

# Contents

## Container Instances

### Overview

- About Azure Container Instances

### Quickstarts

- Deploy a container instance - CLI

- Deploy a container instance - Portal

- Deploy a container instance - PowerShell

### Tutorials

- Create and deploy container image

  - 1 - Create container image

  - 2 - Create container registry

  - 3 - Deploy application

- Deploy a multi-container group

  - Deploy a container group - YAML

  - Deploy a container group - Resource Manager

### Samples

- Code samples

- Resource Manager templates

### Concepts

- Container groups

- Quotas and limits

- Region availability

- Relationship to orchestrators

### How-to guides

- Deploy

  - Deploy in a virtual network (preview)

  - Deploy from Azure Container Registry

- Container scenarios

  - Set restart policy for run-once tasks

Set environment variables (env)

Override container entrypoint

Execute a command (exec)

Run Jenkins build job

Use a managed identity (preview)

Use GPU resources (preview)

## Mount data volumes

Azure file share

Secret

Empty directory

Cloned Git repo

## Manage running containers

Configure liveness probes

Stop and start containers

Update running containers

## Monitor and log

Monitor CPU and memory usage

Get container logs and events

Logging with Azure Log Analytics

## Troubleshoot

Troubleshoot common issues

## Reference

Azure CLI

REST

PowerShell

.NET

Python

Java

Node.js

Resource Manager template

## Resources

Build your skills with Microsoft Learn

[Region availability](#)

[Pricing](#)

[Roadmap](#)

[Provide product feedback](#)

[Stack Overflow](#)

[Videos](#)

# What is Azure Container Instances?

3/26/2019 • 2 minutes to read • [Edit Online](#)

Containers are becoming the preferred way to package, deploy, and manage cloud applications. Azure Container Instances offers the fastest and simplest way to run a container in Azure, without having to manage any virtual machines and without having to adopt a higher-level service.

Azure Container Instances is a great solution for any scenario that can operate in isolated containers, including simple applications, task automation, and build jobs. For scenarios where you need full container orchestration, including service discovery across multiple containers, automatic scaling, and coordinated application upgrades, we recommend [Azure Kubernetes Service \(AKS\)](#).

## Fast startup times

Containers offer significant startup benefits over virtual machines (VMs). Azure Container Instances can start containers in Azure in seconds, without the need to provision and manage VMs.

## Public IP connectivity and DNS name

Azure Container Instances enables exposing your containers directly to the internet with an IP address and a fully qualified domain name (FQDN). When you create a container instance, you can specify a custom DNS name label so your application is reachable at `customlabel.azureregion.azurecontainer.io`.

## Hypervisor-level security

Historically, containers have offered application dependency isolation and resource governance but have not been considered sufficiently hardened for hostile multi-tenant usage. Azure Container Instances guarantees your application is as isolated in a container as it would be in a VM.

## Custom sizes

Containers are typically optimized to run just a single application, but the exact needs of those applications can differ greatly. Azure Container Instances provides optimum utilization by allowing exact specifications of CPU cores and memory. You pay based on what you need and get billed by the second, so you can fine-tune your spending based on actual need.

For compute-intensive jobs such as machine learning, Azure Container Instances can schedule Linux containers to use NVIDIA Tesla [GPU resources](#) (preview).

## Persistent storage

To retrieve and persist state with Azure Container Instances, we offer direct [mounting of Azure Files shares](#).

## Linux and Windows containers

Azure Container Instances can schedule both Windows and Linux containers with the same API. Simply specify the OS type when you create your [container groups](#).

Some features are currently restricted to Linux containers:

- Multiple containers per container group

- Volume mounting ([Azure Files](#), [emptyDir](#), [GitRepo](#), [secret](#))
- [Resource usage metrics](#) with Azure Monitor
- [Virtual network deployment](#) (preview)
- [GPU resources](#) (preview)

Azure Container Instances currently supports Windows Server 2016 images based on Long-Term Servicing Channel (LTSC) versions. Windows Semi-Annual Channel (SAC) releases like 1709 and 1803 are unsupported.

## Co-scheduled groups

Azure Container Instances supports scheduling of [multi-container groups](#) that share a host machine, local network, storage, and lifecycle. This enables you to combine your main application container with other supporting role containers, such as logging sidecars.

## Virtual network deployment (preview)

Currently in preview, this feature of Azure Container Instances enables [deployment of container instances into an Azure virtual network](#). By deploying container instances into a subnet within your virtual network, they can communicate securely with other resources in the virtual network, including those that are on premises (through [VPN gateway](#) or [ExpressRoute](#)).

### IMPORTANT

Certain features of Azure Container Instances are in preview, and some [limitations apply](#). Previews are made available to you on the condition that you agree to the [supplemental terms of use](#). Some aspects of these features may change prior to general availability (GA).

## Next steps

Try deploying a container to Azure with a single command using our quickstart guide:

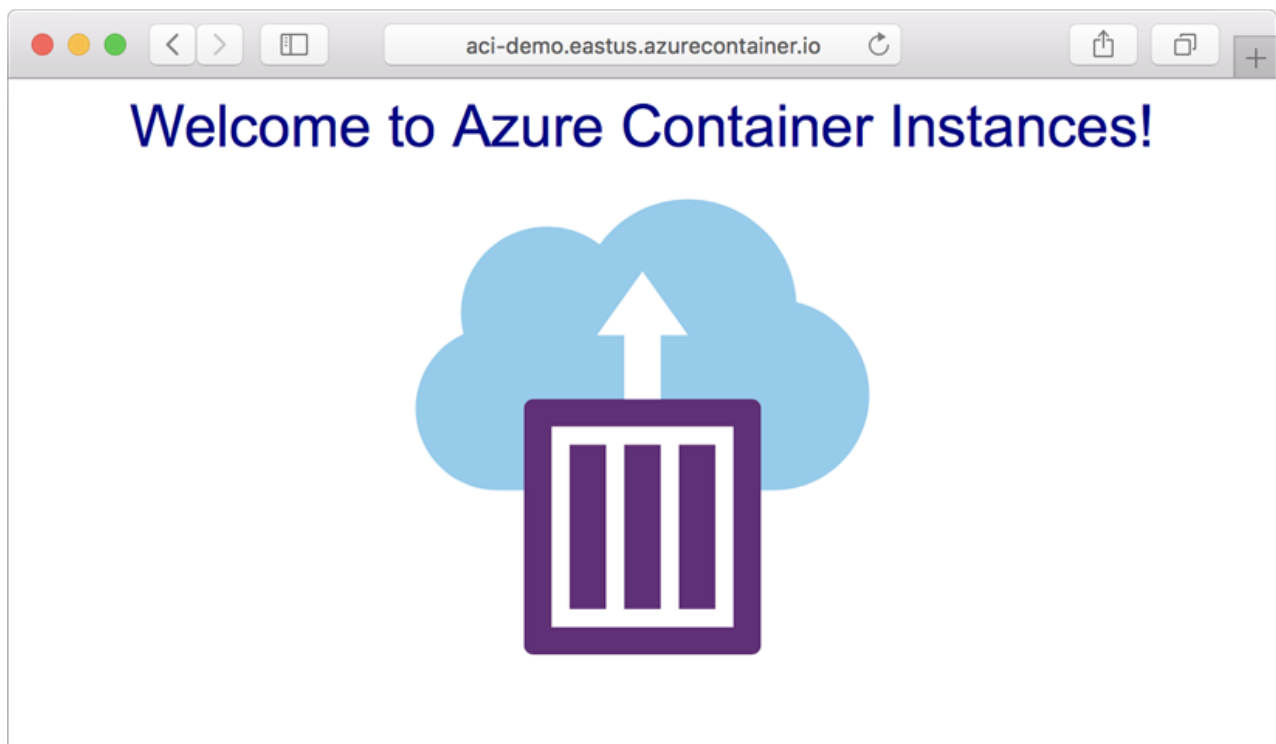
[Azure Container Instances Quickstart](#)

# Quickstart: Deploy a container instance in Azure using the Azure CLI

3/22/2019 • 5 minutes to read • [Edit Online](#)

Use Azure Container Instances to run serverless Docker containers in Azure with simplicity and speed. Deploy an application to a container instance on-demand when you don't need a full container orchestration platform like Azure Kubernetes Service.




In this quickstart, you use the Azure CLI to deploy an isolated Docker container and make its application available with a fully qualified domain name (FQDN). A few seconds after you execute a single deployment command, you can browse to the application running in the container:



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

## Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Select **Copy** to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select <b>Try It</b> in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu in the upper-right corner of the <a href="#">Azure portal</a> .	

You can use the Azure Cloud Shell or a local installation of the Azure CLI to complete this quickstart. If you'd like to use it locally, version 2.0.55 or later is recommended. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create a resource group

Azure container instances, like all Azure resources, must be deployed into a resource group. Resource groups allow you to organize and manage related Azure resources.

First, create a resource group named *myResourceGroup* in the *eastus* location with the following [az group create](#) command:

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

## Create a container

Now that you have a resource group, you can run a container in Azure. To create a container instance with the Azure CLI, provide a resource group name, container instance name, and Docker container image to the [az container create](#) command. In this quickstart, you use the public `mcr.microsoft.com/azuredocs/aci-helloworld` image. This image packages a small web app written in Node.js that serves a static HTML page.

You can expose your containers to the internet by specifying one or more ports to open, a DNS name label, or both. In this quickstart, you deploy a container with a DNS name label so that the web app is publicly reachable.

Execute a command similar to the following to start a container instance. Set a `--dns-name-label` value that's unique within the Azure region where you create the instance. If you receive a "DNS name label not available" error message, try a different DNS name label.

```
az container create --resource-group myResourceGroup --name mycontainer --image  
mcr.microsoft.com/azuredocs/aci-helloworld --dns-name-label aci-demo --ports 80
```

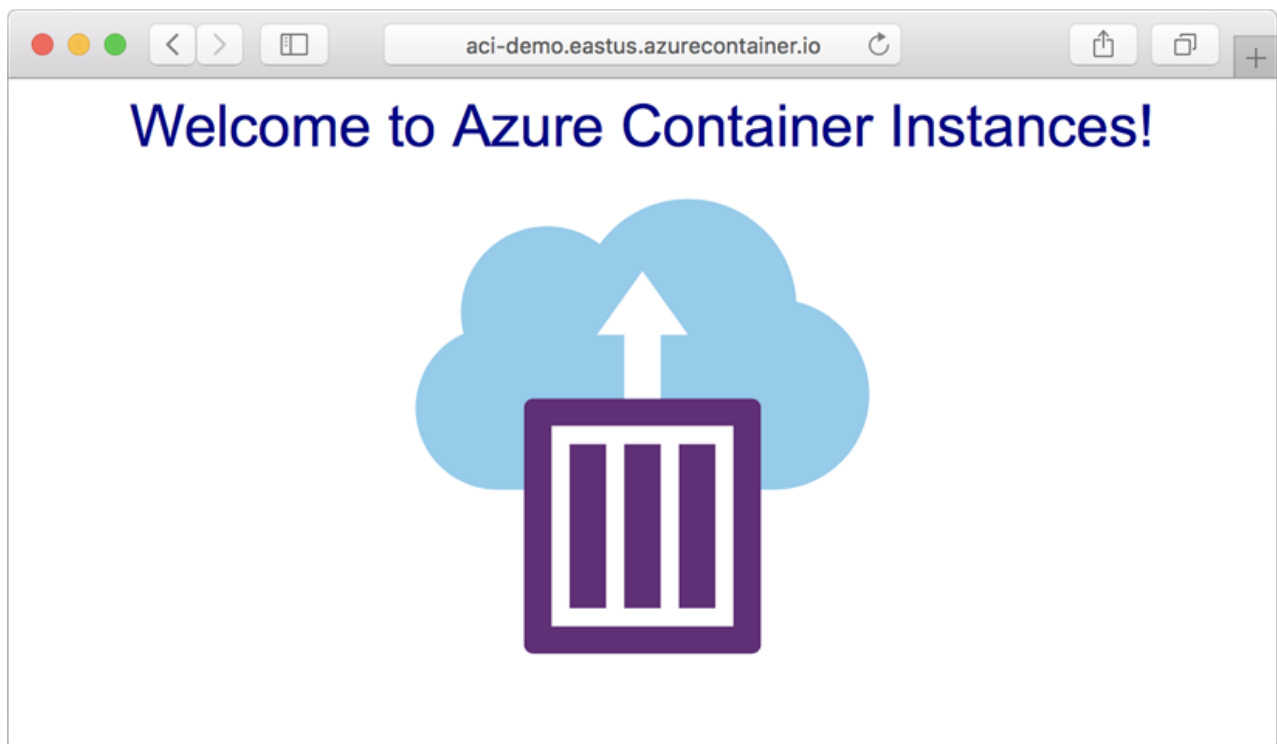
Within a few seconds, you should get a response from the Azure CLI indicating that the deployment has completed. Check its status with the [az container show](#) command:

```
az container show --resource-group myResourceGroup --name mycontainer --query "  
{FQDN:ipAddress.fqdn,ProvisioningState:provisioningState}" --out table
```

When you run the command, the container's fully qualified domain name (FQDN) and its provisioning state are displayed.

```
$ az container show --resource-group myResourceGroup --name mycontainer --query "  
{FQDN:ipAddress.fqdn,ProvisioningState:provisioningState}" --out table  
FQDN                               ProvisioningState  
-----  
aci-demo.eastus.azurecontainer.io  Succeeded
```

If the container's `ProvisioningState` is **Succeeded**, navigate to its FQDN in your browser. If you see a web page similar to the following, congratulations! You've successfully deployed an application running in a Docker container to Azure.



If at first the application isn't displayed, you might need to wait a few seconds while DNS propagates, then try refreshing your browser.

## Pull the container logs

When you need to troubleshoot a container or the application it runs (or just see its output), start by viewing the container instance's logs.

Pull the container instance logs with the [az container logs](#) command:

```
az container logs --resource-group myResourceGroup --name mycontainer
```

The output displays the logs for the container, and should show the HTTP GET requests generated when you viewed the application in your browser.

```
$ az container logs --resource-group myResourceGroup --name mycontainer
listening on port 80
::ffff:10.240.255.55 - - [21/Mar/2019:17:43:53 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
::ffff:10.240.255.55 - - [21/Mar/2019:17:44:36 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
::ffff:10.240.255.55 - - [21/Mar/2019:17:44:36 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
```

## Attach output streams

In addition to viewing the logs, you can attach your local standard out and standard error streams to that of the container.

First, execute the [az container attach](#) command to attach your local console to the container's output streams:

```
az container attach --resource-group myResourceGroup --name mycontainer
```

Once attached, refresh your browser a few times to generate some additional output. When you're done, detach



your console with `Control+C`. You should see output similar to the following:

```
$ az container attach --resource-group myResourceGroup --name mycontainer
Container 'mycontainer' is in state 'Running'...
(count: 1) (last timestamp: 2019-03-21 17:27:20+00:00) pulling image "mcr.microsoft.com/azuredocs/aci-helloworld"
(count: 1) (last timestamp: 2019-03-21 17:27:24+00:00) Successfully pulled image "mcr.microsoft.com/azuredocs/aci-helloworld"
(count: 1) (last timestamp: 2019-03-21 17:27:27+00:00) Created container
(count: 1) (last timestamp: 2019-03-21 17:27:27+00:00) Started container

Start streaming logs:
listening on port 80

::ffff:10.240.255.55 - - [21/Mar/2019:17:43:53 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
::ffff:10.240.255.55 - - [21/Mar/2019:17:44:36 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
::ffff:10.240.255.55 - - [21/Mar/2019:17:44:36 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
::ffff:10.240.255.55 - - [21/Mar/2019:17:47:01 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
::ffff:10.240.255.56 - - [21/Mar/2019:17:47:12 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
```

## Clean up resources

When you're done with the container, remove it using the [az container delete](#) command:

```
az container delete --resource-group myResourceGroup --name mycontainer
```

To verify that the container has been deleted, execute the [az container list](#) command:

```
az container list --resource-group myResourceGroup --output table
```

The **mycontainer** container should not appear in the command's output. If you have no other containers in the resource group, no output is displayed.

If you're done with the *myResourceGroup* resource group and all the resources it contains, delete it with the [az group delete](#) command:

```
az group delete --name myResourceGroup
```

## Next steps

In this quickstart, you created an Azure container instance by using a public Microsoft image. If you'd like to build a container image and deploy it from a private Azure container registry, continue to the [Azure Container Instances tutorial](#).

[Azure Container Instances tutorial](#)

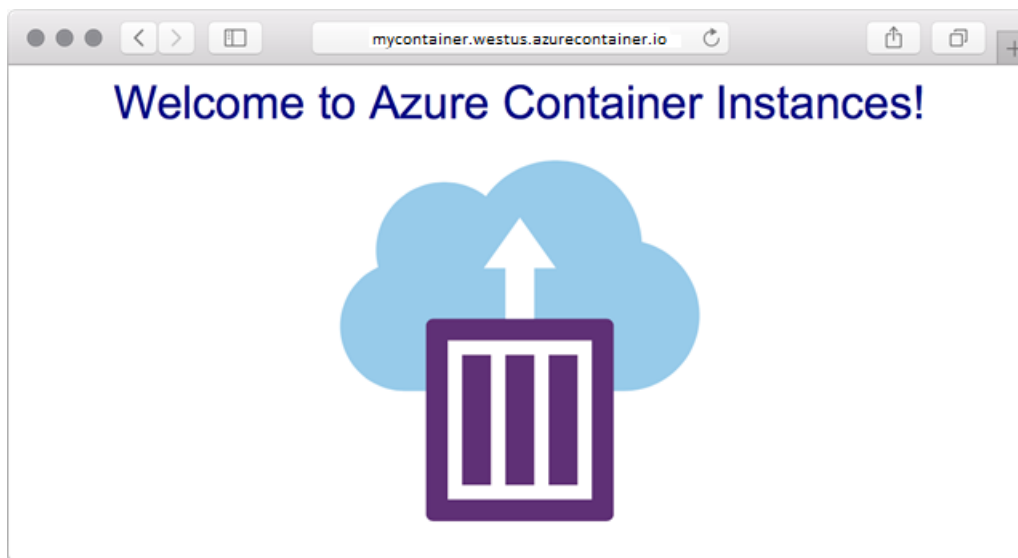
To try out options for running containers in an orchestration system on Azure, see the [Azure Kubernetes Service \(AKS\) quickstarts](#).

# Quickstart: Deploy a container instance in Azure using the Azure portal

4/18/2019 • 2 minutes to read • [Edit Online](#)

Use Azure Container Instances to run serverless Docker containers in Azure with simplicity and speed. Deploy an application to a container instance on-demand when you don't need a full container orchestration platform like Azure Kubernetes Service.

In this quickstart, you use the Azure portal to deploy an isolated Docker container and make its application available with a fully qualified domain name (FQDN). After configuring a few settings and deploying the container, you can browse to the running application:



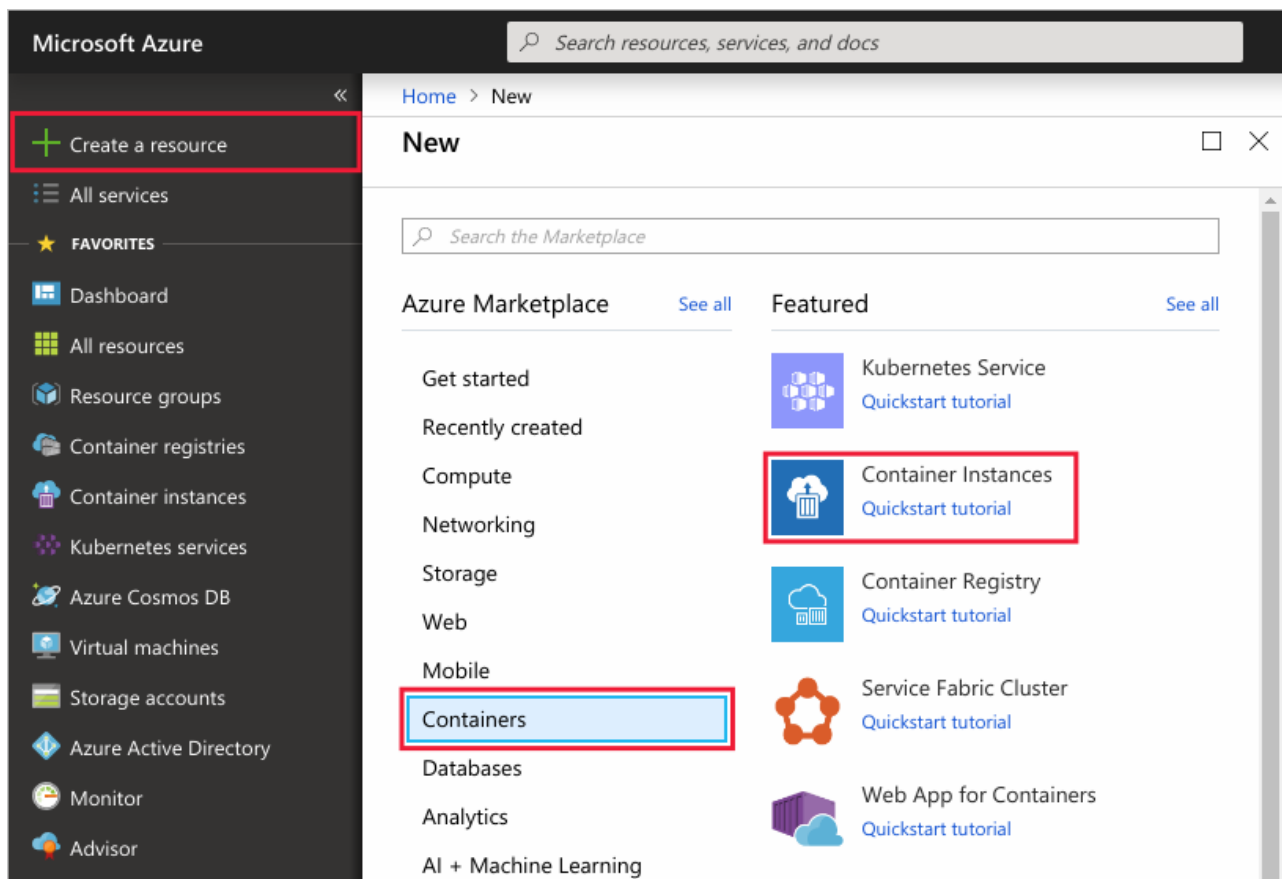
## Sign in to Azure

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

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

## Create a container instance

Select the **Create a resource** > **Containers** > **Container Instances**.



On the **Basics** page, enter the following values in the **Resource group**, **Container name**, and **Container image** text boxes. Leave the other values at their defaults, then select **OK**.

- Resource group: **Create new** > `myresourcegroup`
- Container name: `mycontainer`
- Container image: `mcr.microsoft.com/azuredocs/aci-helloworld`

[Home](#) > [New](#) > Create container instance

Create container instance

[Basics](#) [Networking](#) [Advanced](#) [Tags](#) [Review + create](#)

Azure Container Instances (ACI) allows you to quickly and easily run containers on Azure without managing servers or having to learn new tools. ACI offers per-second billing to minimize the cost of running containers on the cloud. [Learn more about Azure Container Instances](#)

#### PROJECT DETAILS

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

\* Subscription ⓘ

Visual Studio Enterprise

\* Resource group ⓘ

(New) myresourcegroup

[Create new](#)

#### CONTAINER DETAILS

\* Container name ⓘ

mycontainer

\* Region ⓘ

West US

\* Image type ⓘ

☒ Public ☐ Private

\* Image name ⓘ

mcr.microsoft.com/azuredocs/aci-helloworld

\* OS type

☒ Linux ☐ Windows

\* Size ⓘ

1 vcpu, 1.5 GB memory, 0 gpus

[Change size](#)

Review + create

Previous

Next : Networking >

For this quickstart, you use the default **Image type** setting of **Public** to deploy the public Microsoft `aci-helloworld` image. This Linux image packages a small web app written in Node.js that serves a static HTML page.

On the **Networking** page, specify a **DNS name label** for your container. The name must be unique within the Azure region where you create the container instance. Your container will be publicly reachable at `<dns-name-label>.<region>.azurecontainer.io`. If you receive a "DNS name label not available" error message, try a different DNS name label.

Home > New > Create container instance

Create container instance

Basics Networking Advanced Tags Review + create

You can configure networking settings for your container, such as ports and protocols as well as a DNS name label. If you choose not to include a public IP address, you will still be able to access your container and logs using the command line.  
[Learn more about Azure Container Instances networking](#)

Include public IP address ☒ Yes ☐ No

Ports ⓘ

PORTS	PORTS PROTOCOL	
80	TCP	🗑️
<input type="text"/>	<input type="text"/>	

DNS name label ⓘ

mycontainer

✓

.westus.azurecontainer.io

Review + create

Previous

Next : Advanced >

Leave the other settings at their defaults, then select **Review + create**.

When the validation completes, you're shown a summary of the container's settings. Select **Create** to submit your container deployment request.

Home > New > Create container instance

Create container instance

✓ Validation passed

Basics Networking Advanced Tags Review + create

**BASICS**

Subscription

Visual Studio Enterprise

Resource group

(new) myresourcegroup

Region

West US

Container name

mycontainer

Image type

Public

Image name

mcr.microsoft.com/azuredocs/aci-helloworld

OS type

Linux

Memory (GB)

1.5

Number of CPU cores

1

GPU type

None

Number of GPU cores

0

**NETWORKING**

Include public IP address

Yes

Ports

80 (TCP)

DNS name label

mycontainer

**ADVANCED**

Restart policy

On failure

**TAGS**

(none)

Create

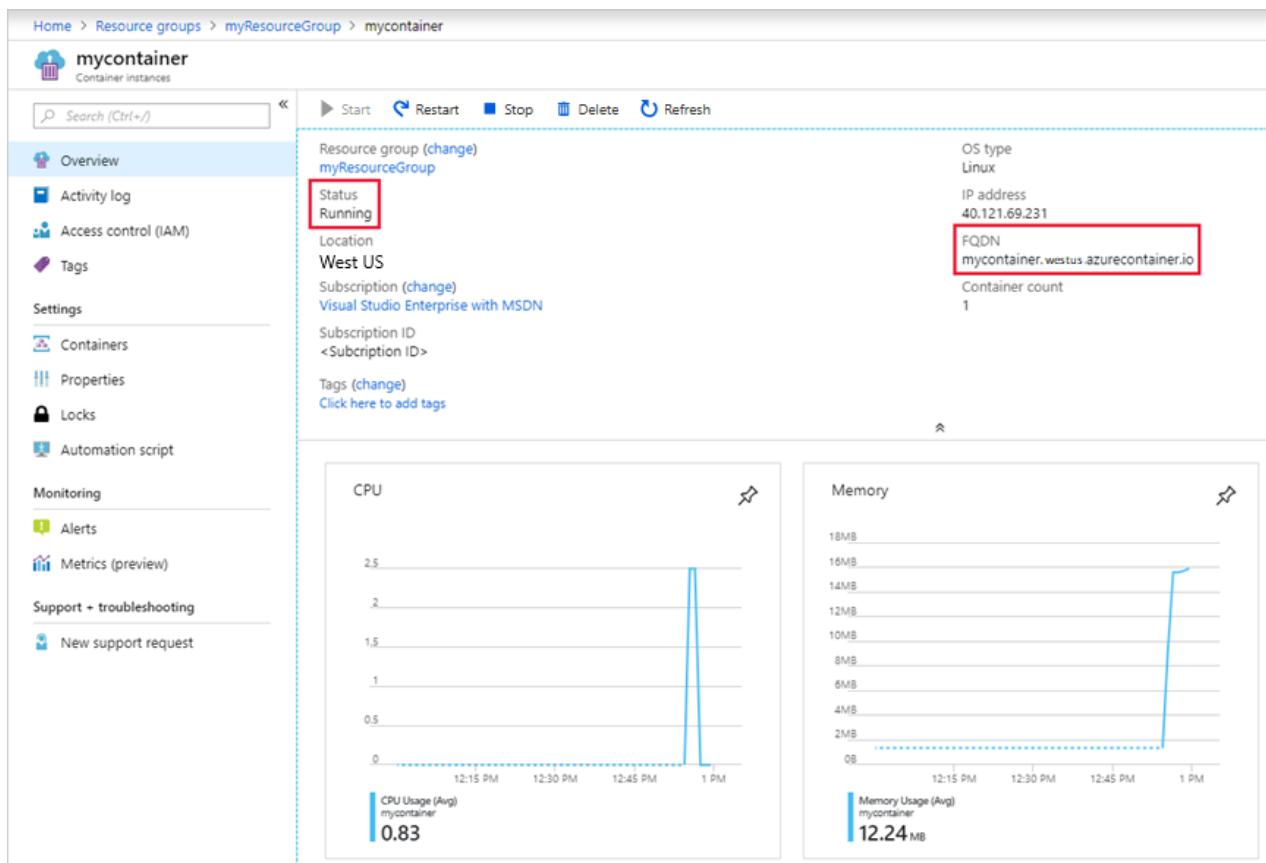
Previous

Next

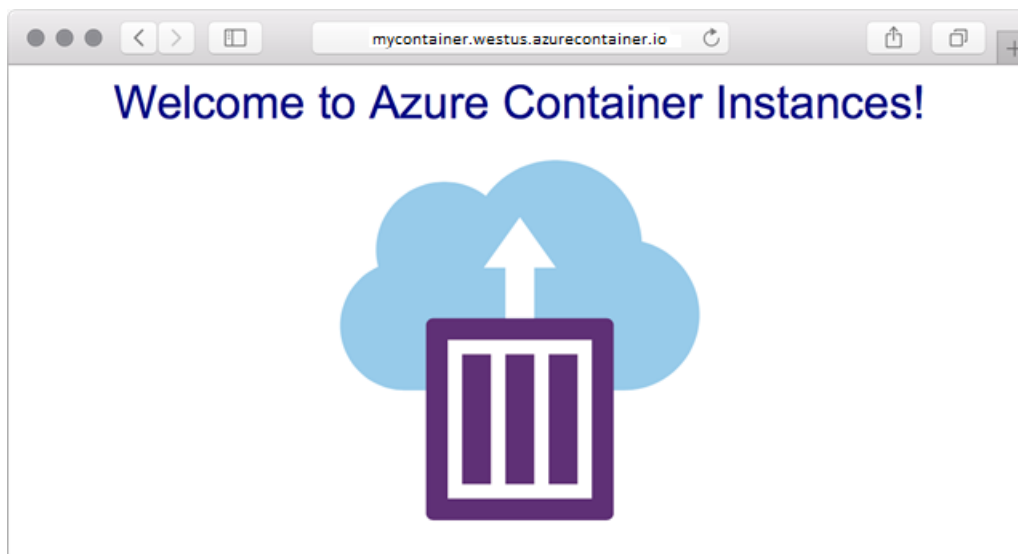
Download a template for automation

When deployment starts, a notification appears indicating the deployment is in progress. Another notification is displayed when the container group has been deployed.

Open the overview for the container group by navigating to **Resource Groups > myresourcegroup > mycontainer**. Take note of the **FQDN** (the fully qualified domain name) of the container instance, as well its **Status**.



Once its **Status** is *Running*, navigate to the container's FQDN in your browser.



Congratulations! By configuring just a few settings, you've deployed a publicly accessible application in Azure Container Instances.

## View container logs

Viewing the logs for a container instance is helpful when troubleshooting issues with your container or the application it runs.

To view the container's logs, under **Settings**, select **Containers**, then **Logs**. You should see the HTTP GET request generated when you viewed the application in your browser.

Home > myResourceGroup > mycontainer - Containers

**mycontainer** - Containers  
Container instances

Search (Ctrl+/) Refresh

1 container

NAME	IMAGE	STATE	PREVIOUS STATE	START TIME	RESTART COUNT
mycontainer	mcr.microsoft.co...	Running	-	2019-03-21T17:4...	0

Events Properties **Logs** Connect

```

listening on port 80
::ffff:10.240.255.55 - - [21/Mar/2019:17:43:53 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
::ffff:10.240.255.55 - - [21/Mar/2019:17:44:36 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
::ffff:10.240.255.55 - - [21/Mar/2019:17:44:36 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/72.0.3626.121 Safari/537.36"
  
```

## Clean up resources

When you're done with the container, select **Overview** for the *mycontainer* container instance, then select **Delete**.

Home > Resource groups > myResourceGroup > mycontainer

**mycontainer**  
Container instances

Search (Ctrl+/) Start Restart Stop **Delete** Refresh

**Overview**

Resource group (change)  
myResourceGroup

Status  
Running

Location  
West US

Subscription (change)  
Visual Studio Enterprise with MSDN

Subscription ID  
<Subscription ID>

Tags (change)  
Click here to add tags

OS type  
Linux

IP address  
40.121.69.231

FQDN  
mycontainer.westus.azurecontainer.io

Container count  
1

Select **Yes** when the confirmation dialog appears.

Delete container instances
  
Do you want to delete all container instances in container group 'mycontainer'?

## Next steps

In this quickstart, you created an Azure container instance from a public Microsoft image. If you'd like to build a container image and deploy it from a private Azure container registry, continue to the [Azure Container Instances tutorial](#).

[Azure Container Instances tutorial](#)

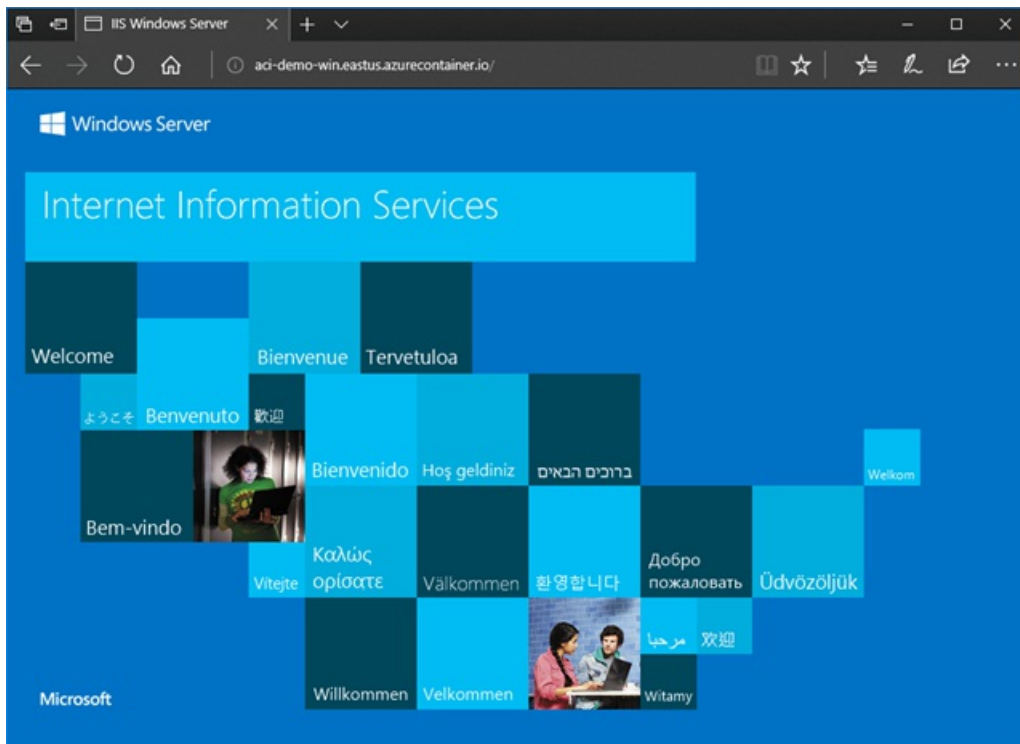


# Quickstart: Deploy a container instance in Azure using Azure PowerShell

3/22/2019 • 3 minutes to read • [Edit Online](#)

Use Azure Container Instances to run serverless Docker containers in Azure with simplicity and speed. Deploy an application to a container instance on-demand when you don't need a full container orchestration platform like Azure Kubernetes Service.

In this quickstart, you use Azure PowerShell to deploy an isolated Windows container and make its application available with a fully qualified domain name (FQDN). A few seconds after you execute a single deployment command, you can browse to the application running in the container:






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

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## Launch Azure Cloud Shell

The Azure Cloud Shell is a free interactive shell that you can use to run the steps in this article. It has common Azure tools preinstalled and configured to use with your account. Just click the **Copy** to copy the code, paste it into the Cloud Shell, and then press enter to run it. There are a few ways to launch the Cloud Shell:

Click <b>Try It</b> in the upper right corner of a code block.	
Open Cloud Shell in your browser.	
Click the <b>Cloud Shell</b> button on the menu in the upper right of the Azure portal.	

If you choose to install and use the PowerShell locally, this tutorial requires the Azure PowerShell module. Run `Get-Module -ListAvailable Az` to find the version. If you need to upgrade, see [Install Azure PowerShell module](#). If you are running PowerShell locally, you also need to run `Connect-AzAccount` to create a connection with Azure.

## Create a resource group

Azure container instances, like all Azure resources, must be deployed into a resource group. Resource groups allow you to organize and manage related Azure resources.

First, create a resource group named *myResourceGroup* in the *eastus* location with the following [New-AzResourceGroup](#) command:

```
New-AzResourceGroup -Name myResourceGroup -Location EastUS
```

## Create a container

Now that you have a resource group, you can run a container in Azure. To create a container instance with Azure PowerShell, provide a resource group name, container instance name, and Docker container image to the [New-AzContainerGroup](#) cmdlet. In this quickstart, you use the public `mcr.microsoft.com/windows/servercore/iis:nanoserver` image. This image packages Microsoft Internet Information Services (IIS) to run in Nano Server.

You can expose your containers to the internet by specifying one or more ports to open, a DNS name label, or both. In this quickstart, you deploy a container with a DNS name label so that IIS is publicly reachable.

Execute a command similar to the following to start a container instance. Set a `-DnsNameLabel` value that's unique within the Azure region where you create the instance. If you receive a "DNS name label not available" error message, try a different DNS name label.

```
New-AzContainerGroup -ResourceGroupName myResourceGroup -Name mycontainer -Image
mcr.microsoft.com/windows/servercore/iis:nanoserver -OsType Windows -DnsNameLabel aci-demo-win
```

Within a few seconds, you should receive a response from Azure. The container's `ProvisioningState` is initially **Creating**, but should move to **Succeeded** within a minute or two. Check the deployment state with the [Get-AzContainerGroup](#) cmdlet:

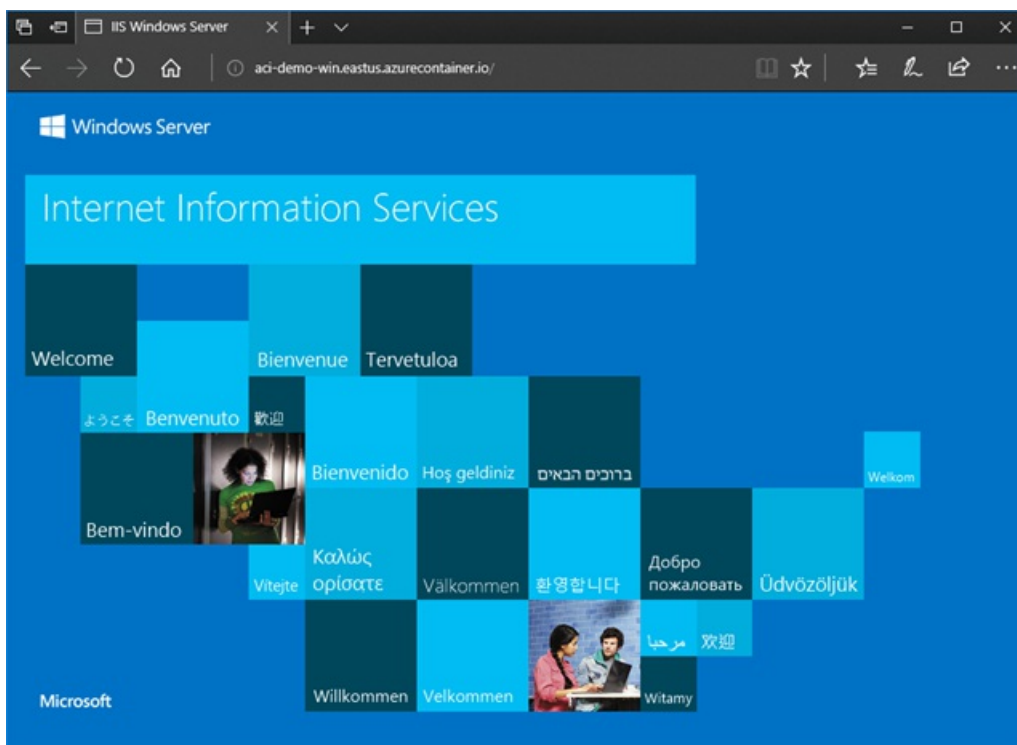
```
Get-AzContainerGroup -ResourceGroupName myResourceGroup -Name mycontainer
```

The container's provisioning state, fully qualified domain name (FQDN), and IP address appear in the cmdlet's output:

```
PS Azure:\> Get-AzContainerGroup -ResourceGroupName myResourceGroup -Name mycontainer
```

```
ResourceGroupName      : myResourceGroup
Id                     : /subscriptions/<Subscription
ID>/resourceGroups/myResourceGroup/providers/Microsoft.ContainerInstance/containerGroups/mycontainer
Name                   : mycontainer
Type                   : Microsoft.ContainerInstance/containerGroups
Location               : eastus
Tags                   :
ProvisioningState      : Creating
Containers              : {mycontainer}
ImageRegistryCredentials :
RestartPolicy          : Always
IpAddress              : 52.226.19.87
DnsNameLabel           : aci-demo-win
Fqdn                   : aci-demo-win.eastus.azurecontainer.io
Ports                  : {80}
OsType                 : Windows
Volumes                :
State                  : Pending
Events                 : {}
```

Once the container's `ProvisioningState` is **Succeeded**, navigate to its `Fqdn` in your browser. If you see a web page similar to the following, congratulations! You've successfully deployed an application running in a Docker container to Azure.



## Clean up resources

When you're done with the container, remove it with the `Remove-AzContainerGroup` cmdlet:

```
Remove-AzContainerGroup -ResourceGroupName myResourceGroup -Name mycontainer
```

## Next steps

In this quickstart, you created an Azure container instance from an image in the public Docker Hub registry. If

you'd like to build a container image and deploy it from a private Azure container registry, continue to the Azure Container Instances tutorial.

[Azure Container Instances tutorial](#)

# Tutorial: Create a container image for deployment to Azure Container Instances

3/14/2019 • 3 minutes to read • [Edit Online](#)

Azure Container Instances enables deployment of Docker containers onto Azure infrastructure without provisioning any virtual machines or adopting a higher-level service. In this tutorial, you package a small Node.js web application into a container image that can be run using Azure Container Instances.

In this article, part one of the series, you:

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

In tutorial parts two and three, you upload your image to Azure Container Registry, and then deploy it to Azure Container Instances.

## Before you begin

You must satisfy the following requirements to complete this tutorial:

**Azure CLI:** You must have Azure CLI version 2.0.29 or later installed on your local computer. Run `az --version` to find the version. If you need to install or upgrade, see [Install the Azure CLI](#).

**Docker:** This tutorial assumes a basic understanding of core Docker concepts like containers, container images, and basic `docker` commands. For a primer on Docker and container basics, see the [Docker overview](#).

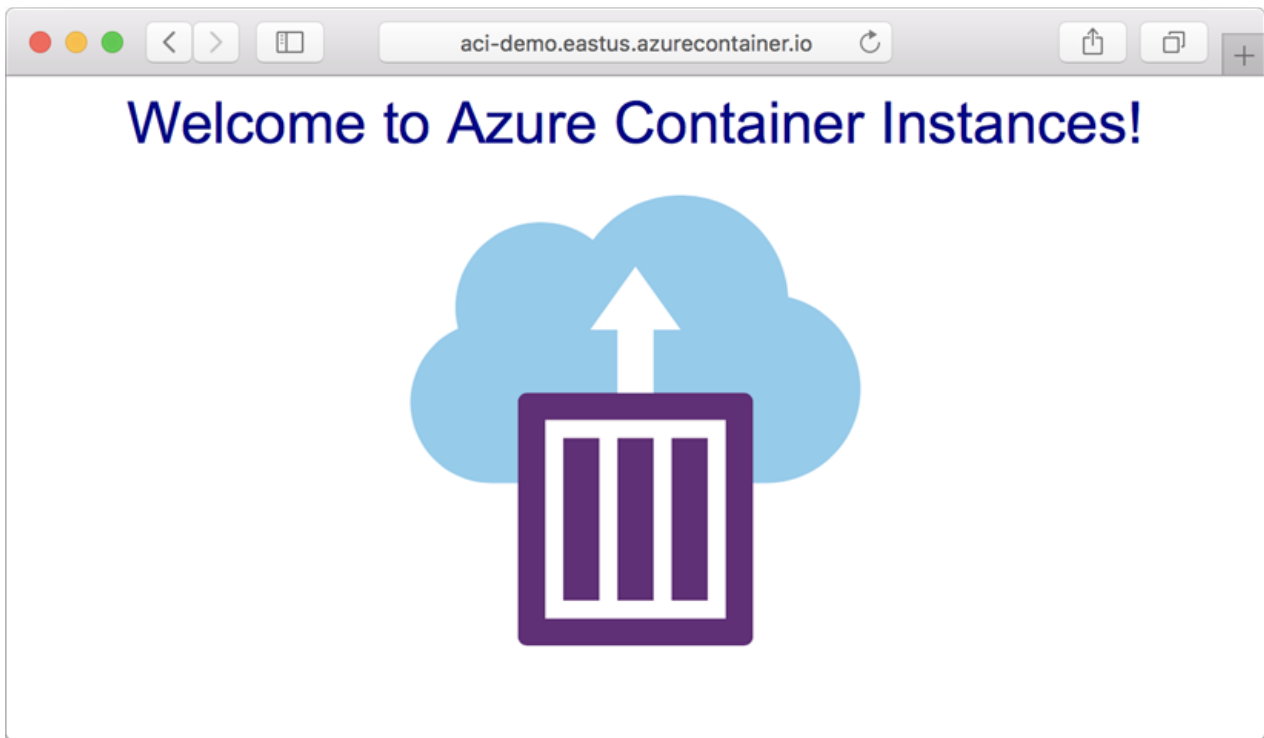
**Docker Engine:** To complete this tutorial, you need Docker Engine installed locally. Docker provides packages that configure the Docker environment on [macOS](#), [Windows](#), and [Linux](#).

### IMPORTANT

Because the Azure Cloud shell does not include the Docker daemon, you *must* install both the Azure CLI and Docker Engine on your *local computer* to complete this tutorial. You cannot use the Azure Cloud Shell for this tutorial.

## Get application code

The sample application in this tutorial is a simple web app built in [Node.js](#). The application serves a static HTML page, and looks similar to the following screenshot:



Use Git to clone the sample application's repository:

```
git clone https://github.com/Azure-Samples/aci-helloworld.git
```

You can also [download the ZIP archive](#) from GitHub directly.

## Build the container image

The Dockerfile in the sample application shows how the container is built. It starts from an [official Node.js image](#) based on [Alpine Linux](#), a small distribution that is well suited for use with containers. It then copies the application files into the container, installs dependencies using the Node Package Manager, and finally, starts the application.

```
FROM node:8.9.3-alpine
RUN mkdir -p /usr/src/app
COPY ./app/ /usr/src/app/
WORKDIR /usr/src/app
RUN npm install
CMD node /usr/src/app/index.js
```

Use the [docker build](#) command to create the container image and tag it as *aci-tutorial-app*:

```
docker build ./aci-helloworld -t aci-tutorial-app
```

Output from the [docker build](#) command is similar to the following (truncated for readability):

```
$ docker build ./aci-helloworld -t aci-tutorial-app
Sending build context to Docker daemon 119.3kB
Step 1/6 : FROM node:8.9.3-alpine
8.9.3-alpine: Pulling from library/node
88286f41530e: Pull complete
84f3a4bf8410: Pull complete
d0d9b2214720: Pull complete
Digest: sha256:c73277ccc763752b42bb2400d1aaecb4e3d32e3a9dbedd0e49885c71bea07354
Status: Downloaded newer image for node:8.9.3-alpine
--> 90f5ee24bee2
...
Step 6/6 : CMD node /usr/src/app/index.js
--> Running in f4a1ea099eec
--> 6edad76d09e9
Removing intermediate container f4a1ea099eec
Successfully built 6edad76d09e9
Successfully tagged aci-tutorial-app:latest
```

Use the `docker images` command to see the built image:

```
docker images
```

Your newly built image should appear in the list:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
aci-tutorial-app	latest	5c745774dfa9	39 seconds ago	68.1 MB

## Run the container locally

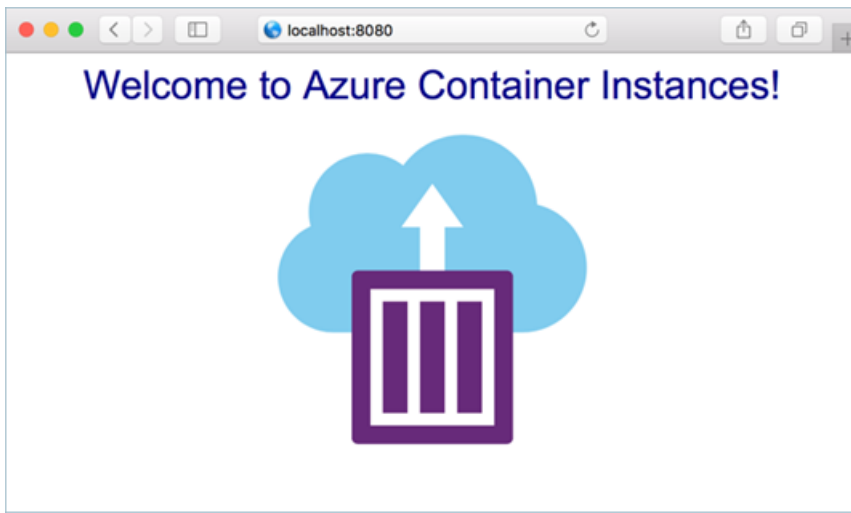
Before you deploy the container to Azure Container Instances, use `docker run` to run it locally and confirm that it works. The `-d` switch lets the container run in the background, while `-p` allows you to map an arbitrary port on your computer to port 80 in the container.

```
docker run -d -p 8080:80 aci-tutorial-app
```

Output from the `docker run` command displays the running container's ID if the command was successful:

```
$ docker run -d -p 8080:80 aci-tutorial-app
a2e3e4435db58ab0c664ce521854c2e1a1bda88c9cf2fcff46aedf48df86cccf
```

Now, navigate to `http://localhost:8080` in your browser to confirm that the container is running. You should see a web page similar to the following:



## Next steps

In this tutorial, you created a container image that can be deployed in Azure Container Instances, and verified that it runs locally. So far, you've done the following:

- Cloned the application source from GitHub
- Created a container image from the application source
- Tested the container locally

Advance to the next tutorial in the series to learn about storing your container image in Azure Container Registry:

[Push image to Azure Container Registry](#)



# Tutorial: Deploy an Azure container registry and push a container image

3/6/2019 • 5 minutes to read • [Edit Online](#)

This is part two of a three-part tutorial. [Part one](#) of the tutorial created a Docker container image for a Node.js web application. In this tutorial, you push the image to Azure Container Registry. If you haven't yet created the container image, return to [Tutorial 1 – Create container image](#).

Azure Container Registry is your private Docker registry in Azure. In this tutorial, you create an Azure Container Registry instance in your subscription, then push the previously created container image to it. In this article, part two of the series, you:

- Create an Azure Container Registry instance
- Tag a container image for your Azure container registry
- Upload the image to your registry

In the next article, the last in the series, you deploy the container from your private registry to Azure Container Instances.

## Before you begin

You must satisfy the following requirements to complete this tutorial:

**Azure CLI:** You must have Azure CLI version 2.0.29 or later installed on your local computer. Run `az --version` to find the version. If you need to install or upgrade, see [Install the Azure CLI](#).

**Docker:** This tutorial assumes a basic understanding of core Docker concepts like containers, container images, and basic `docker` commands. For a primer on Docker and container basics, see the [Docker overview](#).

**Docker Engine:** To complete this tutorial, you need Docker Engine installed locally. Docker provides packages that configure the Docker environment on [macOS](#), [Windows](#), and [Linux](#).

### IMPORTANT

Because the Azure Cloud shell does not include the Docker daemon, you *must* install both the Azure CLI and Docker Engine on your *local computer* to complete this tutorial. You cannot use the Azure Cloud Shell for this tutorial.

## Create Azure container registry

Before you create your container registry, you need a *resource group* to deploy it to. A resource group is a logical collection into which all Azure resources are deployed and managed.

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

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

Once you've created the resource group, create an Azure container registry with the `az acr create` command. The container registry name must be unique within Azure, and contain 5-50 alphanumeric characters. Replace `<acrName>` with a unique name for your registry:

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

Here's example output for a new Azure container registry named *mycontainerregistry082* (shown here truncated):

```
$ az acr create --resource-group myResourceGroup --name mycontainerregistry082 --sku Basic --admin-enabled true
...
{
  "adminUserEnabled": true,
  "creationDate": "2018-03-16T21:54:47.297875+00:00",
  "id": "/subscriptions/<SubscriptionID>/resourceGroups/myResourceGroup/providers/Microsoft.ContainerRegistry/registries/mycontainerregistry082",
  "location": "eastus",
  "loginServer": "mycontainerregistry082.azurecr.io",
  "name": "mycontainerregistry082",
  "provisioningState": "Succeeded",
  "resourceGroup": "myResourceGroup",
  "sku": {
    "name": "Basic",
    "tier": "Basic"
  },
  "status": null,
  "storageAccount": null,
  "tags": {},
  "type": "Microsoft.ContainerRegistry/registries"
}
```

The rest of the tutorial refers to `<acrName>` as a placeholder for the container registry name that you chose in this step.

## Log in to container registry

You must log in to your Azure Container Registry instance before pushing images to it. Use the [az acr login](#) command to complete the operation. You must provide the unique name you chose for the container registry when you created it.

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

The command returns `Login Succeeded` once completed:

```
$ az acr login --name mycontainerregistry082
Login Succeeded
```

## Tag container image

To push a container image to a private registry like Azure Container Registry, you must first tag the image with the full name of the registry's login server.

First, get the full login server name for your Azure container registry. Run the following [az acr show](#) command, and replace `<acrName>` with the name of registry you just created:

```
az acr show --name <acrName> --query loginServer --output table
```

For example, if your registry is named *mycontainerregistry082*:

```
$ az acr show --name mycontainerregistry082 --query loginServer --output table
Result
-----
mycontainerregistry082.azurecr.io
```

Now, display the list of your local images with the `docker images` command:

```
docker images
```

Along with any other images you have on your machine, you should see the *aci-tutorial-app* image you built in the [previous tutorial](#):

```
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
aci-tutorial-app    latest      5c745774dfa9     39 minutes ago   68.1 MB
```

Tag the *aci-tutorial-app* image with the loginServer of your container registry. Also, add the `:v1` tag to the end of the image name to indicate the image version number. Replace `<acrLoginServer>` with the result of the [az acr show](#) command you executed earlier.

```
docker tag aci-tutorial-app <acrLoginServer>/aci-tutorial-app:v1
```

Run `docker images` again to verify the tagging operation:

```
$ docker images
REPOSITORY          TAG          IMAGE ID          CREATED           SIZE
aci-tutorial-app    latest      5c745774dfa9     39 minutes ago   68.1 MB
mycontainerregistry082.azurecr.io/aci-tutorial-app    v1          5c745774dfa9     7 minutes ago    68.1 MB
```

## Push image to Azure Container Registry

Now that you've tagged the *aci-tutorial-app* image with the full login server name of your private registry, you can push it to the registry with the `docker push` command. Replace `<acrLoginServer>` with the full login server name you obtained in the earlier step.

```
docker push <acrLoginServer>/aci-tutorial-app:v1
```

The `push` operation should take a few seconds to a few minutes depending on your internet connection, and output is similar to the following:

```
$ docker push mycontainerregistry082.azurecr.io/aci-tutorial-app:v1
The push refers to a repository [mycontainerregistry082.azurecr.io/aci-tutorial-app]
3db9cac20d49: Pushed
13f653351004: Pushed
4cd158165f4d: Pushed
d8fbd47558a8: Pushed
44ab46125c35: Pushed
5bef08742407: Pushed
v1: digest: sha256:ed67fff971da47175856505585dcd92d1270c3b37543e8afd46014d328f05715 size: 1576
```

## List images in Azure Container Registry

To verify that the image you just pushed is indeed in your Azure container registry, list the images in your registry with the [az acr repository list](#) command. Replace `<acrName>` with the name of your container registry.

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

For example:

```
$ az acr repository list --name mycontainerregistry082 --output table
Result
-----
aci-tutorial-app
```

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

```
az acr repository show-tags --name <acrName> --repository aci-tutorial-app --output table
```

You should see output similar to the following:

```
$ az acr repository show-tags --name mycontainerregistry082 --repository aci-tutorial-app --output table
Result
-----
v1
```

## Next steps

In this tutorial, you prepared an Azure container registry for use with Azure Container Instances, and pushed a container image to the registry. The following steps were completed:

- Deployed an Azure Container Registry instance
- Tagged a container image for Azure Container Registry
- Uploaded an image to Azure Container Registry

Advance to the next tutorial to learn how to deploy the container to Azure using Azure Container Instances:

[Deploy container to Azure Container Instances](#)

# Tutorial: Deploy a container application to Azure Container Instances

2/15/2019 • 4 minutes to read • [Edit Online](#)

This is the final tutorial in a three-part series. Earlier in the series, [a container image was created](#) and [pushed to Azure Container Registry](#). This article completes the series by deploying the container to Azure Container Instances.

In this tutorial, you:

- Deploy the container from Azure Container Registry to Azure Container Instances
- View the running application in the browser
- Display the container's logs

## Before you begin

You must satisfy the following requirements to complete this tutorial:

**Azure CLI:** You must have Azure CLI version 2.0.29 or later installed on your local computer. Run `az --version` to find the version. If you need to install or upgrade, see [Install the Azure CLI](#).

**Docker:** This tutorial assumes a basic understanding of core Docker concepts like containers, container images, and basic `docker` commands. For a primer on Docker and container basics, see the [Docker overview](#).

**Docker Engine:** To complete this tutorial, you need Docker Engine installed locally. Docker provides packages that configure the Docker environment on [macOS](#), [Windows](#), and [Linux](#).

### IMPORTANT

Because the Azure Cloud shell does not include the Docker daemon, you *must* install both the Azure CLI and Docker Engine on your *local computer* to complete this tutorial. You cannot use the Azure Cloud Shell for this tutorial.

## Deploy the container using the Azure CLI

In this section, you use the Azure CLI to deploy the image built in the [first tutorial](#) and pushed to Azure Container Registry in the [second tutorial](#). Be sure you've completed those tutorials before proceeding.

### Get registry credentials

When you deploy an image that's hosted in a private container registry like the one created in the [second tutorial](#), you must supply credentials to access the registry. As shown in [Authenticate with Azure Container Registry from Azure Container Instances](#), a best practice for many scenarios is to create and configure an Azure Active Directory service principal with *pull* permissions to your registry. See that article for sample scripts to create a service principal with the necessary permissions. Take note of the service principal ID and service principal password. You use these credentials when you deploy the container.

You also need the full name of the container registry login server (replace `<acrName>` with the name of your registry):

```
az acr show --name <acrName> --query loginServer
```

## Deploy container

Now, use the [az container create](#) command to deploy the container. Replace `<acrLoginServer>` with the value you obtained from the previous command. Replace `<service-principal-ID>` and `<service-principal-password>` with the service principal ID and password that you created to access the registry. Replace `<aciDnsLabel>` with a desired DNS name.

```
az container create --resource-group myResourceGroup --name aci-tutorial-app --image <acrLoginServer>/aci-tutorial-app:v1 --cpu 1 --memory 1 --registry-login-server <acrLoginServer> --registry-username <service-principal-ID> --registry-password <service-principal-password> --dns-name-label <aciDnsLabel> --ports 80
```

Within a few seconds, you should receive an initial response from Azure. The `--dns-name-label` value must be unique within the Azure region you create the container instance. Modify the value in the preceding command if you receive a **DNS name label** error message when you execute the command.

## Verify deployment progress

To view the state of the deployment, use [az container show](#):

```
az container show --resource-group myResourceGroup --name aci-tutorial-app --query instanceView.state
```

Repeat the [az container show](#) command until the state changes from *Pending* to *Running*, which should take under a minute. When the container is *Running*, proceed to the next step.

## View the application and container logs

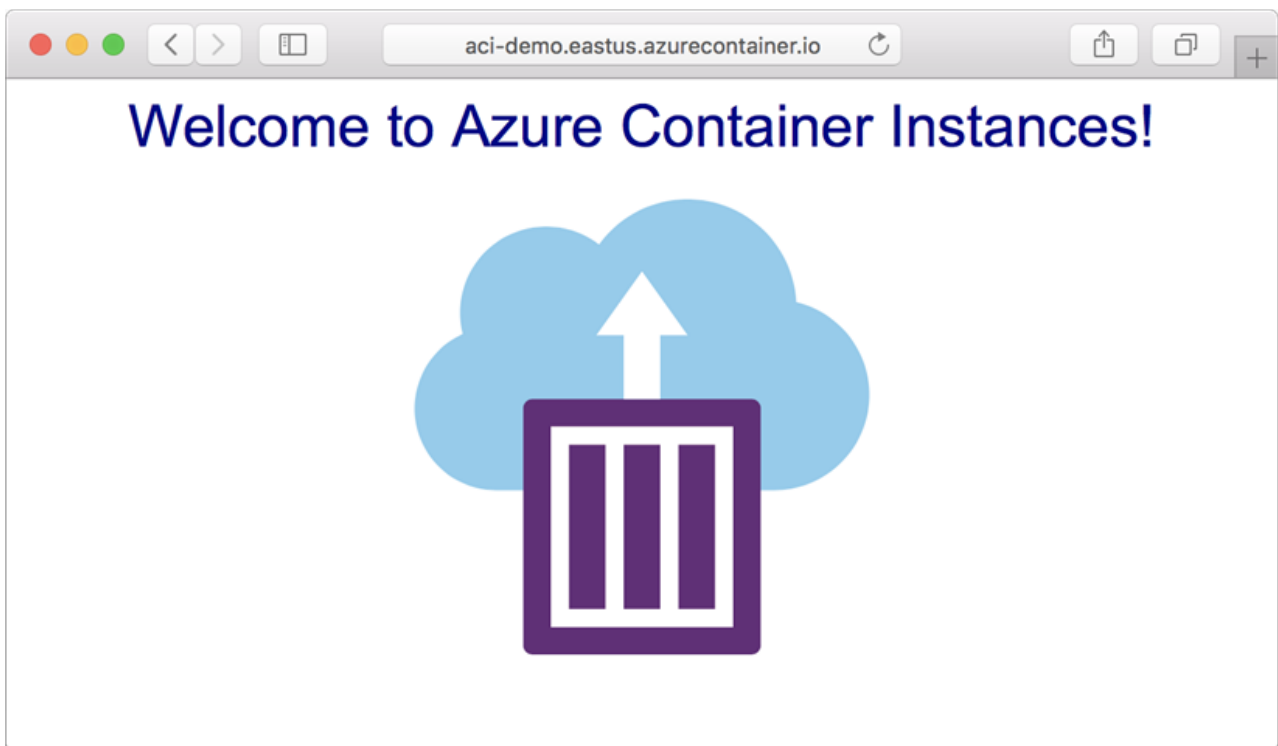
Once the deployment succeeds, display the container's fully qualified domain name (FQDN) with the [az container show](#) command:

```
az container show --resource-group myResourceGroup --name aci-tutorial-app --query ipAddress.fqdn
```

For example:

```
$ az container show --resource-group myResourceGroup --name aci-tutorial-app --query ipAddress.fqdn
"aci-demo.eastus.azurecontainer.io"
```

To see the running application, navigate to the displayed DNS name in your favorite browser:



You can also view the log output of the container:

```
az container logs --resource-group myResourceGroup --name aci-tutorial-app
```

Example output:

```
$ az container logs --resource-group myResourceGroup --name aci-tutorial-app
listening on port 80
::ffff:10.240.0.4 - - [21/Jul/2017:06:00:02 +0000] "GET / HTTP/1.1" 200 1663 "-" "Mozilla/5.0 (Macintosh;
Intel Mac OS X 10_12_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115 Safari/537.36"
::ffff:10.240.0.4 - - [21/Jul/2017:06:00:02 +0000] "GET /favicon.ico HTTP/1.1" 404 150 "http://aci-
demo.eastus.azurecontainer.io/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_5) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/59.0.3071.115 Safari/537.36"
```

## Clean up resources

If you no longer need any of the resources you created in this tutorial series, you can execute the [az group delete](#) command to remove the resource group and all resources it contains. This command deletes the container registry you created, as well as the running container, and all related resources.

```
az group delete --name myResourceGroup
```

## Next steps

In this tutorial, you completed the process of deploying your container to Azure Container Instances. The following steps were completed:

- Deployed the container from Azure Container Registry using the Azure CLI
- Viewed the application in the browser
- Viewed the container logs

Now that you have the basics down, move on to learning more about Azure Container Instances, such as how container groups work:





# Tutorial: Deploy a multi-container group using a YAML file

4/4/2019 • 4 minutes to read • [Edit Online](#)

Azure Container Instances supports the deployment of multiple containers onto a single host using a [container group](#). A container group is useful when building an application sidecar for logging, monitoring, or any other configuration where a service needs a second attached process.

In this tutorial, you follow steps to run a simple two-container sidecar configuration by deploying a YAML file using the Azure CLI. A YAML file provides a concise format for specifying the instance settings. You learn how to:

- Configure a YAML file
- Deploy the container group
- View the logs of the containers


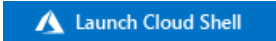

## NOTE

Multi-container groups are currently restricted to Linux containers.

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

## Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Select **Copy** to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select <b>Try It</b> in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu in the upper-right corner of the <a href="#">Azure portal</a> .	

## Configure a YAML file

To deploy a multi-container group with the `az container create` command in the Azure CLI, you must specify the container group configuration in a YAML file. Then pass the YAML file as a parameter to the command.

Start by copying the following YAML into a new file named **deploy-aci.yaml**. In Azure Cloud Shell, you can use Visual Studio Code to create the file in your working directory:

```
code deploy-aci.yaml
```

This YAML file defines a container group named "myContainerGroup" with two containers, a public IP address,

and two exposed ports. The containers are deployed from public Microsoft images. The first container in the group runs an internet-facing web application. The second container, the sidecar, periodically makes HTTP requests to the web application running in the first container via the container group's local network.

```
apiVersion: 2018-10-01
location: eastus
name: myContainerGroup
properties:
  containers:
    - name: aci-tutorial-app
      properties:
        image: mcr.microsoft.com/azuredocs/aci-helloworld:latest
        resources:
          requests:
            cpu: 1
            memoryInGb: 1.5
        ports:
          - port: 80
          - port: 8080
    - name: aci-tutorial-sidecar
      properties:
        image: mcr.microsoft.com/azuredocs/aci-tutorial-sidecar
        resources:
          requests:
            cpu: 1
            memoryInGb: 1.5
  osType: Linux
  ipAddress:
    type: Public
    ports:
      - protocol: tcp
        port: '80'
      - protocol: tcp
        port: '8080'
  tags: null
  type: Microsoft.ContainerInstance/containerGroups
```

To use a private container image registry, add the `imageRegistryCredentials` property to the container group, with values modified for your environment:

```
imageRegistryCredentials:
  - server: imageRegistryLoginServer
    username: imageRegistryUsername
    password: imageRegistryPassword
```

## Deploy the container group

Create a resource group with the [az group create](#) command:

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

Deploy the container group with the [az container create](#) command, passing the YAML file as an argument:

```
az container create --resource-group myResourceGroup --file deploy-aci.yaml
```

Within a few seconds, you should receive an initial response from Azure.

## View deployment state

To view the state of the deployment, use the following [az container show](#) command:

```
az container show --resource-group myResourceGroup --name myContainerGroup --output table
```

If you'd like to view the running application, navigate to its IP address in your browser. For example, the IP is `52.168.26.124` in this example output:

Name	ResourceGroup	Status	Image		
IP:ports	Network	CPU/Memory	OsType	Location	
myContainerGroup	danlep0318r	Running	mcr.microsoft.com/azuredocs/aci-tutorial-sidecar,mcr.microsoft.com/azuredocs/aci-helloworld:latest	20.42.26.114:80,8080	Public
Linux	eastus				1.0 core/1.5 gb

## View container logs

View the log output of a container using the [az container logs](#) command. The `--container-name` argument specifies the container from which to pull logs. In this example, the `aci-tutorial-app` container is specified.

```
az container logs --resource-group myResourceGroup --name myContainerGroup --container-name aci-tutorial-app
```

Output:

```
listening on port 80
::1 - - [21/Mar/2019:23:17:48 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
::1 - - [21/Mar/2019:23:17:51 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
::1 - - [21/Mar/2019:23:17:54 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
```

To see the logs for the sidecar container, run a similar command specifying the `aci-tutorial-sidecar` container.

```
az container logs --resource-group myResourceGroup --name myContainerGroup --container-name aci-tutorial-sidecar
```

Output:

```
Every 3s: curl -I http://localhost                2019-03-21 20:36:41

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
  0  1663    0     0    0     0      0     0  --:--:--  --:--:--  --:--:--    0
HTTP/1.1 200 OK
X-Powered-By: Express
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Wed, 29 Nov 2017 06:40:40 GMT
ETag: W/"67f-16006818640"
Content-Type: text/html; charset=UTF-8
Content-Length: 1663
Date: Thu, 21 Mar 2019 20:36:41 GMT
Connection: keep-alive
```

As you can see, the sidecar is periodically making an HTTP request to the main web application via the group's local network to ensure that it is running. This sidecar example could be expanded to trigger an alert if it received an HTTP response code other than `200 OK`.

## Next steps

In this tutorial, you used a YAML file to deploy a multi-container group in Azure Container Instances. You learned how to:

- Configure a YAML file for a multi-container group
- Deploy the container group
- View the logs of the containers

You can also specify a multi-container group using a [Resource Manager template](#). A Resource Manager template can be readily adapted for scenarios when you need to deploy additional Azure service resources with the container group.

# Tutorial: Deploy a multi-container group using a Resource Manager template

4/4/2019 • 4 minutes to read • [Edit Online](#)

Azure Container Instances supports the deployment of multiple containers onto a single host using a [container group](#). A container group is useful when building an application sidecar for logging, monitoring, or any other configuration where a service needs a second attached process.

In this tutorial, you follow steps to run a simple two-container sidecar configuration by deploying an Azure Resource Manager template using the Azure CLI. You learn how to:

- Configure a multi-container group template
- Deploy the container group
- View the logs of the containers

A Resource Manager template can be readily adapted for scenarios when you need to deploy additional Azure service resources (for example, an Azure Files share or a virtual network) with the container group.


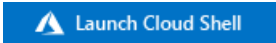

## NOTE

Multi-container groups are currently restricted to Linux containers.

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

## Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Select **Copy** to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select <b>Try It</b> in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu in the upper-right corner of the <a href="#">Azure portal</a> .	

## Configure a template

Start by copying the following JSON into a new file named `azuredeploy.json`. In Azure Cloud Shell, you can use Visual Studio Code to create the file in your working directory:

```
code azuredeploy.json
```

This Resource Manager template defines a container group with two containers, a public IP address, and two

exposed ports. The first container in the group runs an internet-facing web application. The second container, the sidecar, makes an HTTP request to the main web application via the group's local network.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "containerGroupName": {
      "type": "string",
      "defaultValue": "myContainerGroup",
      "metadata": {
        "description": "Container Group name."
      }
    }
  },
  "variables": {
    "container1name": "aci-tutorial-app",
    "container1image": "mcr.microsoft.com/azuredocs/aci-helloworld:latest",
    "container2name": "aci-tutorial-sidecar",
    "container2image": "mcr.microsoft.com/azuredocs/aci-tutorial-sidecar"
  },
  "resources": [
    {
      "name": "[parameters('containerGroupName')]",
      "type": "Microsoft.ContainerInstance/containerGroups",
      "apiVersion": "2018-10-01",
      "location": "[resourceGroup().location]",
      "properties": {
        "containers": [
          {
            "name": "[variables('container1name')]",
            "properties": {
              "image": "[variables('container1image')]",
              "resources": {
                "requests": {
                  "cpu": 1,
                  "memoryInGb": 1.5
                }
              },
              "ports": [
                {
                  "port": 80
                },
                {
                  "port": 8080
                }
              ]
            }
          },
          {
            "name": "[variables('container2name')]",
            "properties": {
              "image": "[variables('container2image')]",
              "resources": {
                "requests": {
                  "cpu": 1,
                  "memoryInGb": 1.5
                }
              }
            }
          }
        ]
      },
      "osType": "Linux",
      "ipAddress": {
        "type": "Public",
        "ports": [
          {
            "protocol": "tcp"
          }
        ]
      }
    }
  ]
}
```

```

        "protocol": "tcp",
        "port": "80"
      },
      {
        "protocol": "tcp",
        "port": "8080"
      }
    ]
  }
}
],
"outputs": {
  "containerIPv4Address": {
    "type": "string",
    "value": "[reference(resourceId('Microsoft.ContainerInstance/containerGroups/',
parameters('containerGroupName'))).ipAddress.ip]"
  }
}
}

```

To use a private container image registry, add an object to the JSON document with the following format. For an example implementation of this configuration, see the [ACI Resource Manager template reference](#) documentation.

```

"imageRegistryCredentials": [
  {
    "server": "[parameters('imageRegistryLoginServer')]",
    "username": "[parameters('imageRegistryUsername')]",
    "password": "[parameters('imageRegistryPassword')]"
  }
]

```

## Deploy the template

Create a resource group with the [az group create](#) command.

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

Deploy the template with the [az group deployment create](#) command.

```
az group deployment create --resource-group myResourceGroup --template-file azuredeploy.json
```

Within a few seconds, you should receive an initial response from Azure.

## View deployment state

To view the state of the deployment, use the following [az container show](#) command:

```
az container show --resource-group myResourceGroup --name myContainerGroup --output table
```

If you'd like to view the running application, navigate to its IP address in your browser. For example, the IP is `52.168.26.124` in this example output:

Name	ResourceGroup	Status	Image	Location
IP:ports	Network	CPU/Memory	OsType	
-----	-----	-----	-----	-----
myContainerGroup	danlep0318r	Running	mcr.microsoft.com/azuredocs/aci-tutorial-sidecar,mcr.microsoft.com/azuredocs/aci-helloworld:latest	20.42.26.114:80,8080 Public
Linux	eastus			1.0 core/1.5 gb

## View container logs

View the log output of a container using the `az container logs` command. The `--container-name` argument specifies the container from which to pull logs. In this example, the `aci-tutorial-app` container is specified.

```
az container logs --resource-group myResourceGroup --name myContainerGroup --container-name aci-tutorial-app
```

Output:

```
listening on port 80
::1 - - [21/Mar/2019:23:17:48 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
::1 - - [21/Mar/2019:23:17:51 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
::1 - - [21/Mar/2019:23:17:54 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
```

To see the logs for the sidecar container, run a similar command specifying the `aci-tutorial-sidecar` container.

```
az container logs --resource-group myResourceGroup --name myContainerGroup --container-name aci-tutorial-sidecar
```

Output:

```
Every 3s: curl -I http://localhost                2019-03-21 20:36:41

  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
  0  1663    0    0    0     0      0      0  --:--:-- --:--:-- --:--:--    0
HTTP/1.1 200 OK
X-Powered-By: Express
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Last-Modified: Wed, 29 Nov 2017 06:40:40 GMT
ETag: W/"67f-16006818640"
Content-Type: text/html; charset=UTF-8
Content-Length: 1663
Date: Thu, 21 Mar 2019 20:36:41 GMT
Connection: keep-alive
```

As you can see, the sidecar is periodically making an HTTP request to the main web application via the group's local network to ensure that it is running. This sidecar example could be expanded to trigger an alert if it received an HTTP response code other than `200 OK`.

## Next steps

In this tutorial, you used an Azure Resource Manager template to deploy a multi-container group in Azure Container Instances. You learned how to:

- Configure a multi-container group template



- Deploy the container group
- View the logs of the containers

For additional template samples, see [Azure Resource Manager templates for Azure Container Instances](#).

You can also specify a multi-container group using a [YAML file](#). Due to the YAML format's more concise nature, deployment with a YAML file is a good choice when your deployment includes only container instances.

# Azure Resource Manager templates for Azure Container Instances

4/4/2019 • 2 minutes to read • [Edit Online](#)

The following sample templates deploy container instances in various configurations.

For deployment options, see the [Deployment](#) section. If you'd like to create your own templates, the Azure Container Instances [Resource Manager template reference](#) details template format and available properties.

## Sample templates

<b>Applications</b>	
<a href="#">WordPress</a>	Creates a WordPress website and its MySQL database in a container group. The WordPress site content and MySQL database are persisted to an Azure Files share. Also creates an application gateway to expose public network access to WordPress.
<a href="#">MS NAV with SQL Server and IIS</a>	Deploys a single Windows container with a fully featured self-contained Dynamics NAV / Dynamics 365 Business Central environment.
<b>Volumes</b>	
<a href="#">emptyDir</a>	Deploys two Linux containers that share an emptyDir volume.
<a href="#">gitRepo</a>	Deploys a Linux container that clones a GitHub repo and mounts it as a volume.
<a href="#">secret</a>	Deploys a Linux container with a PFX cert mounted as a secret volume.
<b>Networking</b>	
<a href="#">UDP-exposed container</a>	Deploys a Windows or Linux container that exposes a UDP port.
<a href="#">Linux container with public IP</a>	Deploys a single Linux container accessible via a public IP.
<a href="#">Deploy a container group with a virtual network (preview)</a>	Deploys a new virtual network, subnet, network profile, and container group.
<b>Azure resources</b>	
<a href="#">Create Azure Storage account and Files share</a>	Uses the Azure CLI in a container instance to create a storage account and an Azure Files share.

# Deployment

You have several options for deploying resources with Resource Manager templates:

[Azure CLI](#)

[Azure PowerShell](#)

[Azure portal](#)

[REST API](#)

# Container groups in Azure Container Instances

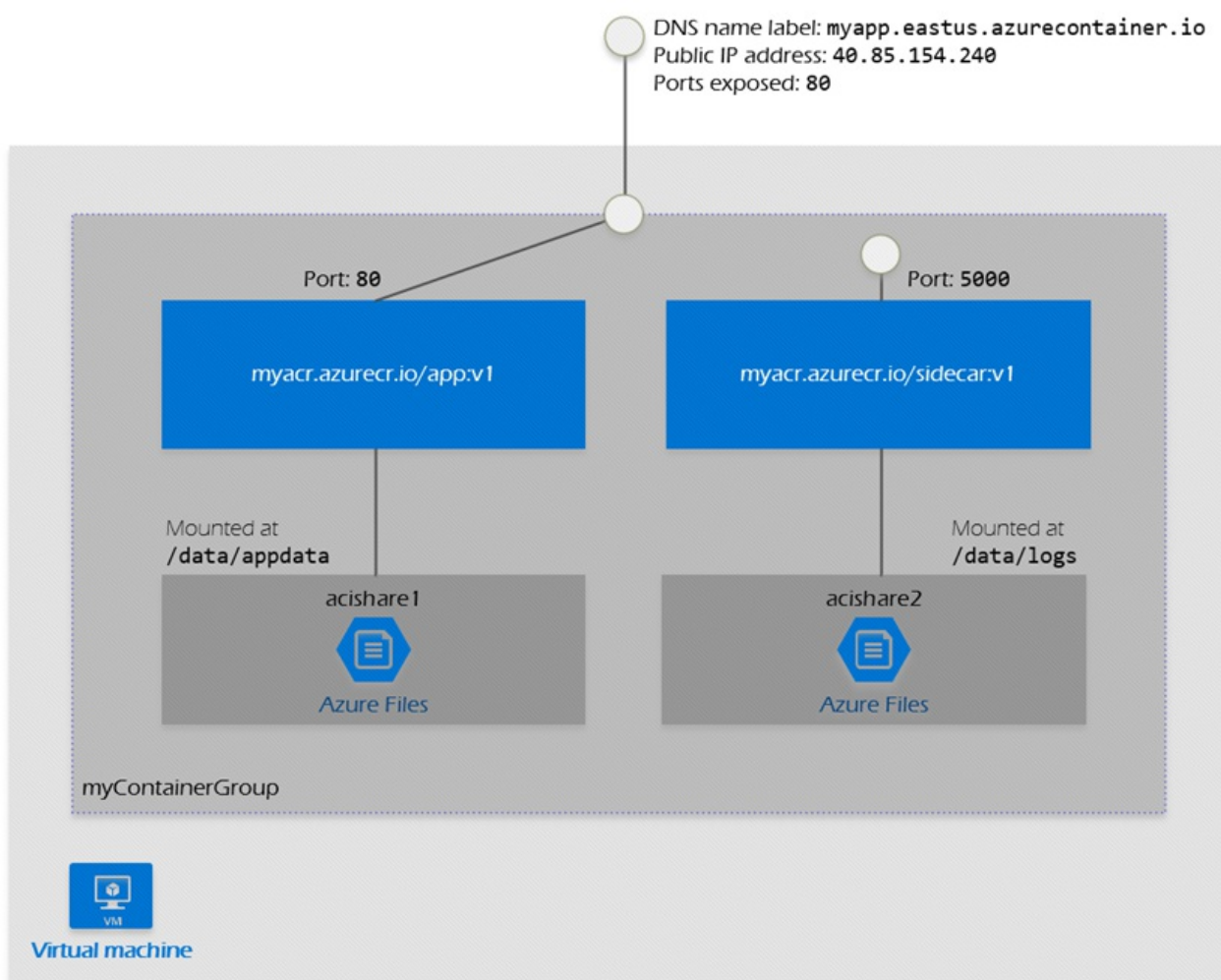
4/4/2019 • 4 minutes to read • [Edit Online](#)

The top-level resource in Azure Container Instances is the *container group*. This article describes what container groups are and the types of scenarios they enable.

## How a container group works

A container group is a collection of containers that get scheduled on the same host machine. The containers in a container group share a lifecycle, resources, local network, and storage volumes. It's similar in concept to a *pod* in [Kubernetes](#).

The following diagram shows an example of a container group that includes multiple containers:



This example container group:

- Is scheduled on a single host machine.
- Is assigned a DNS name label.
- Exposes a single public IP address, with one exposed port.
- Consists of two containers. One container listens on port 80, while the other listens on port 5000.
- Includes two Azure file shares as volume mounts, and each container mounts one of the shares locally.

#### NOTE

Multi-container groups currently support only Linux containers. For Windows containers, Azure Container Instances only supports deployment of a single instance. While we are working to bring all features to Windows containers, you can find current platform differences in the service [Overview](#).

## Deployment

Here are two common ways to deploy a multi-container group: use a [Resource Manager template](#) or a [YAML file](#). A Resource Manager template is recommended when you need to deploy additional Azure service resources (for example, an [Azure Files share](#)) when you deploy the container instances. Due to the YAML format's more concise nature, a YAML file is recommended when your deployment includes only container instances.

To preserve a container group's configuration, you can export the configuration to a YAML file by using the Azure CLI command [az container export](#). Export allows you to store your container group configurations in version control for "configuration as code." Or, use the exported file as a starting point when developing a new configuration in YAML.

## Resource allocation

Azure Container Instances allocates resources such as CPUs, memory, and optionally [GPUs](#) (preview) to a container group by adding the [resource requests](#) of the instances in the group. Taking CPU resources as an example, if you create a container group with two instances, each requesting 1 CPU, then the container group is allocated 2 CPUs.

The maximum resources available for a container group depend on the [Azure region](#) used for the deployment.

### Container resource requests and limits

- By default, container instances in a group share the requested resources of the group. In a group with two instances each requesting 1 CPU, the group as whole has access to 2 CPUs. Each instance can use up to the 2 CPUs and the instances may compete for CPU resource while they are running.
- To limit resource usage by an instance in a group, optionally set a [resource limit](#) for the instance. In a group with two instances requesting 1 CPU, one of your containers might require more CPUs to run than the other.

In this scenario, you could set a resource limit of 0.5 CPU for one instance, and a limit of 2 CPUs for the second. This configuration limits the first container's resource usage to 0.5 CPU, allowing the second container to use up to the full 2 CPUs if available.

For more information, see the [ResourceRequirements](#) property in the container groups REST API.

### Minimum and maximum allocation

- Allocate a **minimum** of 1 CPU and 1 GB of memory to a container group. Individual container instances within a group can be provisioned with less than 1 CPU and 1 GB of memory.
- For the **maximum** resources in a container group, see the [resource availability][aci-region-availability] for Azure Container Instances in the deployment region.

## Networking

Container groups share an IP address and a port namespace on that IP address. To enable external clients to reach a container within the group, you must expose the port on the IP address and from the container. Because containers within the group share a port namespace, port mapping isn't supported. Containers within a group can reach each other via localhost on the ports that they have exposed, even if those ports aren't exposed

externally on the group's IP address.

Optionally deploy container groups into an [Azure virtual network](#) (preview) to allow containers to communicate securely with other resources in the virtual network.

## Storage

You can specify external volumes to mount within a container group. You can map those volumes into specific paths within the individual containers in a group.

## Common scenarios

Multi-container groups are useful in cases where you want to divide a single functional task into a small number of container images. These images can then be delivered by different teams and have separate resource requirements.

Example usage could include:

- A container serving a web application and a container pulling the latest content from source control.
- An application container and a logging container. The logging container collects the logs and metrics output by the main application and writes them to long-term storage.
- An application container and a monitoring container. The monitoring container periodically makes a request to the application to ensure that it's running and responding correctly, and raises an alert if it's not.
- A front-end container and a back-end container. The front end might serve a web application, with the back end running a service to retrieve data.

## Next steps

Learn how to deploy a multi-container container group with an Azure Resource Manager template:

[Deploy a container group](#)

# Quotas and limits for Azure Container Instances

3/6/2019 • 2 minutes to read • [Edit Online](#)

All Azure services include certain default limits and quotas for resources and features. This article details the default quotas and limits for Azure Container Instances.

For the availability of Azure Container Instances features and resources in Azure regions, see [Resource availability for Azure Container Instances](#).

## Service quotas and limits

RESOURCE	DEFAULT LIMIT
Container groups per <a href="#">subscription</a>	100 <sup>1</sup>
Number of containers per container group	60
Number of volumes per container group	20
Ports per IP	5
Container instance log size - running instance	4 MB
Container instance log size - stopped instance	16 KB or 1,000 lines
Container creates per hour	300 <sup>1</sup>
Container creates per 5 minutes	100 <sup>1</sup>
Container deletes per hour	300 <sup>1</sup>
Container deletes per 5 minutes	100 <sup>1</sup>

<sup>1</sup>To request a limit increase, create an [Azure Support request](#).

## Next steps

Certain default limits and quotas can be increased. To request an increase of one or more resources that support such an increase, please submit an [Azure support request](#) (select "Quota" for **Issue type**).

# Resource availability for Azure Container Instances in Azure regions

3/6/2019 • 2 minutes to read • [Edit Online](#)

This article details the availability of Azure Container Instances compute and memory resources in Azure regions.

Values presented are the maximum resources available per deployment of a [container group](#). Values are current at time of publication. For up-to-date information, use the [List Capabilities](#) API.

## NOTE

Container groups created within these resource limits are subject to availability within the deployment region. When a region is under heavy load, you may experience a failure when deploying instances. To mitigate such a deployment failure, try deploying instances with lower resource settings, or try your deployment at a later time.

For information about quotas and other limits in your deployments, see [Quotas and limits for Azure Container Instances](#).

## Availability - General

LOCATION	OS	CPU	MEMORY (GB)
Canada Central, Central US, East US 2, South Central US	Linux	4	16
East US, North Europe, West Europe, West US, West US 2	Linux	4	14
Japan East	Linux	2	8
Australia East, Southeast Asia	Linux	2	7
Central India, East Asia, North Central US, South India	Linux	2	3.5
East US, West Europe, West US	Windows	4	14
Australia East, Canada Central, Central India, Central US, East Asia, East US 2, Japan East, North Central US, North Europe, South Central US, South India, Southeast Asia, West US 2	Windows	2	3.5

## Availability - Virtual network deployment (preview)



The following regions and resources are available to a container group deployed in an [Azure virtual network](#) (preview),

### Regions and resource availability

LOCATION	OS	CPU	MEMORY (GB)
West Europe	Linux	4	14
East US, West US	Linux	2	3.5
Australia East, North Europe	Linux	1	1.5

## Availability - GPU resources (preview)

The following regions and resources are available to a container group deployed with [GPU resources](#) (preview),

### Supported regions

- East US
- West US 2
- South Central US
- West Europe
- North Europe
- East Asia
- Central India

### Resource availability

OS	GPU SKU	GPU COUNT	CPU	MEMORY (GB)
Linux	K80	1	6	56
Linux	K80	2	12	112
Linux	K80	4	24	224
Linux	P100	1	6	112
Linux	P100	2	12	224
Linux	P100	4	24	448
Linux	V100	1	6	112
Linux	V100	2	12	224
Linux	V100	4	24	448

## Next steps

Let the team know if you'd like to see additional regions or increased resource availability at [aka.ms/aci/feedback](https://aka.ms/aci/feedback).

For information on troubleshooting container instance deployment, see [Troubleshoot deployment issues with](#)



# Azure Container Instances and container orchestrators

3/12/2019 • 3 minutes to read • [Edit Online](#)

Because of their small size and application orientation, containers are well suited for agile delivery environments and microservice-based architectures. The task of automating and managing a large number of containers and how they interact is known as *orchestration*. Popular container orchestrators include Kubernetes, DC/OS, and Docker Swarm.

Azure Container Instances provides some of the basic scheduling capabilities of orchestration platforms. And while it does not cover the higher-value services that those platforms provide, Azure Container Instances can be complementary to them. This article describes the scope of what Azure Container Instances handles, and how full container orchestrators might interact with it.

## Traditional orchestration

The standard definition of orchestration includes the following tasks:

- **Scheduling:** Given a container image and a resource request, find a suitable machine on which to run the container.
- **Affinity/Anti-affinity:** Specify that a set of containers should run nearby each other (for performance) or sufficiently far apart (for availability).
- **Health monitoring:** Watch for container failures and automatically reschedule them.
- **Failover:** Keep track of what is running on each machine, and reschedule containers from failed machines to healthy nodes.
- **Scaling:** Add or remove container instances to match demand, either manually or automatically.
- **Networking:** Provide an overlay network for coordinating containers to communicate across multiple host machines.
- **Service discovery:** Enable containers to locate each other automatically, even as they move between host machines and change IP addresses.
- **Coordinated application upgrades:** Manage container upgrades to avoid application downtime, and enable rollback if something goes wrong.

## Orchestration with Azure Container Instances: A layered approach

Azure Container Instances enables a layered approach to orchestration, providing all of the scheduling and management capabilities required to run a single container, while allowing orchestrator platforms to manage multi-container tasks on top of it.

Because the underlying infrastructure for container instances is managed by Azure, an orchestrator platform does not need to concern itself with finding an appropriate host machine on which to run a single container. The elasticity of the cloud ensures that one is always available. Instead, the orchestrator can focus on the tasks that simplify the development of multi-container architectures, including scaling and coordinated upgrades.

## Scenarios

While orchestrator integration with Azure Container Instances is still nascent, we anticipate that a few different environments will emerge:

### Orchestration of container instances exclusively

Because they start quickly and bill by the second, an environment based exclusively on Azure Container Instances offers the fastest way to get started and to deal with highly variable workloads.

### Combination of container instances and containers in Virtual Machines

For long-running, stable workloads, orchestrating containers in a cluster of dedicated virtual machines is typically cheaper than running the same containers with Azure Container Instances. However, container instances offer a great solution for quickly expanding and contracting your overall capacity to deal with unexpected or short-lived spikes in usage.

Rather than scaling out the number of virtual machines in your cluster, then deploying additional containers onto those machines, the orchestrator can simply schedule the additional containers in Azure Container Instances, and delete them when they're no longer needed.

## Sample implementation: virtual nodes for Azure Kubernetes Service (AKS)

To rapidly scale application workloads in an [Azure Kubernetes Service](#) (AKS) cluster, you can use *virtual nodes* created dynamically in Azure Container Instances. Currently in preview, virtual nodes enable network communication between pods that run in ACI and the AKS cluster.

Virtual nodes currently support Linux container instances. Get started with virtual nodes using the [Azure CLI](#) or [Azure portal](#).

Virtual nodes use the open source [Virtual Kubelet](#) to mimic the Kubernetes [kubelet](#) by registering as a node with unlimited capacity. The Virtual Kubelet dispatches the creation of [pods](#) as container groups in Azure Container Instances.

See the [Virtual Kubelet](#) project for additional examples of extending the Kubernetes API into serverless container platforms.

## Next steps

Create your first container with Azure Container Instances using the [quickstart guide](#).

# Deploy container instances into an Azure virtual network

4/26/2019 • 11 minutes to read • [Edit Online](#)

[Azure Virtual Network](#) provides secure, private networking for your Azure and on-premises resources. By deploying container groups into an Azure virtual network, your containers can communicate securely with other resources in the virtual network.

Container groups deployed into an Azure virtual network enable scenarios like:

- Direct communication between container groups in the same subnet
- Send [task-based](#) workload output from container instances to a database in the virtual network
- Retrieve content for container instances from a [service endpoint](#) in the virtual network
- Container communication with virtual machines in the virtual network
- Container communication with on-premises resources through a [VPN gateway](#) or [ExpressRoute](#)

## IMPORTANT

This feature is currently in preview, and some [limitations](#) apply. Previews are made available to you on the condition that you agree to the [supplemental terms of use](#). Some aspects of this feature may change prior to general availability (GA).

## Virtual network deployment limitations

Certain limitations apply when you deploy container groups to a virtual network.

- To deploy container groups to a subnet, the subnet cannot contain any other resource types. Remove all existing resources from an existing subnet prior to deploying container groups to it, or create a new subnet.
- You cannot use a [managed identity](#) in a container group deployed to a virtual network.
- Due to the additional networking resources involved, deploying a container group to a virtual network is typically somewhat slower than deploying a standard container instance.

## Preview limitations

While this feature is in preview, the following limitations apply when deploying container groups to a virtual network.

### Regions and resource availability

LOCATION	OS	CPU	MEMORY (GB)
West Europe	Linux	4	14
East US, West US	Linux	2	3.5
Australia East, North Europe	Linux	1	1.5

Container resource limits may differ from limits for non-networked container instances in these regions. Currently only Linux containers are supported for this feature. Windows support is planned.

### Unsupported networking scenarios

- **Azure Load Balancer** - Placing an Azure Load Balancer in front of container instances in a networked container group is not supported
- **Virtual network peering** - You can't peer a virtual network containing a subnet delegated to Azure Container Instances to another virtual network
- **Route tables** - User-defined routes can't be set up in a subnet delegated to Azure Container Instances
- **Network security groups** - Outbound security rules in NSGs applied to a subnet delegated to Azure Container Instances aren't currently enforced
- **Public IP or DNS label** - Container groups deployed to a virtual network don't currently support exposing containers directly to the internet with a public IP address or a fully qualified domain name
- **Internal name resolution** - Name resolution for Azure resources in the virtual network via the internal Azure DNS is not supported

**Network resource deletion** requires [additional steps](#) once you've deployed container groups to the virtual network.

## Required network resources

There are three Azure Virtual Network resources required for deploying container groups to a virtual network: the [virtual network](#) itself, a [delegated subnet](#) within the virtual network, and a [network profile](#).

### Virtual network

A virtual network defines the address space in which you create one or more subnets. You then deploy Azure resources (like container groups) into the subnets in your virtual network.

### Subnet (delegated)

Subnets segment the virtual network into separate address spaces usable by the Azure resources you place in them. You create one or several subnets within a virtual network.

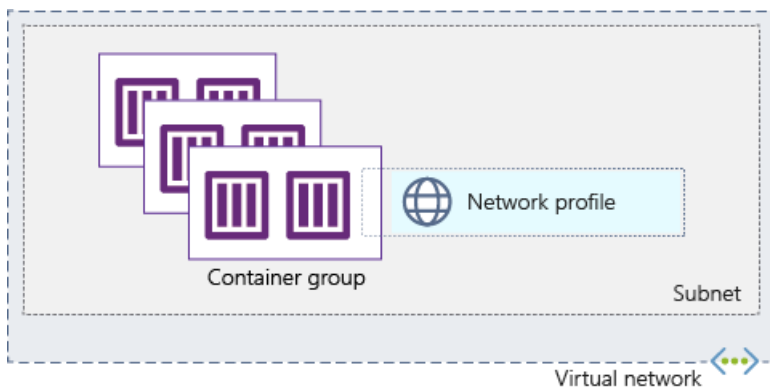
The subnet that you use for container groups may contain only container groups. When you first deploy a container group to a subnet, Azure delegates that subnet to Azure Container Instances. Once delegated, the subnet can be used only for container groups. If you attempt to deploy resources other than container groups to a delegated subnet, the operation fails.

### Network profile

A network profile is a network configuration template for Azure resources. It specifies certain network properties for the resource, for example, the subnet into which it should be deployed. When you first use the [az container create](#) command to deploy a container group to a subnet (and thus a virtual network), Azure creates a network profile for you. You can then use that network profile for future deployments to the subnet.

To use a Resource Manager template, YAML file, or a programmatic method to deploy a container group to a subnet, you need to provide the full Resource Manager resource ID of a network profile. You can use a profile previously created using [az container create](#), or create a profile using a Resource Manager template (see [template example](#) and [reference](#)). To get the ID of a previously created profile, use the [az network profile list](#) command.

In the following diagram, several container groups have been deployed to a subnet delegated to Azure Container Instances. Once you've deployed one container group to a subnet, you can deploy additional container groups to it by specifying the same network profile.



## Deployment scenarios

You can use [az container create](#) to deploy container groups to a new virtual network and allow Azure to create the required network resources for you, or deploy to an existing virtual network.

### New virtual network

To deploy to a new virtual network and have Azure create the network resources for you automatically, specify the following when you execute [az container create](#):

- Virtual network name
- Virtual network address prefix in CIDR format
- Subnet name
- Subnet address prefix in CIDR format

The virtual network and subnet address prefixes specify the address spaces for the virtual network and subnet, respectively. These values are represented in Classless Inter-Domain Routing (CIDR) notation, for example `10.0.0.0/16`. For more information about working with subnets, see [Add, change, or delete a virtual network subnet](#).

Once you've deployed your first container group with this method, you can deploy to the same subnet by specifying the virtual network and subnet names, or the network profile that Azure automatically creates for you. Because Azure delegates the subnet to Azure Container Instances, you can deploy *only* container groups to the subnet.

### Existing virtual network

To deploy a container group to an existing virtual network:

1. Create a subnet within your existing virtual network, or empty an existing subnet of *all* other resources
2. Deploy a container group with [az container create](#) and specify one of the following:
  - Virtual network name and subnet name
  - Virtual network resource ID and subnet resource ID, which allows using a virtual network from a different resource group
  - Network profile name or ID, which you can obtain using [az network profile list](#)

Once you deploy your first container group to an existing subnet, Azure delegates that subnet to Azure Container Instances. You can no longer deploy resources other than container groups to that subnet.

## Deployment examples

The following sections describe how to deploy container groups to a virtual network with the Azure CLI. The command examples are formatted for the **Bash** shell. If you prefer another shell such as PowerShell or Command Prompt, adjust the line continuation characters accordingly.

### Deploy to a new virtual network

First, deploy a container group and specify the parameters for a new virtual network and subnet. When you specify these parameters, Azure creates the virtual network and subnet, delegates the subnet to Azure Container instances, and also creates a network profile. Once these resources are created, your container group is deployed to the subnet.

Run the following `az container create` command that specifies settings for a new virtual network and subnet. You need to supply the name of a resource group that was created in a region that [supports](#) container groups in a virtual network. This command deploys the public Microsoft [aci-helloworld](#) container that runs a small Node.js webserver serving a static web page. In the next section, you'll deploy a second container group to the same subnet, and test communication between the two container instances.

```
az container create \
  --name appcontainer \
  --resource-group myResourceGroup \
  --image mcr.microsoft.com/azuredocs/aci-helloworld \
  --vnet aci-vnet \
  --vnet-address-prefix 10.0.0.0/16 \
  --subnet aci-subnet \
  --subnet-address-prefix 10.0.0.0/24
```

When you deploy to a new virtual network by using this method, the deployment can take a few minutes while the network resources are created. After the initial deployment, additional container group deployments complete more quickly.

### Deploy to existing virtual network

Now that you've deployed a container group to a new virtual network, deploy a second container group to the same subnet, and verify communication between the two container instances.

First, get the IP address of the first container group you deployed, the *appcontainer*:

```
az container show --resource-group myResourceGroup --name appcontainer --query ipAddress.ip --output tsv
```

The output should display the IP address of the container group in the private subnet:

```
$ az container show --resource-group myResourceGroup --name appcontainer --query ipAddress.ip --output tsv
10.0.0.4
```

Now, set `CONTAINER_GROUP_IP` to the IP you retrieved with the `az container show` command, and execute the following `az container create` command. This second container, *commchecker*, runs an Alpine Linux-based image and executes `wget` against the first container group's private subnet IP address.

```
CONTAINER_GROUP_IP=<container-group-IP-here>

az container create \
  --resource-group myResourceGroup \
  --name commchecker \
  --image alpine:3.5 \
  --command-line "wget $CONTAINER_GROUP_IP" \
  --restart-policy never \
  --vnet aci-vnet \
  --subnet aci-subnet
```

After this second container deployment has completed, pull its logs so you can see the output of the `wget` command it executed:



```
az container logs --resource-group myResourceGroup --name commchecker
```

If the second container communicated successfully with the first, output should be similar to:

```
$ az container logs --resource-group myResourceGroup --name commchecker
Connecting to 10.0.0.4 (10.0.0.4:80)
index.html      100% |*****| 1663    0:00:00 ETA
```

The log output should show that `wget` was able to connect and download the index file from the first container using its private IP address on the local subnet. Network traffic between the two container groups remained within the virtual network.

### Deploy to existing virtual network - YAML

You can also deploy a container group to an existing virtual network by using a YAML file. To deploy to a subnet in a virtual network, you specify several additional properties in the YAML:

- `ipAddress`: The IP address settings for the container group.
  - `ports`: The ports to open, if any.
  - `protocol`: The protocol (TCP or UDP) for the opened port.
- `networkProfile`: Specifies network settings like the virtual network and subnet for an Azure resource.
  - `id`: The full Resource Manager resource ID of the `networkProfile`.

To deploy a container group to a virtual network with a YAML file, you first need to get the ID of the network profile. Execute the [az network profile list](#) command, specifying the name of the resource group that contains your virtual network and delegated subnet.

```
az network profile list --resource-group myResourceGroup --query [0].id --output tsv
```

The output of the command displays the full resource ID for the network profile:

```
$ az network profile list --resource-group myResourceGroup --query [0].id --output tsv
/subscriptions/<Subscription
ID>/resourceGroups/myResourceGroup/providers/Microsoft.Network/networkProfiles/aci-network-profile-aci-vnet-
aci-subnet
```

Once you have the network profile ID, copy the following YAML into a new file named `vnet-deploy-aci.yaml`. Under `networkProfile`, replace the `id` value with ID you just retrieved, then save the file. This YAML creates a container group named `appcontaineryaml` in your virtual network.

```

apiVersion: '2018-09-01'
location: westus
name: appcontaineryaml
properties:
  containers:
    - name: appcontaineryaml
      properties:
        image: mcr.microsoft.com/azuredocs/aci-helloworld
        ports:
          - port: 80
            protocol: TCP
        resources:
          requests:
            cpu: 1.0
            memoryInGB: 1.5
      ipAddress:
        type: Private
        ports:
          - protocol: tcp
            port: '80'
      networkProfile:
        id: /subscriptions/<Subscription
ID>/resourceGroups/container/providers/Microsoft.Network/networkProfiles/aci-network-profile-aci-vnet-subnet
        osType: Linux
        restartPolicy: Always
      tags: null
      type: Microsoft.ContainerInstance/containerGroups

```

Deploy the container group with the [az container create](#) command, specifying the YAML file name for the `--file` parameter:

```
az container create --resource-group myResourceGroup --file vnet-deploy-aci.yaml
```

Once the deployment has completed, run the [az container show](#) command to display its status:

```

$ az container show --resource-group myResourceGroup --name appcontaineryaml --output table

```

Name	ResourceGroup	Status	Image	IP:ports	Network
CPU/Memory	OsType	Location			
-----	-----	-----	-----	-----	-----
appcontaineryaml	myResourceGroup	Running	mcr.microsoft.com/azuredocs/aci-helloworld	10.0.0.5:80	Private
1.0 core/1.5 gb	Linux	westus			

## Clean up resources

### Delete container instances

When you're done working with the container instances you created, delete them with the following commands:

```

az container delete --resource-group myResourceGroup --name appcontainer -y
az container delete --resource-group myResourceGroup --name commchecker -y
az container delete --resource-group myResourceGroup --name appcontaineryaml -y

```

### Delete network resources

The initial preview of this feature requires several additional commands to delete the network resources you created earlier. If you used the example commands in previous sections of this article to create your virtual network and subnet, then you can use the following script to delete those network resources.

Before executing the script, set the `RES_GROUP` variable to the name of the resource group containing the virtual network and subnet that should be deleted. Update the names of the virtual network and subnet if you did not use the `aci-vnet` and `aci-subnet` names suggested earlier. The script is formatted for the Bash shell. If you prefer another shell such as PowerShell or Command Prompt, you'll need to adjust variable assignment and accessors accordingly.

#### WARNING

This script deletes resources! It deletes the virtual network and all subnets it contains. Be sure that you no longer need *any* of the resources in the virtual network, including any subnets it contains, prior to running this script. Once deleted, **these resources are unrecoverable**.

```
# Replace <my-resource-group> with the name of your resource group
RES_GROUP=<my-resource-group>

# Get network profile ID
NETWORK_PROFILE_ID=$(az network profile list --resource-group $RES_GROUP --query [0].id --output tsv)

# Delete the network profile
az network profile delete --id $NETWORK_PROFILE_ID -y

# Get the service association link (SAL) ID
# Replace aci-vnet and aci-subnet with your VNet and subnet names in the following commands

SAL_ID=$(az network vnet subnet show --resource-group $RES_GROUP --vnet-name aci-vnet --name aci-subnet --query id --output tsv)/providers/Microsoft.ContainerInstance/serviceAssociationLinks/default

# Delete the default SAL ID for the subnet
az resource delete --ids $SAL_ID --api-version 2018-07-01

# Delete the subnet delegation to Azure Container Instances
az network vnet subnet update --resource-group $RES_GROUP --vnet-name aci-vnet --name aci-subnet --remove delegations 0

# Delete the subnet
az network vnet subnet delete --resource-group $RES_GROUP --vnet-name aci-vnet --name aci-subnet

# Delete virtual network
az network vnet delete --resource-group $RES_GROUP --name aci-vnet
```

## Next steps

To deploy a new virtual network, subnet, network profile, and container group using a Resource Manager template, see [Create an Azure container group with VNet](#).

Several virtual network resources and features were discussed in this article, though briefly. The Azure Virtual Network documentation covers these topics extensively:

- [Virtual network](#)
- [Subnet](#)
- [Service endpoints](#)
- [VPN Gateway](#)
- [ExpressRoute](#)

# Deploy to Azure Container Instances from Azure Container Registry

3/4/2019 • 5 minutes to read • [Edit Online](#)

[Azure Container Registry](#) is an Azure-based, managed container registry service used to store private Docker container images. This article describes how to deploy container images stored in an Azure container registry to Azure Container Instances.

## Prerequisites

**Azure container registry:** You need an Azure container registry--and at least one container image in the registry--to complete the steps in this article. If you need a registry, see [Create a container registry using the Azure CLI](#).

**Azure CLI:** The command-line examples in this article use the [Azure CLI](#) and are formatted for the Bash shell. You can [install the Azure CLI](#) locally, or use the [Azure Cloud Shell](#).

## Configure registry authentication

In any production scenario, access to an Azure container registry should be provided by using [service principals](#). Service principals allow you to provide [role-based access control](#) to your container images. For example, you can configure a service principal with pull-only access to a registry.

In the following section, you create an Azure key vault and a service principal, and store the service principal's credentials in the vault.

### Create key vault

If you don't already have a vault in [Azure Key Vault](#), create one with the Azure CLI using the following commands.

Update the `RES_GROUP` variable with the name of an existing resource group in which to create the key vault, and `ACR_NAME` with the name of your container registry. Specify a name for your new key vault in `AKV_NAME`. The vault name must be unique within Azure and must be 3-24 alphanumeric characters in length, begin with a letter, end with a letter or digit, and cannot contain consecutive hyphens.

```
RES_GROUP=myresourcegroup # Resource Group name
ACR_NAME=myregistry        # Azure Container Registry registry name
AKV_NAME=mykeyvault        # Azure Key Vault vault name

az keyvault create -g $RES_GROUP -n $AKV_NAME
```

### Create service principal and store credentials

You now need to create a service principal and store its credentials in your key vault.

The following command uses `az ad sp create-for-rbac` to create the service principal, and `az keyvault secret set` to store the service principal's **password** in the vault.

```
# Create service principal, store its password in AKV (the registry *password*)
az keyvault secret set \
  --vault-name $AKV_NAME \
  --name $ACR_NAME-pull-pwd \
  --value $(az ad sp create-for-rbac \
    --name http://$ACR_NAME-pull \
    --scopes $(az acr show --name $ACR_NAME --query id --output tsv) \
    --role acrpull \
    --query password \
    --output tsv)
```

The `--role` argument in the preceding command configures the service principal with the *acrpull* role, which grants it pull-only access to the registry. To grant both push and pull access, change the `--role` argument to *acrpush*.

Next, store the service principal's *appId* in the vault, which is the **username** you pass to Azure Container Registry for authentication.

```
# Store service principal ID in AKV (the registry *username*)
az keyvault secret set \
  --vault-name $AKV_NAME \
  --name $ACR_NAME-pull-usr \
  --value $(az ad sp show --id http://$ACR_NAME-pull --query appId --output tsv)
```

You've created an Azure Key Vault and stored two secrets in it:

- `$ACR_NAME-pull-usr`: The service principal ID, for use as the container registry **username**.
- `$ACR_NAME-pull-pwd`: The service principal password, for use as the container registry **password**.

You can now reference these secrets by name when you or your applications and services pull images from the registry.

## Deploy container with Azure CLI

Now that the service principal credentials are stored in Azure Key Vault secrets, your applications and services can use them to access your private registry.

First get the registry's login server name by using the [az acr show](#) command. The login server name is all lowercase and similar to `myregistry.azurecr.io`.

```
ACR_LOGIN_SERVER=$(az acr show --name $ACR_NAME --resource-group $RES_GROUP --query "loginServer" --output tsv)
```

Execute the following [az container create](#) command to deploy a container instance. The command uses the service principal's credentials stored in Azure Key Vault to authenticate to your container registry, and assumes you've previously pushed the [aci-helloworld](#) image to your registry. Update the `--image` value if you'd like to use a different image from your registry.

```
az container create \
  --name aci-demo \
  --resource-group $RES_GROUP \
  --image $ACR_LOGIN_SERVER/aci-helloworld:v1 \
  --registry-login-server $ACR_LOGIN_SERVER \
  --registry-username $(az keyvault secret show --vault-name $AKV_NAME -n $ACR_NAME-pull-usr --query value -o tsv) \
  --registry-password $(az keyvault secret show --vault-name $AKV_NAME -n $ACR_NAME-pull-pwd --query value -o tsv) \
  --dns-name-label aci-demo-$RANDOM \
  --query ipAddress.fqdn
```

The `--dns-name-label` value must be unique within Azure, so the preceding command appends a random number to the container's DNS name label. The output from the command displays the container's fully qualified domain name (FQDN), for example:

```
$ az container create --name aci-demo --resource-group $RES_GROUP --image $ACR_LOGIN_SERVER/aci-helloworld:v1 -
-registry-login-server $ACR_LOGIN_SERVER --registry-username $(az keyvault secret show --vault-name $AKV_NAME -
n $ACR_NAME-pull-usr --query value -o tsv) --registry-password $(az keyvault secret show --vault-name $AKV_NAME
-n $ACR_NAME-pull-pwd --query value -o tsv) --dns-name-label aci-demo-$RANDOM --query ipAddress.fqdn
"aci-demo-25007.eastus.azurecontainer.io"
```

Once the container has started successfully, you can navigate to its FQDN in your browser to verify the application is running successfully.

## Deploy with Azure Resource Manager template

You can specify the properties of your Azure Container Registry in an Azure Resource Manager template by including the `imageRegistryCredentials` property in the container group definition:

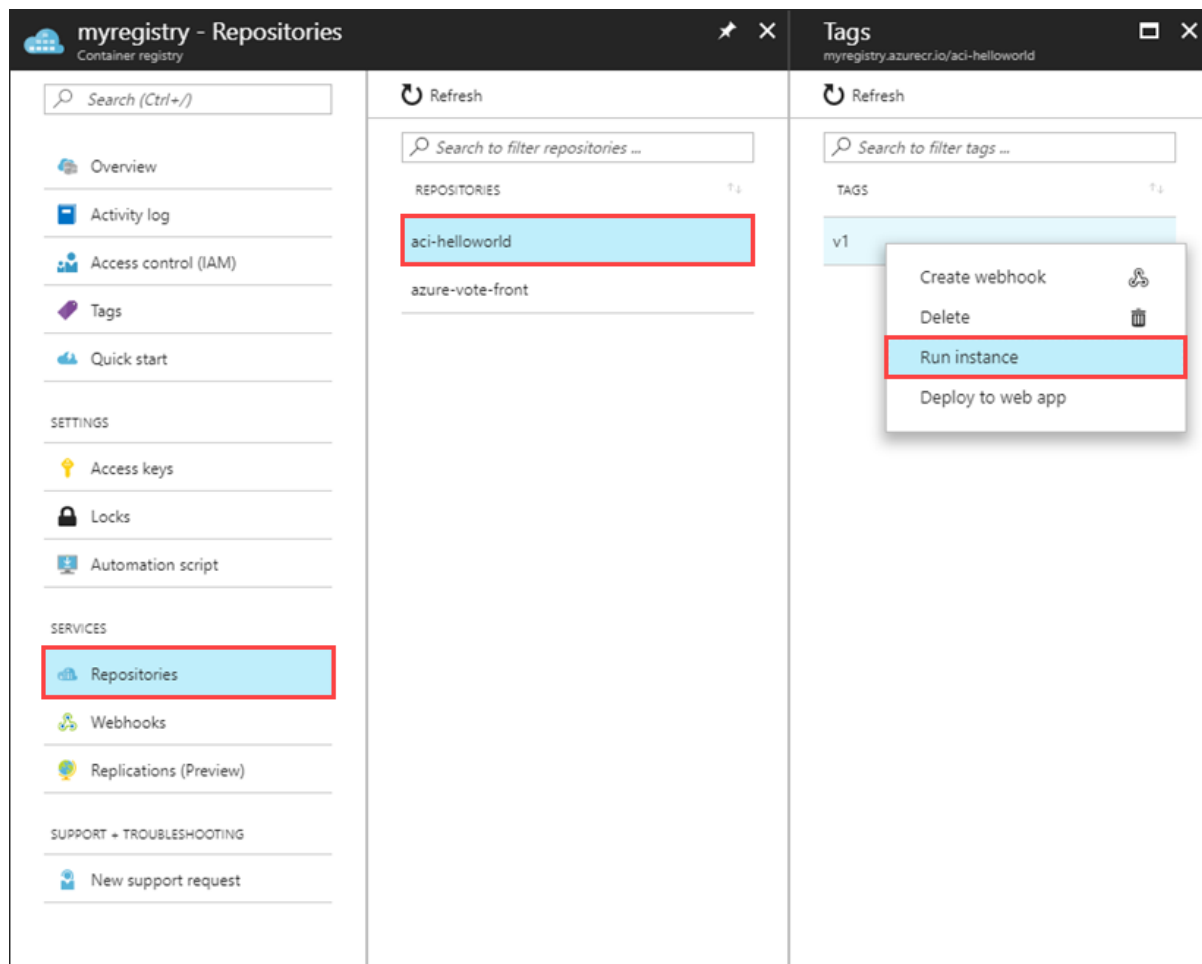
```
"imageRegistryCredentials": [
  {
    "server": "imageRegistryLoginServer",
    "username": "imageRegistryUsername",
    "password": "imageRegistryPassword"
  }
]
```

For details on referencing Azure Key Vault secrets in a Resource Manager template, see [Use Azure Key Vault to pass secure parameter value during deployment](#).

## Deploy with Azure portal

If you maintain container images in an Azure container registry, you can easily create a container in Azure Container Instances using the Azure portal. When using the portal to deploy a container instance from a container registry, you must enable the registry's [admin account](#). The admin account is designed for a single user to access the registry, mainly for testing purposes.

1. In the Azure portal, navigate to your container registry.
2. To confirm that the admin account is enabled, select **Access keys**, and under **Admin user** select **Enable**.
3. Select **Repositories**, then select the repository that you want to deploy from, right-click the tag for the container image you want to deploy, and select **Run instance**.



4. Enter a name for the container and a name for the resource group. You can also change the default values if you wish.

Create container instance

Container name

mycontainer

Container image

myregistry.azurecr.io/aci-helloworld:v1

OS type

Linux Windows

Subscription

Visual Studio Enterprise with MSDN

Resource group

Create new Use existing

myResourceGroup

Location

East US

Number of cores

1

Memory (GB)

1.5

Public IP address

Yes No

Port

80

OK

- Once the deployment completes, you can navigate to the container group from the notifications pane to find its IP address and other properties.

mycontainer

Container group

Search (Ctrl+I)

Overview

Activity log

Access control (IAM)

Tags

SETTINGS

Properties

Locks

Automation script

Delete

Resource group (change)

myResourceGroup

Status

Running

Location

East US

Subscription (change)

Visual Studio Enterprise with MSDN

Subscription ID

<Subscription ID>

OS type

Linux

IP address

52.170.115.171

1 container

NAME	IMAGE	STATE	START TIME	RESTART COUNT
mycontainer	myregistry.azurecr.io/aci-h...	Running	2017-12-22T22:44:41Z	0

## Next steps

For more information about Azure Container Registry authentication, see [Authenticate with an Azure container registry](#).



# Run containerized tasks with restart policies

4/15/2019 • 2 minutes to read • [Edit Online](#)

The ease and speed of deploying containers in Azure Container Instances provides a compelling platform for executing run-once tasks like build, test, and image rendering in a container instance.

With a configurable restart policy, you can specify that your containers are stopped when their processes have completed. Because container instances are billed by the second, you're charged only for the compute resources used while the container executing your task is running.

The examples presented in this article use the Azure CLI. You must have Azure CLI version 2.0.21 or greater [installed locally](#), or use the CLI in the [Azure Cloud Shell](#).

## Container restart policy

When you create a [container group](#) in Azure Container Instances, you can specify one of three restart policy settings.

RESTART POLICY	DESCRIPTION
<code>Always</code>	Containers in the container group are always restarted. This is the <b>default</b> setting applied when no restart policy is specified at container creation.
<code>Never</code>	Containers in the container group are never restarted. The containers run at most once.
<code>OnFailure</code>	Containers in the container group are restarted only when the process executed in the container fails (when it terminates with a nonzero exit code). The containers are run at least once.

## Specify a restart policy

How you specify a restart policy depends on how you create your container instances, such as with the Azure CLI, Azure PowerShell cmdlets, or in the Azure portal. In the Azure CLI, specify the `--restart-policy` parameter when you call [az container create](#).

```
az container create \  
  --resource-group myResourceGroup \  
  --name mycontainer \  
  --image mycontainerimage \  
  --restart-policy OnFailure
```

## Run to completion example

To see the restart policy in action, create a container instance from the Microsoft [aci-wordcount](#) image, and specify the `OnFailure` restart policy. This example container runs a Python script that, by default, analyzes the text of Shakespeare's [Hamlet](#), writes the 10 most common words to STDOUT, and then exits.

Run the example container with the following [az container create](#) command:

```
az container create \  
  --resource-group myResourceGroup \  
  --name mycontainer \  
  --image mcr.microsoft.com/azuredocs/aci-wordcount:latest \  
  --restart-policy OnFailure
```

Azure Container Instances starts the container, and then stops it when its application (or script, in this case) exits. When Azure Container Instances stops a container whose restart policy is `Never` or `OnFailure`, the container's status is set to **Terminated**. You can check a container's status with the [az container show](#) command:

```
az container show --resource-group myResourceGroup --name mycontainer --query  
containers[0].instanceView.currentState.state
```

Example output:

```
"Terminated"
```

Once the example container's status shows *Terminated*, you can see its task output by viewing the container logs. Run the [az container logs](#) command to view the script's output:

```
az container logs --resource-group myResourceGroup --name mycontainer
```

Output:

```
[('the', 990),  
 ('and', 702),  
 ('of', 628),  
 ('to', 610),  
 ('I', 544),  
 ('you', 495),  
 ('a', 453),  
 ('my', 441),  
 ('in', 399),  
 ('HAMLET', 386)]
```

This example shows the output that the script sent to STDOUT. Your containerized tasks, however, might instead write their output to persistent storage for later retrieval. For example, to an [Azure file share](#).

## Next steps

Task-based scenarios, such as batch processing a large dataset with several containers, can take advantage of custom [environment variables](#) or [command lines](#) at runtime.

For details on how to persist the output of your containers that run to completion, see [Mounting an Azure file share with Azure Container Instances](#).

# Set environment variables in container instances

4/18/2019 • 5 minutes to read • [Edit Online](#)

Setting environment variables in your container instances allows you to provide dynamic configuration of the application or script run by the container. This is similar to the `--env` command-line argument to `docker run`.

To set environment variables in a container, specify them when you create a container instance. This article shows examples of setting environment variables when you start a container with the [Azure CLI](#), [Azure PowerShell](#), and the [Azure portal](#).

For example, if you run the Microsoft [aci-wordcount](#) container image, you can modify its behavior by specifying the following environment variables:

*NumWords*: The number of words sent to STDOUT.

*MinLength*: The minimum number of characters in a word for it to be counted. A higher number ignores common words like "of" and "the."

If you need to pass secrets as environment variables, Azure Container Instances supports [secure values](#) for both Windows and Linux containers.

## NOTE

This article has been updated to use the new Azure PowerShell Az module. You can still use the AzureRM module, which will continue to receive bug fixes until at least December 2020. To learn more about the new Az module and AzureRM compatibility, see [Introducing the new Azure PowerShell Az module](#). For Az module installation instructions, see [Install Azure PowerShell](#).

## Azure CLI example

To see the default output of the [aci-wordcount](#) container, run it first with this [az container create](#) command (no environment variables specified):

```
az container create \
  --resource-group myResourceGroup \
  --name mycontainer1 \
  --image mcr.microsoft.com/azuredocs/aci-wordcount:latest \
  --restart-policy OnFailure
```

To modify the output, start a second container with the `--environment-variables` argument added, specifying values for the *NumWords* and *MinLength* variables. (This example assume you are running the CLI in a Bash shell or Azure Cloud Shell. If you use the Windows Command Prompt, specify the variables with double-quotes, such as `--environment-variables "NumWords"="5" "MinLength"="8" .`)

```
az container create \
  --resource-group myResourceGroup \
  --name mycontainer2 \
  --image mcr.microsoft.com/azuredocs/aci-wordcount:latest \
  --restart-policy OnFailure \
  --environment-variables 'NumWords'='5' 'MinLength'='8'
```

Once both containers' state shows as *Terminated* (use [az container show](#) to check state), display their logs with [az](#)

[container logs](#) to see the output.

```
az container logs --resource-group myResourceGroup --name mycontainer1
az container logs --resource-group myResourceGroup --name mycontainer2
```

The output of the containers show how you've modified the second container's script behavior by setting environment variables.

```
azureuser@Azure:~$ az container logs --resource-group myResourceGroup --name mycontainer1
[('the', 990),
 ('and', 702),
 ('of', 628),
 ('to', 610),
 ('I', 544),
 ('you', 495),
 ('a', 453),
 ('my', 441),
 ('in', 399),
 ('HAMLET', 386)]

azureuser@Azure:~$ az container logs --resource-group myResourceGroup --name mycontainer2
[('CLAUDIUS', 120),
 ('POLONIUS', 113),
 ('GERTRUDE', 82),
 ('ROSENCRANTZ', 69),
 ('GUILDENSTERN', 54)]
```

## Azure PowerShell example

Setting environment variables in PowerShell is similar to the CLI, but uses the `-EnvironmentVariable` command-line argument.

First, launch the [aci-wordcount](#) container in its default configuration with this [New-AzContainerGroup](#) command:

```
New-AzContainerGroup `
  -ResourceGroupName myResourceGroup `
  -Name mycontainer1 `
  -Image mcr.microsoft.com/azuredocs/aci-wordcount:latest
```

Now run the following [New-AzContainerGroup](#) command. This one specifies the *NumWords* and *MinLength* environment variables after populating an array variable, `envVars`:

```
$envVars = @{'NumWords'='5';'MinLength'='8'}
New-AzContainerGroup `
  -ResourceGroupName myResourceGroup `
  -Name mycontainer2 `
  -Image mcr.microsoft.com/azuredocs/aci-wordcount:latest `
  -RestartPolicy OnFailure `
  -EnvironmentVariable $envVars
```

Once both containers' state is *Terminated* (use [Get-AzContainerInstanceLog](#) to check state), pull their logs with the [Get-AzContainerInstanceLog](#) command.

```
Get-AzContainerInstanceLog -ResourceGroupName myResourceGroup -ContainerGroupName mycontainer1
Get-AzContainerInstanceLog -ResourceGroupName myResourceGroup -ContainerGroupName mycontainer2
```

The output for each container shows how you've modified the script run by the container by setting environment

variables.

```
PS Azure:\> Get-AzContainerInstanceLog -ResourceGroupName myResourceGroup -ContainerGroupName mycontainer1
[('the', 990),
 ('and', 702),
 ('of', 628),
 ('to', 610),
 ('I', 544),
 ('you', 495),
 ('a', 453),
 ('my', 441),
 ('in', 399),
 ('HAMLET', 386)]
```

Azure:\

```
PS Azure:\> Get-AzContainerInstanceLog -ResourceGroupName myResourceGroup -ContainerGroupName mycontainer2
[('CLAUDIUS', 120),
 ('POLONIUS', 113),
 ('GERTRUDE', 82),
 ('ROSENCRANTZ', 69),
 ('GUILDENSTERN', 54)]
```

Azure:\

## Azure portal example

To set environment variables when you start a container in the Azure portal, specify them in the **Advanced** page when you create the container.

1. On the **Advanced** page, set the **Restart policy** to *On failure*
2. Under **Environment variables**, enter `NumWords` with a value of `5` for the first variable, and enter `MinLength` with a value of `8` for the second variable.
3. Select **Review + create** to verify and then deploy the container.

### Create container instance

Basics Networking **Advanced** Tags Review + create

Configure additional container properties and variables.

Restart policy ⓘ On failure

Environment variables

KEY	VALUE	
NumWords	5	
MinLength	8	

Command override ⓘ Example: `/bin/bash -c "echo hello", /bin/bash -c "echo have a good day"`

Review + createPreviousNext : Tags >

To view the container's logs, under **Settings** select **Containers**, then **Logs**. Similar to the output shown in the previous CLI and PowerShell sections, you can see how the script's behavior has been modified by the environment variables. Only five words are displayed, each with a minimum length of eight characters.

Home > mycontainer - Containers

mycontainer - Containers

Container instances

Search (Ctrl+/)

Refresh

1 container

NAME	IMAGE	STATE	START TIME	RESTART COUNT
mycontainer	microsoft/aci-wordcount	Terminated	2018-05-15T17:23:34Z	0

Events Properties **Logs**

```
[ ('CLAUDIUS', 120),
  ('POLONIUS', 113),
  ('GERTRUDE', 82),
  ('ROSENCRANTZ', 69),
  ('GUILDENSTERN', 54)]
```

## Secure values

Objects with secure values are intended to hold sensitive information like passwords or keys for your application. Using secure values for environment variables is both safer and more flexible than including it in your container's image. Another option is to use secret volumes, described in [Mount a secret volume in Azure Container Instances](#).

Environment variables with secure values aren't visible in your container's properties--their values can be accessed only from within the container. For example, container properties viewed in the Azure portal or Azure CLI display only a secure variable's name, not its value.

Set a secure environment variable by specifying the `secureValue` property instead of the regular `value` for the variable's type. The two variables defined in the following YAML demonstrate the two variable types.

### YAML deployment

Create a `secure-env.yaml` file with the following snippet.

```
apiVersion: 2018-10-01
location: eastus
name: securetest
properties:
  containers:
    - name: mycontainer
      properties:
        environmentVariables:
          - name: 'NOTSECRET'
            value: 'my-exposed-value'
          - name: 'SECRET'
            secureValue: 'my-secret-value'
        image: nginx
        ports: []
        resources:
          requests:
            cpu: 1.0
            memoryInGB: 1.5
        osType: Linux
        restartPolicy: Always
      tags: null
    type: Microsoft.ContainerInstance/containerGroups
```

Run the following command to deploy the container group with YAML (adjust the resource group name as necessary):

```
az container create --resource-group myResourceGroup --file secure-env.yaml
```

## Verify environment variables

Run the [az container show](#) command to query your container's environment variables:

```
az container show --resource-group myResourceGroup --name securetest --query  
'containers[].environmentVariables'
```

The JSON response shows both the insecure environment variable's key and value, but only the name of the secure environment variable:

```
[  
  [  
    {  
      "name": "NOTSECRET",  
      "secureValue": null,  
      "value": "my-exposed-value"  
    },  
    {  
      "name": "SECRET",  
      "secureValue": null,  
      "value": null  
    }  
  ]  
]
```

With the [az container exec](#) command, which enables executing a command in a running container, you can verify that the secure environment variable has been set. Run the following command to start an interactive bash session in the container:

```
az container exec --resource-group myResourceGroup --name securetest --exec-command "/bin/bash"
```

Once you've opened an interactive shell within the container, you can access the `SECRET` variable's value:

```
root@caas-ef3ee231482549629ac8a40c0d3807fd-3881559887-53741:/# echo $SECRET  
my-secret-value
```

## Next steps

Task-based scenarios, such as batch processing a large dataset with several containers, can benefit from custom environment variables at runtime. For more information about running task-based containers, see [Run containerized tasks with restart policies](#).

# Set the command line in a container instance to override the default command line operation

4/26/2019 • 3 minutes to read • [Edit Online](#)

When you create a container instance, optionally specify a command to override the default command line instruction baked into the container image. This behavior is similar to the `--entrypoint` command-line argument to `docker run`.

Like setting [environment variables](#) for container instances, specifying a starting command line is useful for batch jobs where you need to prepare each container dynamically with task-specific configuration.

## Command line guidelines

- By default, the command line specifies a *single process that starts without a shell* in the container. For example, the command line might run a Python script or executable file.
- To execute multiple commands, begin your command line by setting a shell environment that is supported in the container operating system. Examples:

OPERATING SYSTEM	DEFAULT SHELL
Ubuntu	<code>/bin/bash</code>
Alpine	<code>/bin/sh</code>
Windows	<code>cmd</code>

Follow the conventions of the shell to combine multiple commands to run in sequence.

- Depending on the container configuration, you might need to set a full path to the command line executable or arguments.
- Set an appropriate [restart policy](#) for the container instance, depending on whether the command-line specifies a long-running task or a run-once task. For example, a restart policy of `Never` or `OnFailure` is recommended for a run-once task.
- If you need information about the default entrypoint set in a container image, use the [docker image inspect](#) command.

## Command line syntax

The command line syntax varies depending on the Azure API or tool used to create the instances. If you specify a shell environment, also observe the command syntax conventions of the shell.

- [az container create](#) command: Pass a string with the `--command-line` parameter. Example:  
`--command-line "python myscript.py arg1 arg2"`.
- [New-AzureRmContainerGroup](#) Azure PowerShell cmdlet: Pass a string with the `-Command` parameter. Example: `-Command "echo hello"`.
- Azure portal: In the **Command override** property of the container configuration, provide a comma-



separated list of strings, without quotes. Example: `python, myscript.py, arg1, arg2` ).

- Resource Manager template or YAML file, or one of the Azure SDKs: Specify the command line property as an array of strings. Example: the JSON array `["python", "myscript.py", "arg1", "arg2"]` in a Resource Manager template.

If you're familiar with [Dockerfile](#) syntax, this format is similar to the `exec` form of the CMD instruction.

## Examples

	AZURE CLI	PORTAL	TEMPLATE
Single command	<pre>--command-line "python myscript.py arg1 arg2"</pre>	<b>Command override:</b> <pre>python, myscript.py, arg1, arg2</pre>	<pre>"command": ["python", "myscript.py", "arg1", "arg2"]</pre>
Multiple commands	<pre>--command-line "/bin/bash -c 'mkdir test; touch test/myfile; tail -f /dev/null'"</pre>	<b>Command override:</b> <pre>/bin/bash, -c, mkdir test; touch test/myfile; tail -f /dev/null</pre>	<pre>"command": ["/bin/bash", "-c", "mkdir test; touch test/myfile; tail -f /dev/null"]</pre>

## Azure CLI example

As an example, modify the behavior of the [microsoft/aci-wordcount](#) container image, which analyzes text in Shakespeare's *Hamlet* to find the most frequently occurring words. Instead of analyzing *Hamlet*, you could set a command line that points to a different text source.

To see the output of the [microsoft/aci-wordcount](#) container when it analyzes the default text, run it with the following [az container create](#) command. No start command line is specified, so the default container command runs. For illustration purposes, this example sets [environment variables](#) to find the top 3 words that are at least five letters long:

```
az container create \  
  --resource-group myResourceGroup \  
  --name mycontainer1 \  
  --image mcr.microsoft.com/azuredocs/aci-wordcount:latest \  
  --environment-variables NumWords=3 MinLength=5 \  
  --restart-policy OnFailure
```

Once the container's state shows as *Terminated* (use [az container show](#) to check state), display the log with [az container logs](#) to see the output.

```
az container logs --resource-group myResourceGroup --name mycontainer1
```

Output:

```
[('HAMLET', 386), ('HORATIO', 127), ('CLAUDIUS', 120)]
```

Now set up a second example container to analyze different text by specifying a different command line. The Python script executed by the container, *wordcount.py*, accepts a URL as an argument, and processes that page's content instead of the default.

For example, to determine the top 3 words that are at least five letters long in *Romeo and Juliet*:

```
az container create \  
  --resource-group myResourceGroup \  
  --name mycontainer2 \  
  --image mcr.microsoft.com/azuredocs/aci-wordcount:latest \  
  --restart-policy OnFailure \  
  --environment-variables NumWords=3 MinLength=5 \  
  --command-line "python wordcount.py http://shakespeare.mit.edu/romeo_juliet/full.html"
```

Again, once the container is *Terminated*, view the output by showing the container's logs:

```
az container logs --resource-group myResourceGroup --name mycontainer2
```

Output:

```
[('ROMEO', 177), ('JULIET', 134), ('CAPULET', 119)]
```

## Next steps

Task-based scenarios, such as batch processing a large dataset with several containers, can benefit from custom command lines at runtime. For more information about running task-based containers, see [Run containerized tasks with restart policies](#).

# Execute a command in a running Azure container instance

10/8/2018 • 2 minutes to read • [Edit Online](#)

Azure Container Instances supports executing a command in a running container. Running a command in a container you've already started is especially helpful during application development and troubleshooting. The most common use of this feature is to launch an interactive shell so that you can debug issues in a running container.

## Run a command with Azure CLI

Execute a command in a running container with `az container exec` in the [Azure CLI](#):

```
az container exec --resource-group <group-name> --name <container-group-name> --exec-command "<command>"
```

For example, to launch a Bash shell in an Nginx container:

```
az container exec --resource-group myResourceGroup --name mynginx --exec-command "/bin/bash"
```

In the example output below, the Bash shell is launched in a running Linux container, providing a terminal in which `ls` is executed:

```
$ az container exec --resource-group myResourceGroup --name mynginx --exec-command "/bin/bash"
root@caas-83e6c883014b427f9b277a2bba3b7b5f-708716530-2qv47:/# ls
bin  dev  home  lib64  mnt  proc  run  srv  tmp  var
boot  etc  lib  media  opt  root  sbin  sys  usr
root@caas-83e6c883014b427f9b277a2bba3b7b5f-708716530-2qv47:/# exit
exit
Bye.
```

In this example, Command Prompt is launched in a running Nanoserver container:

```
$ az container exec --resource-group myResourceGroup --name myiis --exec-command "cmd.exe"
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.

C:\>dir
Volume in drive C has no label.
Volume Serial Number is 76E0-C852

Directory of C:\

03/23/2018  09:13 PM    <DIR>          inetpub
11/20/2016  11:32 AM                1,894 License.txt
03/23/2018  09:13 PM    <DIR>          Program Files
07/16/2016  12:09 PM    <DIR>          Program Files (x86)
03/13/2018  08:50 PM        171,616 ServiceMonitor.exe
03/23/2018  09:13 PM    <DIR>          Users
03/23/2018  09:12 PM    <DIR>          var
03/23/2018  09:22 PM    <DIR>          Windows
                2 File(s)          173,510 bytes
                6 Dir(s)  21,171,609,600 bytes free

C:\>exit
Bye.
```

## Multi-container groups

If your [container group](#) has multiple containers, such as an application container and a logging sidecar, specify the name of the container in which to run the command with `--container-name`.

For example, in the container group *mynginx* are two containers, *nginx-app* and *logger*. To launch a shell on the *nginx-app* container:

```
az container exec --resource-group myResourceGroup --name mynginx --container-name nginx-app --exec-command
"/bin/bash"
```

## Restrictions

Azure Container Instances currently supports launching a single process with [az container exec](#), and you cannot pass command arguments. For example, you cannot chain commands like in `sh -c "echo FOO && echo BAR"`, or execute `echo FOO`.

## Next steps

Learn about other troubleshooting tools and common deployment issues in [Troubleshoot container and deployment issues in Azure Container Instances](#).

# Use Azure Container Instances as a Jenkins build agent

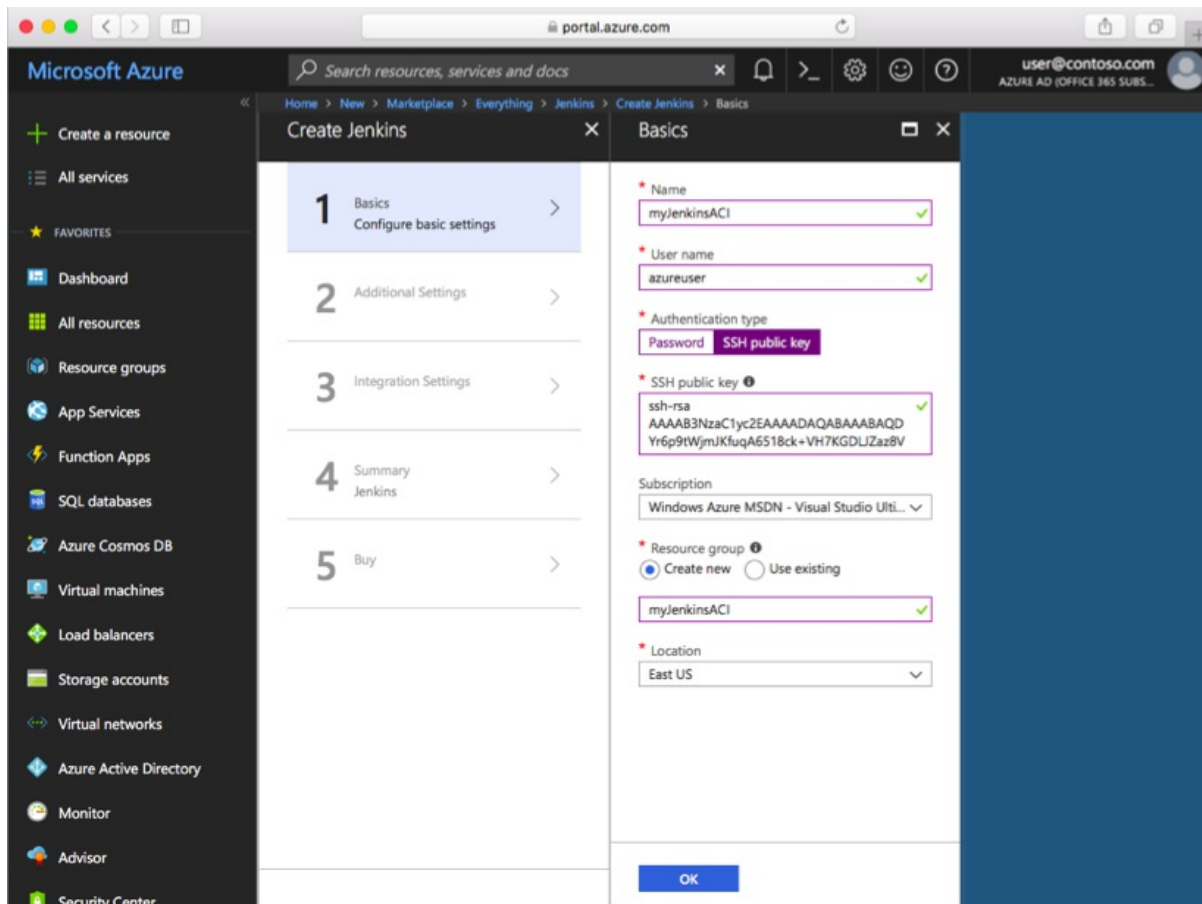
3/14/2019 • 4 minutes to read • [Edit Online](#)

Azure Container Instances (ACI) provides an on-demand, burstable, and isolated environment for running containerized workloads. Because of these attributes, ACI makes a great platform for running Jenkins build jobs at a large scale. This article walks through deploying and using a Jenkins server that's pre-configured with ACI as a build target.

For more information on Azure Container Instances, see [About Azure Container Instances](#).

## Deploy a Jenkins server

1. In the Azure portal, select **Create a resource** and search for **Jenkins**. Select the Jenkins offering with a publisher of **Microsoft**, and then select **Create**.
2. Enter the following information on the **Basics** form, and then select **OK**.
  - **Name:** Enter a name for the Jenkins deployment.
  - **User name:** Enter a name for the admin user of the Jenkins virtual machine.
  - **Authentication type:** We recommend an SSH public key for authentication. If you select this option, paste in an SSH public key to be used for logging in to the Jenkins virtual machine.
  - **Subscription:** Select an Azure subscription.
  - **Resource group:** Create a resource group or select an existing one.
  - **Location:** Select a location for the Jenkins server.



The screenshot shows the 'Create Jenkins' Basics form in the Azure portal. The form is titled 'Create Jenkins' and has a 'Basics' tab selected. The form contains the following fields and options:

- Name:** myjenkinsACI (with a green checkmark)
- User name:** azureuser (with a green checkmark)
- Authentication type:** Password (selected) and SSH public key (with a green checkmark)
- SSH public key:** ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQD Yr6p9tWjmKfuqA6518ck+VH7KGDLIZaz8V (with a green checkmark)
- Subscription:** Windows Azure MSDN - Visual Studio Ulti... (with a dropdown arrow)
- Resource group:** Create new (selected) and Use existing (with a green checkmark)
- Location:** East US (with a dropdown arrow)

The form also includes a 'Buy' button at the bottom right.

3. On the **Additional Settings** form, complete the following items:

- **Size:** Select the appropriate sizing option for your Jenkins virtual machine.
- **VM disk type:** Specify either **HDD** (hard-disk drive) or **SSD** (solid-state drive) for the Jenkins server.
- **Virtual network:** Select the arrow if you want to modify the default settings.
- **Subnets:** Select the arrow, verify the information, and select **OK**.
- **Public IP address:** Select the arrow to give the public IP address a custom name, configure the SKU, and set the assignment method.
- **Domain name label:** Specify a value to create a fully qualified URL to the Jenkins virtual machine.
- **Jenkins release type:** Select the desired release type from the options: **LTS**, **Weekly build**, or **Azure Verified**.

Home > New > Marketplace > Everything > Jenkins > Create Jenkins > Additional Settings

### Create Jenkins

1 Basics  
Done ✓

2 Additional Settings >

3 Integration Settings >

4 Summary  
Jenkins >

5 Buy >

\* Size  
1x Standard DS2 v2 >

VM disk type ⓘ  
**SSD** HDD

\* Virtual network  
(new) jenkins-vnet >

\* Subnets  
Review subnet configuration >

\* Public IP address ⓘ  
(new) jenkins-pip >

\* Domain name label  
**myjenkinsaci** ✓  
eastus.cloudapp.azure.com

Jenkins release type  
LTS ▼

OK

4. For service principal integration, select **Auto(MSI)** to have [managed identities for Azure resources](#) automatically create an authentication identity for the Jenkins instance. Select **Manual** to provide your own service principal credentials.
5. Cloud agents configure a cloud-based platform for Jenkins build jobs. For the sake of this article, select **ACI**. With the ACI cloud agent, each Jenkins build job is run in a container instance.

Home > New > Marketplace > Everything > Jenkins > Create Jenkins > Jenkins Integration Settings

Create Jenkins

Jenkins Integration Settings

1 Basics Done ✓

2 Additional Settings Done ✓

3 Integration Settings >

4 Summary Jenkins >

5 Buy >

Service Principal Integration ⓘ  
Auto(MSI) ▾

Enable Cloud Agents ⓘ  

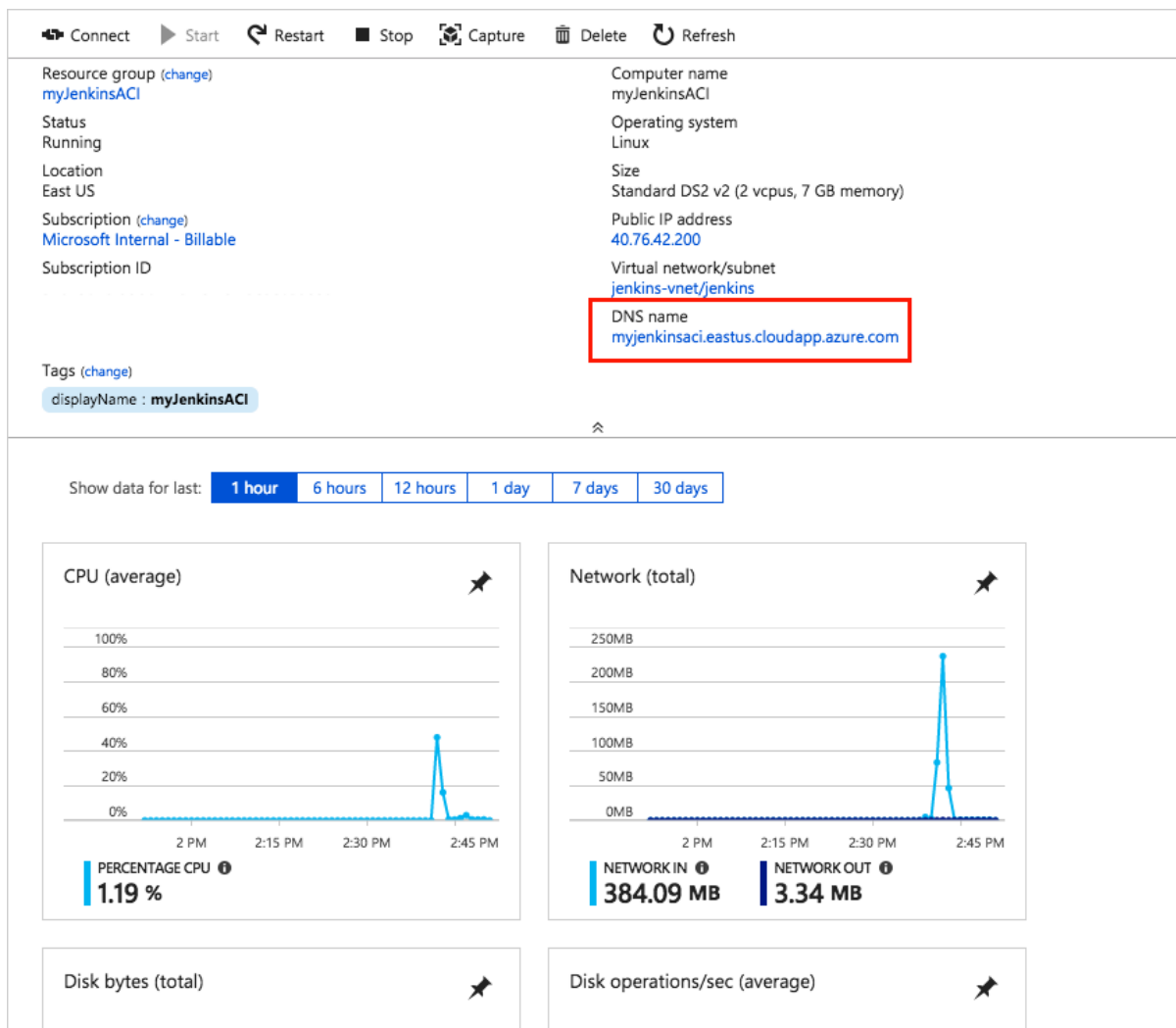
No ACI VM

OK

6. When you're done with the integration settings, select **OK**, and then select **OK** again on the validation summary. Select **Create** on the **Terms of use** summary. The Jenkins server takes a few minutes to deploy.

## Configure Jenkins

1. In the Azure portal, browse to the Jenkins resource group, select the Jenkins virtual machine, and take note of the DNS name.



2. Browse to the DNS name of the Jenkins VM and copy the returned SSH string.





3. Open a terminal session on your development system, and paste in the SSH string from the last step. Update `username` to the username that you specified when you deployed the Jenkins server.
4. After the session is connected, run the following command to retrieve the initial admin password:

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

5. Leave the SSH session and tunnel running, and go to `http://localhost:8080` in a browser. Paste the initial admin password into the box, and then select **Continue**.

**Getting Started**

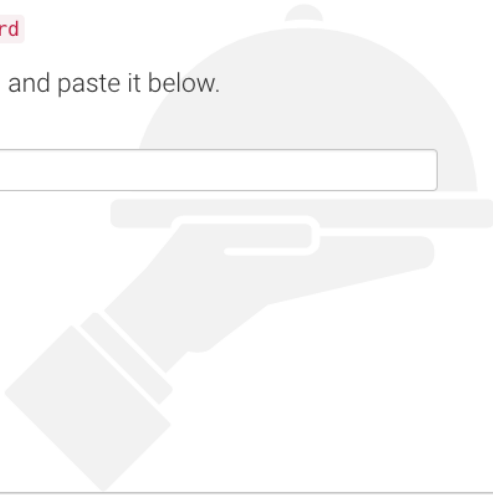
# Unlock Jenkins

To ensure Jenkins is securely set up by the administrator, a password has been written to the log ([not sure where to find it?](#)) and this file on the server:

```
/var/lib/jenkins/secrets/initialAdminPassword
```

Please copy the password from either location and paste it below.

**Administrator password**



Continue

6. Select **Install suggested plugins** to install all recommended Jenkins plugins.

Getting Started

# Customize Jenkins

Plugins extend Jenkins with additional features to support many different needs.

## Install suggested plugins

Install plugins the Jenkins community finds most useful.

## Select plugins to install

Select and install plugins most suitable for your needs.

Jenkins 2.121.1

7. Create an admin user account. This account is used for logging in to and working with your Jenkins instance.

Getting Started

# Create First Admin User

Username:

azureuser

Password:

.....

Confirm password:

.....

Full name:

first last

E-mail address:

user@contoso.com

Jenkins 2.121.1

[Continue as admin](#)[Save and Continue](#)

8. Select **Save and Finish**, and then select **Start using Jenkins** to complete the configuration.

Jenkins is now configured and ready to build and deploy code. For this example, a simple Java application is used to demonstrate a Jenkins build on Azure Container Instances.

## Create a build job


Now, a Jenkins build job is created to demonstrate Jenkins builds on an Azure container instance.

1. Select **New Item**, give the build project a name such as **aci-demo**, select **Freestyle project**, and select **OK**.


### Enter an item name

aci-demo


» Required field

**Freestyle project**


This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

**Pipeline**


Orchestrates long-running activities that can span multiple build agents. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.

**Multi-configuration project**


Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.

**Folder**

Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.

**GitHub Organization**

Scans a GitHub organization (or user account) for all repositories matching some defined markers.

**Multibranch Pipeline**

Creates a set of Pipeline projects according to detected branches in one SCM repository.

OK

2. Under **General**, ensure that **Restrict where this project can be run** is selected. Enter **linux** for the label expression. This configuration ensures that this build job runs on the ACI cloud.

**General** Source Code Management Build Triggers Build Environment Build Post-build Actions

>>

Description

[Plain text] [Preview](#)

- ☐ Enable project-based security
- ☐ Discard old builds ?
- ☐ GitHub project
- ☐ Permission to Copy Artifact
- ☐ This project is parameterized ?
- ☐ Throttle builds ?
- ☐ Disable this project ?
- ☐ Execute concurrent builds if necessary ?
- ☒ Restrict where this project can be run ?

Label Expression  ?

[Label linux](#) is serviced by no nodes and 1 cloud. Permissions or other restrictions provided by plugins may prevent this job from running on those nodes.

[Advanced...](#)

3. Under **Build**, select **Add build step** and select **Execute Shell**. Enter `echo "aci-demo"` as the command.

**Build**

**Execute shell** X ?

Command `echo "aci-demo"`

See [the list of available environment variables](#)

[Advanced...](#)

[Add build step](#) ▼

**Post-build Actions**

[Add post-build action](#) ▼

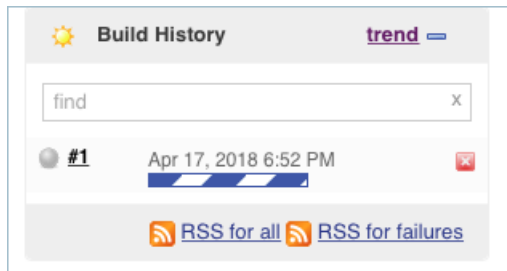
[Save](#) [Apply](#)

4. Select **Save**.

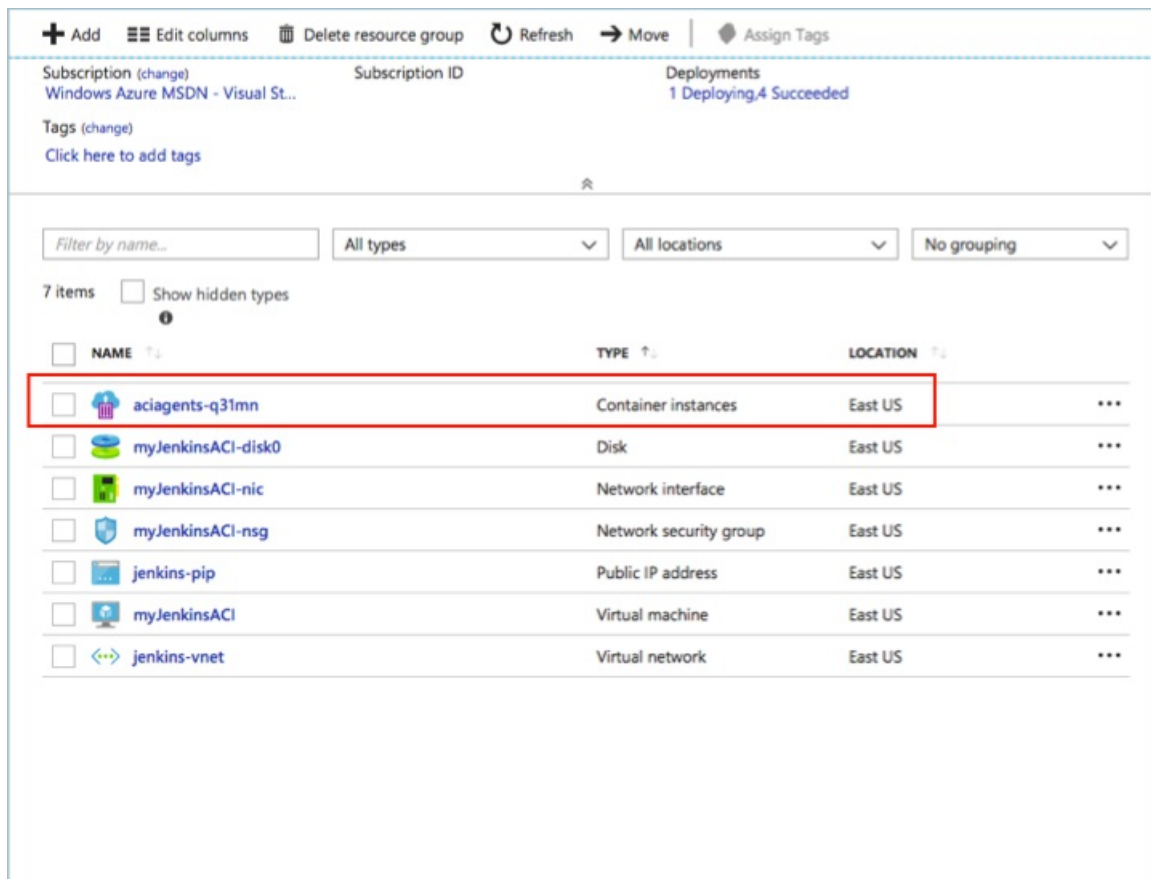
## Run the build job

To test the build job and observe Azure Container Instances as the build platform, manually start a build.

1. Select **Build Now** to start a build job. It takes a few minutes for the job to start. You should see a status that's similar to the following image:



2. While the job is running, open the Azure portal and look at the Jenkins resource group. You should see that a container instance has been created. The Jenkins job is running inside this instance.



3. As Jenkins runs more jobs than the configured number of Jenkins executors (default 2), multiple container instances are created.

+ Add   Edit columns   Delete resource group   Refresh   Move   Assign Tags

Subscription (change)   Subscription ID   Deployments  
 Windows Azure MSDN - Visual St...   1 Deploying, 5 Succeeded

Tags (change)  
 Click here to add tags

---

Filter by name...   All types   All locations   No grouping

8 items   ☐ Show hidden types

<input type="checkbox"/>	NAME ↑↓	TYPE ↑↓	LOCATION ↑↓	
<input type="checkbox"/>	aciagents-1cdk2	Container instances	East US	...
<input type="checkbox"/>	aciagents-jmkcb	Container instances	East US	...
<input type="checkbox"/>	myJenkinsACI-disk0	Disk	East US	...
<input type="checkbox"/>	myJenkinsACI-nic	Network interface	East US	...
<input type="checkbox"/>	myJenkinsACI-nsg	Network security group	East US	...
<input type="checkbox"/>	jenkins-pip	Public IP address	East US	...
<input type="checkbox"/>	myJenkinsACI	Virtual machine	East US	...
<input type="checkbox"/>	jenkins-vnet	Virtual network	East US	...

4. After all build jobs have finished, the container instances are removed.

+ Add   Edit columns   Delete resource group   Refresh   Move   Assign Tags

Subscription (change)   Subscription ID   Deployments  
 Windows Azure MSDN - Visual St...   4 Succeeded

Tags (change)  
 Click here to add tags

---

Filter by name...   All types   All locations   No grouping

6 items   ☐ Show hidden types

<input type="checkbox"/>	NAME ↑↓	TYPE ↑↓	LOCATION ↑↓	
<input type="checkbox"/>	myJenkinsACI-disk0	Disk	East US	...
<input type="checkbox"/>	myJenkinsACI-nic	Network interface	East US	...
<input type="checkbox"/>	myJenkinsACI-nsg	Network security group	East US	...
<input type="checkbox"/>	jenkins-pip	Public IP address	East US	...
<input type="checkbox"/>	myJenkinsACI	Virtual machine	East US	...
<input type="checkbox"/>	jenkins-vnet	Virtual network	East US	...

## Troubleshooting the Jenkins plugin

If you encounter any bugs with the Jenkins plugins, file an issue in the [Jenkins JIRA](#) for the specific component.

## Next steps

To learn more about Jenkins on Azure, see [Azure and Jenkins](#).

# How to use managed identities with Azure Container Instances

3/21/2019 • 11 minutes to read • [Edit Online](#)

Use [managed identities for Azure resources](#) to run code in Azure Container Instances that interacts with other Azure services - without maintaining any secrets or credentials in code. The feature provides an Azure Container Instances deployment with an automatically managed identity in Azure Active Directory.

In this article, you learn more about managed identities in Azure Container Instances and:

- Enable a user-assigned or system-assigned identity in a container group
- Grant the identity access to an Azure Key Vault
- Use the managed identity to access a Key Vault from a running container

Adapt the examples to enable and use identities in Azure Container Instances to access other Azure services. These examples are interactive. However, in practice your container images would run code to access Azure services.

## NOTE

Currently you cannot use a managed identity in a container group deployed to a virtual network.

## Why use a managed identity?

Use a managed identity in a running container to authenticate to any [service that supports Azure AD authentication](#) without managing credentials in your container code. For services that don't support AD authentication, you can store secrets in Azure Key Vault and use the managed identity to access Key Vault to retrieve credentials. For more information about using a managed identity, see [What is managed identities for Azure resources?](#)

## IMPORTANT

This feature is currently in preview. Previews are made available to you on the condition that you agree to the [supplemental terms of use](#). Some aspects of this feature may change prior to general availability (GA). Currently, managed identities are only supported on Linux container instances.

### Enable a managed identity

In Azure Container Instances, managed identities for Azure resources are supported as of REST API version 2018-10-01 and corresponding SDKs and tools. When you create a container group, enable one or more managed identities by setting a [ContainerGroupIdentity](#) property. You can also enable or update managed identities after a container group is running; either action causes the container group to restart. To set the identities on a new or existing container group, use the Azure CLI, a Resource Manager template, or a YAML file.

Azure Container Instances supports both types of managed Azure identities: user-assigned and system-assigned. On a container group, you can enable a system-assigned identity, one or more user-assigned identities, or both types of identities.

- A **user-assigned** managed identity is created as a standalone Azure resource in the Azure AD tenant that's trusted by the subscription in use. After the identity is created, the identity can be assigned to one or more Azure resources (in Azure Container Instances or other Azure services). The lifecycle of a user-assigned



identity is managed separately from the lifecycle of the container groups or other service resources to which it's assigned. This behavior is especially useful in Azure Container Instances. Because the identity extends beyond the lifetime of a container group, you can reuse it along with other standard settings to make your container group deployments highly repeatable.

- A **system-assigned** managed identity is enabled directly on a container group in Azure Container Instances. When it's enabled, Azure creates an identity for the group in the Azure AD tenant that's trusted by the subscription of the instance. After the identity is created, the credentials are provisioned in each container in the container group. The lifecycle of a system-assigned identity is directly tied to the container group that it's enabled on. When the group is deleted, Azure automatically cleans up the credentials and the identity in Azure AD.




### Use a managed identity

To use a managed identity, the identity must initially be granted access to one or more Azure service resources (such as a Web App, a Key Vault, or a Storage Account) in the subscription. To access the Azure resources from a running container, your code must acquire an *access token* from an Azure AD endpoint. Then, your code sends the access token on a call to a service that supports Azure AD authentication.

Using a managed identity in a running container is essentially the same as using an identity in an Azure VM. See the VM guidance for using a [token](#), [Azure PowerShell](#) or [Azure CLI](#), or the [Azure SDKs](#).

## Open Azure Cloud Shell

Azure Cloud Shell is a free, interactive shell that you can use to run the steps in this article. Common Azure tools are preinstalled and configured in Cloud Shell for you to use with your account. Select **Copy** to copy the code, paste it in Cloud Shell, and then press Enter to run it. There are a few ways to open Cloud Shell:

Select <b>Try It</b> in the upper-right corner of a code block.	
Open Cloud Shell in your browser.	
Select the <b>Cloud Shell</b> button on the menu in the upper-right corner of the <a href="#">Azure portal</a> .	

If you choose to install and use the CLI locally, this article requires that you are running the Azure CLI version 2.0.49 or later. Run `az --version` to find the version. If you need to install or upgrade, see [Install Azure CLI](#).

## Create an Azure Key Vault

The examples in this article use a managed identity in Azure Container Instances to access an Azure Key Vault secret.

First, create a resource group named *myResourceGroup* in the *eastus* location with the following [az group create](#) command:

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

Use the [az keyvault create](#) command to create a Key Vault. Be sure to specify a unique Key Vault name.

```
az keyvault create --name mykeyvault --resource-group myResourceGroup --location eastus
```

Store a sample secret in the Key Vault using the [az keyvault secret set](#) command:

```
az keyvault secret set --name SampleSecret --value "Hello Container Instances!" --description ACISecret --vault-name mykeyvault
```

Continue with the following examples to access the Key Vault using either a user-assigned or system-assigned managed identity in Azure Container Instances.

## Example 1: Use a user-assigned identity to access Azure Key Vault

### Create an identity

First create an identity in your subscription using the [az identity create](#) command. You can use the same resource group used to create the Key Vault, or use a different one.

```
az identity create --resource-group myResourceGroup --name myACIID
```

To use the identity in the following steps, use the [az identity show](#) command to store the identity's service principal ID and resource ID in variables.

```
# Get service principal ID of the user-assigned identity
spID=$(az identity show --resource-group myResourceGroup --name myACIID --query principalId --output tsv)

# Get resource ID of the user-assigned identity
resourceID=$(az identity show --resource-group myResourceGroup --name myACIID --query id --output tsv)
```

### Enable a user-assigned identity on a container group

Run the following [az container create](#) command to create a container instance based on Ubuntu Server. This example provides a single-container group that you can use to interactively access other Azure services. The `--assign-identity` parameter passes your user-assigned managed identity to the group. The long-running command keeps the container running. This example uses the same resource group used to create the Key Vault, but you could specify a different one.

```
az container create --resource-group myResourceGroup --name mycontainer --image microsoft/azure-cli --assign-identity $resourceID --command-line "tail -f /dev/null"
```

Within a few seconds, you should get a response from the Azure CLI indicating that the deployment has completed. Check its status with the [az container show](#) command.

```
az container show --resource-group myResourceGroup --name mycontainer
```

The `identity` section in the output looks similar to the following, showing the identity is set in the container group. The `principalID` under `userAssignedIdentities` is the service principal of the identity you created in Azure Active Directory:

```
...
"identity": {
  "principalId": "null",
  "tenantId": "xxxxxxx-f292-4e60-9122-xxxxxxxxxxxx",
  "type": "UserAssigned",
  "userAssignedIdentities": {
    "/subscriptions/xxxxxxx-0903-4b79-a55a-
xxxxxxx/resourcegroups/danlep1018/providers/Microsoft.ManagedIdentity/userAssignedIdentities/myACIID": {
      "clientId": "xxxxxxx-5523-45fc-9f49-xxxxxxxxxxxx",
      "principalId": "xxxxxxx-f25b-4895-b828-xxxxxxxxxxxx"
    }
  }
},
...
```

## Grant user-assigned identity access to the Key Vault

Run the following [az keyvault set-policy](#) command to set an access policy on the Key Vault. The following example allows the user-assigned identity to get secrets from the Key Vault:

```
az keyvault set-policy --name mykeyvault --resource-group myResourceGroup --object-id $spID --secret-
permissions get
```

## Use user-assigned identity to get secret from Key Vault

Now you can use the managed identity to access the Key Vault within the running container instance. For this example, first launch a bash shell in the container:

```
az container exec --resource-group myResourceGroup --name mycontainer --exec-command "/bin/bash"
```

Run the following commands in the bash shell in the container. To get an access token to use Azure Active Directory to authenticate to Key Vault, run the following command:

```
curl 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-
01&resource=https%3A%2F%2Fvault.azure.net%2F' -H Metadata:true -s
```

Output:

```
{"access_token":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Imk2bEdrM0ZaenhSY1ViMkMzbkVRN3
N5SEpsWSIsImtpZCI6Imk2bEdrM0ZaenhSY1ViMkMzbkVRN3N5SEpsWSJ9.....xxxxxxxxxxxxxxxx","refresh_token":"","expires
_in":"28799","expires_on":"1539927532","not_before":"1539898432","resource":"https://vault.azure.net/","token_
type":"Bearer"}
```

To store the access token in a variable to use in subsequent commands to authenticate, run the following command:

```
token=$(curl 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-
01&resource=https%3A%2F%2Fvault.azure.net' -H Metadata:true | jq -r '.access_token')
```

Now use the access token to authenticate to Key Vault and read a secret. Be sure to substitute the name of your key vault in the URL (<https://mykeyvault.vault.azure.net/>):

```
curl https://mykeyvault.vault.azure.net/secrets/SampleSecret/?api-version=2016-10-01 -H "Authorization: Bearer
$token"
```

The response looks similar to the following, showing the secret. In your code, you would parse this output to obtain the secret. Then, use the secret in a subsequent operation to access another Azure resource.

```
{
  "value": "Hello Container Instances!",
  "contentType": "ACIsecret",
  "id": "https://mykeyvault.vault.azure.net/secrets/SampleSecret/xxxxxxxxxx",
  "attributes": {
    "enabled": true,
    "created": 1539965967,
    "updated": 1539965967,
    "recoveryLevel": "Purgeable",
    "tags": {
      "file-encoding": "utf-8"
    }
  }
}
```

## Example 2: Use a system-assigned identity to access Azure Key Vault

### Enable a system-assigned identity on a container group

Run the following [az container create](#) command to create a container instance based on Ubuntu Server. This example provides a single-container group that you can use to interactively access other Azure services. The `--assign-identity` parameter with no additional value enables a system-assigned managed identity on the group. The long-running command keeps the container running. This example uses the same resource group used to create the Key Vault, but you could specify a different one.

```
az container create --resource-group myResourceGroup --name mycontainer --image microsoft/azure-cli --assign-identity --command-line "tail -f /dev/null"
```

Within a few seconds, you should get a response from the Azure CLI indicating that the deployment has completed. Check its status with the [az container show](#) command.

```
az container show --resource-group myResourceGroup --name mycontainer
```

The `identity` section in the output looks similar to the following, showing that a system-assigned identity is created in Azure Active Directory:

```
...
"identity": {
  "principalId": "xxxxxxxx-528d-7083-b74c-xxxxxxxxxxxx",
  "tenantId": "xxxxxxx-f292-4e60-9122-xxxxxxxxxxxx",
  "type": "SystemAssigned",
  "userAssignedIdentities": null
},
...
```

Set a variable to the value of `principalId` (the service principal ID) of the identity, to use in later steps.

```
spID=$(az container show --resource-group myResourceGroup --name mycontainer --query identity.principalId --out tsv)
```

### Grant container group access to the Key Vault

Run the following [az keyvault set-policy](#) command to set an access policy on the Key Vault. The following example allows the system-managed identity to get secrets from the Key Vault:

```
az keyvault set-policy --name mykeyvault --resource-group myResourceGroup --object-id $spID --secret-permissions get
```

### Use container group identity to get secret from Key Vault

Now you can use the managed identity to access the Key Vault within the running container instance. For this

example, first launch a bash shell in the container:

```
az container exec --resource-group myResourceGroup --name mycontainer --exec-command "/bin/bash"
```

Run the following commands in the bash shell in the container. To get an access token to use Azure Active Directory to authenticate to Key Vault, run the following command:

```
curl 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https%3A%2F%2Fvault.azure.net%2F' -H Metadata:true -s
```

Output:

```
{"access_token":"xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Imk2bEdrM0ZaenhSY1ViMkMzbkVRN3N5SEpsWSIsImtpZCI6Imk2bEdrM0ZaenhSY1ViMkMzbkVRN3N5SEpsWSJ9.....xxxxxxxxxxxxxxxxxxxx","refresh_token":"","expires_in":"28799","expires_on":"1539927532","not_before":"1539898432","resource":"https://vault.azure.net/","token_type":"Bearer"}
```

To store the access token in a variable to use in subsequent commands to authenticate, run the following command:

```
token=$(curl 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https%3A%2F%2Fvault.azure.net' -H Metadata:true | jq -r '.access_token')
```

Now use the access token to authenticate to Key Vault and read a secret. Be sure to substitute the name of your key vault in the URL (*<https://mykeyvault.vault.azure.net/>*):

```
curl https://mykeyvault.vault.azure.net/secrets/SampleSecret/?api-version=2016-10-01 -H "Authorization: Bearer $token"
```

The response looks similar to the following, showing the secret. In your code, you would parse this output to obtain the secret. Then, use the secret in a subsequent operation to access another Azure resource.

```
{"value":"Hello Container Instances!","contentType":"ACISecret","id":"https://mykeyvault.vault.azure.net/secrets/SampleSecret/xxxxxxxxxxxxxxxx","attributes":{"enabled":true,"created":1539965967,"updated":1539965967,"recoveryLevel":"Purgeable"},"tags":{"file-encoding":"utf-8"}}
```

## Enable managed identity using Resource Manager template

To enable a managed identity in a container group using a [Resource Manager template](#), set the `identity` property of the `Microsoft.ContainerInstance/containerGroups` object with a `ContainerGroupIdentity` object. The following snippets show the `identity` property configured for different scenarios. See the [Resource Manager template reference](#). Specify an `apiVersion` of `2018-10-01`.

### User-assigned identity

A user-assigned identity is a resource ID of the form:

```
"/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{identityName}"
```

You can enable one or more user-assigned identities.

```
"identity": {
  "type": "UserAssigned",
  "userAssignedIdentities": {
    "myResourceID1": {
    }
  }
}
```

### System-assigned identity

```
"identity": {
  "type": "SystemAssigned"
}
```

### System- and user-assigned identities

On a container group, you can enable both a system-assigned identity and one or more user-assigned identities.

```
"identity": {
  "type": "System Assigned, UserAssigned",
  "userAssignedIdentities": {
    "myResourceID1": {
    }
  }
}
...
```

## Enable managed identity using YAML file

To enable a managed identity in a container group deployed using a [YAML file](#), include the following YAML. Specify an `apiVersion` of `2018-10-01`.

### User-assigned identity

A user-assigned identity is a resource ID of the form

```
'/subscriptions/{subscriptionId}/resourceGroups/{resourceGroupName}/providers/Microsoft.ManagedIdentity/userAssignedIdentities/{identityName}'
```

You can enable one or more user-assigned identities.

```
identity:
  type: UserAssigned
  userAssignedIdentities:
    {'myResourceID1':{}}
```

### System-assigned identity

```
identity:
  type: SystemAssigned
```

### System- and user-assigned identities

On a container group, you can enable both a system-assigned identity and one or more user-assigned identities.

```
identity:
  type: SystemAssigned, UserAssigned
  userAssignedIdentities:
    {'myResourceID1':{}}
```

## Next steps

In this article, you learned about managed identities in Azure Container Instances and how to:

- Enable a user-assigned or system-assigned identity in a container group
- Grant the identity access to an Azure Key Vault
- Use the managed identity to access a Key Vault from a running container
- Learn more about [managed identities for Azure resources](#).
- See an [Azure Go SDK example](#) of using a managed identity to access a Key Vault from Azure Container Instances.

# Deploy container instances that use GPU resources

4/18/2019 • 5 minutes to read • [Edit Online](#)

To run certain compute-intensive workloads on Azure Container Instances, deploy your [container groups](#) with *GPU resources*. The container instances in the group can access one or more NVIDIA Tesla GPUs while running container workloads such as CUDA and deep learning applications.

This article shows how to add GPU resources when you deploy a container group by using a [YAML file](#) or [Resource Manager template](#). You can also specify GPU resources when you deploy a container instance using the Azure portal.

## IMPORTANT

This feature is currently in preview, and some [limitations apply](#). Previews are made available to you on the condition that you agree to the [supplemental terms of use](#). Some aspects of this feature may change prior to general availability (GA).

## Preview limitations

In preview, the following limitations apply when using GPU resources in container groups.

### Supported regions

- East US
- West US 2
- South Central US
- West Europe
- North Europe
- East Asia
- Central India

Support will be added for additional regions over time.

**Supported OS types:** Linux only

**Additional limitations:** GPU resources can't be used when deploying a container group into a [virtual network](#).

## About GPU resources

### Count and SKU

To use GPUs in a container instance, specify a *GPU resource* with the following information:

- **Count** - The number of GPUs: **1**, **2**, or **4**.
- **SKU** - The GPU SKU: **K80**, **P100**, or **V100**. Each SKU maps to the NVIDIA Tesla GPU in one the following Azure GPU-enabled VM families:

SKU	VM FAMILY
K80	<a href="#">NC</a>
P100	<a href="#">NCv2</a>



SKU	VM FAMILY
V100	NCv3

### Resource availability

OS	GPU SKU	GPU COUNT	CPU	MEMORY (GB)
Linux	K80	1	6	56
Linux	K80	2	12	112
Linux	K80	4	24	224
Linux	P100	1	6	112
Linux	P100	2	12	224
Linux	P100	4	24	448
Linux	V100	1	6	112
Linux	V100	2	12	224
Linux	V100	4	24	448

When deploying GPU resources, set CPU and memory resources appropriate for the workload, up to the maximum values shown in the preceding table. These values are currently larger than the CPU and memory resources available in container groups without GPU resources.

### Things to know

- **Deployment time** - Creation of a container group containing GPU resources takes up to **8-10 minutes**. This is due to the additional time to provision and configure a GPU VM in Azure.
- **Pricing** - Similar to container groups without GPU resources, Azure bills for resources consumed over the *duration* of a container group with GPU resources. The duration is calculated from the time to pull your first container's image until the container group terminates. It does not include the time to deploy the container group.

See [pricing details](#).

- **CUDA drivers** - Container instances with GPU resources are pre-provisioned with NVIDIA CUDA drivers and container runtimes, so you can use container images developed for CUDA workloads.

We support CUDA 9.0 at this stage. For example, you can use following base images for your Docker file:

- [nvidia/cuda:9.0-base-ubuntu16.04](#)
- [tensorflow/tensorflow: 1.12.0-gpu-py3](#)

## YAML example

One way to add GPU resources is to deploy a container group by using a [YAML file](#). Copy the following YAML into a new file named *gpu-deploy-aci.yaml*, then save the file. This YAML creates a container group named *gpucontainergroup* specifying a container instance with a K80 GPU. The instance runs a sample CUDA vector addition application. The resource requests are sufficient to run the workload.

```
additional_properties: {}
apiVersion: '2018-10-01'
name: gpucontainergroup
properties:
  containers:
  - name: gpucontainer
    properties:
      image: k8s-gcrio.azureedge.net/cuda-vector-add:v0.1
      resources:
        requests:
          cpu: 1.0
          memoryInGB: 1.5
          gpu:
            count: 1
            sku: K80
    osType: Linux
    restartPolicy: OnFailure
```

Deploy the container group with the [az container create](#) command, specifying the YAML file name for the `--file` parameter. You need to supply the name of a resource group and a location for the container group such as *eastus* that supports GPU resources.

```
az container create --resource-group myResourceGroup --file gpu-deploy-aci.yaml --location eastus
```

The deployment takes several minutes to complete. Then, the container starts and runs a CUDA vector addition operation. Run the [az container logs](#) command to view the log output:

```
az container logs --resource-group myResourceGroup --name gpucontainergroup --container-name gpucontainer
```

Output:

```
[Vector addition of 50000 elements]
Copy input data from the host memory to the CUDA device
CUDA kernel launch with 196 blocks of 256 threads
Copy output data from the CUDA device to the host memory
Test PASSED
Done
```

## Resource Manager template example

Another way to deploy a container group with GPU resources is by using a [Resource Manager template](#). Start by creating a file named `gpudeploy.json`, then copy the following JSON into it. This example deploys a container instance with a V100 GPU that runs a [TensorFlow](#) training job against the [MNIST dataset](#). The resource requests are sufficient to run the workload.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "containerGroupName": {
      "type": "string",
      "defaultValue": "gpucontainergroupm",
      "metadata": {
        "description": "Container Group name."
      }
    }
  },
  "variables": {
    "containername": "gpucontainer",
    "containerimage": "microsoft/samples-tf-mnist-demo:gpu"
  },
  "resources": [
    {
      "name": "[parameters('containerGroupName')]",
      "type": "Microsoft.ContainerInstance/containerGroups",
      "apiVersion": "2018-10-01",
      "location": "[resourceGroup().location]",
      "properties": {
        "containers": [
          {
            "name": "[variables('containername')]",
            "properties": {
              "image": "[variables('containerimage')]",
              "resources": {
                "requests": {
                  "cpu": 4.0,
                  "memoryInGb": 12.0,
                  "gpu": {
                    "count": 1,
                    "sku": "V100"
                  }
                }
              }
            }
          }
        ],
        "osType": "Linux",
        "restartPolicy": "OnFailure"
      }
    }
  ]
}
```

Deploy the template with the [az group deployment create](#) command. You need to supply the name of a resource group that was created in a region such as *eastus* that supports GPU resources.

```
az group deployment create --resource-group myResourceGroup --template-file gpudeploy.json
```

The deployment takes several minutes to complete. Then, the container starts and runs the TensorFlow job. Run the [az container logs](#) command to view the log output:

```
az container logs --resource-group myResourceGroup --name gpucontainergroupm --container-name gpucontainer
```

Output:

```
2018-10-25 18:31:10.155010: I tensorflow/core/platform/cpu_feature_guard.cc:137] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: SSE4.1 SSE4.2 AVX AVX2 FMA
2018-10-25 18:31:10.305937: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1030] Found device 0 with
properties:
name: Tesla K80 major: 3 minor: 7 memoryClockRate(GHz): 0.8235
pciBusID: ccb6:00:00.0
totalMemory: 11.92GiB freeMemory: 11.85GiB
2018-10-25 18:31:10.305981: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1120] Creating TensorFlow
device (/device:GPU:0) -> (device: 0, name: Tesla K80, pci bus id: ccb6:00:00.0, compute capability: 3.7)
2018-10-25 18:31:14.941723: I tensorflow/stream_executor/dso_loader.cc:139] successfully opened CUDA library
libcupti.so.8.0 locally
Successfully downloaded train-images-idx3-ubyte.gz 9912422 bytes.
Extracting /tmp/tensorflow/input_data/train-images-idx3-ubyte.gz
Successfully downloaded train-labels-idx1-ubyte.gz 28881 bytes.
Extracting /tmp/tensorflow/input_data/train-labels-idx1-ubyte.gz
Successfully downloaded t10k-images-idx3-ubyte.gz 1648877 bytes.
Extracting /tmp/tensorflow/input_data/t10k-images-idx3-ubyte.gz
Successfully downloaded t10k-labels-idx1-ubyte.gz 4542 bytes.
Extracting /tmp/tensorflow/input_data/t10k-labels-idx1-ubyte.gz
Accuracy at step 0: 0.097
Accuracy at step 10: 0.6993
Accuracy at step 20: 0.8208
Accuracy at step 30: 0.8594
...
Accuracy at step 990: 0.969
Adding run metadata for 999
```

## Clean up resources

Because using GPU resources may be expensive, ensure that your containers don't run unexpectedly for long periods. Monitor your containers in the Azure portal, or check the status of a container group with the [az container show](#) command. For example:

```
az container show --resource-group myResourceGroup --name gpucontainergroup --output table
```

When you're done working with the container instances you created, delete them with the following commands:

```
az container delete --resource-group myResourceGroup --name gpucontainergroup -y
az container delete --resource-group myResourceGroup --name gpucontainergroup -y
```

## Next steps

- Learn more about deploying a container group using a [YAML file](#) or [Resource Manager template](#).
- Learn more about [GPU optimized VM sizes](#) in Azure.

# Mount an Azure file share in Azure Container Instances

3/22/2019 • 3 minutes to read • [Edit Online](#)

By default, Azure Container Instances are stateless. If the container crashes or stops, all of its state is lost. To persist state beyond the lifetime of the container, you must mount a volume from an external store. This article shows how to mount an Azure file share created with [Azure Files](#) for use with Azure Container Instances. Azure Files offers fully managed file shares in the cloud that are accessible via the industry standard Server Message Block (SMB) protocol. Using an Azure file share with Azure Container Instances provides file-sharing features similar to using an Azure file share with Azure virtual machines.

## NOTE

Mounting an Azure Files share is currently restricted to Linux containers. While we are working to bring all features to Windows containers, you can find current platform differences in [Quotas and region availability for Azure Container Instances](#).

## Create an Azure file share

Before using an Azure file share with Azure Container Instances, you must create it. Run the following script to create a storage account to host the file share, and the share itself. The storage account name must be globally unique, so the script adds a random value to the base string.

```
# Change these four parameters as needed
ACI_PERS_RESOURCE_GROUP=myResourceGroup
ACI_PERS_STORAGE_ACCOUNT_NAME=mystorageaccount$RANDOM
ACI_PERS_LOCATION=eastus
ACI_PERS_SHARE_NAME=acishare

# Create the storage account with the parameters
az storage account create \
  --resource-group $ACI_PERS_RESOURCE_GROUP \
  --name $ACI_PERS_STORAGE_ACCOUNT_NAME \
  --location $ACI_PERS_LOCATION \
  --sku Standard_LRS

# Create the file share
az storage share create --name $ACI_PERS_SHARE_NAME --account-name $ACI_PERS_STORAGE_ACCOUNT_NAME
```

## Get storage credentials

To mount an Azure file share as a volume in Azure Container Instances, you need three values: the storage account name, the share name, and the storage access key.

If you used the script above, the storage account name was stored in the `$ACI_PERS_STORAGE_ACCOUNT_NAME` variable. To see the account name, type:

```
echo $ACI_PERS_STORAGE_ACCOUNT_NAME
```

The share name is already known (defined as *acishare* in the script above), so all that remains is the storage

account key, which can be found using the following command:

```
STORAGE_KEY=$(az storage account keys list --resource-group $ACI_PERS_RESOURCE_GROUP --account-name  
$ACI_PERS_STORAGE_ACCOUNT_NAME --query "[0].value" --output tsv)  
echo $STORAGE_KEY
```

## Deploy container and mount volume

To mount an Azure file share as a volume in a container, specify the share and volume mount point when you create the container with [az container create](#). If you've followed the previous steps, you can mount the share you created earlier by using the following command to create a container:

```
az container create \  
  --resource-group $ACI_PERS_RESOURCE_GROUP \  
  --name hellofiles \  
  --image mcr.microsoft.com/azuredocs/aci-hellofiles \  
  --dns-name-label aci-demo \  
  --ports 80 \  
  --azure-file-volume-account-name $ACI_PERS_STORAGE_ACCOUNT_NAME \  
  --azure-file-volume-account-key $STORAGE_KEY \  
  --azure-file-volume-share-name $ACI_PERS_SHARE_NAME \  
  --azure-file-volume-mount-path /aci/logs/
```

The `--dns-name-label` value must be unique within the Azure region you create the container instance. Update the value in the preceding command if you receive a **DNS name label** error message when you execute the command.

## Manage files in mounted volume

Once the container starts up, you can use the simple web app deployed via the Microsoft [aci-hellofiles](#) image to create small text files in the Azure file share at the mount path you specified. Obtain the web app's fully qualified domain name (FQDN) with the [az container show](#) command:

```
az container show --resource-group $ACI_PERS_RESOURCE_GROUP --name hellofiles --query ipAddress.fqdn
```

You can use the [Azure portal](#) or a tool like the [Microsoft Azure Storage Explorer](#) to retrieve and inspect the file written to the file share.

## Mount multiple volumes

To mount multiple volumes in a container instance, you must deploy using an [Azure Resource Manager template](#) or a YAML file.

To use a template, provide the share details and define the volumes by populating the `volumes` array in the `properties` section of the template. For example, if you've created two Azure Files shares named *share1* and *share2* in storage account *myStorageAccount*, the `volumes` array would appear similar to the following:

```

"volumes": [{
  "name": "myvolume1",
  "azureFile": {
    "shareName": "share1",
    "storageAccountName": "myStorageAccount",
    "storageAccountKey": "<storage-account-key>"
  }
},
{
  "name": "myvolume2",
  "azureFile": {
    "shareName": "share2",
    "storageAccountName": "myStorageAccount",
    "storageAccountKey": "<storage-account-key>"
  }
}]

```

Next, for each container in the container group in which you'd like to mount the volumes, populate the `volumeMounts` array in the `properties` section of the container definition. For example, this mounts the two volumes, *myvolume1* and *myvolume2*, previously defined:

```

"volumeMounts": [{
  "name": "myvolume1",
  "mountPath": "/mnt/share1/"
},
{
  "name": "myvolume2",
  "mountPath": "/mnt/share2/"
}]

```

To see an example of container instance deployment with an Azure Resource Manager template, see [Deploy a container group](#). For an example using a YAML file, see [Deploy a multi-container group with YAML](#).

## Next steps

Learn how to mount other volume types in Azure Container Instances:

- [Mount an emptyDir volume in Azure Container Instances](#)
- [Mount a gitRepo volume in Azure Container Instances](#)
- [Mount a secret volume in Azure Container Instances](#)

# Mount a secret volume in Azure Container Instances

3/22/2019 • 3 minutes to read • [Edit Online](#)

Use a *secret* volume to supply sensitive information to the containers in a container group. The *secret* volume stores your secrets in files within the volume, accessible by the containers in the container group. By storing secrets in a *secret* volume, you can avoid adding sensitive data like SSH keys or database credentials to your application code.

All *secret* volumes are backed by [tmpfs](#), a RAM-backed filesystem; their contents are never written to non-volatile storage.

## NOTE

*Secret* volumes are currently restricted to Linux containers. Learn how to pass secure environment variables for both Windows and Linux containers in [Set environment variables](#). While we're working to bring all features to Windows containers, you can find current platform differences in [Quotas and region availability for Azure Container Instances](#).

## Mount secret volume - Azure CLI

To deploy a container with one or more secrets by using the Azure CLI, include the `--secrets` and `--secrets-mount-path` parameters in the [az container create](#) command. This example mounts a *secret* volume consisting of two secrets, "mysecret1" and "mysecret2," at `/mnt/secrets`:

```
az container create \
  --resource-group myResourceGroup \
  --name secret-volume-demo \
  --image mcr.microsoft.com/azuredocs/aci-helloworld \
  --secrets mysecret1="My first secret F00" mysecret2="My second secret BAR" \
  --secrets-mount-path /mnt/secrets
```

The following [az container exec](#) output shows opening a shell in the running container, listing the files within the secret volume, then displaying their contents:

```
$ az container exec --resource-group myResourceGroup --name secret-volume-demo --exec-command "/bin/sh"
/usr/src/app # ls -l /mnt/secrets
mysecret1
mysecret2
/usr/src/app # cat /mnt/secrets/mysecret1
My first secret F00
/usr/src/app # cat /mnt/secrets/mysecret2
My second secret BAR
/usr/src/app # exit
Bye.
```

## Mount secret volume - YAML

You can also deploy container groups with the Azure CLI and a [YAML template](#). Deploying by YAML template is the preferred method when deploying container groups consisting of multiple containers.

When you deploy with a YAML template, the secret values must be **Base64-encoded** in the template. However, the secret values appear in plaintext within the files in the container.



The following YAML template defines a container group with one container that mounts a *secret* volume at `/mnt/secrets`. The secret volume has two secrets, "mysecret1" and "mysecret2."

```
apiVersion: '2018-06-01'
location: eastus
name: secret-volume-demo
properties:
  containers:
    - name: aci-tutorial-app
      properties:
        environmentVariables: []
        image: mcr.microsoft.com/azuredocs/aci-helloworld:latest
        ports: []
        resources:
          requests:
            cpu: 1.0
            memoryInGB: 1.5
        volumeMounts:
          - mountPath: /mnt/secrets
            name: secretvolume1
      osType: Linux
      restartPolicy: Always
      volumes:
        - name: secretvolume1
          secret:
            mysecret1: TXkgZmlyc3Qgc2VjcmV0IEZPTwo=
            mysecret2: TXkgc2Vjb25kIHNLy3JldCBCQVlK
      tags: {}
  type: Microsoft.ContainerInstance/containerGroups
```

To deploy with the YAML template, save the preceding YAML to a file named `deploy-aci.yaml`, then execute the [az container create](#) command with the `--file` parameter:

```
# Deploy with YAML template
az container create --resource-group myResourceGroup --file deploy-aci.yaml
```

## Mount secret volume - Resource Manager

In addition to CLI and YAML deployment, you can deploy a container group using an Azure [Resource Manager template](#).

First, populate the `volumes` array in the container group `properties` section of the template. When you deploy with a Resource Manager template, the secret values must be **Base64-encoded** in the template. However, the secret values appear in plaintext within the files in the container.

Next, for each container in the container group in which you'd like to mount the *secret* volume, populate the `volumeMounts` array in the `properties` section of the container definition.

The following Resource Manager template defines a container group with one container that mounts a *secret* volume at `/mnt/secrets`. The secret volume has two secrets, "mysecret1" and "mysecret2."

```

{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "variables": {
    "container1name": "aci-tutorial-app",
    "container1image": "microsoft/aci-helloworld:latest"
  },
  "resources": [
    {
      "name": "secret-volume-demo",
      "type": "Microsoft.ContainerInstance/containerGroups",
      "apiVersion": "2018-06-01",
      "location": "[resourceGroup().location]",
      "properties": {
        "containers": [
          {
            "name": "[variables('container1name')]",
            "properties": {
              "image": "[variables('container1image')]",
              "resources": {
                "requests": {
                  "cpu": 1,
                  "memoryInGb": 1.5
                }
              },
              "ports": [
                {
                  "port": 80
                }
              ],
              "volumeMounts": [
                {
                  "name": "secretvolume1",
                  "mountPath": "/mnt/secrets"
                }
              ]
            }
          }
        ],
        "osType": "Linux",
        "ipAddress": {
          "type": "Public",
          "ports": [
            {
              "protocol": "tcp",
              "port": "80"
            }
          ]
        },
        "volumes": [
          {
            "name": "secretvolume1",
            "secret": {
              "mysecret1": "TXkgZmlyc3Qgc2VjcmV0IEZPTwo=",
              "mysecret2": "TXkgc2Vjb25kIHNNlY3JldCBCQVlK"
            }
          }
        ]
      }
    }
  ]
}

```

To deploy with the Resource Manager template, save the preceding JSON to a file named `deploy-aci.json`, then execute the `az group deployment create` command with the `--template-file` parameter:

```
# Deploy with Resource Manager template
az group deployment create --resource-group myResourceGroup --template-file deploy-aci.json
```

## Next steps

### Volumes

Learn how to mount other volume types in Azure Container Instances:

- [Mount an Azure file share in Azure Container Instances](#)
- [Mount an emptyDir volume in Azure Container Instances](#)
- [Mount a gitRepo volume in Azure Container Instances](#)

### Secure environment variables

Another method for providing sensitive information to containers (including Windows containers) is through the use of [secure environment variables](#).

# Mount an emptyDir volume in Azure Container Instances

10/8/2018 • 2 minutes to read • [Edit Online](#)

Learn how to mount an *emptyDir* volume to share data between the containers in a container group in Azure Container Instances.

## NOTE

Mounting an *emptyDir* volume is currently restricted to Linux containers. While we are working to bring all features to Windows containers, you can find current platform differences in [Quotas and region availability for Azure Container Instances](#).

## emptyDir volume

The *emptyDir* volume provides a writable directory accessible to each container in a container group. Containers in the group can read and write the same files in the volume, and it can be mounted using the same or different paths in each container.

Some example uses for an *emptyDir* volume:

- Scratch space
- Checkpointing during long-running tasks
- Store data retrieved by a sidecar container and served by an application container

Data in an *emptyDir* volume is persisted through container crashes. Containers that are restarted, however, are not guaranteed to persist the data in an *emptyDir* volume.

## Mount an emptyDir volume

To mount an *emptyDir* volume in a container instance, you must deploy using an [Azure Resource Manager template](#).

First, populate the `volumes` array in the container group `properties` section of the template. Next, for each container in the container group in which you'd like to mount the *emptyDir* volume, populate the `volumeMounts` array in the `properties` section of the container definition.

For example, the following Resource Manager template creates a container group consisting of two containers, each of which mounts the *emptyDir* volume:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "variables": {
    "container1name": "aci-tutorial-app",
    "container1image": "microsoft/aci-helloworld:latest",
    "container2name": "aci-tutorial-sidecar",
    "container2image": "microsoft/aci-tutorial-sidecar"
  },
  "resources": [
    {
      "name": "volume-demo-emptydir",
      "type": "Microsoft.ContainerInstance/containerGroups"
```

```

    type: "Microsoft.ContainerInstance/containerGroups",
    "apiVersion": "2018-02-01-preview",
    "location": "[resourceGroup().location]",
    "properties": {
      "containers": [
        {
          "name": "[variables('container1name')]",
          "properties": {
            "image": "[variables('container1image')]",
            "resources": {
              "requests": {
                "cpu": 1,
                "memoryInGb": 1.5
              }
            },
            "ports": [
              {
                "port": 80
              }
            ],
            "volumeMounts": [
              {
                "name": "emptydir1",
                "mountPath": "/mnt/empty"
              }
            ]
          }
        },
        {
          "name": "[variables('container2name')]",
          "properties": {
            "image": "[variables('container2image')]",
            "resources": {
              "requests": {
                "cpu": 1,
                "memoryInGb": 1.5
              }
            },
            "volumeMounts": [
              {
                "name": "emptydir1",
                "mountPath": "/mnt/empty"
              }
            ]
          }
        }
      ],
      "osType": "Linux",
      "ipAddress": {
        "type": "Public",
        "ports": [
          {
            "protocol": "tcp",
            "port": "80"
          }
        ]
      },
      "volumes": [
        {
          "name": "emptydir1",
          "emptyDir": {}
        }
      ]
    }
  }
}

```

To see an example of container instance deployment with an Azure Resource Manager template, see [Deploy multi-container groups in Azure Container Instances](#).

## Next steps

Learn how to mount other volume types in Azure Container Instances:

- [Mount an Azure file share in Azure Container Instances](#)
- [Mount a gitRepo volume in Azure Container Instances](#)
- [Mount a secret volume in Azure Container Instances](#)

# Mount a gitRepo volume in Azure Container Instances

3/22/2019 • 4 minutes to read • [Edit Online](#)

Learn how to mount a *gitRepo* volume to clone a Git repository into your container instances.

## NOTE

Mounting a *gitRepo* volume is currently restricted to Linux containers. While we are working to bring all features to Windows containers, you can find current platform differences in [Quotas and region availability for Azure Container Instances](#).

## gitRepo volume

The *gitRepo* volume mounts a directory and clones the specified Git repository into it at container startup. By using a *gitRepo* volume in your container instances, you can avoid adding the code for doing so in your applications.

When you mount a *gitRepo* volume, you can set three properties to configure the volume:

PROPERTY	REQUIRED	DESCRIPTION
<code>repository</code>	Yes	The full URL, including <code>http://</code> or <code>https://</code> , of the Git repository to be cloned.
<code>directory</code>	No	Directory into which the repository should be cloned. The path must not contain or start with " <code>..</code> ". If you specify " <code>.</code> ", the repository is cloned into the volume's directory. Otherwise, the Git repository is cloned into a subdirectory of the given name within the volume directory.
<code>revision</code>	No	The commit hash of the revision to be cloned. If unspecified, the <code>HEAD</code> revision is cloned.

## Mount gitRepo volume: Azure CLI

To mount a *gitRepo* volume when you deploy container instances with the [Azure CLI](#), supply the `--gitrepo-url` and `--gitrepo-mount-path` parameters to the [az container create](#) command. You can optionally specify the directory within the volume to clone into (`--gitrepo-dir`) and the commit hash of the revision to be cloned (`--gitrepo-revision`).

This example command clones the Microsoft [aci-helloworld](#) sample application into `/mnt/aci-helloworld` in the container instance:

```
az container create \
  --resource-group myResourceGroup \
  --name hellogitrepo \
  --image mcr.microsoft.com/azuredocs/aci-helloworld \
  --dns-name-label aci-demo \
  --ports 80 \
  --gitrepo-url https://github.com/Azure-Samples/aci-helloworld \
  --gitrepo-mount-path /mnt/aci-helloworld
```

To verify the gitRepo volume was mounted, launch a shell in the container with [az container exec](#) and list the directory:

```
$ az container exec --resource-group myResourceGroup --name hellogitrepo --exec-command /bin/sh
/usr/src/app # ls -l /mnt/aci-helloworld/
total 16
-rw-r--r-- 1 root root 144 Apr 16 16:35 Dockerfile
-rw-r--r-- 1 root root 1162 Apr 16 16:35 LICENSE
-rw-r--r-- 1 root root 1237 Apr 16 16:35 README.md
drwxr-xr-x 2 root root 4096 Apr 16 16:35 app
```

## Mount gitRepo volume: Resource Manager

To mount a gitRepo volume when you deploy container instances with an [Azure Resource Manager template](#), first populate the `volumes` array in the container group `properties` section of the template. Then, for each container in the container group in which you'd like to mount the *gitRepo* volume, populate the `volumeMounts` array in the `properties` section of the container definition.

For example, the following Resource Manager template creates a container group consisting of a single container. The container clones two GitHub repositories specified by the *gitRepo* volume blocks. The second volume includes additional properties specifying a directory to clone to, and the commit hash of a specific revision to clone.

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "variables": {
    "container1name": "aci-tutorial-app",
    "container1image": "microsoft/aci-helloworld:latest"
  },
  "resources": [
    {
      "name": "volume-demo-gitrepo",
      "type": "Microsoft.ContainerInstance/containerGroups",
      "apiVersion": "2018-02-01-preview",
      "location": "[resourceGroup().location]",
      "properties": {
        "containers": [
          {
            "name": "[variables('container1name')]",
            "properties": {
              "image": "[variables('container1image')]",
              "resources": {
                "requests": {
                  "cpu": 1,
                  "memoryInGb": 1.5
                }
              },
              "ports": [
                {
                  "port": 80
                }
              ]
            }
          }
        ]
      }
    }
  ]
}
```



```

        "volumeMounts": [
            {
                "name": "gitrepo1",
                "mountPath": "/mnt/repo1"
            },
            {
                "name": "gitrepo2",
                "mountPath": "/mnt/repo2"
            }
        ]
    },
    "osType": "Linux",
    "ipAddress": {
        "type": "Public",
        "ports": [
            {
                "protocol": "tcp",
                "port": "80"
            }
        ]
    },
    "volumes": [
        {
            "name": "gitrepo1",
            "gitRepo": {
                "repository": "https://github.com/Azure-Samples/aci-helloworld"
            }
        },
        {
            "name": "gitrepo2",
            "gitRepo": {
                "directory": "my-custom-clone-directory",
                "repository": "https://github.com/Azure-Samples/aci-helloworld",
                "revision": "d5ccfcedc0d81f7ca5e3dbe6e5a7705b579101f1"
            }
        }
    ]
}

```

The resulting directory structure of the two cloned repos defined in the preceding template is:

```

/mnt/repo1/aci-helloworld
/mnt/repo2/my-custom-clone-directory

```

To see an example of container instance deployment with an Azure Resource Manager template, see [Deploy multi-container groups in Azure Container Instances](#).

## Private Git repo authentication

To mount a gitRepo volume for a private Git repository, specify credentials in the repository URL. Typically, credentials are in the form of a user name and a personal access token (PAT) that grants scoped access to the repository.

For example, the Azure CLI `--gitrepo-url` parameter for a private GitHub repository would appear similar to the following (where "gituser" is the GitHub user name, and "abcdef1234fdsa4321abcdef" is the user's personal access token):

```
--gitrepo-url https://gituser:abcdef1234fdsa4321abcdef@github.com/GitUser/some-private-repository
```

For an Azure Repos Git repository, specify any user name (you can use "azurereposuser" as in the following example) in combination with a valid PAT:

```
--gitrepo-url https://azurereposuser:abcdef1234fdsa4321abcdef@dev.azure.com/your-org/_git/some-private-repository
```

For more information about personal access tokens for GitHub and Azure Repos, see the following:

GitHub: [Creating a personal access token for the command line](#)

Azure Repos: [Create personal access tokens to authenticate access](#)

## Next steps

Learn how to mount other volume types in Azure Container Instances:

- [Mount an Azure file share in Azure Container Instances](#)
- [Mount an emptyDir volume in Azure Container Instances](#)
- [Mount a secret volume in Azure Container Instances](#)

# Configure liveness probes

3/12/2019 • 2 minutes to read • [Edit Online](#)

Containerized applications may run for extended periods of time resulting in broken states that may need to be repaired by restarting the container. Azure Container Instances supports liveness probes to include configurations so that your container can restart if critical functionality is not working.

This article explains how to deploy a container group that includes a liveness probe, demonstrating the automatic restart of a simulated unhealthy container.

## YAML deployment

Create a `liveness-probe.yaml` file with the following snippet. This file defines a container group that consists of an NGNIX container that eventually becomes unhealthy.

```
apiVersion: 2018-06-01
location: eastus
name: livenessstest
properties:
  containers:
  - name: mycontainer
    properties:
      image: nginx
      command:
      - "/bin/sh"
      - "-c"
      - "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"
    ports: []
    resources:
      requests:
        cpu: 1.0
        memoryInGB: 1.5
    livenessProbe:
      exec:
        command:
        - "cat"
        - "/tmp/healthy"
      periodSeconds: 5
    osType: Linux
    restartPolicy: Always
  tags: null
type: Microsoft.ContainerInstance/containerGroups
```

Run the following command to deploy this container group with the above YAML configuration:

```
az container create --resource-group myResourceGroup --name livenessstest -f liveness-probe.yaml
```

### Start command

The deployment defines a starting command to be run when the container first starts running, defined by the `command` property which accepts an array of strings. In this example, it will start a bash session and create a file called `healthy` within the `/tmp` directory by passing this command:

```
/bin/sh -c "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"
```

It will then sleep for 30 seconds before deleting the file, then enters a 10 minute sleep.

## Liveness command

This deployment defines a `livenessProbe` which supports an `exec` liveness command that acts as the liveness check. If this command exits with a non-zero value, the container will be killed and restarted, signaling the `healthy` file could not be found. If this command exits successfully with exit code 0, no action will be taken.

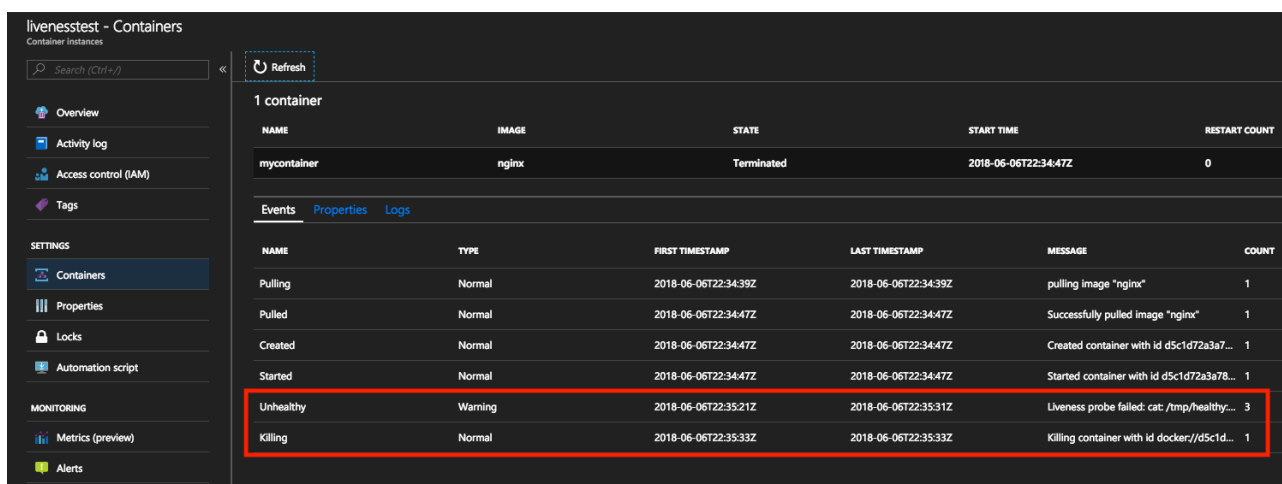
The `periodSeconds` property designates the liveness command should execute every 5 seconds.

## Verify liveness output

Within the first 30 seconds, the `healthy` file created by the start command exists. When the liveness command checks for the `healthy` file's existence, the status code returns a zero, signaling success, so no restarting occurs.

After 30 seconds, the `cat /tmp/healthy` will begin to fail, causing unhealthy and killing events to occur.

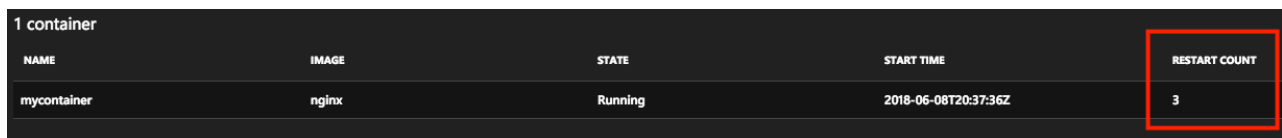
These events can be viewed from the Azure portal or Azure CLI.



liveness-test - Containers					
Container Instances					
Search (Ctrl+F)					
Refresh					
1 container					
NAME	IMAGE	STATE	START TIME	RESTART COUNT	
mycontainer	nginx	Terminated	2018-06-06T22:34:47Z	0	
Events Properties Logs					
NAME	TYPE	FIRST TIMESTAMP	LAST TIMESTAMP	MESSAGE	COUNT
Pulling	Normal	2018-06-06T22:34:39Z	2018-06-06T22:34:39Z	pulling image "nginx"	1
Pulled	Normal	2018-06-06T22:34:47Z	2018-06-06T22:34:47Z	Successfully pulled image "nginx"	1
Created	Normal	2018-06-06T22:34:47Z	2018-06-06T22:34:47Z	Created container with id d5c1d72a3a7...	1
Started	Normal	2018-06-06T22:34:47Z	2018-06-06T22:34:47Z	Started container with id d5c1d72a3a7...	1
Unhealthy	Warning	2018-06-06T22:35:21Z	2018-06-06T22:35:31Z	Liveness probe failed: cat: /tmp/healthy...	3
Killing	Normal	2018-06-06T22:35:33Z	2018-06-06T22:35:33Z	Killing container with id docker://d5c1d...	1

By viewing the events in the Azure portal, events of type `Unhealthy` will be triggered upon the liveness command failing. The subsequent event will be of type `Killing`, signifying a container deletion so a restart can begin. The restart count for the container will increment each time this occurs.

Restarts are completed in-place so resources like public IP addresses and node-specific contents will be preserved.



1 container				
NAME	IMAGE	STATE	START TIME	RESTART COUNT
mycontainer	nginx	Running	2018-06-08T20:37:36Z	3

If the liveness probe continuously fails and triggers too many restarts, your container will enter an exponential back off delay.

## Liveness probes and restart policies

Restart policies supersede the restart behavior triggered by liveness probes. For example, if you set a `restartPolicy = Never` and a liveness probe, the container group will not restart in the event of a failed liveness check. The container group will instead adhere to the container group's restart policy of `Never`.

## Next steps

Task-based scenarios may require a liveness probe to enable automatic restarts if a pre-requisite function is not working properly. For more information about running task-based containers, see [Run containerized tasks in Azure Container Instances](#).

# Manually stop or start containers in Azure Container Instances

4/15/2019 • 2 minutes to read • [Edit Online](#)

The [restart policy](#) setting of a container group determines how container instances start or stop by default. You can override the default setting by manually stopping or starting a container group.

## Stop

Manually stop a running container group - for example, by using the [az container stop](#) command or Azure portal. For certain container workloads, you might want to stop a long-running container group after a defined period to save on costs.

*When a container group enters the Stopped state, it terminates and recycles all the containers in the group. It does not preserve container state.*

Although the containers in a Stopped container group are recycled, the [resources](#) remain allocated for your use. Therefore, billing continues for a Stopped container group.

The stop action has no effect if the container group already terminated (is in either a Succeeded or Failed state). For example, a container group with run-once container tasks that ran successfully terminates in the Succeeded state. Attempts to stop the group in that state do not change the state.

## Start

When a container group is stopped - either because the containers terminated on their own or you manually stopped the group - you can start the containers. For example, use the [az container start](#) command or Azure portal to manually start the containers in the group. If the container image for any container is updated, a new image is pulled.

Starting a container group begins a new deployment with the same container configuration. This action can help you quickly reuse a known container group configuration that works as you expect. You don't have to create a new container group to run the same workload.

All containers in a container group are started by this action. You can't start a specific container in the group.

After you manually start or restart a container group, the container group runs according to the configured restart policy.

## Restart

You can restart a container group while it is running - for example, by using the [az container restart](#) command. This action restarts all containers in the container group. If the container image for any container is updated, a new image is pulled.

Restarting a container group is helpful when you want to troubleshoot a deployment problem. For example, if a temporary resource limitation prevents your containers from running successfully, restarting the group might solve the problem.

All containers in a container group are restarted by this action. You can't restart a specific container in the group.

After you manually restart a container group, the container group runs according to the configured restart policy.

## Next steps

Learn more about [restart policy settings](#) in Azure Container Instances.

In addition to manually stopping and starting a container group with the existing configuration, you can [update the settings](#) of a running container group.

# Update containers in Azure Container Instances

10/8/2018 • 3 minutes to read • [Edit Online](#)

During normal operation of your container instances, you may find it necessary to update the containers in a container group. For example, you might wish to update the image version, change a DNS name, update environment variables, or refresh the state of a container whose application has crashed.

## Update a container group

Update the containers in a container group by redeploying an existing group with at least one modified property. When you update a container group, all running containers in the group are restarted in-place.

Redeploy an existing container group by issuing the create command (or use the Azure portal) and specify the name of an existing group. Modify at least one valid property of the group when you issue the create command to trigger the redeployment. Not all container group properties are valid for redeployment. See [Properties that require delete](#) for a list of unsupported properties.

The following Azure CLI example updates a container group with a new DNS name label. Because the DNS name label property of the group is modified, the container group is redeployed, and its containers restarted.

Initial deployment with DNS name label *myapplication-staging*:

```
# Create container group
az container create --resource-group myResourceGroup --name mycontainer \
  --image nginx:alpine --dns-name-label myapplication-staging
```

Update the container group with a new DNS name label, *myapplication*:

```
# Update container group (restarts container)
az container create --resource-group myResourceGroup --name mycontainer \
  --image nginx:alpine --dns-name-label myapplication
```

## Update benefits

The primary benefit of updating an existing container group is faster deployment. When you redeploy an existing container group, its container image layers are pulled from those cached by the previous deployment. Instead of pulling all image layers fresh from the registry as is done with new deployments, only modified layers (if any) are pulled.

Applications based on larger container images like Windows Server Core can see significant improvement in deployment speed when you update instead of delete and deploy new.

## Limitations

Not all properties of a container group support updates. To change some properties of a container group, you must first delete, then redeploy the group. For details, see [Properties that require container delete](#).

All containers in a container group are restarted when you update the container group. You can't perform an update or in-place restart of a specific container in a multi-container group.

The IP address of a container won't typically change between updates, but it's not guaranteed to remain the same.

As long as the container group is deployed to the same underlying host, the container group retains its IP address. Although rare, and while Azure Container Instances makes every effort to redeploy to the same host, there are some Azure-internal events that can cause redeployment to a different host. To mitigate this issue, always use a DNS name label for your container instances.

Terminated or deleted container groups can't be updated. Once a container group has stopped (is in the *Terminated* state) or has been deleted, the group is deployed as new.

## Properties that require container delete

As mentioned earlier, not all container group properties can be updated. For example, to change the ports or restart policy of a container, you must first delete the container group, then create it again.

These properties require container group deletion prior to redeployment:

- OS type
- CPU
- Memory
- Restart policy
- Ports

When you delete a container group and recreate it, it's not "redployed," but created new. All image layers are pulled fresh from the registry, not from those cached by a previous deployment. The IP address of the container might also change due to being deployed to a different underlying host.

## Next steps

Mentioned several times in this article is the **container group**. Every container in Azure Container Instances is deployed in a container group, and container groups can contain more than one container.

[Container groups in Azure Container Instances](#)

[Deploy a multi-container group](#)



# Monitor container resources in Azure Container Instances

4/26/2019 • 3 minutes to read • [Edit Online](#)

[Azure Monitor](#) provides insight into the compute resources used by your containers instances. This resource usage data helps you determine the best resource settings for your container groups. Azure Monitor also provides metrics that track network activity in your container instances.

This document details gathering Azure Monitor metrics for container instances using both the Azure portal and Azure CLI.

## IMPORTANT

Azure Monitor metrics in Azure Container Instances are currently in preview, and some [limitations apply](#). Previews are made available to you on the condition that you agree to the [supplemental terms of use](#). Some aspects of this feature may change prior to general availability (GA).

## Preview limitations

At this time, Azure Monitor metrics are only available for Linux containers.

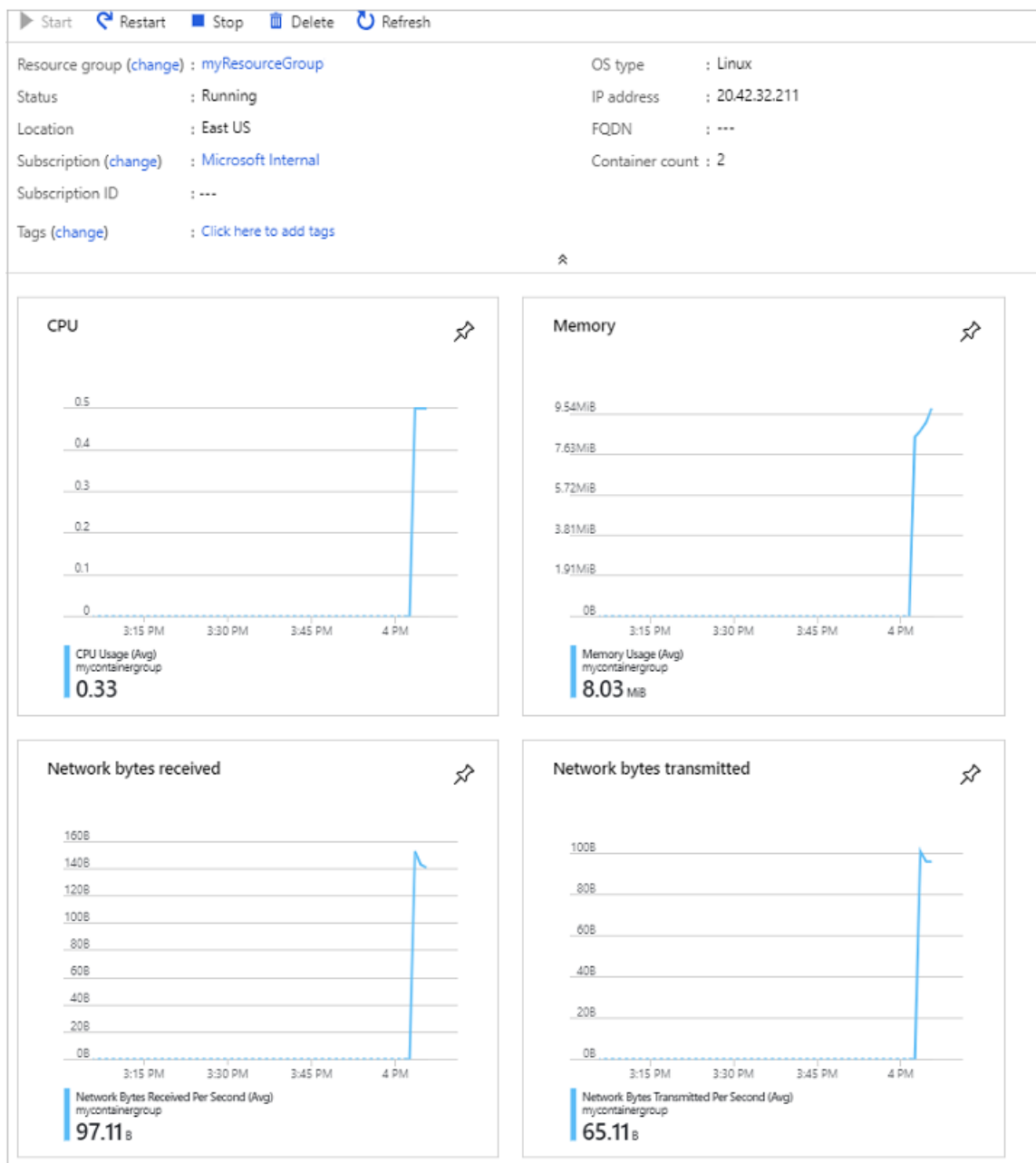
## Available metrics

Azure Monitor provides the following [metrics for Azure Container Instances](#). These metrics are available for a container group and individual containers.

- **CPU Usage** - measured in **millicores**. One millicore is 1/1000th of a CPU core, so 500 millicores (or 500 m) represents 50% usage of a CPU core. Aggregated as **average usage** across all cores.
- **Memory Usage** - aggregated as **average bytes**.
- **Network Bytes Received Per Second** and **Network Bytes Transmitted Per Second** - aggregated as **average bytes per second**.

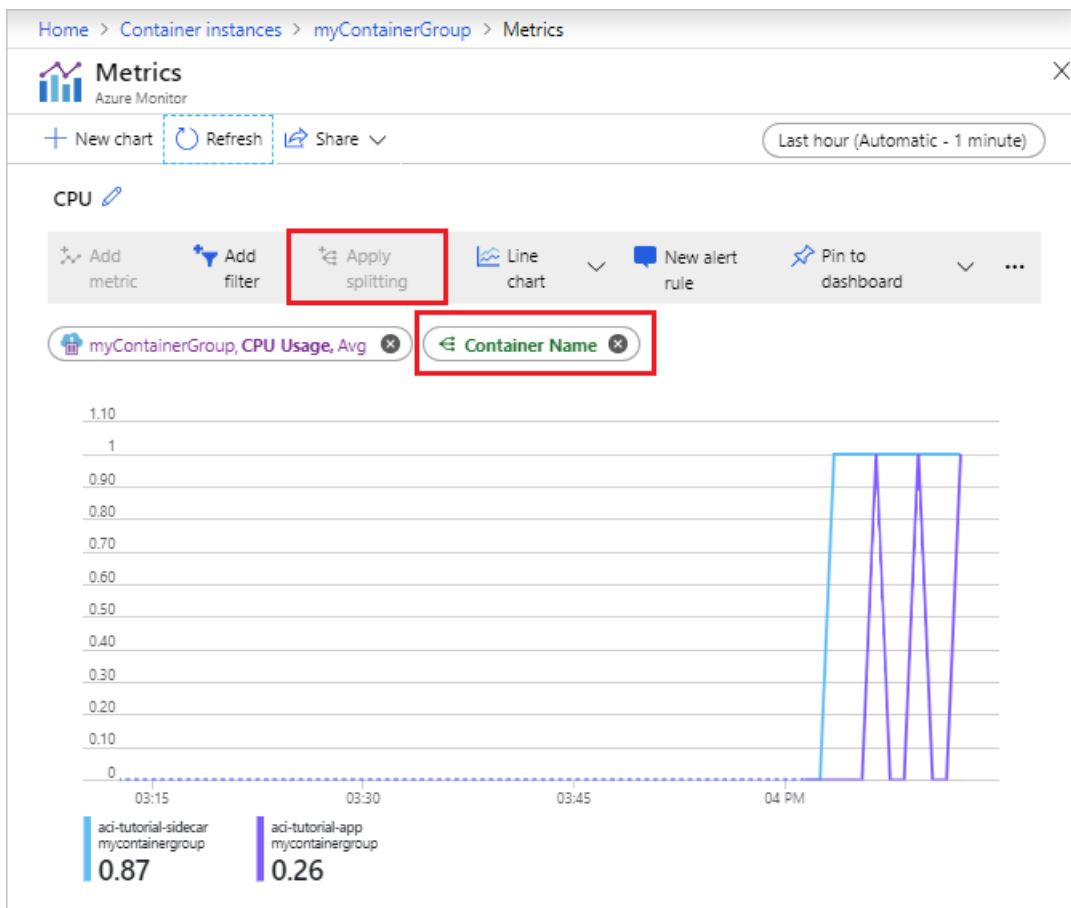
## Get metrics - Azure portal

When a container group is created, Azure Monitor data is available in the Azure portal. To see metrics for a container group, go to the **Overview** page for the container group. Here you can see pre-created charts for each of the available metrics.



In a container group that contains multiple containers, use a [dimension](#) to present metrics by container. To create a chart with individual container metrics, perform the following steps:

1. In the **Overview** page, select one of the metric charts, such as **CPU**.
2. Select the **Apply splitting** button, and select **Container Name**.



## Get metrics - Azure CLI

Metrics for container instances can also be gathered using the Azure CLI. First, get the ID of the container group using the following command. Replace `<resource-group>` with your resource group name and `<container-group>` with the name of your container group.

```
CONTAINER_GROUP=$(az container show --resource-group <resource-group> --name <container-group> --query id --output tsv)
```

Use the following command to get **CPU** usage metrics.

```
$ az monitor metrics list --resource $CONTAINER_GROUP --metric CPUUsage --output table
```

Timestamp	Name	Average
2019-04-23 22:59:00	CPU Usage	
2019-04-23 23:00:00	CPU Usage	
2019-04-23 23:01:00	CPU Usage	0.0
2019-04-23 23:02:00	CPU Usage	0.0
2019-04-23 23:03:00	CPU Usage	0.5
2019-04-23 23:04:00	CPU Usage	0.5
2019-04-23 23:05:00	CPU Usage	0.5
2019-04-23 23:06:00	CPU Usage	1.0
2019-04-23 23:07:00	CPU Usage	0.5
2019-04-23 23:08:00	CPU Usage	0.5
2019-04-23 23:09:00	CPU Usage	1.0
2019-04-23 23:10:00	CPU Usage	0.5

Change the value of the `--metric` parameter in the command to get other [supported metrics](#). For example, use the following command to get **memory** usage metrics.

```
$ az monitor metrics list --resource $CONTAINER_GROUP --metric MemoryUsage --output table
```

Timestamp	Name	Average
2019-04-23 22:59:00	Memory Usage	
2019-04-23 23:00:00	Memory Usage	
2019-04-23 23:01:00	Memory Usage	0.0
2019-04-23 23:02:00	Memory Usage	8859648.0
2019-04-23 23:03:00	Memory Usage	9181184.0
2019-04-23 23:04:00	Memory Usage	9580544.0
2019-04-23 23:05:00	Memory Usage	10280960.0
2019-04-23 23:06:00	Memory Usage	7815168.0
2019-04-23 23:07:00	Memory Usage	7739392.0
2019-04-23 23:08:00	Memory Usage	8212480.0
2019-04-23 23:09:00	Memory Usage	8159232.0
2019-04-23 23:10:00	Memory Usage	8093696.0

For a multi-container group, the `containerName` dimension can be added to return metrics per container.

```
$ az monitor metrics list --resource $CONTAINER_GROUP --metric MemoryUsage --dimension containerName --output table
```

Timestamp	Name	Containername	Average
2019-04-23 22:59:00	Memory Usage	aci-tutorial-app	
2019-04-23 23:00:00	Memory Usage	aci-tutorial-app	
2019-04-23 23:01:00	Memory Usage	aci-tutorial-app	0.0
2019-04-23 23:02:00	Memory Usage	aci-tutorial-app	16834560.0
2019-04-23 23:03:00	Memory Usage	aci-tutorial-app	17534976.0
2019-04-23 23:04:00	Memory Usage	aci-tutorial-app	18329600.0
2019-04-23 23:05:00	Memory Usage	aci-tutorial-app	19742720.0
2019-04-23 23:06:00	Memory Usage	aci-tutorial-app	14786560.0
2019-04-23 23:07:00	Memory Usage	aci-tutorial-app	14651392.0
2019-04-23 23:08:00	Memory Usage	aci-tutorial-app	15470592.0
2019-04-23 23:09:00	Memory Usage	aci-tutorial-app	15450112.0
2019-04-23 23:10:00	Memory Usage	aci-tutorial-app	15339520.0
2019-04-23 22:59:00	Memory Usage	aci-tutorial-sidecar	
2019-04-23 23:00:00	Memory Usage	aci-tutorial-sidecar	
2019-04-23 23:01:00	Memory Usage	aci-tutorial-sidecar	0.0
2019-04-23 23:02:00	Memory Usage	aci-tutorial-sidecar	884736.0
2019-04-23 23:03:00	Memory Usage	aci-tutorial-sidecar	827392.0
2019-04-23 23:04:00	Memory Usage	aci-tutorial-sidecar	831488.0
2019-04-23 23:05:00	Memory Usage	aci-tutorial-sidecar	819200.0
2019-04-23 23:06:00	Memory Usage	aci-tutorial-sidecar	843776.0
2019-04-23 23:07:00	Memory Usage	aci-tutorial-sidecar	827392.0
2019-04-23 23:08:00	Memory Usage	aci-tutorial-sidecar	954368.0
2019-04-23 23:09:00	Memory Usage	aci-tutorial-sidecar	868352.0
2019-04-23 23:10:00	Memory Usage	aci-tutorial-sidecar	847872.0

## Next steps

Learn more about Azure Monitoring at the [Azure Monitoring overview](#).

Learn how to create [metric alerts](#) to get notified when a metric for Azure Container Instances crosses a threshold.

# Retrieve container logs and events in Azure Container Instances

3/22/2019 • 2 minutes to read • [Edit Online](#)

When you have a misbehaving container, start by viewing its logs with [az container logs](#), and streaming its standard out and standard error with [az container attach](#).

## View logs

To view logs from your application code within a container, you can use the [az container logs](#) command.

The following is log output from the example task-based container in [Run a containerized task in ACI](#), after having fed it an invalid URL to process:

```
$ az container logs --resource-group myResourceGroup --name mycontainer
Traceback (most recent call last):
  File "wordcount.py", line 11, in <module>
    urllib.request.urlretrieve (sys.argv[1], "foo.txt")
  File "/usr/local/lib/python3.6/urllib/request.py", line 248, in urlretrieve
    with contextlib.closing(urlopen(url, data)) as fp:
  File "/usr/local/lib/python3.6/urllib/request.py", line 223, in urlopen
    return opener.open(url, data, timeout)
  File "/usr/local/lib/python3.6/urllib/request.py", line 532, in open
    response = meth(req, response)
  File "/usr/local/lib/python3.6/urllib/request.py", line 642, in http_response
    'http', request, response, code, msg, hdrs)
  File "/usr/local/lib/python3.6/urllib/request.py", line 570, in error
    return self._call_chain(*args)
  File "/usr/local/lib/python3.6/urllib/request.py", line 504, in _call_chain
    result = func(*args)
  File "/usr/local/lib/python3.6/urllib/request.py", line 650, in http_error_default
    raise HTTPError(req.full_url, code, msg, hdrs, fp)
urllib.error.HTTPError: HTTP Error 404: Not Found
```

## Attach output streams

The [az container attach](#) command provides diagnostic information during container startup. Once the container has started, it streams STDOUT and STDERR to your local console.

For example, here is output from the task-based container in [Run a containerized task in ACI](#), after having supplied a valid URL of a large text file to process:

```
$ az container attach --resource-group myResourceGroup --name mycontainer
Container 'mycontainer' is in state 'Unknown'...
Container 'mycontainer' is in state 'Waiting'...
Container 'mycontainer' is in state 'Running'...
(count: 1) (last timestamp: 2019-03-21 19:42:39+00:00) pulling image "mcr.microsoft.com/azuredocs/aci-wordcount:latest"
Container 'mycontainer1' is in state 'Running'...
(count: 1) (last timestamp: 2019-03-21 19:42:39+00:00) pulling image "mcr.microsoft.com/azuredocs/aci-wordcount:latest"
(count: 1) (last timestamp: 2019-03-21 19:42:52+00:00) Successfully pulled image
"mcr.microsoft.com/azuredocs/aci-wordcount:latest"
(count: 1) (last timestamp: 2019-03-21 19:42:55+00:00) Created container
(count: 1) (last timestamp: 2019-03-21 19:42:55+00:00) Started container

Start streaming logs:
[('the', 22979),
 ('I', 20003),
 ('and', 18373),
 ('to', 15651),
 ('of', 15558),
 ('a', 12500),
 ('you', 11818),
 ('my', 10651),
 ('in', 9707),
 ('is', 8195)]
```

## Get diagnostic events

If your container fails to deploy successfully, you need to review the diagnostic information provided by the Azure Container Instances resource provider. To view the events for your container, run the `[az container show][az-container-show]` command:

```
az container show --resource-group myResourceGroup --name mycontainer
```

The output includes the core properties of your container, along with deployment events (shown here truncated):

```

{
  "containers": [
    {
      "command": null,
      "environmentVariables": [],
      "image": "mcr.microsoft.com/azuredocs/aci-helloworld",
      ...
      "events": [
        {
          "count": 1,
          "firstTimestamp": "2019-03-21T19:46:22+00:00",
          "lastTimestamp": "2019-03-21T19:46:22+00:00",
          "message": "pulling image \"mcr.microsoft.com/azuredocs/aci-helloworld\"",
          "name": "Pulling",
          "type": "Normal"
        },
        {
          "count": 1,
          "firstTimestamp": "2019-03-21T19:46:28+00:00",
          "lastTimestamp": "2019-03-21T19:46:28+00:00",
          "message": "Successfully pulled image \"mcr.microsoft.com/azuredocs/aci-helloworld\"",
          "name": "Pulled",
          "type": "Normal"
        },
        {
          "count": 1,
          "firstTimestamp": "2019-03-21T19:46:31+00:00",
          "lastTimestamp": "2019-03-21T19:46:31+00:00",
          "message": "Created container",
          "name": "Created",
          "type": "Normal"
        },
        {
          "count": 1,
          "firstTimestamp": "2019-03-21T19:46:31+00:00",
          "lastTimestamp": "2019-03-21T19:46:31+00:00",
          "message": "Started container",
          "name": "Started",
          "type": "Normal"
        }
      ],
      "previousState": null,
      "restartCount": 0
    },
    {
      "name": "mycontainer",
      "ports": [
        {
          "port": 80,
          "protocol": null
        }
      ],
      ...
    }
  ],
  ...
}

```

## Next steps

Learn how to [troubleshoot common container and deployment issues](#) for Azure Container Instances.

# Container instance logging with Azure Monitor logs

2/26/2019 • 4 minutes to read • [Edit Online](#)

Log Analytics workspaces provide a centralized location for storing and querying log data from not only Azure resources, but also on premises resources and resources in other clouds. Azure Container Instances includes built-in support for sending data to Azure Monitor logs.

To send container instance data to Azure Monitor logs, you must create a container group by using the Azure CLI (or Cloud Shell) and a YAML file. The following sections describe creating a logging-enabled container group and querying logs.

## NOTE

This article was recently updated to use the term Azure Monitor logs instead of Log Analytics. Log data is still stored in a Log Analytics workspace and is still collected and analyzed by the same Log Analytics service. We are updating the terminology to better reflect the role of [logs in Azure Monitor](#). See [Azure Monitor terminology changes](#) for details.

## Prerequisites

To enable logging in your container instances, you need the following:

- [Log Analytics workspace](#)
- [Azure CLI](#) (or [Cloud Shell](#))

## Get Log Analytics credentials

Azure Container Instances needs permission to send data to your Log Analytics workspace. To grant this permission and enable logging, you must provide the Log Analytics workspace ID and one of its keys (either primary or secondary) when you create the container group.

To obtain the log analytics workspace ID and primary key:

1. Navigate to your Log Analytics workspace in the Azure portal
2. Under **SETTINGS**, select **Advanced settings**
3. Select **Connected Sources** > **Windows Servers** (or **Linux Servers**--the ID and keys are the same for both)
4. Take note of:
  - **WORKSPACE ID**
  - **PRIMARY KEY**

## Create container group

Now that you have the log analytics workspace ID and primary key, you're ready to create a logging-enabled container group.

The following examples demonstrate two ways to create a container group with a single [fluentd](#) container: Azure CLI, and Azure CLI with a YAML template. The Fluentd container produces several lines of output in its default configuration. Because this output is sent to your Log Analytics workspace, it works well for demonstrating the viewing and querying of logs.

### Deploy with Azure CLI



To deploy with the Azure CLI, specify the `--log-analytics-workspace` and `--log-analytics-workspace-key` parameters in the [az container create](#) command. Replace the two workspace values with the values you obtained in the previous step (and update the resource group name) before running the following command.

```
az container create \
  --resource-group myResourceGroup \
  --name mycontainergroup001 \
  --image fluent/fluentd \
  --log-analytics-workspace <WORKSPACE_ID> \
  --log-analytics-workspace-key <WORKSPACE_KEY>
```

## Deploy with YAML

Use this method if you prefer to deploy container groups with YAML. The following YAML defines a container group with a single container. Copy the YAML into a new file, then replace `LOG_ANALYTICS_WORKSPACE_ID` and `LOG_ANALYTICS_WORKSPACE_KEY` with the values you obtained in the previous step. Save the file as **deploy-aci.yaml**.

```
apiVersion: 2018-06-01
location: eastus
name: mycontainergroup001
properties:
  containers:
  - name: mycontainer001
    properties:
      environmentVariables: []
      image: fluent/fluentd
      ports: []
      resources:
        requests:
          cpu: 1.0
          memoryInGB: 1.5
    osType: Linux
    restartPolicy: Always
    diagnostics:
      logAnalytics:
        workspaceId: LOG_ANALYTICS_WORKSPACE_ID
        workspaceKey: LOG_ANALYTICS_WORKSPACE_KEY
  tags: null
type: Microsoft.ContainerInstance/containerGroups
```

Next, execute the following command to deploy the container group; replace `myResourceGroup` with a resource group in your subscription (or first create a resource group named "myResourceGroup"):

```
az container create --resource-group myResourceGroup --name mycontainergroup001 --file deploy-aci.yaml
```

You should receive a response from Azure containing deployment details shortly after issuing the command.

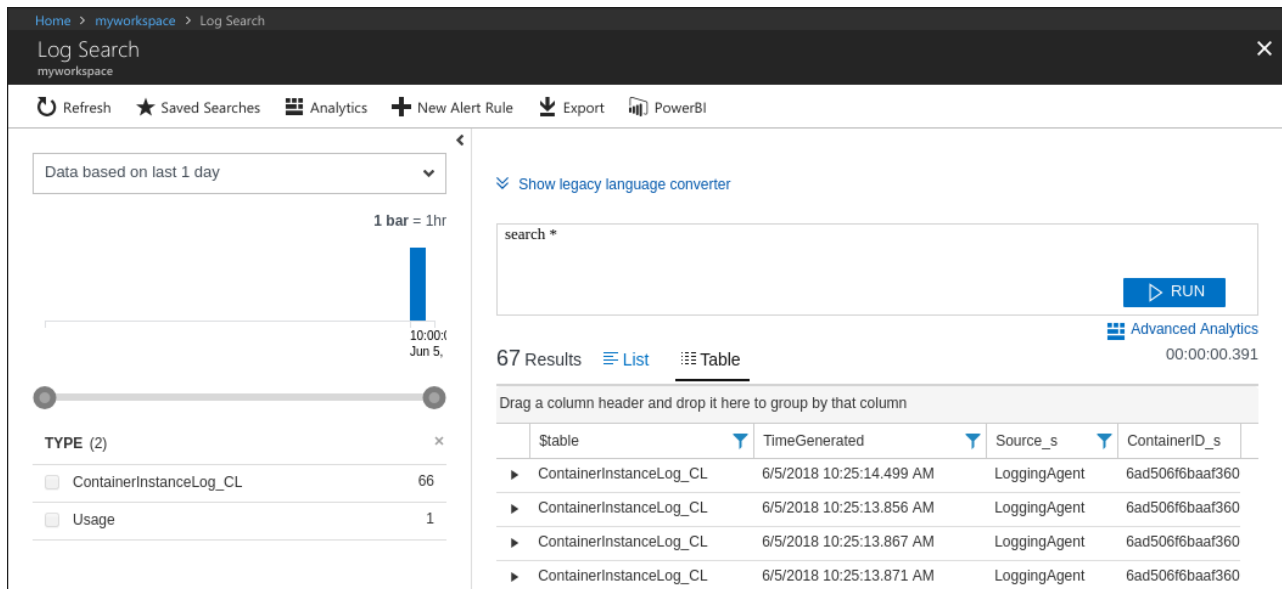
## View logs in Azure Monitor logs

After you've deployed the container group, it can take several minutes (up to 10) for the first log entries to appear in the Azure portal. To view the container group's logs, open your Log Analytics workspace, then:

1. In the **OMS workspace** overview, select **Log Search**. OMS workspaces are now referred to as Log Analytics workspaces.
2. Under **A few more queries to try**, select the **All collected data** link

You should see several results displayed by the `search *` query. If at first you don't see any results, wait a few minutes, then select the **RUN** button to execute the query again. By default, log entries are displayed in "List" view-

-select **Table** to see the log entries in a more condensed format. You can then expand a row to see the contents of an individual log entry.



## Query container logs

Azure Monitor logs includes an extensive [query language](#) for pulling information from potentially thousands of lines of log output.

The Azure Container Instances logging agent sends entries to the `ContainerInstanceLog_CL` table in your Log Analytics workspace. The basic structure of a query is the source table ( `ContainerInstanceLog_CL` ) followed by a series of operators separated by the pipe character ( `|` ). You can chain several operators to refine the results and perform advanced functions.

To see example query results, paste the following query into the query text box (under "Show legacy language converter"), and select the **RUN** button to execute the query. This query displays all log entries whose "Message" field contains the word "warn":

```
ContainerInstanceLog_CL
| where Message contains("warn")
```

More complex queries are also supported. For example, this query displays only those log entries for the "mycontainergroup001" container group generated within the last hour:

```
ContainerInstanceLog_CL
| where (ContainerGroup_s == "mycontainergroup001")
| where (TimeGenerated > ago(1h))
```

## Next steps

### Azure Monitor logs

For more information about querying logs and configuring alerts in Azure Monitor logs, see:

- [Understanding log searches in Azure Monitor logs](#)
- [Unified alerts in Azure Monitor](#)

### Monitor container CPU and memory

For information about monitoring container instance CPU and memory resources, see:

- [Monitor container resources in Azure Container Instances.](#)

# Troubleshoot common issues in Azure Container Instances

3/22/2019 • 7 minutes to read • [Edit Online](#)

This article shows how to troubleshoot common issues for managing or deploying containers to Azure Container Instances.

## Naming conventions

When defining your container specification, certain parameters require adherence to naming restrictions. Below is a table with specific requirements for container group properties. For more information on Azure naming conventions, see [Naming conventions](#) in the Azure Architecture Center.

SCOPE	LENGTH	CASING	VALID CHARACTERS	SUGGESTED PATTERN	EXAMPLE
Container group name	1-64	Case insensitive	Alphanumeric, and hyphen anywhere except the first or last character	<code>&lt;name&gt;-&lt;role&gt;-CG&lt;number&gt;</code>	<code>web-batch-CG1</code>
Container name	1-64	Case insensitive	Alphanumeric, and hyphen anywhere except the first or last character	<code>&lt;name&gt;-&lt;role&gt;-CG&lt;number&gt;</code>	<code>web-batch-CG1</code>
Container ports	Between 1 and 65535	Integer	Integer between 1 and 65535	<code>&lt;port-number&gt;</code>	<code>443</code>
DNS name label	5-63	Case insensitive	Alphanumeric, and hyphen anywhere except the first or last character	<code>&lt;name&gt;</code>	<code>frontend-site1</code>
Environment variable	1-63	Case insensitive	Alphanumeric, and underscore ( ) anywhere except the first or last character	<code>&lt;name&gt;</code>	<code>MY_VARIABLE</code>
Volume name	5-63	Case insensitive	Lowercase letters and numbers, and hyphens anywhere except the first or last character. Cannot contain two consecutive hyphens.	<code>&lt;name&gt;</code>	<code>batch-output-volume</code>

# OS version of image not supported

If you specify an image that Azure Container Instances doesn't support, an `OsVersionNotSupported` error is returned. The error is similar to following, where `{0}` is the name of the image you attempted to deploy:

```
{
  "error": {
    "code": "OsVersionNotSupported",
    "message": "The OS version of image '{0}' is not supported."
  }
}
```

This error is most often encountered when deploying Windows images that are based on a Semi-Annual Channel (SAC) release. For example, Windows versions 1709 and 1803 are SAC releases, and generate this error upon deployment.

Azure Container Instances currently supports Windows images based only on the **Windows Server 2016 Long-Term Servicing Channel (LTSC)** release. To mitigate this issue when deploying Windows containers, always deploy Windows Server 2016 (LTSC)-based images. Images based on Windows Server 2019 (LTSC) are not supported.

For details about the LTSC and SAC versions of Windows, see [Windows Server Semi-Annual Channel overview](#).

## Unable to pull image

If Azure Container Instances is initially unable to pull your image, it retries for a period of time. If the image pull operation continues to fail, ACI eventually fails the deployment, and you may see a `Failed to pull image` error.

To resolve this issue, delete the container instance and retry your deployment. Ensure that the image exists in the registry, and that you've typed the image name correctly.

If the image can't be pulled, events like the following are shown in the output of `az container show`:

```
"events": [
  {
    "count": 3,
    "firstTimestamp": "2017-12-21T22:56:19+00:00",
    "lastTimestamp": "2017-12-21T22:57:00+00:00",
    "message": "pulling image \"mcr.microsoft.com/azuredocs/aci-helloworld\"",
    "name": "Pulling",
    "type": "Normal"
  },
  {
    "count": 3,
    "firstTimestamp": "2017-12-21T22:56:19+00:00",
    "lastTimestamp": "2017-12-21T22:57:00+00:00",
    "message": "Failed to pull image \"mcr.microsoft.com/azuredocs/aci-helloworld\": rpc error: code 2 desc Error: image t/aci-helloworld:latest not found",
    "name": "Failed",
    "type": "Warning"
  },
  {
    "count": 3,
    "firstTimestamp": "2017-12-21T22:56:20+00:00",
    "lastTimestamp": "2017-12-21T22:57:16+00:00",
    "message": "Back-off pulling image \"mcr.microsoft.com/azuredocs/aci-helloworld\"",
    "name": "BackOff",
    "type": "Normal"
  }
],
```

# Container continually exits and restarts (no long-running process)

Container groups default to a [restart policy](#) of **Always**, so containers in the container group always restart after they run to completion. You may need to change this to **OnFailure** or **Never** if you intend to run task-based containers. If you specify **OnFailure** and still see continual restarts, there might be an issue with the application or script executed in your container.

When running container groups without long-running processes you may see repeated exits and restarts with images such as Ubuntu or Alpine. Connecting via [EXEC](#) will not work as the container has no process keeping it alive. To resolve this problem, include a start command like the following with your container group deployment to keep the container running.

```
## Deploying a Linux container
az container create -g MyResourceGroup --name myapp --image ubuntu --command-line "tail -f /dev/null"
```

```
## Deploying a Windows container
az container create -g myResourceGroup --name mywindowsapp --os-type Windows --image
mcr.microsoft.com/windows/servercore:ltsc2016
--command-line "ping -t localhost"
```

The Container Instances API and Azure portal includes a `restartCount` property. To check the number of restarts for a container, you can use the [az container show](#) command in the Azure CLI. In the following example output (which has been truncated for brevity), you can see the `restartCount` property at the end of the output.

```

...
"events": [
  {
    "count": 1,
    "firstTimestamp": "2017-11-13T21:20:06+00:00",
    "lastTimestamp": "2017-11-13T21:20:06+00:00",
    "message": "Pulling: pulling image \"myregistry.azurecr.io/aci-tutorial-app:v1\"",
    "type": "Normal"
  },
  {
    "count": 1,
    "firstTimestamp": "2017-11-13T21:20:14+00:00",
    "lastTimestamp": "2017-11-13T21:20:14+00:00",
    "message": "Pulled: Successfully pulled image \"myregistry.azurecr.io/aci-tutorial-app:v1\"",
    "type": "Normal"
  },
  {
    "count": 1,
    "firstTimestamp": "2017-11-13T21:20:14+00:00",
    "lastTimestamp": "2017-11-13T21:20:14+00:00",
    "message": "Created: Created container with id
bf25a6ac73a925687cafcec792c9e3723b0776f683d8d1402b20cc9fb5f66a10",
    "type": "Normal"
  },
  {
    "count": 1,
    "firstTimestamp": "2017-11-13T21:20:14+00:00",
    "lastTimestamp": "2017-11-13T21:20:14+00:00",
    "message": "Started: Started container with id
bf25a6ac73a925687cafcec792c9e3723b0776f683d8d1402b20cc9fb5f66a10",
    "type": "Normal"
  }
],
"previousState": null,
"restartCount": 0
...
}

```

#### NOTE

Most container images for Linux distributions set a shell, such as bash, as the default command. Since a shell on its own is not a long-running service, these containers immediately exit and fall into a restart loop when configured with the default **Always** restart policy.

## Container takes a long time to start

The two primary factors that contribute to container startup time in Azure Container Instances are:

- [Image size](#)
- [Image location](#)

Windows images have [additional considerations](#).

### Image size

If your container takes a long time to start, but eventually succeeds, start by looking at the size of your container image. Because Azure Container Instances pulls your container image on demand, the startup time you see is directly related to its size.

You can view the size of your container image by using the `docker images` command in the Docker CLI:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
mcr.microsoft.com/azuredocs/aci-helloworld	latest	7367f3256b41	15 months ago	67.6MB

The key to keeping image sizes small is ensuring that your final image does not contain anything that is not required at runtime. One way to do this is with [multi-stage builds](#). Multi-stage builds make it easy to ensure that the final image contains only the artifacts you need for your application, and not any of the extra content that was required at build time.

### Image location

Another way to reduce the impact of the image pull on your container's startup time is to host the container image in [Azure Container Registry](#) in the same region where you intend to deploy container instances. This shortens the network path that the container image needs to travel, significantly shortening the download time.

### Cached Windows images

Azure Container Instances uses a caching mechanism to help speed container startup time for images based on common Windows and Linux images. For a detailed list of cached images and tags, use the [List Cached Images](#) API.

To ensure the fastest Windows container startup time, use one of the **three most recent** versions of the following **two images** as the base image:

- [Windows Server Core 2016](#) (LTSC only)
- [Windows Server 2016 Nano Server](#)

### Windows containers slow network readiness

On initial creation, Windows containers may have no inbound or outbound connectivity for up to 30 seconds (or longer, in rare cases). If your container application needs an Internet connection, add delay and retry logic to allow 30 seconds to establish Internet connectivity. After initial setup, container networking should resume appropriately.

## Resource not available error

Due to varying regional resource load in Azure, you might receive the following error when attempting to deploy a container instance:

```
The requested resource with 'x' CPU and 'y.z' GB memory is not available in the location 'example region' at this moment. Please retry with a different resource request or in another location.
```

This error indicates that due to heavy load in the region in which you are attempting to deploy, the resources specified for your container can't be allocated at that time. Use one or more of the following mitigation steps to help resolve your issue.

- Verify your container deployment settings fall within the parameters defined in [Region availability for Azure Container Instances](#)
- Specify lower CPU and memory settings for the container
- Deploy to a different Azure region
- Deploy at a later time

## Cannot connect to underlying Docker API or run privileged containers

Azure Container Instances does not expose direct access to the underlying infrastructure that hosts container groups. This includes access to the Docker API running on the container's host and running privileged containers. If you require Docker interaction, check the [REST reference documentation](#) to see what the ACI API supports. If there is something missing, submit a request on the [ACI feedback forums](#).



## IPs may not be accessible due to mismatched ports

Azure Container Instances does not currently support port mapping like with regular docker configuration, however this fix is on the roadmap. If you find IPs are not accessible when you believe it should be, ensure you have configured your container image to listen to the same ports you expose in your container group with the `ports` property.

## Next steps

Learn how to [retrieve container logs and events](#) to help debug your containers.