E-book series

Microsoft Azure

# Kubernetes
# Best Practices

Fallow: LifeTimeDevOps

# Contents

# Kubernetes best practices

## Introduction

This guide gives recommendations around multi-tenancy, security, network and storage, and running enterprise-ready workloads. It is aimed at developers and application owners who are familiar with Kubernetes and have a good understanding of Azure Kubernetes Service (AKS) core concepts. The following best practices are based on real-world deployments of Kubernetes that we have gathered directly from our experience in the field.

If you are following our *Kubernetes Learning Path*, this content is part of the last step, "Operational best practices for Kubernetes," see 50 days from zero to hero with Kubernetes for more.

# Multi-tenancy

## Best practices for cluster isolation

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.

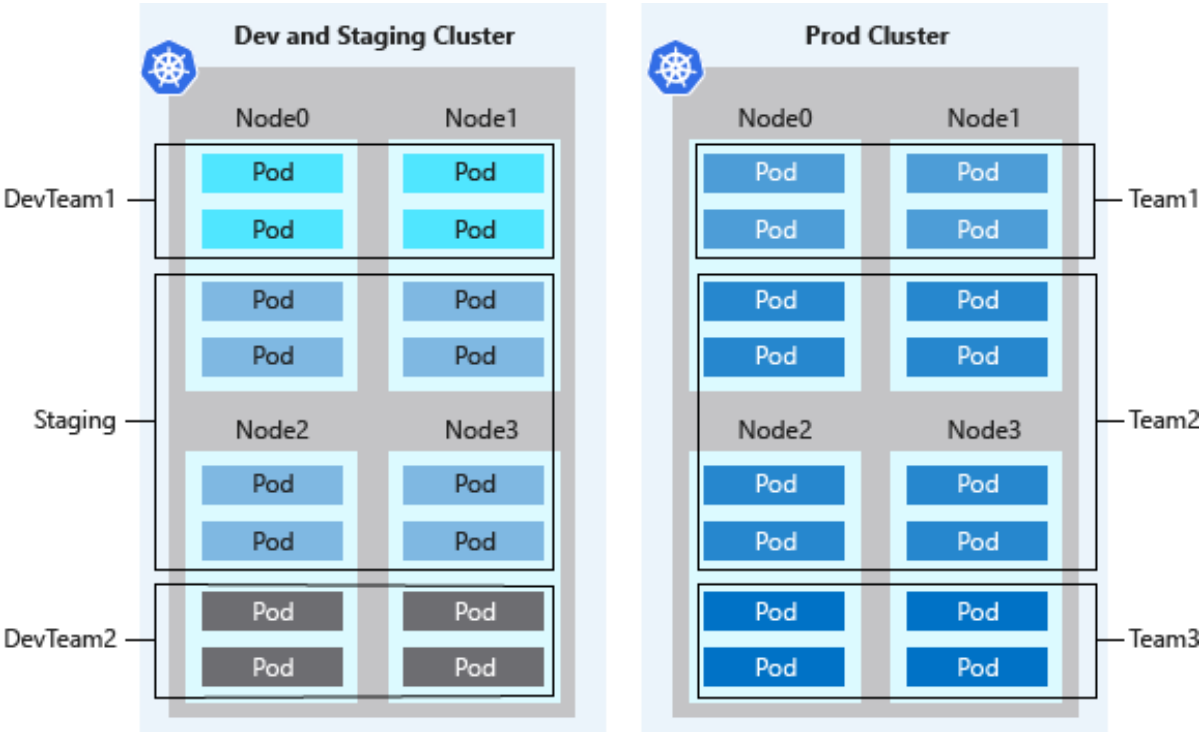
- **Authentication and authorization** include the user of role-based access control (RBAC) and Azure Active Directory (AD) integration, pod identities, and secrets in Azure Key Vault. For more information about these features, see Best practices for authentication and authorization in AKS.

- **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.



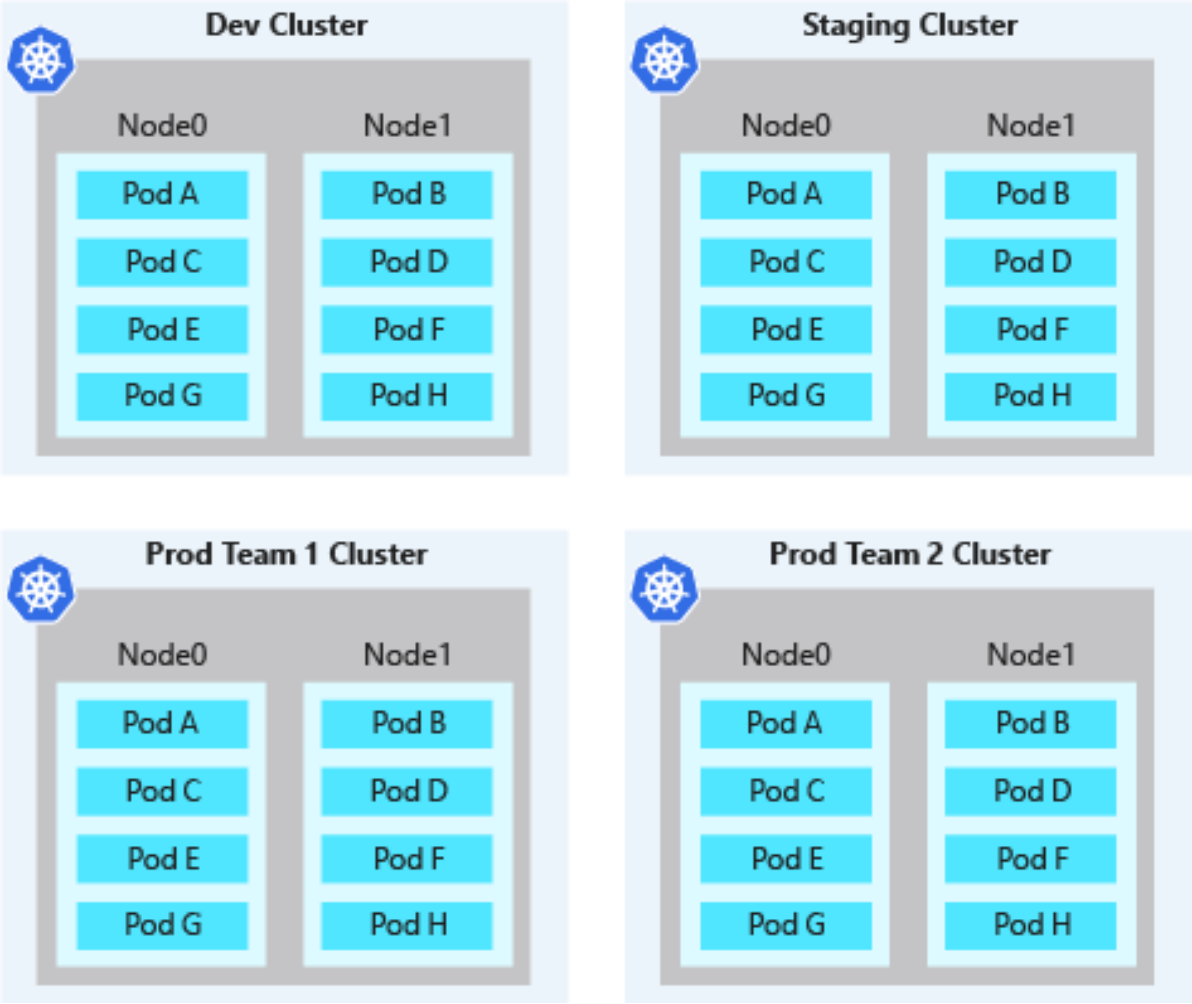

Logical separation of clusters usually provides a higher pod density than physically isolated clusters. There's less excess compute capacity that sits idle in the cluster. When combined with the Kubernetes cluster autoscaler, you can scale the number of nodes up or down to meet demands. This best practice 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. 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 is 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.

# Multi-tenancy

## Best practices for basic scheduler features

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 as 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.

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.

# Multi-tenancy

## Best practices for advanced scheduler features

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 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.

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 nodepool 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.

## 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 without virtual machine scale support**
  - 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 *node*1 becomes *node3*, and *node2* becomes *node1*.
- **Clusters that use virtual machine scale sets** (currently in preview in AKS)
  - Again, 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.

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 not 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 Colocate pods on the same node.

# Multi-tenancy

## Best practices for authentication and authorization

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:

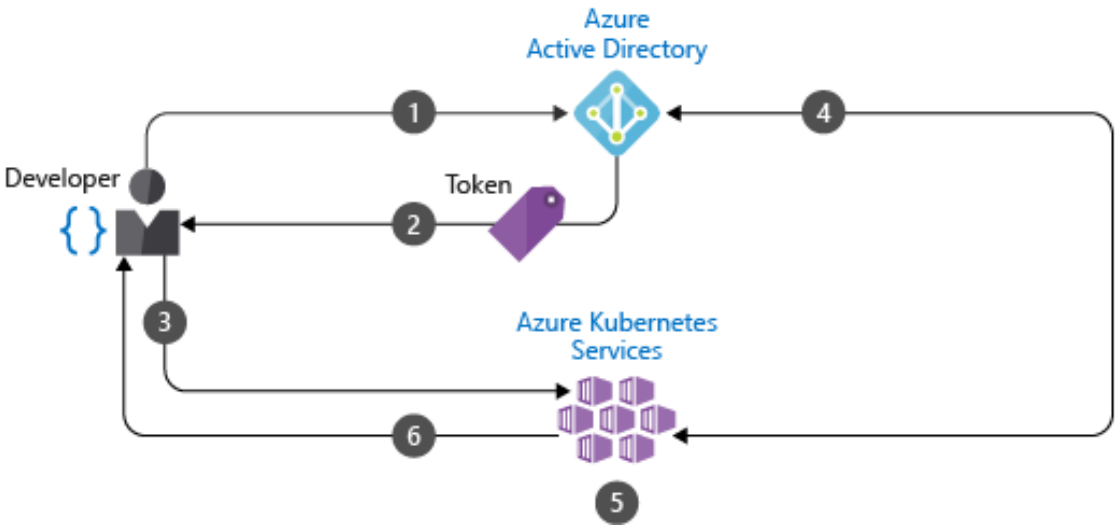
- Authenticate AKS cluster users with Azure Active Directory
- Control access to resources with role-based access controls (RBAC)
- Use a managed identity to authenticate themselves with other services

## 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 the 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.

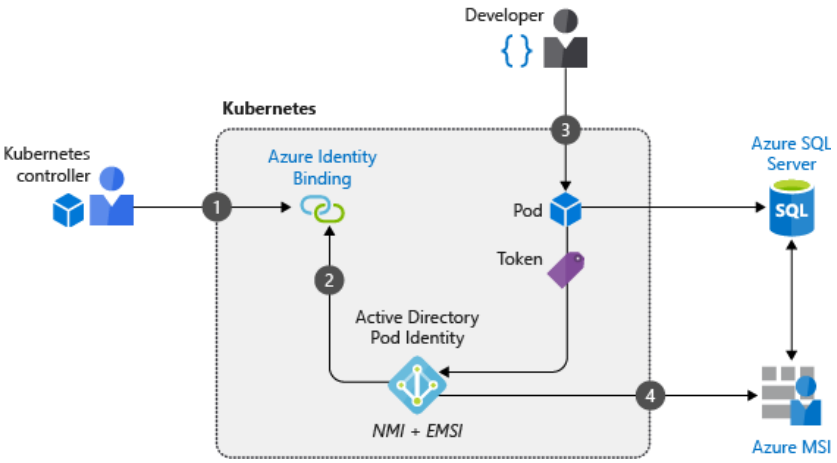
When pods need access to other Azure services, such as Cosmos DB, Key Vault, or Blob Storage, the pod needs access credentials. These access credentials could be defined with the container image or injected as a Kubernetes secret, but need to be manually created and assigned. Often, the credentials are 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.

Managed pod identities is an AKS open source project, and is not supported by Azure technical support. It is provided to gather feedback and bugs from our community. The project is not recommended for production use.

To use pod identities, see Azure Active Directory identities for Kubernetes applications.

# Security

## Best practices for cluster security and upgrades

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 update to date and automatically apply security patches

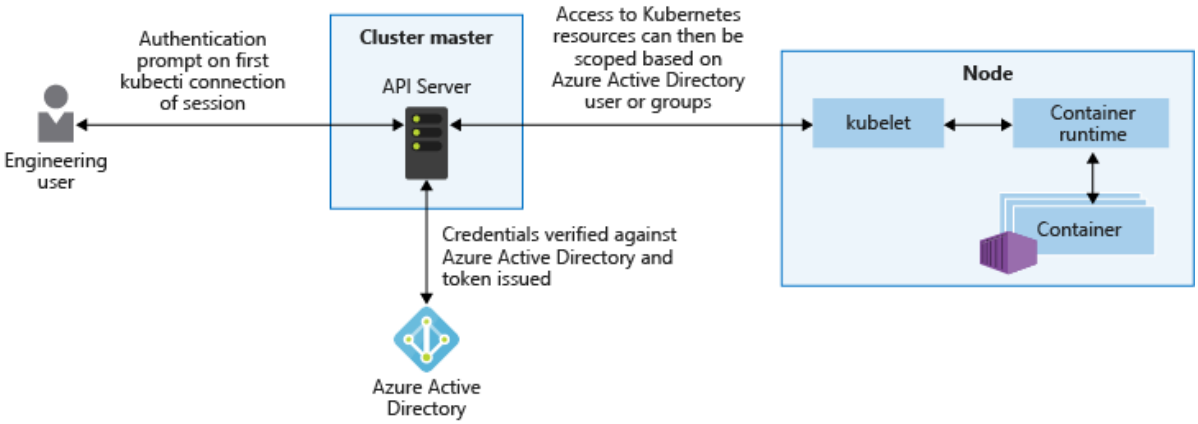
You can also read the best practices for container image management and for pod security.

### 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.

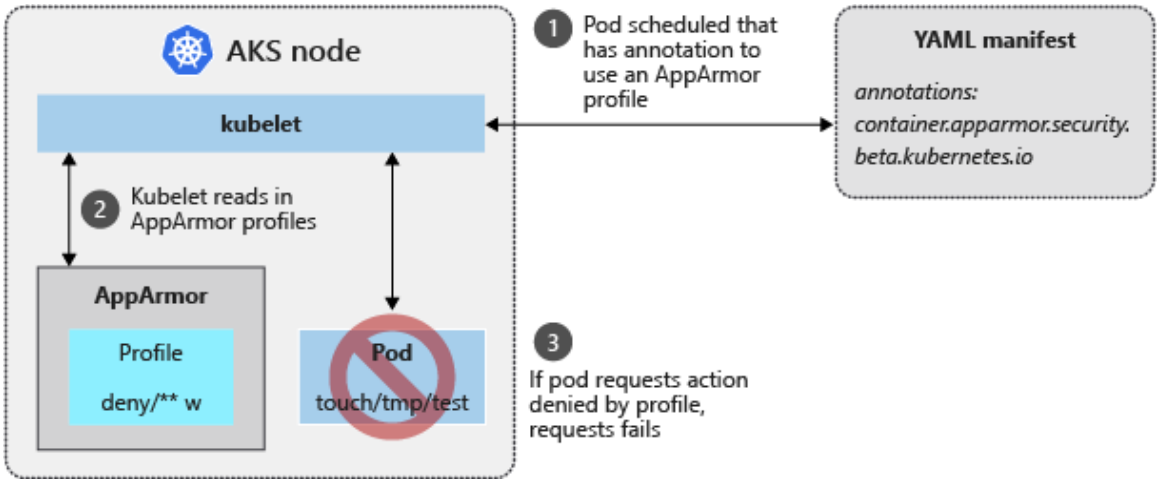
In the same way that you should grant users or groups the least number of privileges required, containers should also be limited to only the actions and processes that they need. To minimize the risk of attack, don't configure applications and containers that require escalated privileges or root access. 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.

> 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` 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:
    seccomp.security.alpha.kubernetes.io/pod: localhost/prevent-chmod
spec:
  containers:
  - name: chmod
    image: busybox
    command:
      - “chmod”
    args:
     - “777”
     - /etc/hostname
  restartPolicy: Never
```

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 1.11.8
```

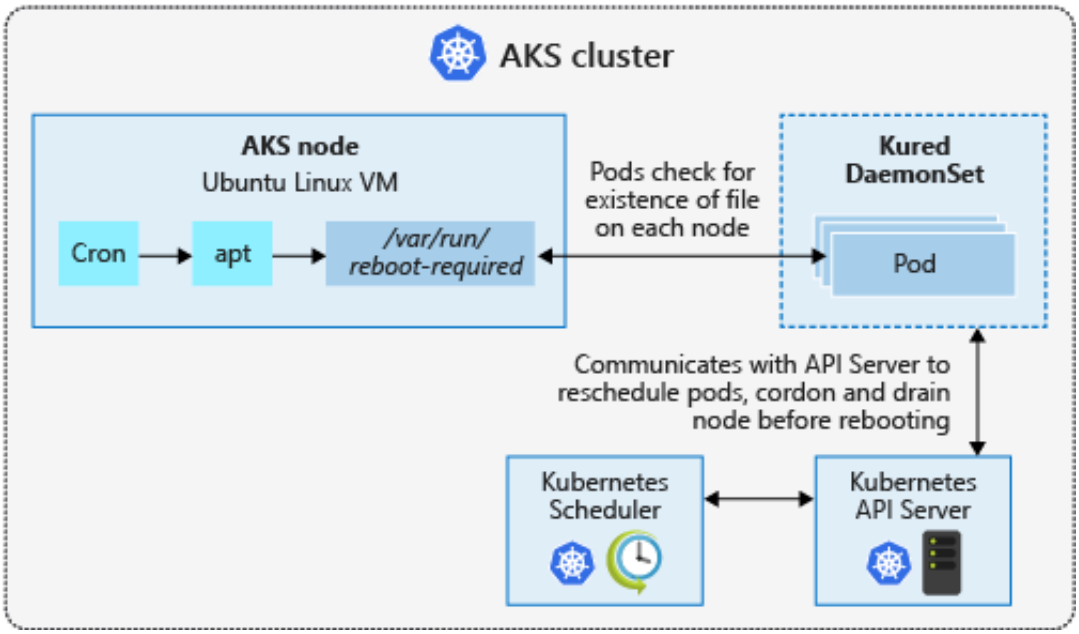
For more information about upgrades in AKS, see Supported Kubernetes versions in AKS and Upgrade an AKS cluster.

# Process node updates and reboots using kured

**Best practice guidance -** AKS automatically downloads and installs security fixes on each of the worker 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.

Each evening, the AKS nodes 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 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.

# Security

## Best practices for container image management and security

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:

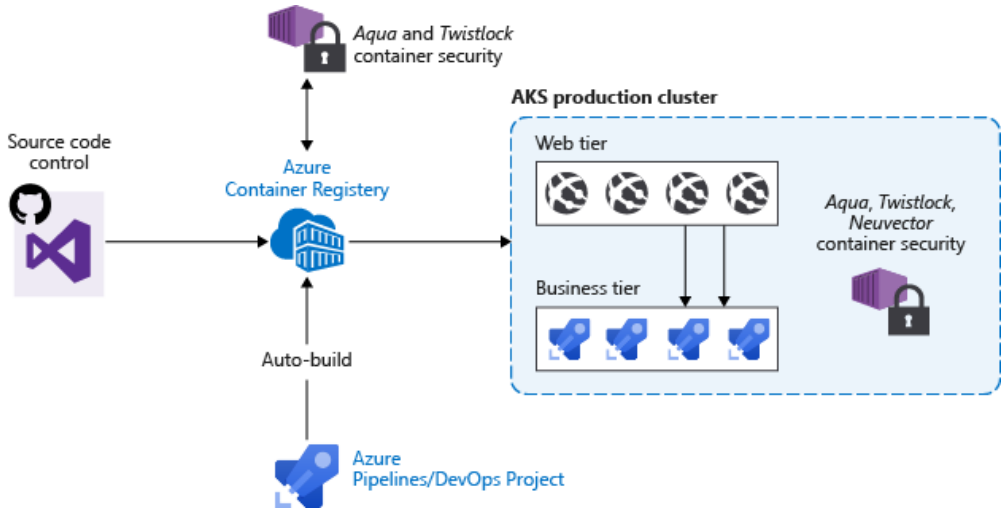
- Scan for and remediate image vulnerabilities
- Use a trusted registry with digitally signed container images
- Automatically trigger and redeploy container images when a base image is updated

You can also read the best practices for cluster security and for pod security.

# 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 makes 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.

# Security

## Best practices for pod security

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 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. 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.

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.

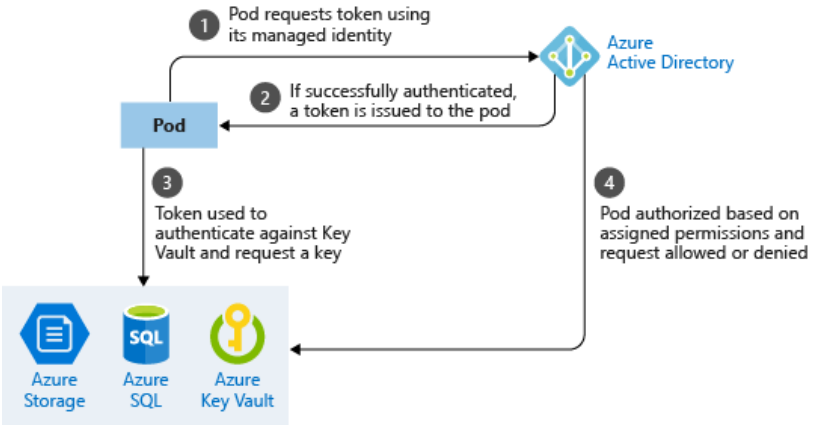
The following associated AKS open source projects let you automatically authenticate pods or request credentials and keys from a digital vault:

- 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 any service in Azure that supports it such as Storage, 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:



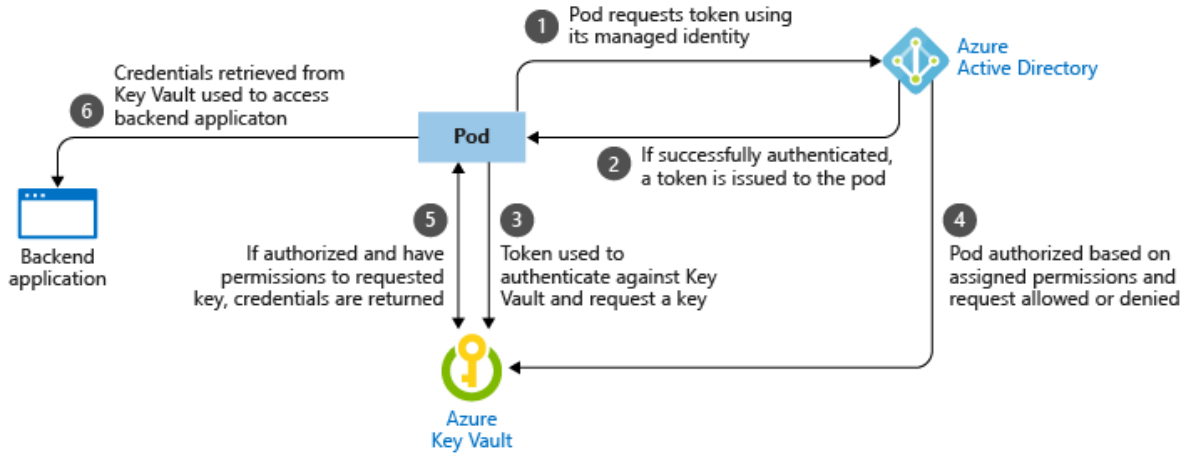
With a managed identity, your application code doesn't need to include credentials to access a service, such as Azure Storage. As each pod authenticates with its own identity, so you can audit and review access. If your application connects with other Azure services, use managed identities to limit credential 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.

# Network and storage

## Best practices for network connectivity and security

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 firewalls (WAF)
- Securely connect to cluster nodes

## Choose the appropriate network model

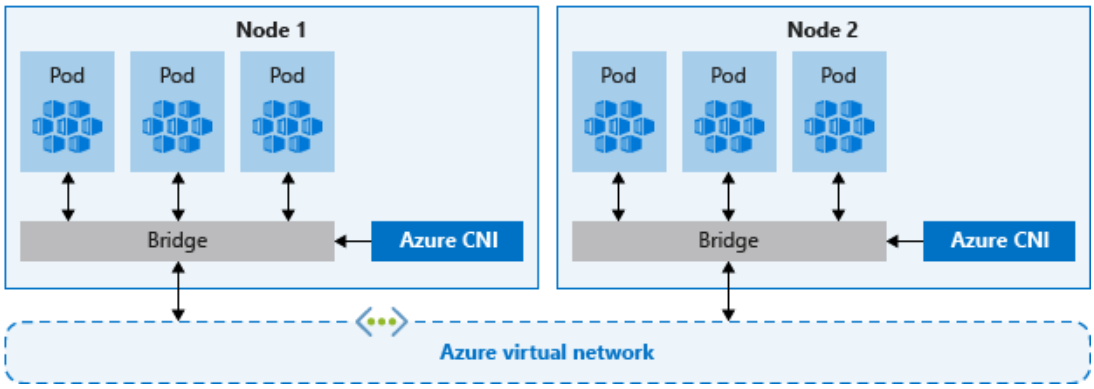
**Best practice guidance -** For integration with existing virtual networks or on-premises networks, use Azure CNI networking in AKS. This network model also allows greater separation of resources and controls in an enterprise environment.

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 up events or cluster upgrades, you also need additional IP addresses available for use in the assigned subnet.

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:

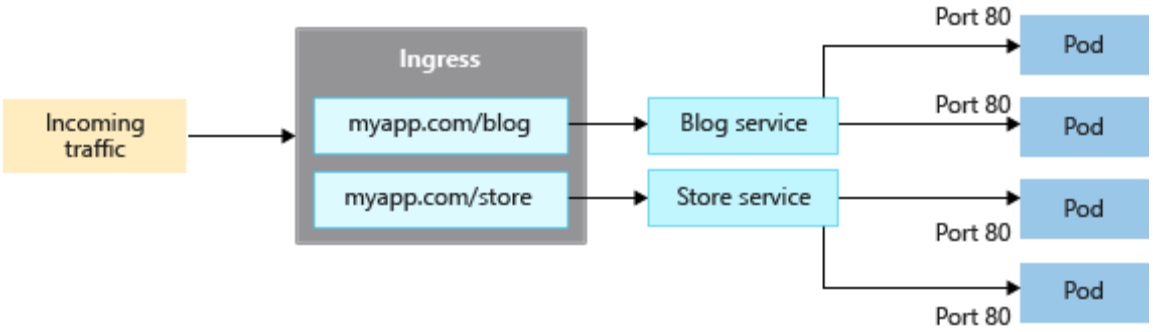
- Nodes and pods are placed on different IP subnets. User Defined Routing (UDR) and IP forwarding is used to route traffic between pods and nodes. This additional routing may reduce network performance.
- Connections to existing on-premises networks or peering to other Azure virtual networks can be complex.

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 Kuberenetes 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.

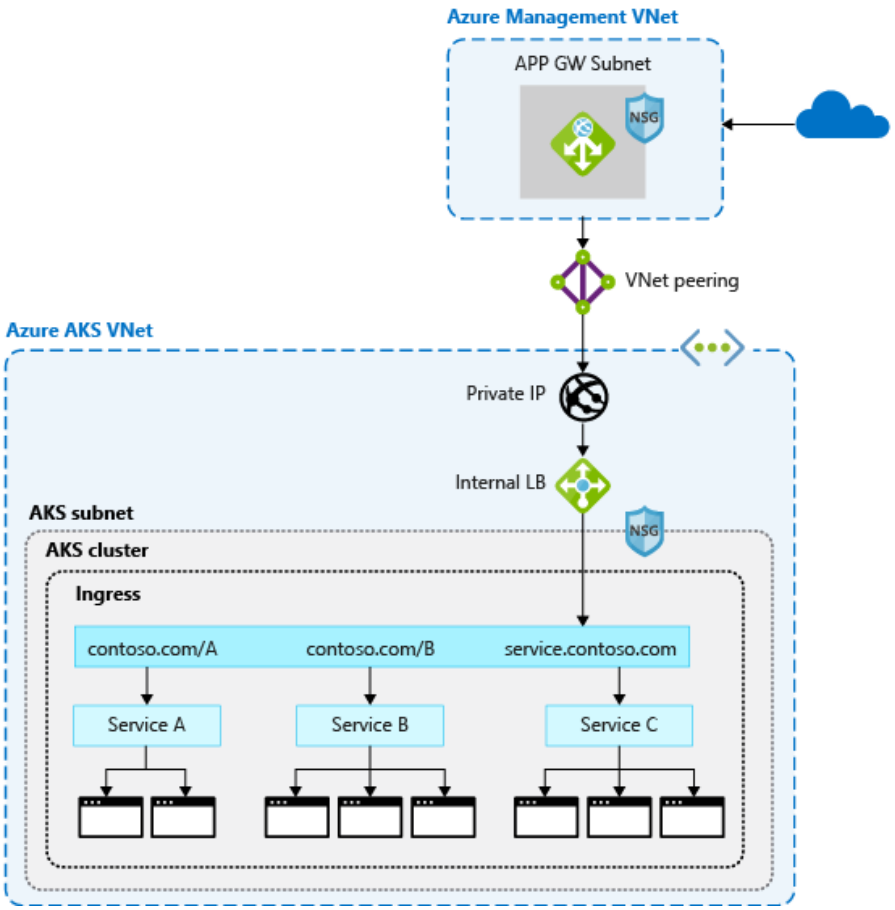
There are many scenarios for ingress, including the following how-to guides:

- 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 (currently in preview in AKS) 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.

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:

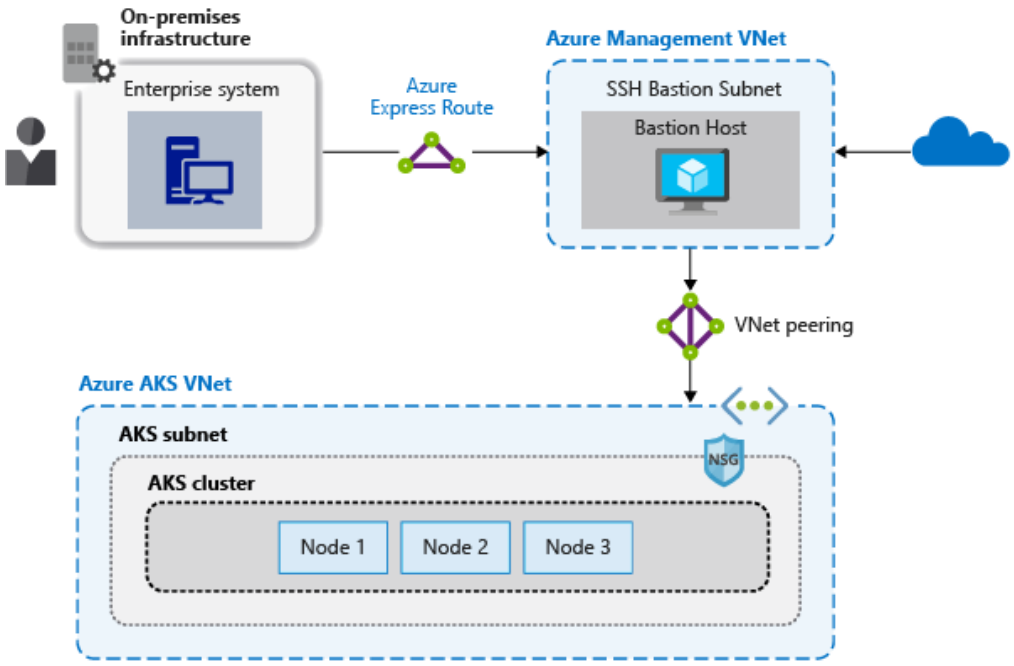
```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: backend-policy
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
```

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.

# Network and storage

## Best practices for storage and backups

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:

The following table outlines the available storage types and their capabilities:

| Use case | Volume plugin | Read/write once | Read-only many | Read/write many |
|---|---|---|---|---|
| Shared configuration | Azure Files | Yes | Yes | Yes |
| Structured app data | Azure Disks | Yes | No | No |
| App data, read-only shares | Dysk (preview) | Yes | Yes | No |
| Unstructured data, file system operations | BlobFuse (preview) | Yes | Yes | Yes |

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 |

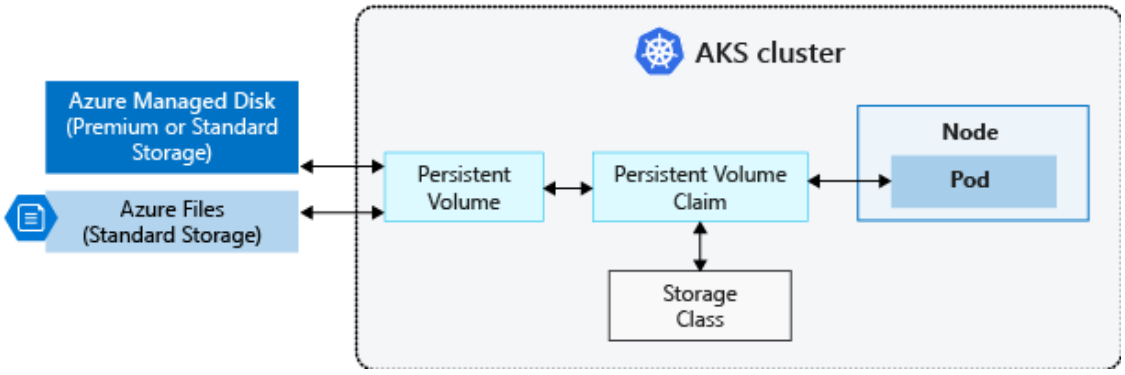
Here, the Standard_DS2_v2 allows double the number of attached disks, and provides three to four times the amount of IOPS and disk throughput. If you only looked at the core compute resources and compared costs, you may choose the Standard_B2ms VM size and have poor storage performance and limitations. 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 a hook 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.

# Running enterprise-ready workloads

## Best practices for business continuity and disaster recovery

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. These multiple nodes don't protect you from a region failure. To maximize your uptime, implement some business continuity and disaster recovery features.

This best practices article focuses on considerations that help you 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 with 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 multi-region deployment

**Best practice guidance -** 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 yourself from region failure, deploy your application into multiple AKS clusters across different regions. When you plan what regions to deploy your AKS cluster, the following considerations apply:

- AKS region availability
  - Choose regions close to your users. AKS is continually expanding into new regions.
- Azure paired regions
  - For your geographic area, choose two regions that are paired with each other. These regions coordinate platform updates, and prioritize recovery efforts where needed.
- Service Availability Level (Hot/Hot, Hot/Warm, Hot/Cold)
  - Do you want to run both regions at the same time, with one region ready to start serving traffic, or one region that needs time to get ready to serve traffic.

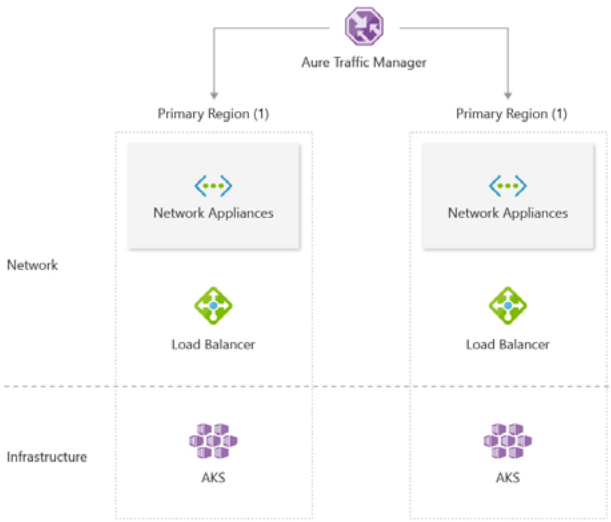
AKS region availability and paired regions are joint consideration. Deploy your AKS clusters into paired regions that are designed to manage region disaster recovery together. For example, AKS is available in East US and West US. These regions are also paired. These two regions would be recommended when creating an AKS BC/DR strategy.

When you deploy your application, you must also add another step to your CI/CD pipeline to deploy to these multiple AKS clusters. If you don't update your deployment pipelines, application deployments may only be deployed into one of your regions and AKS clusters. Customer traffic that is directed to a secondary region won't receive the latest code updates.

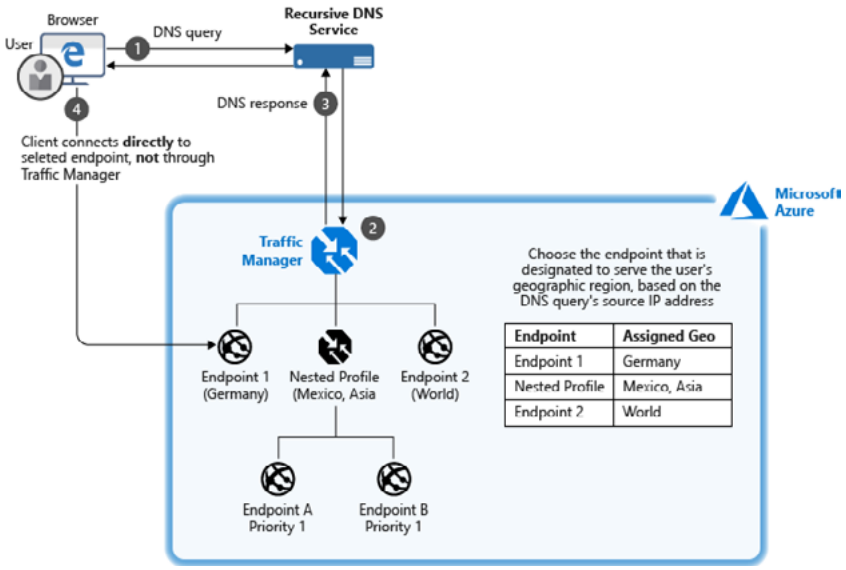
# Use Azure Traffic Manager to route traffic

**Best practice guidance -** 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 going to your AKS cluster.

With multiple AKS clusters in different regions, you need to control how traffic is directed 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. You can route users based on cluster response time, or based on geography.




With a single AKS cluster, customers typically connect to the Service IP or DNS name of a given application. In a multi-cluster deployment, customers should connect to a Traffic Manager DNS name that points to the services on each AKS cluster. These services are defined using Traffic Manager endpoints. Each endpoint is the Service Load Balancer IP. This configuration lets you direct network traffic from the Traffic Manager endpoint in one region to the endpoint in a different region.




Traffic Manager is used to perform the DNS lookups and return the most appropriate endpoint for a user. Nested profiles can be used, with priority given for a primary location. For example, a user should primarily connect to their closest geographic region. If that region has a problem, Traffic Manager instead directs them to a secondary region. This approach makes sure customers can always connect to an application instance, even if their closest geographic region is unavailable.

For steps on how to set up these endpoints and routing, see Configure the geographic traffic routing method using Traffic Manager.

## Layer 7 application routing with Azure Front Door

Azure Traffic Manager uses DNS (layer 3) to shape traffic. Azure Front Door (currently in preview) provides an HTTP/HTTPS (layer 7) routing option. Additional features of Front Door 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.

# Enable geo-replication for container images

**Best practice guidance -** 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. Azure Container Registry (ACR) can integrate with AKS to securely store your container images or Helm charts. ACR supports multi-master geo-replication to automatically replicate your images to Azure regions around the world. To improve performance and availability, use ACR 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 ACR registry in the same region:


The benefits of using ACR geo-replication include the following:

- **Pulling images from the same region is faster.** You pull images from high-speed, low latency network connections within the same Azure region.
- **Pulling images from the same region is more reliable.** If a region is unavailable, your AKS cluster pulls the image from a different ACR registry that remains available.
- **Pulling images from the same region is cheaper.** There's no network egress charge between datacenters.

Geo-replication is a feature of Premium SKU ACR registries. For steps on how to configure replication, see Azure Container Registry geo-replication.

# Remove service state from inside containers

**Best practice guidance -** Where possible, don't store service state inside the container. Instead, use Azure PaaS services that support multi-region 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 may also include files or other resources that are stored on disk. For example, the files a database would use to store data and transaction logs.

State can be either externalized or colocated with the code that is manipulating the state. Externalization of state is typically done by using a database or other data store that runs on different machines over the network or out of process on the same machine.

Containers and microservices are most resilient when the processes that run inside them do not retain state. As your applications almost always contain some state, use a Platform as a Service solution such as Azure Database for MySQL/Postgres or Azure SQL.

For details on for how to build applications that are more portable, see the following guidelines:

- The Twelve-Factor App Methodology
- Run a web application in multiple Azure Regions

# Create a storage migration plan

**Best practice guidance -** If you use Azure Storage, prepare and test how to migrate your storage from the primary to the backup region.

Your applications may use Azure Storage for their data. As your applications are spread across multiple AKS clusters in different regions, you need to keep the storage synchronized. Two common ways of replicating storage include the following approaches:

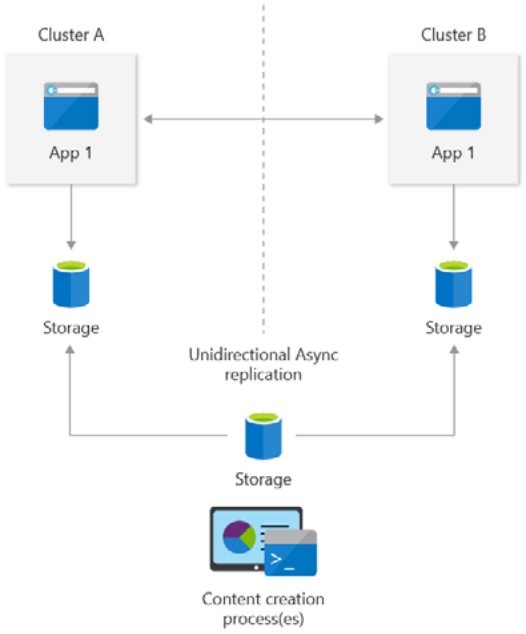
- Application-based asynchronous replication
- Infrastructure-based asynchronous replication

## Infrastructure-based asynchronous replication

Your applications may require persistent storage even after a pod is deleted. In Kubernetes, you can use Persistent Volumes to persist data storage. These persistent volumes are mounted to node VM, and then exposed to the pods. Persistent volumes follow pods, even if the pod is moved to a different node inside the same cluster.

Depending on the storage solution use, replication strategies may be different. Common storage solutions such as Gluster, CEPH, Rook, and Portworx all have their own guidance.

The central approach is a common storage point for applications to write their data. This data is then replicated across regions and then accessed locally.

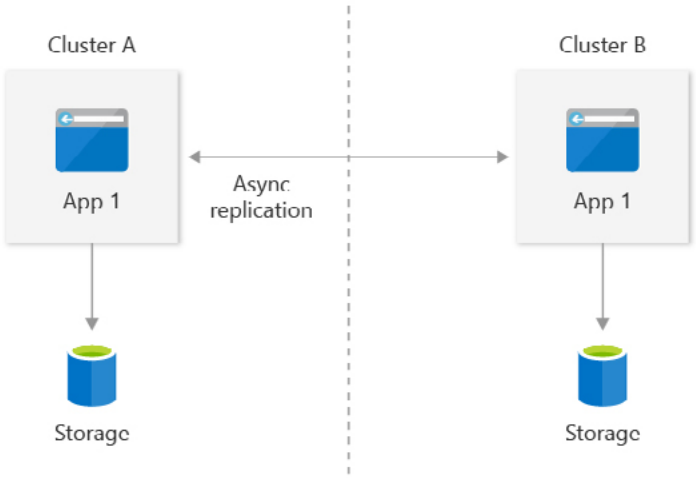



If you use Azure Managed Disks, available replication and DR solutions include using one of the following approaches:

- Ark on Azure
- Azure Site Recovery

## Application-based Asynchronous Replication

There is currently no Kubernetes-native implementation for application-based asynchronous replication. With the loosely coupled nature of containers and Kubernetes, any traditional application or language approach should work. The central approach is for the applications themselves to replicate the storage requests that are then written to each cluster's underlying data storage.



# Conclusion

We hope you have found the Kubernetes Best Practices guide helpful as a reference for implementing applications and managing cluster operations. Please check out our *Kubernetes Learning Path* program for an end-to-end series of resources on Kubernetes and about Azure Kubernetes Service (AKS) core concepts, including infrastructure components, access and identity, security, network, storage, and scaling. See our infographic 50 days from zero to hero with Kubernetes for complete details