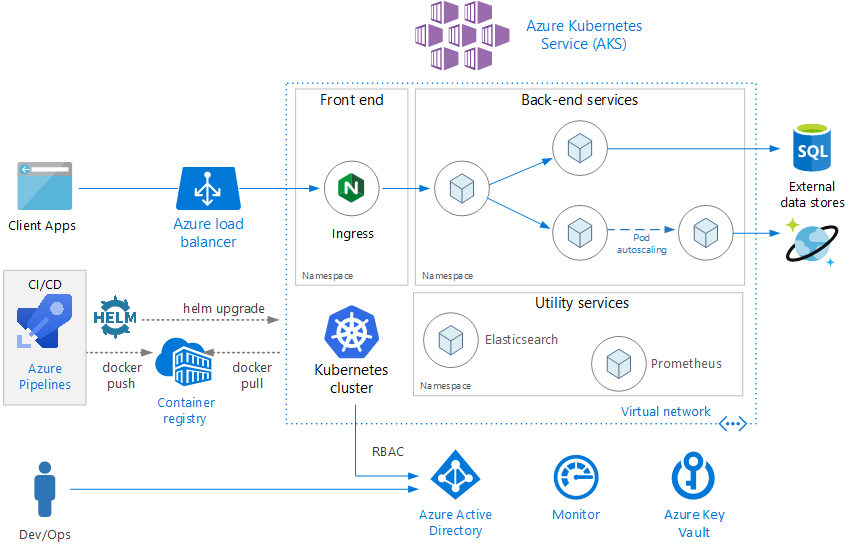
# Microservices architecture on Azure Kubernetes Service (AKS)

**Introduction**

This reference architectures shows how a microservices application deployed to Azure Kubernetes Service (AKS).

In this Document I have tried to explain the Azure services which can be integrated with AKS services and its Purpose of using it her.

# Reference Architecture



**Architecture**

The architecture consists of the following components.

* **Azure Kubernetes Service** (AKS). AKS is an Azure service that deploys a managed Kubernetes cluster.
* **Kubernetes cluster**. AKS is responsible for deploying the Kubernetes cluster and for managing the Kubernetes masters. We only manage the agent nodes.
* **Virtual network.** By default, AKS creates a virtual network to deploy the agent nodes into. which lets you control things like how the subnets are configured, on-premises connectivity, and IP addressing.
* **Ingress.** An ingress exposes HTTP(S)routes to services inside the cluster**.**

**External data stores**. Microservices are typically stateless and write state to external data stores, such as Azure SQL Database or Cosmos DB.

* **Azure Active Directory**. AKS uses an Azure Active Directory (Azure AD) identity to create and manage other Azure resources. Azure AD is also recommended for user authentication in client applications.
* **Azure Container Registry**. Its a Container Registry in Azure to store private Docker images. AKS can authenticate with Container Registry using its Azure AD identity. AKS can use other container registries, such as Docker Hub.
* **Azure Pipelines**. Pipelines is part of Azure DevOps Services and runs automated builds, tests, and deployments. You can also use third-party CI/CD solutions such as Jenkins.
* **Helm.** Helm is as a package manager for Kubernetes. It’s a way to bundle Kubernetes objects into a single unit that you can publish, deploy, version, and update.
* **Azure Monitor**. Azure Monitor collects and stores metrics and logs, including platform metrics for the Azure services in the solution and application telemetry. Azure Monitor integrates with AKS to collect metrics from controllers, nodes, and containers, as well as container logs and master node logs.
* **Ingress .**The Kubernetes **Ingress** resource type works in conjunction with an ingress controller. There are ingress controllers for Nginx, HAProxy, Traefik, and Application Gateway (preview), among others.

The Ingress Controller has access to the Kubernetes API, so it can make intelligent decisions about routing and load balancing. For example, the Nginx ingress controller bypasses the kube-proxy network proxy.

* **Data storage** Avoid storing persistent data in local cluster storage, because that ties the data to the node. Instead,Use an external service such as Azure SQL Database or Cosmos DB, *or* Mount a persistent volume using Azure Disks or Azure Files. Use Azure Files if the same volume needs to be shared by multiple pods.
* **Namespaces.** Every object in a Kubernetes cluster belongs to a namespace. By default, when you create a new object, it goes into the default namespace. But it's a good practice to create namespaces that are more descriptive to help Apply resource constraints to a namespace, so that the total set of pods assigned to that namespace cannot exceed the resource quota of the namespace. Apply policies at the namespace level, including RBAC and security policies.

**Scalability considerations**

AKS Kubernetes supports scale-out at two levels:

* Scale the number of pods allocated to a deployment.
* Scale the nodes in the cluster, to increase the total compute resources available to the cluster.

Although you can scale out pods and nodes manually, we recommend using autoscaling, to minimize the chance that services will become resource starved under high load. An autoscaling strategy must take both pods and nodes into account. If you just scale out the pods, eventually you will reach the resource limits of the nodes.

* **Pod autoscaling**

The **Horizontal Pod Autoscaler (HPA)** scales pods based on observed CPU, memory, or custom metrics. To configure horizontal pod scaling, you specify a target metric (for example, 70% of CPU), and the minimum and maximum number of replicas.

* Use readiness probes to let Kubernetes know when a new pod is ready to accept traffic.
* Use pod disruption budgets to limit how many pods can be evicted from a service at a time.
* **Cluster autoscaling**

The cluster autoscaler scales the number of nodes. If pods can't be scheduled because of resource constraints, the cluster autoscaler will provision more nodes. (Note: Integration between AKS and the cluster autoscaler is currently in preview.)

**Health probes**

Kubernetes defines two types of health probe that a pod can expose:

* Readiness probe: Tells Kubernetes whether the pod is ready to accept requests.
* Liveness probe: Tells Kubernetes whether a pod should be removed and a new instance started.

**Security considerations**

**Role based access control (RBAC)**

Kubernetes and Azure both have mechanisms for role-based access control (RBAC):

* Azure RBAC controls access to resources in Azure, Permissions can be assigned to users, groups, or service principals.
* Kubernetes RBAC controls permissions to the Kubernetes API. For example, creating pods and listing pods are actions that can be authorized (or denied) to a user through RBAC.
* To assign Kubernetes permissions to users, you create *roles* and *role bindings*:
  + A Role is a set of permissions that apply within a namespace. Permissions are defined as get, update, create, delete) on resources (pods, deployments, etc.).
  + A RoleBinding assigns users or groups to a Role.
  + There is also a ClusterRole object, which is like a Role but applies to the entire cluster, across all namespaces. To assign users or groups to a ClusterRole, create a ClusterRoleBinding.
* **Secrets management and application credentials**
* Azure Key Vault. In AKS, you can mount one or more secrets from Key Vault as a volume. The volume reads the secrets from Key Vault. The pod can then read the secrets just like a regular volume.
* The pod authenticates itself by using either a pod identity or by using an Azure AD Service Principal along with a client secret. Using pod identities is recommended because the client secret isn't needed in that case.
* Kubernetes secrets. Another option is simply to use Kubernetes secrets. This option is the easiest to configure but has some challenges. Secrets are stored in etcd, which is a distributed key-value store.
* Using a Azure Key Vault provides several advantages, such as:
* Centralized control of secrets.
* Ensuring that all secrets are encrypted at rest.
* Centralized key management.
* Access control of secrets.
* Auditing

**Pod and container security**

* Don't run containers in privileged mode. Privileged mode gives a container access to all devices on the host. You can set Pod Security Policy to disallow containers from running in privileged mode.
* When possible, avoid running processes as root inside containers. Containers do not provide complete isolation from a security standpoint, so it's better to run a container process as a non-privileged user.
* Store images in a trusted private registry, such as Azure Container Registry or Docker Trusted Registry.
* Ensure that pods can only pull images from the trusted registry.
* Scan images for known vulnerabilities, using a scanning solution such as Twistlock and Aqua, which are available through the Azure Marketplace.

**Deployment (CI/CD) considerations**

Here are some goals of a robust CI/CD process for a microservices architecture:

* Each team can build and deploy the services that it owns independently, without affecting or disrupting other teams.
* Before a new version of a service is deployed to production, it gets deployed to dev/test/QA environments for validation. Quality gates are enforced at each stage.
* A new version of a service can be deployed side-by-side with the previous version.
* Sufficient access control policies are in place.
* You can trust the container images that are deployed to production.

**Isolation of environments**

In Kubernetes, you have a choice between physical isolation and logical isolation. Physical isolation means deploying to separate clusters. Logical isolation makes use of namespaces and policies.

# Create AKS Cluster

**Sign in to Azure**

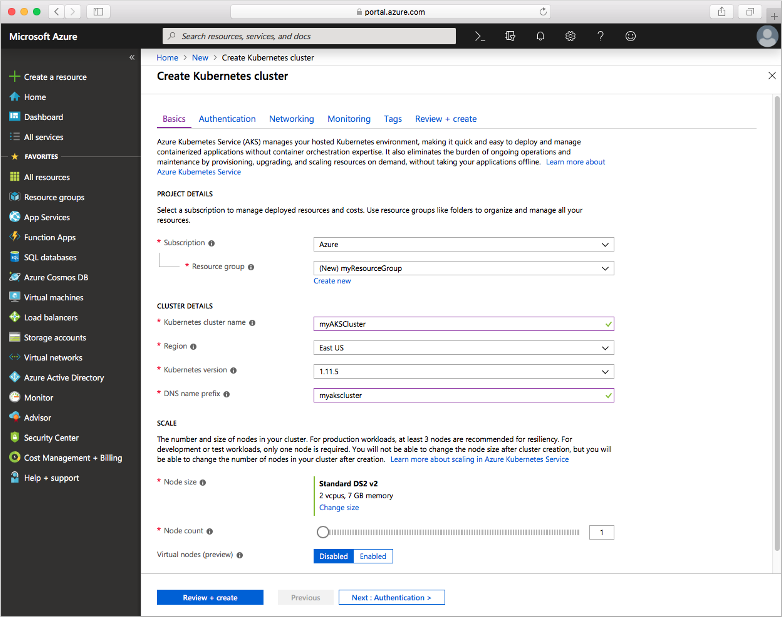
Sign in to the Azure portal at <https://portal.azure.com>.

**Create an AKS cluster**

In the top left-hand corner of the Azure portal, select **+ Create a resource** > **Kubernetes Service**.

To create an AKS cluster, complete the following steps:

1. **Basics** - Configure the following options:
   * *PROJECT DETAILS*: Select an Azure subscription, then select or create an Azure resource group, such as *myResourceGroup*. Enter a **Kubernetes cluster name**, such as *myAKSCluster*.
   * *CLUSTER DETAILS*: Select a region, Kubernetes version, and DNS name prefix for the AKS cluster.
   * *SCALE*: Select a VM size for the AKS nodes. The VM size **cannot** be changed once an AKS cluster has been deployed.
     + Select the number of nodes to deploy into the cluster. For this quickstart, set **Node count** to *1*. Node count **can** be adjusted after the cluster has been deployed.

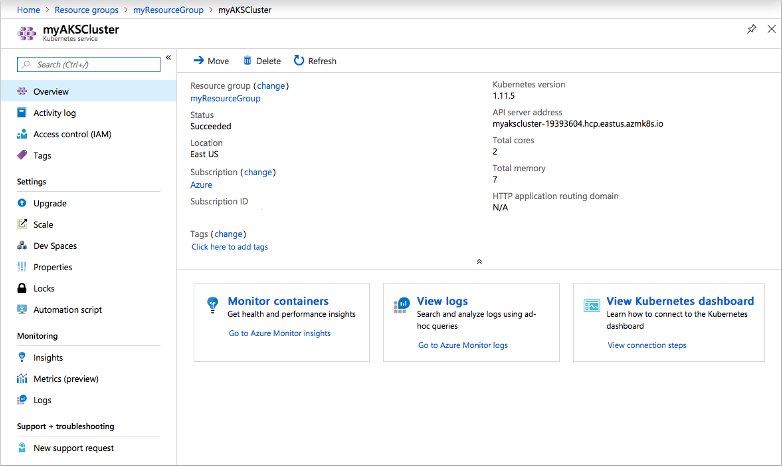


Select **Next: Authentication** when complete.

1. **Authentication**: Configure the following options:
   * Create a new service principal or *Configure* to use an existing one. When using an existing SPN, you need to provide the SPN client ID and secret.
   * Enable the option for Kubernetes role-based access controls (RBAC). These controls provide more fine-grained control over access to the Kubernetes resources deployed in your AKS cluster.

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

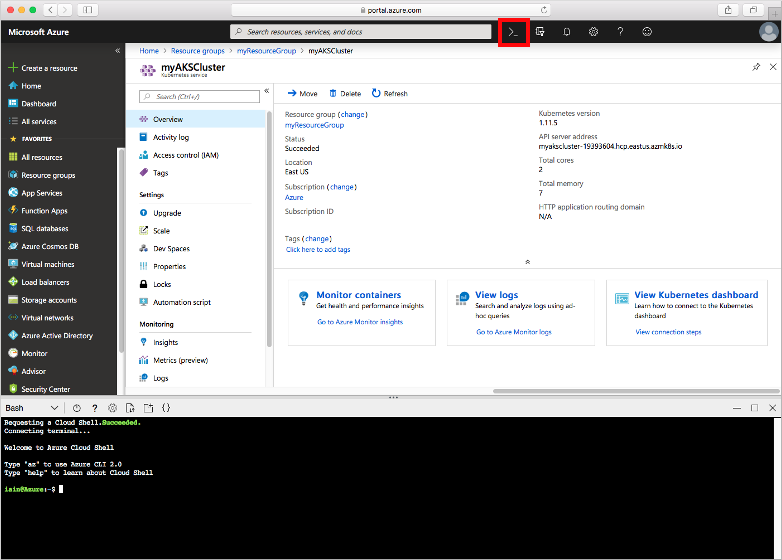
It takes a few minutes to create the AKS cluster and to be ready for use. When finished, browse to the AKS cluster resource group, such as *myResourceGroup*, and select the AKS resource, such as *myAKSCluster*. The AKS cluster dashboard is shown, as in the following example screenshot:



**Connect to the cluster**

To manage a Kubernetes cluster, you use [kubectl](https://kubernetes.io/docs/user-guide/kubectl/), the Kubernetes command-line client. The kubectl client is pre-installed in the Azure Cloud Shell.

Open Cloud Shell using the button on the top right-hand corner of the Azure portal.



To configure kubectl to connect to your Kubernetes cluster, use the [az aks get-credentials](https://docs.microsoft.com/en-us/cli/azure/aks?view=azure-cli-latest#az-aks-get-credentials) command. This command downloads credentials and configures the Kubernetes CLI to use them. The following example gets credentials for the cluster name *myAKSCluster* in the resource group named *myResourceGroup*:

Azure CLI

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

To verify the connection to your cluster, use the [kubectl get](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#get) command to return a list of the cluster nodes.

Azure CLI

kubectl get nodes

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

NAME STATUS ROLES AGE VERSION

aks-agentpool-14693408-0 Ready agent 15m v1.11.5