# AZ-104T00A – Administer PaaS Compute Options

Good day everyone. Last time we discussed about Virtual Machine, one of the most important pillars of basically every Azure services. Today we are going to talk about PaaS options in Azure, Azure App Services, Container Instances and Azure Kubernetes Service. Let’s start with Azure App Service Plan.

## Configure Azure App Service Plans

An App Service plan defines a set of compute resources for a web application to run. The compute resources are analogous to a server farm in conventional web hosting. One or more applications can be configured to run on the same computing resources (or in the same App Service plan).

## Determine App Service Plan Pricing

The three main settings you have to choose in an App Service Plan are the operating system, the region, and the pricing tier. The operating system can be either Windows or Linux. For the region, you typically choose the one that’s closest to where most of your users are located, such as Switzerland North. Bear in mind that you can’t change a service plan’s region. So, if you decide to move an app to another region later, you’ll have to make a copy of it and put it in a new App Service Plan. Those first two settings are usually pretty easy to choose, but the pricing tier is much more complex. This is where you decide what type of compute resources to allocate and what App Service features you want. This table shows a summary of the different tiers, although it doesn’t actually show all of the options and features in each tier. I’ll walk you through some of the highlights.

**Free Tier**

If you’ll be using this plan for development and testing, then you can probably get away with using a low-powered and inexpensive option. The Free tier is a good choice as long as you only need 60 CPU minutes per day. If you need more CPU time than that, then you’ll have to go with a plan that costs money.

**Shared Tier**

The next step up is the Shared tier, which is also known as the D1 tier. It gives you up to 240 CPU minutes per day. It’s called the Shared tier because you have to share compute resources with other Azure customers. That’s the case with the Free tier as well. Both of these tiers have just one gig of RAM and one gig of storage, so you can’t do too much with them. The Shared tier also allows you to use a custom domain name for your app’s website, which isn’t something you can do with the Free tier.

**Basic Tier**

If you need more resources, then you can go with the Basic tier, which puts your apps on dedicated virtual machines instead of shared VMs. There are several options that vary based on the number of virtual CPU cores and the amount of memory. All of the options come with 10 gig of storage.

**Standard Tier**

The next step up is the Standard tier, but although it has 50 gig of storage instead of 10 gig, the CPU and memory options are exactly the same as the Basic tier. The big difference is that the Standard tier supports autoscaling, which means it can automatically add more virtual machines to handle increased traffic. It lets you use up to 10 VM instances. That’s significantly more than the Basic tier, which has a maximum of 3 VM instances.

**Premium Tier**

The Premium tier has options for more CPU, memory, and storage, and it lets you use up to 30 VM instances in most Azure regions.

**Isolated Tier**

The Isolated tier gives you a private, dedicated environment that’s in its own virtual network. While the Standard and Premium tiers allow you to connect your dedicated VMs to a virtual network, the Isolated tier actually puts your VMs in a private virtual network so they’re isolated from other App Service instances. It lets you have up to 100 VM instances.

## Scale Up and Scale Out the App Service Plan

Regardless of which pricing tier you choose, it’s possible to put multiple apps in your service plan. But all of those apps will share the same compute resources, so if you put too many apps in the same service plan, then you’ll run into performance issues. One potential way to resolve this problem would be to add more virtual machine instances up to the limit of the number of instances allowed in the pricing tier you’ve chosen. This is known as scaling out. If scaling out doesn’t solve the problem, then you can switch your plan to a pricing tier that has more powerful instances and/or a higher limit on the number of instances you can have. This is called scaling up, and it’s very easy to do. You can simply select the new pricing tier, and it will only take seconds to go into effect. When App Service scales your app to a new tier, it will take steps to ensure your application remains available during the scaling process. It will first allocate one or more VMs of the selected size and it will then copy any apps you have in the App Service plan to the new VMs. While this happens, the App Service front-end will continue to send any incoming requests to your current VMs. Only when your apps are on the new VMs and ready to process requests will the front-end start sending new requests to the new VMs. That might seem like it would take a bit of time, but in fact, App Service can perform all of these steps quickly.

## Configure App Service Plan Scaling

If you need to add more resources to an app running on App Service, one way is to scale up its resources by switching its App Service Plan to a higher pricing tier. This is very easy to do, and it only takes a few seconds, but it’s not a very dynamic solution. For example, what if your app gets used a lot more on weekdays than on weekends? You wouldn’t want to scale up and scale down your app’s service plan every week. A more elegant solution is to have Azure add more virtual machines when your app is busier and remove VMs when it’s less busy. This is known as scaling out and scaling in rather than scaling up and scaling down. It’s only available on Basic plans and higher. It’s not available on Free or Shared plans. Keep in mind, though, that even the Basic tier probably won’t give you what you need. That’s because it only allows you to scale in and out manually. If you want Azure to scale without manual intervention, then you need autoscaling, which is only available on Standard plans and higher. The service that actually handles autoscaling is Azure Monitor. So, you can get to the autoscale settings either by selecting “Scale out” from the menu in your Service Plan or by going to Azure Monitor and selecting “Autoscale” and then selecting your Service Plan. The way it works is that you can create one or more scale conditions that specify exactly what will trigger Azure to add or remove virtual machines. In each scale condition, you need to add one or more rules. For example, you could say that if the average CPU percentage of the VMs in your Service Plan is over 75% for at least 10 minutes, then it should increase the number of VM instances by one. There are a huge number of variations in the rules that you can create. First, there are different metrics to choose from. Aside from CPU percentage, there’s also memory percentage, data in, data out, disk queue length, and a wide variety of network metrics, such as “Socket count for inbound requests”. You can also tell it to look for the minimum or maximum value of a metric rather than the average. For example, you could tell it to scale out if the CPU percentage reaches 75% at any time during a 10-minute period. It doesn’t have to be a 10-minute period either. You can set it to aggregate the metric over any number of minutes you want. And, of course, it doesn’t need to be 75%. You can set that to whatever percentage you want, too. Then there’s the scaling action that you want it to take. You can tell it to scale out by exactly a certain number of VM instances, such as 1 in our example, or scale out to a total number of instances, such as 3, or you can tell it to scale out by a percentage. For example, you can tell it to scale out the number of instances by 50% if the condition is met. You can also set something called a “cool down” period. This is the number of minutes to wait after a scaling operation before it can scale again. By default, it’s set to 5 minutes. This gives the metrics a chance to stabilize again after the scaling operation. For example, suppose you have a rule saying that if the average CPU percentage is over 70% for 5 minutes, then add an instance. Then suppose you have 2 instances, and the average CPU percentage reaches 90%, which causes it to add another instance. If you didn’t have a cool-down period, the CPU percentage might still be above 70% for a little while until the new instance has spun up and taken some load off of the other instances, which could trigger the unnecessary addition of yet another instance. So far, I’ve only mentioned scaling out, but you can create rules for scaling in as well. They work the same way except that everything is reversed. For example, you could tell it to scale in by one instance if the average CPU percentage is below 25% during a 10-minute period.

Now, things really get interesting when you have multiple rules. For example, suppose that you have these rules:

* Scale out by one instance if CPU utilization is above 65 percent
* Scale out by one instance if disk queue length reaches more than 1700
* Scale in by one instance if CPU utilization drops below 35 percent

Now suppose that your instances are at 30 percent CPU utilization, and the disk queue length is 1900 messages. Will it scale in or scale out or stay the same? The answer isn’t obvious because there are conflicting rules. The CPU utilization is below 35 percent, so maybe it will scale in by one instance. But the disk queue length is more than 1700, so maybe it will scale out by one instance. Do the two rules cancel each other out, or does one of them have priority? Well, Azure autoscaling has a very sensible policy in these situations. Scale-out rules always win over scale-in rules. That’s because if any aspect of the system needs more resources, then it should scale out. Scaling in would just make performance worse. Okay, that was a lot of information, but believe it or not, there are a couple of other important autoscaling options that I haven’t mentioned yet: scaling to a specific instance count and scheduling. Previously, I showed scaling based on a metric, such as CPU percentage, but it’s also possible to create a scale condition that doesn’t involve metrics. To do this, you select “Scale to a specific instance count” instead of “Scale based on a metric” when you create a scale condition. This is much simpler than the metric-based process. You just tell it the number of instances you want it to scale to. Of course, this doesn’t really sound like autoscaling, does it? Without a metric, how will it know when to scale to a specific instance? That’s where scheduling comes in. You simply set a start time and an end time for when this condition should be in effect. If you want it to happen on a recurring basis, you can select “Repeat specific days” and tell it which days you want it to happen every week. For example, you could scale out to a specific instance count on weekdays. It’s also possible to use a schedule for a metric-based condition. For example, you could configure it to scale based on CPU percentage only on weekends but scale out to a specific instance count during weekdays. In this example, you’d have both metric-based and specific instance-based conditions in the same Service Plan, but since you scheduled them for different days, there wouldn’t be a conflict.

## Configure Azure App Services

When you need to host a website on Azure, there are lots of potential choices. If it’s a static website that doesn’t require any interactions with users, then you can just store your website in Azure Storage and serve it from there. If you need to host something more sophisticated, then you could deploy it on an Azure virtual machine that’s running a content management system, such as WordPress. If you’ve built a microservices-based application in Docker containers, then you could run it on Azure Kubernetes Service. But the most popular way to do it is to use Azure App Service. Why? Well, because App Service tries to make deploying a web application as easy as possible while still providing lots of flexibility and features. It’s considered to be a platform-as-a-service rather than infrastructure-as-a-service because it manages the underlying infrastructure for you. This means you don’t have to worry about things like patching the operating system.

## Implement Azure App Service

App Service supports a wide variety of programming languages and frameworks, including ASP.NET, ASP.NET Core, Java, Ruby, Node.js, PHP, and Python. Not only do these come preinstalled, but they’re also updated when new patches or releases are available. For most of these frameworks, you can choose whether to run it on Windows or Linux (although that’s not the case with ASP.NET Framework 4.8, which can only run on Windows). Even if you want to use a different programming language, you can still use App Service if you put your application in a Docker container that has the right dependencies installed in it. There are other key development and deployment features, too. First, it’s well-integrated with Visual Studio and Visual Studio Code, which are the preferred programming environments for many software developers. It’s also integrated with continuous integration / continuous delivery tools, such as Azure DevOps, GitHub, and BitBucket. You can also use App Service to host mobile backends and application programming interfaces. Aside from the ease of deployment and maintenance, one of the biggest advantages of using App Service is its enterprise-class features. First, there’s high availability. Microsoft guarantees 99.95% uptime in most cases for apps running on this service. Then there’s scalability. You can configure it to automatically add more resources to your application when demand increases. Of course, security is important, too, so Microsoft provides authentication capabilities if you only want authorized users to be able to access your apps. It also lets you control which IP addresses are allowed or denied access to your site if you want.

## Create an App Service

So far, you’ve seen how to create an App Service plan. Now let’s look at how you can create an App Service app that runs in that App Service plan. To create a new web app, search for Web App in the Azure Marketplace. When creating your web app, you’ll specify a name for the app. You have the option of creating your app using your own code or to run a Docker container in your App Service plan. If you choose **Code**, you can select from a wide range of runtime stacks, including .NET, Java, Node, PHP, Ruby, and Python. I’ve selected my existing App Service plan for this web app, but you can also create a new App Service plan by clicking **Create New**. Once everything is configured the way you want it, click **Review + Create** to create the web app. Once the web app has been created, you can access it by browsing to [https://web\_app\_name.azurewebsites.net.](https://web_app_name.azurewebsites.net/) If you don’t see an existing App Service plan when creating a new web app, make sure you have selected the OS that matches the App Service plan’s OS. You also need to ensure the region you select is the region where the App Service plan is deployed.

## Demo Azure App Service

*dotnet new webapp -n DemoAppServiceApp -f net6.0 && cd DemoAppServiceApp*

az webapp up –resource-group “Demo” –plan “demo-plan01” –name demoazwebnetapp01 –location switzerlandnorth

**Modify index.cshtml to add “Welcome to Azure” in the body**

*az webapp up*

## Create Deployment Slots

The Azure portal provides out-of-the-box continuous integration and deployment with Azure DevOps, GitHub, Bitbucket, FTP, or a local Git repository on your development machine. You can connect your web app with any of the above sources and App Service handles the rest for you. App Service auto-synchronizes your code and any future changes to the code into your web app. With Azure DevOps, you can also define your own build and release process. Compile your source code, run tests, and build and deploy the release into your web app every time you commit the code. All of the operations happen implicitly without any need for human administration. App Service makes it easy to deploy a new version of an app to production. All you have to do is run the “az webapp up” command as we see in the demo. But you’d normally want to test a new version thoroughly before putting it into production. So, you need a testing environment that’s as close as possible to your production environment. App Service has a great solution. It’s called a deployment slot. By default, an app has one deployment slot called “Production”. But you can create other deployment slots, too, and give them names like “testing” and “staging”. Then when you deploy an app, you can choose which deployment slot to put it in. Here’s a typical way to do it. First, you clone your production slot to a staging slot. When you do this, you can choose to clone the production slot’s configuration, including app settings, connection strings, the language framework, and many other settings. That way your staging slot’s configuration will be as close as possible to your production slot’s configuration, which will make your testing more valid. Once you’ve cloned the slot to a new one, though, you can change any of those settings in the new slot if you need to. For example, you might change the version of the language framework to a newer version, such as going from Java 8 to Java 11. You might expect it to clone the actual app as well, and not just the settings, but it doesn’t. That’s because it expects you to upload a new version of the app to the staging slot, so there’s no point in cloning the production app. Note that if you use the command line to upload the app to the staging slot, you can’t use the “az webapp up” command. Instead, you have to use the “az webapp deployment source” command, which takes a bit more work. The most common deployment source is a git repository. Once you’ve tested the new version in your staging slot, the next step is to swap the staging slot with the production slot. You start by setting the source slot and the target slot. The source slot contains the new version of the app, so it’s the staging slot. The target slot is the one that will be running the new version once the swap has been completed, so it’s the production slot. When you tell it to swap the source and target slots, it swaps not only the two different versions of your app but also any settings that you changed in the staging slot, such as the language framework if you changed that. However, there is an exception to this. You can create slot-specific settings. For example, it’s usually a good idea to have a separate test database so you don’t mess up the production one when you’re testing a new version of your app in the staging slot. In this case, you’d need to have a different connection string for the database in each slot, and you wouldn’t want this setting to be swapped. It would need to stay with the slot rather than the app, which is why it’s called slot-specific. So, when you add or change an application setting or a connection string in a non-production slot, you have the option of making it a slot-specific setting. Aside from giving you the ability to properly test a new version before putting it into production, there are a couple of other big advantages to using a swap to deploy an app. First, when you deploy a new version of an app, it will often cause a reboot of the virtual machines that are used to host the app. During the warm-up period when the [VM instances](https://cloudacademy.com/course/getting-started-with-azure-virtual-machines-988/what-is-a-virtual-machine/) are spinning up, the app is not available, so if you were to simply deploy the new version of the app to production, it would cause downtime. But if you deploy to a staging slot first, then the VM instances have a chance to warm up before you swap the slot with production. Second, if there’s a problem after you swap the new version into production, you can easily roll back and revert to the previous version. All you have to do is swap the two slots again. If you’ve ever had to quickly roll back a new version that you’ve deployed, then I’m sure you can appreciate how helpful this feature is. There’s also an option that makes the swap even safer if you have slot-specific settings. It’s called “swap with preview”. This does the swap in two phases. First, it temporarily copies the slot-specific settings from the target slot (that is, the production slot) to the source slot (that is, the staging slot). It then gives you a chance to test the new version of the app with all of the settings it will have after the swap is completed. If it doesn’t work, then you can cancel the swap. If it does work, then you can tell it to complete the swap. At that point, it swaps the app and the other settings between the two slots. But it leaves the slot-specific settings with each slot. So, the version of the app that was previously in production and is now in the staging slot has the connection string for the test database, for example. There’s also a riskier option for doing a swap. If you enable “auto swap” on a slot, then every time you push a new version of your app to that slot, it will automatically swap it with whichever slot you specify. So, for example, if you enable “auto swap” on the staging slot, and you set the “Auto swap deployment slot” to the Production slot, then every time you upload a new version of the app to the staging slot, it will automatically be swapped into Production. Why on earth would you do such a thing? Well, you might want to do this as part of an [automated DevOps process](https://cloudacademy.com/course/implementing-a-build-strategy-for-continuous-integration-with-azure-devops/introduction/). If you have a continuous integration system that automatically tests your app every time you commit new changes to the source code repository, then you might feel comfortable putting new versions into production without doing any manual testing. So, if that’s the case, then why bother pushing it to the staging slot first and then auto-swapping it into production? Why not just push it straight to production? Well, as I mentioned earlier, when you push an app to the staging slot first, it has a chance to warm up before going into production. In fact, it’s even possible to create a custom warm-up. If your app needs to be initialized before it can serve requests from users, then you can create a custom warm-up that makes sure the initialization is complete before it performs the swap. There are two ways to set this up. If your app is running on an IIS web server, then you can configure the applicationInitialization element in the web.config file. Alternatively, regardless of the web server you’re using, you can set WEBSITE\_SWAP\_WARMUP\_PING\_PATH and WEBSITE\_SWAP\_WARMUP\_PING\_STATUSES in the app settings in [Azure App Service](https://cloudacademy.com/course/getting-started-with-azure-app-service-949/introduction/). So far, I’ve only talked about having two slots and having all of your end-user traffic go to the production slot. But you can have more complicated configurations if you want. First, you can have more than two slots. For example, you might want to have separate development, testing, staging, and production slots to handle the different stages of your software development process. Second, you can make things really interesting by sending some of your end-user traffic to non-production slots. For example, suppose you want to do canary testing, which means that you want to reduce the risk involved with deploying a new version of your app by releasing the new version to only a small percentage of your users. Then if those users don’t have any problems with it, you can release it to the rest of your users. App Service supports this type of testing by allowing you to set a different percentage of traffic for each slot. Normally, the production slot is set to take 100% of the traffic and all other slots are set to take 0%. But if you wanted to do a canary test, you could set the staging slot to 5% and the production slot to 95%. Then after you’ve done your testing, you could swap the two slots, which would put the staging version into production, and then set the production slot to take 100% of the traffic again. I should mention that the total number of slots you can use depends on the pricing tier of the App Service Plan you’re using. Deployment slots are only supported in the Standard, Premium, and Isolated tiers. The Standard tier supports up to 5 slots, but the Premium and Isolated tiers support up to 20. So, if you’re using the Standard tier, and for some reason, you need more than 5 slots, you can just scale up your service plan to the Premium tier.

## Secure an App Service

Azure App Service provides built-in authentication and authorization support. You can sign in users and access data by writing minimal or no code in your web app, API, and mobile backend, and also your Azure Functions apps. Secure authentication and authorization require deep understanding of security, including federation, encryption, JSON web tokens (JWT) management, grant types, and so on. App Service provides these utilities so you can spend more time and energy on providing business value to your customer. As is typical with a PaaS service, App Service makes it simple to implement security for your web app. You can secure you app with Azure Active Directory, and you can also easily implement security using Facebook, Google, and Twitter so that users can authenticate to your app using their existing logins.on-premises. To configure authentication, click the Authentication menu item after opening your web app in the Azure portal. When you first do this, you’ll need to add an identity provider. For example, you can use Facebook as identity provider. The App ID and App Secret are from the Facebook developer’s site. The **Authentication** setting has been set to **Require Authentication**, thereby preventing users from browsing my web app without logging in. Directly under that setting, unauthenticated requests are configured to redirect to Facebook so that users can log in. Once the settings are configured, click **Add** to add the provider. App Service uses OAUTH authentication when configuring a third-party identity provider. Secrets that you provide to configure the provider are securely stored in Azure Key Vault. You can configure multiple identity providers for your app. However, doing so will require you to create a page that presents users with a list of authentication providers and links to sign in.

## Create a Custom Domain

As we saw before, you can browse to your app in App Service using *https://app\_name.azurewebsites.net*, but it’s likely you’ll want to use your own custom domain name. App Service makes using a custom domain easy. In order to map an existing custom domain to your web app, you’ll need to go through a series of steps, some of which are carried out at your domain registrar’s website. Therefore, some of the steps may differ slightly depending on your provider.

## Backup an App Service

App Service provides easy backup and restore of your apps. These backups can be created manually, or they can be scheduled on a regular basis. Backups can be retained for an indefinite amount of time. When you back up your app, App Service can back up not only the app’s content and configuration, but it can also back up SQL Database, Azure Database for MySQL, Azure Database for PostreSQL, and MySQL in-app databases. App Service backups are stored in Azure storage, and each backup is a complete copy of the app. Backups are not incremental. You can manually back up your app by clicking on the Backups menu option and clicking on Backup. Note that if you haven’t already configured the storage account for your backups, you’ll first need to do that. Once you’ve created a backup, you can restore the backup by clicking **Restore**. You can choose to overwrite your existing app or restore the backup to a new app. If you’d prefer, you can create a scheduled backup so that App Service backs up your app automatically at configured times. App Service backups cannot exceed 10 GB, and that includes the app’s content and any databases. It’s also important to remember that you have to be running your App Service plan in the Standard tier or higher. Backup is not available for lower tiers.

## Demonstration – Create an App Service

Task 1: Create an Azure web app called demoappserv2111.

* Create a web app by using the Azure portal.
* The web app should run on **Windows** and use the **.net 6.0** runtime stack.

Task 2: Create a staging deployment slot.

* Verify there's a production deployment slot.
* Create a new staging deployment slot.

Task 3: Configure Web App deployment settings.

* Deploy your web app from a local Git session.
* Provide the authentication credentials.

Task 4: Deploy code to the staging deployment slot.

* Use Azure PowerShell to clone the remote repository ***git clone https://github.com/Azure-Samples/******dotnetcore-docs-hello-world*** and set the local path ***Set-Location -Path $HOME/dotnetcore-docs-hello-world***
* Add the remote Git session by using the authentication credentials. Git remote **url remote**
* **git push demoappserv2111 master**
* Display the default web page in a new browser tab.
* Push the sample web app code from the local repository to the Azure Web App staging deployment slot.

Task 5: Swap the staging slots.

* Swap the deployment slots.
* Verify the default web page is replaced with the Hello World page.

Task 6: Configure and test autoscaling of your Azure web app.

* Configure a custom autoscale rule on the production deployment slot.
* The scale rule should use the CPU percentage to increase the resource count.
* Use Azure PowerShell to start an infinite loop that sends the HTTP requests to your web app:

$rgName = “demo-app”

$webapp = Get-AzWebApp -ResourceGroupName $rgName

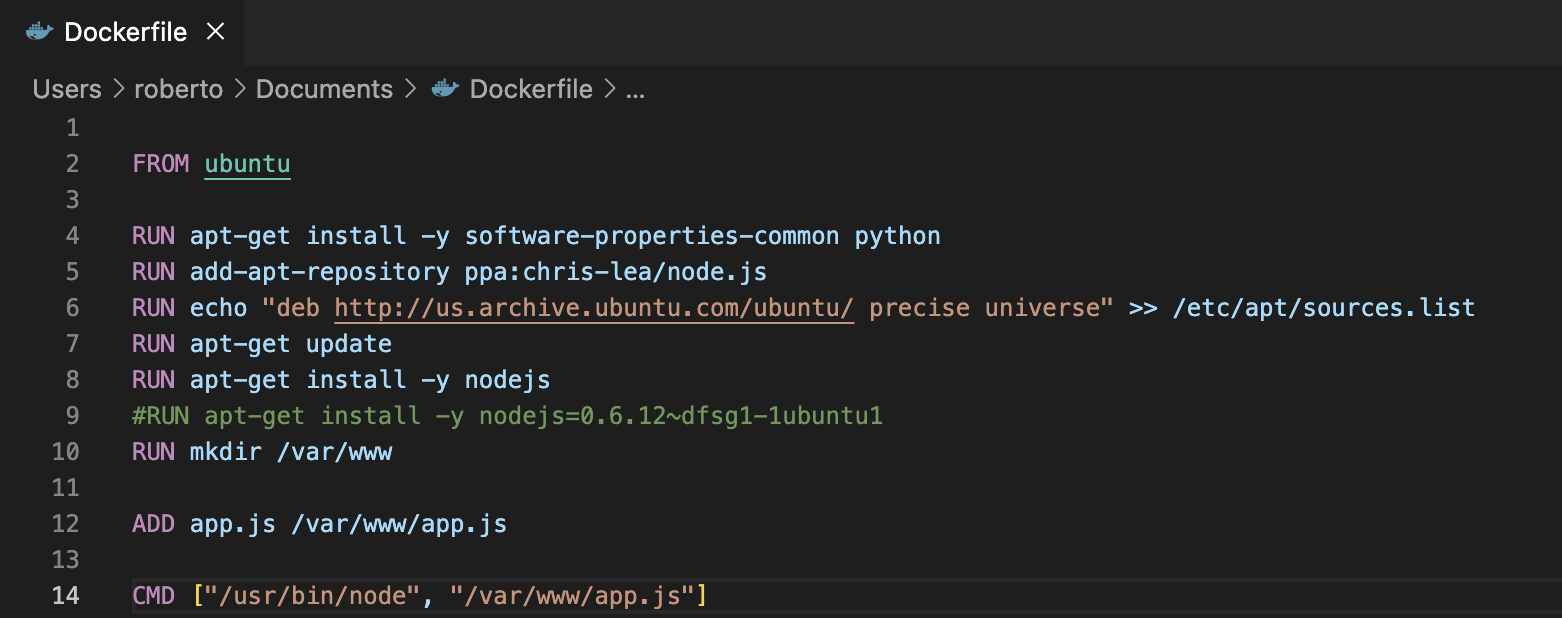
while ($true) {Invoke-WebRequest -uri $webapp.DefaultHostName }

* Confirm the resource count automatically scales.

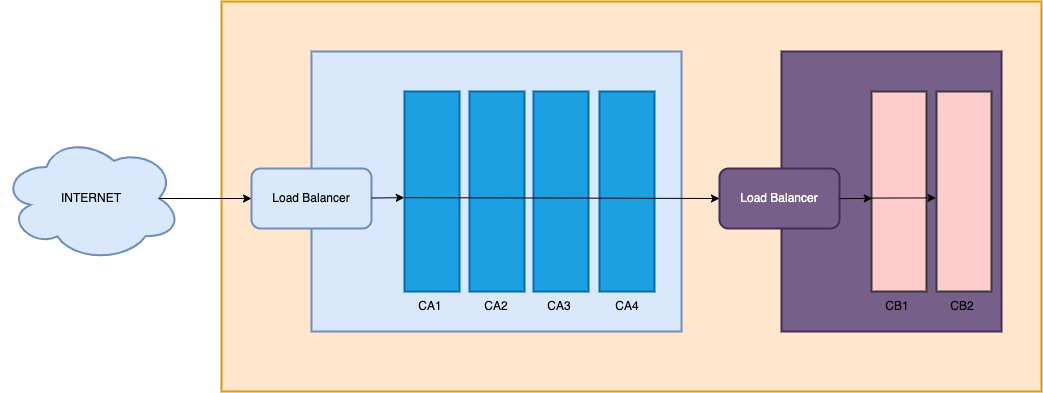
## Configure Azure Container Instances

Containers are a bit of a it thing in technology right now. The reason for this is simple, they're a very powerful tool which can streamline your development and ops processes, save your company money, and make life for your developers much easier. However, the flip side of this is that they're a new paradigm to understand and require that apps be built with a specific architecture to take full advantage of their features.

## Compare Containers to Virtual Machines

So, what is a container? One way to think about them is that they're lightweight virtual machines. With a virtual machine, you have to virtualize an entire operating system as well as the software you want to run. This makes VMs really resource heavy. The operating systems is often the single largest most resource-intensive piece of hardware on your computer and so running multiple OS on the same computer, just so that you can have separate environments, uses a lot of your resources. To overcome this issue, the Linux operating system began implementing containers. The idea is simple, if you're running a Linux OS on your computer already, why run a new OS for each VM?

Instead, you can use the core of the OS, called the kernel, for each VM. This way the VMs only run the software that they need to. The difficulty with this is that it is important that the VMs not be able to affect each other, or the underlying computer they're running on. And containers need to replicate this functionality. So, the Linux team had to implement some safety features into the kernel itself. Features such as being able to block off different parts of the kernel processor and memory for the different containers, so that the code running on one container can't accidentally access another container through the kernel. Now that these containers were implemented at the kernel level, any amount of software could be run inside of one and it would be like running it in its own VM, or own physical machine. And because all Linux distros share the same fundamental Linux kernel, you can easily run containers with different distros, just as easily as you can run containers using the same distro. The software that makes each distribution unique all runs on top the kernel and it's only the kernel that is shared across all the containers and the host OS. Once containers are implemented at the most basic fundamental part of the Linux OS, software which made it easier to implement these Linux containers begin to pop up. One of the first and most successful container software projects is called Docker. Docker makes it easy to define, manage and use Linux containers by simply writing plain text documents to define the software that you want running inside of a particular container. In addition, Docker and other companies began building software that could link containers together into a single app, as well as orchestrate spinning them up and down in the cloud rapidly. In addition to Docker, there are other container systems. But I'm mostly going to use Docker as the example of how containers work in this course, because it is the most frequently used and frankly the most easily explained of all these systems.

As an example, a Dockerfile is used to define a container. This Dockerfile starts with a standard image, usually provided by a software project, such as, a Linux distro or web technology like node.js. From there you can add new pieces to that image in a certain order, usually by running commands telling the image to install and setup new software. Once the file is written and saved, it can be sent in plain text to other people and built in just a few seconds on any computer that has Docker installed. This is very different from VMs, where you have to send a multi-gigabyte executable file to other people who want to run the VM. Here you're just sending a few kilobytes of instructions on how to build the container yourself. Docker builds these images in layers, for instance, if you were to run a Docker build on this Docker file, first Docker would take the official Ubuntu image, then it would run apt-get install-y software-properties-common-python to create a new image, and then from that image, it would run add-apt-repository etcetera, to create another image, and so on, and so on. Once the Docker file is built, you can run the image inside a container, or copy it to multiple times, to run it in as many containers as you want.

For instance, here there are four instances of container A and two instances of container B. Further software can be used to network containers to each other, the same way VMs or physical machines can be networked together. So that your containers can communicate with each other to create one large system built with many small containers. Here we have a fairly standard networking arrangement, where all the containers live in one VPN, and load balancers direct the traffic in into each subnet towards the least used container at the moment. The details of how this is implemented may change depending on what sort of system your containers are running on. And we'll dive deeper into the specifics in a later lesson. But this should give you an idea of how to use them. But that's just on Linux. What if you want to run containers on another operating system? While Docker lets you run Linux containers on Mac or Windows by first starting a really lightweight Linux VM that mostly just runs the kernel, and then running all the other containers inside that VM. So, this is slower than running Linux containers on the Linux, because you do have a VM, but it's faster than the old paradigm of using a bunch of VMs, because you're only running one, and you get the other benefits of containers along with it. In addition, Microsoft has been working to build Windows containers. These are containers that are built into the Windows operating systems, so that instead of running a Linux distro and a container, you can run Windows in Windows software in a container. Windows has been working really closely with Docker on this project, so they work with Docker. However, running a Windows container on Linux or Mac doesn't really work at this point. Finally, there's currently no way to run Mac OS containers. It's just not something Apple has implemented in the Mac OS kernel, and since Mac OS is almost never used to run servers, no one is really asking for it. There are container systems for other operating systems, like BSD, but that's a bit out of scope for this course, since they're rarely used commercially. So that's what containers are. A way to run multiple computers on a single machine. Each with a different operating system software installed, in the fast and secure manner. In the next lesson we'll talk about why you'd want to use containers. And then show you how to containerize a app.

## Explore Azure Container Instance Benefits

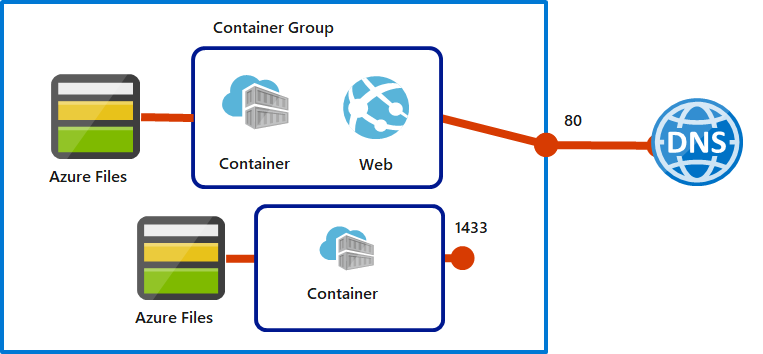
The thing with containers is that their benefits do not stop at being faster than VMs; they start there. A development team might indeed even work faster and more efficiently once they're comfortable with using containers. This is because containers can streamline and even make possible various development and operations tasks, such as testing, DevOps, and continuous deployment. I've seen it argued very well that the biggest benefit to containers is not the security or speed of your app, but rather, the speed with which you can build, test and deploy new features when using them. So, we'll talk about how containers can help you, both from a technological standpoint, but also from a management perspective. The first thing that containers enable is fast testing. Containers can be built and spun up very quickly, and code inside a container will run the same pretty much no matter what computer it's running on. As such, if you build an app with containers, every time you check in code, you can quickly build a new container with that code and play around with it. If you run your app in containers in production, then the containerized app that a developer has on their computer is exactly the same as the app that will run on the cloud. Gone are the days of each developer managing the dependencies on their computer to ensure that the app that they're working on can run locally. This quick feedback loop for your developers can lead to better code being written, and for less need for troubleshooting when strange bugs pop up that are caused by the developer having a slightly different version of the software on their computer than that you're running in production. In addition, this makes it easier for others to get up and running with your code as well. For instance, an open-source project might include a Docker file as part of their GitHub repo. This way, anyone who downloads the code to their machine in order to edit it can get a version up and running to test their changes with just a few Docker commands, whereas before, the developer would have to set up a new VM on their machine which followed a number of instructions on how to get exactly the software that the open-source project needs to run. Now they only need to run one or two commands and the rest is done automatically by Docker. Containers are also really useful when it comes to the DevOps process. DevOps is a way of doing development work where the developer and IT teams work closely together, and as much of the process is automated as possible. For instance, with continuous deployment, any accepted change to the main code base is deployed to the users, and this process is made much easier with containers. Container systems are immutable, which makes upgrading them easy. Without containers, you may have to deploy the new code by taking machines offline, updating the app on that machine, and then spinning it up again, doing this for each machine until they're all running the same software. With containers, you simply build a new image with the new code and push that out to your cluster. You never update a container itself because they are unchangeable or immutable. Your orchestration software will direct new traffic to the new containers, and as the old containers are no longer needed, because they're no longer seeing new traffic, they'll be shut down. This way, your users experience a seamless transition to the new software, and you don't have to worry about periods of having fewer servers online during the transition. Docker, in particular, is very easily automated, having been designed with that automation in mind. It's easy to plug Docker into your CI/CD process so that containers are built from new code, tested and then pushed into production all automatically. Finally, once an app is up and running, this same orchestration software makes scaling the app to the amount of traffic it's seeing very easy. It's kind of like a load balancer in a typical cloud setup. The orchestration software will look at what containers are getting sluggish from too much traffic and will spin up more of that kind of container. If you've built your app using microservices, as is suggested when containerizing an app, then the orchestration software has a lot of fine detailed control over what gets spun up or down. Maybe only your database server needs more help. The front end is fine.

It takes some time for teams to containerize an app and update their workflow to work best with containers, but once they do, they should be writing better code, getting more feedback from different parts of the organization, and deploying and managing the apps more easily.

In this slide you can read some of the benefits of using Azure Container Instances as for example: Fast startup times. Containers can start in seconds without the need to provision and manage virtual machines. Public IP connectivity and DNS names. Containers can be directly exposed to the internet with an IP address and FQDN (fully qualified domain name). Hypervisor-level security. Container applications are as isolated in a container as they would be in a virtual machine. Custom sizes. Container nodes can be scaled dynamically to match actual resource demands for an application. Persistent storage. Containers support direct mounting of Azure Files file shares. Linux and Windows containers. Container Instances can schedule both Windows and Linux containers. Specify the operating system type when you create your container groups. Coscheduled groups. Container Instances supports scheduling of multi-container groups that share host machine resources. Virtual network deployment. Container Instances can be deployed into an Azure virtual network.

## Implement Container Groups

The top-level resource in Azure Container Instances is the container group. A container group is a collection of containers that get scheduled on the same host machine. The containers in a container group share a lifecycle, resources, local network, and storage volumes. A container group is similar to a pod in Kubernetes. A pod typically has a 1:1 mapping with a container, but a pod can contain multiple containers. The containers in a multi-container pod can share related resources. Azure Container Instances allocates resources to a multi-container group by adding together the resource requests of all containers in the group. Resources can include items such as CPUs, memory, and GPUs. Consider a container group that has two containers that each require CPU resources. Each container requests one CPU. Azure Container Instances allocates two CPUs for the container group. There are two common ways to deploy a multi-container group: Azure Resource Manager (ARM) templates and YAML files. ARM template. An ARM template is recommended for deploying other Azure service resources when you deploy your container instances, such as an Azure Files file share. YAML file. Due to the concise nature of the YAML format, a YAML file is recommended when your deployment includes only container instances. Container groups can share an external-facing IP address, one or more ports on the IP address, and a DNS label with an FQDN. External client access. You must expose the port on the IP address and from the container to enable external clients to reach a container in your group. Port mapping. Port mapping isn't supported because containers in a group share a port namespace. Deleted groups. When a container group is deleted, its IP address and FQDN are released. Consider the following example of a multi-container group with two containers.



The multi-container group has the following characteristics and configuration:

* The container group is scheduled on a single host machine, and is assigned a DNS name label.
* The container group exposes a single public IP address with one exposed port.
* One container in the group listens on port 80. The other container listens on port 1433.
* The group includes two Azure Files file shares as volume mounts. Each container in the group mounts one of the file shares locally.

Multi-container groups are useful when you want to divide a single functional task into a few container images. The images can be delivered by different teams and have separate resource requirements.

## Review the Docker Platform

Docker Hub provides a large global repository of container images from developers, open-source projects, and independent software vendors. You can access Docker Hub to find and share container images for your app and containers. Docker Hosts are machines that run Docker and allow you to run your apps as containers. The following illustration shows how Docker Hub communicates with Docker Host. Before you begin using Docker and Azure Container Instances to create, build, and test containers, it's helpful to be familiar with the terminology and concepts.

* **Container:** An instance of a Docker image. A container represents the execution of a single application, process, or service. It consists of the contents of a Docker image, an execution environment, and a standard set of instructions. When scaling a service, you create multiple instances of a container from the same image. A batch job can create multiple containers from the same image, and pass different parameters to each instance.
* **Container image:** A package with all the dependencies and information required to create a container. The dependencies include frameworks and the deployment and execution configuration that a container runtime uses. Usually, an image derives from multiple base images that are layers stacked on top of each other to form the container's file system. An image is immutable after it's created.
* **Build:** The process of creating a container image based on the information and context provided by the Dockerfile. The build also includes any other necessary files. You build images by using the Docker docker build command.
* **Pull:** The process of downloading a Docker container image from a container registry.
* **Push:** The process of uploading a Docker container image to a container registry.

## Demonstration – Deploy Azure Container Instances

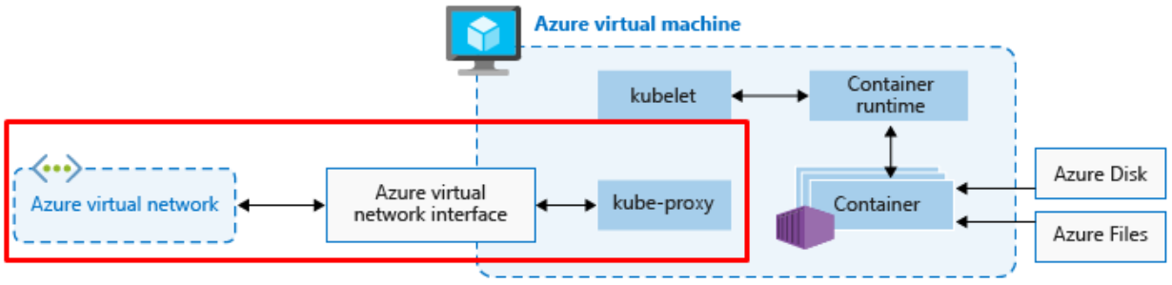
## Configure Azure Kubernetes Service

If your entire application is in a single container, then it’s very easy to run it on Azure. But running a single instance of a container is usually not sufficient. Most production applications need to be both scalable and highly available, so you need to run multiple copies of your container on multiple virtual machines. Furthermore, it’s becoming a common practice to break an application up into multiple microservices that run in separate containers. When you combine this with the need for high availability and scalability, the number of containers you need to manage can be overwhelming. The solution is to use a container orchestrator that takes care of the underlying technical details for you. By far the most popular orchestrator is Kubernetes, which is an open-source solution. Kubernetes lets you create clusters of virtual machines that run containers. Each virtual machine is called a node, and each node runs one or more pods. Each pod typically contains one container. The cluster also contains a control plane, which manages the nodes and pods. This is a very basic overview. Kubernetes has all kinds of features that help with storage, networking, security, and management. Despite the fact that Kubernetes is designed to simplify container deployments, it is quite a complex system, so it can take a bit of work to get it installed and running on a new cluster. Fortunately, Microsoft has made it much easier to do this by using Azure Kubernetes Service, or AKS for short. With AKS, you can create and configure a Kubernetes cluster quite easily. It's helpful to be familiar with the terminology and concepts.

* **Pools**: A pool is a group of nodes that have an identical configuration.
* **Nodes**: A node is an individual virtual machine that runs containerized applications.
* **Pods**: A pod is a single instance of an application. A pod can contain multiple containers.
* **Container**: A container is a lightweight and portable executable image that contains software and all of its dependencies.
* **Deployment**: A deployment has one or more identical pods managed by Kubernetes.
* **Manifest**: The manifest is the YAML file that describes a deployment.

## Understand AKS Clusters and Nodes

An Azure Kubernetes Service cluster is divided into two components: Azure-managed nodes and customer-managed nodes. Azure-managed nodes provide the core Kubernetes services and orchestration of application workloads in your AKS cluster. Customer-managed nodes run your application workloads in your AKS cluster. The following slide shows an example AKS cluster.

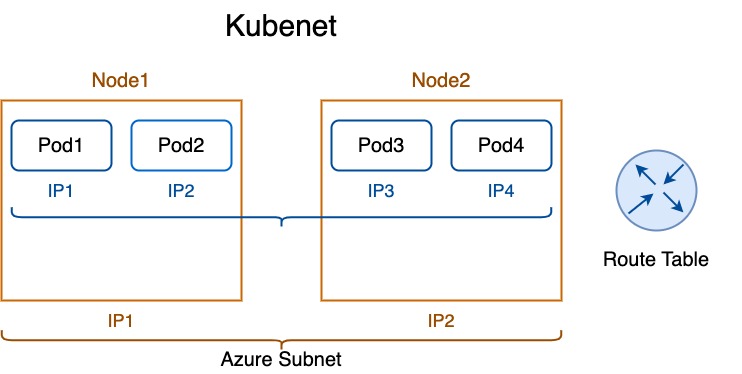
The Azure-managed node has a scheduler, controller, API server, and storage. The customer-managed node has a container runtime, container, kubelet agent, and kube-proxy component. We'll examine these items in the following section. Let's take a closer look at how Azure Kubernetes Service implements clusters and nodes.

* To run your applications and supporting services, you need a Kubernetes node for your AKS cluster. Each AKS cluster contains one or more nodes that run the Kubernetes node components and the container runtime.
* Nodes are instances of Azure Virtual Machines. Nodes of the same configuration are grouped together into node pools. A Kubernetes cluster contains one or more node pools.
* The initial number of nodes and size are defined when you create an AKS cluster, which creates the default node pool. The default node pool in AKS contains the underlying virtual machines that run your agent nodes.
* When you create an AKS cluster, an Azure-managed cluster node is automatically created and configured. This node is provided as a managed Azure resource that's abstracted from the user.
* The kubelet is the Kubernetes agent that processes the orchestration requests from the Azure-managed node, and scheduling of running the requested containers.
* The kube-proxy component handles virtual networking on each node. The proxy routes network traffic and manages IP addressing for services and pods.
* The container runtime component allows containerized applications to run and interact with other resources such as the virtual network and storage.
  + AKS clusters with Kubernetes version 1.19 node pools and later use containerd as the container runtime.
  + AKS clusters with node pools that use Kubernetes versions earlier than v1.19 implement Moby (upstream Docker) as the container runtime.
* When you implement Azure Kubernetes Service clusters, you pay only for running agent nodes in your cluster.

## Configure AKS Networking

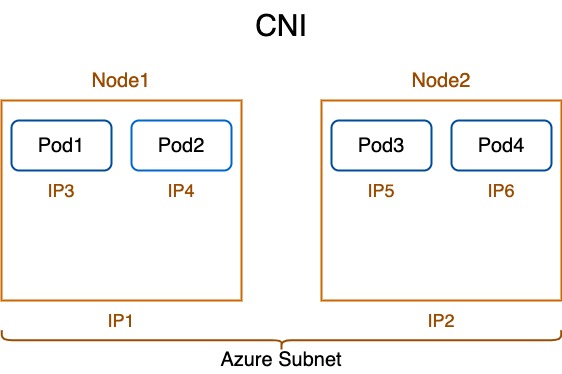
Kubernetes provides an abstraction layer for virtual networking to allow access to your applications, or for application components to communicate with each other. Kubernetes uses pods to run an instance of your application, and provides different services to logically group pods. This arrangement allows for direct access via an IP address or domain name system (DNS) name and on a specific port. Azure Kubernetes Service expands on the Kubernetes features to simplify the process to support networking. When creating an AKS cluster, you have two options for networking: kubenet and Azure Container Networking Interface (CNI). By default, AKS will use kubenet networking, also called *basic* networking. However, you can specify to use CNI (or *advanced*) networking if desired.

So, for Kubenet, we have our standard worker nodes as we see here.



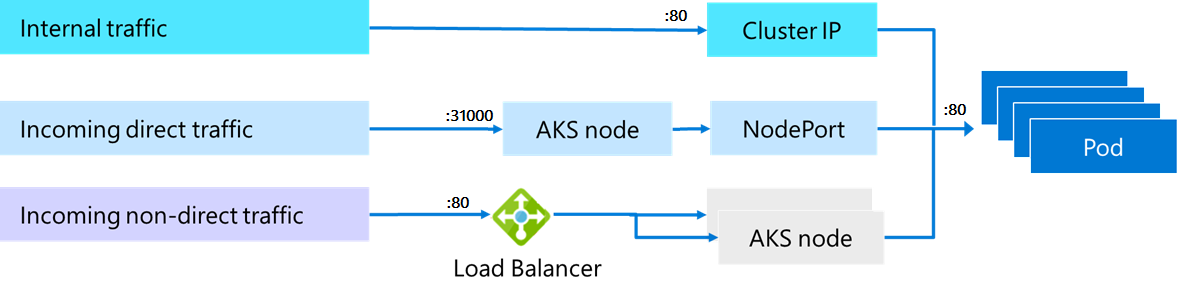
These nodes reside on the Azure subnet where they are allocated two IP addresses, one for each node. So what about our Pods? How do we communicate with them? Well, Kubenet takes care of that by assigning a CIDR IP block to each pod. Now this is not the same as the Azure VNet. No, far from it. This gives us the ability to have more flexibility with our Pods and our IP addressing and management. So, the worker nodes get their IPs from the subnet, and then the Pods get their IPs from Kubenet. Well, okay, great Mike. How do they talk? Well, it's sort of bridged if you will. There's a route table created at deployment, and you can find it in the deployed infrastructure resource group. So, let's say Pod1 wants to say hello to Pod3. Its next hop is going to be set to node 1's IP address, and that node 1 IP address knows who maintains the range of IPs on node 2. So, let's take a look at the portal. And here we can see in our Kubernetes infrastructure resource group, we have a route table automagically created for us. And as we see in our three‑node pool, each node has an address prefix defined and then its next hop, which is the IP address of the nodes.

Then we have Azure CNI.



So, here's our nodes again. Now the big takeaway here, again, is who's in charge of the IP addresses, and how do we communicate? So similar to Kubenet, the worker nodes get their IP addresses from the Azure subnet. So, it's still the same. But here's the difference between the two. The Pods and the worker nodes also get IPs allocated from the subnet. So, the plus side is we have direct contact with the Pods from our subnet. The downside is, well, we have to consider the number of IPs we have and the ability to allocate and manage those IPs. Now that's a big difference between the two. Be careful in the wording of the questions. Is it an Azure CNI network? Well, then you have to count every Pod and node with a valid IP. If it's Kubenet, only the worker nodes will have an IP allocated from the subnet. And again, in Kubenet, all communication is done via a route table, so that's how they know how to get back and forth between the different subnets.

Kubenet does reduce the number of IP addresses you need for your cluster, but because all pods are using an IP address internal to the cluster, network address translation (NAT) is required in order for the pods to establish a network connection to other Azure resources. It also means that other VMs in Azure or on-premises can’t directly establish communication with those pods.



Whether you’re using kubenet or CNI, there are challenges with networking in Kubernetes. Kubernetes is designed to orchestrate pods, and that means that it spins them up and tears them down as needed. Because of that, the IP addresses for your pods are constantly changing. For that reason, Kubernetes implements the concept of a *service* that sits between incoming network traffic and one or more identical pods. The service gets an IP address from a specific IP address pool set aside for services, and because the service is always running, it’s not affected by pod lifecycle. When network traffic needs to reach a particular pod (for example, a pod running a website that needs to process an HTTP request), the traffic is received by the service. The service will then balance the traffic to the pods using a round robin algorithm. There are multiple service types:

**Cluster IP** Provides an internal IP address that can only be used within the AKS cluster.

**NodePort** Provides a port mapping on the node, allowing network traffic to reach the node using the specified port. (Note that a different port can then be used from the service to the actual pod.)

**LoadBalancer** Provides an Azure Load Balancer and an external IP address to allow access to the node as per load balancing rules that are created. (Internal load balancers can be created to restrict access from the Internet.)

**ExternalName** Provides a DNS entry for AKS nodes.

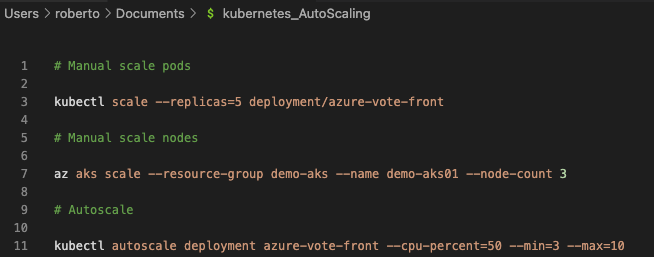
When using a LoadBalancer service type, the load balancer distributes traffic based on the originating port. However, there may be situations where you need more control over where traffic is routed. For example, suppose your cluster is hosting a web service, and you need traffic to be distributed to specific pods based on the incoming URL. A LoadBalancer service is unable to implement rules to deal with that, so in a case such as this, you can use an ingress controller to handle the traffic. You can use NGINX for an ingress controller in AKS, but you can also use other methods such as the AKS HTTP application routing feature or the Application Gateway Ingress Controller (AGIC) add-on. AGIC uses Application Gateway in Azure to make services in your cluster available over the Internet. Network security in an AKS cluster is handled using NSGs and network policy. Azure creates NSG rules for you as you create resources. Network policy is a feature in Kubernetes that enables you to control network traffic between pods.

## Configure AKS Storage

Next, let's talk about Azure Kubernetes Storage. So let's talk about a couple of concepts here. First, we have volumes. Volumes represent a way to store, retrieve, and persist data across the Pods and through the application lifecycle. Now they can use Azure Disk, as well as Azure Files. Now Azure Disks, as well Azure Files, use either premium or standard storage and are mounted as read‑write once and can only be used for a single Pod. Now if you need storage that can be accessed by multiple clusters, you would choose Azure Files. Something to jot down for the exam prep. Then we have persistent volumes, which are volumes defined and created as part of the Pod lifecycle and only exist until you delete the Pod. Now as with volumes, we can use both Azure Files, as well as Disks. And of course, it's determined which to use based on the concurrent access. And we have storage classes, which are used to define the tiers of storage, such as premium or standard. The StorageClass type also defines the *reclaimPolicy* actions for the storage. The *reclaimPolicy* definition controls the behavior of the underlying Azure Storage resource when the pod is deleted and the persistent volume might no longer be required. The underlying Storage resource can be deleted, or retained for use with a future pod. Now it can also define the reclaim policy of a Pod for when you choose to delete the Pod. This makes sure that storage resources are really, in fact, deleted when persistent volumes are, in fact, deleted. And then we have persistent volume claims, which request either disk or file storage of a particular storage class, like access mode and size.

## Configure AKS Scaling

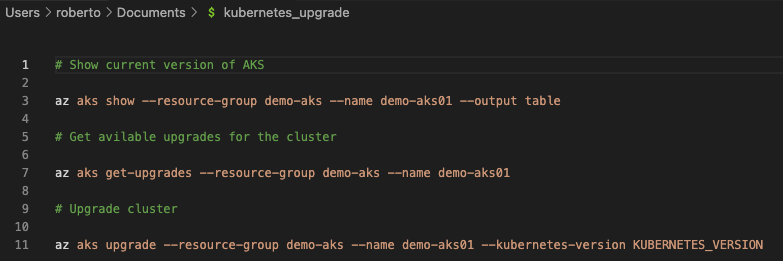
Now the first option for scaling is to do it manually. Now this can be done both in the portal, as well as in the shell. Then we have the horizontal Pod auto‑scaler or HPA. Now Kubernetes uses this to monitor demand and automatically scales the number of replicas. This can scale both up and back down based on demand or lack thereof, thereby controlling your Azure spend, which is really important. When you configure this, you create guardrails by starting the minimum and maximum numbers of replicas that can run. Now a variety of metrics can be configured to determine if and when to scale up or down. As with other auto‑scalers, we also have what's called a cooldown period. This prevents the HPA from changing the number of replicas before a previous scale event occurs. So you essentially set a delay value which ensures the auto‑scaler waits before another event can be triggered. Again, trying to save on your Azure spend. So the HPA is the Pod scaler. And next, we have the cluster auto‑scaler, which auto scales the nodes of the cluster based on the requested compute demand. Again, this can scale up and down. This is typically run in conjunction with the horizontal Pod scale. Now one thing I would watch for is a scenario in which they give you so‑called configure metrics, and it's up to you to determine how many Pods or worker nodes may be scaled up or down. And also, don't assume that it'll scale at all. So, in the exam, you're going to probably see some code for auto scaling.



So here we have the first one, which is manual scale. And we've defined the number of replicas here. So we have five replicas for our app, azure‑vote‑front. The next command is scaling for nodes. And as we see here, we're doing a node count of 3. And then finally, we're actually doing some auto scaling here where we have kubectl, which auto scales deployment for azure‑vote‑front, and we're looking at CPU percentage, as I mentioned earlier. We're looking at CPU usage of 50%. Now we're going to have a minimum of 3 nodes versus a maximum of 10, ensuring that we don't scale beyond our means. Now pay attention to the command structure, specifically the order of operations. You may have to choose which option of four goes into which slot, just like the other courses. Also notice, we're using both Azure CLI and kubectl.

## Upgrading a cluster

And finally, let's touch on the process of upgrading an AKS cluster. Here, we can show where we're at.



This is going to display what version we're at. Then we see what upgrades we have available. And then finally, upgrading to our target version. Now, here's the fine point on this is you can only upgrade one minor version at a time. No skipsies. You can go from 1.14.x to 1.15.x, but you cannot go from 1.14.x to 1.16.x. You can't skip. So be careful on the exam. Always remember upgrading the AKS cluster is an iterative process.

## Demo Application on Kubernetes Cluster

git clone <https://github.com/lumezit/tetris-game.git> javascript-tetris

cd javascript-tetris

* Open the Dockerfile:

The Dockerfile lists the commands required to create a container image for the application:

1. line 1: Start from the nginx:alpine image. Nginx is the web server used to serve the application. Alpine is a lightweight version of an Nginx image which will make the application image smaller in size.
2. line 3: Copy the source files into the /www directory in the image
3. line 4: Copy the included Nginx configuration (nginx.conf) into the image's default configuration path. This causes the container to load the given configuration by default.
4. line 6: Expose port 8000 so that incoming connections on port 8000 are allowed. The provided Nginx configuration causes the server to listen on port 8000.
5. line 8: Set the container start up command to write log messages to standard output and start Nginx. The daemon off string is required to start Nginx in a container.

The Dev Environment itself utilizes a Docker container to run. Therefore Cloud Shell cannot run a Docker daemon. So to test the program you run a web server provided by a Node.js package rather than using a container. In practice, you should use the same image for testing and production.

* Create container registry on portal

# Store ACR name  
acr\_name=$(az acr list --query [0].name -o tsv)  
# Run an ACR quick task to build image and push to app/tetris repository in ACR  
az acr build -t app/tetris:1.0 -r $acr\_name .

* Create a Kubernetes Cluster in the portal
* Get credentials for the Kubernetes cluster:

# Store the resource group's name

group\_name=$(az group list --query [0].name -o tsv)

# Store the AKS cluster's name

cluster\_name=$(az aks list --query [0].name -o tsv)

# Get credentials for cluster

az aks get-credentials --name $cluster\_name -g $group\_name --admin

* Confirm that the credentials grant you access to the cluster by listing the nodes in the cluster:

kubectl get pods --all-namespaces

* Create a Kubernetes deployment manifest file and open it in the code editor:

touch app.yaml

* Copy the content in the file

apiVersion: apps/v1

kind: Deployment

metadata:

name: tetris

spec:

replicas: 2

selector:

matchLabels:

app: tetris

template:

metadata:

labels:

app: tetris

spec:

containers:

- name: tetris

image: {{acr\_name}}/app/tetris:1.0 # image in ACR

resources: # include resources for better scheduling

requests:

cpu: 100m

memory: 128Mi

limits:

cpu: 250m

memory: 256Mi

ports:

- containerPort: 8000

---

apiVersion: v1

kind: Service

metadata:

name: tetris

spec:

ports:

- port: 80 # Access on service port 80

protocol: TCP

targetPort: 8000

selector:

app: tetris

type: LoadBalancer # External Access via load balancer service

Read through the file to see how the resources are configured. The manifest file is comprised of a Deployment, and Service. The Deployment creates two replicas of the application and pulls the image from ACR. The Service creates a load balancer with an external IP for accessing the Pods created by the Deployment. The Service is available on port 80, the standard HTTP port.

* In the terminal, execute the following commands to replace the placeholders in the manifest with the required values:

# Store ACR DNS name

acr\_name=$(az acr list --query [0].loginServer -o tsv)

# Replace the placeholders with the values using stream editor (sed)

sed -i "s/{{acr\_name}}/$acr\_name/" app.yaml

* Create the resources in the manifest file to deploy the application:

kubectl config set-context --current --namespace=default

kubectl create -f app.yaml

* Watch the service until the **EXTERNAL-IP**appears:

kubectl get svc

* Copy the **EXTERNAL-IP**and navigate to it in a new browser tab

Open Insight to check monitoring part