Course Overview
Course Overview

Hi everyone. My name Manoj Ravikumar Nair. Welcome to my course, Azure Kubernetes Services - The Big Picture. I work as a cloud solutions architect with Microsoft. Whether you work for a startup, working on the next Pokemon Go or for an enterprise modernizing your applications workloads, containers are quickly becoming the preferred choice of computing for organizations worldwide. While working with a couple of containers is absolute fun, tens and thousands of containers can quickly spiral out of control. Thankfully, Kubernetes solves this problem and is quickly becoming the most sought after container orchestration service with excellent community support. This course is a quick introduction to Azure Kubernetes Service, or AKS, which is a managed Kubernetes service running on the enterprise-grade cloud offering powered Microsoft Azure. No experience in Docker or Kubernetes is required. Some of the major topics that we will cover include examining the use case for Kubernetes; understanding the fundamentals of Docker and Kubernetes; exploring the container ecosystem on Microsoft Azure; deploying, and scaling, and updating applications on the Azure Kubernetes Service. By the end of this course, you'll have a solid understanding of the fundamentals of the Azure Kubernetes Service and be ready to make an informed choice on how AKS would fit into your containerization or your modernization strategy. From here, continue your learning by diving into Azure Container Services with courses on Azure Container Instances and Designing a Compute Strategy for Microsoft Azure. I hope you'll join me in this journey to learn Azure Kubernetes Service, or AKS, with the Azure Kubernetes Services - The Big Picture course here, at Pluralsight. On that note, let's get kubing.

What Is Azure Kubernetes Service (AKS)?

Version Check

Module Introduction

Unless you've been living under a rock, you might have heard about containers or, if I have to be vendor-specific, Docker containers are the CoreOS's rkt. If not, you're certainly making this guy angry, in fact very angry. All right, I'm joking. But containers are indeed one of the most sought-after technologies in today's computing world. Whether you work for an enterprise that is adopting containers to modernize your application stack or you're a part of this awesome startup that is working on the next Pokemon Go, containers are becoming the de facto compute service for organizations worldwide. And for Kubernetes, well, Kubernetes has become the preferred choice for container orchestration. Hold on a second. It's just a minute in this course, and I have already thrown a bunch of buzz words. Before you think about closing your browser window and thinking that this is going to be a keyword heavy course, bear with me for a moment. Well, as with every big picture course, to get the most out of this, you and I must be on the same page. You may or may not have heard about Kubernetes or all this Docker hype. That's okay. This course will cement the core concepts of Kubernetes and its orchestration magic and top it up with the Microsoft Azure's magic serum of AKS or the Azure Kubernetes Service. So in this module, my intention is to set the field straight. We will demystify a lot of terms, terms like containers, container orchestration via Kubernetes, microservices, and a whole lot more. Once we are on the same page in terms of

Docker containers and what Kubernetes is, we will dig into the meat of this course. What is this Azure Kubernetes Service or AKS? There are plenty of cloud vendors out there that offer a managed Kubernetes offering. So why choose Microsoft Azure? Well, that's a great question, and I'll cover that off in this module. I am so excited to take you on a journey with Azure Kubernetes Service, so let's get started.

The Case for Kubernetes

Traditionally, software applications were built as big monoliths, running either as a single process or a small number of processes spread across a handful of servers. At the end of every release cycle, developers had to package the whole system and hand it over to the ITOps team, which then deploys it and monitors its health. In case of any hardware failures, the ITOps team manually migrates it to remaining healthy servers. Fast-forward 2018, these big monolithic legacy applications are being broken down into much smaller, independently running components called microservices. Microservices are decoupled from each other, and therefore, they can be developed, deployed, updated, and scaled individually. By adopting the microservices architectural pattern, you can change components quickly and quite often as you need to keep up with today's rapidly changing business requirements. Each microservice exposes its functionality or services via an interface, typically a RESTful API. This allows other microservices to consume those APIs using any language of their choice. Because each microservice is a standalone process with a relatively static external API, it is possible to develop and deploy each microservice separately. A change to one of them doesn't require changes or redeployment of any other microservice provided that the API doesn't change, or change is only in a backward-compatible way. Let me explain this with the help of an example. Let's say you build a web application that accepts an image and performs a few image manipulation tasks, like converting into a grayscale image, applying brightness, applying contrast, etc. Also, all this functionality is currently built into a single code base. You package this code, and you deploy it on a server. The application becomes famous. So to keep up with demand, you either scale vertically, that is you add more CPU, RAM, and make the server beefy. Or you can scale horizontally and add more servers that contain the code base and place a load balancer in front of them. Now, let's say that the most popular feature of this application turns out to be the grayscaling of the images. Since all this functionality is tightly coupled into a single code base, you can no longer scale just this feature of grayscaling images. Now let's say you decompose your application into multiple services, each developed independently and exposes its functionality as an API. So let's say you have a web front-end server that accepts the image and stores into a cloud storage service like the Microsoft Azure storage hub. Then, based on the request, you either call the grayscale service or the apply brightness service or the apply contrast service. If in future, you notice that a particular feature gets more popular, you can simply deploy more instances of that feature. And boom, you already have easily scaled one specific aspect of your application. Also, the development team working on these features can work independently and build the servers with the technology best suited for their implementation. Awesome, isn't it? But like anything in the software world, microservices solves and addresses a lot of problems that monoliths faced, but also introduces a fresh set of challenges. When your system consists of only a small number of deployable components, managing these components is very easy. It's trivial to decide where to deploy each component because there aren't much choices. For example, if my application

is contained in a single code base, I can simply deploy this code base to multiple servers and potentially load balance them. When the number of these components increases, deployment-ready decisions become increasingly difficult because not only does the number of deployment combinations increase, but the number of interdependencies between these components increases by even a greater factor. It's much harder to figure out where to put each of these components to achieve high resource utilization and thereby to keep the hardware costs down. Doing all of this manually is a hard work. For instance, let's say in the earlier example, if the apply brightness service requires a GPU to work, you need to make sure that you deploy this microservice on a host that has a GPU. We need automation, which includes the automatic scheduling of those components to our servers, automatic configuration, supervision, and failure handling. In other words, if you're going to host microservices in containers, you need something that orchestrates this for you. You need a container orchestrator, and this is where Kubernetes comes in. Kubernetes enables developers to deploy their applications themselves as often as they want without requiring any assistance from the ITOps team. But Kubernetes doesn't just benefit the developers. It also helps the ops team by automatically monitoring and rescheduling those apps in the event of a hardware failure. The focus for sysadmins shift from supervising the individual applications to mostly supervising and managing Kubernetes and the rest of the infrastructure, while Kubernetes itself takes care of the applications. And here's my favorite feature of Kubernetes. Kubernetes abstracts away the hardware infrastructure and exposes your whole data center as a single gigantic computational resource. It allows you to deploy and run your software components without having to know about the actual servers underneath. It doesn't care whether the server is a physical box or a virtual machine. When you deploy a multi-component application through Kubernetes, it selects a server for each component, deploys it, and enables it to easily find and communicate with all the other components of your application. This makes Kubernetes great for most of the on-premises data centers. But where it starts to shine and is almost godsend is that when it's used in one of the world's largest data centers, the ones that are run by Microsoft. With the enterprise-grade cloud offered by Microsoft and the scale of features it provides, Kubernetes plus Microsoft Azure is a win-win situation for both developers, as well as ITOps. And as a technical decision maker, you know that if your developers and ITOps team are happy at the same time, productivity increases, and your business goals can be successfully met.

Containers 101

If you point your browser to kubernetes.io, you will notice this tagline on its homepage, Production-Grade Container Orchestration. Also, note the one-line definition, so to speak. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. So to better understand Kubernetes, you must first have a good understanding of what containers are. In simple words, containers offer the lightweightness of a process and the isolation of an operating system. Just like with virtual machines, you can virtualize the hardware and have multiple virtual machines running on a single physical hardware box. With containers, you virtualize the operating system and get isolated copies of the operating system that can offer compute services to your applications. Since the host operating system is already booted, spinning up new containers is almost instantaneous. Virtual machines run on virtualized hardware, but they still contain a

full-blown operating system, an operating system that needs to be patched, upgraded, and maintained. You then run your applications inside the virtual machine. On the other hand, containers running on a Linux operating system or with Windows Server 2016 on Windows operating system use namespaces. Namespaces offer isolation. All system resources, such as file systems, process IDs, user IDs, and network interfaces, belong to a single namespace. You can create additional namespaces and organize resources across them. Now, here's the most important thing that I want you to remember. The process will only see resources that are inside the same namespace. Now, multiple kinds of namespaces exist. So a process doesn't belong to one namespace, but to one namespace of each kind. There are multiple kinds of namespaces, like mount, process ID, network, inter-process communication, UTS, and user ID. Each namespace kind is used to isolate a certain group of resources. For example, the UTS namespace determines what host name and domain name the process running inside the namespace sees. But assigning two different UTS namespaces to a pair of processes, you can make them see different local host names. In other words, to the two processes, it will appear as though they are running on two different machines, at least as far as the host name is concerned. Likewise, what network namespace a process belongs to determines which network interfaces the application running inside the process sees. Each network interface belongs to exactly one namespace, but can be moved from one namespace to other. Each container uses its own network namespace. And therefore, each container sees only its own set of network interfaces. What I've covered so far should give you a basic idea of how namespaces are used to isolate applications running inside the container from each other. The other half of container isolation deals with limiting the amount of system resources a container can consume. This is achieved with Cgroups, a Linux kernel feature that limits the resource usage of a process or a group of processes. A process can't use more than the configured amount of CPU, memory, and network bandwidth. This way, processes cannot hog resources reserved for other processes, which is very similar when each process runs on a separate machine. While container technologies have been around for a long time, they have become more widely known with the rise of Docker container platform. Docker was the first container system that made containers easily portable across different machines. It simplified the process of packaging up not only the application, but also all its libraries and other dependencies, even the whole OS file system, into a simple portable package that can be used to provision the application to any other machine running Docker. When you run an application package with Docker, it sees the exact file system contents that you have bundled with it. It sees the same files whether it's running on your development machine or on a production machine, even if the production server is running a completely different Linux operating system. The application won't see anything from the server it's running on. So it doesn't matter if the server has a completely different set of installed libraries compared to your development machine. For example, if you package your application with the files of the whole Ubuntu operating system, the application will believe that it's running inside Ubuntu. But when you run it on your development machine that runs CentOS or when you run it on a server running Red Hat or some other Linux distribution, only the kernel may be different. This is similar to creating a virtual machine image by installing an operating system inside a VM, installing the app inside it, and then distributing the whole VM image around and running it. Docker achieves the same effect. But instead of using VMs to achieve application isolation, it uses Linux containers technologies like namespaces and control groups to provide almost the same level of isolation that virtual machines do. So instead of using big monolithic VM images, it uses container images, which are usually much, much smaller. So what does Docker offer? Well, Docker is a platform for

packaging, distributing, and running applications. There are three important concepts in Docker that you need to be aware of. Images. A Docker-based container image is essentially what you package your application and its environment into. The image contains a file system and the path to the application executable when you run the image. Registries. A Docker registry is a repository that stores your Docker images and allows easy sharing of those images between different people and computers. When you build your image, you can either run it on the computer you have built it on, or you can upload the image to a registry and put it on another computer and run it there. And finally, containers. A Docker-based container is a regular Linux or a Windows container created from a Docker-based container image. A running container is a process running on the host running Docker, but it's completely isolated from both the host, as well as the other processes running on it. A typical Docker workflow for building and deploying applications is pretty straightforward. The developer first builds an image and then pushes it to the registry. The image is thus available to anyone who can access the registry. They can pull the image to any machine of their choice running Docker and then run the image. Docker creates an isolated container based on the image and runs the binary executable specified as a part of the image. Well, this is all that you need to know about containers to get the most out of this course. However, feel free to look at other courses in the Pluralsight library to learn more about Docker. Now, the fact is that you're not going to work with just 1 or maybe 10 containers in a production environment. There are going to be hundreds and thousands of them. And we've already talked about how difficult it is to deploy and manage thousands of containers in the real world. Enter Kubernetes.

What Is Kubernetes

Kubernetes is a software system that allows you to easily deploy and manage containerized applications on top of it. It enables you to run your software applications on thousands of computer nodes as if all these nodes were a single anonymous computer. It abstracts away the underlying infrastructure and doesn't care whether it contains physical machines or virtual machines. When you deploy your applications through Kubernetes, the process is always the same, whether your cluster contains only a couple of nodes or thousands of them. The size of the cluster makes no difference at all. Additional cluster nodes simply translate to additional amount of resources available to your deployed applications. Here's the simplest possible view of a Kubernetes system. The system is composed of a master node and any number of worker nodes. The worker nodes were initially called as minions, but now just referred as worker nodes. It all starts with the developer. Let me repeat that. It all starts with the developer. The developer submits the list of applications, typically called an app descriptor, to the Kubernetes master. The master looks at the app descriptor, figures out what to do, and then deploys those applications onto the worker nodes. As a developer, you don't care which node your application runs on. Your only concern is that you want X amount of copies of your application running, and Kubernetes handles that for you. The developer can specify that certain applications must run together, and Kubernetes will deploy them on the same worker node. Others will be spread around the cluster, but they can still talk to each other in the same way, regardless of where they are deployed. Let's take a closer look at what a Kubernetes cluster is composed of. At the hardware level, a Kubernetes cluster is composed of many nodes, which can be split into two types, the master node, which hosts the Kubernetes control plane that controls and manages the whole Kubernetes system, worker

nodes that run the actual applications that you deploy. The control plane, or the master node, is what controls the cluster and makes it function. It is what I call the heart and soul of Kubernetes. It consists of multiple components that can run on a single master node or be split across multiple nodes and replicated to ensure high availability. Let's take a look at these components one by one: the Kubernetes API server, which you and the other control plane components communicate with, the scheduler, which schedules your applications, basically assigning a worker node to each deployable component of your application, the controller manager, which performs the cluster-level functions, such as replicating component, keeping track of worker nodes, handling node failures and so on, etcd, which is a reliable distributed data store that persistently stores the cluster configuration. Note that these components of the control plane hold and control the state of the cluster, but they don't run your applications. This is done by the workers nodes. The workers are the machines that run your containerized applications. It consists of three important components, the container runtime, which can be Docker or rkt, the kubelet, which talks with the API server and manages containers on its node, and the Kubernetes service proxy, which load balances network traffic between the application components. All right, now that you have a good understanding of the Kubernetes architecture, let's take a step back and try to understand how a developer or a DevOps engineer interacts with the Kubernetes cluster. To run an application in Kubernetes, you first need to package it up into one or more containerized images, push those images to an image repository, like an image registry like Docker Hub or Azure Container Registry, and then post a description of your application to the Kubernetes API server. The description includes information such as the container image or images that contain your application components, how those components are related to each other, and which ones need to be run collocated. That is, they need to be run together on the same node. For each component, you can also specify how many copies or replicas you want to run. When the API server processes your application's description, the scheduler schedules specified groups of containers onto available worker nodes based on the computational resources required by each group and the unlocking of resources on each node at the moment. The kubelet on those nodes will then instruct the container runtime, which in this case is Docker, to pull the required container images and run the containers. Now let's take a look at some of the benefits of using Kubernetes. First, it simplifies application deployment. By exposing your worker nodes as a single deployment platform, it abstracts developers from the underlying infrastructure, making deploying applications easy as a pie. Better hardware utilization. While you can certainly manually choose the node you want to deploy your containerized applications onto, outsourcing that to Kubernetes is a much more intelligent choice. Kubernetes chooses the most appropriate node to run your application based on the description of the application's resource requirements and the available resources on each node. Health monitoring and self-healing. In the unfortunate event that a worker node dies or becomes inaccessible, Kubernetes will select the new nodes for all the containers that were running on the node and run them on the newly selected nodes. Automatic scaling. While the application is running, you can decide whether you want to increase or decrease the number of copies, and Kubernetes will spin up additional ones or simply stop the excess ones. You can even leave the job of deciding the optimal number of copies to Kubernetes. Now I'm pretty sure you're all excited to get your hands dirty with Kubernetes and see things in action. But before that, we need to be very clear on Kubernetes objects or resources as they are also referred to as, and that's what I'm going to cover next.

Kubernetes Objects

Let's start with kubectl. Whether you pronounce it as kubectl or kube ctl, well that's an internal debate within the Kubernetes Kubernetes. For the purpose of this course, I'm going to pronounce it as kubectl. Just like as you interact with Azure using the Azure CLI or using the Azure PowerShell module, kubectl is a command line interface for running commands against Kubernetes clusters. To run kubectl commands from your terminal window, we use the syntax as shown on your screen, kubectl followed by the command or the operation you want to perform, then the type of the Kubernetes object that you want to work with, which is then followed by the resource name and optionally any flags. For example, to get the list of nodes running in your Kubernetes cluster, you can run kubectl get nodes. To learn more about kubectl, you can simply refer to the help documentation. For example, to get more help on kubectl, you can type in kubectl and then type in help and then Enter. Pods. A Pod is the smallest unit that Kubernetes manages, and it's a fundamental unit that the rest of the Kubernetes system is built on. It is made of one or more containers and the information associated with those containers. When you ask Kubernetes about a Pod, it will return a data structure that includes a list of one or more containers along with the metadata that Kubernetes users to coordinate the Pod with other Pods and policies of how Kubernetes should act and react if the program fails. The metadata can also define things such as affinity, which influences where a Pod should be scheduled in a cluster, expectations around how to get the container images, and much more. Now one thing to bear in mind is that the Pod is not intended to be treated as a durable or a long-lived entity. Here are some of the characteristics of a Kubernetes Pod. All the containers for a Pod will be run on the same node. Any container running within a Pod will share the node's network and any other containers in the same Pod. Containers within a Pod can share files through volumes attached to the containers. A Pod has an explicit lifecycle and will always remain on the node in which it was started. For all practical purposes, when you want to know what's running on a Kubernetes cluster, you are generally going to want to know about the Pods running with your Kubernetes cluster and their state. Let's tackle namespaces now. Pods are collected into namespaces, which are used to group Pods together for a variety of purposes. For example, to see the status of all the Pods in the cluster, you can run kubectl get pods with the --all -namespaces option. Namespaces can be used to provide quotas and limits around the resource usage, have an impact on the DNS names that Kubernetes creates internal to the cluster, and, in future, may even impact access control policies. If no namespace is specified when interacting with the Kubernetes through kubectl, the command assumes that you're working with the default namespace, which is typically called default. Note, a node is a machine typically running Linux that has been added to the Kubernetes cluster. It can be a physical machine or a virtual machine. As we saw earlier, the master node is the brain of Kubernetes, while the worker nodes do the actual work pulling container images and running Pods. Networks. All the containers in a Pod share the nodes network. In addition, all nodes in a Kubernetes cluster are expected to be connected to each other and share a private cluster-wide network. When Kubernetes runs containers in a Pod, it does so within this isolated network. Kubernetes is also responsible for handling IP addresses, creating DNS entries, and making sure that a Pod can communicate with the other Pod in the same Kubernetes cluster. Services, which we'll dig into a bit later, is what Kubernetes uses to expose Pods to each other over this private network or to handle connections in and out of the cluster. Controllers. Kubernetes is built with the notion that you tell it what you want, and it knows how to do it, just like PowerShell's desired state configuration. When you interact with Kubernetes, you're asserting you want one or more

resources to be in a certain state and with specific version. Controllers are where the brains exist for tracking those resources and attempt to run your software as you describe. These descriptions can include how many copies of a container image are running, updating the software version running within a Pod, and handing the case of a node failure where you unexpectedly lose a part of your cluster. There are a variety of controllers used within Kubernetes, and they are mostly hidden behind two key resources, deployments and replica sets. Let's first start with replica sets. A ReplicaSet is associated with a Pod and indicates how many instances of that Pod should be running within that cluster. A ReplicaSet also implies that Kubernetes has a controller that watches the ongoing state and knows how many of your Pods to keep running. This is where Kubernetes is really starting to do the work for you. If you specify three Pods in a ReplicaSet and one failed, Kubernetes will automatically schedule and run another Pod for you. A ReplicaSet is commonly wrapped in turn by a deployment. Now let's talk a little bit more about deployments. The most common and recommended way to run code on Kubernetes is with the deployment, which is managed by a deployment controller. A Pod by itself is interesting, but limited, specifically because it is intended to be ephemeral. If a node were to die, all the Pods on the node would stop running. ReplicaSets provides self-healing capabilities. Self-healing, hmm. What does that mean? Well, ReplicaSets work within the cluster to recognize when a Pod is no longer available and which will attempt to schedule another Pod typically to bring a service back online. The deployment controller wraps around an extensive ReplicaSet controller, and it is primarily responsible for rolling out software updates and managing processes of that rollout when you update your deployment resource with new versions of your software. The deployment controller includes metadata settings to know how many Pods to keep running so that you can enable a seamless rolling update of your software by adding new versions of a container and stopping all versions when you request it. Services. A service is the Kubernetes resource used to provision and abstraction through to your Pods that is agnostic of the specific instances that are running. Providing a layer between what one container or a set of containers provides, such as a front-end web application and other layers such as a database, it allows Kubernetes to scale them independently, update them, handle scaling issues, and much more. A service can also contain a policy by which data should be transferred. So you might consider a service as a software load balancer within Kubernetes. Finally, let's talk about representing Kubernetes resources. Kubernetes resources can be generally represented as either a JSON or a YAML structure. Kubernetes is specifically built so that you can save this file. And when you want to run your software, you can use a command such as kubectl deploy and provide the definitions you created previously, and it uses that to run your software. In the next module, we'll use one such YAML file to deploy our application on a cluster managed by Kubernetes. I think that's pretty much it. Well, of course, there are tons of other things that you can learn about Kubernetes. But what we have covered so far, it should give you a pretty solid understanding of the fundamentals of Kubernetes.

What Is AKS

After watching the first few clips, you might be tempted to try out Kubernetes, and I wouldn't blame you for that. I felt the same when I started to read upon Kubernetes. I just wanted to dive right in and get a Kube cluster up and running. You can do this yourself and manually install Kubernetes master Node and a bunch of Kubernetes worker Nodes. If you're deploying this in production, then you also must take care of its high availability, patching,

maintaining, adding more Nodes if your application requires more compute resources. Doing the deployment of a Kubernetes cluster manually and then making sure that it's highly available, well, that's a lot of work. All we want is to take advantage of the amazing Kubernetes platform without going through the unnecessary underlying plumbing to get Kubernetes up and running. Well, fortunately for us, we have Microsoft Azure. Microsoft Azure offers Azure Kubernetes Service that simplifies deployment, management, and operations of Kubernetes. At the time of recording this course, Azure Kubernetes Service, or AKS as it's generally referred to as, is currently in public preview. Though there is a very high chance by the time you're watching this course, it will be generally available or GA. Azure Kubernetes Service manages your hosted Kubernetes environment, making it quick and very easy to deploy and manage containerized applications without any container orchestration expertise. And here is the best part. It elevates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand without taking your applications offline. In other words, just like as Azure App Services abstracts you from the underlying virtual machines used to host your web or API apps, AKS abstracts the complex infrastructure of a Kubernetes cluster using Azure virtual machines as Kubernetes worker Nodes, Nodes that you don't have to babysit or take care of. In fact, so good is AKS that you don't even see or have to worry about the master Node. That is completely managed by Azure. Trust me. This is indeed a winner because if you ever speak to a system administrator whose job is to maintain the high availability of the Kubernetes master Node, he will cringe on how daunting that entire process can be. By using AKS, you can take advantage of the enterprise-grade features of Microsoft Azure, while still maintaining the application portability through Kubernetes and through Docker. AKS reduces the complexity and operational overhead of managing a Kubernetes cluster by offloading much of the responsibility to Microsoft. As a hosted Kubernetes service, Azure handles the critical tasks such as health monitoring and maintenance for you. In addition, you only pay for the agent Nodes within your clusters, not for the masters. Now how cool is that? The goal of Azure Kubernetes Service is to provide a container hosting environment by using open source tools and technologies that are popular among customers today. With AKS, you can use familiar tools like kubectl to manage and interact with your Kubernetes environment. In fact, you don't even have to install kubectl. If you use the Azure Cloud Shell, then kubectl is a part of that. Let's take a look at the key benefits that Azure Kubernetes Service offers. With AKS, you get automated Kubernetes version upgrades and patching. So you don't have to worry about upgrading to a new version of Kubernetes once it's released. Microsoft takes care of performing version upgrades and patching so that you can concentrate on your applications and not on the infrastructure. Easy cluster scaling. Do you want to scale your cluster from 1 to 10 worker Nodes? Well, that's easy as a pie with AKS. You can use the Azure CLI or the SDKs to make a simple API call. And boom, your cluster gets scaled. Self-healing hosted control plane or masters. As I said earlier, you don't even see the master Nodes, so one less thing to worry about. Cost savings. You only have to pay for the running agent pool Nodes. If you're like me, I'm pretty sure you must be sold on Azure Kubernetes Service. Now, you might wonder though, why should I use AKS when there are other vendors out there offering a manage Kubernetes platform? The answer to that question is very simple. AKS is just one service within Microsoft Azure that offers container solutions. There are others. Let's take a look at these next.

Beyond Management Kubernetes

All right, Azure offers a managed Kubernetes platform. Well, that's great. But why Azure? Why not someone else who offers a managed Kubernetes platform? We've already seen the benefits that AKS offers, so I'm not going to hop on it again. But when you evaluate a cloud vendor, you typically look at the bigger picture or the wider ecosystem the cloud vendor brings onto the table. Since we're talking about Azure and containers, I will focus more on the container solutions on Azure. Here is a quick overview of the Azure container ecosystem. When you deploy container workloads on Azure's AKS or Azure Kubernetes Service, you get the benefits of a managed Kubernetes platform plus the entire portfolio of Azure service offerings. Let's start with Azure Container Instances. So if you just want to deploy a container without worrying about the underlying host or about any orchestrator, Azure Container Instance fits the bill. With ACI, you can easily start deploying containers to support your targeted use cases or other application development and testing scenarios. So Dockerize your application and execute immediately with just one click. For example, let's say you want a Jenkins server to run a few CI/CD tasks. With ACI, you can spin up a Jenkins Docker container, perform your builds, tests, etc., and then ted on ACI. There are no virtual machines, no hypervisors to manage. Here is one example. I'm using the Azure CLI to create an ACI instance called mycontainer that uses the Microsoft ACI helloworld container image, and I see that I want a public IP address. As you can see, it spins up a container for me in 20 seconds or less. And boom, I can now connect to the public IP address of my web application. Of course, you can also do this via Azure portal. When ACI was launched, it had a very interesting capability in it. The ACI connector for Kubernetes, which bridges the world of ACI with the world of Kubernetes. The ACI connector can dispatch Pods to ACI. So you get the benefits of the per-second billing and the cloud-scale container capabilities of ACI. Think of this as your Kubernetes worker Nodes, but running on ACI. This is all possible thanks to virtual kubelet. The virutal kubelet registers itself as a Node and allows developers to program their own behaviors for operations on Pods and containers. Next, Azure Container Registry. For containers, we obviously need a storage layer to house our Docker images. Well, that's where Azure Container Registry comes in. Azure Container Registry, or ACR, allows you to store images for all types of container deployments, including DC/OS, Docker Swarm, Kubernetes, and Azure services such as the App Service, Batch, Service Fabric, and others. Your DevOps team can manage the configuration of apps isolated from the configuration of the hosting environment. You stop worrying about high availability. You can efficiently manage a single registry, which is replicated across multiple regions. Now, thanks to georeplication, you can manage global deployment as one entity to simplify operations and management. You don't even have to learn new APIs or commands because ACR, or Azure Container Registry, is compatible with open-source Docker Registry version 2 APIs. So you can use the same open-source Docker CLI tools you already are aware of. You can get up and running with Azure Container Registry via portal in just a few clicks. Let's talk about Service Fabric next. Well, what can I say about Service Fabric? Service Fabric is the foundational technology powering core Azure infrastructure, as well as Microsoft services, such as Skype for Business, Intune, Azure Event Hubs, Azure Data Factory, Cosmos DB, SQL Database, Dynamics 365, and Cortana. It is designed to deliver highly available and durable services at cloud scale. Service Fabric understands the available infrastructure and the resource needs of your application, and it allows you to automatically enable scaling, rolling upgrades, and self-healing from faults when they occur. Service Fabric is the Microsoft's container orchestration deploying microservices across a cluster of machines. Microservices

can be deployed in many ways. Either you use the Service Fabric programming models, ASP.NET Core, or deploying any core of your choice. If you just want to deploy and manage containers, Service Fabric is a perfect choice for that as well. It's a great container orchestrator. Service Fabric runs everywhere. You can create clusters of Service Fabric in your on-premises environment, in Azure, or in any other cloud vendor, and you can use Windows servers or Linux for the underlying resources. You can build both stateful and stateless microservices, which is possible using Service Fabric. A key differentiating factor with Service Fabric is its strong focus on building stateful services, either with the built-in programming models that it offers or by using containerized stateful services. In the recent Build conference, Microsoft also announced the new Service Fabric Mesh. Now this is one thing I'm really excited about. It's a service that offers the same reliability and mission control performance of Service Fabric, but with no overhead of cluster management and patching operations. Next, Azure App Services or Web App for Containers. If you work with Azure, you know that Azure App Service has been the industry leader for Platform as a Service offering for running of web applications or your API apps. With Web App for Containers, you get all the benefits of Azure App Service along with the ability to easily deploy and run containerized web applications that scale with your business. You can also take advantage of the built-in autoscaling and load balancing, streamline continuous integration and continuous delivery with Docker Hub, Azure Container Registry, and GitHub. What this means is that the Azure App Service creates an association with the selected repository so your apps can be updated each time whenever your source code changes. You can schedule performance and quality tests in staging environment and use deployment slots to swap staging to production in seconds or roll back previous versions without downtime. And finally, the Azure Batch Service. Azure Batch Service basically offers you cloud-scale job scheduling and compute management that can scale to tens and hundreds and even thousands of virtual machines to power your high-performance computing or the HPC workloads. Azure Batch earlier used to run batch tasks only on Azure virtual machines. However, now you can also use batch pool to run tasks in Docker containers. Using containers provides an easy way to run batch tasks without the need to manage application packages and dependencies. You can use both Windows and Linux containers. And if you're excited about running your batch workloads on containers via Azure Batch, be sure to check out the Shipyard toolkit at the link shown on your screen. Batch Shipyard is a tool to help provision and execute containerized-based batch processing and high-performance compute workloads on Azure Batch compute pools. As you saw, Microsoft Azure offers a whole gamut of services that help you deploy and manage containerized workloads on Azure. For a quick summary, let's take a look at this table. If you want to simplify the deployment, management, and operations of Kubernetes, well use Azure Kubernetes Service, or AKS. To easily run containers on Azure with a single command, use Azure Container Instances. To store and manage containerized images across all types of Azure deployments, use Azure Container Registry. For developing microservices and orchestrating containers on Windows and Linux, use Service Fabric. To deploy web applications on Linux using containers, use Azure App Service or Web App for Containers. And finally, to run repetitive compute jobs using containers, use Azure Batch. Now you can also take advantage of other services like Cosmos DB that offers global scale, multi-model database or, my personal favorite, Azure Event Hubs or Azure Functions for going serverless, and the list can just go on. So when you use AKS, what you get is an access to a full suite of Azure services that can help you build an awesome application that you always wanted to build with the global scalability and performance offered by Microsoft Azure.

Module Summary

Wow, we covered a lot in this module, isn't it? We looked at how organizations leveraging microservices architecture have a problem of managing these disparate services running in containers, and we needed a solution that can orchestrate the deployment, the management, the failover, and the high availability of the container host. Speaking of containers, we looked at the key benefits that containers offer, the lightweight design of the process and the isolation of an operating system, and a consistent and repeatable environment for both developers and system administrators. We saw how Kubernetes abstracts the entire data center as a single computational resource and offers portable, extensible, open-source platform for managing containerized workloads and services that facilitates both declarative configuration, that is desired state, as well as automation. While Kubernetes solves a lot of challenges with container management and orchestration, of course, well, it does introduce one challenge though. Who is going to deploy and manage Kubernetes? Well, thanks to Microsoft, Azure Kubernetes Service, or AKS, you get a managed Kubernetes platform that can be easily provisioned and scaled on demand, or you can also do that manually. You get a self-healed and a hosted control plane or the master Node, and you will only pay for the running worker Nodes. And finally, we saw how Microsoft Azure offers a whole gamut of services to help you run your containerized workload. Its rich ecosystem of services makes it the best place to deploy and run your application. Remember, it's not just about an individual service. It's about utilizing the entire suit of services that Microsoft offers that can help you make the most out of Azure. Now, I am pretty sure you must be itching to see AKS in action. So, see you in the next module where we will get our hands dirty with AKS.

AKS in Action

Module Introduction

Now that we have a solid understanding of Kubernetes and Azure Kubernetes Service, it's time for us to get our feet wet with AKS. This module is full of demonstrations where we will walk through a typical workflow of deploying an application to an AKS cluster. We will kick things off by reviewing the sample application we want to deploy to our AKS cluster, verify a few prerequisites we need to have in place in order to test our sample application. For the purpose of this course, we'll be using an ASP.NET Core Razor Pages application that is built using the .NET SDK. Let me emphasize. It doesn't matter what platform or runtime you use. It could be a Node application or a Ruby on Rails application. As long as you Dockerize the application, you should be good to follow along and deploy your application to an AKS cluster. With your application image ready, we will spin off a container based on this image and verify if our application is working as expected. We will then deploy this application to an AKS cluster using the Kubernetes deployment object, simulate a scaling event for our application, as well as the worker Nodes, update the application by deploying a new image of our application, see how we can roll back changes made, and then, finally, wrap this modular by looking up at the declarative approach of deploying applications to a Kubernetes cluster. There's a lot to cover in this module, so grab your favorite beverage and let's get Kubing.

A Quick Tour of Your Development Environment

I have installed the Docker Desktop for Windows and switched the configuration to use Linux containers. If you're following along on a macOS, you can download Docker Desktop for Mac. And on a Linux institution, you can use the appropriate package manager to install Docker on your box. Docker for desktop offers an easy way to set up a single-node Kubernetes cluster. All you need to do is to enable the checkbox to enable Kubernetes and hit Apply. This will install the necessary components. And once done, you have a single-node Kubernetes cluster available locally to deploy our applications into. However, since we are focusing on Azure Kubernetes Service here, you don't need to enable Kubernetes locally. We will let Microsoft take care of that for us. For the purpose of this course, I have created a sample ASP.NET application based on the .NET 5 Docker image. The code for the sample application, including the Dockerfile, is available on my GitHub repository, the link for which is up on your screen now. If you review the code, you will notice that there are four branches, version 1, version 2, version 3, and version 4, each corresponding to the new version of the application. You can very well clone the application and build a Docker image yourself using the docker build command. However, I have already created the images for the applications and their versions, and they are available in my Docker Hub repository. As you can see, there are four tags of this myapp Docker image, each corresponding to the application version we just described. Let's test our application by running a container based on the myapp Docker image. I'll use the docker container run command. Specify a name. Let's say myapp. Use the -d detach switch to run this app as a daemon or background process. Specify that I want to map the port 8081 of my host machine to the port 80 of the container. And finally, specify the name of the image, which, in our case, is manojnair/myapp:v1. This will pull down the necessary images from Docker Hub if not available locally. And once done, run a container based on that image. I can now connect to the port 8081 of my localhost. And as you can see, it serves up my web application. Pretty simple web application that displays the application version, whether we are running this application in a container, which we are right now, and shows the machine name, which is the container name of our application. Now to show you how this version, version v1, differs from the other versions, I'll run the same command we used earlier, but map port 8082 to version 2, 8083 to version 3, 8084 to version 4 of my application. Awesome. As you can see, I've created four versions of my application, each running as a Docker container. We will leverage these versions later in the module. Now, of course, you're welcome to use these images or feel free to utilize your own Docker images if you wish to do so. Now that we have all the prerequisites in place, let's go ahead and spin up a Kubernetes cluster in Microsoft Azure.

Deploying an Azure Kubernetes Service AKS) Cluster

Okay, this is the moment we have been waiting for. It's time for us to sprinkle some AKS magic. Let's create a resource, and under Categories, I will choose Containers and then select the Kubernetes Service. We will create a new resource group for our Kubernetes cluster. Let's call it aks-rg1, and let's call our cluster aksdemo1. Choose the region as Central US. Let's use the default version of Kubernetes, which, at the time of recording this course, is version 1.19.7. Note that I'm using a DS2 v2 as the VM size for the Node. This is actually a great choice if you need to take advantage of the enhanced networking capabilities that it offers, especially when you want your Kubernetes deployments to talk to Azure services like,

let's say, Azure MySQL, for example. Now for the Node count, we will set the Node count to 1. This is intentional because later in the course, we will scale the cluster Nodes to 2 using the Azure CLI. Now, if you're doing this in production, I highly recommend that you choose a value of 3 or higher. Clicking on Next will launch the Node pools blade where we will stick to the default. Next up is the Authentication blade, and here is where you can go the old school route of either creating a service principal manually or using a system-assigned managed identity. I do want to point out that using a system-assigned managed identity is a better choice as it alleviates the burden of renewing the service principal credentials to keep your cluster working. If you're new to the concept of managed identities, head over to my course Implementing Managed Identities in Microsoft Azure here, at Pluralsight. We will keep the RBAC enabled and choose the default encryption type and click on Next. In the Networking blade under Network configuration, we will choose Azure CNI, which stands for the Azure Container Networking interface. With Azure CNI, every Pod gets an IP address from the subnet, which gives the Pod full virtual network connectivity. The Pods can also be directly reached via their private IP addresses from connected networks. For example, let's say you have a virtual machine in the same VNet. It can access the Pod using the private IP address, and this communication works both ways. Since we don't have a VNet provisioned in my resource group, I'll create one. Leave the IP address ranges at the defaults and provide a DNS prefix of aksdemo1. Leaving the rest of the configuration as default, we will move on to integrations Now, we do want to use Azure Container Registry for our demos, but I will not create it now. We will create it later in the course. Let's use the default workspace created for monitoring our cluster in Azure Monitor. It's a good idea to tag your resources in production. But here, I will just skip it and go to the review screen. It will then run a final validation. And once passed, let's click on the Create button to kick off our cluster deployment. This is going to take a few minutes, and once done, we will have a single-node Kubernetes cluster in Azure already for us to deploy our applications into. To connect to our AKS cluster, let's switch back to PowerShell and use the Azure CLI to get our cluster credentials. First, let me verify that I have logged into the right subscription by using the az account show command. Now to set the default resource group for all subsequent Azure CLI commands, I'll use the az configure command and send the defaults group to aks-rg1. With that set, let's run the az aks get credentials command and specify the name of our Kubernetes Custer, which is aksdemo1. This will fetch the credentials and merge it into our current context so that we can repurpose the credentials for subsequent commands. Pretty neat, I must admit. Okay, now if you do not have kubectl installed, you can simply run the az aks install-cli command, and this will install the kubectl CLI on your machine. I already have kubectl installed since I've enabled Kubernetes option in my Docker for desktop settings. Okay, so to verify if we have the correct kubectl context, let's run the kubectl config current-context command. And as you can see, it says aksdemo1. Let's quickly run the kubectl get nodes command and verify if we have a single worker Node available. Since we haven't deployed any applications yet, we neither have any developments nor any Pods. So why don't we deploy some applications to this cluster in the next clip? See you there.

Deploying the Application to AKS Cluster Imperatively

With our AKS cluster ready, let's now deploy our application to it. Now, there are many ways to deploy an application to a Kubernetes cluster. You can either use the imperative way, which is basically firing a bunch of kubectl commands to create Kubernetes objects, and it's a great

way to test your applications, especially in a development environment. Alternatively, you can use a declarative approach where we define our Kubernetes objects as YAML or JSON manifests, and this is what is generally practice in most production scenarios. I will show you an example of using a declarative approach later in this course. As I want to keep the focus on AKS and get you up to speed as quickly as possible, we'll be using the imperative way to create Kubernetes objects in most of the demos. All right, so let's run our first application. Our mission is to create a deployment that uses the image we have in my Docker Hub repository. That is the manojnair/myapp:v1. To do so, let's run the command kubectl create deployment. We will specify the deployment name as myapp and the image as manojnair/myapp:v1 and the number of replicas as 1. As you can see, a deployment is created. Now to verify if a deployment was indeed successful, let's run kubectl get deployment. We see that we have a myapp deployment with 1 out of 1 Pods ready. Let's also run kubectl get pods to see the Pod that is powering the deployment. Awesome. Now, we have our app deployed, but how do we connect to our application, especially from outside of the Kubernetes cluster? Interesting question, isn't it? This is where we can take advantage of services in Kubernetes. With AKS, when we create a service of type LoadBalancer, it creates an Azure load balancer that will get a public IP, and we can use the load balancer to connect to our application. To create a service imperatively, we will use the kubectl expose command, kubectl expose deployment. Specify the name of the deployment, which, in our case is myapp, and specify the type as LoadBalancer. We will connect to port 80 of our service, which is our load balancer, to the port 80 of the container. Now, before we run this command, let's open a new PowerShell session. And to keep things a bit more interesting, let's run kubectl get svc command with the --watch switch. This will monitor the creation of the service and also watch for events as they unfold, including creation of a new service and the assignment of the public IP. All right, so let me run the expose deployment command. And as you can see, the kubectl get service command shows our myapp service created. And after a while, it also shows the external IP it got assigned by Microsoft Azure. Great. Time to connect to our web application. While in PowerShell, we can simply type the Start-Process command and specify the URL of our application, which, in this case, is http followed by the external IP address of our load balancer. This will launch the default browser, which, in my case, is Chrome, and there we go. We have our application up and running. Beautiful.

Scaling the Deployment Manually

Let's run kubectl get deployments. And as you can see, we have one deployment called myapp, and it has one replica or a Pod. Please note that we still have our service, myapp, that exposes the deployment using a load balancer To simulate a scaling event, let's use kubectl scale command to scale our deployments to three replicas. Okay, so kubectl scale deployment. We'll specify the deployment name, which is myapp, and then set the number of replicas to 3. That's it. Now, let's run kubectl get deployment, and as you can see, we now have 3 replicas instead of 1. If we now connect to our service's public IP address using a different tab or, let's say, incognito windows, you will see that our application is currently powered by three different Pods or replicas. Let me show you something interesting. Let's say we want to scale our application to use 100 Pods or replicas. We run kubectl scale deployment myapp --replicas and set the value to 100. Now, to monitor our scaling operation, let's use the kubectl get deployments with the --watch switch. You will notice that the number of Pods in

our deployment is gradually increasing, and I'm going to give this some time. And let us reach the steady state of 100 out of 100 Pods being available. Hmm. It is stuck at 99 out of 100. Or, in other words, 99 Pods out of the desired 100 Pods are ready. So the question that needs to be answered is why is the deployment stuck at 99 out of 100 Pods? Now, Pods run containers, and each container consumes vCPUs and memory. So the obvious explanation to this mystery is that Kubernetes is trying its best to be bend back as many Pods as possible, which in turn runs containers on the DS2 v2 Node that we have in our Kubernetes cluster. There is one more important contributing factor. Our Kubernetes cluster only has one Node of type DS2 v2, and the maximum number of Pods it supports is 110. Well, how did I get that number? Easy. We can use the az aks nodepool show command and provide the cluster name and the pool name and query the maxPods property. All right, so you might wonder, Manoj, we are just asking for 100 Pods. Isn't that less than 110? So what's the problem here? That's a great question. Let me explain. Kubernetes uses some system Pods for its own operations. You can see those Pods when you append the --namespace switch and use the kube-system namespace. Let me pipe this to Powershell's Measure command, and now we can see that we have a count of 12. Now this also includes the header, so there are technically 11 Pods. So now if I run kubectl get pods, but this time I'm going to append the --all namespaces switch, I'll pipe this further to PowerShell's Measure command, and now we get 112 - 1, which is the header, so technically 111 Pods. So, Kubernetes is trying its best to bend back as many Pods as possible, including the system ones, that it needs to operate on the single worker Node that happens to be DS2 v2, and that has a maximum gap of 110 Pods. So the question is how do I get 100 Pods for myapp deployment? Well, you guessed it right. We need to add another worker Node to our cluster, which is what we will do next.

Scaling the Nodes Manually

Everything in Azure translates to an API call. This is true for Azure Kubernetes Service as well. With just a single command, you can scale the number of Nodes running in the AKS cluster. Let's check the number of Nodes running in our AKS demo1 cluster by using the command kubectl get nodes. We currently only have one Node running, which happens to be a worker Node. To scale the number of Nodes manually, we can run the command az aks scale, specify the resource group and then the name of the cluster, and, finally, the desired number of Nodes, which, in our case, is going to be 2. We will also add the --no-wait switch so that we don't have to wait for this long running operation to complete before we get a prompt back. To query the status of the operation, we can use the command az aks nodepool show. Specify the name of the Node pool, which, in this case, is agentpool, and the cluster name which is aksdemo1, the resource group, and, finally, the query switch to query the Node count and the provisioning state. This will take a few moments. And once done, we will have our AKS cluster scaled to a two-Node cluster. We can easily verify this using the kubectl get nodes command. And as you can see, now we have two Nodes in our Kubernetes cluster. Now that we have scaled our cluster, let's actually go back and check the status of the deployment that we did in the last clip. Remember, our deployment was stuck at 99 Pods as we maxed out on the number of Pods per Node. Now that we have two Nodes, we should be able to accommodate up to 222 Pods collectively. So now, when we run kubectl get deployments, you can see that our application, myapp, has been successfully scaled to 100 Pods. Pure awesomeness. Let's scale it back to 3 replicas as we really don't need 100 replicas for our

application, kubectl scale deployment. We'll set the replicas count to 3. And finally, kubectl get deployments just to confirm that we have three Pods powering our deployment.

Updating the Application

If you recollect from our previous discussion, we have four versions of our application, v1, v2, v3, and v4. Let's say we have a need to update the current version of our application to version 2. Essentially, what we need to do is to update the image used by our deployment. There are three ways to do so. I'll show you a couple of options here in this clip and reserve the third option for a future clip. Before we talk about updating our applications, let me first draw your attention to rollout. A rollout is used to manage the rollout of Kubernetes resources, one of them being a deployment. You need to instruct the Kubernetes API to record every change you make to the rollout, and this is typically done by appending the --record switch, which is set to true. Let me show you the current rollout history of the myapp deployment. You will run the kubectl rollout history command, and we are tracking the deployment myapp. Now let's capture this output, and we will come back to this later. Let's start by describing a deployment using the kubectl describe deployment myapp. Notice that in the Pod template, we're using the object myapp, and the image used by the myapp container is manojnair/myapp:v1. The first method of updating the image used by a deployment is to use the kubectl set image command. Let's add in kubectl set image deployment, specify the myapp deployment, and we will set the myapp container's image to manojnair/myapp:v2. And we will also append the --record is equal to true switch to track the rollout status later. So the image has been updated now. Let's quickly verify that using the kubectl describe deployment command. Awesome. Let's connect to our service using the browser. And as you can see, we have now version 2 of our application being serviced by our deployment, myapp. Alternatively, you can also use the kubectl edit deployment command, specify the deployment name, which, in our case, is deployment/myapp and along with our --record is equal to true to capture the rollout. This allows us to edit the deployment API resource using the editing tools, which, in Windows, by default, is set to Notepad. And hence, you see this Notepad window popping up. Now, all I need to do is to scroll down to the image section here and change the image to manojnair/myapp:v3. That's it. Save the Notepad file, and voila, the deployment has been updated. Awesome. And as usual, let's connect to our service. And as you can see, we are now presented with the version 3 of our application. Now, as an exercise, I want you to try updating the image with the version 4 of this application with any of the two methods that we discussed in this clip. Make sure you add the --record is equal to true to keep track of the deployment revisions, which is something that we'll talk about next.

Rolling Back the Application to Previous Versions

Let's run kubectl rollout history and specify our deployment, myapp. As you can see, we see a history of changes made to our deployment and the corresponding revision numbers. The last change we have made is to set the image to version 4. To revert to the previous change, we can use kubectl rollout undo deployment/myapp. This will undo our latest change, which, in our case, is going to update the image from version v4 and bring it back to its previous state, which is version v3. Let's verify that by describing a deployment. And now, let's connect to

our service. And surely enough, we can see that our application has been reverted back to version 3. Now if you run kubectl rollout history command, you will see that we no longer have revision 3. And we can see a new revision, revision 5, which is basically pointing to the latest change we made, which is updating the application to use the v3 image. If you wanted to revert to a specific revision number, let's say to revision 1, you can use the same command, but add the --to-revision parameter and specify the revision number, which, in this case, let's say, is revision 1. As you can see, we have now reverted to version 1 of our application.

Using the Declarative Approach to Deploy Kubernetes

Let's create a new deployment called myapp2 with the v2 version of our application. But this time, let's use the declarative approach. To do so, we will need to create a YAML file that defines the Kubernetes deployment object. We start with the API version, which defines the version of the Kubernetes API we'll be using to create our deployment object. Next up is the kind that defines the kind of object we want to create. In our case, that is a deployment. Metadata, as the name suggests, is used to provide data that helps us uniquely identify the object, including a name, which, in our case, we're going to set it as myapp2. Spec defines the desired state of the Kubernetes object we are defining. In case of a deployment, this would be the number of replicas of what we need, the selector, that is used to define what Pods we need to manage as a deployment, and, of course, the Pod definition itself. In our case, we need three replicas. For the selector, we will match all Pods that are labeled as myapp2 to be managed by our deployment object. And for the Pod definition, let's specify the template where we provide the container image as manojnair/myapp:v2. We're going to use the version 2 of our image and the port as 80. So now, we have our deployment YAML ready. But as you already know, we need to expose this deployment as a service, which can also be done declaratively. The service object is defined in the v1 apiVersion. Kind is Service. And for the metadata, let's define the name as myapp2. Now, what really changes here is the spec. Here, we define the selector and the labels that will be used by the service to load balance the Pod. In this case, all Pods that have the label app: myapp2 will be load balanced by the service we are defining here. We will define the type as LoadBalancer and specify the target port and the port as we did with the kubectl expose command. Great. So now we have both the service and the deployment YAML file created. The good thing about YAML is that we can combine them into a single file, let's call it myapp2.yml, and separate our deployment and service objects using three dashes. To create a deployment and service, we can use the kubectl create command and specify the myapp2.yml file. As you can see, it creates both the deployment and the service. And to check our deployment status, let's run the kubectl get deployment command. And now we have 3 out of 3 Pods running for our myapp2 deployment. Great. To check our service status, let's run kubectl get service, and we can see that our myapp2 service has a public IP assigned. Let's open up a browser and connect to that public IP. Beautiful. We can see our application is running version 2 of our myapp. And if you open up a new tab or an incognito window, you will see that the machine name changes as our service load balances between the three Pods of our deployment. The good thing about declarative approach is that now you can define your Kubernetes objects as YAML files, source control them, build your objects as a part of the CI/CD pipeline, similar to the concept of Infrastructure as Code. Now, what if I wanted to scale the number of replicas to 10 or update the image to use the version 4 of our application? Easy. Just make those changes

in the YAML file and then use the kubectl apply command and specify the updated YAML file. That's it. As you can see, our deployment has now 10 replicas, and our service now resolves to version 4 of our application. Before we move on, let's clean up our environment and get rid of the second deployment that we created, as well as the service that we created. Fortunately, that is pretty simple to do so. All we need is to use the kubectl delete command and provide the file name that we used to actually create the deployment, as well as the service, which, in our case, is myapp2.yml. As you can see, it deletes both the myapp2 deployment and the service.

Pushing the Image to the Azure Container Registry (ACR)

We have been using the Docker Hub repository to host our container images. It would be great if we can host our images in a native Azure service. Thankfully, Azure provides a robust registry service called Azure Container Registry. Integrating ACR with AKS will give us the best of both worlds. We cannot only secure our images as they remain private to our environment, but also take advantage of the managed identity assigned to the AKS cluster to pull images of this registry. Let's create an Azure Container Registry by clicking on Create a resource, Containers. Choose Container Registry, and then click on Create. Here, we will provide a registry name. Notice how the complete name of a repo is the name of the registry followed by azurecr.io We will choose the same location where our AKS cluster is hosted. For SKU, we will choose Basic. The higher tier you choose, better will be the performance and the scalability. Networking blade is pretty much grayed out as we are using the basic SKU. I highly recommend you use the premium SKU in production as it provides a private endpoint for the Azure Container Registry, and same is also true for encryption as well. Ignoring the Tags blade, let's run a validation of our inputs. And once done, let's click on Create. I will give this a minute as it deploys our container registry. Now, we need to give our AKS agent pool or, in other words, our Kubernetes worker Nodes the permission to pull container images from this registry, which is our Azure Container Registry. To do so, let's go to the Access control blade of the resource, click on Add, and then choose Add role assignment. For role, we need to only allow our agent pools to pull images off this registry. So the reader role is more than sufficient. Since we're using a system-assigned manage identity, all we need to do here is to type our AKS cluster name, which is aksdemo1, and choose the aksdemo1-agentpool identity. Let's click on Save, and that's it. Our AKS Node pools can now read images off the container registry. Right now, our container registry is empty and has no images. So let's push the v1 image of our myapp application to this registry. To do so, first, let's run the docker image ls to list our Docker images. Now, let's tag the v1 image, which is the login server details of our ACR, docker tag manojnair/myapp:v1 followed by the login server name and the image name and the tag. Let's confirm if the tagging was successful by running the docker image ls command again. And as you can see, we do have our image tag with the login server name. Now, we need to push this image to the registry. Azure CLI makes this very easy by using the az acr login command. So az acr login, and I will specify the name of the registry. Once our login is successful, all we need to do here is to use the Docker push command to push the tag image to our Azure Container Registry. Awesome. We can see that our repository is now available in the registry, along with the v1 tag. Now comes the litmus test. Let's now create a deployment. But this time, instead of using our regular Docker Hub image, let's use the Azure Container Registry image. Remember that this image is not available publicly. And as of now, only my Azure user account and the agent pool Nodes

can pull this image from this registry. So kubectl create deployment, let's call this myappacr, and the image as our ACR login server followed by the image name and the tag. As you can see, the deployment has been created. And to verify the status of the deployment, let's run the kubectl get deployment myappacr. Now, let's expose the deployment using the kubectl expose command followed by the deployment name. Type as the LoadBalancer. Target port and port set to 80. Once we get the public IP assigned to our service, let's connect to it via a browser. And boom, the version 1 of our application is now being served, the image of which is now securely stored in our Azure Container Registry or ACR. Pure awesomeness.

Module Summary

Didn't I say this module is going to be full of hopefully interesting demonstrations? We covered a lot of ground in this module and saw AKS in action. We started the proceedings by reviewing our sample application, spun a Docker container based on that application's Docker image. Next, we created a Kubernetes deployment of our application manually using the imperative approach. We saw how to scale our application by adding more replicas manually and then added Nodes to a worker pools to accommodate for future groups, such as adding more Pods or replicas. Next, we used the set image deployment and the kubectl edit commands to update the image used by our application and then used the Kubernetes rollback command to roll back changes made to our deployments. Ask any DevOps engineer how awesome this feature is, and I'm sure she will nod her head in agreement. Finally, I gave you a glimpse of how you can use the declarative approach to create Kubernetes objects, which can be integrated into your DevOps pipelines. Let's now move to the final module of this course.

Next Steps

Next Steps

Congratulations! You have made it. Well done. As always, thank you so much for your time, and I hope I was able to get you excited about both Kubernetes, as well as AKS. I'm sure you must be excited to dive a bit deeper in AKS, so I wanted to leave you with some additional references and features you might be interested in to explore on your own. AKS will provide you with a Kubernetes farm, but you need to have a good handle on Kubernetes. Thankfully, the Pluralsight course library has crafted some awesome learning paths on Kubernetes. Start with the Certified Kubernetes Application Developer path, even if you're not interested in the CKAD certification. This path will go a bit deeper into the concepts we touched upon in this course and will cover topics like Pod health, understanding storage options in Kubernetes, especially around persistent volumes, persistent volume claims, and also managing secrets and configuration using secrets and config maps in Kubernetes. Once you get a good handle on Kubernetes, check out some other courses in the Pluralsight library on the Azure Kubernetes Service. I do highly recommend watching Mike's, Anthony's, and James's courses on this topic. They are awesome. Finally, take a look at the Azure Kubernetes Service workshop on the Microsoft Learn portal. This workshop will help you deploy a multi-container application to AKS, including setting up SSL and TLS using ingress controllers

with App Gateway. Well, I reckon that's about it. Feel free to reach out to me on Twitter or on LinkedIn, and let me know how you found the course and how I can improve my future courses. I hope you continue on the journey on exploring the awesomeness of Azure Kubernetes Service. On that note, keep calm and az aks.