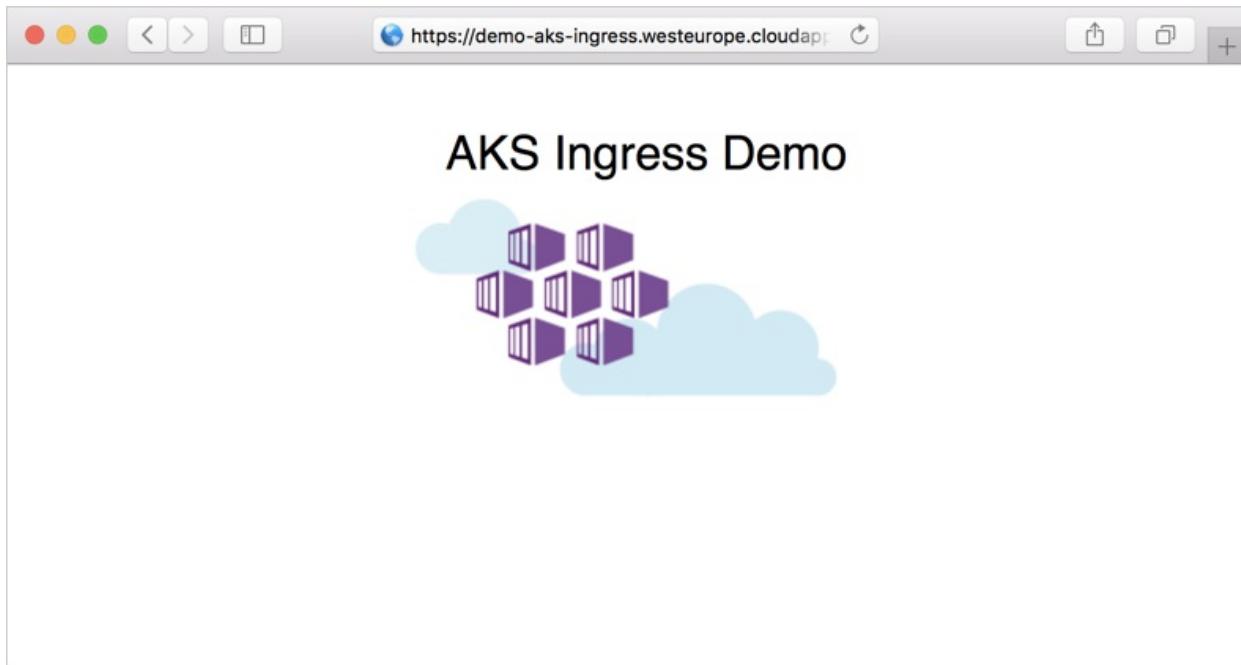


The demo application is shown in the web browser:



Now add the `/hello-world-two` path to the FQDN, such as <https://demo-aks->

[ingress.eastus.cloudapp.azure.com/hello-world-two](https://ingress.eastus.cloudapp.azure.com/hello-world-two). The second demo application with the custom title is shown:



## Clean up resources

This article used Helm to install the ingress components, certificates, and sample apps. When you deploy a Helm chart, a number of Kubernetes resources are created. These resources includes pods, deployments, and services. To clean up these resources, you can either delete the entire sample namespace, or the individual resources.

### Delete the sample namespace and all resources

To delete the entire sample namespace, use the `kubectl delete` command and specify your namespace name. All the resources in the namespace are deleted.

```
kubectl delete namespace ingress-basic
```

Then, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

### Delete resources individually

Alternatively, a more granular approach is to delete the individual resources created. First, remove the certificate resources:

```
kubectl delete -f certificates.yaml  
kubectl delete -f cluster-issuer.yaml
```

Now list the Helm releases with the `helm list` command. Look for charts named *nginx-ingress*, *cert-manager*, and *aks-helloworld*, as shown in the following example output:

NAME	NAMESPACE	REVISION	UPDATED	STATUS	
CHART	APP VERSION				
aks-helloworld	ingress-basic	1	2020-01-11 15:02:21.51172346	deployed	aks-
helloworld-0.1.0					
aks-helloworld-2	ingress-basic	1	2020-01-11 15:03:10.533465598	deployed	aks-
helloworld-0.1.0					
nginx-ingress	ingress-basic	1	2020-01-11 14:51:03.454165006	deployed	
nginx-ingress-1.28.2	0.26.2				
cert-manager	cert-manager	1	2020-01-06 21:19:03.866212286	deployed	
cert-manager-v0.12.0	v0.12.0				

Delete the releases with the `helm uninstall` command. The following example deletes the NGINX ingress deployment, certificate manager, and the two sample AKS hello world apps.

```
$ helm uninstall aks-helloworld aks-helloworld-2 nginx-ingress -n ingress-basic

release "aks-helloworld" deleted
release "aks-helloworld-2" deleted
release "nginx-ingress" deleted

$ helm uninstall cert-manager -n cert-manager

release "cert-manager" deleted
```

Next, remove the Helm repo for the AKS hello world app:

```
helm repo remove azure-samples
```

Delete the `itself` namespace. Use the `kubectl delete` command and specify your namespace name:

```
kubectl delete namespace ingress-basic
```

Finally, remove the static public IP address created for the ingress controller. Provide your `MC_` cluster resource group name obtained in the first step of this article, such as `MC_myResourceGroup_myAKSCluster_eastus`:

```
az network public-ip delete --resource-group MC_myResourceGroup_myAKSCluster_eastus --name myAKSPublicIP
```

## Next steps

This article included some external components to AKS. To learn more about these components, see the following project pages:

- [Helm CLI](#)
- [NGINX ingress controller](#)
- [cert-manager](#)

You can also:

- [Create a basic ingress controller with external network connectivity](#)
- [Enable the HTTP application routing add-on](#)
- [Create an ingress controller that uses an internal, private network and IP address](#)
- [Create an ingress controller that uses your own TLS certificates](#)

- Create an ingress controller with a dynamic public IP and configure Let's Encrypt to automatically generate TLS certificates

# Use a static public IP address for egress traffic in Azure Kubernetes Service (AKS)

2/25/2020 • 3 minutes to read • [Edit Online](#)

By default, the egress IP address from an Azure Kubernetes Service (AKS) cluster is randomly assigned. This configuration is not ideal when you need to identify an IP address for access to external services, for example. Instead, you may need to assign a static IP address that can be whitelisted for service access.

This article shows you how to create and use a static public IP address for use with egress traffic in an AKS cluster.

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the [AKS quickstart using the Azure CLI](#) or [using the Azure portal](#).

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Egress traffic overview

Outbound traffic from an AKS cluster follows [Azure Load Balancer conventions](#). Before the first Kubernetes service of type `LoadBalancer` is created, the agent nodes in an AKS cluster are not part of any Azure Load Balancer pool. In this configuration, the nodes have no instance level Public IP address. Azure translates the outbound flow to a public source IP address that is not configurable or deterministic.

Once a Kubernetes service of type `LoadBalancer` is created, agent nodes are added to an Azure Load Balancer pool. For outbound flow, Azure translates it to the first public IP address configured on the load balancer. This public IP address is only valid for the lifespan of that resource. If you delete the Kubernetes LoadBalancer service, the associated load balancer and IP address are also deleted. If you want to assign a specific IP address or retain an IP address for redeployed Kubernetes services, you can create and use a static public IP address.

## Create a static public IP

Get the resource group name with the `az aks show` command and add the `--query nodeResourceGroup` query parameter. The following example gets the node resource group for the AKS cluster name `myAKSCluster` in the resource group name `myResourceGroup`:

```
$ az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv  
MC_myResourceGroup_myAKSCluster_eastus
```

Now create a static public IP address with the `az network public-ip create` command. Specify the node resource group name obtained in the previous command, and then a name for the IP address resource, such as `myAKSPublicIP`:

```
az network public-ip create \  
--resource-group MC_myResourceGroup_myAKSCluster_eastus \  
--name myAKSPublicIP \  
--allocation-method static
```

The IP address is shown, as shown in the following condensed example output:

```
{  
  "publicIp": {  
    "dnsSettings": null,  
    "etag": "W/\"6b6fb15c-5281-4f64-b332-8f68f46e1358\"",  
    "id":  
      "/subscriptions/<SubscriptionID>/resourceGroups/MC_myResourceGroup_myAKScluster_eastus/providers/Microsoft.Network/publicIPAddresses/myAKSPublicIP",  
    "idleTimeoutInMinutes": 4,  
    "ipAddress": "40.121.183.52",  
    [...]  
  }  
}
```

You can later get the public IP address using the [az network public-ip list](#) command. Specify the name of the node resource group, and then query for the *ipAddress* as shown in the following example:

```
$ az network public-ip list --resource-group MC_myResourceGroup_myAKScluster_eastus --query [0].ipAddress --  
output tsv  
  
40.121.183.52
```

## Create a service with the static IP

To create a service with the static public IP address, add the `loadBalancerIP` property and the value of the static public IP address to the YAML manifest. Create a file named `egress-service.yaml` and copy in the following YAML. Provide your own public IP address created in the previous step.

```
apiVersion: v1  
kind: Service  
metadata:  
  name: azure-egress  
spec:  
  loadBalancerIP: 40.121.183.52  
  type: LoadBalancer  
  ports:  
    - port: 80
```

Create the service and deployment with the `kubectl apply` command.

```
kubectl apply -f egress-service.yaml
```

This service configures a new frontend IP on the Azure Load Balancer. If you do not have any other IPs configured, then **all** egress traffic should now use this address. When multiple addresses are configured on the Azure Load Balancer, egress uses the first IP on that load balancer.

## Verify egress address

To verify that the static public IP address is being used, you can use DNS look-up service such as `checkip.dyndns.org`.

Start and attach to a basic *Debian* pod:

```
kubectl run -it --rm aks-ip --image=debian --generator=run-pod/v1
```

To access a web site from within the container, use `apt-get` to install `curl` into the container.

```
apt-get update && apt-get install curl -y
```

Now use curl to access the `checkip.dyndns.org` site. The egress IP address is shown, as displayed in the following example output. This IP address matches the static public IP address created and defined for the loadBalancer service:

```
$ curl -s checkip.dyndns.org
<html><head><title>Current IP Check</title></head><body>Current IP Address: 40.121.183.52</body></html>
```

## Next steps

To avoid maintaining multiple public IP addresses on the Azure Load Balancer, you can instead use an ingress controller. Ingress controllers provide additional benefits such as SSL/TLS termination, support for URI rewrites, and upstream SSL/TLS encryption. For more information, see [Create a basic ingress controller in AKS](#).

# Customize CoreDNS with Azure Kubernetes Service

2/25/2020 • 4 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) uses the [CoreDNS](#) project for cluster DNS management and resolution with all 1.12.x and higher clusters. Previously, the kube-dns project was used. This kube-dns project is now deprecated. For more information about CoreDNS customization and Kubernetes, see the [official upstream documentation](#).

As AKS is a managed service, you cannot modify the main configuration for CoreDNS (a *CoreFile*). Instead, you use a Kubernetes *ConfigMap* to override the default settings. To see the default AKS CoreDNS ConfigMaps, use the `kubectl get configmaps --namespace=kube-system coredns -o yaml` command.

This article shows you how to use ConfigMaps for basic customization options of CoreDNS in AKS. This approach differs from configuring CoreDNS in other contexts such as using the *CoreFile*. Verify the version of CoreDNS you are running as the configuration values may change between versions.

## NOTE

`kube-dns` offered different [customization options](#) via a Kubernetes config map. CoreDNS is **not** backwards compatible with kube-dns. Any customizations you previously used must be updated for use with CoreDNS.

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the [AKS quickstart using the Azure CLI](#) or [using the Azure portal](#).

## What is supported/unsupported

All built-in CoreDNS plugins are supported. No add-on/third party plugins are supported.

## Rewrite DNS

One scenario you have is to perform on-the-fly DNS name rewrites. In the following example, replace `<domain to be written>` with your own fully qualified domain name. Create a file named `corednsms.yaml` and paste the following example configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns-custom
  namespace: kube-system
data:
  test.server: |
    <domain to be rewritten>.com:53 {
      errors
      cache 30
      rewrite name substring <domain to be rewritten>.com default.svc.cluster.local
      forward . /etc/resolv.conf # you can redirect this to a specific DNS server such as 10.0.0.10
    }
```

Create the ConfigMap using the `kubectl apply configmap` command and specify the name of your YAML manifest:

```
kubectl apply -f corednsms.yaml
```

To verify the customizations have been applied, use the [kubectl get configmaps](#) and specify your `coredns-custom` ConfigMap:

```
kubectl get configmaps --namespace=kube-system coredns-custom -o yaml
```

Now force CoreDNS to reload the ConfigMap. The [kubectl delete pod](#) command isn't destructive and doesn't cause down time. The `kube-dns` pods are deleted, and the Kubernetes Scheduler then recreates them. These new pods contain the change in TTL value.

```
kubectl delete pod --namespace kube-system -l k8s-app=kube-dns
```

#### NOTE

The command above is correct. While we're changing `coredns`, the deployment is under the `kube-dns` name.

## Custom forward server

If you need to specify a forward server for your network traffic, you can create a ConfigMap to customize DNS. In the following example, update the `forward` name and address with the values for your own environment. Create a file named `corednsms.yaml` and paste the following example configuration:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns-custom
  namespace: kube-system
data:
  test.server: | # you may select any name here, but it must end with the .server file extension
    <domain to be rewritten>.com:53 {
      forward foo.com 1.1.1.1
    }
```

As in the previous examples, create the ConfigMap using the [kubectl apply configmap](#) command and specify the name of your YAML manifest. Then, force CoreDNS to reload the ConfigMap using the [kubectl delete pod](#) for the Kubernetes Scheduler to recreate them:

```
kubectl apply -f corednsms.yaml
kubectl delete pod --namespace kube-system --selector k8s-app=kube-dns
```

## Use custom domains

You may want to configure custom domains that can only be resolved internally. For example, you may want to resolve the custom domain `puglife.local`, which isn't a valid top-level domain. Without a custom domain ConfigMap, the AKS cluster can't resolve the address.

In the following example, update the custom domain and IP address to direct traffic to with the values for your own environment. Create a file named `corednsms.yaml` and paste the following example configuration:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns-custom
  namespace: kube-system
data:
  puglife.server: |
    puglife.local:53 {
      errors
      cache 30
      forward . 192.11.0.1 # this is my test/dev DNS server
    }

```

As in the previous examples, create the ConfigMap using the [kubectl apply configmap](#) command and specify the name of your YAML manifest. Then, force CoreDNS to reload the ConfigMap using the [kubectl delete pod](#) for the Kubernetes Scheduler to recreate them:

```

kubectl apply -f corednsms.yaml
kubectl delete pod --namespace kube-system --selector k8s-app=kube-dns

```

## Stub domains

CoreDNS can also be used to configure stub domains. In the following example, update the custom domains and IP addresses with the values for your own environment. Create a file named `corednsms.yaml` and paste the following example configuration:

```

apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns-custom
  namespace: kube-system
data:
  test.server: |
    abc.com:53 {
      errors
      cache 30
      forward . 1.2.3.4
    }
  my.cluster.local:53 {
    errors
    cache 30
    forward . 2.3.4.5
  }

```

As in the previous examples, create the ConfigMap using the [kubectl apply configmap](#) command and specify the name of your YAML manifest. Then, force CoreDNS to reload the ConfigMap using the [kubectl delete pod](#) for the Kubernetes Scheduler to recreate them:

```

kubectl apply -f corednsms.yaml
kubectl delete pod --namespace kube-system --selector k8s-app=kube-dns

```

## Hosts plugin

As all built-in plugins are supported this means that the CoreDNS [Hosts](#) plugin is available to customize as well:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns-custom # this is the name of the configmap you can overwrite with your changes
  namespace: kube-system
data:
  test.override: |
    hosts example.hosts example.org { # example.hosts must be a file
      10.0.0.1 example.org
      fallthrough
    }
```

## Enable logging for DNS query debugging

To enable DNS query logging, apply the following configuration in your coredns-custom ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: coredns-custom
  namespace: kube-system
data:
  log.override: |
    log
```

## Next steps

This article showed some example scenarios for CoreDNS customization. For information on the CoreDNS project, see [the CoreDNS upstream project page](#).

To learn more about core network concepts, see [Network concepts for applications in AKS](#).

# Service principals with Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

To interact with Azure APIs, an AKS cluster requires an [Azure Active Directory \(AD\) service principal](#). The service principal is needed to dynamically create and manage other Azure resources such as an Azure load balancer or container registry (ACR).

This article shows how to create and use a service principal for your AKS clusters.

## Before you begin

To create an Azure AD service principal, you must have permissions to register an application with your Azure AD tenant, and to assign the application to a role in your subscription. If you don't have the necessary permissions, you might need to ask your Azure AD or subscription administrator to assign the necessary permissions, or pre-create a service principal for you to use with the AKS cluster.

If you are using a service principal from a different Azure AD tenant, there are additional considerations around the permissions available when you deploy the cluster. You may not have the appropriate permissions to read and write directory information. For more information, see [What are the default user permissions in Azure Active Directory?](#)

You also need the Azure CLI version 2.0.59 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Automatically create and use a service principal

When you create an AKS cluster in the Azure portal or using the `az aks create` command, Azure can automatically generate a service principal.

In the following Azure CLI example, a service principal is not specified. In this scenario, the Azure CLI creates a service principal for the AKS cluster. To successfully complete the operation, your Azure account must have the proper rights to create a service principal.

```
az aks create --name myAKSCluster --resource-group myResourceGroup
```

## Manually create a service principal

To manually create a service principal with the Azure CLI, use the `az ad sp create-for-rbac` command. In the following example, the `--skip-assignment` parameter prevents any additional default assignments being assigned:

```
az ad sp create-for-rbac --skip-assignment --name myAKSClusterServicePrincipal
```

The output is similar to the following example. Make a note of your own `appId` and `password`. These values are used when you create an AKS cluster in the next section.

```
{
  "appId": "559513bd-0c19-4c1a-87cd-851a26af5fc",
  "displayName": "myAKSclusterServicePrincipal",
  "name": "http://myAKSclusterServicePrincipal",
  "password": "e763725a-5eee-40e8-a466-dc88d980f415",
  "tenant": "72f988bf-86f1-41af-91ab-2d7cd011db48"
}
```

## Specify a service principal for an AKS cluster

To use an existing service principal when you create an AKS cluster using the `az aks create` command, use the `--service-principal` and `--client-secret` parameters to specify the `appId` and `password` from the output of the `az ad sp create-for-rbac` command:

```
az aks create \
--resource-group myResourceGroup \
--name myAKScluster \
--service-principal <appId> \
--client-secret <password>
```

### NOTE

If you're using an existing service principal with customized secret, ensure the secret is no longer than 190 bytes.

If you deploy an AKS cluster using the Azure portal, on the *Authentication* page of the **Create Kubernetes cluster** dialog, choose to **Configure service principal**. Select **Use existing**, and specify the following values:

- **Service principal client ID** is your `appId`
- **Service principal client secret** is the `password` value

The screenshot shows the 'Create Kubernetes cluster' dialog in the Azure portal. The 'Authentication' tab is selected. On the left, under 'Cluster infrastructure', there is a 'Service principal' field with '(new) default service principal' and a 'Configure service principal' button. Under 'Kubernetes authentication and authorization', there is a 'Enable RBAC' switch set to 'Yes'. At the bottom, there are 'Review + create', '< Previous', 'Next : Networking >', and 'Ok' buttons.

## Delegate access to other Azure resources

The service principal for the AKS cluster can be used to access other resources. For example, if you want to deploy your AKS cluster into an existing Azure virtual network subnet or connect to Azure Container Registry (ACR), you need to delegate access to those resources to the service principal.

To delegate permissions, create a role assignment using the `az role assignment create` command. Assign the `appId`

to a particular scope, such as a resource group or virtual network resource. A role then defines what permissions the service principal has on the resource, as shown in the following example:

```
az role assignment create --assignee <appId> --scope <resourceScope> --role Contributor
```

The `--scope` for a resource needs to be a full resource ID, such as `/subscriptions/<guid>/resourceGroups/myResourceGroup` or `/subscriptions/<guid>/resourceGroups/myResourceGroupVnet/providers/Microsoft.Network/virtualNetworks/myVnet`

The following sections detail common delegations that you may need to make.

### Azure Container Registry

If you use Azure Container Registry (ACR) as your container image store, you need to grant permissions to the service principal for your AKS cluster to read and pull images. Currently, the recommended configuration is to use the [az aks create](#) or [az aks update](#) command to integrate with a registry and assign the appropriate role for the service principal. For detailed steps, see [Authenticate with Azure Container Registry from Azure Kubernetes Service](#).

### Networking

You may use advanced networking where the virtual network and subnet or public IP addresses are in another resource group. Assign one of the following set of role permissions:

- Create a [custom role](#) and define the following role permissions:
  - `Microsoft.Network/virtualNetworks/subnets/join/action`
  - `Microsoft.Network/virtualNetworks/subnets/read`
  - `Microsoft.Network/virtualNetworks/subnets/write`
  - `Microsoft.Network/publicIPAddresses/join/action`
  - `Microsoft.Network/publicIPAddresses/read`
  - `Microsoft.Network/publicIPAddresses/write`
- Or, assign the [Network Contributor](#) built-in role on the subnet within the virtual network

### Storage

You may need to access existing Disk resources in another resource group. Assign one of the following set of role permissions:

- Create a [custom role](#) and define the following role permissions:
  - `Microsoft.Compute/disks/read`
  - `Microsoft.Compute/disks/write`
- Or, assign the [Storage Account Contributor](#) built-in role on the resource group

### Azure Container Instances

If you use Virtual Kubelet to integrate with AKS and choose to run Azure Container Instances (ACI) in resource group separate to the AKS cluster, the AKS service principal must be granted *Contributor* permissions on the ACI resource group.

## Additional considerations

When using AKS and Azure AD service principals, keep the following considerations in mind.

- The service principal for Kubernetes is a part of the cluster configuration. However, don't use the identity to deploy the cluster.
- By default, the service principal credentials are valid for one year. You can [update or rotate the service principal](#)

credentials at any time.

- Every service principal is associated with an Azure AD application. The service principal for a Kubernetes cluster can be associated with any valid Azure AD application name (for example: <https://www.contoso.org/example>). The URL for the application doesn't have to be a real endpoint.
- When you specify the service principal **Client ID**, use the value of the `appId`.
- On the agent node VMs in the Kubernetes cluster, the service principal credentials are stored in the file `/etc/kubernetes/azure.json`
- When you use the `az aks create` command to generate the service principal automatically, the service principal credentials are written to the file `~/.azure/aksServicePrincipal.json` on the machine used to run the command.
- If you do not specifically pass a service principal in additional AKS CLI commands, the default service principal located at `~/.azure/aksServicePrincipal.json` is used.
- You can also optionally remove the `aksServicePrincipal.json` file, and AKS will create a new service principal.
- When you delete an AKS cluster that was created by `az aks create`, the service principal that was created automatically is not deleted.
  - To delete the service principal, query for your cluster `servicePrincipalProfile.clientId` and then delete with `az ad app delete`. Replace the following resource group and cluster names with your own values:

```
az ad sp delete --id $(az aks show -g myResourceGroup -n myAKSCluster --query servicePrincipalProfile.clientId -o tsv)
```

## Troubleshoot

The service principal credentials for an AKS cluster are cached by the Azure CLI. If these credentials have expired, you encounter errors deploying AKS clusters. The following error message when running `az aks create` may indicate a problem with the cached service principal credentials:

```
Operation failed with status: 'Bad Request'.
Details: The credentials in ServicePrincipalProfile were invalid. Please see https://aka.ms/aks-sp-help for more details.
(Details: adal: Refresh request failed. Status Code = '401'.
```

Check the age of the credentials file using the following command:

```
ls -la $HOME/.azure/aksServicePrincipal.json
```

The default expiration time for the service principal credentials is one year. If your `aksServicePrincipal.json` file is older than one year, delete the file and try to deploy an AKS cluster again.

## Next steps

For more information about Azure Active Directory service principals, see [Application and service principal objects](#).

For information on how to update the credentials, see [Update or rotate the credentials for a service principal in AKS](#).

# Preview - Use managed identities in Azure Kubernetes Service

2/25/2020 • 2 minutes to read • [Edit Online](#)

Currently, an Azure Kubernetes Service (AKS) cluster (specifically, the Kubernetes cloud provider) requires a *service principal* to create additional resources like load balancers and managed disks in Azure. Either you must provide a service principal or AKS creates one on your behalf. Service principals typically have an expiration date. Clusters eventually reach a state in which the service principal must be renewed to keep the cluster working. Managing service principals adds complexity.

*Managed identities* are essentially a wrapper around service principals, and make their management simpler. To learn more, read about [managed identities for Azure resources](#).

AKS creates two managed identities:

- **System-assigned managed identity:** The identity that the Kubernetes cloud provider uses to create Azure resources on behalf of the user. The life cycle of the system-assigned identity is tied to that of the cluster. The identity is deleted when the cluster is deleted.
- **User-assigned managed identity:** The identity that's used for authorization in the cluster. For example, the user-assigned identity is used to authorize AKS to use access control records (ACRs), or to authorize the kubelet to get metadata from Azure.

In this preview period, a service principal is still required. It's used for authorization of add-ons such as monitoring, virtual nodes, Azure Policy, and HTTP application routing. Work is underway to remove the dependency of add-ons on the service principal name (SPN). Eventually, the requirement of an SPN in AKS will be removed completely.

## IMPORTANT

AKS preview features are available on a self-service, opt-in basis. Previews are provided "as-is" and "as available," and are excluded from the Service Level Agreements and limited warranty. AKS previews are partially covered by customer support on best-effort basis. As such, these features are not meant for production use. For more information, see the following support articles:

- [AKS Support Policies](#)
- [Azure Support FAQ](#)

## Before you begin

You must have the following resources installed:

- The Azure CLI, version 2.0.70 or later
- The aks-preview 0.4.14 extension

To install the aks-preview 0.4.14 extension or later, use the following Azure CLI commands:

```
az extension add --name aks-preview  
az extension list
```

## Caution

After you register a feature on a subscription, you can't currently unregister that feature. When you enable some

preview features, defaults might be used for all AKS clusters created afterward in the subscription. Don't enable preview features on production subscriptions. Instead, use a separate subscription to test preview features and gather feedback.

```
az feature register --name MSIPreview --namespace Microsoft.ContainerService
```

It might take several minutes for the status to show as **Registered**. You can check the registration status by using the [az feature list](#) command:

```
az feature list -o table --query "[?contains(name, 'Microsoft.ContainerService/MSIPreview')].{Name:name,State:properties.state}"
```

When the status shows as registered, refresh the registration of the `Microsoft.ContainerService` resource provider by using the [az provider register](#) command:

```
az provider register --namespace Microsoft.ContainerService
```

## Create an AKS cluster with managed identities

You can now create an AKS cluster with managed identities by using the following CLI commands.

First, create an Azure resource group:

```
# Create an Azure resource group
az group create --name myResourceGroup --location westus2
```

Then, create an AKS cluster:

```
az aks create -g MyResourceGroup -n MyManagedCluster --enable-managed-identity
```

Finally, get credentials to access the cluster:

```
az aks get-credentials --resource-group myResourceGroup --name MyManagedCluster
```

The cluster will be created in a few minutes. You can then deploy your application workloads to the new cluster and interact with it just as you've done with service-principal-based AKS clusters.

### IMPORTANT

- AKS clusters with managed identities can be enabled only during creation of the cluster.
- Existing AKS clusters cannot be updated or upgraded to enable managed identities.

# Use Azure role-based access controls to define access to the Kubernetes configuration file in Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

You can interact with Kubernetes clusters using the `kubectl` tool. The Azure CLI provides an easy way to get the access credentials and configuration information to connect to your AKS clusters using `kubectl`. To limit who can get that Kubernetes configuration (*kubeconfig*) information and to limit the permissions they then have, you can use Azure role-based access controls (RBAC).

This article shows you how to assign RBAC roles that limit who can get the configuration information for an AKS cluster.

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart [using the Azure CLI](#) or [using the Azure portal](#).

This article also requires that you are running the Azure CLI version 2.0.65 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Available cluster roles permissions

When you interact with an AKS cluster using the `kubectl` tool, a configuration file is used that defines cluster connection information. This configuration file is typically stored in `~/.kube/config`. Multiple clusters can be defined in this *kubeconfig* file. You switch between clusters using the `kubectl config use-context` command.

The `az aks get-credentials` command lets you get the access credentials for an AKS cluster and merges them into the *kubeconfig* file. You can use Azure role-based access controls (RBAC) to control access to these credentials. These Azure RBAC roles let you define who can retrieve the *kubeconfig* file, and what permissions they then have within the cluster.

The two built-in roles are:

- **Azure Kubernetes Service Cluster Admin Role**
  - Allows access to `Microsoft.ContainerService/managedClusters/listClusterAdminCredential/action` API call. This API call [lists the cluster admin credentials](#).
  - Downloads *kubeconfig* for the *clusterAdmin* role.
- **Azure Kubernetes Service Cluster User Role**
  - Allows access to `Microsoft.ContainerService/managedClusters/listClusterUserCredential/action` API call. This API call [lists the cluster user credentials](#).
  - Downloads *kubeconfig* for *clusterUser* role.

These RBAC roles can be applied to an Azure Active Directory (AD) user or group.

![NOTE] On clusters that use Azure AD, users with the *clusterUser* role have an empty *kubeconfig* file that prompts a log in. Once logged in, users have access based on their Azure AD user or group settings. Users with the *clusterAdmin* role have admin access.

Clusters that do not use Azure AD only use the *clusterAdmin* role.

# Assign role permissions to a user or group

To assign one of the available roles, you need to get the resource ID of the AKS cluster and the ID of the Azure AD user account or group. The following example commands:

- Get the cluster resource ID using the `az aks show` command for the cluster named *myAKSCluster* in the *myResourceGroup* resource group. Provide your own cluster and resource group name as needed.
- Use the `az account show` and `az ad user show` commands to get your user ID.
- Finally, assign a role using the `az role assignment create` command.

The following example assigns the *Azure Kubernetes Service Cluster Admin Role* to an individual user account:

```
# Get the resource ID of your AKS cluster
AKS_CLUSTER=$(az aks show --resource-group myResourceGroup --name myAKSCluster --query id -o tsv)

# Get the account credentials for the logged in user
ACCOUNT_UPN=$(az account show --query user.name -o tsv)
ACCOUNT_ID=$(az ad user show --id $ACCOUNT_UPN --query objectId -o tsv)

# Assign the 'Cluster Admin' role to the user
az role assignment create \
    --assignee $ACCOUNT_ID \
    --scope $AKS_CLUSTER \
    --role "Azure Kubernetes Service Cluster Admin Role"
```

## TIP

If you want to assign permissions to an Azure AD group, update the `--assignee` parameter shown in the previous example with the object ID for the *group* rather than a *user*. To obtain the object ID for a group, use the `az ad group show` command. The following example gets the object ID for the Azure AD group named *appdev*:

```
az ad group show --group appdev --query objectId -o tsv
```

You can change the previous assignment to the *Cluster User Role* as needed.

The following example output shows the role assignment has been successfully created:

```
{
  "canDelegate": null,
  "id": "/subscriptions/<guid>/resourcegroups/myResourceGroup/providers/Microsoft.ContainerService/managedClusters/myAKSCluster/providers/Microsoft.Authorization/roleAssignments/b2712174-5a41-4ecb-82c5-12b8ad43d4fb",
  "name": "b2712174-5a41-4ecb-82c5-12b8ad43d4fb",
  "principalId": "946016dd-9362-4183-b17d-4c416d1f8f61",
  "resourceGroup": "myResourceGroup",
  "roleDefinitionId": "/subscriptions/<guid>/providers/Microsoft.Authorization/roleDefinitions/0ab01a8-8aac-4efd-b8c2-3ee1fb270be8",
  "scope": "/subscriptions/<guid>/resourcegroups/myResourceGroup/providers/Microsoft.ContainerService/managedClusters/myAKSCluster",
  "type": "Microsoft.Authorization/roleAssignments"
}
```

# Get and verify the configuration information

With RBAC roles assigned, use the `az aks get-credentials` command to get the *kubeconfig* definition for your AKS cluster. The following example gets the `--admin` credentials, which work correctly if the user has been granted the *Cluster Admin Role*:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --admin
```

You can then use the [kubectl config view](#) command to verify that the *context* for the cluster shows that the admin configuration information has been applied:

```
$ kubectl config view

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://myaksclust-myresourcegroup-19da35-4839be06.hcp.eastus.azmk8s.io:443
    name: myAKSCluster
contexts:
- context:
    cluster: myAKSCluster
    user: clusterAdmin_myResourceGroup_myAKSCluster
    name: myAKSCluster-admin
current-context: myAKSCluster-admin
kind: Config
preferences: {}
users:
- name: clusterAdmin_myResourceGroup_myAKSCluster
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED
    token: e9f2f819a4496538b02cefff94e61d35
```

## Remove role permissions

To remove role assignments, use the [az role assignment delete](#) command. Specify the account ID and cluster resource ID, as obtained in the previous commands. If you assigned the role to a group rather than a user, specify the appropriate group object ID rather than account object ID for the `--assignee` parameter:

```
az role assignment delete --assignee $ACCOUNT_ID --scope $AKS_CLUSTER
```

## Next steps

For enhanced security on access to AKS clusters, [integrate Azure Active Directory authentication](#).

# Secure traffic between pods using network policies in Azure Kubernetes Service (AKS)

2/25/2020 • 12 minutes to read • [Edit Online](#)

When you run modern, microservices-based applications in Kubernetes, you often want to control which components can communicate with each other. The principle of least privilege should be applied to how traffic can flow between pods in an Azure Kubernetes Service (AKS) cluster. Let's say you likely want to block traffic directly to back-end applications. The *Network Policy* feature in Kubernetes lets you define rules for ingress and egress traffic between pods in a cluster.

This article shows you how to install the network policy engine and create Kubernetes network policies to control the flow of traffic between pods in AKS. Network policy should only be used for Linux-based nodes and pods in AKS.

## Before you begin

You need the Azure CLI version 2.0.61 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

### TIP

If you used the network policy feature during preview, we recommend that you [create a new cluster](#).

If you wish to continue using existing test clusters that used network policy during preview, upgrade your cluster to a new Kubernetes versions for the latest GA release and then deploy the following YAML manifest to fix the crashing metrics server and Kubernetes dashboard. This fix is only required for clusters that used the Calico network policy engine.

As a security best practice, [review the contents of this YAML manifest](#) to understand what is deployed into the AKS cluster.

```
kubectl delete -f https://raw.githubusercontent.com/Azure/aks-engine/master/docs/topics/calico-3.3.1-cleanup-after-upgrade.yaml
```

## Overview of network policy

All pods in an AKS cluster can send and receive traffic without limitations, by default. To improve security, you can define rules that control the flow of traffic. Back-end applications are often only exposed to required front-end services, for example. Or, database components are only accessible to the application tiers that connect to them.

Network Policy is a Kubernetes specification that defines access policies for communication between Pods. Using Network Policies, you define an ordered set of rules to send and receive traffic and apply them to a collection of pods that match one or more label selectors.

These network policy rules are defined as YAML manifests. Network policies can be included as part of a wider manifest that also creates a deployment or service.

### Network policy options in AKS

Azure provides two ways to implement network policy. You choose a network policy option when you create an AKS cluster. The policy option can't be changed after the cluster is created:

- Azure's own implementation, called *Azure Network Policies*.
- *Calico Network Policies*, an open-source network and network security solution founded by [Tigera](#).

Both implementations use Linux *IPTables* to enforce the specified policies. Policies are translated into sets of allowed and disallowed IP pairs. These pairs are then programmed as IPTable filter rules.

### Differences between Azure and Calico policies and their capabilities

CAPABILITY	AZURE	CALICO
Supported platforms	Linux	Linux
Supported networking options	Azure CNI	Azure CNI and kubenet
Compliance with Kubernetes specification	All policy types supported	All policy types supported
Additional features	None	Extended policy model consisting of Global Network Policy, Global Network Set, and Host Endpoint. For more information on using the <code>calicctl</code> CLI to manage these extended features, see <a href="#">calicctl user reference</a> .
Support	Supported by Azure support and Engineering team	Calico community support. For more information on additional paid support, see <a href="#">Project Calico support options</a> .
Logging	Rules added / deleted in IPTables are logged on every host under <code>/var/log/azure-npm.log</code>	For more information, see <a href="#">Calico component logs</a>

## Create an AKS cluster and enable network policy

To see network policies in action, let's create and then expand on a policy that defines traffic flow:

- Deny all traffic to pod.
- Allow traffic based on pod labels.
- Allow traffic based on namespace.

First, let's create an AKS cluster that supports network policy.

#### IMPORTANT

The network policy feature can only be enabled when the cluster is created. You can't enable network policy on an existing AKS cluster.

To use Azure Network Policy, you must use the [Azure CNI plug-in](#) and define your own virtual network and subnets. For more detailed information on how to plan out the required subnet ranges, see [configure advanced networking](#). Calico Network Policy could be used with either this same Azure CNI plug-in or with the Kubenet CNI plug-in.

The following example script:

- Creates a virtual network and subnet.
- Creates an Azure Active Directory (Azure AD) service principal for use with the AKS cluster.
- Assigns *Contributor* permissions for the AKS cluster service principal on the virtual network.
- Creates an AKS cluster in the defined virtual network and enables network policy.
  - The *azure* network policy option is used. To use Calico as the network policy option instead, use the

```
--network-policy calico parameter. Note: Calico could be used with either --network-plugin azure or  
--network-plugin kubenet.
```

Provide your own secure *SP\_PASSWORD*. You can replace the *RESOURCE\_GROUP\_NAME* and *CLUSTER\_NAME* variables:

```
RESOURCE_GROUP_NAME=myResourceGroup-NP
CLUSTER_NAME=myAKSCluster
LOCATION=canadaeast

# Create a resource group
az group create --name $RESOURCE_GROUP_NAME --location $LOCATION

# Create a virtual network and subnet
az network vnet create \
    --resource-group $RESOURCE_GROUP_NAME \
    --name myVnet \
    --address-prefixes 10.0.0.0/8 \
    --subnet-name myAKSSubnet \
    --subnet-prefix 10.240.0.0/16

# Create a service principal and read in the application ID
SP=$(az ad sp create-for-rbac --output json)
SP_ID=$(echo $SP | jq -r .appId)
SP_PASSWORD=$(echo $SP | jq -r .password)

# Wait 15 seconds to make sure that service principal has propagated
echo "Waiting for service principal to propagate..."
sleep 15

# Get the virtual network resource ID
VNET_ID=$(az network vnet show --resource-group $RESOURCE_GROUP_NAME --name myVnet --query id -o tsv)

# Assign the service principal Contributor permissions to the virtual network resource
az role assignment create --assignee $SP_ID --scope $VNET_ID --role Contributor

# Get the virtual network subnet resource ID
SUBNET_ID=$(az network vnet subnet show --resource-group $RESOURCE_GROUP_NAME --vnet-name myVnet --name myAKSSubnet --query id -o tsv)

# Create the AKS cluster and specify the virtual network and service principal information
# Enable network policy by using the `--network-policy` parameter
az aks create \
    --resource-group $RESOURCE_GROUP_NAME \
    --name $CLUSTER_NAME \
    --node-count 1 \
    --generate-ssh-keys \
    --network-plugin azure \
    --service-cidr 10.0.0.0/16 \
    --dns-service-ip 10.0.0.10 \
    --docker-bridge-address 172.17.0.1/16 \
    --vnet-subnet-id $SUBNET_ID \
    --service-principal $SP_ID \
    --client-secret $SP_PASSWORD \
    --network-policy azure
```

It takes a few minutes to create the cluster. When the cluster is ready, configure `kubectl` to connect to your Kubernetes cluster by using the [az aks get-credentials](#) command. This command downloads credentials and configures the Kubernetes CLI to use them:

```
az aks get-credentials --resource-group $RESOURCE_GROUP_NAME --name $CLUSTER_NAME
```

## Deny all inbound traffic to a pod

Before you define rules to allow specific network traffic, first create a network policy to deny all traffic. This policy gives you a starting point to begin to whitelist only the desired traffic. You can also clearly see that traffic is dropped when the network policy is applied.

For the sample application environment and traffic rules, let's first create a namespace called *development* to run the example pods:

```
kubectl create namespace development  
kubectl label namespace/development purpose=development
```

Create an example back-end pod that runs NGINX. This back-end pod can be used to simulate a sample back-end web-based application. Create this pod in the *development* namespace, and open port *80* to serve web traffic. Label the pod with *app=webapp,role=backend* so that we can target it with a network policy in the next section:

```
kubectl run backend --image=nginx --labels app=webapp,role=backend --namespace development --expose --port 80  
--generator=run-pod/v1
```

Create another pod and attach a terminal session to test that you can successfully reach the default NGINX webpage:

```
kubectl run --rm -it --image=alpine network-policy --namespace development --generator=run-pod/v1
```

At the shell prompt, use `wget` to confirm that you can access the default NGINX webpage:

```
wget -qO- http://backend
```

The following sample output shows that the default NGINX webpage returned:

```
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>  
[...]
```

Exit out of the attached terminal session. The test pod is automatically deleted.

```
exit
```

### Create and apply a network policy

Now that you've confirmed you can use the basic NGINX webpage on the sample back-end pod, create a network policy to deny all traffic. Create a file named `backend-policy.yaml` and paste the following YAML manifest. This manifest uses a *podSelector* to attach the policy to pods that have the *app:webapp,role:backend* label, like your sample NGINX pod. No rules are defined under *ingress*, so all inbound traffic to the pod is denied:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: backend-policy
  namespace: development
spec:
  podSelector:
    matchLabels:
      app: webapp
      role: backend
  ingress: []
```

Apply the network policy by using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f backend-policy.yaml
```

### Test the network policy

Let's see if you can use the NGINX webpage on the back-end pod again. Create another test pod and attach a terminal session:

```
kubectl run --rm -it --image=alpine network-policy development --generator=run-pod/v1
```

At the shell prompt, use `wget` to see if you can access the default NGINX webpage. This time, set a timeout value to 2 seconds. The network policy now blocks all inbound traffic, so the page can't be loaded, as shown in the following example:

```
$ wget -qO- --timeout=2 http://backend
wget: download timed out
```

Exit out of the attached terminal session. The test pod is automatically deleted.

```
exit
```

## Allow inbound traffic based on a pod label

In the previous section, a back-end NGINX pod was scheduled, and a network policy was created to deny all traffic. Let's create a front-end pod and update the network policy to allow traffic from front-end pods.

Update the network policy to allow traffic from pods with the labels `app:webapp,role:frontend` and in any namespace. Edit the previous `backend-policy.yaml` file, and add `matchLabels` ingress rules so that your manifest looks like the following example:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: backend-policy
  namespace: development
spec:
  podSelector:
    matchLabels:
      app: webapp
      role: backend
  ingress:
  - from:
    - namespaceSelector: {}
      podSelector:
        matchLabels:
          app: webapp
          role: frontend
```

#### NOTE

This network policy uses a `namespaceSelector` and a `podSelector` element for the ingress rule. The YAML syntax is important for the ingress rules to be additive. In this example, both elements must match for the ingress rule to be applied. Kubernetes versions prior to 1.12 might not interpret these elements correctly and restrict the network traffic as you expect. For more about this behavior, see [Behavior of to and from selectors](#).

Apply the updated network policy by using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f backend-policy.yaml
```

Schedule a pod that is labeled as `app=webapp,role=frontend` and attach a terminal session:

```
kubectl run --rm -it frontend --image=alpine --labels app=webapp,role=frontend --namespace development --generator=run-pod/v1
```

At the shell prompt, use `wget` to see if you can access the default NGINX webpage:

```
wget -qO- http://backend
```

Because the ingress rule allows traffic with pods that have the labels `app: webapp,role: frontend`, the traffic from the front-end pod is allowed. The following example output shows the default NGINX webpage returned:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
[...]
```

Exit out of the attached terminal session. The pod is automatically deleted.

```
exit
```

### Test a pod without a matching label

The network policy allows traffic from pods labeled `app: webapp,role: frontend`, but should deny all other traffic. Let's test to see whether another pod without those labels can access the back-end NGINX pod. Create another test pod and attach a terminal session:

```
kubectl run --rm -it --image=alpine network-policy --namespace development --generator=run-pod/v1
```

At the shell prompt, use `wget` to see if you can access the default NGINX webpage. The network policy blocks the inbound traffic, so the page can't be loaded, as shown in the following example:

```
$ wget -qO- --timeout=2 http://backend
wget: download timed out
```

Exit out of the attached terminal session. The test pod is automatically deleted.

```
exit
```

## Allow traffic only from within a defined namespace

In the previous examples, you created a network policy that denied all traffic, and then updated the policy to allow traffic from pods with a specific label. Another common need is to limit traffic to only within a given namespace. If the previous examples were for traffic in a `development` namespace, create a network policy that prevents traffic from another namespace, such as `production`, from reaching the pods.

First, create a new namespace to simulate a production namespace:

```
kubectl create namespace production
kubectl label namespace/production purpose=production
```

Schedule a test pod in the `production` namespace that is labeled as `app=webapp,role=frontend`. Attach a terminal session:

```
kubectl run --rm -it frontend --image=alpine --labels app=webapp,role=frontend --namespace production --
generator=run-pod/v1
```

At the shell prompt, use `wget` to confirm that you can access the default NGINX webpage:

```
wget -qO- http://backend.production
```

Because the labels for the pod match what is currently permitted in the network policy, the traffic is allowed. The network policy doesn't look at the namespaces, only the pod labels. The following example output shows the default NGINX webpage returned:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
[...]
```

Exit out of the attached terminal session. The test pod is automatically deleted.

```
exit
```

## Update the network policy

Let's update the ingress rule *namespaceSelector* section to only allow traffic from within the *development* namespace. Edit the *backend-policy.yaml* manifest file as shown in the following example:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: backend-policy
  namespace: development
spec:
  podSelector:
    matchLabels:
      app: webapp
      role: backend
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          purpose: development
      podSelector:
        matchLabels:
          app: webapp
          role: frontend
```

In more complex examples, you could define multiple ingress rules, like a *namespaceSelector* and then a *podSelector*.

Apply the updated network policy by using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f backend-policy.yaml
```

## Test the updated network policy

Schedule another pod in the *production* namespace and attach a terminal session:

```
kubectl run --rm -it frontend --image=alpine --labels app=webapp,role=frontend --namespace production --
generator=run-pod/v1
```

At the shell prompt, use `wget` to see that the network policy now denies traffic:

```
$ wget -qO- --timeout=2 http://backend.development
wget: download timed out
```

Exit out of the test pod:

```
exit
```

With traffic denied from the *production* namespace, schedule a test pod back in the *development* namespace and attach a terminal session:

```
kubectl run --rm -it frontend --image=alpine --labels app=webapp,role=frontend --namespace development --generator=run-pod/v1
```

At the shell prompt, use `wget` to see that the network policy allows the traffic:

```
wget -qO- http://backend
```

Traffic is allowed because the pod is scheduled in the namespace that matches what's permitted in the network policy. The following sample output shows the default NGINX webpage returned:

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
[...]
```

Exit out of the attached terminal session. The test pod is automatically deleted.

```
exit
```

## Clean up resources

In this article, we created two namespaces and applied a network policy. To clean up these resources, use the `kubectl delete` command and specify the resource names:

```
kubectl delete namespace production
kubectl delete namespace development
```

## Next steps

For more about network resources, see [Network concepts for applications in Azure Kubernetes Service \(AKS\)](#).

To learn more about policies, see [Kubernetes network policies](#).

# Preview - Secure your cluster using pod security policies in Azure Kubernetes Service (AKS)

2/28/2020 • 15 minutes to read • [Edit Online](#)

To improve the security of your AKS cluster, you can limit what pods can be scheduled. Pods that request resources you don't allow can't run in the AKS cluster. You define this access using pod security policies. This article shows you how to use pod security policies to limit the deployment of pods in AKS.

## IMPORTANT

AKS preview features are self-service opt-in. Previews are provided "as-is" and "as available" and are excluded from the service level agreements and limited warranty. AKS Previews are partially covered by customer support on best effort basis. As such, these features are not meant for production use. For additional information, please see the following support articles:

- [AKS Support Policies](#)
- [Azure Support FAQ](#)

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the [AKS quickstart using the Azure CLI](#) or [using the Azure portal](#).

You need the Azure CLI version 2.0.61 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

### Install `aks-preview` CLI extension

To use pod security policies, you need the `aks-preview` CLI extension version 0.4.1 or higher. Install the `aks-preview` Azure CLI extension using the [az extension add](#) command, then check for any available updates using the [az extension update](#) command:

```
# Install the aks-preview extension
az extension add --name aks-preview

# Update the extension to make sure you have the latest version installed
az extension update --name aks-preview
```

### Register pod security policy feature provider

To create or update an AKS cluster to use pod security policies, first enable a feature flag on your subscription. To register the `PodSecurityPolicyPreview` feature flag, use the [az feature register](#) command as shown in the following example:

#### Caution

When you register a feature on a subscription, you can't currently un-register that feature. After you enable some preview features, defaults may be used for all AKS clusters then created in the subscription. Don't enable preview features on production subscriptions. Use a separate subscription to test preview features and gather feedback.

```
az feature register --name PodSecurityPolicyPreview --namespace Microsoft.ContainerService
```

It takes a few minutes for the status to show *Registered*. You can check on the registration status using the [az](#)

[feature list](#) command:

```
az feature list -o table --query "[?contains(name, 'Microsoft.ContainerService/PodSecurityPolicyPreview')].{Name:name,State:properties.state}"
```

When ready, refresh the registration of the *Microsoft.ContainerService* resource provider using the [az provider register](#) command:

```
az provider register --namespace Microsoft.ContainerService
```

## Overview of pod security policies

In a Kubernetes cluster, an admission controller is used to intercept requests to the API server when a resource is to be created. The admission controller can then *validate* the resource request against a set of rules, or *mutate* the resource to change deployment parameters.

*PodSecurityPolicy* is an admission controller that validates a pod specification meets your defined requirements. These requirements may limit the use of privileged containers, access to certain types of storage, or the user or group the container can run as. When you try to deploy a resource where the pod specifications don't meet the requirements outlined in the pod security policy, the request is denied. This ability to control what pods can be scheduled in the AKS cluster prevents some possible security vulnerabilities or privilege escalations.

When you enable pod security policy in an AKS cluster, some default policies are applied. These default policies provide an out-of-the-box experience to define what pods can be scheduled. However, cluster users may run into problems deploying pods until you define your own policies. The recommended approach is to:

- Create an AKS cluster
- Define your own pod security policies
- Enable the pod security policy feature

To show how the default policies limit pod deployments, in this article we first enable the pod security policies feature, then create a custom policy.

## Enable pod security policy on an AKS cluster

You can enable or disable pod security policy using the [az aks update](#) command. The following example enables pod security policy on the cluster name *myAKSCluster* in the resource group named *myResourceGroup*.

### NOTE

For real-world use, don't enable the pod security policy until you have defined your own custom policies. In this article, you enable pod security policy as the first step to see how the default policies limit pod deployments.

```
az aks update \
--resource-group myResourceGroup \
--name myAKSCluster \
--enable-pod-security-policy
```

## Default AKS policies

When you enable pod security policy, AKS creates one default policy named *privileged*. Don't edit or remove the default policy. Instead, create your own policies that define the settings you want to control. Let's first look at what

these default policies are how they impact pod deployments.

To view the policies available, use the [kubectl get psp](#) command, as shown in the following example

```
$ kubectl get psp
```

NAME	PRIV	CAPS	SELINUX	RUNASUSER	FSGROUP	SUPGROUP	READONLYROOTFS	VOLUMES
privileged	true	*	RunAsAny	RunAsAny	RunAsAny	RunAsAny	false	*
			configMap,emptyDir,projected,secret,downwardAPI,persistentVolumeClaim					

The *privileged* pod security policy is applied to any authenticated user in the AKS cluster. This assignment is controlled by ClusterRoles and ClusterRoleBindings. Use the [kubectl get clusterrolebindings](#) command and search for the *default:privileged*: binding:

```
kubectl get clusterrolebindings default:privileged -o yaml
```

As shown in the following condensed output, the *psp:restricted* ClusterRole is assigned to any *system:authenticated* users. This ability provides a basic level of restrictions without your own policies being defined.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  [...]
  name: default:privileged
  [...]
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp:privileged
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:authenticated
```

It's important to understand how these default policies interact with user requests to schedule pods before you start to create your own pod security policies. In the next few sections, let's schedule some pods to see these default policies in action.

## Create a test user in an AKS cluster

By default, when you use the [az aks get-credentials](#) command, the *admin* credentials for the AKS cluster are added to your `kubectl` config. The admin user bypasses the enforcement of pod security policies. If you use Azure Active Directory integration for your AKS clusters, you could sign in with the credentials of a non-admin user to see the enforcement of policies in action. In this article, let's create a test user account in the AKS cluster that you can use.

Create a sample namespace named *psp-aks* for test resources using the [kubectl create namespace](#) command. Then, create a service account named *nonadmin-user* using the [kubectl create serviceaccount](#) command:

```
kubectl create namespace psp-aks
kubectl create serviceaccount --namespace psp-aks nonadmin-user
```

Next, create a RoleBinding for the *nonadmin-user* to perform basic actions in the namespace using the [kubectl create rolebinding](#) command:

```
kubectl create rolebinding \
--namespace psp-aks \
psp-aks-editor \
--clusterrole=edit \
--serviceaccount=psp-aks:nonadmin-user
```

## Create alias commands for admin and non-admin user

To highlight the difference between the regular admin user when using `kubectl` and the non-admin user created in the previous steps, create two command-line aliases:

- The **kubectl-admin** alias is for the regular admin user, and is scoped to the *psp-aks* namespace.
- The **kubectl-nonadminuser** alias is for the *nonadmin-user* created in the previous step, and is scoped to the *psp-aks* namespace.

Create these two aliases as shown in the following commands:

```
alias kubectl-admin='kubectl --namespace psp-aks'
alias kubectl-nonadminuser='kubectl --as=system:serviceaccount:psp-aks:nonadmin-user --namespace psp-aks'
```

## Test the creation of a privileged pod

Let's first test what happens when you schedule a pod with the security context of `privileged: true`. This security context escalates the pod's privileges. In the previous section that showed the default AKS pod security policies, the *restricted* policy should deny this request.

Create a file named `nginx-privileged.yaml` and paste the following YAML manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-privileged
spec:
  containers:
    - name: nginx-privileged
      image: nginx:1.14.2
      securityContext:
        privileged: true
```

Create the pod using the `kubectl apply` command and specify the name of your YAML manifest:

```
kubectl-nonadminuser apply -f nginx-privileged.yaml
```

The pod fails to be scheduled, as shown in the following example output:

```
$ kubectl-nonadminuser apply -f nginx-privileged.yaml
Error from server (Forbidden): error when creating "nginx-privileged.yaml": pods "nginx-privileged" is
forbidden: unable to validate against any pod security policy: [spec.containers[0].securityContext.privileged:
Invalid value: true: Privileged containers are not allowed]
```

The pod doesn't reach the scheduling stage, so there are no resources to delete before you move on.

## Test creation of an unprivileged pod

In the previous example, the pod specification requested privileged escalation. This request is denied by the default *restricted* pod security policy, so the pod fails to be scheduled. Let's try now running that same NGINX pod without the privilege escalation request.

Create a file named `nginx-unprivileged.yaml` and paste the following YAML manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-unprivileged
spec:
  containers:
    - name: nginx-unprivileged
      image: nginx:1.14.2
```

Create the pod using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl-nonadminuser apply -f nginx-unprivileged.yaml
```

The Kubernetes scheduler accepts the pod request. However, if you look at the status of the pod using

`kubectl get pods`, there's an error:

```
$ kubectl-nonadminuser get pods

NAME          READY   STATUS            RESTARTS   AGE
nginx-unprivileged   0/1     CreateContainerConfigError   0          26s
```

Use the [kubectl describe pod](#) command to look at the events for the pod. The following condensed example shows the container and image require root permissions, even though we didn't request them:

```
$ kubectl-nonadminuser describe pod nginx-unprivileged

Name:           nginx-unprivileged
Namespace:      psp-aks
Priority:       0
PriorityClassName:  <none>
Node:           aks-agentpool-34777077-0/10.240.0.4
Start Time:     Thu, 28 Mar 2019 22:05:04 +0000
[...]
Events:
  Type  Reason  Age           From           Message
  ----  -----  --           ----           -----
  Normal  Scheduled  7m14s        default-scheduler  Successfully assigned psp-
  aks/nginx-unprivileged to aks-agentpool-34777077-0
  Warning Failed   5m2s (x12 over 7m13s)  kubelet, aks-agentpool-34777077-0  Error: container has
  runAsNonRoot and image will run as root
  Normal  Pulled    2m10s (x25 over 7m13s)  kubelet, aks-agentpool-34777077-0  Container image "nginx:1.14.2"
  already present on machine
```

Even though we didn't request any privileged access, the container image for NGINX needs to create a binding for port *80*. To bind ports *1024* and below, the *root* user is required. When the pod tries to start, the *restricted* pod security policy denies this request.

This example shows that the default pod security policies created by AKS are in effect and restrict the actions a user can perform. It's important to understand the behavior of these default policies, as you may not expect a basic NGINX pod to be denied.

Before you move on to the next step, delete this test pod using the [kubectl delete pod](#) command:

```
kubectl-nonadminuser delete -f nginx-unprivileged.yaml
```

## Test creation of a pod with a specific user context

In the previous example, the container image automatically tried to use root to bind NGINX to port 80. This request was denied by the default *restricted* pod security policy, so the pod fails to start. Let's try now running that same NGINX pod with a specific user context, such as `runAsUser: 2000`.

Create a file named `nginx-unprivileged-nonroot.yaml` and paste the following YAML manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-unprivileged-nonroot
spec:
  containers:
    - name: nginx-unprivileged
      image: nginx:1.14.2
      securityContext:
        runAsUser: 2000
```

Create the pod using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl-nonadminuser apply -f nginx-unprivileged-nonroot.yaml
```

The Kubernetes scheduler accepts the pod request. However, if you look at the status of the pod using `kubectl get pods`, there's a different error than the previous example:

```
$ kubectl-nonadminuser get pods
NAME                  READY   STATUS            RESTARTS   AGE
nginx-unprivileged-nonroot  0/1     CrashLoopBackOff   1          3s
```

Use the [kubectl describe pod](#) command to look at the events for the pod. The following condensed example shows the pod events:

```
$ kubectl-nonadminuser describe pods nginx-unprivileged
Name:           nginx-unprivileged
Namespace:      psp-aks
Priority:       0
PriorityClassName: <none>
Node:           aks-agentpool-34777077-0/10.240.0.4
Start Time:     Thu, 28 Mar 2019 22:05:04 +0000
[...]
Events:
  Type  Reason  Age           From           Message
  ----  -----  --           --           --
  Normal Scheduled  2m14s       default-scheduler  Successfully assigned psp-
  aks/nginx-unprivileged-nonroot to aks-agentpool-34777077-0
  Normal Pulled    118s (x3 over 2m13s)  kubelet, aks-agentpool-34777077-0  Container image "nginx:1.14.2"
already present on machine
  Normal Created    118s (x3 over 2m13s)  kubelet, aks-agentpool-34777077-0  Created container
  Normal Started    118s (x3 over 2m12s)  kubelet, aks-agentpool-34777077-0  Started container
  Warning BackOff   105s (x5 over 2m11s)  kubelet, aks-agentpool-34777077-0  Back-off restarting failed
  container
```

The events indicate that the container was created and started. There's nothing immediately obvious as to why the pod is in a failed state. Let's look at the pod logs using the [kubectl logs](#) command:

```
kubectl-nonadminuser logs nginx-unprivileged-nonroot --previous
```

The following example log output gives an indication that within the NGINX configuration itself, there's a permissions error when the service tries to start. This error is again caused by needing to bind to port 80. Although the pod specification defined a regular user account, this user account isn't sufficient in the OS-level for the NGINX service to start and bind to the restricted port.

```
$ kubectl-nonadminuser logs nginx-unprivileged-nonroot --previous

2019/03/28 22:38:29 [warn] 1#1: the "user" directive makes sense only if the master process runs with super-user privileges, ignored in /etc/nginx/nginx.conf:2
nginx: [warn] the "user" directive makes sense only if the master process runs with super-user privileges, ignored in /etc/nginx/nginx.conf:2
2019/03/28 22:38:29 [emerg] 1#1: mkdir() "/var/cache/nginx/client_temp" failed (13: Permission denied)
nginx: [emerg] mkdir() "/var/cache/nginx/client_temp" failed (13: Permission denied)
```

Again, it's important to understand the behavior of the default pod security policies. This error was a little harder to track down, and again, you may not expect a basic NGINX pod to be denied.

Before you move on to the next step, delete this test pod using the [kubectl delete pod](#) command:

```
kubectl-nonadminuser delete -f nginx-unprivileged-nonroot.yaml
```

## Create a custom pod security policy

Now that you've seen the behavior of the default pod security policies, let's provide a way for the *nonadmin-user* to successfully schedule pods.

Let's create a policy to reject pods that request privileged access. Other options, such as *runAsUser* or allowed *volumes*, aren't explicitly restricted. This type of policy denies a request for privileged access, but otherwise lets the cluster run the requested pods.

Create a file named `psp-deny-privileged.yaml` and paste the following YAML manifest:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: psp-deny-privileged
spec:
  privileged: false
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
    - '*'
```

Create the policy using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f psp-deny-privileged.yaml
```

To view the policies available, use the [kubectl get psp](#) command, as shown in the following example. Compare the *psp-deny-privileged* policy with the default *restricted* policy that was enforced in the previous examples to create a pod. Only the use of *PRIV* escalation is denied by your policy. There are no restrictions on the user or group for the *psp-deny-privileged* policy.

```
$ kubectl get psp
```

NAME	PRIV	CAPS	SELINUX	RUNASUSER	FSGROUP	SUPGROUP	READONLYROOTFS
VOLUMES							
privileged	true	*	RunAsAny	RunAsAny	RunAsAny	RunAsAny	false *
psp-deny-privileged	false		RunAsAny	RunAsAny	RunAsAny	RunAsAny	false *
configMap,emptyDir,projected,secret,downwardAPI,persistentVolumeClaim							

## Allow user account to use the custom pod security policy

In the previous step, you created a pod security policy to reject pods that request privileged access. To allow the policy to be used, you create a *Role* or a *ClusterRole*. Then, you associate one of these roles using a *RoleBinding* or *ClusterRoleBinding*.

For this example, create a *ClusterRole* that allows you to *use* the *psp-deny-privileged* policy created in the previous step. Create a file named `psp-deny-privileged-clusterrole.yaml` and paste the following YAML manifest:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psp-deny-privileged-clusterrole
rules:
- apiGroups:
  - extensions
  resources:
  - podsecuritypolicies
  resourceNames:
  - psp-deny-privileged
  verbs:
  - use
```

Create the *ClusterRole* using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f psp-deny-privileged-clusterrole.yaml
```

Now create a *ClusterRoleBinding* to use the *ClusterRole* created in the previous step. Create a file named `psp-deny-privileged-clusterrolebinding.yaml` and paste the following YAML manifest:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: psp-deny-privileged-clusterrolebinding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: psp-deny-privileged-clusterrole
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:serviceaccounts
```

Create a ClusterRoleBinding using the [kubectl apply](#) command and specify the name of your YAML manifest:

```
kubectl apply -f psp-deny-privileged-clusterrolebinding.yaml
```

#### NOTE

In the first step of this article, the pod security policy feature was enabled on the AKS cluster. The recommended practice was to only enable the pod security policy feature after you've defined your own policies. This is the stage where you would enable the pod security policy feature. One or more custom policies have been defined, and user accounts have been associated with those policies. Now you can safely enable the pod security policy feature and minimize problems caused by the default policies.

## Test the creation of an unprivileged pod again

With your custom pod security policy applied and a binding for the user account to use the policy, let's try to create an unprivileged pod again. Use the same `nginx-privileged.yaml` manifest to create the pod using the [kubectl apply](#) command:

```
kubectl-nonadminuser apply -f nginx-unprivileged.yaml
```

The pod is successfully scheduled. When you check the status of the pod using the [kubectl get pods](#) command, the pod is *Running*:

```
$ kubectl-nonadminuser get pods

NAME           READY   STATUS    RESTARTS   AGE
nginx-unprivileged   1/1     Running   0          7m14s
```

This example shows how you can create custom pod security policies to define access to the AKS cluster for different users or groups. The default AKS policies provide tight controls on what pods can run, so create your own custom policies to then correctly define the restrictions you need.

Delete the NGINX unprivileged pod using the [kubectl delete](#) command and specify the name of your YAML manifest:

```
kubectl-nonadminuser delete -f nginx-unprivileged.yaml
```

## Clean up resources

To disable pod security policy, use the [az aks update](#) command again. The following example disables pod security policy on the cluster name *myAKSCluster* in the resource group named *myResourceGroup*:

```
az aks update \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --disable-pod-security-policy
```

Next, delete the ClusterRole and ClusterRoleBinding:

```
kubectl delete -f psp-deny-privileged-clusterrolebinding.yaml
kubectl delete -f psp-deny-privileged-clusterrole.yaml
```

Delete the security policy using [kubectl delete](#) command and specify the name of your YAML manifest:

```
kubectl delete -f psp-deny-privileged.yaml
```

Finally, delete the *psp-aks* namespace:

```
kubectl delete namespace psp-aks
```

## Next steps

This article showed you how to create a pod security policy to prevent the use of privileged access. There are lots of features that a policy can enforce, such as type of volume or the RunAs user. For more information on the available options, see the [Kubernetes pod security policy reference docs](#).

For more information about limiting pod network traffic, see [Secure traffic between pods using network policies in AKS](#).

# Secure access to the API server using authorized IP address ranges in Azure Kubernetes Service (AKS)

2/25/2020 • 5 minutes to read • [Edit Online](#)

In Kubernetes, the API server receives requests to perform actions in the cluster such as to create resources or scale the number of nodes. The API server is the central way to interact with and manage a cluster. To improve cluster security and minimize attacks, the API server should only be accessible from a limited set of IP address ranges.

This article shows you how to use API server authorized IP address ranges to limit which IP addresses and CIDRs can access control plane.

## IMPORTANT

On new clusters, API server authorized IP address ranges are only supported on the *Standard* SKU load balancer. Existing clusters with the *Basic* SKU load balancer and API server authorized IP address ranges configured will continue work as is but cannot be migrated to a *Standard* SKU load balancer. Those existing clusters will also continue to work if their Kubernetes version or control plane are upgraded.

## Before you begin

API server authorized IP ranges only work for new AKS clusters that you create. This article shows you how to create an AKS cluster using the Azure CLI.

You need the Azure CLI version 2.0.76 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Overview of API server authorized IP ranges

The Kubernetes API server is how the underlying Kubernetes APIs are exposed. This component provides the interaction for management tools, such as `kubectl` or the Kubernetes dashboard. AKS provides a single-tenant cluster master, with a dedicated API server. By default, the API server is assigned a public IP address, and you should control access using role-based access controls (RBAC).

To secure access to the otherwise publicly accessible AKS control plane / API server, you can enable and use authorized IP ranges. These authorized IP ranges only allow defined IP address ranges to communicate with the API server. A request made to the API server from an IP address that is not part of these authorized IP ranges is blocked. Continue to use RBAC to authorize users and the actions they request.

For more information about the API server and other cluster components, see [Kubernetes core concepts for AKS](#).

## Create an AKS cluster with API server authorized IP ranges enabled

API server authorized IP ranges only work for new AKS clusters. Create a cluster using the `az aks create` and specify the `--api-server-authorized-ip-ranges` parameter to provide a list of authorized IP address ranges. These IP address ranges are usually address ranges used by your on-premises networks or public IPs. When you specify a CIDR range, start with the first IP address in the range. For example, `137.117.106.90/29` is a valid range, but make sure you specify the first IP address in the range, such as `137.117.106.88/29`.

## IMPORTANT

By default, your cluster uses the [Standard SKU load balancer](#) which you can use to configure the outbound gateway. When you enable API server authorized IP ranges during cluster creation, the public IP for your cluster is also allowed by default in addition to the ranges you specify. If you specify "" or no value for `--api-server-authorized-ip-ranges`, API server authorized IP ranges will be disabled.

The following example creates a single-node cluster named `myAKSCluster` in the resource group named `myResourceGroup` with API server authorized IP ranges enabled. The IP address ranges allowed are `73.140.245.0/24`:

```
az aks create \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --node-count 1 \
    --vm-set-type VirtualMachineScaleSets \
    --load-balancer-sku standard \
    --api-server-authorized-ip-ranges 73.140.245.0/24 \
    --generate-ssh-keys
```

## NOTE

You should add these ranges to an allow list:

- The firewall public IP address
- Any range that represents networks that you'll administer the cluster from
- If you are using Azure Dev Spaces on your AKS cluster, you have to allow [additional ranges based on your region](#).

## Specify the outbound IPs for the Standard SKU load balancer

When creating an AKS cluster, if you specify the outbound IP addresses or prefixes for the cluster, those addresses or prefixes are allowed as well. For example:

```
az aks create \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --node-count 1 \
    --vm-set-type VirtualMachineScaleSets \
    --load-balancer-sku standard \
    --api-server-authorized-ip-ranges 73.140.245.0/24 \
    --load-balancer-outbound-ips <publicIpId1>,<publicIpId2> \
    --generate-ssh-keys
```

In the above example, all IPs provided in the parameter `--load-balancer-outbound-ip-prefixes` are allowed along with the IPs in the `--api-server-authorized-ip-ranges` parameter.

Alternatively, you can specify the `--load-balancer-outbound-ip-prefixes` parameter to allow outbound load balancer IP prefixes.

## Allow only the outbound public IP of the Standard SKU load balancer

When you enable API server authorized IP ranges during cluster creation, the outbound public IP for the Standard SKU load balancer for your cluster is also allowed by default in addition to the ranges you specify. To allow only the outbound public IP of the Standard SKU load balancer, use `0.0.0.0/32` when specifying the `--api-server-authorized-ip-ranges` parameter.

In the following example, only the outbound public IP of the Standard SKU load balancer is allowed, and you can

only access the API server from the nodes within the cluster.

```
az aks create \
--resource-group myResourceGroup \
--name myAKScluster \
--node-count 1 \
--vm-set-type VirtualMachineScaleSets \
--load-balancer-sku standard \
--api-server-authorized-ip-ranges 0.0.0.0/32 \
--generate-ssh-keys
```

## Update a cluster's API server authorized IP ranges

To update the API server authorized IP ranges on an existing cluster, use [az aks update](#) command and use the `--api-server-authorized-ip-ranges`, `--load-balancer-outbound-ip-prefixes`, `--load-balancer-outbound-ips`, or `--load-balancer-outbound-ip-prefixes` parameters.

The following example updates API server authorized IP ranges on the cluster named *myAKScluster* in the resource group named *myResourceGroup*. The IP address range to authorize is `73.140.245.0/24`:

```
az aks update \
--resource-group myResourceGroup \
--name myAKScluster \
--api-server-authorized-ip-ranges 73.140.245.0/24
```

You can also use `0.0.0.0/32` when specifying the `--api-server-authorized-ip-ranges` parameter to allow only the public IP of the Standard SKU load balancer.

## Disable authorized IP ranges

To disable authorized IP ranges, use [az aks update](#) and specify an empty range to disable API server authorized IP ranges. For example:

```
az aks update \
--resource-group myResourceGroup \
--name myAKScluster \
--api-server-authorized-ip-ranges ""
```

## Next steps

In this article, you enabled API server authorized IP ranges. This approach is one part of how you can run a secure AKS cluster.

For more information, see [Security concepts for applications and clusters in AKS](#) and [Best practices for cluster security and upgrades in AKS](#).

# Understand Azure Policy for Azure Kubernetes Service

12/23/2019 • 7 minutes to read • [Edit Online](#)

Azure Policy integrates with the [Azure Kubernetes Service \(AKS\)](#) to apply at-scale enforcements and safeguards on your clusters in a centralized, consistent manner. By extending use of [Gatekeeper](#) v2, an *admission controller webhook* for [Open Policy Agent](#) (OPA), Azure Policy makes it possible to manage and report on the compliance state of your Azure resources and AKS clusters from one place.

## NOTE

Azure Policy for AKS is in Limited Preview and only supports built-in policy definitions.

## Overview

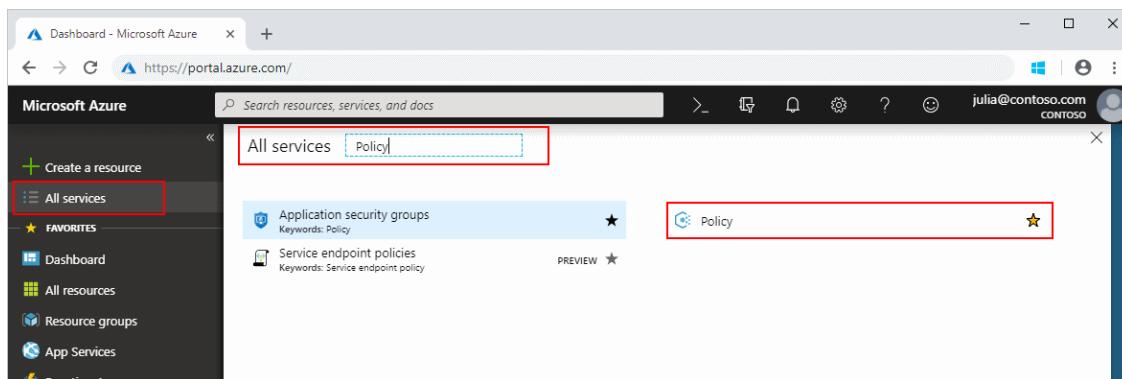
To enable and use Azure Policy for AKS with your AKS cluster, take the following actions:

- [Opt-in for preview features](#)
- [Install the Azure Policy Add-on](#)
- [Assign a policy definition for AKS](#)
- [Wait for validation](#)

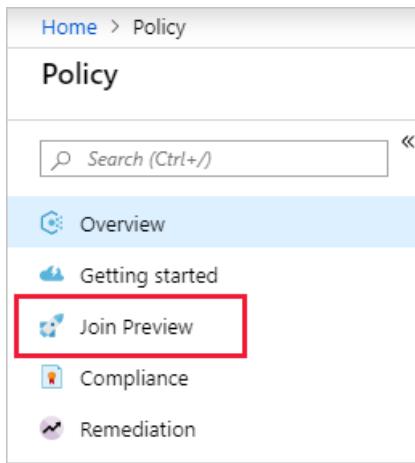
## Opt-in for preview

Before installing the Azure Policy Add-on or enabling any of the service features, your subscription must enable the **Microsoft.ContainerService** resource provider and the **Microsoft.PolicyInsights** resource provider, then be approved to join the preview. To join the preview, follow these steps in either the Azure portal or with Azure CLI:

- Azure portal:
  1. Register the **Microsoft.ContainerService** and **Microsoft.PolicyInsights** resource providers. For steps, see [Resource providers and types](#).
  2. Launch the Azure Policy service in the Azure portal by clicking **All services**, then searching for and selecting **Policy**.



3. Select **Join Preview** on the left side of the Azure Policy page.



4. Select the row of the subscription you want added to the preview.

5. Select the **Opt-in** button at the top of the list of subscriptions.

- Azure CLI:

```
# Log in first with az login if you're not using Cloud Shell

# Provider register: Register the Azure Kubernetes Services provider
az provider register --namespace Microsoft.ContainerService

# Provider register: Register the Azure Policy provider
az provider register --namespace Microsoft.PolicyInsights

# Feature register: enables installing the add-on
az feature register --namespace Microsoft.ContainerService --name AKS-AzurePolicyAutoApprove

# Use the following to confirm the feature has registered
az feature list -o table --query "[?contains(name, 'Microsoft.ContainerService/AKS-AzurePolicyAutoApprove')].[Name:name,State:properties.state]"

# Once the above shows 'Registered' run the following to propagate the update
az provider register -n Microsoft.ContainerService

# Feature register: enables the add-on to call the Azure Policy resource provider
az feature register --namespace Microsoft.PolicyInsights --name AKS-DataplaneAutoApprove

# Use the following to confirm the feature has registered
az feature list -o table --query "[?contains(name, 'Microsoft.PolicyInsights/AKS-DataplaneAutoApprove')].[Name:name,State:properties.state]"

# Once the above shows 'Registered' run the following to propagate the update
az provider register -n Microsoft.PolicyInsights
```

## Azure Policy Add-on

The *Azure Policy Add-on* for Kubernetes connects the Azure Policy service to the Gatekeeper admission controller. The add-on, which is installed into the *azure-policy* namespace, enacts the following functions:

- Checks with Azure Policy for assignments to the AKS cluster
- Downloads and caches policy details, including the *rego* policy definition, as **configmaps**
- Runs a full scan compliance check on the AKS cluster
- Reports auditing and compliance details back to Azure Policy

### Installing the add-on

#### Prerequisites

Before you install the add-on in your AKS cluster, the preview extension must be installed. This step is done with Azure CLI:

1. You need the Azure CLI version 2.0.62 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install the Azure CLI](#).
2. The AKS cluster must be version 1.10 or higher. Use the following script to validate your AKS cluster version:

```
# Log in first with az login if you're not using Cloud Shell  
  
# Look for the value in kubernetesVersion  
az aks list
```

3. Install version 0.4.0 of the Azure CLI preview extension for AKS, `aks-preview`:

```
# Log in first with az login if you're not using Cloud Shell  
  
# Install/update the preview extension  
az extension add --name aks-preview  
  
# Validate the version of the preview extension  
az extension show --name aks-preview --query [version]
```

#### NOTE

If you've previously installed the `aks-preview` extension, install any updates using the

```
az extension update --name aks-preview
```

#### Installation steps

Once the prerequisites are completed, install the Azure Policy Add-on in the AKS cluster you want to manage.

- Azure portal
1. Launch the AKS service in the Azure portal by clicking **All services**, then searching for and selecting **Kubernetes services**.
  2. Select one of your AKS clusters.
  3. Select **Policies (preview)** on the left side of the Kubernetes service page.

The screenshot shows the Azure portal interface for managing an AKS cluster named 'MyAKSCluster'. The left sidebar contains a search bar and a navigation menu with the following items:

- Overview
- Activity log
- Access control (IAM)
- Tags
- Settings
  - Node pools (preview)
  - Upgrade
  - Scale
  - Dev Spaces
  - Deployment center (preview)
  - Policies (preview) **(highlighted with a red box)**
  - Properties

4. In the main page, select the **Enable add-on** button.

### Onboard to Azure Policy for Azure Kubernetes Service (AKS)

With this integration, you can apply at-scale enforcements and safeguards for AKS clusters in a centralized, consistent manner through Azure Policy. This capability is currently under Private Preview. To register for Preview, [please sign up here](#). Unless you have been pre-approved, the 'enable add-on' button is disabled. [Learn more about Azure Policy](#)

**Enable add-on**

#### NOTE

If the **Enable add-on** button is grayed out, the subscription has not yet been added to the preview. See [Opt-in for preview](#) for the required steps.

- Azure CLI

```
# Log in first with az login if you're not using Cloud Shell  
az aks enable-addons --addons azure-policy --name MyAKSCluster --resource-group MyResourceGroup
```

#### Validation and reporting frequency

The add-on checks in with Azure Policy for changes in policy assignments every 5 minutes. During this refresh cycle, the add-on removes all *configmaps* in the *azure-policy* namespace then recreates the *configmaps* for Gatekeeper use.

#### NOTE

While a *cluster admin* may have permission to the *azure-policy* namespace, it's not recommended or supported to make changes to the namespace. Any manual changes made are lost during the refresh cycle.

Every 5 minutes, the add-on calls for a full scan of the cluster. After gathering details of the full scan and any real-

time evaluations by Gatekeeper of attempted changes to the cluster, the add-on reports the results back to Azure Policy for inclusion in [compliance details](#) like any Azure Policy assignment. Only results for active policy assignments are returned during the audit cycle.

## Policy language

The Azure Policy language structure for managing AKS follows that of existing policies. The effect [EnforceRegoPolicy](#) is used to manage your AKS clusters and takes *details* properties specific to working with OPA and Gatekeeper v2. For details and examples, see the [EnforceRegoPolicy](#) effect.

As part of the *details.policy* property in the policy definition, Azure Policy passes the URI of a rego policy to the add-on. Rego is the language that OPA and Gatekeeper support to validate or mutate a request to the Kubernetes cluster. By supporting an existing standard for Kubernetes management, Azure Policy makes it possible to reuse existing rules and pair them with Azure Policy for a unified cloud compliance reporting experience. For more information, see [What is Rego?](#).

## Built-in policies

To find the built-in policies for managing AKS using the Azure portal, follow these steps:

1. Start the Azure Policy service in the Azure portal. Select **All services** in the left pane and then search for and select **Policy**.
2. In the left pane of the Azure Policy page, select **Definitions**.
3. From the Category drop-down list box, use **Select all** to clear the filter and then select **Kubernetes service**.
4. Select the policy definition, then select the **Assign** button.

### NOTE

When assigning the Azure Policy for AKS definition, the **Scope** must include the AKS cluster resource.

Alternately, use the [Assign a policy - Portal](#) quickstart to find and assign an AKS policy. Search for a Kubernetes policy definition instead of the sample 'audit vms'.

### IMPORTANT

Built-in policies in category **Kubernetes service** are only for use with AKS.

## Logging

### Azure Policy Add-on logs

As a Kubernetes controller/container, the Azure Policy Add-on keeps logs in the AKS cluster. The logs are exposed in the **Insights** page of the AKS cluster. For more information, see [Understand AKS cluster performance with Azure Monitor for containers](#).

### Gatekeeper logs

To enable Gatekeeper logs for new resource requests, follow the steps in [Enable and review Kubernetes master node logs in AKS](#). Here is an example query to view denied events on new resource requests:

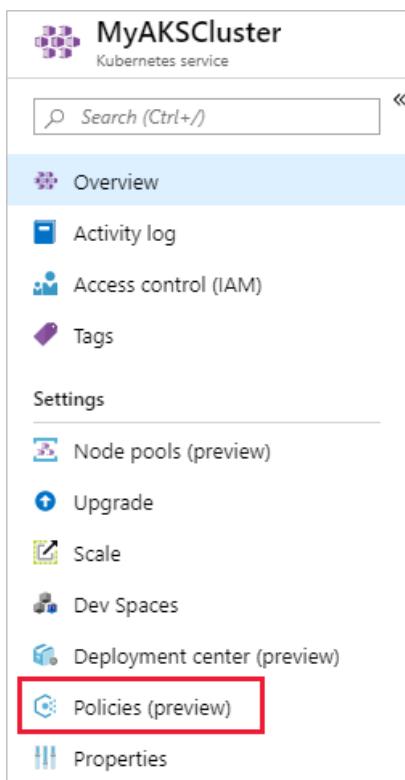
```
| where Category == "kube-audit"  
| where log_s contains "admission webhook"  
| limit 100
```

To view logs from Gatekeeper containers, follow the steps in [Enable and review Kubernetes master node logs in AKS](#) and check the *kube-apiserver* option in the **Diagnostic settings** pane.

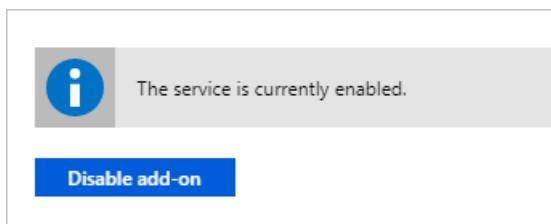
## Remove the add-on

To remove the Azure Policy Add-on from your AKS cluster, use either the Azure portal or Azure CLI:

- Azure portal
  1. Launch the AKS service in the Azure portal by clicking **All services**, then searching for and selecting **Kubernetes services**.
  2. Select your AKS cluster where you want to disable the Azure Policy Add-on.
  3. Select **Policies (preview)** on the left side of the Kubernetes service page.



4. In the main page, select the **Disable add-on** button.



- Azure CLI

```
# Log in first with az login if you're not using Cloud Shell  
az aks disable-addons --addons azure-policy --name MyAKSCluster --resource-group MyResourceGroup
```

## Diagnostic data collected by Azure Policy Add-on

The Azure Policy Add-on for Kubernetes collects limited cluster diagnostic data. This diagnostic data is vital technical data related to software and performance. It's used in the following ways:

- Keep Azure Policy Add-on up to date
- Keep Azure Policy Add-on secure, reliable, performant
- Improve Azure Policy Add-on - through the aggregate analysis of the use of the add-on

The information collected by the add-on isn't personal data. The following details are currently collected:

- Azure Policy Add-on agent version
- Cluster type
- Cluster region
- Cluster resource group
- Cluster resource ID
- Cluster subscription ID
- Cluster OS (Example: Linux)
- Cluster city (Example: Seattle)
- Cluster state or province (Example: Washington)
- Cluster country or region (Example: United States)
- Exceptions/errors encountered by Azure Policy Add-on during agent installation on policy evaluation
- Number of Gatekeeper policies not installed by Azure Policy Add-on

## Next steps

- Review examples at [Azure Policy samples](#).
- Review the [Policy definition structure](#).
- Review [Understanding policy effects](#).
- Understand how to [programmatically create policies](#).
- Learn how to [get compliance data](#).
- Learn how to [remediate non-compliant resources](#).
- Review what a management group is with [Organize your resources with Azure management groups](#).

# Update or rotate the credentials for a service principal in Azure Kubernetes Service (AKS)

2/25/2020 • 2 minutes to read • [Edit Online](#)

By default, AKS clusters are created with a service principal that has a one-year expiration time. As you near the expiration date, you can reset the credentials to extend the service principal for an additional period of time. You may also want to update, or rotate, the credentials as part of a defined security policy. This article details how to update these credentials for an AKS cluster.

## Before you begin

You need the Azure CLI version 2.0.65 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Choose to update or create a service principal

When you want to update the credentials for an AKS cluster, you can choose to:

- update the credentials for the existing service principal used by the cluster, or
- create a service principal and update the cluster to use these new credentials.

### Update Existing Service Principal Expiration

To update the credentials for the existing service principal, get the service principal ID of your cluster using the `az aks show` command. The following example gets the ID for the cluster named *myAKSCluster* in the *myResourceGroup* resource group. The service principal ID is set as a variable named *SP\_ID* for use in additional command.

```
SP_ID=$(az aks show --resource-group myResourceGroup --name myAKSCluster \
--query servicePrincipalProfile.clientId -o tsv)
```

With a variable set that contains the service principal ID, now reset the credentials using `az ad sp credential reset`. The following example lets the Azure platform generate a new secure secret for the service principal. This new secure secret is also stored as a variable.

```
SP_SECRET=$(az ad sp credential reset --name $SP_ID --query password -o tsv)
```

Now continue on to [update AKS cluster with new credentials](#). This step is necessary for the Service Principal changes to reflect on the AKS cluster.

### Create a New Service Principal

If you chose to update the existing service principal credentials in the previous section, skip this step. Continue to [update AKS cluster with new credentials](#).

To create a service principal and then update the AKS cluster to use these new credentials, use the `az ad sp create-for-rbac` command. In the following example, the `--skip-assignment` parameter prevents any additional default assignments being assigned:

```
az ad sp create-for-rbac --skip-assignment
```

The output is similar to the following example. Make a note of your own `appId` and `password`. These values are used in the next step.

```
{  
  "appId": "7d837646-b1f3-443d-874c-fd83c7c739c5",  
  "name": "7d837646-b1f3-443d-874c-fd83c7c739c",  
  "password": "a5ce83c9-9186-426d-9183-614597c7f2f7",  
  "tenant": "a4342dc8-cd0e-4742-a467-3129c469d0e5"  
}
```

Now define variables for the service principal ID and client secret using the output from your own [az ad sp create-for-rbac](#) command, as shown in the following example. The *SP\_ID* is your *appId*, and the *SP\_SECRET* is your *password*:

```
SP_ID=7d837646-b1f3-443d-874c-fd83c7c739c5  
SP_SECRET=a5ce83c9-9186-426d-9183-614597c7f2f7
```

Now continue on to [update AKS cluster with new credentials](#). This step is necessary for the Service Principal changes to reflect on the AKS cluster.

## Update AKS cluster with new credentials

Regardless of whether you chose to update the credentials for the existing service principal or create a service principal, you now update the AKS cluster with your new credentials using the [az aks update-credentials](#) command. The variables for the `--service-principal` and `--client-secret` are used:

```
az aks update-credentials \  
  --resource-group myResourceGroup \  
  --name myAKSCluster \  
  --reset-service-principal \  
  --service-principal $SP_ID \  
  --client-secret $SP_SECRET
```

It takes a few moments for the service principal credentials to be updated in the AKS.

## Next steps

In this article, the service principal for the AKS cluster itself was updated. For more information on how to manage identity for workloads within a cluster, see [Best practices for authentication and authorization in AKS](#).

# Control egress traffic for cluster nodes in Azure Kubernetes Service (AKS)

2/25/2020 • 9 minutes to read • [Edit Online](#)

By default, AKS clusters have unrestricted outbound (egress) internet access. This level of network access allows nodes and services you run to access external resources as needed. If you wish to restrict egress traffic, a limited number of ports and addresses must be accessible to maintain healthy cluster maintenance tasks. Your cluster is configured by default to only use base system container images from Microsoft Container Registry (MCR) or Azure Container Registry (ACR). Configure your preferred firewall and security rules to allow these required ports and addresses.

This article details what network ports and fully qualified domain names (FQDNs) are required and optional if you restrict egress traffic in an AKS cluster.

## IMPORTANT

This document covers only how to lock down the traffic leaving the AKS subnet. AKS has no ingress requirements. Blocking internal subnet traffic using network security groups (NSGs) and firewalls is not supported. To control and block the traffic within the cluster, use [Network Policies](#).

## Before you begin

You need the Azure CLI version 2.0.66 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Egress traffic overview

For management and operational purposes, nodes in an AKS cluster need to access certain ports and fully qualified domain names (FQDNs). These actions could be to communicate with the API server, or to download and then install core Kubernetes cluster components and node security updates. By default, egress (outbound) internet traffic is not restricted for nodes in an AKS cluster. The cluster may pull base system container images from external repositories.

To increase the security of your AKS cluster, you may wish to restrict egress traffic. The cluster is configured to pull base system container images from MCR or ACR. If you lock down the egress traffic in this manner, define specific ports and FQDNs to allow the AKS nodes to correctly communicate with required external services. Without these authorized ports and FQDNs, your AKS nodes can't communicate with the API server or install core components.

You can use [Azure Firewall](#) or a 3rd-party firewall appliance to secure your egress traffic and define these required ports and addresses. AKS does not automatically create these rules for you. The following ports and addresses are for reference as you create the appropriate rules in your network firewall.

### IMPORTANT

When you use Azure Firewall to restrict egress traffic and create a user-defined route (UDR) to force all egress traffic, make sure you create an appropriate DNAT rule in Firewall to correctly allow ingress traffic. Using Azure Firewall with a UDR breaks the ingress setup due to asymmetric routing. (The issue occurs if the AKS subnet has a default route that goes to the firewall's private IP address, but you're using a public load balancer - ingress or Kubernetes service of type: LoadBalancer). In this case, the incoming load balancer traffic is received via its public IP address, but the return path goes through the firewall's private IP address. Because the firewall is stateful, it drops the returning packet because the firewall isn't aware of an established session. To learn how to integrate Azure Firewall with your ingress or service load balancer, see [Integrate Azure Firewall with Azure Standard Load Balancer](#). You can lock down the traffic for TCP port 9000 and TCP port 22 using a network rule between the egress worker node IP(s) and the IP for the API server.

In AKS, there are two sets of ports and addresses:

- The [required ports and address for AKS clusters](#) details the minimum requirements for authorized egress traffic.
- The [optional recommended addresses and ports for AKS clusters](#) aren't required for all scenarios, but integration with other services such as Azure Monitor won't work correctly. Review this list of optional ports and FQDNs, and authorize any of the services and components used in your AKS cluster.

### NOTE

Limiting egress traffic only works on new AKS clusters. For existing clusters, [perform a cluster upgrade operation](#) using the `az aks upgrade` command before you limit the egress traffic.

## Required ports and addresses for AKS clusters

The following outbound ports / network rules are required for an AKS cluster:

- TCP port 443
- TCP [IPAddrOfYourAPIServer]:443 is required if you have an app that needs to talk to the API server. This change can be set after the cluster is created.
- TCP port 9000 and TCP port 22 for the tunnel front pod to communicate with the tunnel end on the API server.
  - To get more specific, see the `*.hcp.<location>.azmk8s.io` and `*.tun.<location>.azmk8s.io` addresses in the following table.
- UDP port 123 for Network Time Protocol (NTP) time synchronization (Linux nodes).
- UDP port 53 for DNS is also required if you have pods directly accessing the API server.

The following FQDN / application rules are required:

### IMPORTANT

**\*.blob.core.windows.net** and **aksrepos.azurecr.io** are no longer required FQDN rules for egress lockdown. For existing clusters, [perform a cluster upgrade operation](#) using the `az aks upgrade` command to remove these rules.

- Azure Global

FQDN	PORT	USE
<code>*.hcp.&lt;location&gt;.azmk8s.io</code>	HTTPS:443, TCP:22, TCP:9000	This address is the API server endpoint. Replace <code>&lt;location&gt;</code> with the region where your AKS cluster is deployed.

FQDN	PORT	USE
*.tun.<location>.azmk8s.io	HTTPS:443, TCP:22, TCP:9000	This address is the API server endpoint. Replace <location> with the region where your AKS cluster is deployed.
mcr.microsoft.com	HTTPS:443	This address is required to access images in Microsoft Container Registry (MCR). This registry contains first-party images/charts(for example, moby, etc.) required for the functioning of the cluster during upgrade and scale of the cluster
*.cdn.mscl.io	HTTPS:443	This address is required for MCR storage backed by the Azure content delivery network (CDN).
management.azure.com	HTTPS:443	This address is required for Kubernetes GET/PUT operations.
login.microsoftonline.com	HTTPS:443	This address is required for Azure Active Directory authentication.
ntp.ubuntu.com	UDP:123	This address is required for NTP time synchronization on Linux nodes.
packages.microsoft.com	HTTPS:443	This address is the Microsoft packages repository used for cached <i>apt-get</i> operations. Example packages include Moby, PowerShell, and Azure CLI.
acs-mirror.azureedge.net	HTTPS:443	This address is for the repository required to install required binaries like kubenet and Azure CNI.

- Azure China 21Vianet

FQDN	PORT	USE
*.hcp.<location>.cx.prod.service.azk8s.cn	HTTPS:443, TCP:22, TCP:9000	This address is the API server endpoint. Replace <location> with the region where your AKS cluster is deployed.
*.tun.<location>.cx.prod.service.azk8s.cn	HTTPS:443, TCP:22, TCP:9000	This address is the API server endpoint. Replace <location> with the region where your AKS cluster is deployed.
*.azk8s.cn	HTTPS:443	This address is required to download required binaries and images

FQDN	PORT	USE
mcr.microsoft.com	HTTPS:443	This address is required to access images in Microsoft Container Registry (MCR). This registry contains first-party images/charts(for example, moby, etc.) required for the functioning of the cluster during upgrade and scale of the cluster
*.cdn.mscl.io	HTTPS:443	This address is required for MCR storage backed by the Azure Content Delivery Network (CDN).
management.chinacloudapi.cn	HTTPS:443	This address is required for Kubernetes GET/PUT operations.
login.chinacloudapi.cn	HTTPS:443	This address is required for Azure Active Directory authentication.
ntp.ubuntu.com	UDP:123	This address is required for NTP time synchronization on Linux nodes.
packages.microsoft.com	HTTPS:443	This address is the Microsoft packages repository used for cached <i>apt-get</i> operations. Example packages include Moby, PowerShell, and Azure CLI.

- Azure Government

FQDN	PORT	USE
*.hcp.<location>.cx.aks.containerservice.azure.us	HTTPS:443, TCP:22, TCP:9000	This address is the API server endpoint. Replace <location> with the region where your AKS cluster is deployed.
*.tun.<location>.cx.aks.containerservice.azure.us	HTTPS:443, TCP:22, TCP:9000	This address is the API server endpoint. Replace <location> with the region where your AKS cluster is deployed.
mcr.microsoft.com	HTTPS:443	This address is required to access images in Microsoft Container Registry (MCR). This registry contains first-party images/charts(for example, moby, etc.) required for the functioning of the cluster during upgrade and scale of the cluster
*.cdn.mscl.io	HTTPS:443	This address is required for MCR storage backed by the Azure Content Delivery Network (CDN).
management.usgovcloudapi.net	HTTPS:443	This address is required for Kubernetes GET/PUT operations.
login.microsoftonline.us	HTTPS:443	This address is required for Azure Active Directory authentication.

FQDN	PORT	USE
ntp.ubuntu.com	UDP:123	This address is required for NTP time synchronization on Linux nodes.
packages.microsoft.com	HTTPS:443	This address is the Microsoft packages repository used for cached <i>apt-get</i> operations. Example packages include Moby, PowerShell, and Azure CLI.
acs-mirror.azureedge.net	HTTPS:443	This address is for the repository required to install required binaries like kubenet and Azure CNI.

## Optional recommended addresses and ports for AKS clusters

The following outbound ports / network rules are optional for an AKS cluster:

The following FQDN / application rules are recommended for AKS clusters to function correctly:

FQDN	PORT	USE
security.ubuntu.com, azure.archive.ubuntu.com, changelogs.ubuntu.com	HTTP:80	This address lets the Linux cluster nodes download the required security patches and updates.

## Required addresses and ports for GPU enabled AKS clusters

The following FQDN / application rules are required for AKS clusters that have GPU enabled:

FQDN	PORT	USE
nvidia.github.io	HTTPS:443	This address is used for correct driver installation and operation on GPU-based nodes.
us.download.nvidia.com	HTTPS:443	This address is used for correct driver installation and operation on GPU-based nodes.
apt.dockerproject.org	HTTPS:443	This address is used for correct driver installation and operation on GPU-based nodes.

## Required addresses and ports with Azure Monitor for containers enabled

The following FQDN / application rules are required for AKS clusters that have the Azure Monitor for containers enabled:

FQDN	PORT	USE

FQDN	PORT	USE
dc.services.visualstudio.com	HTTPS:443	This is for correct metrics and monitoring telemetry using Azure Monitor.
*.ods.opinsights.azure.com	HTTPS:443	This is used by Azure Monitor for ingesting log analytics data.
*.oms.opinsights.azure.com	HTTPS:443	This address is used by omsagent, which is used to authenticate the log analytics service.
*.microsoftonline.com	HTTPS:443	This is used for authenticating and sending metrics to Azure Monitor.
*.monitoring.azure.com	HTTPS:443	This is used to send metrics data to Azure Monitor.

## Required addresses and ports with Azure Dev Spaces enabled

The following FQDN / application rules are required for AKS clusters that have the Azure Dev Spaces enabled:

FQDN	PORT	USE
cloudflare.docker.com	HTTPS:443	This address is used to pull linux alpine and other Azure Dev Spaces images
gcr.io	HTTP:443	This address is used to pull helm/tiller images
storage.googleapis.com	HTTP:443	This address is used to pull helm/tiller images
azds-..azds.io	HTTPS:443	To communicate with Azure Dev Spaces backend services for your controller. The exact FQDN can be found in the "dataplaneFqdn" in %USERPROFILE%.azds\settings.json

## Required addresses and ports for AKS clusters with Azure Policy (in public preview) enabled

### Caution

Some of the features below are in preview. The suggestions in this article are subject to change as the feature moves to public preview and future release stages.

The following FQDN / application rules are required for AKS clusters that have the Azure Policy enabled.

FQDN	PORT	USE
gov-prod-policy-data.trafficmanager.net	HTTPS:443	This address is used for correct operation of Azure Policy. (currently in preview in AKS)

FQDN	PORT	USE
raw.githubusercontent.com	HTTPS:443	This address is used to pull the built-in policies from GitHub to ensure correct operation of Azure Policy. (currently in preview in AKS)
*.gk.azmk8s.io	HTTPS:443	Azure policy add-on that talks to Gatekeeper audit endpoint running in master server to get the audit results.
dc.services.visualstudio.com	HTTPS:443	Azure policy add-on that sends telemetry data to applications insights endpoint.

## Required by Windows Server based nodes (in public preview) enabled

### Caution

Some of the features below are in preview. The suggestions in this article are subject to change as the feature moves to public preview and future release stages.

The following FQDN / application rules are required for Windows Server based AKS clusters:

FQDN	PORT	USE
onegetcdn.azureedge.net, winlayers.blob.core.windows.net, winlayers.cdn.mscr.io, go.microsoft.com	HTTPS:443	To install windows-related binaries
mp.microsoft.com, www.msftconnecttest.com, ctldl.windowsupdate.com	HTTP:80	To install windows-related binaries
kms.core.windows.net	TCP:1688	To install windows-related binaries

## Next steps

In this article, you learned what ports and addresses to allow if you restrict egress traffic for the cluster. You can also define how the pods themselves can communicate and what restrictions they have within the cluster. For more information, see [Secure traffic between pods using network policies in AKS](#).

# Integrate Azure Active Directory with Azure Kubernetes Service using the Azure CLI

2/25/2020 • 8 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) can be configured to use Azure Active Directory (AD) for user authentication. In this configuration, you can log into an AKS cluster using an Azure AD authentication token. Cluster operators can also configure Kubernetes role-based access control (RBAC) based on a user's identity or directory group membership.

This article shows you how to create the required Azure AD components, then deploy an Azure AD-enabled cluster and create a basic RBAC role in the AKS cluster. You can also [complete these steps using the Azure portal](#).

For the complete sample script used in this article, see [Azure CLI samples - AKS integration with Azure AD](#).

The following limitations apply:

- Azure AD can only be enabled when you create a new, RBAC-enabled cluster. You can't enable Azure AD on an existing AKS cluster.

## Before you begin

You need the Azure CLI version 2.0.61 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

For consistency and to help run the commands in this article, create a variable for your desired AKS cluster name. The following example uses the name *myakscluster*:

```
aksname="myakscluster"
```

## Azure AD authentication overview

Azure AD authentication is provided to AKS clusters with OpenID Connect. OpenID Connect is an identity layer built on top of the OAuth 2.0 protocol. For more information on OpenID Connect, see the [Open ID connect documentation](#).

From inside of the Kubernetes cluster, Webhook Token Authentication is used to verify authentication tokens. Webhook token authentication is configured and managed as part of the AKS cluster. For more information on Webhook token authentication, see the [webhook authentication documentation](#).

### NOTE

When configuring Azure AD for AKS authentication, two Azure AD applications are configured. This operation must be completed by an Azure tenant administrator.

## Create Azure AD server component

To integrate with AKS, you create and use an Azure AD application that acts as an endpoint for the identity requests. The first Azure AD application you need gets Azure AD group membership for a user.

Create the server application component using the `az ad app create` command, then update the group

membership claims using the [az ad app update](#) command. The following example uses the `aksname` variable defined in the [Before you begin](#) section, and creates a variable

```
# Create the Azure AD application
serverApplicationId=$(az ad app create \
    --display-name "${aksname}Server" \
    --identifier-uris "https://${aksname}Server" \
    --query appId -o tsv)

# Update the application group membership claims
az ad app update --id $serverApplicationId --set groupMembershipClaims=All
```

Now create a service principal for the server app using the [az ad sp create](#) command. This service principal is used to authenticate itself within the Azure platform. Then, get the service principal secret using the [az ad sp credential reset](#) command and assign to the variable named `serverApplicationSecret` for use in one of the following steps:

```
# Create a service principal for the Azure AD application
az ad sp create --id $serverApplicationId

# Get the service principal secret
serverApplicationSecret=$(az ad sp credential reset \
    --name $serverApplicationId \
    --credential-description "AKSPassword" \
    --query password -o tsv)
```

The Azure AD needs permissions to perform the following actions:

- Read directory data
- Sign in and read user profile

Assign these permissions using the [az ad app permission add](#) command:

```
az ad app permission add \
    --id $serverApplicationId \
    --api 00000003-0000-0000-c000-000000000000 \
    --api-permissions e1fe6dd8-ba31-4d61-89e7-88639da4683d=Scope 06da0dbc-49e2-44d2-8312-53f166ab848a=Scope
    7ab1d382-f21e-4acd-a863-ba3e13f7da61=Role
```

Finally, grant the permissions assigned in the previous step for the server application using the [az ad app permission grant](#) command. This step fails if the current account is not a tenant admin. You also need to add permissions for Azure AD application to request information that may otherwise require administrative consent using the [az ad app permission admin-consent](#):

```
az ad app permission grant --id $serverApplicationId --api 00000003-0000-0000-c000-000000000000
az ad app permission admin-consent --id $serverApplicationId
```

## Create Azure AD client component

The second Azure AD application is used when a user logs to the AKS cluster with the Kubernetes CLI (`kubectl`). This client application takes the authentication request from the user and verifies their credentials and permissions. Create the Azure AD app for the client component using the [az ad app create](#) command:

```
clientApplicationId=$(az ad app create \
--display-name "${aksname}Client" \
--native-app \
--reply-urls "https://${aksname}Client" \
--query appId -o tsv)
```

Create a service principal for the client application using the [az ad sp create](#) command:

```
az ad sp create --id $clientApplicationId
```

Get the oAuth2 ID for the server app to allow the authentication flow between the two app components using the [az ad app show](#) command. This oAuth2 ID is used in the next step.

```
oAuthPermissionId=$(az ad app show --id $serverApplicationId --query "oauth2Permissions[0].id" -o tsv)
```

Add the permissions for the client application and server application components to use the oAuth2 communication flow using the [az ad app permission add](#) command. Then, grant permissions for the client application to communicate with the server application using the [az ad app permission grant](#) command:

```
az ad app permission add --id $clientApplicationId --api $serverApplicationId --api-permissions \
${oAuthPermissionId}=Scope
az ad app permission grant --id $clientApplicationId --api $serverApplicationId
```

## Deploy the cluster

With the two Azure AD applications created, now create the AKS cluster itself. First, create a resource group using the [az group create](#) command. The following example creates the resource group in the *EastUS* region:

Create a resource group for the cluster:

```
az group create --name myResourceGroup --location EastUS
```

Get the tenant ID of your Azure subscription using the [az account show](#) command. Then, create the AKS cluster using the [az aks create](#) command. The command to create the AKS cluster provides the server and client application IDs, the server application service principal secret, and your tenant ID:

```
tenantId=$(az account show --query tenantId -o tsv)

az aks create \
--resource-group myResourceGroup \
--name $aksname \
--node-count 1 \
--generate-ssh-keys \
--aad-server-app-id $serverApplicationId \
--aad-server-app-secret $serverApplicationSecret \
--aad-client-app-id $clientApplicationId \
--aad-tenant-id $tenantId
```

Finally, get the cluster admin credentials using the [az aks get-credentials](#) command. In one of the following steps, you get the regular *user* cluster credentials to see the Azure AD authentication flow in action.

```
az aks get-credentials --resource-group myResourceGroup --name $aksname --admin
```

## Create RBAC binding

Before an Azure Active Directory account can be used with the AKS cluster, a role binding or cluster role binding needs to be created. *Roles* define the permissions to grant, and *bindings* apply them to desired users. These assignments can be applied to a given namespace, or across the entire cluster. For more information, see [Using RBAC authorization](#).

Get the user principal name (UPN) for the user currently logged in using the `az ad signed-in-user show` command. This user account is enabled for Azure AD integration in the next step.

```
az ad signed-in-user show --query userPrincipalName -o tsv
```

### IMPORTANT

If the user you grant the RBAC binding for is in the same Azure AD tenant, assign permissions based on the *userPrincipalName*. If the user is in a different Azure AD tenant, query for and use the *objectId* property instead.

Create a YAML manifest named `basic-azure-ad-binding.yaml` and paste the following contents. On the last line, replace *userPrincipalName\_or\_objectId* with the UPN or object ID output from the previous command:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: contoso-cluster-admins
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: userPrincipalName_or_objectId
```

Create the ClusterRoleBinding using the `kubectl apply` command and specify the filename of your YAML manifest:

```
kubectl apply -f basic-azure-ad-binding.yaml
```

## Access cluster with Azure AD

Now let's test the integration of Azure AD authentication for the AKS cluster. Set the `kubectl` config context to use regular user credentials. This context passes all authentication requests back through Azure AD.

```
az aks get-credentials --resource-group myResourceGroup --name $aksname --overwrite-existing
```

Now use the `kubectl get pods` command to view pods across all namespaces:

```
kubectl get pods --all-namespaces
```

You receive a sign in prompt to authenticate using Azure AD credentials using a web browser. After you've successfully authenticated, the `kubectl` command displays the pods in the AKS cluster, as shown in the following example output:

```
$ kubectl get pods --all-namespaces
```

To sign in, use a web browser to open the page <https://microsoft.com/devicelogin> and enter the code BYMK7UXVD to authenticate.

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
kube-system	coredns-754f947b4-2v75r	1/1	Running	0	23h
kube-system	coredns-754f947b4-tghwh	1/1	Running	0	23h
kube-system	coredns-autoscaler-6fcdb7d64-4wkvp	1/1	Running	0	23h
kube-system	heapster-5fb7488d97-t5wzk	2/2	Running	0	23h
kube-system	kube-proxy-2nd5m	1/1	Running	0	23h
kube-system	kube-svc-redirect-swp9r	2/2	Running	0	23h
kube-system	kubernetes-dashboard-847bb4ddc6-trt7m	1/1	Running	0	23h
kube-system	metrics-server-7b97f9cd9-btxzz	1/1	Running	0	23h
kube-system	tunnelfront-6ff887cffb-xkfmq	1/1	Running	0	23h

The authentication token received for `kubectl` is cached. You are only reprompted to sign in when the token has expired or the Kubernetes config file is re-created.

If you see an authorization error message after you've successfully signed in using a web browser as in the following example output, check the following possible issues:

```
error: You must be logged in to the server (Unauthorized)
```

- You defined the appropriate object ID or UPN, depending on if the user account is in the same Azure AD tenant or not.
- The user is not a member of more than 200 groups.
- Secret defined in the application registration for server matches the value configured using

```
--aad-server-app-secret
```

## Next steps

For the complete script that contains the commands shown in this article, see the [Azure AD integration script in the AKS samples repo](#).

To use Azure AD users and groups to control access to cluster resources, see [Control access to cluster resources using role-based access control and Azure AD identities in AKS](#).

For more information about how to secure Kubernetes clusters, see [Access and identity options for AKS](#).

For best practices on identity and resource control, see [Best practices for authentication and authorization in AKS](#).

# Integrate Azure Active Directory with Azure Kubernetes Service

2/25/2020 • 9 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) can be configured to use Azure Active Directory (Azure AD) for user authentication. In this configuration, you can sign in to an AKS cluster by using your Azure AD authentication token.

Cluster administrators can configure Kubernetes role-based access control (RBAC) based on a user's identity or directory group membership.

This article explains how to:

- Deploy the prerequisites for AKS and Azure AD.
- Deploy an Azure AD-enabled cluster.
- Create a basic RBAC role in the AKS cluster by using the Azure portal.

You can also complete these steps by using the [Azure CLI](#).

## NOTE

Azure AD can only be enabled when you create a new RBAC-enabled cluster. You can't enable Azure AD on an existing AKS cluster.

## Authentication details

Azure AD authentication is provided to AKS clusters that have OpenID Connect. OpenID Connect is an identity layer built on top of the OAuth 2.0 protocol.

For more information about OpenID Connect, see [Authorize access to web applications using OpenID Connect and Azure AD](#).

Inside a Kubernetes cluster, webhook token authentication is used to authenticate tokens. Webhook token authentication is configured and managed as part of the AKS cluster.

For more information about webhook token authentication, see the [Webhook Token Authentication](#) section in Kubernetes Documentation.

To provide Azure AD authentication for an AKS cluster, two Azure AD applications are created. The first application is a server component that provides user authentication. The second application is a client component that's used when you're prompted by the CLI for authentication. This client application uses the server application for the actual authentication of the credentials provided by the client.

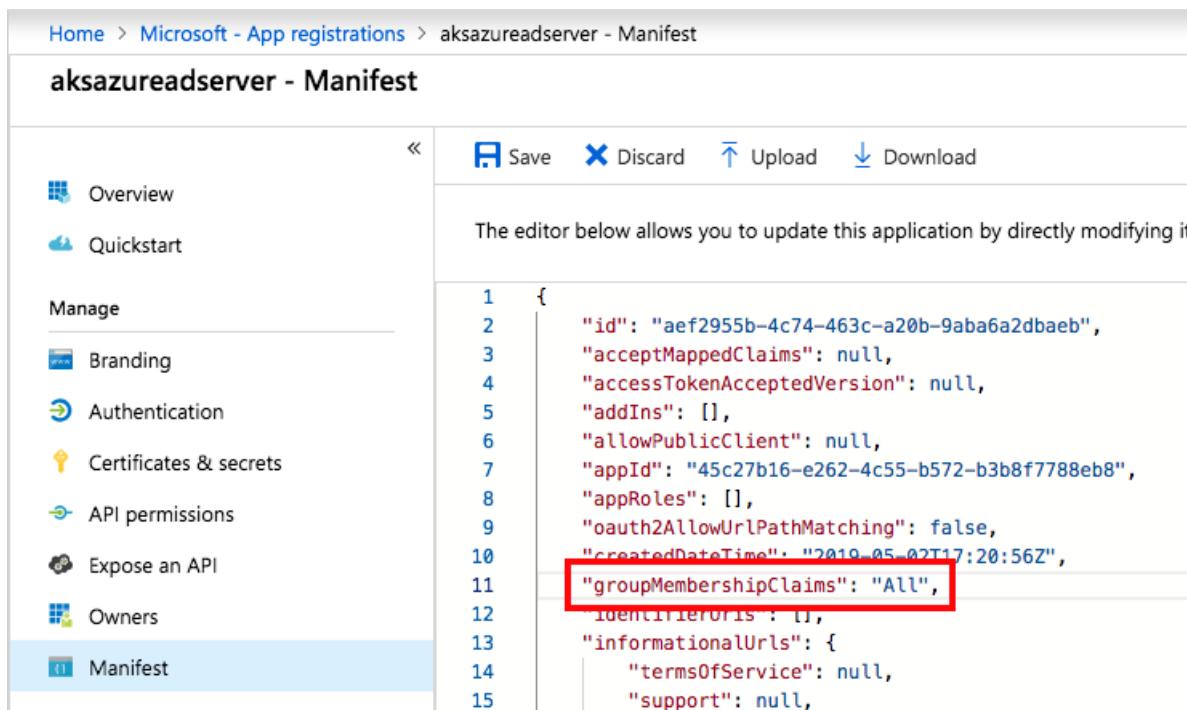
## NOTE

When you configure Azure AD for AKS authentication, two Azure AD applications are configured. The steps to delegate permissions for each application must be completed by an Azure tenant administrator.

## Create the server application

The first Azure AD application is applied to get a user's Azure AD group membership. To create this application in the Azure portal:

1. Select **Azure Active Directory > App registrations > New registration**.
  - a. Give the application a name, such as *AKSAzureADServer*.
  - b. For **Supported account types**, select **Accounts in this organizational directory only**.
  - c. Choose **Web** for the Redirect URI type, and then enter any URI-formatted value, such as <https://aksazureadserver>.
  - d. Select **Register** when you're finished.
2. Select **Manifest**, and then edit the **groupMembershipClaims**: value as **All**. When you're finished with the updates, select **Save**.



The screenshot shows the Azure AD Manifest editor for the application 'aksazureadserver'. The left sidebar has a 'Manage' section with options like Overview, Quickstart, Branding, Authentication, Certificates & secrets, API permissions, Expose an API, Owners, and Manifest. The 'Manifest' option is selected. The right pane contains a JSON editor with the following code, where the line 'groupMembershipClaims': 'All' is highlighted with a red box:

```
1  {
2    "id": "aef2955b-4c74-463c-a20b-9aba6a2dbaeb",
3    "acceptMappedClaims": null,
4    "accessTokenAcceptedVersion": null,
5    "addIns": [],
6    "allowPublicClient": null,
7    "appId": "45c27b16-e262-4c55-b572-b3b8f7788eb8",
8    "appRoles": [],
9    "oauth2AllowUrlPathMatching": false,
10   "createdDateTime": "2019-05-07T17:20:56Z",
11   "groupMembershipClaims": "All",
12   "identifierUris": [],
13   "informationalUrls": {
14     "termsOfService": null,
15     "support": null,
```

3. In the left pane of the Azure AD application, select **Certificates & secrets**.
  - a. Select **+ New client secret**.
  - b. Add a key description, such as *AKS Azure AD server*. Choose an expiration time, and then select **Add**.
  - c. Note the key value, which is displayed only at this time. When you deploy an Azure AD-enabled AKS cluster, this value is called the server application secret.
4. In the left pane of the Azure AD application, select **API permissions**, and then select **+ Add a permission**.
  - a. Under **Microsoft APIs**, select **Microsoft Graph**.
  - b. Select **Delegated permissions**, and then select the check box next to **Directory > Directory.Read.All (Read directory data)**.
  - c. If a default delegated permission for **User > User.Read (Sign in and read user profile)** doesn't exist, select the check box next to it.
  - d. Select **Application permissions**, and then select the check box next to **Directory > Directory.Read.All (Read directory data)**.

## API permissions

Applications are authorized to use APIs by requesting permissions. These permissions show up during the consent process where users are given the opportunity to grant/deny access.

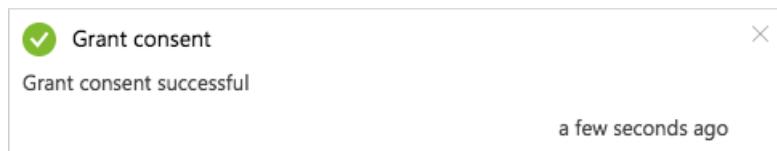
API / PERMISSIONS NAME				TYPE	DESCRIPTION	ADMIN CONSENT REQUIRED
▼ Microsoft Graph (3)						
Directory.Read.All	Delegated	Read directory data	Yes	⚠ Not granted for Micro...		
Directory.Read.All	Application	Read directory data	Yes	⚠ Not granted for Micro...		
User.Read	Delegated	Sign in and read user profile	-			

These are the permissions that this application requests statically. You may also request user consen-table permissions dynamically through code. [See best practices for requesting permissions](#)

e. Select **Add permissions** to save the updates.

f. Under **Grant consent**, select **Grant admin consent**. This button won't be available the current account being used is not listed as a tenant admin.

When permissions are successfully granted, the following notification is displayed in the portal:



5. In the left pane of the Azure AD application, select **Expose an API**, and then select **+ Add a scope**.

a. Enter a **Scope name**, an **Admin consent display name**, and then an **Admin consent description** such as *AKSAzureADServer*.

b. Make sure **State** is set to **Enabled**.

A screenshot of the "Add a scope" dialog. It includes fields for "Scope name" (AKSAzureADServer), "Who can consent?" (Admins only selected), "Admin consent display name" (AKSAzureADServer), "Admin consent description" (AKSAzureADServer), and "State" (Enabled selected).

c. Select **Add scope**.

6. Return to the application **Overview** page and note the **Application (client) ID**. When you deploy an Azure AD-enabled AKS cluster, this value is called the server application ID.

aksazureadserver

Display name : aksazureadserver

Application (client) ID : 45c27b16-e262-4c55-b572-b3b8f7788eb8

Object ID : aef2955b-4c74-463c-a20b-9aba6a2dbaeb

Supported account types : My organization only

Redirect URIs : 1 web, 0 public client

Managed application in ... : aksazureadserver

## Create the client application

The second Azure AD application is used when you sign in with the Kubernetes CLI (`kubectl`).

1. Select **Azure Active Directory > App registrations > New registration**.

a. Give the application a name, such as *AKSAzureADClient*.

b. For **Supported account types**, select **Accounts in this organizational directory only**.

c. Select **Web** for the Redirect URI type, and then enter any URL-formatted value such as `https://aksazureadclient`.

### NOTE

If you are creating a new RBAC-enabled cluster to support Azure Monitor for containers, add the following two additional redirect URLs to this list as **Web** application types. The first base URL value should be

`https://afd.hosting.portal.azure.net/monitoring/Content/iframe/infrainights.app/web/base-libs/auth/auth.html`

and the second base URL value should be

`https://monitoring.hosting.portal.azure.net/monitoring/Content/iframe/infrainights.app/web/base-libs/auth/auth.html`

If you're using this feature in Azure China, the first base URL value should be

`https://afd.hosting.azureportal.chinaloudapi.cn/monitoring/Content/iframe/infrainights.app/web/base-libs/auth/auth.html`

and the second base URL value should be

`https://monitoring.hosting.azureportal.chinaloudapi.cn/monitoring/Content/iframe/infrainights.app/web/base-libs/auth/auth.html`

For further information, see [How to setup the Live Data \(preview\) feature](#) for Azure Monitor for containers, and the steps for configuring authentication under the [Configure AD integrated authentication](#) section.

d. Select **Register** when you're finished.

2. In the left pane of the Azure AD application, select **API permissions**, and then select **+ Add a permission**.

a. Select **My APIs**, and then choose your Azure AD server application created in the previous step, such as *AKSAzureADServer*.

b. Select **Delegated permissions**, and then select the check box next to your Azure AD server app.

**Request API permissions**

[All APIs](#)

**AKSAzureADServer**  
api://6affb2f2-2985-4075-bb05-7f51ed4f052b

What type of permissions does your application require?

**Delegated permissions**  
Your application needs to access the API as the signed-in user.

**Application permissions**  
Your application runs as a background service or daemon without a signed-in user.

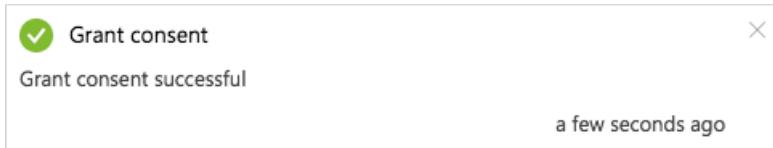
Select permissions [expand all](#)

Type to search

PERMISSION	ADMIN CONSENT REQUIRED
<input checked="" type="checkbox"/> AKSAzureADServer AKSAzureADServer <a href="#">@</a>	Yes

c. Select **Add permissions**.

d. Under **Grant consent**, select **Grant admin consent**. This button isn't available if the current account isn't a tenant admin. When permissions are granted, the following notification is displayed in the portal:



3. In the left pane of the Azure AD application, select **Authentication**.

- Under **Default client type**, select **Yes** to **Treat the client as a public client**.

4. In the left pane of the Azure AD application, note the application ID. When you deploy an Azure AD-enabled AKS cluster, this value is called the client application ID.

**AKSAzureADClient**

Overview	Delete	Endpoints
Quickstart		
Manage		
Branding		

Display name : AKSAzureADClient

Application (client) ID : f8cd1fd9-154f-4da7-b348-595f283c13a3

Directory (tenant) ID :

Object ID : e9604487-a951-4fe1-98c4-c23bd0cddcb1

## Get the tenant ID

Next, get the ID of your Azure tenant. This value is used when you create the AKS cluster.

From the Azure portal, select **Azure Active Directory > Properties** and note the **Directory ID**. When you create an Azure AD-enabled AKS cluster, this value is called the tenant ID.

Save Discard

\* Name  
Azure AD (Office 365 Subscription)

Country or region  
United States

Location  
United States datacenters

Notification language  
English

Global admin can manage Azure Subscriptions and Management Groups  
Yes No

Directory ID

Technical contact

Global privacy contact

Privacy statement URL

## Deploy the AKS cluster

Use the [az group create](#) command to create a resource group for the AKS cluster.

```
az group create --name myResourceGroup --location eastus
```

Use the [az aks create](#) command to deploy the AKS cluster. Next, replace the values in the following sample command. Use the values collected when you created the Azure AD applications for the server app ID, app secret, client app ID, and tenant ID.

```
az aks create \
--resource-group myResourceGroup \
--name myAKScluster \
--generate-ssh-keys \
--aad-server-app-id b1536b67-29ab-4b63-b60f-9444d0c15df1 \
--aad-server-app-secret wHYomLe2i1mHR2B3/d4sFrooHwADZccKwfQWK2QHg= \
--aad-client-app-id 8aaaf8bd5-1bdd-4822-99ad-02bfaa63eea7 \
--aad-tenant-id 72f988bf-0000-0000-0000-2d7cd011db47
```

An AKS cluster takes a few minutes to create.

## Create an RBAC binding

### NOTE

The cluster role binding name is case sensitive.

Before you use an Azure Active Directory account with an AKS cluster, you must create role-binding or cluster role-binding. Roles define the permissions to grant, and bindings apply them to desired users. These assignments

can be applied to a given namespace, or across the entire cluster. For more information, see [Using RBAC authorization](#).

First, use the `az aks get-credentials` command with the `--admin` argument to sign in to the cluster with admin access.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --admin
```

Next, create ClusterRoleBinding for an Azure AD account that you want to grant access to the AKS cluster. The following example gives the account full access to all namespaces in the cluster:

- If the user you grant the RBAC binding for is in the same Azure AD tenant, assign permissions based on the user principal name (UPN). Move on to the step to create the YAML manifest for ClusterRoleBinding.
- If the user is in a different Azure AD tenant, query for and use the **objectId** property instead. If needed, get the objectId of the required user account by using the `az ad user show` command. Provide the user principal name (UPN) of the required account:

```
az ad user show --upn-or-object-id user@contoso.com --query objectId -o tsv
```

Create a file, such as `rbac-aad-user.yaml`, and then paste the following contents. On the last line, replace **userPrincipalName\_or\_objectId** with the UPN or object ID. The choice depends on whether the user is the same Azure AD tenant or not.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: contoso-cluster-admins
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: userPrincipalName_or_objectId
```

Apply the binding by using the `kubectl apply` command as shown in the following example:

```
kubectl apply -f rbac-aad-user.yaml
```

A role binding can also be created for all members of an Azure AD group. Azure AD groups are specified by using the group object ID, as shown in the following example.

Create a file, such as `rbac-aad-group.yaml`, and then paste the following contents. Update the group object ID with one from your Azure AD tenant:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: contoso-cluster-admins
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: "894656e1-39f8-4bfe-b16a-510f61af6f41"
```

Apply the binding by using the [kubectl apply](#) command as shown in the following example:

```
kubectl apply -f rbac-aad-group.yaml
```

For more information on securing a Kubernetes cluster with RBAC, see [Using RBAC Authorization](#).

## Access the cluster with Azure AD

Pull the context for the non-admin user by using the [az aks get-credentials](#) command.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

After you run the `kubectl` command, you'll be prompted to authenticate by using Azure. Follow the on-screen instructions to finish the process, as shown in the following example:

```
$ kubectl get nodes

To sign in, use a web browser to open https://microsoft.com/devicelogin. Next, enter the code BUJHWDGNL to
authenticate.

NAME           STATUS   ROLES      AGE    VERSION
aks-nodepool1-79590246-0  Ready    agent      1h    v1.13.5
aks-nodepool1-79590246-1  Ready    agent      1h    v1.13.5
aks-nodepool1-79590246-2  Ready    agent      1h    v1.13.5
```

When the process is finished, the authentication token is cached. You're only prompted to sign in again when the token expires, or the Kubernetes config file is re-created.

If you see an authorization error message after you successfully sign in, check the following criteria:

```
error: You must be logged in to the server (Unauthorized)
```

- You defined the appropriate object ID or UPN, depending on if the user account is in the same Azure AD tenant or not.
- The user isn't a member of more than 200 groups.
- The secret defined in the application registration for server matches the value configured by using `--aad-server-app-secret`.

## Next steps

To use Azure AD users and groups to control access to cluster resources, see [Control access to cluster resources using role-based access control and Azure AD identities in AKS](#).

For more information about how to secure Kubernetes clusters, see [Access and identity options for AKS](#).

To learn more about identity and resource control, see [Best practices for authentication and authorization in AKS](#).

# Control access to cluster resources using role-based access control and Azure Active Directory identities in Azure Kubernetes Service

2/25/2020 • 10 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) can be configured to use Azure Active Directory (AD) for user authentication. In this configuration, you sign in to an AKS cluster using an Azure AD authentication token. You can also configure Kubernetes role-based access control (RBAC) to limit access to cluster resources based a user's identity or group membership.

This article shows you how to use Azure AD group membership to control access to namespaces and cluster resources using Kubernetes RBAC in an AKS cluster. Example groups and users are created in Azure AD, then Roles and RoleBindings are created in the AKS cluster to grant the appropriate permissions to create and view resources.

## Before you begin

This article assumes that you have an existing AKS cluster enabled with Azure AD integration. If you need an AKS cluster, see [Integrate Azure Active Directory with AKS](#).

You need the Azure CLI version 2.0.61 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create demo groups in Azure AD

In this article, let's create two user roles that can be used to show how Kubernetes RBAC and Azure AD control access to cluster resources. The following two example roles are used:

- **Application developer**
  - A user named *aksdev* that is part of the *appdev* group.
- **Site reliability engineer**
  - A user named *akssre* that is part of the *opssre* group.

In production environments, you can use existing users and groups within an Azure AD tenant.

First, get the resource ID of your AKS cluster using the `az aks show` command. Assign the resource ID to a variable named *AKS\_ID* so that it can be referenced in additional commands.

```
AKS_ID=$(az aks show \
    --resource-group myResourceGroup \
    --name myAKSCluster \
    --query id -o tsv)
```

Create the first example group in Azure AD for the application developers using the `az ad group create` command. The following example creates a group named *appdev*:

```
APPDEV_ID=$(az ad group create --display-name appdev --mail-nickname appdev --query objectId -o tsv)
```

Now, create an Azure role assignment for the *appdev* group using the `az role assignment create` command. This

assignment lets any member of the group use `kubectl` to interact with an AKS cluster by granting them the *Azure Kubernetes Service Cluster User Role*.

```
az role assignment create \
--assignee $APPDEV_ID \
--role "Azure Kubernetes Service Cluster User Role" \
--scope $AKS_ID
```

#### TIP

If you receive an error such as

```
Principal 35bfec9328bd4d8d9b54dea6dac57b82 does not exist in the directory a5443cdcd-cd0e-494d-a387-3039b419f0d5.
```

, wait a few seconds for the Azure AD group object ID to propagate through the directory then try the

```
az role assignment create
```

command again.

Create a second example group, this one for SREs named *opssre*:

```
OPSSRE_ID=$(az ad group create --display-name opssre --mail-nickname opssre --query objectId -o tsv)
```

Again, create an Azure role assignment to grant members of the group the *Azure Kubernetes Service Cluster User Role*:

```
az role assignment create \
--assignee $OPSSRE_ID \
--role "Azure Kubernetes Service Cluster User Role" \
--scope $AKS_ID
```

## Create demo users in Azure AD

With two example groups created in Azure AD for our application developers and SREs, now lets create two example users. To test the RBAC integration at the end of the article, you sign in to the AKS cluster with these accounts.

Create the first user account in Azure AD using the `az ad user create` command.

The following example creates a user with the display name *AKS Dev* and the user principal name (UPN) of `aksdev@contoso.com`. Update the UPN to include a verified domain for your Azure AD tenant (replace `contoso.com` with your own domain), and provide your own secure `--password` credential:

```
AKSDEV_ID=$(az ad user create \
--display-name "AKS Dev" \
--user-principal-name aksdev@contoso.com \
--password P@ssw0rd1 \
--query objectId -o tsv)
```

Now add the user to the *appdev* group created in the previous section using the `az ad group member add` command:

```
az ad group member add --group appdev --member-id $AKSDEV_ID
```

Create a second user account. The following example creates a user with the display name *AKS SRE* and the user principal name (UPN) of `akssre@contoso.com`. Again, update the UPN to include a verified domain for your Azure

AD tenant (replace `contoso.com` with your own domain), and provide your own secure `--password` credential:

```
# Create a user for the SRE role
AKSSRE_ID=$(az ad user create \
--display-name "AKS SRE" \
--user-principal-name akssre@contoso.com \
--password P@ssw0rd1 \
--query objectId -o tsv)

# Add the user to the opssre Azure AD group
az ad group member add --group opssre --member-id $AKSSRE_ID
```

## Create the AKS cluster resources for app devs

The Azure AD groups and users are now created. Azure role assignments were created for the group members to connect to an AKS cluster as a regular user. Now, let's configure the AKS cluster to allow these different groups access to specific resources.

First, get the cluster admin credentials using the [az aks get-credentials](#) command. In one of the following sections, you get the regular *user* cluster credentials to see the Azure AD authentication flow in action.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --admin
```

Create a namespace in the AKS cluster using the [kubectl create namespace](#) command. The following example creates a namespace name `dev`:

```
kubectl create namespace dev
```

In Kubernetes, *Roles* define the permissions to grant, and *RoleBindings* apply them to desired users or groups. These assignments can be applied to a given namespace, or across the entire cluster. For more information, see [Using RBAC authorization](#).

First, create a Role for the `dev` namespace. This role grants full permissions to the namespace. In production environments, you can specify more granular permissions for different users or groups.

Create a file named `role-dev-namespace.yaml` and paste the following YAML manifest:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dev-user-full-access
  namespace: dev
rules:
- apiGroups: [ "", "extensions", "apps" ]
  resources: [ "*" ]
  verbs: [ "*" ]
- apiGroups: [ "batch" ]
  resources:
  - jobs
  - cronjobs
  verbs: [ "*" ]
```

Create the Role using the [kubectl apply](#) command and specify the filename of your YAML manifest:

```
kubectl apply -f role-dev-namespace.yaml
```

Next, get the resource ID for the *appdev* group using the [az ad group show](#) command. This group is set as the subject of a RoleBinding in the next step.

```
az ad group show --group appdev --query objectId -o tsv
```

Now, create a RoleBinding for the *appdev* group to use the previously created Role for namespace access. Create a file named `rolebinding-dev-namespace.yaml` and paste the following YAML manifest. On the last line, replace `groupObjectId` with the group object ID output from the previous command:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: dev-user-access
  namespace: dev
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: dev-user-full-access
subjects:
- kind: Group
  namespace: dev
  name: groupObjectId
```

Create the RoleBinding using the [kubectl apply](#) command and specify the filename of your YAML manifest:

```
kubectl apply -f rolebinding-dev-namespace.yaml
```

## Create the AKS cluster resources for SREs

Now, repeat the previous steps to create a namespace, Role, and RoleBinding for the SREs.

First, create a namespace for *sre* using the [kubectl create namespace](#) command:

```
kubectl create namespace sre
```

Create a file named `role-sre-namespace.yaml` and paste the following YAML manifest:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: sre-user-full-access
  namespace: sre
rules:
- apiGroups: [ "", "extensions", "apps" ]
  resources: [ "*" ]
  verbs: [ "*" ]
- apiGroups: [ "batch" ]
  resources:
  - jobs
  - cronjobs
  verbs: [ "*" ]
```

Create the Role using the [kubectl apply](#) command and specify the filename of your YAML manifest:

```
kubectl apply -f role-sre-namespace.yaml
```

Get the resource ID for the *opssre* group using the [az ad group show](#) command:

```
az ad group show --group opssre --query objectId -o tsv
```

Create a RoleBinding for the *opssre* group to use the previously created Role for namespace access. Create a file named `rolebinding-sre-namespace.yaml` and paste the following YAML manifest. On the last line, replace `groupObjectId` with the group object ID output from the previous command:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  name: sre-user-access
  namespace: sre
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: sre-user-full-access
subjects:
- kind: Group
  namespace: sre
  name: groupObjectId
```

Create the RoleBinding using the [kubectl apply](#) command and specify the filename of your YAML manifest:

```
kubectl apply -f rolebinding-sre-namespace.yaml
```

## Interact with cluster resources using Azure AD identities

Now, let's test the expected permissions work when you create and manage resources in an AKS cluster. In these examples, you schedule and view pods in the user's assigned namespace. Then, you try to schedule and view pods outside of the assigned namespace.

First, reset the *kubeconfig* context using the [az aks get-credentials](#) command. In a previous section, you set the context using the cluster admin credentials. The admin user bypasses Azure AD sign in prompts. Without the `--admin` parameter, the user context is applied that requires all requests to be authenticated using Azure AD.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --overwrite-existing
```

Schedule a basic NGINX pod using the [kubectl run](#) command in the *dev* namespace:

```
kubectl run --generator=run-pod/v1 nginx-dev --image=nginx --namespace dev
```

As the sign in prompt, enter the credentials for your own `appdev@contoso.com` account created at the start of the article. Once you are successfully signed in, the account token is cached for future `kubectl` commands. The NGINX is successfully scheduled, as shown in the following example output:

```
$ kubectl run --generator=run-pod/v1 nginx-dev --image=nginx --namespace dev
```

To sign in, use a web browser to open the page <https://microsoft.com/devicelogin> and enter the code `B24ZD6FP8` to authenticate.

```
pod/nginx-dev created
```

Now use the [kubectl get pods](#) command to view pods in the *dev* namespace.

```
kubectl get pods --namespace dev
```

As shown in the following example output, the NGINX pod is successfully *Running*:

```
$ kubectl get pods --namespace dev
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-dev	1/1	Running	0	4m

### Create and view cluster resources outside of the assigned namespace

Now try to view pods outside of the *dev* namespace. Use the [kubectl get pods](#) command again, this time to see `--all-namespaces` as follows:

```
kubectl get pods --all-namespaces
```

The user's group membership does not have a Kubernetes Role that allows this action, as shown in the following example output:

```
$ kubectl get pods --all-namespaces
```

```
Error from server (Forbidden): pods is forbidden: User "aksdev@contoso.com" cannot list resource "pods" in API group "" at the cluster scope
```

In the same way, try to schedule a pod in different namespace, such as the *sre* namespace. The user's group membership does not align with a Kubernetes Role and RoleBinding to grant these permissions, as shown in the following example output:

```
$ kubectl run --generator=run-pod/v1 nginx-dev --image=nginx --namespace sre
```

```
Error from server (Forbidden): pods is forbidden: User "aksdev@contoso.com" cannot create resource "pods" in API group "" in the namespace "sre"
```

### Test the SRE access to the AKS cluster resources

To confirm that our Azure AD group membership and Kubernetes RBAC work correctly between different users and groups, try the previous commands when signed in as the *opssre* user.

Reset the *kubeconfig* context using the [az aks get-credentials](#) command that clears the previously cached authentication token for the *aksdev* user:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --overwrite-existing
```

Try to schedule and view pods in the assigned *sre* namespace. When prompted, sign in with your own `opssre@contoso.com` credentials created at the start of the article:

```
kubectl run --generator=run-pod/v1 nginx-sre --image=nginx --namespace sre
```

```
kubectl get pods --namespace sre
```

As shown in the following example output, you can successfully create and view the pods:

```
$ kubectl run --generator=run-pod/v1 nginx-sre --image=nginx --namespace sre

To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code BM4RHP3FD
to authenticate.

pod/nginx-sre created

$ kubectl get pods --namespace sre

NAME      READY   STATUS    RESTARTS   AGE
nginx-sre  1/1     Running   0          10s
```

Now, try to view or schedule pods outside of assigned SRE namespace:

```
kubectl get pods --all-namespaces
kubectl run --generator=run-pod/v1 nginx-sre --image=nginx --namespace dev
```

These `kubectl` commands fail, as shown in the following example output. The user's group membership and Kubernetes Role and RoleBindings don't grant permissions to create or manager resources in other namespaces:

```
$ kubectl get pods --all-namespaces
Error from server (Forbidden): pods is forbidden: User "akssre@contoso.com" cannot list pods at the cluster
scope

$ kubectl run --generator=run-pod/v1 nginx-sre --image=nginx --namespace dev
Error from server (Forbidden): pods is forbidden: User "akssre@contoso.com" cannot create pods in the
namespace "dev"
```

## Clean up resources

In this article, you created resources in the AKS cluster and users and groups in Azure AD. To clean up all these resources, run the following commands:

```
# Get the admin kubeconfig context to delete the necessary cluster resources
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster --admin

# Delete the dev and sre namespaces. This also deletes the pods, Roles, and RoleBindings
kubectl delete namespace dev
kubectl delete namespace sre

# Delete the Azure AD user accounts for aksdev and akssre
az ad user delete --upn-or-object-id $AKSDEV_ID
az ad user delete --upn-or-object-id $AKSSRE_ID

# Delete the Azure AD groups for appdev and opssre. This also deletes the Azure role assignments.
az ad group delete --group appdev
az ad group delete --group opssre
```

## Next steps

For more information about how to secure Kubernetes clusters, see [Access and identity options for AKS](#).

For best practices on identity and resource control, see [Best practices for authentication and authorization in AKS](#).

# Rotate certificates in Azure Kubernetes Service (AKS)

2/28/2020 • 3 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) uses certificates for authentication with many of its components. Periodically, you may need to rotate those certificates for security or policy reasons. For example, you may have a policy to rotate all your certificates every 90 days.

This article shows you how to rotate the certificates in your AKS cluster.

## Before you begin

This article requires that you are running the Azure CLI version 2.0.77 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## AKS certificates, Certificate Authorities, and Service Accounts

AKS generates and uses the following certificates, Certificate Authorities, and Service Accounts:

- The AKS API server creates a Certificate Authority (CA) called the Cluster CA.
- The API server has a Cluster CA, which signs certificates for one-way communication from the API server to kubelets.
- Each kubelet also creates a Certificate Signing Request (CSR), which is signed by the Cluster CA, for communication from the kubelet to the API server.
- The etcd key value store has a certificate signed by the Cluster CA for communication from etcd to the API server.
- The etcd key value store creates a CA that signs certificates to authenticate and authorize data replication between etcd replicas in the AKS cluster.
- The API aggregator uses the Cluster CA to issue certificates for communication with other APIs, such as Open Service Broker for Azure. The API aggregator can also have its own CA for issuing those certificates, but it currently uses the Cluster CA.
- Each node uses a Service Account (SA) token, which is signed by the Cluster CA.
- The `kubectl` client has a certificate for communicating with the AKS cluster.

### NOTE

AKS clusters created prior to March 2019 have certificates that expire after two years. Any cluster created after March 2019 or any cluster that has its certificates rotated have certificates that expire after 30 years. To verify when your cluster was created, use `kubectl get nodes` to see the *Age* of your node pools.

Additionally, you can check the expiration date of your cluster's certificate. For example, the following command displays the certificate details for the *myAKSCluster* cluster.

```
kubectl config view --raw -o jsonpath=".clusters[?(@.name == 'myAKSCluster')].cluster.certificate-authority-data" | base64 -d > my-cert.crt  
openssl x509 -in my-cert.crt -text
```

## Rotate your cluster certificates

## WARNING

Rotating your certificates using `az aks rotate-certs` can cause up to 30 minutes of downtime for your AKS cluster.

Use `az aks get-credentials` to sign in to your AKS cluster. This command also downloads and configures the `kubectl` client certificate on your local machine.

```
az aks get-credentials -g $RESOURCE_GROUP_NAME -n $CLUSTER_NAME
```

Use `az aks rotate-certs` to rotate all certificates, CAs, and SAs on your cluster.

```
az aks rotate-certs -g $RESOURCE_GROUP_NAME -n $CLUSTER_NAME
```

## IMPORTANT

It may take up to 30 minutes for `az aks rotate-certs` to complete. If the command fails before completing, use `az aks show` to verify the status of the cluster is *Certificate Rotating*. If the cluster is in a failed state, rerun `az aks rotate-certs` to rotate your certificates again.

Verify that the old certificates are no longer valid by running a `kubectl` command. Since you have not updated the certificates used by `kubectl`, you will see an error. For example:

```
$ kubectl get no
Unable to connect to the server: x509: certificate signed by unknown authority (possibly because of
"crypto/rsa: verification error" while trying to verify candidate authority certificate "ca")
```

Update the certificate used by `kubectl` by running `az aks get-credentials`.

```
az aks get-credentials -g $RESOURCE_GROUP_NAME -n $CLUSTER_NAME --overwrite-existing
```

Verify the certificates have been updated by running a `kubectl` command, which will now succeed. For example:

```
kubectl get no
```

## NOTE

If you have any services that run on top of AKS, such as [Azure Dev Spaces](#), you may need to [update certificates related to those services](#) as well.

## Next steps

This article showed you how to automatically rotate your cluster's certificates, CAs, and SAs. You can see [Best practices for cluster security and upgrades in Azure Kubernetes Service \(AKS\)](#) for more information on AKS security best practices.

# Create a private Azure Kubernetes Service cluster (preview)

2/26/2020 • 5 minutes to read • [Edit Online](#)

In a private cluster, the control plane or API server has internal IP addresses that are defined in the [RFC1918 - Address Allocation for Private Internets](#) document. By using a private cluster, you can ensure that network traffic between your API server and your node pools remains on the private network only.

The control plane or API server is in an Azure Kubernetes Service (AKS)-managed Azure subscription. A customer's cluster or node pool is in the customer's subscription. The server and the cluster or node pool can communicate with each other through the [Azure Private Link service](#) in the API server virtual network and a private endpoint that's exposed in the subnet of the customer's AKS cluster.

## IMPORTANT

AKS preview features are self-service and are offered on an opt-in basis. Previews are provided *as is* and *as available* and are excluded from the service-level agreement (SLA) and limited warranty. AKS previews are partially covered by customer support on a *best effort* basis. Therefore, the features aren't meant for production use. For more information, see the following support articles:

- [AKS Support Policies](#)
- [Azure Support FAQ](#)

## Prerequisites

- The Azure CLI version 2.0.77 or later, and the Azure CLI AKS Preview extension version 0.4.18

## Currently supported regions

- Australia East
- Australia Southeast
- Brazil South
- Canada Central
- Canada East
- Central US
- East Asia
- East US
- East US 2
- East US 2 EUAP
- France Central
- Germany North
- Japan East
- Japan West
- Korea Central
- Korea South
- North Central US
- North Europe

- North Europe
- South Central US
- UK South
- West Europe
- West US
- West US 2
- East US 2

## Currently Supported Availability Zones

- Central US
- East US
- East US 2
- France Central
- Japan East
- North Europe
- Southeast Asia
- UK South
- West Europe
- West US 2

## Install the latest Azure CLI AKS Preview extension

To use private clusters, you need the Azure CLI AKS Preview extension version 0.4.18 or later. Install the Azure CLI AKS Preview extension by using the [az extension add](#) command, and then check for any available updates by using the following [az extension update](#) command:

```
# Install the aks-preview extension
az extension add --name aks-preview

# Update the extension to make sure you have the latest version installed
az extension update --name aks-preview
```

**Caution**

When you register a feature on a subscription, you can't currently un-register that feature. After you enable some preview features, you can use default settings for all AKS clusters that were created in the subscription. Don't enable preview features on production subscriptions. Use a separate subscription to test preview features and gather feedback.

```
az feature register --name AKSPrivateLinkPreview --namespace Microsoft.ContainerService
```

It might take several minutes for the registration status to show as *Registered*. You can check on the status by using the following [az feature list](#) command:

```
az feature list -o table --query "[?contains(name, 'Microsoft.ContainerService/AKSPrivateLinkPreview')].
{Name:name,State:properties.state}"
```

When the state is registered, refresh the registration of the *Microsoft.ContainerService* resource provider by using the following [az provider register](#) command:

```
az provider register --namespace Microsoft.ContainerService  
az provider register --namespace Microsoft.Network
```

## Create a private AKS cluster

### Default basic networking

```
az aks create -n <private-cluster-name> -g <private-cluster-resource-group> --load-balancer-sku standard --enable-private-cluster
```

Where `--enable-private-cluster` is a mandatory flag for a private cluster.

### Advanced networking

```
az aks create \  
    --resource-group <private-cluster-resource-group> \  
    --name <private-cluster-name> \  
    --load-balancer-sku standard \  
    --enable-private-cluster \  
    --network-plugin azure \  
    --vnet-subnet-id <subnet-id> \  
    --docker-bridge-address 172.17.0.1/16 \  
    --dns-service-ip 10.2.0.10 \  
    --service-cidr 10.2.0.0/24
```

Where `--enable-private-cluster` is a mandatory flag for a private cluster.

#### NOTE

If the Docker bridge address CIDR (172.17.0.1/16) clashes with the subnet CIDR, change the Docker bridge address appropriately.

## Connect to the private cluster

The API server endpoint has no public IP address. Consequently, you must create an Azure virtual machine (VM) in a virtual network and connect to the API server. To do so, do the following:

1. Get credentials to connect to the cluster.

```
az aks get-credentials --name MyManagedCluster --resource-group MyResourceGroup
```

2. Do either of the following:

- Create a VM in the same virtual network as the AKS cluster.
- Create a VM in a different virtual network, and peer this virtual network with the AKS cluster virtual network.

If you create a VM in a different virtual network, set up a link between this virtual network and the private DNS zone. To do so:

- a. Go to the MC\_\* resource group in the Azure portal.
- b. Select the private DNS zone.
- c. In the left pane, select the **Virtual network** link.
- d. Create a new link to add the virtual network of the VM to the private DNS zone. It takes a few

- minutes for the DNS zone link to become available.
- e. Go back to the MC\_\* resource group in the Azure portal.
  - f. In the right pane, select the virtual network. The virtual network name is in the form *aks-vnet-\**.
  - g. In the left pane, select **Peerings**.
  - h. Select **Add**, add the virtual network of the VM, and then create the peering.
  - i. Go to the virtual network where you have the VM, select **Peerings**, select the AKS virtual network, and then create the peering. If the address ranges on the AKS virtual network and the VM's virtual network clash, peering fails. For more information, see [Virtual network peering](#).
3. Access the VM via Secure Shell (SSH).
  4. Install the Kubectl tool, and run the Kubectl commands.

## Dependencies

- The Private Link service is supported on Standard Azure Load Balancer only. Basic Azure Load Balancer isn't supported.
- To use a custom DNS server, deploy an AD server with DNS to forward to this IP 168.63.129.16

## Limitations

- IP authorized ranges cannot be applied to the private api server endpoint, they only apply to the public API server
- Availability Zones are currently supported for certain regions, see the beginning of this document
- [Azure Private Link service limitations](#) apply to private clusters, Azure private endpoints, and virtual network service endpoints, which aren't currently supported in the same virtual network.
- No support for virtual nodes in a private cluster to spin private Azure Container Instances (ACI) in a private Azure virtual network
- No support for Azure DevOps integration out of the box with private clusters
- For customers that need to enable Azure Container Registry to work with private AKS, the Container Registry virtual network must be peered with the agent cluster virtual network.
- No current support for Azure Dev Spaces
- No support for converting existing AKS clusters into private clusters
- Deleting or modifying the private endpoint in the customer subnet will cause the cluster to stop functioning.
- Azure Monitor for containers Live Data isn't currently supported.
- *Bring your own DNS* isn't currently supported.

# Bring your own keys (BYOK) with Azure disks in Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

Azure Storage encrypts all data in a storage account at rest. By default, data is encrypted with Microsoft-managed keys. For additional control over encryption keys, you can supply [customer-managed keys](#) to use for encryption at rest for both the OS and data disks for your AKS clusters.

## NOTE

BYOK Linux and Windows based AKS clusters are available in [Azure regions](#) that support server side encryption of Azure managed disks.

## Before you begin

- This article assumes that you are creating a *new AKS cluster*.
- You must enable soft delete and purge protection for *Azure Key Vault* when using Key Vault to encrypt managed disks.
- You need the Azure CLI version 2.0.79 or later and the `aks-preview` 0.4.26 extension

## IMPORTANT

AKS preview features are self-service opt-in. Previews are provided "as-is" and "as available" and are excluded from the service level agreements and limited warranty. AKS Previews are partially covered by customer support on best effort basis. As such, these features are not meant for production use. For additional information, please see the following support articles:

- [AKS Support Policies](#)
- [Azure Support FAQ](#)

## Install latest AKS CLI preview extension

To use customer-managed keys, you need the `aks-preview` CLI extension version 0.4.26 or higher. Install the `aks-preview` Azure CLI extension using the [az extension add](#) command, then check for any available updates using the [az extension update](#) command:

```
# Install the aks-preview extension
az extension add --name aks-preview

# Update the extension to make sure you have the latest version installed
az extension update --name aks-preview
```

## Create an Azure Key Vault instance

Use an Azure Key Vault instance to store your keys. You can optionally use the Azure portal to [Configure customer-managed keys with Azure Key Vault](#)

Create a new *resource group*, then create a new *Key Vault* instance and enable soft delete and purge protection.

Ensure you use the same region and resource group names for each command.

```
# Optionally retrieve Azure region short names for use on upcoming commands  
az account list-locations
```

```
# Create new resource group in a supported Azure region  
az group create -l myAzureRegionName -n myResourceGroup
```

```
# Create an Azure Key Vault resource in a supported Azure region  
az keyvault create -n myKeyVaultName -g myResourceGroup -l myAzureRegionName --enable-purge-protection true --enable-soft-delete true
```

## Create an instance of a DiskEncryptionSet

Replace *myKeyVaultName* with the name of your key vault. You will also need a *key* stored in Azure Key Vault to complete the following steps. Either store your existing Key in the Key Vault you created on the previous steps, or [generate a new key](#) and replace *myKeyName* below with the name of your key.

```
# Retrieve the Key Vault Id and store it in a variable  
keyVaultId=$(az keyvault show --name myKeyVaultName --query [id] -o tsv)  
  
# Retrieve the Key Vault key URL and store it in a variable  
keyVaultKeyUrl=$(az keyvault key show --vault-name myKeyVaultName --name myKeyName --query [key.kid] -o tsv)  
  
# Create a DiskEncryptionSet  
az disk-encryption-set create -n myDiskEncryptionSetName -l myAzureRegionName -g myResourceGroup --source-vault $keyVaultId --key-url $keyVaultKeyUrl
```

## Grant the DiskEncryptionSet access to key vault

Use the DiskEncryptionSet and resource groups you created on the prior steps, and grant the DiskEncryptionSet resource access to the Azure Key Vault.

```
# Retrieve the DiskEncryptionSet value and set a variable  
desIdentity=$(az disk-encryption-set show -n myDiskEncryptionSetName -g myResourceGroup --query [identity.principalId] -o tsv)  
  
# Update security policy settings  
az keyvault set-policy -n myKeyVaultName -g myResourceGroup --object-id $desIdentity --key-permissions wrapkey unwrapkey get  
  
# Assign the reader role  
az role assignment create --assignee $desIdentity --role Reader --scope $keyVaultId
```

## Create a new AKS cluster and encrypt the OS disk

Create a **new resource group** and AKS cluster, then use your key to encrypt the OS disk. Customer-managed keys are only supported in Kubernetes versions greater than 1.17.

### IMPORTANT

Ensure you create a new resoruce group for your AKS cluster

```

# Retrieve the DiskEncryptionSet value and set a variable
diskEncryptionSetId=$(az resource show -n myDiskEncryptionSetName -g myResourceGroup --resource-type
"Microsoft.Compute/diskEncryptionSets" --query [id] -o tsv)

# Create a resource group for the AKS cluster
az group create -n myResourceGroup -l myAzureRegionName

# Create the AKS cluster
az aks create -n myAKSCluster -g myResourceGroup --node-osdisk-diskencryptionset-id $diskEncryptionSetId --
kubernetes-version 1.17.0 --generate-ssh-keys

```

When new node pools are added to the cluster created above, the customer-managed key provided during the create is used to encrypt the OS disk.

## Encrypt your AKS cluster data disk

You can also encrypt the AKS data disks with your own keys.

### IMPORTANT

Ensure you have the proper AKS credentials. The Service principal will need to have contributor access to the resource group where the diskencryptionset is deployed. Otherwise, you will get an error suggesting that the service principal does not have permissions.

```

# Retrieve your Azure Subscription Id from id property as shown below
az account list

```

```

someuser@Azure:~$ az account list
[
  {
    "cloudName": "AzureCloud",
    "id": "666e66d8-1e43-4136-be25-f25bb5de5893",
    "isDefault": true,
    "name": "MyAzureSubscription",
    "state": "Enabled",
    "tenantId": "3ebcdf90-2069-4529-a1ab-7bdcb24df7cd",
    "user": {
      "cloudShellID": true,
      "name": "someuser@azure.com",
      "type": "user"
    }
  }
]

```

Create a file called **byok-azure-disk.yaml** that contains the following information. Replace myAzureSubscriptionId, myResourceGroup, and myDiskEncryptionSetName with your values, and apply the yaml. Make sure to use the resource group where your DiskEncryptionSet is deployed. If you use the Azure Cloud Shell, this file can be created using vi or nano as if working on a virtual or physical system:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: hdd
provisioner: kubernetes.io/azure-disk
parameters:
  skuname: Standard_LRS
  kind: managed
  diskEncryptionSetID:
    "/subscriptions/{myAzureSubscriptionId}/resourceGroups/{myResourceGroup}/providers/Microsoft.Compute/diskEncryptionSets/{myDiskEncryptionSetName}"
```

Next, run this deployment in your AKS cluster:

```
# Get credentials
az aks get-credentials --name myAksCluster --resource-group myResourceGroup --output table

# Update cluster
kubectl apply -f byok-azure-disk.yaml
```

## Limitations

- BYOK is only currently available in GA and Preview in certain [Azure regions](#)
- OS Disk Encryption supported with Kubernetes version 1.17 and above
- Available only in regions where BYOK is supported
- Encryption with customer-managed keys currently is for new AKS clusters only, existing clusters cannot be upgraded
- AKS cluster using Virtual Machine Scale Sets are required, no support for Virtual Machine Availability Sets

## Next steps

Review [best practices for AKS cluster security](#)

# Azure Monitor for containers overview

2/13/2020 • 3 minutes to read • [Edit Online](#)

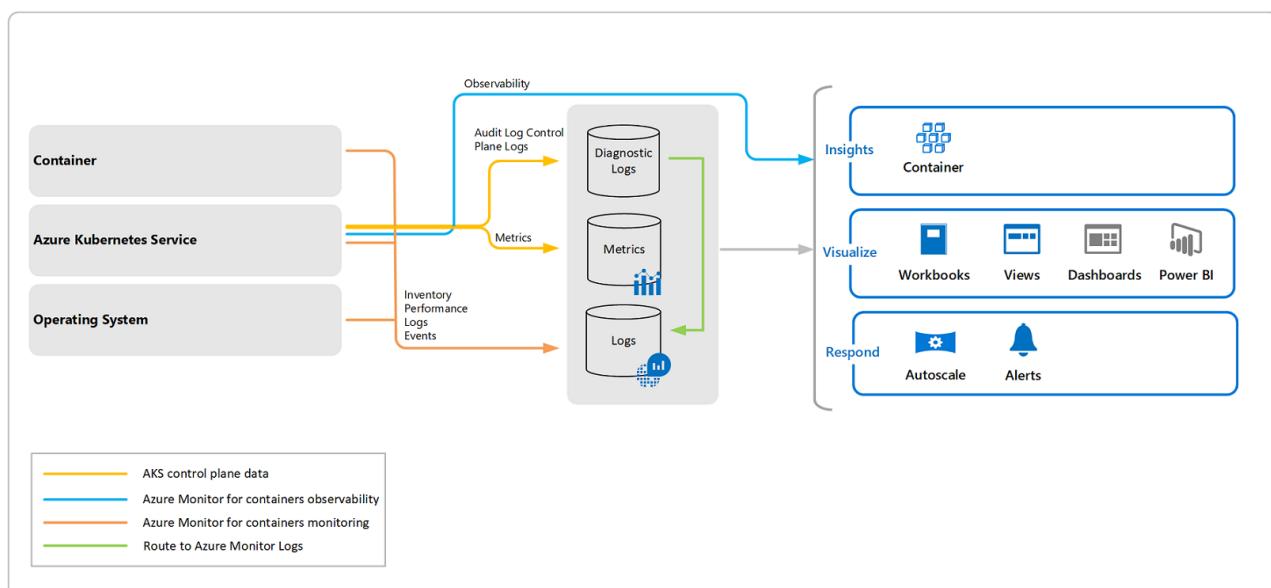
Azure Monitor for containers is a feature designed to monitor the performance of container workloads deployed to:

- Managed Kubernetes clusters hosted on [Azure Kubernetes Service \(AKS\)](#)
- Self-managed Kubernetes clusters hosted on Azure using [AKS Engine](#)
- [Azure Container Instances](#)
- Self-managed Kubernetes clusters hosted on [Azure Stack](#) or on-premises
- [Azure Red Hat OpenShift](#)

Azure Monitor for containers supports clusters running the Linux and Windows Server 2019 operating system.

Monitoring your containers is critical, especially when you're running a production cluster, at scale, with multiple applications.

Azure Monitor for containers gives you performance visibility by collecting memory and processor metrics from controllers, nodes, and containers that are available in Kubernetes through the Metrics API. Container logs are also collected. After you enable monitoring from Kubernetes clusters, metrics and logs are automatically collected for you through a containerized version of the Log Analytics agent for Linux. Metrics are written to the metrics store and log data is written to the logs store associated with your [Log Analytics](#) workspace.



## What does Azure Monitor for containers provide?

Azure Monitor for containers delivers a comprehensive monitoring experience using different features of Azure Monitor. These features enable you to understand the performance and health of your Kubernetes cluster running Linux and Windows Server 2019 operating system, and the container workloads. With Azure Monitor for containers you can:

- Identify AKS containers that are running on the node and their average processor and memory utilization. This knowledge can help you identify resource bottlenecks.
- Identify processor and memory utilization of container groups and their containers hosted in Azure Container Instances.

- Identify where the container resides in a controller or a pod. This knowledge can help you view the controller's or pod's overall performance.
- Review the resource utilization of workloads running on the host that are unrelated to the standard processes that support the pod.
- Understand the behavior of the cluster under average and heaviest loads. This knowledge can help you identify capacity needs and determine the maximum load that the cluster can sustain.
- Configure alerts to proactively notify you or record it when CPU and memory utilization on nodes or containers exceed your thresholds, or when a health state change occurs in the cluster at the infrastructure or nodes health rollup.
- Integrate with [Prometheus](#) to view application and workload metrics it collects from nodes and Kubernetes using [queries](#) to create custom alerts, dashboards, and detailed perform detailed analysis.
- Monitor container workloads [deployed to AKS Engine on-premises](#) and [AKS Engine on Azure Stack](#).
- Monitor container workloads [deployed to Azure Red Hat OpenShift](#).

**NOTE**

Support for Azure Red Hat OpenShift is a feature in public preview at this time.

The main differences in monitoring a Windows Server cluster compared to a Linux cluster are the following:

- Memory RSS metric isn't available for Windows node and containers.
- Disk storage capacity information isn't available for Windows nodes.
- Container logs aren't available for containers running in Windows nodes.
- Live Data (preview) feature support is available with the exception of Windows container logs.
- Only pod environments are monitored, not Docker environments.
- With the preview release, a maximum of 30 Windows Server containers are supported. This limitation doesn't apply to Linux containers.

Check out the following video providing an intermediate level deep dive to help you learn about monitoring your AKS cluster with Azure Monitor for containers.

## How do I access this feature?

You can access Azure Monitor for containers two ways, from Azure Monitor or directly from the selected AKS cluster. From Azure Monitor, you have a global perspective of all the containers deployed, which are monitored and which are not, allowing you to search and filter across your subscriptions and resource groups, and then drill into Azure Monitor for containers from the selected container. Otherwise, you can access the feature directly from a selected AKS container from the AKS page.

The screenshot shows the Azure Monitor - Containers interface. On the left, there's a sidebar with navigation links like Overview, Activity log, Alerts, Metrics, Logs, Service Health, and Workbooks (preview). The main area displays a Cluster Status Summary with metrics: Total 6, Critical 0, Warning 2, Unknown 1, AKS Healthy 2, AKS-engine Healthy 0, and Non-monitored 1. Below this, there are sections for Monitored clusters (5) and Non-monitored clusters (1). The right side shows a detailed view for a monitored cluster named ContosoSH360Win - Insights, which is a Kubernetes service. It includes tabs for Overview, Activity log, Access control (IAM), Tags, Settings (Node pools (preview), Upgrade, Scale, Dev Spaces, Deployment center (preview), Policies (preview), Properties, Locks, Export template), Monitoring, and Insights. The Insights tab shows a table of nodes with metrics like CPU Usage (millicores) over a last 6-hour time range. The table includes columns for NAME, STATUS, 95TH %, CONTAINERS, UPTIME, CONTROLLER, and TREND 95TH % (1 BAR = 15M). The data for the first few nodes is as follows:

NAME	STATUS	95TH %	CONTAINERS	UPTIME	CONTROLLER	TREND 95TH % (1 BAR = 15M)
aks-nodepool1-55...	Ok	6%	126 mc	16	19 hours	-
aks-linuxpool2-55...	Ok	3%	67 mc	6	1 hour	-
aks-linuxpool2-55...	Ok	3%	65 mc	6	1 hour	-
aksnwpwin000001	Ok	-	-	1	-	-
aksnwpwin000000	Ok	-	-	10	-	-

If you are interested in monitoring and managing your Docker and Windows container hosts running outside of AKS to view configuration, audit, and resource utilization, see the [Container Monitoring solution](#).

## Next steps

To begin monitoring your Kubernetes cluster, review [How to enable the Azure Monitor for containers](#) to understand the requirements and available methods to enable monitoring.

# Enable and review Kubernetes master node logs in Azure Kubernetes Service (AKS)

2/25/2020 • 4 minutes to read • [Edit Online](#)

With Azure Kubernetes Service (AKS), the master components such as the *kube-apiserver* and *kube-controller-manager* are provided as a managed service. You create and manage the nodes that run the *kubelet* and container runtime, and deploy your applications through the managed Kubernetes API server. To help troubleshoot your application and services, you may need to view the logs generated by these master components. This article shows you how to use Azure Monitor logs to enable and query the logs from the Kubernetes master components.

## Before you begin

This article requires an existing AKS cluster running in your Azure account. If you do not already have an AKS cluster, create one using the [Azure CLI](#) or [Azure portal](#). Azure Monitor logs works with both RBAC and non-RBAC enabled AKS clusters.

## Enable diagnostics logs

To help collect and review data from multiple sources, Azure Monitor logs provides a query language and analytics engine that provides insights to your environment. A workspace is used to collate and analyze the data, and can integrate with other Azure services such as Application Insights and Security Center. To use a different platform to analyze the logs, you can instead choose to send diagnostic logs to an Azure storage account or event hub. For more information, see [What is Azure Monitor logs?](#).

Azure Monitor logs are enabled and managed in the Azure portal. To enable log collection for the Kubernetes master components in your AKS cluster, open the Azure portal in a web browser and complete the following steps:

1. Select the resource group for your AKS cluster, such as *myResourceGroup*. Don't select the resource group that contains your individual AKS cluster resources, such as *MC\_myResourceGroup\_myAKSCluster\_eastus*.
2. On the left-hand side, choose **Diagnostic settings**.
3. Select your AKS cluster, such as *myAKSCluster*, then choose to **Add diagnostic setting**.
4. Enter a name, such as *myAKSClusterLogs*, then select the option to **Send to Log Analytics**.
5. Select an existing workspace or create a new one. If you create a workspace, provide a workspace name, a resource group, and a location.
6. In the list of available logs, select the logs you wish to enable. Common logs include the *kube-apiserver*, *kube-controller-manager*, and *kube-scheduler*. You can enable additional logs, such as *kube-audit* and *cluster-autoscaler*. You can return and change the collected logs once Log Analytics workspaces are enabled.
7. When ready, select **Save** to enable collection of the selected logs.

The following example portal screenshot shows the *Diagnostics settings* window:

## Diagnostics settings

Save  Discard  Delete

\* Name

myAKSClusterLogs 

Archive to a storage account

Stream to an event hub

Send to Log Analytics

Subscription

Azure

Log Analytics Workspace

DefaultWorkspace-8fa5cd83-7fbb-431a-af16-4a20dede8802-EUS ( eastus )

LOG

kube-apiserver

kube-controller-manager

kube-scheduler

kube-audit

cluster-autoscaler

METRIC

AllMetrics

## Schedule a test pod on the AKS cluster

To generate some logs, create a new pod in your AKS cluster. The following example YAML manifest can be used to create a basic NGINX instance. Create a file named `nginx.yaml` in an editor of your choice and paste the following content:

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
    - name: mypod
      image: nginx:1.15.5
      resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 250m
          memory: 256Mi
      ports:
        - containerPort: 80

```

Create the pod with the [kubectl create](#) command and specify your YAML file, as shown in the following example:

```

$ kubectl create -f nginx.yaml

pod/nginx created

```

## View collected logs

It may take a few minutes for the diagnostics logs to be enabled and appear in the Log Analytics workspace. In the Azure portal, select the resource group for your Log Analytics workspace, such as *myResourceGroup*, then choose your log analytics resource, such as *myAKSLogs*.

<input type="checkbox"/> NAME ↗	<input type="checkbox"/> TYPE ↗	<input type="checkbox"/> LOCATION ↗	<input type="checkbox"/> ...
<input type="checkbox"/> myAKSCluster	Kubernetes service	East US	<input type="checkbox"/>
<input type="checkbox"/> myAKSLogs	Log Analytics	East US	<input type="checkbox"/>

On the left-hand side, choose **Logs**. To view the *kube-apiserver*, enter the following query in the text box:

```

AzureDiagnostics
| where Category == "kube-apiserver"
| project log_s

```

Many logs are likely returned for the API server. To scope down the query to view the logs about the NGINX pod created in the previous step, add an additional *where* statement to search for *pods/nginx* as shown in the following example query:

```

AzureDiagnostics
| where Category == "kube-apiserver"
| where log_s contains "pods/nginx"
| project log_s

```

The specific logs for your NGINX pod are displayed, as shown in the following example screenshot:

The screenshot shows the Azure Log Analytics interface. On the left, there's a sidebar with navigation links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings (Locks, Automation script, Advanced settings), General (Quick Start, Workspace summary, View Designer, Logs), Solutions, Saved searches, Pricing tier, Usage and estimated costs, Properties, and Service Map. Below these are sections for Workspace Data Sources and Manual machine. The main area is titled "DefaultWorkspace-19da35d3-9a1a-4f3b-9b9c-3c56ef409565-EUS - Logs". It has a search bar, a "New Query 1" button, and a "Run" button. A time range selector shows "Last 24 hours". To the right of the search bar are Save, Copy Link, Export, Set alert, and Pin buttons. The main content area displays a query results table with columns for TABLE, CHART, and Columns. The table shows completed log entries from the last 24 hours. The first few rows of the log table are as follows:

log_s
> [1025 22:47:33.141090 1 wrnp.go:42] GET /api/v1/namespaces/default/pods/nginx: {3.024572ms} 200 [[kubectl/v1.9.11 (linux/amd64) kubernetes/1bfeeb6] 172.31...
> [1025 22:47:33.152601 1 wrnp.go:42] PUT /api/v1/namespaces/default/pods/nginx/status: {10.212304ms} 200 [[kubectl/v1.9.11 (linux/amd64) kubernetes/1bfeeb6] ...]
> [1025 22:47:34.198246 1 wrnp.go:42] GET /api/v1/namespaces/default/pods/nginx: {3.799964ms} 200 [[kubectl/v1.12.1 (linux/amd64) kubernetes/4ed3216] 172.31...
> [1025 22:47:34.399200 1 wrnp.go:42] GET /api/v1/namespaces/default/pods/nginx: {3.615766ms} 200 [[kubectl/v1.12.1 (linux/amd64) kubernetes/4ed3216] 172.31...
> [1025 22:47:21.014045 1 wrnp.go:42] POST /api/v1/namespaces/default/pods/nginx/binding: {6.561339ms} 201 [[hyperkube/v1.9.11 (linux/amd64) kubernetes/1bf6...
> [1025 22:47:21.034467 1 wrnp.go:42] GET /api/v1/namespaces/default/pods/nginx: {3.31199ms} 200 [[kubectl/v1.9.11 (linux/amd64) kubernetes/1bfeeb6] 172.31...
> [1025 22:47:21.046264 1 wrnp.go:42] PUT /api/v1/namespaces/default/pods/nginx/status: {10.627901ms} 200 [[kubectl/v1.9.11 (linux/amd64) kubernetes/1bfeeb6] ...]

To view additional logs, you can update the query for the *Category* name to *kube-controller-manager* or *kube-scheduler*, depending on what additional logs you enable. Additional *where* statements can then be used to refine the events you are looking for.

For more information on how to query and filter your log data, see [View or analyze data collected with log analytics log search](#).

## Log event schema

To help analyze the log data, the following table details the schema used for each event:

FIELD NAME	DESCRIPTION
<i>resourceId</i>	Azure resource that produced the log
<i>time</i>	Timestamp of when the log was uploaded
<i>category</i>	Name of container/component generating the log
<i>operationName</i>	Always <i>Microsoft.ContainerService/managedClusters/diagnosticLogs/Read</i>
<i>properties.log</i>	Full text of the log from the component
<i>properties.stream</i>	<i>stderr</i> or <i>stdout</i>
<i>properties.pod</i>	Pod name that the log came from
<i>properties.containerID</i>	ID of the docker container this log came from

## Log Roles

ROLE	DESCRIPTION
<i>aksService</i>	The display name in audit log for the control plane operation (from the hcpService)
<i>masterclient</i>	The display name in audit log for MasterClientCertificate, the certificate you get from az aks get-credentials
<i>nodeclient</i>	The display name for ClientCertificate, which is used by agent nodes

## Next steps

In this article, you learned how to enable and review the logs for the Kubernetes master components in your AKS cluster. To monitor and troubleshoot further, you can also [view the Kubelet logs](#) and [enable SSH node access](#).

# Get kubelet logs from Azure Kubernetes Service (AKS) cluster nodes

2/25/2020 • 2 minutes to read • [Edit Online](#)

As part of operating an AKS cluster, you may need to review logs to troubleshoot a problem. Built-in to the Azure portal is the ability to view logs for the [AKS master components](#) or [containers in an AKS cluster](#). Occasionally, you may need to get *kubelet* logs from an AKS node for troubleshooting purposes.

This article shows you how you can use `journalctl` to view the *kubelet* logs on an AKS node.

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart [using the Azure CLI](#) or [using the Azure portal](#).

## Create an SSH connection

First, create an SSH connection with the node on which you need to view *kubelet* logs. This operation is detailed in the [SSH into Azure Kubernetes Service \(AKS\) cluster nodes](#) document.

## Get kubelet logs

Once you have connected to the node, run the following command to pull the *kubelet* logs:

```
sudo journalctl -u kubelet -o cat
```

The following sample output shows the *kubelet* log data:

I0508 12:26:17.905042	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:26:27.943494	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:26:28.920125	8672 server.go:796] GET /stats/summary: (10.370874ms) 200 [[Ruby] 10.244.0.2:52292]
I0508 12:26:37.964650	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:26:47.996449	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:26:58.019746	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:27:05.107680	8672 server.go:796] GET /stats/summary/: (24.853838ms) 200 [[Go-http-client/1.1] 10.244.0.3:44660]
I0508 12:27:08.041736	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:27:18.068505	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:27:28.094889	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:27:38.121346	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:27:44.015205	8672 server.go:796] GET /stats/summary: (30.236824ms) 200 [[Ruby] 10.244.0.2:52588]
I0508 12:27:48.145640	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:27:58.178534	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:28:05.040375	8672 server.go:796] GET /stats/summary/: (27.78503ms) 200 [[Go-http-client/1.1] 10.244.0.3:44660]
I0508 12:28:08.214158	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:28:18.242160	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:28:28.274408	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:28:38.296074	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:28:48.321952	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"
I0508 12:28:58.344656	8672 kubelet_node_status.go:497] Using Node Hostname from cloudprovider: "aks-agentpool-11482510-0"

## Next steps

If you need additional troubleshooting information from the Kubernetes master, see [view Kubernetes master node logs in AKS](#).

# How to view Kubernetes logs, events, and pod metrics in real-time

12/13/2019 • 6 minutes to read • [Edit Online](#)

Azure Monitor for containers includes the Live Data (preview) feature, which is an advanced diagnostic feature allowing you direct access to your Azure Kubernetes Service (AKS) container logs (stdout/stderror), events, and pod metrics. It exposes direct access to `kubectl logs -c`, `kubectl get` events, and `kubectl top pods`. A console pane shows the logs, events, and metrics generated by the container engine to further assist in troubleshooting issues in real-time.

This article provides a detailed overview and helps you understand how to use this feature.

## NOTE

AKS clusters enabled as [private clusters](#) are not supported with this feature. This feature relies on directly accessing the Kubernetes API through a proxy server from your browser. Enabling networking security to block the Kubernetes API from this proxy will block this traffic.

## NOTE

This feature is available in all Azure regions, including Azure China. It is currently not available in Azure US Government.

For help setting up or troubleshooting the Live Data (preview) feature, review our [setup guide](#). This feature directly access the Kubernetes API, and additional information about the authentication model can be found [here](#).

## Live Data (preview) functionality overview

### Search

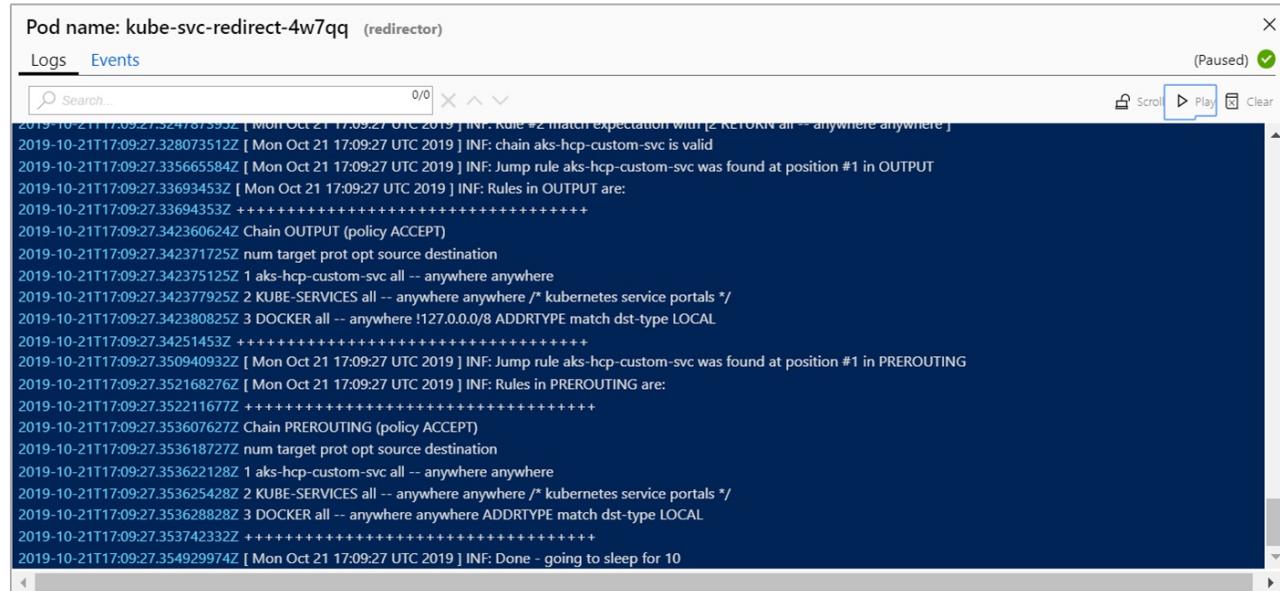
The screenshot shows a search interface for a specific pod named "kube-svc-redirect-t8b91". The search bar at the top contains the text "destination". Below the search bar, there are tabs for "Logs" and "Events", with "Logs" being the active tab. To the right of the search bar, there is a status message "(No New Data)" with a green checkmark. At the bottom of the search bar area, there are buttons for "Scroll", "Pause", and "Clear". The main content area displays log entries in a dark-themed terminal window. The logs show various system messages related to network policies and rules, such as DNAT and RETURN statements, and chain definitions. Some log entries are highlighted in yellow, indicating they match the search term "destination".

```
Pod name: kube-svc-redirect-t8b91 (redirector)
Logs Events (No New Data) ✓
destination 1/16 X ^ v
2019-10-21T16:54:06.137209651Z target prot opt source destination
2019-10-21T16:54:06.137214751Z DNAT tcp -- anywhere 10.0.0.1 to:127.0.0.1:14612
2019-10-21T16:54:06.137218452Z RETURN all -- anywhere anywhere
2019-10-21T16:54:06.141669262Z [ Mon Oct 21 16:54:06 UTC 2019 ] INF: found expected rule count in chain:aks-hcp-custom-svc
2019-10-21T16:54:06.141682766Z [ Mon Oct 21 16:54:06 UTC 2019 ] INF: Will validate the following rules:
2019-10-21T16:54:06.141687062Z ++++++
2019-10-21T16:54:06.145364646Z Chain aks-hcp-custom-svc (2 references)
2019-10-21T16:54:06.14535986Z num target prot opt source destination
2019-10-21T16:54:06.145364162Z 1 DNAT tcp -- anywhere 10.0.0.1 to:127.0.0.1:14612
2019-10-21T16:54:06.145367762Z 2 RETURN all -- anywhere anywhere
2019-10-21T16:54:06.14537116Z ++++++
2019-10-21T16:54:06.150201984Z [ Mon Oct 21 16:54:06 UTC 2019 ] INF: Rule #1 match expectation with [1 DNAT tcp -- anywhere 10.0.0.1 to:127.0.0.1:14612]
2019-10-21T16:54:06.156271639Z [ Mon Oct 21 16:54:06 UTC 2019 ] INF: Rule #2 match expectation with [2 RETURN all -- anywhere anywhere]
2019-10-21T16:54:06.157287965Z [ Mon Oct 21 16:54:06 UTC 2019 ] INF: chain aks-hcp-custom-svc is valid
2019-10-21T16:54:06.16411234Z [ Mon Oct 21 16:54:06 UTC 2019 ] INF: Jump rule aks-hcp-custom-svc was found at position #1 in OUTPUT
2019-10-21T16:54:06.165113766Z [ Mon Oct 21 16:54:06 UTC 2019 ] INF: Rules in OUTPUT are:
2019-10-21T16:54:06.16526637Z ++++++
2019-10-21T16:54:06.168171844Z Chain OUTPUT (policy ACCEPT)
2019-10-21T16:54:06.16818334Z num target prot opt source destination
2019-10-21T16:54:06.168187145Z 1 aks-hcp-custom-svc all -- anywhere anywhere
```

The Live Data (preview) feature includes search functionality. In the **Search** field, you can filter results by typing a key word or term and any matching results are highlighted to allow quick review. While viewing events, you can additionally limit the results using the **Filter** pill found to the right of the search bar. Depending on what resource you have selected, the pill lists a Pod, Namespace, or cluster to chose from.

## Scroll Lock and Pause

To suspend autoscroll and control the behavior of the pane, allowing you to manually scroll through the new data read, you can use the **Scroll** option. To re-enable autoscroll, simply select the **Scroll** option again. You can also pause retrieval of log or event data by selecting the the **Pause** option, and when you are ready to resume, simply select **Play**.



```
Pod name: kube-svc-redirect-4w7qq (redactor)
Logs Events
X (Paused) ✓
Search... 0/0 X ^ v
2019-10-21T17:09:27.324783595Z [ Mon Oct 21 17:09:27 UTC 2019 ] INF: Rule #2 match expectation with [2 RETURN all -- anywhere anywhere ]
2019-10-21T17:09:27.328073512Z [ Mon Oct 21 17:09:27 UTC 2019 ] INF: chain aks-hcp-custom-svc is valid
2019-10-21T17:09:27.335665584Z [ Mon Oct 21 17:09:27 UTC 2019 ] INF: Jump rule aks-hcp-custom-svc was found at position #1 in OUTPUT
2019-10-21T17:09:27.336934532Z [ Mon Oct 21 17:09:27 UTC 2019 ] INF: Rules in OUTPUT are:
2019-10-21T17:09:27.336943532Z ++++++
2019-10-21T17:09:27.342360624Z Chain OUTPUT (policy ACCEPT)
2019-10-21T17:09:27.342371725Z num target prot opt source destination
2019-10-21T17:09:27.342375125Z 1 aks-hcp-custom-svc all -- anywhere anywhere
2019-10-21T17:09:27.342377925Z 2 KUBE-SERVICES all -- anywhere anywhere /* kubernetes service portals */
2019-10-21T17:09:27.342380825Z 3 DOCKER all -- anywhere !127.0.0.0/8 ADDRTYPE match dst-type LOCAL
2019-10-21T17:09:27.342514532Z ++++++
2019-10-21T17:09:27.350940932Z [ Mon Oct 21 17:09:27 UTC 2019 ] INF: Jump rule aks-hcp-custom-svc was found at position #1 in PREROUTING
2019-10-21T17:09:27.352168276Z [ Mon Oct 21 17:09:27 UTC 2019 ] INF: Rules in PREROUTING are:
2019-10-21T17:09:27.352211677Z ++++++
2019-10-21T17:09:27.353607627Z Chain PREROUTING (policy ACCEPT)
2019-10-21T17:09:27.353618727Z num target prot opt source destination
2019-10-21T17:09:27.353622128Z 1 aks-hcp-custom-svc all -- anywhere anywhere
2019-10-21T17:09:27.353625428Z 2 KUBE-SERVICES all -- anywhere anywhere /* kubernetes service portals */
2019-10-21T17:09:27.353628828Z 3 DOCKER all -- anywhere anywhere ADDRTYPE match dst-type LOCAL
2019-10-21T17:09:27.353742332Z ++++++
2019-10-21T17:09:27.354929974Z [ Mon Oct 21 17:09:27 UTC 2019 ] INF: Done - going to sleep for 10
```

### IMPORTANT

We recommend only suspending or pausing autoscroll for a short period of time while troubleshooting an issue. These requests may impact the availability and throttling of the Kubernetes API on your cluster.

### IMPORTANT

No data is stored permanently during operation of this feature. All information captured during the session is deleted when you close your browser or navigate away from it. Data only remains present for visualization inside the five minute window of the metrics feature; any metrics older than five minutes are also deleted. The Live Data (preview) buffer queries within reasonable memory usage limits (need to be more specific here, what is reasonable?).

## View logs

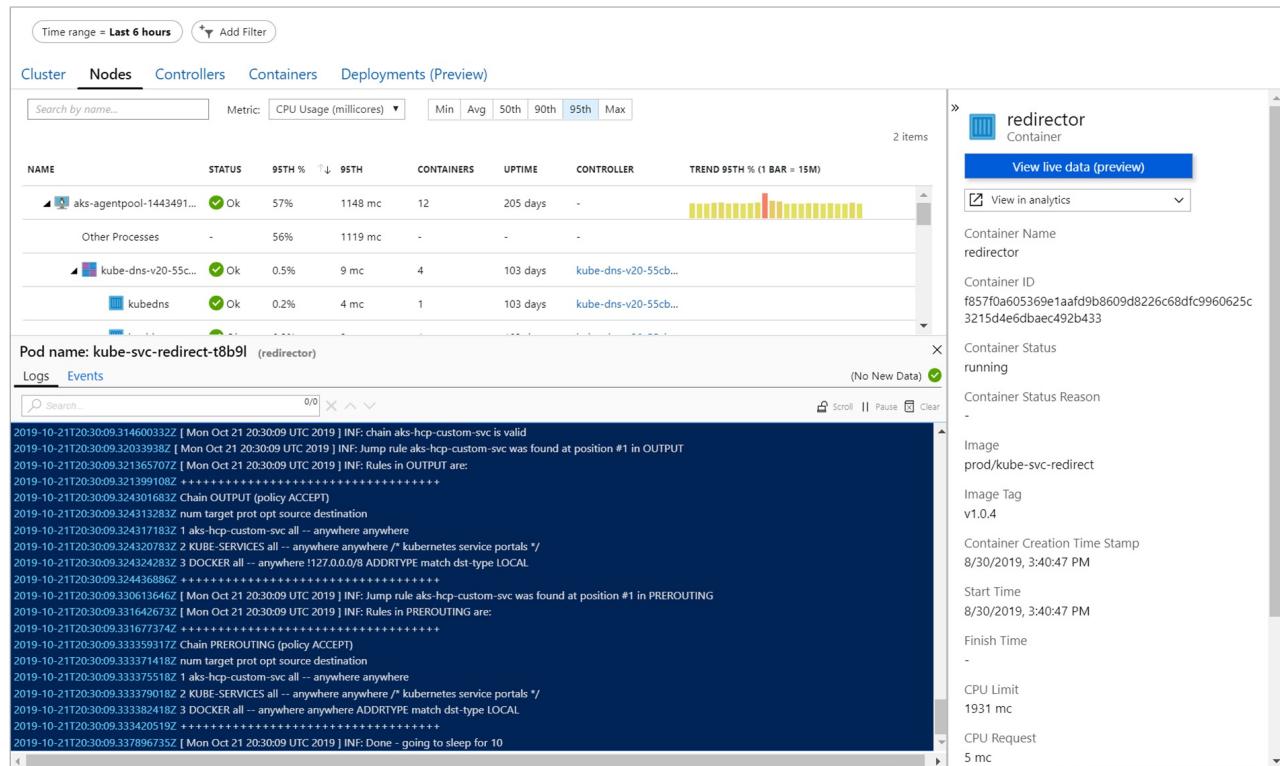
You can view real-time log data as they are generated by the container engine from the **Nodes**, **Controllers**, and **Containers** view. To view log data, perform the following steps.

1. In the Azure portal, browse to the AKS cluster resource group and select your AKS resource.
2. On the AKS cluster dashboard, under **Monitoring** on the left-hand side, choose **Insights**.
3. Select either the **Nodes**, **Controllers**, or **Containers** tab.
4. Select an object from the performance grid, and on the properties pane found on the right side, select **View live data (preview)** option. If the AKS cluster is configured with single sign-on using Azure AD, you are prompted to authenticate on first use during that browser session. Select your account and complete authentication with Azure.

## NOTE

When viewing the data from your Log Analytics workspace by selecting the **View in analytics** option from the properties pane, the log search results will potentially show **Nodes**, **Daemon Sets**, **Replica Sets**, **Jobs**, **Cron Jobs**, **Pods**, and **Containers** which may no longer exist. Attempting to search logs for a container which isn't available in `kubectl` will also fail here. Review the **View in analytics** feature to learn more about viewing historical logs, events and metrics.

After successfully authenticating, the Live Data (preview) console pane will appear below the performance data grid where you can view log data in a continuous stream. If the fetch status indicator shows a green check mark, which is on the far right of the pane, it means data can be retrieved and it begins streaming to your console.



## View events

You can view real-time event data as they are generated by the container engine from the **Nodes**, **Controllers**, **Containers**, and **Deployments (preview)** view when a container, pod, node, ReplicaSet, DaemonSet, job, CronJob or Deployment is selected. To view events, perform the following steps.

1. In the Azure portal, browse to the AKS cluster resource group and select your AKS resource.
2. On the AKS cluster dashboard, under **Monitoring** on the left-hand side, choose **Insights**.
3. Select either the **Nodes**, **Controllers**, **Containers**, or **Deployments (preview)** tab.
4. Select an object from the performance grid, and on the properties pane found on the right side, select **View live data (preview)** option. If the AKS cluster is configured with single sign-on using Azure AD, you are prompted to authenticate on first use during that browser session. Select your account and complete authentication with Azure.

#### NOTE

When viewing the data from your Log Analytics workspace by selecting the **View in analytics** option from the properties pane, the log search results will potentially show **Nodes**, **Daemon Sets**, **Replica Sets**, **Jobs**, **Cron Jobs**, **Pods**, and **Containers** which may no longer exist. Attempting to search logs for a container which isn't available in `kubectl` will also fail here. Review the [View in analytics](#) feature to learn more about viewing historical logs, events and metrics.

After successfully authenticating, the Live Data (preview) console pane will appear below the performance data grid. If the fetch status indicator shows a green check mark, which is on the far right of the pane, it means data can be retrieved and it begins streaming to your console.

If the object you selected was a container, select the **Events** option in the pane. If you selected a Node, Pod, or controller, viewing events is automatically selected.

```
![Controller properties pane view events](./media/container-insights-livedata-overview/controller-properties-live-events.png)
```

The pane title shows the name of the Pod the container is grouped with.

#### Filter events

While viewing events, you can additionally limit the results using the **Filter** pill found to the right of the search bar. Depending on what resource you have selected, the pill lists a Pod, Namespace, or cluster to chose from.

## View metrics

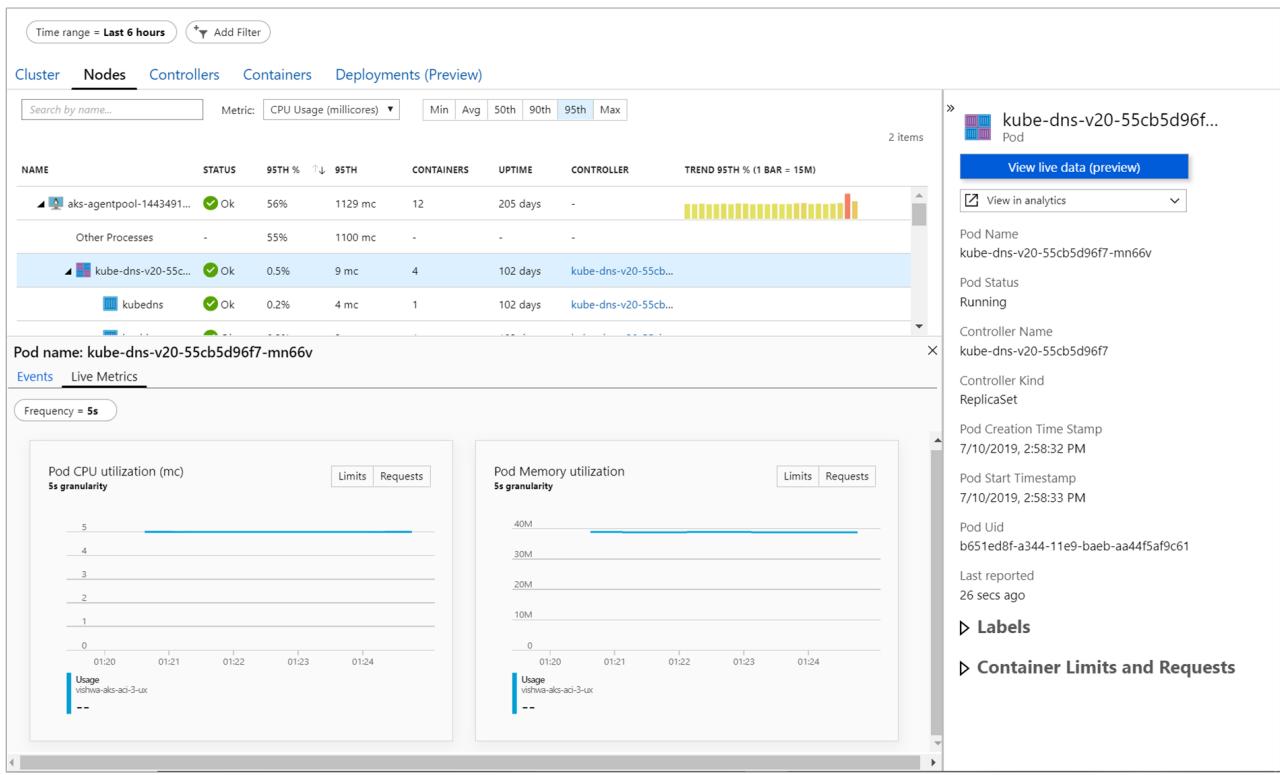
You can view real-time metric data as they are generated by the container engine from the **Nodes** or **Controllers** view only when a **Pod** is selected. To view metrics, perform the following steps.

1. In the Azure portal, browse to the AKS cluster resource group and select your AKS resource.
2. On the AKS cluster dashboard, under **Monitoring** on the left-hand side, choose **Insights**.
3. Select either the **Nodes** or **Controllers** tab.
4. Select a **Pod** object from the performance grid, and on the properties pane found on the right side, select **View live data (preview)** option. If the AKS cluster is configured with single sign-on using Azure AD, you are prompted to authenticate on first use during that browser session. Select your account and complete authentication with Azure.

#### NOTE

When viewing the data from your Log Analytics workspace by selecting the **View in analytics** option from the properties pane, the log search results will potentially show **Nodes**, **Daemon Sets**, **Replica Sets**, **Jobs**, **Cron Jobs**, **Pods**, and **Containers** which may no longer exist. Attempting to search logs for a container which isn't available in `kubectl` will also fail here. Review the [View in analytics](#) feature to learn more about viewing historical logs, events and metrics.

After successfully authenticating, the Live Data (preview) console pane will appear below the performance data grid. Metric data is retrieved and begins streaming to your console for presentation in the two charts. The pane title shows the name of the pod the container is grouped with.



## Next steps

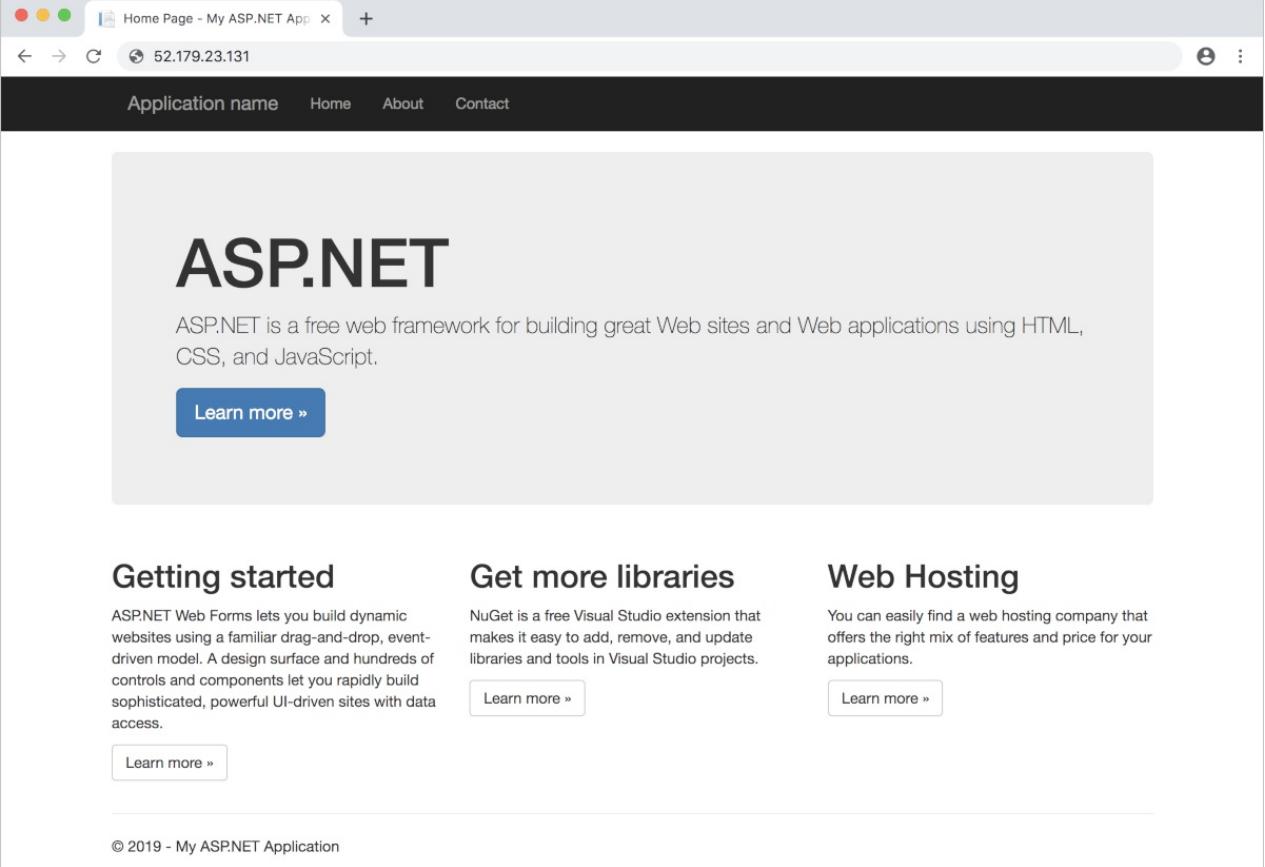
- To continue learning how to use Azure Monitor and monitor other aspects of your AKS cluster, see [View Azure Kubernetes Service health](#).
- View [log query examples](#) to see predefined queries and examples to create alerts, visualizations, or perform further analysis of your clusters.

# Preview - Create a Windows Server container on an Azure Kubernetes Service (AKS) cluster using the Azure CLI

2/25/2020 • 10 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) is a managed Kubernetes service that lets you quickly deploy and manage clusters. In this article, you deploy an AKS cluster using the Azure CLI. You also deploy an ASP.NET sample application in a Windows Server container to the cluster.

This feature is currently in preview.



The screenshot shows a web browser window with the title "Home Page - My ASP.NET App". The address bar displays the IP address "52.179.23.131". The page content includes a navigation bar with links for "Application name", "Home", "About", and "Contact". The main content area features a large "ASP.NET" logo, a brief description of the framework, and a "Learn more »" button. Below this, there are three sections: "Getting started", "Get more libraries", and "Web Hosting", each with a brief description and a "Learn more »" button. At the bottom of the page, there is a copyright notice: "© 2019 - My ASP.NET Application".

This article assumes a basic understanding of Kubernetes concepts. For more information, see [Kubernetes core concepts for Azure Kubernetes Service \(AKS\)](#).

If you don't have an Azure subscription, create a [free account](#) before you begin.

## Use Azure Cloud Shell

Azure hosts Azure Cloud Shell, an interactive shell environment that you can use through your browser. You can use either Bash or PowerShell with Cloud Shell to work with Azure services. You can use the Cloud Shell preinstalled commands to run the code in this article without having to install anything on your local environment.

To start Azure Cloud Shell:

OPTION	EXAMPLE/LINK
Select <b>Try It</b> in the upper-right corner of a code block. Selecting <b>Try It</b> doesn't automatically copy the code to Cloud Shell.	
Go to <a href="https://shell.azure.com">https://shell.azure.com</a> , or select the <b>Launch Cloud Shell</b> button to open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu bar at the upper right in the <a href="#">Azure portal</a> .	

To run the code in this article in Azure Cloud Shell:

1. Start Cloud Shell.
2. Select the **Copy** button on a code block to copy the code.
3. Paste the code into the Cloud Shell session by selecting **Ctrl+Shift+V** on Windows and Linux or by selecting **Cmd+Shift+V** on macOS.
4. Select **Enter** to run the code.

If you choose to install and use the CLI locally, this article requires that you are running the Azure CLI version 2.0.61 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Before you begin

You must add an additional node pool after you create your cluster that can run Windows Server containers. Adding an additional node pool is covered in a later step, but you first need to enable a few preview features.

### IMPORTANT

AKS preview features are self-service opt-in. Previews are provided "as-is" and "as available" and are excluded from the service level agreements and limited warranty. AKS Previews are partially covered by customer support on best effort basis. As such, these features are not meant for production use. For additional information, please see the following support articles:

- [AKS Support Policies](#)
- [Azure Support FAQ](#)

### Install aks-preview CLI extension

To use Windows Server containers, you need the *aks-preview* CLI extension version 0.4.12 or higher. Install the *aks-preview* Azure CLI extension using the [az extension add](#) command, then check for any available updates using the [az extension update](#) command:

```
# Install the aks-preview extension
az extension add --name aks-preview

# Update the extension to make sure you have the latest version installed
az extension update --name aks-preview
```

### Register Windows preview feature

To create an AKS cluster that can use multiple node pools and run Windows Server containers, first enable the *WindowsPreview* feature flags on your subscription. The *WindowsPreview* feature also uses multi-node pool

clusters and virtual machine scale set to manage the deployment and configuration of the Kubernetes nodes. Register the *WindowsPreview* feature flag using the [az feature register](#) command as shown in the following example:

```
az feature register --name WindowsPreview --namespace Microsoft.ContainerService
```

#### NOTE

Any AKS cluster you create after you've successfully registered the *WindowsPreview* feature flag use this preview cluster experience. To continue to create regular, fully-supported clusters, don't enable preview features on production subscriptions. Use a separate test or development Azure subscription for testing preview features.

It takes a few minutes for the registration to complete. Check on the registration status using the [az feature list](#) command:

```
az feature list -o table --query "[?contains(name, 'Microsoft.ContainerService/WindowsPreview')].{Name:name,State:properties.state}"
```

When the registration state is `Registered`, press Ctrl-C to stop monitoring the state. Then refresh the registration of the *Microsoft.ContainerService* resource provider using the [az provider register](#) command:

```
az provider register --namespace Microsoft.ContainerService
```

#### Limitations

The following limitations apply when you create and manage AKS clusters that support multiple node pools:

- You can't delete the first node pool.

While this feature is in preview, the following additional limitations apply:

- The AKS cluster can have a maximum of eight node pools.
- The AKS cluster can have a maximum of 400 nodes across those eight node pools.
- The Windows Server node pool name has a limit of 6 characters.

## Create a resource group

An Azure resource group is a logical group in which Azure resources are deployed and managed. When you create a resource group, you are asked to specify a location. This location is where resource group metadata is stored, it is also where your resources run in Azure if you don't specify another region during resource creation. Create a resource group using the [az group create](#) command.

The following example creates a resource group named *myResourceGroup* in the *eastus* location.

#### NOTE

This article uses Bash syntax for the commands in this tutorial. If you are using Azure Cloud Shell, ensure that the dropdown in the upper-left of the Cloud Shell window is set to **Bash**.

```
az group create --name myResourceGroup --location eastus
```

The following example output shows the resource group created successfully:

```
{  
  "id": "/subscriptions/<guid>/resourceGroups/myResourceGroup",  
  "location": "eastus",  
  "managedBy": null,  
  "name": "myResourceGroup",  
  "properties": {  
    "provisioningState": "Succeeded"  
  },  
  "tags": null,  
  "type": null  
}
```

## Create an AKS cluster

In order to run an AKS cluster that supports node pools for Windows Server containers, your cluster needs to use a network policy that uses [Azure CNI](#) (advanced) network plugin. For more detailed information to help plan out the required subnet ranges and network considerations, see [configure Azure CNI networking](#). Use the `az aks create` command to create an AKS cluster named *myAKSCluster*. This command will create the necessary network resources if they don't exist.

- The cluster is configured with two nodes
- The `windows-admin-password` and `windows-admin-username` parameters set the admin credentials for any Windows Server containers created on the cluster.

### NOTE

To ensure your cluster to operate reliably, you should run at least 2 (two) nodes in the default node pool.

Provide your own secure `PASSWORD_WIN` (remember that the commands in this article are entered into a BASH shell):

```
PASSWORD_WIN="P@ssw0rd1234"  
  
az aks create \  
  --resource-group myResourceGroup \  
  --name myAKSCluster \  
  --node-count 2 \  
  --enable-addons monitoring \  
  --kubernetes-version 1.15.7 \  
  --generate-ssh-keys \  
  --windows-admin-password $PASSWORD_WIN \  
  --windows-admin-username azureuser \  
  --vm-set-type VirtualMachineScaleSets \  
  --load-balancer-sku standard \  
  --network-plugin azure
```

### NOTE

If you get a password validation error, try creating your resource group in another region. Then try creating the cluster with the new resource group.

### NOTE

If you are unable to create the AKS cluster because the version is not supported in this region then you can use the `[az aks get-versions --location eastus]` command to find the supported version list for this region.

After a few minutes, the command completes and returns JSON-formatted information about the cluster. Occasionally the cluster can take longer than a few minutes to provision. Allow up to 10 minutes in these cases.

## Add a Windows Server node pool

By default, an AKS cluster is created with a node pool that can run Linux containers. Use `az aks nodepool add` command to add an additional node pool that can run Windows Server containers.

```
az aks nodepool add \
--resource-group myResourceGroup \
--cluster-name myAKSCluster \
--os-type Windows \
--name npwin \
--node-count 1 \
--kubernetes-version 1.15.7
```

The above command creates a new node pool named *npwin* and adds it to the *myAKSCluster*. When creating a node pool to run Windows Server containers, the default value for *node-vm-size* is *Standard\_D2s\_v3*. If you choose to set the *node-vm-size* parameter, please check the list of [restricted VM sizes](#). The minimum recommended size is *Standard\_D2s\_v3*. The above command also uses the default subnet in the default vnet created when running `az aks create`.

## Connect to the cluster

To manage a Kubernetes cluster, you use [kubectl](#), the Kubernetes command-line client. If you use Azure Cloud Shell, `kubectl` is already installed. To install `kubectl` locally, use the `az aks install-cli` command:

```
az aks install-cli
```

To configure `kubectl` to connect to your Kubernetes cluster, use the `az aks get-credentials` command. This command downloads credentials and configures the Kubernetes CLI to use them.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

To verify the connection to your cluster, use the `kubectl get` command to return a list of the cluster nodes.

```
kubectl get nodes
```

The following example output shows the all the nodes in the cluster. Make sure that the status of all nodes is *Ready*:

NAME	STATUS	ROLES	AGE	VERSION
aks-nodepool1-12345678-vmssfedcba	Ready	agent	13m	v1.15.7
aksnpwin987654	Ready	agent	108s	v1.15.7

## Run the application

A Kubernetes manifest file defines a desired state for the cluster, such as what container images to run. In this article, a manifest is used to create all objects needed to run the ASP.NET sample application in a Windows Server container. This manifest includes a [Kubernetes deployment](#) for the ASP.NET sample application and an external [Kubernetes service](#) to access the application from the internet.

The ASP.NET sample application is provided as part of the [.NET Framework Samples](#) and runs in a Windows Server container. AKS requires Windows Server containers to be based on images of *Windows Server 2019* or greater. The Kubernetes manifest file must also define a [node selector](#) to tell your AKS cluster to run your ASP.NET sample application's pod on a node that can run Windows Server containers.

Create a file named `sample.yaml` and copy in the following YAML definition. If you use the Azure Cloud Shell, this file can be created using `vi` or `nano` as if working on a virtual or physical system:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sample
  labels:
    app: sample
spec:
  replicas: 1
  template:
    metadata:
      name: sample
      labels:
        app: sample
    spec:
      nodeSelector:
        "beta.kubernetes.io/os": windows
      containers:
        - name: sample
          image: mcr.microsoft.com/dotnet/framework/samples:aspnetapp
          resources:
            limits:
              cpu: 1
              memory: 800M
            requests:
              cpu: .1
              memory: 300M
          ports:
            - containerPort: 80
      selector:
        matchLabels:
          app: sample
---
apiVersion: v1
kind: Service
metadata:
  name: sample
spec:
  type: LoadBalancer
  ports:
    - protocol: TCP
      port: 80
  selector:
    app: sample
```

Deploy the application using the `kubectl apply` command and specify the name of your YAML manifest:

```
kubectl apply -f sample.yaml
```

The following example output shows the Deployment and Service created successfully:

```
deployment.apps/sample created
service/sample created
```

# Test the application

When the application runs, a Kubernetes service exposes the application front end to the internet. This process can take a few minutes to complete. Occasionally the service can take longer than a few minutes to provision. Allow up to 10 minutes in these cases.

To monitor progress, use the `kubectl get service` command with the `--watch` argument.

```
kubectl get service sample --watch
```

Initially the *EXTERNAL-IP* for the *sample* service is shown as *pending*.

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
sample	LoadBalancer	10.0.37.27	<pending>	80:30572/TCP	6s

When the *EXTERNAL-IP* address changes from *pending* to an actual public IP address, use `CTRL-C` to stop the `kubectl` watch process. The following example output shows a valid public IP address assigned to the service:

```
sample  LoadBalancer  10.0.37.27  52.179.23.131  80:30572/TCP  2m
```

To see the sample app in action, open a web browser to the external IP address of your service.

The screenshot shows a web browser window with the title "Home Page - My ASP.NET App". The address bar shows the URL "52.179.23.131". The page content includes:

- ASP.NET**: A large heading. Below it, text reads: "ASP.NET is a free web framework for building great Web sites and Web applications using HTML, CSS, and JavaScript." A blue "Learn more »" button is present.
- Getting started**: Text: "ASP.NET Web Forms lets you build dynamic websites using a familiar drag-and-drop, event-driven model. A design surface and hundreds of controls and components let you rapidly build sophisticated, powerful UI-driven sites with data access." A blue "Learn more »" button is present.
- Get more libraries**: Text: "NuGet is a free Visual Studio extension that makes it easy to add, remove, and update libraries and tools in Visual Studio projects." A blue "Learn more »" button is present.
- Web Hosting**: Text: "You can easily find a web hosting company that offers the right mix of features and price for your applications." A blue "Learn more »" button is present.

At the bottom of the page, there is a copyright notice: "© 2019 - My ASP.NET Application".

## NOTE

If you receive a connection timeout when trying to load the page then you should verify the sample app is ready with the following command [`kubectl get pods --watch`]. Sometimes the windows container will not be started by the time your external IP address is available.

## Delete cluster

When the cluster is no longer needed, use the [az group delete](#) command to remove the resource group, container service, and all related resources.

```
az group delete --name myResourceGroup --yes --no-wait
```

### NOTE

When you delete the cluster, the Azure Active Directory service principal used by the AKS cluster is not removed. For steps on how to remove the service principal, see [AKS service principal considerations and deletion](#).

## Next steps

In this article, you deployed a Kubernetes cluster and deployed an ASP.NET sample application in a Windows Server container to it. [Access the Kubernetes web dashboard](#) for the cluster you just created.

To learn more about AKS, and walk through a complete code to deployment example, continue to the Kubernetes cluster tutorial.

[AKS tutorial](#)

# Connect with RDP to Azure Kubernetes Service (AKS) cluster Windows Server nodes for maintenance or troubleshooting

2/25/2020 • 5 minutes to read • [Edit Online](#)

Throughout the lifecycle of your Azure Kubernetes Service (AKS) cluster, you may need to access an AKS Windows Server node. This access could be for maintenance, log collection, or other troubleshooting operations. You can access the AKS Windows Server nodes using RDP. Alternatively, if you want to use SSH to access the AKS Windows Server nodes and you have access to the same keypair that was used during cluster creation, you can follow the steps in [SSH into Azure Kubernetes Service \(AKS\) cluster nodes](#). For security purposes, the AKS nodes are not exposed to the internet.

Windows Server node support is currently in preview in AKS.

This article shows you how to create an RDP connection with an AKS node using their private IP addresses.

## Before you begin

This article assumes that you have an existing AKS cluster with a Windows Server node. If you need an AKS cluster, see the article on [creating an AKS cluster with a Windows container using the Azure CLI](#). You need the Windows administrator username and password for the Windows Server node you want to troubleshoot. You also need an RDP client such as [Microsoft Remote Desktop](#).

You also need the Azure CLI version 2.0.61 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Deploy a virtual machine to the same subnet as your cluster

The Windows Server nodes of your AKS cluster don't have externally accessible IP addresses. To make an RDP connection, you can deploy a virtual machine with a publicly accessible IP address to the same subnet as your Windows Server nodes.

The following example creates a virtual machine named *myVM* in the *myResourceGroup* resource group.

First, get the subnet used by your Windows Server node pool. To get the subnet id, you need the name of the subnet. To get the name of the subnet, you need the name of the vnet. Get the vnet name by querying your cluster for its list of networks. To query the cluster, you need its name. You can get all of these by running the following in the Azure Cloud Shell:

```
CLUSTER_RG=$(az aks show -g myResourceGroup -n myAKScluster --query nodeResourceGroup -o tsv)
VNET_NAME=$(az network vnet list -g $CLUSTER_RG --query [0].name -o tsv)
SUBNET_NAME=$(az network vnet subnet list -g $CLUSTER_RG --vnet-name $VNET_NAME --query [0].name -o tsv)
SUBNET_ID=$(az network vnet subnet show -g $CLUSTER_RG --vnet-name $VNET_NAME --name $SUBNET_NAME --query id -o tsv)
```

Now that you have the SUBNET\_ID, run the following command in the same Azure Cloud Shell window to create the VM:

```
az vm create \
--resource-group myResourceGroup \
--name myVM \
--image win2019datacenter \
--admin-username azureuser \
--admin-password myP@ssw0rd12 \
--subnet $SUBNET_ID \
--query publicIpAddress -o tsv
```

The following example output shows the VM has been successfully created and displays the public IP address of the virtual machine.

```
13.62.204.18
```

Record the public IP address of the virtual machine. You will use this address in a later step.

## Allow access to the virtual machine

AKS node pool subnets are protected with NSGs (Network Security Groups) by default. To get access to the virtual machine, you'll have to enable access in the NSG.

### NOTE

The NSGs are controlled by the AKS service. Any change you make to the NSG will be overwritten at any time by the control plane.

First, get the resource group and nsg name of the nsg to add the rule to:

```
CLUSTER_RG=$(az aks show -g myResourceGroup -n myAKSCluster --query nodeResourceGroup -o tsv)
NSG_NAME=$(az network nsg list -g $CLUSTER_RG --query [].name -o tsv)
```

Then, create the NSG rule:

```
az network nsg rule create --name tempRDPAccess --resource-group $CLUSTER_RG --nsg-name $NSG_NAME --priority
100 --destination-port-range 3389 --protocol Tcp --description "Temporary RDP access to Windows nodes"
```

## Get the node address

To manage a Kubernetes cluster, you use [kubectl](#), the Kubernetes command-line client. If you use Azure Cloud Shell, `kubectl` is already installed. To install `kubectl` locally, use the `az aks install-cli` command:

```
az aks install-cli
```

To configure `kubectl` to connect to your Kubernetes cluster, use the `az aks get-credentials` command. This command downloads credentials and configures the Kubernetes CLI to use them.

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

List the internal IP address of the Windows Server nodes using the `kubectl get` command:

```
kubectl get nodes -o wide
```

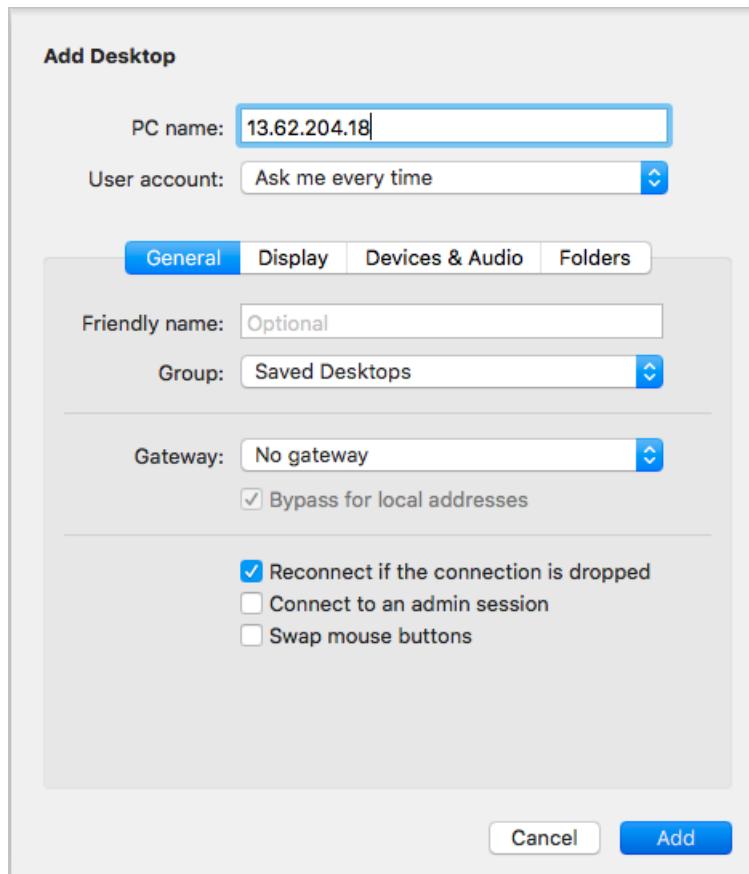
The follow example output shows the internal IP addresses of all the nodes in the cluster, including the Windows Server nodes.

```
$ kubectl get nodes -o wide
NAME                  STATUS   ROLES   AGE    VERSION   INTERNAL-IP   EXTERNAL-IP   OS-IMAGE
KERNEL-VERSION        CONTAINER-RUNTIME
aks-nodepool1-42485177-vmss000000  Ready    agent    18h    v1.12.7   10.240.0.4   <none>       Ubuntu
16.04.6 LTS           4.15.0-1040-azure docker://3.0.4
aksnpwin000000        Ready    agent    13h    v1.12.7   10.240.0.67  <none>       Windows
Server Datacenter     10.0.17763.437
```

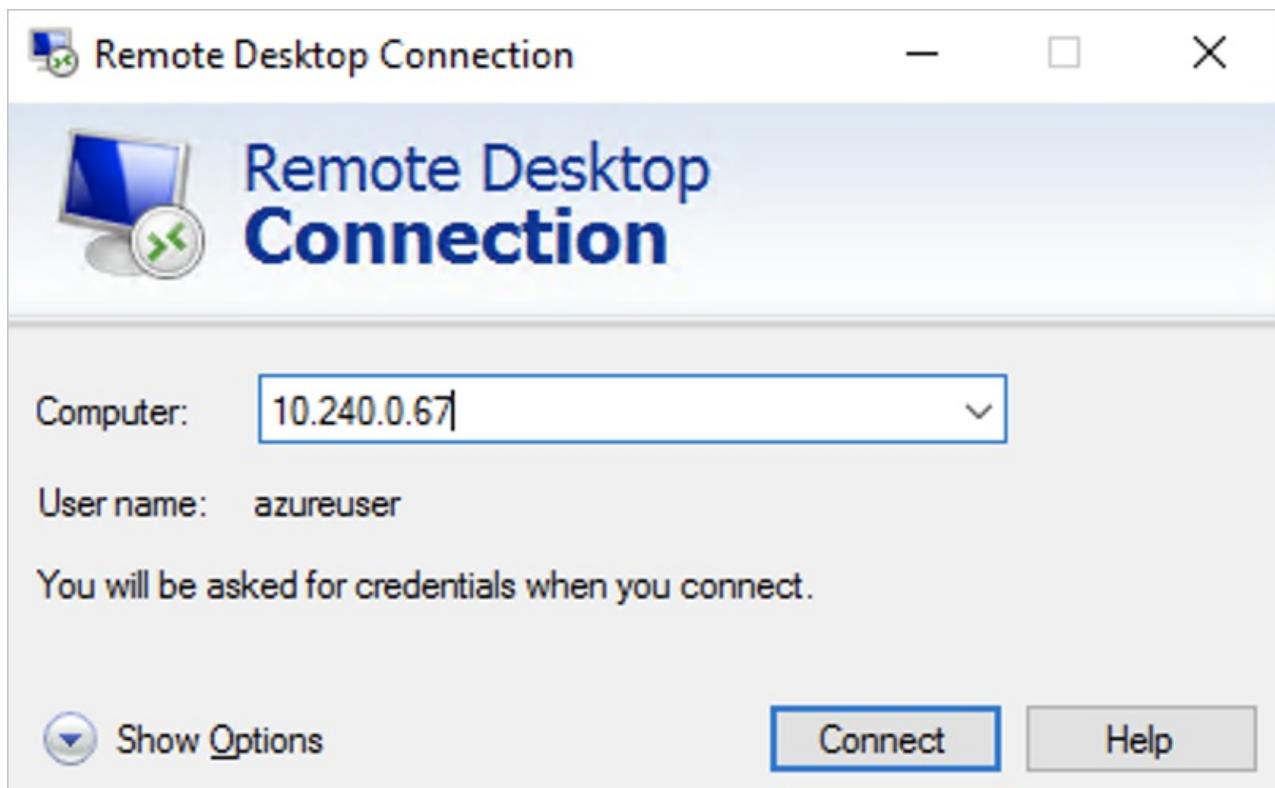
Record the internal IP address of the Windows Server node you wish to troubleshoot. You will use this address in a later step.

## Connect to the virtual machine and node

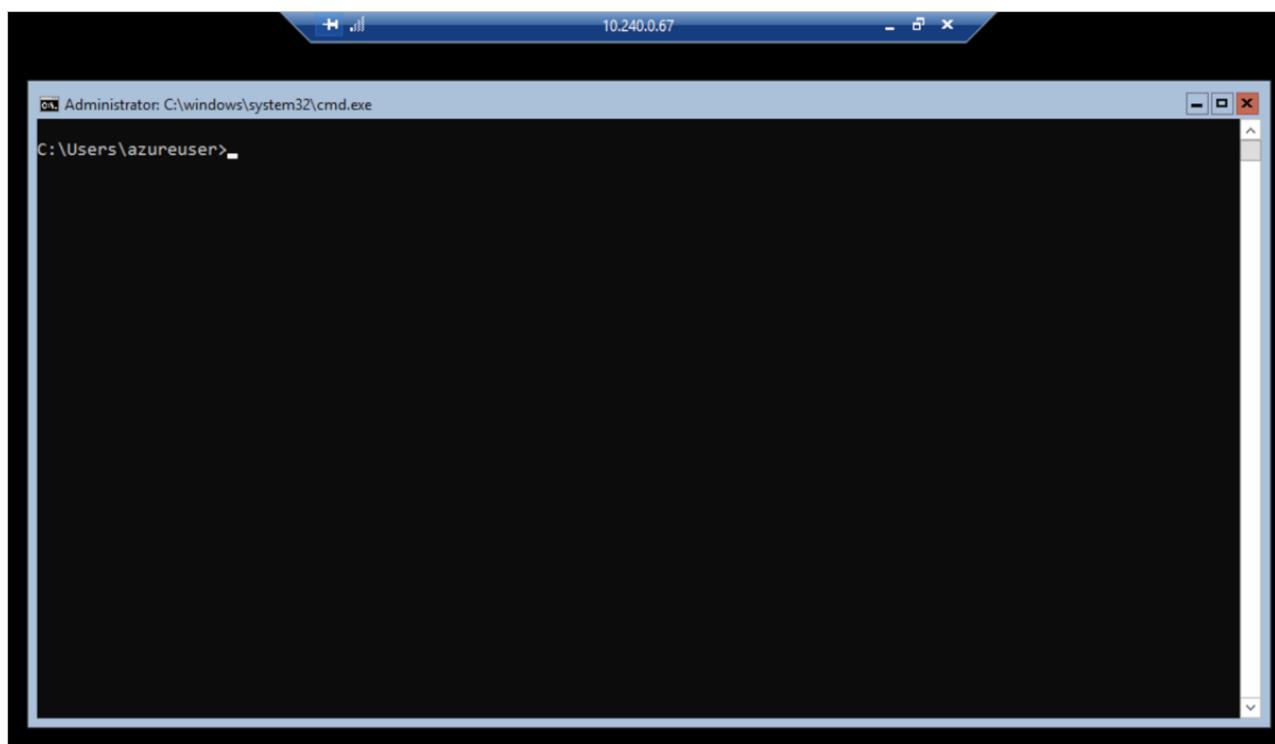
Connect to the public IP address of the virtual machine you created earlier using an RDP client such as [Microsoft Remote Desktop](#).



After you've connected to your virtual machine, connect to the *internal IP address* of the Windows Server node you want to troubleshoot using an RDP client from within your virtual machine.



You are now connected to your Windows Server node.



You can now run any troubleshooting commands in the *cmd* window. Since Windows Server nodes use Windows Server Core, there's not a full GUI or other GUI tools when you connect to a Windows Server node over RDP.

## Remove RDP access

When done, exit the RDP connection to the Windows Server node then exit the RDP session to the virtual machine. After you exit both RDP sessions, delete the virtual machine with the [az vm delete](#) command:

```
az vm delete --resource-group myResourceGroup --name myVM
```

And the NSG rule:

```
CLUSTER_RG=$(az aks show -g myResourceGroup -n myAKSCluster --query nodeResourceGroup -o tsv)
NSG_NAME=$(az network nsg list -g $CLUSTER_RG --query [].name -o tsv)
```

```
az network nsg rule delete --resource-group $CLUSTER_RG --nsg-name $NSG_NAME --name tempRDPAccess
```

## Next steps

If you need additional troubleshooting data, you can [view the Kubernetes master node logs](#) or [Azure Monitor](#).

# Current limitations for Windows Server node pools and application workloads in Azure Kubernetes Service (AKS)

2/25/2020 • 5 minutes to read • [Edit Online](#)

In Azure Kubernetes Service (AKS), you can create a node pool that runs Windows Server as the guest OS on the nodes. These nodes can run native Windows container applications, such as those built on the .NET Framework. As there are major differences in how the Linux and Windows OS provides container support, some common Kubernetes and pod-related features are not currently available for Windows node pools.

This article outlines some of the limitations and OS concepts for Windows Server nodes in AKS. Node pools for Windows Server are currently in preview.

## IMPORTANT

AKS preview features are self-service opt-in. Previews are provided "as-is" and "as available" and are excluded from the service level agreements and limited warranty. AKS Previews are partially covered by customer support on best effort basis. As such, these features are not meant for production use. For additional information, please see the following support articles:

- [AKS Support Policies](#)
- [Azure Support FAQ](#)

## Which Windows operating systems are supported?

AKS uses Windows Server 2019 as the host OS version and only supports process isolation. Container images built using other Windows Server versions are not supported. [Windows container version compatibility](#)

## Is Kubernetes different on Windows and Linux?

Windows Server node pool support includes some limitations that are part of the upstream Windows Server in Kubernetes project. These limitations are not specific to AKS. For more information on this upstream support for Windows Server in Kubernetes, see the [Supported Functionality and Limitations](#) section of the [Intro to Windows support in Kubernetes](#) document, from the Kubernetes project.

Kubernetes is historically Linux-focused. Many examples used in the upstream [Kubernetes.io](#) website are intended for use on Linux nodes. When you create deployments that use Windows Server containers, the following considerations at the OS-level apply:

- **Identity** - Linux identifies a user by an integer user identifier (UID). A user also has an alphanumeric user name for logging on, which Linux translates to the user's UID. Similarly Linux identifies a user group by an integer group identifier (GID) and translates a group name to its corresponding GID.
  - Windows Server uses a larger binary security identifier (SID) which is stored in the Windows Security Access Manager (SAM) database. This database is not shared between the host and containers, or between containers.
- **File permissions** - Windows Server uses an access control list based on SIDs, rather than a bitmask of permissions and UID+GID
- **File paths** - convention on Windows Server is to use \ instead of /.

- In pod specs that mount volumes, specify the path correctly for Windows Server containers. For example, rather than a mount point of `/mnt/volume` in a Linux container, specify a drive letter and location such as `/K/Volume` to mount as the `K:` drive.

## What kind of disks are supported for Windows?

Azure Disks and Azure Files are the supported volume types, accessed as NTFS volumes in the Windows Server container.

## Can I run Windows only clusters in AKS?

The master nodes (the control plane) in an AKS cluster are hosted by AKS the service, you will not be exposed to the operating system of the nodes hosting the master components. All AKS cluster are created with a default first node pool, which is Linux based. This node pool contains system services, which are needed for the cluster to function. It's recommended to run at least two nodes in the first node pool to ensure reliability of your cluster and the ability to do cluster operations. The first Linux-based node pool can't be deleted unless the AKS cluster itself is deleted.

## What network plug-ins are supported?

AKS clusters with Windows node pools must use the Azure CNI (advanced) networking model. Kubenet (basic) networking is not supported. For more information on the differences in network models, see [Network concepts for applications in AKS](#). - The Azure CNI network model requires additional planning and considerations for IP address management. For more information on how to plan and implement Azure CNI, see [Configure Azure CNI networking in AKS](#).

## Can I change the max. # of pods per node?

It is currently a requirement to be set to a maximum of 30 pods to ensure the reliability of your clusters.

## How do patch my Windows nodes?

Windows Server nodes in AKS must be *upgraded* to get the latest patch fixes and updates. Windows Updates are not enabled on nodes in AKS. AKS releases new node pool images as soon as patches are available, it is the customers responsibility to upgrade node pools to stay current on patches and hotfix. This is also true for the Kubernetes version being used. AKS release notes will indicate when new versions are available. For more information on upgrading a Windows Server node pool, see [Upgrade a node pool in AKS](#).

### NOTE

The updated Windows Server image will only be used if a cluster upgrade (control plane upgrade) has been performed prior to upgrading the node pool

## How do I rotate the service principal for my Windows node pool?

During preview, Windows node pools do not support service principal rotation as a preview limitation. In order to update the service principal, create a new Windows node pool and migrate your pods from the older pool to the new one. Once this is complete, delete the older node pool.

## How many node pools can I create?

The AKS cluster can have a maximum of eight (8) node pools. You can have a maximum of 400 nodes across those node pools. [Node pool limitations](#).

## What can I name my Windows node pools?

You have to keep the name to a maximum of 6 (six) characters. This is a current limitation of AKS.

## Are all features supported with Windows nodes?

Network policies and kubenet are currently not supported with Windows nodes.

## Can I run ingress controllers on Windows nodes?

Yes, an ingress-controller which supports Windows Server containers can run on Windows nodes in AKS.

## Can I use Azure Dev Spaces with Windows nodes?

Azure Dev Spaces is currently only available for Linux-based node pools.

## Can my Windows Server containers use gMSA?

Group managed service accounts (gMSA) support is not currently available in AKS.

## Can I use Azure Monitor for containers with Windows nodes and containers?

Yes you can, however Azure Monitor does not gather logs (stdout) from Windows containers. You can still attach to the live stream of stdout logs from a Windows container.

## What if I need a feature which is not supported?

We work hard to bring all the features you need to Windows in AKS, but if you do encounter gaps, the open-source, upstream [aks-engine](#) project provides an easy and fully customizable way of running Kubernetes in Azure, including Windows support. Please make sure to check out our roadmap of features coming [AKS roadmap](#).

## Next steps

To get started with Windows Server containers in AKS, [create a node pool that runs Windows Server in AKS](#).

# Access the Kubernetes web dashboard in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

Kubernetes includes a web dashboard that can be used for basic management operations. This dashboard lets you view basic health status and metrics for your applications, create and deploy services, and edit existing applications. This article shows you how to access the Kubernetes dashboard using the Azure CLI, then guides you through some basic dashboard operations.

For more information on the Kubernetes dashboard, see [Kubernetes Web UI Dashboard](#).

## Before you begin

The steps detailed in this document assume that you have created an AKS cluster and have established a `kubectl` connection with the cluster. If you need to create an AKS cluster, see the [AKS quickstart](#).

You also need the Azure CLI version 2.0.46 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Start the Kubernetes dashboard

To start the Kubernetes dashboard, use the `az aks browse` command. The following example opens the dashboard for the cluster named *myAKSCluster* in the resource group named *myResourceGroup*:

```
az aks browse --resource-group myResourceGroup --name myAKSCluster
```

This command creates a proxy between your development system and the Kubernetes API, and opens a web browser to the Kubernetes dashboard. If a web browser doesn't open to the Kubernetes dashboard, copy and paste the URL address noted in the Azure CLI, typically `http://127.0.0.1:8001`.

### IMPORTANT

If your AKS cluster uses RBAC, a *ClusterRoleBinding* must be created before you can correctly access the dashboard. By default, the Kubernetes dashboard is deployed with minimal read access and displays RBAC access errors. The Kubernetes dashboard does not currently support user-provided credentials to determine the level of access, rather it uses the roles granted to the service account. A cluster administrator can choose to grant additional access to the *kubernetes-dashboard* service account, however this can be a vector for privilege escalation. You can also integrate Azure Active Directory authentication to provide a more granular level of access.

To create a binding, use the `kubectl create clusterrolebinding` command. The following example shows how to create a sample binding, however, this sample binding does not apply any additional authentication components and may lead to insecure use. The Kubernetes dashboard is open to anyone with access to the URL. Do not expose the Kubernetes dashboard publicly.

```
kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-admin --serviceaccount=kube-system:kubernetes-dashboard
```

For more information on using the different authentication methods, see the Kubernetes dashboard wiki on [access controls](#).

The screenshot shows the Kubernetes Dashboard's Overview page. On the left, a sidebar lists Cluster components like Namespaces, Nodes, Persistent Volumes, Roles, and Storage Classes, along with a Namespace dropdown set to 'default'. The main area features three green circular status indicators for Deployments, Pods, and Replica Sets, each showing 100.00% completion. Below these are two tables: 'Deployments' and 'Pods'. The 'Deployments' table has one row for 'nginx' with details: Name: nginx, Labels: k8s-app: nginx, Pods: 1 / 1, Age: 55 seconds, Images: nginx:1.15.5. The 'Pods' table has one row for 'nginx-f597dd9c' with details: Name: nginx-f597dd9c, Node: aks-nodepool1-79590246-0, Status: Running, Restarts: 0, Age: 55 seconds, CPU (cores): -, Memory (bytes): 2.047 Mi.

## Create an application

To see how the Kubernetes dashboard can reduce the complexity of management tasks, let's create an application. You can create an application from the Kubernetes dashboard by providing text input, a YAML file, or through a graphical wizard.

To create an application, complete the following steps:

1. Select the **Create** button in the upper right window.
2. To use the graphical wizard, choose to **Create an app**.
3. Provide a name for the deployment, such as *nginx*
4. Enter the name for the container image to use, such as *nginx:1.15.5*
5. To expose port 80 for web traffic, you create a Kubernetes service. Under **Service**, select **External**, then enter **80** for both the port and target port.
6. When ready, select **Deploy** to create the app.

Cluster

- Namespaces
- Nodes
- Persistent Volumes
- Roles
- Storage Classes

Namespace

default

Overview

Workloads

- Cron Jobs
- Daemon Sets
- Deployments
- Jobs
- Pods
- Replica Sets
- Replication Controllers
- Stateful Sets

Discovery and Load Balancing

CREATE FROM TEXT INPUT   CREATE FROM FILE   CREATE AN APP

App name \* nginx

Container image \* nginx:1.15.5

Number of pods \* 1

Service \* External

Port *	Target port *	Protocol *
80	80	TCP

Port      Target port      Protocol \*

TCP

[SHOW ADVANCED OPTIONS](#)

[DEPLOY](#)   [CANCEL](#)

It takes a minute or two for a public external IP address to be assigned to the Kubernetes service. On the left-hand side, under **Discovery and Load Balancing** select **Services**. Your application's service is listed, including the *External endpoints*, as shown in the following example:

Discovery and load balancing > Services

Overview

Workloads

- Cron Jobs
- Daemon Sets
- Deployments
- Jobs
- Pods
- Replica Sets
- Replication Controllers
- Stateful Sets

Discovery and Load Balancing

Ingresses

[Services](#)

Services

Name	Labels	Cluster IP	Internal endpoints	External endpoints	Age
nginx	k8s-app: nginx	10.0.180.34	nginx:80 TCP nginx:30344 TCP	23.96.99.120:80	41 minutes
kubernetes	component: ap...	10.0.0.1	kubernetes:443 T provider: kuber...	-	7 days

Select the endpoint address to open a web browser window to the default NGINX page:

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

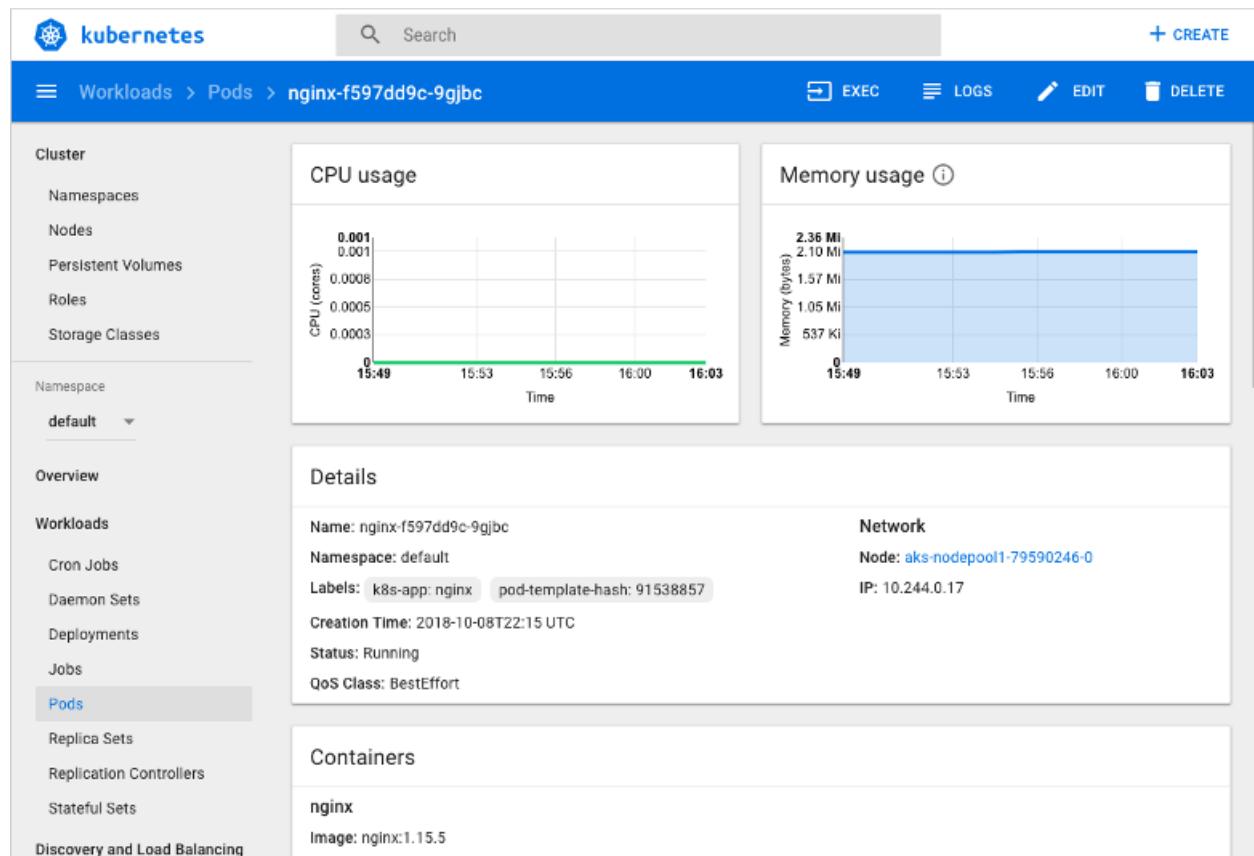
For online documentation and support please refer to [nginx.org](http://nginx.org). Commercial support is available at [nginx.com](http://nginx.com).

Thank you for using nginx.

## View pod information

The Kubernetes dashboard can provide basic monitoring metrics and troubleshooting information such as logs.

To see more information about your application pods, select **Pods** in the left-hand menu. The list of available pods is shown. Choose your *nginx* pod to view information, such as resource consumption:



## Edit the application

In addition to creating and viewing applications, the Kubernetes dashboard can be used to edit and update application deployments. To provide additional redundancy for the application, let's increase the number of NGINX replicas.

To edit a deployment:

1. Select **Deployments** in the left-hand menu, and then choose your *nginx* deployment.
2. Select **Edit** in the upper right-hand navigation bar.
3. Locate the `spec.replicas` value, at around line 20. To increase the number of replicas for the application, change this value from 1 to 3.
4. Select **Update** when ready.

```

15: "annotations": {
16:   "deployment.kubernetes.io/revision": 1
17: }
18: },
19: "spec": {
20:   "replicas": 3,
21:   "selector": {
22:     "matchLabels": {
23:       "k8s-app": "nginx"
24:     }
25:   },
26:   "template": {
27:     "metadata": {
28:       "name": "nginx",
29:       "creationTimestamp": null,
30:       "labels": {

```

CANCEL    COPY    UPDATE

It takes a few moments for the new pods to be created inside a replica set. On the left-hand menu, choose **Replica Sets**, and then choose your *nginx* replica set. The list of pods now reflects the updated replica count, as shown in the following example output:

Name	Node	Status	Restarts	Age	CPU (cores)	Memory (bytes)
nginx-f597dd9c	aks-nodepool1-79590246-0	Running	0	56 minutes	0	2.098 Mi
nginx-f597dd9c	aks-nodepool1-79590246-0	Running	0	10 seconds	-	2.059 Mi
nginx-f597dd9c	aks-nodepool1-79590246-0	Running	0	10 seconds	-	1.980 Mi

## Next steps

For more information about the Kubernetes dashboard, see the [Kubernetes Web UI Dashboard](#).

2 minutes to read

# Create a Kubernetes dev space: Visual Studio Code and Java with Azure Dev Spaces

2/25/2020 • 8 minutes to read • [Edit Online](#)

In this guide, you will learn how to:

- Create a Kubernetes-based environment in Azure that is optimized for development - a *dev space*.
- Iteratively develop code in containers using VS Code and the command line.
- Productively develop and test your code in a team environment.

## NOTE

If you get stuck at any time, see the [Troubleshooting](#) section.

## Install the Azure CLI

Azure Dev Spaces requires minimal local machine setup. Most of your dev space's configuration gets stored in the cloud, and is shareable with other users. Start by downloading and running the [Azure CLI](#).

### Sign in to Azure CLI

Sign in to Azure. Type the following command in a terminal window:

```
az login
```

## NOTE

If you don't have an Azure subscription, you can create a [free account](#).

### If you have multiple Azure subscriptions...

You can view your subscriptions by running:

```
az account list --output table
```

Locate the subscription which has *True* for *IsDefault*. If this isn't the subscription you want to use, you can change the default subscription:

```
az account set --subscription <subscription ID>
```

## Create a Kubernetes cluster enabled for Azure Dev Spaces

At the command prompt, create the resource group in a [region that supports Azure Dev Spaces](#).

```
az group create --name MyResourceGroup --location <region>
```

Create a Kubernetes cluster with the following command:

```
az aks create -g MyResourceGroup -n MyAKS --location <region> --generate-ssh-keys
```

It takes a few minutes to create the cluster.

### Configure your AKS cluster to use Azure Dev Spaces

Enter the following Azure CLI command, using the resource group that contains your AKS cluster, and your AKS cluster name. The command configures your cluster with support for Azure Dev Spaces.

```
az aks use-dev-spaces -g MyResourceGroup -n MyAKS
```

#### IMPORTANT

The Azure Dev Spaces configuration process will remove the `azds` namespace in the cluster, if it exists.

## Get Kubernetes debugging for VS Code

Rich features like Kubernetes debugging are available for .NET Core and Node.js developers using VS Code.

1. If you don't have it, install [VS Code](#).
2. Download and install the [VS Azure Dev Spaces extension](#). Click Install once on the extension's Marketplace page, and again in VS Code.

In order to debug Java applications with Azure Dev Spaces, download and install the [Java Debugger for Azure Dev Spaces](#) extension for VS Code. Click Install once on the extension's Marketplace page, and again in VS Code.

## Create a web app running in a container

In this section, you'll create a Java web application and get it running in a container in Kubernetes.

### Create a Java web app

Download code from GitHub by navigating to <https://github.com/Azure/dev-spaces> and select **Clone or Download** to download the GitHub repository to your local environment. The code for this guide is in `samples/java/getting-started/webfrontend`.

## Preparing code for Docker and Kubernetes development

So far, you have a basic web app that can run locally. You'll now containerize it by creating assets that define the app's container and how it will deploy to Kubernetes. This task is easy to do with Azure Dev Spaces:

1. Launch VS Code and open the `webfrontend` folder. (You can ignore any default prompts to add debug assets or restore the project.)
2. Open the Integrated Terminal in VS Code (using the **View > Integrated Terminal** menu).
3. Run this command (be sure that **webfrontend** is your current folder):

```
azds prep --enable-ingress
```

The Azure CLI's `azds prep` command generates Docker and Kubernetes assets with default settings:

- `./Dockerfile` describes the app's container image, and how the source code is built and runs within the container.

- A [Helm chart](#) under `./charts/webfrontend` describes how to deploy the container to Kubernetes.

#### TIP

The [Dockerfile](#) and [Helm chart](#) for your project is used by Azure Dev Spaces to build and run your code, but you can modify these files if you want to change how the project is built and ran.

For now, it isn't necessary to understand the full content of these files. It's worth pointing out, however, that **the same Kubernetes and Docker configuration-as-code assets can be used from development through to production, thus providing better consistency across different environments.**

A file named `./azds.yaml` is also generated by the `prep` command, and it is the configuration file for Azure Dev Spaces. It complements the Docker and Kubernetes artifacts with additional configuration that enables an iterative development experience in Azure.

## Build and run code in Kubernetes

Let's run our code! In the terminal window, run this command from the **root code folder**, `webfrontend`:

```
azds up
```

Keep an eye on the command's output, you'll notice several things as it progresses:

- Source code is synced to the dev space in Azure.
- A container image is built in Azure, as specified by the Docker assets in your code folder.
- Kubernetes objects are created that utilize the container image as specified by the Helm chart in your code folder.
- Information about the container's endpoint(s) is displayed. In our case, we're expecting a public HTTP URL.
- Assuming the above stages complete successfully, you should begin to see `stdout` (and `stderr`) output as the container starts up.

#### NOTE

These steps will take longer the first time the `up` command is run, but subsequent runs should be quicker.

## Test the web app

Scan the console output for information about the public URL that was created by the `up` command. It will be in the form:

```
(pending registration) Service 'webfrontend' port 'http' will be available at <url>
Service 'webfrontend' port 'http' is available at http://webfrontend.1234567890abcdef1234.eus.azds.io/
Service 'webfrontend' port 80 (TCP) is available at 'http://localhost:<port>'
```

Identify the public URL for the service in the output from the `up` command. It ends in `.azds.io`. In the above example, the public URL is `http://webfrontend.1234567890abcdef1234.eus.azds.io/`.

To see your web app, open the public URL in a browser. Also, notice `stdout` and `stderr` output is streamed to the `azds trace` terminal window as you interact with your web app. You'll also see tracking information for HTTP requests as they go through the system. This makes it easier for you to track complex multi-service calls during development. The instrumentation added by Dev Spaces provides this request tracking.

## NOTE

In addition to the public URL, you can use the alternative `http://localhost:<portnumber>` URL that is displayed in the console output. If you use the localhost URL, it may seem like the container is running locally, but actually it is running in AKS. Azure Dev Spaces uses Kubernetes *port-forward* functionality to map the localhost port to the container running in AKS. This facilitates interacting with the service from your local machine.

## Update a content file

Azure Dev Spaces isn't just about getting code running in Kubernetes - it's about enabling you to quickly and iteratively see your code changes take effect in a Kubernetes environment in the cloud.

1. In the terminal window, press `Ctrl+C` (to stop `azds up`).
2. Open `src/main/java/com/ms/sample/webfrontend/Application.java`, and edit the greeting message on [line 19](#):

```
return "Hello from webfrontend in Azure!";
```

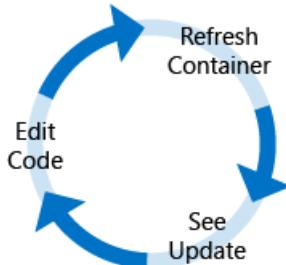
3. Save the file.
4. Run `azds up` in the terminal window.

This command rebuilds the container image and redeploys the Helm chart. To see your code changes take effect in the running application, simply refresh the browser.

But there is an even *faster method* for developing code, which you'll explore in the next section.

## Debug a container in Kubernetes

In this section, you'll use VS Code to directly debug our container running in Azure. You'll also learn how to get a faster edit-run-test loop.



## NOTE

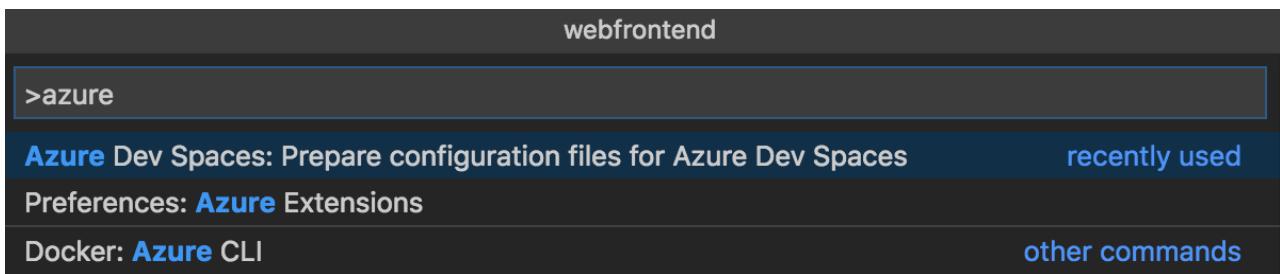
If you get stuck at any time, see the [Troubleshooting](#) section, or post a comment on this page.

## Initialize debug assets with the VS Code extension

You first need to configure your code project so VS Code will communicate with our dev space in Azure. The VS Code extension for Azure Dev Spaces provides a helper command to set up debug configuration.

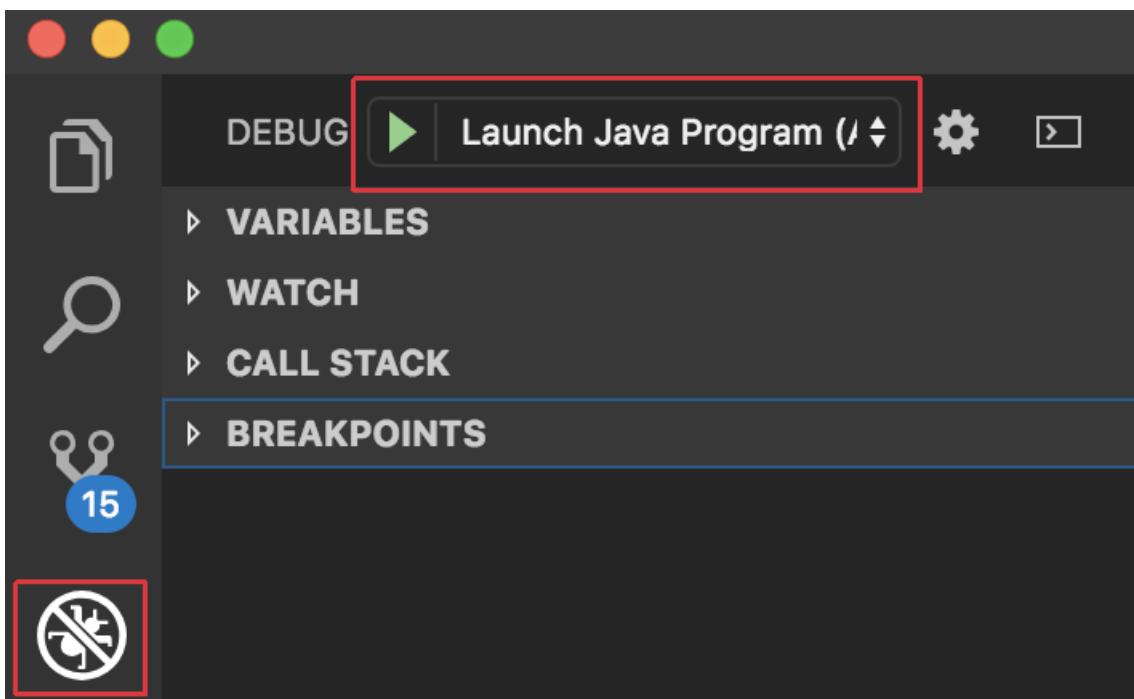
Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command: `Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces`.

This adds debug configuration for Azure Dev Spaces under the `.vscode` folder. This command is not to be confused with the `azds prep` command, which configures the project for deployment.



### Select the AZDS debug configuration

1. To open the Debug view, click on the Debug icon in the **Activity Bar** on the side of VS Code.
2. Select **Launch Java Program (AZDS)** as the active debug configuration.



#### NOTE

If you don't see any Azure Dev Spaces commands in the Command Palette, ensure you have installed the VS Code extension for Azure Dev Spaces. Be sure the workspace you opened in VS Code is the folder that contains `azds.yaml`.

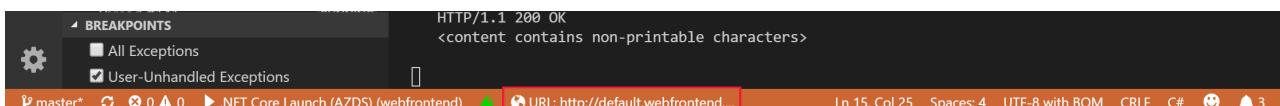
### Debug the container in Kubernetes

Hit **F5** to debug your code in Kubernetes.

As with the `up` command, code is synced to the dev space, and a container is built and deployed to Kubernetes. This time, of course, the debugger is attached to the remote container.

#### TIP

The VS Code status bar will turn orange, indicating that the debugger is attached. It will also display a clickable URL, which you can use to open your application.



Set a breakpoint in a server-side code file, for example within the `greeting()` function in the `src/main/java/com/ms/sample/webfrontend/Application.java` source file. Refreshing the browser page causes the

breakpoint to hit.

You have full access to debug information just like you would if the code was executing locally, such as the call stack, local variables, exception information, etc.

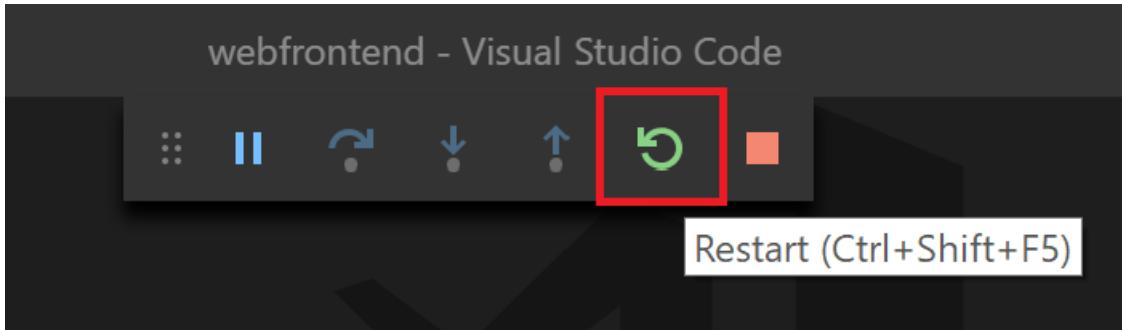
### Edit code and refresh

With the debugger active, make a code edit. For example, modify the greeting in

```
src/main/java/com/ms/sample/webfrontend/Application.java .
```

```
public String greeting()
{
    return "I'm debugging Java code in Azure!";
}
```

Save the file, and in the **Debug actions pane**, click the **Restart** button.



Instead of rebuilding and redeploying a new container image each time code edits are made, which will often take considerable time, Azure Dev Spaces will incrementally recompile code within the existing container to provide a faster edit/debug loop.

Refresh the web app in the browser. You should see your custom message appear in the UI.

**Now you have a method for rapidly iterating on code and debugging directly in Kubernetes!** Next, you'll see how you can create and call a second container.

## Next steps

[Learn about multi-service development](#)

# Running multiple dependent services: Java and Visual Studio Code with Azure Dev Spaces

12/11/2019 • 2 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to develop multi-service applications using Azure Dev Spaces, along with some of the added benefits that Dev Spaces provides.

## Call a service running in a separate container

In this section, you create a second service, `mywebapi`, and have `webfrontend` call it. Each service will run in separate containers. You'll then debug across both containers.



### Download sample code for `mywebapi`

For the sake of time, let's download sample code from a GitHub repository. Go to <https://github.com/Azure/dev-spaces> and select **Clone or Download** to download the GitHub repository. The code for this section is in `samples/java/getting-started/mywebapi`.

### Run `mywebapi`

1. Open the folder `mywebapi` in a *separate VS Code window*.
2. Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command: `Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces`.
3. Hit F5, and wait for the service to build and deploy. You'll know it's ready when a message similar to the below appears in the debug console:

```
2019-03-11 17:02:35.935 INFO 216 --- [           main] com.ms.sample.mywebapi.Application      : Started Application in 8.164 seconds (JVM running for 9.272)
```

4. The endpoint URL will look something like `http://localhost:<portnumber>`. **Tip: The VS Code status bar will turn orange and display a clickable URL.** It might seem like the container is running locally, but actually it is running in our dev space in Azure. The reason for the localhost address is because `mywebapi` has not defined any public endpoints and can only be accessed from within the Kubernetes instance. For your convenience, and to facilitate interacting with the private service from your local machine, Azure Dev Spaces creates a temporary SSH tunnel to the container running in Azure.
5. When `mywebapi` is ready, open your browser to the localhost address.
6. If all the steps were successful, you should be able to see a response from the `mywebapi` service.

### Make a request from `webfrontend` to `mywebapi`

Let's now write code in `webfrontend` that makes a request to `mywebapi`.

1. Switch to the VS Code window for `webfrontend`.
2. Add the following `import` statements under the `package` statement:

```
import java.io.*;
import java.net.*;
```

3. Replace the code for the greeting method:

```
@RequestMapping(value = "/greeting", produces = "text/plain")
public String greeting(@RequestHeader(value = "azds-route-as", required = false) String azdsRouteAs)
throws Exception {
    URLConnection conn = new URL("http://mywebapi/").openConnection();
    conn.setRequestProperty("azds-route-as", azdsRouteAs); // propagate dev space routing header
    try (BufferedReader reader = new BufferedReader(new InputStreamReader(conn.getInputStream())))
    {
        return "Hello from webfrontend and " + reader.lines().reduce("\n", String::concat);
    }
}
```

The preceding code example forwards the `azds-route-as` header from the incoming request to the outgoing request. You'll see later how this helps teams with collaborative development.

### Debug across multiple services

1. At this point, `mywebapi` should still be running with the debugger attached. If it is not, hit F5 in the `mywebapi` project.
2. Set a breakpoint in the `index()` method of the `mywebapi` project, on [line 19 of `Application.java`](#)
3. In the `webfrontend` project, set a breakpoint just before it sends a GET request to `mywebapi`, on the line starting with `try`.
4. Hit F5 in the `webfrontend` project (or restart the debugger if currently running).
5. Invoke the web app, and step through code in both services.
6. In the web app, the About page will display a message concatenated by the two services: "Hello from webfrontend and Hello from mywebapi."

### Well done!

You now have a multi-container application where each container can be developed and deployed separately.

## Next steps

[Learn about team development in Dev Spaces](#)

# Team development using Java and Visual Studio Code with Azure Dev Spaces

12/11/2019 • 9 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how a team of developers can simultaneously collaborate in the same Kubernetes cluster using Dev Spaces.

## Learn about team development

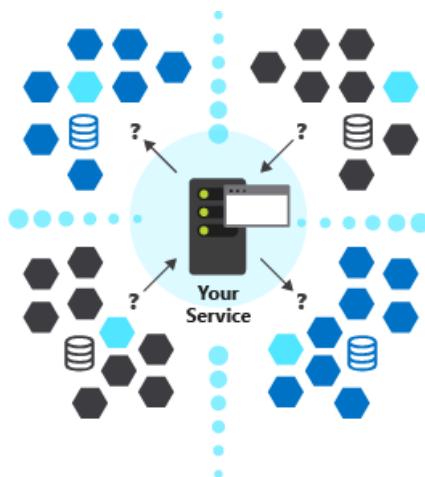
So far you've been running your application's code as if you were the only developer working on the app. In this section, you'll learn how Azure Dev Spaces streamlines team development:

- Enable a team of developers to work in the same environment, by working in a shared dev space or in distinct dev spaces as needed.
- Supports each developer iterating on their code in isolation and without fear of breaking others.
- Test code end-to-end, prior to code commit, without having to create mocks or simulate dependencies.

### Challenges with developing microservices

Your sample application isn't very complex at the moment. But in real-world development, challenges soon emerge as you add more services and the development team grows. It can become unrealistic to run everything locally for development.

- Your development machine may not have enough resources to run every service you need at once.
- Some services may need to be publicly reachable. For example, a service may need to have an endpoint that responds to a webhook.
- If you want to run a subset of services, you have to know the full dependency hierarchy between all your services. Determining this can be difficult, especially as your number of services increase.
- Some developers resort to simulating, or mocking up, many of their service dependencies. This approach can help, but managing those mocks can soon impact development cost. Plus, this approach leads to your development environment looking very different from production, which allows subtle bugs to creep in.
- It follows that doing any type of integration testing becomes difficult. Integration testing can only realistically happen post-commit, which means you see problems later in the development cycle.



### Work in a shared dev space

With Azure Dev Spaces, you can set up a *shared* dev space in Azure. Each developer can focus on just their part of the application, and can iteratively develop *pre-commit code* in a dev space that already contains all the other services and cloud resources that their scenarios depend on. Dependencies are always up-to-date, and developers are working in a way that mirrors production.

### Work in your own space

As you develop code for your service, and before you're ready to check it in, code often won't be in a good state. You're still iteratively shaping it, testing it, and experimenting with solutions. Azure Dev Spaces provides the concept of a **space**, which allows you to work in isolation, and without the fear of breaking your team members.

## Use Dev Spaces for team development

Let's demonstrate these ideas with a concrete example using our *webfrontend -> mywebapi* sample application. We'll imagine a scenario where a developer, Scott, needs to make a change to the *mywebapi* service, and *only* that service. The *webfrontend* won't need to change as part of Scott's update.

*Without* using Dev Spaces, Scott would have a few ways to develop and test his update, none of which are ideal:

- Run ALL components locally. This requires a more powerful development machine with Docker installed, and potentially MiniKube.
- Run ALL components in an isolated namespace on the Kubernetes cluster. Since *webfrontend* isn't changing, this is a waste of cluster resources.
- ONLY run *mywebapi*, and make manual REST calls to test. This doesn't test the full end-to-end flow.
- Add development-focused code to *webfrontend* that allows the developer to send requests to a different instance of *mywebapi*. This complicates the *webfrontend* service.

### Set up your baseline

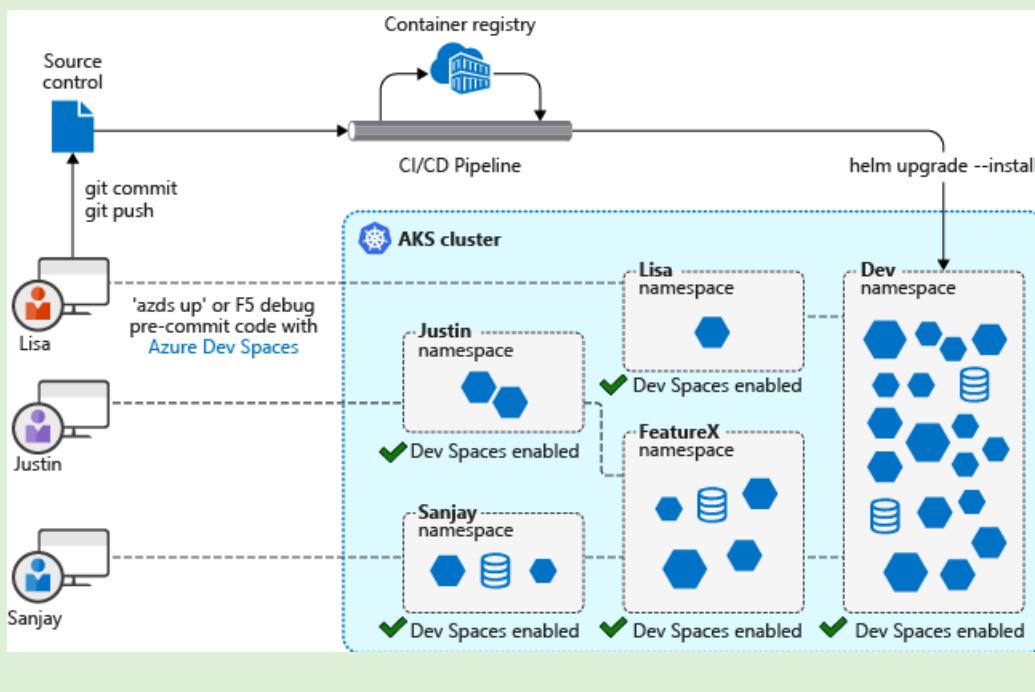
First we'll need to deploy a baseline of our services. This deployment will represent the "last known good" so you can easily compare the behavior of your local code vs. the checked-in version. We'll then create a child space based on this baseline so we can test our changes to *mywebapi* within the context of the larger application.

1. Clone the [Dev Spaces sample application](#): `git clone https://github.com/Azure/dev-spaces && cd dev-spaces`
2. Checkout the remote branch *azds\_updates*: `git checkout -b azds_updates origin/azds_updates`
3. Select the *dev* space: `azds space select --name dev`. When prompted to select a parent dev space, select *<none>*.
4. Navigate to the *mywebapi* directory and execute: `azds up -d`
5. Navigate to the *webfrontend* directory and execute: `azds up -d`
6. Execute `azds list-uris` to see the public endpoint for *webfrontend*

## TIP

The above steps manually set up a baseline, but we recommend teams use CI/CD to automatically keep your baseline up to date with committed code.

Check out our [guide to setting up CI/CD with Azure DevOps](#) to create a workflow similar to the following diagram.



At this point your baseline should be running. Run the `azds list-up --all` command, and you'll see output similar to the following:

Name	DevSpace	Type	Updated	Status
mywebapi	dev	Service	3m ago	Running
mywebapi-56c8f45d9-zs4mw	dev	Pod	3m ago	Running
webfrontend	dev	Service	1m ago	Running
webfrontend-6b6ddbb98f-fgvnc	dev	Pod	1m ago	Running

The DevSpace column shows that both services are running in a space named *dev*. Anyone who opens the public URL and navigates to the web app will invoke the checked-in code path that runs through both services. Now suppose you want to continue developing *mywebapi*. How can you make code changes and test them and not interrupt other developers who are using the dev environment? To do that, you'll set up your own space.

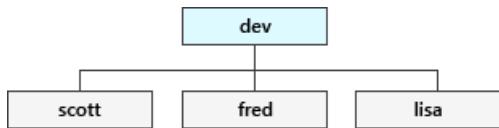
## Create a dev space

To run your own version of *mywebapi* in a space other than *dev*, you can create your own space by using the following command:

```
azds space select --name scott
```

When prompted, select *dev* as the **parent dev space**. This means our new space, *dev/scott*, will derive from the space *dev*. We'll shortly see how this will help us with testing.

Keeping with our introductory hypothetical, we've used the name *scott* for the new space so peers can identify who is working in it. But it can be called anything you like, and be flexible about what it means, like *sprint4* or *demo*. Whatever the case, *dev* serves as the baseline for all developers working on a piece of this application:



Run the `azds space list` command to see a list of all the spaces in the `dev` environment. The *Selected* column indicates which space you currently have selected (true/false). In your case, the space named `dev/scott` was automatically selected when it was created. You can select another space at any time with the `azds space select` command.

Let's see it in action.

### Make a code change

Go to the VS Code window for `mywebapi` and make a code edit to the `String index()` method in `src/main/java/com/ms/sample/mywebapi/Application.java`, for example:

```

@RequestMapping(value = "/", produces = "text/plain")
public String index() {
    return "Hello from mywebapi says something new";
}

```

### Run the service

To run the service, hit F5 (or type `azds up` in the Terminal Window) to run the service. The service will automatically run in your newly selected space `dev/scott`. Confirm that your service is running in its own space by running `azds list-up`:

```

$ azds list-up

Name          DevSpace  Type      Updated   Status
mywebapi      scott     Service   3m ago   Running
webfrontend   dev       Service   26m ago  Running

```

Notice an instance of `mywebapi` is now running in the `dev/scott` space. The version running in `dev` is still running but it is not listed.

List the URLs for the current space by running `azds list-uris`.

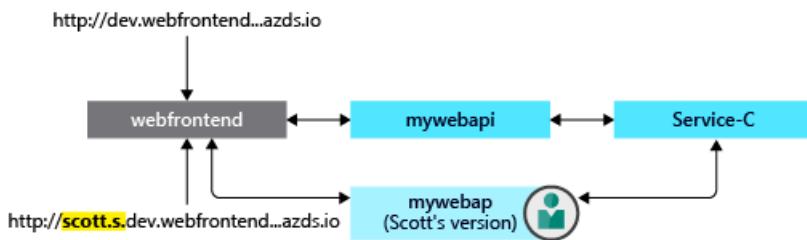
```

$ azds list-uris

Uri                                         Status
-----
http://localhost:53831 => mywebapi.scott:80           Tunneled
http://scott.s.dev.webfrontend.6364744826e042319629.ce.azds.io/ Available

```

Notice the public access point URL for `webfrontend` is prefixed with `scott.s`. This URL is unique to the `dev/scott` space. This URL prefix tells the Ingress controller to route requests to the `dev/scott` version of a service. When a request with this URL is handled by Dev Spaces, the Ingress Controller first tries to route the request to the `webfrontend` service in the `dev/scott` space. If that fails, the request will be routed to the `webfrontend` service in the `dev` space as a fallback. Also notice there is a localhost URL to access the service over localhost using the Kubernetes *port-forward* functionality. For more information about URLs and routing in Azure Dev Spaces, see [How Azure Dev Spaces works and is configured](#).



This built-in feature of Azure Dev Spaces lets you test code in a shared space without requiring each developer to re-create the full stack of services in their space. This routing requires your app code to forward propagation headers, as illustrated in the previous step of this guide.

### Test code in a space

To test your new version of *mywebapi* with *webfrontend*, open your browser to the public access point URL for *webfrontend* and go to the About page. You should see your new message displayed.

Now, remove the "scott.s." part of the URL, and refresh the browser. You should see the old behavior (with the *mywebapi* version running in *dev*).

Once you have a *dev* space, which always contains your latest changes, and assuming your application is designed to take advantage of DevSpace's space-based routing as described in this tutorial section, hopefully it becomes easy to see how Dev Spaces can greatly assist in testing new features within the context of the larger application. Rather than having to deploy *all* services to your private space, you can create a private space that derives from *dev*, and only "up" the services you're actually working on. The Dev Spaces routing infrastructure will handle the rest by utilizing as many services out of your private space as it can find, while defaulting back to the latest version running in the *dev* space. And better still, *multiple* developers can actively develop different services at the same time in their own space without disrupting each other.

### Well done!

You've completed the getting started guide! You learned how to:

- Set up Azure Dev Spaces with a managed Kubernetes cluster in Azure.
- Iteratively develop code in containers.
- Independently develop two separate services, and used Kubernetes' DNS service discovery to make a call to another service.
- Productively develop and test your code in a team environment.
- Establish a baseline of functionality using Dev Spaces to easily test isolated changes within the context of a larger microservice application

Now that you've explored Azure Dev Spaces, [share your dev space with a team member](#) and begin collaborating.

## Clean up

To completely delete an Azure Dev Spaces instance on a cluster, including all the dev spaces and running services within it, use the `az aks remove-dev-spaces` command. Bear in mind that this action is irreversible. You can add support for Azure Dev Spaces again on the cluster, but it will be as if you are starting again. Your old services and spaces won't be restored.

The following example lists the Azure Dev Spaces controllers in your active subscription, and then deletes the Azure Dev Spaces controller that is associated with AKS cluster 'myaks' in resource group 'myaks-rg'.

```

azds controller list
az aks remove-dev-spaces --name myaks --resource-group myaks-rg

```

# Create a Kubernetes dev space: Visual Studio Code and .NET Core with Azure Dev Spaces

2/25/2020 • 9 minutes to read • [Edit Online](#)

In this guide, you will learn how to:

- Create a Kubernetes-based environment in Azure that is optimized for development - a *dev space*.
- Iteratively develop code in containers using VS Code and the command line.
- Productively develop and test your code in a team environment.

## NOTE

If you get stuck at any time, see the [Troubleshooting](#) section.

## Install the Azure CLI

Azure Dev Spaces requires minimal local machine setup. Most of your dev space's configuration gets stored in the cloud, and is shareable with other users. Start by downloading and running the [Azure CLI](#).

### Sign in to Azure CLI

Sign in to Azure. Type the following command in a terminal window:

```
az login
```

## NOTE

If you don't have an Azure subscription, you can create a [free account](#).

### If you have multiple Azure subscriptions...

You can view your subscriptions by running:

```
az account list --output table
```

Locate the subscription which has *True* for *IsDefault*. If this isn't the subscription you want to use, you can change the default subscription:

```
az account set --subscription <subscription ID>
```

## Create a Kubernetes cluster enabled for Azure Dev Spaces

At the command prompt, create the resource group in a [region that supports Azure Dev Spaces](#).

```
az group create --name MyResourceGroup --location <region>
```

Create a Kubernetes cluster with the following command:

```
az aks create -g MyResourceGroup -n MyAKS --location <region> --generate-ssh-keys
```

It takes a few minutes to create the cluster.

### Configure your AKS cluster to use Azure Dev Spaces

Enter the following Azure CLI command, using the resource group that contains your AKS cluster, and your AKS cluster name. The command configures your cluster with support for Azure Dev Spaces.

```
az aks use-dev-spaces -g MyResourceGroup -n MyAKS
```

#### IMPORTANT

The Azure Dev Spaces configuration process will remove the `azds` namespace in the cluster, if it exists.

## Get Kubernetes debugging for VS Code

Rich features like Kubernetes debugging are available for .NET Core and Node.js developers using VS Code.

1. If you don't have it, install [VS Code](#).
2. Download and install the [VS Azure Dev Spaces](#) and [C#](#) extensions. For each extension, click install on the extension's Marketplace page, and again in VS Code.

## Create a web app running in a container

In this section, you'll create an ASP.NET Core web app and get it running in a container in Kubernetes.

### Create an ASP.NET Core web app

Clone or download the [Azure Dev Spaces sample application](#). This article uses the code in the `samples/dotnetcore/getting-started/webfrontend` directory.

## Preparing code for Docker and Kubernetes development

So far, you have a basic web app that can run locally. You'll now containerize it by creating assets that define the app's container and how it will deploy to Kubernetes. This task is easy to do with Azure Dev Spaces:

1. Launch VS Code and open the `webfrontend` folder. (You can ignore any default prompts to add debug assets or restore the project.)
2. Open the Integrated Terminal in VS Code (using the **View > Integrated Terminal** menu).
3. Run this command (be sure that `webfrontend` is your current folder):

```
azds prep --enable-ingress
```

The Azure CLI's `azds prep` command generates Docker and Kubernetes assets with default settings:

- `./Dockerfile` describes the app's container image, and how the source code is built and runs within the container.
- A [Helm chart](#) under `./charts/webfrontend` describes how to deploy the container to Kubernetes.

### TIP

The [Dockerfile](#) and [Helm chart](#) for your project is used by Azure Dev Spaces to build and run your code, but you can modify these files if you want to change how the project is built and ran.

For now, it isn't necessary to understand the full content of these files. It's worth pointing out, however, that **the same Kubernetes and Docker configuration-as-code assets can be used from development through to production, thus providing better consistency across different environments.**

A file named `./azds.yaml` is also generated by the `prep` command, and it is the configuration file for Azure Dev Spaces. It complements the Docker and Kubernetes artifacts with additional configuration that enables an iterative development experience in Azure.

## Build and run code in Kubernetes

Let's run our code! In the terminal window, run this command from the **root code folder**, `webfrontend`:

```
azds up
```

Keep an eye on the command's output, you'll notice several things as it progresses:

- Source code is synced to the dev space in Azure.
- A container image is built in Azure, as specified by the Docker assets in your code folder.
- Kubernetes objects are created that utilize the container image as specified by the Helm chart in your code folder.
- Information about the container's endpoint(s) is displayed. In our case, we're expecting a public HTTP URL.
- Assuming the above stages complete successfully, you should begin to see `stdout` (and `stderr`) output as the container starts up.

### NOTE

These steps will take longer the first time the `up` command is run, but subsequent runs should be quicker.

## Test the web app

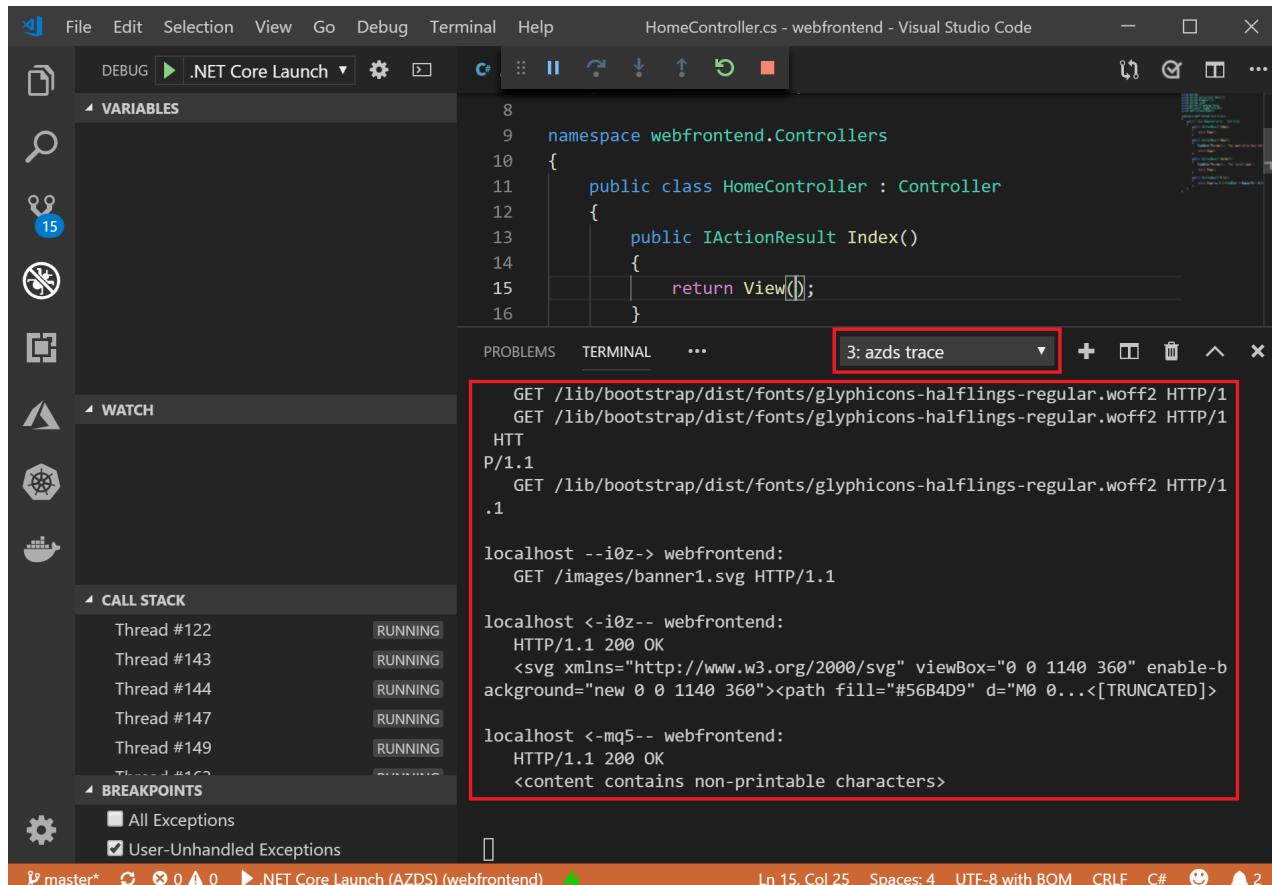
Scan the console output for the *Application started* message, confirming that the `up` command has completed:

```
Service 'webfrontend' port 80 (TCP) is available at 'http://localhost:<port>'  
Service 'webfrontend' port 'http' is available at http://webfrontend.1234567890abcdef1234.eus.azds.io/  
Microsoft (R) Build Engine version 15.9.20+g88f5fadfbe for .NET Core  
Copyright (C) Microsoft Corporation. All rights reserved.  
  
webfrontend -> /src/bin/Debug/netcoreapp2.2/webfrontend.dll  
webfrontend -> /src/bin/Debug/netcoreapp2.2/webfrontend.Views.dll  
  
Build succeeded.  
  0 Warning(s)  
  0 Error(s)  
  
Time Elapsed 00:00:00.94  
[...]  
webfrontend-5798f9dc44-99fsd: Now listening on: http://[::]:80  
webfrontend-5798f9dc44-99fsd: Application started. Press Ctrl+C to shut down.
```

Identify the public URL for the service in the output from the `up` command. It ends in `.azds.io`. In the above

example, the public URL is `http://webfrontend.1234567890abcdef1234.eus.azds.io/`.

To see your web app, open the public URL in a browser. Also, notice `stdout` and `stderr` output is streamed to the `azds trace` terminal window as you interact with your web app. You'll also see tracking information for HTTP requests as they go through the system. This makes it easier for you to track complex multi-service calls during development. The instrumentation added by Dev Spaces provides this request tracking.



#### NOTE

In addition to the public URL, you can use the alternative `http://localhost:<portnumber>` URL that is displayed in the console output. If you use the localhost URL, it may seem like the container is running locally, but actually it is running in AKS. Azure Dev Spaces uses Kubernetes *port-forward* functionality to map the localhost port to the container running in AKS. This facilitates interacting with the service from your local machine.

#### Update a content file

Azure Dev Spaces isn't just about getting code running in Kubernetes - it's about enabling you to quickly and iteratively see your code changes take effect in a Kubernetes environment in the cloud.

1. Locate the file `./Views/Home/Index.cshtml` and make an edit to the HTML. For example, change line 73 that reads `<h2>Application uses</h2>` to something like:

```
<h2>Hello k8s in Azure!</h2>
```

2. Save the file. Moments later, in the Terminal window you'll see a message saying a file in the running container was updated.
3. Go to your browser and refresh the page. You should see the web page display the updated HTML.

What happened? Edits to content files, like HTML and CSS, don't require recompilation in a .NET Core web app, so an active `azds up` command automatically syncs any modified content files into the running container in Azure, so

you can see your content edits right away.

## Update a code file

Updating code files requires a little more work, because a .NET Core app needs to rebuild and produce updated application binaries.

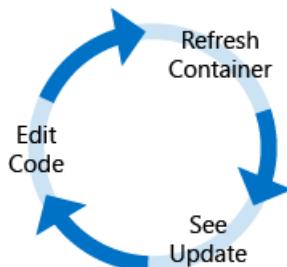
1. In the terminal window, press `Ctrl+C` (to stop `azds up`).
2. Open the code file named `Controllers/HomeController.cs`, and edit the message that the About page will display: `ViewData["Message"] = "Your application description page.";`
3. Save the file.
4. Run `azds up` in the terminal window.

This command rebuilds the container image and redeploys the Helm chart. To see your code changes take effect in the running application, go to the About menu in the web app.

But there is an even *faster method* for developing code, which you'll explore in the next section.

## Debug a container in Kubernetes

In this section, you'll use VS Code to directly debug our container running in Azure. You'll also learn how to get a faster edit-run-test loop.



### NOTE

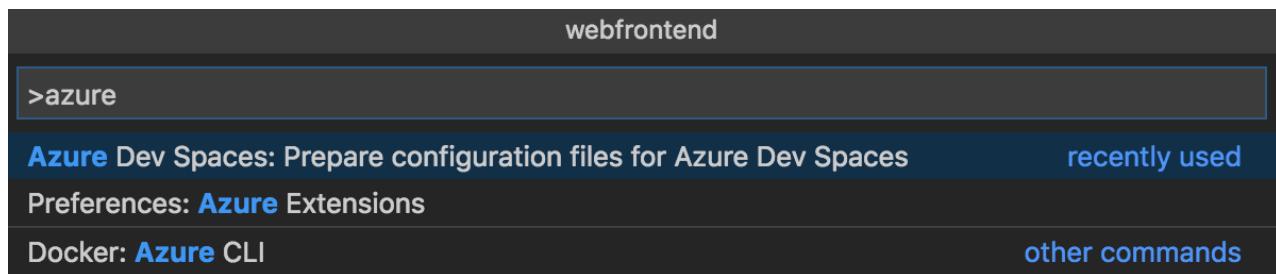
If you get stuck at any time, see the [Troubleshooting](#) section, or post a comment on this page.

### Initialize debug assets with the VS Code extension

You first need to configure your code project so VS Code will communicate with our dev space in Azure. The VS Code extension for Azure Dev Spaces provides a helper command to set up debug configuration.

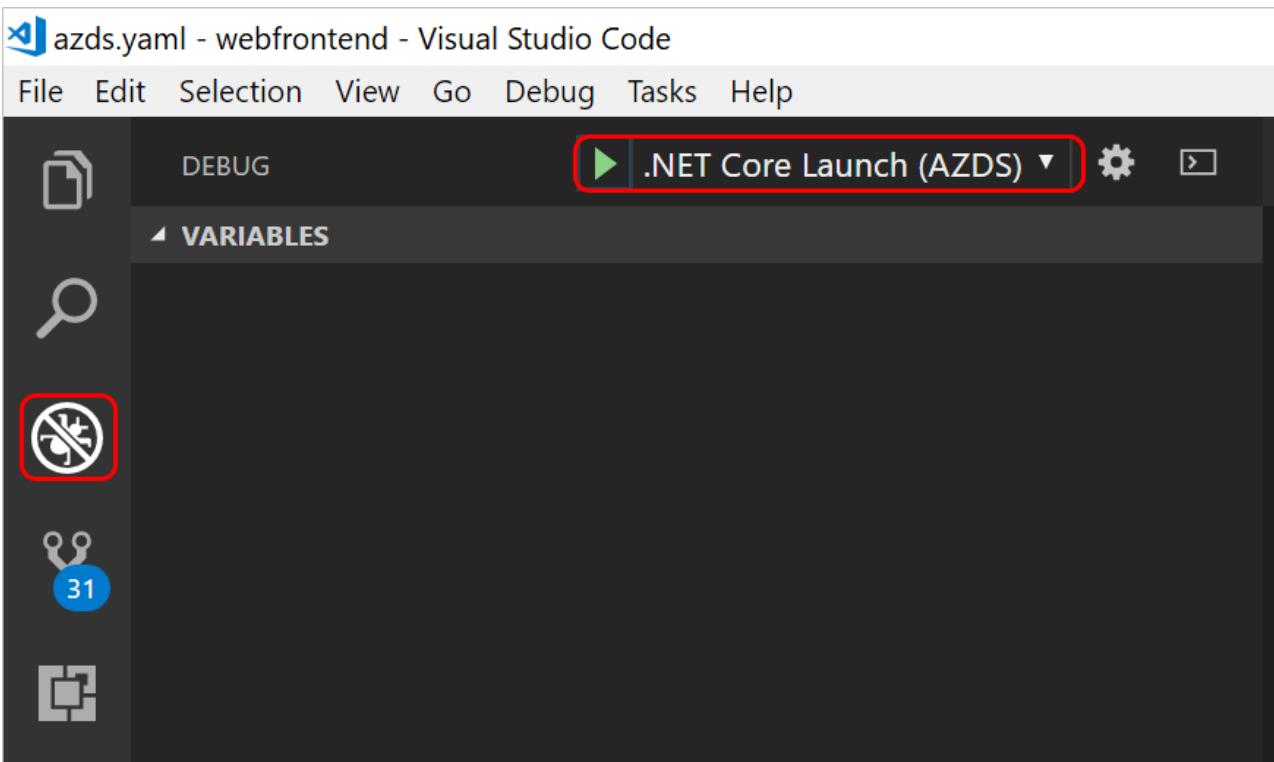
Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command: `Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces`.

This adds debug configuration for Azure Dev Spaces under the `.vscode` folder. This command is not to be confused with the `azds prep` command, which configures the project for deployment.



### Select the AZDS debug configuration

1. To open the Debug view, click on the Debug icon in the **Activity Bar** on the side of VS Code.
2. Select **.NET Core Launch (AZDS)** as the active debug configuration.



#### NOTE

If you don't see any Azure Dev Spaces commands in the Command Palette, ensure you have installed the VS Code extension for Azure Dev Spaces. Be sure the workspace you opened in VS Code is the folder that contains `azds.yaml`.

### Debug the container in Kubernetes

Hit **F5** to debug your code in Kubernetes.

As with the `up` command, code is synced to the dev space, and a container is built and deployed to Kubernetes. This time, of course, the debugger is attached to the remote container.

#### TIP

The VS Code status bar will turn orange, indicating that the debugger is attached. It will also display a clickable URL, which you can use to open your site.



Set a breakpoint in a server-side code file, for example within the `About()` function in the `Controllers/HomeController.cs` source file. Refreshing the browser page causes the breakpoint to hit.

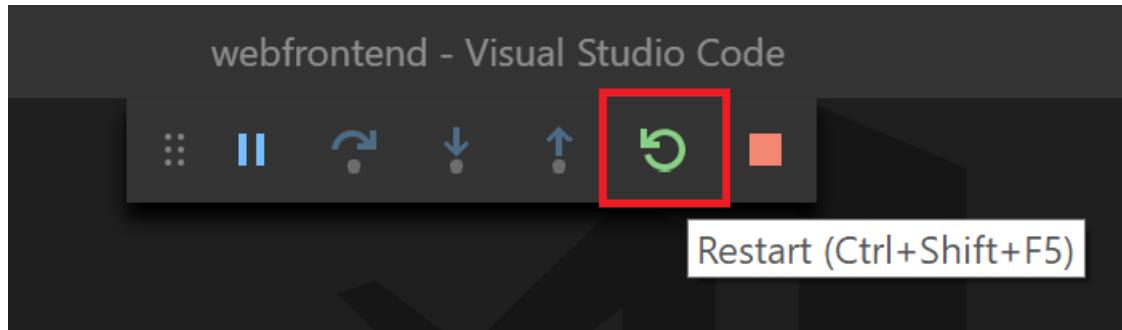
You have full access to debug information just like you would if the code was executing locally, such as the call stack, local variables, exception information, etc.

### Edit code and refresh

With the debugger active, make a code edit. For example, modify the About page's message in `Controllers/HomeController.cs`.

```
public IActionResult About()
{
    ViewData["Message"] = "My custom message in the About page.";
    return View();
}
```

Save the file, and in the **Debug actions pane**, click the **Restart** button.



Instead of rebuilding and redeploying a new container image each time code edits are made, which will often take considerable time, Azure Dev Spaces will incrementally recompile code within the existing container to provide a faster edit/debug loop.

Refresh the web app in the browser, and go to the About page. You should see your custom message appear in the UI.

**Now you have a method for rapidly iterating on code and debugging directly in Kubernetes!** Next, you'll see how you can create and call a second container.

## Next steps

[Learn about multi-service development](#)

# Running multiple dependent services: .NET Core and Visual Studio Code with Azure Dev Spaces

12/11/2019 • 3 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to develop multi-service applications using Azure Dev Spaces, along with some of the added benefits that Dev Spaces provides.

## Call a service running in a separate container

In this section, you will create a second service, `mywebapi`, and have `webfrontend` call it. Each service will run in separate containers. You'll then debug across both containers.



### Download sample code for `mywebapi`

For the sake of time, let's download sample code from a GitHub repository. Go to <https://github.com/Azure/dev-spaces> and select **Clone or Download** to download the GitHub repository. The code for this section is in `samples/dotnetcore/getting-started/mywebapi`.

### Run `mywebapi`

1. Open the folder `mywebapi` in a *separate VS Code window*.
2. Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command: `Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces`. This command is not to be confused with the `azds prep` command, which configures the project for deployment.
3. Hit F5, and wait for the service to build and deploy. You'll know it's ready when the *Application started. Press Ctrl+C to shut down.* message appears in the debug console.
4. The endpoint URL will look something like `http://localhost:<portnumber>`. **Tip: The VS Code status bar will turn orange and display a clickable URL.** It might seem like the container is running locally, but actually it is running in our dev space in Azure. The reason for the localhost address is because `mywebapi` has not defined any public endpoints and can only be accessed from within the Kubernetes instance. For your convenience, and to facilitate interacting with the private service from your local machine, Azure Dev Spaces creates a temporary SSH tunnel to the container running in Azure.
5. When `mywebapi` is ready, open your browser to the localhost address. Append `/api/values` to the URL to invoke the default GET API for the `ValuesController`.
6. If all the steps were successful, you should be able to see a response from the `mywebapi` service.

### Make a request from `webfrontend` to `mywebapi`

Let's now write code in `webfrontend` that makes a request to `mywebapi`.

1. Switch to the VS Code window for `webfrontend`.
2. Replace the code for the About method in `HomeController.cs`:

```

public async Task<IActionResult> About()
{
    ViewData["Message"] = "Hello from webfrontend";

    using (var client = new System.Net.Http.HttpClient())
    {
        // Call *mywebapi*, and display its response in the page
        var request = new System.Net.Http.HttpRequestMessage();
        request.RequestUri = new Uri("http://mywebapi/api/values/1");
        if (this.Request.Headers.ContainsKey("azds-route-as"))
        {
            // Propagate the dev space routing header
            request.Headers.Add("azds-route-as", this.Request.Headers["azds-route-as"] as
IEquatable<string>);
        }
        var response = await client.SendAsync(request);
        ViewData["Message"] += " and " + await response.Content.ReadAsStringAsync();
    }

    return View();
}

```

The preceding code example forwards the `azds-route-as` header from the incoming request to the outgoing request. You'll see later how this helps teams with [collaborative development](#).

### Debug across multiple services

- At this point, `mywebapi` should still be running with the debugger attached. If it is not, hit F5 in the `mywebapi` project.
- Set a breakpoint inside the `Get(int id)` method that handles `api/values/{id}` GET requests. This is around [line 23 in the \*Controllers/ValuesController.cs\* file](#).
- In the `webfrontend` project, set a breakpoint just before it sends a GET request to `mywebapi/api/values`. This is around line 32 in the [\*Controllers/HomeController.cs\* file](#) that you modified in the previous section.
- Hit F5 in the `webfrontend` project.
- Invoke the web app, and step through code in both services.
- In the web app, the About page will display a message concatenated by the two services: "Hello from webfrontend and Hello from mywebapi."

### Well done!

You now have a multi-container application where each container can be developed and deployed separately.

## Next steps

[Learn about team development in Dev Spaces](#)

# Team development using .NET Core and Visual Studio Code with Azure Dev Spaces

12/11/2019 • 9 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how a team of developers can simultaneously collaborate in the same Kubernetes cluster using Dev Spaces.

## Learn about team development

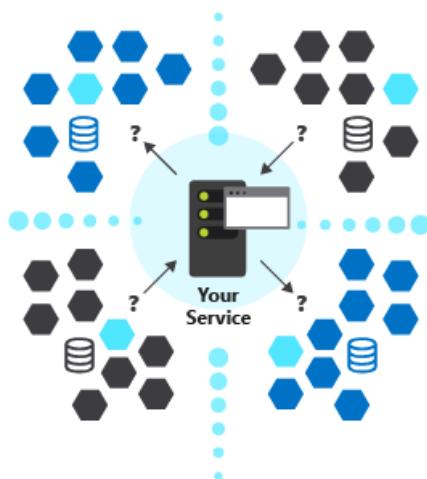
So far you've been running your application's code as if you were the only developer working on the app. In this section, you'll learn how Azure Dev Spaces streamlines team development:

- Enable a team of developers to work in the same environment, by working in a shared dev space or in distinct dev spaces as needed.
- Supports each developer iterating on their code in isolation and without fear of breaking others.
- Test code end-to-end, prior to code commit, without having to create mocks or simulate dependencies.

### Challenges with developing microservices

Your sample application isn't very complex at the moment. But in real-world development, challenges soon emerge as you add more services and the development team grows. It can become unrealistic to run everything locally for development.

- Your development machine may not have enough resources to run every service you need at once.
- Some services may need to be publicly reachable. For example, a service may need to have an endpoint that responds to a webhook.
- If you want to run a subset of services, you have to know the full dependency hierarchy between all your services. Determining this can be difficult, especially as your number of services increase.
- Some developers resort to simulating, or mocking up, many of their service dependencies. This approach can help, but managing those mocks can soon impact development cost. Plus, this approach leads to your development environment looking very different from production, which allows subtle bugs to creep in.
- It follows that doing any type of integration testing becomes difficult. Integration testing can only realistically happen post-commit, which means you see problems later in the development cycle.



### Work in a shared dev space

With Azure Dev Spaces, you can set up a *shared* dev space in Azure. Each developer can focus on just their part of the application, and can iteratively develop *pre-commit code* in a dev space that already contains all the other services and cloud resources that their scenarios depend on. Dependencies are always up-to-date, and developers are working in a way that mirrors production.

## Work in your own space

As you develop code for your service, and before you're ready to check it in, code often won't be in a good state. You're still iteratively shaping it, testing it, and experimenting with solutions. Azure Dev Spaces provides the concept of a **space**, which allows you to work in isolation, and without the fear of breaking your team members.

# Use Dev Spaces for team development

Let's demonstrate these ideas with a concrete example using our *webfrontend -> mywebapi* sample application. We'll imagine a scenario where a developer, Scott, needs to make a change to the *mywebapi* service, and *only* that service. The *webfrontend* won't need to change as part of Scott's update.

*Without* using Dev Spaces, Scott would have a few ways to develop and test his update, none of which are ideal:

- Run ALL components locally. This requires a more powerful development machine with Docker installed, and potentially MiniKube.
- Run ALL components in an isolated namespace on the Kubernetes cluster. Since *webfrontend* isn't changing, this is a waste of cluster resources.
- ONLY run *mywebapi*, and make manual REST calls to test. This doesn't test the full end-to-end flow.
- Add development-focused code to *webfrontend* that allows the developer to send requests to a different instance of *mywebapi*. This complicates the *webfrontend* service.

## Set up your baseline

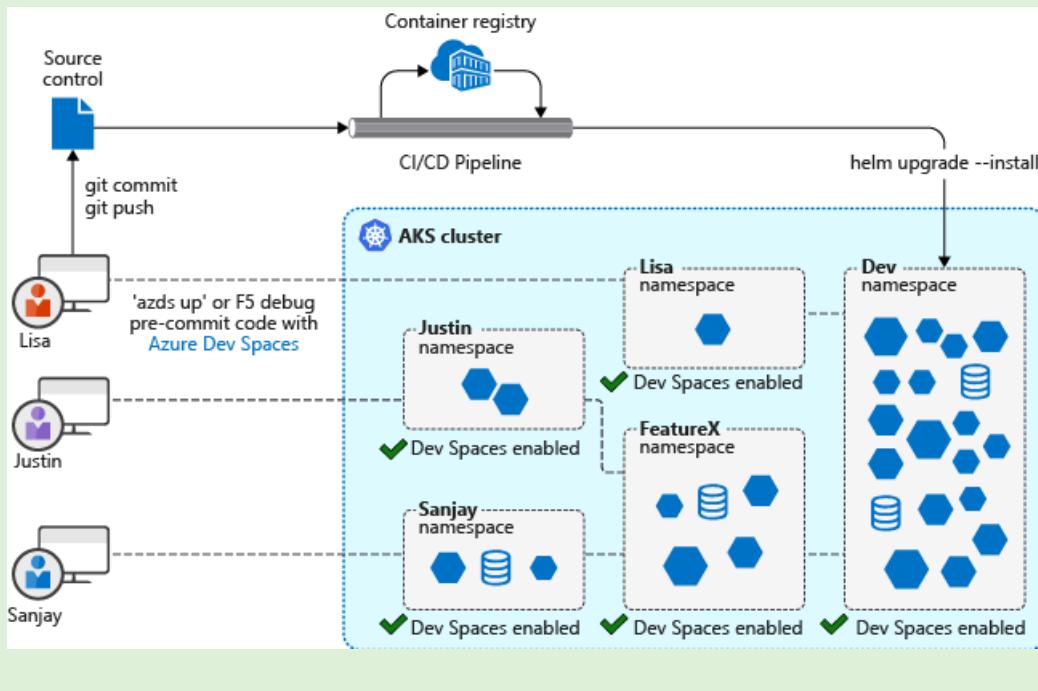
First we'll need to deploy a baseline of our services. This deployment will represent the "last known good" so you can easily compare the behavior of your local code vs. the checked-in version. We'll then create a child space based on this baseline so we can test our changes to *mywebapi* within the context of the larger application.

1. Clone the [Dev Spaces sample application](#): `git clone https://github.com/Azure/dev-spaces && cd dev-spaces`
2. Checkout the remote branch *azds\_updates*: `git checkout -b azds_updates origin/azds_updates`
3. Select the *dev* space: `azds space select --name dev`. When prompted to select a parent dev space, select *<none>*.
4. Navigate to the *mywebapi* directory and execute: `azds up -d`
5. Navigate to the *webfrontend* directory and execute: `azds up -d`
6. Execute `azds list-uris` to see the public endpoint for *webfrontend*

## TIP

The above steps manually set up a baseline, but we recommend teams use CI/CD to automatically keep your baseline up to date with committed code.

Check out our [guide to setting up CI/CD with Azure DevOps](#) to create a workflow similar to the following diagram.



At this point your baseline should be running. Run the `azds list-up --all` command, and you'll see output similar to the following:

```
$ azds list-up --all
```

Name	DevSpace	Type	Updated	Status
mywebapi	dev	Service	3m ago	Running
mywebapi-56c8f45d9-zs4mw	dev	Pod	3m ago	Running
webfrontend	dev	Service	1m ago	Running
webfrontend-6b6ddbb98f-fgvnc	dev	Pod	1m ago	Running

The DevSpace column shows that both services are running in a space named *dev*. Anyone who opens the public URL and navigates to the web app will invoke the checked-in code path that runs through both services. Now suppose you want to continue developing *mywebapi*. How can you make code changes and test them and not interrupt other developers who are using the dev environment? To do that, you'll set up your own space.

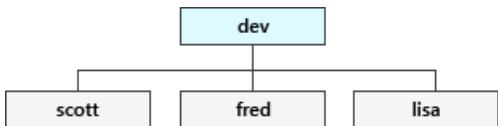
## Create a dev space

To run your own version of *mywebapi* in a space other than *dev*, you can create your own space by using the following command:

```
azds space select --name scott
```

When prompted, select *dev* as the **parent dev space**. This means our new space, *dev/scott*, will derive from the space *dev*. We'll shortly see how this will help us with testing.

Keeping with our introductory hypothetical, we've used the name *scott* for the new space so peers can identify who is working in it. But it can be called anything you like, and be flexible about what it means, like *sprint4* or *demo*. Whatever the case, *dev* serves as the baseline for all developers working on a piece of this application:



Run the `azds space list` command to see a list of all the spaces in the dev environment. The *Selected* column indicates which space you currently have selected (true/false). In your case, the space named *dev/scott* was automatically selected when it was created. You can select another space at any time with the `azds space select` command.

Let's see it in action.

### Make a code change

Go to the VS Code window for `mywebapi` and make a code edit to the `string Get(int id)` method in `Controllers/ValuesController.cs`, for example:

```
[HttpGet("{id}")]
public string Get(int id)
{
    return "mywebapi now says something new";
}
```

### Run the service

To run the service, hit F5 (or type `azds up` in the Terminal Window) to run the service. The service will automatically run in your newly selected space *dev/scott*. Confirm that your service is running in its own space by running `azds list-up`:

```
$ azds list-up

Name          DevSpace  Type      Updated   Status
mywebapi     scott     Service   3m ago   Running
webfrontend  dev       Service   26m ago  Running
```

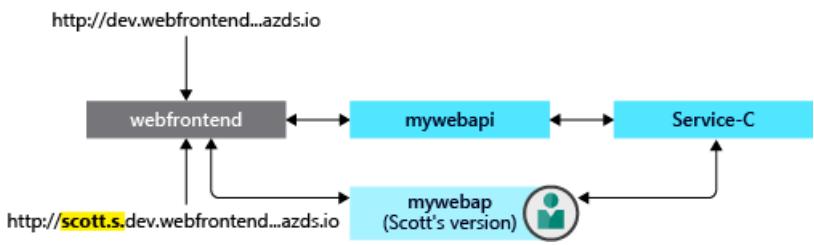
Notice an instance of *mywebapi* is now running in the *dev/scott* space. The version running in *dev* is still running but it is not listed.

List the URLs for the current space by running `azds list-uris`.

```
$ azds list-uris

Uri                                         Status
-----
http://localhost:53831 => mywebapi.scott:80           Tunneled
http://scott.s.dev.webfrontend.6364744826e042319629.ce.azds.io/ Available
```

Notice the public access point URL for *webfrontend* is prefixed with *scott.s*. This URL is unique to the *dev/scott* space. This URL prefix tells the Ingress controller to route requests to the *dev/scott* version of a service. When a request with this URL is handled by Dev Spaces, the Ingress Controller first tries to route the request to the *webfrontend* service in the *dev/scott* space. If that fails, the request will be routed to the *webfrontend* service in the *dev* space as a fallback. Also notice there is a localhost URL to access the service over localhost using the Kubernetes *port-forward* functionality. For more information about URLs and routing in Azure Dev Spaces, see [How Azure Dev Spaces works and is configured](#).



This built-in feature of Azure Dev Spaces lets you test code in a shared space without requiring each developer to re-create the full stack of services in their space. This routing requires your app code to forward propagation headers, as illustrated in the previous step of this guide.

### Test code in a space

To test your new version of *mywebapi* with *webfrontend*, open your browser to the public access point URL for *webfrontend* and go to the About page. You should see your new message displayed.

Now, remove the "scott.s." part of the URL, and refresh the browser. You should see the old behavior (with the *mywebapi* version running in *dev*).

Once you have a *dev* space, which always contains your latest changes, and assuming your application is designed to take advantage of DevSpace's space-based routing as described in this tutorial section, hopefully it becomes easy to see how Dev Spaces can greatly assist in testing new features within the context of the larger application. Rather than having to deploy *all* services to your private space, you can create a private space that derives from *dev*, and only "up" the services you're actually working on. The Dev Spaces routing infrastructure will handle the rest by utilizing as many services out of your private space as it can find, while defaulting back to the latest version running in the *dev* space. And better still, *multiple* developers can actively develop different services at the same time in their own space without disrupting each other.

### Well done!

You've completed the getting started guide! You learned how to:

- Set up Azure Dev Spaces with a managed Kubernetes cluster in Azure.
- Iteratively develop code in containers.
- Independently develop two separate services, and used Kubernetes' DNS service discovery to make a call to another service.
- Productively develop and test your code in a team environment.
- Establish a baseline of functionality using Dev Spaces to easily test isolated changes within the context of a larger microservice application

Now that you've explored Azure Dev Spaces, [share your dev space with a team member](#) and begin collaborating.

## Clean up

To completely delete an Azure Dev Spaces instance on a cluster, including all the dev spaces and running services within it, use the `az aks remove-dev-spaces` command. Bear in mind that this action is irreversible. You can add support for Azure Dev Spaces again on the cluster, but it will be as if you are starting again. Your old services and spaces won't be restored.

The following example lists the Azure Dev Spaces controllers in your active subscription, and then deletes the Azure Dev Spaces controller that is associated with AKS cluster 'myaks' in resource group 'myaks-rg'.

```

azds controller list
az aks remove-dev-spaces --name myaks --resource-group myaks-rg

```

# Create a Kubernetes dev space: Visual Studio and .NET Core with Azure Dev Spaces

1/8/2020 • 5 minutes to read • [Edit Online](#)

In this guide, you will learn how to:

- Set up Azure Dev Spaces with a managed Kubernetes cluster in Azure.
- Iteratively develop code in containers using Visual Studio.
- Independently develop two separate services, and used Kubernetes' DNS service discovery to make a call to another service.
- Productively develop and test your code in a team environment.

## NOTE

If you get stuck at any time, see the [Troubleshooting](#) section.

## Create a Kubernetes cluster enabled for Azure Dev Spaces

1. Sign in to the Azure portal at <https://portal.azure.com>.
2. Choose **Create a resource** > search for **Kubernetes** > select **Kubernetes Service** > **Create**.

Complete the following steps under each heading of the *Create Kubernetes cluster* form and verify your selected [region supports Azure Dev Spaces](#).

- **PROJECT DETAILS:** select an Azure subscription and a new or existing Azure resource group.
- **CLUSTER DETAILS:** enter a name, region, version, and DNS name prefix for the AKS cluster.
- **SCALE:** select a VM size for the AKS agent nodes and the number of nodes. If you're getting started with Azure Dev Spaces, one node is enough to explore all the features. The node count can be easily adjusted any time after the cluster is deployed. Note that the VM size can't be changed once an AKS cluster has been created. However, once an AKS cluster has been deployed, you can easily create a new AKS cluster with larger VMs and use Dev Spaces to redeploy to that larger cluster if you need to scale up.

Home > New > Create Kubernetes cluster

## Create Kubernetes cluster

---

**Basics**   **Authentication**   **Networking**   **Monitoring**   **Tags**   **Review + create**

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. [Learn more about Azure Kubernetes Service](#)

**PROJECT DETAILS**

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

\* Subscription

\* Resource group  Create new  Use existing

\* Kubernetes cluster name

**CLUSTER DETAILS**

\* Region

\* Kubernetes version

\* DNS name prefix

**SCALE**

The number and size of nodes in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. You will not be able to change the node size after cluster creation, but you will be able to change the number of nodes in your cluster after creation. [Learn more about scaling in Azure Kubernetes Service](#)

\* Node size

[Change size](#)

\* Node count

---

[Review + create](#)   [Next: Authentication »](#)   [Download a template for automation](#)

Select **Next: Authentication** when complete.

3. Choose your desired setting for Role-based Access Control (RBAC). Azure Dev Spaces supports clusters with RBAC enabled, or disabled.

The **cluster infrastructure** service principal is used by the Kubernetes cluster to manage cloud resources attached to the cluster.

**Kubernetes authentication and authorization** is used by the Kubernetes cluster to control user access to the cluster as well as what the user may do once authenticated.

[Learn more about authentication and authorization in Azure Kubernetes Service](#)

#### CLUSTER INFRASTRUCTURE

\* Service principal [?](#)

(new) default service principal

[Configure service principal](#)

#### KUBERNETES AUTHENTICATION AND AUTHORIZATION

Enable RBAC [?](#)

No Yes

4. Select **Review + create** and then **Create** when complete.

## Get the Visual Studio tools

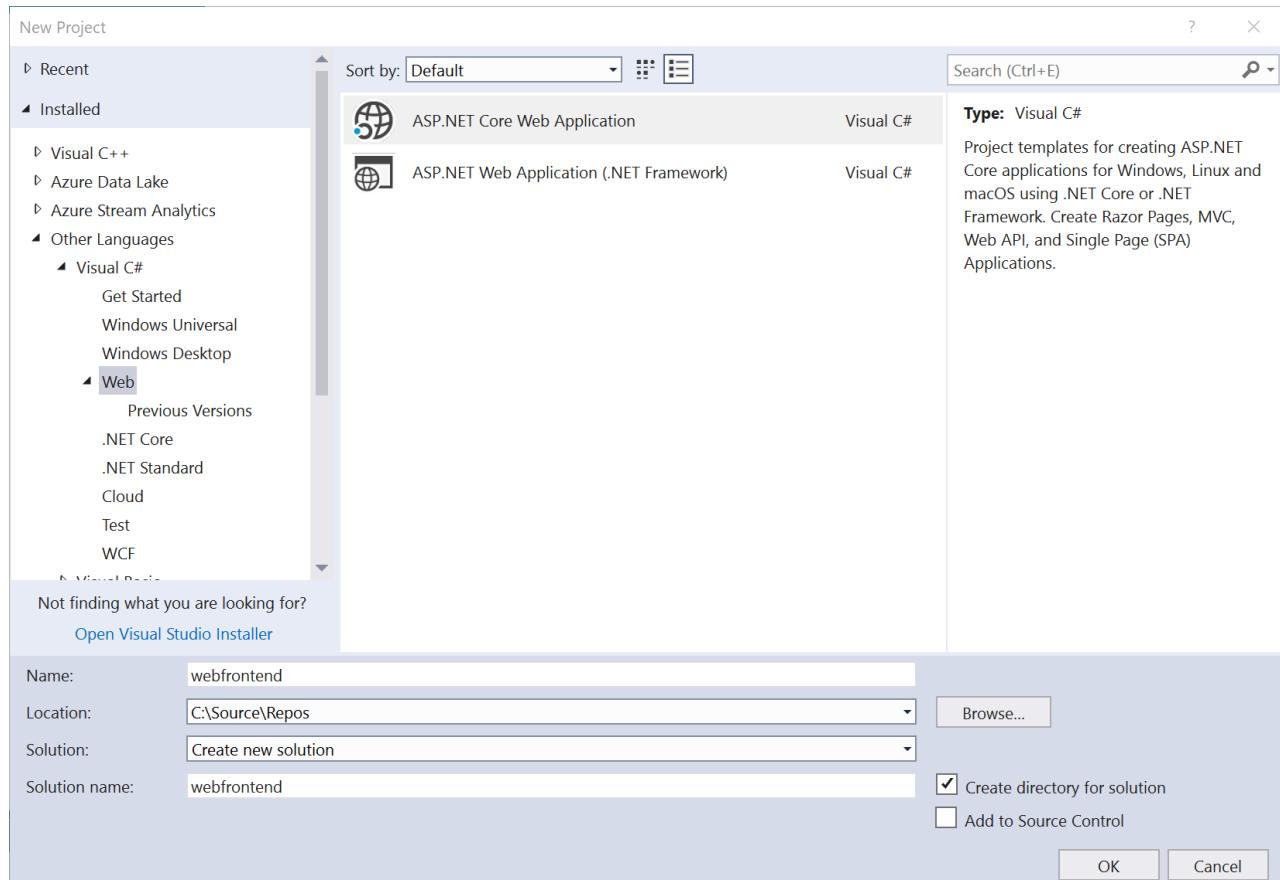
Install the latest version of [Visual Studio](#). For Visual Studio 2019 on Windows you need to install the Azure Development workload. For Visual Studio 2017 on Windows you need to install the ASP.NET and web development workload as well as [Visual Studio Tools for Kubernetes](#).

## Create a web app running in a container

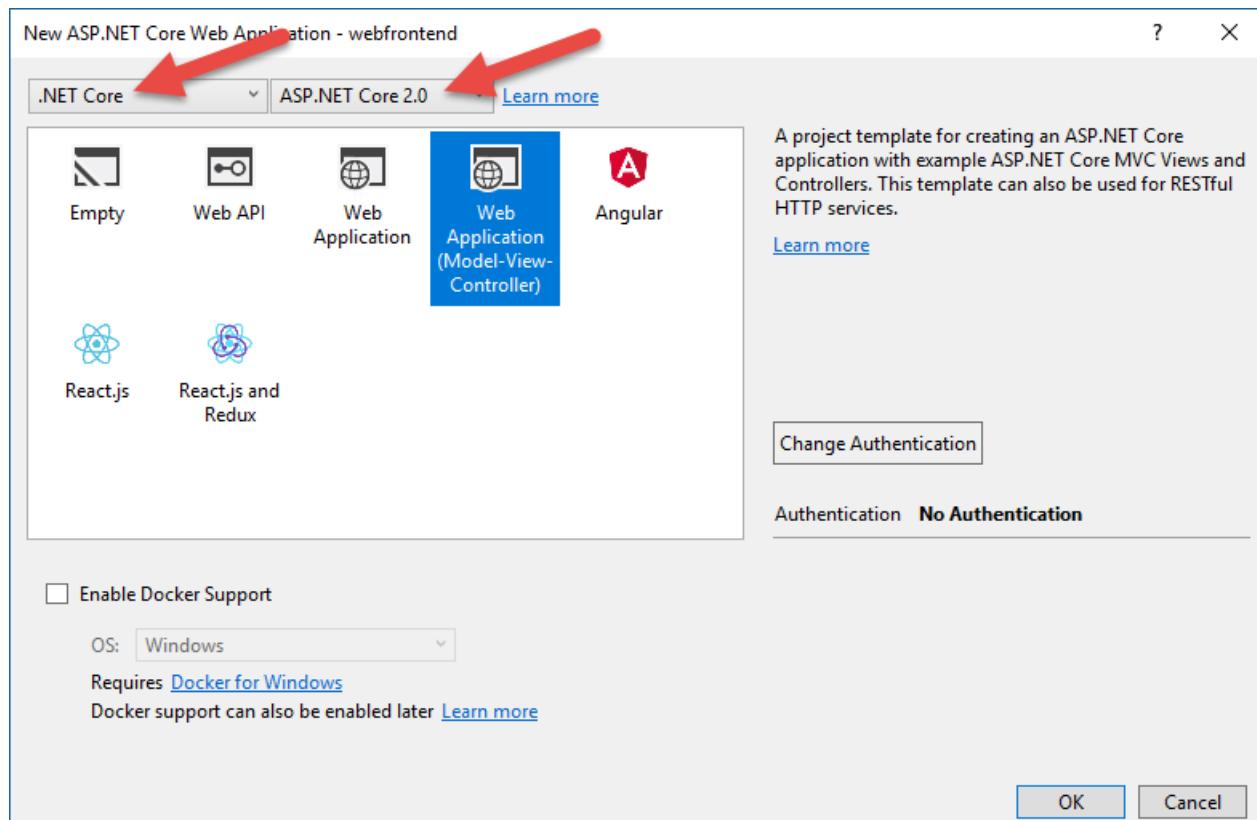
In this section, you'll create an ASP.NET Core web app and get it running in a container in Kubernetes.

### Create an ASP.NET web app

From within Visual Studio, create a new project. Currently, the project must be an **ASP.NET Core Web Application**. Name the project '**webfrontend**'.

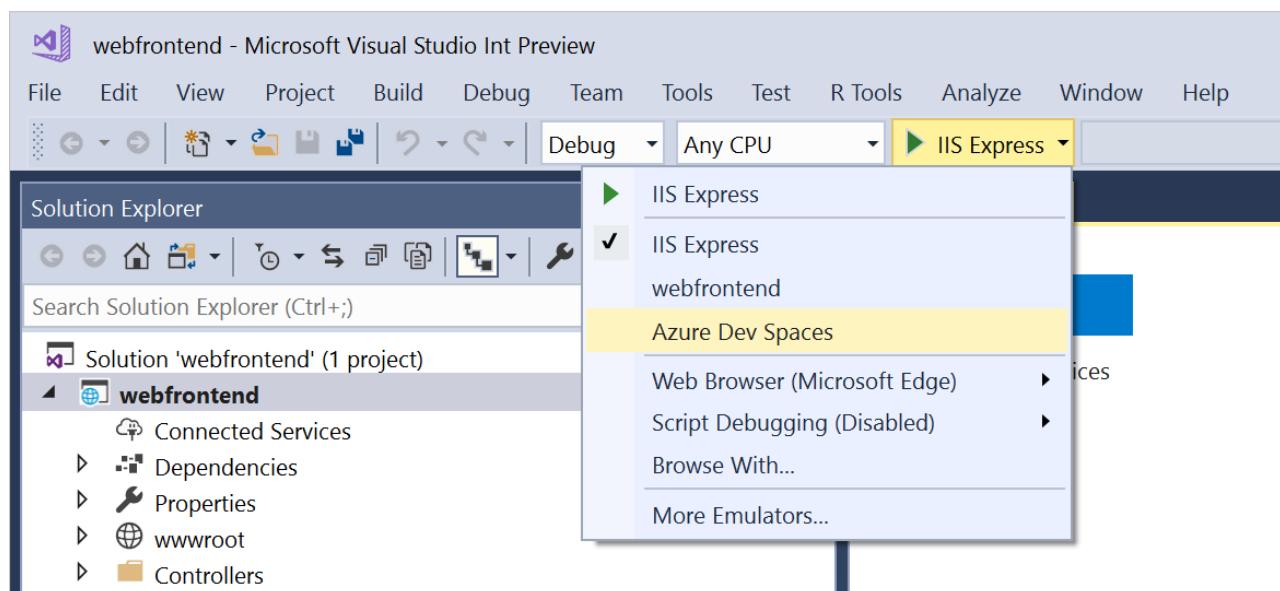


Select the **Web Application (Model-View-Controller)** template and be sure you're targeting **.NET Core** and **ASP.NET Core 2.0** in the two dropdowns at the top of the dialog. Click **OK** to create the project.



### Enable Dev Spaces for an AKS cluster

With the project you just created, select **Azure Dev Spaces** from the launch settings dropdown, as shown below.



In the dialog that is displayed next, make sure you are signed in with the appropriate account, and then either select an existing Kubernetes cluster.



Subscription:

Azure Kubernetes Service cluster:

MyAKS

[Create a new AKS cluster](#)

Space:

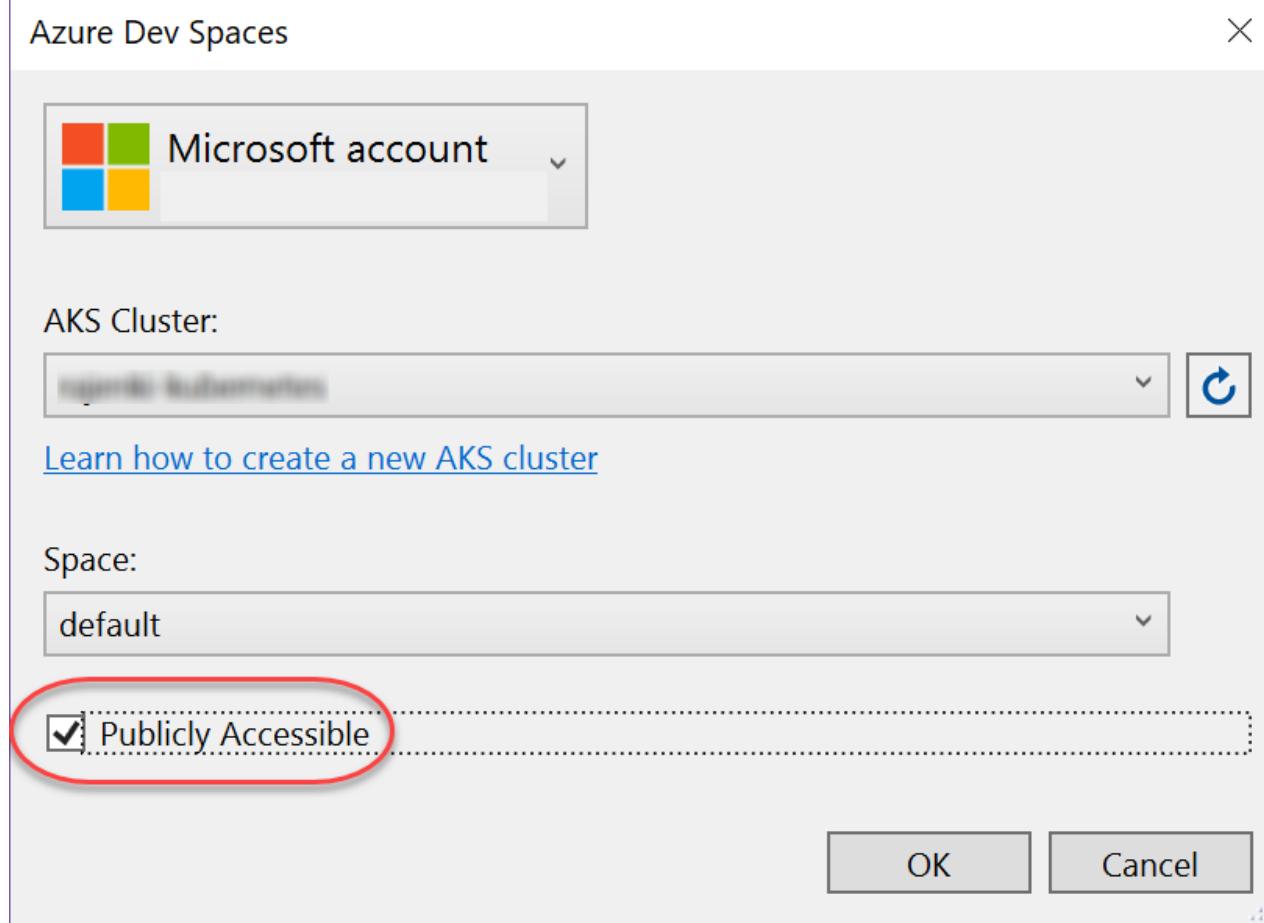
default

[Create a new space...](#) Publicly Accessible

OK

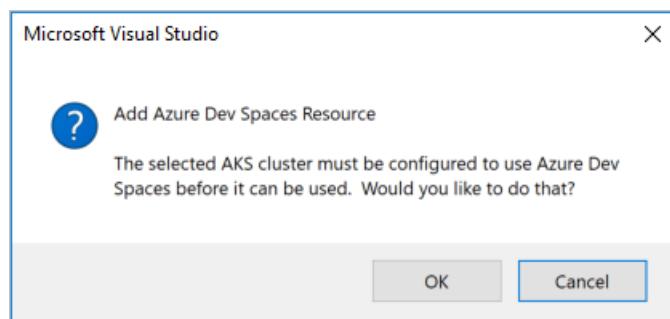
Cancel

Leave the **Space** dropdown defaulted to `default` for now. Later, you'll learn more about this option. Check the **Publicly Accessible** checkbox so the web app will be accessible via a public endpoint. This setting isn't required, but it will be helpful to demonstrate some concepts later in this walkthrough. But don't worry, in either case you will be able to debug your website using Visual Studio.



Click **OK** to select or create the cluster.

If you choose a cluster that hasn't been enabled to work with Azure Dev Spaces, you'll see a message asking if you want to configure it.

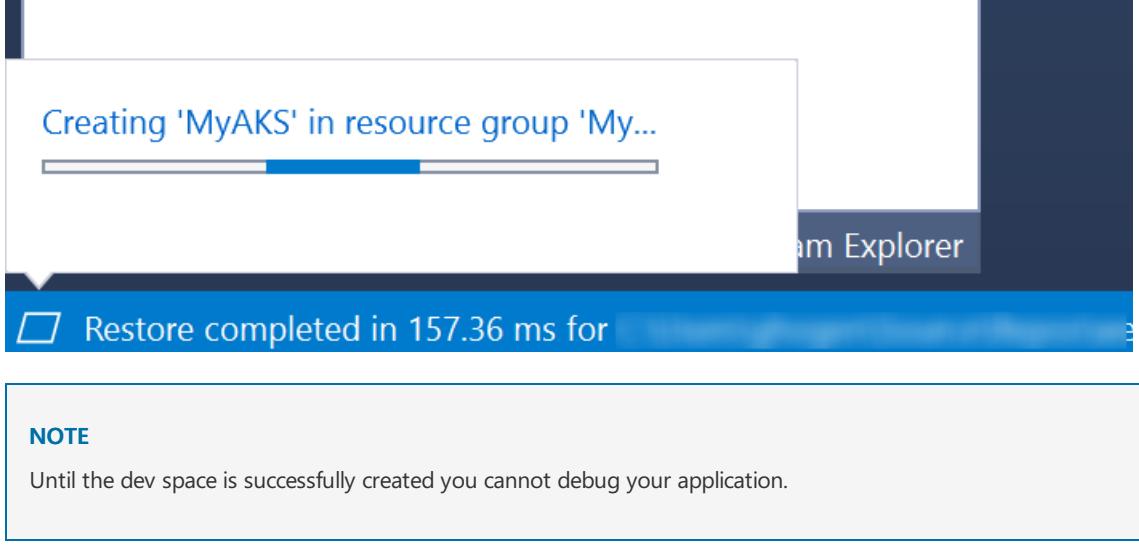


Choose **OK**.

#### IMPORTANT

The Azure Dev Spaces configuration process will remove the `azds` namespace in the cluster, if it exists.

A background task will be started to accomplish this. It will take a number of minutes to complete. To see if it's still being created, hover your pointer over the **Background tasks** icon in the bottom left corner of the status bar, as shown in the following image.



#### NOTE

Until the dev space is successfully created you cannot debug your application.

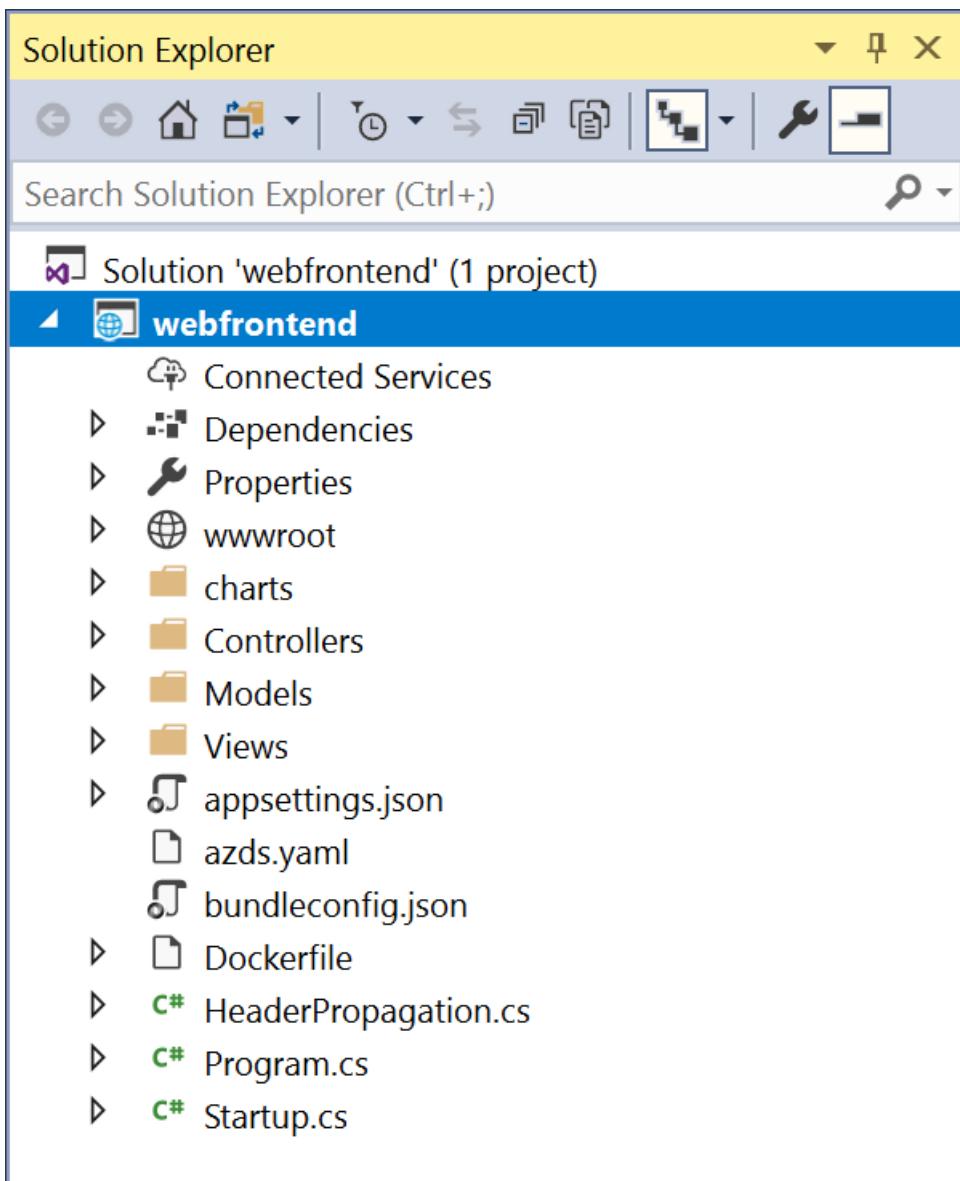
### Look at the files added to project

While you wait for the dev space to be created, look at the files that have been added to your project when you chose to use a dev space.

First, you can see a folder named `charts` has been added and within this folder a [Helm chart](#) for your application has been scaffolded. These files are used to deploy your application into the dev space.

You will see a file named `Dockerfile` has been added. This file has information needed to package your application in the standard Docker format.

Lastly, you will see a file named `azds.yaml`, which contains development-time configuration that is needed by the dev space.



## Debug a container in Kubernetes

Once the dev space is successfully created, you can debug the application. Set a breakpoint in the code, for example on line 20 in the file `HomeController.cs` where the `Message` variable is set. Click **F5** to start debugging.

Visual Studio will communicate with the dev space to build and deploy the application and then open a browser with the web app running. It might seem like the container is running locally, but actually it's running in the dev space in Azure. The reason for the localhost address is because Azure Dev Spaces creates a temporary SSH tunnel to the container running in AKS.

Click on the **About** link at the top of the page to trigger the breakpoint. You have full access to debug information just like you would if the code was executing locally, such as the call stack, local variables, exception information, and so on.

## Iteratively develop code

Azure Dev Spaces isn't just about getting code running in Kubernetes - it's about enabling you to quickly and iteratively see your code changes take effect in a Kubernetes environment in the cloud.

### Update a content file

1. Locate the file `./Views/Home/Index.cshtml` and make an edit to the HTML. For example, change line 73 that reads `<h2>Application uses</h2>` to something like:

```
<h2>Hello k8s in Azure!</h2>
```

2. Save the file.
3. Go to your browser and refresh the page. You should see the web page display the updated HTML.

What happened? Edits to content files, like HTML and CSS, don't require recompilation in a .NET Core web app, so an active F5 session automatically syncs any modified content files into the running container in AKS, so you can see your content edits right away.

### Update a code file

Updating code files requires a little more work, because a .NET Core app needs to rebuild and produce updated application binaries.

1. Stop the debugger in Visual Studio.
2. Open the code file named `Controllers/HomeController.cs`, and edit the message that the About page will display: `ViewData["Message"] = "Your application description page.";`
3. Save the file.
4. Press **F5** to start debugging again.

Instead of rebuilding and redeploying a new container image each time code edits are made, which will often take considerable time, Azure Dev Spaces will incrementally recompile code within the existing container to provide a faster edit/debug loop.

Refresh the web app in the browser, and go to the About page. You should see your custom message appear in the UI.

## Next steps

[Learn about multi-service development](#)

# Running multiple dependent services: .NET Core and Visual Studio with Azure Dev Spaces

12/11/2019 • 3 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to develop multi-service applications using Azure Dev Spaces, along with some of the added benefits that Dev Spaces provides.

## Call another container

In this section, you're going to create a second service, `mywebapi`, and have `webfrontend` call it. Each service will run in separate containers. You'll then debug across both containers.

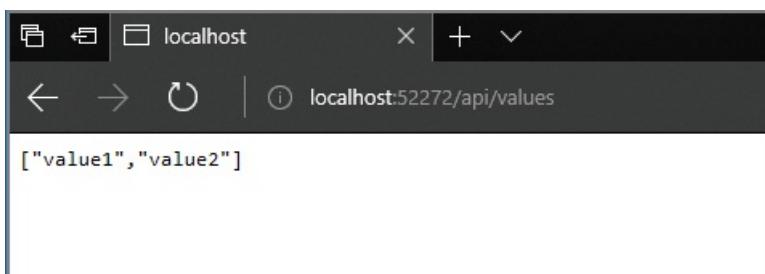


### Download sample code for `mywebapi`

For the sake of time, let's download sample code from a GitHub repository. Go to <https://github.com/Azure/dev-spaces> and select **Clone or Download** to download the GitHub repository. The code for this section is in `samples/dotnetcore/getting-started/mywebapi`.

### Run `mywebapi`

1. Open the project `mywebapi` in a *separate Visual Studio window*.
2. Select **Azure Dev Spaces** from the launch settings dropdown as you did previously for the `webfrontend` project. Rather than creating a new AKS cluster this time, select the same one you already created. As before, leave the Space defaulted to `default` and click **OK**. In the Output window, you may notice Visual Studio starts to "warm up" this new service in your dev space in order to speed up things when you start debugging.
3. Hit F5, and wait for the service to build and deploy. You'll know it's ready when the Visual Studio status bar turns orange
4. Take note of the endpoint URL displayed in the **Azure Dev Spaces for AKS** pane in the **Output** window. It will look something like `http://localhost:<portnumber>`. It might seem like the container is running locally, but actually it's running in the dev space in Azure.
5. When `mywebapi` is ready, open your browser to the localhost address and append `/api/values` to the URL to invoke the default GET API for the `ValuesController`.
6. If all the steps were successful, you should be able to see a response from the `mywebapi` service that looks like this.



### Make a request from `webfrontend` to `mywebapi`

Let's now write code in `webfrontend` that makes a request to `mywebapi`. Switch to the Visual Studio window that has the `webfrontend` project. In the `HomeController.cs` file, replace the code for the About method with the following code:

```
public async Task<IActionResult> About()
{
    ViewData["Message"] = "Hello from webfrontend";

    using (var client = new System.Net.Http.HttpClient())
    {
        // Call *mywebapi*, and display its response in the page
        var request = new System.Net.Http.HttpRequestMessage();
        request.RequestUri = new Uri("http://mywebapi/api/values/1");
        if (this.Request.Headers.ContainsKey("azds-route-as"))
        {
            // Propagate the dev space routing header
            request.Headers.Add("azds-route-as", this.Request.Headers["azds-route-as"] as
IEquatable<string>);
        }
        var response = await client.SendAsync(request);
        ViewData["Message"] += " and " + await response.Content.ReadAsStringAsync();
    }

    return View();
}
```

The preceding code example forwards the `azds-route-as` header from the incoming request to the outgoing request. You'll see later how this facilitates a more productive development experience in [team scenarios](#).

### Debug across multiple services

1. At this point, `mywebapi` should still be running with the debugger attached. If it is not, hit F5 in the `mywebapi` project.
2. Set a breakpoint in the `Get(int id)` method in the `Controllers/ValuesController.cs` file that handles `api/values/{id}` GET requests.
3. In the `webfrontend` project where you pasted the above code, set a breakpoint just before it sends a GET request to `mywebapi/api/values`.
4. Hit F5 in the `webfrontend` project. Visual Studio will again open a browser to the appropriate localhost port and the web app will be displayed.
5. Click on the “**About**” link at the top of the page to trigger the breakpoint in the `webfrontend` project.
6. Hit F10 to proceed. The breakpoint in the `mywebapi` project will now be triggered.
7. Hit F5 to proceed and you will be back in the code in the `webfrontend` project.
8. Hitting F5 one more time will complete the request and return a page in the browser. In the web app, the About page will display a message concatenated by the two services: “Hello from webfrontend and Hello from mywebapi.”

### Well done!

You now have a multi-container application where each container can be developed and deployed separately.

## Next steps

[Learn about team development in Dev Spaces](#)

# Team development using .NET Core and Visual Studio with Azure Dev Spaces

12/11/2019 • 8 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how a team of developers can simultaneously collaborate in the same Kubernetes cluster using Dev Spaces.

## Learn about team development

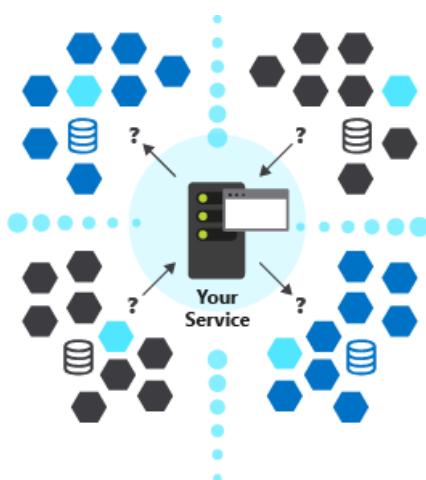
So far you've run your application's code as if you were the only developer working on the app. In this section, you'll learn how Azure Dev Spaces streamlines team development:

- Enable a team of developers to work in the same environment, by working in a shared dev space or in distinct dev spaces as needed.
- Supports each developer iterating on their code in isolation and without fear of breaking others.
- Test code end-to-end, prior to code commit, without having to create mocks or simulate dependencies.

### Challenges with developing microservices

Your sample application isn't complex at the moment. But in real-world development, challenges soon emerge as you add more services and the development team grows.

- Your development machine may not have enough resources to run every service you need at once.
- Some services may need to be publicly reachable. For example, a service may need to have an endpoint that responds to a webhook.
- If you want to run a subset of services, you have to know the full dependency hierarchy between all your services. Determining this hierarchy can be difficult, especially as your number of services increase.
- Some developers resort to simulating, or mocking up, many of their service dependencies. This approach can help, but managing those mocks can soon impact development cost. Plus, this approach leads to your development environment looking different from production, which can lead to subtle bugs occurring.
- It follows that doing any type of integration testing becomes difficult. Integration testing can only realistically happen post-commit, which means you see problems later in the development cycle.



### Work in a shared dev space

With Azure Dev Spaces, you can set up a *shared* dev space in Azure. Each developer can focus on just their part of

the application, and can iteratively develop *pre-commit code* in a dev space that already contains all the other services and cloud resources that their scenarios depend on. Dependencies are always up-to-date, and developers are working in a way that mirrors production.

## Work in your own space

As you develop code for your service, and before you're ready to check it in, code often won't be in a good state. You're still iteratively shaping it, testing it, and experimenting with solutions. Azure Dev Spaces provides the concept of a **space**, which allows you to work in isolation, and without the fear of breaking your team members.

# Use Dev Spaces for team development

Let's demonstrate these ideas with a concrete example using our *webfrontend -> mywebapi* sample application. We'll imagine a scenario where a developer, Scott, needs to make a change to the *mywebapi* service, and *only* that service. The *webfrontend* won't need to change as part of Scott's update.

Without using Dev Spaces, Scott would have a few ways to develop and test his update, none of which are ideal:

- Run ALL components locally, which requires a more powerful development machine with Docker installed, and potentially MiniKube.
- Run ALL components in an isolated namespace on the Kubernetes cluster. Since *webfrontend* isn't changing, using an isolated namespace is a waste of cluster resources.
- ONLY run *mywebapi*, and make manual REST calls to test. This type of testing doesn't test the full end-to-end flow.
- Add development-focused code to *webfrontend* that allows the developer to send requests to a different instance of *mywebapi*. Adding this code complicates the *webfrontend* service.

## Set up your baseline

First we'll need to deploy a baseline of our services. This deployment will represent the "last known good" so you can easily compare the behavior of your local code vs. the checked-in version. We'll then create a child space based on this baseline so we can test our changes to *mywebapi* within the context of the larger application.

1. Clone the [Dev Spaces sample application](#): `git clone https://github.com/Azure/dev-spaces && cd dev-spaces`
2. Check out the remote branch *azds\_updates*: `git checkout -b azds_updates origin/azds_updates`
3. Close any F5/debug sessions for both services, but keep the projects open in their Visual Studio windows.
4. Switch to the Visual Studio window with the *mywebapi* project.
5. Right-click on the project in **Solution Explorer** and select **Properties**.
6. Select the **Debug** tab on the left to show the Azure Dev Spaces settings.
7. Select **Change** to create the space that will be used when you F5 or Ctrl+F5 the service.
8. In the Space dropdown, select **<Create New Space...>**.
9. Make sure the parent space is set to **<none>**, and enter space name **dev**. Click OK.
10. Press Ctrl+F5 to run *mywebapi* without the debugger attached.
11. Switch to the Visual Studio window with the *webfrontend* project and press Ctrl+F5 to run it as well.

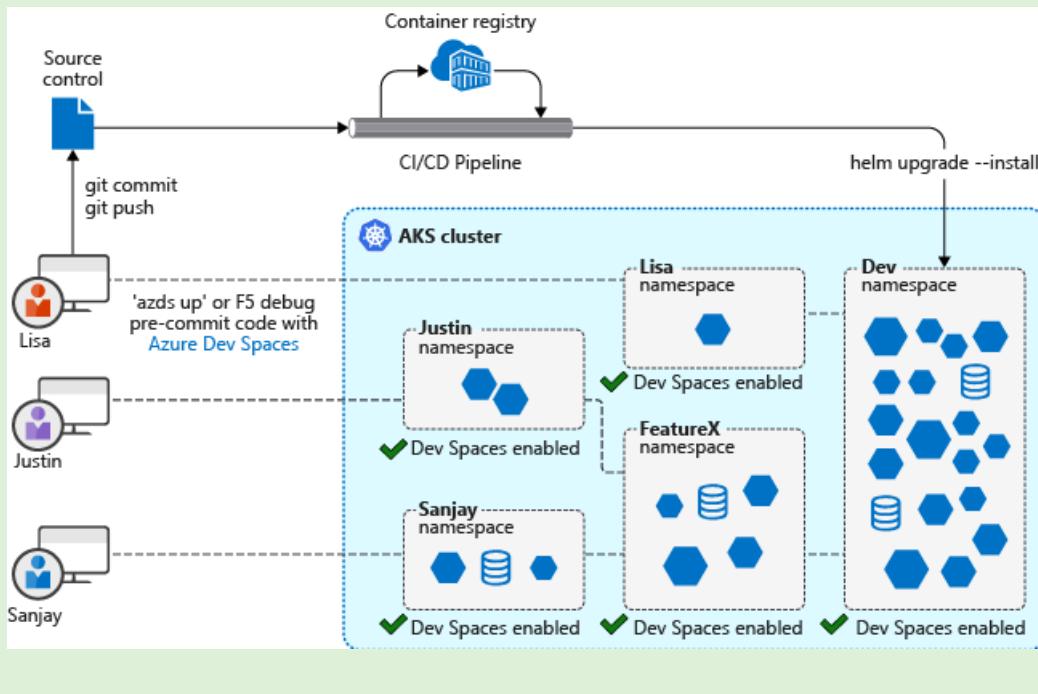
### NOTE

It is sometimes necessary to refresh your browser after the web page is initially displayed following a Ctrl+F5.

## TIP

The above steps manually set up a baseline, but we recommend teams use CI/CD to automatically keep your baseline up to date with committed code.

Check out our [guide to setting up CI/CD with Azure DevOps](#) to create a workflow similar to the following diagram.



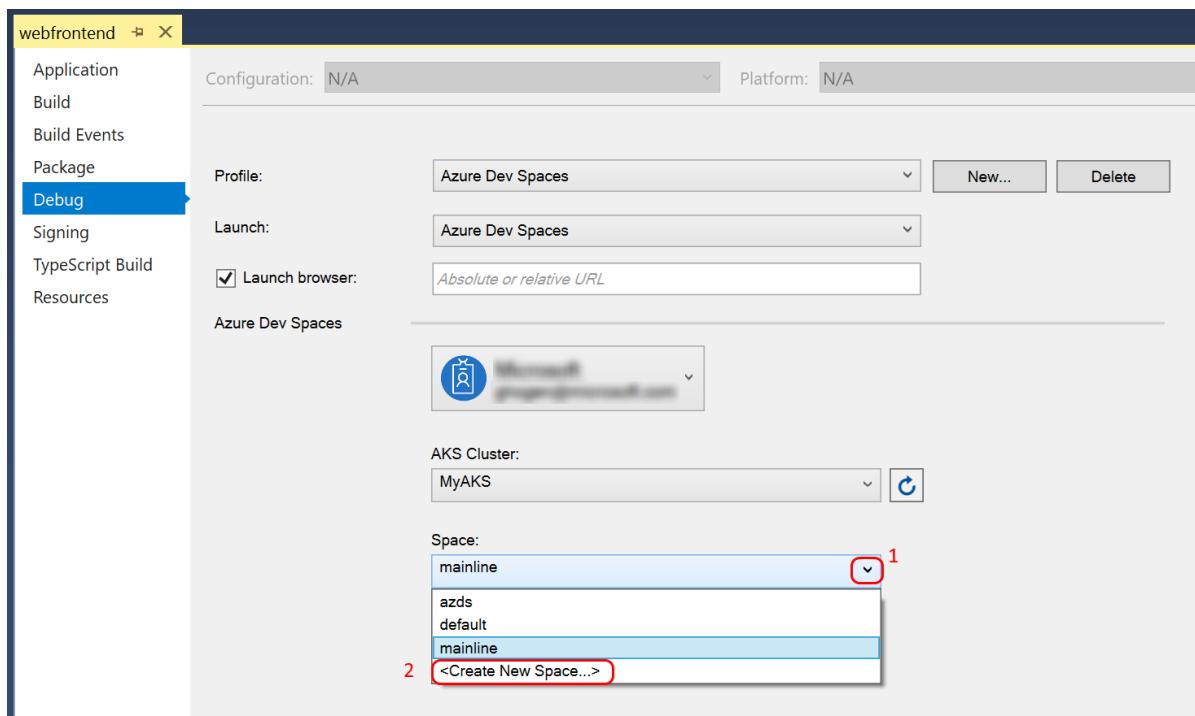
Anyone who opens the public URL and navigates to the web app will invoke the code path you have written which runs through both services using the default *dev* space. Now suppose you want to continue developing *mywebapi* - how can you do this and not interrupt other developers who are using the dev space? To do that, you'll set up your own space.

### Create a new dev space

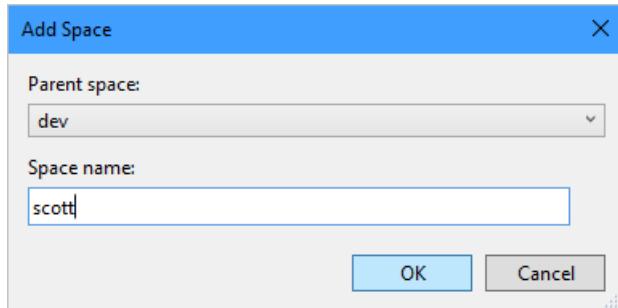
From within Visual Studio, you can create additional spaces that will be used when you F5 or Ctrl+F5 your service. You can call a space anything you'd like, and you can be flexible about what it means (ex. *sprint4* or *demo*).

Do the following to create a new space:

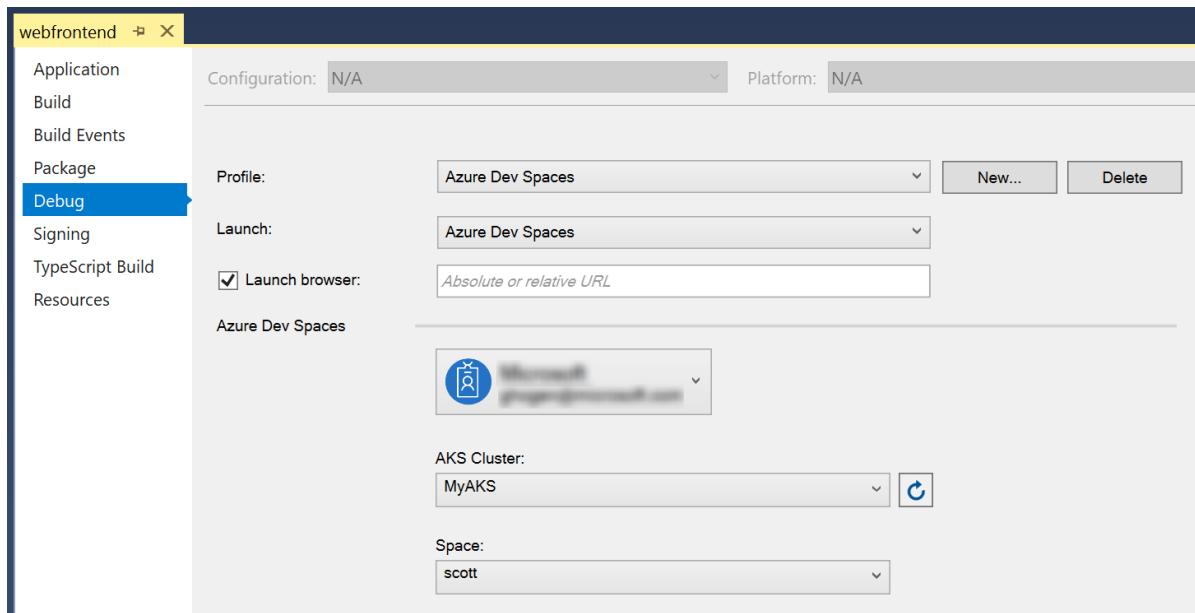
1. Switch to the Visual Studio window with the *mywebapi* project.
2. Right-click on the project in **Solution Explorer** and select **Properties**.
3. Select the **Debug** tab on the left to show the Azure Dev Spaces settings.
4. From here, you can change or create the cluster and/or space that will be used when you F5 or Ctrl+F5.  
*Make sure the Azure Dev Space you created earlier is selected.*
5. In the Space dropdown, select **<Create New Space...>**.



- In the **Add Space** dialog, set the parent space to **dev**, and enter a name for your new space. You can use your name (for example, "scott") for the new space so that it is identifiable to your peers what space you're working in. Click **OK**.



- You should now see your AKS cluster and new Space selected on the project properties page.



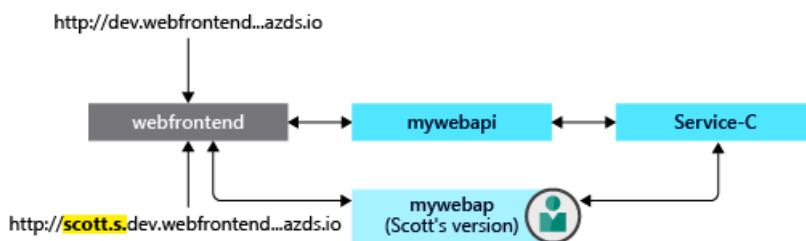
### Update code for *mywebapi*

- In the *mywebapi* project make a code change to the `string Get(int id)` method in file `Controllers/ValuesController.cs` as follows:

```
[HttpGet("{id}")]
public string Get(int id)
{
    return "mywebapi now says something new";
}
```

2. Set a breakpoint in this updated block of code (you may already have one set from before).
3. Hit F5 to start the *mywebapi* service, which will start the service in your cluster using the selected space. The selected space in this case is *scott*.

Here is a diagram that will help you understand how the different spaces work. The purple path shows a request via the *dev* space, which is the default path used if no space is prepended to the URL. The pink path shows a request via the *dev/scott* space.



This built-in capability of Azure Dev Spaces enables you to test code end-to-end in a shared environment without requiring each developer to re-create the full stack of services in their space. This routing requires propagation headers to be forwarded in your app code, as illustrated in the previous step of this guide.

### Test code running in the *dev/scott* space

To test your new version of *mywebapi* in conjunction with *webfrontend*, open your browser to the public access point URL for *webfrontend* (for example, <http://dev.webfrontend.123456abcdef.eus.azds.io>) and go to the About page. You should see the original message "Hello from webfrontend and Hello from mywebapi".

Now, add the "scott.s." part to the URL so it reads something like

<http://scott.s.dev.webfrontend.123456abcdef.eus.azds.io> and refresh the browser. The breakpoint you set in your *mywebapi* project should get hit. Click F5 to proceed and in your browser you should now see the new message "Hello from webfrontend and mywebapi now says something new." This is because the path to your updated code in *mywebapi* is running in the *dev/scott* space.

Once you have a *dev* space that always contains your latest changes, and assuming your application is designed to take advantage of DevSpace's space-based routing as described in this tutorial section, hopefully it becomes easy to see how Dev Spaces can greatly assist in testing new features within the context of the larger application. Rather than having to deploy *all* services to your private space, you can create a private space that derives from *dev*, and only "up" the services you're actually working on. The Dev Spaces routing infrastructure will handle the rest by utilizing as many services out of your private space as it can find, while defaulting back to the latest version running in the *dev* space. And better still, *multiple* developers can actively develop different services at the same time in their own space without disrupting each other.

### Well done!

You've completed the getting started guide! You learned how to:

- Set up Azure Dev Spaces with a managed Kubernetes cluster in Azure.
- Iteratively develop code in containers.
- Independently develop two separate services, and used Kubernetes' DNS service discovery to make a call to another service.
- Productively develop and test your code in a team environment.
- Establish a baseline of functionality using Dev Spaces to easily test isolated changes within the context of a

larger microservice application

Now that you've explored Azure Dev Spaces, [share your dev space with a team member](#) and help them see how easy it is to collaborate together.

## Clean up

To completely delete an Azure Dev Spaces instance on a cluster, including all the dev spaces and running services within it, use the `az aks remove-dev-spaces` command. Bear in mind that this action is irreversible. You can add support for Azure Dev Spaces again on the cluster, but it will be as if you are starting again. Your old services and spaces won't be restored.

The following example lists the Azure Dev Spaces controllers in your active subscription, and then deletes the Azure Dev Spaces controller that is associated with AKS cluster 'myaks' in resource group 'myaks-rg'.

```
azds controller list  
az aks remove-dev-spaces --name myaks --resource-group myaks-rg
```

# Create a Kubernetes dev space: Visual Studio Code and Node.js with Azure Dev Spaces

2/25/2020 • 10 minutes to read • [Edit Online](#)

In this guide, you will learn how to:

- Create a Kubernetes-based environment in Azure that is optimized for development - a *dev space*.
- Iteratively develop code in containers using VS Code and the command line.
- Productively develop and test your code in a team environment.

## NOTE

If you get stuck at any time, see the [Troubleshooting](#) section.

## Install the Azure CLI

Azure Dev Spaces requires minimal local machine setup. Most of your dev space's configuration gets stored in the cloud, and is shareable with other users. Start by downloading and running the [Azure CLI](#).

### Sign in to Azure CLI

Sign in to Azure. Type the following command in a terminal window:

```
az login
```

## NOTE

If you don't have an Azure subscription, you can create a [free account](#).

### If you have multiple Azure subscriptions...

You can view your subscriptions by running:

```
az account list --output table
```

Locate the subscription which has *True* for *IsDefault*. If this isn't the subscription you want to use, you can change the default subscription:

```
az account set --subscription <subscription ID>
```

## Create a Kubernetes cluster enabled for Azure Dev Spaces

At the command prompt, create the resource group in a [region that supports Azure Dev Spaces](#).

```
az group create --name MyResourceGroup --location <region>
```

Create a Kubernetes cluster with the following command:

```
az aks create -g MyResourceGroup -n MyAKS --location <region> --generate-ssh-keys
```

It takes a few minutes to create the cluster.

### Configure your AKS cluster to use Azure Dev Spaces

Enter the following Azure CLI command, using the resource group that contains your AKS cluster, and your AKS cluster name. The command configures your cluster with support for Azure Dev Spaces.

```
az aks use-dev-spaces -g MyResourceGroup -n MyAKS
```

#### IMPORTANT

The Azure Dev Spaces configuration process will remove the `azds` namespace in the cluster, if it exists.

## Get Kubernetes debugging for VS Code

Rich features like Kubernetes debugging are available for .NET Core and Node.js developers using VS Code.

1. If you don't have it, install [VS Code](#).
2. Download and install the [VS Azure Dev Spaces extension](#). Click Install once on the extension's Marketplace page, and again in VS Code.

## Create a Node.js container in Kubernetes

In this section, you'll create a Node.js web app and get it running in a container in Kubernetes.

### Create a Node.js Web App

Download code from GitHub by navigating to <https://github.com/Azure/dev-spaces> and select **Clone or Download** to download the GitHub repository to your local environment. The code for this guide is in `samples/nodejs/getting-started/webfrontend`.

## Prepare code for Docker and Kubernetes development

So far, you have a basic web app that can run locally. You'll now containerize it by creating assets that define the app's container and how it will deploy to Kubernetes. This task is easy to do with Azure Dev Spaces:

1. Launch VS Code and open the `webfrontend` folder. (You can ignore any default prompts to add debug assets or restore the project.)
2. Open the Integrated Terminal in VS Code (using the **View > Integrated Terminal** menu).
3. Run this command (be sure that **webfrontend** is your current folder):

```
azds prep --enable-ingress
```

The Azure CLI's `azds prep` command generates Docker and Kubernetes assets with default settings:

- `./Dockerfile` describes the app's container image, and how the source code is built and runs within the container.
- A [Helm chart](#) under `./charts/webfrontend` describes how to deploy the container to Kubernetes.

### TIP

The [Dockerfile](#) and [Helm chart](#) for your project is used by Azure Dev Spaces to build and run your code, but you can modify these files if you want to change how the project is built and ran.

For now, it isn't necessary to understand the full content of these files. It's worth pointing out, however, that **the same Kubernetes and Docker configuration-as-code assets can be used from development through to production, thus providing better consistency across different environments.**

A file named `./azds.yaml` is also generated by the `prep` command, and it is the configuration file for Azure Dev Spaces. It complements the Docker and Kubernetes artifacts with additional configuration that enables an iterative development experience in Azure.

## Build and run code in Kubernetes

Let's run our code! In the terminal window, run this command from the **root code folder**, `webfrontend`:

```
azds up
```

Keep an eye on the command's output, you'll notice several things as it progresses:

- Source code is synced to the dev space in Azure.
- A container image is built in Azure, as specified by the Docker assets in your code folder.
- Kubernetes objects are created that utilize the container image as specified by the Helm chart in your code folder.
- Information about the container's endpoint(s) is displayed. In our case, we're expecting a public HTTP URL.
- Assuming the above stages complete successfully, you should begin to see `stdout` (and `stderr`) output as the container starts up.

### NOTE

These steps will take longer the first time the `up` command is run, but subsequent runs should be quicker.

## Test the web app

Scan the console output for information about the public URL that was created by the `up` command. It will be in the form:

```
(pending registration) Service 'webfrontend' port 'http' will be available at <url>
Service 'webfrontend' port 'http' is available at http://webfrontend.1234567890abcdef1234.eus.azds.io/
Service 'webfrontend' port 80 (TCP) is available at 'http://localhost:<port>'
```

Identify the public URL for the service in the output from the `up` command. It ends in `.azds.io`. In the above example, the public URL is `http://webfrontend.1234567890abcdef1234.eus.azds.io/`.

To see your web app, open the public URL in a browser. Also, notice `stdout` and `stderr` output is streamed to the `azds trace` terminal window as you interact with your web app. You'll also see tracking information for HTTP requests as they go through the system. This makes it easier for you to track complex multi-service calls during development. The instrumentation added by Dev Spaces provides this request tracking.

## NOTE

In addition to the public URL, you can use the alternative `http://localhost:<portnumber>` URL that is displayed in the console output. If you use the localhost URL, it may seem like the container is running locally, but actually it is running in Azure. Azure Dev Spaces uses Kubernetes *port-forward* functionality to map the localhost port to the container running in AKS. This facilitates interacting with the service from your local machine.

## Update a content file

Azure Dev Spaces isn't just about getting code running in Kubernetes - it's about enabling you to quickly and iteratively see your code changes take effect in a Kubernetes environment in the cloud.

1. Locate the file `./public/index.html` and make an edit to the HTML. For example, change the page's background color to a shade of blue [on line 15](#):

```
<body style="background-color: #95B9C7; margin-left:10px; margin-right:10px;">
```

2. Save the file. Moments later, in the Terminal window you'll see a message saying a file in the running container was updated.
3. Go to your browser and refresh the page. You should see your color update.

What happened? Edits to content files, like HTML and CSS, don't require the Node.js process to restart, so an active `azds up` command will automatically sync any modified content files directly into the running container in Azure, thereby providing a fast way to see your content edits.

## Test from a mobile device

Open the web app on a mobile device using the public URL for webfrontend. You may want to copy and send the URL from your desktop to your device to save you from entering the long address. When the web app loads in your mobile device, you will notice that the UI does not display properly on a small device.

To fix this issue, you'll add a `viewport` meta tag:

1. Open the file `./public/index.html`
2. Add a `viewport` meta tag in the existing `head` element that starts [on line 6](#):

```
<head>
  <!-- Add this line -->
  <meta name="viewport" content="width=device-width, initial-scale=1">
</head>
```

3. Save the file.
4. Refresh your device's browser. You should now see the web app rendered correctly.

This example shows that some problems just aren't found until you test on the devices where an app is meant to be used. With Azure Dev Spaces, you can rapidly iterate on your code and validate any changes on target devices.

## Update a code file

Updating server-side code files requires a little more work, because a Node.js app needs to restart.

1. In the terminal window, press `Ctrl+C` (to stop `azds up`).
2. Open the code file named `server.js`, and edit service's hello message:

```
res.send('Hello from webfrontend running in Azure!');
```

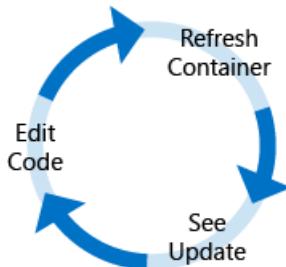
3. Save the file.
4. Run `azds up` in the terminal window.

This command rebuilds the container image and redeploys the Helm chart. Reload the browser page to see your code changes take effect.

But there is an even *faster method* for developing code, which you'll explore in the next section.

## Debug a container in Kubernetes

In this section, you'll use VS Code to directly debug our container running in Azure. You'll also learn how to get a faster edit-run-test loop.



### NOTE

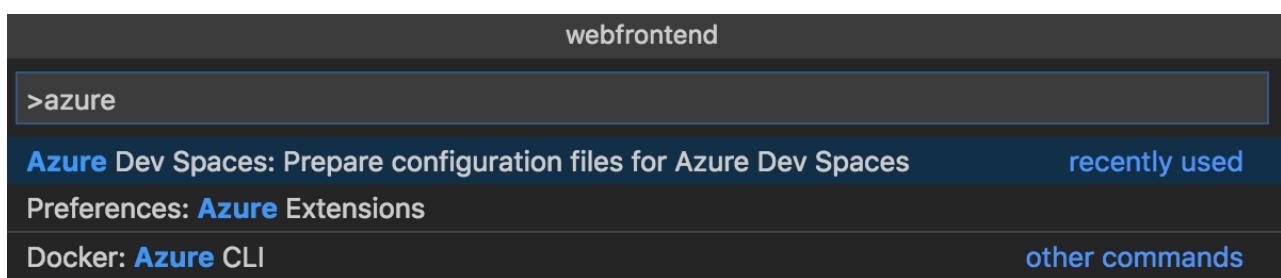
If you get stuck at any time, see the [Troubleshooting](#) section, or post a comment on this page.

### Initialize debug assets with the VS Code extension

You first need to configure your code project so VS Code will communicate with our dev space in Azure. The VS Code extension for Azure Dev Spaces provides a helper command to set up debug configuration.

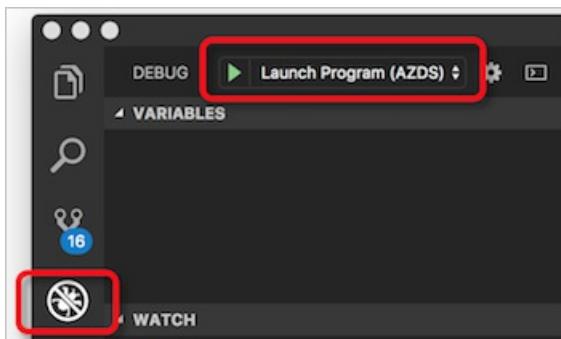
Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command: `Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces`.

This adds debug configuration for Azure Dev Spaces under the `.vscode` folder. This command is not to be confused with the `azds prep` command, which configures the project for deployment.



### Select the AZDS debug configuration

1. To open the Debug view, click on the Debug icon in the **Activity Bar** on the side of VS Code.
2. Select **Launch Program (AZDS)** as the active debug configuration.



#### NOTE

If you don't see any Azure Dev Spaces commands in the Command Palette, ensure you have [installed the VS Code extension for Azure Dev Spaces](#).

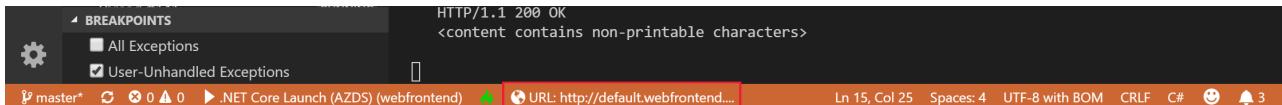
### Debug the container in Kubernetes

Hit **F5** to debug your code in Kubernetes!

Similar to the `up` command, code is synced to the development environment when you start debugging, and a container is built and deployed to Kubernetes. This time, the debugger is attached to the remote container.

#### TIP

The VS Code status bar will turn orange, indicating that the debugger is attached. It will also display a clickable URL, which you can use to quickly open your site.



Set a breakpoint in a server-side code file, for example within the `app.get('/api'...` on [line 13 of `server.js`](#).

```
```javascript
app.get('/api', function (req, res) {
  res.send('Hello from webfrontend');
});
```

```

Refresh the browser page, or press the *Say It Again* button, and you should hit the breakpoint and be able to step through code.

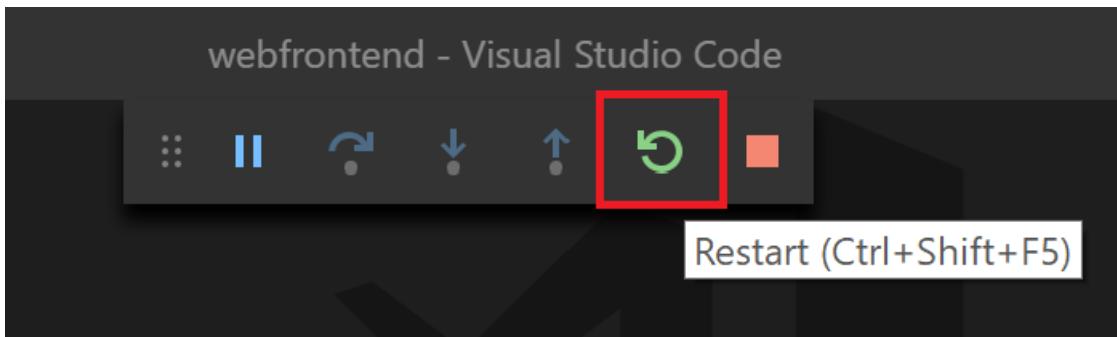
You have full access to debug information just like you would if the code was executing locally, such as the call stack, local variables, exception information, etc.

### Edit code and refresh the debug session

With the debugger active, make a code edit; for example, modify the hello message on [line 13 of `server.js`](#) again:

```
app.get('/api', function (req, res) {
  res.send('**** Hello from webfrontend running in Azure! ****');
});
```

Save the file, and in the **Debug actions pane**, click the **Restart** button.



Instead of rebuilding and redeploying a new container image each time code edits are made, which will often take considerable time, Azure Dev Spaces will restart the Node.js process in between debug sessions to provide a faster edit/debug loop.

Refresh the web app in the browser, or press the *Say It Again* button. You should see your custom message appear in the UI.

### Use NodeMon to develop even faster

*Nodemon* is a popular tool that Node.js developers use for rapid development. Instead of manually restarting the Node process each time a server-side code edit is made, developers will often configure their Node project to have *nodemon* monitor file changes and automatically restart the server process. In this style of working, the developer just refreshes their browser after making a code edit.

With Azure Dev Spaces, you can use many of the same development workflows you use when developing locally. To illustrate this point, the sample `webfrontend` project was configured to use *nodemon* (it is configured as a dev dependency in `package.json`).

Try the following steps:

1. Stop the VS Code debugger.
2. Click on the Debug icon in the **Activity Bar** on the side of VS Code.
3. Select **Attach (AZDS)** as the active debug configuration.
4. Hit F5.

In this configuration, the container is configured to start *nodemon*. When server code edits are made, *nodemon* automatically restarts the Node process, just like it does when you develop locally.

1. Edit the hello message again in `server.js`, and save the file.
2. Refresh the browser, or click the *Say It Again* button, to see your changes take effect!

**Now you have a method for rapidly iterating on code and debugging directly in Kubernetes!** Next, you'll see how you can create and call a second container.

## Next steps

[Learn about multi-service development](#)

# Running multiple dependent services: Node.js and Visual Studio Code with Azure Dev Spaces

12/11/2019 • 2 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how to develop multi-service applications using Azure Dev Spaces, along with some of the added benefits that Dev Spaces provides.

## Call a service running in a separate container

In this section you're going to create a second service, `mywebapi`, and have `webfrontend` call it. Each service will run in separate containers. You'll then debug across both containers.



### Open sample code for `mywebapi`

You should already have the sample code for `mywebapi` for this guide under a folder named `samples` (if not, go to <https://github.com/Azure/dev-spaces> and select **Clone or Download** to download the GitHub repository.) The code for this section is in `samples/nodejs/getting-started/mywebapi`.

### Run `mywebapi`

1. Open the folder `mywebapi` in a *separate VS Code window*.
2. Open the **Command Palette** (using the **View | Command Palette** menu), and use auto-complete to type and select this command: `Azure Dev Spaces: Prepare configuration files for Azure Dev Spaces`. This command is not to be confused with the `azds prep` command, which configures the project for deployment.
3. Hit F5, and wait for the service to build and deploy. You'll know it's ready when the *Listening on port 80* message appears in the debug console.
4. Take note of the endpoint URL, it will look something like `http://localhost:<portnumber>`. **Tip: The VS Code status bar will turn orange and display a clickable URL.** It may seem like the container is running locally, but actually it is running in your development environment in Azure. The reason for the localhost address is because `mywebapi` has not defined any public endpoints and can only be accessed from within the Kubernetes instance. For your convenience, and to facilitate interacting with the private service from your local machine, Azure Dev Spaces creates a temporary SSH tunnel to the container running in Azure.
5. When `mywebapi` is ready, open your browser to the localhost address. You should see a response from the `mywebapi` service ("Hello from mywebapi").

### Make a request from `webfrontend` to `mywebapi`

Let's now write code in `webfrontend` that makes a request to `mywebapi`.

1. Switch to the VS Code window for `webfrontend`.
2. Add these lines of code at the top of `server.js`:

```
var request = require('request');
```

3. Replace the code for the `/api` GET handler. When handling a request, it in turn makes a call to `mywebapi`, and then returns the results from both services.

```
app.get('/api', function (req, res) {
  request({
    uri: 'http://mywebapi',
    headers: {
      /* propagate the dev space routing header */
      'azds-route-as': req.headers['azds-route-as']
    }
  }, function (error, response, body) {
    res.send('Hello from webfrontend and ' + body);
  });
});
```

- a. Remove the `server.close()` line at the end of `server.js`

The preceding code example forwards the `azds-route-as` header from the incoming request to the outgoing request. You'll see later how this helps teams with collaborative development.

### Debug across multiple services

1. At this point, `mywebapi` should still be running with the debugger attached. If it is not, hit F5 in the `mywebapi` project.
2. Set a breakpoint inside the default GET `/` handler [on line 8 of `server.js`](#).
3. In the `webfrontend` project, set a breakpoint just before it sends a GET request to `http://mywebapi`.
4. Hit F5 in the `webfrontend` project.
5. Open the web app, and step through code in both services. The web app should display a message concatenated by the two services: "Hello from webfrontend and Hello from mywebapi."

### Well done!

You now have a multi-container application where each container can be developed and deployed separately.

## Next steps

[Learn about team development in Dev Spaces](#)

# Team development using Node.js and Visual Studio Code with Azure Dev Spaces

12/11/2019 • 9 minutes to read • [Edit Online](#)

In this tutorial, you'll learn how a team of developers can simultaneously collaborate in the same Kubernetes cluster using Dev Spaces.

## Learn about team development

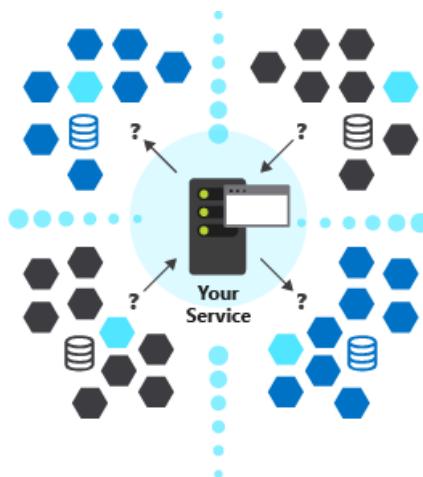
So far you've been running your application's code as if you were the only developer working on the app. In this section, you'll learn how Azure Dev Spaces streamlines team development:

- Enable a team of developers to work in the same environment, by working in a shared dev space or in distinct dev spaces as needed.
- Supports each developer iterating on their code in isolation and without fear of breaking others.
- Test code end-to-end, prior to code commit, without having to create mocks or simulate dependencies.

### Challenges with developing microservices

Your sample application isn't very complex at the moment. But in real-world development, challenges soon emerge as you add more services and the development team grows. It can become unrealistic to run everything locally for development.

- Your development machine may not have enough resources to run every service you need at once.
- Some services may need to be publicly reachable. For example, a service may need to have an endpoint that responds to a webhook.
- If you want to run a subset of services, you have to know the full dependency hierarchy between all your services. Determining this can be difficult, especially as your number of services increase.
- Some developers resort to simulating, or mocking up, many of their service dependencies. This approach can help, but managing those mocks can soon impact development cost. Plus, this approach leads to your development environment looking very different from production, which allows subtle bugs to creep in.
- It follows that doing any type of integration testing becomes difficult. Integration testing can only realistically happen post-commit, which means you see problems later in the development cycle.



### Work in a shared dev space

With Azure Dev Spaces, you can set up a *shared* dev space in Azure. Each developer can focus on just their part of the application, and can iteratively develop *pre-commit code* in a dev space that already contains all the other services and cloud resources that their scenarios depend on. Dependencies are always up-to-date, and developers are working in a way that mirrors production.

### Work in your own space

As you develop code for your service, and before you're ready to check it in, code often won't be in a good state. You're still iteratively shaping it, testing it, and experimenting with solutions. Azure Dev Spaces provides the concept of a **space**, which allows you to work in isolation, and without the fear of breaking your team members.

## Use Dev Spaces for team development

Let's demonstrate these ideas with a concrete example using our *webfrontend -> mywebapi* sample application. We'll imagine a scenario where a developer, Scott, needs to make a change to the *mywebapi* service, and *only* that service. The *webfrontend* won't need to change as part of Scott's update.

*Without* using Dev Spaces, Scott would have a few ways to develop and test his update, none of which are ideal:

- Run ALL components locally. This requires a more powerful development machine with Docker installed, and potentially MiniKube.
- Run ALL components in an isolated namespace on the Kubernetes cluster. Since *webfrontend* isn't changing, this is a waste of cluster resources.
- ONLY run *mywebapi*, and make manual REST calls to test. This doesn't test the full end-to-end flow.
- Add development-focused code to *webfrontend* that allows the developer to send requests to a different instance of *mywebapi*. This complicates the *webfrontend* service.

### Set up your baseline

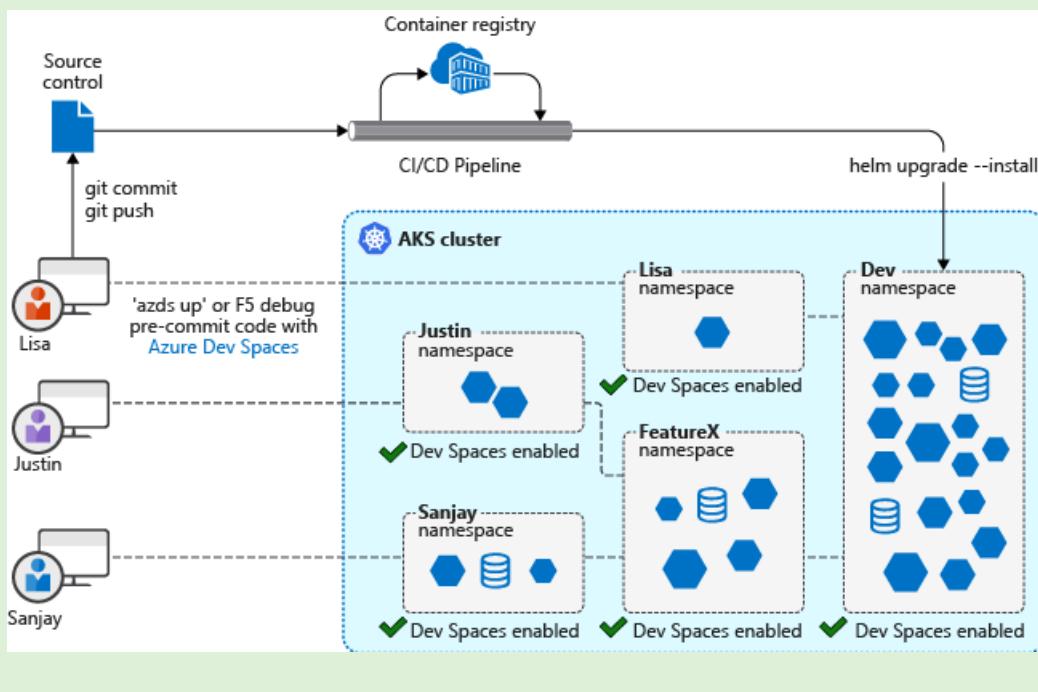
First we'll need to deploy a baseline of our services. This deployment will represent the "last known good" so you can easily compare the behavior of your local code vs. the checked-in version. We'll then create a child space based on this baseline so we can test our changes to *mywebapi* within the context of the larger application.

1. Clone the [Dev Spaces sample application](#): `git clone https://github.com/Azure/dev-spaces && cd dev-spaces`
2. Checkout the remote branch *azds\_updates*: `git checkout -b azds_updates origin/azds_updates`
3. Select the *dev* space: `azds space select --name dev`. When prompted to select a parent dev space, select *<none>*.
4. Navigate to the *mywebapi* directory and execute: `azds up -d`
5. Navigate to the *webfrontend* directory and execute: `azds up -d`
6. Execute `azds list-uris` to see the public endpoint for *webfrontend*

## TIP

The above steps manually set up a baseline, but we recommend teams use CI/CD to automatically keep your baseline up to date with committed code.

Check out our [guide to setting up CI/CD with Azure DevOps](#) to create a workflow similar to the following diagram.



At this point your baseline should be running. Run the `azds list-up --all` command, and you'll see output similar to the following:

| Name                         | DevSpace | Type    | Updated | Status  |
|------------------------------|----------|---------|---------|---------|
| mywebapi                     | dev      | Service | 3m ago  | Running |
| mywebapi-56c8f45d9-zs4mw     | dev      | Pod     | 3m ago  | Running |
| webfrontend                  | dev      | Service | 1m ago  | Running |
| webfrontend-6b6ddbb98f-fgvnc | dev      | Pod     | 1m ago  | Running |

The DevSpace column shows that both services are running in a space named *dev*. Anyone who opens the public URL and navigates to the web app will invoke the checked-in code path that runs through both services. Now suppose you want to continue developing *mywebapi*. How can you make code changes and test them and not interrupt other developers who are using the dev environment? To do that, you'll set up your own space.

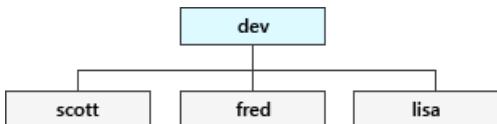
## Create a dev space

To run your own version of *mywebapi* in a space other than *dev*, you can create your own space by using the following command:

```
azds space select --name scott
```

When prompted, select *dev* as the **parent dev space**. This means our new space, *dev/scott*, will derive from the space *dev*. We'll shortly see how this will help us with testing.

Keeping with our introductory hypothetical, we've used the name *scott* for the new space so peers can identify who is working in it. But it can be called anything you like, and be flexible about what it means, like *sprint4* or *demo*. Whatever the case, *dev* serves as the baseline for all developers working on a piece of this application:



Run the `azds space list` command to see a list of all the spaces in the *dev* environment. The *Selected* column indicates which space you currently have selected (true/false). In your case, the space named *dev/scott* was automatically selected when it was created. You can select another space at any time with the `azds space select` command.

Let's see it in action.

### Make a code change

Go to the VS Code window for `mywebapi` and make a code edit to the default GET `/` handler in `server.js`, for example:

```

app.get('/', function (req, res) {
    res.send('mywebapi now says something new');
});
```

### Run the service

To run the service, hit F5 (or type `azds up` in the Terminal Window) to run the service. The service will automatically run in your newly selected space *dev/scott*. Confirm that your service is running in its own space by running `azds list-up`:

```

$ azds list-up

Name          DevSpace  Type     Updated   Status
mywebapi      scott    Service  3m ago   Running
webfrontend   dev      Service  26m ago  Running
```

Notice an instance of *mywebapi* is now running in the *dev/scott* space. The version running in *dev* is still running but it is not listed.

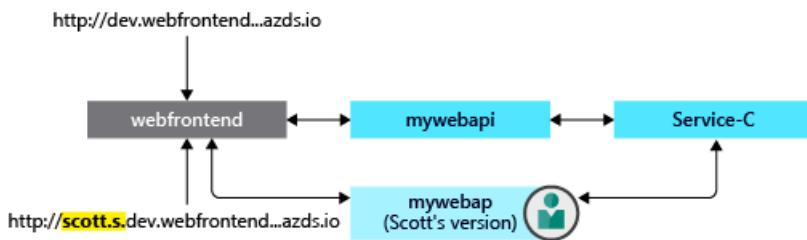
List the URLs for the current space by running `azds list-uris`.

```

$ azds list-uris

Uri                                Status
-----
http://localhost:53831 => mywebapi.scott:80           Tunneled
http://scott.s.dev.webfrontend.6364744826e042319629.ce.azds.io/ Available
```

Notice the public access point URL for *webfrontend* is prefixed with *scott.s*. This URL is unique to the *dev/scott* space. This URL prefix tells the Ingress controller to route requests to the *dev/scott* version of a service. When a request with this URL is handled by Dev Spaces, the Ingress Controller first tries to route the request to the *webfrontend* service in the *dev/scott* space. If that fails, the request will be routed to the *webfrontend* service in the *dev* space as a fallback. Also notice there is a localhost URL to access the service over localhost using the Kubernetes *port-forward* functionality. For more information about URLs and routing in Azure Dev Spaces, see [How Azure Dev Spaces works and is configured](#).



This built-in feature of Azure Dev Spaces lets you test code in a shared space without requiring each developer to re-create the full stack of services in their space. This routing requires your app code to forward propagation headers, as illustrated in the previous step of this guide.

### Test code in a space

To test your new version of *mywebapi* with *webfrontend*, open your browser to the public access point URL for *webfrontend* and go to the About page. You should see your new message displayed.

Now, remove the "scott.s." part of the URL, and refresh the browser. You should see the old behavior (with the *mywebapi* version running in *dev*).

Once you have a *dev* space, which always contains your latest changes, and assuming your application is designed to take advantage of DevSpace's space-based routing as described in this tutorial section, hopefully it becomes easy to see how Dev Spaces can greatly assist in testing new features within the context of the larger application. Rather than having to deploy *all* services to your private space, you can create a private space that derives from *dev*, and only "up" the services you're actually working on. The Dev Spaces routing infrastructure will handle the rest by utilizing as many services out of your private space as it can find, while defaulting back to the latest version running in the *dev* space. And better still, *multiple* developers can actively develop different services at the same time in their own space without disrupting each other.

### Well done!

You've completed the getting started guide! You learned how to:

- Set up Azure Dev Spaces with a managed Kubernetes cluster in Azure.
- Iteratively develop code in containers.
- Independently develop two separate services, and used Kubernetes' DNS service discovery to make a call to another service.
- Productively develop and test your code in a team environment.
- Establish a baseline of functionality using Dev Spaces to easily test isolated changes within the context of a larger microservice application

Now that you've explored Azure Dev Spaces, [share your dev space with a team member](#) and begin collaborating.

## Clean up

To completely delete an Azure Dev Spaces instance on a cluster, including all the dev spaces and running services within it, use the `az aks remove-dev-spaces` command. Bear in mind that this action is irreversible. You can add support for Azure Dev Spaces again on the cluster, but it will be as if you are starting again. Your old services and spaces won't be restored.

The following example lists the Azure Dev Spaces controllers in your active subscription, and then deletes the Azure Dev Spaces controller that is associated with AKS cluster 'myaks' in resource group 'myaks-rg'.

```

azds controller list
az aks remove-dev-spaces --name myaks --resource-group myaks-rg

```

# Install applications with Helm in Azure Kubernetes Service (AKS)

2/25/2020 • 10 minutes to read • [Edit Online](#)

Helm is an open-source packaging tool that helps you install and manage the lifecycle of Kubernetes applications. Similar to Linux package managers such as *APT* and *Yum*, Helm is used to manage Kubernetes charts, which are packages of preconfigured Kubernetes resources.

This article shows you how to configure and use Helm in a Kubernetes cluster on AKS.

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart using the [Azure CLI](#) or [using the Azure portal](#).

You also need the Helm CLI installed, which is the client that runs on your development system. It allows you to start, stop, and manage applications with Helm. If you use the Azure Cloud Shell, the Helm CLI is already installed. For installation instructions on your local platform, see [Installing Helm](#).

### IMPORTANT

Helm is intended to run on Linux nodes. If you have Windows Server nodes in your cluster, you must ensure that Helm pods are only scheduled to run on Linux nodes. You also need to ensure that any Helm charts you install are also scheduled to run on the correct nodes. The commands in this article use [node-selectors](#) to make sure pods are scheduled to the correct nodes, but not all Helm charts may expose a node selector. You can also consider using other options on your cluster, such as [taints](#).

## Verify your version of Helm

Use the `helm version` command to verify the version of Helm you have installed:

```
helm version
```

The following example shows Helm version 3.0.0 installed:

```
$ helm version  
  
version.BuildInfo{Version:"v3.0.0", GitCommit:"e29ce2a54e96cd02ccfce88bee4f58bb6e2a28b6",  
GitTreeState:"clean", GoVersion:"go1.13.4"}
```

For Helm v3, follow the steps in the [Helm v3 section](#). For Helm v2, follow the steps in the [Helm v2 section](#)

## Install an application with Helm v3

### Add the official Helm stable charts repository

Use the `helm repo` command to add the official Helm stable charts repository.

```
helm repo add stable https://kubernetes-charts.storage.googleapis.com/
```

## Find Helm charts

Helm charts are used to deploy applications into a Kubernetes cluster. To search for pre-created Helm charts, use the [helm search](#) command:

```
helm search repo stable
```

The following condensed example output shows some of the Helm charts available for use:

| NAME   | CHART VERSION | APP VERSION | DESCRIPTION                  |
|--|---------------|-------------|------------------------------|
| stable/acs-engine-autoscaler<br>nodes within agent pools                           | 2.2.2         | 2.1.1       | DEPRECATED Scales worker     |
| stable/aerospike<br>in Kubernetes  | 0.3.1         | v4.5.0.5    | A Helm chart for Aerospike   |
| stable/airflow<br>programmatically autho...  | 4.10.0        | 1.10.4      | Airflow is a platform to     |
| stable/ambassador<br>Ambassador  | 4.4.7         | 0.85.0      | A Helm chart for Datawire    |
| stable/anchore-engine<br>and policy evaluatio...                                   | 1.3.7         | 0.5.2       | Anchore container analysis   |
| stable/apm-server<br>from the Elastic APM a...                                     | 2.1.5         | 7.0.0       | The server receives data     |
| stable/ark<br>ark  | 4.2.2         | 0.10.2      | DEPRECATED A Helm chart for  |
| stable/artifactory<br>Repository Manager support...                                | 7.3.1         | 6.1.0       | DEPRECATED Universal         |
| stable/artifactory-ha<br>Repository Manager support...                             | 0.4.1         | 6.2.0       | DEPRECATED Universal         |
| stable/atlantis<br><a href="https://www.runatlant...">https://www.runatlant...</a> | 3.8.4         | v0.8.2      | A Helm chart for Atlantis    |
| stable/auditbeat<br>audit the activities o...                                      | 1.1.0         | 6.7.0       | A lightweight shipper to     |
| stable/aws-cluster-autoscaler<br>autoscaling groups.                               | 0.3.3         |             | Scales worker nodes within   |
| stable/aws-iam-authenticator<br>authenticator                                      | 0.1.1         | 1.0         | A Helm chart for aws-iام-    |
| stable/bitcoind<br>payment network and a ...                                       | 0.2.2         | 0.17.1      | Bitcoin is an innovative     |
| stable/bookstack<br>hosted, easy-to-use...   | 1.1.2         | 0.27.4-1    | BookStack is a simple, self- |
| stable/buildkite<br>Buildkite  | 0.2.4         | 3           | DEPRECATED Agent for         |
| stable/burrow<br>smart contract machine  | 1.5.2         | 0.29.0      | Burrow is a permissionable   |
| stable/centrifugo<br>messaging server.   | 3.1.0         | 2.1.0       | Centrifugo is a real-time    |
| stable/cerebro<br>web admin tool tha...  | 1.3.1         | 0.8.5       | A Helm chart for Cerebro - a |
| stable/cert-manager<br>manager   | v0.6.7        | v0.6.2      | A Helm chart for cert-       |
| stable/chaoskube<br>random pods in you...  | 3.1.2         | 0.14.0      | Chaoskube periodically kills |
| stable/chartmuseum<br>Repository   | 2.4.0         | 0.8.2       | Host your own Helm Chart     |
| stable/chronograf<br>written in Go and R...  | 1.1.0         | 1.7.12      | Open-source web application  |
| stable/clamav<br>engine for detecting t...   | 1.0.4         | 1.6         | An Open-Source antivirus     |
| stable/cloudserver<br>implementation of the Am...                                  | 1.0.3         | 8.1.5       | An open-source Node.js       |
| stable/cluster-autoscaler<br>autoscaling groups.                                   | 6.2.0         | 1.14.6      | Scales worker nodes within   |
| stable/cluster-overprovisioner<br>that overprovisions t...                         | 0.2.6         | 1.0         | Installs the a deployment    |

|                           |        |         |  |
|---------------------------|--------|---------|--|
| stable/cockroachdb        | 2.1.16 | 19.1.5  | CockroachDB is a scalable, survivable, strongly... |
| stable/collabora-code     | 1.0.5  | 4.0.3.1 | A Helm chart for Collabora                         |
| Office - CODE-Edition     |        |         |  |
| stable/concourse          | 8.2.7  | 5.6.0   | Concourse is a simple and scalable CI system.      |
| stable/consul             | 3.9.2  | 1.5.3   | Highly available and distributed service discov... |
| stable/contour            | 0.1.0  | v0.15.0 | Contour Ingress controller for Kubernetes          |
| stable/coredns            | 1.7.4  | 1.6.4   | CoreDNS is a DNS server that chains plugins and... |
| stable/cosbench           | 1.0.1  | 0.0.6   | A benchmark tool for cloud object storage services |
| stable/coscale            | 1.0.0  | 3.16.0  | CoScale Agent                                      |
| stable/couchbase-operator | 1.0.1  | 1.2.1   | A Helm chart to deploy the Couchbase Autonomous... |
| stable/couchdb            | 2.3.0  | 2.3.1   | DEPRECATED A database featuring seamless multi-... |
| stable/dask               | 3.1.0  | 1.1.5   | Distributed computation in Python with task sch... |
| stable/dask-distributed   | 2.0.2  |         | DEPRECATED: Distributed computation in Python      |
| stable/datadog            | 1.38.3 | 6.14.0  | DataDog Agent                                      |
| ...                       |        |         |  |

To update the list of charts, use the [helm repo update](#) command. The following example shows a successful repo update:

```
$ helm repo update

Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "stable" chart repository
Update Complete. 🎉 Happy Helming! 🎉
```

## Run Helm charts

To install charts with Helm, use the [helm install](#) command and specify a release name and the name of the chart to install. To see installing a Helm chart in action, let's install a basic nginx deployment using a Helm chart.

```
helm install my-nginx-ingress stable/nginx-ingress \
--set controller.nodeSelector."beta\\.kubernetes\\.io/os"=linux \
--set defaultBackend.nodeSelector."beta\\.kubernetes\\.io/os"=linux
```

The following condensed example output shows the deployment status of the Kubernetes resources created by the Helm chart:

```
$ helm install my-nginx-ingress stable/nginx-ingress \
>     --set controller.nodeSelector."beta\\.kubernetes\\.io/os"=linux \
>     --set defaultBackend.nodeSelector."beta\\.kubernetes\\.io/os"=linux

NAME: my-nginx-ingress
LAST DEPLOYED: Fri Nov 22 10:08:06 2019
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The nginx-ingress controller has been installed.
It may take a few minutes for the LoadBalancer IP to be available.
You can watch the status by running 'kubectl --namespace default get services -o wide -w my-nginx-ingress-controller'
...

```

Use the `kubectl get services` command to get the *EXTERNAL-IP* of your service. For example, the below command shows the *EXTERNAL-IP* for the *mv-nainx-ingress-controller* service:

```
$ kubectl --namespace default get services -o wide -w my-nginx-ingress-controller
```

| NAME  | TYPE         | CLUSTER-IP | EXTERNAL-IP   | PORT(S)                    | AGE |
|---|--------------|------------|---------------|----------------------------|-----|
| my-nginx-ingress-controller                                     | LoadBalancer | 10.0.123.1 | <EXTERNAL-IP> | 80:31301/TCP,443:31623/TCP | 96s |
| app=nginx-ingress,component=controller,release=my-nginx-ingress |              |            |               |                            |     |

## List releases

To see a list of releases installed on your cluster, use the `helm list` command.

helm list

The following example shows the *my-nginx-ingress* release deployed in the previous step:

```
$ helm list
```

| NAME             | NAMESPACE | REVISION | UPDATED                              | STATUS   | CHART                | APP    |
|------------------|-----------|----------|--------------------------------------|----------|----------------------|--------|
| VERSION          |           |          |                                      |          |                      |        |
| my-nginx-ingress | default   | 1        | 2019-11-22 10:08:06.048477 -0600 CST | deployed | nginx-ingress-1.25.0 | 0.26.1 |

### **Clean up resources**

When you deploy a Helm chart, a number of Kubernetes resources are created. These resources include pods, deployments, and services. To clean up these resources, use the `helm uninstall` command and specify your release name, as found in the previous `helm list` command.

```
helm uninstall my-nginx-ingress
```

The following example shows the release named *my-nginx-ingress* has been uninstalled:

```
$ helm uninstall my-nginx-ingress  
release "my-nginx-ingress" uninstalled
```

## Install an application with Helm v2

## Create a service account

Before you can deploy Helm in an RBAC-enabled AKS cluster, you need a service account and role binding for the Tiller service. For more information on securing Helm / Tiller in an RBAC enabled cluster, see [Tiller, Namespaces, and RBAC](#). If your AKS cluster is not RBAC enabled, skip this step.

Create a file named `helm-rbac.yaml` and copy in the following YAML:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
```

Create the service account and role binding with the `kubectl apply` command:

```
kubectl apply -f helm-rbac.yaml
```

## Secure Tiller and Helm

The Helm client and Tiller service authenticate and communicate with each other using TLS/SSL. This authentication method helps to secure the Kubernetes cluster and what services can be deployed. To improve security, you can generate your own signed certificates. Each Helm user would receive their own client certificate, and Tiller would be initialized in the Kubernetes cluster with certificates applied. For more information, see [Using TLS/SSL between Helm and Tiller](#).

With an RBAC-enabled Kubernetes cluster, you can control the level of access Tiller has to the cluster. You can define the Kubernetes namespace that Tiller is deployed in, and restrict what namespaces Tiller can then deploy resources in. This approach lets you create Tiller instances in different namespaces and limit deployment boundaries, and scope the users of Helm client to certain namespaces. For more information, see [Helm role-based access controls](#).

## Configure Helm

To deploy a basic Tiller into an AKS cluster, use the `helm init` command. If your cluster is not RBAC enabled, remove the `--service-account` argument and value. The following examples also set the `history-max` to 200.

If you configured TLS/SSL for Tiller and Helm, skip this basic initialization step and instead provide the required `--tiller-tls-` as shown in the next example.

```
helm init --history-max 200 --service-account tiller --node-selectors "beta.kubernetes.io/os=linux"
```

If you configured TLS/SSL between Helm and Tiller provide the `--tiller-tls-*` parameters and names of your own certificates, as shown in the following example:

```
helm init \
--tiller-tls \
--tiller-tls-cert tiller.cert.pem \
--tiller-tls-key tiller.key.pem \
--tiller-tls-verify \
--tls-ca-cert ca.cert.pem \
--history-max 200 \
--service-account tiller \
--node-selectors "beta.kubernetes.io/os=linux"
```

## Find Helm charts

Helm charts are used to deploy applications into a Kubernetes cluster. To search for pre-created Helm charts, use the [helm search](#) command:

```
helm search
```

The following condensed example output shows some of the Helm charts available for use:

```
$ helm search

NAME          CHART VERSION APP VERSION DESCRIPTION
stable/aerospike    0.1.7      v3.14.1.2   A Helm chart for Aerospike in Kubernetes
stable/anchore-engine 0.1.7      0.1.10     Anchore container analysis and policy evaluatio...
stable/apm-server   0.1.0      6.2.4      The server receives data from the Elastic APM a...
stable/ark          1.0.1      0.8.2      A Helm chart for ark
stable/artifactory  7.2.1      6.0.0      Universal Repository Manager supporting all maj...
stable/artifactory-ha 0.2.1      6.0.0      Universal Repository Manager supporting all maj...
stable/auditbeat    0.1.0      6.2.4      A lightweight shipper to audit the activities o...
stable/aws-cluster-autoscaler 0.3.3
stable/bitcoind     0.1.3      0.15.1     Bitcoin is an innovative payment network and a ...
stable/buildkite    0.2.3      3          Agent for Buildkite
stable/burrow        0.4.4      0.17.1     Burrow is a permissionable smart contract machine
stable/centrifugo   2.0.1      1.7.3      Centrifugo is a real-time messaging server.
stable/cerebro       0.1.0      0.7.3      A Helm chart for Cerebro - a web admin tool tha...
stable/cert-manager v0.3.3     v0.3.1     A Helm chart for cert-manager
stable/chaoskube    0.7.0      0.8.0      Chaoskube periodically kills random pods in you...
stable/chartmuseum  1.5.0      0.7.0      Helm Chart Repository with support for Amazon S...
stable/chronograf   0.4.5      1.3        Open-source web application written in Go and R...
stable/cluster-autoscaler 0.6.4
stable/cockroachdb  1.1.1      2.0.0      CockroachDB is a scalable, survivable, strongly...
stable/concourse    1.10.1     3.14.1     Concourse is a simple and scalable CI system.
stable/consul        3.2.0      1.0.0      Highly available and distributed service discov...
stable/coredns      0.9.0      1.0.6      CoreDNS is a DNS server that chains plugins and...
stable/coscale       0.2.1      3.9.1      CoScale Agent
stable/dask          1.0.4      0.17.4     Distributed computation in Python with task sch...
stable/dask-distributed 2.0.2
stable/datadog      0.18.0     6.3.0      DataDog Agent
...
```

To update the list of charts, use the [helm repo update](#) command. The following example shows a successful repo update:

```
$ helm repo update

Hold tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
Update Complete.
```

## Run Helm charts

To install charts with Helm, use the `helm install` command and specify the name of the chart to install. To see installing a Helm chart in action, let's install a basic nginx deployment using a Helm chart. If you configured TLS/SSL, add the `--tls` parameter to use your Helm client certificate.

```
helm install stable/nginx-ingress \
--set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \
--set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux
```

The following condensed example output shows the deployment status of the Kubernetes resources created by the Helm chart:

```
$ helm install stable/nginx-ingress --set controller.nodeSelector."beta\.kubernetes\.io/os"=linux --set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux

NAME: flailing-alpaca
LAST DEPLOYED: Thu May 23 12:55:21 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME                      DATA   AGE
flailing-alpaca-nginx-ingress-controller  1      0s

==> v1/Pod(related)
NAME                           READY   STATUS        RESTARTS   AGE
flailing-alpaca-nginx-ingress-controller-56666df9f-bq4cl  0/1    ContainerCreating  0          0s
flailing-alpaca-nginx-ingress-default-backend-66bc89dc44-m87bp  0/1    ContainerCreating  0          0s

==> v1/Service
NAME              TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)
AGE
flailing-alpaca-nginx-ingress-controller   LoadBalancer   10.0.109.7   <pending>
80:31219/TCP,443:32421/TCP  0s
flailing-alpaca-nginx-ingress-default-backend ClusterIP   10.0.44.97   <none>       80/TCP
0s
...
...
```

It takes a minute or two for the *EXTERNAL-IP* address of the nginx-ingress-controller service to be populated and allow you to access it with a web browser.

## List Helm releases

To see a list of releases installed on your cluster, use the `helm list` command. The following example shows the nginx-ingress release deployed in the previous step. If you configured TLS/SSL, add the `--tls` parameter to use your Helm client certificate.

```
$ helm list

NAME      REVISION UPDATED      STATUS      CHART      APP VERSION NAMESPACE
flailing-alpaca  1      Thu May 23 12:55:21 2019  DEPLOYED  nginx-ingress-1.6.13  0.24.1      default
```

## Clean up resources

When you deploy a Helm chart, a number of Kubernetes resources are created. These resources include pods, deployments, and services. To clean up these resources, use the `helm delete` command and specify your release name, as found in the previous `helm list` command. The following example deletes the release named *flailing-alpaca*:

```
$ helm delete flailing-alpaca  
release "flailing-alpaca" deleted
```

## Next steps

For more information about managing Kubernetes application deployments with Helm, see the Helm documentation.

[Helm documentation](#)

# Integrate with Azure-managed services using Open Service Broker for Azure (OSBA)

2/25/2020 • 3 minutes to read • [Edit Online](#)

Together with the [Kubernetes Service Catalog](#), Open Service Broker for Azure (OSBA) allows developers to utilize Azure-managed services in Kubernetes. This guide focuses on deploying Kubernetes Service Catalog, Open Service Broker for Azure (OSBA), and applications that use Azure-managed services using Kubernetes.

## Prerequisites

- An Azure subscription
- Azure CLI: [install it locally](#), or use it in the [Azure Cloud Shell](#).
- Helm CLI 2.7+: [install it locally](#), or use it in the [Azure Cloud Shell](#).
- Permissions to create a service principal with the Contributor role on your Azure subscription
- An existing Azure Kubernetes Service (AKS) cluster. If you need an AKS cluster, follow the[Create an AKS cluster](#) quickstart.

## Install Service Catalog

The first step is to install Service Catalog in your Kubernetes cluster using a Helm chart. Upgrade your Tiller (Helm server) installation in your cluster with:

```
helm init --upgrade
```

Now, add the Service Catalog chart to the Helm repository:

```
helm repo add svc-cat https://svc-catalog-charts.storage.googleapis.com
```

Finally, install Service Catalog with the Helm chart. If your cluster is RBAC-enabled, run this command.

```
helm install svc-cat/catalog --name catalog --namespace catalog --set  
apiserver.storage.etcd.persistence.enabled=true --set apiserver.healthcheck.enabled=false --set  
controllerManager.healthcheck.enabled=false --set apiserver.verbosity=2 --set controllerManager.verbosity=2
```

If your cluster is not RBAC-enabled, run this command.

```
helm install svc-cat/catalog --name catalog --namespace catalog --set rbacEnable=false --set  
apiserver.storage.etcd.persistence.enabled=true --set apiserver.healthcheck.enabled=false --set  
controllerManager.healthcheck.enabled=false --set apiserver.verbosity=2 --set controllerManager.verbosity=2
```

After the Helm chart has been run, verify that `servicecatalog` appears in the output of the following command:

```
kubectl get apiservice
```

For example, you should see output similar to the following (show here truncated):

| NAME                          | AGE |
|-------------------------------|-----|
| v1.                           | 10m |
| v1.authentication.k8s.io      | 10m |
| ...                           |     |
| v1beta1.servicecatalog.k8s.io | 34s |
| v1beta1.storage.k8s.io        | 10  |

## Install Open Service Broker for Azure

The next step is to install [Open Service Broker for Azure](#), which includes the catalog for the Azure-managed services. Examples of available Azure services are Azure Database for PostgreSQL, Azure Database for MySQL, and Azure SQL Database.

Start by adding the Open Service Broker for Azure Helm repository:

```
helm repo add azure https://kubernetescharts.blob.core.windows.net/azure
```

Create a [Service Principal](#) with the following Azure CLI command:

```
az ad sp create-for-rbac
```

Output should be similar to the following. Take note of the `appId`, `password`, and `tenant` values, which you use in the next step.

```
{
  "appId": "7248f250-0000-0000-0000-dbdeb8400d85",
  "displayName": "azure-cli-2017-10-15-02-20-15",
  "name": "http://azure-cli-2017-10-15-02-20-15",
  "password": "77851d2c-0000-0000-0000-cb3ebc97975a",
  "tenant": "72f988bf-0000-0000-0000-2d7cd011db47"
}
```

Set the following environment variables with the preceding values:

```
AZURE_CLIENT_ID=<appId>
AZURE_CLIENT_SECRET=<password>
AZURE_TENANT_ID=<tenant>
```

Now, get your Azure subscription ID:

```
az account show --query id --output tsv
```

Again, set the following environment variable with the preceding value:

```
AZURE_SUBSCRIPTION_ID=[your Azure subscription ID from above]
```

Now that you've populated these environment variables, execute the following command to install the Open Service Broker for Azure using the Helm chart:

```
helm install azure/open-service-broker-azure --name osba --namespace osba \
--set azure.subscriptionId=${AZURE_SUBSCRIPTION_ID} \
--set azure.tenantId=${AZURE_TENANT_ID} \
--set azure.clientId=${AZURE_CLIENT_ID} \
--set azure.clientSecret=${AZURE_CLIENT_SECRET}
```

Once the OSBA deployment is complete, install the [Service Catalog CLI](#), an easy-to-use command-line interface for querying service brokers, service classes, service plans, and more.

Execute the following commands to install the Service Catalog CLI binary:

```
curl -sLO https://servicecatalogcli.blob.core.windows.net/cli/latest/$(uname -s)/$(uname -m)/svcat
chmod +x ./svcat
```

Now, list installed service brokers:

```
./svcat get brokers
```

You should see output similar to the following:

| NAME | URL  | STATUS |
|------|--|--------|
| osba | http://osba-open-service-broker-azure.osba.svc.cluster.local | Ready  |

Next, list the available service classes. The displayed service classes are the available Azure-managed services that can be provisioned through Open Service Broker for Azure.

```
./svcat get classes
```

Finally, list all available service plans. Service plans are the service tiers for the Azure-managed services. For example, for Azure Database for MySQL, plans range from `basic50` for Basic tier with 50 Database Transaction Units (DTUs), to `standard800` for Standard tier with 800 DTUs.

```
./svcat get plans
```

## Install WordPress from Helm chart using Azure Database for MySQL

In this step, you use Helm to install an updated Helm chart for WordPress. The chart provisions an external Azure Database for MySQL instance that WordPress can use. This process can take a few minutes.

```
helm install azure/wordpress --name wordpress --namespace wordpress --set resources.requests.cpu=0 --set replicaCount=1
```

In order to verify the installation has provisioned the right resources, list the installed service instances and bindings:

```
./svcat get instances -n wordpress
./svcat get bindings -n wordpress
```

List installed secrets:

```
kubectl get secrets -n wordpress -o yaml
```

## Next steps

By following this article, you deployed Service Catalog to an Azure Kubernetes Service (AKS) cluster. You used Open Service Broker for Azure to deploy a WordPress installation that uses Azure-managed services, in this case Azure Database for MySQL.

Refer to the [Azure/helm-charts](#) repository to access other updated OSBA-based Helm charts. If you're interested in creating your own charts that work with OSBA, refer to [Creating a New Chart](#).

# Integrate existing MongoDB application with Azure Cosmos DB API for MongoDB and Open Service Broker for Azure (OSBA)

1/19/2020 • 6 minutes to read • [Edit Online](#)

Azure Cosmos DB is a globally distributed, multi-model database service. It also provides wire protocol compatibility with several NoSQL APIs including for MongoDB. The Cosmos DB API for MongoDB allows you to use Cosmos DB with your existing MongoDB application without having to change your application's database drivers or implementation. You can also provision a Cosmos DB service using Open Service Broker for Azure.

In this article, you take an existing Java application that uses a MongoDB database and update it to use a Cosmos DB database using Open Service Broker for Azure.

## Prerequisites

Before you can proceed, you must:

- Have an [Azure Kubernetes Service cluster](#) created.
- Have [Open Service Broker for Azure installed and configured on your AKS cluster](#).
- Have the [Service Catalog CLI](#) installed and configured to run `svcat` commands.
- Have an existing [MongoDB](#) database. For example, you could have MongoDB running on your [development machine](#) or in an [Azure VM](#).
- Have a way of connecting to and querying the MongoDB database, such as the [mongo shell](#).

## Get application code

In this article, you use the [spring music sample application from Cloud Foundry](#) to demonstrate an application that uses a MongoDB database.

Clone the application from GitHub and navigate into its directory:

```
git clone https://github.com/cloudfoundry-samples/spring-music
cd spring-music
```

## Prepare the application to use your MongoDB database

The spring music sample application provides many options for datasources. In this article, you configure it to use an existing MongoDB database.

Add the YAML following to the end of `src/main/resources/application.yml`. This addition creates a profile called `mongodb` and configures a URI and database name. Replace the URI with the connection information to your existing MongoDB database. Adding the URI, which contains a username and password, directly to this file is for **development use only** and **should never be added to version control**.

```
---
spring:
  profiles: mongodb
  data:
    mongodb:
      uri: "mongodb://user:password@serverAddress:port/musicdb"
      database: musicdb
```

When you start your application and tell it to use the *mongodb* profile, it connects to your MongoDB database and use it to store the application's data.

To build your application:

```
./gradlew clean assemble

Starting a Gradle Daemon (subsequent builds will be faster)

BUILD SUCCESSFUL in 10s
4 actionable tasks: 4 executed
```

Start your application and tell it to use the *mongodb* profile:

```
java -jar -Dspring.profiles.active=mongodb build/libs/spring-music-1.0.jar
```

Navigate to <http://localhost:8080> in your browser.

| Album                | Artist                  | Year | Genre |
|----------------------|-------------------------|------|-------|
| Nevermind            | Nirvana                 | 1991 | Rock  |
| Pet Sounds           | The Beach Boys          | 1966 | Rock  |
| What's Going On      | Marvin Gaye             | 1971 | Rock  |
| Are You Experienced? | Jimi Hendrix Experience | 1967 | Rock  |
| The Joshua Tree      | U2                      | 1987 | Rock  |
| Abbey Road           | The Beatles             | 1969 | Rock  |
| Rumours              | Fleetwood Mac           | 1977 | Rock  |
| Sun Sessions         | Elvis Presley           | 1976 | Rock  |
| Thriller             | Michael Jackson         | 1982 | Pop   |
| Exile on Main Street | The Rolling Stones      | 1972 | Rock  |
| Born to Run          | Bruce Springsteen       | 1975 | Rock  |
| London Calling       | The Clash               | 1980 | Rock  |
| Hotel California     | Led Zeppelin            | 1973 | Rock  |
| Led Zeppelin         | Led Zeppelin            | 1971 | Rock  |
| IV                   | Led Zeppelin            | 1971 | Rock  |
| Synchronicity        | Police                  | 1983 | Pop   |

Notice the application has been populated with some [default data](#). Interact with it by deleting a few existing albums and creating a few new ones.

You can verify your application is using your MongoDB database by connecting to it and querying the *musicdb* database:

```
mongo serverAddress:port/musicdb -u user -p password
use musicdb
db.album.find()

{ "_id" : ObjectId("5c1bb6f5df0e66f13f9c446d"), "title" : "Nevermind", "artist" : "Nirvana", "releaseYear" :
"1991", "genre" : "Rock", "trackCount" : 0, "_class" : "org.cloudfoundry.samples.music.domain.Album" }
{ "_id" : ObjectId("5c1bb6f5df0e66f13f9c446e"), "title" : "Pet Sounds", "artist" : "The Beach Boys",
"releaseYear" : "1966", "genre" : "Rock", "trackCount" : 0, "_class" :
"org.cloudfoundry.samples.music.domain.Album" }
{ "_id" : ObjectId("5c1bb6f5df0e66f13f9c446f"), "title" : "What's Going On", "artist" : "Marvin Gaye",
"releaseYear" : "1971", "genre" : "Rock", "trackCount" : 0, "_class" :
"org.cloudfoundry.samples.music.domain.Album" }
...
...
```

The preceding example uses the [mongo shell](#) to connect to the MongoDB database and query it. You can also verify your changes are persisted by stopping your application, restarting it, and navigating back to it in your browser. Notice the changes you have made are still there.

## Create a Cosmos DB database

To create a Cosmos DB database in Azure using Open Service Broker, use the `svcat provision` command:

```
svcat provision musicdb --class azure-cosmosdb-mongo-account --plan account --params-json '{
  "location": "eastus",
  "resourceGroup": "MyResourceGroup",
  "ipFilters" : {
    "allowedIPRanges" : ["0.0.0.0/0"]
  }
}'
```

The preceding command provisions a Cosmos DB database in Azure in the resource group *MyResourceGroup* in the *eastus* region. More information on *resourceGroup*, *location*, and other Azure-specific JSON parameters is available in the [Cosmos DB module reference documentation](#).

To verify your database has completed provisioning, use the `svcat get instance` command:

```
$ svcat get instance musicdb

  NAME      NAMESPACE      CLASS      PLAN      STATUS
+-----+-----+-----+-----+
  musicdb  default  azure-cosmosdb-mongo-account  account  Ready
```

Your database is ready when see *Ready* under *STATUS*.

Once your database has completed provisioning, you need to bind its metadata to a [Kubernetes secret](#). Other applications can then access that data after it has been bound to a secret. To bind the metadata of your database to a secret, use the `svcat bind` command:

```
$ svcat bind musicdb

Name:      musicdb
Namespace: default
Status:
Secret:    musicdb
Instance:  musicdb

Parameters:
No parameters defined
```

## Use the Cosmos DB database with your application

To use the Cosmos DB database with your application, you need to know the URI to connect to it. To get this information, use the `kubectl get secret` command:

```
$ kubectl get secret musicdb -o=jsonpath='{.data.uri}' | base64 --decode

mongodb://12345678-90ab-cdef-1234-
567890abcdef:aaaabbbbccccdddeeeeffffggghhhiiiijjjjkkkkllllmmmmnnnnooooppppqqqrrrrsssstttuuuvvvv@098765432
-aaaa-bbbb-cccc-1234567890ab.documents.azure.com:10255/?ssl=true&replicaSet=globaldb
```

The preceding command gets the *musicdb* secret and displays only the URI. Secrets are stored in base64 format so the preceding command also decodes it.

Using the URI of the Cosmos DB database, update *src/main/resources/application.yml* to use it:

```
...
---
spring:
  profiles: mongodb
  data:
    mongodb:
      uri: "mongodb://12345678-90ab-cdef-1234-
567890abcdef:aaaabbbbccccdddeeeeffffggghhhiiiijjjjkkkkllllmmmmnnnnooooppppqqqrrrrsssstttuuuvvvv@098765432
-aaaa-bbbb-cccc-1234567890ab.documents.azure.com:10255/?ssl=true&replicaSet=globaldb"
      database: musicdb
```

Updating the URI, which contains a username and password, directly to this file is for **development use only** and **should never be added to version control**.

Rebuild and start your application to begin using the Cosmos DB database:

```
./gradlew clean assemble

java -jar -Dspring.profiles.active=mongodb build/libs/spring-music-1.0.jar
```

Notice your application still uses the *mongodb* profile and a URI that begins with *mongodb://* to connect to the Cosmos DB database. The [Azure Cosmos DB API for MongoDB](#) provides this compatibility. It allows your application to continue to operate as if it is using a MongoDB database, but it is actually using Cosmos DB.

Navigate to `http://localhost:8080` in your browser. Notice the default data has been restored. Interact with it by deleting a few existing albums and creating a few new ones. You can verify your changes are persisted by stopping your application, restarting it, and navigating back to it in your browser. Notice the changes you have made are still there. The changes are persisted to the Cosmos DB you created using Open Service Broker for Azure.

# Run your application on your AKS cluster

You can use [Azure Dev Spaces](#) to deploy the application to your AKS cluster. Azure Dev Spaces helps you generate artifacts, such as Dockerfiles and Helm charts, and deploy and run an application in AKS.

To enable Azure Dev Spaces in your AKS cluster:

```
az aks enable-addons --addons http_application_routing -g MyResourceGroup -n MyAKS  
az aks use-dev-spaces -g MyResourceGroup -n MyAKS
```

Use the Azure Dev Spaces tooling to prepare your application to run in AKS:

```
azds prep --public
```

This command generates several artifacts, including a *charts/* folder, which is your Helm chart, at the root of the project. This command cannot generate a *Dockerfile* for this specific project so you have to create it.

Create a file at the root of your project named *Dockerfile* with this content:

```
FROM openjdk:8-jdk-alpine  
EXPOSE 8080  
WORKDIR /app  
COPY build/libs/spring-music-1.0.jar .  
ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-Dspring.profiles.active=mongodb","-jar","/app/spring-music-1.0.jar"]
```

In addition, you need to update the *configurations.develop.build* property in *azds.yaml* to *false*:

```
...  
configurations:  
  develop:  
    build:  
      useGitIgnore: false
```

You also need to update the *containerPort* attribute to *8080* in *charts/spring-music/templates/deployment.yaml*:

```
...  
spec:  
  ...  
  template:  
    ...  
    spec:  
      containers:  
        - name: {{ .Chart.Name }}  
          image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"  
          imagePullPolicy: {{ .Values.image.pullPolicy }}  
          ports:  
            - name: http  
              containerPort: 8080  
              protocol: TCP
```

To deploy your application to AKS:

```
$ azds up

Using dev space 'default' with target 'MyAKS'
Synchronizing files...1m 18s
Installing Helm chart...5s
Waiting for container image build...23s
Building container image...
Step 1/5 : FROM openjdk:8-jdk-alpine
Step 2/5 : EXPOSE 8080
Step 3/5 : WORKDIR /app
Step 4/5 : COPY build/libs/spring-music-1.0.jar .
Step 5/5 : ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-Dspring.profiles.active=mongodb","-jar","/app/spring-music-1.0.jar"]
Built container image in 21s
Waiting for container...8s
Service 'spring-music' port 'http' is available at http://spring-music.1234567890abcdef1234.eastus.aksapp.io/
Service 'spring-music' port 8080 (TCP) is available at http://localhost:57892
press Ctrl+C to detach
...
```

Navigate to the URL displayed in the logs. In the preceding example, you would use <http://spring-music.1234567890abcdef1234.eastus.aksapp.io/>.

Verify you see the application along with your changes.

## Next steps

This article described how to update an existing application from using MongoDB to using Cosmos DB API for MongoDB. This article also covered how to provision a Cosmos DB service using Open Service Broker for Azure and deploying that application to AKS with Azure Dev Spaces.

For more information about Cosmos DB, Open Service Broker for Azure, and Azure Dev Spaces, see:

- [Cosmos DB](#)
- [Open Service Broker for Azure](#)
- [Develop with Dev Spaces](#)

# Using OpenFaaS on AKS

2/25/2020 • 4 minutes to read • [Edit Online](#)

OpenFaaS is a framework for building serverless functions through the use of containers. As an open source project, it has gained large-scale adoption within the community. This document details installing and using OpenFaaS on an Azure Kubernetes Service (AKS) cluster.

## Prerequisites

In order to complete the steps within this article, you need the following.

- Basic understanding of Kubernetes.
- An Azure Kubernetes Service (AKS) cluster and AKS credentials configured on your development system.
- Azure CLI installed on your development system.
- Git command-line tools installed on your system.

## Add the OpenFaaS helm chart repo

OpenFaaS maintains its own helm charts to keep up to date with all the latest changes.

```
helm repo add openfaas https://openfaas.github.io/faas-netes/
helm repo update
```

## Deploy OpenFaaS

As a good practice, OpenFaaS and OpenFaaS functions should be stored in their own Kubernetes namespace.

Create a namespace for the OpenFaaS system and functions:

```
kubectl apply -f https://raw.githubusercontent.com/openfaas/faas-netes/master/namespaces.yaml
```

Generate a password for the OpenFaaS UI Portal and REST API:

```
# generate a random password
PASSWORD=$(head -c 12 /dev/urandom | shasum| cut -d' ' -f1)

kubectl -n openfaas create secret generic basic-auth \
--from-literal=basic-auth-user=admin \
--from-literal=basic-auth-password="$PASSWORD"
```

You can get the value of the secret with `echo $PASSWORD`.

The password we create here will be used by the helm chart to enable basic authentication on the OpenFaaS Gateway, which is exposed to the Internet through a cloud LoadBalancer.

A Helm chart for OpenFaaS is included in the cloned repository. Use this chart to deploy OpenFaaS into your AKS cluster.

```
helm upgrade openfaas --install openfaas/openfaas \
--namespace openfaas \
--set basic_auth=true \
--set functionNamespace=openfaas-fn \
--set serviceType=LoadBalancer
```

Output:

```
NAME:    openfaas
LAST DEPLOYED: Wed Feb 28 08:26:11 2018
NAMESPACE: openfaas
STATUS:  DEPLOYED

RESOURCES:
==> v1/ConfigMap
NAME          DATA   AGE
prometheus-config   2      20s
alertmanager-config 1      20s

{snip}

NOTES:
To verify that openfaas has started, run:

  kubectl --namespace=openfaas get deployments -l "release=openfaas, app=openfaas"
```

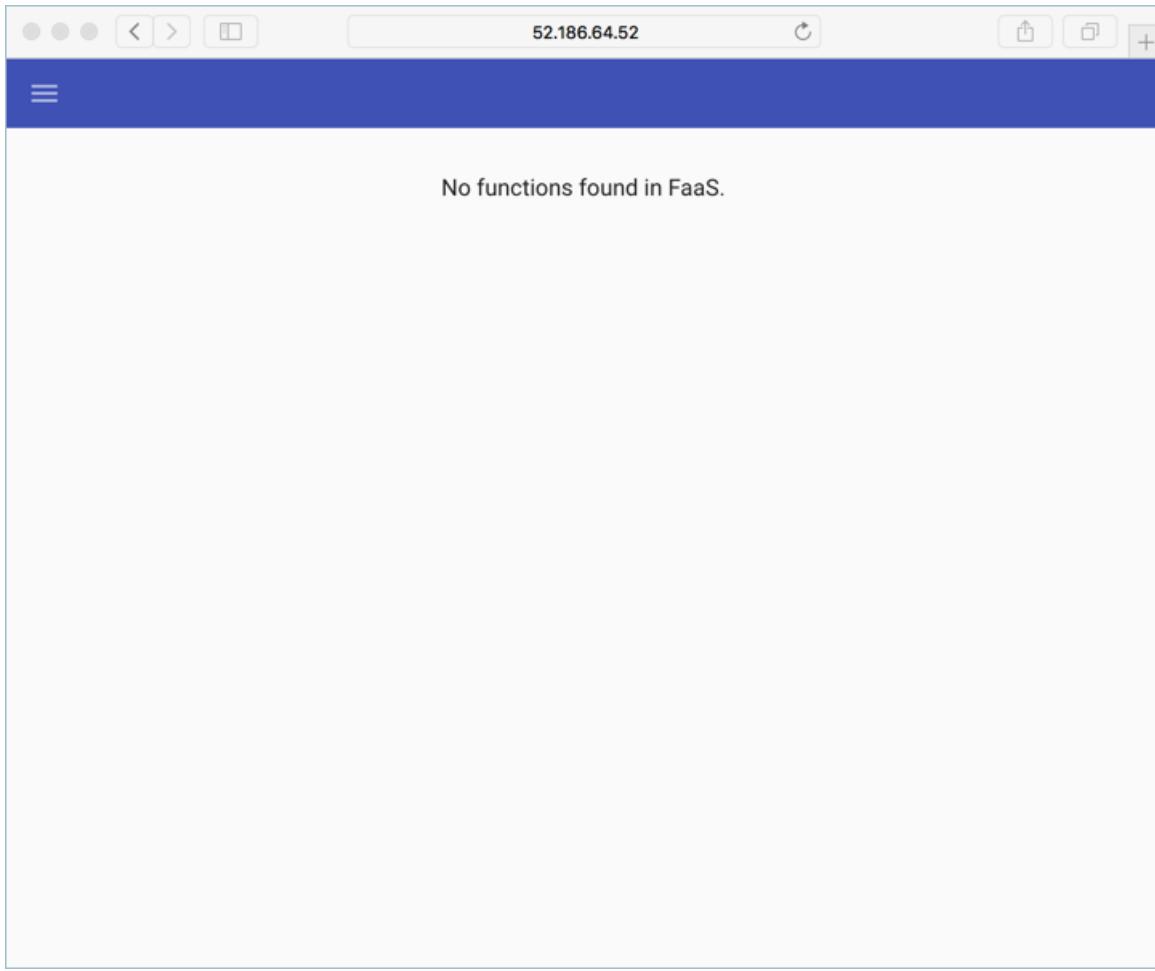
A public IP address is created for accessing the OpenFaaS gateway. To retrieve this IP address, use the [kubectl get service](#) command. It may take a minute for the IP address to be assigned to the service.

```
kubectl get service -l component=gateway --namespace openfaas
```

Output.

| NAME             | TYPE         | CLUSTER-IP   | EXTERNAL-IP  | PORT(S)        | AGE |
|------------------|--------------|--------------|--------------|----------------|-----|
| gateway          | ClusterIP    | 10.0.156.194 | <none>       | 8080/TCP       | 7m  |
| gateway-external | LoadBalancer | 10.0.28.18   | 52.186.64.52 | 8080:30800/TCP | 7m  |

To test the OpenFaaS system, browse to the external IP address on port 8080, <http://52.186.64.52:8080> in this example. You will be prompted to log in. To fetch your password, enter `echo $PASSWORD`.



Finally, install the OpenFaaS CLI. This example used brew, see the [OpenFaaS CLI documentation](#) for more options.

```
brew install faas-cli
```

Set `$OPENFAAS_URL` to the public IP found above.

Log in with the Azure CLI:

```
export OPENFAAS_URL=http://52.186.64.52:8080
echo -n $PASSWORD | ./faas-cli login -g $OPENFAAS_URL -u admin --password-stdin
```

## Create first function

Now that OpenFaaS is operational, create a function using the OpenFaaS portal.

Click on **Deploy New Function** and search for **Figlet**. Select the Figlet function, and click **Deploy**.

## Deploy A New Function

X

FROM STORE      MANUALLY

---

Search for Function

figlet

---

**Figlet**

OpenFaaS Figlet image. This repository comes with the blog post <http://jmkhael.io/create-a-serverless-ascii-banner-with-faas/>

F

🔗

---

CLOSE DIALOG      DEPLOY

Use curl to invoke the function. Replace the IP address in the following example with that of your OpenFaas gateway.

```
curl -X POST http://52.186.64.52:8080/function/figlet -d "Hello Azure"
```

## Output:



## Create second function

Now create a second function. This example will be deployed using the OpenFaaS CLI and includes a custom container image and retrieving data from a Cosmos DB. Several items need to be configured before creating the function.

First, create a new resource group for the Cosmos DB.

```
az group create --name serverless-backing --location eastus
```

Deploy a CosmosDB instance of kind `MongoDB`. The instance needs a unique name, update `openfaas-cosmos` to something unique to your environment.

```
az cosmosdb create --resource-group serverless-backing --name openfaas-cosmos --kind MongoDB
```

Get the Cosmos database connection string and store it in a variable.

Update the value for the `--resource-group` argument to the name of your resource group, and the `--name` argument to the name of your Cosmos DB.

```
COSMOS=$(az cosmosdb list-connection-strings \
--resource-group serverless-backing \
--name openfaas-cosmos \
--query connectionStrings[0].connectionString \
--output tsv)
```

Now populate the Cosmos DB with test data. Create a file named `plans.json` and copy in the following json.

```
{
  "name" : "two_person",
  "friendlyName" : "Two Person Plan",
  "portionSize" : "1-2 Person",
  "mealsPerWeek" : "3 Unique meals per week",
  "price" : 72,
  "description" : "Our basic plan, delivering 3 meals per week, which will feed 1-2 people.",
  "__v" : 0
}
```

Use the `mongoimport` tool to load the CosmosDB instance with data.

If needed, install the MongoDB tools. The following example installs these tools using brew, see the [MongoDB documentation](#) for other options.

```
brew install mongodb
```

Load the data into the database.

```
mongoimport --uri=$COSMOS -c plans < plans.json
```

Output:

```
2018-02-19T14:42:14.313+0000      connected to: localhost
2018-02-19T14:42:14.918+0000      imported 1 document
```

Run the following command to create the function. Update the value of the `-g` argument with your OpenFaaS gateway address.

```
faas-cli deploy -g http://52.186.64.52:8080 --image=shanepeckham/openfaascosmos --name=cosmos-query --
env=NODE_ENV=$COSMOS
```

Once deployed, you should see your newly created OpenFaaS endpoint for the function.

```
Deployed. 202 Accepted.
URL: http://52.186.64.52:8080/function/cosmos-query
```

Test the function using curl. Update the IP address with the OpenFaaS gateway address.

```
curl -s http://52.186.64.52:8080/function/cosmos-query
```

Output:

```
[{"ID": "", "Name": "two_person", "FriendlyName": "", "PortionSize": "", "MealsPerWeek": "", "Price": 72, "Description": "Our basic plan, delivering 3 meals per week, which will feed 1-2 people."}]
```

You can also test the function within the OpenFaaS UI.

The screenshot shows the OpenFaaS Invoke function interface. At the top, there are browser-style buttons for back, forward, and refresh, followed by the IP address '52.186.64.52'. Below the address bar is a toolbar with icons for upload, download, and a plus sign. The main area has a title 'Invoke function' and a button labeled 'INVOKE'. Underneath are three radio buttons: 'Text' (selected), 'JSON', and 'Download'. A section labeled 'Request body' is empty. Below that is 'Response status' with the value '200'. Under 'Round-trip (s)' is the value '0.721'. The 'Response body' section contains a JSON array with one element:

```
[{"ID": "", "Name": "two_person", "FriendlyName": "", "PortionSize": "", "MealsPerWeek": "", "Price": 72, "Description": "Our basic plan, delivering 3 meals per week, which will feed 1-2 people."}]
```

## Next Steps

You can continue to learn with the OpenFaaS workshop through a set of hands-on labs that cover topics such as how to create your own GitHub bot, consuming secrets, viewing metrics, and auto-scaling.

# Running Apache Spark jobs on AKS

2/25/2020 • 6 minutes to read • [Edit Online](#)

Apache Spark is a fast engine for large-scale data processing. As of the [Spark 2.3.0 release](#), Apache Spark supports native integration with Kubernetes clusters. Azure Kubernetes Service (AKS) is a managed Kubernetes environment running in Azure. This document details preparing and running Apache Spark jobs on an Azure Kubernetes Service (AKS) cluster.

## Prerequisites

In order to complete the steps within this article, you need the following.

- Basic understanding of Kubernetes and [Apache Spark](#).
- [Docker Hub](#) account, or an [Azure Container Registry](#).
- Azure CLI [installed](#) on your development system.
- [JDK 8](#) installed on your system.
- SBT ([Scala Build Tool](#)) installed on your system.
- Git command-line tools installed on your system.

## Create an AKS cluster

Spark is used for large-scale data processing and requires that Kubernetes nodes are sized to meet the Spark resources requirements. We recommend a minimum size of `Standard_D3_v2` for your Azure Kubernetes Service (AKS) nodes.

If you need an AKS cluster that meets this minimum recommendation, run the following commands.

Create a resource group for the cluster.

```
az group create --name mySparkCluster --location eastus
```

Create a Service Principal for the cluster. After it is created, you will need the Service Principal appId and password for the next command.

```
az ad sp create-for-rbac --name SparkSP
```

Create the AKS cluster with nodes that are of size `Standard_D3_v2`, and values of appId and password passed as service-principal and client-secret parameters.

```
az aks create --resource-group mySparkCluster --name mySparkCluster --node-vm-size Standard_D3_v2 --generate-ssh-keys --service-principal <APPID> --client-secret <PASSWORD>
```

Connect to the AKS cluster.

```
az aks get-credentials --resource-group mySparkCluster --name mySparkCluster
```

If you are using Azure Container Registry (ACR) to store container images, configure authentication between AKS and ACR. See the [ACR authentication documentation](#) for these steps.

# Build the Spark source

Before running Spark jobs on an AKS cluster, you need to build the Spark source code and package it into a container image. The Spark source includes scripts that can be used to complete this process.

Clone the Spark project repository to your development system.

```
git clone -b branch-2.4 https://github.com/apache/spark
```

Change into the directory of the cloned repository and save the path of the Spark source to a variable.

```
cd spark  
sparkdir=$(pwd)
```

If you have multiple JDK versions installed, set `JAVA_HOME` to use version 8 for the current session.

```
export JAVA_HOME=`/usr/libexec/java_home -d 64 -v "1.8*`
```

Run the following command to build the Spark source code with Kubernetes support.

```
./build/mvn -Pkubernetes -DskipTests clean package
```

The following commands create the Spark container image and push it to a container image registry. Replace `registry.example.com` with the name of your container registry and `v1` with the tag you prefer to use. If using Docker Hub, this value is the registry name. If using Azure Container Registry (ACR), this value is the ACR login server name.

```
REGISTRY_NAME=registry.example.com  
REGISTRY_TAG=v1
```

```
./bin/docker-image-tool.sh -r $REGISTRY_NAME -t $REGISTRY_TAG build
```

Push the container image to your container image registry.

```
./bin/docker-image-tool.sh -r $REGISTRY_NAME -t $REGISTRY_TAG push
```

# Prepare a Spark job

Next, prepare a Spark job. A jar file is used to hold the Spark job and is needed when running the `spark-submit` command. The jar can be made accessible through a public URL or pre-packaged within a container image. In this example, a sample jar is created to calculate the value of Pi. This jar is then uploaded to Azure storage. If you have an existing jar, feel free to substitute.

Create a directory where you would like to create the project for a Spark job.

```
mkdir myprojects  
cd myprojects
```

Create a new Scala project from a template.

```
sbt new sbt/scala-seed.g8
```

When prompted, enter `SparkPi` for the project name.

```
name [Scala Seed Project]: SparkPi
```

Navigate to the newly created project directory.

```
cd sparkpi
```

Run the following commands to add an SBT plugin, which allows packaging the project as a jar file.

```
touch project/assembly.sbt
echo 'addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.14.10")' >> project/assembly.sbt
```

Run these commands to copy the sample code into the newly created project and add all necessary dependencies.

```
EXAMPLESDIR="src/main/scala/org/apache/spark/examples"
mkdir -p $EXAMPLESDIR
cp $sparkdir/examples/$EXAMPLESDIR/SparkPi.scala $EXAMPLESDIR/SparkPi.scala

cat <<EOT >> build.sbt
// https://mvnrepository.com/artifact/org.apache.spark/spark-sql
libraryDependencies += "org.apache.spark" %% "spark-sql" % "2.3.0" % "provided"
EOT

sed -ie 's/scalaVersion.*/scalaVersion := "2.11.11"/' build.sbt
sed -ie 's/name.*/name := "SparkPi"/' build.sbt
```

To package the project into a jar, run the following command.

```
sbt assembly
```

After successful packaging, you should see output similar to the following.

```
[info] Packaging /Users/me/myprojects/sparkpi/target/scala-2.11/SparkPi-assembly-0.1.0-SNAPSHOT.jar ...
[info] Done packaging.
[success] Total time: 10 s, completed Mar 6, 2018 11:07:54 AM
```

## Copy job to storage

Create an Azure storage account and container to hold the jar file.

```
RESOURCE_GROUP=sparkdemo
STORAGE_ACCT=sparkdemo$RANDOM
az group create --name $RESOURCE_GROUP --location eastus
az storage account create --resource-group $RESOURCE_GROUP --name $STORAGE_ACCT --sku Standard_LRS
export AZURE_STORAGE_CONNECTION_STRING=`az storage account show-connection-string --resource-group
$RESOURCE_GROUP --name $STORAGE_ACCT -o tsv`
```

Upload the jar file to the Azure storage account with the following commands.

```

CONTAINER_NAME=jars
BLOB_NAME=SparkPi-assembly-0.1.0-SNAPSHOT.jar
FILE_TO_UPLOAD=target/scala-2.11/SparkPi-assembly-0.1.0-SNAPSHOT.jar

echo "Creating the container..."
az storage container create --name $CONTAINER_NAME
az storage container set-permission --name $CONTAINER_NAME --public-access blob

echo "Uploading the file..."
az storage blob upload --container-name $CONTAINER_NAME --file $FILE_TO_UPLOAD --name $BLOB_NAME

jarUrl=$(az storage blob url --container-name $CONTAINER_NAME --name $BLOB_NAME | tr -d '"')

```

Variable `jarUrl` now contains the publicly accessible path to the jar file.

## Submit a Spark job

Start kube-proxy in a separate command-line with the following code.

```
kubectl proxy
```

Navigate back to the root of Spark repository.

```
cd $sparkdir
```

Create a service account that has sufficient permissions for running a job.

```

kubectl create serviceaccount spark
kubectl create clusterrolebinding spark-role --clusterrole=edit --serviceaccount=default:spark --
namespace=default

```

Submit the job using `spark-submit`.

```

./bin/spark-submit \
--master k8s://http://127.0.0.1:8001 \
--deploy-mode cluster \
--name spark-pi \
--class org.apache.spark.examples.SparkPi \
--conf spark.executor.instances=3 \
--conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \
--conf spark.kubernetes.container.image=$REGISTRY_NAME/spark:$REGISTRY_TAG \
$jarUrl

```

This operation starts the Spark job, which streams job status to your shell session. While the job is running, you can see Spark driver pod and executor pods using the `kubectl get pods` command. Open a second terminal session to run these commands.

| NAME   | READY | STATUS   | RESTARTS | AGE |
|--|-------|----------|----------|-----|
| spark-pi-2232778d0f663768ab27edc35cb73040-driver | 1/1   | Running  | 0        | 16s |
| spark-pi-2232778d0f663768ab27edc35cb73040-exec-1 | 0/1   | Init:0/1 | 0        | 4s  |
| spark-pi-2232778d0f663768ab27edc35cb73040-exec-2 | 0/1   | Init:0/1 | 0        | 4s  |
| spark-pi-2232778d0f663768ab27edc35cb73040-exec-3 | 0/1   | Init:0/1 | 0        | 4s  |

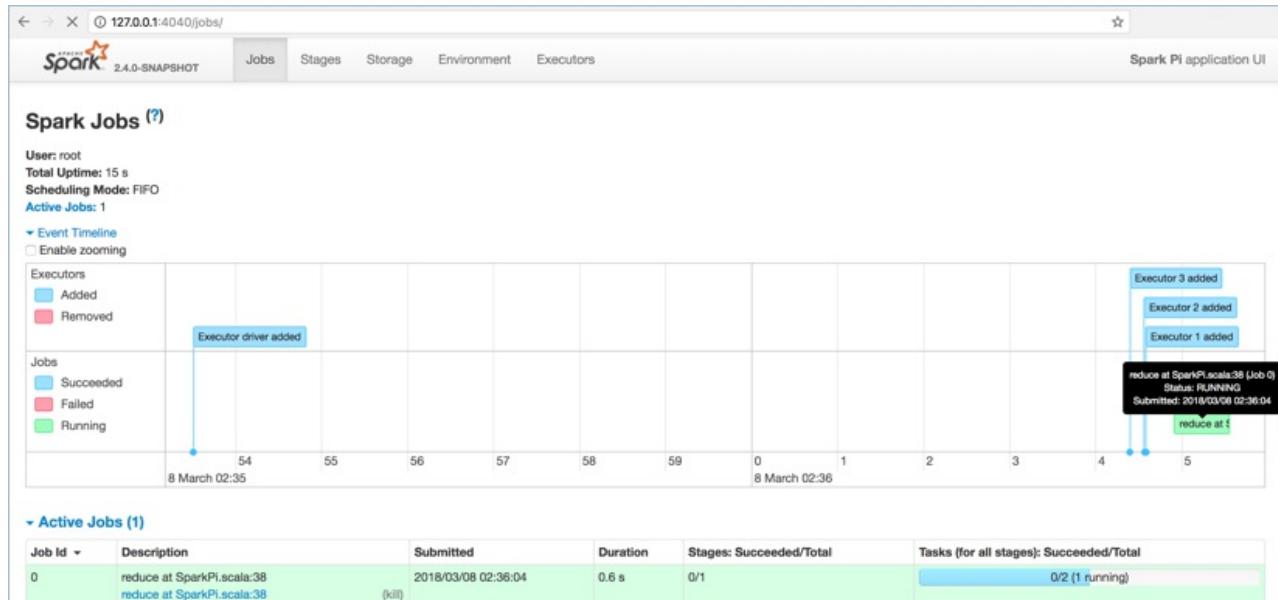
While the job is running, you can also access the Spark UI. In the second terminal session, use the

```
kubectl port-forward
```

 command provide access to Spark UI.

```
kubectl port-forward spark-pi-2232778d0f663768ab27edc35cb73040-driver 4040:4040
```

To access Spark UI, open the address `127.0.0.1:4040` in a browser.



## Get job results and logs

After the job has finished, the driver pod will be in a "Completed" state. Get the name of the pod with the following command.

```
kubectl get pods --show-all
```

Output:

| NAME   | READY | STATUS    | RESTARTS | AGE |
|--|-------|-----------|----------|-----|
| spark-pi-2232778d0f663768ab27edc35cb73040-driver | 0/1   | Completed | 0        | 1m  |

Use the `kubectl logs` command to get logs from the spark driver pod. Replace the pod name with your driver pod's name.

```
kubectl logs spark-pi-2232778d0f663768ab27edc35cb73040-driver
```

Within these logs, you can see the result of the Spark job, which is the value of Pi.

```
Pi is roughly 3.152155760778804
```

## Package jar with container image

In the above example, the Spark jar file was uploaded to Azure storage. Another option is to package the jar file into custom-built Docker images.

To do so, find the `dockerfile` for the Spark image located at

```
$sparkdir/resource-managers/kubernetes/docker/src/main/dockerfiles/spark/
```

 directory. Add an `ADD` statement for

the Spark job `jar` somewhere between `WORKDIR` and `ENTRYPOINT` declarations.

Update the jar path to the location of the `SparkPi-assembly-0.1.0-SNAPSHOT.jar` file on your development system. You can also use your own custom jar file.

```
WORKDIR /opt/spark/work-dir  
  
ADD /path/to/SparkPi-assembly-0.1.0-SNAPSHOT.jar SparkPi-assembly-0.1.0-SNAPSHOT.jar  
  
ENTRYPOINT [ "/opt/entrypoint.sh" ]
```

Build and push the image with the included Spark scripts.

```
./bin/docker-image-tool.sh -r <your container repository name> -t <tag> build  
./bin/docker-image-tool.sh -r <your container repository name> -t <tag> push
```

When running the job, instead of indicating a remote jar URL, the `local://` scheme can be used with the path to the jar file in the Docker image.

```
./bin/spark-submit \  
--master k8s://https://<k8s-apiserver-host>:<k8s-apiserver-port> \  
--deploy-mode cluster \  
--name spark-pi \  
--class org.apache.spark.examples.SparkPi \  
--conf spark.executor.instances=3 \  
--conf spark.kubernetes.authenticate.driver.serviceAccountName=spark \  
--conf spark.kubernetes.container.image=<spark-image> \  
local:///opt/spark/work-dir/<your-jar-name>.jar
```

#### WARNING

From Spark documentation: "The Kubernetes scheduler is currently experimental. In future versions, there may be behavioral changes around configuration, container images and entrypoints".

## Next steps

Check out Spark documentation for more details.

[Spark documentation](#)

# Use GPUs for compute-intensive workloads on Azure Kubernetes Service (AKS)

2/25/2020 • 7 minutes to read • [Edit Online](#)

Graphical processing units (GPUs) are often used for compute-intensive workloads such as graphics and visualization workloads. AKS supports the creation of GPU-enabled node pools to run these compute-intensive workloads in Kubernetes. For more information on available GPU-enabled VMs, see [GPU optimized VM sizes in Azure](#). For AKS nodes, we recommend a minimum size of `Standard_NC6`.

## NOTE

GPU-enabled VMs contain specialized hardware that is subject to higher pricing and region availability. For more information, see the [pricing](#) tool and [region availability](#).

Currently, using GPU-enabled node pools is only available for Linux node pools.

## Before you begin

This article assumes that you have an existing AKS cluster with nodes that support GPUs. Your AKS cluster must run Kubernetes 1.10 or later. If you need an AKS cluster that meets these requirements, see the first section of this article to [create an AKS cluster](#).

You also need the Azure CLI version 2.0.64 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an AKS cluster

If you need an AKS cluster that meets the minimum requirements (GPU-enabled node and Kubernetes version 1.10 or later), complete the following steps. If you already have an AKS cluster that meets these requirements, [skip to the next section](#).

First, create a resource group for the cluster using the `az group create` command. The following example creates a resource group name `myResourceGroup` in the `eastus` region:

```
az group create --name myResourceGroup --location eastus
```

Now create an AKS cluster using the `az aks create` command. The following example creates a cluster with a single node of size `Standard_NC6`:

```
az aks create \
--resource-group myResourceGroup \
--name myAKSCluster \
--node-vm-size Standard_NC6 \
--node-count 1
```

Get the credentials for your AKS cluster using the `az aks get-credentials` command:

```
az aks get-credentials --resource-group myResourceGroup --name myAKSCluster
```

## Install nVidia drivers

Before the GPUs in the nodes can be used, you must deploy a DaemonSet for the NVIDIA device plugin. This DaemonSet runs a pod on each node to provide the required drivers for the GPUs.

First, create a namespace using the [kubectl create namespace](#) command, such as *gpu-resources*:

```
kubectl create namespace gpu-resources
```

Create a file named *nvidia-device-plugin-ds.yaml* and paste the following YAML manifest. This manifest is provided as part of the [NVIDIA device plugin for Kubernetes project](#).

```
apiVersion: extensions/v1beta1
kind: DaemonSet
metadata:
  name: nvidia-device-plugin-daemonset
  namespace: gpu-resources
spec:
  updateStrategy:
    type: RollingUpdate
  template:
    metadata:
      # Mark this pod as a critical add-on; when enabled, the critical add-on scheduler
      # reserves resources for critical add-on pods so that they can be rescheduled after
      # a failure. This annotation works in tandem with the toleration below.
      annotations:
        scheduler.alpha.kubernetes.io/critical-pod: ""
      labels:
        name: nvidia-device-plugin-ds
    spec:
      tolerations:
        # Allow this pod to be rescheduled while the node is in "critical add-ons only" mode.
        # This, along with the annotation above marks this pod as a critical add-on.
        - key: CriticalAddonsOnly
          operator: Exists
        - key: nvidia.com/gpu
          operator: Exists
          effect: NoSchedule
      containers:
        - image: nvidia/k8s-device-plugin:1.11
          name: nvidia-device-plugin-ctr
          securityContext:
            allowPrivilegeEscalation: false
            capabilities:
              drop: ["ALL"]
          volumeMounts:
            - name: device-plugin
              mountPath: /var/lib/kubelet/device-plugins
      volumes:
        - name: device-plugin
          hostPath:
            path: /var/lib/kubelet/device-plugins
```

Now use the [kubectl apply](#) command to create the DaemonSet and confirm the nVidia device plugin is created successfully, as shown in the following example output:

```
$ kubectl apply -f nvidia-device-plugin-ds.yaml
daemonset "nvidia-device-plugin" created
```

## Confirm that GPUs are schedutable

With your AKS cluster created, confirm that GPUs are schedutable in Kubernetes. First, list the nodes in your cluster using the [kubectl get nodes](#) command:

```
$ kubectl get nodes
```

| NAME                     | STATUS | ROLES | AGE | VERSION |
|--------------------------|--------|-------|-----|---------|
| aks-nodepool1-28993262-0 | Ready  | agent | 13m | v1.12.7 |

Now use the [kubectl describe node](#) command to confirm that the GPUs are schedutable. Under the *Capacity* section, the GPU should list as `nvidia.com/gpu: 1`.

The following condensed example shows that a GPU is available on the node named `aks-nodepool1-18821093-0`:

```

$ kubectl describe node aks-nodepool1-28993262-0

Name:           aks-nodepool1-28993262-0
Roles:          agent
Labels:         accelerator=nvidia

[...]

Capacity:
attachable-volumes-azure-disk: 24
cpu:                          6
ephemeral-storage:            101584140Ki
hugepages-1Gi:                0
hugepages-2Mi:                0
memory:                      57713784Ki
nvidia.com/gpu:               1
pods:                         110

Allocatable:
attachable-volumes-azure-disk: 24
cpu:                          5916m
ephemeral-storage:            93619943269
hugepages-1Gi:                0
hugepages-2Mi:                0
memory:                      51702904Ki
nvidia.com/gpu:               1
pods:                         110

System Info:
Machine ID:          b0cd6fb49ffe4900b56ac8df2eaa0376
System UUID:          486A1C08-C459-6F43-AD6B-E9CD0F8AEC17
Boot ID:              f134525f-385d-4b4e-89b8-989f3abb490b
Kernel Version:       4.15.0-1040-azure
OS Image:             Ubuntu 16.04.6 LTS
Operating System:    linux
Architecture:        amd64
Container Runtime Version: docker://1.13.1
Kubelet Version:      v1.12.7
Kube-Proxy Version:   v1.12.7
PodCIDR:              10.244.0.0/24
ProviderID:
azure:///subscriptions/<guid>/resourceGroups/MC_myResourceGroup_myAKSCluster_eastus/providers/Microsoft.Compute/virtualMachines/aks-nodepool1-28993262-0

Non-terminated Pods: (9 in total)
  Namespace          Name                 CPU Requests  CPU Limits  Memory
  Requests  Memory Limits  AGE
  -----  -----  -----  -----
  -  -  -  -
  kube-system        nvidia-device-plugin-daemonset-bbjlq    0 (0%)     0 (0%)     0 (0%)
  0 (0%)    2m39s

[...]

```

## Run a GPU-enabled workload

To see the GPU in action, schedule a GPU-enabled workload with the appropriate resource request. In this example, let's run a [Tensorflow](#) job against the [MNIST dataset](#).

Create a file named `samples-tf-mnist-demo.yaml` and paste the following YAML manifest. The following job manifest includes a resource limit of `nvidia.com/gpu: 1`:

#### NOTE

If you receive a version mismatch error when calling into drivers, such as, CUDA driver version is insufficient for CUDA runtime version, review the nVidia driver matrix compatibility chart - <https://docs.nvidia.com/Deploy/CUDA-Compatibility/index.html>

```
apiVersion: batch/v1
kind: Job
metadata:
  labels:
    app: samples-tf-mnist-demo
  name: samples-tf-mnist-demo
spec:
  template:
    metadata:
      labels:
        app: samples-tf-mnist-demo
    spec:
      containers:
        - name: samples-tf-mnist-demo
          image: microsoft/samples-tf-mnist-demo:gpu
          args: [--max_steps, "500"]
          imagePullPolicy: IfNotPresent
          resources:
            limits:
              nvidia.com/gpu: 1
          restartPolicy: OnFailure
```

Use the `kubectl apply` command to run the job. This command parses the manifest file and creates the defined Kubernetes objects:

```
kubectl apply -f samples-tf-mnist-demo.yaml
```

## View the status and output of the GPU-enabled workload

Monitor the progress of the job using the `kubectl get jobs` command with the `--watch` argument. It may take a few minutes to first pull the image and process the dataset. When the `COMPLETIONS` column shows `1/1`, the job has successfully finished. Exit the `kubectl --watch` command with `Ctrl-C`:

```
$ kubectl get jobs samples-tf-mnist-demo --watch
NAME          COMPLETIONS   DURATION   AGE
samples-tf-mnist-demo  0/1          3m29s   3m29s
samples-tf-mnist-demo  1/1          3m10s   3m36s
```

To look at the output of the GPU-enabled workload, first get the name of the pod with the `kubectl get pods` command:

```
$ kubectl get pods --selector app=samples-tf-mnist-demo
NAME           READY   STATUS    RESTARTS   AGE
samples-tf-mnist-demo-mtd44  0/1     Completed   0          4m39s
```

Now use the `kubectl logs` command to view the pod logs. The following example pod logs confirm that the appropriate GPU device has been discovered, `Tesla K80`. Provide the name for your own pod:

```
$ kubectl logs samples-tf-mnist-demo-smnrv6

2019-05-16 16:08:31.258328: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2019-05-16 16:08:31.396846: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1030] Found device 0 with
properties:
name: Tesla K80 major: 3 minor: 7 memoryClockRate(GHz): 0.8235
pciBusID: 2fd7:00:00.0
totalMemory: 11.17GiB freeMemory: 11.10GiB
2019-05-16 16:08:31.396886: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120] Creating TensorFlow
device (/device:GPU:0) -> (device: 0, name: Tesla K80, pci bus id: 2fd7:00:00.0, compute capability: 3.7)
2019-05-16 16:08:36.076962: I tensorflow/stream_executor/dso_loader.cc:139] successfully opened CUDA library
libcupti.so.8.0 locally
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/tensorflow/input_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/tensorflow/input_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/tensorflow/input_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/tensorflow/input_data/t10k-labels-idx1-ubyte.gz
Accuracy at step 0: 0.1081
Accuracy at step 10: 0.7457
Accuracy at step 20: 0.8233
Accuracy at step 30: 0.8644
Accuracy at step 40: 0.8848
Accuracy at step 50: 0.8889
Accuracy at step 60: 0.8898
Accuracy at step 70: 0.8979
Accuracy at step 80: 0.9087
Accuracy at step 90: 0.9099
Adding run metadata for 99
Accuracy at step 100: 0.9125
Accuracy at step 110: 0.9184
Accuracy at step 120: 0.922
Accuracy at step 130: 0.9161
Accuracy at step 140: 0.9219
Accuracy at step 150: 0.9151
Accuracy at step 160: 0.9199
Accuracy at step 170: 0.9305
Accuracy at step 180: 0.9251
Accuracy at step 190: 0.9258
Adding run metadata for 199
Accuracy at step 200: 0.9315
Accuracy at step 210: 0.9361
Accuracy at step 220: 0.9357
Accuracy at step 230: 0.9392
Accuracy at step 240: 0.9387
Accuracy at step 250: 0.9401
Accuracy at step 260: 0.9398
Accuracy at step 270: 0.9407
Accuracy at step 280: 0.9434
Accuracy at step 290: 0.9447
Adding run metadata for 299
Accuracy at step 300: 0.9463
Accuracy at step 310: 0.943
Accuracy at step 320: 0.9439
Accuracy at step 330: 0.943
Accuracy at step 340: 0.9457
Accuracy at step 350: 0.9497
Accuracy at step 360: 0.9481
Accuracy at step 370: 0.9466
Accuracy at step 380: 0.9514
Accuracy at step 390: 0.948
Adding run metadata for 399
Accuracy at step 400: 0.9469
Accuracy at step 410: 0.9489
Accuracy at step 420: 0.9529
```

```
Accuracy at step 430: 0.9507
Accuracy at step 440: 0.9504
Accuracy at step 450: 0.951
Accuracy at step 460: 0.9512
Accuracy at step 470: 0.9539
Accuracy at step 480: 0.9533
Accuracy at step 490: 0.9494
Adding run metadata for 499
```

## Clean up resources

To remove the associated Kubernetes objects created in this article, use the [kubectl delete job](#) command as follows:

```
kubectl delete jobs samples-tf-mnist-demo
```

## Next steps

To run Apache Spark jobs, see [Run Apache Spark jobs on AKS](#).

For more information about running machine learning (ML) workloads on Kubernetes, see [Kubeflow Labs](#).

# Connecting Azure Kubernetes Service and Azure Database for PostgreSQL - Single Server

12/3/2019 • 2 minutes to read • [Edit Online](#)

Azure Kubernetes Service (AKS) provides a managed Kubernetes cluster you can use in Azure. Below are some options to consider when using AKS and Azure Database for PostgreSQL together to create an application.

## Accelerated networking

Use accelerated networking-enabled underlying VMs in your AKS cluster. When accelerated networking is enabled on a VM, there is lower latency, reduced jitter, and decreased CPU utilization on the VM. Learn more about how accelerated networking works, the supported OS versions, and supported VM instances for [Linux](#).

From November 2018, AKS supports accelerated networking on those supported VM instances. Accelerated networking is enabled by default on new AKS clusters that use those VMs.

You can confirm whether your AKS cluster has accelerated networking:

1. Go to the Azure portal and select your AKS cluster.
2. Select the Properties tab.
3. Copy the name of the **Infrastructure Resource Group**.
4. Use the portal search bar to locate and open the infrastructure resource group.
5. Select a VM in that resource group.
6. Go to the VM's **Networking** tab.
7. Confirm whether **Accelerated networking** is 'Enabled.'

Or through the Azure CLI using the following two commands:

```
az aks show --resource-group myResourceGroup --name myAKSCluster --query "nodeResourceGroup"
```

The output will be the generated resource group that AKS creates containing the network interface. Take the "nodeResourceGroup" name and use it in the next command. **EnableAcceleratedNetworking** will either be true or false:

```
az network nic list --resource-group nodeResourceGroup -o table
```

## Open Service Broker for Azure

[Open Service Broker for Azure](#) (OSBA) lets you provision Azure services directly from Kubernetes or Cloud Foundry. It is an [Open Service Broker API](#) implementation for Azure.

With OSBA, you can create an Azure Database for PostgreSQL server and bind it to your AKS cluster using Kubernetes' native language. Learn about how to use OSBA and Azure Database for PostgreSQL together on the [OSBA GitHub page](#).

## Connection pooling

A connection pooler minimizes the cost and time associated with creating and closing new connections to the

database. The pool is a collection of connections that can be reused.

There are multiple connection poolers you can use with PostgreSQL. One of these is [PgBouncer](#). In the Microsoft Container Registry, we provide a lightweight containerized PgBouncer that can be used in a sidecar to pool connections from AKS to Azure Database for PostgreSQL. Visit the [docker hub page](#) to learn how to access and use this image.

## Next steps

- [Create an Azure Kubernetes Service cluster](#)

# Use Azure API Management with microservices deployed in Azure Kubernetes Service

12/11/2019 • 7 minutes to read • [Edit Online](#)

Microservices are perfect for building APIs. With [Azure Kubernetes Service](#) (AKS), you can quickly deploy and operate a [microservices-based architecture](#) in the cloud. You can then leverage [Azure API Management](#) (API Management) to publish your microservices as APIs for internal and external consumption. This article describes the options of deploying API Management with AKS. It assumes basic knowledge of Kubernetes, API Management, and Azure networking.

## Background

When publishing microservices as APIs for consumption, it can be challenging to manage the communication between the microservices and the clients that consume them. There is a multitude of cross-cutting concerns such as authentication, authorization, throttling, caching, transformation, and monitoring. These concerns are valid regardless of whether the microservices are exposed to internal or external clients.

The [API Gateway](#) pattern addresses these concerns. An API gateway serves as a front door to the microservices, decouples clients from your microservices, adds an additional layer of security, and decreases the complexity of your microservices by removing the burden of handling cross cutting concerns.

[Azure API Management](#) is a turnkey solution to solve your API gateway needs. You can quickly create a consistent and modern gateway for your microservices and publish them as APIs. As a full-lifecycle API management solution, it also provides additional capabilities including a self-service developer portal for API discovery, API lifecycle management, and API analytics.

When used together, AKS and API Management provide a platform for deploying, publishing, securing, monitoring, and managing your microservices-based APIs. In this article, we will go through a few options of deploying AKS in conjunction with API Management.

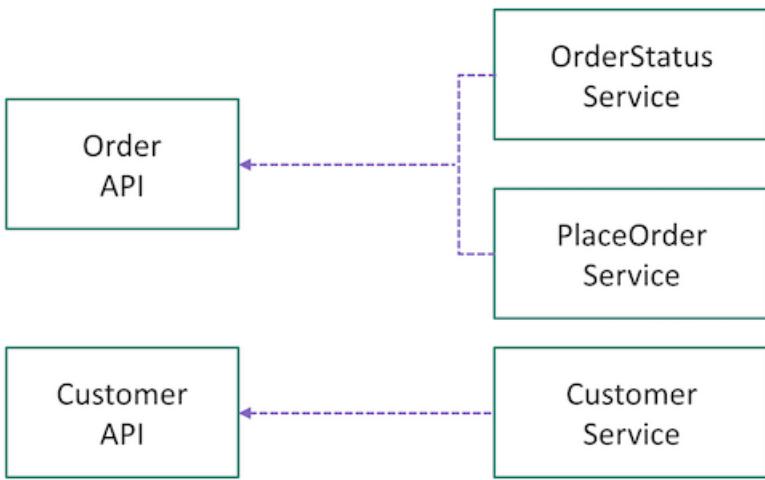
## Kubernetes Services and APIs

In a Kubernetes cluster, containers are deployed in [Pods](#), which are ephemeral and have a lifecycle. When a worker node dies, the Pods running on the node are lost. Therefore, the IP address of a Pod can change anytime. We cannot rely on it to communicate with the pod.

To solve this problem, Kubernetes introduced the concept of [Services](#). A Kubernetes Service is an abstraction layer which defines a logic group of Pods and enables external traffic exposure, load balancing and service discovery for those Pods.

When we are ready to publish our microservices as APIs through API Management, we need to think about how to map our Services in Kubernetes to APIs in API Management. There are no set rules. It depends on how you designed and partitioned your business capabilities or domains into microservices at the beginning. For instance, if the pods behind a Service is responsible for all operations on a given resource (e.g., Customer), the Service may be mapped to one API. If operations on a resource are partitioned into multiple microservices (e.g., GetOrder, PlaceOrder), then multiple Services may be logically aggregated into one single API in API management (See Fig. 1).

The mappings can also evolve. Since API Management creates a façade in front of the microservices, it allows us to refactor and right-size our microservices over time.



## Deploy API Management in front of AKS

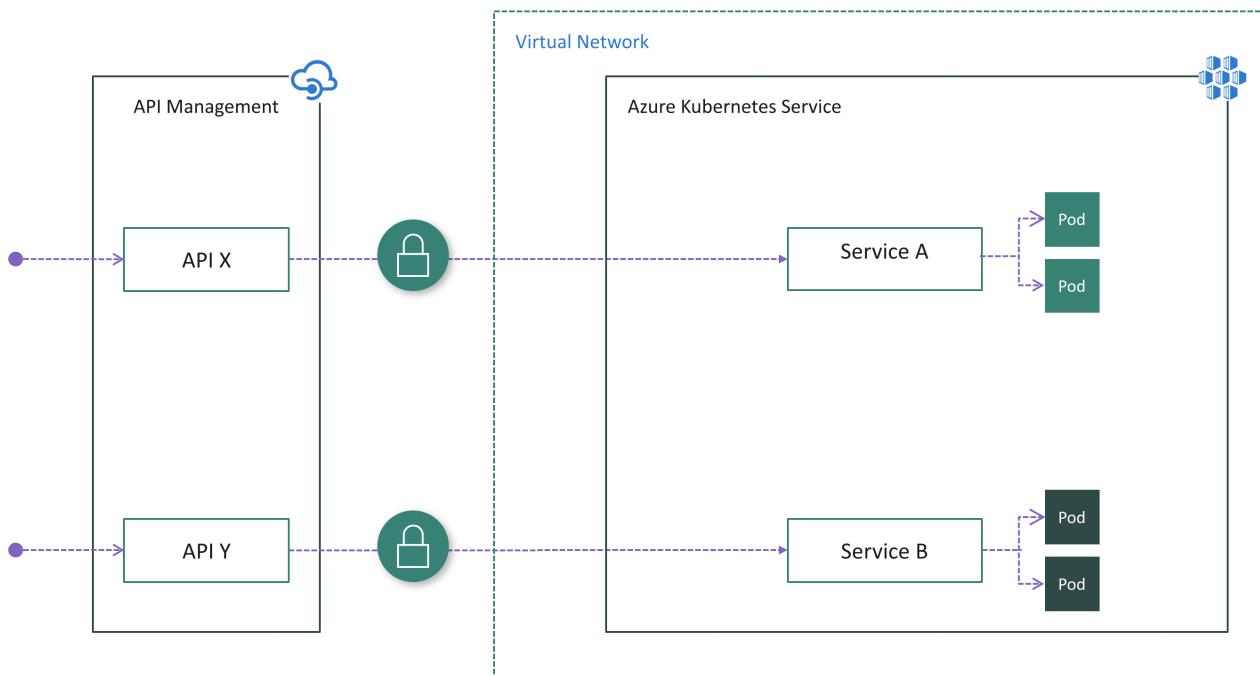
There are a few options of deploying API Management in front of an AKS cluster.

While an AKS cluster is always deployed in a virtual network (VNet), an API Management instance is not required to be deployed in a VNet. When API Management does not reside within the cluster VNet, the AKS cluster has to publish public endpoints for API Management to connect to. In that case, there is a need to secure the connection between API Management and AKS. In other words, we need to ensure the cluster can only be accessed exclusively through API Management. Let's go through the options.

### Option 1: Expose Services publicly

Services in an AKS cluster can be exposed publicly using [Service types](#) of NodePort, LoadBalancer, or ExternalName. In this case, Services are accessible directly from public internet. After deploying API Management in front of the cluster, we need to ensure all inbound traffic goes through API Management by applying authentication in the microservices. For instance, API Management can include an access token in each request made to the cluster. Each microservice is responsible for validating the token before processing the request.

This might be the easiest option to deploy API Management in front of AKS, especially if you already have authentication logic implemented in your microservices.



Pros:

- Easy configuration on the API Management side because it does not need to be injected into the cluster VNet
- No change on the AKS side if Services are already exposed publicly and authentication logic already exists in microservices

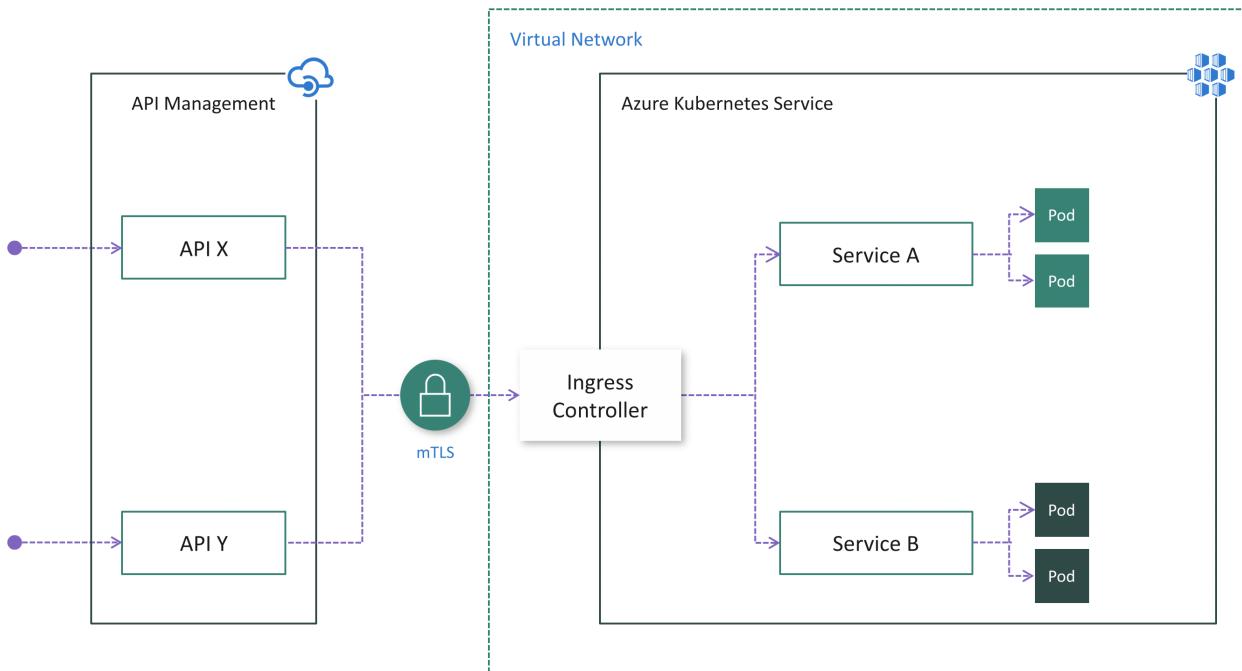
Cons:

- Potential security risk due to public visibility of Service endpoints
- No single-entry point for inbound cluster traffic
- Complicates microservices with duplicate authentication logic

## Option 2: Install an Ingress Controller

Although Option 1 might be easier, it has notable drawbacks as mentioned above. If an API Management instance does not reside in the cluster VNet, Mutual TLS authentication (mTLS) is a robust way of ensuring the traffic is secure and trusted in both directions between an API Management instance and an AKS cluster.

Mutual TLS authentication is [natively supported](#) by API Management and can be enabled in Kubernetes by [installing an Ingress Controller](#) (Fig. 3). As a result, authentication will be performed in the Ingress Controller, which simplifies the microservices. Additionally, you can add the IP addresses of API Management to the allowed list by Ingress to make sure only API Management has access to the cluster.



Pros:

- Easy configuration on the API Management side because it does not need to be injected into the cluster VNet and mTLS is natively supported
- Centralizes protection for inbound cluster traffic at the Ingress Controller layer
- Reduces security risk by minimizing publicly visible cluster endpoints

Cons:

- Increases complexity of cluster configuration due to extra work to install, configure and maintain the Ingress Controller and manage certificates used for mTLS
- Security risk due to public visibility of Ingress Controller endpoint(s)

When you publish APIs through API Management, it's easy and common to secure access to those APIs by using subscription keys. Developers who need to consume the published APIs must include a valid subscription key in HTTP requests when they make calls to those APIs. Otherwise, the calls are rejected immediately by the API Management gateway. They aren't forwarded to the back-end services.

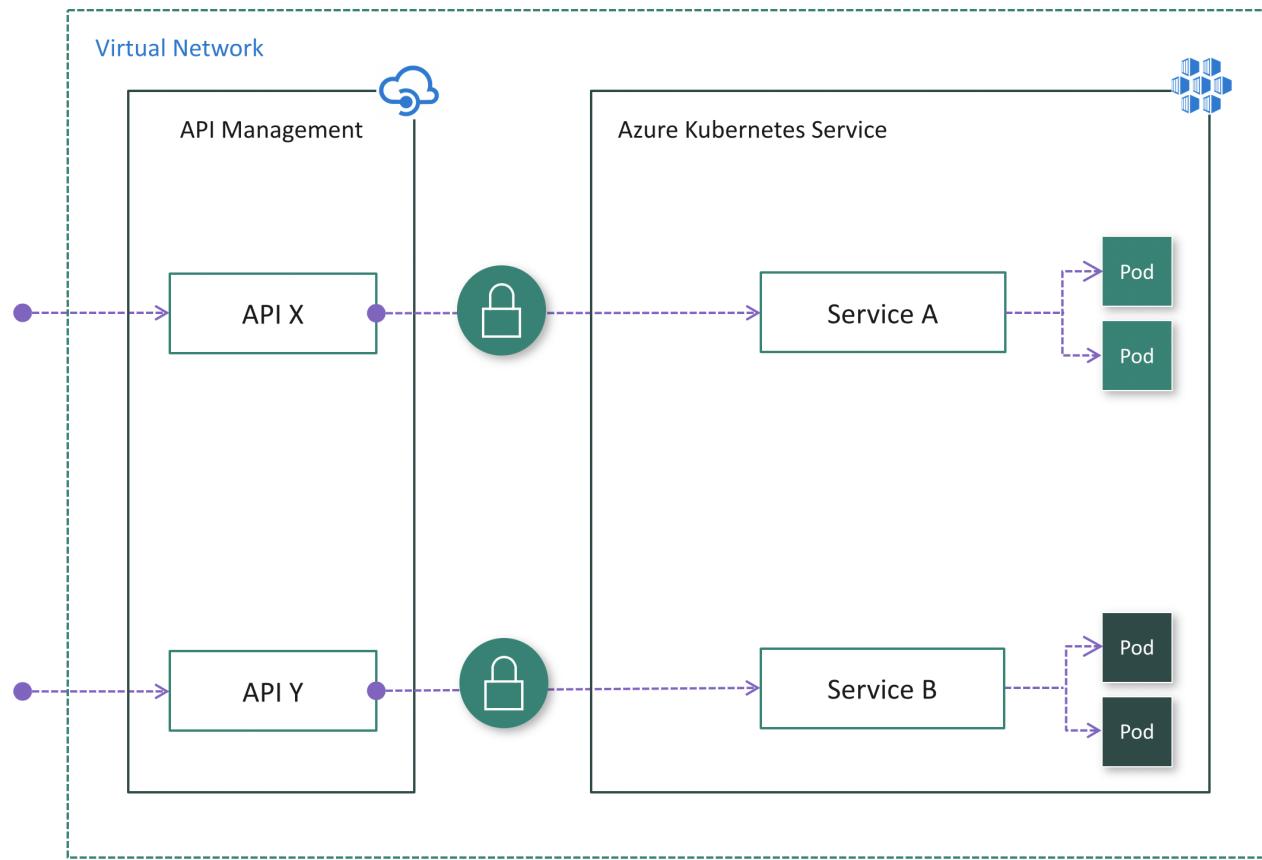
To get a subscription key for accessing APIs, a subscription is required. A subscription is essentially a named container for a pair of subscription keys. Developers who need to consume the published APIs can get subscriptions. And they don't need approval from API publishers. API publishers can also create subscriptions directly for API consumers.

### Option 3: Deploy APIM inside the cluster VNet

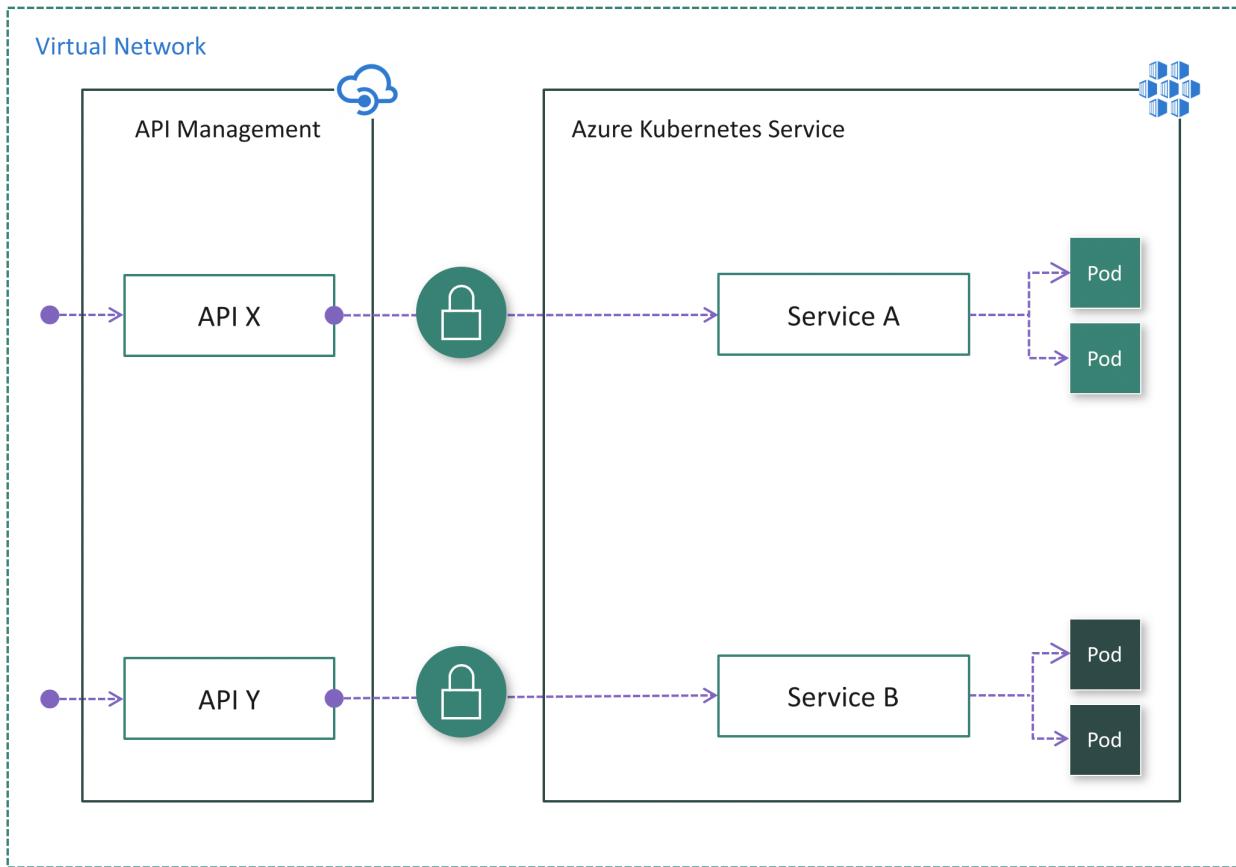
In some cases, customers with regulatory constraints or strict security requirements may find Option 1 and 2 not viable solutions due to publicly exposed endpoints. In others, the AKS cluster and the applications that consume the microservices might reside within the same VNet, hence there is no reason to expose the cluster publicly as all API traffic will remain within the VNet. For these scenarios, you can deploy API Management into the cluster VNet. [API Management Premium tier](#) supports VNet deployment.

There are two modes of [deploying API Management into a VNet](#) – External and Internal.

If API consumers do not reside in the cluster VNet, the External mode (Fig. 4) should be used. In this mode, the API Management gateway is injected into the cluster VNet but accessible from public internet via an external load balancer. It helps to hide the cluster completely while still allow external clients to consume the microservices. Additionally, you can use Azure networking capabilities such as Network Security Groups (NSG) to restrict network traffic.



If all API consumers reside within the cluster VNet, then the Internal mode (Fig. 5) could be used. In this mode, the API Management gateway is injected into the cluster VNET and accessible only from within this VNet via an internal load balancer. There is no way to reach the API Management gateway or the AKS cluster from public internet.



In both cases, the AKS cluster is not publicly visible. Compared to Option 2, the Ingress Controller may not be necessary. Depending on your scenario and configuration, authentication might still be required between API Management and your microservices. For instance, if a Service Mesh is adopted, it always requires mutual TLS authentication.

Pros:

- The most secure option because the AKS cluster has no public endpoint
- Simplifies cluster configuration since it has no public endpoint
- Ability to hide both API Management and AKS inside the VNet using the Internal mode
- Ability to control network traffic using Azure networking capabilities such as Network Security Groups (NSG)

Cons:

- Increases complexity of deploying and configuring API Management to work inside the VNet

## Next steps

- Learn more about [Network concepts for applications in AKS](#)
- Learn more about [How to use API Management with virtual networks](#)

# About service meshes

2/25/2020 • 4 minutes to read • [Edit Online](#)

A service mesh provides capabilities like traffic management, resiliency, policy, security, strong identity, and observability to your workloads. Your application is decoupled from these operational capabilities and the service mesh moves them out of the application layer, and down to the infrastructure layer.

## Scenarios

These are some of the scenarios that can be enabled for your workloads when you use a service mesh:

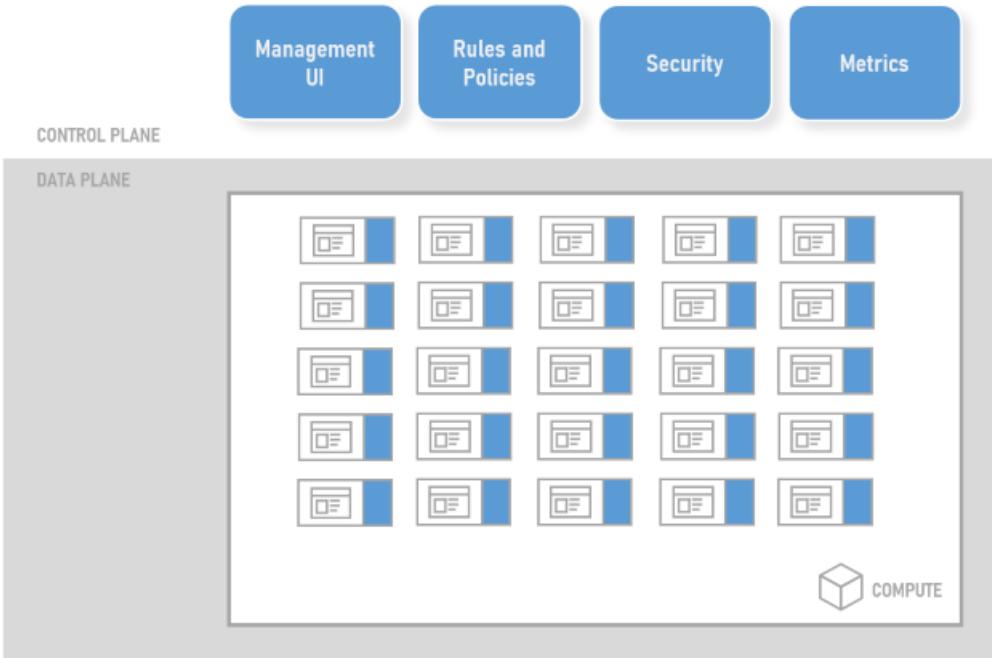
- **Encrypt all traffic in cluster** - Enable mutual TLS between specified services in the cluster. This can be extended to ingress and egress at the network perimeter. Provides a secure by default option with no changes needed for application code and infrastructure.
- **Canary and phased rollouts** - Specify conditions for a subset of traffic to be routed to a set of new services in the cluster. On successful test of canary release, remove conditional routing and phase gradually increasing % of all traffic to new service. Eventually all traffic will be directed to new service.
- **Traffic management and manipulation** - Create a policy on a service that will rate limit all traffic to a version of a service from a specific origin. Or a policy that applies a retry strategy to classes of failures between specified services. Mirror live traffic to new versions of services during a migration or to debug issues. Inject faults between services in a test environment to test resiliency.
- **Observability** - Gain insight into how your services are connected the traffic that flows between them. Obtain metrics, logs, and traces for all traffic in cluster, and ingress/egress. Add distributed tracing abilities to your applications.

## Architecture

A service mesh is typically composed of a control plane and the data plane.

The **control plane** has a number of components that support managing the service mesh. This will typically include a management interface which could be a UI or an API. There will also typically be components that manage the rule and policy definitions that define how the service mesh should implement specific capabilities. There are also components that manage aspects of security like strong identity and certificates for mTLS. Service meshes will also typically have a metrics or observability component that collects and aggregates metrics and telemetry from the workloads.

The **data plane** typically consists of a proxy that is transparently injected as a sidecar to your workloads. This proxy is configured to control all network traffic in and out of the pod containing your workload. This allows the proxy to be configured to secure traffic via mTLS, dynamically route traffic, apply policies to traffic and to collect metrics and tracing information.



## Capabilities

Each of the service meshes have a natural fit and focus on supporting specific scenarios, but you'll typically find that most will implement a number of, if not all, of the following capabilities.

### Traffic management

- **Protocol** – layer 7 (http, grpc)
- **Dynamic Routing** – conditional, weighting, mirroring
- **Resiliency** – timeouts, retries, circuit breakers
- **Policy** – access control, rate limits, quotas
- **Testing** - fault injection

### Security

- **Encryption** – mTLS, certificate management, external CA
- **Strong Identity** – SPIFFE or similar
- **Auth** – authentication, authorisation

### Observability

- **Metrics** – golden metrics, prometheus, grafana
- **Tracing** - traces across workloads
- **Traffic** – cluster, ingress/egress

### Mesh

- **Supported Compute** - Kubernetes, virtual machines
- **Multi-cluster** - gateways, federation

## Selection criteria

Before you select a service mesh, ensure that you understand your requirements and the reasons for installing a service mesh. Try asking the following questions.

- **Is an Ingress Controller sufficient for my needs?** - Sometimes having a capability like a/b testing or traffic splitting at the ingress is sufficient to support the required scenario. Don't add complexity to your

environment with no upside.

- **Can my workloads and environment tolerate the additional overheads?** - All the additional components required to support the service mesh require additional resources like cpu and memory. In addition, all the proxies and their associated policy checks add latency to your traffic. If you have workloads that are very sensitive to latency or cannot provide the additional resources to cover the service mesh components, then re-consider.
- **Is this adding additional complexity unnecessarily?** - If the reason for installing a service mesh is to gain a capability that is not necessarily critical to the business or operational teams, then consider whether the additional complexity of installation, maintenance, and configuration is worth it.
- **Can this be adopted in an incremental approach?** - Some of the service meshes that provide a lot of capabilities can be adopted in a more incremental approach. Install just the components you need to ensure your success. Once you are more confident and additional capabilities are required, then explore those. Resist the urge to install *everything* from the start.

If, after careful consideration, you decide that you need a service mesh to provide the capabilities required, then your next decision is *which service mesh?*

Consider the following areas and which of them are most aligned with your requirements. This will guide you towards the best fit for your environment and workloads. The [Next steps](#) section will take you to further detailed information about specific service meshes and how they map to these areas.

- **Technical** - traffic management, policy, security, observability
- **Business** - commercial support, foundation (CNCF), OSS license, governance
- **Operational** – installation/upgrades, resource requirements, performance requirements, integrations (metrics, telemetry, dashboards, tools, SMI), mixed workloads (Linux and Windows node pools), compute (Kubernetes, virtual machines), multi-cluster
- **Security** - auth, identity, certificate management and rotation, pluggable external CA

## Next steps

The following documentation provides more information about service meshes that you can try out on Azure Kubernetes Service (AKS):

[Learn more about Istio ...](#)

[Learn more about Linkerd ...](#)

[Learn more about Consul ...](#)

You may also want to explore Service Mesh Interface (SMI), a standard interface for service meshes on Kubernetes:

- [Service Mesh Interface \(SMI\)](#)

# Istio

2/25/2020 • 2 minutes to read • [Edit Online](#)

## Overview

Istio is a full featured, customisable, and extensible service mesh.

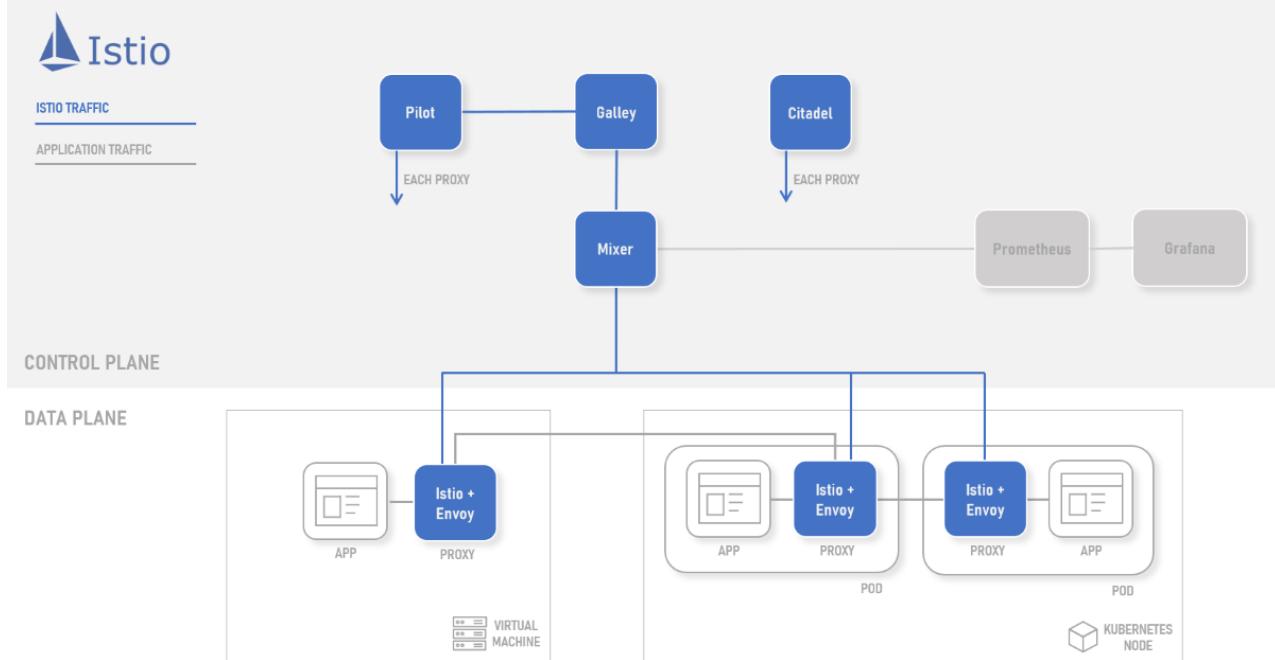
## Architecture

Istio provides a data plane that is composed of [Envoy](#)-based sidecars. These intelligent proxies control all network traffic in and out of your meshed apps and workloads.

The control plane manages the configuration, policy, and telemetry via the following [components](#):

- **Mixer** - Enforces access control and usage policies. Collects telemetry from the proxies that is pushed into [Prometheus](#).
- **Pilot** - Provides service discovery and traffic management policy/configuration for the proxies.
- **Citadel** - Provides identity and security capabilities that allow for mTLS between services.
- **Galley** - Abstracts and provides configuration to components.

The following architecture diagram demonstrates how the various components within the data plane and control plane interact.



## Selection criteria

It's important to understand and consider the following areas when evaluating Istio for your workloads:

- [Design Goals](#)
- [Capabilities](#)
- [Scenarios](#)

## Design goals

The following design goals [guide](#) the Istio project:

- **Maximize Transparency** - Allow adoption with the minimum amount of work to get real value from the system.
- **Extensibility** - Must be able to grow and adapt with changing needs.
- **Portability** - Run easily in different kinds of environments - cloud, on-premises.
- **Policy Uniformity** - Consistency in policy definition across variety of resources.

## Capabilities

Istio provides the following set of capabilities:

- **Mesh** – gateways (multi-cluster), virtual machines (mesh expansion)
- **Traffic Management** – routing, splitting, timeouts, circuit breakers, retries, ingress, egress
- **Policy** – access control, rate limit, quota, custom policy adapters
- **Security** – authentication (jwt), authorisation, encryption (mTLS), external CA (HashiCorp Vault)
- **Observability** – golden metrics, mirror, tracing, custom adapters, prometheus, grafana

## Scenarios

Istio is well suited to and suggested for the following scenarios:

- Require extensibility and rich set of capabilities
- Mesh expansion to include VM based workloads
- Multi-cluster service mesh

## Next steps

The following documentation describes how you can install Istio on Azure Kubernetes Service (AKS):

### [Install Istio in Azure Kubernetes Service \(AKS\)](#)

You can also further explore Istio concepts and additional deployment models:

- [Istio Concepts](#)
- [Istio Deployment Models](#)

# Install and use Istio in Azure Kubernetes Service (AKS)

2/25/2020 • 15 minutes to read • [Edit Online](#)

Istio is an open-source service mesh that provides a key set of functionality across the microservices in a Kubernetes cluster. These features include traffic management, service identity and security, policy enforcement, and observability. For more information about Istio, see the official [What is Istio?](#) documentation.

This article shows you how to install Istio. The Istio `istioctl` client binary is installed onto your client machine and the Istio components are installed into a Kubernetes cluster on AKS.

## NOTE

The following instructions reference Istio version `1.4.0`.

The Istio `1.4.x` releases have been tested by the Istio team against Kubernetes versions `1.13`, `1.14`, `1.15`. You can find additional Istio versions at [GitHub - Istio Releases](#), information about each of the releases at [Istio News](#) and supported Kubernetes versions at [Istio General FAQ](#).

In this article, you learn how to:

- Download and install the Istio `istioctl` client binary
- Install Istio on AKS
- Validate the Istio installation
- Access the add-ons
- Uninstall Istio from AKS

## Before you begin

The steps detailed in this article assume that you've created an AKS cluster (Kubernetes `1.13` and above, with RBAC enabled) and have established a `kubectl` connection with the cluster. If you need help with any of these items, then see the [AKS quickstart](#).

Make sure that you have read the [Istio Performance and Scalability](#) documentation to understand the additional resource requirements for running Istio in your AKS cluster. The core and memory requirements will vary based on your specific workload. Choose an appropriate number of nodes and VM size to cater for your setup.

This article separates the Istio installation guidance into several discrete steps. The end result is the same in structure as the official Istio installation [guidance](#).

## Download and install the Istio `istioctl` client binary

In a bash-based shell on Linux or [Windows Subsystem for Linux](#), use `curl` to download the Istio release and then extract with `tar` as follows:

```
# Specify the Istio version that will be leveraged throughout these instructions
ISTIO_VERSION=1.4.0

curl -sL "https://github.com/istio/istio/releases/download/$ISTIO_VERSION/istio-$ISTIO_VERSION-linux.tar.gz" |
tar xz
```

The `istioctl` client binary runs on your client machine and allows you to interact with the Istio service mesh. Use the following commands to install the Istio `istioctl` client binary in a bash-based shell on Linux or [Windows Subsystem for Linux](#). These commands copy the `istioctl` client binary to the standard user program location in your `PATH`.

```
cd istio-$ISTIO_VERSION
sudo cp ./bin/istioctl /usr/local/bin/istioctl
sudo chmod +x /usr/local/bin/istioctl
```

If you'd like command-line completion for the Istio `istioctl` client binary, then set it up as follows:

```
# Generate the bash completion file and source it in your current shell
mkdir -p ~/completions && istioctl collateral --bash -o ~/completions
source ~/completions/istioctl.bash

# Source the bash completion file in your .bashrc so that the command-line completions
# are permanently available in your shell
echo "source ~/completions/istioctl.bash" >> ~/.bashrc
```

## Download and install the Istio `istioctl` client binary

In a bash-based shell on MacOS, use `curl` to download the Istio release and then extract with `tar` as follows:

```
# Specify the Istio version that will be leveraged throughout these instructions
ISTIO_VERSION=1.4.0

curl -sL "https://github.com/istio/istio/releases/download/$ISTIO_VERSION/istio-$ISTIO_VERSION-osx.tar.gz" |
tar xz
```

The `istioctl` client binary runs on your client machine and allows you to interact with the Istio service mesh. Use the following commands to install the Istio `istioctl` client binary in a bash-based shell on MacOS. These commands copy the `istioctl` client binary to the standard user program location in your `PATH`.

```
cd istio-$ISTIO_VERSION
sudo cp ./bin/istioctl /usr/local/bin/istioctl
sudo chmod +x /usr/local/bin/istioctl
```

If you'd like command-line completion for the Istio `istioctl` client binary, then set it up as follows:

```
# Generate the bash completion file and source it in your current shell
mkdir -p ~/completions && istioctl collateral --bash -o ~/completions
source ~/completions/istioctl.bash

# Source the bash completion file in your .bashrc so that the command-line completions
# are permanently available in your shell
echo "source ~/completions/istioctl.bash" >> ~/.bashrc
```

## Download and install the Istio `istioctl` client binary

In a PowerShell-based shell on Windows, use `Invoke-WebRequest` to download the Istio release and then extract with `Expand-Archive` as follows:

```

# Specify the Istio version that will be leveraged throughout these instructions
$ISTIO_VERSION="1.4.0"

# Enforce TLS 1.2
[Net.ServicePointManager]::SecurityProtocol = "tls12"
$ProgressPreference = 'SilentlyContinue'; Invoke-WebRequest -URI
"https://github.com/istio/istio/releases/download/$ISTIO_VERSION/istio-$ISTIO_VERSION-win.zip" -OutFile
"istio-$ISTIO_VERSION.zip"
Expand-Archive -Path "istio-$ISTIO_VERSION.zip" -DestinationPath .

```

The `istioctl` client binary runs on your client machine and allows you to interact with the Istio service mesh. Use the following commands to install the Istio `istioctl` client binary in a PowerShell-based shell on Windows. These commands copy the `istioctl` client binary to an Istio folder and then make it available both immediately (in current shell) and permanently (across shell restarts) via your `PATH`. You don't need elevated (Admin) privileges to run these commands and you don't need to restart your shell.

```

# Copy istioctl.exe to C:\Istio
cd istio-$ISTIO_VERSION
New-Item -ItemType Directory -Force -Path "C:\Istio"
Copy-Item -Path .\bin\istioctl.exe -Destination "C:\Istio\"

# Add C:\Istio to PATH.
# Make the new PATH permanently available for the current User, and also immediately available in the current shell.
$PATH = [environment]::GetEnvironmentVariable("PATH", "User") + "; C:\Istio\
[environment]::SetEnvironmentVariable("PATH", $PATH, "User")
[environment]::SetEnvironmentVariable("PATH", $PATH)

```

## Install the Istio components on AKS

We'll be installing [Grafana](#) and [Kiali](#) as part of our Istio installation. Grafana provides analytics and monitoring dashboards, and Kiali provides a service mesh observability dashboard. In our setup, each of these components requires credentials that must be provided as a [Secret](#).

Before we can install the Istio components, we must create the secrets for both Grafana and Kiali. These secrets need to be installed into the `istio-system` namespace that will be used by Istio, so we'll need to create the namespace too. We need to use the `--save-config` option when creating the namespace via `kubectl create` so that the Istio installer can run `kubectl apply` on this object in the future.

```
kubectl create namespace istio-system --save-config
```

### Add Grafana Secret

Replace the `REPLACE_WITH_YOUR_SECURE_PASSWORD` token with your password and run the following commands:

```

GRAFANA_USERNAME=$(echo -n "grafana" | base64)
GRAFANA_PASSPHRASE=$(echo -n "REPLACE_WITH_YOUR_SECURE_PASSWORD" | base64)

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: grafana
  namespace: istio-system
  labels:
    app: grafana
type: Opaque
data:
  username: $GRAFANA_USERNAME
  passphrase: $GRAFANA_PASSPHRASE
EOF

```

## Add Kiali Secret

Replace the `REPLACE_WITH_YOUR_SECURE_PASSWORD` token with your password and run the following commands:

```

KIALI_USERNAME=$(echo -n "kiali" | base64)
KIALI_PASSPHRASE=$(echo -n "REPLACE_WITH_YOUR_SECURE_PASSWORD" | base64)

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: kiali
  namespace: istio-system
  labels:
    app: kiali
type: Opaque
data:
  username: $KIALI_USERNAME
  passphrase: $KIALI_PASSPHRASE
EOF

```

## Add Grafana Secret

Replace the `REPLACE_WITH_YOUR_SECURE_PASSWORD` token with your password and run the following commands:

```

GRAFANA_USERNAME=$(echo -n "grafana" | base64)
GRAFANA_PASSPHRASE=$(echo -n "REPLACE_WITH_YOUR_SECURE_PASSWORD" | base64)

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: grafana
  namespace: istio-system
  labels:
    app: grafana
type: Opaque
data:
  username: $GRAFANA_USERNAME
  passphrase: $GRAFANA_PASSPHRASE
EOF

```

## Add Kiali Secret

Replace the `REPLACE_WITH_YOUR_SECURE_PASSWORD` token with your password and run the following commands:

```

KIALI_USERNAME=$(echo -n "kiali" | base64)
KIALI_PASSPHRASE=$(echo -n "REPLACE_WITH_YOUR_SECURE_PASSWORD" | base64)

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: kiali
  namespace: istio-system
  labels:
    app: kiali
type: Opaque
data:
  username: $KIALI_USERNAME
  passphrase: $KIALI_PASSPHRASE
EOF

```

## Add Grafana Secret

Replace the `REPLACE_WITH_YOUR_SECURE_PASSWORD` token with your password and run the following commands:

```

$GRAFANA_USERNAME=[Convert]::ToString([System.Text.Encoding]::UTF8.GetBytes("grafana"))
$GRAFANA_PASSPHRASE=
[Convert]::ToString([System.Text.Encoding]::UTF8.GetBytes("REPLACE_WITH_YOUR_SECURE_PASSWORD"))

"apiVersion: v1
kind: Secret
metadata:
  name: grafana
  namespace: istio-system
  labels:
    app: grafana
type: Opaque
data:
  username: $GRAFANA_USERNAME
  passphrase: $GRAFANA_PASSPHRASE" | kubectl apply -f -

```

## Add Kiali Secret

Replace the `REPLACE_WITH_YOUR_SECURE_PASSWORD` token with your password and run the following commands:

```

$KIALI_USERNAME=[Convert]::ToString([System.Text.Encoding]::UTF8.GetBytes("kiali"))
$KIALI_PASSPHRASE=
[Convert]::ToString([System.Text.Encoding]::UTF8.GetBytes("REPLACE_WITH_YOUR_SECURE_PASSWORD"))

"apiVersion: v1
kind: Secret
metadata:
  name: kiali
  namespace: istio-system
  labels:
    app: kiali
type: Opaque
data:
  username: $KIALI_USERNAME
  passphrase: $KIALI_PASSPHRASE" | kubectl apply -f -

```

## Install Istio components

Now that we've successfully created the Grafana and Kiali secrets in our AKS cluster, it's time to install the Istio components.

The [Helm](#) installation approach for Istio will be deprecated in the future. The new installation approach for Istio

leverages the `istioctl` client binary, the [Istio configuration profiles](#), and the new [Istio control plane spec and api](#). This new approach is what we'll be using to install Istio.

#### NOTE

Istio currently must be scheduled to run on Linux nodes. If you have Windows Server nodes in your cluster, you must ensure that the Istio pods are only scheduled to run on Linux nodes. We'll use [node selectors](#) to make sure pods are scheduled to the correct nodes.

#### Caution

The [SDS \(secret discovery service\)](#) and [Istio CNI](#) Istio features are currently in [Alpha](#), so thought should be given before enabling these. In addition, the [Service Account Token Volume Projection](#) Kubernetes feature (a requirement for SDS) is not enabled in current AKS versions. Create a file called `istio.aks.yaml` with the following content. This file will hold the [Istio control plane spec](#) details for configuring Istio.

```
apiVersion: install.istio.io/v1alpha2
kind: IstioControlPlane
spec:
  # Use the default profile as the base
  # More details at: https://istio.io/docs/setup/additional-setup/config-profiles/
  profile: default
  values:
    global:
      # Ensure that the Istio pods are only scheduled to run on Linux nodes
      defaultNodeSelector:
        beta.kubernetes.io/os: linux
      # Enable mutual TLS for the control plane
      controlPlaneSecurityEnabled: true
      mtls:
        # Require all service to service communication to have mtls
        enabled: false
    grafana:
      # Enable Grafana deployment for analytics and monitoring dashboards
      enabled: true
      security:
        # Enable authentication for Grafana
        enabled: true
    kiali:
      # Enable the Kiali deployment for a service mesh observability dashboard
      enabled: true
    tracing:
      # Enable the Jaeger deployment for tracing
      enabled: true
```

Install istio using the `istioctl apply` command and the above `istio.aks.yaml` Istio control plane spec file as follows:

```
istioctl manifest apply -f istio.aks.yaml --logtostderr --set
installPackagePath=./install/kubernetes/operator/charts
```

The installer will deploy a number of [CRDs](#) and then manage dependencies to install all of the relevant objects defined for this configuration of Istio. You should see something like the following output snippet.

```
Applying manifests for these components:
- Tracing
- EgressGateway
- NodeAgent
- Grafana
- Policy
- Citadel
```

```
- CertManager
- IngressGateway
- Injector
- Prometheus
- PrometheusOperator
- Kiali
- Telemetry
- Galley
- Cni
- Pilot
- Base
- CoreDNS

NodeAgent is waiting on a prerequisite...
Telemetry is waiting on a prerequisite...
Galley is waiting on a prerequisite...
Cni is waiting on a prerequisite...
Grafana is waiting on a prerequisite...
Policy is waiting on a prerequisite...
Citadel is waiting on a prerequisite...
EgressGateway is waiting on a prerequisite...
Tracing is waiting on a prerequisite...
Kiali is waiting on a prerequisite...
PrometheusOperator is waiting on a prerequisite...
IngressGateway is waiting on a prerequisite...
Prometheus is waiting on a prerequisite...
CertManager is waiting on a prerequisite...
Injector is waiting on a prerequisite...
Pilot is waiting on a prerequisite...
Applying manifest for component Base
Waiting for CRDs to be applied.
CRDs applied.

Finished applying manifest for component Base
Prerequisite for Tracing has completed, proceeding with install.
Prerequisite for Injector has completed, proceeding with install.
Prerequisite for Telemetry has completed, proceeding with install.
Prerequisite for Policy has completed, proceeding with install.
Prerequisite for PrometheusOperator has completed, proceeding with install.
Prerequisite for NodeAgent has completed, proceeding with install.
Prerequisite for IngressGateway has completed, proceeding with install.
Prerequisite for Kiali has completed, proceeding with install.
Prerequisite for EgressGateway has completed, proceeding with install.
Prerequisite for Galley has completed, proceeding with install.
Prerequisite for Grafana has completed, proceeding with install.
Prerequisite for Cni has completed, proceeding with install.
Prerequisite for Citadel has completed, proceeding with install.

Applying manifest for component Tracing
Prerequisite for Prometheus has completed, proceeding with install.
Prerequisite for Pilot has completed, proceeding with install.
Prerequisite for CertManager has completed, proceeding with install.

Applying manifest for component Kiali
Applying manifest for component Prometheus
Applying manifest for component IngressGateway
Applying manifest for component Policy
Applying manifest for component Telemetry
Applying manifest for component Citadel
Applying manifest for component Galley
Applying manifest for component Pilot
Applying manifest for component Injector
Applying manifest for component Grafana
Finished applying manifest for component Kiali
Finished applying manifest for component Tracing
Finished applying manifest for component Prometheus
Finished applying manifest for component Citadel
Finished applying manifest for component Policy
Finished applying manifest for component IngressGateway
Finished applying manifest for component Injector
Finished applying manifest for component Galley
Finished applying manifest for component Pilot
Finished applying manifest for component Grafana
```

```
FINISHED APPLYING MANIFEST FOR COMPONENT Telemetry
Finished applying manifest for component Telemetry

Component IngressGateway installed successfully:
=====
serviceaccount/istio-ingressgateway-service-account created
deployment.apps/istio-ingressgateway created
gateway.networking.istio.io/ingressgateway created
sidecar.networking.istio.io/default created
poddisruptionbudget.policy/ingressgateway created
horizontalpodautoscaler.autoscaling/istio-ingressgateway created
service/istio-ingressgateway created

...
```

At this point, you've deployed Istio to your AKS cluster. To ensure that we have a successful deployment of Istio, let's move on to the next section to [Validate the Istio installation](#).

## Validate the Istio installation

First confirm that the expected services have been created. Use the `kubectl get svc` command to view the running services. Query the `istio-system` namespace, where the Istio and add-on components were installed by the `istio` Helm chart:

```
kubectl get svc --namespace istio-system --output wide
```

The following example output shows the services that should now be running:

- `istio-*` services
- `jaeger-*`, `tracing`, and `zipkin` add-on tracing services
- `prometheus` add-on metrics service
- `grafana` add-on analytics and monitoring dashboard service
- `kiali` add-on service mesh dashboard service

If the `istio-ingressgateway` shows an external ip of `<pending>`, wait a few minutes until an IP address has been assigned by Azure networking.

| NAME                                       | TYPE         | CLUSTER-IP   | EXTERNAL-IP  | PORT(S)                                |
|--|--------------|--------------|--|--|
| grafana                                    | ClusterIP    | 10.0.116.147 | <none>   | 3000/TCP                               |
| 92s app=grafana                            |              |              |  |  |
| istio-citadel                              | ClusterIP    | 10.0.248.152 | <none>   | 8060/TCP,15014/TCP                     |
| 94s app=citadel                            |              |              |  |  |
| istio-galley                               | ClusterIP    | 10.0.50.100  | <none>   | 443/TCP,15014/TCP,9901/TCP,15019/TCP   |
| 93s istio=galley                           |              |              |  |  |
| istio-ingressgateway                       | LoadBalancer | 10.0.36.213  | 20.188.221.111<br>15020:30369/TCP,80:31368/TCP,443:30045/TCP,15029:32011/TCP,15030:31212/TCP,15031:32411/TCP,15032:30009/TCP,15443:30010/TCP |  |
| 93s app=istio-ingressgateway               |              |              |  |  |
| istio-pilot                                | ClusterIP    | 10.0.23.222  | <none>   | 15010/TCP,15011/TCP,8080/TCP,15014/TCP |
| 93s istio=pilot                            |              |              |  |  |
| istio-policy                               | ClusterIP    | 10.0.59.250  | <none>   | 9091/TCP,15004/TCP,15014/TCP           |
| 93s istio-mixer-type=policy,istio=mixer    |              |              |  |  |
| istio-sidecar-injector                     | ClusterIP    | 10.0.123.219 | <none>   | 443/TCP                                |
| 93s istio=sidecar-injector                 |              |              |  |  |
| istio-telemetry                            | ClusterIP    | 10.0.216.9   | <none>   | 9091/TCP,15004/TCP,15014/TCP,42422/TCP |
| 89s istio-mixer-type=telemetry,istio=mixer |              |              |  |  |
| jaeger-agent                               | ClusterIP    | None         | <none>   | 5775/UDP,6831/UDP,6832/UDP             |
| 96s app=jaeger                             |              |              |  |  |
| jaeger-collector                           | ClusterIP    | 10.0.221.24  | <none>   | 14267/TCP,14268/TCP,14250/TCP          |
| 95s app=jaeger                             |              |              |  |  |
| jaeger-query                               | ClusterIP    | 10.0.46.154  | <none>   | 16686/TCP                              |
| 95s app=jaeger                             |              |              |  |  |
| kiali                                      | ClusterIP    | 10.0.174.97  | <none>   | 20001/TCP                              |
| 94s app=kiali                              |              |              |  |  |
| prometheus                                 | ClusterIP    | 10.0.245.226 | <none>   | 9090/TCP                               |
| 94s app=prometheus                         |              |              |  |  |
| tracing                                    | ClusterIP    | 10.0.249.95  | <none>   | 9411/TCP                               |
| 95s app=jaeger                             |              |              |  |  |
| zipkin                                     | ClusterIP    | 10.0.154.89  | <none>   | 9411/TCP                               |
| 94s app=jaeger                             |              |              |  |  |

Next, confirm that the required pods have been created. Use the `kubectl get pods` command, and again query the `istio-system` namespace:

```
kubectl get pods --namespace istio-system
```

The following example output shows the pods that are running:

- the `istio-*` pods
- the `prometheus-*` add-on metrics pod
- the `grafana-*` add-on analytics and monitoring dashboard pod
- the `kiali` add-on service mesh dashboard pod

| NAME                                    | READY | STATUS  | RESTARTS | AGE |
|---|-------|---------|----------|-----|
| grafana-6bc97ff99-k9sk4                 | 1/1   | Running | 0        | 92s |
| istio-citadel-6b5c754454-tb8nf          | 1/1   | Running | 0        | 94s |
| istio-galley-7d6d78d7c5-zshsd           | 2/2   | Running | 0        | 94s |
| istio-ingressgateway-85869c5cc7-x5d76   | 1/1   | Running | 0        | 95s |
| istio-pilot-787d6995b5-n5vrj            | 2/2   | Running | 0        | 94s |
| istio-policy-6cf4fbc8dc-sdsg5           | 2/2   | Running | 2        | 94s |
| istio-sidecar-injector-5d5b978668-wrz2s | 1/1   | Running | 0        | 94s |
| istio-telemetry-5498db684-6kdnw         | 2/2   | Running | 1        | 94s |
| istio-tracing-78548677bc-74tx6          | 1/1   | Running | 0        | 96s |
| kiali-59b7fd7f68-92zrh                  | 1/1   | Running | 0        | 95s |
| prometheus-7c7cf9dbd6-rjxcv             | 1/1   | Running | 0        | 94s |

All of the pods should show a status of `Running`. If your pods don't have these statuses, wait a minute or two until

they do. If any pods report an issue, use the `kubectl describe pod` command to review their output and status.

## Accessing the add-ons

A number of add-ons were installed by Istio in our setup above that provide additional functionality. The web applications for the add-ons are **not** exposed publicly via an external ip address.

To access the add-on user interfaces, use the `istioctl dashboard` command. This command leverages `kubectl port-forward` and a random port to create a secure connection between your client machine and the relevant pod in your AKS cluster. It will then automatically open the add-on web application in your default browser.

We added an additional layer of security for Grafana and Kiali by specifying credentials for them earlier in this article.

### Grafana

The analytics and monitoring dashboards for Istio are provided by [Grafana](#). Remember to use the credentials you created via the Grafana secret earlier when prompted. Open the Grafana dashboard securely as follows:

```
istioctl dashboard grafana
```

### Prometheus

Metrics for Istio are provided by [Prometheus](#). Open the Prometheus dashboard securely as follows:

```
istioctl dashboard prometheus
```

### Jaeger

Tracing within Istio is provided by [Jaeger](#). Open the Jaeger dashboard securely as follows:

```
istioctl dashboard jaeger
```

### Kiali

A service mesh observability dashboard is provided by [Kiali](#). Remember to use the credentials you created via the Kiali secret earlier when prompted. Open the Kiali dashboard securely as follows:

```
istioctl dashboard kiali
```

### Envoy

A simple interface to the [Envoy](#) proxies is available. It provides configuration information and metrics for an Envoy proxy running in a specified pod. Open the Envoy interface securely as follows:

```
istioctl dashboard envoy <pod-name>.<namespace>
```

## Uninstall Istio from AKS

### WARNING

Deleting Istio from a running system may result in traffic related issues between your services. Ensure that you have made provisions for your system to still operate correctly without Istio before proceeding.

## Remove Istio components and namespace

To remove Istio from your AKS cluster, use the `istioctl manifest generate` command with the `istio.aks.yaml` Istio control plane spec file. This will generate the deployed manifest, which we will pipe to `kubectl delete` in order to remove all the installed components and the `istio-system` namespace.

```
istioctl manifest generate -f istio.aks.yaml -o istio-components-aks --logtostderr --set  
installPackagePath=./install/kubernetes/operator/charts  
  
kubectl delete -f istio-components-aks -R
```

## Remove Istio CRDs and Secrets

The above commands delete all the Istio components and namespace, but we are still left with generated Istio secrets.

To delete the secrets, run the following command:

```
kubectl get secret --all-namespaces -o json | jq '.items[].metadata | ["kubectl delete secret -n", .namespace,  
.name] | join(" ")' -r | fgrep "istio." | xargs -t0 bash -c
```

To delete the secrets, run the following command:

```
kubectl get secret --all-namespaces -o json | jq '.items[].metadata | ["kubectl delete secret -n", .namespace,  
.name] | join(" ")' -r | fgrep "istio." | xargs -t0 bash -c
```

To delete the secrets, run the following command:

```
(kubectl get secret --all-namespaces -o json | ConvertFrom-Json).items.metadata |% { if ($_.name -match  
"istio.") { "Deleting {0}.{1}" -f $_.namespace, $_.name; kubectl delete secret -n $_.namespace $_.name } }
```

## Next steps

The following documentation describes how you can use Istio to provide intelligent routing to roll out a canary release:

### [AKS Istio intelligent routing scenario](#)

To explore more installation and configuration options for Istio, see the following official Istio guidance:

- [Istio - installation guides](#)

You can also follow additional scenarios using:

- [Istio Bookinfo Application example](#)

To learn how to monitor your AKS application using Application Insights and Istio, see the following Azure Monitor documentation:

- [Zero instrumentation application monitoring for Kubernetes hosted applications](#)

# Use intelligent routing and canary releases with Istio in Azure Kubernetes Service (AKS)

2/25/2020 • 15 minutes to read • [Edit Online](#)

Istio is an open-source service mesh that provides a key set of functionality across the microservices in a Kubernetes cluster. These features include traffic management, service identity and security, policy enforcement, and observability. For more information about Istio, see the official [What is Istio?](#) documentation.

This article shows you how to use the traffic management functionality of Istio. A sample AKS voting app is used to explore intelligent routing and canary releases.

In this article, you learn how to:

- Deploy the application
- Update the application
- Roll out a canary release of the application
- Finalize the rollout

## Before you begin

### NOTE

This scenario has been tested against Istio version [1.3.2](#).

The steps detailed in this article assume you've created an AKS cluster (Kubernetes [1.13](#) and above, with RBAC enabled) and have established a `kubectl` connection with the cluster. You'll also need Istio installed in your cluster.

If you need help with any of these items, then see the [AKS quickstart](#) and [Install Istio in AKS](#) guidance.

## About this application scenario

The sample AKS voting app provides two voting options (**Cats** or **Dogs**) to users. There is a storage component that persists the number of votes for each option. Additionally, there is an analytics component that provides details around the votes cast for each option.

In this application scenario, you start by deploying version [1.0](#) of the voting app and version [1.0](#) of the analytics component. The analytics component provides simple counts for the number of votes. The voting app and analytics component interact with version [1.0](#) of the storage component, which is backed by Redis.

You upgrade the analytics component to version [1.1](#), which provides counts, and now totals and percentages.

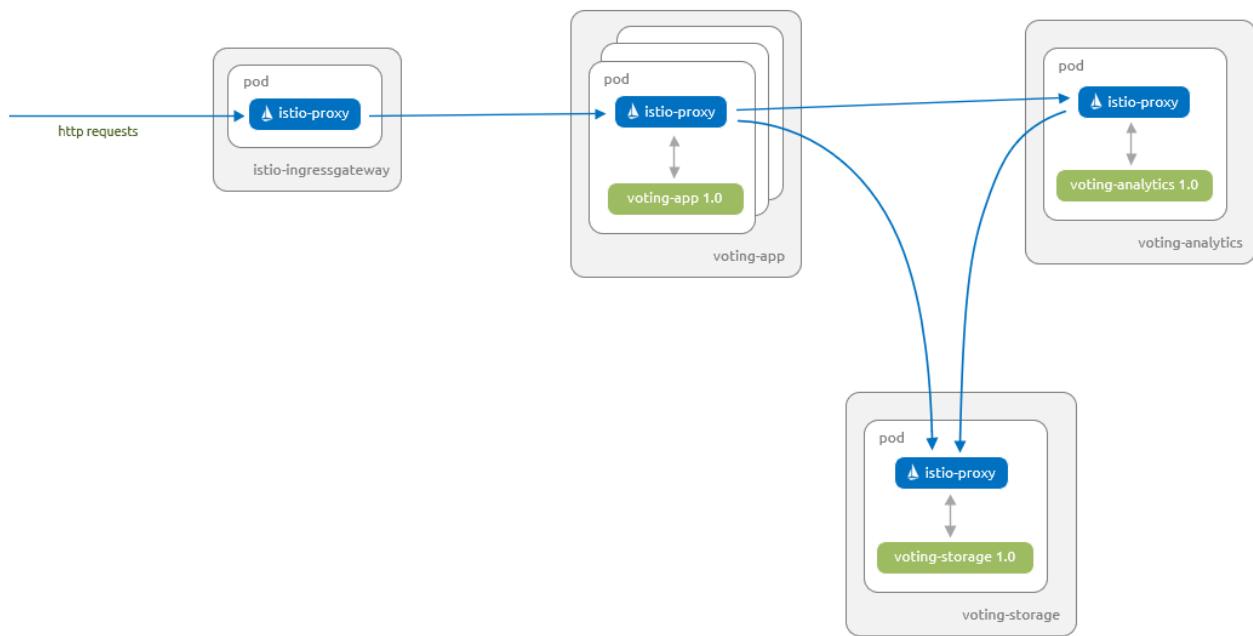
A subset of users test version [2.0](#) of the app via a canary release. This new version uses a storage component that is backed by a MySQL database.

Once you're confident that version [2.0](#) works as expected on your subset of users, you roll out version [2.0](#) to all your users.

## Deploy the application

Let's start by deploying the application into your Azure Kubernetes Service (AKS) cluster. The following diagram

shows what runs by the end of this section - version 1.0 of all components with inbound requests serviced via the Istio ingress gateway:



The artifacts you need to follow along with this article are available in the [Azure-Samples/aks-voting-app](https://github.com/Azure-Samples/aks-voting-app) GitHub repo. You can either download the artifacts or clone the repo as follows:

```
git clone https://github.com/Azure-Samples/aks-voting-app.git
```

Change to the following folder in the downloaded / cloned repo and run all subsequent steps from this folder:

```
cd aks-voting-app/scenarios/intelligent-routing-with-istio
```

First, create a namespace in your AKS cluster for the sample AKS voting app named `voting` as follows:

```
kubectl create namespace voting
```

Label the namespace with `istio-injection=enabled`. This label instructs Istio to automatically inject the istio-proxies as sidecars into all of your pods in this namespace.

```
kubectl label namespace voting istio-injection=enabled
```

Now let's create the components for the AKS Voting app. Create these components in the `voting` namespace created in a previous step.

```
kubectl apply -f kubernetes/step-1-create-voting-app.yaml --namespace voting
```

The following example output shows the resources being created:

```
deployment.apps/voting-storage-1-0 created
service/voting-storage created
deployment.apps/voting-analytics-1-0 created
service/voting-analytics created
deployment.apps/voting-app-1-0 created
service/voting-app created
```

#### NOTE

Istio has some specific requirements around pods and services. For more information, see the [Istio Requirements for Pods and Services documentation](#).

To see the pods that have been created, use the [kubectl get pods](#) command as follows:

```
kubectl get pods -n voting --show-labels
```

The following example output shows there are three instances of the `voting-app` pod and a single instance of both the `voting-analytics` and `voting-storage` pods. Each of the pods has two containers. One of these containers is the component, and the other is the `istio-proxy`:

| NAME                                  | READY | STATUS  | RESTARTS | AGE | LABELS  |
|---------------------------------------|-------|---------|----------|-----|---|
| voting-analytics-1-0-57c7fccb44-ng7dl | 2/2   | Running | 0        | 39s | app=voting-analytics,pod-template-hash=57c7fccb44,version=1.0 |
| voting-app-1-0-956756fd-d5w7z         | 2/2   | Running | 0        | 39s | app=voting-app,pod-template-hash=956756fd,version=1.0         |
| voting-app-1-0-956756fd-f6h69         | 2/2   | Running | 0        | 39s | app=voting-app,pod-template-hash=956756fd,version=1.0         |
| voting-app-1-0-956756fd-wsxvt         | 2/2   | Running | 0        | 39s | app=voting-app,pod-template-hash=956756fd,version=1.0         |
| voting-storage-1-0-5d8fcc89c4-2jhms   | 2/2   | Running | 0        | 39s | app=voting-storage,pod-template-hash=5d8fcc89c4,version=1.0   |

To see information about the pod, we'll use the [kubectl describe pod](#) command with label selectors to select the `voting-analytics` pod. We'll filter the output to show the details of the two containers present in the pod:

```
kubectl describe pod -l "app=voting-analytics, version=1.0" -n voting | egrep "istio-proxy|voting-analytics:" -A2
```

The `istio-proxy` container has automatically been injected by Istio to manage the network traffic to and from your components, as shown in the following example output:

```
voting-analytics:
  Container ID: docker://35efa1f31d95ca737ff2e2229ab8fe7d9f2f8a39ac11366008f31287be4cea4d
  Image: mcr.microsoft.com/aks/samples/voting/analytics:1.0
  --
  istio-proxy:
  Container ID: docker://1fa4eb43e8d4f375058c23cc062084f91c0863015e58eb377276b20c809d43c6
  Image: docker.io/istio/proxyv2:1.3.2
```

```
kubectl describe pod -l "app=voting-analytics, version=1.0" -n voting | egrep "istio-proxy|voting-analytics:" -A2
```

The `istio-proxy` container has automatically been injected by Istio to manage the network traffic to and from your

components, as shown in the following example output:

```
voting-analytics:  
  Container ID: docker://35efa1f31d95ca737ff2e2229ab8fe7d9f2f8a39ac11366008f31287be4cea4d  
  Image: mcr.microsoft.com/aks/samples/voting/analytics:1.0  
--  
istio-proxy:  
  Container ID: docker://1fa4eb43e8d4f375058c23cc062084f91c0863015e58eb377276b20c809d43c6  
  Image: docker.io/istio/proxyv2:1.3.2
```

```
kubectl describe pod -l "app=voting-analytics, version=1.0" -n voting | Select-String -Pattern "istio-proxy:|voting-analytics:" -Context 0,2
```

The `istio-proxy` container has automatically been injected by Istio to manage the network traffic to and from your components, as shown in the following example output:

```
> voting-analytics:  
  Container ID: docker://35efa1f31d95ca737ff2e2229ab8fe7d9f2f8a39ac11366008f31287be4cea4d  
  Image: mcr.microsoft.com/aks/samples/voting/analytics:1.0  
> istio-proxy:  
  Container ID: docker://1fa4eb43e8d4f375058c23cc062084f91c0863015e58eb377276b20c809d43c6  
  Image: docker.io/istio/proxyv2:1.3.2
```

You can't connect to the voting app until you create the Istio [Gateway](#) and [Virtual Service](#). These Istio resources route traffic from the default Istio ingress gateway to our application.

#### NOTE

A **Gateway** is a component at the edge of the service mesh that receives inbound or outbound HTTP and TCP traffic.

A **Virtual Service** defines a set of routing rules for one or more destination services.

Use the `kubectl apply` command to deploy the Gateway and Virtual Service yaml. Remember to specify the namespace that these resources are deployed into.

```
kubectl apply -f istio/step-1-create-voting-app-gateway.yaml --namespace voting
```

The following example output shows the new Gateway and Virtual Service being created:

```
virtualservice.networking.istio.io/voting-app created  
gateway.networking.istio.io/voting-app-gateway created
```

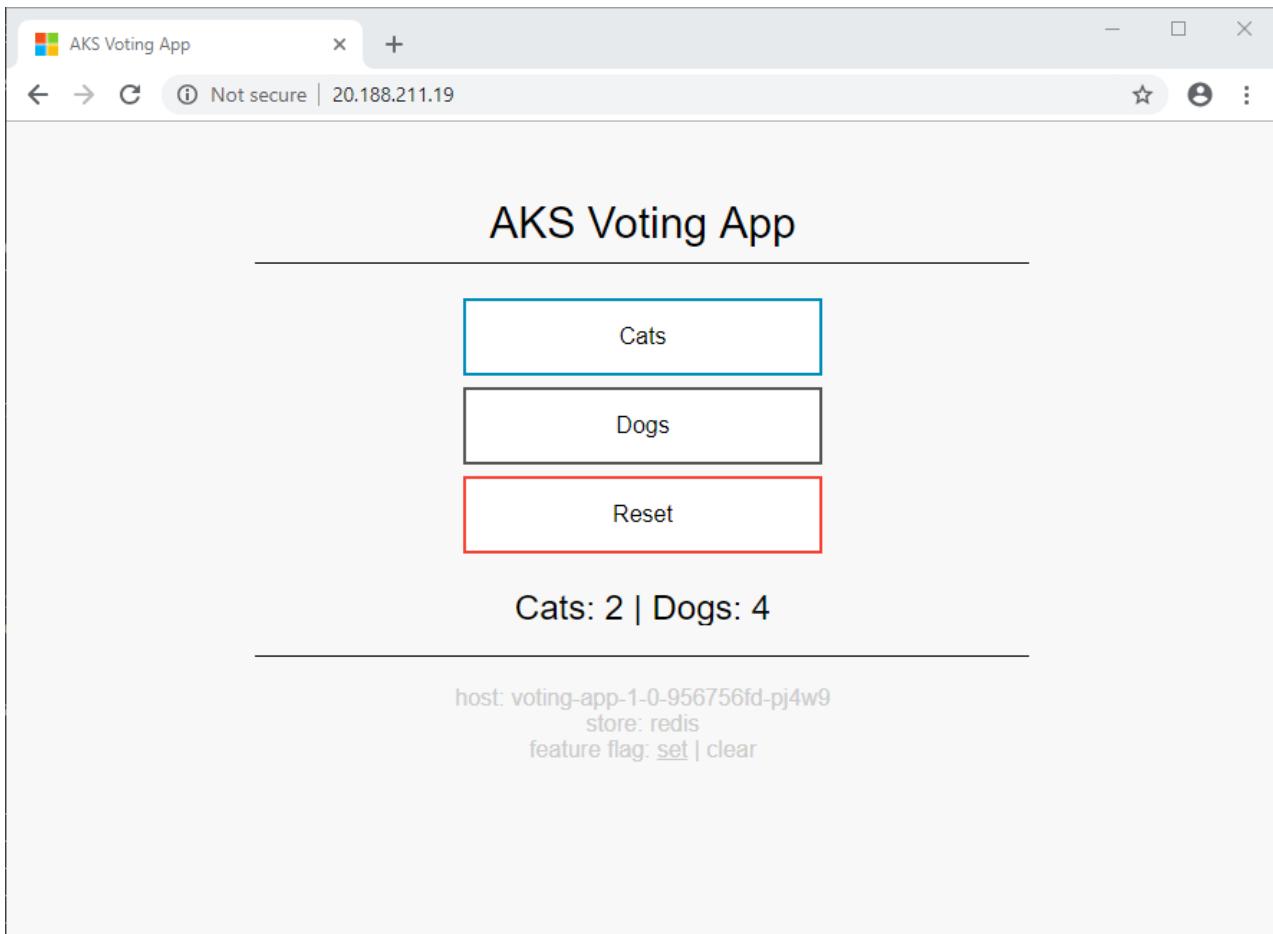
Obtain the IP address of the Istio Ingress Gateway using the following command:

```
kubectl get service istio-ingressgateway --namespace istio-system -o  
jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

The following example output shows the IP address of the Ingress Gateway:

```
20.188.211.19
```

Open up a browser and paste in the IP address. The sample AKS voting app is displayed.

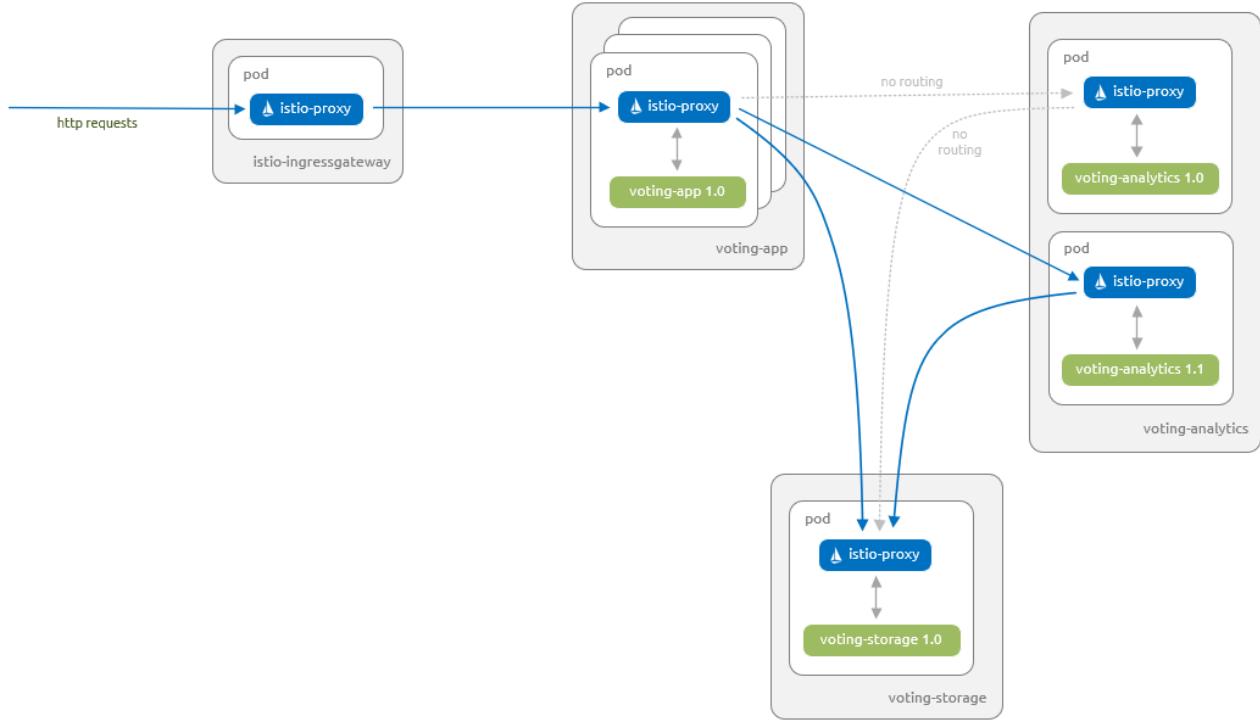


The information at the bottom of the screen shows that the app uses version `1.0` of `voting-app` and version `1.0` of `voting-storage` (Redis).

## Update the application

Let's deploy a new version of the analytics component. This new version `1.1` displays totals and percentages in addition to the count for each category.

The following diagram shows what will be running at the end of this section - only version `1.1` of our `voting-analytics` component has traffic routed from the `voting-app` component. Even though version `1.0` of our `voting-analytics` component continues to run and is referenced by the `voting-analytics` service, the Istio proxies disallow traffic to and from it.



Let's deploy version `1.1` of the `voting-analytics` component. Create this component in the `voting` namespace:

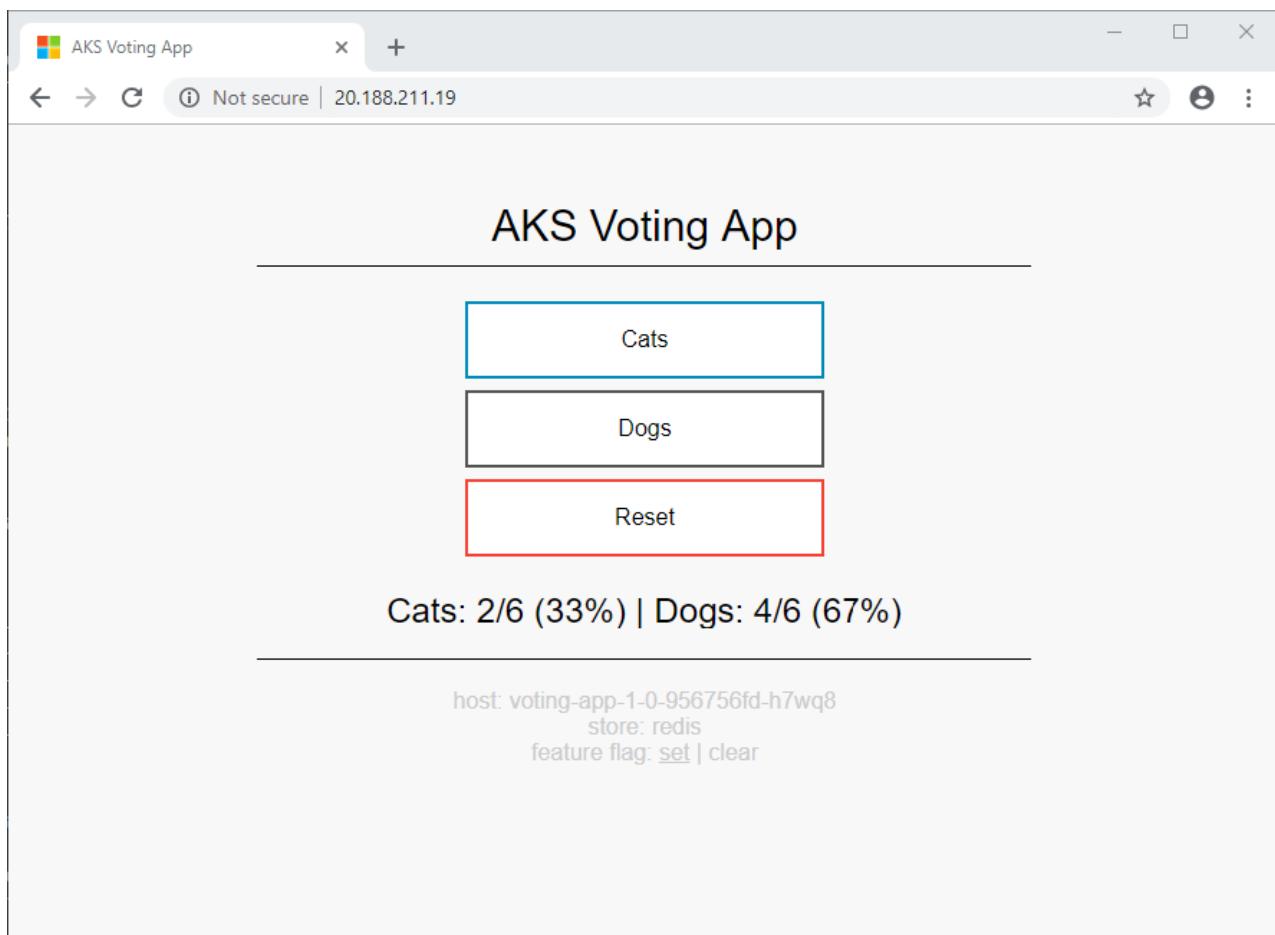
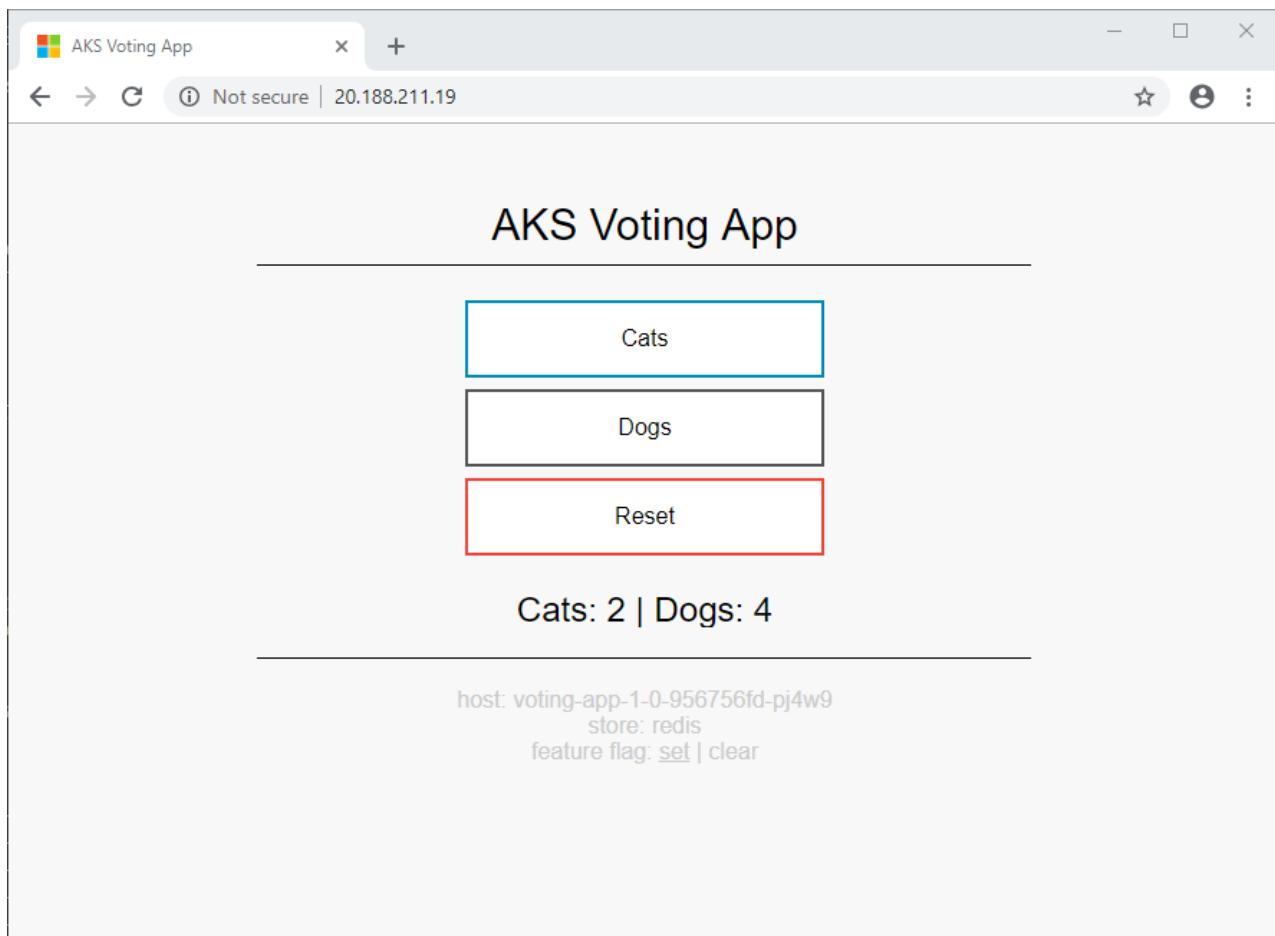
```
kubectl apply -f kubernetes/step-2-update-voting-analytics-to-1.1.yaml --namespace voting
```

The following example output shows the resources being created:

```
deployment.apps/voting-analytics-1-1 created
```

Open the sample AKS voting app in a browser again, using the IP address of the Istio Ingress Gateway obtained in the previous step.

Your browser alternates between the two views shown below. Since you are using a Kubernetes [Service](#) for the `voting-analytics` component with only a single label selector (`app: voting-analytics`), Kubernetes uses the default behavior of round-robin between the pods that match that selector. In this case, it is both version `1.0` and `1.1` of your `voting-analytics` pods.



You can visualize the switching between the two versions of the `voting-analytics` component as follows.  
Remember to use the IP address of your own Istio Ingress Gateway.

```
INGRESS_IP=20.188.211.19
for i in {1..5}; do curl -si $INGRESS_IP | grep results; done
```

```
INGRESS_IP=20.188.211.19
for i in {1..5}; do curl -si $INGRESS_IP | grep results; done
```

```
$INGRESS_IP="20.188.211.19"
(1..5) |% { (Invoke-WebRequest -Uri $INGRESS_IP).Content.Split("`n") | Select-String -Pattern "results" }
```

The following example output shows the relevant part of the returned web site as the site switches between versions:

```
<div id="results"> Cats: 2 | Dogs: 4 </div>
<div id="results"> Cats: 2 | Dogs: 4 </div>
<div id="results"> Cats: 2/6 (33%) | Dogs: 4/6 (67%) </div>
<div id="results"> Cats: 2 | Dogs: 4 </div>
<div id="results"> Cats: 2/6 (33%) | Dogs: 4/6 (67%) </div>
```

### Lock down traffic to version 1.1 of the application

Now let's lock down traffic to only version `1.1` of the `voting-analytics` component and to version `1.0` of the `voting-storage` component. You then define routing rules for all of the other components.

- A **Virtual Service** defines a set of routing rules for one or more destination services.
- A **Destination Rule** defines traffic policies and version specific policies.
- A **Policy** defines what authentication methods can be accepted on workload(s).

Use the `kubectl apply` command to replace the Virtual Service definition on your `voting-app` and add `Destination Rules` and `Virtual Services` for the other components. You will add a `Policy` to the `voting` namespace to ensure that all communication between services is secured using mutual TLS and client certificates.

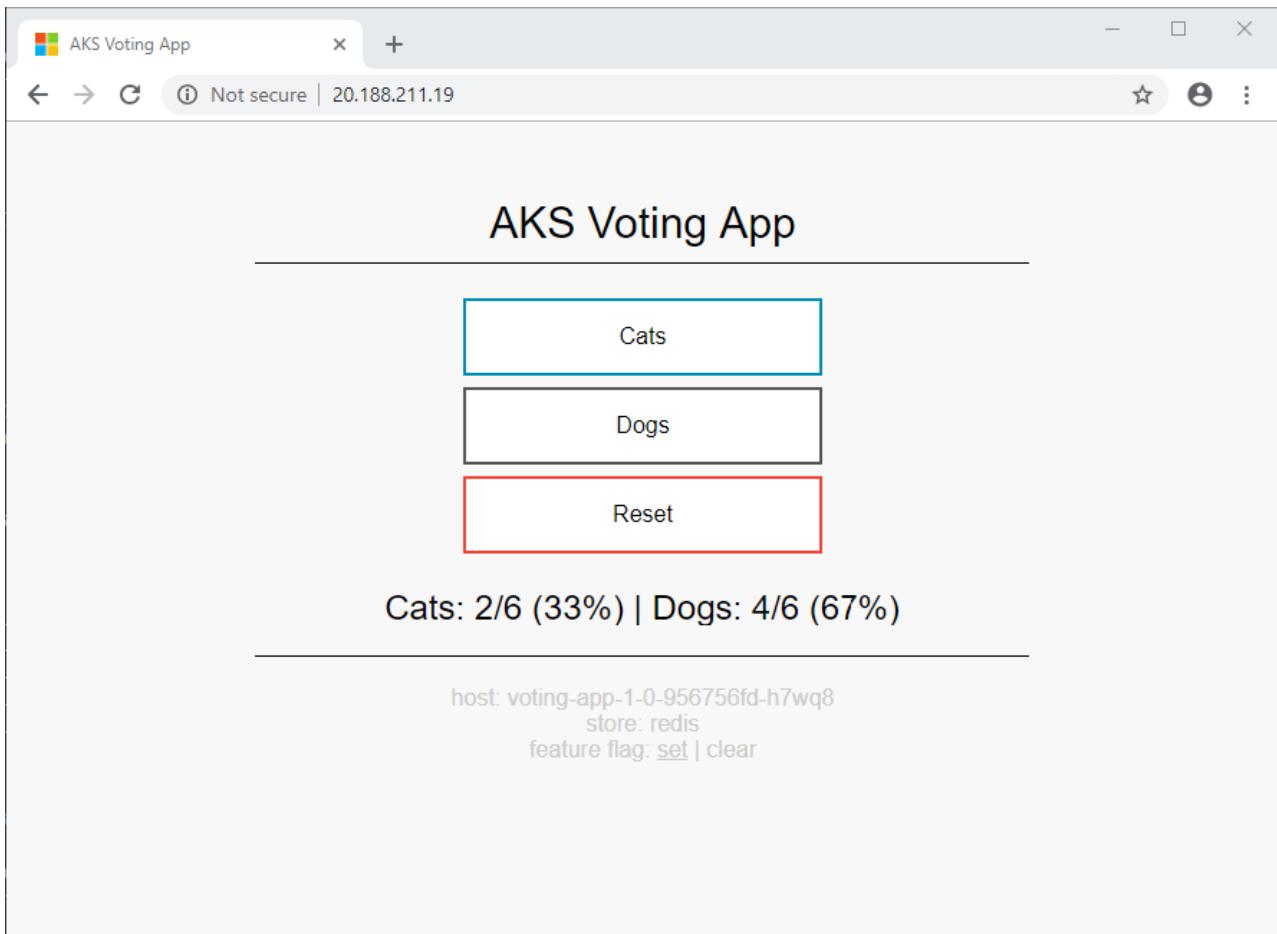
- The Policy has `peers.mtls.mode` set to `STRICT` to ensure that mutual TLS is enforced between your services within the `voting` namespace.
- We also set the `trafficPolicy.tls.mode` to `ISTIO_MUTUAL` in all our Destination Rules. Istio provides services with strong identities and secures communications between services using mutual TLS and client certificates that Istio transparently manages.

```
kubectl apply -f istio/step-2-update-and-add-routing-for-all-components.yaml --namespace voting
```

The following example output shows the new Policy, Destination Rules, and Virtual Services being updated/created:

```
virtualservice.networking.istio.io/voting-app configured
policy.authentication.istio.io/default created
destinationrule.networking.istio.io/voting-app created
destinationrule.networking.istio.io/voting-analytics created
virtualservice.networking.istio.io/voting-analytics created
destinationrule.networking.istio.io/voting-storage created
virtualservice.networking.istio.io/voting-storage created
```

If you open the AKS Voting app in a browser again, only the new version `1.1` of the `voting-analytics` component is used by the `voting-app` component.



You can visualize that you are now only routed to version `1.1` of your `voting-analytics` component as follows. Remember to use the IP address of your own Istio Ingress Gateway:

```
INGRESS_IP=20.188.211.19
for i in {1..5}; do curl -si $INGRESS_IP | grep results; done
```

```
INGRESS_IP=20.188.211.19
for i in {1..5}; do curl -si $INGRESS_IP | grep results; done
```

```
$INGRESS_IP="20.188.211.19"
(1..5) |% { (Invoke-WebRequest -Uri $INGRESS_IP).Content.Split("`n") | Select-String -Pattern "results" }
```

The following example output shows the relevant part of the returned web site:

```
<div id="results"> Cats: 2/6 (33%) | Dogs: 4/6 (67%) </div>
<div id="results"> Cats: 2/6 (33%) | Dogs: 4/6 (67%) </div>
<div id="results"> Cats: 2/6 (33%) | Dogs: 4/6 (67%) </div>
<div id="results"> Cats: 2/6 (33%) | Dogs: 4/6 (67%) </div>
<div id="results"> Cats: 2/6 (33%) | Dogs: 4/6 (67%) </div>
```

Let's now confirm that Istio is using mutual TLS to secure communications between each of our services. For this we will use the `authn tls-check` command on the `istioctl` client binary, which takes the following form.

```
istioctl authn tls-check <pod-name[.namespace> [<service>]
```

This set of commands provide information about the access to the specified services, from all pods that are in a namespace and match a set of labels:

```

# mTLS configuration between each of the istio ingress pods and the voting-app service
kubectl get pod -n istio-system -l app=istio-ingressgateway | grep Running | cut -d ' ' -f1 | xargs -n1 -I{} istioctl authn tls-check {}.istio-system voting-app.voting.svc.cluster.local

# mTLS configuration between each of the voting-app pods and the voting-analytics service
kubectl get pod -n voting -l app=voting-app | grep Running | cut -d ' ' -f1 | xargs -n1 -I{} istioctl authn tls-check {}.voting voting-analytics.voting.svc.cluster.local

# mTLS configuration between each of the voting-app pods and the voting-storage service
kubectl get pod -n voting -l app=voting-app | grep Running | cut -d ' ' -f1 | xargs -n1 -I{} istioctl authn tls-check {}.voting voting-storage.voting.svc.cluster.local

# mTLS configuration between each of the voting-analytics version 1.1 pods and the voting-storage service
kubectl get pod -n voting -l app=voting-analytics,version=1.1 | grep Running | cut -d ' ' -f1 | xargs -n1 -I{} istioctl authn tls-check {}.voting voting-storage.voting.svc.cluster.local

```

```

# mTLS configuration between each of the istio ingress pods and the voting-app service
kubectl get pod -n istio-system -l app=istio-ingressgateway | grep Running | cut -d ' ' -f1 | xargs -n1 -I{} istioctl authn tls-check {}.istio-system voting-app.voting.svc.cluster.local

# mTLS configuration between each of the voting-app pods and the voting-analytics service
kubectl get pod -n voting -l app=voting-app | grep Running | cut -d ' ' -f1 | xargs -n1 -I{} istioctl authn tls-check {}.voting voting-analytics.voting.svc.cluster.local

# mTLS configuration between each of the voting-app pods and the voting-storage service
kubectl get pod -n voting -l app=voting-app | grep Running | cut -d ' ' -f1 | xargs -n1 -I{} istioctl authn tls-check {}.voting voting-storage.voting.svc.cluster.local

# mTLS configuration between each of the voting-analytics version 1.1 pods and the voting-storage service
kubectl get pod -n voting -l app=voting-analytics,version=1.1 | grep Running | cut -d ' ' -f1 | xargs -n1 -I{} istioctl authn tls-check {}.voting voting-storage.voting.svc.cluster.local

```

```

# mTLS configuration between each of the istio ingress pods and the voting-app service
(kubectl get pod -n istio-system -l app=istio-ingressgateway | Select-String -Pattern "Running").Line |% { $_.Split()[0] |% { istioctl authn tls-check $($_.Split()[0]) voting-app.voting.svc.cluster.local } }

# mTLS configuration between each of the voting-app pods and the voting-analytics service
(kubectl get pod -n voting -l app=voting-app | Select-String -Pattern "Running").Line |% { $_.Split()[0] |% { istioctl authn tls-check $($_.Split()[0]) voting-analytics.voting.svc.cluster.local } }

# mTLS configuration between each of the voting-app pods and the voting-storage service
(kubectl get pod -n voting -l app=voting-app | Select-String -Pattern "Running").Line |% { $_.Split()[0] |% { istioctl authn tls-check $($_.Split()[0]) voting-storage.voting.svc.cluster.local } }

# mTLS configuration between each of the voting-analytics version 1.1 pods and the voting-storage service
(kubectl get pod -n voting -l app=voting-analytics,version=1.1 | Select-String -Pattern "Running").Line |% { $_.Split()[0] |% { istioctl authn tls-check $($_.Split()[0]) voting-storage.voting.svc.cluster.local } }

```

This following example output shows that mutual TLS is enforced for each of our queries above. The output also shows the Policy and Destination Rules that enforces the mutual TLS:

```

# mTLS configuration between istio ingress pods and the voting-app service
HOST:PORT STATUS SERVER CLIENT AUTHN POLICY DESTINATION
RULE
voting-app.voting.svc.cluster.local:8080 OK mTLS mTLS default/voting voting-
app/voting

# mTLS configuration between each of the voting-app pods and the voting-analytics service
HOST:PORT STATUS SERVER CLIENT AUTHN POLICY
DESTINATION RULE
voting-analytics.voting.svc.cluster.local:8080 OK mTLS mTLS default/voting voting-
analytics/voting
HOST:PORT STATUS SERVER CLIENT AUTHN POLICY
DESTINATION RULE
voting-analytics.voting.svc.cluster.local:8080 OK mTLS mTLS default/voting voting-
analytics/voting
HOST:PORT STATUS SERVER CLIENT AUTHN POLICY
DESTINATION RULE
voting-analytics.voting.svc.cluster.local:8080 OK mTLS mTLS default/voting voting-
analytics/voting

# mTLS configuration between each of the voting-app pods and the voting-storage service
HOST:PORT STATUS SERVER CLIENT AUTHN POLICY
DESTINATION RULE
voting-storage.voting.svc.cluster.local:6379 OK mTLS mTLS default/voting voting-
storage/voting
HOST:PORT STATUS SERVER CLIENT AUTHN POLICY
DESTINATION RULE
voting-storage.voting.svc.cluster.local:6379 OK mTLS mTLS default/voting voting-
storage/voting
HOST:PORT STATUS SERVER CLIENT AUTHN POLICY
DESTINATION RULE
voting-storage.voting.svc.cluster.local:6379 OK mTLS mTLS default/voting voting-
storage/voting

# mTLS configuration between each of the voting-analytics version 1.1 pods and the voting-storage service
HOST:PORT STATUS SERVER CLIENT AUTHN POLICY
DESTINATION RULE
voting-storage.voting.svc.cluster.local:6379 OK mTLS mTLS default/voting voting-
storage/voting

```

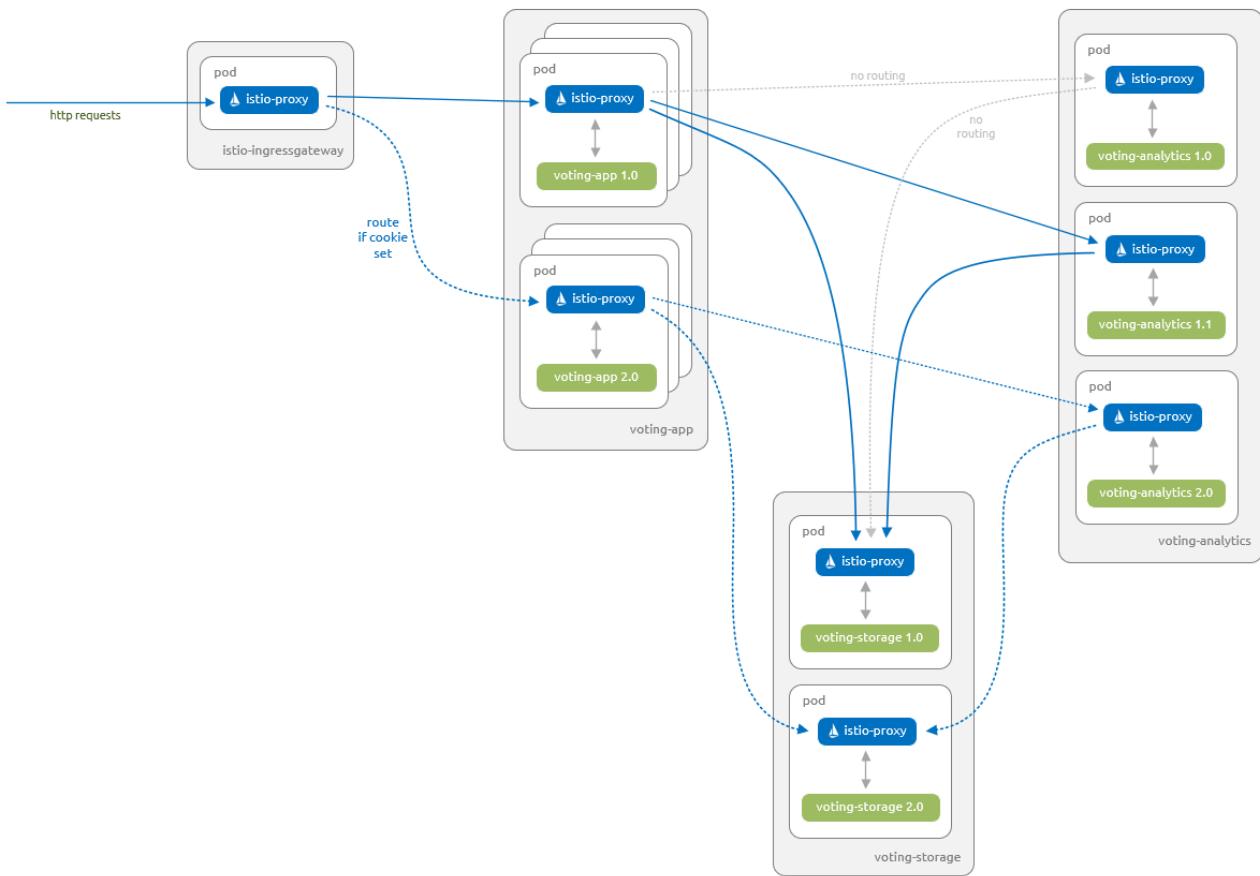
## Roll out a canary release of the application

Now let's deploy a new version 2.0 of the voting-app, voting-analytics, and voting-storage components. The new voting-storage component uses MySQL instead of Redis, and the voting-app and voting-analytics components are updated to allow them to use this new voting-storage component.

The voting-app component now supports feature flag functionality. This feature flag allows you to test the canary release capability of Istio for a subset of users.

The following diagram shows what you will have running at the end of this section.

- Version 1.0 of the voting-app component, version 1.1 of the voting-analytics component and version 1.0 of the voting-storage component are able to communicate with each other.
- Version 2.0 of the voting-app component, version 2.0 of the voting-analytics component and version 2.0 of the voting-storage component are able to communicate with each other.
- Version 2.0 of the voting-app component are only accessible to users that have a specific feature flag set. This change is managed using a feature flag via a cookie.



First, update the Istio Destination Rules and Virtual Services to cater for these new components. These updates ensure that you don't route traffic incorrectly to the new components and users don't get unexpected access:

```
kubectl apply -f istio/step-3-add-routing-for-2.0-components.yaml --namespace voting
```

The following example output shows the Destination Rules and Virtual Services being updated:

```
destinationrule.networking.istio.io/voting-app configured
virtualservice.networking.istio.io/voting-app configured
destinationrule.networking.istio.io/voting-analytics configured
virtualservice.networking.istio.io/voting-analytics configured
destinationrule.networking.istio.io/voting-storage configured
virtualservice.networking.istio.io/voting-storage configured
```

Next, let's add the Kubernetes objects for the new version `2.0` components. You also update the `voting-storage` service to include the `3306` port for MySQL:

```
kubectl apply -f kubernetes/step-3-update-voting-app-with-new-storage.yaml --namespace voting
```

The following example output shows the Kubernetes objects are successfully updated or created:

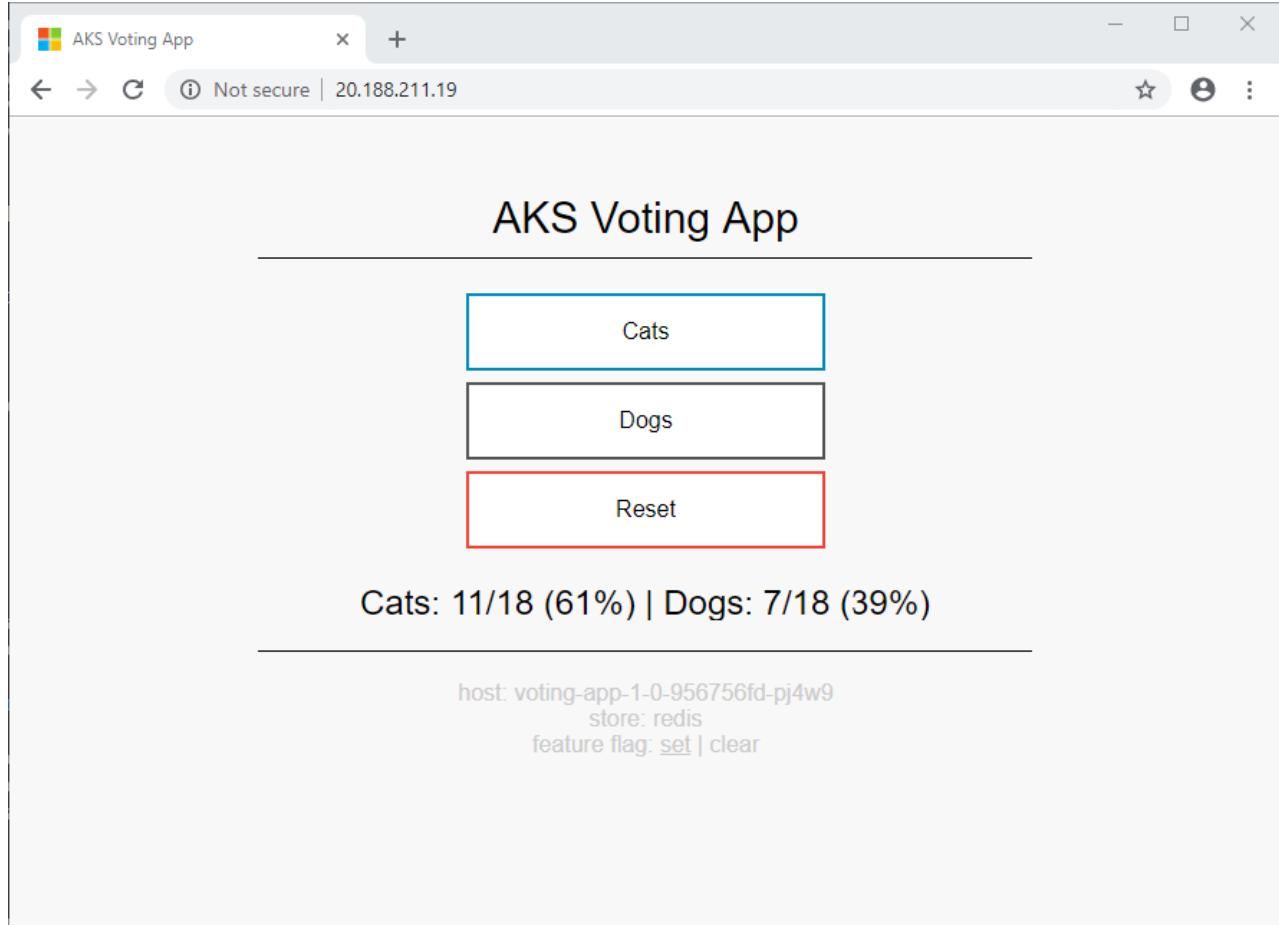
```
service/voting-storage configured
secret/voting-storage-secret created
deployment.apps/voting-storage-2-0 created
persistentvolumeclaim/mysql-pv-claim created
deployment.apps/voting-analytics-2-0 created
deployment.apps/voting-app-2-0 created
```

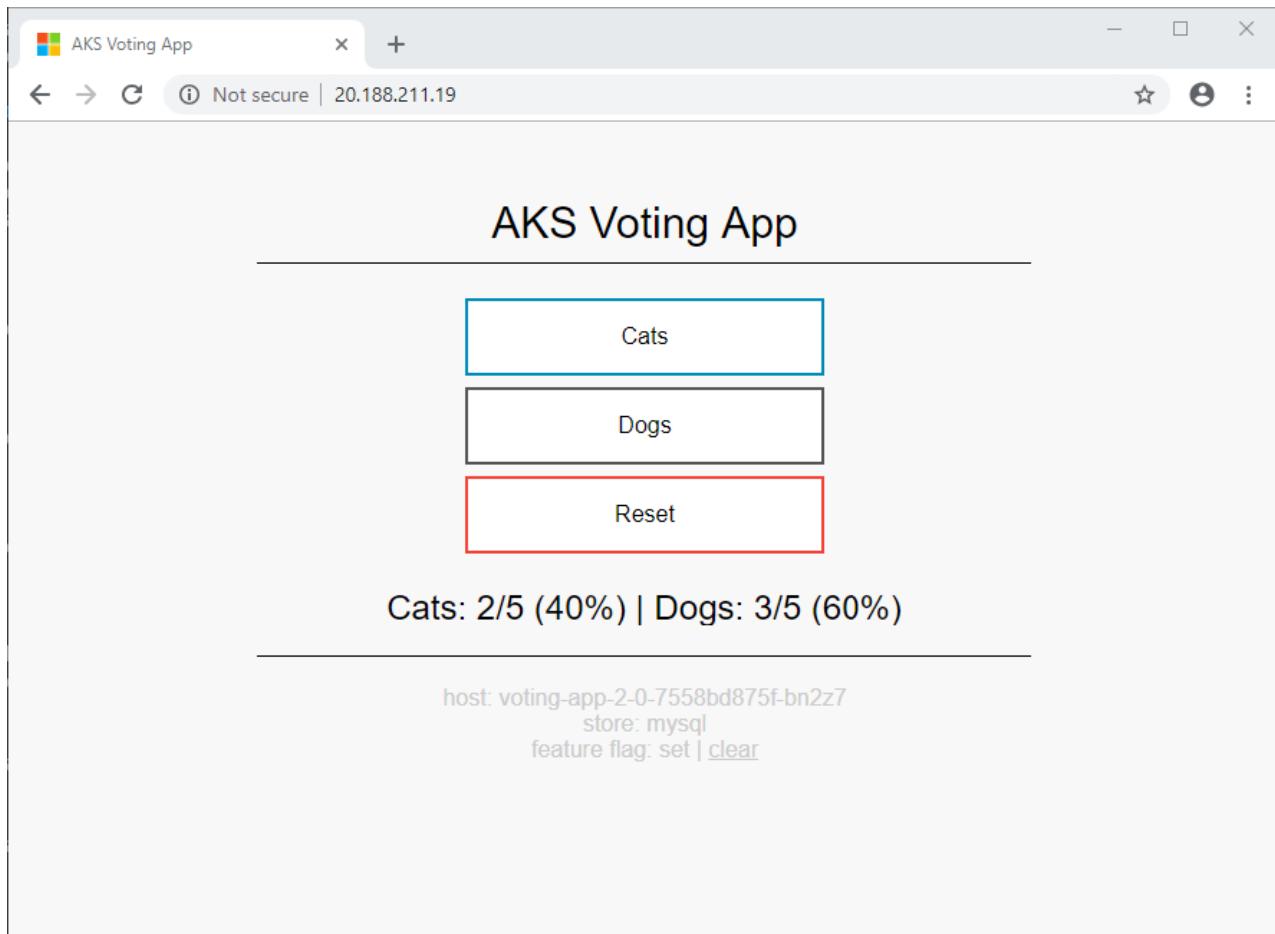
Wait until all the version `2.0` pods are running. Use the `kubectl get pods` command with the `-w` watch switch to

watch for changes on all pods in the `voting` namespace:

```
kubectl get pods --namespace voting -w
```

You should now be able to switch between the version `1.0` and version `2.0` (canary) of the voting application. The feature flag toggle at the bottom of the screen sets a cookie. This cookie is used by the `voting-app` Virtual Service to route users to the new version `2.0`.

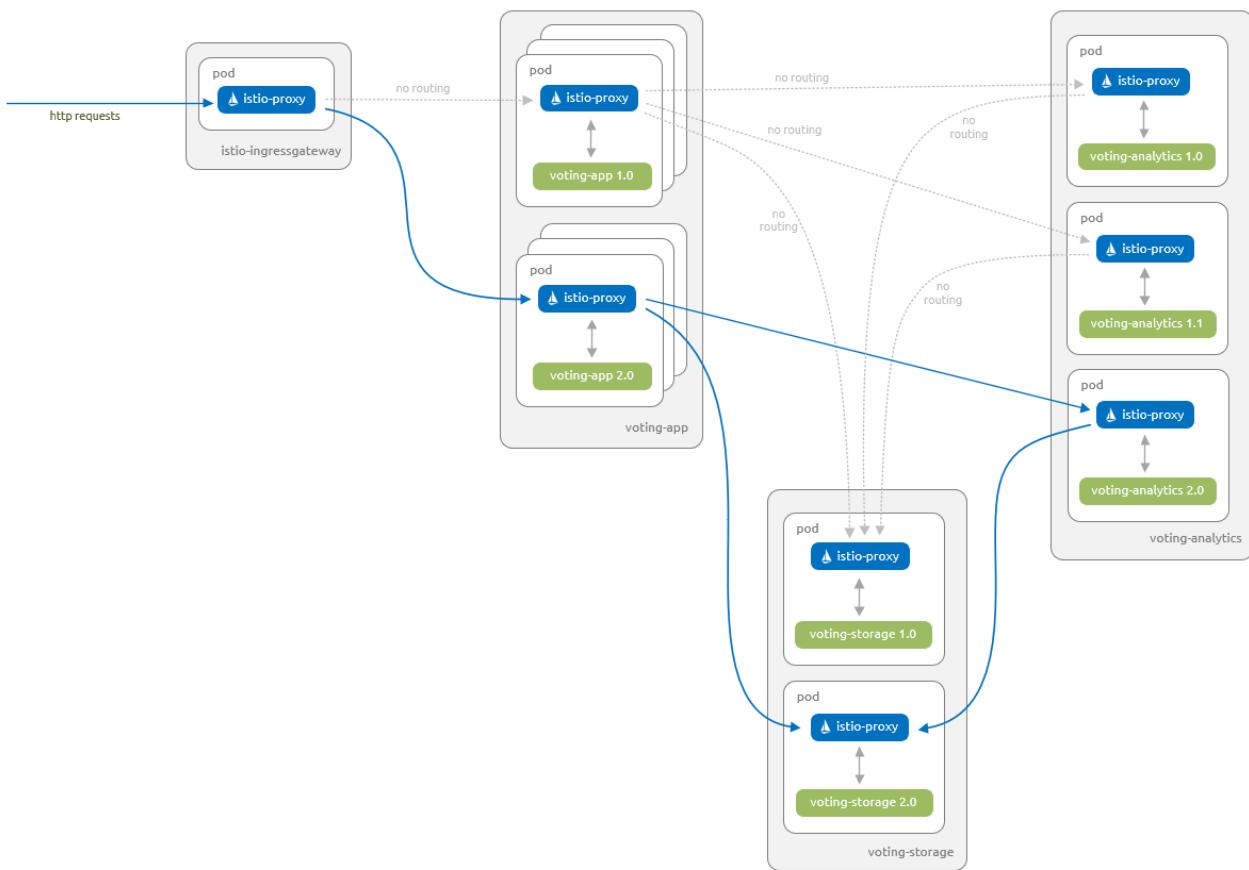




The vote counts are different between the versions of the app. This difference highlights that you are using two different storage backends.

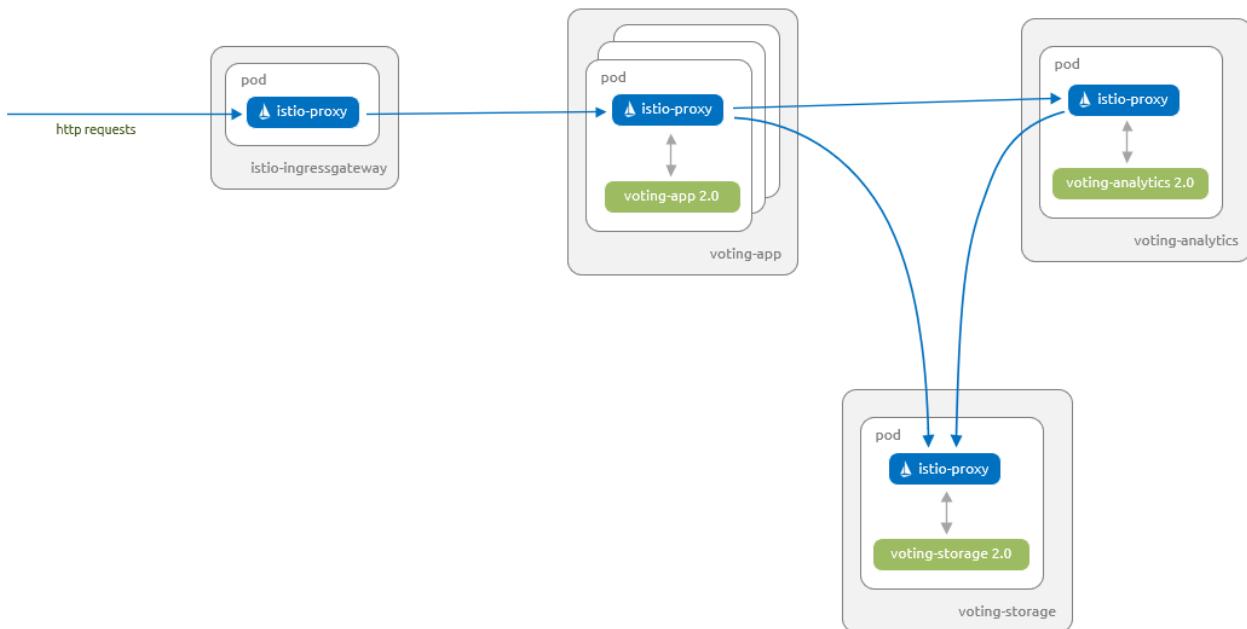
## Finalize the rollout

Once you've successfully tested the canary release, update the `voting-app` Virtual Service to route all traffic to version `2.0` of the `voting-app` component. All users then see version `2.0` of the application, regardless of whether the feature flag is set or not:



Update all the Destination Rules to remove the versions of the components you no longer want active. Then, update all the Virtual Services to stop referencing those versions.

Since there's no longer any traffic to any of the older versions of the components, you can now safely delete all the deployments for those components.



You have now successfully rolled out a new version of the AKS Voting App.

## Clean up

You can remove the AKS voting app we used in this scenario from your AKS cluster by deleting the `voting`

namespace as follows:

```
kubectl delete namespace voting
```

The following example output shows that all the components of the AKS voting app have been removed from your AKS cluster.

```
namespace "voting" deleted
```

## Next steps

You can explore additional scenarios using the [Istio Bookinfo Application example](#).

# Linkerd

2/25/2020 • 2 minutes to read • [Edit Online](#)

## Overview

Linkerd is an easy to use and lightweight service mesh.

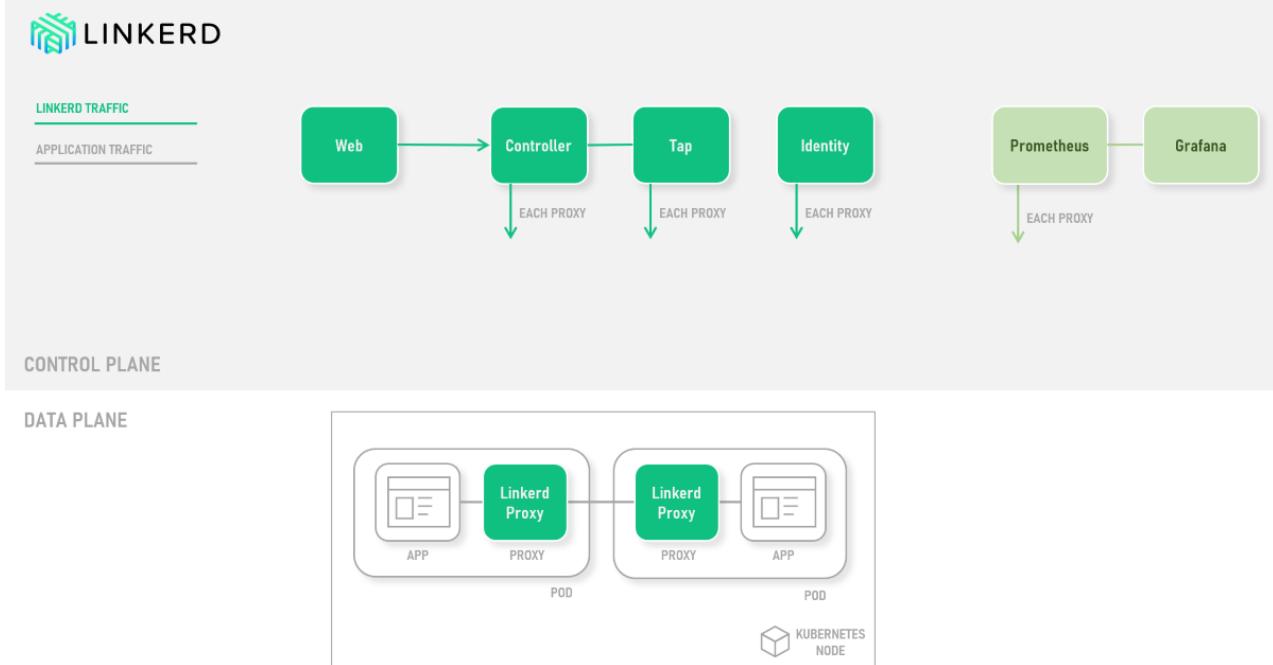
## Architecture

Linkerd provides a data plane that is composed of ultralight Linkerd specialised proxy sidecars. These intelligent proxies control all network traffic in and out of your meshed apps and workloads. The proxies also expose metrics via [Prometheus](#) metrics endpoints.

The control plane manages the configuration and aggregated telemetry via the following [components](#):

- **Controller** - Provides api that drives the Linkerd CLI and Dashboard. Provides configuration for proxies.
- **Tap** - Establish real-time watches on requests and responses.
- **Identity** - Provides identity and security capabilities that allow for mTLS between services.
- **Web** - Provides the Linkerd dashboard.

The following architecture diagram demonstrates how the various components within the data plane and control plane interact.



## Selection criteria

It's important to understand and consider the following areas when evaluating Linkerd for your workloads:

- [Design Principles](#)
- [Capabilities](#)
- [Scenarios](#)

## Design principles

The following design principles [guide](#) the Linkerd project:

- **Keep it Simple** - Must be easy to use and understand.
- **Minimize Resource Requirements** - Impose minimal performance and resource cost.
- **Just Work** - Don't break existing applications and don't require complex configuration.

## Capabilities

Linkerd provides the following set of capabilities:

- **Mesh** – built in debugging option
- **Traffic Management** – splitting, timeouts, retries, ingress
- **Security** – encryption (mTLS), certificates autorotated every 24 hours
- **Observability** – golden metrics, tap, tracing, service profiles and per route metrics, web dashboard with topology graphs, prometheus, grafana

## Scenarios

Linkerd is well suited to and suggested for the following scenarios:

- Simple to use with just the essential set of capability requirements
- Low latency, low overhead, with focus on observability and simple traffic management

## Next steps

The following documentation describes how you can install Linkerd on Azure Kubernetes Service (AKS):

### [Install Linkerd in Azure Kubernetes Service \(AKS\)](#)

You can also further explore Linkerd features and architecture:

- [Linkerd Features](#)
- [Linkerd Architecture](#)

# Install Linkerd in Azure Kubernetes Service (AKS)

2/25/2020 • 8 minutes to read • [Edit Online](#)

Linkerd is an open-source service mesh and [CNCF incubating project](#). Linkerd is an ultralight service mesh that provides features that include traffic management, service identity and security, reliability, and observability. For more information about Linkerd, see the official [Linkerd FAQ](#) and [Linkerd Architecture](#) documentation.

This article shows you how to install Linkerd. The Linkerd `linkerd` client binary is installed onto your client machine and the Linkerd components are installed into a Kubernetes cluster on AKS.

## NOTE

These instructions reference Linkerd version `stable-2.6.0`.

The Linkerd `stable-2.6.x` can be run against Kubernetes versions `1.13+`. You can find additional stable and edge Linkerd versions at [GitHub - Linkerd Releases](#).

In this article, you learn how to:

- Download and install the Linkerd linkerd client binary
- Install Linkerd on AKS
- Validate the Linkerd installation
- Access the Dashboard
- Uninstall Linkerd from AKS

## Before you begin

The steps detailed in this article assume that you've created an AKS cluster (Kubernetes `1.13` and above, with RBAC enabled) and have established a `kubectl` connection with the cluster. If you need help with any of these items, then see the [AKS quickstart](#).

All Linkerd pods must be scheduled to run on Linux nodes - this setup is the default in the installation method detailed below and requires no additional configuration.

This article separates the Linkerd installation guidance into several discrete steps. The result is the same in structure as the official Linkerd getting started [guidance](#).

## Download and install the Linkerd linkerd client binary

In a bash-based shell on Linux or [Windows Subsystem for Linux](#), use `curl` to download the Linkerd release as follows:

```
# Specify the Linkerd version that will be leveraged throughout these instructions
LINKERD_VERSION=stable-2.6.0

curl -sLO "https://github.com/linkerd/linkerd2/releases/download/$LINKERD_VERSION/linkerd2-cli-$LINKERD_VERSION-linux"
```

The `linkerd` client binary runs on your client machine and allows you to interact with the Linkerd service mesh. Use the following commands to install the Linkerd `linkerd` client binary in a bash-based shell on Linux or [Windows Subsystem for Linux](#). These commands copy the `linkerd` client binary to the standard user program

location in your `PATH`.

```
sudo cp ./linkerd2-cli-$LINKERD_VERSION-linux /usr/local/bin/linkerd
sudo chmod +x /usr/local/bin/linkerd
```

If you'd like command-line completion for the Linkerd `linkerd` client binary, then set it up as follows:

```
# Generate the bash completion file and source it in your current shell
mkdir -p ~/completions && linkerd completion bash > ~/completions/linkerd.bash
source ~/completions/linkerd.bash

# Source the bash completion file in your .bashrc so that the command-line completions
# are permanently available in your shell
echo "source ~/completions/linkerd.bash" >> ~/.bashrc
```

## Download and install the Linkerd linkerd client binary

In a bash-based shell on MacOS, use `curl` to download the Linkerd release as follows:

```
# Specify the Linkerd version that will be leveraged throughout these instructions
$LINKERD_VERSION=stable-2.6.0

curl -sLO "https://github.com/linkerd/linkerd2/releases/download/$LINKERD_VERSION/linkerd2-cli-$LINKERD_VERSION-darwin"
```

The `linkerd` client binary runs on your client machine and allows you to interact with the Linkerd service mesh. Use the following commands to install the Linkerd `linkerd` client binary in a bash-based shell on MacOS. These commands copy the `linkerd` client binary to the standard user program location in your `PATH`.

```
sudo cp ./linkerd2-cli-$LINKERD_VERSION-darwin /usr/local/bin/linkerd
sudo chmod +x /usr/local/bin/linkerd
```

If you'd like command-line completion for the Linkerd `linkerd` client binary, then set it up as follows:

```
# Generate the bash completion file and source it in your current shell
mkdir -p ~/completions && linkerd completion bash > ~/completions/linkerd.bash
source ~/completions/linkerd.bash

# Source the bash completion file in your .bashrc so that the command-line completions
# are permanently available in your shell
echo "source ~/completions/linkerd.bash" >> ~/.bashrc
```

## Download and install the Linkerd linkerd client binary

In a PowerShell-based shell on Windows, use `Invoke-WebRequest` to download the Linkerd release as follows:

```
# Specify the Linkerd version that will be leveraged throughout these instructions
$LINKERD_VERSION="stable-2.6.0"

# Enforce TLS 1.2
[Net.ServicePointManager]::SecurityProtocol = "tls12"
$ProgressPreference = 'SilentlyContinue'; Invoke-WebRequest -URI
"https://github.com/linkerd/linkerd2/releases/download/$LINKERD_VERSION/linkerd2-cli-$LINKERD_VERSION-windows.exe" -OutFile "linkerd2-cli-$LINKERD_VERSION-windows.exe"
```

The `linkerd` client binary runs on your client machine and allows you to interact with the Linkerd service mesh. Use the following commands to install the Linkerd `linkerd` client binary in a PowerShell-based shell on Windows. These commands copy the `linkerd` client binary to a Linkerd folder and then make it available both immediately (in current shell) and permanently (across shell restarts) via your `PATH`. You don't need elevated (Admin) privileges to run these commands and you don't need to restart your shell.

```
# Copy linkerd.exe to C:\Linkerd
New-Item -ItemType Directory -Force -Path "C:\Linkerd"
Copy-Item -Path ".\linkerd2-cli-$LINKERD_VERSION-windows.exe" -Destination "C:\Linkerd\linkerd.exe"

# Add C:\Linkerd to PATH.
# Make the new PATH permanently available for the current User, and also immediately available in the current
shell.
$PATH = [environment]::GetEnvironmentVariable("PATH", "User") + "; C:\Linkerd\
[environment]::SetEnvironmentVariable("PATH", $PATH, "User")
[environment]::SetEnvironmentVariable("PATH", $PATH)
```

## Install Linkerd on AKS

Before we install Linkerd, we'll run pre-installation checks to determine if the control plane can be installed on our AKS cluster:

```
linkerd check --pre
```

You should see something like the following to indicate that your AKS cluster is a valid installation target for Linkerd:

```
kubernetes-api
-----
✓ can initialize the client
✓ can query the Kubernetes API

kubernetes-version
-----
✓ is running the minimum Kubernetes API version
✓ is running the minimum kubectl version

pre-kubernetes-setup
-----
✓ control plane namespace does not already exist
✓ can create Namespaces
✓ can create ClusterRoles
✓ can create ClusterRoleBindings
✓ can create CustomResourceDefinitions
✓ can create PodSecurityPolicies
✓ can create ServiceAccounts
✓ can create Services
✓ can create Deployments
✓ can create CronJobs
✓ can create ConfigMaps
✓ no clock skew detected

pre-kubernetes-capability
-----
✓ has NET_ADMIN capability
✓ has NET_RAW capability

pre-linkerd-global-resources
-----
✓ no ClusterRoles exist
✓ no ClusterRoleBindings exist
✓ no CustomResourceDefinitions exist
✓ no MutatingWebhookConfigurations exist
✓ no ValidatingWebhookConfigurations exist
✓ no PodSecurityPolicies exist

linkerd-version
-----
✓ can determine the latest version
✓ cli is up-to-date

Status check results are ✓
```

Now it's time to install the Linkerd components. Use the `linkerd` and `kubectl` binaries to install the Linkerd components into your AKS cluster. A `linkerd` namespace will be automatically created, and the components will be installed into this namespace.

```
linkerd install | kubectl apply -f -
```

Linkerd deploys a number of objects. You'll see the list from the output of your `linkerd install` command above. The deployment of the Linkerd components should take around 1 minute to complete, depending on your cluster environment.

At this point, you've deployed Linkerd to your AKS cluster. To ensure we have a successful deployment of Linkerd, let's move on to the next section to [Validate the Linkerd installation](#).

## Validate the Linkerd installation

Confirm that the resources have been successfully created. Use the `kubectl get svc` and `kubectl get pod` commands

to query the `linkerd` namespace, where the Linkerd components were installed by the `linkerd install` command:

```
kubectl get svc --namespace linkerd --output wide  
kubectl get pod --namespace linkerd --output wide
```

The following example output shows the services and pods (scheduled on Linux nodes) that should now be running:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR	
linkerd-controller-api	ClusterIP	10.0.110.67	<none>	8085/TCP	66s	linkerd.io/control-plane-component=controller	
linkerd-destination	ClusterIP	10.0.224.29	<none>	8086/TCP	66s	linkerd.io/control-plane-component=destination	
linkerd-dst	ClusterIP	10.0.225.148	<none>	8086/TCP	66s	linkerd.io/control-plane-component=grafana	
linkerd-grafana	ClusterIP	10.0.61.124	<none>	3000/TCP	65s	linkerd.io/control-plane-component=identity	
linkerd-identity	ClusterIP	10.0.6.104	<none>	8080/TCP	67s	linkerd.io/control-plane-component=prometheus	
linkerd-prometheus	ClusterIP	10.0.27.168	<none>	9090/TCP	65s	linkerd.io/control-plane-component=proxy-injector	
linkerd-proxy-injector	ClusterIP	10.0.100.133	<none>	443/TCP	64s	linkerd.io/control-plane-component=sp-validator	
linkerd-sp-validator	ClusterIP	10.0.221.5	<none>	443/TCP	64s	linkerd.io/control-plane-component=tap	
linkerd-tap	ClusterIP	10.0.18.14	<none>	8088/TCP,443/TCP	64s	linkerd.io/control-plane-component=web	
linkerd-web	ClusterIP	10.0.37.108	<none>	8084/TCP,9994/TCP	66s	linkerd.io/control-plane-component=grafana	
NAME	NOMINATED NODE	READY	STATUS	RESTARTS	AGE	IP	NODE
linkerd-controller-66ddc9f94f-cm9kt	vmss000001	3/3	Running	0	66s	10.240.0.50	aks-linux-16165125-0
linkerd-destination-c94bc454-qpkng	vmss000002	2/2	Running	0	66s	10.240.0.78	aks-linux-16165125-0
linkerd-grafana-6868fdcb66-4cmq2	vmss000002	2/2	Running	0	65s	10.240.0.69	aks-linux-16165125-0
linkerd-identity-74d8df4b85-tqq8f	vmss000001	2/2	Running	0	66s	10.240.0.48	aks-linux-16165125-0
linkerd-prometheus-699587cf8-k8ghg	vmss000001	2/2	Running	0	65s	10.240.0.41	aks-linux-16165125-0
linkerd-proxy-injector-6556447f64-n29wr	vmss000000	2/2	Running	0	64s	10.240.0.32	aks-linux-16165125-0
linkerd-sp-validator-56745cd567-v4x7h	vmss000000	2/2	Running	0	64s	10.240.0.6	aks-linux-16165125-0
linkerd-tap-5cd9fc566-ct988	vmss000000	2/2	Running	0	64s	10.240.0.15	aks-linux-16165125-0
linkerd-web-774c79b6d5-dhhwf	vmss000002	2/2	Running	0	65s	10.240.0.70	aks-linux-16165125-0

Linkerd provides a command via the `linkerd` client binary to validate that the Linkerd control plane was successfully installed and configured.

```
linkerd check
```

You should see something like the following to indicate that your installation was successful:

```

kubernetes-api
-----
✓ can initialize the client
✓ can query the Kubernetes API

kubernetes-version
-----
✓ is running the minimum Kubernetes API version
✓ is running the minimum kubectl version

linkerd-config
-----
✓ control plane Namespace exists
✓ control plane ClusterRoles exist
✓ control plane ClusterRoleBindings exist
✓ control plane ServiceAccounts exist
✓ control plane CustomResourceDefinitions exist
✓ control plane MutatingWebhookConfigurations exist
✓ control plane ValidatingWebhookConfigurations exist
✓ control plane PodSecurityPolicies exist

linkerd-existence
-----
✓ 'linkerd-config' config map exists
✓ heartbeat ServiceAccount exist
✓ control plane replica sets are ready
✓ no unschedulable pods
✓ controller pod is running
✓ can initialize the client
✓ can query the control plane API

linkerd-api
-----
✓ control plane pods are ready
✓ control plane self-check
✓ [kubernetes] control plane can talk to Kubernetes
✓ [prometheus] control plane can talk to Prometheus
✓ no invalid service profiles

linkerd-version
-----
✓ can determine the latest version
✓ cli is up-to-date

control-plane-version
-----
✓ control plane is up-to-date
✓ control plane and cli versions match

Status check results are ✓

```

## Access the dashboard

Linkerd comes with a dashboard that provides insight into the service mesh and workloads. To access the dashboard, use the `linkerd dashboard` command. This command leverages [kubectl port-forward](#) to create a secure connection between your client machine and the relevant pods in your AKS cluster. It will then automatically open the dashboard in your default browser.

```
linkerd dashboard
```

The command will also create a port-forward and return a link for the Grafana dashboards.

```
Linkerd dashboard available at:  
http://127.0.0.1:50750  
Grafana dashboard available at:  
http://127.0.0.1:50750/grafana  
Opening Linkerd dashboard in the default browser
```

## Uninstall Linkerd from AKS

### WARNING

Deleting Linkerd from a running system may result in traffic related issues between your services. Ensure that you have made provisions for your system to still operate correctly without Linkerd before proceeding.

First you'll need to remove the data plane proxies. Remove any Automatic Proxy Injection [annotations](#) from workload namespaces and roll out your workload deployments. Your workloads should no longer have any associated data plane components.

Finally, remove the control plane as follows:

```
linkerd install --ignore-cluster | kubectl delete -f -
```

## Next steps

To explore more installation and configuration options for Linkerd, see the following official Linkerd guidance:

- [Linkerd - Helm installation](#)
- [Linkerd - Multi-stage installation to cater for role privileges](#)

You can also follow additional scenarios using:

- [Linkerd emojivoto demo](#)
- [Linkerd books demo](#)

# Consul

2/25/2020 • 2 minutes to read • [Edit Online](#)

## Overview

Consul is a multi data centre aware service networking solution to connect and secure services across runtime platforms. [Connect](#) is the component that provides service mesh capabilities.

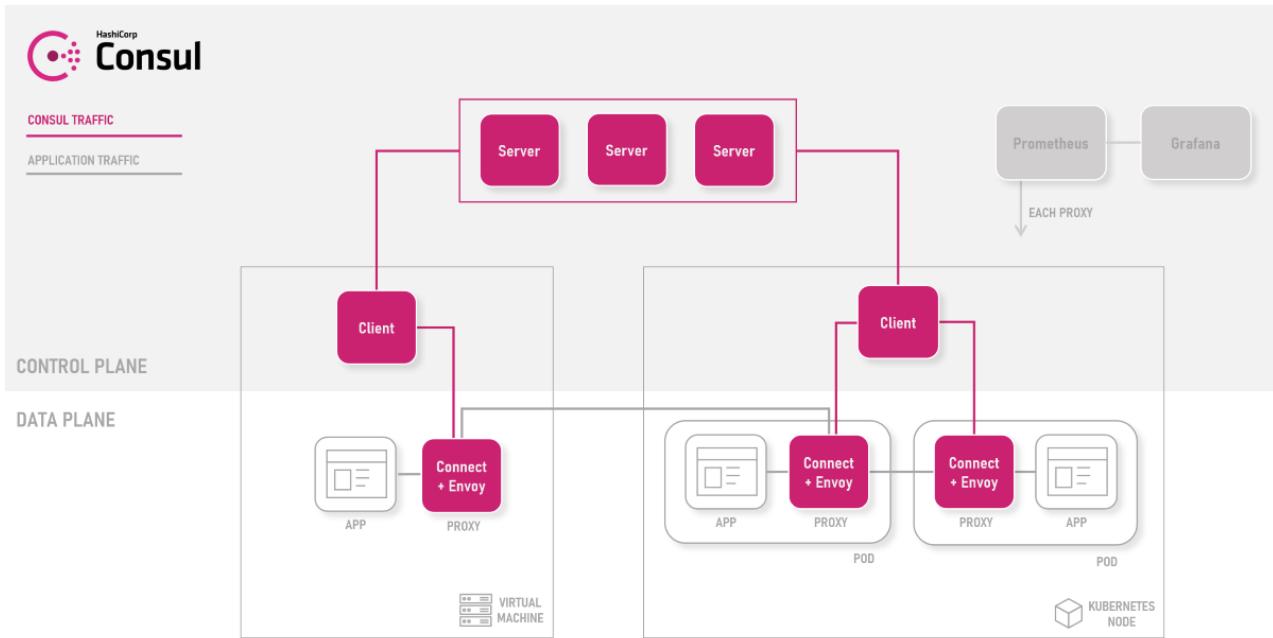
## Architecture

Consul provides a data plane that is composed of [Envoy](#)-based [sidecars](#) by default. Consul has a pluggable proxy architecture. These intelligent proxies control all network traffic in and out of your meshed apps and workloads.

The control plane manages the configuration, and policy via the following [components](#):

- **Server** - A Consul Agent running in Server mode that maintains Consul cluster state.
- **Client** - A Consul Agent running in lightweight Client Mode. Each compute node must have a Client agent running. This client brokers configuration and policy between the workloads and the Consul configuration.

The following architecture diagram demonstrates how the various components within the data plane and control plane interact.



## Selection criteria

It's important to understand and consider the following areas when evaluating Consul for your workloads:

- [Consul Principles](#)
- [Capabilities](#)
- [Scenarios](#)

### Consul principles

The following principles [guide](#) the Consul project:

- **API-Driven** - Codify all configuration and policy.
- **Run and Connect Anywhere** - Connect workloads across runtime platforms (Kubernetes, VMs, Serverless).
- **Extend and Integrate** - Securely connect workloads across infrastructure.

## Capabilities

Consul provides the following set of capabilities:

- **Mesh** – gateway (multi data centre), virtual machines (out of cluster nodes), service sync, built in debugging option
- **Proxies** – Envoy, built-in proxy, pluggable, I4 proxy available for Windows workloads
- **Traffic Management** – routing, splitting, resolution
- **Policy** – intentions, ACLs
- **Security** – authorisation, authentication, encryption, SPIFFE-based identities, external CA (Vault), certificate management, and rotation
- **Observability** – metrics, ui dashboard, prometheus, grafana

## Scenarios

Consul is well suited to and suggested for the following scenarios:

- Extending existing Consul connected workloads
- Compliance requirements around certificate management
- Multi cluster service mesh
- VM-based workloads to be included in the service mesh

## Next steps

The following documentation describes how you can install Consul on Azure Kubernetes Service (AKS):

### [Install Consul in Azure Kubernetes Service \(AKS\)](#)

You can also further explore Consul features and architecture:

- [Consul Features](#)
- [Consul Architecture](#)
- [Consul - How Connect Works](#)

# Install and use Consul in Azure Kubernetes Service (AKS)

2/25/2020 • 6 minutes to read • [Edit Online](#)

Consul is an open-source service mesh that provides a key set of functionality across the microservices in a Kubernetes cluster. These features include service discovery, health checking, service segmentation, and observability. For more information about Consul, see the official [What is Consul?](#) documentation.

This article shows you how to install Consul. The Consul components are installed into a Kubernetes cluster on AKS.

## NOTE

These instructions reference Consul version `1.6.0`, and use at least Helm version `2.14.2`.

The Consul `1.6.x` releases can be run against Kubernetes versions `1.13+`. You can find additional Consul versions at [GitHub - Consul Releases](#) and information about each of the releases at [Consul- Release Notes](#).

In this article, you learn how to:

- Install the Consul components on AKS
- Validate the Consul installation
- Uninstall Consul from AKS

## Before you begin

The steps detailed in this article assume that you've created an AKS cluster (Kubernetes `1.13` and above, with RBAC enabled) and have established a `kubectl` connection with the cluster. If you need help with any of these items, then see the [AKS quickstart](#). Ensure that your cluster has at least 3 nodes in the Linux node pool.

You'll need [Helm](#) to follow these instructions and install Consul. It's recommended that you have the latest stable version correctly installed and configured in your cluster. If you need help with installing Helm, then see the [AKS Helm installation guidance](#). All Consul pods must also be scheduled to run on Linux nodes.

This article separates the Consul installation guidance into several discrete steps. The end result is the same in structure as the official Consul installation [guidance](#).

### Install the Consul components on AKS

We'll start by downloading version `v0.10.0` of the Consul Helm chart. This version of the chart includes Consul version `1.6.0`.

In a bash-based shell on Linux, [Windows Subsystem for Linux](#) or MacOS, use `curl` to download the Consul Helm chart release as follows:

```
# Specify the Consul Helm chart version that will be leveraged throughout these instructions  
CONSUL_HELM_VERSION=0.10.0  
  
curl -sL "https://github.com/hashicorp/consul-helm/archive/v$CONSUL_HELM_VERSION.tar.gz" | tar xz  
mv consul-helm-$CONSUL_HELM_VERSION consul-helm
```

In a bash-based shell on Linux, [Windows Subsystem for Linux](#) or MacOS, use `curl` to download the Consul Helm

chart release as follows:

```
# Specify the Consul Helm chart version that will be leveraged throughout these instructions  
CONSUL_HELM_VERSION=0.10.0  
  
curl -sL "https://github.com/hashicorp/consul-helm/archive/v$CONSUL_HELM_VERSION.tar.gz" | tar xz  
mv consul-helm-$CONSUL_HELM_VERSION consul-helm
```

In a PowerShell-based shell on Windows, use `Invoke-WebRequest` to download the Consul Helm chart release and then extract with `Expand-Archive` as follows:

```
# Specify the Consul Helm chart version that will be leveraged throughout these instructions  
$CONSUL_HELM_VERSION="0.10.0"  
  
# Enforce TLS 1.2  
[Net.ServicePointManager]::SecurityProtocol = "tls12"  
$ProgressPreference = 'SilentlyContinue'; Invoke-WebRequest -URI "https://github.com/hashicorp/consul-helm/archive/v$CONSUL_HELM_VERSION.zip" -OutFile "consul-helm-$CONSUL_HELM_VERSION.zip"  
Expand-Archive -Path "consul-helm-$CONSUL_HELM_VERSION.zip" -DestinationPath .  
Move-Item -Path consul-helm-$CONSUL_HELM_VERSION -Destination consul-helm
```

Use Helm and the downloaded `consul-helm` chart to install the Consul components into the `consul` namespace in your AKS cluster.

## NOTE

### Installation options

We are using the following options as part of our installation:

- `connectInject.enabled=true` - enable proxies to be injected into pods
- `client.enabled=true` - enable Consul clients to run on every node
- `client.grpc=true` - enable gRPC listener for connectInject
- `syncCatalog.enabled=true` - sync Kubernetes and Consul services

### Node selectors

Consul currently must be scheduled to run on Linux nodes. If you have Windows Server nodes in your cluster, you must ensure that the Consul pods are only scheduled to run on Linux nodes. We'll use [node selectors](#) to make sure pods are scheduled to the correct nodes.

```
helm install -f consul-helm/values.yaml --name consul --namespace consul ./consul-helm \  
--set connectInject.enabled=true --set connectInject.nodeSelector="beta.kubernetes.io/os: linux" \  
--set client.enabled=true --set client.grpc=true --set client.nodeSelector="beta.kubernetes.io/os: linux" \  
--set server.nodeSelector="beta.kubernetes.io/os: linux" \  
--set syncCatalog.enabled=true --set syncCatalog.nodeSelector="beta.kubernetes.io/os: linux"
```

```
helm install -f consul-helm/values.yaml --name consul --namespace consul ./consul-helm \  
--set connectInject.enabled=true --set connectInject.nodeSelector="beta.kubernetes.io/os: linux" \  
--set client.enabled=true --set client.grpc=true --set client.nodeSelector="beta.kubernetes.io/os: linux" \  
--set server.nodeSelector="beta.kubernetes.io/os: linux" \  
--set syncCatalog.enabled=true --set syncCatalog.nodeSelector="beta.kubernetes.io/os: linux"
```

```
helm install -f consul-helm/values.yaml --name consul --namespace consul ./consul-helm \
--set connectInject.enabled=true --set connectInject.nodeSelector="beta.kubernetes.io/os: linux" \
--set client.enabled=true --set client.grpc=true --set client.nodeSelector="beta.kubernetes.io/os: linux" \
--set server.nodeSelector="beta.kubernetes.io/os: linux" \
--set syncCatalog.enabled=true --set syncCatalog.nodeSelector="beta.kubernetes.io/os: linux"
```

The `Consul` Helm chart deploys a number of objects. You can see the list from the output of your `helm install` command above. The deployment of the Consul components can take around 3 minutes to complete, depending on your cluster environment.

At this point, you've deployed Consul to your AKS cluster. To ensure that we have a successful deployment of Consul, let's move on to the next section to validate the Consul installation.

## Validate the Consul installation

Confirm that the resources have been successfully created. Use the `kubectl get svc` and `kubectl get pod` commands to query the `consul` namespace, where the Consul components were installed by the `helm install` command:

```
kubectl get svc --namespace consul --output wide
kubectl get pod --namespace consul --output wide
```

The following example output shows the services and pods (scheduled on Linux nodes) that should now be running:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)		
AGE	SELECTOR	ExternalName	<none>	consul.service.consul	<none>	
38s	<none>					
3m26s	app=consul,component=connect-injector,release=consul	ClusterIP	10.0.98.102	<none>		443/TCP
	consul-consul-dns	ClusterIP	10.0.46.194	<none>		53/TCP,53/UDP
	consul-consul-server	ClusterIP	None	<none>		
	consul-consul-ui	ClusterIP	10.0.50.188	<none>		80/TCP
3m26s	app=consul,component=server,release=consul					
NAME	NOMINATED NODE	READY	STATUS	RESTARTS	AGE	IP
NODE		GATES				
10.240.0.68	aks-linux-92468653-vmss000002	<none>	<none>		3m9s	
10.240.0.44	aks-linux-92468653-vmss000001	<none>	<none>		3m9s	
10.240.0.70	aks-linux-92468653-vmss000002	<none>	<none>		3m9s	
10.240.0.91	aks-linux-92468653-vmss000002	<none>	<none>		3m9s	
10.240.0.38	aks-linux-92468653-vmss000001	<none>	<none>		3m9s	
10.240.0.10	aks-linux-92468653-vmss000000	<none>	<none>		3m9s	
10.240.0.94	aks-linux-92468653-vmss000002	<none>	<none>		3m9s	
10.240.0.12	aks-linux-92468653-vmss000000	<none>	<none>		3m9s	

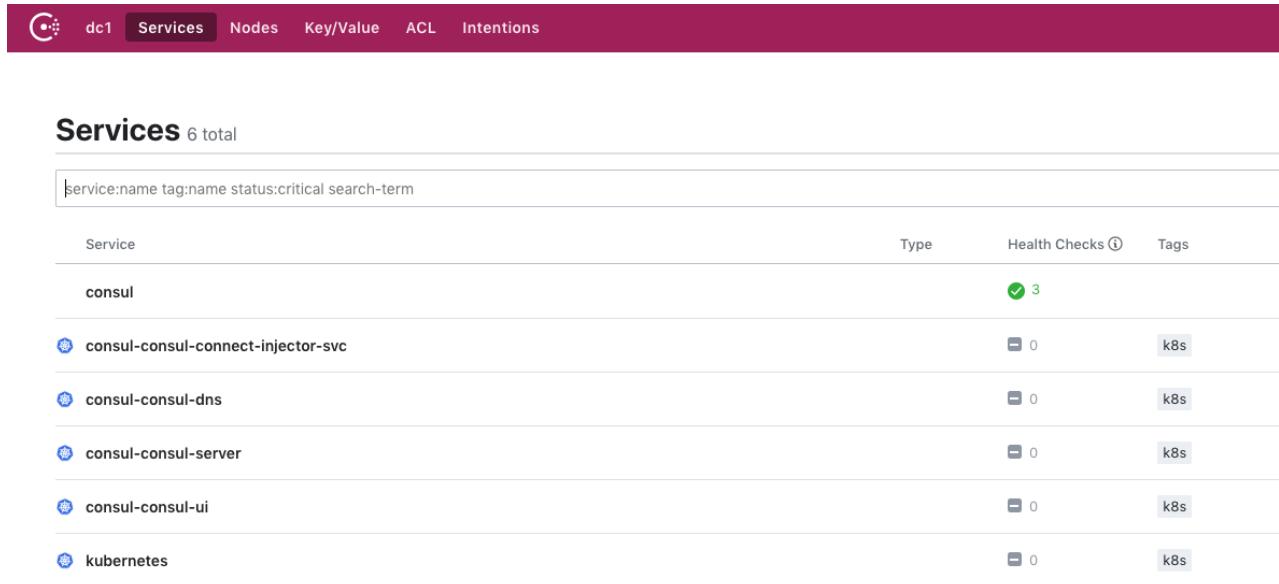
All of the pods should show a status of `Running`. If your pods don't have these statuses, wait a minute or two until they do. If any pods report an issue, use the `kubectl describe pod` command to review their output and status.

# Accessing the Consul UI

The Consul UI was installed in our setup above and provides UI based configuration for Consul. The UI for Consul is not exposed publicly via an external ip address. To access the Consul user interface, use the `kubectl port-forward` command. This command creates a secure connection between your client machine and the relevant pod in your AKS cluster.

```
kubectl port-forward -n consul svc/consul-consul-ui 8080:80
```

You can now open a browser and point it to `http://localhost:8080/ui` to open the Consul UI. You should see the following when you open the UI:



The screenshot shows the Consul UI's Services page. At the top, there are tabs: dc1, Services (which is selected and highlighted in red), Nodes, Key/Value, ACL, and Intentions. Below the tabs, the title "Services" is followed by "6 total". A search bar contains the placeholder text "service:name tag:name status:critical search-term". The main table lists six services:

Service	Type	Health Checks ⓘ	Tags
consul		✓ 3	
consul-consul-connect-injector-svc		0	k8s
consul-consul-dns		0	k8s
consul-consul-server		0	k8s
consul-consul-ui		0	k8s
kubernetes		0	k8s

## Uninstall Consul from AKS

### WARNING

Deleting Consul from a running system may result in traffic related issues between your services. Ensure that you have made provisions for your system to still operate correctly without Consul before proceeding.

### Remove Consul components and namespace

To remove Consul from your AKS cluster, use the following commands. The `helm delete` commands will remove the `consul` chart, and the `kubectl delete namespace` command will remove the `consul` namespace.

```
helm delete --purge consul
kubectl delete namespace consul
```

## Next steps

To explore more installation and configuration options for Consul, see the following official Consul articles:

- [Consul - Helm installation guide](#)
- [Consul - Helm installation options](#)

You can also follow additional scenarios using:

- [Consul Example Application](#)



2 minutes to read

# Tutorial: Deploy ASP.NET Core apps to Azure Kubernetes Service with Azure DevOps Projects

10/4/2019 • 7 minutes to read • [Edit Online](#)

Azure DevOps Projects presents a simplified experience where you can bring your existing code and Git repo or choose a sample application to create a continuous integration (CI) and continuous delivery (CD) pipeline to Azure.

DevOps Projects also:

- Automatically creates Azure resources, such as Azure Kubernetes Service (AKS).
- Creates and configures a release pipeline in Azure DevOps that sets up a build and release pipeline for CI/CD.
- Creates an Azure Application Insights resource for monitoring.
- Enables [Azure Monitor for containers](#) to monitor performance for the container workloads on the AKS cluster

In this tutorial, you will:

- Use DevOps Projects to deploy an ASP.NET Core app to AKS
- Configure Azure DevOps and an Azure subscription
- Examine the AKS cluster
- Examine the CI pipeline
- Examine the CD pipeline
- Commit changes to Git and automatically deploy them to Azure
- Clean up resources

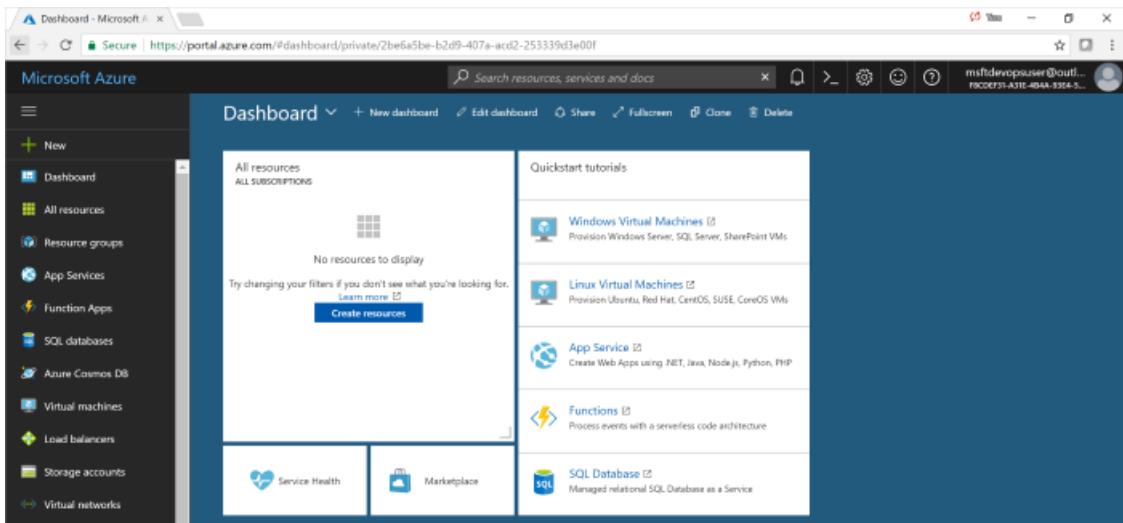
## Prerequisites

- An Azure subscription. You can get one free through [Visual Studio Dev Essentials](#).

## Use DevOps Projects to deploy an ASP.NET Core app to AKS

DevOps Projects creates a CI/CD pipeline in Azure Pipelines. You can create a new Azure DevOps organization or use an existing organization. DevOps Projects also creates Azure resources, such as an AKS cluster, in the Azure subscription of your choice.

1. Sign in to the [Azure portal](#).
2. In the left pane, select **Create a resource**.
3. In the search box, type **DevOps Projects**, and then select **Create**.



4. Select **.NET**, and then select **Next**.
5. Under **Choose an application framework**, select **ASP.NET Core**.
6. Select **Kubernetes Service**, and then select **Next**.

## Configure Azure DevOps and an Azure subscription

1. Create a new Azure DevOps organization, or select an existing organization.
2. Enter a name for your Azure DevOps project.
3. Select your Azure subscription.
4. To view additional Azure configuration settings and to identify the number of nodes for the AKS cluster, select **Change**.  
This pane displays various options for configuring the type and location of Azure services.

5. Exit the Azure configuration area, and then select **Done**.

After a few minutes, the process is completed. A sample ASP.NET Core app is set up in a Git repo in your Azure DevOps organization, an AKS cluster is created, a CI/CD pipeline is executed, and your app is deployed to Azure.

After all this is completed, the Azure DevOps Project dashboard is displayed in the Azure portal. You can also go to the DevOps Projects dashboard directly from **All resources** in the Azure portal.

This dashboard provides visibility into your Azure DevOps code repository, your CI/CD pipeline, and your AKS cluster. You can configure additional CI/CD options in your Azure DevOps pipeline. At the right, select **Browse** to view your running app.

## Examine the AKS cluster

DevOps Projects automatically configures an AKS cluster, which you can explore and customize. To familiarize yourself with the AKS cluster, do the following:

1. Go to the DevOps Projects dashboard.
2. At the right, select the AKS service.  
A pane opens for the AKS cluster. From this view you can perform various actions, such as monitoring container health, searching logs, and opening the Kubernetes dashboard.
3. At the right, select **View Kubernetes dashboard**.  
Optionally, follow the steps to open the Kubernetes dashboard.

## Examine the CI pipeline

DevOps Projects automatically configures a CI/CD pipeline in your Azure DevOps organization. You can explore and customize the pipeline. To familiarize yourself with it, do the following:

1. Go to the DevOps Projects dashboard.
2. At the top of the DevOps Projects dashboard, select **Build Pipelines**.  
A browser tab displays the build pipeline for your new project.
3. Point to the **Status** field, and then select the ellipsis (...).  
A menu displays several options, such as queueing a new build, pausing a build, and editing the build pipeline.
4. Select **Edit**.
5. In this pane, you can examine the various tasks for your build pipeline.  
The build performs various tasks, such as fetching sources from the Git repo, restoring dependencies, and publishing outputs used for deployments.
6. At the top of the build pipeline, select the build pipeline name.
7. Change the name of your build pipeline to something more descriptive, select **Save & queue**, and then select **Save**.
8. Under your build pipeline name, select **History**.  
This pane displays an audit trail of your recent changes for the build. Azure DevOps keeps track of any changes made to the build pipeline, and it allows you to compare versions.
9. Select **Triggers**.  
DevOps Projects automatically creates a CI trigger, and every commit to the repo starts a new build. Optionally, you can choose to include or exclude branches from the CI process.
10. Select **Retention**.  
Depending on your scenario, you can specify policies to keep or remove a certain number of builds.

## Examine the CD release pipeline

DevOps Projects automatically creates and configures the necessary steps to deploy from your Azure DevOps organization to your Azure subscription. These steps include configuring an Azure service connection to authenticate Azure DevOps to your Azure subscription. The automation also creates a release pipeline, which provides the CD to Azure. To learn more about the release pipeline, do the following:

1. Select **Build and Release**, and then select **Releases**.  
DevOps Projects creates a release pipeline to manage deployments to Azure.
2. Select the ellipsis (...) next to your release pipeline, and then select **Edit**.  
The release pipeline contains a *pipeline*, which defines the release process.
3. Under **Artifacts**, select **Drop**.  
The build pipeline you examined in the previous steps produces the output that's used for the artifact.
4. At the right of the **Drop** icon, select **Continuous deployment trigger**.  
This release pipeline has an enabled CD trigger, which executes a deployment every time a new build artifact is available. Optionally, you can disable the trigger so that your deployments require manual execution.
5. At the right, select **View releases** to display a history of releases.
6. Select the ellipsis (...) next to a release, and then select **Open**.

You can explore several menus, such as a release summary, associated work items, and tests.

#### 7. Select **Commits**.

This view shows code commits that are associated with this deployment. Compare releases to view the commit differences between deployments.

#### 8. Select **Logs**.

The logs contain useful information about the deployment process. You can view them both during and after deployments.

## Commit changes to Azure Repos and automatically deploy them to Azure

#### NOTE

The following procedure tests the CI/CD pipeline by making a simple text change.

You're now ready to collaborate with a team on your app by using a CI/CD process that automatically deploys your latest work to your website. Each change to the Git repo starts a build in Azure DevOps, and a CD pipeline executes a deployment to Azure. Follow the procedure in this section, or use another technique to commit changes to your repo. For example, you can clone the Git repo in your favorite tool or IDE, and then push changes to this repo.

1. In the Azure DevOps menu, select **Code > Files**, and then go to your repo.
2. Go to the *Views\Home* directory, select the ellipsis (...) next to the *Index.cshtml* file, and then select **Edit**.
3. Make a change to the file, such as adding some text within one of the div tags.
4. At the top right, select **Commit**, and then select **Commit** again to push your change.  
After a few moments, a build starts in Azure DevOps and a release executes to deploy the changes. Monitor the build status on the DevOps Projects dashboard or in the browser with your Azure DevOps organization.
5. After the release is completed, refresh your app to verify your changes.

## Clean up resources

If you are testing, you can avoid accruing billing charges by cleaning up your resources. When they are no longer needed, you can delete the AKS cluster and related resources that you created in this tutorial. To do so, use the **Delete** functionality on the DevOps Projects dashboard.

#### IMPORTANT

The following procedure permanently deletes resources. The *Delete* functionality destroys the data that's created by the project in DevOps Projects in both Azure and Azure DevOps, and you will be unable to retrieve it. Use this procedure only after you've carefully read the prompts.

1. In the Azure portal, go to the DevOps Projects dashboard.
2. At the top right, select **Delete**.
3. At the prompt, select **Yes** to *permanently delete* the resources.

## Next steps

You can optionally modify these build and release pipelines to meet the needs of your team. You can also use this CI/CD pattern as a template for your other pipelines. In this tutorial, you learned how to:

- Use DevOps Projects to deploy an ASP.NET Core app to AKS
- Configure Azure DevOps and an Azure subscription
- Examine the AKS cluster
- Examine the CI pipeline
- Examine the CD pipeline
- Commit changes to Git and automatically deploy them to Azure
- Clean up resources

To learn more about using the Kubernetes dashboard, see:

[Use the Kubernetes dashboard](#)

# Deployment Center for Azure Kubernetes

2/25/2020 • 4 minutes to read • [Edit Online](#)

Deployment Center in Azure DevOps simplifies setting up a robust Azure DevOps pipeline for your application. By default, Deployment Center configures an Azure DevOps pipeline to deploy your application updates to the Kubernetes cluster. You can extend the default configured Azure DevOps pipeline and also add richer capabilities: the ability to gain approval before deploying, provision additional Azure resources, run scripts, upgrade your application, and even run more validation tests.

In this tutorial, you will:

- Configure an Azure DevOps pipeline to deploy your application updates to the Kubernetes cluster.
- Examine the continuous integration (CI) pipeline.
- Examine the continuous delivery (CD) pipeline.
- Clean up the resources.

## Prerequisites

- An Azure subscription. You can get one free through [Visual Studio Dev Essentials](#).
- An Azure Kubernetes Service (AKS) cluster.

## Create an AKS cluster

1. Sign in to your [Azure portal](#).
2. Select the [Cloud Shell](#) option on the right side of the menu bar in the Azure portal.
3. To create the AKS cluster, run the following commands:

```
# Create a resource group in the South India location:  
  
az group create --name azooaks --location southindia  
  
# Create a cluster named azookubectl with one node.  
  
az aks create --resource-group azooaks --name azookubectl --node-count 1 --enable-addons monitoring --  
generate-ssh-keys
```

## Deploy application updates to a Kubernetes cluster

1. Go to the resource group that you created in the previous section.
2. Select the AKS cluster, and then select **Deployment Center (preview)** on the left blade. Select **Get started**.

The screenshot shows the Azure Deployment Center (preview) interface. On the left, there's a sidebar with various options like Overview, Activity log, Access control (IAM), Tags, Settings (Upgrade, Scale, Dev Spaces, Deployment center (preview)), Properties, Locks, Export template, Monitoring, and Insights. The 'Deployment center (preview)' option is highlighted with a red arrow pointing to it.

**Deployment Center**

Deployment center in Azure DevOps simplifies setting up a robust DevOps pipeline for your application. By default, this configures a DevOps pipeline to deploy your application updates to this Kubernetes cluster. You can extend the default configured DevOps pipeline and add richer DevOps capabilities - approvals before deploying, provisioning additional Azure resources, running scripts, upgrading your application, or even running additional validation tests.

3. Choose the location of the code and select **Next**. Then, select one of the currently supported repositories: **Azure Repos** or **GitHub**.

Azure Repos is a set of version control tools that help you manage your code. Whether your software project is large or small, using version control as early as possible is a good idea.

- **Azure Repos:** Choose a repository from your existing project and organization.

The screenshot shows the 'Select the code location' step in a deployment wizard. At the top, there's a progress bar with four steps: 1 Source (highlighted in blue), 2 Repository, 3 Application, and 4 Resources. Below the progress bar, the text 'Select the code location' is centered.

**Azure Repos**

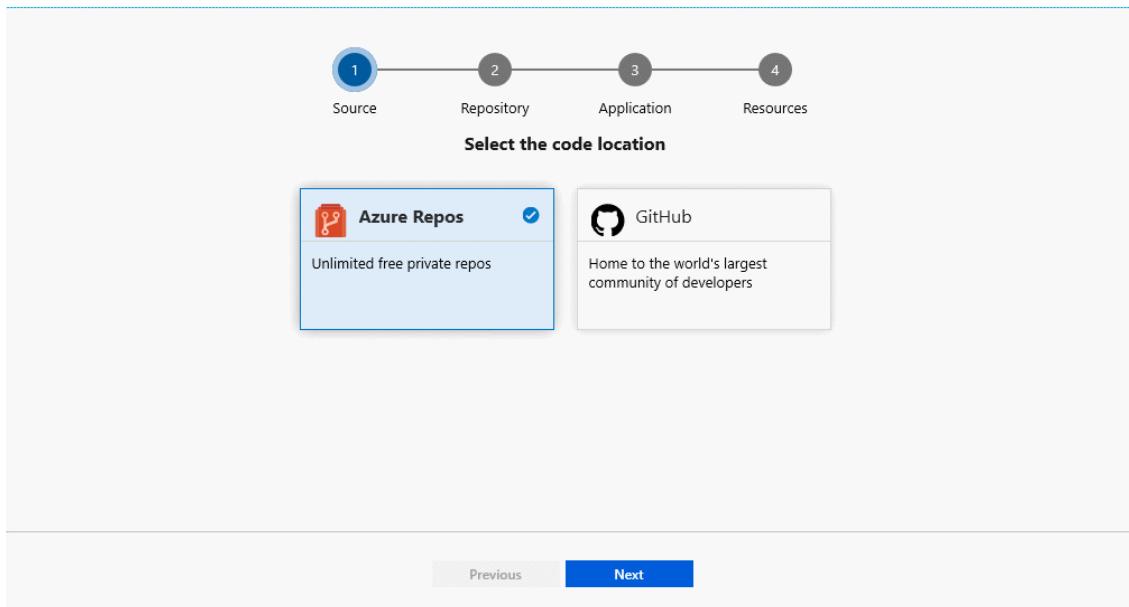
Unlimited free private repos

**GitHub**

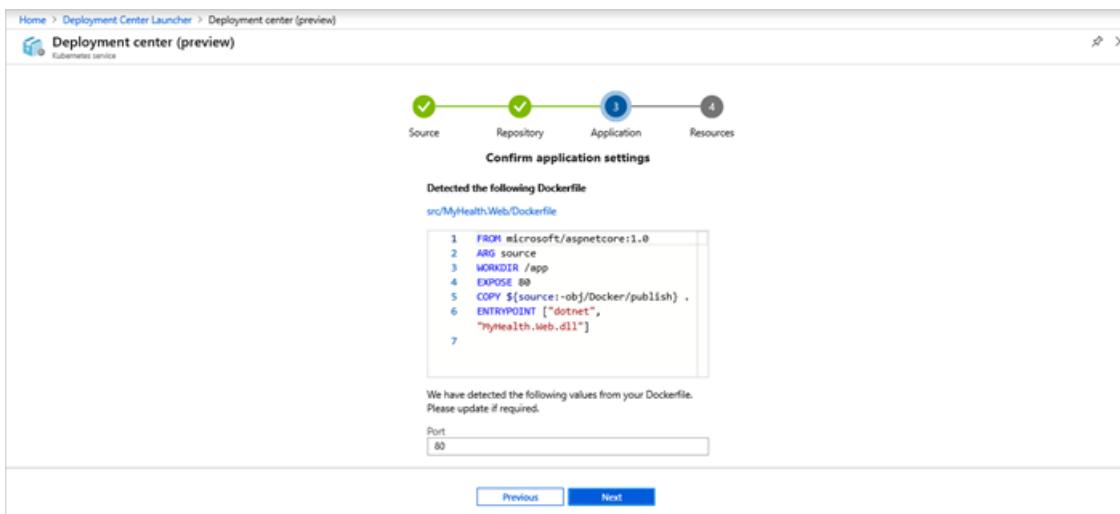
Home to the world's largest community of developers

Previous **Next**

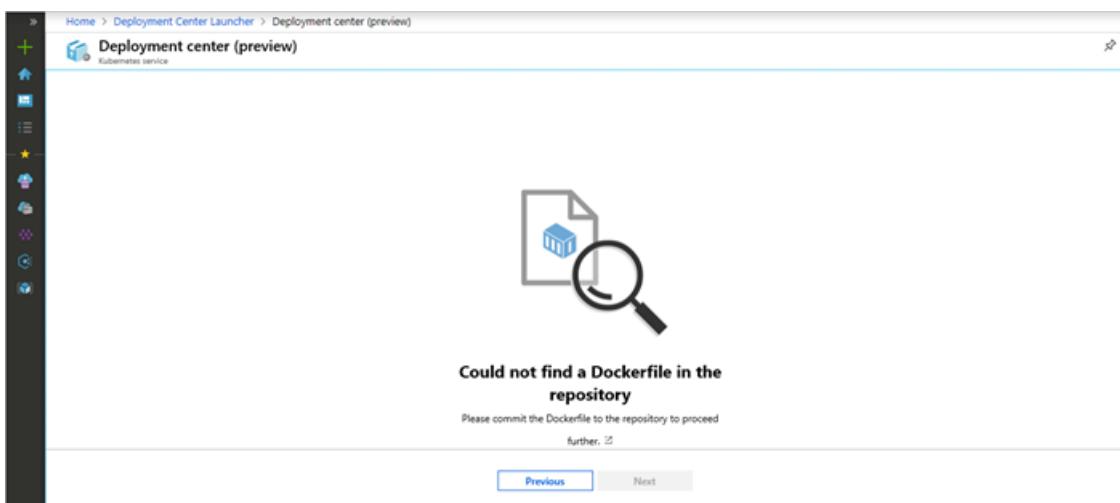
- **GitHub:** Authorize and select the repository for your GitHub account.



- Deployment Center analyzes the repository and detects your Dockerfile. If you want to update the Dockerfile, you can edit the identified port number.



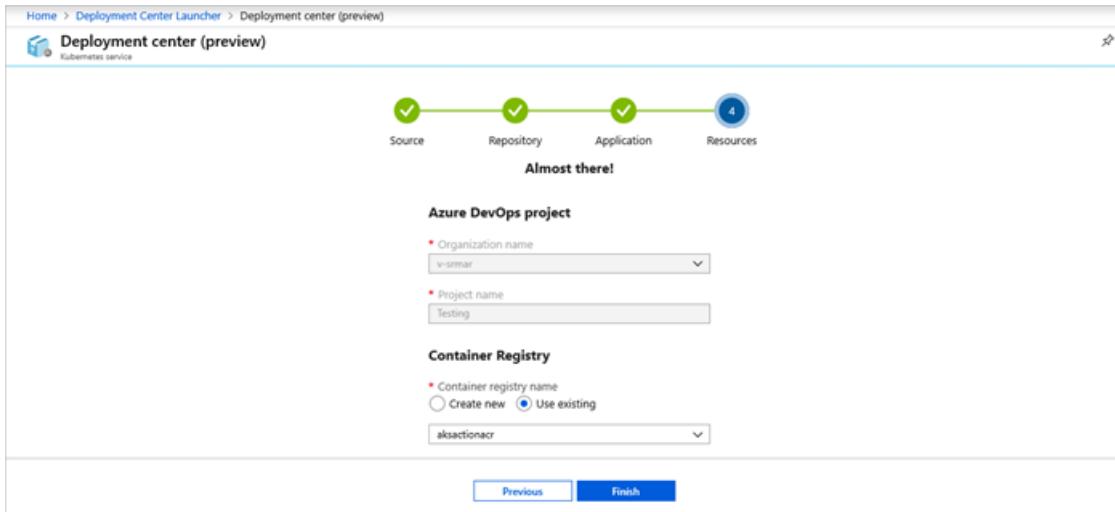
If the repository doesn't contain the Dockerfile, the system displays a message to commit one.



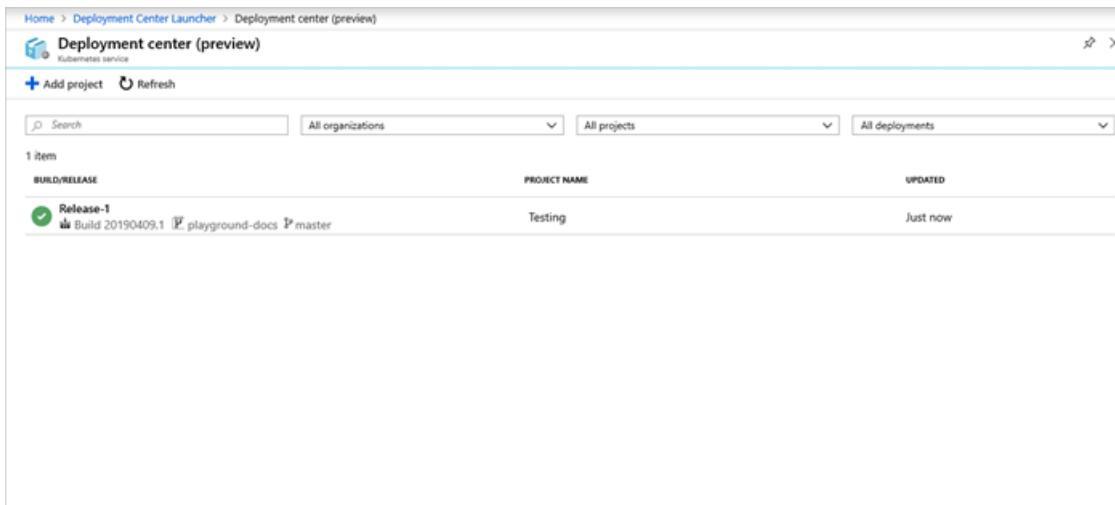
- Select an existing container registry or create one, and then select **Finish**. The pipeline is created automatically and queues a build in [Azure Pipelines](#).

Azure Pipelines is a cloud service that you can use to automatically build and test your code project and make it available to other users. Azure Pipelines combines continuous integration and continuous delivery

to constantly and consistently test and build your code and ship it to any target.



6. Select the link to see the ongoing pipeline.
7. You'll see the successful logs after deployment is complete.



## Examine the CI pipeline

Deployment Center automatically configures your Azure DevOps organization's CI/CD pipeline. The pipeline can be explored and customized.

1. Go to the Deployment Center dashboard.
2. Select the build number from the list of successful logs to view the build pipeline for your project.
3. Select the ellipsis (...) in the upper-right corner. A menu shows several options, such as queuing a new build, retaining a build, and editing the build pipeline. Select **Edit pipeline**.
4. You can examine the different tasks for your build pipeline in this pane. The build performs various tasks, such as collecting sources from the Git repository, creating an image, pushing an image to the container registry, and publishing outputs that are used for deployments.
5. Select the name of the build pipeline at the top of the pipeline.
6. Change your build pipeline name to something more descriptive, select **Save & queue**, and then select **Save**.
7. Under your build pipeline, select **History**. This pane shows an audit trail of your recent build changes. Azure DevOps monitors any changes made to the build pipeline and allows you to compare versions.

8. Select **Triggers**. You can include or exclude branches from the CI process.
9. Select **Retention**. You can specify policies to keep or remove a number of builds, depending on your scenario.

## Examine the CD pipeline

Deployment Center automatically creates and configures the relationship between your Azure DevOps organization and your Azure subscription. The steps involved include setting up an Azure service connection to authenticate your Azure subscription with Azure DevOps. The automated process also creates a release pipeline, which provides continuous delivery to Azure.

1. Select **Pipelines**, and then select **Releases**.
2. To edit the release pipeline, select **Edit**.
3. Select **Drop** from the **Artifacts** list. In the previous steps, the construction pipeline you examined produces the output used for the artifact.
4. Select the **Continuous deployment** trigger on the right of the **Drop** option. This release pipeline has an enabled CD trigger that runs a deployment whenever a new build artifact is available. You can also disable the trigger to require manual execution for your deployments.
5. To examine all the tasks for your pipeline, select **Tasks**. The release sets the tiller environment, configures the `imagePullSecrets` parameter, installs Helm tools, and deploys the Helm charts to the Kubernetes cluster.
6. To view the release history, select **View releases**.
7. To see the summary, select **Release**. Select any of the stages to explore multiple menus, such as a release summary, associated work items, and tests.
8. Select **Commits**. This view shows code commits related to this deployment. Compare releases to see the commit differences between deployments.
9. Select **Logs**. The logs contain useful deployment information, which you can view during and after deployments.

## Clean up resources

You can delete the related resources that you created when you don't need them anymore. Use the delete functionality on the DevOps Projects dashboard.

## Next steps

You can modify these build and release pipelines to meet the needs of your team. Or, you can use this CI/CD model as a template for your other pipelines.

# GitHub Actions for deploying to Kubernetes service

2/25/2020 • 3 minutes to read • [Edit Online](#)

[GitHub Actions](#) gives you the flexibility to build an automated software development lifecycle workflow. The Kubernetes action [azure/aks-set-context@v1](#) facilitate deployments to Azure Kubernetes Service clusters. The action sets the target AKS cluster context, which could be used by other actions like [azure/k8s-deploy](#), [azure/k8s-create-secret](#) etc. or run any kubectl commands.

A workflow is defined by a YAML (.yml) file in the `/.github/workflows/` path in your repository. This definition contains the various steps and parameters that make up the workflow.

For a workflow targeting AKS, the file has three sections:

SECTION	TASKS
<b>Authentication</b>	Login to a private container registry (ACR)
<b>Build</b>	Build & push the container image
<b>Deploy</b>	<ol style="list-style-type: none"><li>Set the target AKS cluster</li></ol>
	<ol style="list-style-type: none"><li>Create a generic/docker-registry secret in Kubernetes cluster</li></ol>
	<ol style="list-style-type: none"><li>Deploy to the Kubernetes cluster</li></ol>

## Create a service principal

You can create a [service principal](#) by using the [az ad sp create-for-rbac](#) command in the [Azure CLI](#). You can run this command using [Azure Cloud Shell](#) in the Azure portal or by selecting the **Try it** button.

```
az ad sp create-for-rbac --name "myApp" --role contributor --scopes /subscriptions/<SUBSCRIPTION_ID>/resourceGroups/<RESOURCE_GROUP> --sdk-auth
```

In the above command, replace the placeholders with your subscription ID, and resource group. The output is the role assignment credentials that provide access to your resource. The command should output a JSON object similar to this.

```
{  
  "clientId": "<GUID>",  
  "clientSecret": "<GUID>",  
  "subscriptionId": "<GUID>",  
  "tenantId": "<GUID>",  
  (...)  
}
```

Copy this JSON object, which you can use to authenticate from GitHub.

## Configure the GitHub secrets

Follow the steps to configure the secrets:

- In [GitHub](#), browse to your repository, select **Settings > Secrets > Add a new secret**.

The screenshot shows the GitHub repository settings interface. On the left, there's a sidebar with options like Options, Collaborators, Branches, Webhooks, Notifications, Integrations & services, Deploy keys, **Secrets** (which is selected and highlighted with an orange border), and Actions. On the right, under the **Secrets** heading, there's a note about encrypted environment variables and a link to learn more. Below that is a large button labeled "Add a new secret", which is also highlighted with a red box.

- Paste the contents of the above `az cli` command as the value of secret variable. For example, `AZURE_CREDENTIALS`.
- Similarly, define the following additional secrets for the container registry credentials and set them in Docker login action.
  - `REGISTRY_USERNAME`
  - `REGISTRY_PASSWORD`
- You will see the secrets as shown below once defined.

The screenshot shows the GitHub repository settings interface again. The sidebar on the left is identical to the previous one. On the right, under the **Secrets** heading, there is a list of three secrets: `AZURE_CREDENTIALS`, `REGISTRY_PASSWORD`, and `REGISTRY_USERNAME`. Each secret has a small icon next to it and a "Remove" button to its right. The entire list of secrets is highlighted with a red box.

## Build a container image and deploy to Azure Kubernetes Service cluster

The build and push of the container images is done using `Azure/docker-login@v1` action. To deploy a container image to AKS, you will need to use the `Azure/k8s-deploy@v1` action. This action has five parameters:

PARAMETER	EXPLANATION
<b>namespace</b>	(Optional) Choose the target Kubernetes namespace. If the namespace is not provided, the commands will run in the default namespace

PARAMETER	EXPLANATION
<b>manifests</b>	(Required) Path to the manifest files, that will be used for deployment
<b>images</b>	(Optional) Fully qualified resource URL of the image(s) to be used for substitutions on the manifest files
<b>imagepullsecrets</b>	(Optional) Name of a docker-registry secret that has already been set up within the cluster. Each of these secret names is added under imagePullSecrets field for the workloads found in the input manifest files
<b>kubectl-version</b>	(Optional) Installs a specific version of kubectl binary

## Deploy to Azure Kubernetes Service cluster

End to end workflow for building container images and deploying to an Azure Kubernetes Service cluster.

```
on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@master

      - uses: Azure/docker-login@v1
        with:
          login-server: contoso.azurecr.io
          username: ${{ secrets.REGISTRY_USERNAME }}
          password: ${{ secrets.REGISTRY_PASSWORD }}

      - run: |
        docker build . -t contoso.azurecr.io/k8sdemo:${{ github.sha }}
        docker push contoso.azurecr.io/k8sdemo:${{ github.sha }}

    # Set the target AKS cluster.
    - uses: Azure/aks-set-context@v1
      with:
        creds: '${{ secrets.AZURE_CREDENTIALS }}'
        cluster-name: contoso
        resource-group: contoso-rg

    - uses: Azure/k8s-create-secret@v1
      with:
        container-registry-url: contoso.azurecr.io
        container-registry-username: ${{ secrets.REGISTRY_USERNAME }}
        container-registry-password: ${{ secrets.REGISTRY_PASSWORD }}
        secret-name: demo-k8s-secret

    - uses: Azure/k8s-deploy@v1
      with:
        manifests: |
          manifests/deployment.yml
          manifests/service.yml
        images: |
          demo.azurecr.io/k8sdemo:${{ github.sha }}
        imagepullsecrets: |
          demo-k8s-secret
```

## Next steps

You can find our set of Actions in different repositories on GitHub, each one containing documentation and examples to help you use GitHub for CI/CD and deploy your apps to Azure.

- [setup-kubectl](#)
- [k8s-set-context](#)
- [aks-set-context](#)
- [k8s-create-secret](#)
- [k8s-deploy](#)
- [webapps-container-deploy](#)
- [actions-workflow-samples](#)

# AKS troubleshooting

2/25/2020 • 19 minutes to read • [Edit Online](#)

When you create or manage Azure Kubernetes Service (AKS) clusters, you might occasionally encounter problems. This article details some common problems and troubleshooting steps.

## In general, where do I find information about debugging Kubernetes problems?

Try the [official guide to troubleshooting Kubernetes clusters](#). There's also a [troubleshooting guide](#), published by a Microsoft engineer for troubleshooting pods, nodes, clusters, and other features.

## I'm getting a "quota exceeded" error during creation or upgrade. What should I do?

You need to [request cores](#).

## What is the maximum pods-per-node setting for AKS?

The maximum pods-per-node setting is 30 by default if you deploy an AKS cluster in the Azure portal. The maximum pods-per-node setting is 110 by default if you deploy an AKS cluster in the Azure CLI. (Make sure you're using the latest version of the Azure CLI). This default setting can be changed by using the `--max-pods` flag in the `az aks create` command.

## I'm getting an insufficientSubnetSize error while deploying an AKS cluster with advanced networking. What should I do?

If Azure CNI (advanced networking) is used, AKS allocates IP addresses based on the "max-pods" per node configured. Based on the configured max pods per node, the subnet size must be greater than the product of the number of nodes and the max pod per node setting. The following equation outlines this:

Subnet size > number of nodes in the cluster (taking into consideration the future scaling requirements) \* max pods per node set.

For more information, see [Plan IP addressing for your cluster](#).

## My pod is stuck in CrashLoopBackOff mode. What should I do?

There might be various reasons for the pod being stuck in that mode. You might look into:

- The pod itself, by using `kubectl describe pod <pod-name>`.
- The logs, by using `kubectl logs <pod-name>`.

For more information on how to troubleshoot pod problems, see [Debug applications](#).

## I'm trying to enable RBAC on an existing cluster. How can I do that?

Unfortunately, enabling role-based access control (RBAC) on existing clusters isn't supported at this time. You must explicitly create new clusters. If you use the CLI, RBAC is enabled by default. If you use the AKS portal, a toggle button to enable RBAC is available in the creation workflow.

I created a cluster with RBAC enabled by using either the Azure CLI with defaults or the Azure portal, and now I see many warnings on the Kubernetes dashboard. The dashboard used to work without any warnings. What should I do?

The reason for the warnings on the dashboard is that the cluster is now enabled with RBAC and access to it has been disabled by default. In general, this approach is good practice because the default exposure of the dashboard to all users of the cluster can lead to security threats. If you still want to enable the dashboard, follow the steps in [this blog post](#).

## I can't connect to the dashboard. What should I do?

The easiest way to access your service outside the cluster is to run `kubectl proxy`, which proxies requests sent to your localhost port 8001 to the Kubernetes API server. From there, the API server can proxy to your service:

```
http://localhost:8001/api/v1/namespaces/kube-system/services/kubernetes-dashboard/proxy#!/node?  
namespace=default
```

If you don't see the Kubernetes dashboard, check whether the `kube-proxy` pod is running in the `kube-system` namespace. If it isn't in a running state, delete the pod and it will restart.

I can't get logs by using `kubectl logs` or I can't connect to the API server. I'm getting "Error from server: error dialing backend: dial tcp...". What should I do?

Make sure that the default network security group isn't modified and that both port 22 and 9000 are open for connection to the API server. Check whether the `tunnelfront` pod is running in the `kube-system` namespace using the `kubectl get pods --namespace kube-system` command. If it isn't, force deletion of the pod and it will restart.

I'm trying to upgrade or scale and am getting a "message: Changing property 'imageReference' is not allowed" error. How do I fix this problem?

You might be getting this error because you've modified the tags in the agent nodes inside the AKS cluster. Modifying and deleting tags and other properties of resources in the MC\_\* resource group can lead to unexpected results. Modifying the resources under the MC\_\* group in the AKS cluster breaks the service-level objective (SLO).

I'm receiving errors that my cluster is in failed state and upgrading or scaling will not work until it is fixed

*This troubleshooting assistance is directed from <https://aka.ms/aks-cluster-failed>*

This error occurs when clusters enter a failed state for multiple reasons. Follow the steps below to resolve your cluster failed state before retrying the previously failed operation:

1. Until the cluster is out of `failed` state, `upgrade` and `scale` operations won't succeed. Common root issues and resolutions include:
  - Scaling with **insufficient compute (CRP) quota**. To resolve, first scale your cluster back to a stable goal state within quota. Then follow these [steps to request a compute quota increase](#) before trying to scale up again beyond initial quota limits.
  - Scaling a cluster with advanced networking and **insufficient subnet (networking) resources**. To resolve, first scale your cluster back to a stable goal state within quota. Then follow [these steps to request](#)

- a resource quota increase before trying to scale up again beyond initial quota limits.
2. Once the underlying cause for upgrade failure is resolved, your cluster should be in a succeeded state. Once a succeeded state is verified, retry the original operation.

## I'm receiving errors when trying to upgrade or scale that state my cluster is being currently being upgraded or has failed upgrade

*This troubleshooting assistance is directed from <https://aka.ms/aks-pending-upgrade>*

Upgrade and scale operations on a cluster with a single node pool or a cluster with [multiple node pools](#) are mutually exclusive. You cannot have a cluster or node pool simultaneously upgrade and scale. Instead, each operation type must complete on the target resource prior to the next request on that same resource. As a result, operations are limited when active upgrade or scale operations are occurring or attempted and subsequently failed.

To help diagnose the issue run `az aks show -g myResourceGroup -n myAKSCluster -o table` to retrieve detailed status on your cluster. Based on the result:

- If cluster is actively upgrading, wait until the operation terminates. If it succeeded, retry the previously failed operation again.
- If cluster has failed upgrade, follow steps outlined in previous section.

## Can I move my cluster to a different subscription or my subscription with my cluster to a new tenant?

If you have moved your AKS cluster to a different subscription or the cluster owning subscription to a new tenant, the cluster will lose functionality due to losing role assignments and service principals rights. **AKS does not support moving clusters across subscriptions or tenants** due to this constraint.

## I'm receiving errors trying to use features that require virtual machine scale sets

*This troubleshooting assistance is directed from <aka.ms/aks-vmss-enablement>*

You may receive errors that indicate your AKS cluster is not on a virtual machine scale set, such as the following example:

### **AgentPool 'agentpool' has set auto scaling as enabled but is not on Virtual Machine Scale Sets**

To use features such as the cluster autoscaler or multiple node pools, AKS clusters must be created that use virtual machine scale sets. Errors are returned if you try to use features that depend on virtual machine scale sets and you target a regular, non-virtual machine scale set AKS cluster.

Follow the *Before you begin* steps in the appropriate doc to correctly create an AKS cluster:

- [Use the cluster autoscaler](#)
- [Create and use multiple node pools](#)

## What naming restrictions are enforced for AKS resources and parameters?

*This troubleshooting assistance is directed from <aka.ms/aks-naming-rules>*

Naming restrictions are implemented by both the Azure platform and AKS. If a resource name or parameter breaks one of these restrictions, an error is returned that asks you provide a different input. The following common naming guidelines apply:

- Cluster names must be 1-63 characters. The only allowed characters are letters, numbers, dashes, and underscores. The first and last character must be a letter or a number.
- The AKS MC\_ resource group name combines resource group name and resource name. The auto-generated syntax of `MC_resourceGroupName_resourceName_AzureRegion` must be no greater than 80 chars. If needed, reduce the length of your resource group name or AKS cluster name.
- The `dnsPrefix` must start and end with alphanumeric values and must be between 1-54 characters. Valid characters include alphanumeric values and hyphens (-). The `dnsPrefix` can't include special characters such as a period (.)

I'm receiving errors when trying to create, update, scale, delete or upgrade cluster, that operation is not allowed as another operation is in progress.

*This troubleshooting assistance is directed from aka.ms/aks-pending-operation*

Cluster operations are limited when a previous operation is still in progress. To retrieve a detailed status of your cluster, use the `az aks show -g myResourceGroup -n myAKSCluster -o table` command. Use your own resource group and AKS cluster name as needed.

Based on the output of the cluster status:

- If the cluster is in any provisioning state other than *Succeeded* or *Failed*, wait until the operation (*Upgrading / Updating / Creating / Scaling / Deleting / Migrating*) terminates. When the previous operation has completed, re-try your latest cluster operation.
- If the cluster has a failed upgrade, follow the steps outlined [I'm receiving errors that my cluster is in failed state and upgrading or scaling will not work until it is fixed.](#)

I'm receiving errors that my service principal was not found when I try to create a new cluster without passing in an existing one.

When creating an AKS cluster it requires a service principal to create resources on your behalf. AKS offers the ability to have a new one created at cluster creation time, but this requires Azure Active Directory to fully propagate the new service principal in a reasonable time in order to have the cluster succeed in creation. When this propagation takes too long, the cluster will fail validation to create as it cannot find an available service principal to do so.

Use the following workarounds for this:

1. Use an existing service principal which has already propagated across regions and exists to pass into AKS at cluster create time.
2. If using automation scripts, add time delays between service principal creation and AKS cluster creation.
3. If using Azure portal, return to the cluster settings during create and retry the validation page after a few minutes.

I'm receiving errors after restricting my egress traffic

When restricting egress traffic from an AKS cluster, there are [required and optional recommended](#) outbound ports / network rules and FQDN / application rules for AKS. If your settings are in conflict with any of these rules, you may not be able to run certain `kubectl` commands. You may also see errors when creating an AKS cluster.

Verify that your settings are not conflicting with any of the required or optional recommended outbound ports / network rules and FQDN / application rules.

# Azure Storage and AKS Troubleshooting

## What are the recommended stable versions of Kubernetes for Azure disk?

KUBERNETES VERSION	RECOMMENDED VERSION
1.12	1.12.9 or later
1.13	1.13.6 or later
1.14	1.14.2 or later

## What versions of Kubernetes have Azure Disk support on the Sovereign Cloud?

KUBERNETES VERSION	RECOMMENDED VERSION
1.12	1.12.0 or later
1.13	1.13.0 or later
1.14	1.14.0 or later

## WaitForAttach failed for Azure Disk: parsing "/dev/disk/azure/scsi1/lun1": invalid syntax

In Kubernetes version 1.10, MountVolume.WaitForAttach may fail with an the Azure Disk remount.

On Linux, you may see an incorrect DevicePath format error. For example:

```
MountVolume.WaitForAttach failed for volume "pvc-f1562ecb-3e5f-11e8-ab6b-000d3af9f967" : azureDisk - Wait for attach expect device path as a lun number, instead got: /dev/disk/azure/scsi1/lun1 (strconv.Atoi: parsing "/dev/disk/azure/scsi1/lun1": invalid syntax)
Warning FailedMount          1m (x10 over 21m)  kubelet, k8s-agentpool-66825246-0  Unable to mount volumes for pod
```

On Windows, you may see a wrong DevicePath(LUN) number error. For example:

```
Warning FailedMount          1m    kubelet, 15282k8s9010  MountVolume.WaitForAttach failed for volume "disk01" : azureDisk - WaitForAttach failed within timeout node (15282k8s9010) diskId:(andy-mghyb1102-dynamic-pvc-6c526c51-4a18-11e8-ab5c-000d3af7b38e) lun:(4)
```

This issue has been fixed in the following versions of Kubernetes:

KUBERNETES VERSION	FIXED VERSION
1.10	1.10.2 or later
1.11	1.11.0 or later
1.12 and later	N/A

## Failure when setting uid and gid in mountOptions for Azure Disk

Azure Disk uses the ext4,xfs filesystem by default and mountOptions such as uid=x,gid=x can't be set at mount time. For example if you tried to set mountOptions uid=999,gid=999, would see an error like:

```
Warning FailedMount          63s                 kubelet, aks-nodepool1-29460110-0
MountVolume.MountDevice failed for volume "pvc-d783d0e4-85a1-11e9-8a90-369885447933" : azureDisk -
mountDevice:FormatAndMount failed with mount failed: exit status 32
Mounting command: systemd-run
Mounting arguments: --description=Kubernetes transient mount for /var/lib/kubelet/plugins/kubernetes.io/azure-
disk/mounts/m436970985 --scope -- mount -t xfs -o dir_mode=0777,file_mode=0777,uid=1000,gid=1000,defaults
/dev/disk/azure/scsi1/lun2 /var/lib/kubelet/plugins/kubernetes.io/azure-disk/mounts/m436970985
Output: Running scope as unit run-rb21966413ab449b3a242ae9b0fb9c9398.scope.
mount: wrong fs type, bad option, bad superblock on /dev/sde,
      missing codepage or helper program, or other error
```

You can mitigate the issue by doing one the following:

- [Configure the security context for a pod](#) by setting uid in runAsUser and gid in fsGroup. For example, the following setting will set pod run as root, make it accessible to any file:

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 0
    fsGroup: 0
```

#### NOTE

Since gid and uid are mounted as root or 0 by default. If gid or uid are set as non-root, for example 1000, Kubernetes will use `chown` to change all directories and files under that disk. This operation can be time consuming and may make mounting the disk very slow.

- Use `chown` in initContainers to set gid and uid. For example:

```
initContainers:
- name: volume-mount
  image: busybox
  command: ["sh", "-c", "chown -R 100:100 /data"]
  volumeMounts:
  - name: <your data volume>
    mountPath: /data
```

#### Error when deleting Azure Disk PersistentVolumeClaim in use by a pod

If you try to delete an Azure Disk PersistentVolumeClaim that is being used by a pod, you may see an error. For example:

```
$ kubectl describe pv pvc-d8eebc1d-74d3-11e8-902b-e22b71bb1c06
...
Message:      disk.DisksClient#Delete: Failure responding to request: StatusCode=409 -- Original Error:
autorest/azure: Service returned an error. Status=409 Code="OperationNotAllowed" Message="Disk kubernetes-
dynamic-pvc-d8eebc1d-74d3-11e8-902b-e22b71bb1c06 is attached to VM /subscriptions/{subs-
id}/resourceGroups/MC_markito-aks-pvc_markito-aks-pvc_westus/providers/Microsoft.Compute/virtualMachines/aks-
agentpool-25259074-0."
```

In Kubernetes version 1.10 and later, there is a PersistentVolumeClaim protection feature enabled by default to prevent this error. If you are using a version of Kubernetes that does not have the fix for this issue, you can mitigate this issue by deleting the pod using the PersistentVolumeClaim before deleting the PersistentVolumeClaim.

## Error "Cannot find Lun for disk" when attaching a disk to a node

When attaching a disk to a node, you may see the following error:

```
MountVolume.WaitForAttach failed for volume "pvc-12b458f4-c23f-11e8-8d27-46799c22b7c6" : Cannot find Lun for
disk kubernetes-dynamic-pvc-12b458f4-c23f-11e8-8d27-46799c22b7c6
```

This issue has been fixed in the following versions of Kubernetes:

KUBERNETES VERSION	FIXED VERSION
1.10	1.10.10 or later
1.11	1.11.5 or later
1.12	1.12.3 or later
1.13	1.13.0 or later
1.14 and later	N/A

If you are using a version of Kubernetes that does not have the fix for this issue, you can mitigate the issue by waiting several minutes and retrying.

## Azure Disk attach/detach failure, mount issues, or I/O errors during multiple attach/detach operations

Starting in Kubernetes version 1.9.2, when running multiple attach/detach operations in parallel, you may see the following disk issues due to a dirty VM cache:

- Disk attach/detach failures
- Disk I/O errors
- Unexpected disk detachment from VM
- VM running into failed state due to attaching non-existing disk

This issue has been fixed in the following versions of Kubernetes:

KUBERNETES VERSION	FIXED VERSION
1.10	1.10.12 or later
1.11	1.11.6 or later
1.12	1.12.4 or later
1.13	1.13.0 or later
1.14 and later	N/A

If you are using a version of Kubernetes that does not have the fix for this issue, you can mitigate the issue by trying the below:

- If there a disk is waiting to detach for a long period of time, try detaching the disk manually

## Azure Disk waiting to detach indefinitely

In some cases, if an Azure Disk detach operation fails on the first attempt, it will not retry the detach operation and will remain attached to the original node VM. This error can occur when moving a disk from one node to another.

For example:

```
[Warning] AttachVolume.Attach failed for volume "pvc-7b7976d7-3a46-11e9-93d5-dee1946e6ce9" : Attach volume "kubernetes-dynamic-pvc-7b7976d7-3a46-11e9-93d5-dee1946e6ce9" to instance "/subscriptions/XXX/resourceGroups/XXX/providers/Microsoft.Compute/virtualMachines/aks-agentpool-57634498-0" failed with compute.VirtualMachinesClient#CreateOrUpdate: Failure sending request: StatusCode=0 -- Original Error: autorest/azure: Service returned an error. Status= Code="ConflictingUserInput" Message="Disk '/subscriptions/XXX/resourceGroups/XXX/providers/Microsoft.Compute/disks/kubernetes-dynamic-pvc-7b7976d7-3a46-11e9-93d5-dee1946e6ce9' cannot be attached as the disk is already owned by VM '/subscriptions/XXX/resourceGroups/XXX/providers/Microsoft.Compute/virtualMachines/aks-agentpool-57634498-1'."
```

This issue has been fixed in the following versions of Kubernetes:

KUBERNETES VERSION	FIXED VERSION
1.11	1.11.9 or later
1.12	1.12.7 or later
1.13	1.13.4 or later
1.14 and later	N/A

If you are using a version of Kubernetes that does not have the fix for this issue, you can mitigate the issue by manually detaching the disk.

#### Azure Disk detach failure leading to potential race condition issue and invalid data disk list

When an Azure Disk fails to detach, it will retry up to six times to detach the disk using exponential back off. It will also hold a node-level lock on the data disk list for about 3 minutes. If the disk list is updated manually during that period of time, such as a manual attach or detach operation, this will cause the disk list held by the node-level lock to be obsolete and cause instability on the node VM.

This issue has been fixed in the following versions of Kubernetes:

KUBERNETES VERSION	FIXED VERSION
1.12	1.12.9 or later
1.13	1.13.6 or later
1.14	1.14.2 or later
1.15 and later	N/A

If you are using a version of Kubernetes that does not have the fix for this issue and your node VM has an obsolete disk list, you can mitigate the issue by detaching all non-existing disks from the VM as a single, bulk operation.

**Individually detaching non-existing disks may fail.**

#### Large number of Azure Disks causes slow attach/detach

When the number of Azure Disks attached to a node VM is larger than 10, attach and detach operations may be slow. This issue is a known issue and there are no workarounds at this time.

#### Azure Disk detach failure leading to potential node VM in failed state

In some edge cases, an Azure Disk detach may partially fail and leave the node VM in a failed state.

This issue has been fixed in the following versions of Kubernetes:

KUBERNETES VERSION	FIXED VERSION
1.12	1.12.10 or later
1.13	1.13.8 or later
1.14	1.14.4 or later
1.15 and later	N/A

If you are using a version of Kubernetes that does not have the fix for this issue and your node VM is in a failed state, you can mitigate the issue by manually updating the VM status using one of the below:

- For an availability set-based cluster:

```
az vm update -n <VM_NAME> -g <RESOURCE_GROUP_NAME>
```

- For a VMSS-based cluster:

```
az vmss update-instances -g <RESOURCE_GROUP_NAME> --name <VMSS_NAME> --instance-id <ID>
```

## Azure Files and AKS Troubleshooting

### What are the recommended stable versions of Kubernetes for Azure files?

KUBERNETES VERSION	RECOMMENDED VERSION
1.12	1.12.6 or later
1.13	1.13.4 or later
1.14	1.14.0 or later

### What versions of Kubernetes have Azure Files support on the Sovereign Cloud?

KUBERNETES VERSION	RECOMMENDED VERSION
1.12	1.12.0 or later
1.13	1.13.0 or later
1.14	1.14.0 or later

### What are the default mountOptions when using Azure Files?

Recommended settings:

KUBERNETES VERSION	FILEMODE AND DIRMODE VALUE
1.12.0 - 1.12.1	0755
1.12.2 and later	0777

If using a cluster with Kuberetes version 1.8.5 or greater and dynamically creating the persistent volume with a storage class, mount options can be specified on the storage class object. The following example sets 0777:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azurefile
provisioner: kubernetes.io/azure-file
mountOptions:
  - dir_mode=0777
  - file_mode=0777
  - uid=1000
  - gid=1000
  - mfsymlinks
  - nobrl
  - cache=none
parameters:
  skuName: Standard_LRS
```

Some additional useful *mountOptions* settings:

- *mfsymlinks* will make Azure Files mount (cifs) support symbolic links
- *nobrl* will prevent sending byte range lock requests to the server. This setting is necessary for certain applications that break with cifs style mandatory byte range locks. Most cifs servers do not yet support requesting advisory byte range locks. If not using *nobrl*, applications that break with cifs style mandatory byte range locks may cause error messages similar to:

```
Error: SQLITE_BUSY: database is locked
```

### Error "could not change permissions" when using Azure Files

When running PostgreSQL on the Azure Files plugin, you may see an error similar to:

```
initdb: could not change permissions of directory "/var/lib/postgresql/data": Operation not permitted
fixing permissions on existing directory /var/lib/postgresql/data
```

This error is caused by the Azure Files plugin using the cifs/SMB protocol. When using the cifs/SMB protocol, the file and directory permissions couldn't be changed after mounting.

To resolve this issue, use *subPath* together with the Azure Disk plugin.

#### NOTE

For ext3/4 disk type, there is a lost+found directory after the disk is formatted.

### Azure Files has high latency compared to Azure Disk when handling many small files

In some case, such as handling many small files, you may experience high latency when using Azure Files when compared to Azure Disk.

### Error when enabling "Allow access allow access from selected network" setting on storage account

If you enable *allow access from selected network* on a storage account that is used for dynamic provisioning in AKS, you will get an error when AKS creates a file share:

```
persistentvolume-controller (combined from similar events): Failed to provision volume with StorageClass "azurefile": failed to create share kubernetes-dynamic-pvc-xxx in account xxx: failed to create file share, err: storage: service returned error: StatusCode=403, ErrorCode=AuthorizationFailure, ErrorMessage=This request is not authorized to perform this operation.
```

This error is because of the Kubernetes `persistentvolume-controller` not being on the network chosen when setting `allow access from selected network`.

You can mitigate the issue by using [static provisioning with Azure Files](#).

### Azure Files fails to remount in Windows pod

If a Windows pod with an Azure Files mount is deleted and then scheduled to be recreated on the same node, the mount will fail. This failure is because of the `New-SmbGlobalMapping` command failing since the Azure Files mount is already mounted on the node.

For example, you may see an error similar to:

```
E0118 08:15:52.041014    2112 nestedpendingoperations.go:267] Operation for "\"kubernetes.io/azure-file/42c0ea39-1af9-11e9-8941-000d3af95268-pvc-d7e1b5f9-1af3-11e9-8941-000d3af95268\" ("42c0ea39-1af9-11e9-8941-000d3af95268")" failed. No retries permitted until 2019-01-18 08:15:53.0410149 +0000 GMT m+=+732.446642701 (durationBeforeRetry 1s). Error: "MountVolume.SetUp failed for volume \"pvc-d7e1b5f9-1af3-11e9-8941-000d3af95268\" (UniqueName: \"kubernetes.io/azure-file/42c0ea39-1af9-11e9-8941-000d3af95268-pvc-d7e1b5f9-1af3-11e9-8941-000d3af95268\") pod \"deployment-azurefile-697f98d559-6zrlf\" (UID: \"42c0ea39-1af9-11e9-8941-000d3af95268\") : azureMount: SmbGlobalMapping failed: exit status 1, only SMB mount is supported now, output: \"New-SmbGlobalMapping : Generic failure \\r\\nAt line:1 char:190\\r\\n+ ... ser, $PWord;New-SmbGlobalMapping - RemotePath $Env:smbremotepath -Cred ...\\r\\n+ ~~~~~ + CategoryInfo          : NotSpecified: (MSFT_SmbGlobalMapping:ROOT/Microsoft/...mbGlobalMapping) [New-SmbGlobalMapping\\r\\n      ], CimException\\r\\n + FullyQualifiedErrorId : HRESULT 0x80041001,New-SmbGlobalMapping\\r\\n      \\r\\n      \""
```

This issue has been fixed in the following versions of Kubernetes:

KUBERNETES VERSION	FIXED VERSION
1.12	1.12.6 or later
1.13	1.13.4 or later
1.14 and later	N/A

### Azure Files mount fails due to storage account key changed

If your storage account key has changed, you may see Azure Files mount failures.

You can mitigate the issue by doing manually updating the `azurestorageaccountkey` field manually in Azure file secret with your base64-encoded storage account key.

To encode your storage account key in base64, you can use `base64`. For example:

```
echo X+ALAAUgMhWHL7QmQ87E1kSF1qLKfgC03Guy7/xk9MyIg2w4Jzqe60CVw2r/dm6v6E0DWHTnJUEJGVQAoPaBc== | base64
```

To update your Azure secret file, use `kubectl edit secret`. For example:

```
kubectl edit secret azure-storage-account-{storage-account-name}-secret
```

After a few minutes, the agent node will retry the azure file mount with the updated storage key.

### **Cluster autoscaler fails to scale with error failed to fix node group sizes**

If your cluster autoscaler is not scaling up/down and you see an error like the below on the [cluster autoscaler logs](#).

```
E1114 09:58:55.367731 1 static_autoscaler.go:239] Failed to fix node group sizes: failed to decrease aks-default-35246781-vmss: attempt to delete existing nodes
```

This error is due to an upstream cluster autoscaler race condition where the cluster autoscaler ends with a different value than the one that is actually in the cluster. To get out of this state, simply disable and re-enable the [cluster autoscaler](#).

# Checking for Kubernetes best practices in your cluster

2/25/2020 • 2 minutes to read • [Edit Online](#)

There are several best practices that you should follow on your Kubernetes deployments to ensure the best performance and resilience for your applications. You can use the kube-advisor tool to look for deployments that aren't following those suggestions.

## About kube-advisor

The [kube-advisor tool](#) is a single container designed to be run on your cluster. It queries the Kubernetes API server for information about your deployments and returns a set of suggested improvements.

The kube-advisor tool can report on resource request and limits missing in PodSpecs for Windows applications as well as Linux applications, but the kube-advisor tool itself must be scheduled on a Linux pod. You can schedule a pod to run on a node pool with a specific OS using a [node selector](#) in the pod's configuration.

### NOTE

The kube-advisor tool is supported by Microsoft on a best-effort basis. Issues and suggestions should be filed on GitHub.

## Running kube-advisor

To run the tool on a cluster that is configured for [role-based access control \(RBAC\)](#), using the following commands. The first command creates a Kubernetes service account. The second command runs the tool in a pod using that service account and configures the pod for deletion after it exits.

```
kubectl apply -f https://raw.githubusercontent.com/Azure/kube-advisor/master/sa.yaml  
kubectl run --rm -i -t kubeadvisor --image=mcr.microsoft.com/aks/kubeadvisor --restart=Never --overrides="{"  
  \"apiVersion\": \"v1\", \"spec\": { \"serviceAccountName\": \"kube-advisor\" } }"
```

If you aren't using RBAC, you can run the command as follows:

```
kubectl run --rm -i -t kubeadvisor --image=mcr.microsoft.com/aks/kubeadvisor --restart=Never
```

Within a few seconds, you should see a table describing potential improvements to your deployments.

NAMESPACE	POD NAME	POD CPU/MEMORY	CONTAINER	ISSUE
postgresdemo	postgresdemo-worker-0	1m / 9440Ki	postgresdemo-worker	CPU Resource Limits Missing Memory Resource Limits Missing
	postgresdemo-worker-1	1m / 9152Ki		CPU Resource Limits Missing

## Checks performed

The tool validates several Kubernetes best practices, each with their own suggested remediation.

### Resource requests and limits

Kubernetes supports defining [resource requests and limits on pod specifications](#). The request defines the

minimum CPU and memory required to run the container. The limit defines the maximum CPU and memory that should be allowed.

By default, no requests or limits are set on pod specifications. This can lead to nodes being overscheduled and containers being starved. The kube-advisor tool highlights pods without requests and limits set.

## Cleaning up

If your cluster has RBAC enabled, you can clean up the `clusterRoleBinding` after you've run the tool using the following command:

```
kubectl delete -f https://raw.githubusercontent.com/Azure/kube-advisor/master/sa.yaml
```

If you are running the tool against a cluster that is not RBAC-enabled, no cleanup is required.

## Next steps

- [Troubleshoot issues with Azure Kubernetes Service](#)

# Connect with SSH to Azure Kubernetes Service (AKS) cluster nodes for maintenance or troubleshooting

2/25/2020 • 7 minutes to read • [Edit Online](#)

Throughout the lifecycle of your Azure Kubernetes Service (AKS) cluster, you may need to access an AKS node. This access could be for maintenance, log collection, or other troubleshooting operations. You can access AKS nodes using SSH, including Windows Server nodes (currently in preview in AKS). You can also [connect to Windows Server nodes using remote desktop protocol \(RDP\) connections](#). For security purposes, the AKS nodes aren't exposed to the internet. To SSH to the AKS nodes, you use the private IP address.

This article shows you how to create an SSH connection with an AKS node using their private IP addresses.

## Before you begin

This article assumes that you have an existing AKS cluster. If you need an AKS cluster, see the AKS quickstart [using the Azure CLI](#) or [using the Azure portal](#).

By default, SSH keys are obtained, or generated, then added to nodes when you create an AKS cluster. This article shows you how to specify different SSH keys than the SSH keys used when you created your AKS cluster. The article also shows you how to determine the private IP address of your node and connect to it using SSH. If you don't need to specify a different SSH key, then you may skip the step for adding the SSH public key to the node.

This article also assumes you have an SSH key. You can create an SSH key using [macOS or Linux](#) or [Windows](#). If you use PuTTY Gen to create the key pair, save the key pair in an OpenSSH format rather than the default PuTTY private key format (.ppk file).

You also need the Azure CLI version 2.0.64 or later installed and configured. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Configure virtual machine scale set-based AKS clusters for SSH access

To configure your virtual machine scale set-based for SSH access, find the name of your cluster's virtual machine scale set and add your SSH public key to that scale set.

Use the `az aks show` command to get the resource group name of your AKS cluster, then the `az vmss list` command to get the name of your scale set.

```
CLUSTER_RESOURCE_GROUP=$(az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv)
SCALE_SET_NAME=$(az vmss list --resource-group $CLUSTER_RESOURCE_GROUP --query [0].name -o tsv)
```

The above example assigns the name of the cluster resource group for the *myAKSCluster* in *myResourceGroup* to *CLUSTER\_RESOURCE\_GROUP*. The example then uses *CLUSTER\_RESOURCE\_GROUP* to list the scale set name and assign it to *SCALE\_SET\_NAME*.

## IMPORTANT

At this time, you should only update your SSH keys for your virtual machine scale set-based AKS clusters using the Azure CLI.

For Linux nodes, SSH keys can currently only be added using the Azure CLI. If you want to connect to a Windows Server node using SSH, use the SSH keys provided when you created the AKS cluster and skip the next set of commands for adding your SSH public key. You will still need the IP address of the node you wish to troubleshoot, which is shown in the final command of this section. Alternatively, you can [connect to Windows Server nodes using remote desktop protocol \(RDP\) connections](#) instead of using SSH.

To add your SSH keys to the nodes in a virtual machine scale set, use the [az vmss extension set](#) and [az vmss update-instances](#) commands.

```
az vmss extension set \
--resource-group $CLUSTER_RESOURCE_GROUP \
--vmss-name $SCALE_SET_NAME \
--name VMAccessForLinux \
--publisher Microsoft.OSTCExtensions \
--version 1.4 \
--protected-settings "{\"username\":\"azureuser\", \"ssh_key\":$(cat ~/.ssh/id_rsa.pub)}"
```

```
az vmss update-instances --instance-ids '*' \
--resource-group $CLUSTER_RESOURCE_GROUP \
--name $SCALE_SET_NAME
```

The above example uses the `CLUSTER_RESOURCE_GROUP` and `SCALE_SET_NAME` variables from the previous commands. The above example also uses `~/.ssh/id_rsa.pub` as the location for your SSH public key.

## NOTE

By default, the username for the AKS nodes is `azureuser`.

After you add your SSH public key to the scale set, you can SSH into a node virtual machine in that scale set using its IP address. View the private IP addresses of the AKS cluster nodes using the [kubectl get command](#).

```
kubectl get nodes -o wide
```

The follow example output shows the internal IP addresses of all the nodes in the cluster, including a Windows Server node.

```
$ kubectl get nodes -o wide
```

NAME	KERNEL-VERSION	CONTAINER-RUNTIME	STATUS	ROLES	AGE	VERSION	INTERNAL-IP	EXTERNAL-IP	OS-IMAGE
aks-nodepool11-42485177-vmss000000	16.04.6 LTS	4.15.0-1040-azure	Ready	agent	18h	v1.12.7	10.240.0.4	<none>	Ubuntu
aksnpwin000000	Server Datacenter		Ready	agent	13h	v1.12.7	10.240.0.67	<none>	Windows

Record the internal IP address of the node you wish to troubleshoot.

To access your node using SSH, follow the steps in [Create the SSH connection](#).

## Configure virtual machine availability set-based AKS clusters for SSH

## access

To configure your virtual machine availability set-based AKS cluster for SSH access, find the name of your cluster's Linux node, and add your SSH public key to that node.

Use the [az aks show](#) command to get the resource group name of your AKS cluster, then the [az vm list](#) command to list the virtual machine name of your cluster's Linux node.

```
CLUSTER_RESOURCE_GROUP=$(az aks show --resource-group myResourceGroup --name myAKSCluster --query nodeResourceGroup -o tsv)
az vm list --resource-group $CLUSTER_RESOURCE_GROUP -o table
```

The above example assigns the name of the cluster resource group for the *myAKSCluster* in *myResourceGroup* to *CLUSTER\_RESOURCE\_GROUP*. The example then uses *CLUSTER\_RESOURCE\_GROUP* to list the virtual machine name. The example output shows the name of the virtual machine:

Name	ResourceGroup	Location
aks-nodepool1-79590246-0	MC_myResourceGroupAKS_myAKSClusterRBAC_eastus	eastus

To add your SSH keys to the node, use the [az vm user update](#) command.

```
az vm user update \
--resource-group $CLUSTER_RESOURCE_GROUP \
--name aks-nodepool1-79590246-0 \
--username azureuser \
--ssh-key-value ~/.ssh/id_rsa.pub
```

The above example uses the *CLUSTER\_RESOURCE\_GROUP* variable and the node virtual machine name from previous commands. The above example also uses *~/.ssh/id\_rsa.pub* as the location for your SSH public key. You could also use the contents of your SSH public key instead of specifying a path.

### NOTE

By default, the username for the AKS nodes is *azureuser*.

After you add your SSH public key to the node virtual machine, you can SSH into that virtual machine using its IP address. View the private IP address of an AKS cluster node using the [az vm list-ip-addresses](#) command.

```
az vm list-ip-addresses --resource-group $CLUSTER_RESOURCE_GROUP -o table
```

The above example uses the *CLUSTER\_RESOURCE\_GROUP* variable set in the previous commands. The following example output shows the private IP addresses of the AKS nodes:

VirtualMachine	PrivateIPAddresses
aks-nodepool1-79590246-0	10.240.0.4

## Create the SSH connection

To create an SSH connection to an AKS node, you run a helper pod in your AKS cluster. This helper pod provides you with SSH access into the cluster and then additional SSH node access. To create and use this helper pod,

complete the following steps:

1. Run a `debian` container image and attach a terminal session to it. This container can be used to create an SSH session with any node in the AKS cluster:

```
kubectl run --generator=run-pod/v1 -it --rm aks-ssh --image=debian
```

#### TIP

If you use Windows Server nodes (currently in preview in AKS), add a node selector to the command to schedule the Debian container on a Linux node:

```
kubectl run -it --rm aks-ssh --image=debian --overrides='{"apiVersion": "apps/v1", "spec": {"template": {"spec": {"nodeSelector": {"beta.kubernetes.io/os": "linux"} }}}}'
```

2. Once the terminal session is connected to the container, install an SSH client using `apt-get`:

```
apt-get update && apt-get install openssh-client -y
```

3. Open a new terminal window, not connected to your container, copy your private SSH key into the helper pod. This private key is used to create the SSH into the AKS node.

If needed, change `~/.ssh/id_rsa` to location of your private SSH key:

```
kubectl cp ~/.ssh/id_rsa $(kubectl get pod -l run=aks-ssh -o jsonpath='{.items[0].metadata.name}'):/id_rsa
```

4. Return to the terminal session to your container, update the permissions on the copied `id_rsa` private SSH key so that it is user read-only:

```
chmod 0600 id_rsa
```

5. Create an SSH connection to your AKS node. Again, the default username for AKS nodes is `azureuser`. Accept the prompt to continue with the connection as the SSH key is first trusted. You are then provided with the bash prompt of your AKS node:

```
$ ssh -i id_rsa azureuser@10.240.0.4

ECDSA key fingerprint is SHA256:A6rnRkfpG21TaZ8XmQCCgdi9G/MYIMc+gFAuY9RUY70.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '10.240.0.4' (ECDSA) to the list of known hosts.

Welcome to Ubuntu 16.04.5 LTS (GNU/Linux 4.15.0-1018-azure x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

Get cloud support with Ubuntu Advantage Cloud Guest:
https://www.ubuntu.com/business/services/cloud

[...]

azureuser@aks-nodepool1-79590246-0:~$
```

## Remove SSH access

When done, `exit` the SSH session and then `exit` the interactive container session. When this container session closes, the pod used for SSH access from the AKS cluster is deleted.

## Next steps

If you need additional troubleshooting data, you can [view the kubelet logs](#) or [view the Kubernetes master node logs](#).

# Linux Performance Troubleshooting

2/27/2020 • 11 minutes to read • [Edit Online](#)

Resource exhaustion on Linux machines is a common issue and can manifest through a wide variety of symptoms. This document provides a high-level overview of the tools available to help diagnose such issues.

Many of these tools accept an interval on which to produce rolling output. This output format typically makes spotting patterns much easier. Where accepted, the example invocation will include `[interval]`.

Many of these tools have an extensive history and wide set of configuration options. This page provides only a simple subset of invocations to highlight common problems. The canonical source of information is always the reference documentation for each particular tool. That documentation will be much more thorough than what is provided here.

## Guidance

Be systematic in your approach to investigating performance issues. Two common approaches are USE (utilization, saturation, errors) and RED (rate, errors, duration). RED is typically used in the context of services for request-based monitoring. USE is typically used for monitoring resources: for each resource in a machine, monitor utilization, saturation, and errors. The four main kinds of resources on any machine are cpu, memory, disk, and network. High utilization, saturation, or error rates for any of these resources indicates a possible problem with the system. When a problem exists, investigate the root cause: why is disk IO latency high? Are the disks or virtual machine SKU throttled? What processes are writing to the devices, and to what files?

Some examples of common issues and indicators to diagnose them:

- IOPS throttling: use iostat to measure per-device IOPS. Ensure no individual disk is above its limit, and the sum for all disks is less than the limit for the virtual machine.
- Bandwidth throttling: use iostat as for IOPS, but measuring read/write throughput. Ensure both per-device and aggregate throughput are below the bandwidth limits.
- SNAT exhaustion: this can manifest as high active (outbound) connections in SAR.
- Packet loss: this can be measured by proxy via TCP retransmit count relative to sent/received count. Both `sar` and `netstat` can show this information.

## General

These tools are general purpose and cover basic system information. They are a good starting point for further investigation.

### `uptime`

```
$ uptime  
19:32:33 up 17 days, 12:36,  0 users,  load average: 0.21, 0.77, 0.69
```

`uptime` provides system uptime and 1, 5, and 15-minute load averages. These load averages roughly correspond to threads doing work or waiting for uninterruptible work to complete. In absolute these numbers can be difficult to interpret, but measured over time they can tell us useful information:

- 1-minute average > 5-minute average means load is increasing.
- 1-minute average < 5-minute average means load is decreasing.

uptime can also illuminate why information is not available: the issue may have resolved on its own or by a restart before the user could access the machine.

Load averages higher than the number of CPU threads available may indicate a performance issue with a given workload.

## dmesg

```
$ dmesg | tail  
$ dmesg --level=err | tail
```

dmesg dumps the kernel buffer. Events like OOMKill add an entry to the kernel buffer. Finding an OOMKill or other resource exhaustion messages in dmesg logs is a strong indicator of a problem.

## top

```
$ top  
Tasks: 249 total, 1 running, 158 sleeping, 0 stopped, 0 zombie  
%Cpu(s): 2.2 us, 1.3 sy, 0.0 ni, 95.4 id, 1.0 wa, 0.0 hi, 0.2 si, 0.0 st  
KiB Mem : 65949064 total, 43415136 free, 2349328 used, 20184600 buff/cache  
KiB Swap: 0 total, 0 free, 0 used. 62739060 avail Mem  
  
 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND  
116004 root      20   0 144400  41124  27028 S 11.8  0.1 248:45.45 coredns  
  4503 root      20   0 1677980 167964  89464 S  5.9  0.3 1326:25 kubelet  
     1 root      20   0 120212  6404   4044 S  0.0  0.0  48:20.38 systemd  
    ...
```

`top` provides a broad overview of current system state. The headers provide some useful aggregate information:

- state of tasks: running, sleeping, stopped.
- CPU utilization, in this case mostly showing idle time.
- total, free, and used system memory.

`top` may miss short-lived processes; alternatives like `htop` and `atop` provide similar interfaces while fixing some of these shortcomings.

## CPU

These tools provide CPU utilization information. This is especially useful with rolling output, where patterns become easy to spot.

## mpstat

```
$ mpstat -P ALL [interval]  
Linux 4.15.0-1064-azure (aks-main-10212767-vmss000001) 02/10/20          _x86_64_          (8 CPU)  
  
19:49:03    CPU    %usr    %nice    %sys %iowait    %irq    %soft    %steal    %guest    %gnice    %idle  
19:49:04    all    1.01    0.00    0.63    2.14    0.00    0.13    0.00    0.00    0.00    0.00    96.11  
19:49:04      0    1.01    0.00    1.01   17.17    0.00    0.00    0.00    0.00    0.00    0.00    80.81  
19:49:04      1    1.98    0.00    0.99    0.00    0.00    0.00    0.00    0.00    0.00    0.00    97.03  
19:49:04      2    1.01    0.00    0.00    0.00    0.00    1.01    0.00    0.00    0.00    0.00    97.98  
19:49:04      3    0.00    0.00    0.99    0.00    0.00    0.99    0.00    0.00    0.00    0.00    98.02  
19:49:04      4    1.98    0.00    1.98    0.00    0.00    0.00    0.00    0.00    0.00    0.00    96.04  
19:49:04      5    1.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    98.00  
19:49:04      6    1.00    0.00    1.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    98.00  
19:49:04      7    1.98    0.00    0.99    0.00    0.00    0.00    0.00    0.00    0.00    0.00    97.03
```

`mpstat` prints similar CPU information to `top`, but broken down by CPU thread. Seeing all cores at once can be

useful for detecting highly imbalanced CPU usage, for example when a single threaded application uses one core at 100% utilization. This problem may be more difficult to spot when aggregated over all CPUs in the system.

## vmstat

```
$ vmstat [interval]
procs -----memory----- --swap-- -----io---- -system-- -----cpu-----
r b swpd free buff cache si so bi bo in cs us sy id wa st
2 0      0 43300372 545716 19691456   0   0    3   50   3   3 2 1 95 1 0
```

`vmstat` provides similar information `mpstat` and `top`, enumerating number of processes waiting on CPU (r column), memory statistics, and percent of CPU time spent in each work state.

## Memory

Memory is a very important, and thankfully easy, resource to track. Some tools can report both CPU and memory, like `vmstat`. But tools like `free` may still be useful for quick debugging.

## free

```
$ free -m
              total        used        free      shared  buff/cache   available
Mem:       64403         2338       42485           1        19579       61223
Swap:          0           0           0
```

`free` presents basic information about total memory as well as used and free memory. `vmstat` may be more useful even for basic memory analysis due to its ability to provide rolling output.

## Disk

These tools measure disk IOPS, wait queues, and total throughput.

## iostat

```
$ iostat -xy [interval] [count]
$ iostat -xy 1 1
Linux 4.15.0-1064-azure (aks-main-10212767-vmss000001) 02/10/20      _x86_64_      (8 CPU)

avg-cpu: %user  %nice %system %iowait  %steal  %idle
          3.42   0.00   2.92   1.90   0.00  91.76

Device:    rrqm/s   wrqm/s     r/s     w/s   rkB/s   wkB/s  avgrrq-sz avgqu-sz   await r_await w_await
svctm  %util
loop0      0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
0.00  0.00
sdb       0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
0.00  0.00
sda       0.00    56.00     0.00    65.00     0.00   504.00    15.51     0.01    3.02     0.00     3.02
0.12  0.80
scd0      0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
0.00  0.00
```

`iostat` provides deep insights into disk utilization. This invocation passes `-x` for extended statistics, `-y` to skip the initial output printing system averages since boot, and `1 1` to specify we want 1-second interval, ending after one block of output.

`iostat` exposes many useful statistics:

- `r/s` and `w/s` are reads per second and writes per second. The sum of these values is IOPS.

- `rkB/s` and `wkB/s` are kilobytes read/written per second. The sum of these values is throughput.
- `await` is the average iowait time in milliseconds for queued requests.
- `avgqu-sz` is the average queue size over the provided interval.

On an Azure VM:

- the sum of `r/s` and `w/s` for an individual block device may not exceed that disk's SKU limits.
- the sum of `rkB/s` and `wkB/s` for an individual block device may not exceed that disk's SKU limits
- the sum of `r/s` and `w/s` for all block devices may not exceed the limits for the VM SKU.
- the sum of `rkB/s` and `wkB/s` for all block devices may not exceed the limits for the VM SKU.

Note that the OS disk counts as a managed disk of the smallest SKU corresponding to its capacity. For example, a 1024GB OS Disk corresponds to a P30 disk. Ephemeral OS disks and temporary disks do not have individual disk limits; they are only limited by the full VM limits.

Non-zero values of `await` or `avgqu-sz` are also good indicators of IO contention.

## Network

These tools measure network statistics like throughput, transmission failures, and utilization. Deeper analysis can expose fine-grained TCP statistics about congestion and dropped packets.

### `sar`

```
$ sar -n DEV [interval]
22:36:57      IFACE    rxpck/s    txpck/s    rxkB/s    txkB/s    rxcmp/s    txcmp/s    rxmcst/s    %ifutil
22:36:58    docker0     0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
22:36:58    azv604be75d832   1.00    9.00     0.06     1.04     0.00     0.00     0.00     0.00
22:36:58    azure0     68.00   79.00    27.79    52.79     0.00     0.00     0.00     0.00
22:36:58    azv4a8e7704a5b   202.00  207.00    37.51    21.86     0.00     0.00     0.00     0.00
22:36:58    azve83c28f6d1c   21.00   30.00    24.12     4.11     0.00     0.00     0.00     0.00
22:36:58    eth0      314.00  321.00    70.87   163.28     0.00     0.00     0.00     0.00
22:36:58    azva3128390bff   12.00   20.00     1.14     2.29     0.00     0.00     0.00     0.00
22:36:58    azvf46c95dde4   10.00   18.00    31.47     1.36     0.00     0.00     0.00     0.00
22:36:58    enP1s1      74.00  374.00    29.36   166.94     0.00     0.00     0.00     0.00
22:36:58    lo        0.00     0.00     0.00     0.00     0.00     0.00     0.00     0.00
22:36:58    azvdbf16b0b2fc   9.00   19.00     3.36     1.18     0.00     0.00     0.00     0.00
```

`sar` is a powerful tool for a wide range of analysis. While this example uses its ability to measure network stats, it is equally powerful for measuring CPU and memory consumption. This example invokes `sar` with `-n` flag to specify the `DEV` (network device) keyword, displaying network throughput by device.

- The sum of `rxKb/s` and `txKb/s` is total throughput for a given device. When this value exceeds the limit for the provisioned Azure NIC, workloads on the machine will experience increased network latency.
- `%ifutil` measures utilization for a given device. As this value approaches 100%, workloads will experience increased network latency.

```
$ sar -n TCP,ETCP [interval]
Linux 4.15.0-1064-azure (aks-main-10212767-vmss000001) 02/10/20          _x86_64_      (8 CPU)

22:50:08    active/s passive/s    iseg/s    oseg/s
22:50:09        2.00       0.00     19.00     24.00

22:50:08    atmptf/s estres/s retrans/s isegerr/s   orsts/s
22:50:09        0.00       0.00       0.00       0.00       0.00

Average:    active/s passive/s    iseg/s    oseg/s
Average:        2.00       0.00     19.00     24.00

Average:    atmptf/s estres/s retrans/s isegerr/s   orsts/s
Average:        0.00       0.00       0.00       0.00       0.00
```

This invocation of `sar` uses the `TCP,ETCP` keywords to examine TCP connections. The third column of the last row, "retrans", is the number of TCP retransmits per second. High values for this field indicate an unreliable network connection. In The first and third rows, "active" means a connection originated from the local device, while "remote" indicates an incoming connection. A common issue on Azure is SNAT port exhaustion, which `sar` can help detect. SNAT port exhaustion would manifest as high "active" values, since the problem is due to a high rate of outbound, locally-initiated TCP connections.

As `sar` takes an interval, it prints rolling output and then prints final rows of output containing the average results from the invocation.

## netstat

```
$ netstat -s
Ip:
    71046295 total packets received
    78 forwarded
    0 incoming packets discarded
    71046066 incoming packets delivered
    83774622 requests sent out
    40 outgoing packets dropped
Icmp:
    103 ICMP messages received
    0 input ICMP message failed.
    ICMP input histogram:
        destination unreachable: 103
    412802 ICMP messages sent
    0 ICMP messages failed
    ICMP output histogram:
        destination unreachable: 412802
IcmpMsg:
    InType3: 103
    OutType3: 412802
Tcp:
    11487089 active connections openings
    592 passive connection openings
    1137 failed connection attempts
    404 connection resets received
    17 connections established
    70880911 segments received
    95242567 segments send out
    176658 segments retransmited
    3 bad segments received.
    163295 resets sent
Udp:
    164968 packets received
    84 packets to unknown port received.
    0 packet receive errors
    165082 packets sent
UdpLite:
```

```

TcpExt:
    5 resets received for embryonic SYN_RECV sockets
    1670559 TCP sockets finished time wait in fast timer
    95 packets rejects in established connections because of timestamp
    756870 delayed acks sent
    2236 delayed acks further delayed because of locked socket
    Quick ack mode was activated 479 times
    11983969 packet headers predicted
    25061447 acknowledgments not containing data payload received
    5596263 predicted acknowledgments
    19 times recovered from packet loss by selective acknowledgements
    Detected reordering 114 times using SACK
    Detected reordering 4 times using time stamp
    5 congestion windows fully recovered without slow start
    1 congestion windows partially recovered using Hoe heuristic
    5 congestion windows recovered without slow start by DSACK
    111 congestion windows recovered without slow start after partial ack
    73 fast retransmits
    26 retransmits in slow start
    311 other TCP timeouts
    TCPLossProbes: 198845
    TCPLossProbeRecovery: 147
    480 DSACKs sent for old packets
    175310 DSACKs received
    316 connections reset due to unexpected data
    272 connections reset due to early user close
    5 connections aborted due to timeout
    TCPDSACKIgnoredNoUndo: 8498
    TCPSpuriousRTT0s: 1
    TCPSackShifted: 3
    TCPSackMerged: 9
    TCPSackShiftFallback: 177
    IPReversePathFilter: 4
    TCPRcvCoalesce: 1501457
    TCPOFQQueue: 9898
    TCPChallengeACK: 342
    TCPSYNChallenge: 3
    TCPSpuriousRtxHostQueues: 17
    TCPAutoCorking: 2315642
    TCPFromZeroWindowAdv: 483
    TCPToZeroWindowAdv: 483
    TCPWantZeroWindowAdv: 115
    TCPSynRetrans: 885
    TCPOrigDataSent: 51140171
    TCPHystartTrainDetect: 349
    TCPHystartTrainCwnd: 7045
    TCPHystartDelayDetect: 26
    TCPHystartDelayCwnd: 862
    TCPACKSkippedPAWS: 3
    TCPACKSkippedSeq: 4
    TCPKeepAlive: 62517

IpExt:
    InOctets: 36416951539
    OutOctets: 41520580596
    InNoECTPkts: 86631440
    InECT0Pkts: 14

```

`netstat` can introspect a wide variety of network stats, here invoked with summary output. There are many useful fields here depending on the issue. One useful field in the TCP section is "failed connection attempts". This may be an indication of SNAT port exhaustion or other issues making outbound connections. A high rate of retransmitted segments (also under the TCP section) may indicate issues with packet delivery.

# Support policies for Azure Kubernetes Service

2/25/2020 • 10 minutes to read • [Edit Online](#)

This article provides details about technical support policies and limitations for Azure Kubernetes Service (AKS). The article also details worker node management, managed control plane components, third-party open-source components, and security or patch management.

## Service updates and releases

- For release information, see [AKS release notes](#).
- For information on features in preview, see [AKS preview features and related projects](#).

## Managed features in AKS

Base infrastructure as a service (IaaS) cloud components, such as compute or networking components, give users access to low-level controls and customization options. By contrast, AKS provides a turnkey Kubernetes deployment that gives customers the common set of configurations and capabilities they need. AKS customers have limited customization, deployment, and other options. These customers don't need to worry about or manage Kubernetes clusters directly.

With AKS, the customer gets a fully managed *control plane*. The control plane contains all of the components and services the customer needs to operate and provide Kubernetes clusters to end users. All Kubernetes components are maintained and operated by Microsoft.

Microsoft manages and monitors the following components through the control pane:

- Kubelet or Kubernetes API servers
- Etcd or a compatible key-value store, providing Quality of Service (QoS), scalability, and runtime
- DNS services (for example, kube-dns or CoreDNS)
- Kubernetes proxy or networking

AKS isn't a completely managed cluster solution. Some components, such as worker nodes, have *shared responsibility*, where users must help maintain the AKS cluster. User input is required, for example, to apply a worker node operating system (OS) security patch.

The services are *managed* in the sense that Microsoft and the AKS team deploys, operates, and is responsible for service availability and functionality. Customers can't alter these managed components. Microsoft limits customization to ensure a consistent and scalable user experience. For a fully customizable solution, see [AKS Engine](#).

### NOTE

AKS worker nodes appear in the Azure portal as regular Azure IaaS resources. But these virtual machines are deployed into a custom Azure resource group (prefixed with MC\\*). It's possible to change AKS worker nodes. For example, you can use Secure Shell (SSH) to change AKS worker nodes the way you change normal virtual machines (you can't, however, change the base OS image, and changes might not persist through an update or reboot), and you can attach other Azure resources to AKS worker nodes. But when you make changes *out of band management and customization*, the AKS cluster can become unsupported. Avoid changing worker nodes unless Microsoft Support directs you to make changes.

Issuing unsupported operations as defined above, such as out of band deallocation of all agent nodes, renders the cluster unsupported. AKS reserves the right to archive control planes that have been configured out of

support guidelines for extended periods equal to and beyond 30 days. AKS maintains backups of cluster etcd metadata and can readily reallocate the cluster. This reallocation can be initiated by any PUT operation bringing the cluster back into support, such as an upgrade or scale to active agent nodes.

## Shared responsibility

When a cluster is created, the customer defines the Kubernetes worker nodes that AKS creates. Customer workloads are executed on these nodes. Customers own and can view or modify the worker nodes.

Because customer cluster nodes execute private code and store sensitive data, Microsoft Support can access them in only a limited way. Microsoft Support can't sign in to, execute commands in, or view logs for these nodes without express customer permission or assistance.

Because worker nodes are sensitive, Microsoft takes great care to limit their background management. In many cases, your workload will continue to run even if the Kubernetes master nodes, etcd, and other Microsoft-managed components fail. Carelessly modified worker nodes can cause losses of data and workloads and can render the cluster unsupportable.

## AKS support coverage

Microsoft provides technical support for the following:

- Connectivity to all Kubernetes components that the Kubernetes service provides and supports, such as the API server.
- Management, uptime, QoS, and operations of Kubernetes control plane services (Kubernetes master nodes, API server, etcd, and kube-dns, for example).
- Etcd. Support includes automated, transparent backups of all etcd data every 30 minutes for disaster planning and cluster state restoration. These backups aren't directly available to customers or users. They ensure data reliability and consistency.
- Any integration points in the Azure cloud provider driver for Kubernetes. These include integrations into other Azure services such as load balancers, persistent volumes, or networking (Kubernetes and Azure CNI).
- Questions or issues about customization of control plane components such as the Kubernetes API server, etcd, and kube-dns.
- Issues about networking, such as Azure CNI, kubenet, or other network access and functionality issues. Issues could include DNS resolution, packet loss, routing, and so on. Microsoft supports various networking scenarios:
  - Kubenet (basic) and advanced networking (Azure CNI) within the cluster and associated components
  - Connectivity to other Azure services and applications
  - Ingress controllers and ingress or load balancer configurations
  - Network performance and latency

Microsoft doesn't provide technical support for the following:

- Questions about how to use Kubernetes. For example, Microsoft Support doesn't provide advice on how to create custom ingress controllers, use application workloads, or apply third-party or open-source software packages or tools.

### NOTE

Microsoft Support can advise on AKS cluster functionality, customization, and tuning (for example, Kubernetes operations issues and procedures).

- Third-party open-source projects that aren't provided as part of the Kubernetes control plane or deployed with AKS clusters. These projects might include Istio, Helm, Envoy, or others.

**NOTE**

Microsoft can provide best-effort support for third-party open-source projects such as Helm and Kured. Where the third-party open-source tool integrates with the Kubernetes Azure cloud provider or other AKS-specific bugs, Microsoft supports examples and applications from Microsoft documentation.

- Third-party closed-source software. This software can include security scanning tools and networking devices or software.
- Issues about multicloud or multivendor build-outs. For example, Microsoft doesn't support issues related to running a federated multipublic cloud vendor solution.
- Network customizations other than those listed in the [AKS documentation](#).

**NOTE**

Microsoft does support issues and bugs related to network security groups (NSGs). For example, Microsoft Support can answer questions about an NSG failure to update or an unexpected NSG or load balancer behavior.

## AKS support coverage for worker nodes

### Microsoft responsibilities for AKS worker nodes

Microsoft and customers share responsibility for Kubernetes worker nodes where:

- The base OS image has required additions (such as monitoring and networking agents).
- The worker nodes receive OS patches automatically.
- Issues with the Kubernetes control plane components that run on the worker nodes are automatically remediated. Components include the following:
  - Kube-proxy
  - Networking tunnels that provide communication paths to the Kubernetes master components
  - Kubelet
  - Docker or Moby daemon

**NOTE**

On a worker node, if a control plane component is not operational, the AKS team might need to reboot individual components or the entire worker node. These reboot operations are automated and provide auto-remediation for common issues. These reboots occur only on the *node* level and not the cluster unless there is an emergency maintenance or outage.

### Customer responsibilities for AKS worker nodes

Microsoft doesn't automatically reboot worker nodes to apply OS-level patches. Although OS patches are delivered to the worker nodes, the *customer* is responsible for rebooting the worker nodes to apply the changes. Shared libraries, daemons such as solid-state hybrid drive (SSHD), and other components at the level of the system or OS are automatically patched.

Customers are responsible for executing Kubernetes upgrades. They can execute upgrades through the Azure control panel or the Azure CLI. This applies for updates that contain security or functionality improvements to Kubernetes.

**NOTE**

Because AKS is a *managed service*, its end goals include removing responsibility for patches, updates, and log collection to make the service management more complete and hands-off. As the service's capacity for end-to-end management increases, future releases might omit some functions (for example, node rebooting and automatic patching).

## Security issues and patching

If a security flaw is found in one or more components of AKS, the AKS team will patch all affected clusters to mitigate the issue. Alternatively, the team will give users upgrade guidance.

For worker nodes that a security flaw affects, if a zero-downtime patch is available, the AKS team will apply that patch and notify users of the change.

When a security patch requires worker node reboots, Microsoft will notify customers of this requirement. The customer is responsible for rebooting or updating to get the cluster patch. If users don't apply the patches according to AKS guidance, their cluster will continue to be vulnerable to the security issue.

## Node maintenance and access

Worker nodes are a shared responsibility and are owned by customers. Because of this, customers have the ability to sign in to their worker nodes and make potentially harmful changes such as kernel updates and installing or removing packages.

If customers make destructive changes or cause control plane components to go offline or become nonfunctional, AKS will detect this failure and automatically restore the worker node to the previous working state.

Although customers can sign in to and change worker nodes, doing this is discouraged because changes can make a cluster unsupportable.

## Network ports, access, and NSGs

As a managed service, AKS has specific networking and connectivity requirements. These requirements are less flexible than requirements for normal IaaS components. In AKS, operations like customizing NSG rules, blocking a specific port (for example, using firewall rules that block outbound port 443), and whitelisting URLs can make your cluster unsupportable.

**NOTE**

Currently, AKS doesn't allow you to completely lock down egress traffic from your cluster. To control the list of URLs and ports your cluster can use for outbound traffic see [limit egress traffic](#).

## Unsupported alpha and beta Kubernetes features

AKS supports only stable features within the upstream Kubernetes project. Unless otherwise documented, AKS doesn't support alpha and beta features that are available in the upstream Kubernetes project.

In two scenarios, alpha or beta features might be rolled out before they're generally available:

- Customers have met with the AKS product, support, or engineering teams and have been asked to try these new features.
- These features have been [enabled by a feature flag](#). Customers must explicitly opt in to use these features.

## Preview features or feature flags

For features and functionality that require extended testing and user feedback, Microsoft releases new preview features or features behind a feature flag. Consider these features as prerelease or beta features.

Preview features or feature-flag features aren't meant for production. Ongoing changes in APIs and behavior, bug fixes, and other changes can result in unstable clusters and downtime.

Features in public preview are fall under 'best effort' support as these features are in preview and not meant for production and are supported by the AKS technical support teams during business hours only. For additional information please see:

- [Azure Support FAQ](#)

**NOTE**

Preview features take effect at the Azure *subscription* level. Don't install preview features on a production subscription. On a production subscription, preview features can change default API behavior and affect regular operations.

## Upstream bugs and issues

Given the speed of development in the upstream Kubernetes project, bugs invariably arise. Some of these bugs can't be patched or worked around within the AKS system. Instead, bug fixes require larger patches to upstream projects (such as Kubernetes, node or worker operating systems, and kernels). For components that Microsoft owns (such as the Azure cloud provider), AKS and Azure personnel are committed to fixing issues upstream in the community.

When a technical support issue is root-caused by one or more upstream bugs, AKS support and engineering teams will:

- Identify and link the upstream bugs with any supporting details to help explain why this issue affects your cluster or workload. Customers receive links to the required repositories so they can watch the issues and see when a new release will provide fixes.
- Provide potential workarounds or mitigations. If the issue can be mitigated, a [known issue](#) will be filed in the AKS repository. The known-issue filing explains:
  - The issue, including links to upstream bugs.
  - The workaround and details about an upgrade or another persistence of the solution.
  - Rough timelines for the issue's inclusion, based on the upstream release cadence.

# Frequently asked questions about Azure Kubernetes Service (AKS)

2/27/2020 • 9 minutes to read • [Edit Online](#)

This article addresses frequent questions about Azure Kubernetes Service (AKS).

## Which Azure regions currently provide AKS?

For a complete list of available regions, see [AKS regions and availability](#).

## Does AKS support node autoscaling?

Yes, the ability to automatically scale agent nodes horizontally in AKS is currently available in preview. See [Automatically scale a cluster to meet application demands in AKS](#) for instructions. AKS autoscaling is based on the [Kubernetes autoscaler](#).

## Can I deploy AKS into my existing virtual network?

Yes, you can deploy an AKS cluster into an existing virtual network by using the [advanced networking feature](#).

## Can I limit who has access to the Kubernetes API server?

Yes, you can limit access to the Kubernetes API server using [API Server Authorized IP Ranges](#).

## Can I make the Kubernetes API server accessible only within my virtual network?

Not at this time, but this is planned. You can track progress on the [AKS GitHub repo](#).

## Can I have different VM sizes in a single cluster?

Yes, you can use different virtual machine sizes in your AKS cluster by creating [multiple node pools](#).

## Are security updates applied to AKS agent nodes?

Azure automatically applies security patches to the Linux nodes in your cluster on a nightly schedule. However, you are responsible for ensuring that those Linux nodes are rebooted as required. You have several options for rebooting nodes:

- Manually, through the Azure portal or the Azure CLI.
- By upgrading your AKS cluster. The cluster upgrades [cordon and drain nodes](#) automatically and then bring a new node online with the latest Ubuntu image and a new patch version or a minor Kubernetes version. For more information, see [Upgrade an AKS cluster](#).
- By using [Kured](#), an open-source reboot daemon for Kubernetes. Kured runs as a [DaemonSet](#) and monitors each node for the presence of a file that indicates that a reboot is required. Across the cluster, OS reboots are managed by the same [cordon and drain process](#) as a cluster upgrade.

For more information about using kured, see [Apply security and kernel updates to nodes in AKS](#).

### Windows Server nodes

For Windows Server nodes (currently in preview in AKS), Windows Update does not automatically run and apply the latest updates. On a regular schedule around the Windows Update release cycle and your own validation process, you should perform an upgrade on the cluster and the Windows Server node pool(s) in your AKS cluster. This upgrade process creates nodes that run the latest Windows Server image and patches, then removes the older nodes. For more information on this process, see [Upgrade a node pool in AKS](#).

## Why are two resource groups created with AKS?

AKS builds upon a number of Azure infrastructure resources, including virtual machine scale sets, virtual networks, and managed disks. This enables you to leverage many of the core capabilities of the Azure platform within the managed Kubernetes environment provided by AKS. For example, most Azure virtual machine types can be used directly with AKS and Azure Reservations can be used to receive discounts on those resources automatically.

To enable this architecture, each AKS deployment spans two resource groups:

1. You create the first resource group. This group contains only the Kubernetes service resource. The AKS resource provider automatically creates the second resource group during deployment. An example of the second resource group is *MC\_myResourceGroup\_myAKSCluster\_eastus*. For information on how to specify the name of this second resource group, see the next section.
2. The second resource group, known as the *node resource group*, contains all of the infrastructure resources associated with the cluster. These resources include the Kubernetes node VMs, virtual networking, and storage. By default, the node resource group has a name like *MC\_myResourceGroup\_myAKSCluster\_eastus*. AKS automatically deletes the node resource whenever the cluster is deleted, so it should only be used for resources which share the cluster's lifecycle.

## Can I provide my own name for the AKS node resource group?

Yes. By default, AKS will name the node resource group *MC\_resourcegroupname\_clusternamespace\_location*, but you can also provide your own name.

To specify your own resource group name, install the [aks-preview](#) Azure CLI extension version 0.3.2 or later. When you create an AKS cluster by using the `az aks create` command, use the `--node-resource-group` parameter and specify a name for the resource group. If you [use an Azure Resource Manager template](#) to deploy an AKS cluster, you can define the resource group name by using the `nodeResourceGroup` property.

- The secondary resource group is automatically created by the Azure resource provider in your own subscription.
- You can specify a custom resource group name only when you're creating the cluster.

As you work with the node resource group, keep in mind that you cannot:

- Specify an existing resource group for the node resource group.
- Specify a different subscription for the node resource group.
- Change the node resource group name after the cluster has been created.
- Specify names for the managed resources within the node resource group.
- Modify or delete tags of managed resources within the node resource group. (See additional information in the next section.)

## Can I modify tags and other properties of the AKS resources in the node resource group?

If you modify or delete Azure-created tags and other resource properties in the node resource group, you could get unexpected results such as scaling and upgrading errors. AKS allows you to create and modify custom tags.

You might want to create or modify custom tags, for example, to assign a business unit or cost center. By modifying the resources under the node resource group in the AKS cluster, you break the service-level objective (SLO). For more information, see [Does AKS offer a service-level agreement?](#)

## What Kubernetes admission controllers does AKS support? Can admission controllers be added or removed?

AKS supports the following [admission controllers](#):

- *NamespaceLifecycle*
- *LimitRanger*
- *ServiceAccount*
- *DefaultStorageClass*
- *DefaultTolerationSeconds*
- *MutatingAdmissionWebhook*
- *ValidatingAdmissionWebhook*
- *ResourceQuota*

Currently, you can't modify the list of admission controllers in AKS.

## Is Azure Key Vault integrated with AKS?

AKS isn't currently natively integrated with Azure Key Vault. However, the [Azure Key Vault FlexVolume for Kubernetes project](#) enables direct integration from Kubernetes pods to Key Vault secrets.

## Can I run Windows Server containers on AKS?

Yes, Windows Server containers are available in preview. To run Windows Server containers in AKS, you create a node pool that runs Windows Server as the guest OS. Windows Server containers can use only Windows Server 2019. To get started, see [Create an AKS cluster with a Windows Server node pool](#).

Windows Server support for node pool includes some limitations that are part of the upstream Windows Server in Kubernetes project. For more information on these limitations, see [Windows Server containers in AKS limitations](#).

## Does AKS offer a service-level agreement?

In a service-level agreement (SLA), the provider agrees to reimburse the customer for the cost of the service if the published service level isn't met. Since AKS is free, no cost is available to reimburse, so AKS has no formal SLA. However, AKS seeks to maintain availability of at least 99.5 percent for the Kubernetes API server.

It is important to recognize the distinction between AKS service availability which refers to uptime of the Kubernetes control plane and the availability of your specific workload which is running on Azure Virtual Machines. Although the control plane may be unavailable if the control plane is not ready, your cluster workloads running on Azure VMs can still function. Given Azure VMs are paid resources they are backed by a financial SLA. Read [here for more details](#) on the Azure VM SLA and how to increase that availability with features like [Availability Zones](#).

## Why can't I set maxPods below 30?

In AKS, you can set the `maxPods` value when you create the cluster by using the Azure CLI and Azure Resource Manager templates. However, both Kubenet and Azure CNI require a *minimum value* (validated at creation time):

NETWORKING	MINIMUM	MAXIMUM
Azure CNI	30	250
Kubenet	30	110

Because AKS is a managed service, we deploy and manage add-ons and pods as part of the cluster. In the past, users could define a `maxPods` value lower than the value that the managed pods required to run (for example, 30). AKS now calculates the minimum number of pods by using this formula:  $((\text{maxPods} \text{ or } (\text{maxPods} * \text{vm\_count})) > \text{managed add-on pods minimum})$ .

Users can't override the minimum `maxPods` validation.

## Can I apply Azure reservation discounts to my AKS agent nodes?

AKS agent nodes are billed as standard Azure virtual machines, so if you've purchased [Azure reservations](#) for the VM size that you are using in AKS, those discounts are automatically applied.

## Can I move/migrate my cluster between Azure tenants?

The `az aks update-credentials` command can be used to move an AKS cluster between Azure tenants. Follow the instructions in [Choose to update or create a service principal](#) and then [update aks cluster with new credentials](#).

## Can I move/migrate my cluster between subscriptions?

Movement of clusters between subscriptions is currently unsupported.

## Can I move my AKS clusters from the current azure subscription to another?

Moving your AKS cluster and its associated resources between Azure subscriptions is not supported.

## Why is my cluster delete taking so long?

Most clusters are deleted upon user request; in some cases, especially where customers are bringing their own Resource Group, or doing cross-RG tasks deletion can take additional time or fail. If you have an issue with deletes, double-check that you do not have locks on the RG, that any resources outside of the RG are disassociated from the RG, etc.

## If I have pod / deployments in state 'NodeLost' or 'Unknown' can I still upgrade my cluster?

You can, but AKS does not recommend this. Upgrades should ideally be performed when the state of the cluster is known and healthy.

## If I have a cluster with one or more nodes in an Unhealthy state or shut down, can I perform an upgrade?

No, please delete/remove any nodes in a failed state or otherwise removed from the cluster prior to upgrading.

## I ran a cluster delete, but see the error

## [Errno 11001] getaddrinfo failed

Most commonly, this is caused by users having one or more Network Security Groups (NSGs) still in use and associated with the cluster. Please remove them and attempt the delete again.

## I ran an upgrade, but now my pods are in crash loops, and readiness probes fail?

Please confirm your service principal has not expired. Please see: [AKS service principal](#) and [AKS update credentials](#).

## My cluster was working, but suddenly can not provision LoadBalancers, mount PVCs, etc.?

Please confirm your service principal has not expired. Please see: [AKS service principal](#) and [AKS update credentials](#).

## Can I use the virtual machine scale set APIs to scale manually?

No, scale operations by using the virtual machine scale set APIs aren't supported. Use the AKS APIs (  
`az aks scale`).

## Can I use virtual machine scale sets to manually scale to 0 nodes?

No, scale operations by using the virtual machine scale set APIs aren't supported.

## Can I stop or de-allocate all my VMs?

While AKS has resilience mechanisms to withstand such a config and recover from it, this is not a recommended configuration.

## Can I use custom VM extensions?

No AKS is a managed service, and manipulation of the IaaS resources is not supported. To install custom components, etc. please leverage the Kubernetes APIs and mechanisms. For example, leverage DaemonSets to install required components.