**Best practices for cluster security and upgrades in Azure Kubernetes Service (AKS)**

As you manage clusters in Azure Kubernetes Service (AKS), the security of the 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 can 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](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/operator-best-practices-container-image-management.md) and for [pod security](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/developer-best-practices-pod-security.md).

You can also use [Azure Kubernetes Services integration with Security Center](https://github.com/MicrosoftDocs/azure-docs/blob/master/azure/security-center/azure-kubernetes-service-integration) to help detect threats and view recommendations for securing your AKS clusters.

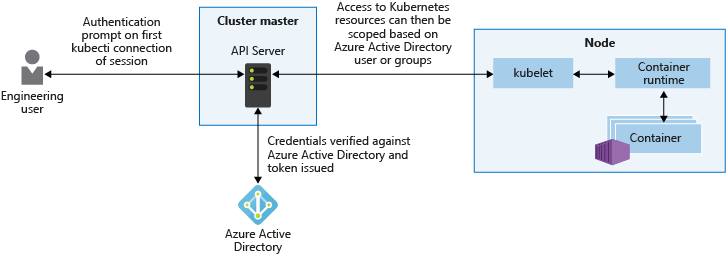
**Secure access to the API server and cluster nodes**

**Best practice guidance**

Securing access to the Kubernetes API-Server is one of the most important things you can do to safeguard the cluster. Integrate Kubernetes role-based access control (RBAC) with Azure Active Directory to control access to the API server. These controls allows you to secure AKS the same way, that you have secured access to your Azure subscriptions.

The Kubernetes API server provides a single connection point for all the 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 are 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 can use the existing user and group accounts to authenticate users to the API server.

[](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/media/operator-best-practices-cluster-security/aad-integration.png)

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 enables you to restrict access to the API server, and provides 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 be updated, but permissions on the AKS cluster would not reflect that. In this scenario, the user ends up being granted more permissions required.

For more information about the Azure AD integration and RBAC, you may refer to the below given link:

[Best practices for authentication and authorization in AKS](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/concepts-identity.md)

**Secure container access to resources**

**Best practice guidance**

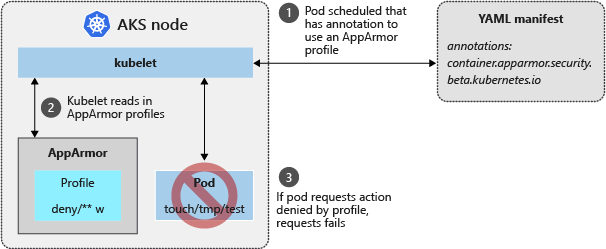
Limit the access to actions that containers can perform. Provide the least number of permissions, and avoid the use of root / privileged escalation.

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, do not 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 allow you to define additional permissions such as the user or group to run as, or what Linux capabilities to expose. For best practices, see [Secure pod access to resources](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/developer-best-practices-pod-security.md" \l "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.

**App Armor**

To limit the actions that containers can perform, you can use the [AppArmor](https://kubernetes.io/docs/tutorials/clusters/apparmor/) Linux kernel security module. AppArmor is available as part of the underlying AKS node OS, and is enabled by default. You can 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.

[](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/media/operator-best-practices-container-security/apparmor.png)

To see AppArmor in action, the following example creates a profile that prevents writing to files. [SSH](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/ssh.md) 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 are 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](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "apply) command:

kubectl apply -f aks-apparmor.yaml

With the pod deployed, use the [kubectl exec](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "exec) command to write to a file. The command cannot 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, you may check the [AppArmor profiles in Kubernetes](https://kubernetes.io/docs/tutorials/clusters/apparmor/).

**Secure computing**

While AppArmor works for any Linux application, [seccomp (](https://kubernetes.io/docs/concepts/policy/pod-security-policy/" \l "seccomp)*[sec](https://kubernetes.io/docs/concepts/policy/pod-security-policy/" \l "seccomp)*[ure](https://kubernetes.io/docs/concepts/policy/pod-security-policy/" \l "seccomp)*[comp](https://kubernetes.io/docs/concepts/policy/pod-security-policy/" \l "seccomp)*[uting)](https://kubernetes.io/docs/concepts/policy/pod-security-policy/" \l "seccomp) 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](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/ssh.md) 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](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "apply) command:

kubectl apply -f ./aks-seccomp.yaml

View the status of the pods using the [kubectl get pods](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands" \l "get) 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, refer to the [Seccomp security profiles for Docker](https://kubernetes.io/docs/concepts/policy/pod-security-policy/" \l "seccomp).

**Regularly update to the latest version of Kubernetes**

**Best practice guidance**

To track the new features and bug fixes, regularly upgrade 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. The 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, you may check the [Supported Kubernetes versions AKS](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/supported-kubernetes-versions.md)

To check the versions that are available for your cluster, use the [az aks get-upgrades](https://github.com/MicrosoftDocs/azure-docs/blob/master/cli/azure/aks" \l "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](https://github.com/MicrosoftDocs/azure-docs/blob/master/cli/azure/aks" \l "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, you are advised to go through the  [Supported Kubernetes versions in AKS](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/supported-kubernetes-versions.md) and [Upgrade an AKS cluster](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/upgrade-cluster.md).

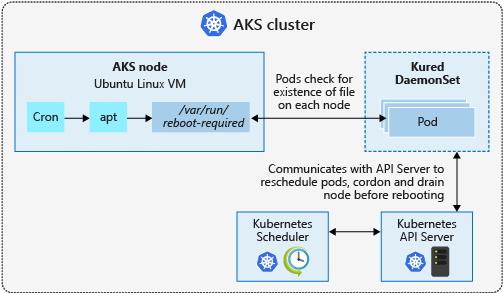
**Process Linux node updates and reboots using kured**

**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, 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)](https://github.com/weaveworks/kured) project by Weaveworks watches for pending node reboots. When a Linux node applies updates that require a reboot, the node is safely cordoned, 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.

[](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/media/operator-best-practices-cluster-security/node-reboot-process.png)

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.

**Best practices for container image management and security in Azure Kubernetes Service (AKS)**

As you develop and run applications in Azure Kubernetes Service (AKS), the security of your containers and container images is a vital issue. 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.

The below section emphasizes 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](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/operator-best-practices-cluster-security.md) and for [pod security](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/developer-best-practices-pod-security.md).

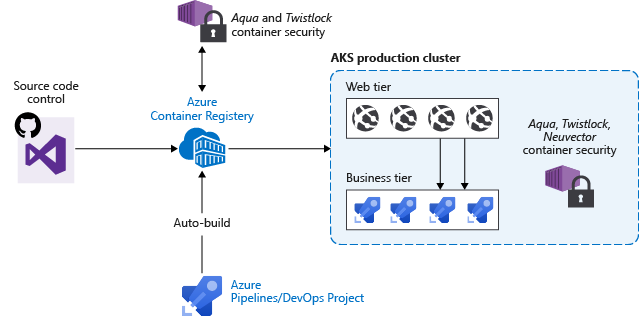
You can also use [Container security in Security Center](https://github.com/MicrosoftDocs/azure-docs/blob/master/azure/security-center/container-security) to help scan your containers for vulnerabilities. There is also [Azure Container Registry integration](https://github.com/MicrosoftDocs/azure-docs/blob/master/azure/security-center/azure-container-registry-integration) 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](https://www.twistlock.com/) or [Aqua](https://www.aquasec.com/), and then only allow verified images to be deployed.

[](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/media/operator-best-practices-container-security/scan-container-images-simplified.png)

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](https://github.com/MicrosoftDocs/azure-docs/blob/master/azure/devops/pipelines/?view=vsts) 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, you can read through the [Automate image builds on base image update with Azure Container Registry Tasks](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/container-registry/container-registry-tutorial-base-image-update.md).

**Best practices for basic scheduler features in Azure Kubernetes Service (AKS)**

As you manage the clusters in Azure Kubernetes Service (AKS), you often need to isolate teams and workloads. The Kubernetes scheduler provides features that enable you to control the distribution of compute resources, or limit the impact of maintenance events.

This phase highlights the basic Kubernetes scheduling features for cluster operators. Here, you can 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 do not define resource requests and limits, reject the deployment. Monitor the resource usage and adjust quotas accordingly.

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, refer to the [Define pod resource requests and limits](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/developer-best-practices-resource-management.md" \l "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 do not offer excess 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](https://kubernetes.io/docs/tasks/administer-cluster/manage-resources/memory-default-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, owners to understand their needs and apply the appropriate resource quotas.

For more information about available resource objects, scopes, and priorities, you can read the [Resource quotas in Kubernetes](https://kubernetes.io/docs/concepts/policy/resource-quotas/).

**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 allow you to plan, 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 the application developers, product managers to learn the requirements and apply the appropriate pod disruption budgets.

For more information about using pod disruption budgets, you may refer to [Specify a disruption budget for your application](https://kubernetes.io/docs/tasks/run-application/configure-pdb/).

**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 do not have resource requests and limits defined.

The [kube-advisor](https://github.com/Azure/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 do not 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](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/concepts-clusters-workloads.md" \l "node-selectors) 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.

**Best practices for advanced scheduler features in Azure Kubernetes Service (AKS)**

As you manage clusters in Azure Kubernetes Service (AKS), you often need to isolate teams and workloads. The Kubernetes scheduler provides advanced features enables you to control the scheduling of pods on certain nodes, or how multi-pod applications can be appropriately distributed across the cluster.

The section emphasizes on advanced Kubernetes scheduling features for cluster operators. You can 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 the resource-intensive applications, such as ingress controllers, to specific nodes. Keep the node resources available for workloads that require them, and do not 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 ensures that you can 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](https://kubernetes.io/docs/concepts/configuration/taint-and-toleration/).

To know more on, how to use multiple node pools in AKS, you can check [Create and manage multiple node pools for a cluster in AKS](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/use-multiple-node-pools.md).

**Behavior of taints and tolerations in AKS**

When you upgrade a node pool in AKS, taints and tolerations follow a set pattern as they are 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 the 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.

To know more about using node selectors, you can check  [Assigning Pods to Nodes](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/).

**Node affinity**

A node selector is a basic way to assign pods to a given node. More flexibility is available by using the *node affinity*. With the node affinity, you can define what happens if the pod can't be matched with a node. It is required that the Kubernetes scheduler matches a pod with a labeled host. You may prefer a match and allow the pod to be scheduled on a different host if a match is not available.

The following example set 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 *preferredDuringSchedulingIgnoreDuringExecution*:

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 which are already scheduled on the nodes.

For more information, you may refer to [Affinity and anti-affinity](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/" \l "affinity-and-anti-affinity).

**Inter-pod affinity and anti-affinity**

One final approach for the Kubernetes scheduler is to logically isolate workloads using the 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 ensures you have 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, you can read the [Co-locate pods on the same node](https://kubernetes.io/docs/concepts/configuration/assign-pod-node/" \l "always-co-located-in-the-same-node).

**Best practices for cluster isolation in Azure Kubernetes Service (AKS)**

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, the multi-tenancy and isolation features need to be understood and implemented.

This section focuses on isolation for cluster operators. It also gives insights on the following:

* 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 enable you to 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](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/concepts-clusters-workloads.md" \l "namespaces) in Kubernetes creates a logical isolation boundary. Additionally, Kubernetes features and considerations for isolation and multi-tenancy include the areas mentioned below:

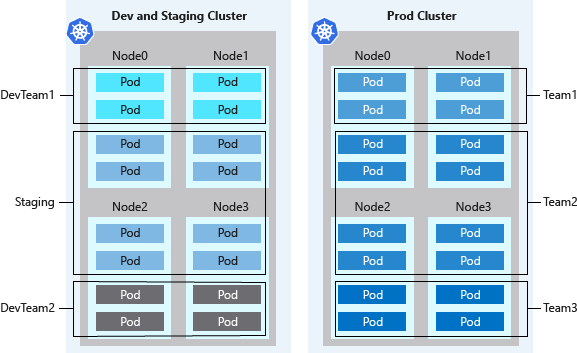
* **Scheduling** includes the use of basic features such as resource quotas and pod disruption budgets. For more information about these features, you can refer to the [Best practices for basic scheduler features in AKS](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/operator-best-practices-scheduler.md).
  + More advanced scheduler features include taints and tolerations, node selectors, and node and pod affinity or anti-affinity. You can read [Best practices for advanced scheduler features in AKS](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/operator-best-practices-advanced-scheduler.md)
* **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. To know more about the features, you can check the [Best practices for authentication and authorization in AKS](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/operator-best-practices-identity.md).
* **Containers** include pod security policies, pod security contexts, scanning images and runtimes for vulnerabilities. This 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 that you deploy to isolate teams or applications.

With the logical isolation, a single AKS cluster can be used for multiple workloads, teams, or environments. Kubernetes [Namespaces](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/concepts-clusters-workloads.md" \l "namespaces) form the logical isolation boundary for workloads and resources.

[](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/media/operator-best-practices-cluster-isolation/logical-isolation.png)

Logical separation of clusters usually provides a higher pod density than physically isolated clusters. There is less 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 in order to meet demands. The best approach is to allow the autoscaling to run only the required number of nodes and reduce the costs.

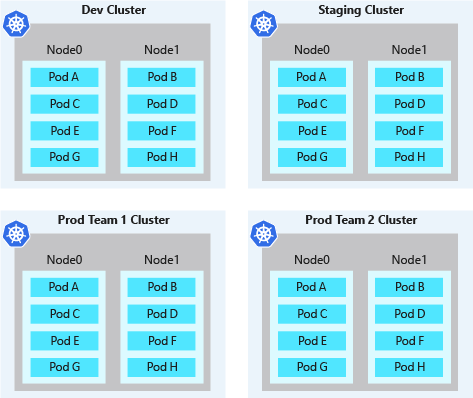
Kubernetes environments, in AKS or elsewhere, are not completely safe for hostile multi-tenant usage. In a multi-tenant environment, various tenants are working on a common, shared infrastructure. As a result if all tenants cannot be trusted, you need to do additional planning to avoid one tenant impacting 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, which has been 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 add to the additional management and financial overhead. You now have to maintain these multiple clusters, individually provide access and assign permissions. You are even billed for all the individual nodes.

[](https://github.com/MicrosoftDocs/azure-docs/blob/master/articles/aks/media/operator-best-practices-cluster-isolation/physical-isolation.png)

Physically separate clusters usually have a low pod density. As each team or workload has their own AKS cluster, this is often over-provisioned with compute resources. Often, a small number of pods are scheduled on those nodes. Unused capacity on the nodes can not be used for applications or services in development by other teams. These excess resources contribute to the additional costs in physically separate clusters.