# Contents

# Azure Container Instances

10/4/2018 • 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.

## 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. While we work to bring feature parity to Windows containers, you can find current platform differences in Quotas and region availability for Azure Container Instances.

Azure Container Instances supports Windows 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**
> Deployment of container groups to a virtual network 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).

## Next steps

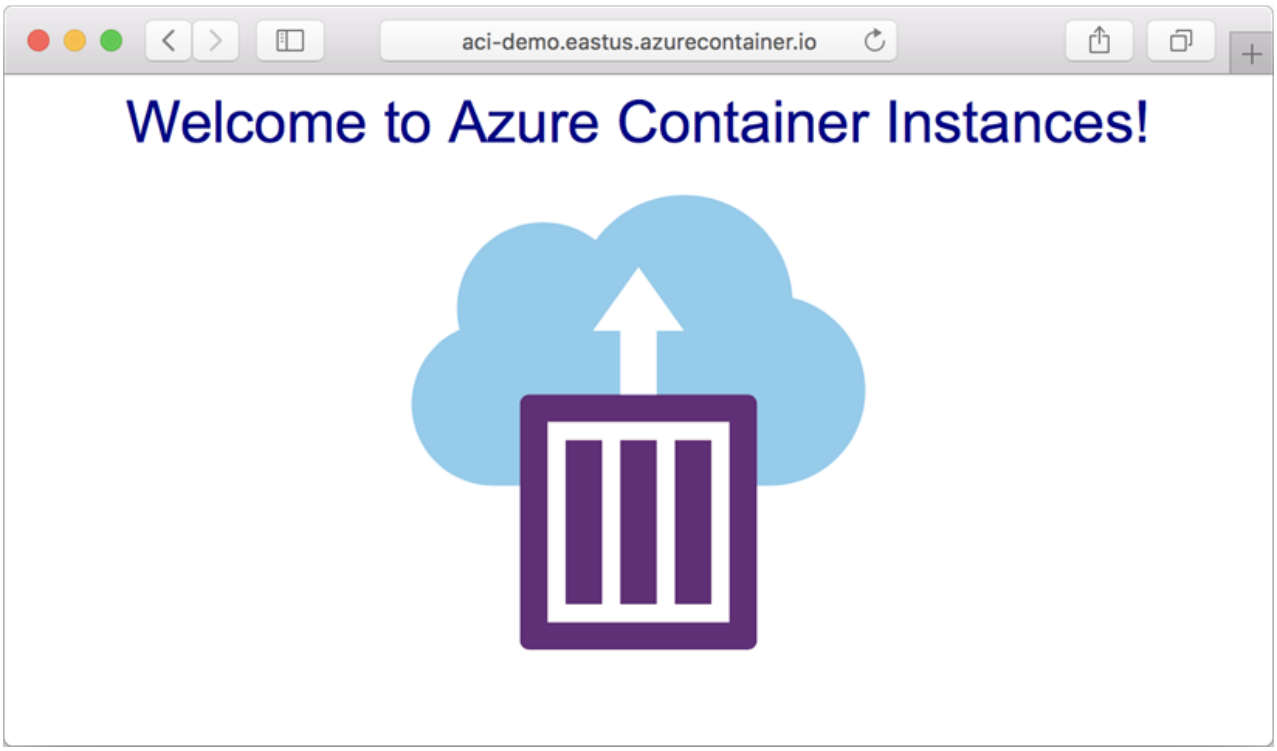
Try deploying a container to Azure with a single command using our quickstart guide:

Azure Container Instances Quickstart

# Quickstart: Run an application in Azure Container Instances

10/8/2018 • 5 minutes to read • Edit Online

Use Azure Container Instances to run Docker containers in Azure with simplicity and speed. You don't need to deploy virtual machines or use a full container orchestration platform like Kubernetes. In this quickstart, you use the Azure portal to create a container in Azure 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 running application:




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. Just select the **Copy** button 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, you can choose any one of them to open Cloud Shell:

| | |
|---|---|
| Select **Try It** in the upper-right corner of a code block. |   |
| Open Cloud Shell in your browser. |   |
| Select the **Cloud Shell** button on the menu in the upper-right corner of the Azure portal. |  |

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, you need version 2.0.27 or later. 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. 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 that hosts a small web app written in Node.js.

Execute the following command to start a container instance. The `--dns-name-label` value must be unique within the Azure region 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 microsoft/aci-helloworld --
dns-name-label aci-demo --ports 80
```

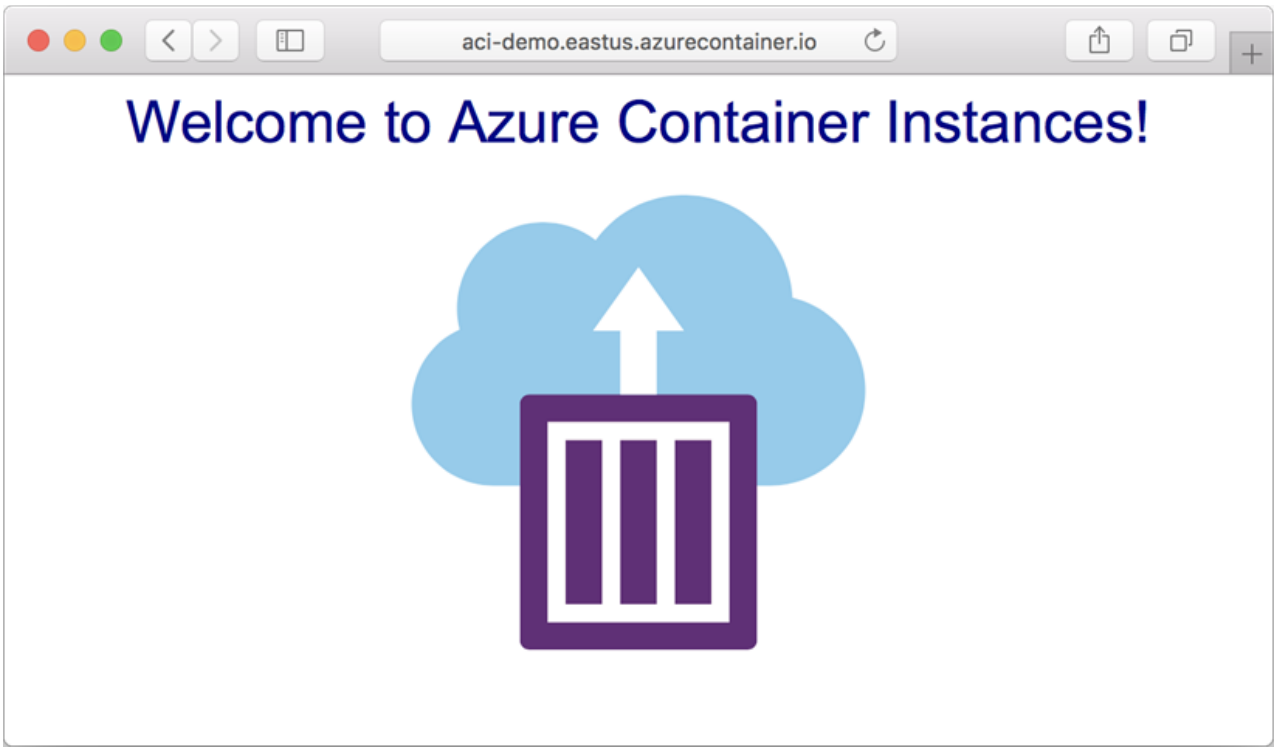
Within a few seconds, you should get a response from the Azure CLI indicating that the deployment has completed. Check its status with the az container show command:

```
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:10.240.255.105 - - [01/Oct/2018:18:25:51 +0000] "GET / HTTP/1.0" 200 1663 "-" "-"
::ffff:10.240.255.106 - - [01/Oct/2018:18:31:04 +0000] "GET / HTTP/1.1" 200 1663 "-" "Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36"
::ffff:10.240.255.106 - - [01/Oct/2018:18:31:04 +0000] "GET /favicon.ico HTTP/1.1" 404 150 "http://aci-
demo.eastus.azurecontainer.io/" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/69.0.3497.100 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 the container's output streams:

```
az container attach --resource-group myResourceGroup -n 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 -n mycontainer
Container 'mycontainer' is in state 'Running'...
(count: 1) (last timestamp: 2018-03-15 21:17:59+00:00) pulling image "microsoft/aci-helloworld"
(count: 1) (last timestamp: 2018-03-15 21:18:05+00:00) Successfully pulled image "microsoft/aci-helloworld"
(count: 1) (last timestamp: 2018-03-15 21:18:05+00:00) Created container with id
3534a1e2ee392d6f47b2c158ce8c1808d1686fc54f17de3a953d356cf5f26a45
(count: 1) (last timestamp: 2018-03-15 21:18:06+00:00) Started container with id
3534a1e2ee392d6f47b2c158ce8c1808d1686fc54f17de3a953d356cf5f26a45

Start streaming logs:
listening on port 80
::ffff:10.240.255.105 - - [15/Mar/2018:21:18:26 +0000] "GET / HTTP/1.1" 200 1663 "-" "Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.146 Safari/537.36"
::ffff:10.240.255.105 - - [15/Mar/2018:21:18:26 +0000] "GET /favicon.ico HTTP/1.1" 404 150 "http://aci-
demo.eastus.azurecontainer.io/" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/65.0.3325.146 Safari/537.36"
::ffff:10.240.255.107 - - [15/Mar/2018:21:18:44 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.146 Safari/537.36"
::ffff:10.240.255.107 - - [15/Mar/2018:21:18:47 +0000] "GET / HTTP/1.1" 304 - "-" "Mozilla/5.0 (X11; Linux
x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.146 Safari/537.36"
```

## Clean up resources

When you're done with the container, remove it using theaz 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 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

To try out options for running containers in an orchestration system on Azure, see the Service Fabric or Azure Kubernetes Service (AKS) quickstarts.

# Quickstart: Run an application in Azure Container Instances

10/8/2018 • 2 minutes to read • Edit Online

Use Azure Container Instances to run Docker containers in Azure with simplicity and speed. You don't need to deploy virtual machines or use a full container orchestration platform like Kubernetes. In this quickstart, you use the Azure portal to create a container in Azure 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:

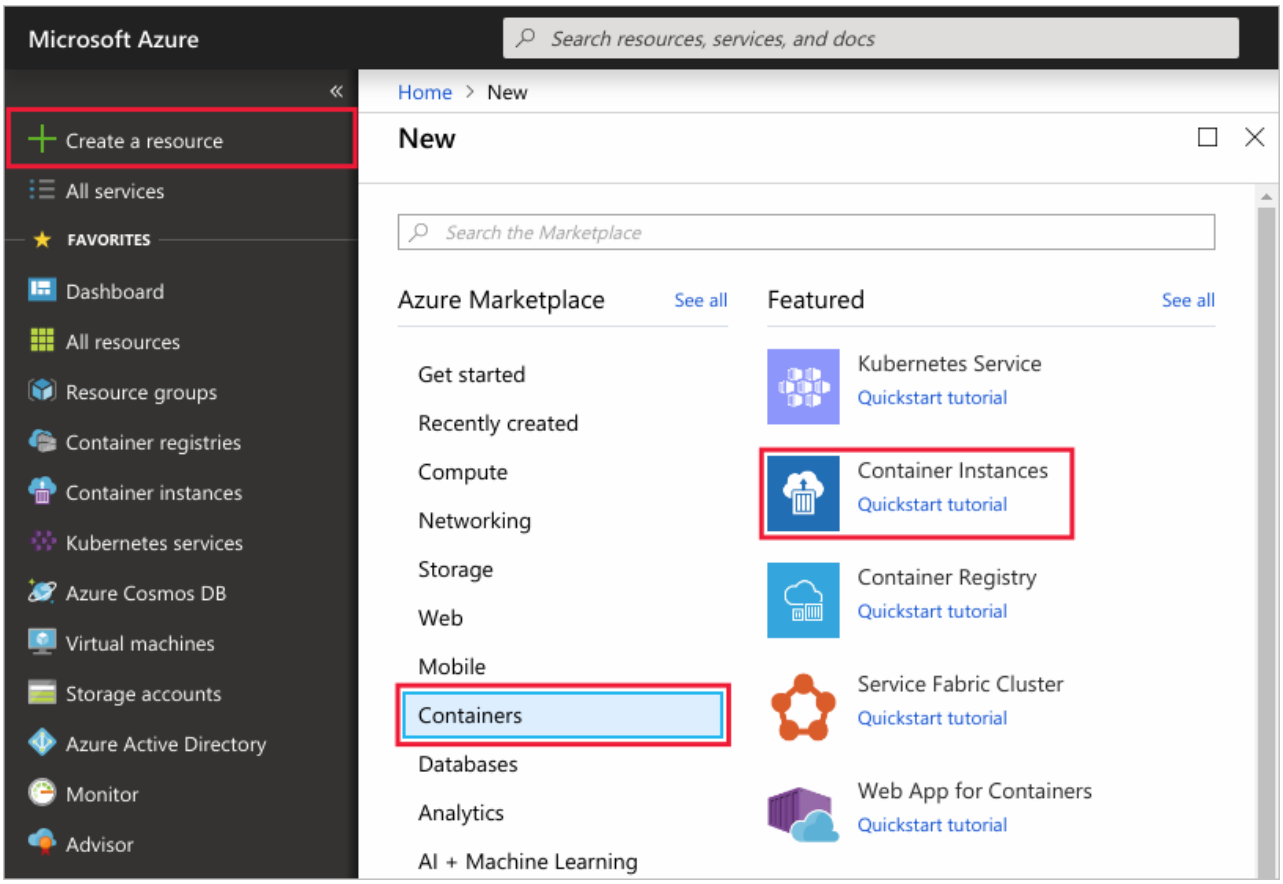

## Sign in to Azure

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.

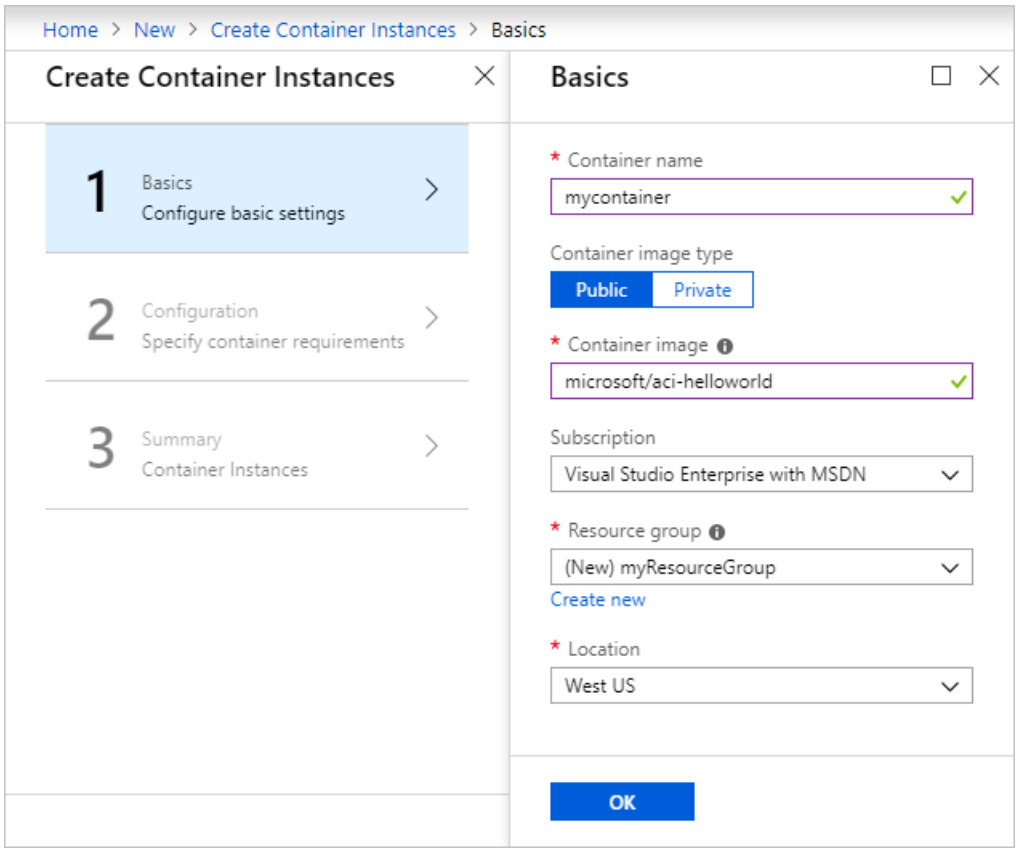
## Create a container instance

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




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

- Container name: `mycontainer`
- Container image: `microsoft/aci-helloworld`
- Resource group: **Create new** > `myResourceGroup`

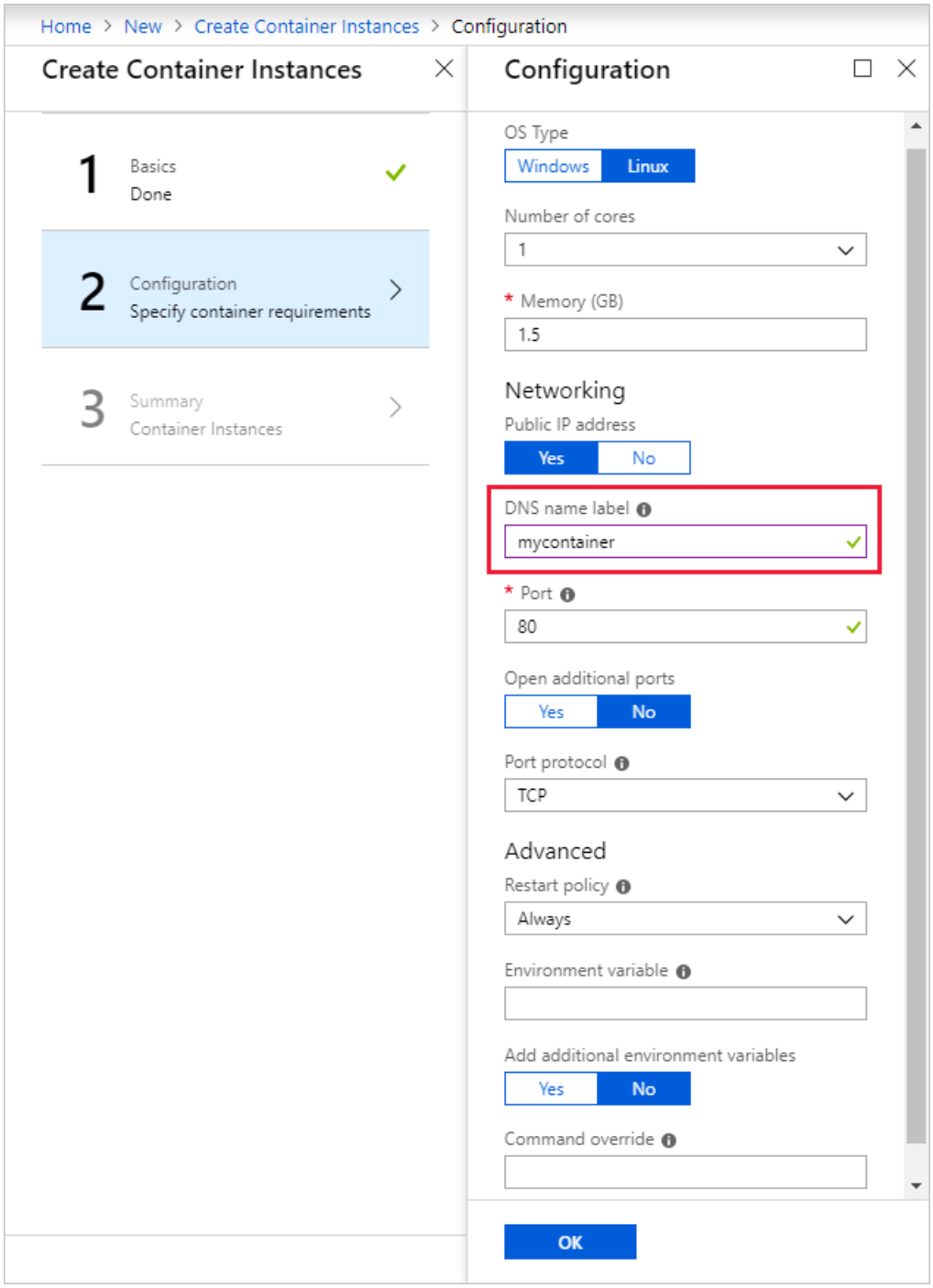



You can create both Windows and Linux containers in Azure Container Instances. For this quickstart, leave the

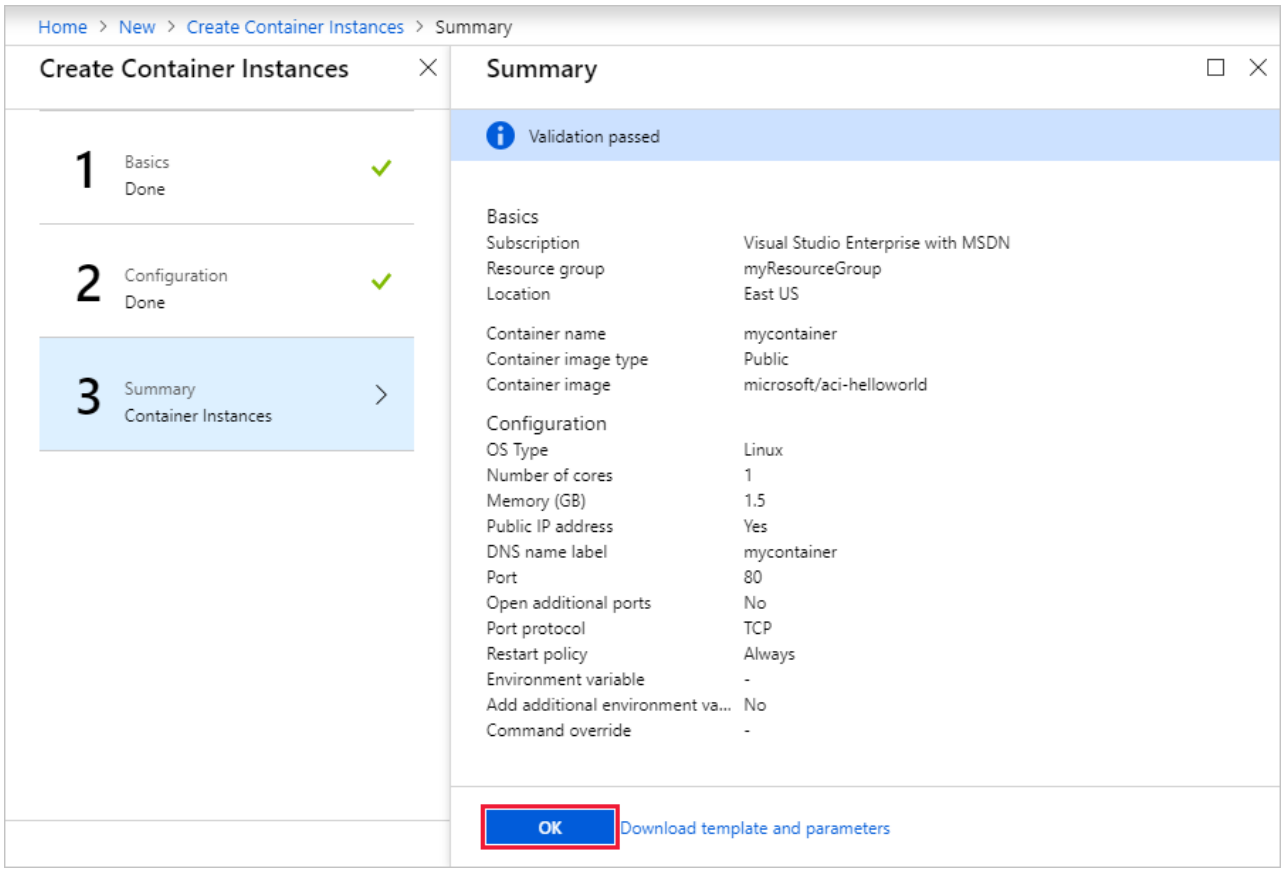
default setting of **Linux** to deploy the Linux-based `microsoft/aci-helloworld` image.

Under **Configuration**, specify a **DNS name label** for your container. The name must be unique within the Azure region you create the container instance. Your container will be publicly reachable at `<dns-name-label>.<region>.azurecontainer.io`.

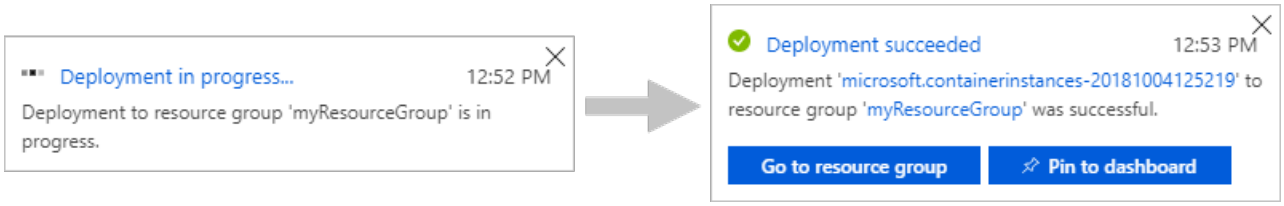
Leave the other settings in **Configuration** at their defaults, then select **OK** to validate the configuration.



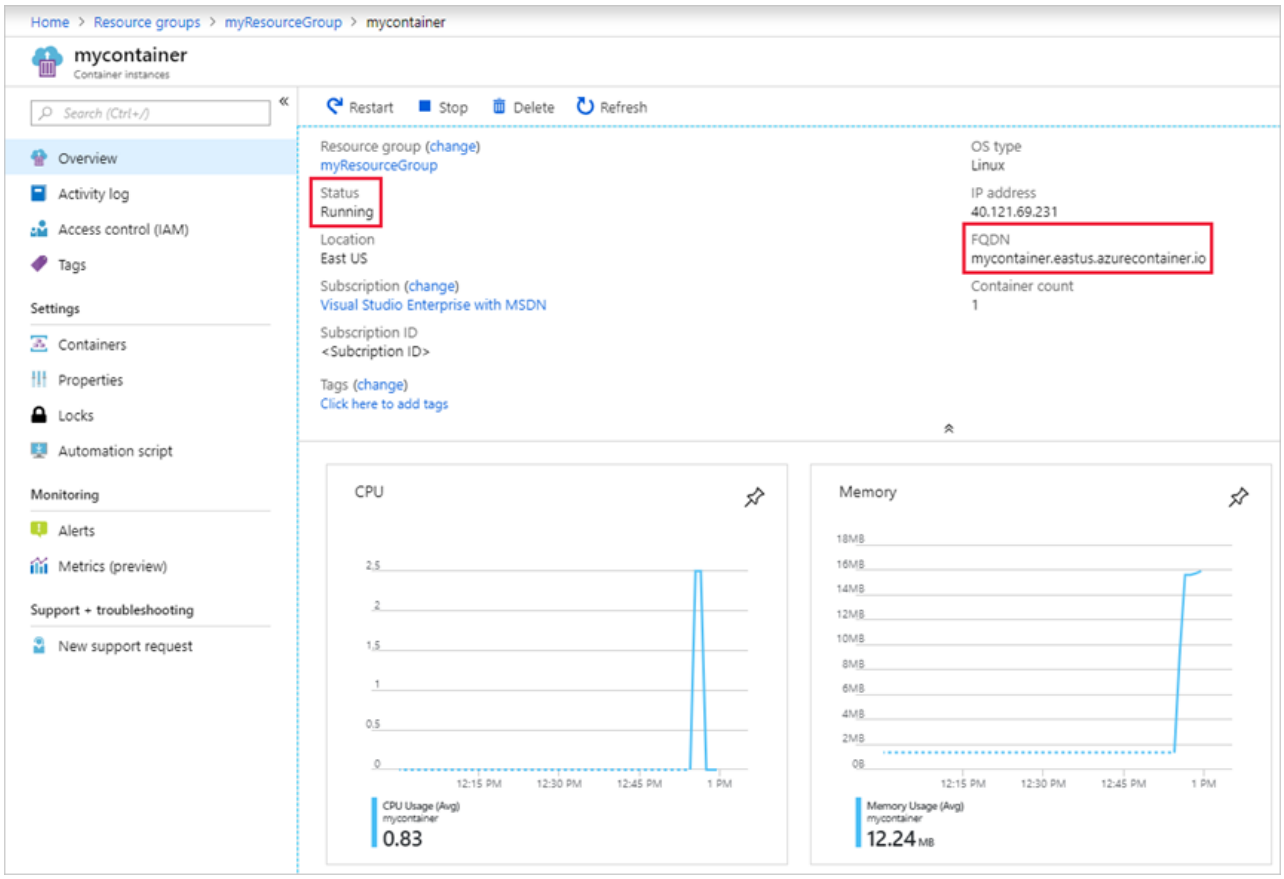

When the validation completes, you're shown a summary of the container's settings. Select **OK** to submit your container deployment request.




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**.



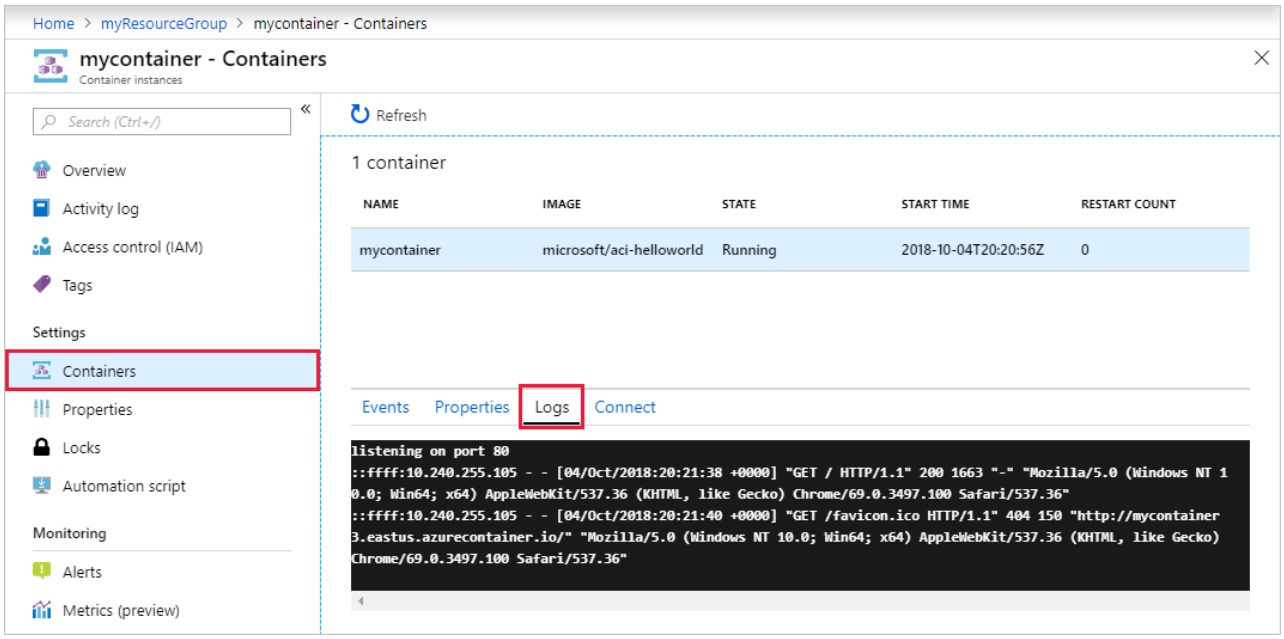

Once its **Status** is *Running*, navigate to the container's FQDN in your browser.


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

## View container logs

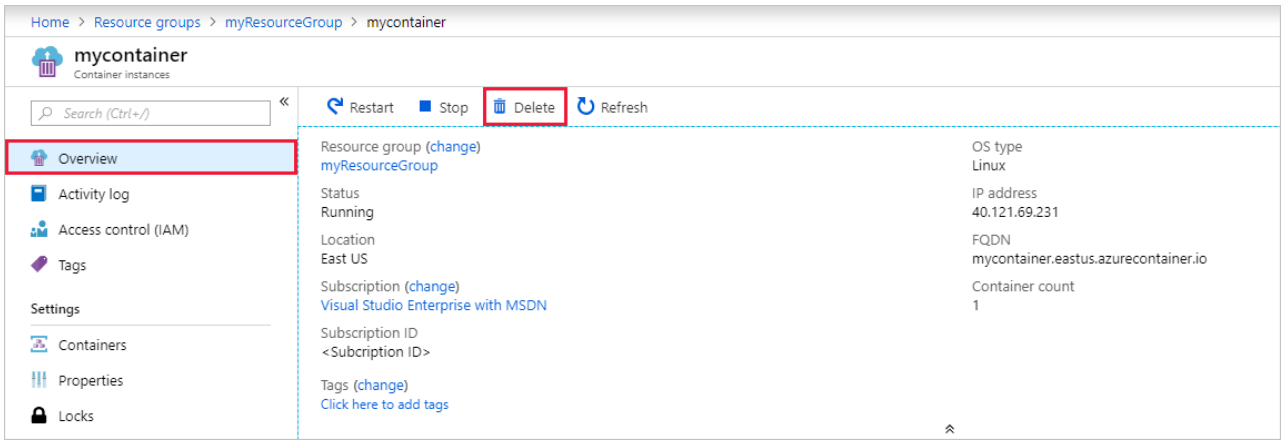
Viewing the logs for a container instance is helpful when troubleshooting issues with your container or the application it runs.

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.

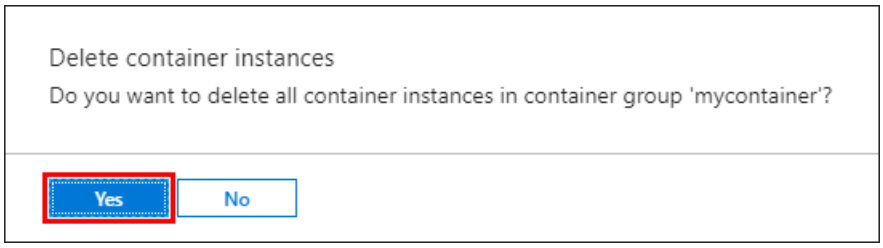



# Clean up resources

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




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




# Next steps

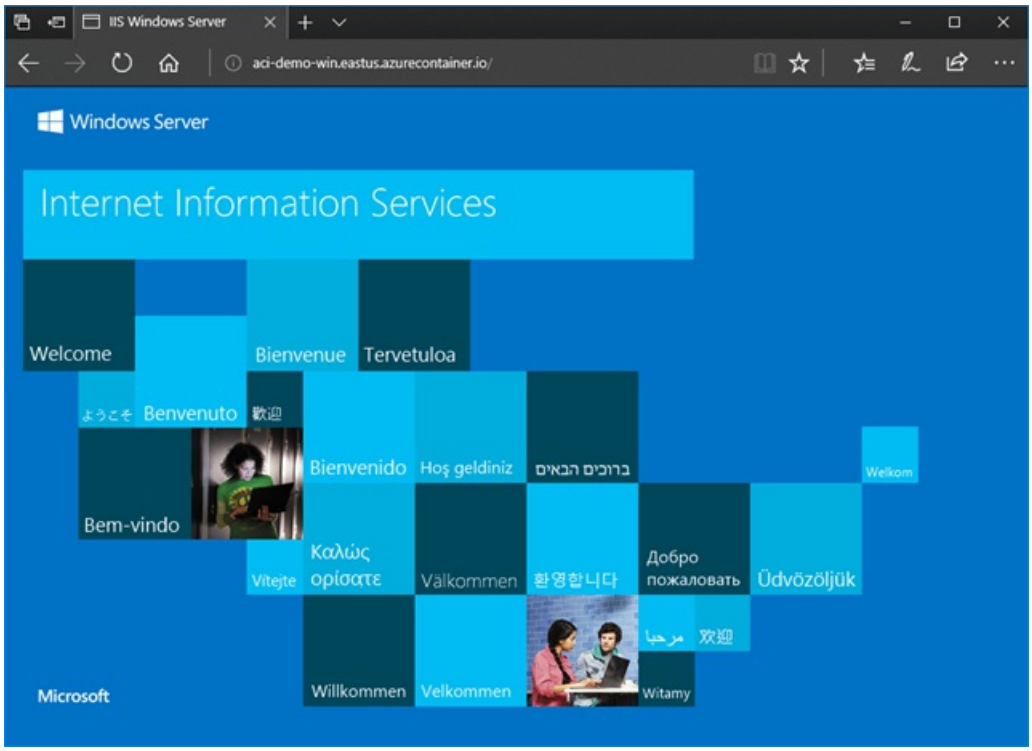
In this quickstart, you created an Azure container instance from an image in the public Docker Hub registry. If you'd like to build a container image and deploy it from a private Azure container registry, continue to the Azure Container Instances tutorial.

Azure Container Instances tutorial

# Quickstart: Run an application in Azure Container Instances

10/8/2018 • 3 minutes to read • Edit Online

Use Azure Container Instances to run Docker containers in Azure with simplicity and speed. You don't need to deploy virtual machines or use a full container orchestration platform like Kubernetes. In this quickstart, you use the Azure portal to create a Windows container in Azure 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 running application:




If you don't have an Azure subscription, create a free account before you begin.

## 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 **Try It** in the upper right corner of a code block. |   |
| Open Cloud Shell in your browser. |   |
| Click the **Cloud Shell** 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 version

5.5 or later. Run `Get-Module -ListAvailable AzureRM` 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-AzureRmAccount` 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-AzureRmResourceGroup command:

```
New-AzureRmResourceGroup -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-AzureRmContainerGroup cmdlet. 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 that hosts Internet Information Services (IIS) running in Nano Server.

Execute the following command to start a container instance. The `-DnsNameLabel` value must be unique within the Azure region you create the instance. If you receive a "DNS name label not available" error message, try a different DNS name label.

```
New-AzureRmContainerGroup -ResourceGroupName myResourceGroup -Name mycontainer -Image microsoft/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-AzureRmContainerGroup cmdlet:

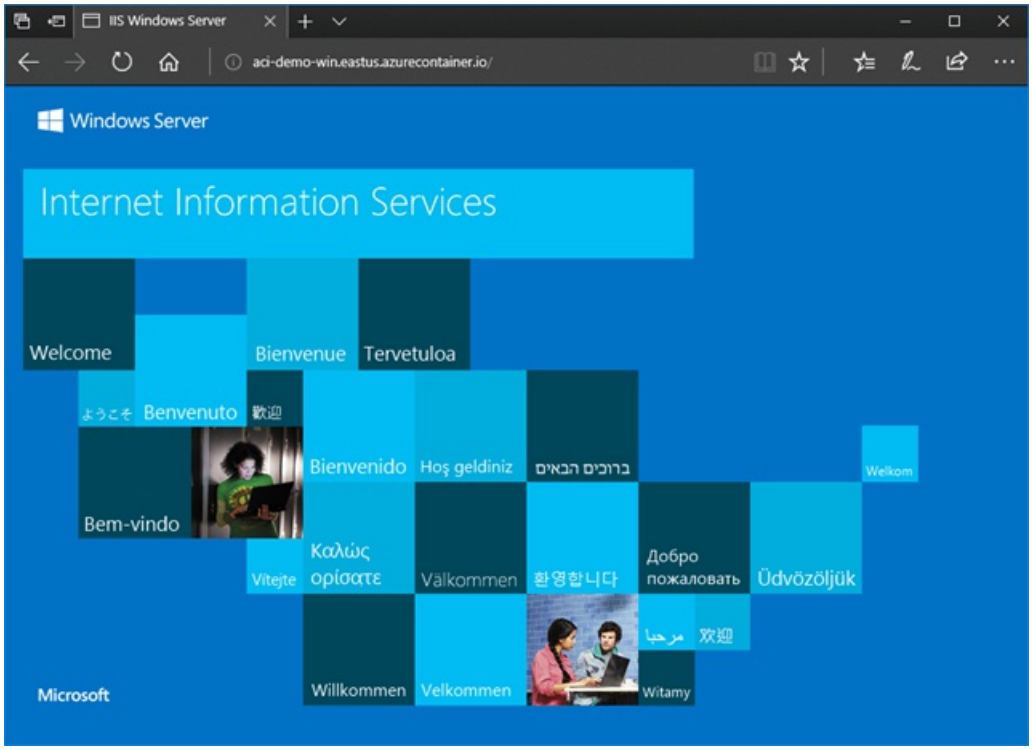
```
Get-AzureRmContainerGroup -ResourceGroupName myResourceGroup -Name mycontainer
```

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

```
PS Azure:\> Get-AzureRmContainerGroup -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 theRemove-AzureRmContainerGroup cmdlet:

```
Remove-AzureRmContainerGroup -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 container for deployment to Azure Container Instances

10/8/2018 • 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.

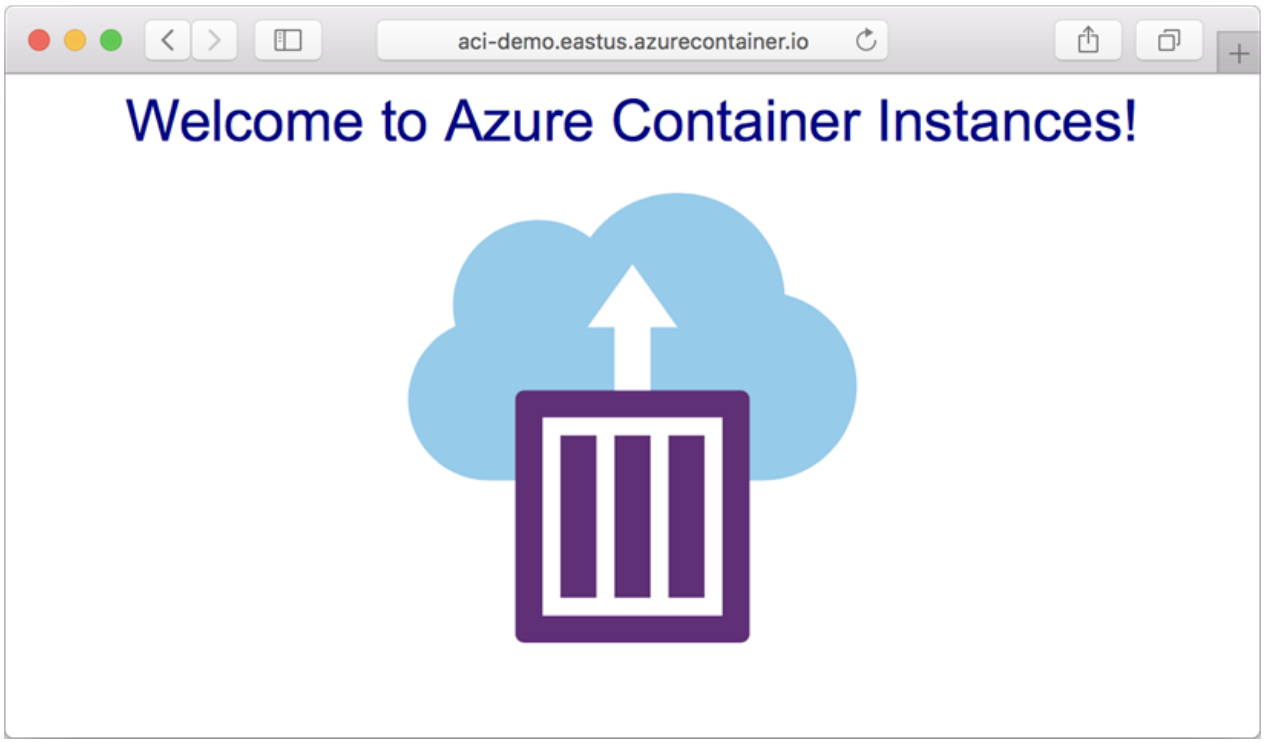
**Docker**: This tutorial assumes a basic understanding of core Docker concepts like containers, container images, and basic `docker` commands. For a primer on Docker and container basics, see the Docker overview.

**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

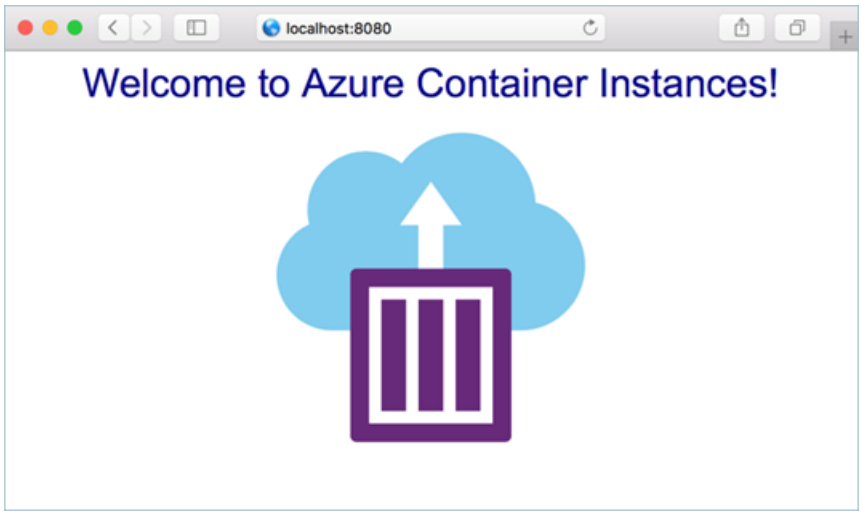
Before you deploy the container to Azure Container Instances, use docker run to run it locally and confirm that it works. The `-d` switch lets the container run in the background, while `-p` allows you to map an arbitrary port on your computer to port 80 in the container.

```
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 and use Azure Container Registry

10/8/2018 • 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/<Subscription
ID>/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 to Azure Container Instances

10/8/2018 • 3 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 the registry's credentials.

First, get the full name of the container registry login server (replace `<acrName>` with the name of your registry):

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

Next, get the container registry password:

```
az acr credential show --name <acrName> --query "passwords[0].value"
```

### Deploy container

Now, use the az container create command to deploy the container. Replace `<acrLoginServer>` and `<acrPassword>` with the values you obtained from the previous two commands. Replace `<acrName>` with the name of your container registry.

```
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 <acrName> --registry-password <acrPassword> --dns-name-label aci-demo --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

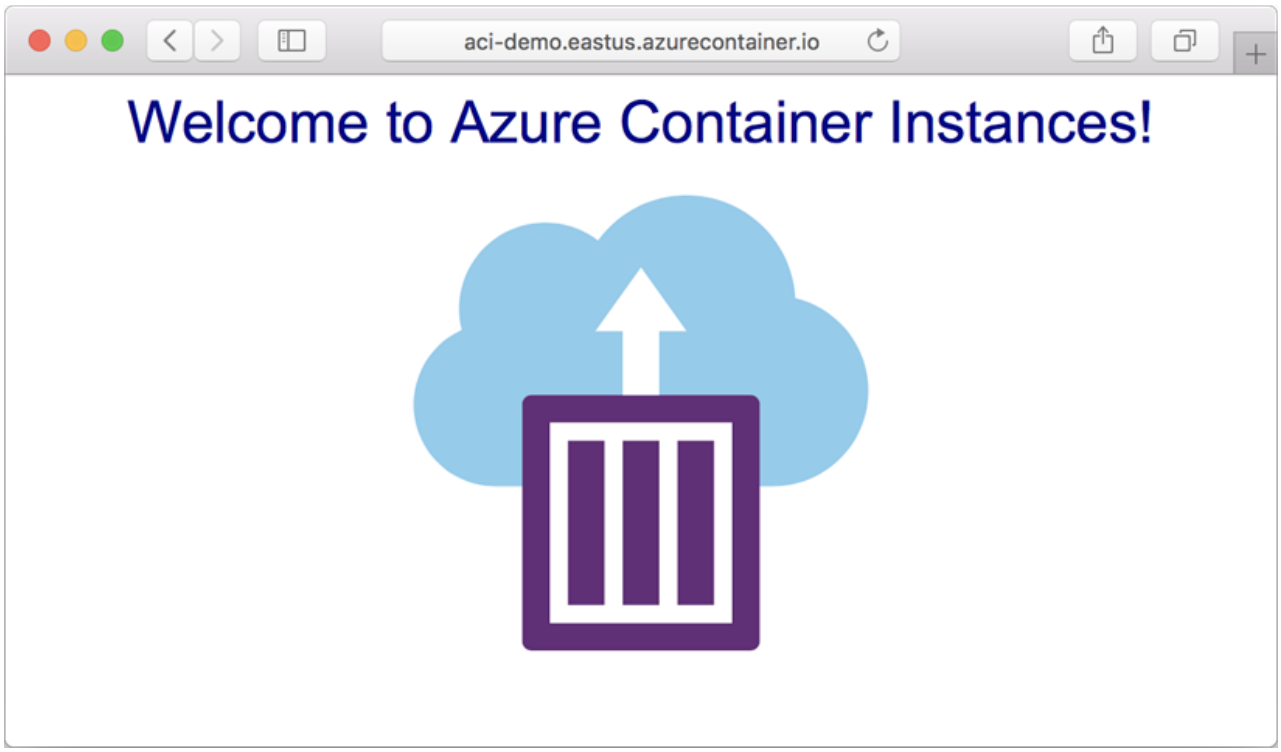
Once the deployment succeeds, display the container's fully qualified domain name (FQDN) with the az container show command:

```
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:

# Azure Resource Manager templates for Azure Container Instances

10/8/2018 • 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

| | |
|---|---|
| **Applications** | |
| Wordpress | Creates a WordPress website and its MySQL database in a container instance. The WordPress site content and MySQL database are persisted to an Azure Files share. |
| MS NAV with SQL Server and IIS | Deploys a single Windows container with a fully featured self-contained Dynamics NAV / Dynamics 365 Business Central environment. |
| **Volumes** | |
| emptyDir | Deploys two Linux containers that share an emptyDir volume. |
| gitRepo | Deploys a Linux container that clones a GitHub repo and mounts it as a volume. |
| secret | Deploys a Linux container with a PFX cert mounted as a secret volume. |
| **Networking** | |
| UDP-exposed container | Deploys a Windows or Linux container that exposes a UDP port. |
| Linux container with public IP | Deploys a single Linux container accessible via a public IP. |
| **Azure resources** | |
| Create Azure Storage account and Files share | 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

# Quotas and region availability for Azure Container Instances

10/9/2018 • 2 minutes to read • Edit Online

All Azure services include certain default limits and quotas for resources and features. The following sections detail the default resource limits for several Azure Container Instances (ACI) resources, as well as the availability of the ACI service in Azure regions.

## Service quotas and limits

| RESOURCE | DEFAULT LIMIT |
|---|---|
| Container groups per subscription | 100[1] |
| Number of containers per container group | 60 |
| Number of volumes per container group | 20 |
| Ports per IP | 5 |
| Container creates per hour | 300[1] |
| Container creates per 5 minutes | 100[1] |
| Container deletes per hour | 300[1] |
| Container deletes per 5 minutes | 100[1] |
| Multiple containers per container group | Linux only[2] |
| Azure Files volumes | Linux only[2] |
| GitRepo volumes | Linux only[2] |
| Secret volumes | Linux only[2] |

[1] Create an Azure support request to request a limit increase.

[2] Windows support for this feature is planned.

## Region availability

Azure Container Instances is available in the following regions with the specified CPU and memory limits.

| LOCATION | OS | CPU | MEMORY (GB) |
|---|---|---|---|
| East US, North Europe, West Europe, West US, West US 2 | Linux | 4 | 14 |

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

Container instances 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 CPU and memory settings, or try your deployment at a later time.

Let the team know of additional regions required or increased CPU/Memory limits at aka.ms/aci/feedback.

For more information on troubleshooting container instance deployment, see Troubleshoot deployment issues with Azure Container Instances.

## 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**).

# Azure Container Instances and container orchestrators

10/4/2018 • 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.

## Potential scenarios

While orchestrator integration with Azure Container Instances is still nascent, we anticipate that a few different environments may 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 Kubelet for Kubernetes

The Virtual Kubelet project demonstrates how container orchestration platforms can integrate with Azure Container Instances.

Virtual Kubelet mimics the Kubernetes kubelet by registering as a node with unlimited capacity and dispatching the creation of pods as container groups in Azure Container Instances.

Connectors for other orchestrators could be built that similarly integrate with platform primitives to combine the power of the orchestrator API with the speed and simplicity of managing containers in Azure Container Instances.

## Next steps

Create your first container with Azure Container Instances using the quickstart guide.

# Container groups in Azure Container Instances

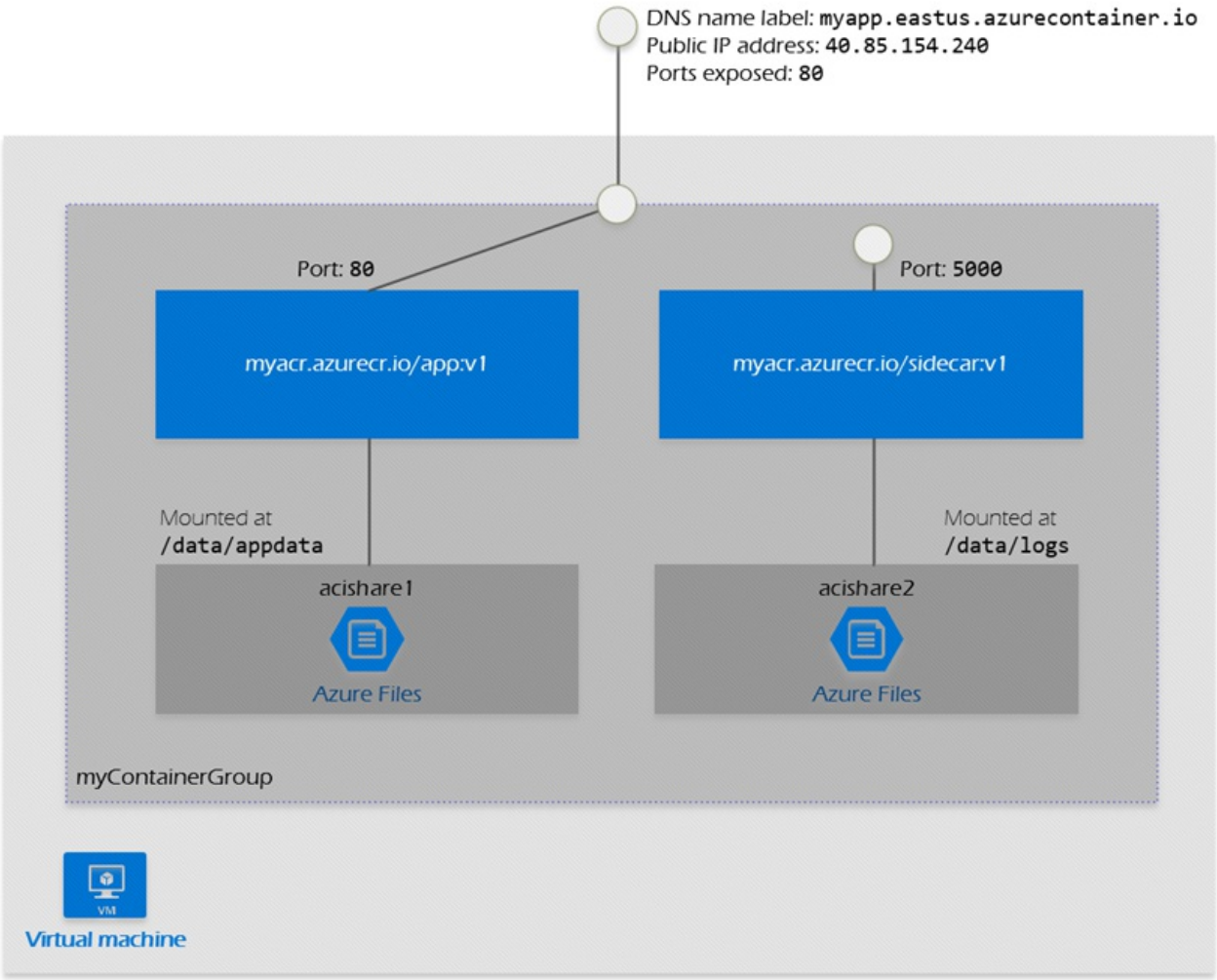
4/30/2018 • 2 minutes to read • Edit Online

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, local network, and storage volumes. It's similar in concept to a *pod* in Kubernetes and DC/OS.

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 are 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.

## Deployment

Container *groups* have a minimum resource allocation of 1 vCPU and 1 GB memory. Individual *containers* within a container group can be provisioned with less than 1 vCPU and 1 GB memory. Within a container group, the distribution of resources can be customized to multiple containers within the limits established at the container group-level. For example, two containers each with 0.5 vCPU residing in a container group that's allocated 1 vCPU.

## 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 is not supported. Containers within a group can reach each other via localhost on the ports that they have exposed, even if those ports are not exposed externally on the group's IP address.

## 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:

- 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 container serving a web application and a container pulling the latest content from source control.

## Next steps

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

Deploy a container group

# Deploy a multi-container container group with YAML

10/8/2018 • 4 minutes to read • Edit Online

Azure Container Instances supports the deployment of multiple containers onto a single host by using a container group. Multi-container container groups are useful when building an application sidecar for logging, monitoring, or any other configuration where a service needs a second attached process.

There are two methods for deploying multi-container groups using the Azure CLI:

- YAML file deployment (this article)
- Resource Manager template deployment

Due to the YAML format's more concise nature, deployment with a YAML file is recommended when your deployment includes *only* container instances. If you need to deploy additional Azure service resources (for example, an Azure Files share) at the time of container instance deployment, Resource Manager template deployment is recommended.

> **NOTE**
> Multi-container groups are currently restricted to Linux containers. 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.

## Configure the YAML file

To deploy a multi-container 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**.

This YAML file defines a container group named "myContainerGroup" 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, periodically makes HTTP requests to the web application running in the first container via the container group's local network.

```
apiVersion: 2018-06-01
location: eastus
name: myContainerGroup
properties:
  containers:
  - name: aci-tutorial-app
    properties:
      image: microsoft/aci-helloworld:latest
      resources:
        requests:
          cpu: 1
          memoryInGb: 1.5
      ports:
      - port: 80
      - port: 8080
  - name: aci-tutorial-sidecar
    properties:
      image: microsoft/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
```

## 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    ProvisioningState    Image
IP:ports              CPU/Memory       OsType    Location
----------------  ---------------  -------------------  ------------------------------------------------------
--------  ---------------------  ---------------  --------  ----------
myContainerGroup  myResourceGroup  Succeeded            microsoft/aci-helloworld:latest,microsoft/aci-
tutorial-sidecar  52.168.26.124:80,8080  1.0 core/1.5 gb  Linux     eastus
```

## View 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 first container is specified.

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

Output:

```
listening on port 80
::1 - - [09/Jan/2018:23:17:48 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
::1 - - [09/Jan/2018:23:17:51 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
::1 - - [09/Jan/2018:23:17:54 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
```

To see the logs for the side-car container, run the same command specifying the second container name.

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

Output:

```
Every 3s: curl -I http://localhost                          2018-01-09 23:25:11

  % 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: Tue, 09 Jan 2018 23:25:11 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.

## Deploy from private registry

To use a private container image registry, include the following YAML with values modified for your environment:

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

For example, the following YAML deploys a container group with a single container whose image is pulled from a private Azure Container Registry named "myregistry":

```
apiVersion: 2018-06-01
location: eastus
name: myContainerGroup2
properties:
  containers:
  - name: aci-tutorial-app
    properties:
      image: myregistry.azurecr.io/aci-helloworld:latest
      resources:
        requests:
          cpu: 1
          memoryInGb: 1.5
      ports:
      - port: 80
  osType: Linux
  ipAddress:
    type: Public
    ports:
    - protocol: tcp
      port: '80'
  imageRegistryCredentials:
  - server: myregistry.azurecr.io
    username: myregistry
    password: REGISTRY_PASSWORD
tags: null
type: Microsoft.ContainerInstance/containerGroups
```

## Export container group to YAML

You can export the configuration of an existing container group to a YAML file by using the Azure CLI command az container export.

Useful for preserving a container group's configuration, 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.

Export the configuration for the container group you created earlier by issuing the following az container export command:

```
az container export --resource-group myResourceGroup --name myContainerGroup --file deployed-aci.yaml
```

No output is displayed if the command is successful, but you can view the contents of the file to see the result. For example, the first few lines with `head`:

```
$ head deployed-aci.yaml
additional_properties: {}
apiVersion: '2018-06-01'
location: eastus
name: myContainerGroup
properties:
  containers:
  - name: aci-tutorial-app
    properties:
      environmentVariables: []
      image: microsoft/aci-helloworld:latest
```

## Next steps

This article covered the steps needed for deploying a multi-container Azure container instance. For an end-to-end Azure Container Instances experience, including using a private Azure container registry, see the Azure Container Instances tutorial.

Azure Container Instances tutorial

# Deploy a container group

10/8/2018 • 3 minutes to read • Edit Online

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

There are two methods for deploying multi-container groups using the Azure CLI:

- Resource Manager template deployment (this article)
- YAML file deployment

Deployment with a Resource Manager template is recommended when you need to deploy additional Azure service resources (for example, an Azure Files share) at the time of container instance deployment. Due to the YAML format's more concise nature, deployment with a YAML file is recommended when your deployment includes *only* container instances.

> **NOTE**
> Multi-container groups are 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.

## Configure the template

The sections in this article walk you through running a simple multi-container sidecar configuration by deploying an Azure Resource Manager template.

Start by creating a file named `azuredeploy.json`, then copy the following JSON into it.

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 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": "microsoft/aci-helloworld:latest",
    "container2name": "aci-tutorial-sidecar",
    "container2image": "microsoft/aci-tutorial-sidecar"
  },
  "resources": [
    {
      "name": "[parameters('containerGroupName')]",
      "type": "Microsoft.ContainerInstance/containerGroups",
      "apiVersion": "2018-04-01",
```

```
      "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",
              "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    ProvisioningState    Image
IP:ports               CPU/Memory       OsType    Location
----------------  ---------------  -------------------  ------------------------------------------------------
--------  ---------------------  ---------------  --------  ----------
myContainerGroup  myResourceGroup  Succeeded            microsoft/aci-helloworld:latest,microsoft/aci-
tutorial-sidecar  52.168.26.124:80,8080  1.0 core/1.5 gb  Linux     westus
```

## View 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 first container is specified.

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

Output:

```
listening on port 80
::1 - - [09/Jan/2018:23:17:48 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
::1 - - [09/Jan/2018:23:17:51 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
::1 - - [09/Jan/2018:23:17:54 +0000] "HEAD / HTTP/1.1" 200 1663 "-" "curl/7.54.0"
```

To see the logs for the side-car container, run the same command specifying the second container name.

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

Output:

```
Every 3s: curl -I http://localhost                          2018-01-09 23:25:11

  % 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: Tue, 09 Jan 2018 23:25:11 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

This article covered the steps needed for deploying a multi-container Azure container instance. For an end-to-end Azure Container Instances experience, see the Azure Container Instances tutorial.

Azure Container Instances tutorial

# Deploy container instances into an Azure virtual network

10/9/2018 • 10 minutes to read • Edit Online

Azure Virtual Network provides secure, private networking including filtering, routing, and peering 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.

- Windows containers are unsupported
- 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.
- Container groups deployed to a virtual network do not currently support public IP addresses or DNS name labels.
- 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 instances to a virtual network.

**Supported** regions:

- West Europe (westeurope)
- West US (westus)

**Unsupported** network resources:

- Network Security Group
- Azure Load Balancer

**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)**

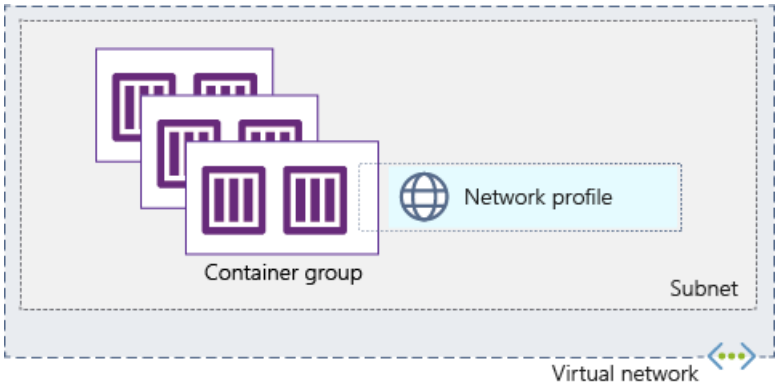
Subnets segment the virtual network into separate address spaces usable by the Azure resources you place in them. You create one or several subnets within a virtual network.

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. The first time you 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.

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.




# Deploy to virtual network

You can 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
     or
   - Network profile name or ID

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.

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 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. This command deploys the 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 microsoft/aci-helloworld \
    --vnet-name 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-name 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: microsoft/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   microsoft/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. 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
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

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

10/8/2018 • 4 minutes to read • Edit Online

The Azure Container Registry is an Azure-based, private registry for your 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 this 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 the 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 $ACR_NAME-pull \
               --scopes $(az acr show --name $ACR_NAME --query id --output tsv) \
               --role reader \
               --query password \
               --output tsv)
```

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

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.

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_NAME.azurecr.io/aci-helloworld:v1 \
    --registry-login-server $ACR_NAME.azurecr.io \
    --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_NAME.azurecr.io/aci-
helloworld:v1 --registry-login-server $ACR_NAME.azurecr.io --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:

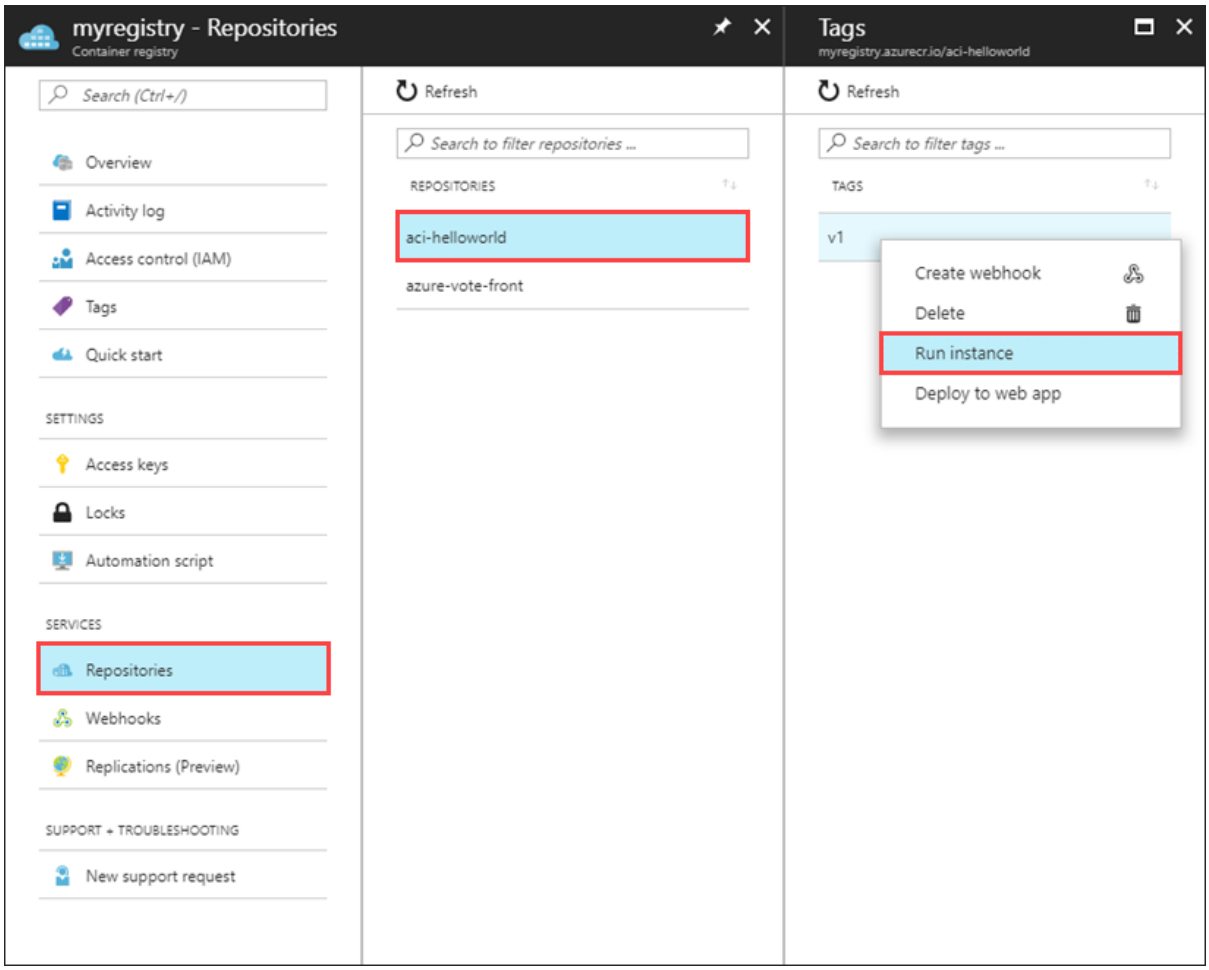
```
"imageRegistryCredentials": [
  {
    "server": "imageRegistryLoginServer",
    "username": "imageRegistryUsername",
    "password": "imageRegistryPassword"
  }
]
```

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.

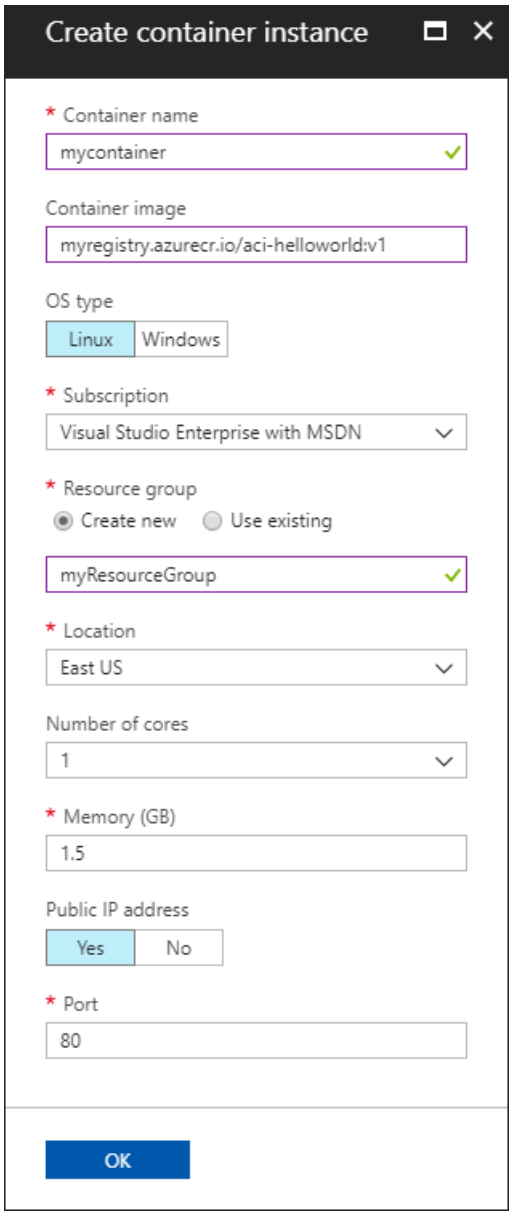
## Deploy with Azure portal

If you maintain container images in the Azure Container Registry, you can easily create a container in Azure Container Instances using the Azure portal.

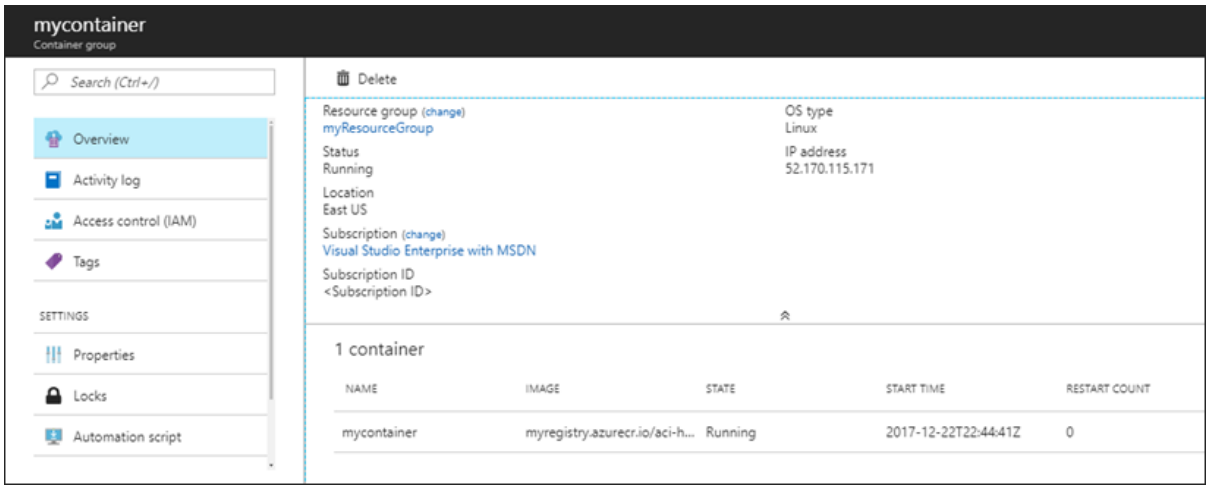
1. In the Azure portal, navigate to your container registry.
2. Select **Repositories**, then select the repository that you want to deploy from, right-click the tag for the container image you want to deploy, and select **Run instance**.




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




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




## Next steps

For more information about Azure Container Registry authentication, see Authenticate with an Azure container registry.

# Set environment variables

10/8/2018 • 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. To set environment variables in a container, specify them when you create a container instance. You can set 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.

## Azure CLI example

To see the default output of the microsoft/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 microsoft/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:

```
az container create \
    --resource-group myResourceGroup \
    --name mycontainer2 \
    --image microsoft/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 microsoft/aci-wordcount container in its default configuration with this New-AzureRmContainerGroup command:

```
New-AzureRmContainerGroup `
    -ResourceGroupName myResourceGroup `
    -Name mycontainer1 `
    -Image microsoft/aci-wordcount:latest
```

Now run the following New-AzureRmContainerGroup command. This one specifies the *NumWords* and *MinLength* environment variables after populating an array variable, `envVars`:

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

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

```
Get-AzureRmContainerInstanceLog -ResourceGroupName myResourceGroup -ContainerGroupName mycontainer1
Get-AzureRmContainerInstanceLog -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-AzureRmContainerInstanceLog -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-AzureRmContainerInstanceLog -ResourceGroupName myResourceGroup -ContainerGroupName mycontainer2
[('CLAUDIUS', 120),
 ('POLONIUS', 113),
 ('GERTRUDE', 82),
 ('ROSENCRANTZ', 69),
 ('GUILDENSTERN', 54)]

Azure:\
```

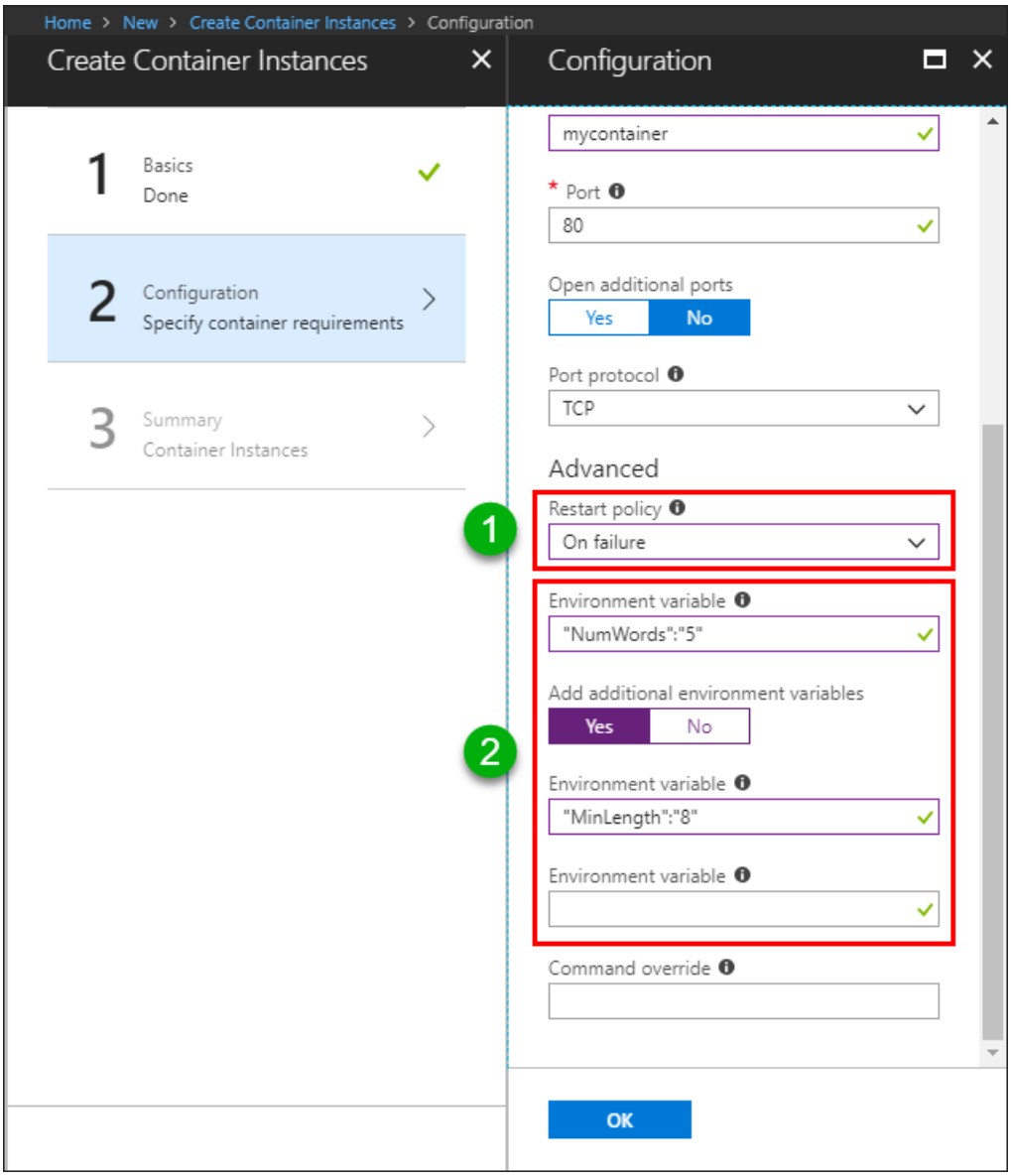
## Azure portal example

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

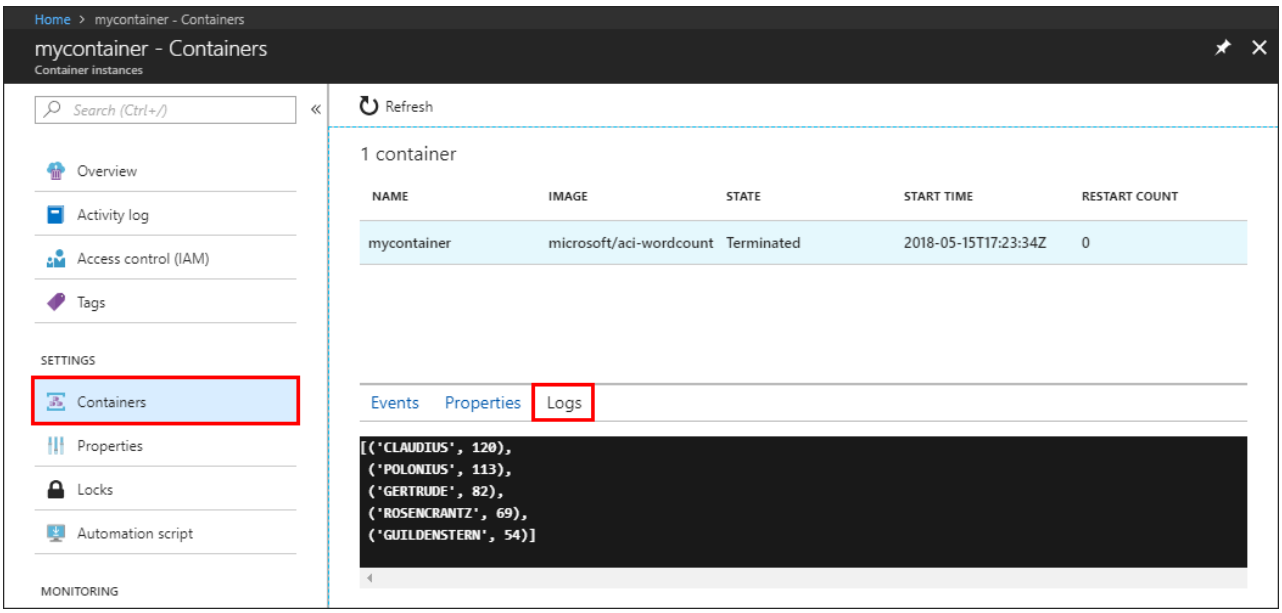
When you deploy with the portal, you're currently limited to three variables, and you must enter them in this format: `"variableName":"value"`

To see an example, start the microsoft/aci-wordcount container with the *NumWords* and *MinLength* variables.

1. In **Configuration**, set the **Restart policy** to *On failure*
2. Enter `"NumWords":"5"` for the first variable, select **Yes** under **Add additional environment variables**, and enter `"MinLength":"8"` for the second variable. Select **OK** to verify and then deploy the container.




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.




## 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: 2018-06-01
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 in Azure Container Instances.

# Configure liveness probes

7/23/2018 • 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: livenesstest
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 livenesstest -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.

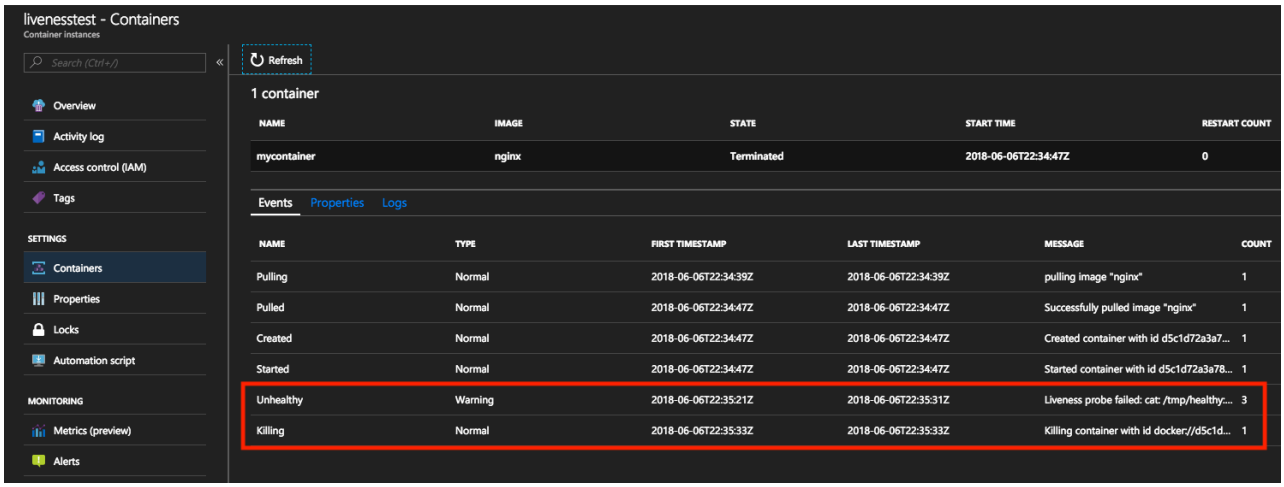
The `periodSeconds` property designates the liveness command should execute every 5 seconds.

## 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.

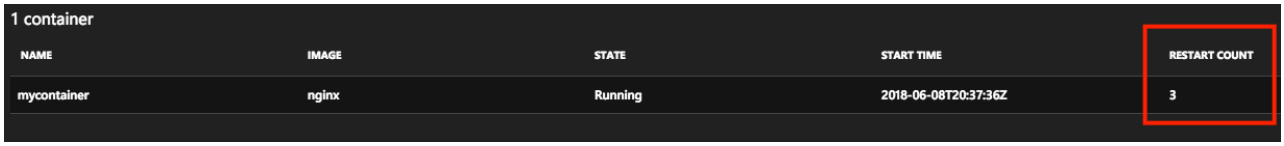
After 30 seconds, the `cat /tmp/healthy` will begin to fail, causing unhealthy and killing events to occur.

These events can be viewed from the Azure portal or Azure CLI.




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.




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.

# 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 "redeployed," 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

# 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.

# Monitor container resources in Azure Container Instances

10/8/2018 • 3 minutes to read • Edit Online

Azure Monitor provides insight into the compute resources used by your containers instances. Use Azure Monitor to track the CPU and memory utilization of container groups and their containers. This resource usage data helps you determine the best CPU and memory settings for your container groups.

This document details gathering CPU and memory usage for container instances using both the Azure portal and Azure CLI.

> **IMPORTANT**
> At this time, resource usage metrics are only available for Linux containers.

## Available metrics

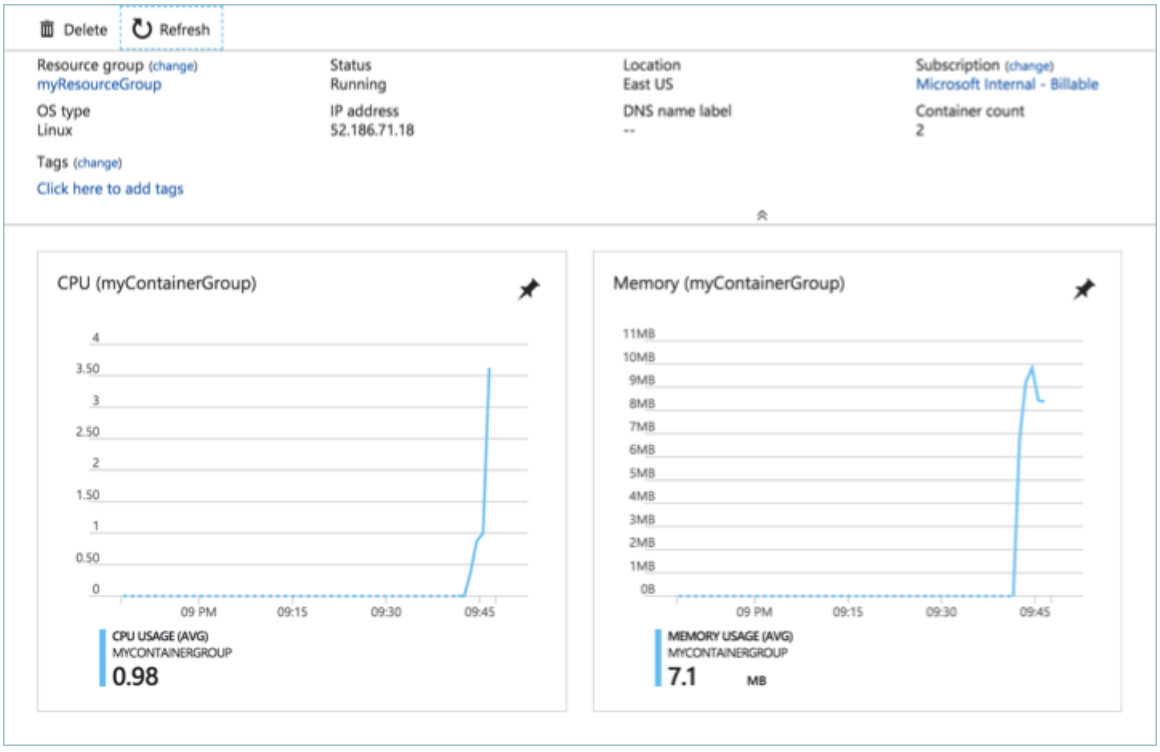
Azure Monitor provides metrics on both **CPU** and **memory** usage for Azure Container Instances. Both metrics are available for a container group and individual containers.

CPU metrics are expressed in **millicores**. One millicore is 1/1000th of a CPU core, so 500 millicores (or 500 m) represents 50% utilization of a CPU core.

Memory metrics are expressed in **bytes**.

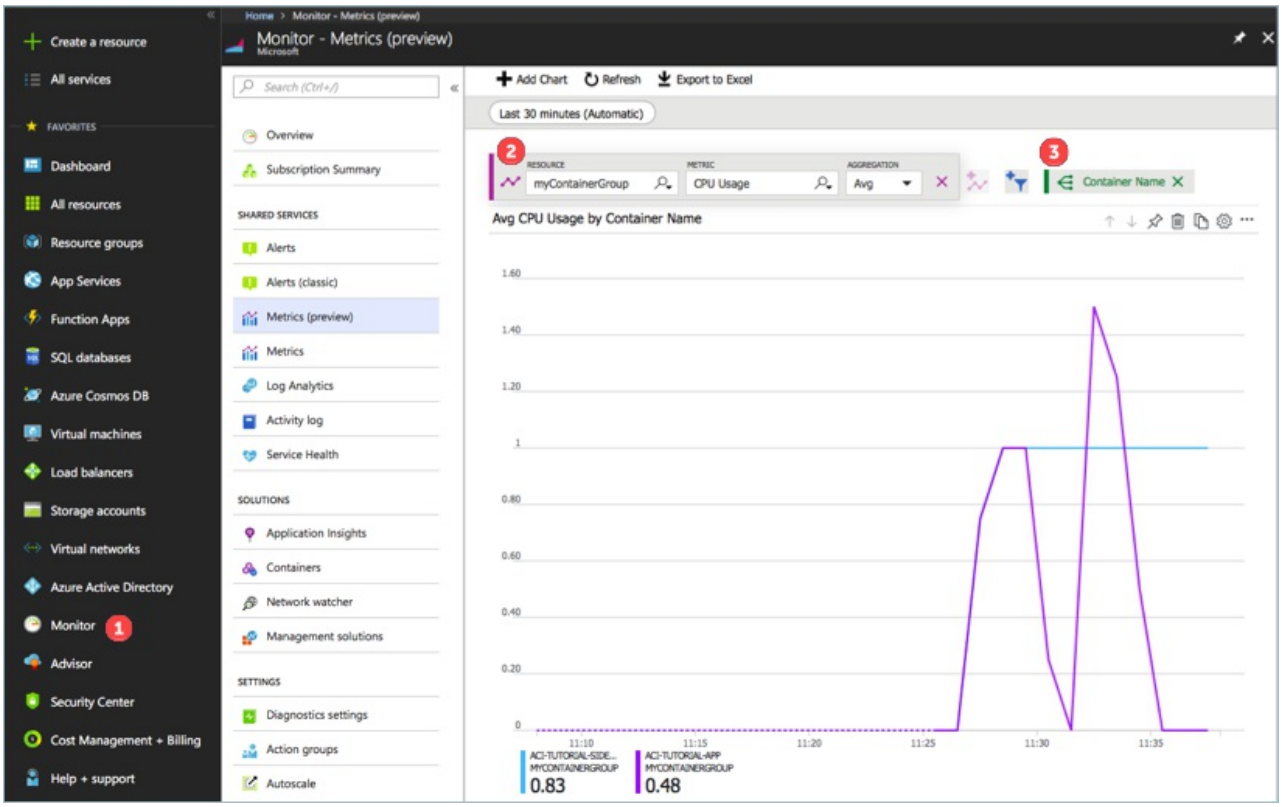
## Get metrics - Azure portal

When a container group is created, Azure Monitor data is available in the Azure portal. To see metrics for a container group, select the resource group and then the container group. Here you can see pre-created charts for both CPU and memory usage.



If you have a container group that contains multiple containers, use a dimension to present metrics for each individual container. To create a chart with individual container metrics, perform the following steps:

1. Select **Monitor** from the left-hand navigation menu.
2. Select a container group and a metric (CPU or Memory).
3. Select the green dimension button, and select **Container Name**.




## Get metrics - Azure CLI

Container instances CPU and memory usage 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
-------------------  ------------  -----------
2018-04-22 04:39:00  CPU Usage
2018-04-22 04:40:00  CPU Usage
2018-04-22 04:41:00  CPU Usage
2018-04-22 04:42:00  CPU Usage
2018-04-22 04:43:00  CPU Usage      0.375
2018-04-22 04:44:00  CPU Usage      0.875
2018-04-22 04:45:00  CPU Usage      1
2018-04-22 04:46:00  CPU Usage      3.625
2018-04-22 04:47:00  CPU Usage      1.5
2018-04-22 04:48:00  CPU Usage      2.75
2018-04-22 04:49:00  CPU Usage      1.625
2018-04-22 04:50:00  CPU Usage      0.625
2018-04-22 04:51:00  CPU Usage      0.5
2018-04-22 04:52:00  CPU Usage      0.5
2018-04-22 04:53:00  CPU Usage      0.5
```

And the following command to get **memory** usage metrics.

```
$ az monitor metrics list --resource $CONTAINER_GROUP --metric MemoryUsage --output table

Timestamp            Name              Average
-------------------  ------------  -----------
2018-04-22 04:38:00  Memory Usage
2018-04-22 04:39:00  Memory Usage
2018-04-22 04:40:00  Memory Usage
2018-04-22 04:41:00  Memory Usage
2018-04-22 04:42:00  Memory Usage  6.76915e+06
2018-04-22 04:43:00  Memory Usage  9.22061e+06
2018-04-22 04:44:00  Memory Usage  9.83552e+06
2018-04-22 04:45:00  Memory Usage  8.42906e+06
2018-04-22 04:46:00  Memory Usage  8.39526e+06
2018-04-22 04:47:00  Memory Usage  8.88013e+06
2018-04-22 04:48:00  Memory Usage  8.89293e+06
2018-04-22 04:49:00  Memory Usage  9.2073e+06
2018-04-22 04:50:00  Memory Usage  9.36243e+06
2018-04-22 04:51:00  Memory Usage  9.30509e+06
2018-04-22 04:52:00  Memory Usage  9.2416e+06
2018-04-22 04:53:00  Memory Usage  9.1008e+06
```

For a multi-container group, the `containerName` dimension can be added to return this data per container.

```
$ az monitor metrics list --resource $CONTAINER_GROUP --metric CPUUsage --dimension containerName --output
table

Timestamp            Name          Containername             Average
-------------------  ------------  --------------------  -----------
2018-04-22 17:03:00  Memory Usage  aci-tutorial-app      1.95338e+07
2018-04-22 17:04:00  Memory Usage  aci-tutorial-app      1.93096e+07
2018-04-22 17:05:00  Memory Usage  aci-tutorial-app      1.91488e+07
2018-04-22 17:06:00  Memory Usage  aci-tutorial-app      1.94335e+07
2018-04-22 17:07:00  Memory Usage  aci-tutorial-app      1.97714e+07
2018-04-22 17:08:00  Memory Usage  aci-tutorial-app      1.96178e+07
2018-04-22 17:09:00  Memory Usage  aci-tutorial-app      1.93434e+07
2018-04-22 17:10:00  Memory Usage  aci-tutorial-app      1.92614e+07
2018-04-22 17:11:00  Memory Usage  aci-tutorial-app      1.90659e+07
2018-04-22 16:12:00  Memory Usage  aci-tutorial-sidecar  1.35373e+06
2018-04-22 16:13:00  Memory Usage  aci-tutorial-sidecar  1.28614e+06
2018-04-22 16:14:00  Memory Usage  aci-tutorial-sidecar  1.31379e+06
2018-04-22 16:15:00  Memory Usage  aci-tutorial-sidecar  1.29536e+06
2018-04-22 16:16:00  Memory Usage  aci-tutorial-sidecar  1.38138e+06
2018-04-22 16:17:00  Memory Usage  aci-tutorial-sidecar  1.41312e+06
2018-04-22 16:18:00  Memory Usage  aci-tutorial-sidecar  1.49914e+06
2018-04-22 16:19:00  Memory Usage  aci-tutorial-sidecar  1.43565e+06
2018-04-22 16:20:00  Memory Usage  aci-tutorial-sidecar  1.408e+06
```

## Next steps

Learn more about Azure Monitoring at the Azure Monitoring overview.

# Container instance logging with Azure Log Analytics

10/8/2018 • 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 Log Analytics.

To send container instance data to Log Analytics, 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.

## 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
```

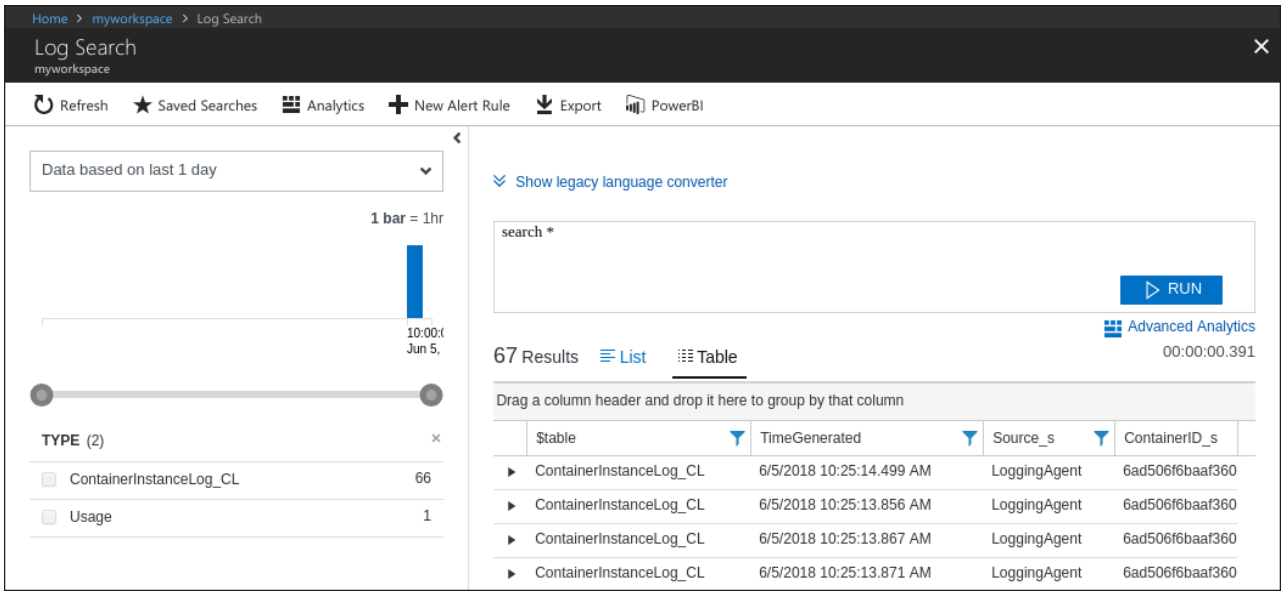
You should receive a response from Azure containing deployment details shortly after issuing the command.

# View logs in Log Analytics

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**
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

Log Analytics 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

### Log Analytics

For more information about querying logs and configuring alerts in Azure Log Analytics, see:

- Understanding log searches in Log Analytics
- 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.

# Run containerized tasks with restart policies

10/8/2018 • 4 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 in Azure Container Instances, you can specify one of three restart policy settings.

| RESTART POLICY | DESCRIPTION |
| --- | --- |
| `Always` | Containers in the container group are always restarted. This is the **default** setting applied when no restart policy is specified at container creation. |
| `Never` | Containers in the container group are never restarted. The containers run at most once. |
| `OnFailure` | 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 microsoft/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.

## Configure containers at runtime

When you create a container instance, you can set its **environment variables**, as well as specify a custom **command line** to execute when the container is started. You can use these settings in your batch jobs to prepare each container with task-specific configuration.

## Environment variables

Set environment variables in your container 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`.

For example, you can modify the behavior of the script in the example container by specifying the following environment variables when you create the container instance:

*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."

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

By specifying `NumWords=5` and `MinLength=8` for the container's environment variables, the container logs should display different output. Once the container status shows as *Terminated* (use `az container show` to check its status), display its logs to see the new output:

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

Output:

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

## Command line override

Specify a command line when you create a container instance to override the command line baked into the container image. This is similar to the `--entrypoint` command-line argument to `docker run`.

For instance, you can have the example container analyze text other than *Hamlet* by specifying a different command line. The Python script executed by the container, *wordcount.py*, accepts a URL as an argument, and will process that page's content instead of the default.

For example, to determine the top 3 five-letter words in *Romeo and Juliet*:

```
az container create \
    --resource-group myResourceGroup \
    --name mycontainer3 \
    --image microsoft/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 mycontainer3
```

Output:

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

## Next steps

**Persist task output**

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.

# Use Azure Container Instances as a Jenkins build agent

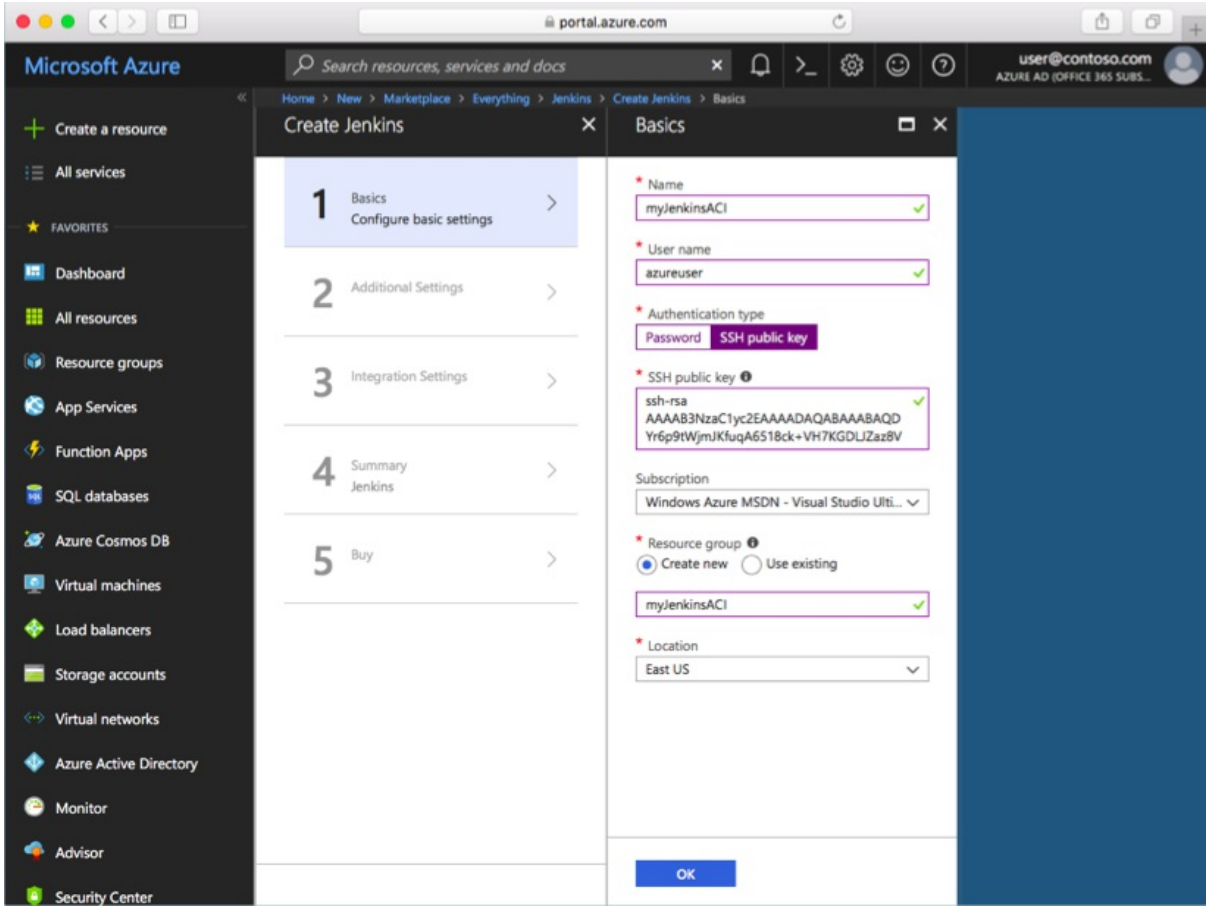
10/8/2018 • 4 minutes to read • Edit Online

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.

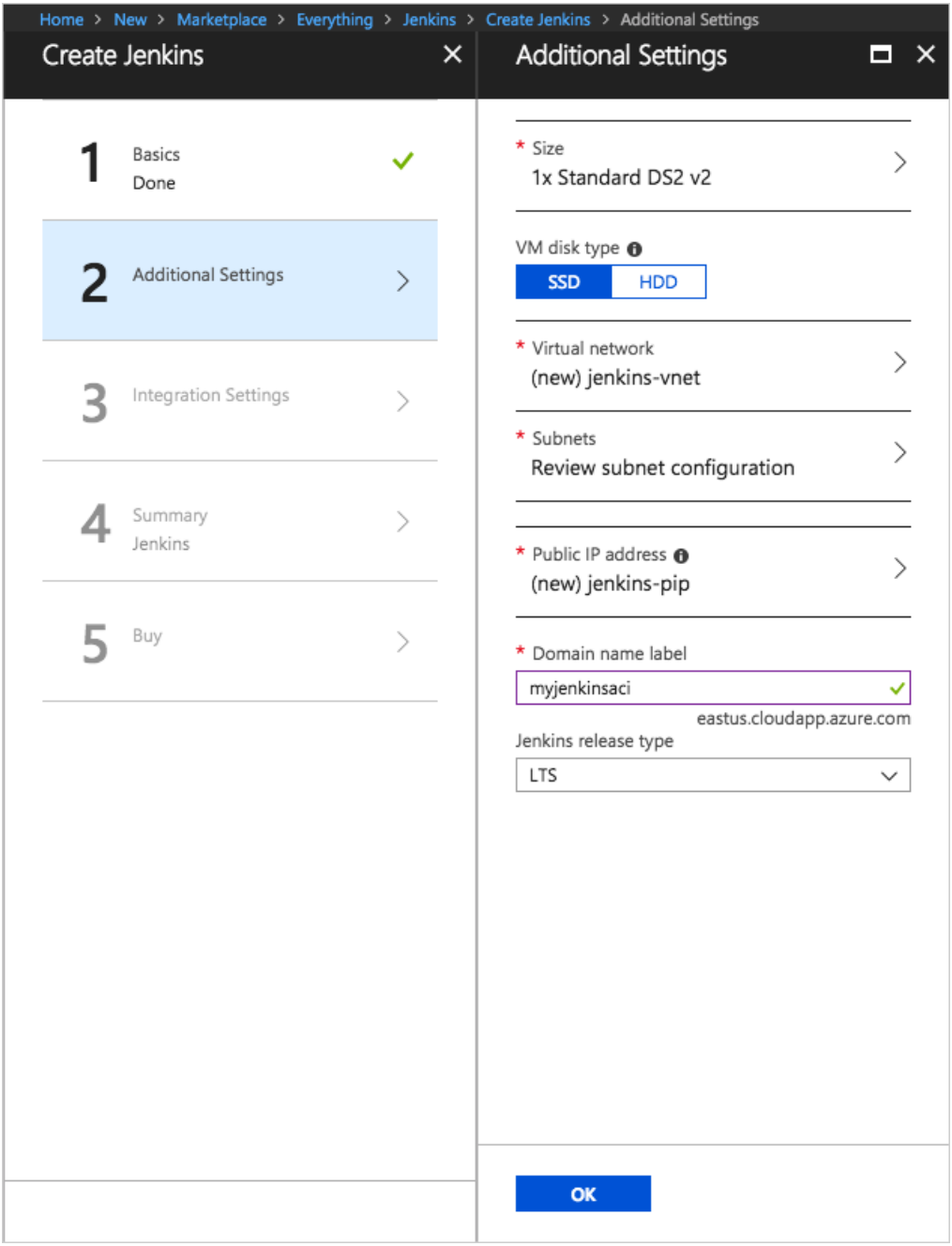
## Deploy a Jenkins server

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.



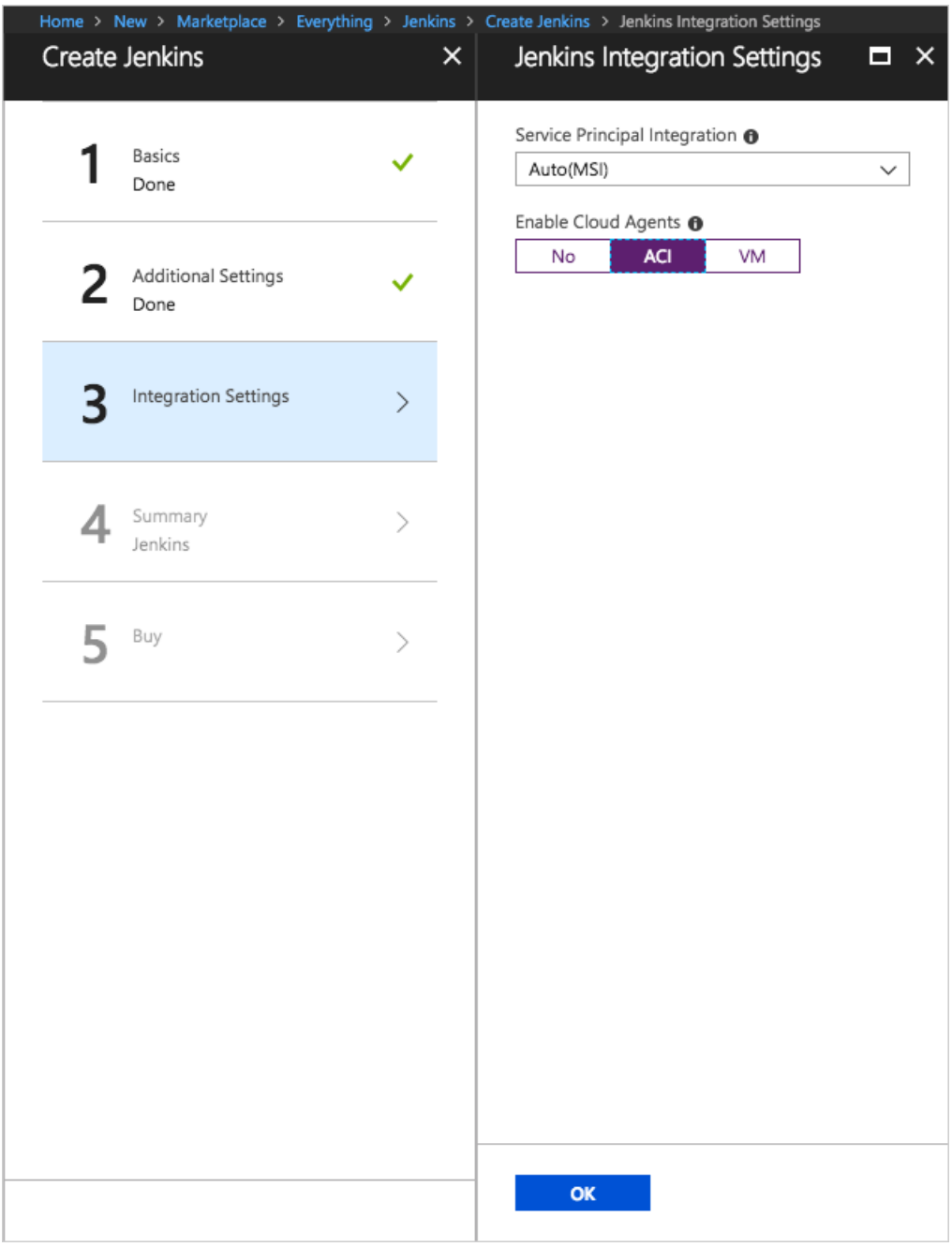
3. On the **Additional Settings** form, complete the following items:

   - **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**.




4. For service principal integration, select **Auto(MSI)** to have [Azure Managed Service Identity][managed-identities-azure-resources] automatically create an authentication identity for the Jenkins instance. Select **Manual** to provide your own service principal credentials.

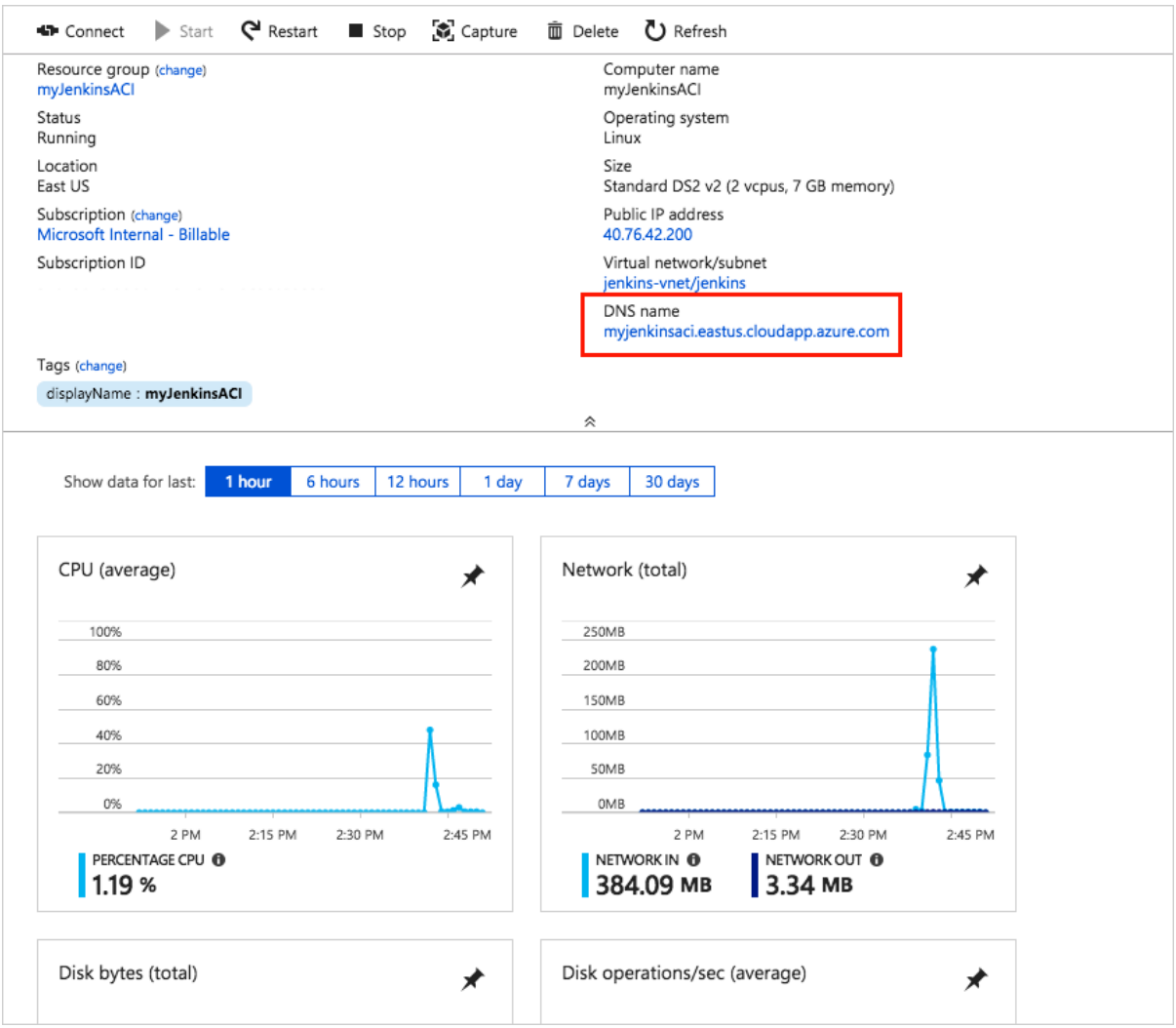
5. Cloud agents configure a cloud-based platform for Jenkins build jobs. For the sake of this article, select **ACI**. With the ACI cloud agent, each Jenkins build job is run in a container instance.




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.



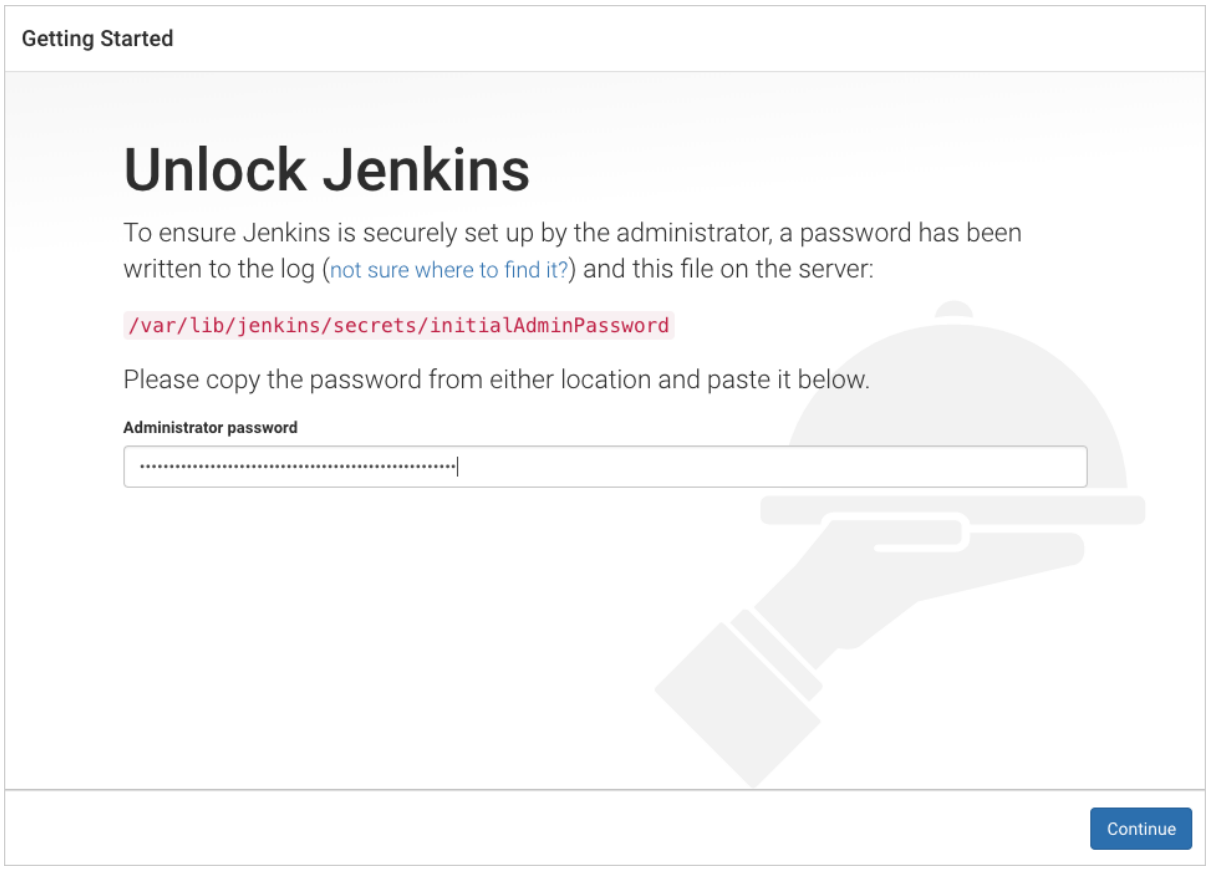

2. Browse to the DNS name of the Jenkins VM and copy the returned SSH string.

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.

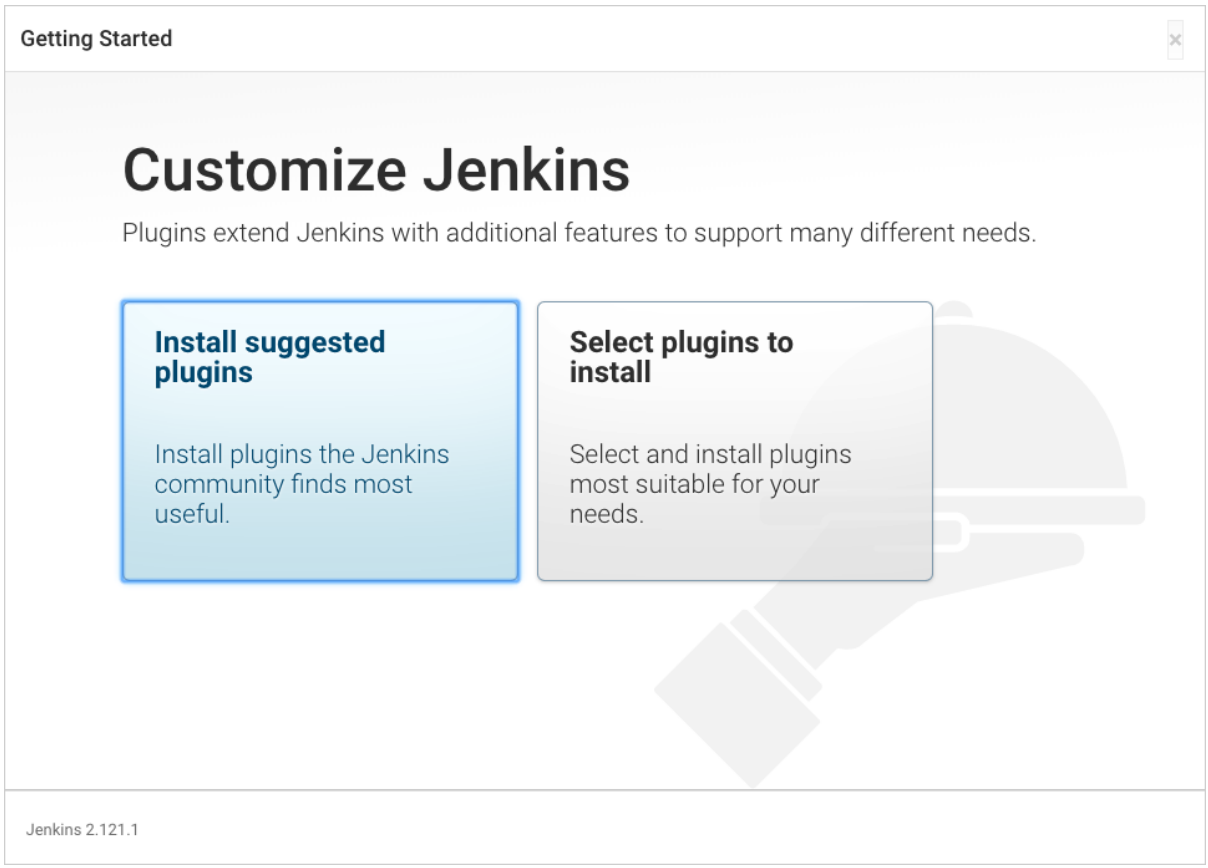
4. After the session is connected, run the following command to retrieve the initial admin password:

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

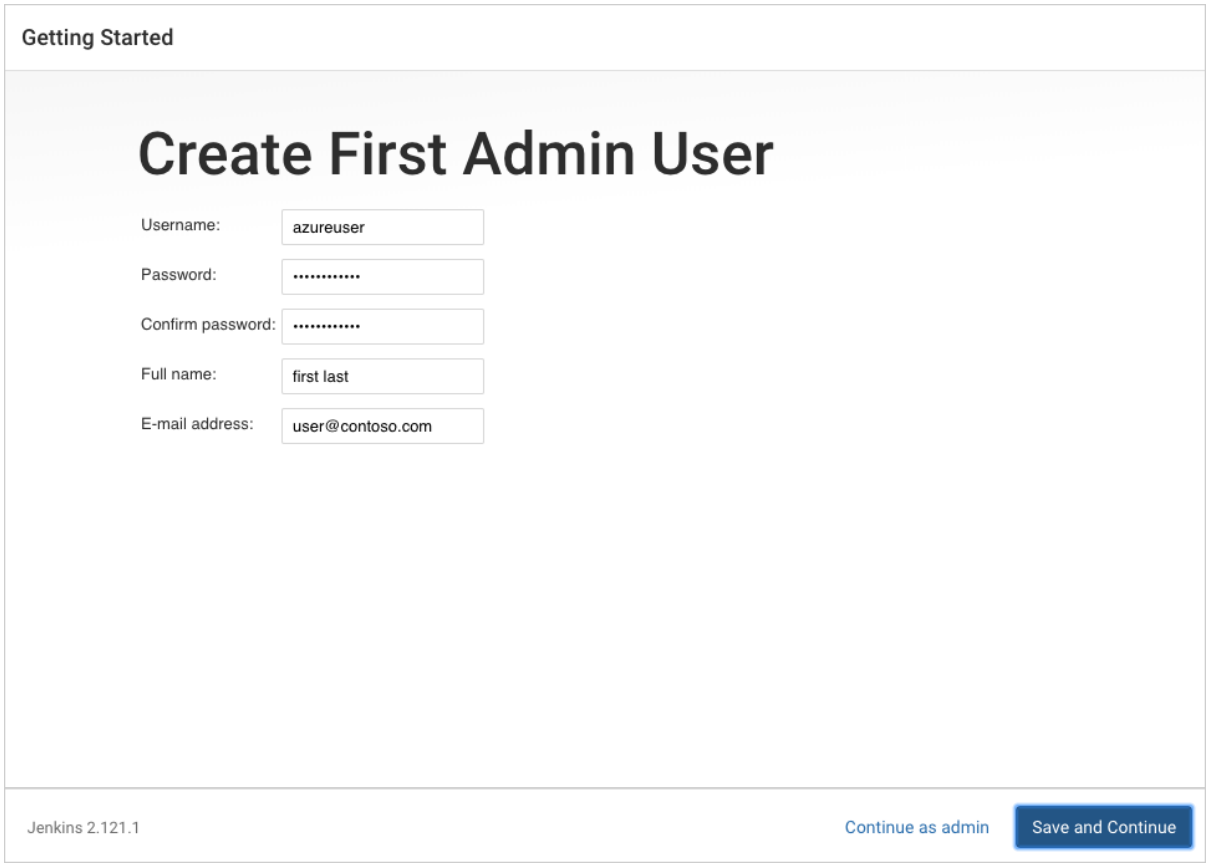
5. Leave the SSH session and tunnel running, and go to http://localhost:8080 in a browser. Paste the initial admin password into the box, and then select **Continue**.




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




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



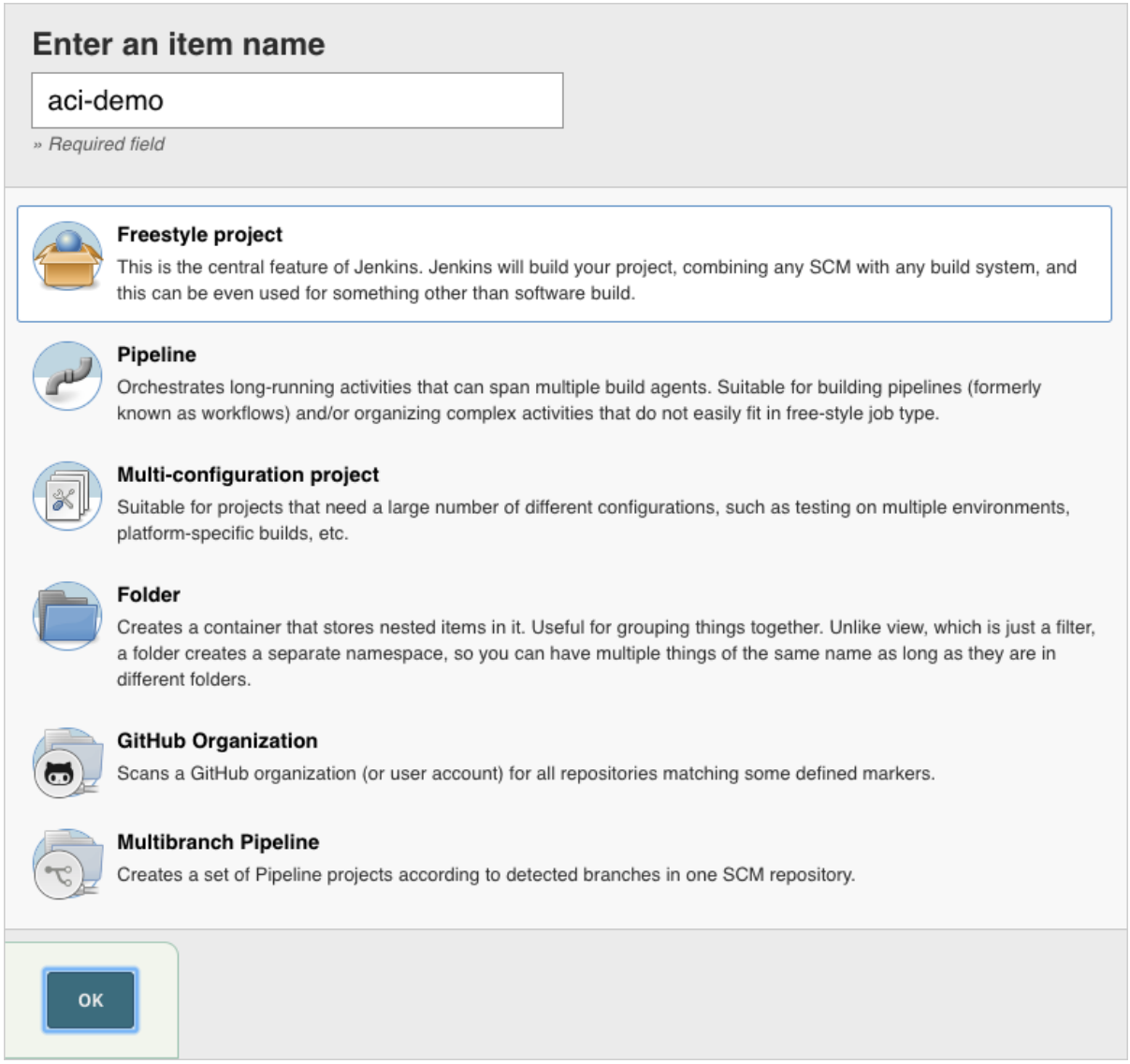

8. Select **Save and Finish**, and then select **Start using Jenkins** to complete the configuration.

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.

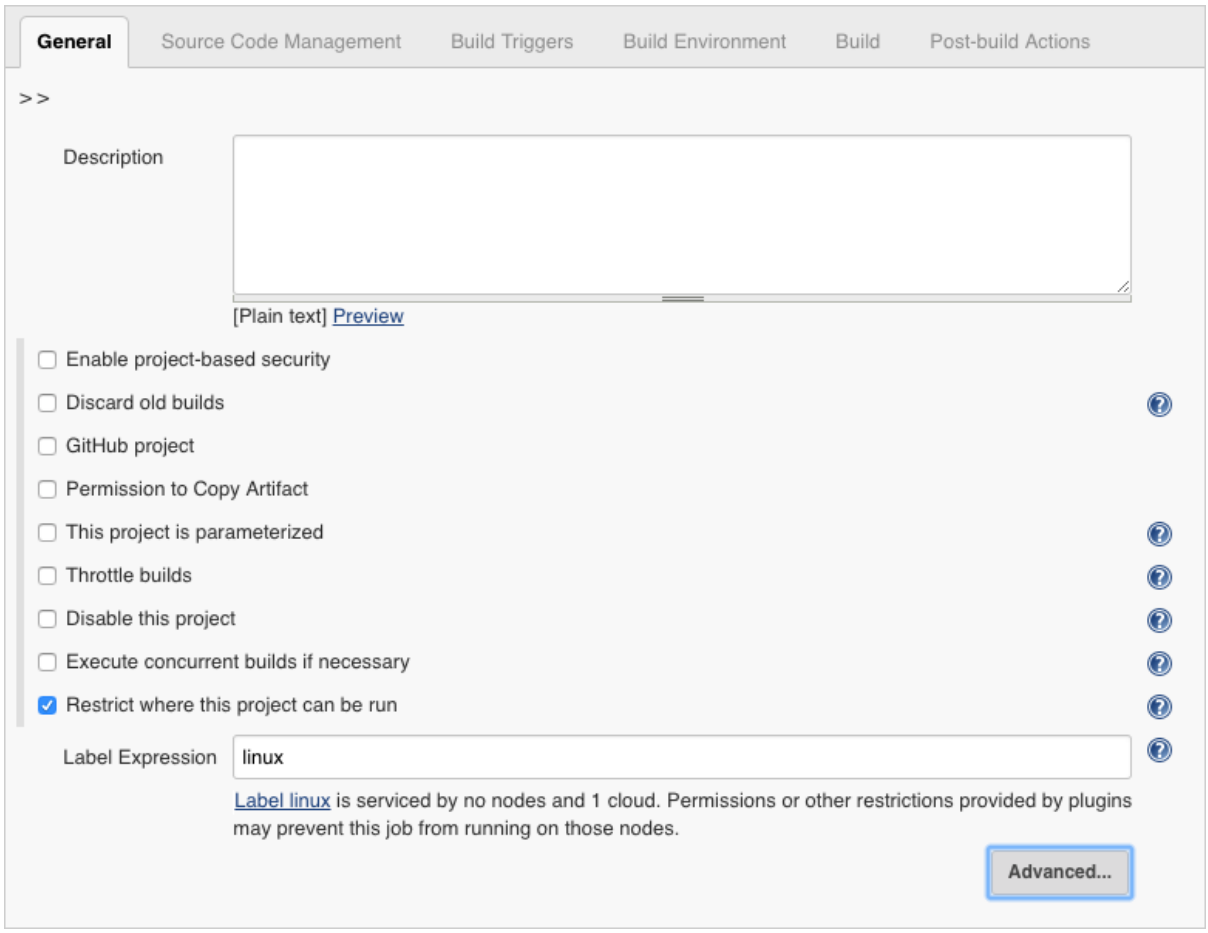
## Create a build job

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

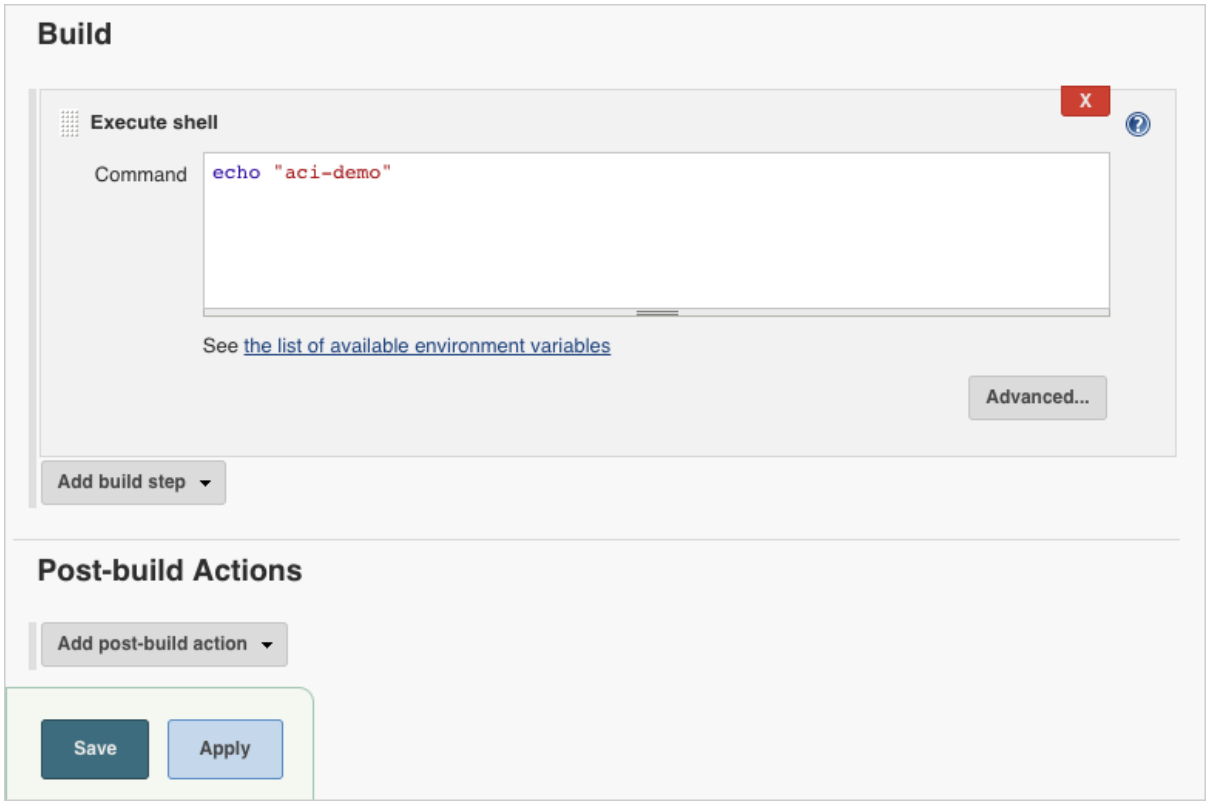
1. Select **New Item**, give the build project a name such as **aci-demo**, select **Freestyle project**, and select **OK**.




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.




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

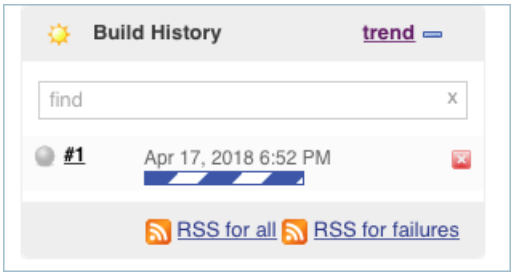



4. Select **Save**.

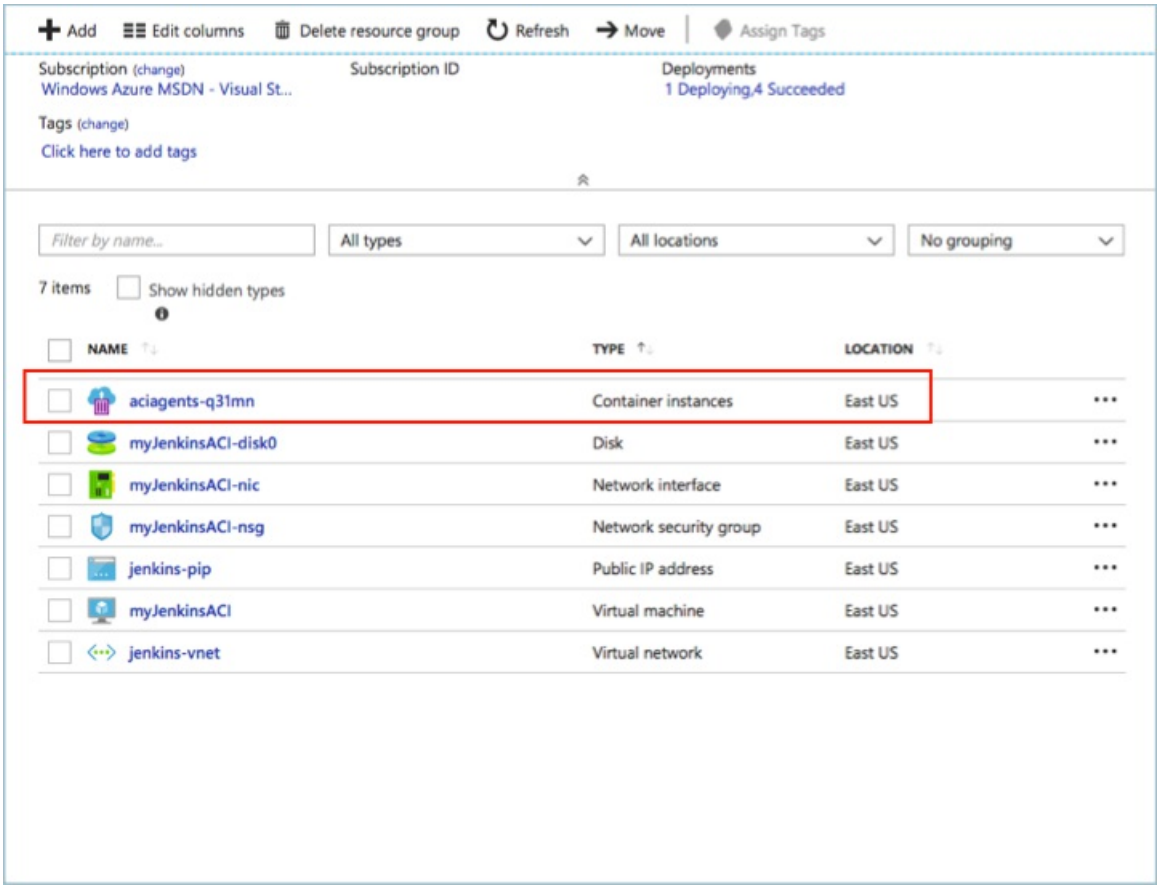
## Run the build job

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

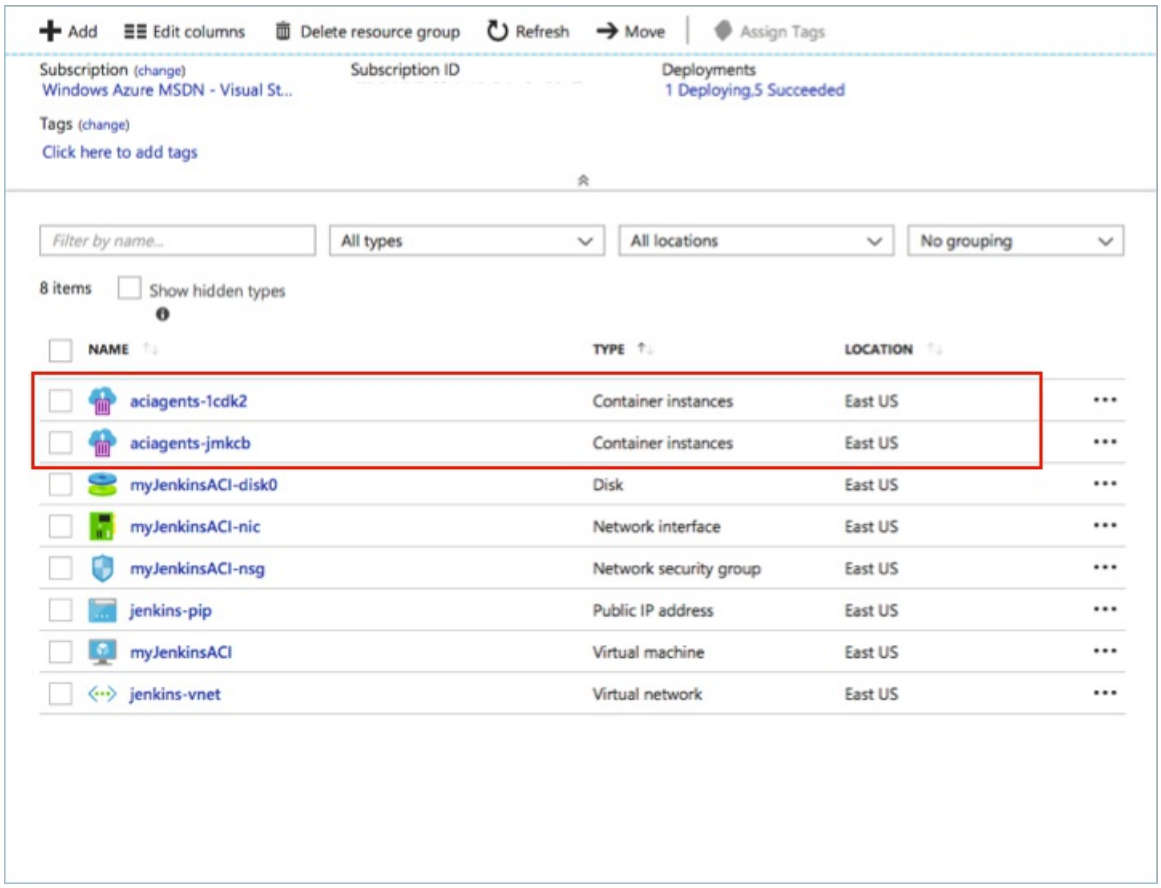
1. Select **Build Now** to start a build job. It takes a few minutes for the job to start. You should see a status that's similar to the following image:



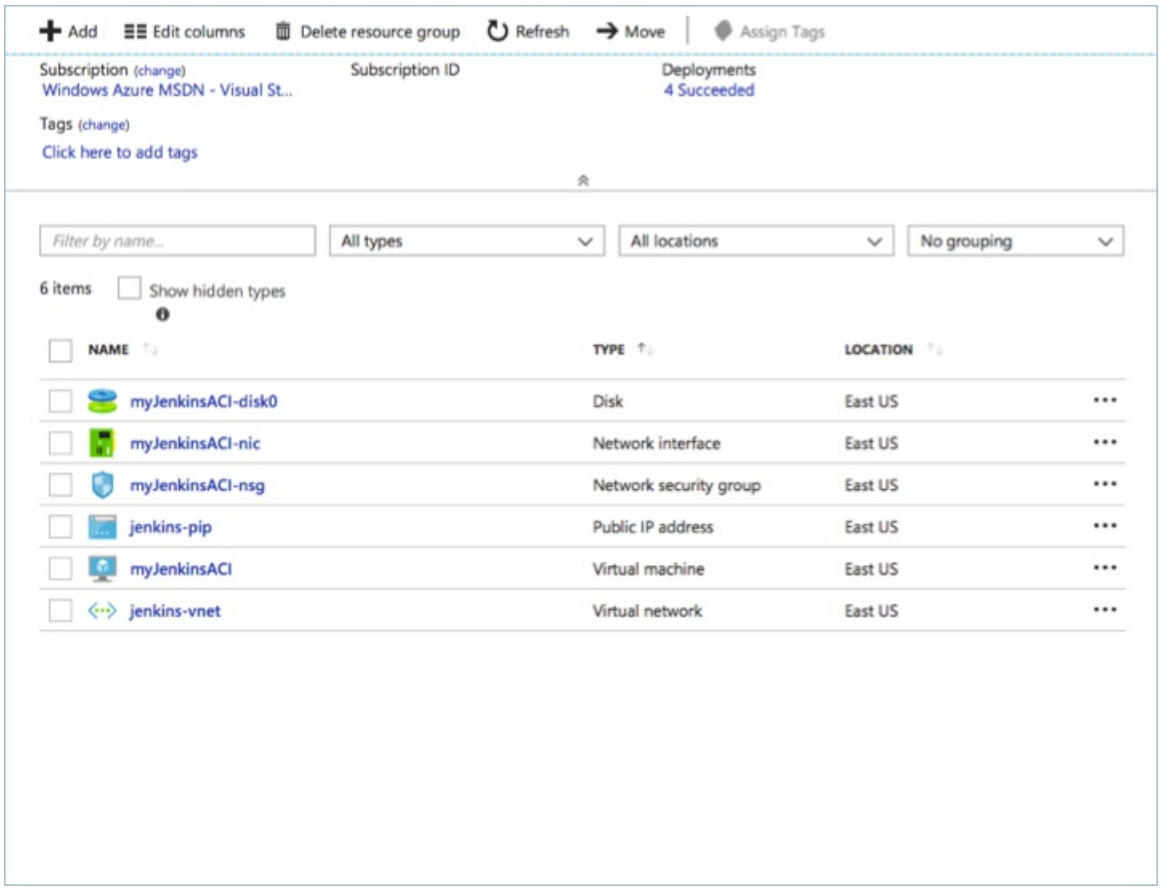

2. While the job is running, open the Azure portal and look at the Jenkins resource group. You should see that a container instance has been created. The Jenkins job is running inside this instance.




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




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




## 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.

# Mount an Azure file share in Azure Container Instances

8/2/2018 • 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 for use with Azure Container Instances.

> **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

# Export the connection string as an environment variable. The following 'az storage share create' command
# references this environment variable when creating the Azure file share.
export AZURE_STORAGE_CONNECTION_STRING=`az storage account show-connection-string --resource-group
$ACI_PERS_RESOURCE_GROUP --name $ACI_PERS_STORAGE_ACCOUNT_NAME --output tsv`

# Create the file share
az storage share create -n $ACI_PERS_SHARE_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 created with a random value at the end. To query the final string (including the random portion), use the following commands:

```
STORAGE_ACCOUNT=$(az storage account list --resource-group $ACI_PERS_RESOURCE_GROUP --query "[?
contains(name,'$ACI_PERS_STORAGE_ACCOUNT_NAME')].[name]" --output tsv)
echo $STORAGE_ACCOUNT
```

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
$STORAGE_ACCOUNT --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 microsoft/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 manage the 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.

First, 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 multi-container groups in Azure Container Instances.

## 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 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

10/8/2018 • 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 |
|---|---|---|
| `repository` | Yes | The full URL, including `http://` or `https://`, of the Git repository to be cloned. |
| `directory` | No | Directory into which the repository should be cloned. The path must not contain or start with "`..`". If you specify "`.`", 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. |
| `revision` | No | The commit hash of the revision to be cloned. If unspecified, the `HEAD` 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 aci-helloworld sample application into `/mnt/aci-helloworld` in the container instance:

```
az container create \
    --resource-group myResourceGroup \
    --name hellogitrepo \
    --image microsoft/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 DevOps Git repository, specify any user name (you can use "azuredevopsuser" as in the following example) in combination with a valid PAT:

```
--gitrepo-url
https://azuredevopsuser:abcdef1234fdsa4321abcdef@azuredevopsorganizationname.visualstudio.com/_git/some-
private-repository
```

For more information about personal access tokens for GitHub and Azure DevOps, see the following:

GitHub: Creating a personal access token for the command line

Azure DevOps: 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

# Mount a secret volume in Azure Container Instances

10/8/2018 • 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 microsoft/aci-helloworld \
    --secrets mysecret1="My first secret FOO" 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 -1 /mnt/secrets
mysecret1
mysecret2
/usr/src/app # cat /mnt/secrets/mysecret1
My first secret FOO
/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: microsoft/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: TXkgc2Vjb25kIHNlY3JldCBCQVIK
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": "TXkgc2Vjb25kIHNlY3JldCBCQVIK"
            }
          }
        ]
      }
    }
  ]
}
```

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.

# Retrieve container logs and events in Azure Container Instances

8/2/2018 • 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: 2018-03-09 23:21:33+00:00) pulling image "microsoft/aci-wordcount:latest"
(count: 1) (last timestamp: 2018-03-09 23:21:49+00:00) Successfully pulled image "microsoft/aci-
wordcount:latest"
(count: 1) (last timestamp: 2018-03-09 23:21:49+00:00) Created container with id
e495ad3e411f0570e1fd37c1e73b0e0962f185aa8a7c982ebd410ad63d238618
(count: 1) (last timestamp: 2018-03-09 23:21:49+00:00) Started container with id
e495ad3e411f0570e1fd37c1e73b0e0962f185aa8a7c982ebd410ad63d238618

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": "microsoft/aci-helloworld",
      ...
        "events": [
          {
            "count": 1,
            "firstTimestamp": "2017-12-21T22:50:49+00:00",
            "lastTimestamp": "2017-12-21T22:50:49+00:00",
            "message": "pulling image \"microsoft/aci-helloworld\"",
            "name": "Pulling",
            "type": "Normal"
          },
          {
            "count": 1,
            "firstTimestamp": "2017-12-21T22:50:59+00:00",
            "lastTimestamp": "2017-12-21T22:50:59+00:00",
            "message": "Successfully pulled image \"microsoft/aci-helloworld\"",
            "name": "Pulled",
            "type": "Normal"
          },
          {
            "count": 1,
            "firstTimestamp": "2017-12-21T22:50:59+00:00",
            "lastTimestamp": "2017-12-21T22:50:59+00:00",
            "message": "Created container with id
2677c7fd54478e5adf6f07e48fb71357d9d18bccebd4a91486113da7b863f91f",
            "name": "Created",
            "type": "Normal"
          },
          {
            "count": 1,
            "firstTimestamp": "2017-12-21T22:50:59+00:00",
            "lastTimestamp": "2017-12-21T22:50:59+00:00",
            "message": "Started container with id
2677c7fd54478e5adf6f07e48fb71357d9d18bccebd4a91486113da7b863f91f",
            "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.

# Troubleshoot common issues in Azure Container Instances

8/2/2018 • 6 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 | `<name>-<role>-CG<number>` | `web-batch-CG1` |
| Container name | 1-64 | Case insensitive | Alphanumeric, and hyphen anywhere except the first or last character | `<name>-<role>-CG<number>` | `web-batch-CG1` |
| Container ports | Between 1 and 65535 | Integer | Integer between 1 and 65535 | `<port-number>` | `443` |
| DNS name label | 5-63 | Case insensitive | Alphanumeric, and hyphen anywhere except the first or last character | `<name>` | `frontend-site1` |
| Environment variable | 1-63 | Case insensitive | Alphanumeric, and underscore (_) anywhere except the first or last character | `<name>` | `MY_VARIABLE` |
| Volume name | 5-63 | Case insensitive | Lowercase letters and numbers, and hyphens anywhere except the first or last character. Cannot contain two consecutive hyphens. | `<name>` | `batch-output-volume` |

## 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 supports Windows images based only on Long-Term Servicing Channel (LTSC) versions. To mitigate this issue when deploying Windows containers, always deploy LTSC-based images.

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 \"microsoft/aci-hellowrld\"",
    "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 \"microsoft/aci-hellowrld\": rpc error: code 2 desc Error: image t/aci-
hellowrld: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 \"microsoft/aci-hellowrld\"",
    "name": "BackOff",
    "type": "Normal"
  }
],
```

## Container continually exits and restarts

If your container runs to completion and automatically restarts, you might need to set a restart policy of **OnFailure** or **Never**. If you specify **OnFailure** and still see continual restarts, there might be an issue with the application or script executed in your container.

The Container Instances API 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 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
microsoft/aci-helloworld    latest    7f78509b568e    13 days ago    68.1MB
```

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 certain Windows images.

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 2016 (LTS only)
- Windows Server 2016 Nano Server

### Windows containers slow network readiness

Windows containers may incur no inbound or outbound connectivity for up to 5 seconds on initial creation. 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 Quotas and 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.

## Next steps

Learn how to retrieve container logs & events to help debug your containers.