Azure Kubernetes Service (AKS) Secure Baseline Reference Implementation

Introduction

Azure Kubernetes Service (AKS) Secure Baseline Reference Implementation

This reference implementation demonstrates the *recommended starting (baseline) infrastructure architecture* for an AKS cluster. This is implementation and document is meant to guide an interdisciplinary team or multiple distinct teams like networking, security and development through the process of getting this secure baseline infrastructure deployed and understanding the components of it.

Introduction

Azure Architecture Center guidance

This project has a companion set of articles that describe challenges, design patterns, and best practices for a secure AKS cluster. You can find this article on the Azure Architecture Center at Baseline architecture for a secure AKS cluster. If you haven't reviewed it, we suggest you read it as it will give added context to the considerations applied in this implementation. Ultimately, this is the direct implementation of that specific architectural guidance.

The Architecture

Architecture

This architecture is infrastructure focused, more so than workload. It concentrates on the AKS cluster itself, including concerns with identity, post-deployment configuration, secret management, and network topologies.

The implementation presented here is the *minimum recommended baseline for most AKS clusters*. This implementation integrates with Azure services that will deliver observability, provide a network topology that will support multi-regional growth, and keep the in-cluster traffic secure as well. This architecture should be considered your starting point for pre-production and production stages.

Throughout the reference implementation, you will see reference to *Contoso Bicycle*. They are a fictional small and fast-growing startup that provides online web services to its clientele on the west coast of North America. They have no on-premises data centers and all their containerized line of business applications are now about to be orchestrated by secure, enterprise-ready AKS clusters. You can read more about their requirements and their IT team composition. This narrative provides grounding for some implementation details, naming conventions, etc. You should adapt as you see fit.

Finally, this implementation uses the ASP.NET Core Docker sample web app as an example workload. This workload purposefully uninteresting, as it is here exclusively to help you experience the baseline infrastructure.

Architecture

Core architecture components

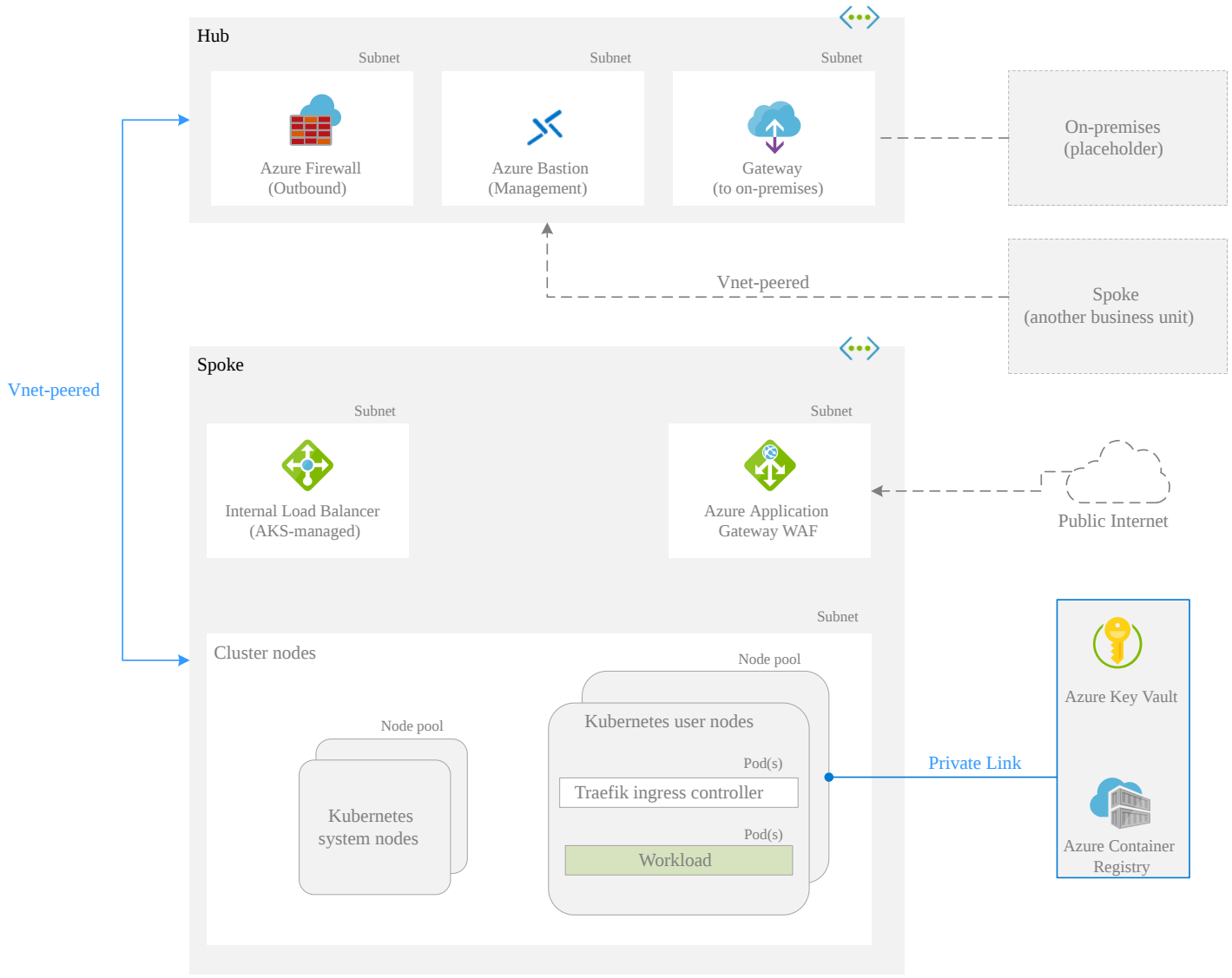
Azure platform

- AKS v1.19
  - System and User node pool separation
  - AKS-managed Azure AD
  - Managed Identities
  - Azure CNI
  - Azure Monitor for containers
- Azure Virtual Networks (hub-spoke)
- Azure Application Gateway (WAF)
- AKS-managed Internal Load Balancers
- Azure Firewall

In-cluster OSS components

- Flux GitOps Operator
- Traefik Ingress Controller
- Azure AD Pod Identity
- Azure KeyVault Secret Store CSI Provider
- Kured

Architecture



The Deployment

Deploy the reference implementation

A deployment of AKS-hosted workloads typically experiences a separation of duties and lifecycle management in the area of prerequisites, the host network, the cluster infrastructure, and finally the workload itself. This reference implementation is similar. Also, be aware our primary purpose is to illustrate the topology and decisions of a baseline cluster. We feel a "step-by-step" flow will help you learn the pieces of the solution and give you insight into the relationship between them. Ultimately, lifecycle/SDLC management of your cluster and its dependencies will depend on your situation (team roles, organizational standards, etc), and will be implemented as appropriate for your needs.

Please start this learning journey in the *Preparing for the cluster* section. If you follow this through the end, you'll have our recommended baseline cluster installed, with an end-to-end sample workload running for you to reference in your own Azure subscription.

In the following sections we will cover:

1. Preparing for the cluster
2. Build target network
3. Deploying the cluster
4. Deploy your workload
5. Validation

1. Preparing for the cluster

Deploy the reference implementation

1. Preparing for the cluster

There are considerations that must be addressed before you start deploying your cluster. Do I have enough permissions in my subscription and AD tenant to do a deployment of this size? How much of this will be handled by my team directly vs having another team be responsible?

- [ ] Begin by ensuring you install and meet the prerequisites
- [ ] Procure client-facing and AKS Ingress Controller TLS certificates
- [ ] Plan your Azure Active Directory integration

Prerequisites

This is the starting point for the instructions on deploying the AKS Secure Baseline reference implementation. There is required access and tooling you'll need in order to accomplish this. Follow the instructions below and on the subsequent pages so that you can get your environment ready to proceed with the AKS cluster creation.

Deploy the reference implementation

1. Preparing for the cluster

Prerequisites

Steps

1. An Azure subscription. If you don't have an Azure subscription, you can create a free account.

   Important: The user or service principal initiating the deployment process ***must*** have the following minimal set of Azure Role-Based Access Control (RBAC) roles and Azure AD permissions assigned:

   - Contributor role is ***required*** at the subscription level to have the ability to create resource groups and perform deployments.
   - User Access Administrator role is ***required*** at the subscription level since you'll be granting least-privilege RBAC access to resources as well as assigning built-in policy definitions to your AKS cluster.
     - One such example is detailed in the Container Insights documentation.

Deploy the reference implementation

1. Preparing for the cluster

Prerequisites

2. An Azure AD tenant to associate your Kubernetes RBAC configuration to.

    Note: The user or service principal initiating the deployment process *must* have the following minimal set of Azure AD permissions assigned:

    - Azure AD User Administrator is ***required*** to create a "break glass" AKS admin Active Directory Security Group and User. Alternatively, you could get your Azure AD admin to create this for you when instructed to do so.
        - If you are not part of the User Administrator group in the tenant associated to your Azure subscription, please consider creating a new tenant to use while evaluating this implementation.

3. Latest Azure CLI installed or you can perform this from Azure Cloud Shell by clicking below.

Deploy the reference implementation

1. Preparing for the cluster

Prerequisites

4. Clone/download this repo locally, or even better fork this repository.

    Note: If you have forked this reference implementation repo, you'll be able to customize some of the files and commands for a more personalized experience; also ensure references to repos mentioned are updated to use your own (e.g. the following `GITHUB_REPO`).

```
export GITHUB_REPO=https://github.com/mspnp/aks-secure-baseline.git
git clone $GITHUB_REPO
```

    Note: The steps shown here and elsewhere in the reference implementation use Bash shell commands. On Windows, you can use the Windows Subsystem for Linux to run Bash.

5. Ensure OpenSSL is installed in order to generate self-signed certs used in this implementation.

Deploy the reference implementation

1. Preparing for the cluster

Prerequisites

Next step

Generate your client-facing TLS certificate

Generate Your Client-Facing and AKS Ingress Controller TLS Certificates

Now that you have the prerequisites met, follow the steps below to create the TLS certificates that Azure Application Gateway will serve for clients connecting to your web app as well as the AKS Ingress Controller. If you already have access to an appropriate certificates, or can procure them from your organization, consider doing so and skipping the certificate generation steps. The following will describe using a self-signed certs for instructive purposes only.

Deploy the reference implementation

Steps

1. Generate a client-facing self-signed TLS certificate

   > Contoso Bicycle needs to procure a CA certificate for the web site. As this is going to be a user-facing site, they purchase an EV cert from their CA. This will serve in front of the Azure Application Gateway. They will also procure another one, a standard cert, to be used with the AKS Ingress Controller. This one is not EV, as it will not be user facing.

   Warning: Do not use the certificate created by this script for actual deployments. The use of self-signed certificates are provided for ease of illustration purposes only. For your cluster, use your organization's requirements for procurement and lifetime management of TLS certificates, *even for development purposes*.

   Create the certificate for Azure Application Gateway with a common name of `bicycle.contoso.com`.

   ```
   openssl req -x509 -nodes -days 365 -newkey rsa:2048 -out appgw.crt -keyout appgw.k
   openssl pkcs12 -export -out appgw.pfx -in appgw.crt -inkey appgw.key -passout pass
   ```

Deploy the reference implementation

1. Preparing for the cluster

Prerequisites

Generate Your Client-Facing and AKS Ingress Controller TLS Certificates

2. Base64 encode the client-facing certificate

Note: No matter if you used a certificate from your organization or you generated one from above, you'll need the certificate (as `.pfx`) to be base 64 encoded for proper storage in Key Vault later.

```
export APP_GATEWAY_LISTENER_CERTIFICATE=$(cat appgw.pfx | base64 | tr -d '\n')
```

3. Generate the wildcard certificate for the AKS Ingress Controller

> Note: Contoso Bicycle will also procure another TLS certificate, a standard cert, to be used with the AKS Ingress Controller. This one is not EV, as it will not be user facing. Finally the app team decides to use a wildcard certificate of `*.aks-ingress.contoso.com` for the ingress controller.

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -out traefik-ingress-internal-ak
```

Deploy the reference implementation

1. Preparing for the cluster

Prerequisites

Generate Your Client-Facing and AKS Ingress Controller TLS Certificates

4. Base64 encode the AKS Ingress Controller certificate

Note: No matter if you used a certificate from your organization or you generated one from above, you'll need the public certificate (as `.crt` or `.cer`) to be base 64 encoded for proper storage in Key Vault later.

```
export AKS_INGRESS_CONTROLLER_CERTIFICATE_BASE64=$(cat traefik-ingress-internal-aks-
```

Deploy the reference implementation

Next step

Prep for Azure Active Directory integration

Prep for Azure Active Directory Integration

In the prior step, you generated the user-facing TLS certificate, now we'll prepare for leveraging Azure AD for Kubernetes role-based access control (RBAC). This is the last of the cluster infrastructure prerequisites.

Deploy the reference implementation

Steps

Note: The Contoso Bicycle Azure AD team requires all admin access to AKS clusters be security-group based. This applies to the new Secure AKS cluster that is being built for Application ID a0008 under the BU001 business unit. Kubernetes RBAC will be AAD-backed and access granted based on a user's identity or directory group membership.

1. Query and save your Azure subscription tenant id

```
export TENANT_ID=$(az account show --query tenantId --output tsv)
```

2. Login into the tenant where you are a Azure AD User Administrator playing the role as the Contoso Bicycle Azure AD team

```
az login --tenant <replace-with-tenant-id-with-user-admin-permissions> --allow-no-
export K8S_RBAC_AAD_PROFILE_TENANTID=$(az account show --query tenantId --output t
```

Deploy the reference implementation

1. Preparing for the cluster

Prerequisites

Generate Your Client-Facing and AKS Ingress Controller TLS Certificates

Prep for Azure Active Directory Integration

3. Create the first the Azure AD group that is going to map the Kubernetes Cluster Role Admin. If you already have a security group that is appropriate for cluster admins, consider using that group and skipping this step. If using your own group, you will need to update group object names throughout the reference implementation.

```
export K8S_RBAC_AAD_PROFILE_ADMIN_GROUP_OBJECTID=$(az ad group create --display-name
```

4. Create a break-glass Cluster Admin user for your AKS cluster

> Note: The organization knows the value of having a break-glass admin user for their critical infrastructure. The app team requests a cluster admin user and Azure AD Admin team proceeds with the creation of the user from Azure AD.

```
export K8S_RBAC_AAD_PROFILE_TENANT_DOMAIN_NAME=$(az ad signed-in-user show --query
export AKS_ADMIN_OBJECTID=$(az ad user create --display-name=bu0001a0008-admin --use
```

Deploy the reference implementation

5. Add the new admin user to new security group so can be granted with the Kubernetes Cluster Admin role.

> Note: The recently created break-glass admin user is added to the Kubernetes Cluster Admin group from Azure AD. After this step the Azure AD Admin team will have finished the app team's request and the outcome are:
>
> - the new app team's user admin credentials
> - and the Azure AD group object ID

```
az ad group member add --group aad-to-bu0001a000800-cluster-admin --member-id $AKS_A
```

This object ID will be used later while creating the cluster. This way, once the cluster gets deployed the new group will get the proper Cluster Role bindings in Kubernetes.

Deploy the reference implementation

1. Preparing for the cluster

Prerequisites

Generate Your Client-Facing and AKS Ingress Controller TLS Certificates

Prep for Azure Active Directory Integration

6. Set up groups to map into other Kubernetes Roles. (Optional, fork required)

    Note: The team knows there will be more than just cluster admins that need group-managed access to the cluster. Out of the box, Kubernetes has other roles like *admin*, *edit*, and *view* which can also be mapped to Azure AD Groups.

In the `user-facing-cluster-role-aad-group.yaml` file, you can replace the four `<replace-with-an-aad-group-object-id-for-this-cluster-role-binding>` placeholders with corresponding new or existing AD groups that map to their purpose for this cluster.

Note: Alternatively, you can make these group associations to Azure RBAC roles. At the time of this writing, this feature is still in *preview*, but will become the preferred way of mapping identities to Kubernetes RBAC roles.

Deploy the reference implementation

Next step

Deploy the hub-spoke network topology

2. Build target network

Deploy the reference implementation

1. Preparing for the cluster

2. Build target network

Microsoft recommends AKS be deploy into a carefully planned network; sized appropriately for your needs and with proper network observability. Organizations typically favor a traditional hub-spoke model, which is reflected in this implementation. While this is a standard hub-spoke model, there are fundamental sizing and portioning considerations included that should be understood.

- [ ] Build the hub-spoke network

Deploy the Hub-Spoke Network Topology

The prerequisites for the AKS secure baseline cluster are now completed with Azure AD group and user work performed in the prior steps. Now we will start with our first Azure resource deployment, the network resources.

1. Login into the Azure subscription that you'll be deploying into.

   > Note: The networking team logins into the Azure subscription that will contain the regional hub. At Contoso Bicycle, all of their regional hubs are in the same, centrally-managed subscription.

```
az login --tenant $TENANT_ID
```

Deploy the reference implementation

2. Create the networking hubs resource group.

   Note: The networking team has all their regional networking hubs in the following resource group. The group's default location does not matter, as it's not tied to the resource locations. (This resource group would have already existed.)

```
# [This takes less than one minute to run.]
az group create --name rg-enterprise-networking-hubs --location centralus
```

3. Create the networking spokes resource group.

   Note: The networking team also keeps all of their spokes in a centrally-managed resource group. As with the hubs resource group, the location of this group does not matter and will not factor into where our network will live. (This resource group would have already existed.)

```
# [This takes less than minute to run.]
az group create --name rg-enterprise-networking-spokes --location centralus
```

Deploy the reference implementation

4. Create the regional network hub.

   Note: When the networking team created the regional hub for eastus2, it didn't have any spokes yet defined, yet the networking team always lays out a base hub following a standard pattern (defined in `hub-default.json`). A hub always contains an Azure Firewall (with some org-wide policies), Azure Bastion, a gateway subnet for VPN connectivity, and Azure Monitor for network observability. They follow Microsoft's recommended sizing for the subnets.

   The networking team has decided that `10.200.[0-9].0` will be where all regional hubs are homed on their organization's network space.

   Note: Azure Bastion and the On-Prem Connectivity is not actually deployed in this reference implementation, just the subnets for them are.

   In addition to the eastus2 regional hub (that you're deploying) you can assume there are similar deployed as well in in other Azure regions in this resource group.

```
# [This takes about five minutes to run.]
az deployment group create --resource-group rg-enterprise-networking-hubs --template
```

Continues on next slide

The hub creation will emit the following:

- `hubVnetId` - which you'll will need to know for future spokes that get created. E.g. `/subscriptions/[subscription id]/resourceGroups/rg-enterprise-networking-hubs/providers/Microsoft.Network/virtualNetworks/vnet-eastus2-hub`

Deploy the reference implementation

5. Create the spoke that will be home to the AKS cluster and its adjacent resources.

   Note: The networking team receives a request from an app team in business unit (BU) 0001 for a network spoke to house their new AKS-based application (Internally know as Application ID: A0008). The network team talks with the app team to understand their requirements and aligns those needs with Microsoft's best practices for a secure AKS cluster deployment. They capture those specific requirements and deploy the spoke, aligning to those specs, and connecting it to the matching regional hub.

```
# [This takes about ten minutes to run.]
HUB_VNET_ID=$(az deployment group show -g rg-enterprise-networking-hubs -n hub-defau
az deployment group create --resource-group rg-enterprise-networking-spokes --templa
```

Deploy the reference implementation

The spoke creation will emit the following:

- `appGwPublicIpAddress` - The Public IP address of the Azure Application Gateway (WAF) that will receive traffic for your workload.
- `clusterVnetResourceId` - The resource ID of the VNet that the cluster will land in. E.g. `/subscriptions/[subscription id]/resourceGroups/rg-enterprise-networking-spokes/providers/Microsoft.Network/virtualNetworks/vnet-hub-spoke-BU0001A0008-00`
- `nodepoolSubnetResourceIds` - An array containing the subnet resource IDs of the AKS node pools in the spoke. E.g. `["/subscriptions/[subscription id]/resourceGroups/rg-enterprise-networking-spokes/providers/Microsoft.Network/virtualNetworks/vnet-hub-spoke-BU0001A0008-00/subnets/snet-clusternodes"]`

Deploy the reference implementation

6. Update the shared, regional hub deployment to account for the requirements of the spoke.

   Note: Now that their hub has its first spoke, the hub can no longer run off of the generic hub template. The networking team creates a named hub template (e.g. `hub-eastus2.json`) to forever represent this specific hub and the features this specific hub needs in order to support its spokes' requirements. As new spokes are attached and new requirements arise for the regional hub, they will be added to this template file.

```
# [This takes about three minutes to run.]
NODEPOOL_SUBNET_RESOURCEIDS=$(az deployment group show -g rg-enterprise-networking-s
az deployment group create --resource-group rg-enterprise-networking-hubs --templat
```

   Note: At this point the networking team has delivered a spoke in which BU 0001's app team can lay down their AKS cluster (ID: A0008). The networking team provides the necessary information to the app team for them to reference in their Infrastructure-as-Code artifacts.

   Hubs and spokes are controlled by the networking team's GitHub Actions workflows. This automation is not included in this reference implementation as this body of work is focused on the AKS baseline and not the networking team's CI/CD practices.

Deploy the reference implementation

1. Preparing for the cluster

2. Build target network

Deploy the Hub-Spoke Network Topology

Next step

Deploy the AKS cluster

3. Deploying the cluster

Deploy the reference implementation

This is the heart of the guidance in this reference implementation; paired with prior network topology guidance. Here you will deploy the Azure resources for your cluster and the adjacent services such as Azure Application Gateway WAF, Azure Monitor, Azure Container Registry, and Azure Key Vault. This is also where you put the cluster under GitOps orchestration.

- [ ] Deploy the AKS cluster and supporting services
- [ ] Place the cluster under GitOps management

We perform the prior steps manually here for you to understand the involved components, but we advocate for an automated DevOps process. Therefore, incorporate the prior steps into your CI/CD pipeline, as you would any infrastructure as code (IaC). We have included a starter GitHub workflow that demonstrates this.

Deploy the AKS Cluster

Now that the hub-spoke network is provisioned, the next step in the AKS secure Baseline reference implementation is deploying the AKS cluster and its adjacent Azure resources.

Deploy the reference implementation

Steps

1. Create the AKS cluster resource group.

   Note: The app team working on behalf of business unit 0001 (BU001) is looking to create an AKS cluster of the app they are creating (Application ID: 0008). They have worked with the organization's networking team and have been provisioned a spoke network in which to lay their cluster and network-aware external resources into (such as Application Gateway). They took that information and added it to their `cluster-stamp.json` and `azuredeploy.parameters.prod.json` files.

   They create this resource group to be the parent group for the application.

```
# [This takes less than one minute.]
az group create --name rg-bu0001a0008 --location eastus2
```

Deploy the reference implementation

2. Get the AKS cluster spoke VNet resource ID.

> Note: The app team will be deploying to a spoke VNet, that was already provisioned by the network team.

```
TARGET_VNET_RESOURCE_ID=$(az deployment group show -g rg-enterprise-networking-spoke
```

3. Deploy the cluster ARM template.
Note: By default, this deployment will allow unrestricted access to your cluster's API Server. You can limit access to the API Server to a set of well-known IP addresses (i.,e. your hub firewall IP, bastion subnet, build agents, or any other networks you'll administer the cluster from) by setting the `clusterAuthorizedIPRanges` parameter in all deployment options.

Option 1 - Deploy in the Azure Portal

Use the following deploy to Azure button to create the baseline cluster from the Azure Portal. You'll need to provide the parameter values as returned from prior steps in this guide.

Deploy the reference implementation

Option 2 - Deploy from the command line

```
# [This takes about 15 minutes.]
az deployment group create --resource-group rg-bu0001a0008 --template-file cluster-s
```

Alteratively, you could have updated the `azuredeploy.parameters.prod.json` file and deployed as above, using `--parameters "@azuredeploy.parameters.prod.json"` instead of the individual key-value pairs.

Deploy the reference implementation

Option 3 - Automated deploy using GitHub Actions (fork is required)

I. Create the Azure Credentials for the GitHub CD workflow.

```
# Create an Azure Service Principal
az ad sp create-for-rbac --name "github-workflow-aks-cluster" --sdk-auth --s-assignment
export APP_ID=$(grep -oP '(?<="clientId": ").*?[^\\](?=",)' sp.json)

# Wait for propagation
 until az ad sp show --id ${APP_ID} &> /dev/null ; do echo "Waiting for Azure AD pagati

# Assign built-in Contributor RBAC role for creating resource groups and perming deploy
 az role assignment create --assignee $APP_ID --role 'Contributor'

# Assign built-in User Access Administrator RBAC role since granting RBAC acc to other
 az role assignment create --assignee $APP_ID --role 'User Access Administrator'
```

II. Create `AZURE_CREDENTIALS` secret in your GitHub repository. For more information, please take a look at Creating encrypted secrets for a repository

Note: Use the content from the `sp.json` file.

```
cat sp.json
```

Deploy the reference implementation

III. Create `APP_GATEWAY_LISTENER_CERTIFICATE_BASE64` secret in your GitHub repository. For more information, please take a look at Creating encrypted secrets for a repository.

Note:

- Use the env var value of `APP_GATEWAY_LISTENER_CERTIFICATE`
- Ideally fetching this secret from a platform-managed secret store such as Azure KeyVault

```
echo $APP_GATEWAY_LISTENER_CERTIFICATE
```

IV. Create `AKS_INGRESS_CONTROLLER_CERTIFICATE_BASE64` secret in your GitHub repository. For more information, please take a look at Creating encrypted secrets for a repository.

Note:

- Use the env var value of `AKS_INGRESS_CONTROLLER_CERTIFICATE_BASE64`
- Ideally fetching this secret from a platform-managed secret store such as Azure Key Vault

```
echo $AKS_INGRESS_CONTROLLER_CERTIFICATE_BASE64
```

Deploy the reference implementation

V. Copy the GitHub workflow file into the expected directory and update the placeholders in it.

```
mkdir -p .github/workflows
cat github-workflow/aks-deploy.yaml | \
    sed "s#<resource-group-location>#eastus2#g" | \
    sed "s#<resource-group-name>#rg-bu0001a0008#g" | \
    sed "s#<resource-group-localtion>#eastus2#g" | \
    sed "s#<geo-redundancy-location>#centralus#g" | \
    sed "s#<cluster-spoke-vnet-resource-id>#$TARGET_VNET_RESOURCE_ID#g" | \
    sed "s#<tenant-id-with-user-admin-permissions>#$K8S_RBAC_AAD_PROFILE_TENANTID#g" |
    sed "s#<azure-ad-aks-admin-group-object-id>#$K8S_RBAC_AAD_PROFILE_ADMIN_GROUP_OBJEC
    > .github/workflows/aks-deploy.yaml
```

VI. Push the changes to your forked repo.

> Note: The DevOps team wants to automate their infrastructure deployments. In this case, they decided to use GitHub Actions. They are going to create a workflow for every AKS cluster instance they have to take care of.

```
git add .github/workflows/aks-deploy.yaml && git commit -m "setup GitHub CD workflow"
git push origin HEAD:kick-off-workflow
```

Note: You might want to convert this GitHub workflow into a template since your organization or team might need to handle multiple AKS clusters. For more information: Sharing Workflow Templates within your organization.

Deploy the reference implementation

VII. Navigate to your GitHub forked repository and open a PR against `main` using the recently pushed changes to the remote branch `kick-off-workflow`.

> Note: The DevOps team configured the GitHub Workflow to preview the changes that will happen when a PR is opened. This will allow them to evaluate the changes before they get deployed. fter the PR reviewers see how resources will change if the AKS cluster ARM template gets deployed, it is possible to merge or discard the pull request. If the decision is made to merge, t will trigger a push event that will kick off the actual deployment process that consists of:
>
> - AKS cluster creation
> - Flux deployment

VIII. Once the GitHub Workflow validation finished successfully, please proceed by merging this PR into `main`.

> Note: The DevOps team monitors this Workflow execution instance. In this instance it will impact a critical piece of infrastructure as well as the management. This flow works for both new or an existing AKS cluster.

XI. The cluster is placed under GitOps managed as part of these GitHub Workflow steps. Therefore, you should proceed straight to Workflow Prerequisites.

Container registry note

Warning: To aid in ease of deployment of this cluster and your experimentation with workloads, Azure Policy is currently configured to allow your cluster to pull images from *public container registries* such as Docker Hub and Quay. For a production system, you'll want to update the Azure Policy named `pa-allowed-registries-images` in your `cluster-stamp.json` file to only list those container registries that you are willing to take a dependency on and what namespaces those policies apply to. This will protect your cluster from unapproved registries being used, which may prevent issues while trying to pull images from a registry which doesn't provide SLA guarantees for your deployment.

This deployment creates an SLA-backed Azure Container Registry for your cluster's needs. Your organization may have a central container registry for you to use, or your registry may be tied specifically to your application's infrastructure (as demonstrated in this implementation).

Only use container registries that satisfy the availability needs of your application.

Deploy the reference implementation

1. Preparing for the cluster

2. Build target network

3. Deploying the cluster

Deploy the AKS Cluster

Next step

Place the cluster under GitOps management

Place the Cluster Under GitOps Management

Now that the AKS cluster has been deployed, the next step to configure a GitOps management solution on our cluster, Flux in this case.

Deploy the reference implementation

Steps

GitOps allows a team to author Kubernetes manifest files, persist them in their git repo, and have them automatically apply to their cluster as changes occur. This reference implementation is focused on the baseline cluster, so Flux is managing cluster-level concerns . This is distinct from workload-level concerns, which would be possible as well to manage via Flux, and would typically be done by additional Flux operators in the cluster. The namespace `cluster-baseline-settings` will be used to provide a logical division of the cluster configuration from workload configuration. Examples of manifests that are applied:

- Cluster Role Bindings for the AKS-managed Azure AD integration
- AAD Pod Identity
- CSI driver and Azure KeyVault CSI Provider
- the workload's namespace named `a0008`

Deploy the reference implementation

1. Install `kubectl` 1.19 or newer. (`kubctl` supports +/-1 Kubernetes version.)

```
sudo az aks install-cli
kubectl version --client
```

2. Get the cluster name.

```
export AKS_CLUSTER_NAME=$(az deployment group show --resource-group rg-bu0001a0008
```

Deploy the reference implementation

3. Get AKS `kubectl` credentials (as a user that has admin permissions to the cluster).

> In the Azure Active Directory Integration step, we placed our cluster under AAD group-backed RBAC. This is the first time we are seeing this used. `az aks get-credentials` allows you to use `kubectl` commands against your cluster. Without the AAD integration, you'd have to use `--admin` here, which isn't what we want to happen.
>
> Here, you'll log in with a user that has been added to the Azure AD security group used to back the Kubernetes RBAC admin role. Executing the command below will invoke the AAD login process to auth the *user of your choice*, which will then be checked against Kubernets RBAC to perform the action. The user you choose to log in with *must be a member of the AAD group bound* to the `cluster-admin` ClusterRole.
>
> For simplicity could either use the "break-glass" admin user created in Azure Active Directory Integration (`bu0001a0008-admin`) or any user you assign to the `cluster-admin` group assignment in your `user-facing-cluster-role-aad-group.yaml` file. If you skipped those steps you can use `--admin` to proceed, but proper AAD group-based RBAC access is a critical security function that you should invest time in setting up.

```
az aks get-credentials -g rg-bu0001a0008 -n $AKS_CLUSTER_NAME
```

Deploy the reference implementation

4. Create the cluster baseline settings namespace.

```
# Verify the user you logged in with has the appropriate permissions, should result
# If you receive "yes" to this command, check which user you authenticated as and er
# assigned to the Azure AD Group you designated for cluster admins.
kubectl auth can-i create namespace -A

kubectl create namespace cluster-baseline-settings
```

5. Import cluster management images to your container registry.

   Public container registries are subject to faults such as outages (no SLA) or request throttling. Interruptions like these can be crippling for a system that needs to pull an image *right now*. To minimize the risks of using public registries, store all applicable container images in a registry that you control, such as the SLA-backed Azure Container Registry.

```
# Get your ACR cluster name
export ACR_NAME=$(az deployment group show --resource-group rg-bu0001a0008 -n cluste

# Import cluster management images hosted in public container registries
az acr import --source docker.io/library/memcached:1.5.20 -n $ACR_NAME
az acr import --source docker.io/fluxcd/flux:1.19.0 -n $ACR_NAME
az acr import --source docker.io/weaveworks/kured:1.4.0 -n $ACR_NAME
```

Deploy the reference implementation

6. Deploy Flux.

   If you used your own fork of this GitHub repo, update the `flux.yaml` file to include reference to your own repo and change the URL below to point to yours as well. Also, since Flux will begin processing the manifests in `cluster-baseline-settings/` now would be a good time to:

   - update the `<replace-with-an-aad-group-object-id-for-this-cluster-role-binding>` placeholder in `user-facing-cluster-role-aad-group.yaml` with the Object IDs for the Azure AD group(s) you created for management purposes. If you don't, the manifest will still apply, but AAD integration will not be mapped to your specific AAD configuration.
   - Update three `image` manifest references to your container registry instead of the default public container registry. See comment in each file for instructions.
     - update the two `image:` values in `flux.yaml`.
     - update the one `image:` values in `kured-1.4.0-dockerhub.yaml`.

Deploy the reference implementation

Warning: Deploying the flux configuration using the `flux.yaml` file unmodified from this repo will be deploying your cluster to take dependencies on public container registries. This is generally okay for exploratory/testing, but not suitable for production. Before going to production, ensure *all* image references are from *your* container registry (as imported in the prior step) or another that you feel confident relying on.

```
kubectl apply -f https://raw.githubusercontent.com/mspnp/aks-secure-baseline/main/cl
```

Deploy the reference implementation

7. Wait for Flux to be ready before proceeding.

```
kubectl wait --namespace cluster-baseline-settings --for=condition=ready pod --selec
```

Generally speaking, this will be the last time you should need to use `kubectl` for day-to-day configuration operations on this cluster (outside of break-fix situations). Between ARM for Azure Resource definitions and the application of manifests via Flux, all normal configuration activities can be performed without the need to use `kubectl`. You will however see us use it for the upcoming workload deployment. This is because the SDLC component of workloads are not in scope for this reference implementation, as this is focused the infrastructure and baseline configuration.

Next step

Prepare for the workload by installing its prerequisites

4. Deploy your workload

Deploy the reference implementation

Without a workload deployed to the cluster it will be hard to see how these decisions come together to work as a reliable application platform for your business. The deployment of this workload would typically follow a CI/CD pattern and may involve even more advanced deployment strategies (blue/green, etc). The following steps represent a manual deployment, suitable for illustration purposes of this infrastructure.

- [ ] Just like the cluster, there are workload prerequisites to address
- [ ] Configure AKS Ingress Controller with Azure Key Vault integration
- [ ] Deploy the workload

Workload Prerequisites

The AKS Cluster has been enrolled in GitOps management, wrapping up the infrastructure focus of the AKS secure Baseline reference implementation. Follow the steps below to import the TLS certificate that the Ingress Controller will serve for Application Gateway to connect to your web app.

Steps

Import the wildcard certificate for the AKS Ingress Controller to Azure Key Vault

> Note:: Contoso Bicycle procured a CA certificate, a standard one, to be used with the AKS Ingress Controller. This one is not EV, as it will not be user facing.

1. Obtain the Azure Key Vault details and give the current user permissions to import certificates.

   > Note:: Finally the app team decides to use a wildcard certificate of `*.aks-ingress.contoso.com` for the ingress controller. They use Azure Key Vault to import and manage the lifecycle of this certificate.

```
KEYVAULT_NAME=$(az deployment group show --resource-group rg-bu0001a0008 -n cluste
az keyvault set-policy --certificate-permissions import list get --upn $(az accoun
```

Deploy the reference implementation

2. Import the AKS Ingress Controller's Wildcard Certificate for `*.aks-ingress.contoso.com`.

Warning: If you already have access to an appropriate certificate, or can procure one from your organization, consider using it for this step. For more information, please take a look at the import certificate tutorial using Azure Key Vault.

Warning: Do not use the certificate created by this script for actual deployments. The use of self-signed certificates are provided for ease of illustration purposes only. For your cluster, use your organization's requirements for procurement and lifetime management of TLS certificates, ***even for development purposes***.

```
cat traefik-ingress-internal-aks-ingress-contoso-com-tls.crt traefik-ingress-interna
az keyvault certificate import -f traefik-ingress-internal-aks-ingress-contoso-com-t
```

3. Remove Azure Key Vault import certificates permissions for current user.

> The Azure Key Vault Policy for your user was a temporary policy to allow you to upload the certificate for this walkthrough. In actual deployments, you would manage these access policies via your ARM templates using Azure RBAC for Key Vault data plane.

```
az keyvault delete-policy --upn $(az account show --query user.name -o tsv) -n $KEYV
```

Check Azure Policies are in place

Note: The app team wants to apply Azure Policy over their cluster like they do other Azure resources. Their pods will be covered using the Azure Policy add-on for AKS. Some of these audits might end up in the denial of a specific Kubernetes API request operation to ensure the pod's specification is compliance with the organization's security best practices. Moreover data is generated by Azure Policy to assist the app team in the process of assessing the current compliance state of the AKS cluster.

The app team is going to assign at the resource group level the Azure Policy for Kubernetes built-in restricted initiative as well as five more built-in individual Azure policies that enforce that pods perform resource requests, define trusted container registries, allow root filesystem access in read-only mode, enforce the usage of internal load balancers, and enforce https-only Kuberentes Ingress objects.

1. Confirm policies are applied to the AKS cluster

```
kubectl get constrainttemplate
```

Deploy the reference implementation

A similar output as the one showed below should be returned

```
NAME                                        AGE
k8sazureallowedcapabilities                 21m
k8sazureallowedseccomp                      21m
... more ...
k8sazurereadonlyrootfilesystem              21m
k8sazurevolumetypes                         21m
```

Deploy the reference implementation

1. Preparing for the cluster

2. Build target network

3. Deploying the cluster

4. Deploy your workload

Workload Prerequisites

Next step

Configure AKS Ingress Controller with Azure Key Vault integration

Configure AKS Ingress Controller with Azure Key Vault integration

Previously you have configured workload prerequisites. These steps configure Traefik, the AKS ingress solution used in this reference implementation, so that it can securely expose the web app to your Application Gateway.

Deploy the reference implementation

Steps

1. Get the AKS Ingress Controller Managed Identity details

```
export TRAEFIK_USER_ASSIGNED_IDENTITY_RESOURCE_ID=$(az deployment group show --res
export TRAEFIK_USER_ASSIGNED_IDENTITY_CLIENT_ID=$(az deployment group show --resou
```

2. Ensure Flux has created the following namespace

```
# press Ctrl-C once you receive a successful response
kubectl get ns a0008 -w
```

Deploy the reference implementation

3. Create Traefik's Azure Managed Identity binding

Create the Traefik Azure Identity and the Azure Identity Binding to let Azure Active Directory Pod Identity to get tokens on behalf of the Traefik's User Assigned Identity and later on assign them to the Traefik's pod.

```
cat <<EOF | kubectl apply -f -
apiVersion: "aadpodidentity.k8s.io/v1"
kind: AzureIdentity
metadata:
  name: aksic-to-keyvault-identity
  namespace: a0008
spec:
  type: 0
  resourceID: $TRAEFIK_USER_ASSIGNED_IDENTITY_RESOURCE_ID
  clientID: $TRAEFIK_USER_ASSIGNED_IDENTITY_CLIENT_ID
---
apiVersion: "aadpodidentity.k8s.io/v1"
kind: AzureIdentityBinding
metadata:
  name: aksic-to-keyvault-identity-binding
  namespace: a0008
spec:
  azureIdentity: aksic-to-keyvault-identity
  selector: traefik-ingress-controller
EOF
```

4. Create the Traefik's Secret Provider Class resource

The Ingress Controller will be exposing the wildcard TLS certificate you created in a prior step. It uses the Azure Key Vault CSI Provider to mount the certificate which is managed and stored in Azure Key Vault. Once mounted, Traefik can use it.

Create a `SecretProviderClass` resource with with your Azure Key Vault parameters for the Azure Key Vault Provider for Secrets Store CSI driver.

Deploy the reference implementation

1. Preparing for the cluster

2. Build target network

3. Deploying the cluster

4. Deploy your workload

Workload Prerequisites

Configure AKS Ingress Controller with Azure Key Vault integration

```
cat <<EOF | kubectl apply -f -
apiVersion: secrets-store.csi.x-k8s.io/v1alpha1
kind: SecretProviderClass
metadata:
  name: aks-ingress-contoso-com-tls-secret-csi-akv
  namespace: a0008
spec:
  provider: azure
  parameters:
    usePodIdentity: "true"
    keyvaultName: "${KEYVAULT_NAME}"
    objects:  |
      array:
        - |
          objectName: traefik-ingress-internal-aks-ingress-contoso-com-tls
          objectAlias: tls.crt
          objectType: cert
        - |
          objectName: traefik-ingress-internal-aks-ingress-contoso-com-tls
          objectAlias: tls.key
          objectType: secret
    tenantId: "${TENANT_ID}"
EOF
```

Deploy the reference implementation

5.Import the Traefik container image to your container registry

> Public container registries are subject to faults such as outages (no SLA) or request throttling. Interruptions like these can be crippling for an application that needs to pull an image *right now*. To minimize the risks of using public registries, store all applicable container images in a registry that you control, such as the SLA-backed Azure Container Registry.

```
# Get your ACR cluster name
export ACR_NAME=$(az deployment group show --resource-group rg-bu0001a0008 -n cluster

# Import ingress controller image hosted in public container registries
az acr import --source docker.io/library/traefik:2.2.1 -n $ACR_NAME
```

Deploy the reference implementation

6. Install the Traefik Ingress Controller

> Install the Traefik Ingress Controller; it will use the mounted TLS certificate provided by the CSI driver, which is the in-cluster secret management solution.
>
> If you used your own fork of this GitHub repo, update the one `image:` value in `traefik.yaml` to reference your container registry instead of the default public container registry and change the URL below to point to yours as well.

Warning: Deploying the traefik `traefik.yaml` file unmodified from this repo will be deploying your workload to take dependencies on a public container registry. This is generally okay for learning/testing, but not suitable for production. Before going to production, ensure *all* image references are from *your* container registry or another that you feel confident relying on.

```
kubectl apply -f https://raw.githubusercontent.com/mspnp/aks-secure-baseline/main/workl
```

Deploy the reference implementation

7. Wait for Traefik to be ready

   During Traefik's pod creation process, AAD Pod Identity will need to retrieve token for Azure Key Vault. This process can take time to complete and it's possible for the pod volume mount to fail during this time but the volume mount will eventually succeed. For more information, please refer to the Pod Identity documentation.

```
kubectl wait --namespace a0008 --for=condition=ready pod --selector=app.kubernetes.io/r
```

Deploy the reference implementation

Next step

Deploy the Workload

Deploy the Workload (ASP.NET Core Docker web app)

The cluster now has an Traefik configured with a TLS certificate. The last step in the process is to deploy the workload, which will demonstrate the system's functions.

Deploy the reference implementation

Steps

Note: The Contoso app team is about to conclude this journey, but they need an app to test their new infrastructure. For this task they've picked out the venerable ASP.NET Core Docker sample web app.

1. Deploy the ASP.NET Core Docker sample web app

   The workload definition demonstrates the inclusion of a Pod Disruption Budget rule, ingress configuration, and pod (anti-)affinity rules for your reference.

```
kubectl apply -f https://raw.githubusercontent.com/mspnp/aks-secure-baseline/main/
```

2. Wait until is ready to process requests running

```
kubectl wait --namespace a0008 --for=condition=ready pod --selector=app.kubernetes
```

Deploy the reference implementation

3. Check your Ingress resource status as a way to confirm the AKS-managed Internal Load Balancer is functioning

> In this moment your Ingress Controller (Traefik) is reading your ingress resource object configuration, updating its status, and creating a router to fulfill the new exposed workloads route. Please take a look at this and notice that the address is set with the Internal Load Balancer IP from the configured subnet.

```
kubectl get ingress aspnetapp-ingress -n a0008
```

> At this point, the route to the workload is established, SSL offloading configured, and a network policy is in place to only allow Traefik to connect to your workload. Therefore, please expect a `403` HTTP response if you attempt to connect to it directly.

4. Give a try and expect a `403` HTTP response

```
kubectl -n a0008 run -i --rm --tty curl --image=mcr.microsoft.com/powershell --limit
```

Deploy the reference implementation

1. Preparing for the cluster

2. Build target network

3. Deploying the cluster

4. Deploy your workload

Workload Prerequisites

Configure AKS Ingress Controller with Azure Key Vault integration

Next step

End-to-End Validation

5. Validation

Deploy the reference implementation

Now that the cluster and the sample workload is deployed; now it's time to look at how the cluster is functioning.

- [ ] Perform end-to-end deployment validation

End-to-End Validation

Now that you have a workload deployed, the ASP.NET Core Docker sample web app, you can start validating and exploring this reference implementation of the AKS secure baseline cluster. In addition to the workload, there are some observability validation you can perform as well.

Validate the Web App

This section will help you to validate the workload is exposed correctly and responding to HTTP requests.

Deploy the reference implementation

Steps

1. Get Public IP of Application Gateway

   Note: The app team conducts a final acceptance test to be sure that traffic is flowing end-to-end as expected, so they place a request against the Azure Application Gateway endpoint.

```
# query the Azure Application Gateway Public Ip
export APPGW_PUBLIC_IP=$(az deployment group show --resource-group rg-enterprise-n
```

2. Create `A` Record for DNS

   Note: You can simulate this via a local hosts file modification. You're welcome to add a real DNS entry for your specific deployment's application domain name, if you have access to do so.

   Map the Azure Application Gateway public IP address to the application domain name. To do that, please edit your hosts file (`C:\Windows\System32\drivers\etc\hosts` or `/etc/hosts`) and add the following record to the end: `${APPGW_PUBLIC_IP} bicycle.contoso.com`

3. Browse to the site (e.g. https://bicycle.contoso.com).

   Note: A TLS warning will be present due to using a self-signed certificate

Validate Web Application Firewall functionality

Your workload is placed behind a Web Application Firewall (WAF), which has rules designed to stop intentionally malicious activity. You can test this by triggering one of the built-in rules with a request that looks malcious.

> Note: This reference implementation enables the built-in OWASP 3.0 ruleset, in **Prevention** mode.

Steps

1. Browse to the site with the following appended to the URL: `?sql=DELETE%20FROM` (e.g. https://bicycle.contoso.com/?sql=DELETE%20FROM).
2. Observe that your request was blocked by Application Gateway's WAF rules and your workload never saw this potentially dangerous request.

3. Blocked requests (along with other gateway data) will be visable in the attached Log Analytics workspace. Execute the following query to show WAF logs, for example.

```
AzureDiagnostics
| where ResourceProvider == "MICROSOFT.NETWORK" and Category == "ApplicationGatewa
```

Validate Azure Monitor Insights and Logs

Monitoring your cluster is critical, especially when you're running a production cluster. Azure Monitor is configured to surface cluster logs, here you can see those logs as they are generated. Azure Monitor for containers is configured on this cluster for this purpose.

Deploy the reference implementation

Steps

1. In the Azure Portal, navigate to your AKS cluster resource.
2. Click *Insights* to see see captured data.

You can also execute queries on the cluster logs captured.

1. In the Azure Portal, navigate to your AKS cluster resource.
2. Click *Logs* to see and query log data. Note: There are several examples on the *Kubernetes Services* category.

Validate Azure Monitor for containers (Prometheus Metrics)

Azure Monitor is configured to scrape Prometheus metrics in your cluster. This reference implementation is configured to collect Prometheus metrics from two namespaces, as configured in `container-azm-ms-agentconfig.yaml`. There are two pods configured to emit Prometheus metrics:

- Treafik (in the `a0008` namespace)
- Kured (in the `cluster-baseline-settings` namespace)

Deploy the reference implementation

Steps

1. In the Azure Portal, navigate to your AKS cluster resource group (`rg-bu0001a0008`).
2. Select your Log Analytic Workspace resource.
3. Click *Saved Searches*.

   Note: This reference implementation ships with some saved queries as an example of how you can write your own and manage them via ARM templates.

4. Type *Prometheus* in the filter.
5. You are able to select and execute the saved query over the scraped metrics.

Validate Workload Logs

The example workload uses the standard dotnet logger interface, which are captured in `ContainerLogs` in Azure Monitor. You could also include additional logging and telemetry frameworks in your workload, such as Application Insights. Here are the steps to view the built-in application logs.

Deploy the reference implementation

Steps

1. In the Azure Portal, navigate to your AKS cluster resource group (`rg-bu0001a0008`).
2. Select your Log Analytic Workspace resource.

3. Execute the following query

```
let podInventory = KubePodInventory
| where ContainerName endswith "aspnetcore-webapp-sample"
| distinct ContainerID, ContainerName
| project-rename Name=ContainerName;
ContainerLog
| project-away Name
| join kind=inner
   podInventory
on ContainerID
| project TimeGenerated, LogEntry, Computer, Name, ContainerID
| order by TimeGenerated desc
```

Validate Azure Alerts

Azure will generate alerts on the health of your cluster and adjacent resources. This reference implementation sets up an alert that all you need to do it subscribe to.

Steps

An alert based on Azure Monitor for containers information using a Kusto query was configured in this reference implementation.

1. In the Azure Portal, navigate to your AKS cluster resource group (`rg-bu0001a0008`).
2. Select *Alerts*, then *Manage Rule Alerts*.
3. There is an alert called "PodFailedScheduledQuery" that will be triggered based on the custom query response.

An Azure Advisor Alert was configured as well in this reference implementation.

1. In the Azure Portal, navigate to your AKS cluster resource group (`rg-bu0001a0008`).
2. Select *Alerts*, then *Manage Rule Alerts*.
3. There is an alert called "AllAzureAdvisorAlert" that will be triggered based on new Azure Advisor alerts.

A series of metric alerts were configured as well in this reference implementation.

1. In the Azure Portal, navigate to your AKS cluster resource group (`rg-bu0001a0008`).
2. Select your cluster, then *Insights*.
3. Select *Recommended alerts* to see those enabled. (Feel free to enable/disable as you see fit.)

Validate Azure Container Registry Image Pulls

If you configured your third-party images to be pulled from your Azure Container Registry vs public registries, you can validate that the container registry logs show `Pull` logs for your cluster when you applied your flux configuration.

Steps

1. In the Azure Portal, navigate to your AKS cluster resource group (`rg-bu0001a0008`) and then your Azure Container Registry instances (starts with `acraks`).
2. Select *Logs*.
3. Execute the following query, for whatever time range is appropriate.

```
ContainerRegistryRepositoryEvents
| where OperationName == 'Pull'
```

4. You should see logs for CSI, flux, kured, memcached, and traefik. You'll see multiple for some as the image was pulled to multiple nodes to satisfy ReplicaSet/DaemonSet placement.

Deploy the reference implementation

1. Preparing for the cluster

2. Build target network

3. Deploying the cluster

4. Deploy your workload

5. Validation

End-to-End Validation

Next step

Clean Up Azure Resources

Cleaning Up

Clean up resources

Most of the Azure resources deployed in the prior steps will incur ongoing charges unless removed.

- [ ] Cleanup all resources

Clean up resources

After you are done exploring your deployed AKS secure baseline cluster, you'll want to delete the created Azure resources to prevent undesired costs from accruing. Follow these steps to delete all resources created as part of this reference implementation.

Clean up resources

Steps

1. Delete the resource groups as a way to delete all contained Azure resources.

   To delete all Azure resources associated with this reference implementation, you'll need to delete the three resource groups created.

   Warning: Ensure you are using the correct subscription, and validate that the only resources that exist in these groups are ones you're okay deleting.

   ```
   az group delete -n rg-bu0001a0008
   az group delete -n rg-enterprise-networking-spokes
   az group delete -n rg-enterprise-networking-hubs
   ```

2. Purge Azure Key Vault

   Because this reference implementation enables soft delete on Key Vault, execute a purge so your next deployment of this implementation doesn't run into a naming conflict.

   ```
   az keyvault purge --name ${KEYVAULT_NAME} --location eastus2
   ```

3. If any temporary changes were made to Azure AD or Azure RBAC permissions consider removing those as well.

Scripts

Inner-loop development scripts

We have provided some sample deployment scripts that you could adapt for your own purposes while doing a POC/spike on this. Those scripts are found in the inner-loop-scripts directory. They include some additional considerations and may include some additional narrative as well. Consider checking them out. They consolidate most of the walk-through performed above into combined execution steps.

Preview Features

Preview features

While this reference implementation tends to avoid *preview* features of AKS to ensure you have the best customer support experience; there are some features you may wish to evaluate in pre-production clusters that augment your posture around security, manageability, etc. Consider trying out and providing feedback on the following. As these features come out of preview, this reference implementation may be updated to incorporate them.

- Azure RBAC for Kubernetes Authentication - An extension of the Azure AD integration already in this reference implementation. Allowing you to bind Kubernetes authentication to Azure RBAC role assignments.
- Host-based encryption - Leverages added data encryption on your VMs' temp and OS disks.
- Generation 2 VM support - Increased memory options, Intel SGX support, and UEFI-based boot architectures.
- Auto Upgrade Profile support
- Customizable Node & Kublet config
- GitOps as an add-on
- Azure AD Pod Identity as an add-on

Advanced Topics

Advanced topics

This reference implementation intentionally does not cover more advanced scenarios. For example topics like the following are not addressed:

- Cluster lifecycle management with regard to SDLC and GitOps
- Workload SDLC integration (including concepts like Bridge to Kubernetes, advanced deployment techniques, etc)
- Mapping decisions to CIS benchmark controls
- Container security
- Multi-region clusters
- Advanced regulatory compliance (FinServ)
- Multiple (related or unrelated) workloads owned by the same team
- Multiple workloads owned by disparate teams (AKS as a shared platform in your organization)
- Cluster-contained state (PVC, etc)
- Windows node pools
- Scale-to-zero node pools and event-based scaling (KEDA)
- Private Kubernetes API Server
- Terraform
- Bedrock
- dapr

Keep watching this space, as we build out reference implementation guidance on topics such as these. Further guidance delivered will use this baseline AKS implementation as their starting point. If you would like to contribute or suggest a pattern built on this baseline, please get in touch.

Final thoughts

Kubernetes is a very flexible platform, giving infrastructure and application operators many choices to achieve their business and technology objectives. At points along your journey, you will need to consider when to take dependencies on Azure platform features, OSS solutions, support channels, regulatory compliance, and operational processes. **We encourage this reference implementation to be the place for you to *start* architectural conversations within your own team; adapting to your specific requirements, and ultimately delivering a solution that delights your customers.**

Related documentation

- Azure Kubernetes Service Documentation
- Microsoft Azure Well-Architected Framework
- Microservices architecture on AKS

Contributions

Please see our contributor guide.

This project has adopted the Microsoft Open Source Code of Conduct. For more information see the Code of Conduct FAQ or contact opencode@microsoft.com with any additional questions or comments.

With :heart: from Microsoft Patterns & Practices, Azure Architecture Center.