



Introducing Azure Kubernetes Service

A Practical Guide to
Container Orchestration

Steve Buchanan
Janaka Rangama
Ned Bellavance

*Foreword by Brendan Burns,
Distinguished Engineer, Microsoft*

apress®

Introducing Azure Kubernetes Service

**A Practical Guide to Container
Orchestration**

**Steve Buchanan
Janaka Rangama
Ned Bellavance**

***Foreword by Brendan Burns,
Distinguished Engineer, Microsoft***

Apress®

Introducing Azure Kubernetes Service: A Practical Guide to Container Orchestration

Steve Buchanan
Plymouth, MN, USA

Janaka Rangama
Victoria, VIC, Australia

Ned Bellavance
New Britain, PA, USA

ISBN-13 (pbk): 978-1-4842-5518-6
<https://doi.org/10.1007/978-1-4842-5519-3>

ISBN-13 (electronic): 978-1-4842-5519-3

Copyright © 2020 by Steve Buchanan, Janaka Rangama, Ned Bellavance

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Coordinating Editor: Jill Balzano

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at <http://www.apress.com/bulk-sales>.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484255186. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

Steve would like to dedicate this book to his wife Aya and kids Malcolm, Sean, Isaac, and Jalen for the support on projects like this.

Janaka would like to dedicate this book to his mother Herath Menike, his wife Aloka, and his daughter Omandi for being the three musketeers in his life.

Ned would like to dedicate this book to his wife Andrea and his kids James, Tess, and Genevieve for their support, encouragement, and patience.

Table of Contents

About the Authors.....	xi
Acknowledgments	xiii
Foreword	xv
Introduction	xvii
 Chapter 1: Inside Docker Containers	 1
The Value of Containers	1
What Is Docker	2
Containers vs. Virtual Machines.....	3
Images and Containers	6
Docker Components (Networking and Storage).....	6
Networking	6
Storage	7
Installing Docker	8
Docker Command Cheat Sheet	10
Understanding the Dockerfile	12
Understanding Docker Build	13
Understanding Docker Compose.....	13
Running a Container	15
Orchestration Platforms	15
Summary.....	16

TABLE OF CONTENTS

Chapter 2: Container Registries..... 17

Overview of Container Registries..... 17

 Registries, Repositories, and Images 18

 Private and Public Registries and Repositories..... 18

 Basic Registry Operations 20

 Image Tagging 25

Common Registries..... 26

 Docker Hub and Docker Registry..... 26

 Azure Container Registry..... 27

Azure Container Registry Expanded..... 28

 Security 28

 Permissions..... 29

 Tasks and Automation 30

 Azure Kubernetes Service Integration..... 33

Summary..... 33

Chapter 3: Inside Kubernetes 35

Kubernetes Interfaces..... 37

Docker Runtime 38

Master Nodes Overview 38

Worker Nodes Overview..... 40

Namespaces 40

Labels and Annotations..... 41

Pods 43

Replicasets..... 43

DaemonSets..... 44

Jobs 44

Services 44

Deployments 45

ConfigMaps	45
Secrets	47
Networking.....	47
Storage.....	48
Summary.....	50
Chapter 4: kubectl Overview	51
Introduction to kubectl.....	51
kubectl Basics	53
Common Operations with kubectl	58
Summary.....	62
Chapter 5: Deploying Azure Kubernetes Service	63
Azure Kubernetes Service Deployment Overview.....	63
Deployment Through the Azure Portal.....	63
Deployment Through Azure CLI	70
Deployment Through Azure Resource Manager Templates.....	72
Create an SSH Key Pair	72
Create a Service Principal	72
Using an Azure Resource Manager QuickStart Template	73
Deployment Through Terraform	76
Connecting to Your AKS Cluster	76
Summary.....	77
Chapter 6: Deploying and Using Rancher with Azure Kubernetes Service	79
What Is Rancher?	79
Why Use Rancher with Kubernetes?	80
How to Deploy Rancher on Azure.....	81
Authenticate Rancher with Azure Active Directory	90
Deploy AKS with Rancher.....	92
Summary.....	99

Chapter 7: Operating Azure Kubernetes Service 101

Cluster Operations in Azure Kubernetes Service 101

 Manually Scaling AKS Cluster Nodes..... 102

Scaling Azure Kubernetes Service 110

 Manually Scaling Pods or Nodes 110

 Automatically Scaling Pods or Nodes 111

Storage Options for Azure Kubernetes Service 115

 Volumes 116

 Persistent Volumes 116

 Storage Classes 117

 Persistent Volume Claims 117

Networking in Azure Kubernetes Service 119

 Kubenet vs. Azure Container Networking Interface (CNI) 119

 Network Security Groups and Network Policies 122

Access and Identity in Azure Kubernetes Service..... 122

 Kubernetes Service Accounts..... 122

 Azure Active Directory Integration 123

 Azure Role-Based Access Controls (RBACs) 123

 Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings 123

 Control Deployments with Azure Policy (Preview) 124

Security Concepts in Azure Kubernetes Service 128

 Master Security 128

 Node Security 128

 Cluster Upgrades 130

 Kubernetes Secrets 130

Monitoring Azure Kubernetes Service..... 131

 Azure Monitor for Containers..... 131

Business Continuity and Disaster Recovery in Azure Kubernetes Service.....	145
Thinking About SLAs and What You Need.....	146
Data Persistence and Replications.....	146
Protecting Against Faults.....	147
Summary.....	149
Chapter 8: Helm Charts for Azure Kubernetes Service.....	151
Helm Overview.....	151
Use Cases.....	152
Advantages over Kubectl.....	152
Key Components.....	153
Cloud Native Application Bundle.....	155
Installing Helm on AKS.....	155
Requirements.....	156
RBAC and Service Account.....	156
TLS Considerations.....	157
Helm init.....	159
Helm Charts.....	163
Chart Contents.....	163
Chart Repositories.....	170
Deployment Process.....	171
Creating a Helm Chart.....	173
Deploying a Helm Chart.....	178
Updating a Release.....	181
Removing a Release.....	184
CI/CD Integrations.....	185
Automating Deployments.....	185
Testing Helm Charts.....	186
Unattended Helm Chart Installs.....	187
Integrating Helm with Azure DevOps.....	187
Summary.....	189

TABLE OF CONTENTS

Chapter 9: CI/CD with Azure Kubernetes Service 191

 CI/CD Overview 192

 Continuous Integration..... 193

 Shared Repository 194

 Build Pipeline..... 195

 Continuous Delivery/Deployment..... 201

 Release Pipeline 202

 Testing..... 212

 Unit Testing 213

 Integration Testing 213

 System Testing 213

 Acceptance Testing..... 214

 Dev Spaces..... 214

 CI/CD Best Practices with AKS 216

 Cluster Operators..... 216

 Application Developers..... 218

 Summary..... 219

Index..... 221

About the Authors

Steve Buchanan is an enterprise cloud architect and Midwest Containers Services Lead on the Cloud Transformation/DevOps team with Avanade, the Microsoft arm of Accenture. He is an eight-time Microsoft MVP and the author of six technical books. He has presented at tech events, including Midwest Management Summit (MMS), Microsoft Ignite, BITCon, Experts Live Europe, OSCON, and user groups. He is active in the technical community and enjoys blogging about his adventures in the world of IT on his blog at buchatech.com.

Janaka Rangama is a Microsoft Azure MVP and a Microsoft Certified Trainer. He is originally from Sri Lanka, “the Pearl of the Indian Ocean,” and now lives in Australia, “the Land Down Under.” Currently a Senior Principal Product Technologist at Dell EMC Azure Stack Product Engineering Team, he is one of the leading hybrid cloud experts in the APAC region. He is a well-known speaker in many international conferences and an expert in both Microsoft and OSS technologies. He co-leads the Melbourne Azure Nights user group and is one of the founding members of the Sri Lanka IT PRO Forum.

Ned Bellavance is a Microsoft Azure MVP and Founder of Ned in the Cloud LLC. As a one-man tech juggernaut, he develops video courses, runs the Day Two Cloud Podcast for Packet Pushers, and creates original content for technology vendors. He is passionate about learning new technologies and sharing that knowledge with others, whether that is through courses, speaking, blogging, or authoring books. He has presented at tech events, including Microsoft Ignite, Cloud Expo NYC, and the Midwest Management Summit (MMS). You can find his musings on the IT industry at his web site at nedinthecloud.com.

Acknowledgments

Steve would like to thank the co-authors Ned and Janaka for taking on this project, the tech reviewers Mike Pfeiffer and Keiko Harada, Brendan Burns for writing the foreword, and the Microsoft teams who do all the cool container things in Azure!

Janaka would like to thank the co-authors Ned and Steve for encouraging him to become part of this book, Keiko Harada (Senior Program Manager, Microsoft Azure) and Nirmal Thewarathanthri (Cloud Solutions Architect, Microsoft Australia) for their continuous guidance and support through his Kubernetes journey, and the Microsoft Azure Product group for their amazing work to augment humanity with the intelligent (cloud + edge).

Ned would like to thank Nigel Poulton for getting him tangled up in the Kubernetes mess, Steve Buchanan for encouraging him to be part of the book, and Justin Luk for sharing his team's knowledge and insight.

Foreword

Kubernetes has revolutionized the way that people approach building and operating distributed systems. Over the last five years, Kubernetes has gone from a small open source project to a ubiquitous part of a broad cloud-native landscape. Kubernetes enables application developers to de-compose their monolithic applications into smaller “two-pizza” teams which radically accelerates autonomy and agility in software development. Additionally Kubernetes includes capabilities for online, self-healing management of applications that also makes distributed systems on Kubernetes more reliable too.

Kubernetes is a critical component of modern application development and digital transformation for many organizations. But it is also a distributed system unto itself. This means that the care and feeding of a Kubernetes cluster is a complicated endeavor. This is made even more complex by the rapid pace of change in the Kubernetes ecosystem, with new versions of Kubernetes released every three to four months and patch releases with fixes and security updates pushed even more quickly.

Because of the complexity of managing your own Kubernetes cluster, consuming it as a managed cloud service becomes a very attractive option. In Microsoft Azure, the Azure Kubernetes Service (AKS) is a managed service for “Kubernetes as a Service.” With AKS, users can harness the power of the Kubernetes API while having the confidence that Azure is ensuring that their clusters are healthy and stable. When updates come, AKS performs extensive testing and vetting of the release to ensure that a user of AKS can upgrade to the latest fixes ensuring that it will work properly for their application. AKS also deeply integrates into the Azure ecosystem and core technologies like Azure Active Directory (AAD). For most people, Kubernetes is only a part of their overall Azure usage, and this integration means that the rest of their digital estate can seamlessly integrate with the Azure ecosystem.

FOREWORD

Whether you are just getting started or a Kubernetes expert, *Introducing Azure Kubernetes Service* is a great resource for ensuring that you get the best out of managed Kubernetes on Azure. I'm grateful to Steve, Janaka, and Ned for providing our users with such great reference material. In Azure, we work tirelessly to ensure that we meet our customers where they are and set them up for greater success.

This book shares those goals and will help you achieve your goals with Kubernetes and Azure. Enjoy!

Brendan Burns
Distinguished Engineer, Microsoft

Introduction

This book is a practical guide to Microsoft's Azure Kubernetes Service (AKS), a container orchestration platform. The goal of this book is to take the reader from 0 to 100 deploying and running a Kubernetes cluster on Microsoft Azure cloud. For anyone embarking on this book, it is ideal to have experience in the IT industry in system administration, DevOps, Azure cloud, or development. Some Docker experience would also be helpful but not required.

This practical guide on AKS scales back on theory content, giving just enough to grasp important concepts while focusing on practical straight to the point knowledge that can be used to go spin up and start running your own AKS.

The book will take the reader on a journey inside Docker containers, container registries, Kubernetes architecture and components, and critical Kubectl commands, along with the deployment and operation of Azure Kubernetes Service including topics such as using Rancher for management, security, networking, storage, monitoring, backup, scaling, identity, package management with HELM, and finally Kubernetes in Continuous Integration and Continuous Delivery/Deployment (CI/CD).

CHAPTER 1

Inside Docker Containers

Welcome to *Introducing Azure Kubernetes Service: A Practical Guide to Container Orchestration*. Before diving into Azure Kubernetes Service, it is important to understand the building blocks and road leading up to Kubernetes and finally Azure Kubernetes Service.

This chapter is not a deep dive into Docker and building applications with Docker. Docker is a large topic and can fill an entire book. The goal of this chapter is to give both those who are not familiar with Docker enough knowledge to get started and those that are familiar with Docker a refresher as must have knowledge as a prerequisite to Kubernetes.

In this first chapter, we are going to dive inside Docker containers. By the end of this chapter, you will have a greater understanding of Docker; images; containers and their value; the underlying Docker components; how to install Docker, run Docker commands, and build Docker images; and Docker Compose and finally an introduction into orchestration platforms.

The Value of Containers

Containerization is not new. In fact, container technology has been around in the Linux world since the 1980s. Containers however only become widely popular with tremendous growth in part due to the launch of the Docker container format in 2013.

Containers are an abstraction of the application layer in an isolated user-space instance. Containers share the operating system (OS) kernel storage and networking from the host they run on. Containers can be thought of like the core components needed for an application running as a process. Containers allow the packaging of an application and its dependencies running in an instance. Containers allow software engineers to develop their applications and replicate across environments such as dev, stage, and prod in a consistent manner. Containers can move through Continuous

Integration and Continuous Deployment pipelines in a manner that keeps the OS, dependencies, and application unchanged, providing ultimate flexibility and agility.

The value of containers from a technical standpoint can be summed under the following: greater density of applications on the same hardware than any other technologies, optimization for developers resulting in improved developer productivity and pipelines, operational simplicity, and cloud and infrastructure agnostic for true platform independence and ultimate portability.

The value of containers to the business are lower TCO, increased speed and time to market, higher customer satisfaction, predictability and dependability, increased agility, and improved operational velocity.

What Is Docker

There are many container formats available; however, Docker has become the de facto standard for both Windows and Linux containers. Docker is an open source container format. Docker can be used to build, run, and store container images and containers. Per Docker, 3.5 million + applications have been containerized. And according to the RightScale 2018 State of the Cloud report, Docker adoption in 2018 rose from 35% to 49%. Let's dive into the Docker components:

- Docker Engine is the core of the Docker solution. It is a Client-server application with the following components:
 - Docker Client is the way users interact with Docker. Docker Client comes with a command line interface (CLI) in which users execute Docker commands. Docker Client can run on the same computer as a Docker daemon or a client computer and connect remotely to a Docker daemon.
 - Docker host runs the Docker daemon. The Docker daemon is a background process that manages Docker images, containers, networks, and storage volumes. The Daemon listens for commands on a REST API or receives commands via the CLI. It also can communicate with other Docker daemons to manage Docker services.

- Docker registry is a repository service where you can host and download container images. A Docker registry can be local, public, or private. Docker has a public registry service named Docker Hub. Most cloud providers offer private Docker registries.
- Docker Objects
 - Docker images are read-only templates used to build Docker containers. These contain the instructions for creating a Docker container. Images include the application code, runtime, system libraries, system tools, and settings.
 - Docker containers are simply the images running at runtime. Docker containers run on the Docker Engine.
 - Docker services allow container scaling across multiple Docker daemons. These multiple daemons act together as a swarm with multiple managers and workers.

Docker is available in two editions:

- Community Edition (CE)
 - CE is a good option for developers, small teams, and anyone starting out with containers.
- Enterprise Edition (EE)
 - EE is good for enterprise teams and production needs.

It is important to know that Kubernetes supports multiple container runtimes; however, overall Docker is the most common image and container format in the tech space today. It is worth investing some time into diving deeper into learning Docker.

Containers vs. Virtual Machines

In IT for a while now, virtual machines (VMs) have pretty much been the standard when there is a need to stand up a server to run an application. Virtual machines require a hypervisor to run on. There are many hypervisors but the popular ones are VMWare and Hyper-V. The hypervisor is installed on top of a physical machine, and then virtual machines are deployed on top of the hypervisors. This allowed the IT industry to pack many virtual machines on physical servers increasing the density and getting more ROI

out of physical hardware. Virtual machines emulate physical servers including storage, networking, and an operating system. They are more portable and faster than physical servers but are still full servers requiring boot up time and the same level of management as physical servers.

Containers take the density and optimizing to the next level. Containers are still a form of virtualization but only virtualize what is core to running an application. With containers, there is no need for a hypervisor as they run directly on the kernel. You can pack many more containers on a physical server. Containers are more lightweight and boot up faster, and the management is streamlined.

With containers some of the underlying components are shared across all the containers running on a host such as storage and networking. Figure 1-1 gives a visual representation of the differences in the architecture between containers and virtual machines.

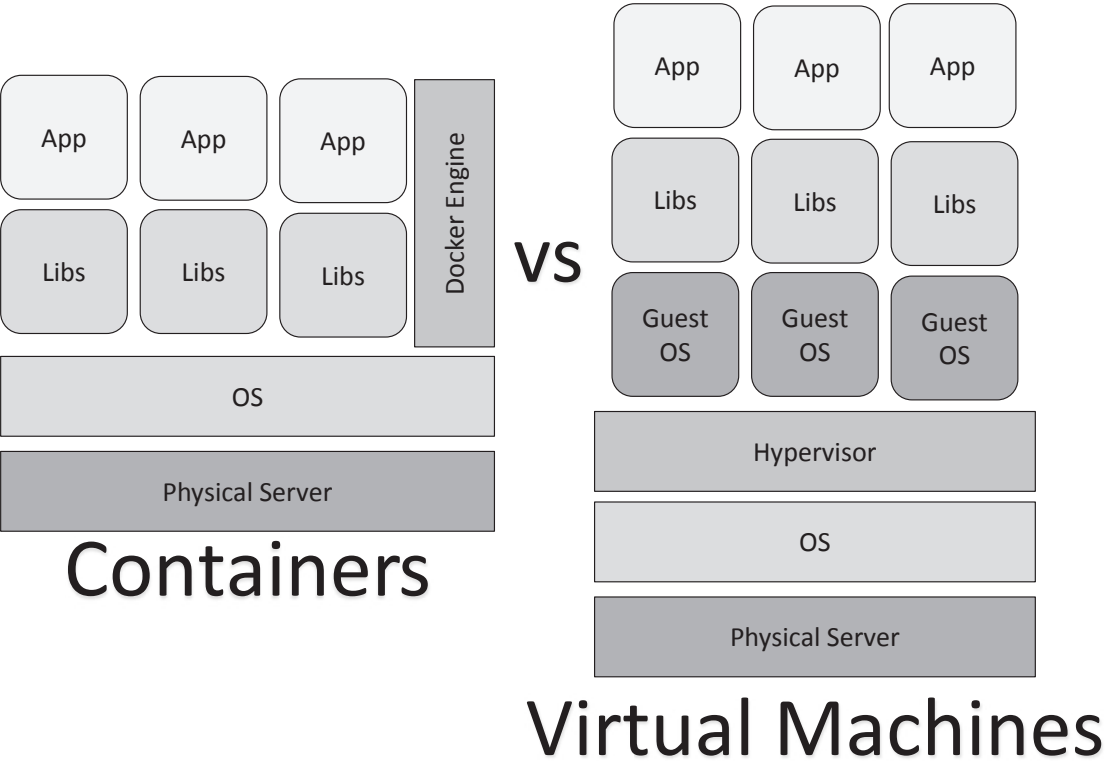


Figure 1-1. Containers vs. virtual machines

As you can see from the image, containers are isolated at an OS-level process, whereas virtual machines are isolated at the hardware abstraction layer. The growth of containers does not mean that virtual machines will go away anytime soon. There are reasons you would use containers over virtual machines due to the benefits. Let's look at some of the reasons you would use containers:

Speed: Docker containers are extremely fast compared to virtual machines. It can take a container anywhere from a few milliseconds to a few seconds to boot up, while it will take a virtual machine at least a few minutes to boot up.

Portability: Containers can be moved and shared across multiple teams, development pipelines, cloud, and infrastructure with the application behaving the same wherever the container runs. This reduces human errors and potential environmental dependency errors.

Microservices: Containers are a good way to decouple and run an applications component to support a microservices-based architecture.

Now let's look at some reasons you may still want to use virtual machines over containers.

Stateful: If you have applications that need state, virtual machines might be a better fit because containers were designed to run stateless applications.

Co-located: If an applications component must all be installed together on the same server, a virtual machine will be a better option as a focus of containers is often to break out an application's services across multiple containers.

With the increase in containers, the footprint of virtual machines will decrease. However, virtual machines are not going to disappear as there are still use cases for them and many workloads today are running just fine on virtual machines.

Images and Containers

Earlier in this chapter, the differences between container images and containers were briefly covered. Let's dive in a little deeper as to what images and containers are. In a nutshell, a container is simply a running instance of an image.

Images are read only. Containers can be modified with changes, but those changes are lost when the container stops. Changes to a container can be retained if they are committed to a new image. Images are a representation of the code, runtime, filesystem, libraries, and settings. An image is a set of commands in a file named Dockerfile that defines the environment inside a container. Listing 1-1 is an example of a simple image Dockerfile that runs on an Ubuntu Linux OS and executes a command that will output Hello World!.

Listing 1-1. Dockerfile content

```
FROM ubuntu:latest  
CMD echo Hello World!
```

After a Dockerfile is built, the docker build command is used to build the actual image. Built docker images are stored locally by default and can be run as a container from there or pushed to a Docker registry. Docker images get a unique ID by default but can be named and tagged. That wraps up this summary of Docker images and containers. Later in this chapter, we will explore the Dockerfile in more detail, using docker build and running a container.

Docker Components (Networking and Storage)

Networking

In your container journey, you will get to a point where you need to expose it to the outside world or you may need to connect several containers together either on the same host or across other hosts. Docker containers have networking options available to fit all scenarios. There is a layer of networking in container technology for the containers to communicate with other containers, the host machine, and the outside world. Docker

supports a few different types of networks. Let's look at each type of network to get a better understanding of how networking works in containers:

- Bridge is the default network for containers. When Docker starts, a bridge network is created, and the containers will be connected to this network unless otherwise specified. With this network type, port mapping is needed for the outside world to access the container. This network type is for containers running on the same Docker daemon host. If containers need to communicate with containers running on other daemon hosts, routing needs to be done at the OS level, or the overlay network type should be used.
- Host uses the host's networking directly. Containers will be accessed using an IP address of the host. This networking type only works on Linux hosts. This is not supported on Docker Desktop. This networking type is also used with swarm.
- Overlay also known as ingress connects Docker daemons together for multi-host network communication. The overlay type runs several layers of network abstraction on top of a physical network. An overlay network is a single layer 2 broadcast domain among containers that are hosted on multiple Docker hosts.
- Macvlan lets you assign MAC addresses directly to containers. When Macvlan is used, containers appear as if they are physically on the network. When this is used, containers can be assigned a public IP address that is accessible from the outside. This type of network connects the container to the host network interfaces. This type uses layer 2 segmentation, and there is no need for network address translation (NAT) or port mapping.

Storage

Containers can store changes made to them. Any container changes will be saved to a writeable layer. This writeable layer requires a storage driver to store these changes. Now by default, containers have nonpersistent storage. What nonpersistent means is that when a container is restarted, the storage is destroyed. In order to retain data indefinitely when a container is restarted or turned off, persistent storage is needed.

With Docker containers, we have four options for persistent storage. The persistent storage options are

- Data volumes sit on the host filesystem outside of the container. These allow you to create persistent storage and manage the volumes such as list them, list the container they are associated with, and rename them.
- Data volume container is when a container is dedicated for hosting a volume for other containers. You can mount the volume from this container in other containers. For example, you may have an application container to host the application and a volume container that hosts the volume for the application container.
- Directory mounts are when you mount the host's local directory into a container.
- Storage plug-ins work with underlying storage devices and can connect to external storage solutions. This can map to external storage solutions including cloud providers Azure, AWS, and GCP; storage arrays like EMC, HPE 3PAR, and NetApp; and storage appliances.

Installing Docker

When you start to work with Docker, you will need to install Docker Desktop on your local machine. Docker Desktop is typically used for local development purposes. Docker Desktop includes Docker Engine, Docker Client, Docker Compose, Docker Machine, and Kitematic. Kitematic is something we have not discussed yet. Kitematic is a GUI for working with Docker images and containers. Kitematic also automates the Docker installation and setup process.

Docker is cross-platform, so it can be installed on Linux, Mac, or Windows. In this section, we are going to cover the steps for installing Docker on Windows. Let's dive right into the steps for installing Docker on Windows.

Requirements:

- Cluster and node management
- Windows 10, 64 bit: Pro, Enterprise, or Education (build 15063 or later)
- Virtualization enabled in the BIOS
- CPU SLAT-capable feature
- Microsoft Hyper-V
- At least 4 GB of RAM

Install steps:

1. Download **Docker Desktop Installer.exe** from <https://download.docker.com/win/stable/Docker%20for%20Windows%20Installer.exe>.
2. Double-click **Docker Desktop Installer.exe** to run the installer.
3. A wizard will pop up. Follow the steps in the wizard including accept the license, authorize the installer, and proceed with the install.
4. Click Finish to complete the Docker Desktop install.
5. Docker will not start automatically. Docker will need to be started. To do this, use Windows search to search for Docker. Click Docker Desktop for Windows.

Note If Hyper-V is not enabled, the Docker Desktop installer will automatically enable it and will reboot the computer if needed.

You can set Docker Desktop to automatically start upon login into Windows as shown in Figure 1-2.

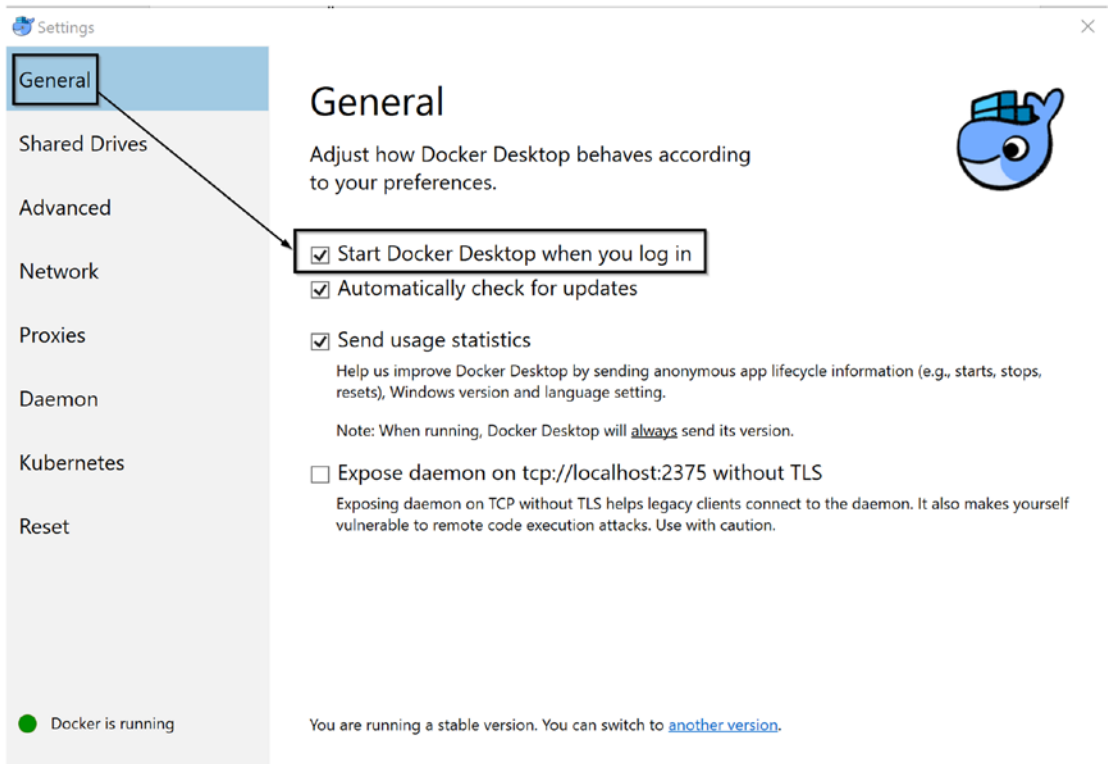


Figure 1-2. Start Docker Desktop on login setting

After Docker is installed, you should see the Docker icon in your task bar tray to reflect that Docker is installed and running. Let’s dive more into utilizing Docker in the next sections.

Docker Command Cheat Sheet

Interacting with Docker is done via command line. Docker was written in Go. Docker stores its configuration files in a directory called `.docker`. Let’s break down the docker command structure. All docker commands start with `docker`, and then there is a space and then the command, another space, and then the management category or option. The docker management command syntax can be seen in Figure 1-3.

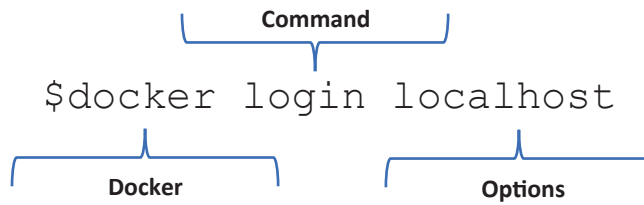


Figure 1-3. *Docker management command structure*

Docker commands that refer directly to a container are slightly different. Commands start with `docker`, and then there is a space and then the command, another space, and then the container name. The docker command syntax referring to a specific container can be seen in Figure 1-4.

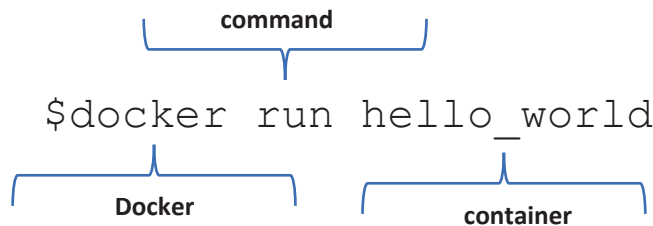


Figure 1-4. *Docker command structure*

Here is a list of critical Docker CLI commands you should know as you get started with docker:

- `docker info`: This will show system-wide information for Docker.
- `docker version`: This will list your current Docker version.
- `docker [COMMAND] help`: This will list the help info about a command.
- `docker images`: This will list the images on your local system.
- `docker run`: This will create and run a container based on an image.
- `docker start`: This will start an existing container.
- `docker stop`: This will stop a running container.
- `docker build`: Used to build an image from a Dockerfile.

- `docker login`: This will log you into a Docker registry.
- `docker logout`: This will log you out of a Docker registry.
- `docker pull`: This will pull an image from a container registry.
- `docker ps`: This will list running containers.
- `docker inspect`: This will show all info on a container including IP addresses.
- `docker rm`: This will delete an image.
- `docker logs`: This will print the docker logs.

Understanding the Dockerfile

A Dockerfile consists of a set of instructions for building an image. A Dockerfile should be named “Dockerfile” with no extension. When an image is built from a Dockerfile, all files that need to be included in the image should be within the same folder as the Dockerfile. Here is an example of a Dockerfile:

```
FROM python:alpine3.7
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
EXPOSE 5000
CMD python ./index.py
```

Let’s break down the commands from our example Dockerfile to gain a better understanding of the Dockerfile structure:

- **FROM**: This defines base image used for the container.
- **COPY**: This will copy files from a source to the container.
- **WORKDIR**: This sets the path where the command, which is defined with CMD, will be executed.
- **RUN**: This defines a set of commands to run within the container when it is first created.

- **EXPOSE:** This will expose a port to the outside world to enable networking access to the container.
- **CMD:** This will execute a specific command within the container when it runs.

The `docker build` command is used as a way to automate the build of an image from the Dockerfile. In the next section, we will take a closer look at `docker build`.

Understanding Docker Build

As stated in the previous section, `docker build` is the command that runs the process to create an image from a Dockerfile. This should be run from within the same directory that contains the Dockerfile. Here is an example of the `docker build` syntax:

```
docker build --tag pythonapp:dev
```

The `--tag` and `:dev` will tag the image with a name and `dev`. This makes it easier to identify the image. Tags are a way to document information about a container image's variant and/or version. Think of tags as adding an alias to container images. After the image build process runs, you can run `docker images` to list the images and verify your name image was created.

Understanding Docker Compose

Dockerfile is a single image. You can create a single image to run a single container using Dockerfile. If you need to create a multi-container application where the containers are connected, then you can use a tool named Docker Compose. We will not dive deep into Docker Compose as this is an advanced topic and out of the scope this chapter. We will however give an overview of Docker Compose.

Docker Compose files are in YAML. Within the Docker Compose file, you reference images; therefore, you still need to build the container images in Dockerfiles. With Docker Compose, you can run a single command to launch all of the containers that make up your application in one shot. Here is an example of a multi-container-based

CHAPTER 1 INSIDE DOCKER CONTAINERS

WordPress application in a docker-compose.yml file made up of a WordPress site and a backend MySQL database server:

```
version: '1.0'

services:
  db:
    image: mysql:latest
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: secret3241
      MYSQL_DATABASE: wp
      MYSQL_USER: wpadmin
      MYSQL_PASSWORD: secret3241

  wordpress:
    depends_on:
      - db
    image: wordpress:latest
    ports:
      - "8000:80"
    restart: always
    environment:
      WORDPRESS_DB_HOST: db:3306
      WORDPRESS_DB_USER: wpadmin
      WORDPRESS_DB_PASSWORD: secret3241
      WORDPRESS_DB_NAME: wp

volumes:
  db_data: {}
```

Docker Compose basically works in the following three steps:

1. Define the needed container images with Dockerfiles.
2. Define the services that make up your multi-container application within a `docker-compose.yml` file.
3. Run the `docker-compose up` command within the directory that has the `docker-compose.yml` file to start and run the multi-container application.

Running a Container

You have learned how to create a container image. You learned about a Dockerfile and then how to create an image using `docker build` and how to list the image. The next step is to create and run the container from the image. You can run the following syntax to create and build the container:

```
docker run pythonapp:dev
```

After the container has been created for the first time, you cannot stop and start the container using `docker stop pythonapp:dev` and `docker start pythonapp:dev`.

Orchestration Platforms

Throughout this chapter, so far you have learned all about containers, Docker, and the many facets of containerization. It is fairly straightforward to build container images and run containers while developing software. Running hundreds or even thousands of containerized applications in production, enterprise ready, and at scale requires a different set of tools and skills not discussed yet. When you need to run containers in production is where container orchestration platforms enter the picture. Container orchestration is all about managing the life cycle of containers. Production container environments are dynamic and require heavy automation. Container orchestration handles

- Cluster and node management
- Container provisioning and deployment
- Container configuration and scheduling

- Container availability and redundancy
- Autoscaling of cluster nodes and containers
- Container load balancing, traffic routing, external access, and service discovery
- Resource management and movement of containers
- Container and host health and performance monitoring
- Container security and access management

Orchestration systems need to cover a lot of ground to handle the life cycle management of containers. There are many container orchestration platforms out there in the market. The top container orchestration platforms are Docker Swarm, Docker Enterprise, Mesosphere, OpenShift, and Kubernetes. Kubernetes is an open source orchestration platform that was developed at Google. Kubernetes has quickly become the de facto standard for container orchestration. The top three cloud providers Microsoft, Amazon, and Google all offer a managed Kubernetes service on their cloud platform. In the rest of this book, we are going to dive deep into Kubernetes and specifically Azure Kubernetes Service.

Note Docker Compose is often referred to as an orchestration tool; however, it is also important to note that Docker Compose is for a dedicated single node compared to orchestration platforms that run many nodes.

Summary

That brings us to a close of this first chapter. In this chapter, we took a journey into the world of Docker containers as this information is foundational to have along the journey into Kubernetes and eventual Azure Kubernetes Service. Within this first chapter, we specifically covered the value of containers, containers compared to virtual machines, all about Docker itself including how to install it, core commands needed for Docker, and all about creating and running container images. Finally, in this chapter, we touched lightly on container orchestration platforms.

CHAPTER 2

Container Registries

Kubernetes is used to deploy applications and services that are based on containers. In many ways, container-based applications are what drove the need for container orchestration technologies like Kubernetes. As mentioned in the previous chapter, containers are instantiated from a read-only copy called an *image*. Images are often stored in a construct called a container registry.

In this chapter, we will discuss the various options for the storage, management, and distribution of images. We will investigate the different types of container registries and further expand on the Azure Container Registry (ACR) in particular. By the end of this chapter, you will be able to perform basic operations on a container registry and understand concepts like image tagging, security, and permissions.

Note For the sake of simplicity, we are going to be focusing on images that use the Open Container Initiative (OCI) image spec and containerd runtime. There are other container image formats (ACI) and container runtimes (rkt), but the essential concepts remain the same.

Overview of Container Registries

When you are deploying an application to a Kubernetes cluster, that application is going to be made up of one or more containers. Kubernetes needs to be able to access the images to instantiate those containers on each node in the Kubernetes cluster. You could create an image on your workstation and manually copy it to each node and then repeat the process each time you update the image. But that would be incredibly inefficient, error prone, and unscalable. A more elegant solution would be a shared location that all nodes can access and download images from that location into their local image cache. That is the basic principle behind a container registry.

Registries, Repositories, and Images

Before we dive into an examination of container registries, it is useful to understand the differences between a registry, a repository, and an image. As we mentioned in Chapter 1, “Inside Docker Containers,” container images are the read-only construct from which a container is instantiated. Each image has a name and optional tags associated with it. Let’s examine the example of an image pulled from Docker Hub shown in Listing 2-1.

Listing 2-1. Image listing of nginx container image

REPOSITORY	TAG	IMAGE ID
Nginx	latest	53f3fd8007f7

The image comes from the nginx repository. It has been tagged as latest. And it has a unique image ID. There are other images stored in the nginx repository, including an image tagged as alpine and another tagged as perl. After pulling both of those images, the updated output of `docker image ls` is shown in Listing 2-2.

Listing 2-2. Image listing of nginx container images

REPOSITORY	TAG	IMAGE ID
nginx	alpine	dd025cdfe837
nginx	perl	4d95835f5c94
nginx	latest	53f3fd8007f7

Each of the images comes from the same repository, but they all have different tags and image IDs. They are unique images. We will dive into tagging and how it is related to images further on in the chapter.

In summary, a container repository contains one or more images. A container registry contains one or more repositories. The images in a particular repository may or may not be related to each other.

Private and Public Registries and Repositories

When it comes to choosing a container registry to host your images, the first question is often whether to create a private or public registry. A **public registry** is hosted on the Internet and is accessible to anyone. It may contain a mix of both public and private

repositories within the registry. A **private registry** is hosted on an internal network and only accessible to systems and users on that internal network. The repositories in a private registry can also hold a mix of public and private repositories, but in this case the scope of a public repository is necessarily more restricted since it is only available to resources on the internal network. While most public registries are run as a managed service, private registries are usually managed by the internal IT team of the organization hosting the registry.

The images in a **public repository** are accessible to anyone who can access the registry's network location. It does not mean that anyone can add, update, or remove images from the repository, but they can download images without any type of authentication. The images in a **private repository** require that anyone wishing to access the repository is authenticated and granted the relevant permissions to download images.

Public registries and repositories are most often used to distribute open source projects and software that are meant to be shared with the world. For instance, the Microsoft repository on Docker Hub is public and is used to publish base images for applications like *Microsoft/dotnet* and *Microsoft/powershell*. Obviously, Microsoft is hoping that you will download these images and use them to build something amazing.

Some common public registries are

- Docker Hub
- Google Container Registry
- Azure Container Registry
- Amazon Elastic Container Registry

Private registries and private repositories are used when images are meant to be kept within a company or organization, and access to those images should be controlled. In addition, private registries are often hosted on an internal network that is not accessible by the wider Internet. You may have images that have proprietary software installed that is considered important intellectual property. For instance, let's say your company is developing software for genomics and deploying it using containers. The images would contain extremely valuable algorithms that should not be available to competitors. Therefore, you would choose to host your images on a private repository and possibly on a private registry as well.

Some common private registries are

- Docker Trusted Registry
- JFrog Artifactory
- Harbor
- GitLab

Basic Registry Operations

All container registries support the same basic operations. These operations include

- Logging into the registry to interact with image repositories
- Searching image repositories for a specific image
- Pulling an image down to the local filesystem
- Pushing an image up to an image repository hosted on the registry

For the following examples, we are going to show operations being performed against Docker Hub. You can create a Docker Hub account for free and follow along with the examples.

Login

Logging into a container registry can be accomplished by using the Docker CLI. The following command will start the login process.

```
docker login [SERVER]
```

The SERVER value can refer to whichever registry you intend to log into. The command also accepts supplying a username and password. If no SERVER value is specified, the Docker CLI will assume that you are logging into Docker Hub. Listing 2-3 shows an example of logging into Docker Hub using the account *iaks*.

Listing 2-3. Logging into Docker Hub**\$ docker login**

Login with your Docker ID to push and pull images from Docker Hub. If you don't have a Docker ID, head over to <https://hub.docker.com> to create one.

Username: iaks

Password: *****

Login Succeeded

Search

The Docker Hub registry has over 100,000 container images available to the public. Private registries will obviously have far fewer images, but there is a need to be able to search through available images to find the one that meets your needs. The `docker search` command provides this functionality. The syntax of the command is

```
docker search [OPTIONS] TERM
```

Let's say we are looking for an `nginx` image to run and host a web server. The search command to do so is shown in Listing 2-4.

Listing 2-4. Searching Docker Hub for `nginx` images**\$ docker search nginx**

NAME	DESCRIPTION	STARS	OFFICIAL
nginx	Official build of Nginx.	11498	[OK]

.

.

[output truncated]

By default, the Docker CLI will search the Docker Hub registry for images. Other registries can be searched by including their address in the search `TERM`. You will need to be authenticated with the registry you are attempting to search prior to executing the search command.

Pull

Pulling an image from a container registry is the act of downloading the hosted image to a local file repository. The location where the files are stored is determined by which storage drive is being used by the Docker daemon. When an image is pulled, Docker first checks the layers included in the image to determine if any of the layers have already been downloaded. Any layers that are not already cached locally will be downloaded from the source repository. Listing 2-5 shows an example of pulling the *alpine/terraform* image with one layer already existing on the local filesystem.

Listing 2-5. Pulling the *alpine/terraform* image from Docker Hub

```
$ docker pull alpine/terraform
```

```
Using default tag: latest
```

```
latest: Pulling from alpine/terraform
```

```
e7c96db7181b: Already exists
```

```
622c94c90cb1: Pull complete
```

```
[output truncated]
```

```
68ced3bc2ce4: Pull complete
```

```
Digest: sha256:4363c7ea68ae6b648d803753884afed380f106eb23e902641ae919b7b02f  
e95a
```

```
Status: Downloaded newer image for alpine/terraform:latest
```

```
docker.io/alpine/terraform:latest
```

In the case of a public registry, anyone can pull an image whether they are authenticated or not. With private registries, the user must first be authenticated and have permissions to pull a given image.

An image can be pulled by issuing the `docker pull` command. The syntax is as follows:

```
docker pull [OPTIONS] NAME[:TAG | @DIGEST]
```

The name refers to the name of the image. Docker will assume that the source repository is on Docker Hub, unless it has been configured otherwise, or the name includes a different registry. For instance, an image can be pulled from Microsoft's public container registry by running

```
docker pull mcr.microsoft.com/azuredocs/aci-helloworld
```

If no TAG is specified, then docker will grab the image in the repository tagged *latest*. There is nothing special about the *latest* tag, and it does not mean that the image pulled will in fact be the latest or most up-to-date image. Generally, it is always best to specify a tag along with the name of the image to be pulled.

Push

Pushing is the act of taking a local image and copying it to a target repository. The repository can be on a public or private registry. In either case, both types of registries will require authentication and proper authorization before allowing the image to be copied.

A new image can be created from a Dockerfile using the `docker build` command and then pushed to the target registry. It is also possible to use an existing image that was pulled from a separate repository and push it to a different repository. Listing 2-6 shows an example Dockerfile that could be used to build a new image.

Listing 2-6. Dockerfile content

```
FROM nginx:stable-alpine
COPY IAKS /usr/share/nginx/html
```

The FROM command will pull the nginx image tagged as stable-alpine from the nginx repository. The COPY command will copy the contents of the IAKS directory to the path /usr/share/nginx/html. We can create this new image by running the following command from the directory containing the Dockerfile:

```
docker build --tag iaks/nginx:v1.
```

By naming it `iaks/nginx:v1`, we are indicating that the target repository for this image will be the iaks Docker Hub account and the name of the image is nginx. We have tagged it as v1, which for the moment is an arbitrary tag. By running `docker image ls`, we can see, in Listing 2-7, that we now have a new image on the local filesystem.

Listing 2-7. Listing the image created by docker build

```
$ docker image ls
```

REPOSITORY	TAG	IMAGE ID
iaks/nginx	v1	bbbdb4e15efd

Finally, we can push the image from our local filesystem to our Docker Hub repository by running the `docker push` command. The syntax for the command is `docker push [OPTIONS] NAME[:TAG]`

In this case, we would run the command shown in Listing 2-8 to push the image.

Listing 2-8. Pushing the image to Docker Hub

\$ docker push iaks/nginx:v1

```
The push refers to repository [docker.io/iaks/nginx]
7dd2de43c03e: Pushed
2bdf88b2699d: Mounted from library/nginx
f1b5933fe4b5: Mounted from library/nginx
v1: digest: sha256:00caf...f4997cea1 size: 94
```

Viewing our Docker Hub account through a browser, we can see in Figure 2-1 that the image has been successfully pushed to our repository.

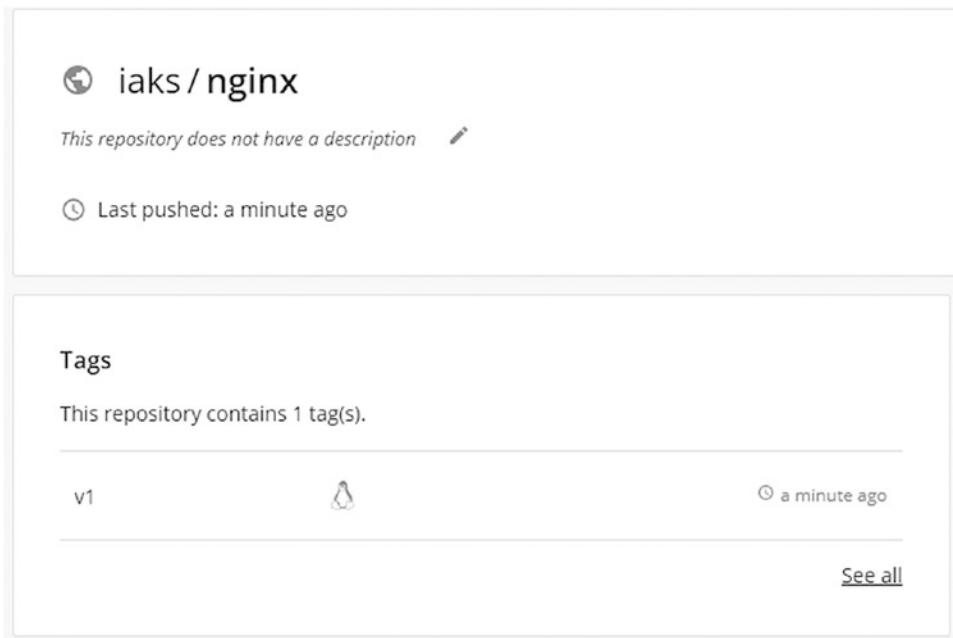


Figure 2-1. Successful push of an image to Docker Hub

It would now be possible to pull this image and run it on any container host with Internet access. That includes worker nodes in an Azure Kubernetes Service cluster.

Image Tagging

Image tags are additional metadata associated with a specific image. As we saw in the section on registries, repositories, and images, an image has a repository, tag, and ID. In Listing 2-9, three nginx images have been pulled, all with different tags and image IDs.

Listing 2-9. Listing of different nginx images

REPOSITORY	TAG	IMAGE ID
nginx	alpine	dd025cdfe837
nginx	perl	4d95835f5c94
nginx	latest	53f3fd8007f7

Multiple tags can be associated with a single image through the use of the `docker image tag` command. Adding another tag to an image does not create a new image and does not take up more space on your local filesystem. Docker simply assigns this new metadata information to the existing image ID. The `docker image tag` command has the following syntax:

```
docker image tag SOURCE_IMAGE[:TAG] TARGET_IMAGE[:TAG]
```

For instance, we can take the existing `nginx:alpine` image and tag it with `v1`.

```
docker image tag nginx:alpine nginx:v1
```

Upon viewing the local image listing in Listing 2-10, both images are present, and both have the same image ID.

Listing 2-10. Logging into Docker Hub

\$ docker image ls

REPOSITORY	TAG	IMAGE ID
nginx	alpine	dd025cdfe837
nginx	v1	dd025cdfe837
nginx	perl	4d95835f5c94
nginx	latest	53f3fd8007f7

We can also tag the image for a totally different repository and then push the image to that repository.

```
docker image tag nginx:alpine iaks/custom
```


The three entries shown in Listing 2-11 will all have the same image ID.

Listing 2-11. Logging into Docker Hub

\$ docker image ls

REPOSITORY	TAG	IMAGE ID
nginx	alpine	dd025cdfe837
nginx	v1	dd025cdfe837
iaks/custom	latest	dd025cdfe837

Tags are simply metadata associated with an image. That includes the mysterious *latest* tag. If no tag is provided for an image – as we did in the preceding command – Docker will automatically give it the *latest* tag. When an image is being pulled or used to instantiate a container, Docker will likewise assume the *latest* tag if no other tag is provided. The *latest* tag does not mean that the image being pulled is the most up-to-date or even the proper image to pull. It is simply the image that was tagged with the latest label, whether that was done on purpose or through omission. For that reason, it is always recommended to specify the tag of an image when pulling an image or running a container.

Common Registries

Docker Hub and Docker Registry

The most common registry that users get started with is Docker Hub, shown in Figure 2-2. Hosted at docker.io, Docker Hub provides a free home for new users to get started with their first repository. Unless otherwise configured, the Docker CLI assumes that Docker Hub is the registry being used. Many software companies choose to host their publicly available containers on Docker Hub, as do several open source projects. Docker Hub supports both public and private repositories, although the private repositories are not free.

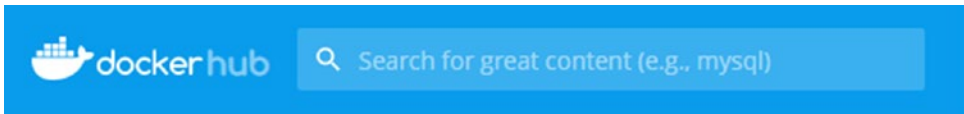


Figure 2-2. Docker Hub web site

Docker Hub is based on the open source project Docker Registry. Docker Registry can also be used to deploy a private registry in your datacenter. Microsoft uses the Docker Registry project as a basis for their deployment of the Azure Container Registry, as do several other public and private registry implementations.

Azure Container Registry

The Azure Container Registry (ACR) is a Software-as-a-Service offering from Microsoft hosted on Azure. ACR is based on the Docker Registry open source project, but it has additional functionality which we will explore later in the chapter. Repositories created on ACR are private in nature and always require some type of authentication for access. ACR has a few different SKUs available – Basic, Standard, and Premium – with the higher tiers offering more robust storage, performance, and replication options.

Images that are stored on ACR can be in the following formats:

- Docker Image Manifest V2, Schema 1
- Docker Image Manifest V2, Schema 2
- Open Container Image Format Specification

The ACR service can also host Helm charts, which we will explore more in Chapter 8, *“Helm Charts for Azure Kubernetes Service.”*

In order to create a registry on ACR, you need to have an Azure subscription. A new ACR registry can be created through the Azure Portal, Azure PowerShell, or the Azure CLI. The examples in the remainder of this section and the next section will use the Azure CLI.

The commands in Listing 2-12 will create a new ACR registry using the Azure CLI.

Listing 2-12. Creating a new ACR registry

```
### Create an Azure Container Registry ###
# Login to Azure and select subscription
az login
az account set --subscription "AZURE_SUBSCRIPTION_NAME"

# Create a resource group and Container Registry
az group create --name RESOURCE_GROUP_NAME --location "LOCATION_NAME"
az acr create --resource-group RESOURCE_GROUP_NAME --name ACR_NAME --sku Basic
```

The ACR_NAME must be globally unique. It will be appended to azurecr.io to create the publicly addressed fqdn for your registry.

In the next section, we will explore some components of the Azure Container Registry in more depth, including how it integrates with the Azure Kubernetes Service.

Azure Container Registry Expanded

While the Azure Container Registry (ACR) service is based on the Docker Registry open source project, it has a number of additional enhancements that are worth making note of. ACR makes use of Azure Active Directory (Azure AD)-powered role-based access control (RBAC) to control access to repositories hosted on the service. ACR has been extended beyond a basic registry to support the capability of running simple or multi-step tasks as part of the service. Since ACR is running in Azure, it has some custom integrations with other Azure services including the Azure Kubernetes Service.

Security

There are three different ways to authenticate with the ACR:

Azure AD individual login: An individual logging into ACR uses their Azure Active Directory account to authenticate against ACR. They are issued a token that is good for one hour before they are required to authenticate again.

Azure AD service principal: Similar to an individual login, however, the service principal can be used for headless authentication and is most commonly used with automation platforms.

Admin account: The admin account is an account not linked to Azure Active Directory. The admin account has full permissions to perform any action on the ACR registry. By default, the admin account is disabled, and it should only be used for testing and emergency scenarios.

In order to log into the ACR we created in the previous section, we can use the following command:

```
az acr login --name ACR_NAME
```

Since we are already logged into Azure, ACR takes our existing credentials and generates a token for use with ACR that is good for one hour. After that time period, the token expires, and the `az acr login` command must be run again. Service principal logins use a username and password when logging in and therefore do not have a token issued based on cached credentials. Service principal logins are the preferred login type when using ACR with an automated process.

Whether using the individual login or service principal option, permissions are assigned through well-defined roles.

Permissions

As with many of the services within Microsoft Azure, permissions within Azure Container Registry are assigned using role-based access control (RBAC). At the time of writing, there are seven roles defined by the service. Table 2-1 outlines the roles and their permissions.

Table 2-1. *RBAC for Azure Container Registry*

Permission/Role	Owner	Contributor	Reader	AcrPush	AcrPull	AcrDelete	AcrImageSigner
Access Resource Manager	✓	✓	✓				
Create/delete registry	✓	✓					
Push image	✓	✓		✓			
Pull image	✓	✓	✓	✓	✓		
Delete image data	✓	✓				✓	
Change policies	✓	✓					
Sign images							✓

The assignment of these roles should follow the principle of least privilege, where the person or service is assigned the least number of permissions required to perform a particular task. The `AcrPush`, `AcrPull`, `AcrDelete`, and `AcrImageSigner` roles are especially designed for services and automation processes that perform specific tasks over their lifetime. For instance, let's say we are using an Azure Kubernetes Service to

deploy containers that are stored in ACR. Assigning the service principal used by the AKS cluster the `AcrPull` role will grant it sufficient privileges to access the container images needed, without also granting access to Resource Manager, a permission which the Reader role includes. Likewise, any CI/CD pipelines that build new container images could be granted the `AcrPush` permission to push the new images up to ACR.

Tasks and Automation

In addition to the storage of container images, the Azure Container Registry service also includes the ability to run simple and multi-step tasks and emit webhooks when certain actions are completed. The tasks and webhooks provide the ability to leverage ACR for common tasks related to image management and assist with integration into a CI/CD pipeline.

Tasks within ACR can be broken into simple tasks, which can be initiated using the `az acr build` or `az acr task` command, and multi-step tasks that are defined by a YAML file and submitted via the `az acr run` command.

Simple Tasks

Simple tasks are used to build a newer version of a container image. The building of the image can be triggered manually by using the `az acr build` command. Doing so off-loads the burden of a container build from your local workstation to ACR, as well as placing the resulting image in ACR without having to push it from your local filesystem. Creating a new image from a local Dockerfile would be performed using the following command:

```
az acr build --registry iaks --image web:v2.
```

The command will build an image tagged `v2` in the image repository `web` on the container registry `iaks`. In addition to off-loading the build process from your local workstation, the same command could be used by a service principal in a CI/CD pipeline to automate new container image builds without using the resources on one of the pipeline agent machines.

Instead of running a task manually, it can be triggered by a git commit or the update of a base image. While the idea of updating based on a git commit makes intuitive sense, the concept of updating an image when a base image is updated bears some explaining.

ACR understands the image dependencies of images stored in its repositories. For instance, your shiny new *web* container image might be based on the *alpine:stable* image from Docker Hub. When that base image is updated, you may want your *web* image to be updated as well to include whatever was changed in the base image. ACR supports the creation of a build task that will be triggered if it detects that the base image for an image in the repository has been updated. The command in that case would look something what is in Listing 2-13.

Listing 2-13. Creating an ACR task to update an image based on base image updates

```
az acr task create \
  --registry iaks \
  --name task-web-service \
  --image web:v2 \
  --arg REGISTRY_NAME=iaks.azurecr.io \
  --context https://dev.azure.com/iaks/_git/iaks.git \
  --file Dockerfile-web \
  --branch master \
  --git-access-token $TOKEN
```

Within the definition of the Dockerfile-web file is a referral to the base image of *alpine:stable*. ACR in turn creates a hook to listen for changes to that base image and will start a build task if a change is detected.

Multi-step Tasks

Mutli-step tasks in ACR build on the existing simple tasks while adding more capabilities. The actions in a multi-step task include

- **Build:** Builds one or more container images
- **Push:** Pushes images to a private or public container registry
- **Cmd:** Runs a container using similar arguments as `docker run`

The actions performed as part of a multi-step task are defined in a YAML-formatted file that is submitted to ACR using the command in Listing 2-14.

Listing 2-14. Creating a multi-step task in ACR

```
az acr run \  
  --registry iaks \  
  -f multi-step-task.yml \  
  https://dev.azure.com/iaks/iaks.git
```

The command instructs ACR to run a task on the registry *iaks* using the file *multi-step-task.yml* found on the referenced git repository.

Multi-step tasks can be used in a workflow to build and test a set of images that make up a container-based application and then update a Helm chart if the tests defined in the task pass. While this is not a replacement for a fully feature CI/CD pipeline, it does provide a way to define workflows in code and have them execute when a new commit is made to a git repository.

Webhooks

When an action is completed in Azure Container Registry, it can notify other services via a webhook. This can assist with sending simple notifications or firing off an automation workflow. The supported actions for triggering a webhook are

- image push
- image delete
- image quarantine
- Helm chart push
- Helm chart delete

Triggering a webhook sends a POST request to the Service URI defined in the webhook configuration. The POST request includes JavaScript Object Notation (JSON)-formatted information that is dependent on the action that triggered the webhook. Custom headers can also be defined in the webhook configuration to be sent with each POST request. These can be leveraged if the target Service URI requires authentication or some other custom data not included in the POST payload.

Creating a webhook that contacts the Postman echo service for an image push would look like Listing 2-15.

Listing 2-15. Logging into Docker Hub

```
az acr webhook create \  
    --registry iaks \  
    --name postmanhook \  
    --actions push \  
    --uri https://postman-echo.com/post
```

The Postman echo service will simply reply back with the contents of the initial POST request, which makes it useful for understanding the information being sent by the webhook.

Azure Kubernetes Service Integration

The Azure Kubernetes Service (AKS) uses both container images and Helm charts to deploy applications to the nodes in an AKS cluster. Conveniently, Azure Container Registry is capable of storing both of those resources and making them available to AKS for consumption. In order to access the resources stored on an ACR registry, AKS can use Azure Active Directory authentication.

When a new AKS cluster is created, it is assigned a service principal in Azure Active Directory. The service principal can be assigned roles to registries hosted in the Azure Container Registry, including the AcrPull role. With this role, the AKS cluster will be able to pull container images and Helm charts that are hosted in that container registry.

Using ACR in tandem with AKS allows you to keep your images hosted in a private registry and use the native authentication service Azure AD to provision a proper level of access for the AKS cluster. You can also make use of ACR Tasks to automate the build, test, and release of container-based applications to your AKS cluster.

Summary

Container registries are a critical component of deploying applications on Kubernetes. Without an image repository, the images would have to be manually copied to every node in the cluster. That's not exactly an ideal situation. Before building and operating applications in Azure Kubernetes Service, it is important to have a proper grounding in container registries and how they are operated.

In this chapter, you learned about the different types of container registries – private and public. We discussed the commands and tools used to interact with an image repository hosted on a container registry, including actions like push, pull, and tagging. Then, we examined the features of the Azure Container Registry service and how it integrates with the Azure Kubernetes Service.

CHAPTER 3

Inside Kubernetes

The first part to a container journey is selecting a container runtime, learning the ins and outs of it, and containerizing applications. The next level is being prepared to run the containers in production at an enterprise level. For running containers in production, you will want an orchestration platform.

The containerization of applications often includes the decoupling of them from monolithic into microservices-based architecture with the components split across many containers. This results in hundreds or even thousands of containers that need to be managed with many of them sharing the same life cycle needing to be managed together.

An orchestration platform is a solution for managing the life cycles of containers. Orchestration platforms control and automate containers in dynamic enterprise environments consisting of the following functionality:

- Provisioning nodes
- Instantiating and scheduling containers
- Container availability and redundancy
- Distribution of containers evenly across nodes
- Allocation of resources across containers
- Scaling of cluster nodes as needed
- Host and container health and performance monitoring
- Scaling up or removing containers as needed
- Moving containers from host to host as needed if there is a shortage of resources on a host, or if a host goes down
- Load balancing of service discovery across containers

- External access to services running in containers
- Application configuration management in relation to the containers that run the application

There are many orchestration platforms on the market. Enter Kubernetes, the most popular container orchestration platform. Kubernetes was created by Google and was designed to work in the environment of your choice such as on-premises on bare metal servers, virtual servers, and public clouds.

Kubernetes has become the gold standard of container orchestration platforms; in fact, the top three cloud providers AWS, GCP, and Azure all offer a managed Kubernetes service as well. Kubernetes is a tightly integrated platform that includes hosting of the Kubernetes components, Docker runtime or Moby runtime, as well as provisioning the host nodes and orchestration of the containers.

Key features of Kubernetes include

- Deployment and replication of containers
- Scale in and out of containers
- Load balancing of containers
- Rolling upgrades of nodes in the cluster
- Resiliency and automated rescheduling of failed containers
- External exposure of container network ports to the outside world

The Kubernetes architecture can be seen as complex. The architecture does have many moving parts, and it is important to understand them. Figure 3-1 is a visual representation of the Kubernetes architecture.

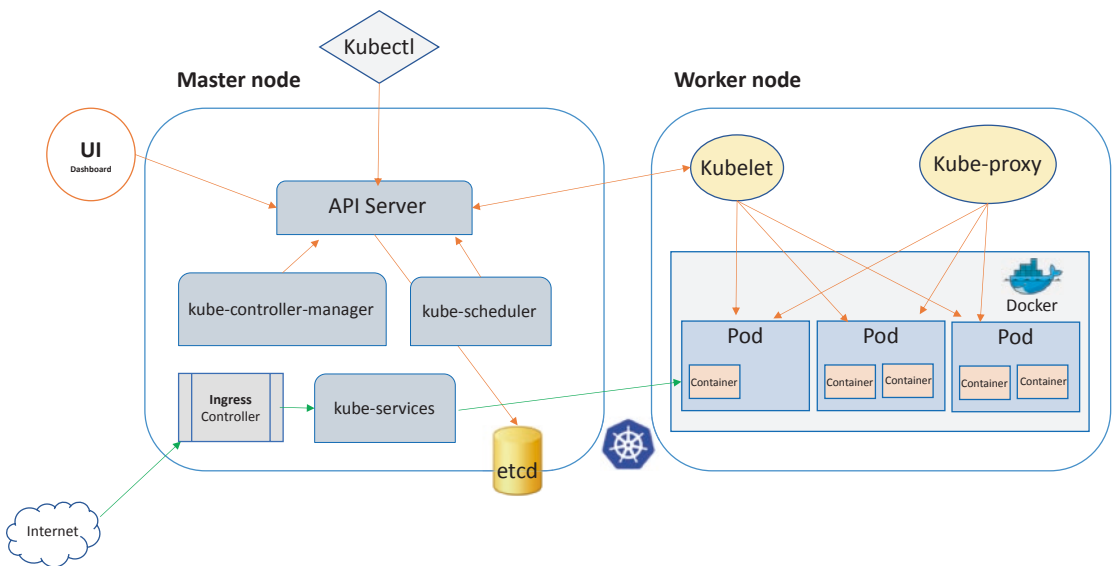


Figure 3-1. *Kubernetes architecture*

As a precursor to working with Azure Kubernetes Service (AKS), it is ideal to have a solid understanding of Kubernetes. In the following sections within this chapter, we will dive deeper into the various components of Kubernetes.

Kubernetes Interfaces

There are multiple ways to interface with Kubernetes. Kubernetes has a REST API, and you can interact with the API directly using REST calls. There are some third-party tools out there that utilize this method such as Rancher bringing Kubernetes management into the Rancher interface.

The second most common way to interface with Kubernetes is through the `kubectl` command line interface. You can use `KubectI` to pretty much do anything in Kubernetes. Some of the tasks you can perform with `KubectI` are deployment of pods, inspect and management of cluster resources, work with nodes, view logs, and upgrade the cluster. In Chapter 4, “`kubectI` Overview,” you will take a deeper dive into `KubectI`.

In addition to the `KubectI` command line interface, there is a web-based user interface for Kubernetes known as the Kubernetes dashboard. This dashboard can be used for basic management operations of Kubernetes. You can manage resources such as pods, deployments, jobs, nodes, volumes, replica sets, and more. It also can be used to

get state and health information on your Kubernetes resources. Figure 3-2 is a screenshot of the Kubernetes dashboard.

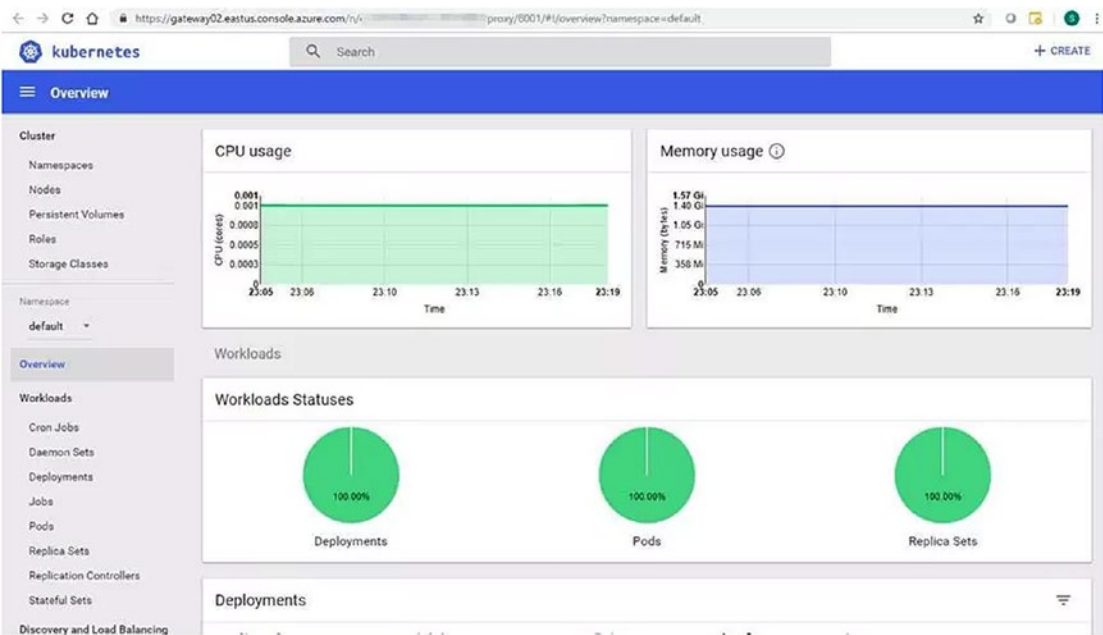


Figure 3-2. Kubernetes dashboard

Docker Runtime

A runtime is needed to run containers. Kubernetes supports both Docker and RKT runtimes. As covered in Chapter 1, “Inside Docker Containers,” of this book, the most widely adopted runtime in the container space is Docker. In Kubernetes, the most common runtime you will find running is Docker. With Docker, you can run Linux- or Windows-based containers. When running Docker in Kubernetes, you can also run Linux or Windows containers.

Master Nodes Overview

In Kubernetes there is a master node that controls and coordinates the cluster. The master node is essentially responsible for managing the cluster. This master node coordinates any activity that happens in the cluster such as provisioning nodes, node-to-node communication, serving as interface for working with Kubernetes,

scheduling containers on the nodes, maintaining the desired state of the containers and applications working through rolling updates, and more. Here are the components that make up the master node:

- **Etcd** is a simple, distributed, consistent key-value store. Etcd stores data about the Kubernetes cluster. It stores data such as nodes, pods, scheduled jobs, services, API objects, namespaces, and another configuration about the cluster. In order to stay secure, it can only be accessed via the API server component.
- **Apiserver** is the central entry point for all REST requests to the Kubernetes cluster. The REST requests can be used to control the cluster and perform actions on components such as pods, deployments, replica sets, services, and more. The Apiserver is also used to communicate with etcd.
- **Kube-controller-manager** watches the shared state of the Kubernetes cluster and makes changes as needed to ensure the cluster meets desired state. An example is ensuring that the correct number of pods are currently running, and the service that points to the pods is running and tracking the pods if they move. The controller manager gets the shared state from the Apiserver. The controller manager also performs controller processes routine tasks in the background.
- **Cloud-controller-manager** is exactly like the Kube-controller-manager except the cloud-controller-manager handles controller processes that depend on an underlying cloud provider. For example, if Kubernetes is running on Azure and is utilizing Azure load balancers, the cloud-controller-manager can ensure the needed load balancers are running.
- **Kube-scheduler** is the component that handles all the scheduling (placement) of pods (containers) on various nodes in the Kubernetes cluster. The scheduler has information about resources available in each node in the cluster so it can place pods properly on nodes that have available capacity.

Worker Nodes Overview

Worker nodes are where the pods and applications run. Worker nodes are virtual machines. In Kubernetes you can have either Linux or Windows worker nodes. The Linux nodes would run containers and applications such as Java, Apache Tomcat, and other Linux-based workloads. A Windows node would run containers and applications such as IIS, .Net, ASP.net, and more. Worker nodes contain all the necessary services such as runtime, networking, scheduling, maintaining container state, and communications to the master node. Here are the components that make up the worker node:

- **Docker** is the runtime engine that runs the containers. The Docker runtime is on each node in a Kubernetes cluster.
- **Kubelet** is a service that handles communication with the master node and etcd. It gets information about new and existing services. Kubelet ensures that the desired containers are healthy and running.
- **kube-proxy** acts as a network proxy and load balancer to expose services to the external world on the worker node. It handles the network routing for TCP and UDP connections.
- **kubectl** is the Kubernetes command line interface that interacts with the Apiserver pushing to the master node.

Namespaces

Namespaces are used as a way to logically segment and organize resources in a Kubernetes cluster between multiple teams. Resources are deployed into a namespace in a Kubernetes cluster. These resources are grouped together for the ability to filter and control them as a single unit.

Namespaces are used to avoid collisions. For example, when teams scale to having thousands of pods, it is possible that deployments could have the same name. In this scenario, you could have multiple namespaces with the deployments with overlapping names existing in different namespaces to avoid collision and for ease of management, organization, and security such as access policies (RBAC). Sometimes namespaces are used for life cycle environments such as development, staging, and production. With namespaces for each of these environments, the same resources could exist in each at the same time because they will be logically separated and will not conflict with each other.

In every Kubernetes cluster, there is a “default” namespace. When deploying resources, if you do not specify a namespace, it will deploy into the default namespace. With Kubernetes, two other namespaces are also deployed by default. These namespaces are kube-system (used for storing Kubernetes components) and kube-public (used for storing public resources globally readable to all users with or without authentication). It is easy to create a custom namespace in a YAML file or by using the Kubectl command. Here is the syntax to do this:

```
kubectl create namespace namespace1
```

Here is an example YAML file:

```
kind: Namespace
apiVersion: v1
metadata:
  name: namespace1  labels:
    name: namespace1
```

Syntax to apply the YAML file to create the namespace:

```
kubectl apply -f namespace1.yaml
```

You can a Kubectl command to list current namespaces:

```
kubectl get namespace
```

To deplete a namespace run:

```
kubectl delete namespace namespace1
```

Labels and Annotations

In Kubernetes when you need to organize, identify, and simply store data about objects, labels and annotations are the go-to features to help with this. If you have spent any time working with a public cloud such as Microsoft Azure, you will be familiar with this need as you may have used tags to help organize your cloud infrastructure. Like public clouds, Kubernetes has a similar set of features in labels and annotations. Labels and annotations take the whole tagging concept to another level. Let’s explore labels and annotations.

Labels are key-value pairs. The keys must be unique. The keys also must have 63 or fewer characters, and the values must be 253 or fewer characters. Labels are designed to be used to organize, query, and identify a set of objects. Labels can be attached to objects when created or at any time:

```
"metadata": {  
  "labels": {  
    "appname" : "webappX",  
    "environment" : "dev"  
  }  
}
```

Annotations are also key-value pairs. Annotations can have more characters compared to labels. Data in annotations is arbitrary, can be structured or unstructured, and is able to include characters not supported in labels. It's important to note that annotations can't be queried. Annotations can be a good way to place metadata to objects in Kubernetes. External systems and tools can consume annotation data. Here are some examples of annotation data: environment such as dev, stage, prod, git branch, pull request number, image information like timestamp or date, version info, app owner, department, and so on. Here is an example of what an annotation looks like:

```
"metadata": {  
  "annotations": {  
    "gitbranch": "brancha",  
    "department": "marketing"  
  }  
}
```

To sum this up, use labels when you will need to query objects in a Kubernetes cluster and use annotations when you need to store general information about objects in Kubernetes but don't need to query it but may also need this information in external systems. Also, labels should be used for identifying objects, and annotations should be used when non-identifying data is needed on objects.

Pods

A pod is one or more containers within Kubernetes that share resources and are coupled together. It represents a unit of deployment. A pod encapsulates an application including the container/s, storage, network IP, and configuration of how to run the containers. Think of pods as a wrapper around containers. Pods are typically deployed into one of two patterns: the first pattern being a pod that runs a single container and the second pattern being a pod that runs multiple containers that need to be tightly coupled together.

The single-container pod is the more common use case in Kubernetes. A multi-container pod is an advanced scenario used when multiple containers make up a single application, share the same life cycle, and need to share resources such as storage and networking. When multiple containers belong to the same pod, they are a single managed entity. When the pod is scheduled, the containers will be placed on the same node, and if the pod needs to be moved to another node, all encompassed containers are moved. Here is an example of what a pod looks like in code:

```
apiVersion: v1
kind: Pod
metadata:
  name: app1-pod
  labels:
    app: app1
spec:
  containers:
    - name: app1-container
      image: nginx
```

Replicaset

A replicaset defines a set of replica pods. A replica set can be used to specify how many identical pods are needed. For example, if you want four copies of a pod to run, you can specify this as in a replicaset. Kubernetes will ensure the four pod replicas are running at all times. If a pod fails, a new one will automatically be deployed to ensure the replicaset maintains running the desired four.

Note Deployments which are covered later in this chapter are able to manage replicaset. Replicasets are typically defined in deployments. It is recommended to utilize deployment sets vs. defining replicasets directly.

DaemonSets

DaemonSets manage groups of replicated pods. They can be used to ensure that all nodes in a Kubernetes cluster run a copy of a specific pod. DaemonSets are typically used when you have some administrative function that is needed on all or specific nodes. An example of when to use a DaemonSet is if you need to perform log collection on all nodes using fluentd. Another example is when you need a monitoring agent such as new relic, AppDynamics, Log Analytics, or Datadog on all nodes.

Jobs

Jobs in Kubernetes supervises pods that run batch processes that run for a finite time to completion. Typical use cases or jobs would be backup, sending emails, transcoding, or calculation operations. Jobs do support parallel and nonparallel.

Services

A service in a Kubernetes cluster is the abstraction that defines a logical set of pods. Service is also a mechanism used to expose external access to pods or an application running on pods. A service is the abstraction on the top of the pod which provides a single IP address and DNS name by which pods can be accessed. It is easy to think of a service as a pointer to a pod or set of pods. When pods are moved from node to node in a Kubernetes cluster service, automatically keep track of where the pods live. There are three types of services as follows:

- **ClusterIP** is the default type used when deploying a service. ClusterIP exposes an IP internal to the cluster only accessible within the cluster.
- **NodePort** exposes a service on a static port on the node.
- **LoadBalancer** is used with cloud providers. LoadBalancer exposes the service externally using the cloud providers load balancer.

Here is an example of what a service looks like in code:

```
apiVersion: v1
kind: Service
metadata:
  name: app1-service
spec:
  selector:
    app: App1
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9523
```

Deployments

Deployments describe the desired state of a replica set and pod. Deployments are manifest yaml files. A deployment controller reconciles the Kubernetes cluster to match the desired state by creating, updating, or deleting replica sets or pods accordingly.

ConfigMaps

With containerized applications, environment configurations should be abstracted from the applications and handled outside of the container and application. This essentially is how configuration management is handled with containers. Keeping configuration separate from containers and applications is one of the techniques to make containers portable. ConfigMaps are a functionality in Kubernetes that helps with configuration management. ConfigMaps hold key-value pairs of configuration data used in pods. ConfigMaps tie configuration artifacts to pods, containers, and system components at runtime. Configuration artifacts consist of command line arguments, configuration files, environment variables, and port numbers.

Note ConfigMaps should be used for nonsensitive configurations that don't need to be secured. For sensitive configurations or data, Secrets within Kubernetes should be used.

ConfigMap key-value data can be literal or from files. ConfigMaps are created using Kubectl. Here is syntax for creating a ConfigMap:

EXAMPLE: `kubectl create configmap [NAME] [DATA]`

EXAMPLE: `kubectl create configmap app1-data --from-file app1-configs/`

`kubectl create configmap` is used to create a ConfigMap holding the key-value pairs. And `--from-file` points to a directory. The files in the directory are used to populate a key in the ConfigMap. The name of the key is the filename. The value of the key is from the content of the file:

EXAMPLE: `kubectl create configmap app1-config --from-literal=app1-config.app1name=myapp1`

After the ConfigMap is created, it can be consumed by a pod via a yaml file. Here is an example yamle file:

```
apiVersion: v1
kind: Pod
metadata:
  name: app1-pod
  labels:
    app: app1
spec:
  containers:
    - name: app1-container
      image: nginx
      env:
        - name: app1name
          valueFrom:
            configMapKeyRef:
              name: app1-config
              key: app1name
```

Secrets

In Kubernetes when you need to secure information, you can use Secret objects. Secrets are a way to store and manage sensitive information in a Kubernetes cluster such as passwords, tokens, SSH keys, and so on. You are able to then reference the secret in pods or container images vs. putting the secret such as a password indirectly. Secrets can be created from a file or literally. Here is an example of creating a secret using literal:

```
kubect1 create secret generic app1-pass --from-  
literal=password=PASSWORDHERE
```

You would then reference the secret in your pod yaml file. Here is an example of this:

```
env:  
- name: APP1_PASSWORD  
valueFrom:  
secretKeyRef:  
name: app1-pass  
key: password
```

Networking

Networking with containers is complicated. At the core, Kubernetes sets out to make the networking with containers easier and more flexible. Kubernetes treats networking with pods are similar to the way it works with virtual machines when it comes to naming, load balancing, port allocation, and even application configuration. Kubernetes by default utilizes an overlay network. Kubernetes gives each pod its own routable unique IP address and single DNS name. This IP is shared by all the containers within the pod.

The address space inside the Kubernetes cluster is flat allowing pods to communicate with each other directly without a proxy. Pods can also communicate with each other across nodes. Kubernetes uses iptables for the network connections between pods. The routable IPs and IP tables make it so you don't have to map host ports to container ports like in Docker.

You may be asking yourself as you read this, "Pod-to-pod communication is good, but how can one get Internet traffic from the Internet to pods?" Services in Kubernetes group pods together logically to provide network connectivity to the applications

running on the pods. There are multiple service types, and these can be used to route traffic to pods. The following service types exist in Kubernetes:

- **NodePort** is a port mapping on the node running the pod, allowing direct access to the application via the node IP and port.
- **ClusterIP** is an internal IP address used within the Kubernetes cluster for internal-only communication.
- **LoadBalancer** is the underlying cloud providers cloud-based load balancer with an external IP address. The load balancer backend pool is connected to the requested pods.
- **ExternalName** is a DNS entry for access to an application running on pods.

In addition to the aforementioned four service types, you also have the Ingress Controller. An Ingress Controller works at layer 7 of the networking OSI model. An Ingress Controller provides configurable traffic routing, TLS termination, and reverse proxy. An Ingress Controller has ingress rules and routes to Kubernetes services. A common use of an Ingress Controller is the ability to route from a single public IP address to multiple services in a Kubernetes cluster. The most common Ingress Controller in Kubernetes is the NGINX ingress controller.

Storage

Files in containers are ephemeral. When containers are restarted, files are lost. If there is a need to preserve data even when a container restarts, Kubernetes volumes can be used. Kubernetes supports many types of volumes; however, the most common options are volumes or persistent volumes. With volumes, when a container is destroyed, the volume will cease to exist as well. With persistent volumes, when a container is destroyed, the data will continue to exist. There are two types of persistent volumes in Kubernetes: the first being persistent volume and the second being persistent volume claim. Persistent volume is a resource in the cluster independent of any pod. Persistent volume claim is requested for a specific pod in the namespace where the pod is. At the

core, a volume is just a directory with data in it that containers in a pod can access. Here is the full list of volumes Kubernetes supports:

```
awsElasticBlockStore
azureDisk
azureFile
cephfs
cinder
configMap
csi
downwardAPI
emptyDir
fc (fibre channel)
flexVolume
flocker
gcePersistentDisk
gitRepo (deprecated)
glusterfs
hostPath
iscsi
local
nfs
persistentVolumeClaim
projected
portworxVolume
quobyte
rbd
scaleIO
secret
storageos
vsphereVolume
```

In Kubernetes we also have something known as storage classes. Storage classes work with dynamic provisioning of persistent storage volumes in Kubernetes. Dynamic storage provisioning is when storage is ordered with a predefined type and configuration without having to know the details about how to provision the physical or cloud storage

device. Storage classes abstract all the details of a specific storage type that is then used by developers or cloud providers. Storage classes give administrators a way to describe the “classes” of storage they offer. Classes map to service levels and/or backup policies.

Summary

In this chapter, we introduced you to Kubernetes. As you embark on this journey into Azure Kubernetes Service, you will now be equipped with core knowledge of Kubernetes and its components. Throughout this chapter, we explored Kubernetes architecture and learned about master and worker nodes and key features such as namespaces, labels, jobs, services, and replicaset. We also learned about configuration management using ConfigMaps and Secrets when information needs to be secured. We learned that Kubernetes has a web-based user interface that can be used for some management tasks. Last but not least, we explored how networking and storage work in Kubernetes.

CHAPTER 4

kubectl Overview

`kubectl` is a command line interface for executing commands against Kubernetes clusters. You can use `kubectl` to deploy applications, check and manage Kubernetes cluster resources, and examine logs.

In this chapter, we will discuss the various `kubectl` commands that you will use for your cluster operations. We will cover the basic commands and provide examples of how to use `kubectl` for common operations such as application management, debugging, and cluster management. By the end of this chapter, you will be able to perform basic operations on a Kubernetes cluster using `kubectl`.

Introduction to kubectl

When you execute an operation in `kubectl`, it looks for a file named `config` in the `$HOME/.kube` directory. If you want to use `kubeconfig` files stored in a different directory, you can do so by either setting the `KUBECONFIG` environment variable or by setting the `--kubeconfig` flag.

`kubeconfig` files are used to organize information about clusters, users, namespaces, and authentication mechanisms. `kubectl` uses `kubeconfig` files to choose a cluster and communicate with the API server of a cluster. Also, you can also define contexts to switch between clusters and namespaces quickly.

In a `kubeconfig` file, A context element is used to group access parameters under a convenient name. There are three parameters for each context: cluster, namespace, and user. `kubectl` uses parameters from the current context to communicate with the cluster by default.

Note `kubectl` is installed by default in Azure Cloud Shell. For a complete guide on organizing cluster access using kubeconfig files, refer to the official Kubernetes documentation found on the following URL: <https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/>.

Almost all `kubectl` commands will typically belong to one of the categories listed in Table 4-1.

Table 4-1. *kubectl Command Categories*

Command Type	Usage	Description
Declarative Resource Management	Development and operations	Used to manage Kubernetes workloads using Resource Config declaratively.
Imperative Resource Management	Development only	Use these commands to manage Kubernetes workloads using command line arguments and flags.
Printing Workload State	Debugging	Includes commands for operations such as printing summarized state and information about resources, printing complete state and information about resources, printing specific fields from resources, and query resources matching labels.
Interacting with Containers	Debugging	Used for debugging operations such as Exec, Attach, Cp, and Logs and includes commands for operations such as printing container logs, printing cluster events, executing or attaching to a container, and copying files from containers in the cluster to a user's filesystem.
Cluster Management	Debugging	Users need to perform operations on cluster nodes, and <code>kubectl</code> supports commands for cluster operations such as drain and cordon nodes.

Note `kubectl` is installed by default in Azure Cloud Shell. To install `kubectl` locally, execute `az aks install-cli` command in Azure CLI.

kubectl Basics

This section provides a high-level overview for the basic kubectl commands. Throughout the book, you will notice various kubectl commands in use. We will only cover the most commonly used commands in this section. The examples shown in this section relates to deploying a simple nginx cluster in Kubernetes.

kubectl Syntax

The following is the syntax to run kubectl commands from a terminal:

```
kubectl [command] [TYPE] [NAME] [flags]
```

- **command:** This specifies the operation a user wants to perform on resources, such as create, get, and delete.
- **TYPE:** This denotes the resource. Remember that resource types are case-insensitive, and you can use singular, plural, or abbreviated forms to reference a resource type.
- **NAME:** This denotes the resource name. A resource name is case-sensitive. If you do not provide a name, details of all the resources will be displayed.
- **flags:** flags are optional. For instance, you can leverage the `-s` or `--server` flag to specify the address and port of the Kubernetes API server.

Note For a complete reference of operations you can perform with kubectl, visit kubectl reference at <https://kubernetes.io/docs/reference/kubectl/kubectl/>.

You can list all the supported resource types and their alias by running `kubectl api-resources` in a terminal.

Formatting Output in kubectl

The default output for any kubectl command is plain text. In order to generate an output in a specific format, you can use the `-o` or `--output` flag. The following is the syntax you need to use:

```
kubectl [command] [TYPE] [NAME] -o <output_format>
```

Table 4-2 lists the output formats supported depending on the kubectl operation that you have executed.

Table 4-2. *kubectl Output Formats*

Output Format	Description
-o custom-columns=<spec>	Displays a table using a comma-separated list of custom columns.
-o custom-columns-file=<filename>	Displays a table using the custom columns template in the <filename> file.
-o json	Prints a JSON-formatted API object.
-o jsonpath=<template>	Displays the fields defined in a jsonpath expression.
-o jsonpath-file=<filename>	Displays the fields defined by the jsonpath expression in the <filename> file.
-o name	Displays only the resource name and nothing else.
-o wide	Displays in the plain text format with any additional information. For pods, the node name is included.
-o yaml	Displays a YAML-formatted API object.

Listing Kubernetes Resources

When you work with Kubernetes clusters, you may need to list the Kubernetes deployment resources in a namespace. Here the deployments are the resources that manage pod replicas. The following example lists the deployments in the kube-system namespace as shown in Listing 4-1.

```
kubectl get deployments --namespace kube-system
```

Listing 4-1. Deployment information for kube-system namespace

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
event-exporter-v0.2.3	1	1	1	1	14d
fluentd-gcp-scaler	1	1	1	1	14d
heapster-v1.6.0-beta.1	1	1	1	1	14d
kube-dns	2	2	2	2	14d
kube-dns-autoscaler	1	1	1	1	14d
l7-default-backend	1	1	1	1	14d
metrics-server-v0.3.1	1	1	1	1	14d

If you want to print detailed information about a specific deployment in a namespace, you can use the following syntax. In this example, we are printing the information about the kube-dns deployment which you can see in Listing 4-2.

```
kubectl describe deployment kube-dns --namespace kube-system
```

Listing 4-2. Getting deployment information for kube-dns

```
Name:                kube-dns
Namespace:           kube-system
CreationTimestamp:    Wed, 29 May 2019 00:28:50 +1030
Labels:               addonmanager.kubernetes.io/mode=Reconcile
                     k8s-app=kube-dns
                     kubernetes.io/cluster-service=true
Annotations:          deployment.kubernetes.io/revision: 2
...
```

Creating a Resource from Config

You can create or update Kubernetes resources from either a remote config hosted in a remote repository such as GitHub or a local config stored in your computer.

Remote Config

```
kubectl apply -f https://k8s.io/examples/application/deployment.yaml
```

Local Config

```
kubectl apply -f ./examples/nginx/nginx.yaml
```

Listing 4-3 shows the common output in both these scenarios.

Listing 4-3. Output of `kubectl apply`

```
service/nginx created
deployment.apps/nginx-deployment created
```

Generating a Config from a Command

You can generate config for a deployment resource, and the config can then be applied to a Kubernetes cluster by writing the output to a file and then executing `kubectl apply -f <yaml-file-name>`. In the following example, we are creating a deployment called `nginx` from the `nginx` image and redirecting the output to a `yaml` file:

```
kubectl create deployment nginx --dry-run -o yaml --image nginx
```

Listing 4-4 shows the output of the `yaml` file.

Listing 4-4. `nginx` deployment `yaml` file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null # delete this
  labels:
    app: nginx
  name: nginx
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx
  strategy: {} # delete this
  template:
    metadata:
```

```

    creationTimestamp: null # delete this
    labels:
      app: nginx
  spec:
    containers:
      - image: nginx
        name: nginx
        resources: {} # delete this
  status: {} # delete this

```

Viewing Pods Associated with Resources

One of the most common scenarios users will encounter while working with Kubernetes clusters is listing pod information. In the following example, we are listing all the pods created by the nginx deployment using pod labels. Listing 4-5 shows the output of this operation.

```
kubectl get pods -l app=nginx
```

Listing 4-5. Listing all the pods of nginx deployment

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-5c678s55ff-b2xfk	1/1	Running	0	10m
nginx-deployment-5c678s55ff-rx569	1/1	Running	0	10m
nginx-deployment-5c678s55ff-s7xcv	1/1	Running	0	10m

Debugging Containers

When users want to debug the containers running on their Kubernetes clusters, first thing to examine are the logs. In the following example, we list the logs from all the pods of the nginx deployment:

```
kubectl logs -l app=nginx
```

If you want to obtain a shell into a specific pod's container, you can leverage exec operation with kubectl as follows:

```
kubectl exec -i -t nginx-deployment-5c678s55ff-b2xfk bash
```


Events are a resource type in Kubernetes that are automatically created when other resources have state changes, errors, or other messages that need to be broadcasted to the system. For an example, `kubectl describe pod <podname>` will list the events at the end of the output for a given pod.

Using `kubectl get events` will allow you to extract the events from the resource's API directly.

Listing 4-6 illustrates filtering events with `kubectl get events` command.

Listing 4-6. `kubectl get events`

```
#Filter warning only
kubectl get events --field-selector type=Warning

#Filter no pod events only
kubectl get events --field-selector involvedObject.kind!=Pod

#Filter events for a single node named "minikube"
kubectl get events --field-selector involvedObject.
kind=Node,involvedObject.name=mi
```

Common Operations with kubectl

To familiarize yourself with how you can use `kubectl` for common operations, we have provided some following examples. Though the following code excerpts don't cover the entire breadth of using `kubectl` for application management, debugging, and cluster management, they provide adequate information to the reader on the most common scenarios they might encounter.

Note As we don't expect to cover every `kubectl` command in this chapter, we would recommend readers to refer the `kubectl` cheat sheet available at <https://kubernetes.io/docs/reference/kubectl/cheatsheet/> for further reading.

kubectl apply

The apply operation will apply or update a Kubernetes resource from a file or stdin. The resource name must be specified, and it will be automatically created if it doesn't exist.

Listing 4-7 shows some common examples of using the apply operation.

Listing 4-7. kubectl apply

```
# Creating a service using the definition in my-service.yaml.
kubectl apply -f my-service.yaml

# Creating a replication controller using the definition in my-controller.yaml.
kubectl apply -f my-controller.yaml

# Creating the objects that are defined in any .yaml, .yml, or .json file
within the <mydirectory> directory.
kubectl apply -f <mydirectory>
```

kubectl get

The get operation lists one or more resources. You can use `kubectl api-resources` command for a complete list of supported resources with get.

Listing 4-8 shows some common examples of using the get operation.

Listing 4-8. kubectl get

```
# List all pods in plain text output format.
kubectl get pods

# List all pods in plain text output format and include additional
information such as node name.
kubectl get pods -o wide

# List the replication controller with the specified name in plain text
output format.
kubectl get replicationcontroller <rc-name>

# List all replication controllers and services together in plain text
output format.
kubectl get rc,services
```

List all daemon sets, including uninitialized ones, in plain text output format.

```
kubectl get ds --include-uninitialized
```

List all pods running on node srv01

```
kubectl get pods --field-selector=spec.nodeName=srv01
```

Note You can shorten and replace the “replicationcontroller” resource type with its alias “rc” as seen in the preceding listing.

kubectl describe

The describe operations shows the detailed information of a specific resource(s), including the uninitialized ones by default. You can use `kubectl api-resources` command for a complete list of supported resources with describe.

Listing 4-9 shows some common examples of using the describe operation.

Listing 4-9. kubectl describe

Displaying the details of the node with name <my-node>.

```
kubectl describe nodes <my-node>
```

Displaying the details of the pod with name <my-pod>.

```
kubectl describe pods/<my-pod>
```

Displaying the details of all the pods that are managed by the replication controller named <rc-myrepctl>.

```
kubectl describe pods <rc-myrepctl >
```

Describe all pods, not including uninitialized ones

```
kubectl describe pods --include-uninitialized=false
```

Note Any pods that are created by a replication controller get prefixed with the name of that replication controller.

kubectl delete

You can use delete operation to delete resources by their filenames, stdin, or specifying label selectors, names, resource selectors, or resources. Keep in mind that only one type of the arguments can be provided: they can be either filenames or resources and names or resources and label selector.

Listing 4-10 shows some common examples of using the delete operation.

Listing 4-10. kubectl delete

```
# Deleting a pod using the type and name specified in the pod.yaml file.
kubectl delete -f pod.yaml

# Deleting all the pods and services that have the label name=<my-label>.
kubectl delete pods,services -l name=<my-label>

# Deleting all the pods and services that have the label name=<my-label>,
including uninitialized ones.
kubectl delete pods,services -l name=<my-label> --include-uninitialized

# Deleting all pods, including uninitialized ones.
kubectl delete pods --all
```

kubectl exec

The exec operation executes a command against a container in a pod.

Listing 4-11 shows some common examples of using the exec operation.

Listing 4-11. kubectl exec

```
# Get output from running 'date' from pod <my-pod>. By default, output is
from the first container.
kubectl exec <my-pod> date

# Get output from running 'date' in container <my-container> of pod <my-pod>
kubectl exec <my-pod> -c <my-container> date

# Get an interactive TTY and run /bin/bash from pod <my-pod>. The default
output is always from the first container in the pod.
kubectl exec -ti <my-pod> /bin/bash
```

kubectl logs

The logs operation can print the logs for a specific container in a pod or for a specified resource in a Kubernetes cluster. If there is only one container in a pod, providing the container name is optional.

Listing 4-12 shows some common examples of using the logs operation.

Listing 4-12. kubectl logs

```
# Returns a snapshot of the logs from pod <my-pod>.
```

```
kubectl logs <my-pod>
```

```
# Start streaming the logs from pod <my-pod>. This is similar to the 'tail  
-f' Linux command.
```

```
kubectl logs -f <my-pod>
```

Summary

kubectl is the primary tooling that you will use to manage your Kubernetes environment. You can use kubectl to declaratively manage applications in Kubernetes, perform debugging, and administer your Kubernetes clusters.

In this chapter, you learned about the basics of kubectl commands. We discussed the basic operations in kubectl by examining code samples on deploying a nginx image in a Kubernetes cluster. Finally, we explored the most common kubectl operations and their usage.

CHAPTER 5

Deploying Azure Kubernetes Service

Azure Kubernetes Service (AKS) simplifies the deployment of a Kubernetes cluster by providing a managed Kubernetes-as-a-Service platform. The operational complexity of managing Kubernetes is reduced by off-loading routine tasks such as health monitoring and maintenance as well as master node management to the Azure platform.

In this chapter, we are going to explore how to deploy an AKS cluster using the Azure Portal, Azure CLI, Azure Resource Manager (ARM) templates, and Terraform. We will review the process for each deployment option followed by explanations on additional feature configurations such as advanced networking, Azure Active Directory integration, and monitoring. By the end of this chapter, you will have a good knowledge about the AKS deployment process, options, and procedure.

Azure Kubernetes Service Deployment Overview

You can deploy an AKS cluster using several methods. Each method has its own merits, and choosing how you want to deploy an AKS cluster is dependent on your preference and scenario. However, you will have to provide a few mandatory parameters that are required to deploy an AKS cluster in all these methods. We will discuss what these parameters are in the upcoming sections.

Deployment Through the Azure Portal

Creating an AKS cluster using the Azure Portal is a straightforward process. The following explains the procedure to do so.

In the Azure Marketplace, select + **Create a resource** ► **Containers** ► **Kubernetes Service**. In the **Create Kubernetes Cluster** page, configure the following options:

1. On the **Basics** section, the following options need to be configured:
 - a. **Project details:** Under this section, select the Azure subscription where you need the AKS cluster to be created.
 - b. **Cluster details:** Select or create an Azure resource group for the AKS cluster, provide a value for **Kubernetes cluster name**, provide an Azure **region** to deploy the AKS cluster, select the desired **Kubernetes version**, and finally provide a **DNS name prefix** for the AKS cluster.
 - c. **Primary node pool:** In this section, you need to select a **VM size** for the AKS nodes from the Azure VM SKUs. Remember that once an AKS cluster is created, the VM size cannot be changed. Select the **VM size** and **the node count**. You can start by setting the Node Count to 1.
 - d. Click **Next: Scale** ►.

[Home](#) >
[New](#) >
[Create Kubernetes cluster](#)

Create Kubernetes cluster

[Basics](#)
[Scale](#)
[Authentication](#)
[Networking](#)
[Monitoring](#)
[Tags](#)
[Review + create](#)

Azure Kubernetes Service (AKS) manages your hosted Kubernetes environment, making it quick and easy to deploy and manage containerized applications without container orchestration expertise. It also eliminates the burden of ongoing operations and maintenance by provisioning, upgrading, and scaling resources on demand, without taking your applications offline. [Learn more about Azure Kubernetes Service](#)

Project details

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

* Subscription ⓘ

MVP Azure

* Resource group ⓘ

(New) jcbaksrcg01

[Create new](#)

Cluster details

* Kubernetes cluster name ⓘ

jcbakscu01

* Region ⓘ

(Asia Pacific) Australia Southeast

* Kubernetes version ⓘ

1.13.10 (default)

* DNS name prefix ⓘ

jcbakscu01-dns

Primary node pool

The number and size of nodes in the primary node pool in your cluster. For production workloads, at least 3 nodes are recommended for resiliency. For development or test workloads, only one node is required. You will not be able to change the node size after cluster creation, but you will be able to change the number of nodes in your cluster after creation. If you would like additional node pools, you will need to enable the "X" feature on the "Scale" tab which will allow you to add more node pools after creating the cluster. [Learn more about node pools in Azure Kubernetes Service](#)

* Node size ⓘ

Standard DS2 v2

2 vcpus, 7 GiB memory

[Change size](#)

* Node count ⓘ

1

Review + create

< Previous

Next : Scale >

Figure 5-1. Create a Kubernetes cluster Basics section

- Keep the default options at the **Scale** section; after that, click **Next: Authentication** ►.

Note We will be discussing the scaling options for AKS in detail in Chapter 7, “Operating Azure Kubernetes Service.”

3. In the **Authentication** section:
 - a. You can either create a new service principal by leaving the Service Principal field as it is or choose **Configure service principal** to use an existing one. Remember if you chose to use an existing one, you will have to provide the **service principal name (SPN) client ID** and **secret** in the next pop-up blade.
 - b. **Enable RBAC**: Set this option to **Yes** to allow Kubernetes role-based access controls (RBAC) which provides more fine-grained control over access AKS cluster resources
 - c. Click **Next: Networking** ➤.

Home > New > Create Kubernetes cluster

Create Kubernetes cluster

Basics Scale **Authentication** Networking Monitoring Tags Review + create

The **cluster infrastructure** service principal is used by the Kubernetes cluster to manage cloud resources attached to the cluster. [Learn more about service principals in AKS](#)

Kubernetes authentication and authorization is used by the Kubernetes cluster to control user access to the cluster as well as what the user may do once authenticated. [Learn more about Kubernetes authentication](#)

Cluster infrastructure

* Service principal ⓘ (new) default service principal
[Configure service principal](#)

Kubernetes authentication and authorization

Enable RBAC ⓘ No Yes

Review + create < Previous Next : Networking >

Figure 5-2. Create a Kubernetes cluster Authentication section

4. Leave the Network configuration radio button to **Basic** settings under the **Networking** section to use **kubenet** with a default VNet configuration. Selecting **Advanced** will redirect you to configure the following which allows you to use an **Azure CNI** with further options to customize your VNet. Click **Next: Monitoring** ➤.

BasicsScaleAuthenticationNetworkingMonitoringTagsReview + create

You can change networking settings for your cluster, including enabling HTTP application routing and configuring your network using either the 'Basic' or 'Advanced' options:

- **'Basic'** networking creates a new VNet for your cluster using default values.
- **'Advanced'** networking allows clusters to use a new or existing VNet with customizable addresses. Application pods are connected directly to the VNet, which allows for native integration with VNet features.

[Learn more about networking in Azure Kubernetes Service](#)

HTTP application routing ⓘ

YesNo

Network configuration ⓘ

BasicAdvanced

Configure virtual networks

* Virtual network ⓘ

(new) jcbaksg01-vnet

Create new

* Cluster subnet ⓘ

(new) default (10.240.0.0/16)

* Kubernetes service address range ⓘ

10.0.0.0/16

✓

* Kubernetes DNS service IP address ⓘ

10.0.0.10

* Docker Bridge address ⓘ

172.17.0.1/16

✓

Figure 5-3. Create a Kubernetes cluster Networking section

5. Under **Monitoring** section, leave the **Enable container monitoring** option to **Yes**. Here you can either create a new Log Analytics workspace for your new AKS cluster or create a new one. Once done, click the **Review + Create** button at the bottom of the screen. Once the validation is completed, click **Create**.

Create Kubernetes cluster

Validation passed

[Basics](#)
[Scale](#)
[Authentication](#)
[Networking](#)
[Monitoring](#)
[Tags](#)
[Review + create](#)

Basics

Subscription	MVP Azure
Resource group	(new) jcbaksrcg01
Region	(Asia Pacific) Australia Southeast
Kubernetes cluster name	jcbakscclu01
Kubernetes version	1.13.10
DNS name prefix	jcbakscclu01-dns
Node count	1
Node size	Standard_DS2_v2

Scale

Virtual nodes	Disabled
VM scale sets	Disabled

Authentication

Enable RBAC	Yes
-------------	-----

Networking

HTTP application routing	No
Network configuration	Advanced
Virtual network	(new) jcbaksrcg01-vnet
Cluster subnet	(new) default
Kubernetes service address range	10.0.0.0/16
Kubernetes DNS service IP address	10.0.0.10
Docker Bridge address	172.17.0.1/16

Monitoring

Enable container monitoring	Yes
Log Analytics workspace	(new) DefaultWorkspace-db2a3cfc-8ce3-4bf1-8959-04dfc9e8282d-ASE

Tags

(none)

[Create](#)
[< Previous](#)
[Next >](#)
[Download a template for automation](#)

Figure 5-4. Create a Kubernetes cluster Validation section

The process to create an AKS cluster will take few minutes to complete the deployment. Once the deployment is completed, you can see the status of your AKS cluster by visiting its dashboard by clicking **Go to resource** under **Next steps** or by searching the resource group name or AKS cluster name in the **search** bar on top of the screen.

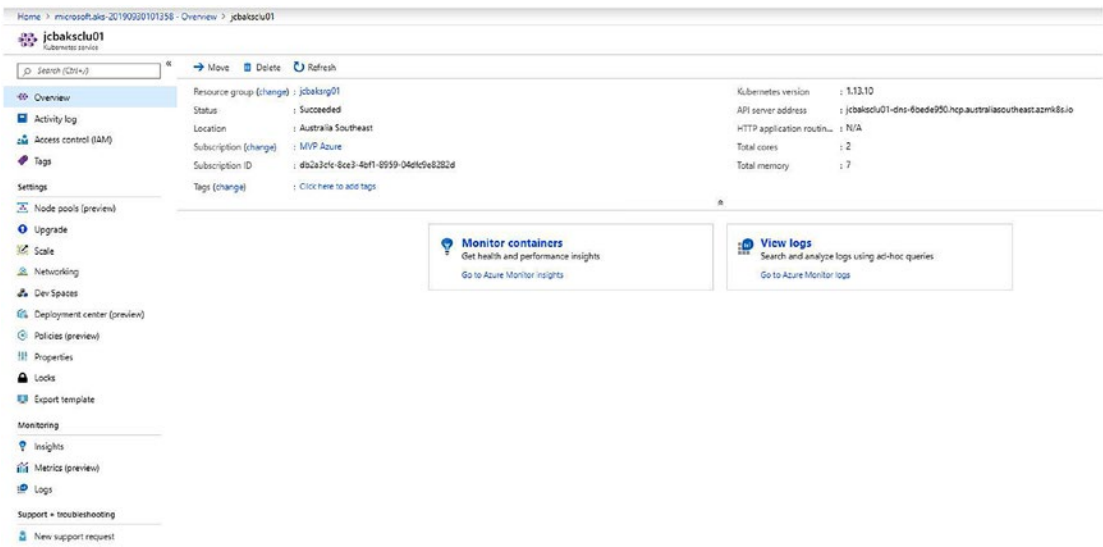


Figure 5-5. AKS cluster dashboard

Deployment Through Azure CLI

Azure CLI is a command line tool for managing your Azure resources. It is designed as a cross-platform tool that can be deployed on Windows, Linux, or MacOS systems. To provide a greater Azure CLI experience, the Azure portal provides **Azure Cloud Shell**, which is an interactive shell environment that you can use using your browser. The advantage of using Azure Cloud Shell is that you can use it with either preinstalled bash or PowerShell Azure CLI commands without installing anything on your local computer.

Note For the purpose of trying the instructions in this section, you may choose to either use Azure CLI installed in your local computer or leverage Azure Cloud Shell.

1. Launch an Azure Cloud Shell session by clicking the **Cloud Shell** button on the top-right menu bar in the Azure Portal.



Figure 5-6. Launch an Azure Cloud Shell

2. Create a resource group to deploy your AKS cluster by entering the following Azure CLI command.

Listing 5-1. Create a resource group for the AKS cluster

```
az group create --name jcbakserg01 --location eastus
```

You should see the following output if the resource group creation was successful.

Listing 5-2. Create a resource group output

```
{
  "id": "/subscriptions/<guid>/jcbaskrg01",
  "location": "eastus",
  "managedBy": null,
  "name": "jcbaskrg02",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null,
  "type": "Microsoft.Resources/resourceGroups"
}
```

3. Use the `az aks create` command to create the AKS cluster. The following example creates a cluster named `jcbakscu02` with one node, and the `--enable-addons monitoring` parameter will enable Azure Monitor for containers for this cluster. After few minutes, once the cluster creation is completed, Azure CLI will return a JSON-formatted cluster information in the Azure Cloud Shell window.

Listing 5-3. Create an AKS cluster

```
az aks create --resource-group jcbakserg01 --name jcbakscu01 --node-count 1
--enable-addons monitoring --generate-ssh-keys
```

Deployment Through Azure Resource Manager Templates

Azure Resource Manager (ARM) templates introduce infrastructure as code capabilities for your Azure deployments. ARM templates are JavaScript Object Notation (JSON) files that define the infrastructure and configuration of an Azure deployment. An ARM template uses a declarative syntax, and you can specify the resources you intend to deploy and their respective configuration by using an ARM template.

Before creating an AKS cluster using an ARM template, you need to provide an SSH public key and Azure Active Directory service principal first.

Create an SSH Key Pair

An SSH key pair is required to connect and access your AKS nodes. By executing the `ssh-keygen` command in an Azure Cloud Shell session according to the following example, you will be generating an SSH public and private key pair with RSA encryption of a 2048-bit length in the `~/.ssh` directory inside your Azure Cloud Shell file storage.

Listing 5-4. Create an SSH key pair in Azure Cloud Shell

```
ssh-keygen -t rsa -b 2048
```

Create a Service Principal

An Azure Active Directory service principal name (SPN) is required to allow an AKS cluster to interact with other Azure resources in your Azure environment. By executing the `az ad sp create-for-rbac` CLI command, you can create the necessary service principal for this exercise. The `--skip-assignment` parameter prevents any additional permissions being assigned. By default, this service principal is valid only for a year.

Listing 5-5. Create a service principal in Azure Cloud Shell

```
az ad sp create-for-rbac --skip-assignment
```

Make a note of the **appId** and **password** values from the output generated. These are required to populate parameters in the ARM template.

Listing 5-6. JSON output of the create service principal operation

```
{
  "appId": "141b2bef-9350-4e80-a0fa-a6aa456750a9",
  "displayName": "azure-cli-2019-09-30-01-39-37",
  "name": "http://azure-cli-2019-09-30-01-39-37",
  "password": "182bb4e7-b53f-4cc4-811d-c72ba828a75d",
  "tenant": "<tenant id>"
}
```

Using an Azure Resource Manager QuickStart Template

If you are not an expert on ARM templates, you can always leverage an Azure Resource Manager QuickStart template to start with.

Note In this example, we are going to use the 101-aks QuickStart template to explain the process of deploying an AKS cluster using an Azure Resource Manager template. For more examples, visit the following URL:

<https://azure.microsoft.com/en-au/resources/templates/?term=Azure%20Kubernetes%20Service>

1. Navigate to the following URL to open the 101-aks QuickStart template and click **Deploying to Azure**:

<https://azure.microsoft.com/en-au/resources/templates/101-aks/>

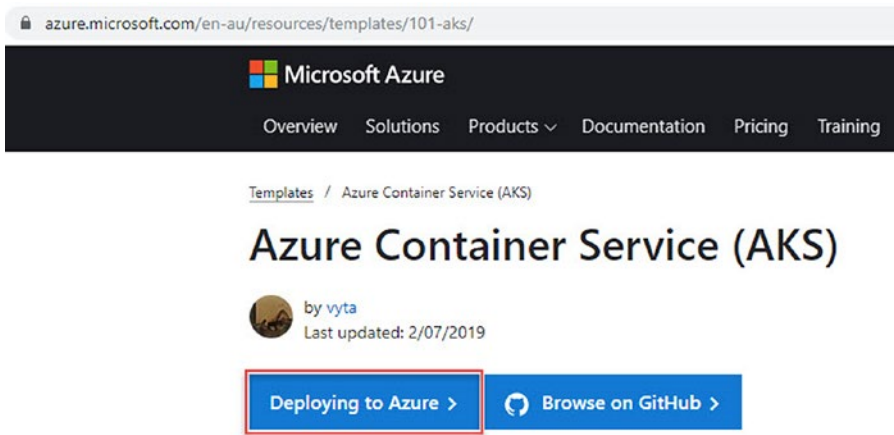


Figure 5-7. Azure QuickStart template 101-aks

2. Enter and/or configure the following values in the template:
 - a. **Subscription:** Select the Azure subscription where you want to deploy the AKS cluster.
 - b. **Resource group:** You can either select an existing resource group or select **Create new** to provide a unique name to create a new resource group and click **OK**.
 - c. **Location:** Select the Azure region for your AKS cluster.
 - d. **Cluster name:** Provide a unique name for the AKS cluster.
 - e. **DNS prefix:** Provide a unique DNS prefix for your cluster.
 - f. **Linux admin username:** Provide a username to connect using SSH.
 - g. **SSH RSA public key:** Enter the public part of your SSH key pair (by default, the contents of `~/.ssh/id_rsa.pub`).
 - h. **Service principal client ID:** Provide the **appId** value generated in the previous section.
 - i. **Service principal client Secret:** Provide the **password** generated in the previous section.
 - j. Click the **I agree to the terms and conditions stated above:** checkbox to agree to the terms and conditions.

[Home](#) > [Azure Container Service \(AKS\)](#)

Azure Container Service (AKS)

Azure quickstart template

101-aks
1 resource

[Edit template](#)
[Edit paramet...](#)
[Learn more](#)

BASICS

* Subscription

MVP Azure

* Resource group

jcbaksrcg01

[Create new](#)

* Location

(Asia Pacific) Australia Southeast

SETTINGS

Cluster Name

jcbakscslu01

Location

[resourceGroup().location]

* Dns Prefix

jcbakscslu01

Os Disk Size GB

0

Agent Count

1

Agent VM Size

Standard_DS2_v2

* Linux Admin Username

jcbakscslu01

* Ssh RSA Public Key

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQACrY21Kr5FNiBbANVMUHG7G68...

* Service Principal Client Id

.....

* Service Principal Client Secret

.....

Os Type

Linux

Kubernetes Version

1.14.6

TERMS AND CONDITIONS

[Template information](#)
[Azure Marketplace Terms](#)
[Azure Marketplace](#)

By clicking "Purchase," I agree to the applicable legal terms associated with the offering; (b) authorize Microsoft to charge or bill my current payment method for the fees associated the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s); and (c) agree that, if the deployment involves 3rd party offerings, Microsoft may share my contact information and other details of such deployment with the publisher of that offering.

☐ I agree to the terms and conditions stated above

Purchase

Figure 5-8. Deployment screen for 101-aks QuickStart template

3. Click **Purchase**. Your AKS cluster deployment will take few minutes to complete.

Note You can use the **Edit Template** or **Edit parameters** buttons to either customize the ARM template or edit the parameters provided in this QuickStart template.

Deployment Through Terraform

Terraform is an Infrastructure-as-Code (IaC) tool designed for building, changing, and versioning infrastructure safely and efficiently. **Configuration files** in Terraform define the components required to run an application. An **execution plan** is generated in Terraform to describe the instructions to reach the desired configuration state, and then it is executed to build the described infrastructure. In case of a configuration change, Terraform is able determine the changes and to create incremental execution plans which can then be applied.

The steps to create an AKS cluster with Terraform is well documented at (<https://docs.microsoft.com/en-us/azure/terraform/terraform-create-k8s-cluster-with-tf-and-aks>).

Note Terraform is preinstalled by default in the Azure Cloud Shell. If you need to set up Terraform locally to follow the instructions in the preceding article, please refer to the following URL:

<https://docs.microsoft.com/en-us/azure/virtual-machines/linux/terraform-install-configure#install-terraform>

Connecting to Your AKS Cluster

You can use `kubectl`, the Kubernetes command line client, in order to manage your AKS cluster. If you are using Azure Cloud Shell, `kubectl` is preinstalled. If you want to install `kubectl` locally on your local computer (where Azure CLI is already installed), you can use the following command.

Listing 5-7. Installing `kubectl` on a local installation of Azure CLI

```
az aks install-cli
```

By using the `az aks get-credentials` command, you can configure `kubectl` to connect to your Kubernetes cluster. This downloads the required credentials and configures the Kubernetes CLI to use them.

Listing 5-8. `az aks get-credentials` command

```
az aks get-credentials --resource-group jcbakserg01 --name jcbaksclu01
```

Then, you can execute the `kubectl get` command to verify the connection to you cluster and see if it returns a list of cluster nodes.

Listing 5-9. `kubectl get` command

```
kubectl get nodes
```

The following is the sample output that shows the nodes in the Kubernetes cluster **jcbaksclu01** created through the previous methods. The status should be **Ready** for all the nodes before you deploy any application to your AKS cluster.

Listing 5-10. `kubectl get` command output for jcbaksclu01

NAME	STATUS	ROLES	AGE	VERSION
aks-agentpool-26412741-0	Ready	agent	120m	v1.13.10

Summary

You can deploy Azure Kubernetes Service using several methods. As explained earlier in this chapter, it is up to you to decide which method suits your deployment requirements. This chapter serves as an introduction to deploying AKS and configuring the basic parameters required to get your AKS cluster up and running.

In this chapter, you learned about what creates an AKS cluster using four different methods, through Azure Portal, Azure CLI (either via a locally installed instance or via Azure Cloud Shell), Azure Resource Manager templates, and finally a very popular third-party Infrastructure-as-a-Code tool called “Terraform.” We reviewed the process of initial configuration for your AKS using each of these methods. Lastly, we briefly discussed about how you can connect to your AKS cluster using `kubectl`, the command line tool for Kubernetes.

CHAPTER 6

Deploying and Using Rancher with Azure Kubernetes Service

As you continue along your journey into the container world, you will get to the point of critical mass. There will be a need to run an orchestration platform to handle the life cycle of containers. Within Chapter 3, “Inside Kubernetes,” we dove into Kubernetes, the most common orchestration platform. In Chapter 5, “Deploying Azure Kubernetes Service,” we covered how to deploy Azure Kubernetes Service (AKS). Kubernetes can be complex for anyone starting out with container orchestration platforms.

Microsoft’s managed Kubernetes service AKS removes some of the complexity from running a Kubernetes cluster; however, it can still be a challenge to run a Kubernetes cluster, including all the things that come along with it such as operating multiple Kubernetes clusters, scaling in the cluster, networking, RBAC, monitoring, deploying bundled solutions via HELM charts (to be covered in a later chapter), and more. There are third-party solutions on the market that can reduce the complexity of running Kubernetes. Rancher is one of this solution if not arguably the best.

In this chapter, we are going to give an overview of Rancher and also will explore how Rancher can be used together with AKS.

What Is Rancher?

In a nutshell, Rancher is an open source solution that can be used to deploy and operate a single or many Kubernetes clusters. Rancher can deploy and manage Kubernetes clusters across on-premises or cloud providers such as AWS, GCP, Digital Ocean, and

Azure. It can be used to deploy and manage your own Kubernetes cluster on your own infrastructure or even managed cluster services from cloud providers, for example, Azure Kubernetes Service.

Unlike many other open source solutions out there on the market, Rancher is completely free. It does not have a community edition and an enterprise edition you have to pay for. With Rancher, you get all of the features when you deploy it. The way Rancher supports itself financially is through paid support options designed for organizations that run Rancher in production.

Because Rancher can operate Kubernetes clusters virtually anywhere, it can also be utilized to migrate resources between providers.

Rancher overall helps simplify the administration of Kubernetes. Some of the ways it simplifies the administration are by centralizing the authentication and access control, bringing in monitoring out of the box with Prometheus and Grafana, having an application library of its own, and HELM charts and streamlined Kubernetes version upgrades.

While Rancher abstracts much of the complexity of managing a Kubernetes cluster, it also allows for advanced administration if desired. For example, a Kubernetes administrator can access `kubectl` right through the Rancher portal.

Why Use Rancher with Kubernetes?

One of the most common questions that come up when someone learns about Rancher is: “Why should I use Rancher with Kubernetes vs. just using Kubernetes on its own?”

The answer to this is there are many reasons to use Rancher with Kubernetes; however, there may be scenarios where it does not make sense to use Rancher. Here we will look at the reasons to use Rancher. Let’s go through them:

- **Deployment and upgrade of Kubernetes clusters:** Deploying and upgrading Kubernetes clusters via Rancher is streamlined and seamless.
- **User interface and API:** Rancher provides a streamlined user interface for those that use Kubernetes. Rancher also provides an API to interface with.
- **Centralize the management of multiple Kubernetes clusters:** Many organizations are taking a multi-cloud approach, and therefore

it is likely they will run a Kubernetes cluster across multiple cloud providers. When Kubernetes clusters are deployed across cloud providers and even on premises, Rancher can be used to centralize the management of all the clusters from one place. Rancher centralizes management of RBAC, security policy management, capacity management, delegated administration, cluster backup and recovery, logging and monitoring, and more.

- **Centralize and streamline the RBAC of Kubernetes:** Kubernetes authorization and access can be managed easily from Rancher.
- **Rancher comes out of the box with Prometheus and Grafana:** Monitoring Kubernetes is critical. Prometheus and Grafana are common monitoring and visualization tools. Being that these are packaged with Rancher and ready to Monitor Kubernetes reduces the effort of deploying these solutions and getting them ready to monitor Kubernetes.
- **Rancher streamlines Helm charts:** Rancher allows you to load a Helm chart library and/or a Rancher library. These libraries make it easy to deploy applications as pods with ease.
- **Kubernetes adoption:** Drive Kubernetes adoption by lowering the Kubernetes learning curve and allowing coders to focus on developing applications vs. running the applications.

As you can see from the previous list, there is a lot of value in using Rancher in combination with Kubernetes. Next, let's look at deploying Rancher and using it with AKS.

How to Deploy Rancher on Azure

Rancher runs as a container on top of Docker. You can deploy Rancher on-premises or on a cloud provider. In this section, we are going to deploy Rancher on an Ubuntu server running Docker on an Azure IaaS VM.

We will deploy this VM and Rancher using an Azure ARM Template. The ARM Template that we will use deploys an Ubuntu VM with Docker and the latest version of Rancher as a container. The Rancher container is deployed from (<https://hub.docker.com/r/rancher/rancher>) on Docker Hub. This ensures that the latest Rancher version will always be deployed.

The ARM Template we will use is named RancherNode.JSON. Here is the ARM Template code:

```
{
  "$schema": "https://schema.management.azure.com/schemas/2015-01-01/
deploymentTemplate.json#",
  "contentVersion": "1.0.0.0",
  "parameters": {
    "adminUsername": {
      "type": "string",
      "metadata": {
        "description": "Username for the Rancher Node Virtual
Machine."
      }
    },
    "adminPassword": {
      "type": "securestring",
      "metadata": {
        "description": "Password for the Rancher Node Virtual
Machine."
      }
    },
    "dnsNameForPublicIP": {
      "type": "string",
      "metadata": {
        "description": "Unique DNS Name for the Public IP used to
access the Rancher Portal."
      }
    },
    "vmSize": {
      "type": "string",
      "defaultValue": "Standard_F1",
      "metadata": {
        "description": "VM size for the Rancher Node."
      }
    }
  },
```



```

"ubuntuOSVersion": {
  "type": "string",
  "defaultValue": "14.04.4-LTS",
  "metadata": {
    "description": "The Ubuntu version for deploying the Docker
containers. This will pick a fully patched image of this
given Ubuntu version. Allowed values: 14.04.4-LTS, 15.10,
16.04.0-LTS"
  },
  "allowedValues": [
    "14.04.4-LTS",
    "15.10",
    "16.04.0-LTS"
  ]
},
"location": {
  "type": "string",
  "defaultValue": "[resourceGroup().location]",
  "metadata": {
    "description": "Location for all resources."
  }
}
},
"variables": {
  "imagePublisher": "Canonical",
  "imageOffer": "UbuntuServer",
  "nicName": "RancherNodeNic",
  "extensionName": "DockerExtension",
  "addressPrefix": "10.0.0.0/16",
  "subnetName": "RancherSubnet",
  "subnetPrefix": "10.0.0.0/24",
  "diskStorageType": "Standard_LRS",
  "publicIPAddressName": "RancherNodePublicIP",
  "publicIPAddressType": "Dynamic",
  "vmName": "RancherNode",
  "virtualNetworkName": "RancherVNet",

```

```

    "subnetRef": "[resourceId('Microsoft.Network/
virtualNetworks/subnets', variables('virtualNetworkName'),
variables('subnetName'))]"
  },
  "resources": [
    {
      "apiVersion": "2017-04-01",
      "type": "Microsoft.Network/publicIPAddresses",
      "name": "[variables('publicIPAddressName')]",
      "location": "[parameters('location')]",
      "properties": {
        "publicIPAllocationMethod": "[variables('publicIPAddress
Type')]",
        "dnsSettings": {
          "domainNameLabel": "[parameters('dnsNameForPublicIP')]"
        }
      }
    },
    {
      "apiVersion": "2017-04-01",
      "type": "Microsoft.Network/virtualNetworks",
      "name": "[variables('virtualNetworkName')]",
      "location": "[parameters('location')]",
      "properties": {
        "addressSpace": {
          "addressPrefixes": [
            "[variables('addressPrefix')]"
          ]
        },
        "subnets": [
          {
            "name": "[variables('subnetName')]",
            "properties": {
              "addressPrefix": "[variables('subnetPrefix')]"
            }
          }
        ]
      }
    }
  ]
}

```

```

    ]
  }
},
{
  "apiVersion": "2017-04-01",
  "type": "Microsoft.Network/networkInterfaces",
  "name": "[variables('nicName')]",
  "location": "[parameters('location')]",
  "dependsOn": [
    "[concat('Microsoft.Network/publicIPAddresses/', variables('publicIPAddressName'))]",
    "[concat('Microsoft.Network/virtualNetworks/', variables('virtualNetworkName'))]"
  ],
  "properties": {
    "ipConfigurations": [
      {
        "name": "ipconfig1",
        "properties": {
          "privateIPAllocationMethod": "Dynamic",
          "publicIPAddress": {
            "id": "[resourceId('Microsoft.Network/publicIPAddresses', variables('publicIPAddressName'))]"
          },
          "subnet": {
            "id": "[variables('subnetRef')]"
          }
        }
      }
    ]
  }
},
{
  "apiVersion": "2017-03-30",
  "type": "Microsoft.Compute/virtualMachines",

```

```

"name": "[variables('vmName')]",
"location": "[parameters('location')]",
"dependsOn": [
    "[concat('Microsoft.Network/networkInterfaces/',
        variables('nicName'))]"
],
"properties": {
    "hardwareProfile": {
        "vmSize": "[parameters('vmSize')]"
    },
    "osProfile": {
        "computerName": "[variables('vmName')]",
        "adminUsername": "[parameters('adminUsername')]",
        "adminPassword": "[parameters('adminPassword')]"
    },
    "storageProfile": {
        "imageReference": {
            "publisher": "[variables('imagePublisher')]",
            "offer": "[variables('imageOffer')]",
            "sku": "[parameters('ubuntuOSVersion')]",
            "version": "latest"
        },
        "osDisk": {
            "name": "[concat(variables('vmName'), '_OSDisk')]",
            "caching": "ReadWrite",
            "createOption": "FromImage",
            "managedDisk": {
                "storageAccountType": "[variables('diskStorage
                    Type')]"
            }
        }
    },
    "networkProfile": {
        "networkInterfaces": [
            {

```

```

        "id": "[resourceId('Microsoft.Network/network
        Interfaces',variables('nicName'))]"
    }
  ]
}
},
{
  "type": "Microsoft.Compute/virtualMachines/extensions",
  "name": "[concat(variables('vmName'),'/',
  variables('extensionName'))]",
  "apiVersion": "2017-03-30",
  "location": "[parameters('location')]",
  "dependsOn": [
    "[concat('Microsoft.Compute/virtualMachines/',
    variables('vmName'))]"
  ],
  "properties": {
    "publisher": "Microsoft.Azure.Extensions",
    "type": "DockerExtension",
    "typeHandlerVersion": "1.0",
    "autoUpgradeMinorVersion": true,
    "settings": {
      "compose": {
        "rancher": {
          "image": "rancher/rancher:stable",
          "ports": [
            "80:80",
            "443:443"
          ],
          "volumes": [
            "/opt/rancher:/var/lib/rancher"
          ]
        }
      }
    }
  }
}
}

```

```
        }
      }
    }
  ]
}
```

This ARM Template can also be downloaded here: <https://github.com/Buchatech/DeployRanchertoAzure>.

Deploy the ARM Template using your deployment option of choice. You will need to provide data for the following parameters:

- Subscription
- Resource group
- Location
- Admin username
- Admin password
- Dns name for public IP
- Vm size
- Ubuntu OS version

After the Ubuntu VM is deployed, you should see the resources in the new resource group as shown in Table 6-1.

Table 6-1. *Rancher on Azure
Resources in Resource Group*

Name	Type
RancherVNet	Virtual network
RancherNode	Virtual machine
RancherNodePublicIP	Public IP address
RancherNodeNic	Network interface
RancherNode_OSDisk	Disk

In order to complete the Rancher deployment, you need to complete the setup via the Rancher portal. The URL is the DNS name of the Rancher Node VM we deployed. You can find the DNS name by clicking the Rancher Node VM in the Azure portal on the overview page. Here is an example of the URL:

<https://NAMEOFTHEVM.centralus.cloudapp.azure.com>

The Rancher portal will prompt you to set a password. This is shown in Figure 6-1.

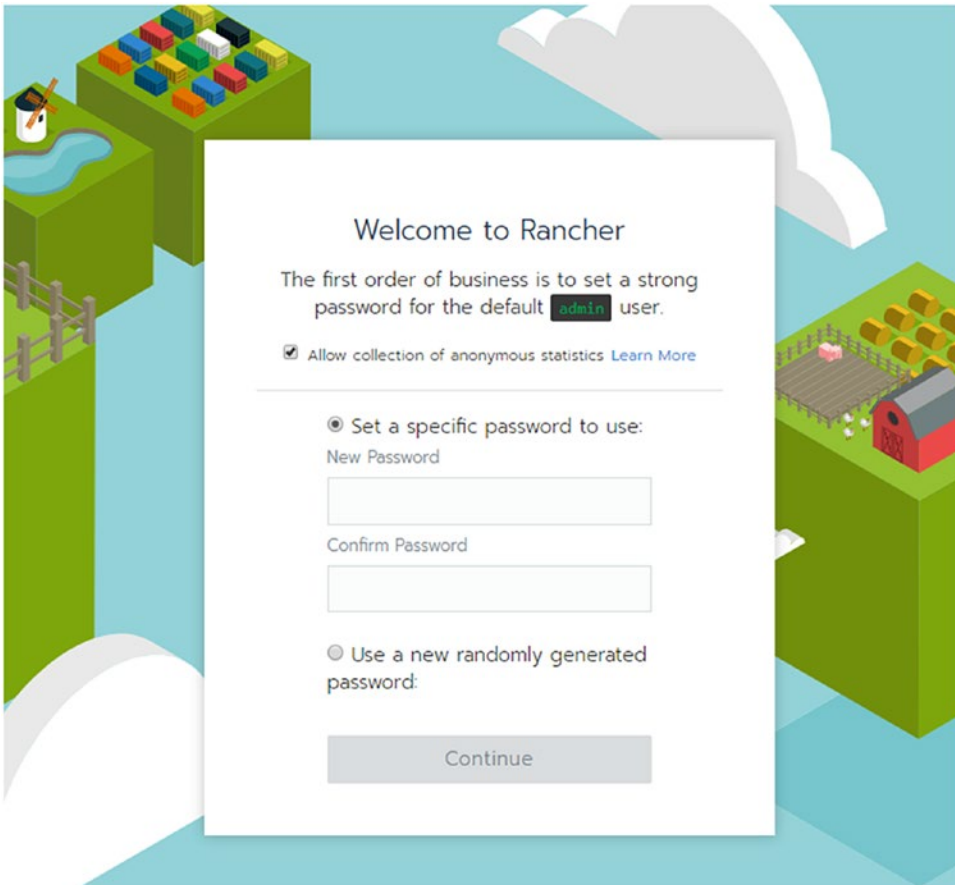


Figure 6-1. Rancher set password

After setting the password, the Rancher portal will prompt you for the correct Rancher Server URL. This will automatically be the Rancher Node VM DNS name as shown in Figure 6-2. Click Save URL.

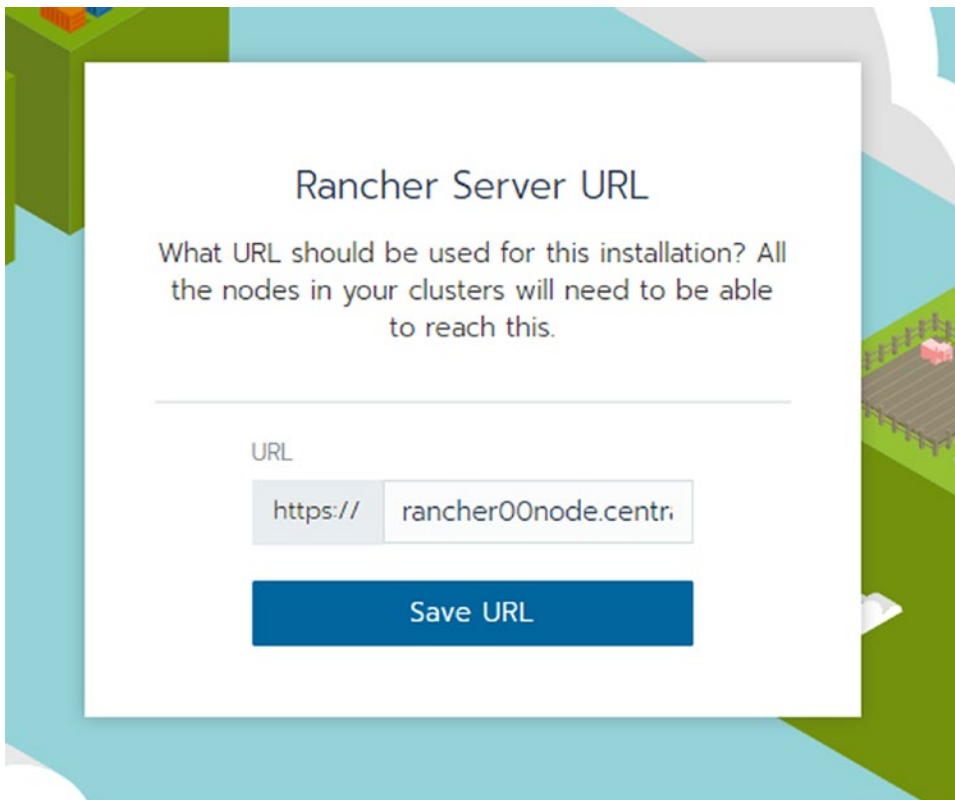


Figure 6-2. *Rancher Save URL*

You will then be logged into the Rancher portal directly on the You will see the clusters page. That wraps up the deployment of deploying Rancher on Azure. Next, we will explore deploying a new AKS cluster and connecting to an existing AKS cluster from within Rancher.

Authenticate Rancher with Azure Active Directory

You will need the authentication from Rancher to Azure working before you can deploy or manage existing AKS clusters. In order to authenticate with Azure from Rancher, you will need a service principal name (SPN) object in Azure Active Directory. This will be used for the authentication to Azure.

To create this SPN, you only need to run one line of syntax. It is recommended that you run this from Bash in Azure Cloud Shell.

Note Use the following steps to open Bash in Azure Cloud Shell:

Log into the Azure Portal.

Launch Cloud Shell from the top navigation by clicking ►.

If this is your first time running Cloud Shell, select a subscription to create a storage account and Microsoft Azure Files share.

When Cloud Shell launches at the bottom of the Azure portal, be sure the environment drop-down on the left-hand side of the Shell window shows Bash and not PowerShell.

Placeholder text. Remove later when able to clean up the note formatting.

Note You will need the subscription ID for the subscription that you want to create your AKS cluster in. You can run the following syntax in Cloud Shell to get a list of subscriptions for the account you are logged in with. This will list subscription property information including IDs. Copy the subscription ID for later use.

```
az account list
```

Use the following syntax to create an SPN with a specific name and assign the contributor role to a specified subscription:

```
az ad sp create-for-rbac --name NAMEOFTHESPNHERE --role contributor  
--scopes /subscriptions/SUBSCRIPTIONIDHERE
```

For example:

```
az ad sp create-for-rbac --name rancherSPN --role contributor --scopes /  
subscriptions/148727f76-9q1b-4941-coa6-92c5d153fe73
```

The output will be similar to:

Changing “rancherSPN” to a valid URI of “<http://rancherSPN>”, which is the required format used for service principal names

Retrying role assignment creation: 1/36

```
{
  "appId": "012d8611-c9a3-4e90-80d9-ad6504c823g8",
  "displayName": " rancherSPN ",
  "name": "http:// rancherSPN",
  "password": "6a4b83fc-31qa-40f0-c4c6-rba8c5av460b",
  "tenant": "0pw0cc24-q010-4f7b-h08e-9o57a72t531d"
}
```

Remember to copy this information somewhere as you will need it to connect to Azure from Rancher when creating your AKS cluster.

Note appId is what you will use for the clientId field in Rancher.

That’s it! That was all you had to do to create the SPN and get the information you need to authenticate to Azure from Rancher. In the next section, we are going to create a new AKS cluster from Rancher.

Deploy AKS with Rancher

At this point, we have Rancher deployed on Azure. Now let’s look at the process for deploying a new AKS cluster using Rancher. Use the following steps to deploy a new AKS from Rancher in Azure.

Note In the following steps, we will call out the required settings such as DNS prefix but not the optional settings such as Azure tags or advanced networking. You can configure optional settings as needed when you deploy your AKS cluster.

Within the Rancher portal, click Clusters in the top navigation menu.

Click Add Cluster and select Azure AKS as shown in Figure 6-3.

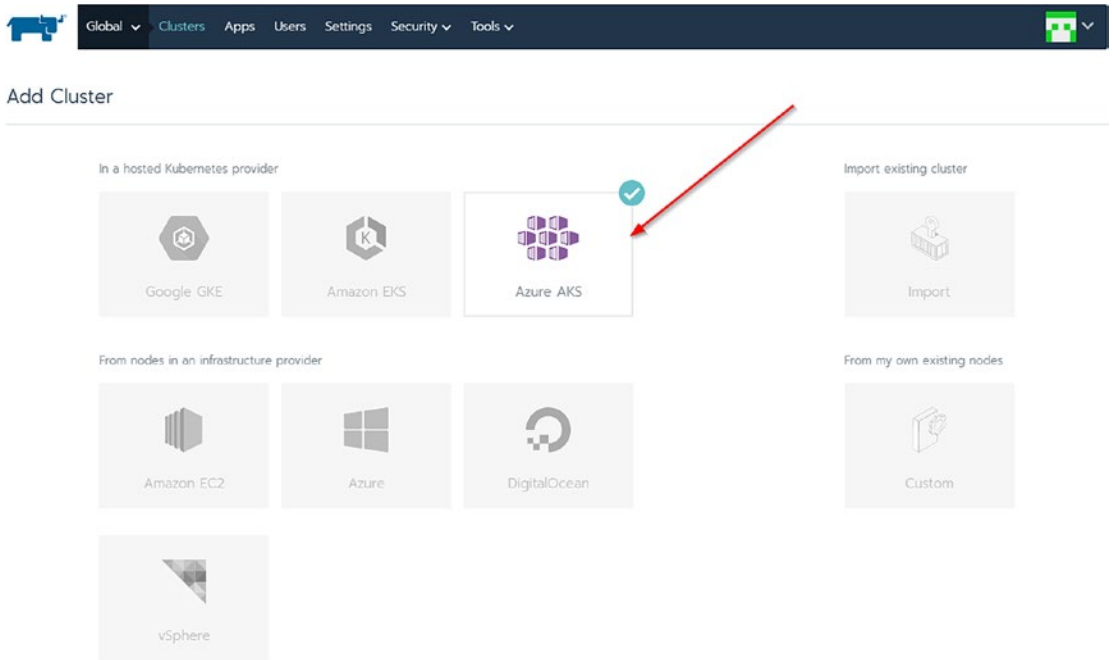


Figure 6-3. Azure AKS hosted Kubernetes provider

You will see the Account Access settings as shown in Figure 6-4.

Figure 6-4. Account Access settings

Input a cluster name as shown in Figure 6-5.

Input the information you copied from the last section after creating the SPN into the Account Access settings.

Click Next: Authentication & configure nodes.

1

Note: Currently Azure AKS will not create an ingress controller when launching a new cluster. If you need this functionality you will have to create an ingress controller manually after cluster creation.

Cluster Name *

RancherAKS

Add a Description

Member Roles

Control who has access to the cluster and what permission they have to change it.

Account Access

Configure the Azure Active Directory (AD) Service Principal that will be used to talk to Azure

Subscription ID *

Tenant ID *

Client ID *

Client Secret *

Location *

East US

You can find instructions on how to create an Azure AD Service Principal here.

Next: Authenticate & configure nodes

Cancel

Figure 6-5. Rancher cluster name

Next, under Cluster Options, give your cluster a DNS prefix as shown in Figure 6-6.

Cluster Options

Customize the cluster that will be created

Kubernetes Version

1.13.7

DNS Prefix

RancherAKS

HTTP Application Routing

Enabled

Disabled

Monitoring

Enabled

Disabled

Tags

+ Add Tag

Figure 6-6. Cluster DNS prefix

Under Nodes, input a name for your resource group in the Cluster Resource Group field as shown in Figure 6-7.

94

Nodes
Customize the nodes that will be created

Admin Username:

Cluster Resource Group:

Node Count:

OS Disk Size: GB

VM Size:

Networking: ☒ Default ☐ Advanced

SSH Public Key:

Figure 6-7. Cluster resource group and SSH key

Also, under Nodes, input an SSH public key and click Create.

Note PuTTY Key Generator is a free utility that can be used to generate a new SSH key. You can download it here: www.puttygen.com.

The AKS cluster will begin to provision as shown in Figure 6-8.

State	Cluster Name	Provider	Nodes	CPU	RAM
Active	[Cluster Name]	Azure AKS v1.13.7	3	2.9/5.8 Cores 50%	3/13.4 GiB 22%
Active	[Cluster Name]	Azure AKS v1.13.7	3	0.9/5.8 Cores 16%	11/13.4 GiB 8%
Provisioning	rancheraks	Azure AKS	0	n/a	n/a

Figure 6-8. AKS cluster provisioning

After the AKS cluster is deployed, it will show as Active in the Rancher portal. The AKS cluster is now deployed, and you could go the Azure portal, navigate to the resource group it created, and see the resources Rancher deployed including the AKS cluster, Log Analytics workspace, and a Containers Insights solution as shown in Figure 6-9.

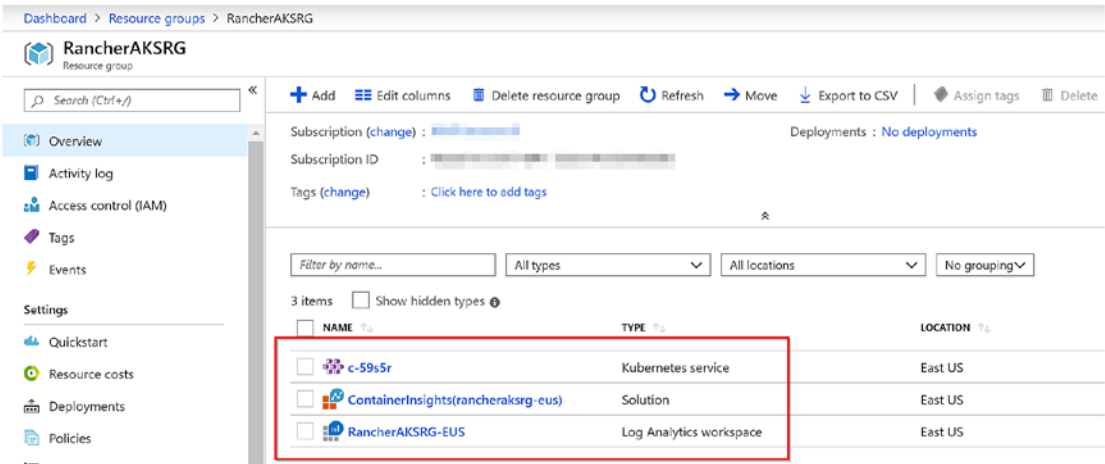


Figure 6-9. AKS cluster resources on Azure

Back in the Rancher portal, you can now click the AKS cluster to access dashboards, monitoring, and the cluster settings, install apps from the Rancher or Helm catalogs, launch Kubectl, and more. Figure 6-10 is an example of the dashboard for an AKS cluster in Rancher.

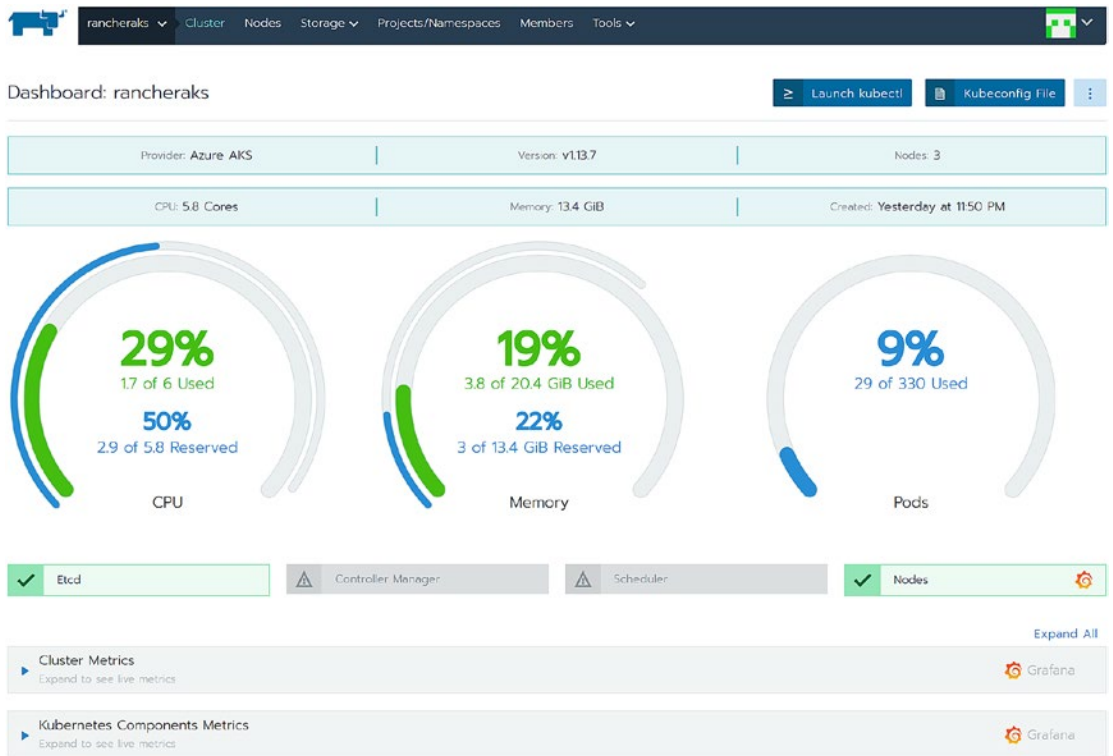


Figure 6-10. AKS cluster dashboard in Rancher

The following two screenshots demonstrate some of the Grafana monitoring that is available in Rancher for an AKS cluster. Figure 6-11 shows live cluster metrics.

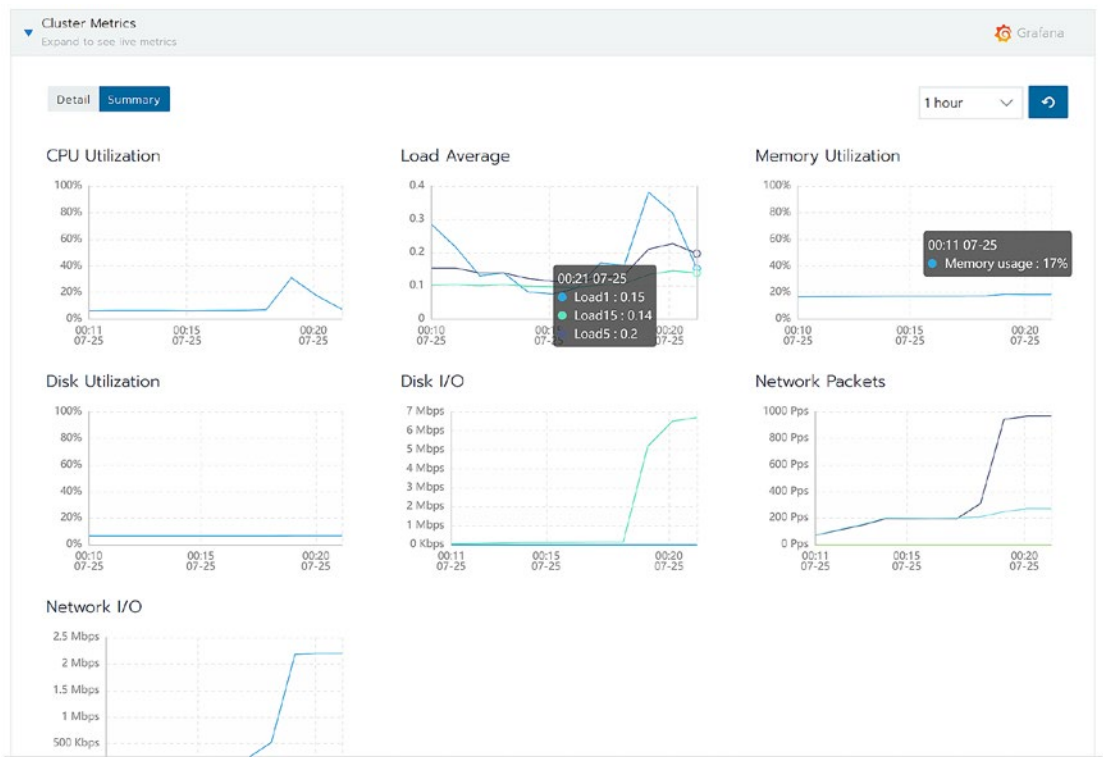


Figure 6-11. Grafana cluster metrics in Rancher

Figure 6-12 shows live metrics for the clusters and Kubernetes components.

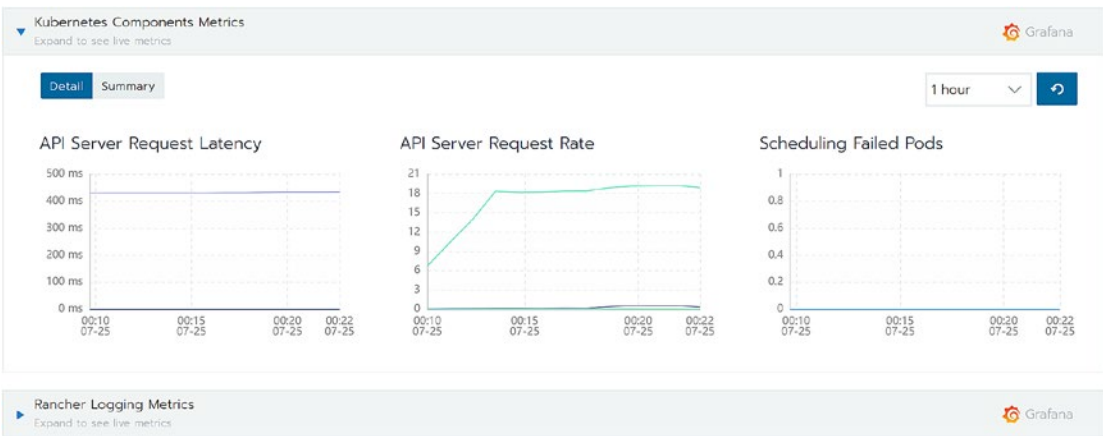


Figure 6-12. Grafana Kubernetes components metrics in Rancher

You also could access the full Grafana UI and system to get deeper insights into your AKS cluster and its resources such as nodes and pods as shown in Figure 6-13.

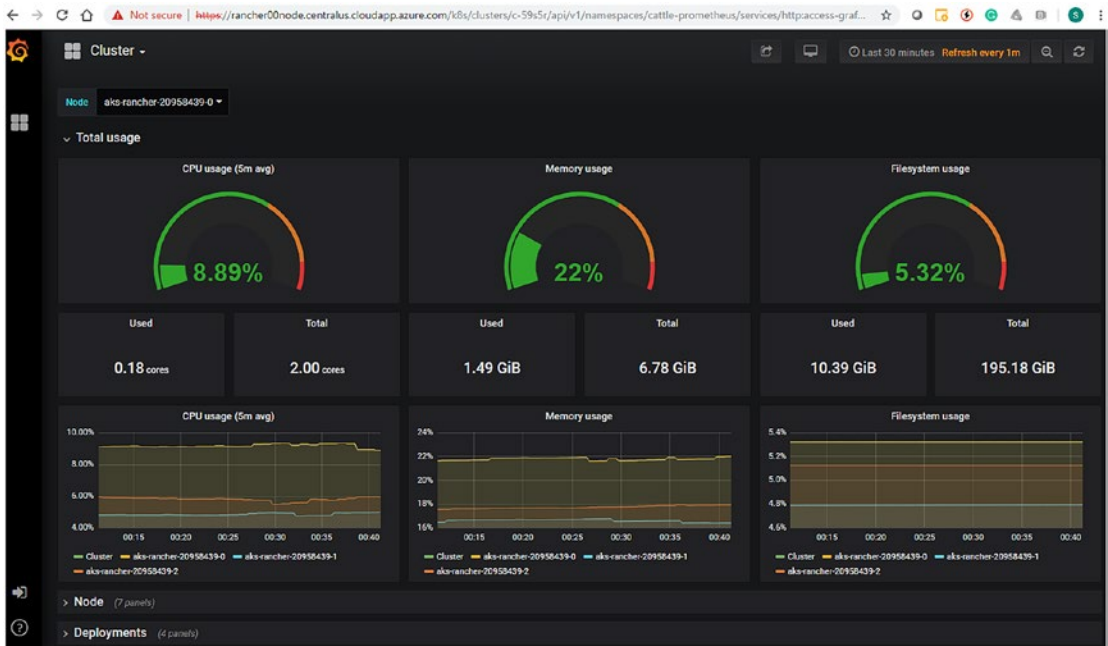


Figure 6-13. Grafana UI

Summary

That brings us to the end of this chapter. Rancher is not as well-known as Docker and Kubernetes are. However, as you learned in this chapter, Rancher is a useful solution when it comes to streamlining aspects of your Kubernetes life cycle. Specifically, in this chapter we covered what Rancher is, why you would use it with Kubernetes, deploying Rancher on Azure, and finally connecting Rancher to Azure so you can deploy a new AKS cluster.

CHAPTER 7

Operating Azure Kubernetes Service

Once you have deployed your first AKS cluster, it is important to understand how you can configure, monitor, and manage the AKS environment. The role of an AKS operator is critical when deploying applications to AKS clusters as cluster optimization is a key operational process in AKS.

In this chapter, we will explore how to operate an AKS cluster from a cluster operator perspective. We will review the processes for handling cluster operations, how to configure data storage for AKS, and how networking, security, and authentication work in AKS. Then, we will dive into the AKS monitoring realm with Azure Monitor for containers. Finally, we will go through the processes and best practices for business continuity and disaster recovery (DR) planning for Azure Kubernetes Service. By the end of this chapter, you will have a thorough understanding of operating an AKS cluster from a configuration, monitoring, and management perspective of an AKS operator.

Cluster Operations in Azure Kubernetes Service

When operating an Azure Kubernetes Services cluster, it is important to get familiarized with cluster common operations. The following sections provide an overview of some of the common cluster operations that you will encounter while working with AKS.

Note The explanations in the following sections are executed within an Azure Cloud Shell. Make sure that you are running Azure CLI version 2.0.65 or later by executing `az --version` if you are using a local installation.

Manually Scaling AKS Cluster Nodes

Resource needs of applications do change time to time. In such scenarios, you can either manually or automatically scale your AKS cluster to increase or decrease the node count. In a scale-down operation, your AKS nodes are carefully cordoned and drained in order to minimize application disruption. In a scale-up operation, until the worker nodes are marked as Ready, AKS waits before pods are scheduled on them.

The following example first obtains the node pool name for the jcbaksclu01 cluster in the jcbakserg01 resource group.

Listing 7-1. az aks show command

```
az aks show --resource-group jcbakserg01 --name jcbaksclu01 --query
agentPoolProfiles
```

You can see the name is nodepool1 in the following output.

Listing 7-2. az aks show command output

```
{
  "count": 1,
  "maxPods": 110,
  "name": "nodepool1",
  "osDiskSizeGb": 30,
  "osType": "Linux",
  "storageProfile": "ManagedDisks",
  "vmSize": "Standard_DS2_v2"
}
```

Then, you can use the `az aks scale` command to scale the cluster nodes. The following example scales the node count of jcbaksclu01 from 1 to 3.

Listing 7-3. az aks scale command

```
az aks scale --resource-group jcbakserg01 --name jcbaksclu01 --node-count 3
--nodepool-name nodepool1
```

You should see the following similar output which shows that the cluster has successfully scaled up to three nodes, as shown in the `agentPoolProfiles` section:

```
{
  "aadProfile": null,
  "addonProfiles": null,
  "agentPoolProfiles": [
    {
      "count": 3,
      "maxPods": 110,
      "name": "nodepool1",
      "osDiskSizeGb": 30,
      "osType": "Linux",
      "storageProfile": "ManagedDisks",
      "vmSize": "Standard_DS2_v2",
      "vnetSubnetId": null
    }
  ],
  [...]
}
```

Upgrading an AKS Cluster

During the life cycle of an AKS cluster, you will need to upgrade it to the latest or a specific Kubernetes version. The following example explains how you can upgrade the master components of a single, default node pool in an AKS cluster.

First, check whether there are any new Kubernetes releases available for your cluster by executing the `az aks get-upgrades` command against your cluster as the following.

Listing 7-4. `az aks get-upgrades` command

```
az aks get-upgrades --resource-group jcbaksrcg01 --name jcbakscld01 --output
table
```

If there are any upgrades available, you should see an output similar to the following. In this example, your cluster can be upgraded to Kubernetes versions 1.14.5 and 1.14.6.

Listing 7-5. az aks get-upgrades command output

Name	ResourceGroup	MasterVersion	NodePoolVersion	Upgrades
-----	-----	-----	-----	-----
default	jcbaksrg01	1.13.10	1.13.10	1.14.5, 1.14.6

If there are no upgrades available, you should see the following error message as the output.

Listing 7-6. No upgrades available error

ERROR: Table output unavailable. Use the --query option to specify an appropriate query. Use --debug for more info.

Note You cannot skip Kubernetes minor versions when upgrading an AKS cluster. For instance, upgrades from 1.12.x to 1.13.x or 1.13.x to 1.14.x are allowed, but upgrade from 1.12.x to 1.14.x is not. To upgrade, from 1.12.x to 1.14.x, first upgrade from 1.12.x to 1.13.x and then upgrade from 1.13.x to 1.14.x.

Now we can upgrade our AKS cluster to Kubernetes version 1.14.5 using az aks upgrade command.

Listing 7-7. az aks upgrade command

```
az aks upgrade --resource-group jcbaksrg01 --name jcbaksclu01 --kubernetes-version 1.14.5
```

Note Depending on the number of nodes you have, it can take some time to upgrade your AKS cluster. The time taken for an upgrade operation can be calculated by **10 minutes x total number of nodes in the cluster**. In this example, the upgrade operation must succeed within 30 minutes, or AKS will fail the operation in order to avoid an unrecoverable cluster state. If you encounter any upgrade failure, retry the cluster upgrade operation after this time-out has been reached.

You can confirm the cluster upgrade was successful by running the following command.

Listing 7-8. Verify AKS upgrade operation

```
az aks show --resource-group jcbakserg01 --name jcbakscu01 --output table
```

You should see an output that confirms the cluster version as 1.14.5

Listing 7-9. Verify AKS upgrade operation output

Name	Location			
ResourceGroup	KubernetesVersion	ProvisioningState	Fqdn	
-----	-----	-----	-----	----
jcbakscu01	australiasoutheast	jcbakserg01	1.14.5	
Succeeded	jcbakscu01-dns-6bede950.hcp.australiasoutheast.			
azmk8s.io				

Deleting an AKS Cluster

Even though you can delete an AKS cluster using a single line of code as shown in Listing 7-10, make sure that you have made backups for your configuration and data before proceeding with this operation.

Listing 7-10. az aks delete command to delete an AKS cluster

```
az aks delete --name jcbakscu01 --resource-group jcbakserg01
```

Creating Virtual Nodes

You can use virtual nodes to rapidly scale application workloads in AKS. The advantage of using virtual nodes is that you can quickly provision pods and only pay per second of their execution time. If you are using the cluster autoscaler (preview feature), you need to wait until the node deployment is completed before running additional pods. Currently, virtual nodes are only supported with Linux nodes and pods.

Virtual nodes are supported in the following Azure regions as of now:

- Australia East (australiaeast)
- Central US (centralus)
- East US (eastus)
- East US 2 (eastus2)
- Japan East (japaneast)
- North Europe (northeurope)
- Southeast Asia (southeastasia)
- West Central US (westcentralus)
- West Europe (westeurope)
- West US (westus)
- West US 2 (westus2)

Bear in mind that the virtual nodes are dependent on the features available in Azure Container Instances (ACI), and therefore the following scenarios are not yet supported with them:

- Using service principal to pull ACR images. You can use Kubernetes secrets as a work-around.
- Virtual network limitations include VNet peering, Kubernetes network policies, and outbound traffic to the Internet with network security groups.
- Init containers.
- Host aliases.
- Arguments for exec in ACI.
- Daemonsets will not deploy pods to the virtual node.
- Windows Server nodes (currently in preview in AKS) are not supported alongside virtual nodes. However, you can use virtual nodes to schedule Windows Server containers without the need for Windows Server nodes in an AKS cluster.

Note Complete step-by-step instructions to create and configure an AKS cluster to use virtual nodes can be found from the following URLs:

Using Azure CLI (<https://docs.microsoft.com/en-au/azure/aks/virtual-nodes-cli?view=azure-cli-latest>)

Using Azure Portal (<https://docs.microsoft.com/en-au/azure/aks/virtual-nodes-portal?view=azure-cli-latest>)

Using Virtual Kubelet with Azure Kubernetes Service

When using Azure Container Instances (ACI) you don't have to manage the underlying compute infrastructure as Azure does this for you. Containers running in ACIs are charged by the second for each running container. You can use the Virtual Kubelet provider for ACI, with both Linux and Windows containers, and it can be scheduled on a container instance as if it were deployed in a regular Kubernetes node.

The following diagram illustrates how Virtual Kubelet works. Essentially, the Virtual Kubelet registers itself as a node in a Kubernetes cluster. This allows developers to allow own APIs to interact with pods and containers by masquerading as a regular kubelet by connecting Kubernetes to other APIs.

Note AKS now provides native support for scheduling containers on ACI using virtual nodes which only supports Linux containers as of now. Hence, it is recommended to use Virtual Kubelet only when you need to schedule Windows container instances.

For step-by-step instructions on using Virtual Kubelet with AKS, refer to the following URL: <https://docs.microsoft.com/en-au/azure/aks/virtual-kubelet?view=azure-cli-latest>.

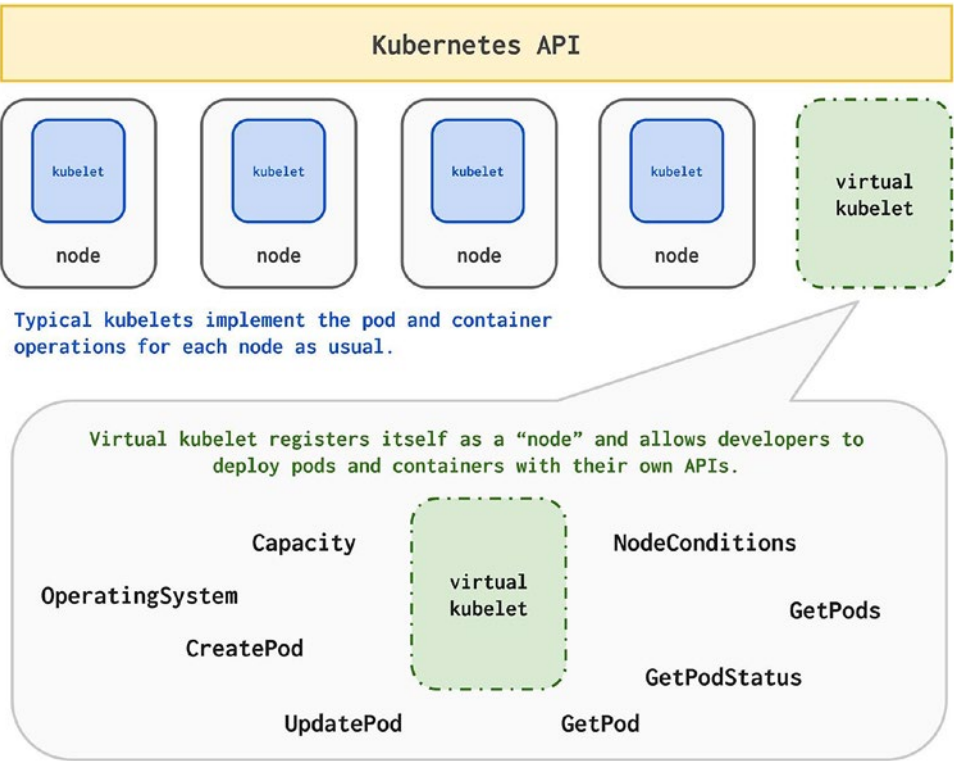


Figure 7-1. *Virtual Kubelet architecture*

Using Kubernetes Dashboard

The default dashboard experience for Kubernetes includes a web dashboard that you can use for basic management tasks. This dashboard allows you to view and monitor basic health status and metrics for you applications, create and deploy container service, as well as modify existing applications. The dashboard is running under **kube-system** namespace.

The following command starts the Kubernetes dashboard for jcbaksclus01 cluster in the jcbaskrg01 resource group.

Note For an RBAC-enabled AKS cluster, make sure that a ClusterRoleBinding is created before you start the Kubernetes dashboard. The Kubernetes dashboard is deployed with minimal reader access by default and can display RBAC access errors. The following code snippets illustrate using the `kubectl create clusterrolebinding` command to create the binding for our example:

```
kubectl create clusterrolebinding kubernetes-dashboard
--clusterrole=cluster-admin --serviceaccount=kube-
system:kubernetes-dashboard.
```

Listing 7-11. Starting the Kubernetes dashboard with `az aks browse` command

```
az aks browse --resource-group jcbaksrcg01 --name jcbakscld01
```

You should be automatically redirected to a new tab in your web browser after executing the preceding command in your Azure Cloud Shell unless pop-up windows are blocked in your browser. If not, copy and paste the URL address displayed in the Azure CLI in your web browser as the following.



```
Bash
janaka@Azure:~$ az aks browse --resource-group jcbaksrcg01 --name jcbakscld01
Merged "jcbakscld01" as current context in /tmp/tmpz2638fu2
To view the console, please open https://gateway14.eastus.console.azure.com/n/cc-6cc79f76/cc-6cc79f76/proxy/8001/ in a new tab
Press CTRL+C to close the tunnel...
```

Figure 7-2. Open Kubernetes dashboard in Azure CLI

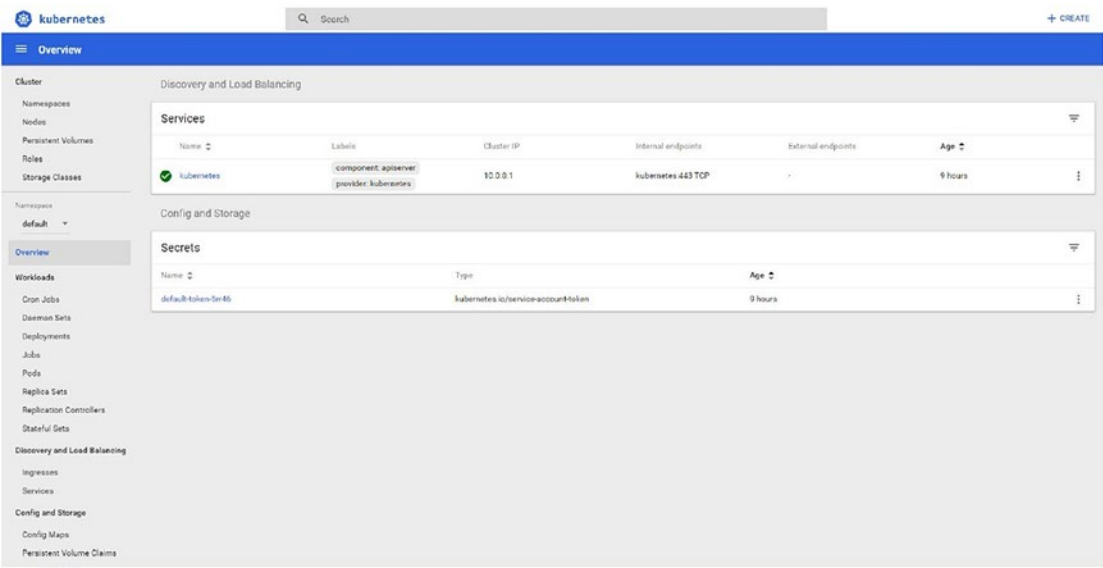


Figure 7-3. *Kubernetes dashboard*

Scaling Azure Kubernetes Service

When you deploy n of applications in Azure Kubernetes Service, it will be required to increase or decrease allocated compute resources based on the demand. This will require the underlying Kubernetes nodes to change accordingly. There will be instances where it is required to quickly provision many additional application instances.

This section explores the core scaling concepts in AKS that will help you to achieve the preceding goals.

Manually Scaling Pods or Nodes

In order to test how your application responds to the resource availability changes in your AKS cluster, you can manually scale pods (replicas) and nodes. By manually scaling these resources, you can define a set number of resources in your AKS cluster. In order to manually scale, first you define the pod or node count, and then the Kubernetes API schedules the creation of additional pods or node draining depending on the pod or node count.

A complete tutorial on scaling pods in an AKS cluster can be found from the following URL: <https://docs.microsoft.com/en-au/azure/aks/tutorial-kubernetes-scale#manually-scale-pods>. In section “Manually Scaling AKS Cluster Nodes,” we have discussed the steps involved in manually scaling your AKS cluster nodes.

Automatically Scaling Pods or Nodes

AKS cluster need a way to automatically scale pods or nodes in order to adjust to the varying application demands, depending on the traffic received by an application. AKS clusters can scale in one of two ways:

- The **horizontal pod autoscaler** (HPA): This leverages the Metrics Server in a Kubernetes cluster to monitor the resource demand of pods. In case if an application requests for more resources, the number of replicas is automatically increased to meet that demand.
- The **cluster autoscaler**: Monitors for pods that cannot be scheduled on nodes due to resource limitations. Then, the cluster can automatically increase the number of nodes.

Horizontal Pod Autoscaler

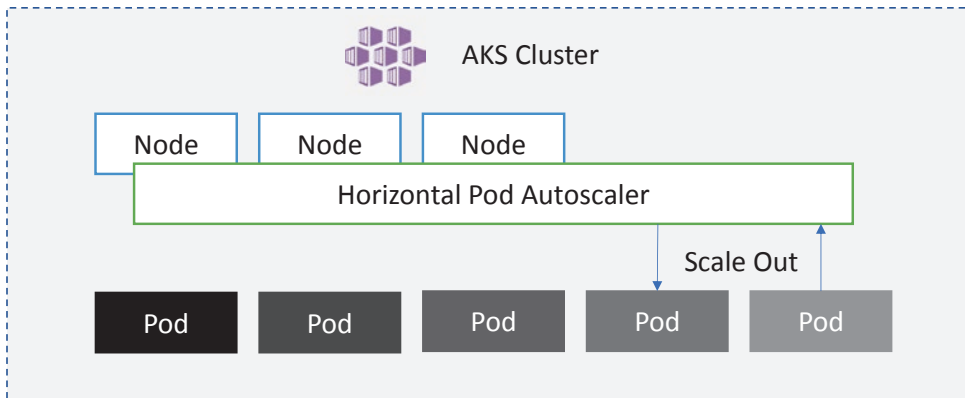


Figure 7-4. Horizontal pod autoscaler architecture

In Kubernetes, the horizontal pod autoscaler (HPA) is used to monitor the resource demand and automatically scale the number of pods. The HPA checks the Metrics API every 30 seconds for any required changes in pod count by default. When and if changes

are required at any point, the number of pods is increased or decreased, respectively. In AKS, HPA is supported with AKS clusters with Metrics Server for Kubernetes 1.8+ deployed.

If you are configuring the horizontal pod autoscaler for an AKS cluster, you will have to define the minimum and maximum number of pods that the cluster can run. In addition to that, you can also declare a metric to monitor and on which to base any scaling decisions, that is, CPU usage.

A complete tutorial on setting up a horizontal autoscaler in an AKS cluster can be found from the following URL: <https://docs.microsoft.com/en-au/azure/aks/tutorial-kubernetes-scale#autoscale-pods>.

Note Previous scale events in an AKS cluster may not have been successfully completed between Metrics API checks that happen every 30 seconds. This phenomenon could potentially cause the HPA to change the number of pods before the previous scale event can grasp the application workload and adjust to the resource demands accordingly. To minimize such **race events**, cooldown or delay values are set in an AKS cluster. These values depict how long the HPA must wait after a scale event before another scale event can be triggered. By doing so, it will allow the new pod count to take effect and the Metrics API to reflect the newly distributed workload. The default delay value on scale-up events is 3 minutes, whereas it is 5 minutes on scale-down events. These cooldown values cannot be set by the user as of now.

Cluster Autoscaler (Preview)

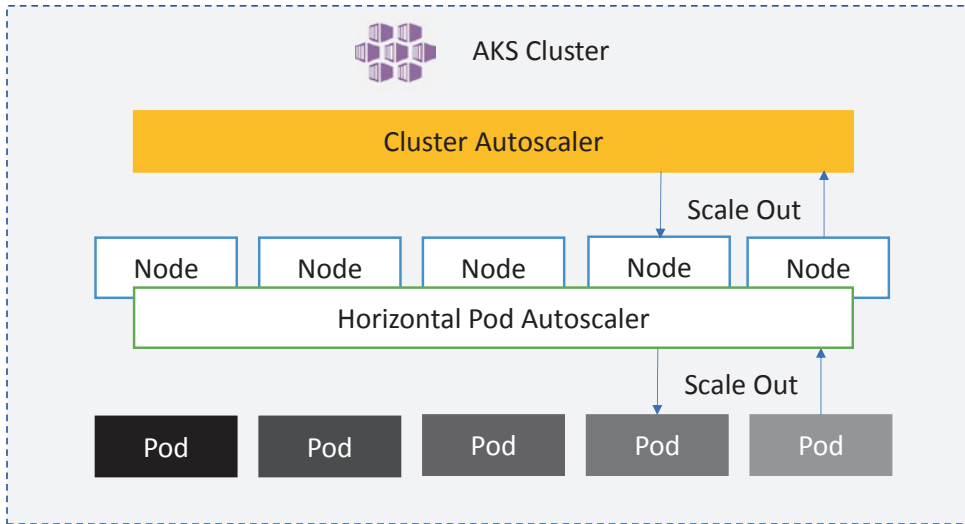


Figure 7-5. Cluster autoscaler architecture

Cluster autoscaler (in preview) can adjust the number of nodes depending on the requested compute resources in the node pool to rapidly respond to varying pod demands. The cluster autoscaler can increase or decrease the number of nodes in your AKS cluster accordingly, if it decides a resource variation is required. The cluster autoscaler feature is supported in RBAC-enabled AKS clusters that run Kubernetes 1.10.x or higher.

Usually the cluster autoscaler is used together with the horizontal pod autoscaler. The HPA increases or decreases the number of pods based on application demand, while the cluster autoscaler adjusts the number of nodes required to run those additional pods.

Note Cluster autoscaler is a preview feature in AKS.

A complete tutorial on getting started with cluster autoscaler can be found from the following URL: <https://docs.microsoft.com/en-au/azure/aks/cluster-autoscaler>.

Burst On Demand with Azure Container Instances

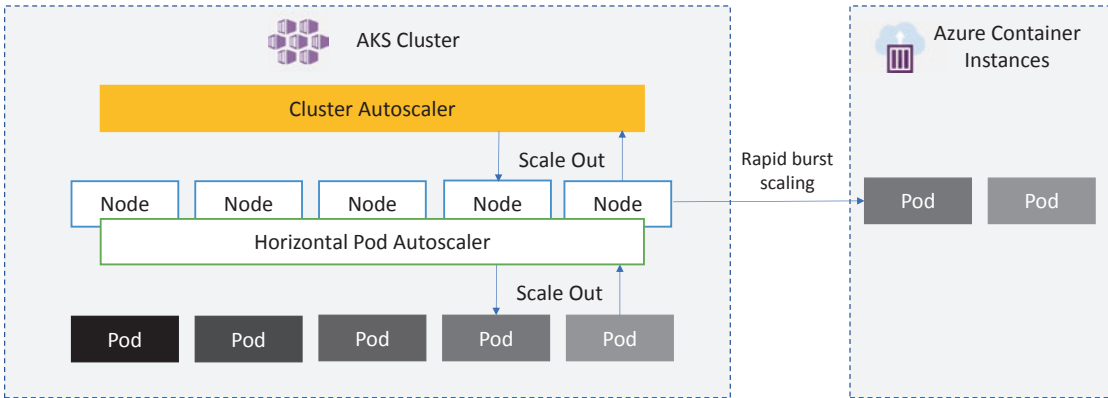


Figure 7-6. *Bursting with Azure Container Instances*

Integrating with Azure Container Instances (ACI) allows you to rapidly scale your AKS clusters. The built-in components in Kubernetes can scale the pod and node count. But if your application demands rapid scaling, the HPA may schedule more pods than what can be provided using the existing compute resources in the node pool. This phenomenon can trigger the cluster autoscaler to deploy additional nodes in the node pool; however, it can take some time for those additional nodes to be provisioned and allow the Kubernetes scheduler to run pods on them.

ACI connected to AKS becomes a secured and logical extension of your AKS cluster. Currently there are two ways to enable ACI on AKS:

- **The Virtual Kubelet:** When installed in your AKS cluster, this component can present ACI as a virtual Kubernetes node. It supports both Linux and Windows nodes.
- **Virtual nodes:** Currently in preview, these are deployed to an additional subnet in the same VNet as your AKS cluster. This VNet allows the traffic between ACI and AKS to be secured. Supports Linux nodes only as of now.

Storage Options for Azure Kubernetes Service

Applications deployed to an AKS cluster require storage to store and retrieve their data. For some workloads, these storages can be local, fast storage on the node and can be released when pods are deleted, while some other workloads may need persistent data storage hosted in Azure. Multiple pods may require sharing the same data volumes and/or reattaching the data volumes if the pods are rescheduled on a different node. It is also may be required to present sensitive data or application configuration into the pods. Figure 7-7 depicts the storage architecture for AKS clusters.

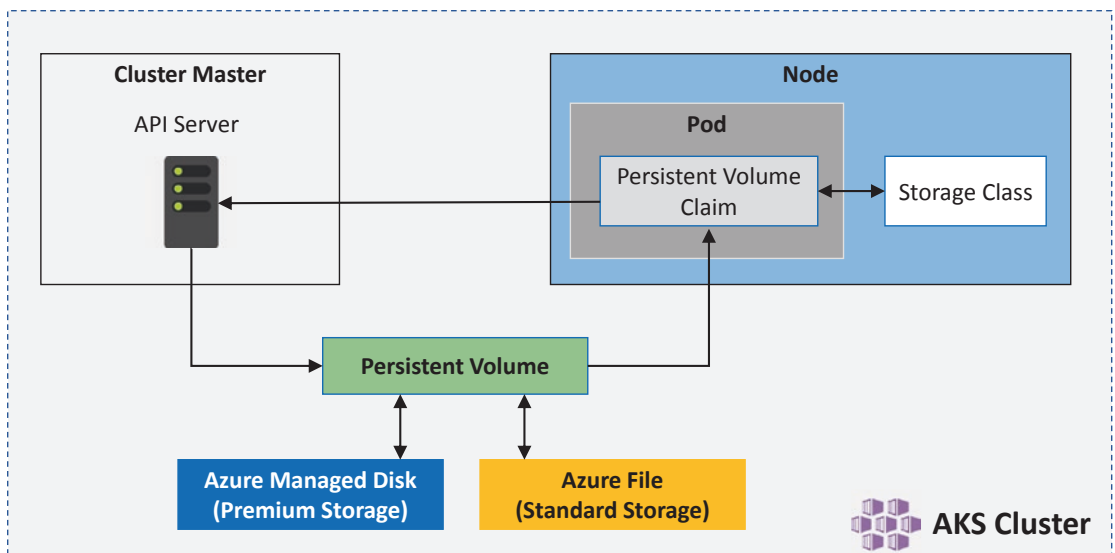


Figure 7-7. Storage architecture in AKS

In this section, let's explore the core concepts in AKS that explains how storage is provided to your application workloads.

Volumes

A *volume* represents a method to store, retrieve, and persist data across pods and through the application life cycle. Usually the volumes required to store and retrieve data on AKS is based on Azure storage. These data volumes can either be manually created and then assigned to pods directly or AKS can automatically create and allocate them when needed. These data volumes can either use

- **Azure Disks:** These can be used to create a Kubernetes DataDisk resource. Azure Disks can use Azure Premium storage or Azure Standard storage. For production and development workloads, it is recommended to use Premium storage. These are mounted as ReadWriteOnce and hence only available to a single node.
- **Azure Files** which are used to mount an SMB 3.0 share backed by an Azure storage account to pods. With Azure Files, you share data across multiple nodes and pods. Both Azure Premium storage and Azure Standard storage are supported with Azure Files.

Kubernetes volumes can also be leveraged to inject data into a pod for use by the containers. Additional volume types in Kubernetes include

- **emptyDir:** Used as temporary space for a pod
- **secret:** Used to inject sensitive data into pods, such as passwords
- **configMap:** Used to inject key-value pair properties into pods, such as application configuration information

Persistent Volumes

A persistent volume (PV) is created and managed by the Kubernetes API. It can exist beyond the lifetime of an individual pod, whereas traditional volumes created as part of a pod life cycle only exist until the pod is deleted. You can use either Azure Disks or Files to provide a PV.

A persistent volume can be either manually created by a cluster admin or dynamically generated by the Kubernetes API server. In case a scheduled requests storage that is not currently available, Kubernetes will create the underlying Azure Disk or Files storage and attach it to the pod. This scenario is called **Dynamic provisioning** and it uses a **StorageClass** to determine what type of Azure storage is required.

Storage Classes

In order to classify different storage tiers, you can create a **StorageClass**. The StorageClass also defines the **reclaimPolicy**. The reclaimPolicy controls the behavior of the Azure storage resource when the pod is deleted, and the persistent volume may no longer be required. When a pod is deleted, the storage resource can either be deleted or retained for use with a future pod.

There are two initial StorageClasses that can be created in AKS:

- **default:** Which utilizes Azure Standard storage to create a Managed Disk. The reclaim policy states that when the corresponding pod is deleted, the Azure Disk will also be deleted.
- **managed-premium:** Which utilizes Azure Premium storage to create a Managed Disk. The reclaim policy states that when the corresponding pod is deleted, the Azure Disk will also be deleted.

When no StorageClass is specified while creating a PV, the default StorageClass is used.

In the following YAML manifest, it is stated that Premium Managed Disks are to be used and the Azure Disk must be retained when the pod is deleted:

Listing 7-12. Defining a storage class in YAML

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: managed-premium-retain
provisioner: kubernetes.io/azure-disk
reclaimPolicy: Retain
parameters:
  storageaccounttype: Premium_LRS
  kind: Managed
```

Persistent Volume Claims

If you want to create either disk or file storage of a defined StorageClass, access mode, and size, define a PersistentVolumeClaim. If there are no existing resources to service the claim based on its StorageClass, the Kubernetes API server can dynamically provision

the underlying storage resource. Once the volume has been connected to the pod, the pod definition will include the volume mount as well.

A PV is bound to a PersistentVolumeClaim after an available storage resource has been assigned to the pod that requests it. The mapping of persistent volumes to claims is 1:1

The following is a sample YAML manifest which denotes a PV claim with managed-premium StorageClass and a disk 5 Gi in size.

Listing 7-13. Defining a PersistentVolumeClaim in YAML

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: azure-managed-disk
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: managed-premium
  resources:
    requests:
      storage: 5Gi
```

A PV claim is specified to request the desired storage when a pod definition is created. Here you can also specify the **volumeMount** for your applications to read and write data. The following YAML manifest illustrates how the previous PV claim can be used to mount a volume at /mnt/azure.

Listing 7-14. Defining a volumeMount in a PersistentVolumeClaim in YAML

```
kind: Pod
apiVersion: v1
metadata:
  name: nginx
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
```

```

- mountPath: "/mnt/azure"
  name: volume
volumes:
- name: volume
  persistentVolumeClaim:
    claimName: azure-managed-disk

```

We have briefly discussed the storage options available for AKS workloads. Next step is to create dynamic and static volumes for AKS. The following articles from the Microsoft documentation provide a holistic overview on how to do so:

- Create a static volume using Azure Disks (<https://docs.microsoft.com/en-au/azure/aks/azure-disk-volume>).
- Create a static volume using Azure Files (<https://docs.microsoft.com/en-au/azure/aks/azure-files-volume>).
- Create a dynamic volume using Azure Disks (<https://docs.microsoft.com/en-au/azure/aks/azure-disks-dynamic-pv>).
- Create a dynamic volume using Azure Files (<https://docs.microsoft.com/en-au/azure/aks/azure-files-dynamic-pv>).

Networking in Azure Kubernetes Service

Application components in a microservices approach must work together to process their desired tasks. This application communication can be achieved using a few components provided by Kubernetes. For an example, the applications can be either exposed internally or externally, can be load balanced for high availability, and have SSL/TLS termination for ingress traffic as well as for routing of multiple components. Furthermore, developers may need you to restrict the flow of network traffic into or between pods and nodes due to security concerns.

In this section, we will dive into core networking concepts of AKS and some of the examples of providing secure network connectivity to your pods and nodes.

Kubenet vs. Azure Container Networking Interface (CNI)

An AKS cluster uses one of the following two networking models:

Kubenet (Basic) Networking

This is the default configuration option for an AKS cluster. In kubenet, the AKS nodes obtain an IP address from the Azure VNet subnet. Pods receive an IP address from a logically different address space to the Azure VNet subnet of the nodes. For the pods to reach resources on the Azure VNet, network address translation (NAT) is then configured. The source IP address of the traffic is NAT'd to the node's primary IP address.

Nodes use the kubenet Kubernetes plugin. You can either allow Azure Fabric to create and configure the VNets for you or deploy your AKS cluster into an existing subnet of a predefined VNet. Even though you are deploying to a predefined VNet, only the nodes will receive a routable IP address; pods use NAT to communicate with other resources external to the AKS cluster.

Azure Container Networking Interface (CNI) - Adadvanced Networking

Each pod gets an IP address from the subnet and can be accessed directly if you are using the Azure CNI model. But remember that these IP addresses must be unique across the VNet network space and must be planned in well advance. There is a configuration parameter for the maximum number of pods that each node supports. An equivalent number of IP addresses per node are then reserved for that node.

The following table lists the behavioral differences between kubenet and Azure CNI.

Table 7-1. *Behavioral Differences Between Kubenet and Azure CNI*

Capability	Kubenet	Azure CNI
Deploy cluster in existing or new virtual network	Supported – UDRs manually applied	Supported
Pod-pod connectivity	Supported	Supported
Pod-VM connectivity; VM in the same virtual network	Works when initiated by pod	Works both ways
Pod-VM connectivity; VM in peered virtual network	Works when initiated by pod	Works both ways
On-premises access using VPN or Express Route	Works when initiated by pod	Works both ways

(continued)

Table 7-1. *(continued)*

Capability	Kubenet	Azure CNI
Access to resources secured by service endpoints	Supported	Supported
Expose Kubernetes services using a load balancer service, App Gateway, or ingress controller	Supported	Supported
Default Azure DNS and Private Zones	Supported	Supported

Table 7-2 lists the advantages and disadvantages of kubenet and Azure CNI at a high level.

Table 7-2. *Advantages and Disadvantages of Kubenet vs. Azure CNI*

Model	Advantages	Disadvantages
Kubenet	<ul style="list-style-type: none"> Conserves IP address space. Uses Kubernetes internal or external load balancer to reach pods from outside of the cluster. 	<ul style="list-style-type: none"> You must manually manage and maintain user-defined routes (UDRs). Maximum of 400 nodes per cluster.
Azure CNI	Pods get full virtual network connectivity and can be directly reached from outside of the cluster.	Requires more IP address space.

Regardless of the network model you have selected, support policies for AKS depict the network tuning capabilities such as service endpoints and UDRs that you can make in your AKS clusters:

- If you manually create the virtual network resources for an AKS cluster, you are supported when configuring your own UDRs or service endpoints.
- If the Azure platform automatically creates the virtual network resources for your AKS cluster, it is not supported to manually change those AKS-managed resources to configure your own UDRs or service endpoints.

Note For a complete record of support policies for AKS, visit the following URL: <https://docs.microsoft.com/en-au/azure/aks/support-policies>.

Network Security Groups and Network Policies

It is not recommended to manually configure network security group rules to filter pod traffic in an AKS cluster. The Azure platform will create and update the appropriate rules as part of the AKS managed service. In order to automatically apply traffic filter rules to pods, you can utilize **Network Policies**. On the one hand, it is a feature available in AKS that allows you to control the traffic between pods. You can decide whether to allow or deny traffic based on settings such as assigned labels, namespace, or traffic port. Network security groups on the other hand are for the AKS nodes, not pods.

Note For step-by-step instructions on securing pod traffic using Azure Network Policies in AKS, visit the following URL: <https://docs.microsoft.com/en-au/azure/aks/use-network-policies>.

Access and Identity in Azure Kubernetes Service

In Azure, there are multiple methods to authenticate and secure AKS clusters. Role-based access controls (RBACs) allow granting users or groups access to only the resources they need. By integrating AKS with Azure Active Directory, you are able to further enhance the security and permissions structure. This section provides a high-level overview of the access and identity options available to you when operating an AKS cluster.

Kubernetes Service Accounts

A service account is a primary user type in Kubernetes, and it exists in and is managed by the Kubernetes API. The service account credentials are stored as Kubernetes secrets, which allows them to be used by authorized pods to communicate with the API server. API requests provide an authentication token for a service account or a regular user account. Regular user accounts are leveraged to provide traditional access to

administrators or developer who are using an AKS cluster, although Kubernetes API itself doesn't provide an identity management solution for such scenarios. By integrating AKS with Azure Active Directory, you can achieve this goal.

Azure Active Directory Integration

Azure Active Directory (AAD) is a multi-tenant, cloud-based directory, and identity management solution that provides core directory services, application access management, and identity protection. You can integrate on-premises identities into AKS clusters to provide unified account management and security processes by integrating AKS with AAD.

For an example, in AAD-integrated AKS clusters, you can grant users or groups access to Kubernetes resources within a namespace or across the cluster. To retrieve a `kubectl` configuration context, a user can execute the `az aks get-credentials` command. Afterwards, when a user interacts with the AKS cluster with `kubectl`, they will be prompted to sign in with their respective Azure AD credentials. This way, the users can only access the resources defined by the AKS cluster administrator.

Azure Role-Based Access Controls (RBACs)

Apart from RBACs provided by Kubernetes API, AKS cluster access can be managed by Azure role-based access controls (RBACs). The difference is that Kubernetes RBAC is designed to work on resources within your AKS cluster, and Azure RBAC is designed to work on resources within your Azure subscription. Azure RBAC enables you to create role definitions that outline the permissions to be applied for your AKS clusters. You can then assign a user or group for a role definition that includes a defined scope, which could be an individual resource, a resource group, or across the Azure subscription.

Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings

In Kubernetes RBAC, you first define permissions as a **Role**. Kubernetes roles grant permissions, and there is no concept of a deny permission. Roles are used to grant permissions within a namespace.

The purpose of a **ClusterRole** is like that of a role, but a ClusterRole can be applied to resources across the entire cluster, not a specific namespace.

When you have roles defined, you assign those Kubernetes RBAC permissions with a **RoleBinding**. In AAD-integrated AKS clusters, bindings are how Azure AD users are granted permissions to perform actions within the cluster. Role bindings are used to assign roles for a defined namespace where you can segregate access to individual clusters.

A **ClusterRoleBinding** on the other hand works in the same way as role bindings but can be applied to resources across the entire cluster, not a specific namespace. This approach is ideal in situations where you need to grant admins or support engineers access to all resources across the board in an AKS cluster.

Control Deployments with Azure Policy (Preview)

Azure Policy can be integrated with AKS so that you can apply policy enforcement to your AKS clusters in a centralized and consistent manner. Further complimented by using **GateKeeper**, which is an admission controller webhook for Open Policy Agent (OPA), Azure Policy allows you to centrally manage and report on the compliance state of your Azure resources and AKS clusters.

Follow the following steps to enable and apply this feature to your AKS clusters.

Enable the Preview

First, you must enable the `Microsoft.ContainerService` resource provider and the `Microsoft.PolicyInsights` resource provider and then be approved to join the preview. The following example illustrates how you can do so using Azure CLI in Azure Cloud Shell.

Listing 7-15. Join the AKS Policy preview via Azure CLI

```
# Provider register: Register the Azure Kubernetes Services provider
az provider register --namespace Microsoft.ContainerService

# Provider register: Register the Azure Policy provider
az provider register --namespace Microsoft.PolicyInsights

# Feature register: enables installing the add-on
az feature register --namespace Microsoft.ContainerService --name AKS-
AzurePolicyAutoApprove
```

```
# Feature register: enables the add-on to call the Azure Policy resource
provider
az feature register --namespace Microsoft.PolicyInsights --name AKS-
DataplaneAutoApprove
```

Azure Policy Add-On

This add-on, installed into the *azure-policy* namespace, connects the Azure Policy service to the GateKeeper admission controller. The following are the functionalities of this add-on:

- Checks with Azure Policy for assignments to the AKS cluster
- Downloads and caches policy details, including the rego policy definition, as configmaps
- Runs a full scan compliance check on the AKS cluster
- Reports auditing and compliance details back to Azure Policy

Installation Prerequisites

You need to install the preview extension before you install the add-on in your AKS cluster. Follow the following procedure to do so:

- Make sure that you are running Azure CLI version 2.0.62. Run `az --version` to find the version.
- The AKS cluster must be version 1.10 or higher. The following Azure CLI excerpt denotes how to check that.

Listing 7-16. Check AKS version

```
az aks list
```

- Install version 0.4.0 of the Azure CLI preview extension for AKS, `aks-preview`.

Listing 7-17. Install Azure CLI preview extension for AKS

```
# Install/update the preview extension
az extension add --name aks-preview

# Validate the version of the preview extension
az extension show --name aks-preview --query [version]
```

Note If the aks-preview extension has been deployed already, please uninstall any updates executing the `az extension update --name aks-preview` command.

Installing the Azure Policy Add-on

Once the preceding prerequisites are installed, you can proceed with installing the Azure Policy add-on. The following Azure CLI excerpt illustrates how to do so.

Listing 7-18. Install Azure Policy add-on

```
az aks enable-addons --addons azure-policy --name jcbaksclu01 --resource-
group jcbaksrg01
```

Assigning Policy Definitions to AKS

Currently Azure Policy for AKS is in limited preview and only supports built-in policy definitions. You can find the built-in policies for managing AKS using the Azure portal as follows:

- Click **All services** in the left pane and then search and select **Policy**.
- In the **Azure Policy** page, select **Definitions**.
- From the **Category** drop-down list, click **Select all** and then select **Kubernetes service**.
- Select the policy definition you want to apply, and then select the **Assign** button.

Note Make sure the **Scope** must include the AKS cluster resource, when assigning the Azure Policy for AKS definition.

Policy Validation

The Azure Policy add-on checks in with Azure Policy Service for changes in policy assignments every 5 minutes. All **configmaps** in the azure-policy namespace are removed and then recreated for GateKeeper during this refresh cycle by the add-on.

The add-on requests for a full scan of the cluster every 5 minutes. Once the details are gathered from the full scan along with any real-time evaluations by GateKeeper of attempted changes to the cluster, the results are reported back to the Azure Policy to include compliance details such as Azure Policy assignment. During the audit cycle, only results for active policy assignments are returned.

Note It's not recommended or supported to make changes to the namespace, although a cluster admin may have permission to the azure-policy namespace and any manual changes made are lost during the refresh cycle.

Azure Policy Add-On Logs

The Azure Policy add-on logs are kept as a Kubernetes controller/container in the AKS cluster. These logs are exposed in the Insights page of the AKS cluster.

GateKeeper Logs

You need to GateKeeper logs for new resource requests. Follow the procedure to enable and review Kubernetes master node logs in AKS in the following URL: <https://docs.microsoft.com/en-au/azure/aks/view-master-logs>.

Listing 7-19 is an example query to view denied events on new resource requests.

Listing 7-19. KQL query to view denied events on new resource requests

```
| where Category == "kube-audit"  
| where log_s contains "admission webhook"  
| limit 100
```

To view logs from GateKeeper containers, follow the steps in the preceding article and check the **kube-apiserver** option in the **Diagnostic settings** pane.

Security Concepts in Azure Kubernetes Service

Security of an AKS cluster is paramount like any other resource in your datacenter. Kubernetes security components such as network policies and secrets are complimented by Azure features such as network security groups and orchestrated AKS cluster upgrades.

Master Security

The Kubernetes master components are part of the AKS managed service provided by Azure. Each AKS cluster has its own single-tenant, dedicated Kubernetes master to provide the API server, scheduler, and so on in Azure. This master is managed and maintained by Azure. The default behavior for the Kubernetes API server to use a public IP address and a fully qualified domain name (FQDN). Access to the API server can be controlled by using Kubernetes RBACs and Azure Active Directory.

Node Security

AKS nodes are Azure VMs that are managed and maintained by yourself. Linux AKS nodes run on an optimized Ubuntu distribution with the Moby container runtime. Windows Server nodes (currently in preview in AKS) run an optimized Windows Server 2019 release with the Moby container runtime. During the creation or a scale-up operation in an AKS cluster, these nodes are automatically deployed with the latest OS security updates and configurations.

The following are some of the facts and considerations when planning for AKS node security:

- OS security patches are automatically applied by the Azure platform to Linux nodes on a nightly basis.
- If a Linux OS security update requires a host reboot, it is not automatically performed.

- You can either manually reboot the Linux nodes or use **Kured**, an open source reboot daemon for Kubernetes.
- Windows Update does not automatically run and apply the latest updates for Windows Server nodes.
- You should perform an upgrade on the Windows Server node pool(s) in your AKS cluster by yourself. This upgrade process creates nodes that run the latest Windows Server image and patches and then removes the older nodes.
- Nodes are by default deployed into a private virtual network subnet, with no public IP addresses assigned. SSH is enabled by default is only available using the internal IP address for troubleshooting and access purposes.
- The nodes use Azure Managed Disks for storage. For most VM SKUs, these are Premium disks with the stored data automatically encrypted at rest within the Azure platform.
- Additional security features such as Pod Security Policies or more fine-grained role-based access controls (RBAC) are needed to secure nodes from exploits that can occur with multi-tenant usage.
- For hostile multi-tenant workloads, you should use physically isolated clusters by leveraging hypervisor-level security where the security domain for Kubernetes becomes the entire cluster, not an individual node.
- The best practice for multi-tenant workloads is to use logical isolation to separate teams and projects. It is recommended to minimize the number of physical AKS clusters you deploy to isolate teams or applications.

Cluster Upgrades

The AKS cluster upgrade process involves individually cordoning the nodes from the cluster so that new pods cannot be scheduled on them. These nodes are then drained and follow the following procedure to upgrade:

- A new node is deployed into the node pool which runs the latest OS image and patches.
- One of the existing nodes is identified and marked for the upgrade. Pods on this node are gracefully terminated and scheduled on the other nodes in the node pool.
- This targeted node is then deleted from the AKS cluster.
- The next node in the cluster is cordoned and drained using the same process until all nodes are successfully replaced as part of the upgrade process.

Kubernetes Secrets

Sensitive data such as access credentials or keys can be ingested into pods using a **Kubernetes Secret**. The secret is first created using the Kubernetes API, and when you define your pod or deployment, a specific Secret can be requested. These secrets are only provided to nodes that have a scheduled pod, which requires a secret, and are stored in **tmpfs**, not written in the disk. A Secret is deleted from the node's tmpfs, when the last pod on a node requests its deletion. Furthermore, Kubernetes secrets are stored within a defined namespace and can only be accessed by pods within the same namespace.

By using Kubernetes secrets, you can minimize the sensitive information that is defined in the pod or service YAML manifest. Here you will request the Secret stored in the Kubernetes API server as part of your YAML manifest. By using this approach, you are only providing specific pod access to the Secret.

Note The raw secret manifest files contain the secret data in base64 format, and hence, this file should be treated as sensitive information and should never be committed to source control.

Monitoring Azure Kubernetes Service

An important part of operating Azure Kubernetes Service is being able to monitor the cluster, the nodes, and the workloads running in that AKS instance. Running production workloads requires a solid level of reliability. Azure comes with Kubernetes, and container monitoring out of the box is available in Azure Monitor. In this section, we are going to dive into the Kubernetes and container monitoring services that are available in Azure Monitor.

Azure Monitor for Containers

Overview

The monitoring service in Azure Monitor is called Azure Monitor for containers. Azure Monitor for containers gives you monitoring from two perspectives: the first one being directly from an AKS cluster and the second one being all AKS clusters in your subscription/s. The monitoring looks at two key areas “health status” and “performance charts” and consists of

Insights: Monitoring for the Kubernetes cluster and containers.

Metrics: Metric-based cluster and pod charts. It is based on a time series db which collects the data directly from the AKS resource provider for basic and standard performance metrics on pods and nodes.

Log Analytics: K8s and container logs viewing and search. It is the platform where Azure Monitor for containers store the data. You can run KQL queries to view all telemetries that Azure Monitor for containers collects such as perf, health, kubernetes events, container logs, and inventory.

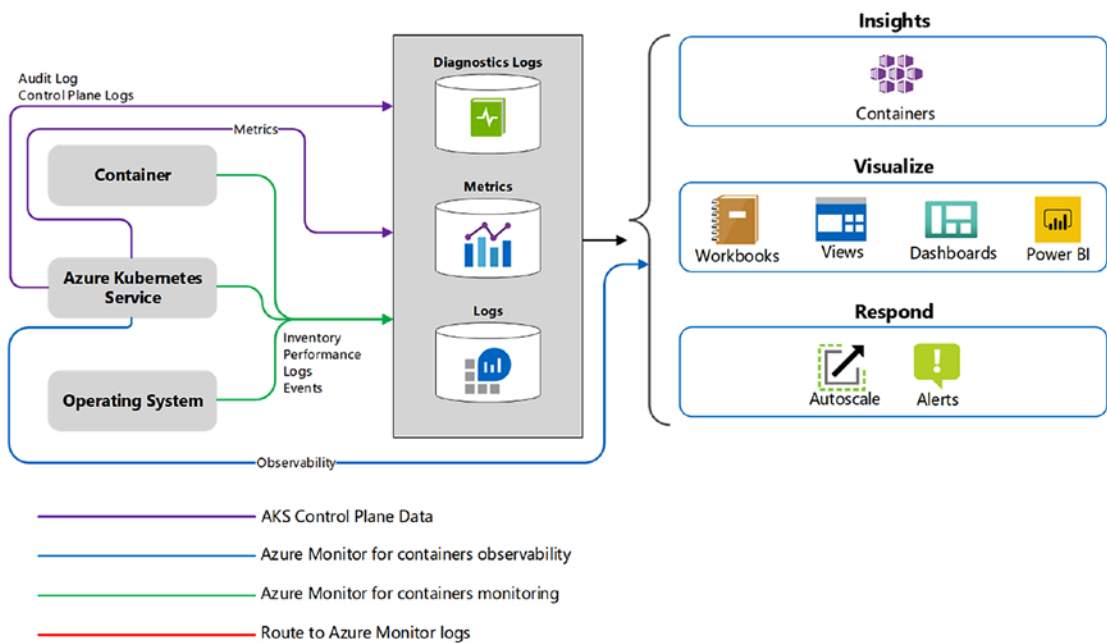


Figure 7-8. Azure Monitor for containers architecture

Enable Monitoring

The easiest way to enable monitoring for AKS is while deploying the AKS cluster. Refer to Chapter 5, “Deploying Azure Kubernetes Service,” for details on deploying an AKS cluster.

Monitoring of an existing AKS cluster can be enabled using one of the following methods:

- Within Azure Monitor or within the AKS cluster in the Azure portal
- Azure PowerShell cmdlet `New-AzResourceGroupDeployment` using the ARM Template from here: <https://docs.microsoft.com/en-us/azure/azure-monitor/insights/container-insights-enable-existing-clusters#enable-using-an-azure-resource-manager-template>
- Azure CLI
- Terraform

The best and fastest way to enable monitoring for an AKS cluster is from the Azure CLI in Azure Cloud Shell. To do this from a web browser, navigate to <https://shell.azure.com> PowerShell and run the following.

Listing 7-20. Enabling Azure Monitor for containers using Azure CLI

```
az aks enable-addons -a monitoring -n ExistingAKSCluster -g
ExistingAKSClusterResourceGroup
```

Azure Monitor

In Azure Monitor, you will find **Containers** under **Insights**. Here you will see a health summary across all AKS clusters in your Azure subscription. Also, you will see how many nodes and system/user pods an AKS cluster has and if there are any health issues with a node or pod. Click a cluster from here, and it will bring you to the Insights section on the AKS cluster itself. Clicking the AKS cluster will bring you to the Insights section of Azure Monitor for containers on the actual AKS cluster. Here you will see Insights, Metrics, and Logs. Let's now dive into each of these three areas.

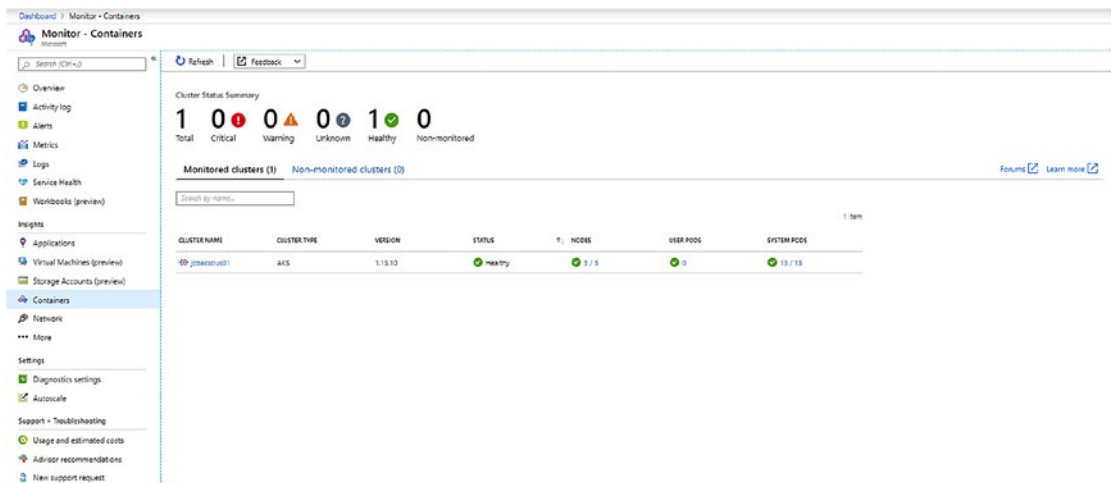


Figure 7-9. Azure Monitor for containers overview page

Insights

Within the Insights area, you will find a lot of useful data in regard to monitoring your AKS cluster. Inside Insights, you have these four areas: Cluster, Nodes, Controllers, and Containers.

Cluster

Within the Cluster tab, you will find charts with key performance metrics for your AKS clusters' health. It has performance charts for your node count with status and pod count with status, along with aggregated node memory and CPU utilization across the cluster. In here you can change the time range from real time, hours to days, and add filters to scope down to specific information such as service, namespace, node pool, and nodes that you want to see.

Nodes

On the Nodes tab, you will see the nodes running in your AKS cluster along with uptime, number of pods on the node, CPU usage, memory working set, and memory RSS. You can click the arrow next to a node to expand it, displaying the pods that are running on it. This provides you a quick way to see the noisy neighbors in your AKS cluster.

Controllers

On the Controllers tab, you will find the health of the cluster's controllers. Again, here you will see CPU usage, memory working set, and memory RSS of each controller and what is running a controller. For example, you could see a `kubernetes-dashboard` pod running on the `kubernetes-dashboard` controller.

You can also view the properties of the `kubernetes-dashboard` pod. The properties will give you information like the pod name, pod status, Uid, label, and more.

Containers

On the Containers tab, you will find all the containers in the AKS cluster. And as with the other tabs, you can see CPU usage, memory working set, and memory RSS. You also will see status, the pod it is part of, the node it's running on, its uptime, and if it has had any restarts.

You also can see a container logs in the containers tab. To do this, select a container to show its properties. Within the properties, you can click View container live logs as shown in the following screenshot or View container logs. Container log data is collected every three minutes. STDOUT and STDERR is the log output from each Docker container that is sent to Log Analytics.

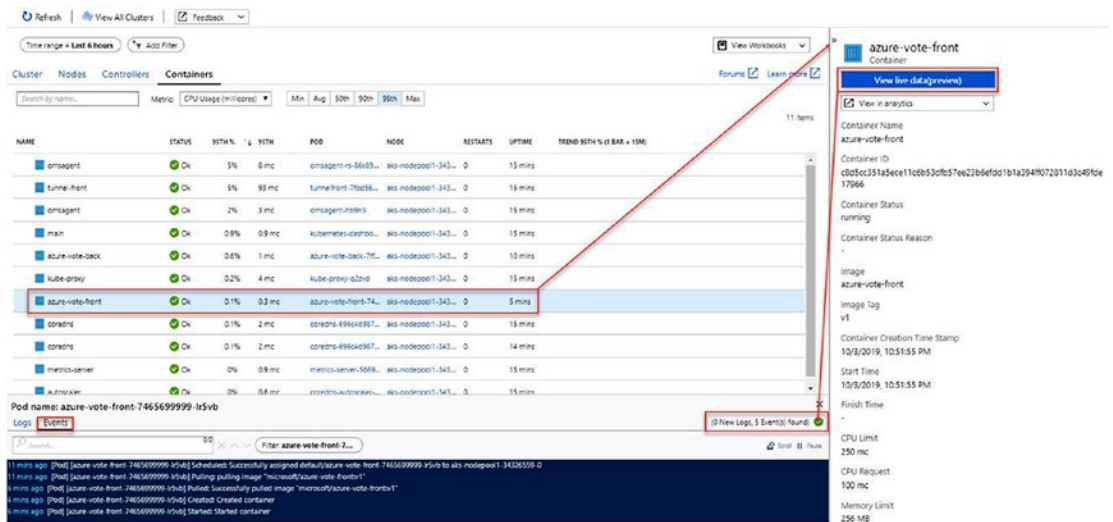


Figure 7-10. Azure Monitor for containers live logs and events

Clicking **View live data (preview)** will bring you to the Log Analytics log search page with that container's logs and events shown in the results pane.

Note Live data is available in node, controller, and containers tabs. They will show you kubernetes events (per cluster, namespace, and/or nodes and/or pods) and container logs.

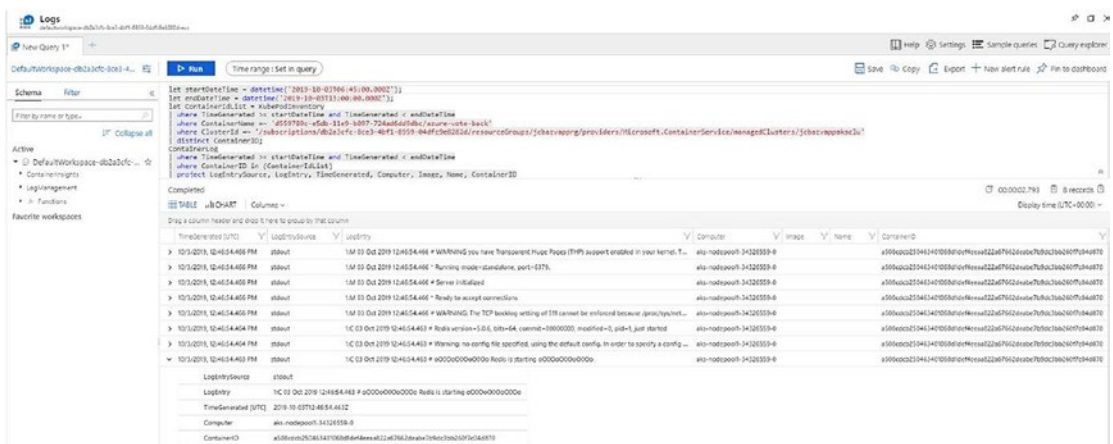


Figure 7-11. Azure Monitor for containers analytics view

kube-system is not currently collected and sent to Log Analytics. If you are not familiar with Docker logs, more information on STDOUT and STDERR can be found on this Docker logging article here: <https://docs.docker.com/config/containers/logging>.

Note If you want to collect logs for kube-system, you can do so by changing the configmap as per the following article: <https://docs.microsoft.com/en-us/azure/azure-monitor/insights/container-insights-agent-config>

Metrics

In the metrics area, you can see metric-based nodes and pod charts that can help you see information that is important to you about an AKS cluster. The following screenshot shows a couple of example charts displaying pods by phase split based on namespace and total of available cores in a cluster.

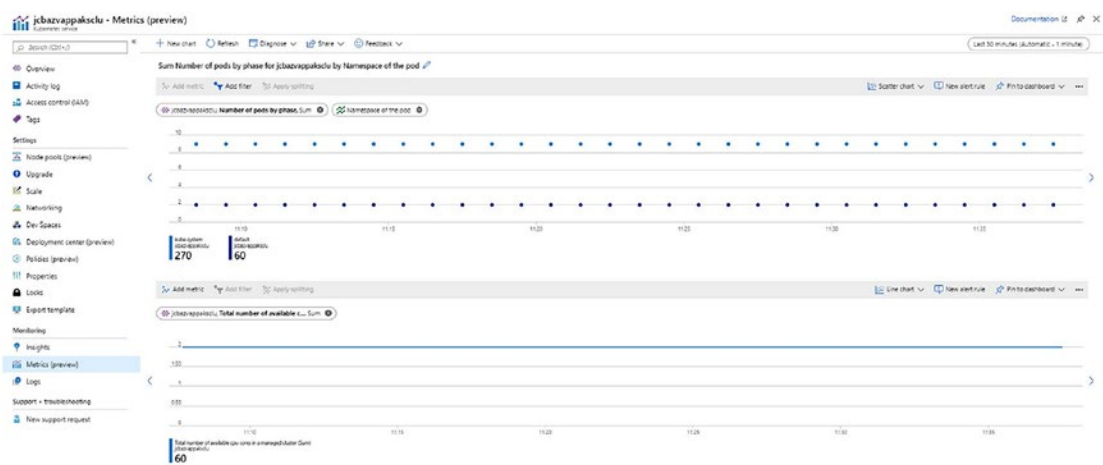


Figure 7-12. Azure Monitor for containers metrics view

At the time of writing this book, the only available standard metric namespace is **microsoft.containerservice/managedclusters** (from AKS resource provider) and custom metrics namespaces **insights.container/nodes** and **insights.container/pods** (from container insights). Aggregation can be Sum or Avg and the metrics you can see in the following screenshot:

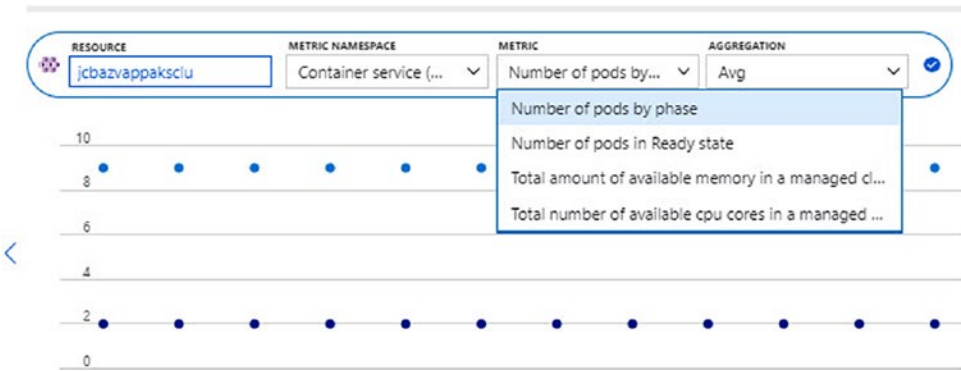


Figure 7-13. Azure Monitor for containers available metrics

Within the metrics area, you can pin charts to your Azure dashboard, and you can create an alert based on a condition such as when pods are in a failed state.

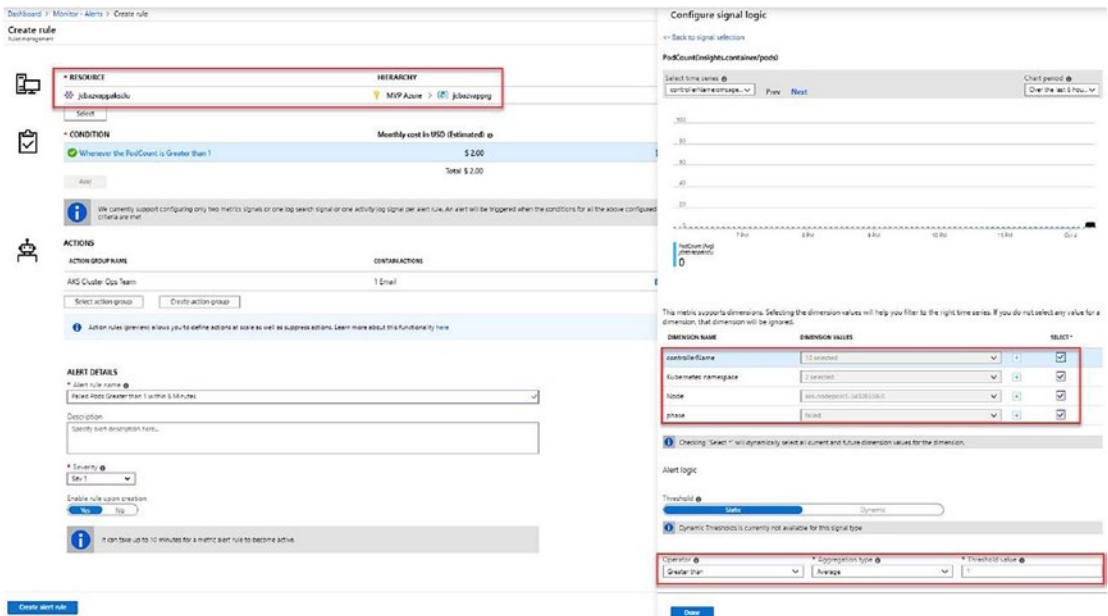


Figure 7-14. Create alert rules for Azure Monitor for containers

Log Analytics

Log Analytics is a feature of Azure Monitor. Log Analytics is utilized for many Azure services for viewing logs and searches; to analyze data to identify trends, patterns, and issues; for anomaly detection through Machine Learning; and more. In Log Analytics you can get deep insights into your AKS cluster and containers. The following screenshot shows the log schema that is collected in Azure Monitor for containers:

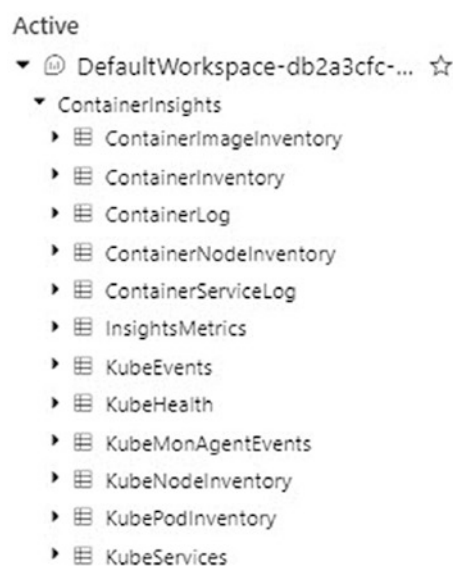


Figure 7-15. Azure Monitor for containers log schema

The data types in the ContainerInsights schema are what appear in Log Analytics search results. One way to show the Log Analytics search page is by clicking Logs from within the AKS cluster. From the search page, you can filter down the results of a search or run a query.

From the Log Analytics search page, you can build queries to retrieve scoped data. Here are three example Log Analytics queries for retrieving AKS data.

Listing 7-21. KQL query samples for retrieving AKS data

Pods that have a restart count greater than 0 in the last 48 hours

```
let startTimestamp = ago(48hrs);
KubePodInventory
| where ClusterName =~ "AKSCLUSTERNAME"
```

```
| where ContainerRestartCount > 0
| where isnotnull(Name)
```

Container lifecycle

ContainerInventory

```
| project Computer, Name, Image, ImageTag, ContainerState, CreatedTime,
  StartedTime, FinishedTime
| render table
```

Kubernetes events

KubeEvents_CL

```
| where not(isempty(Namespace_s))
| sort by TimeGenerated desc
| render table
```

Creating an Alert Rule Through Log Analytics

We are using the following sample query that returns pod phase counts based on all phases – *Failed*, *Pending*, *Unknown*, *Running*, or *Succeeded* – to create an alert rule through Log Analytics queries.

Listing 7-22. KQL query sample to retrieve pod phase counts based on all phases

```
let endDateTime = now();
let startDateTime = ago(1h);
let trendBinSize = 1m;
let clusterName = '<your-cluster-name>';
KubePodInventory
| where TimeGenerated < endDateTime
| where TimeGenerated >= startDateTime
| where ClusterName == clusterName
| distinct ClusterName, TimeGenerated
| summarize ClusterSnapshotCount = count() by bin(TimeGenerated,
  trendBinSize), ClusterName
| join hint.strategy=broadcast (
  KubePodInventory
```



```

| where TimeGenerated < endDateTime
| where TimeGenerated >= startDateTime
| distinct ClusterName, Computer, PodUid, TimeGenerated, PodStatus
| summarize TotalCount = count(),
              PendingCount = sumif(1, PodStatus =~ 'Pending'),
              RunningCount = sumif(1, PodStatus =~ 'Running'),
              SucceededCount = sumif(1, PodStatus =~ 'Succeeded'),
              FailedCount = sumif(1, PodStatus =~ 'Failed')
              by ClusterName, bin(TimeGenerated, trendBinSize)
) on ClusterName, TimeGenerated
| extend UnknownCount = TotalCount - PendingCount - RunningCount -
  SucceededCount - FailedCount
| project TimeGenerated,
          TotalCount = todouble(TotalCount) / ClusterSnapshotCount,
          PendingCount = todouble(PendingCount) / ClusterSnapshotCount,
          RunningCount = todouble(RunningCount) / ClusterSnapshotCount,
          SucceededCount = todouble(SucceededCount) /
            ClusterSnapshotCount,
          FailedCount = todouble(FailedCount) / ClusterSnapshotCount,
          UnknownCount = todouble(UnknownCount) / ClusterSnapshotCount
| summarize AggregatedValue = avg(PendingCount) by bin(TimeGenerated,
trendBinSize)

```

Note The following procedure to create an alert rule for container resource utilization requires leveraging a new log alerts API as in the following URL:

<https://docs.microsoft.com/en-us/azure/azure-monitor/platform/alerts-log-api-switch>.

Follow the following steps to create a log alert in Azure Monitor by using Log Analytics queries:

1. Log into the Azure portal, select **Monitor** from the left pane, navigate to **Insights**, and then select **Containers**.
2. Select a cluster from the list in the **Monitored Clusters** tab.

3. Select **Logs** to open the Azure Monitor logs page under **Monitoring**. In this page, you can write and execute Azure Log Analytics queries.
4. On the **Logs** page, select **+New alert rule**.
5. Under the **Condition** section, select **Whenever the Custom log search is <logic undefined>** custom log condition. Since we're creating an alert rule directly from the Azure Monitor logs page, the custom log search signal type is automatically selected.
6. Paste the query from Listing 7-22 into the **Search query** field.
7. Follow the following steps to configure the alert:
 - a. Select **Metric measurement** under the **Based on** drop-down list. Here a metric measurement creates an alert for each object in the query that has a value above our specified threshold.
 - b. Under **Condition**, select **Greater than**, and enter **75** as an initial baseline Threshold for the CPU and memory utilization alerts. For the low disk space alert, enter **90**. You can enter a different value that meets your criteria.
 - c. Select **Consecutive breaches**, under the **Trigger Alert Based On** section. In the drop-down list, select **Greater than** and enter **2**.
 - d. If you want to configure an alert for container CPU or memory utilization, select **ContainerName** under **Aggregate on**. If you want to configure for cluster node low disk alert, select **ClusterId**.
 - e. Under the **Evaluated based on** section, configure the **Period value** to **60 minutes**. By doing so, the rule will execute every 5 minutes and will return records that were created within the last hour from the current time. When you set the time span to a wider window that will result in potential data latency to ensures that the query returns data to avoid any false negative where the alert is never triggered.
8. Click **Done**.
9. Provide a meaningful name in the **Alert rule name** field. You can also specify a **Description** that provides details about what this alert does. Finally select an appropriate severity level for this alert.

10. Accept the default value for **Enable rule upon creation**, so that the alert is immediately activated.
11. You can select an existing Action Group or create a new group. This is how you can ensure that the same actions are taken every time that this alert is triggered. You can configure this section, depending how your ITSM team manages incidents.
12. Click **Create alert rule** to complete the alert rule. The rule starts executing immediately.

For more information on the creating alerts using the Log Analytics query language, you can visit the Microsoft documentation here: <https://docs.microsoft.com/en-us/azure/azure-monitor/insights/container-insights-alerts>.

Kubelet Logs

If you have issues with a node, you should start your troubleshooting using the node monitoring available in Azure Monitor for containers. If there is a need to go beyond Azure Monitor for containers, you can use the kubelet logs. You can view the kubelet logs from any of the AKS nodes using `journalctl`. To do this, you need to first SSH to the cluster node you want to see the logs for. Once connected to the node through SSH, execute the following syntax.

Listing 7-23. kubelet log retrieval

```
sudo journalctl -u kubelet -o cat
```

That will begin rolling through the kubelet logs giving you insight into activity occurring on the node.

Kubernetes Master Component Logs

It is important to note that with AKS the Kubernetes master node logs are not collected by default. These logs are not collected because AKS is a managed service by Microsoft and they manage the master Kubernetes nodes. Hence, it is not common to dig into troubleshooting master nodes. In the event that you need to see logs from any of the master nodes, you can turn on log collection sending the logs to a Log Analytics workspace.

To enable the master node log collection in the Azure portal, navigate to the AKS resource group. Do not go to the AKS resource group with this name format MC_ResourceGroupName_AKSclusterNAME_REGION. Once in the AKS resource group, click Diagnostic settings. Click the AKS cluster.

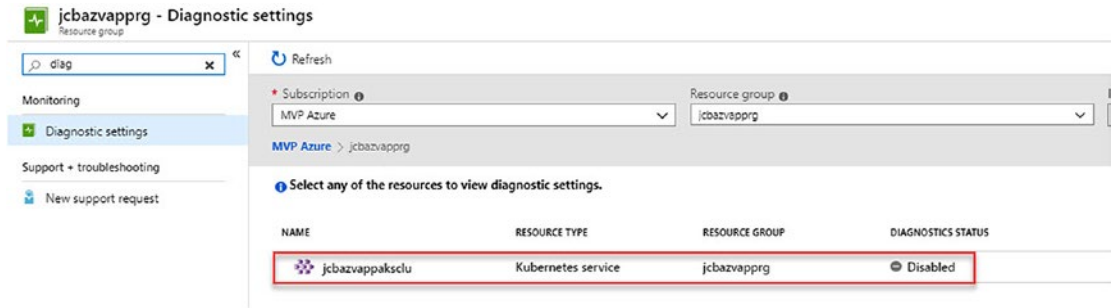


Figure 7-16. Diagnostics settings for AKS cluster

Then click **Add diagnostic setting**.

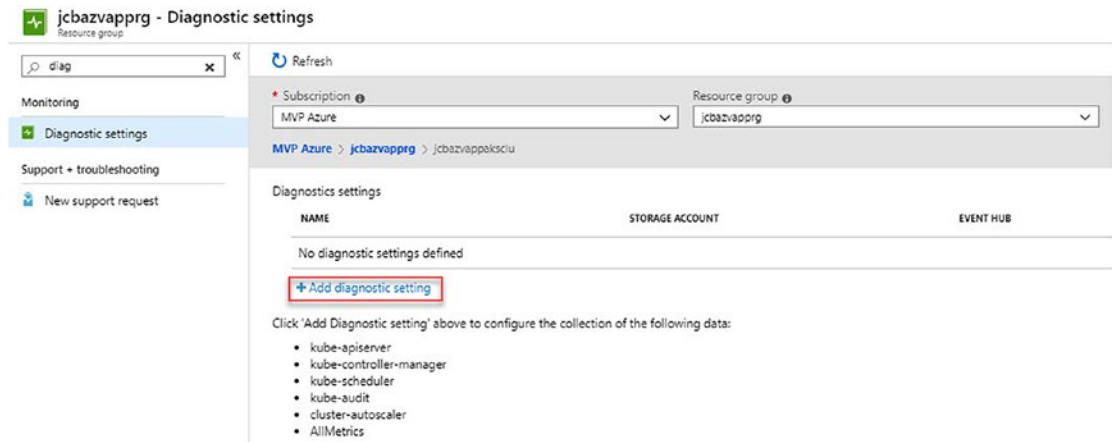


Figure 7-17. Add diagnostics settings for AKS cluster

Configure the diagnostics settings like in the following screenshot to send the logs to a Log Analytics workspace. You will give the diagnostics collection a name, select or create a new Log Analytics workspace, and select the master nodes that you want to collect logs from.

Diagnostics settings

Save

Discard

Delete

Name

jcbazvappaksciuMasterNodeLogs

Archive to a storage account

Stream to an event hub

Send to Log Analytics

Subscription

MVP Azure

Log Analytics Workspace

DefaultWorkspace-db2a3cfc-8ce3-4bf1-8959-04dfc9e6262d-EUS (eastus)

LOG

kube-apiserver

kube-controller-manager

kube-scheduler

kube-audit

cluster-autoscaler

METRIC

AllMetrics

Figure 7-18. *Configure diagnostics settings for AKS cluster*

After you save the diagnostics log settings, you should now see this set on the AKS resource group as shown in Figure 7-19.

jcbazvapprg - Diagnostic settings

Monitoring

Diagnostics settings

Support + troubleshooting

New support request

Refresh

Subscription

MVP Azure

Resource group

jcbazvapprg

Resource type

Kubernetes services

Resource

jcbazvappaciu

Diagnostics settings

NAME	STORAGE ACCOUNT	EVENT HUB	LOG ANALYTIC	EDIT SETTING
jcbazvappaksciuMasterNodeLogs			defaultworkspace-db2a3cfc-8ce3-4bf1-8959-04dfc9e6262d-EUS (eastus)	Edit setting

Add diagnostic setting

Figure 7-19. *Diagnostics settings for AKS cluster configured*

To see the actual logs from the Kubernetes master nodes, go to the Log Analytics workspace that you sent the logs to and run one of the search queries shown in Listing 7-24.

Listing 7-24. KQL queries to retrieve Kubernetes master logs

```
AzureDiagnostics
| Where Category == "kube-apiserver"
| project log_s

AzureDiagnostics
| where Category == "kube-controller-manager"
| project log_s

AzureDiagnostics
| where Category == "kube-scheduler"
| project log_s

AzureDiagnostics
| where Category == "kube-audit"
| project log_s

AzureDiagnostics
| where Category == "guard"
| project log_s

AzureDiagnostics
| where Category == "cluster-autoscaler"
| project log_s
```

Business Continuity and Disaster Recovery in Azure Kubernetes Service

The applications that run in your AKS cluster will have certain service-level objectives and agreements (SLOs and SLAs) that are requested by the business. As an AKS operator, one of your responsibilities is to consider how to deploy and manage AKS to meet those SLAs and SLOs. All services will have outages of one kind or another, and Azure Kubernetes Service is no different. By understanding the underlying components of AKS and how they provide services, you can meet or exceed the expectations of the application owners.

Thinking About SLAs and What You Need

When thinking about disaster recovery, it is useful to understand some basic concepts. There are a few primary measures of protection when in DR, namely, the recovery time objective (RTO) and recovery point objective (RPO). The RTO defines the amount of time it will take to recover service in the event of an outage. The RPO defines the amount of data that is lost when such an outage occurs.

Both terms rely on the declaration of a formal disaster, where an entire site or service is completely unavailable. There may also be situations where a site or service is in a degraded mode and the determination is made to failover to another site or instance of the service. The RTO and RPO are measured against when a disaster declaration is made.

Before a failure is declared, you can ensure that your AKS clusters are deployed in such a way as to protect against common failures. Let's take a look at the various levels of failure that exist and how to use AKS and Azure features to protect against them.

Data Persistence and Replications

Applications running within an AKS cluster may have stateful data that is written to persistent storage. One point of consideration is the replication and protection level afforded by that storage. The storage may be located on any of the following:

- Local storage on the Azure VM worker node
- Azure Managed Disks
- Azure Files
- Other NFS solutions

Local storage on the Azure VM worker nodes uses Azure Managed Disks, as does the in-tree provisioning mechanism for persistent Managed Disk volumes. Managed Disks provide locally redundant storage only, which provides protection against drive failures within a datacenter, but does not provide protection against datacenter failures in Azure. Azure Files can be configured to use Geo-redundant storage, where data is replicated from one Azure datacenter to a paired datacenter in another region. Other NFS solutions may provide different levels of redundancy.

If there is persistent data on the application volumes that must be protected against a site failure, then some type of replication or data protection solution should be put in place to protect that data in the event of a failure.

Protecting Against Faults

In addition to protecting against data failure, there are a number of other faults that can occur in AKS. The following sections will review each of those faults and possible mitigation strategies to protect your applications.

Master Node Failures

Azure Kubernetes Service is a managed service that does not provide visibility into the master node layer of the cluster. In the event of a master node failure, the cluster will automatically replace that master node with a new one. There is very little you can do as an operator to protect against the failure of a master node.

Worker Node Failures

The Azure VMs functioning as worker nodes are subject to occasion faults and failures. In the event that a node fails, the AKS service will replace that node with a functional one. However, there will be a period of time when you are running at reduced capacity. For critical clusters, it is recommended to run with enough spare capacity to absorb an individual node outage without compromising performance.

The cluster can be configured manually with enough nodes to support the current performance requirements with enough overhead to maintain performance during an outage. For instance, let's assume you have a four-node cluster running at 75% capacity. If a node is lost, the other three nodes will need to run at 100% capacity to match the current performance objective until the fourth node is replaced. That is a less than desirable situation. By adding a fifth node to the cluster, the overall cluster utilization will now be at 60%, and a single node loss will result in an increase to 75% during the outage.

Configuring your cluster size manually is an option, but as well all know, cluster consumption is going to be variable. For that reason, it makes more sense to monitor the current utilization and trigger an action to scale the cluster as needed or make use of the cluster autoscale feature currently in preview.

Datacenter Failures

Microsoft Azure regions are comprised of multiple datacenters. Recently, the availability zones have been introduced in many Azure regions. Each availability zone is a geographically separate set of resources with high-bandwidth, low-latency connections

to other availability zones in the same region. The purpose of an availability zone is to provide protection against datacenter failures in a given region.

AKS has a preview feature that allows an AKS cluster to span multiple availability zones. In the event of a datacenter outage, your applications and the AKS management plane will continue to operate uninterrupted, assuming that you have worker nodes available in each availability zone. This feature is likely to become generally available in the near future, but it will require redeploying your cluster to migrate it to availability zones. For the time being, datacenter outages for AKS will need to be treated like regional failures.

Regional Failures

While it is highly uncommon, a regional outage of Azure is not completely unheard of. AKS clusters do not stretch across regions, so the question becomes what level of protection is required for the applications running on your cluster. There are a few different operations models:

- Cold start
- Pilot light
- Warm cluster
- Hot cluster

Each has different cost and recovery characteristics. A cold start would involve the creation of a new AKS cluster in another region. The persistent for applications in the cluster would be recovered from a backup. Once the AKS cluster was provisioned and operational, and the backups restored to the proper storage target, applications could be spun up on the cluster. This is the lowest cost option and will have a high RTO and RPO.

A pilot light scenario would include a running AKS cluster in another region with reduced capacity. Again the persistent data for the applications running in the cluster would be recovered from a backup. In the event of a disaster, the cluster would be scaled up, the backups restored, and applications deployed. This is a low-cost option, due to the reduced capacity of the cluster, and still has a high RTO and RPO.

The warm cluster scenario involved running a fully provisioned AKS cluster in another region with applications already running. Some type of storage replication service would be in place with a lag of several minutes or hours. Recovery would simply involve switching over public facing DNS entries to the warm site. This is a higher-cost

solution due to the higher capacity of the cluster and data replication, but the RTO and RPO are both reduced greatly.

The hot cluster scenario would have a fully provisioned AKS cluster running in another region, with applications already running and serving request. The storage replication solution in this case would need to be nearly synchronous. A failure of one region would simply require scaling up the cluster on the over region to handle the additional load. This is the highest-cost solution by far, but the RTO and RPO approach zero for a failure.

Each of the solutions has its pros and cons; therefore, it is up to the application owners to determine what is an acceptable amount of downtime and data loss for their application compared to the cost of additional protection.

Summary

Before you deploy application in AKS, it is important to understand how to properly administer your AKS resources. The role of an AKS cluster operator is the key here. Although AKS is a managed Kubernetes service, there are management operations such as scaling, identity and access, networking, securing, monitoring, and business continuity planning that need to be planned out well ahead.

In this chapter, you learned about the common cluster management operations in AKS that you will encounter frequently. We explored how to properly scale AKS clusters, what are the storage options available for AKS, and the necessary AKS networking, access and identity, and security concepts for you to get started with managing an AKS cluster. Then, we examined how Azure Monitor for containers can help you to monitor your AKS resources. Finally, we discussed the business continuity and disaster recovery best practices for Azure Kubernetes Service deployments.

CHAPTER 8

Helm Charts for Azure Kubernetes Service

Applications deployed on Kubernetes are typically made up of multiple parts. A common practice is to combine multiple components into a single *yaml* file that will be submitted to the cluster using `kubectl apply` with the `-f` switch. It is also common to deploy the same application across multiple environments, whether those environments are separate Kubernetes clusters or different namespaces within the same cluster. There are several shortcomings to the approach of using a single *yaml* file and `kubectl` for application deployment across multiple environments. Helm was created to address some of these shortcomings.

In this chapter, we will explore the use cases for Helm and how it enhances the application deployment experience on Kubernetes. We will review the process for installing the Helm client and Tiller on an AKS cluster in both a development- and production-type scenario. Then, we will dive into the structure of Helm charts – the basic construct for application deployment in Helm. Finally, we will go through the process of deploying and updating Helm chart releases in AKS. By the end of this chapter, you will have a solid understanding of what Helm is and how it can be used with AKS to simplify and enhance the application deployment process.

Helm Overview

Helm is an open source project maintained by the Cloud Native Computing Foundation in collaboration with Microsoft, Google, Bitnami, and more. The main goal of Helm is to assist in the management of Kubernetes-based applications, including the definition, installation, and upgrade of those applications. One useful way to think of Helm is as a package manager for Kubernetes. In the same vein as `apt`, `yum`, or `Chocolatey`,

Helm has repositories with packages that can be copied and installed locally – locally meaning Kubernetes. It can also handle the upgrade and removal of those applications.

The fundamental construct used by Helm for managing Kubernetes applications is the Helm *chart*. The chart defines the components that make up an application in a standardized format that can be shared and stored in source control. When a chart is combined with configuration information and deployed on a Kubernetes cluster, it is called a *release*.

Use Cases

Helm is intended to simplify application management on Kubernetes. In that regard, there are a few primary use cases that Helm simplifies. Kubernetes applications tend to be composed of multiple resources and components. Helm charts help you manage the complexity of these applications by describing the components and dependencies in a declarative fashion.

Applications are not static deployments, rather they are routinely updated. The update process on Kubernetes can be tricky. Helm provides a simpler update experience, managing revisions and updates for the application as a whole instead of its component parts. New versions of charts can include tests for validation, custom hooks for the release process, and a simple rollback process if the newest chart has issues.

Developers try to follow the DRY principal of “don’t repeat yourself.” Helm expands that concept to the deployment of Kubernetes applications. A Helm chart can reference other charts for dependencies, for example, a single web front-end chart can be reused for multiple applications in an environment. Sharing of the charts can be on a public or private repository, and in the case of Azure Container Registry, the charts can be stored in the same registry as the containers being used by the chart.

Advantages over Kubectl

Kubectl is the CLI tool of choice for managing Kubernetes, and Helm is not intended to replace kubectl for all activities. In fact, kubectl is often used in tandem with the Helm CLI to troubleshoot, investigate, and track application deployments across the cluster. Both kubectl and helm interact with the Kubernetes API to accomplish their work. Helm has the advantage of being able to use the templates written for Helm, and it has several higher-level commands that abstract away multiple kubectl commands. For instance,

when an application is deployed using `helm install`, the Helm software is interacting with the Kubernetes API directly and orchestrating the deployment of resources on the cluster in a way that would take multiple `kubectl` commands.

Key Components

To perform the work of making Kubernetes application management simpler, Helm has several key components that make up an installation.

Note The current major version of Helm is version 2. Version 3 of Helm is in the alpha stage of development and contains several large changes in how Helm is constructed. In particular, the Tiller component will be removed in version 3. For the purpose of this chapter, we will exclusively deal with version 2.

Helm Client

The Helm client is a binary written in Go that runs on the local machine of the user or on some type of CI/CD platform. It is equivalent to `kubectl` in that regard. The client can be installed on multiple operating systems, including Windows, MacOS, and Linux. The most current version of the Helm client binary can always be found on the Helm GitHub releases page (<https://github.com/helm/helm/releases>). To install the client locally, you can use Chocolatey for Windows, Homebrew for Mac, or Snap for some Linux distributions. Listing 8-1 shows an example of installing the Helm client on a Windows machine with Chocolatey.

Listing 8-1. Installing the Helm client on a Windows machine

```
#Install the client
$ choco install kubernetes-helm -y

Chocolatey v0.10.3
Installing the following packages:
kubernetes-helm
By installing you accept licenses for the packages.
...
```

The install of kubernetes-helm was successful.

```
Software installed to 'C:\ProgramData\chocolatey\lib\kubernetes-helm\tools'
```

#Check the client version after installation

```
$ helm version
```

```
Client: &version.Version{SemVer:"v2.14.2", GitCommit:"a8b13cc5ab6a7dbef0a58f5061bcc7c0c61598e7", GitTreeState:"clean"}
```

Since we have not yet configured a connection to a Kubernetes cluster, the server version will come back with an error.

Tiller

Tiller is the server-side component of Helm that takes the commands issued by the Helm client and executes them on the cluster through the Kubernetes API. The Tiller component is most often deployed on the Kubernetes cluster where it will deploy applications, although this is not entirely necessary. It is also possible to run the Tiller component outside of the Kubernetes cluster. The Tiller component is responsible for four primary things:

1. Listening for requests from the Helm client
2. Deploying chart and config info as a release
3. Tracking releases through their life cycle
4. Upgrading or removing releases from the cluster

Tiller can run using a service account with RBAC rules that define what namespaces Tiller has access to. Tiller will have the ability to create and destroy applications on the Kubernetes cluster; thus, it makes sense to employ roles to restrict the actions an instance of Tiller can take. In a production environment, or really any nondevelopment environment, Tiller should be using a service account with proper restrictions in place to control what resources it can manage in the cluster.

When Tiller is installed on a cluster, it creates an in-cluster gRPC endpoint that is unauthenticated by default. Basically, this means that any process within the cluster could issue commands to the Tiller endpoint, and they would be executed. For a development cluster, that might be acceptable. All other cluster environments should endeavor to use TLS to secure authentication on the Tiller endpoints. When

TLS is enabled with Tiller, all communication with the Tiller endpoints is mutually authenticated with TLS certificates issued by a trusted root certificate authority.

Helm Repository

The charts used by Helm can be stored in a repository. The repository can either be hosted privately or publicly. The Helm project maintains an official, public chart repository located on their GitHub site (<https://github.com/helm/charts>). This is a great starting point to find official versions of Helm charts for common applications, such as *Wordpress*, *FluentD*, and *Jenkins*.

The chart repository is simply a web server with an `index.yaml` file that lists out all the charts being stored in the repository, along with some information about each chart. Standing up a Helm repository is outside the scope of this book, but the process is relatively simple and can be accomplished using *ChartMuseum*, *GitHub Pages*, or a simple web server.

Azure Container Registry is also able to store Helm charts. An example of using ACR to store a Helm chart will be given later in the chapter.

Cloud Native Application Bundle

The Cloud Native Application Bundle (CNAB) is an open source project founded by Microsoft and Docker to deal with the packaging of applications that leverage more than just containers and Kubernetes for their deployment. For example, a three-tier web application could be using Azure CosmosDB for database services, AKS for the application and web tier, and Azure Functions for business logic processing. A CNAB bundle would be able to deploy and manage all of these components. Helm is focused solely on application components that are deployed in the context of Kubernetes. While the landscape is still changing, CNAB and Helm accomplish two different, related goals.

Installing Helm on AKS

As mentioned in the Helm components section, there are two basic components to installing Helm. There is the Helm client running locally on a workstation and the Tiller server-side component running on the Kubernetes cluster. To set Helm up to work with AKS, there are a few requirements that need to be fulfilled.

Requirements

Azure Kubernetes clusters are deployed with RBAC enabled by default. Getting the Tiller component working properly with RBAC on the cluster requires that a service account be created and associated with a cluster role. Enabling TLS is also best practice for a nondevelopment environment. In the next two sections, we will walk through the process of setting up the service account and provisioning the necessary certificates to enable TLS authentication between Tiller and the Helm client.

RBAC and Service Account

Role-based access control in Kubernetes includes several different components. The *role* defines a set of actions that an assigned entity can perform on resources within the cluster. There are built-in roles for the cluster, such as *cluster-admin*, *admin*, *edit*, and *view*. A role can be one of two types, Role is namespace specific and ClusterRole is cluster-wide. Roles can be assigned to service accounts, users, and groups by using either the RoleBinding or ClusterRoleBinding type. In Listing 8-2, we are defining a service account for use with Tiller.

Listing 8-2. Definition for a Tiller service account

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
```

Once the service account has been created, it can be assigned a role. In Listing 8-3, we are going to associate the *tiller* service account with the built-in ClusterRole *cluster-admin* by using the ClusterRoleBinding type.

Listing 8-3. Binding the cluster-admin role to the Tiller service account

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: tiller
```



```

roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system

```

Depending on the requirements of the environment, it is also possible to create a custom Role and bind it using RoleBinding to a specific namespace where Tiller will be allowed to deploy resources. For our purposes, Tiller will be allowed to deploy resources across all namespaces in the cluster. In Listing 8-4, both of these configurations have been saved to the file `helm-rbac.yaml`, and `kubectl apply` is being run against an AKS cluster where Tiller will be configured.

Listing 8-4. Binding the cluster-admin role to the Tiller service account

```

$ kubectl apply -f helm-rbac.yaml

serviceaccount "tiller" created
clusterrolebinding.rbac.authorization.k8s.io "tiller" created

```

The service account for Tiller is now available and bound to the *cluster-admin* role.

TLS Considerations

Deploying a full Public Key Infrastructure (PKI) is just a bit outside the scope of this book. In fact, whole books have been written on just that topic. If your organization already has an internal PKI set up, it would make sense to take advantage of it. In the following examples, we are going to create the certificates using `openssl`. There are three certificates in play: the root certificate authority, the tiller certificate, and the helm client certificate. The tiller and helm client certificates will be approved and signed by the root CA certificate, and Tiller and Helm will be configured to trust the root CA certificate. Since they both trust the root CA, they will trust certificates signed by the root CA, meaning that Tiller and Helm will trust each other's certificates to be valid.

All of the commands in Listing 8-5 will create the certificates and keys in the current working directory.

Listing 8-5. Creating TLS certificates for Tiller and Helm communication

#First we must create the root CA.

#Big thanks to this article: <https://medium.com/google-cloud/install-secure-helm-in-gke-254d520061f7>

\$SUBJECT = "/C=US/ST=Pennsylvania/L=Springfield/O=IAKS, Inc./OU=IT/CN=iaks.sh"

#Create a CA key

openssl genrsa -out ca.key.pem 4096

#Create a CA certificate

openssl req -key ca.key.pem -new -x509 -days 7300 -sha256 -out ca.cert.pem
-extensions v3_ca -subj \$SUBJECT

#Then we need to create the certificate request for the Tiller certificate and process it.

#Create a key for the tiller cert

openssl genrsa -out tiller.key.pem 4096

#Create a new certificate request

openssl req -new -sha256 -key tiller.key.pem -out tiller.csr.pem -subj
\$SUBJECT

#Create the certificate from the request

openssl x509 -req -days 365 -CA ca.cert.pem -CAkey ca.key.pem -CAcreateserial
-in tiller.csr.pem -out tiller.cert.pem

#Finally, we need to create the certificate request for the Helm client certificate and process it.

#Create a key for the helm client

openssl genrsa -out helm.key.pem 4096

#Create a new certificate request

openssl req -new -sha256 -key helm.key.pem -out helm.csr.pem -subj \$SUBJECT

#Create the certificate from the request

openssl x509 -req -days 365 -CA ca.cert.pem -CAkey ca.key.pem -CAcreateserial
-in helm.csr.pem -out helm.cert.pem

Now we have all the necessary certificates and their matching private keys. Looking in the current directory, we should see the files in Listing 8-6.

Listing 8-6. Directory listing of TLS certificates and private keys

Mode	LastWriteTime		Length	Name
----	-----		-----	----
-a----	7/16/2019	1:39 PM	2070	ca.cert.pem
-a----	7/16/2019	1:39 PM	3298	ca.key.pem
-a----	7/16/2019	1:40 PM	18	ca.srl
-a----	7/16/2019	1:40 PM	1946	helm.cert.pem
-a----	7/16/2019	1:40 PM	1736	helm.csr.pem
-a----	7/16/2019	1:40 PM	3298	helm.key.pem
-a----	7/16/2019	1:40 PM	1946	tiller.cert.pem
-a----	7/16/2019	1:40 PM	1736	tiller.csr.pem
-a----	7/16/2019	1:39 PM	3294	tiller.key.pem

Each user who will use the Helm client to connect should be issued their own certificate, including any automation accounts running in a CI/CD pipeline. In a production scenario, issuance of certificates would be handled through a certificate authority. While it would be possible to use a third-party certificate authority, an internal CA would make more sense in this context. The Kubernetes cluster will likely be using internal names and be accessed by internal users. Spending money on certificates from a trusted third-party would be unnecessary.

Helm init

Once the prerequisites for the installation of Tiller have been fulfilled, the next step is to run the command `helm init` to initialize the cluster. In a development environment, it is enough to simply run `helm init` with all the defaults that the command implies. Since we will be using a service account and certificates, we will need to add arguments to the `helm init` command.

The commands in Listing 8-7 make use of the *tiller* service account and install the tiller private key, certificate, and root CA certificate. Additionally, the certificate information for Tiller is held in a ConfigMap by default. Due to the sensitive nature of the information, the best practice is to override the default setting and instead use a Secret-type resource to hold the data.

Listing 8-7. Initializing Tiller on the AKS cluster

```
$ helm init /
  --override 'spec.template.spec.containers[0].command={/tiller,--
storage=secret}' /
  --tiller-tls /
  --tiller-tls-cert ".\tiller.cert.pem" /
  --tiller-tls-key ".\tiller.key.pem" /
  --tiller-tls-verify /
  --tls-ca-cert ".\ca.cert.pem" /
  --service-account tiller
```

Tiller is installed as a deployment on Kubernetes. By default, it runs a single pod in a replica set and includes a service with a ClusterIP associated with it. Both the pod and the service are described in Listing 8-8.

Listing 8-8. Tiller pod and service details

```
$ kubectl describe pod tiller-deploy-6656966795-7sxqx --namespace kube-system
```

```
Name:                tiller-deploy-6656966795-7sxqx
Namespace:           kube-system
Priority:             0
PriorityClassName:    <none>
Node:                aks-agentpool-28083664-0/10.240.0.4
Start Time:          Tue, 16 Jul 2019 14:01:20 -0400
Labels:              app=helm
                    name=tiller
                    pod-template-hash=6656966795
Annotations:         <none>
Status:              Running
IP:                  10.244.0.8
Controlled By:       ReplicaSet/tiller-deploy-6656966795
Containers:
  tiller:
    Container ID:     docker://fd05f519da5911b07e0d2aa476b0c9661fe3181ee63a043
                      ca5188eb675bbb64b
    Image:             gcr.io/kubernetes-helm/tiller:v2.14.2
```

```

Image ID:      docker-pullable://gcr.io/kubernetes-helm/tiller@sha256:b
               e79aff05025bd736f027eaf4a1b2716ac1e09b88e0e9493c9626425
               19f19d9c
Ports:         44134/TCP, 44135/TCP
Host Ports:    0/TCP, 0/TCP
Command:
  /tiller
  --storage=secret
State:         Running
  Started:     Tue, 16 Jul 2019 14:01:32 -0400
Ready:         True
Restart Count: 0
Liveness:      http-get http://:44135/liveness delay=1s timeout=1s
               period=10s #success=1 #failure=3
Readiness:     http-get http://:44135/readiness delay=1s timeout=1s
               period=10s #success=1 #failure=3
Environment:
  TILLER_NAMESPACE:  kube-system
  TILLER_HISTORY_MAX: 0
  TILLER_TLS_VERIFY: 1
  TILLER_TLS_ENABLE: 1
  TILLER_TLS_CERTS:  /etc/certs
Mounts:
  /etc/certs from tiller-certs (ro)
  /var/run/secrets/kubernetes.io/serviceaccount from tiller-token-2dbcn
  (ro)
Conditions:
  Type           Status
  Initialized     True
  Ready           True
  ContainersReady True
  PodScheduled    True
Volumes:
  tiller-certs:
    Type:      Secret (a volume populated by a Secret)

```

```

    SecretName: tiller-secret
    Optional:   false
tiller-token-2dbcn:
    Type:       Secret (a volume populated by a Secret)
    SecretName: tiller-token-2dbcn
    Optional:   false
QoS Class:     BestEffort
Node-Selectors: <none>

$ kubectl describe svc tiller-deploy --namespace kube-system

Name:          tiller-deploy
Namespace:     kube-system
Labels:        app=helm
               name=tiller
Annotations:    <none>
Selector:      app=helm,name=tiller
Type:          ClusterIP
IP:            10.0.84.117
Port:          tiller 44134/TCP
TargetPort:    tiller/TCP
Endpoints:     10.244.0.8:44134
Session Affinity: None
Events:        <none>

```

After running the initialization, connectivity to Tiller from the Helm client can be tested by running the command in Listing 8-9.

Listing 8-9. Testing helm client connectivity to Tiller

```

$ helm version --tls --tls-ca-cert ca.cert.pem /
  --tls-cert helm.cert.pem --tls-key helm.key.pem

Client: &version.Version{SemVer:"v2.14.2", GitCommit:"a8b13cc5ab6a7dbef0a58
f5061bcc7c0c61598e7", GitTreeState:"clean"}
Server: &version.Version{SemVer:"v2.14.2", GitCommit:"a8b13cc5ab6a7dbef0a58
f5061bcc7c0c61598e7", GitTreeState:"clean"}

```

In the command, we are specifying that we want to use TLS and also passing the CA cert, the *helm client cert*, and the *helm client key*. Obviously, we don't want to specify these options each time we run a helm command. The helm client will look in the `.helm` directory of the user's home directory for TLS files when the `--tls` flag is used. The commands in Listing 8-10 will copy the files to the correct path with the required file names that the helm client expects.

Listing 8-10. Copying the helm TLS certs and keys to the `.helm` directory

```
copy ca.cert.pem "~\.helm\ca.pem"
copy helm.cert.pem "~\.helm\cert.pem"
copy helm.key.pem "~\.helm\key.pem"
```

With those files copied, it is only necessary to specify `--tls` when running helm client commands. If the `--tls` flag is not set on a command, then the helm client will appear to hang indefinitely.

The Tiller service is up and running and ready to accept Helm commands. Now it's time to build a chart to submit to Tiller.

Helm Charts

Helm charts are the fundamental structure that Helm uses to deploy applications. The chart is combined with configuration settings and submitted to Tiller. Tiller will synthesize the chart and settings into a release and provision that release on the Kubernetes cluster. A chart is a well-defined collection of files and directories. Some files and directories are required, such as the `Chart.yaml` file. Other files and directories are optional depending on the chart.

For the remainder of the chapter, we will be referencing an existing chart called **iaks** that deploys a voting application with a node.js front end and a redis backend.

Chart Contents

A standard file and folder structure for a Helm chart is shown in Listing 8-11. Required files are in **bold**.

Listing 8-11. Standard chart file and folder structure

ChartName (parent directory)

Chart.yaml: Contains information about the chart

LICENSE: Human readable license for the chart

README.md: Human readable markdown file

requirements.yaml: Listing of chart dependencies

values.yaml: Default configuration values for the chart

charts: Directory of charts which this chart depends on

templates: Directory of templates

templates/NOTES.txt: Human readable file with usage notes

Although the *charts* and *templates* directories are not required, they are reserved for use by Helm. Any other files added to the chart will be included, but don't necessarily have any special significance.

Listing 8-12 shows the structure of the **iaks** chart.

Listing 8-12. iaks chart structure

```
C:.\
|  .helmignore
|  Chart.yaml
|  values.yaml
|
|  └── templates
|      NOTES.txt
|      vote-back-deployment.yaml
|      vote-back-service.yaml
|      vote-front-deployment.yaml
|      vote-front-service.yaml
```

Chart.yaml

The `Chart.yaml` file defines values that Helm will use to interpret the chart. Listing 8-13 contains the potential file entries with required entries in **bold**.

Listing 8-13. Chart.yaml file entries

- **apiVersion:** Always set to v1 for now
- **name:** The name of the chart
- **version:** A SemVer 2 version for this chart
- **kubeVersion:** SemVer range of compatible Kubernetes versions
- **description:** Single sentence describing the chart and its purpose
- **keywords:** A list of keywords
- **home:** Project homepage URL
- **sources:** Source code URLs for the project
- **maintainers:** List of maintainers for the project
- **engine:** Name of the template engine (defaults to gotpl)
- **icon:** SVG or PNG image URL
- **appVersion:** Version number for the application
- **deprecated:** Boolean value indicating if the chart is deprecated
- **tillerVersion:** SemVer range of compatible Tiller versions

Listing 8-14 shows the contents of the Chart.yaml file for the **iaks** chart.

Listing 8-14. iaks Chart.yaml contents

```
apiVersion: v1
appVersion: "1.0"
description: A Helm chart for deploying the IAKS Voting App
name: iaks
version: 0.1.0
```

Note that the appVersion and the version entries are not the same. The version of chart may change without the version of the application changing.

Values.yaml

The values.yaml file defines the default settings to be used by the charts and templates in the project. All charts and templates have access to the settings defined in the top-level

values.yaml file. It is also possible to supply values.yaml files in the templates and charts subdirectories. The settings defined in the Values.yaml file can be overridden when the chart is deployed by either supplying an additional *yaml* file with values or by using the `--set` flag and supplying the settings at the command line.

Templates and charts will reference the settings defined in values.yaml by using namespace style reference notation. The namespace starts with the `.` symbolizing the top of the namespace, and then additional strings drill down through the values in the file. For instance, the values.yaml file might have an entry like the one in Listing 8-15.

Listing 8-15. Example values.yaml snippet

```
image:
  repository: iaks/azure-voting-app
  tag: v1-alpine
```

A template would reference the tag by using the notation `.Values.image.tag`.

In addition to the values supplied by files or the command line, there are also predefined values that are accessible to the charts and templates. These include information about the release, chart, and files. An exhaustive list of predefined values is available in Helm's documentation.

Listing 8-16 shows the contents of the values.yaml file for the **iaks** chart.

Listing 8-16. iaks values.yaml contents

```
# Default values for iaks.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

voteBack:
  replicaCount: 1
  appName: azure-vote-back
  image:
    repository: redis
    tag: 5.0.5
    pullPolicy: IfNotPresent
  ports:
    name: redis
    port: 6379
```

```

service:
  port: 6379
voteFront:
  replicaCount: 1
  appName: azure-vote-front
  image:
    repository: iaks/azure-voting-app
    tag: v1-alpine
    pullPolicy: IfNotPresent
    ports:
      name: http
      port: 80
  vote1Value: "Chocolate"
  vote2Value: "Peanut Butter"
  title: "IAKS Voting App"
  service:
    type: LoadBalancer
    port: 80
    targetPort: http
    name: http

```

License

The LICENSE file is written in plain text and is meant to lay out the software license covered by the application being installed in the chart. The license is not read by the Helm client. It is there for the user to parse and implicitly agree to, should they choose to use the chart.

The **iaks** chart does not have a license file.

README.md

The README.md is written in markdown and is meant to assist the user in properly using the chart. At a minimum, it should describe what the chart does, prerequisites for running the chart, and the settings included in the values.yaml file and what the defaults are set to. Any other useful information for deploying the chart should also be included in this document. The README file will be displayed if the chart is published on certain repositories.

If there are some quick getting started notes that should be displayed to the user after deployment, they can be included in a `NOTES.txt` file in the *templates* directory. The `NOTES.txt` file will be evaluated as a template and then displayed on the command line.

Listing 8-17 shows the truncated text from the **iaks** chart `README.md`.

Listing 8-17. iaks README.md contents

```
# IAKS Voting App

Example application for the Helm chapter of the Introducing Azure
Kubernetes Service book.

## Install Chart

To install the IAKS Chart into your Kubernetes cluster :

Clone the chart down to your local file system.

```bash
helm install --namespace "iaks" --name "iaksv1" ./iaks
```

...
```

Requirements.yaml

The chart being defined may use other charts as part of its deployment. Those charts can be manually copied to the *charts* directory and kept there. For teams that need strict control over the version and contents of the dependent charts, it may make sense to copy them directly into the *charts* directory. However, this makes the charts static and requires that they be updated manually.

Charts that are included in the `requirements.yaml` file are pulled dynamically when the `helm dependency update` command is run. The resulting pulled charts are stored as zipped-up charts – aka chart archives – in the *charts* directory. Within the `requirements.yaml` file, each chart is listed out in the dependencies as shown in Listing 8-18.

Listing 8-18. Example requirements.yaml entry

```
dependencies:
  - name: chart_name
    version: 1.2.3
    repository: http://mycharts.com/charts
```

When a new version is available for use, the version number can be updated in the dependencies, and the `helm dependency update` command is run again. This will pull the new version of the chart and store it in the *charts* directory.

There are a few additional optional fields that can be added to a dependency listing. These optional fields are for more advanced deployment cases, and you likely will not need them in your initial attempts with Helm.

If an application requires multiple copies of the same chart, or different versions of the same chart, the `alias` field that can be included. The `alias` field will alter the name of the downloaded chart to match the `alias` value.

The `condition` field specifies a comma-separated list of *yaml* entities in the top parent's `values.yaml`, each resolving to a Boolean value. Setting the value to `false` will stop the chart from being included as a dependency in the chart.

The `tags` field is a list of labels associated with the chart. In the top parent's `values`, each tag can be enabled or disabled using the tag and a Boolean value. If any tag for a dependent chart is enabled, it will be included in the dependencies.

The **iaaks** chart does not have a requirements.yaml file.

Charts Directory

The *charts* directory will contain the charts to be included as dependencies for the parent chart. As mentioned in the Requirements.yaml section, the charts can be populated by copying the files manually, or dynamically by using the requirements.yaml file. The charts contained within the *charts* directory can either be an unpacked chart or a chart archive. Each chart should be its own separate file if using an archive or its own directory if the chart is unpacked.

The **iaaks** chart does not use any other charts as dependencies.

Templates Directory

The *templates* directory contains helm chart templates. When Helm is rendering charts for a release, it evaluates all files contained within the *templates* folder. The template

files use the Go template language for the majority of their functions. Helm also borrows some functions from the Sprig library and includes some specialized functions specific to Helm.

The files in the *template* directory are used to create viable Kubernetes definition files in yaml. The template language is used to manipulate the file content to produce valid yaml files dynamically, incorporating the values supplied by the `values.yaml` file or by the user when `helm install` is run.

The **iaiks** chart has templates for the front-end and backend deployments and services, as well as a `NOTES.txt`. We will examine these files in more detail in the section dealing with template functions.

Chart Repositories

Helm works with charts stored in repositories. The `helm` client has a subset of commands for dealing with both the locally stored charts and remote repositories. Let's start by viewing the list of chart repositories available from a default install of Helm as shown in Listing 8-19.

Listing 8-19. Listing of Helm repositories

```
$ helm repo list
NAME                URL
stable              https://kubernetes-charts.storage.googleapis.com
local               http://127.0.0.1:8879/charts
incubator           http://storage.googleapis.com/kubernetes-charts-incubator
```

As you can see, Helm starts with the *stable* and *incubator* charts from the official Helm repository. It has also created a local repository listening on port 8879. By default, the *local* repository has no charts. We can confirm this by running the command in Listing 8-20.

Listing 8-20. Contents of the local repository

```
$ helm search /local
No results found
```

Running the same command against the *stable* repository as shown in Listing 8-21 results in about 278 charts!

Listing 8-21. Contents of the stable repository

```
$ helm search stable/
```

| NAME | CHART VERSION | APP VERSION |
|--|---------------|-------------|
| DESCRIPTION | | |
| stable/acs-engine-autoscaler | 2.2.2 | 2.1.1 |
| DEPRECATED Scales worker nodes within agent pools | | |
| stable/aerospike | 0.2.7 | v4.5.0.5 |
| A Helm chart for Aerospike in Kubernetes | | |
| stable/airflow | 2.8.2 | 1.10.2 |
| Airflow is a platform to programmatically author, schedul... | | |

The list of charts is cached locally. To update the contents of a repository, the command `helm repo update` can be executed. The process of packaging and pushing a chart to a repository will be covered later in this chapter.

Deployment Process

Deploying a Helm chart as a running application on a Kubernetes cluster is performed through the command `helm install`. The install command allows values to be submitted in the form of an additional `yaml` file or using the `--set` flag in the command. The settings in the values submitted at runtime are merged with the `values.yaml` file in the chart to produce an updated `values.yaml` file containing the final configuration data that will be used during the installation.

For instance, suppose `helm install -f myvalues.yaml ./mychart` is run. The contents of the existing `values.yaml` file in the chart are shown in Listing 8-22.

Listing 8-22. `values.yaml` file with default configuration

```
voteBack:
  replicaCount: 1
  appName: azure-vote-back
  image:
    repository: redis
    tag: 5.0.5
    pullPolicy: IfNotPresent
```

```
ports:
  name: redis
  port: 6379
```

The myvalues.yaml file has the contents shown in Listing 8-23.

Listing 8-23. Contents of myvalues.yaml

```
voteBack:
  replicaCount: 2
  label: mylabel
```

The two files will be merged together with the contents of the myvalues.yaml file taking precedence over the contents of the values.yaml file. Listing 8-24 has the contents of the resulting file.

Listing 8-24. Contents of the new values.yaml file

```
voteBack:
  replicaCount: 2
  label: mylabel
  appName: azure-vote-back
  image:
    repository: redis
    tag: 5.0.5
    pullPolicy: IfNotPresent
  ports:
    name: redis
    port: 6379
```

Tiller accepts the chart and values and creates a set of valid Kubernetes definitions, which are submitted to the cluster through the Kubernetes API. The submitted deployments are called a *release* in Helm parlance.

A Helm release contains several pieces of information describing the release including the following:

- **AppVersion:** Version number based on the AppVersion setting in the Chart.yaml

- **Chart:** The chart name with the version number appended from the Chart.yaml
- **Name:** The name given to the release during the install
- **Namespace:** The namespace in which the release was installed
- **Revision:** Starts at 1 for the first install and increments each time an update or rollback is executed
- **Status:** The current status of the release, typically DEPLOYED for a release that has completed installation
- **Updated:** The last time some aspect of the release changed

The current list of releases can be retrieved by running `helm list` or the shortened version `helm ls`. This command will only show releases with a status of DEPLOYED by default. The flag `--all` can be added to see all releases with any status.

Helm releases can be updated by using one of the following commands:

`helm delete:` Deletes the release from the cluster and changes the status of the release to DELETED

`helm upgrade:` Upgrades the current release with the submitted values

`helm rollback:` Reverts the current release to the submitted revision number

Creating a Helm Chart

There are many excellent charts already available as a starting point for creating you own Helm charts. Helm also includes the tools to begin a chart from a predefined template.

Helm Create

Helm makes it simple to set up the file and directory structure for a new chart. The `helm create` command will create a new directory at the path specified, and within the directory it will create several of the required and optional files for a Helm chart. The command `helm create iaks-chart` will create a new Helm chart called iaks-chart.

The command creates a directory called iaks-chart in the current path and populates it with the files and directories shown in Listing [8-25](#).

Listing 8-25. Contents of the new iaks-chart Helm chart

```

C:.\
|   .helmignore
|   Chart.yaml
|   values.yaml
|
|___charts
|___templates
|   deployment.yaml
|   ingress.yaml
|   NOTES.txt
|   service.yaml
|   _helpers.tpl
|___tests
|   test-connection.yaml

```

The files contain a basic *nginx* application, including an ingress controller, a service for the *nginx* pods, and a deployment of the *nginx* pods. It also includes a test to validate that the deployment of the *nginx* application is successful.

Template Functions

Templates in the helm chart use a combination of the Go template language functions, Sprig functions, and custom functions from Helm. These functions take the contents of the template file and the values submitted during installation and render out valid Kubernetes definitions.

The Go template language is an advanced topic beyond the scope of this humble chapter, but here are some pointers to get started with.

- Any values in the template file that should be evaluated by the template engine will start and end with doubly curly braces `{{ }}`.
- Values from the `value.yaml` file are referenced using namespace path notation, for example, `.Values.dockerTag`.

- Helm has a primer on getting started with template development. You can learn more at this link (https://helm.sh/docs/chart_template_guide/#getting-started-with-a-chart-template).

Looking at an example will help illustrate how the template language is used. Listing 8-26 shows the contents of the `vote-back-service.yaml` file in the **iaks** chart.

Listing 8-26. Contents of the new `vote-back-service.yaml` file

```
apiVersion: v1
kind: Service
metadata:
  name: {{ .Values.voteBack.appName }}
spec:
  ports:
    - port: {{ .Values.voteBack.service.port }}
  selector:
    app: {{ .Values.voteBack.appName }}
```

The `{{ }}` invokes the template engine to evaluate the contents within the double curly braces. In the listing, `metadata.name` is evaluating the expression `.Values.voteBack.appName`. That refers to the setting in the submitted `values.yaml` file for the release. The default for that setting is *azure-vote-back*, and so the template engine will render that portion of the file as shown in Listing 8-27.

Listing 8-27. Rendered value for `metadata.name`

```
metadata:
  name: azure-vote-back
```

Functions can be added to the evaluation in the pipeline to manipulate the value. For instance, suppose that the name needs to be in all lowercase. Listing 8-28 shows how the value can be piped to `lower` to manipulate the text.

Listing 8-28. Using the `lower` function on a value

```
metadata:
  name: {{ .Values.voteBack.appName | lower }}
```

This is a simple example of using template functions. More complicated evaluations are possible depending on the needs of the application.

Chart Tests

Helm doesn't know what the application defined in the chart is supposed to do. If all components of the release are created successfully, then Helm considers the release a success. Chart tests provide a way for the user to validate that the application components are functioning properly. They can also be used in an automation context to validate a release in the pipeline.

Chart tests are template files that reside in the *templates* directory, or more often in a *tests* subdirectory within the *templates* directory. Each test is a *pod* definition. The pod should run some actions and then exit with a value, 0 being considered success and any other value being considered a failure. The pod definitions can be part of a single *yaml* file or broken up into multiple *yaml* files, one per test.

Helm has two test hooks that indicate whether the test should be successful or not, `test-success` and `test-failure`. These hooks are added into the annotations of the pod as shown in Listing 8-29.

Listing 8-29. Helm test hooks

```
metadata:
  annotations:
    "helm.sh/hook": test-success
```

The annotations are what indicates to Helm that these pod definitions are tests, and not part of the application. Tests are invoked by running `helm test` with the release name to be tested.

Packaging a Chart

Once a chart is ready for usage, it can be packaged and uploaded to a chart repository. Packaging a chart creates a versioned archive of that chart. The contents of the chart are zipped up into a `tgz` file. A chart can be packaged by using the `helm package` command and pointing the command to the directory that contains the chart.

Listing 8-30 shows the process of packaging the **iaxs** chart.

Listing 8-30. Packaging the iaks chart

```
$ helm package .\iaks\
Successfully packaged chart and saved it to: C:\gh\Introducing-Azure-
Kubernetes-Service\Helm\aks\iaks-0.1.0.tgz
```

The name of the file is a combination of the chart name and the version of the chart. Both values are found in the `Chart.yaml` file.

The process for uploading the chart archive to a repository will depend on the repository type. The `index.yaml` file for a repository must be updated when the chart is uploaded so that it will be included in repository searches and listings. The Azure Container Registry (ACR) can host packaged helm charts. The first step is to log into an existing ACR repository and add it as a repository for Helm as shown in Listing 8-31.

Listing 8-31. Adding the ACR repo to Helm

```
$ az acr login --name iaks0
$ az acr helm repo add
"iaks0" has been added to your repositories

$ helm repo list
NAME          URL
stable        https://kubernetes-charts.storage.googleapis.com
local         http://127.0.0.1:8879/charts
incubator      http://storage.googleapis.com/kubernetes-charts-incubator
iaks0         https://iaks0.azurecr.io/helm/v1/repo
```

Once the repository has been added to Helm, it is a simple matter of pushing a chart archive to the ACR repository as seen in Listing 8-32. Then the local index of the repository must be updated so that it will show up in the search results.

Listing 8-32. Pushing a package to the ACR repo

```
$ az acr helm push .\iaks-0.1.0.tgz
{
  "saved": true
}
```

```
$helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "incubator" chart repository
...Successfully got an update from the "iaks0" chart repository
...Successfully got an update from the "stable" chart repository
Update Complete.

$ helm search iaks0
NAME                CHART VERSION  APP VERSION     DESCRIPTION
iaks0/iaks          0.1.0          1.0             A Helm chart for deploying
                    the IAKS Voting App
```

The chart archive is now available for consumption through the ACR repository for any other users that have access.

Deploying a Helm Chart

A Helm chart is deployed using the `helm install` command. The command will take the chart, default values, and values submitted in the command and send them to Tiller. Tiller will synthesize them into a release and instantiate that release on the Kubernetes cluster.

Helm Install

Listing 8-33 shows the process of installing the **iaks** chart in the *default* namespace of an AKS cluster.

Listing 8-33. Installing the iaks chart

```
$ helm install --tls --name iaksv1 ./iaks
NAME:    iaksv1
LAST DEPLOYED: Wed Jul 17 11:40:30 2019
NAMESPACE: default
STATUS:  DEPLOYED
```

RESOURCES:

==> v1/Pod(related)

| NAME | READY | STATUS | RESTARTS | AGE |
|----------------------------------|-------|-------------------|----------|-----|
| azure-vote-back-78d97d47df-2hjbr | 0/1 | ContainerCreating | 0 | 0s |
| azure-vote-front-948444d79-m2ms2 | 0/1 | ContainerCreating | 0 | 0s |

==> v1/Service

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|------------------|--------------|--------------|-------------|--------------|-----|
| azure-vote-back | ClusterIP | 10.0.40.25 | <none> | 6379/TCP | 1s |
| azure-vote-front | LoadBalancer | 10.0.212.114 | <pending> | 80:30829/TCP | 0s |

==> v1beta1/Deployment

| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|------------------|-------|------------|-----------|-----|
| azure-vote-back | 0/1 | 1 | 0 | 0s |
| azure-vote-front | 0/1 | 1 | 0 | 0s |

NOTES:

1. Get the application URL by running these commands:

NOTE: It may take a few minutes for the LoadBalancer IP to be available.

You can watch the status of by running 'kubectl get --namespace default svc -w azure-vote-front'

```
export SERVICE_IP=$(kubectl get svc --namespace default azure-vote-front
-o jsonpath='{.status.loadBalancer.ingress[0].ip}')
echo http://$SERVICE_IP:80
```

The installation process displays the resources being created and also prints out the rendered NOTES.txt file found in the templates folder. The NOTES.txt file is evaluated by the template engine and can provide simple directions for getting started with the application.

Figure 8-1 shows the web page that is available once the external IP of the load balancer finishes provisioning.

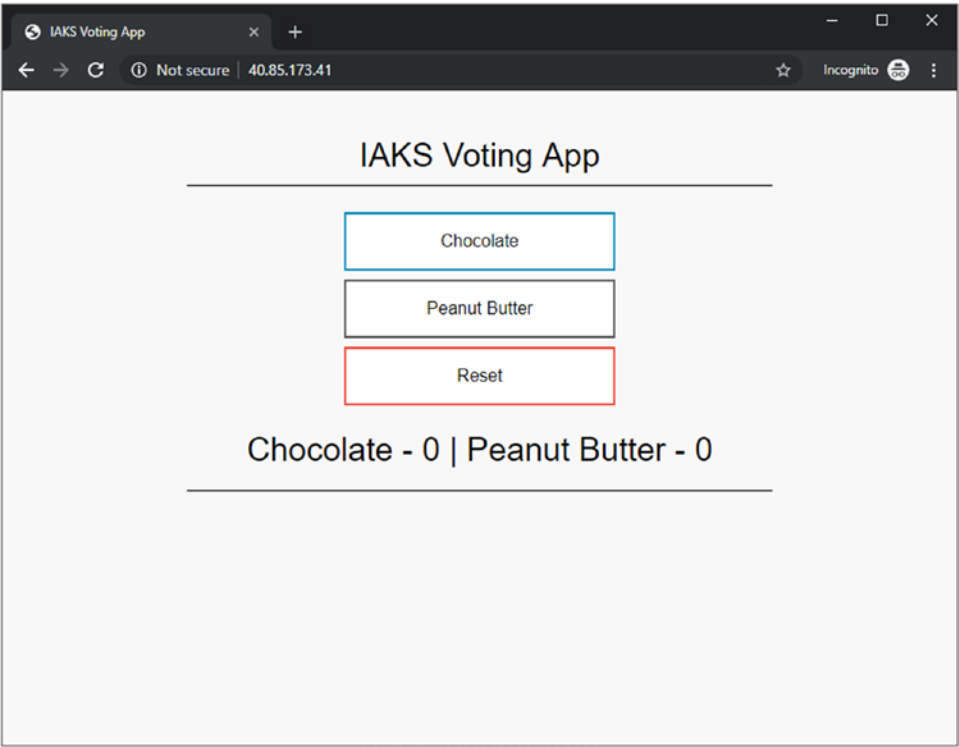


Figure 8-1. Voting App web page

Helm Status

The command `helm status` will get the current status of a release. Listing 8-34 shows the current status of the *iaksv1* release.

Listing 8-34. Status of the *iaksv1* release

```
$ helm status --tls iaksv1
LAST DEPLOYED: Wed Jul 17 11:40:30 2019
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Pod(related)
NAME                                READY  STATUS   RESTARTS  AGE
azure-vote-back-78d97d47df-2hjbr  1/1    Running  0          4m36s
azure-vote-front-948444d79-m2ms2  1/1    Running  0          4m36s
```



```
==> v1/Service
```

| NAME | TYPE | CLUSTER-IP | EXTERNAL-IP | PORT(S) | AGE |
|------------------|--------------|--------------|--------------|--------------|-------|
| azure-vote-back | ClusterIP | 10.0.40.25 | <none> | 6379/TCP | 4m37s |
| azure-vote-front | LoadBalancer | 10.0.212.114 | 40.85.173.41 | 80:30829/TCP | 4m36s |

```
==> v1beta1/Deployment
```

| NAME | READY | UP-TO-DATE | AVAILABLE | AGE |
|------------------|-------|------------|-----------|-------|
| azure-vote-back | 1/1 | 1 | 1 | 4m36s |
| azure-vote-front | 1/1 | 1 | 1 | 4m36s |

NOTES:

1. Get the application URL by running these commands:

NOTE: It may take a few minutes for the LoadBalancer IP to be available.

You can watch the status of by running 'kubectl get --namespace default svc -w azure-vote-front'

```
export SERVICE_IP=$(kubectl get svc --namespace default azure-vote-front
-o jsonpath='{.status.loadBalancer.ingress[0].ip}')
echo http://$SERVICE_IP:80
```

The status gives essentially the same information as the initial installation, including the NOTES.txt section.

Updating a Release

Over the lifetime of the release, it may be necessary to update the chart, application, or settings. The command `helm upgrade` is used to perform such an update. Listing 8-35 shows the process of updating the *iaksv1* release with new values for the voting buttons.

Listing 8-35. Upgrade of the *iaksv1* release

```
$ helm upgrade --tls --set voteFront.vote1Value=Cats,voteFront.
vote2Value=Dogs iaksv1 ./iaks
Release "iaksv1" has been upgraded.
LAST DEPLOYED: Wed Jul 17 11:50:14 2019
```

The output has been truncated for brevity. By running `helm ls` as seen in Listing 8-36, we can retrieve the status of the release and see that the revision number has incremented.

Listing 8-36. Listing of the iaksv1 release

```
$ helm ls --tls iaksv1
```

| NAME | REVISION | UPDATED | STATUS |
|------------|-------------|--------------------------|----------|
| CHART | APP VERSION | NAMESPACE | |
| iaksv1 | 2 | Wed Jul 17 11:50:14 2019 | DEPLOYED |
| iaks-0.1.0 | 1.0 | default | |

Viewing the updated web page in Figure 8-2 shows the updated content of the buttons.

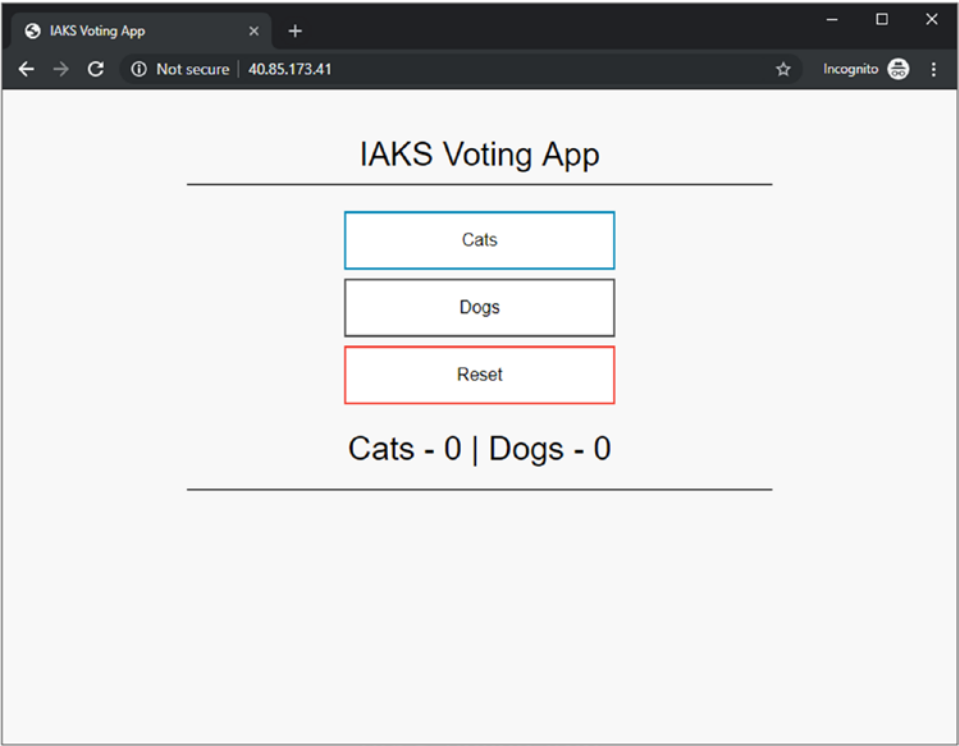


Figure 8-2. Voting App web page updated

It is also possible to roll back to a previous version of the release by using the `helm rollback` command. Listing 8-37 shows the output of rolling back a release.

Listing 8-37. Rollback of the iaksv1 release

```
$ helm rollback --tls iaksv1 1
```

Rollback was a success.

```
$ helm ls --tls iaksv1
```

| NAME | REVISION | UPDATED | STATUS |
|--------------|-------------|--------------------------|----------|
| CHART | APP VERSION | NAMESPACE | |
| iaksv1 | 3 | Wed Jul 17 11:54:45 2019 | DEPLOYED |
| iaksv1-0.1.0 | 1.0 | default | |

By looking at the revision number of the release, we can see that the revision is now at 3 and not 1. The revision number will always increment when a release is altered, regardless of whether the release is being upgraded or rolled back.

Looking at the web site again in Figure 8-3, we can see that the voting buttons have reverted to their previous values.

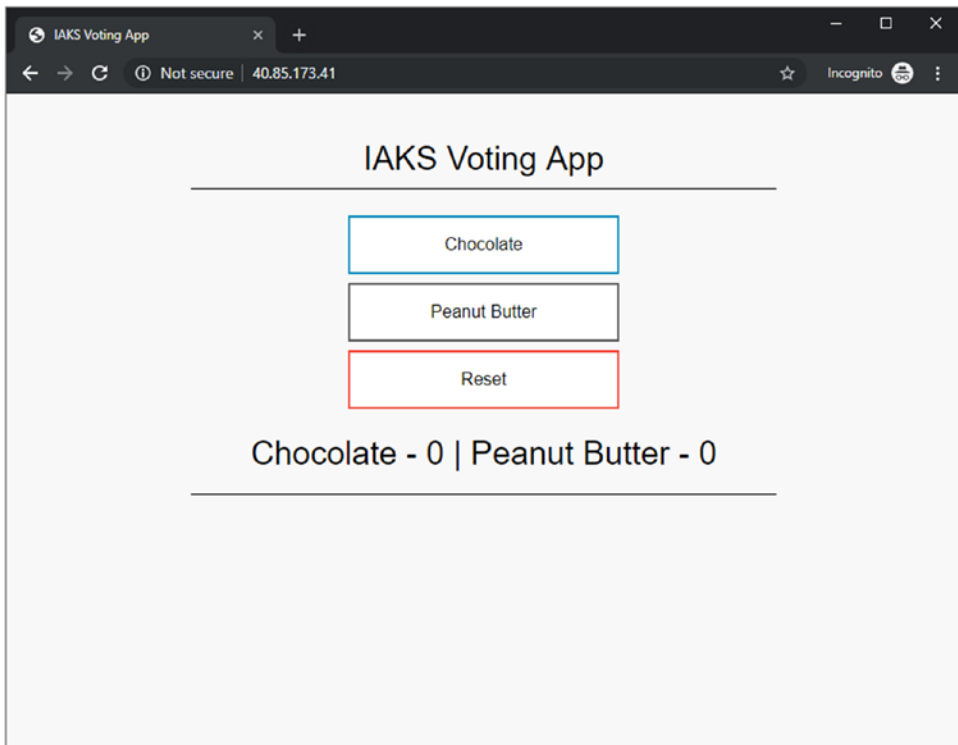


Figure 8-3. Voting App web page rollback

Removing a Release

There comes a time in every release’s life cycle when it is no longer needed. The command for deleting a release is `helm delete`. Listing 8-38 shows the output of deleting `iaksv1`.

Listing 8-38. Delete of the `iaksv1` release

```
$ helm delete --tls iaksv1
release "iaksv1" deleted
```

The resources in the Kubernetes cluster will be removed, but the release is not entirely gone. Listing 8-39 shows all of the releases including deleted ones.

Listing 8-39. Listing of all releases

```
$ helm ls --tls --all
NAME      REVISION      UPDATED              STATUS
CHART     APP VERSION   NAMESPACE
iaksv1    3             Wed Jul 17 11:54:45 2019  DELETED
iaks-0.1.0 1.0           default
```

It is possible to use `helm rollback` to undo the deletion of the release from the Kubernetes cluster.

Note Helm will not recover the exact pods and volumes from the deleted release. Helm will only redeploy the version of the release specified in the rollback command. Any data that was not persisted through other means will be lost during deletion of a release.

To remove the release from Tiller permanently, the `--purge` flag must be used as shown in Listing 8-40.

Listing 8-40. Purge of the `iaksv1` release

```
$ helm delete --tls iaksv1 --purge
release "iaksv1" deleted
```

CI/CD Integrations

Continuous integration and continuous delivery – often abbreviated as CI/CD – are the practice of creating an automated pipeline that moves code from a commit by a developer to deployment into one or more environments. Helm can be incorporated into the CI/CD process as a tool for deploying new releases to a Kubernetes cluster or updating existing releases.

Automating Deployments

When code is committed to a repository, it may kick off a pipeline of events. Using our **iaks** chart example, the chart makes use of the container image `iaks/azure-voting-app`. There could be a pipeline that is triggered when a developer commits a new version of the Dockerfile that builds that image. Once the image has been updated and passes the necessary tests in the pipeline, the **iaks** Helm chart can be tested with the new image version. Assuming all tests pass, the **iaks** Helm chart could be updated to use the new image as the default tag in the `values.yaml` file.

Figure 8-4 demonstrates a potential pipeline.

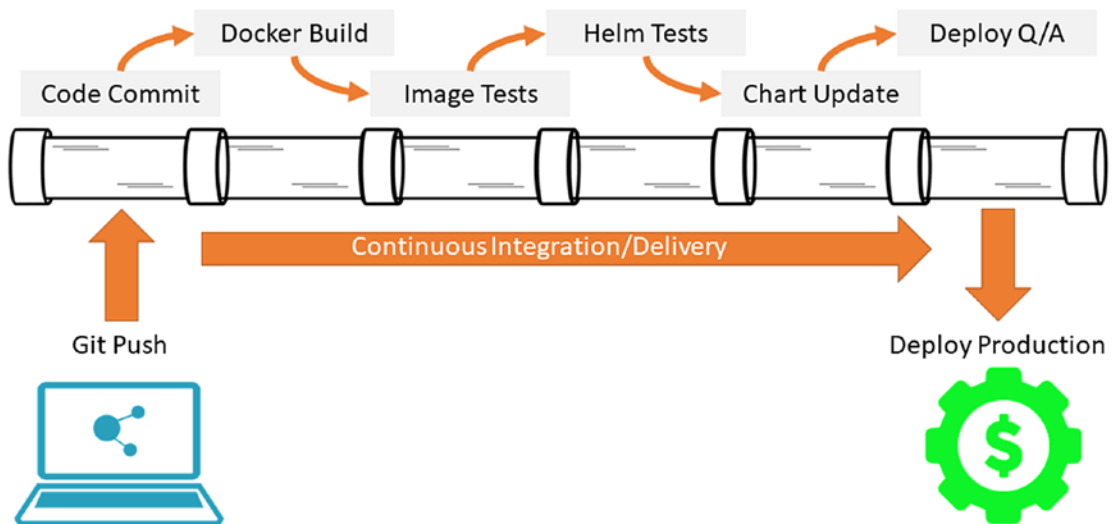


Figure 8-4. Potential CI/CD pipeline

Testing Helm Charts

Helm charts can be tested in several ways. A test could be run to validate that the chart is syntactically correct using the `helm lint` command. Listing 8-41 shows the output of running `helm lint` against the **iaks** chart.

Listing 8-41. Linting the iaks chart

```
$ helm lint .\iaks\
==> Linting .\iaks\
[INFO] Chart.yaml: icon is recommended

1 chart(s) linted, no failures
```

The output is meant to be human readable, but it can be machine parsed to see if there are any errors or warnings prior to moving the chart to the next stage of testing.

The next testing step could be locally rendering the templates with several different possible values and validating that the generated Kubernetes definitions are valid.

Listing 8-42 shows the output of running `helm template` against the templates in the **iaks** chart. The output has been truncated for brevity.

Listing 8-42. Helm template rendering

```
helm template .\iaks\
---
# Source: iaks/templates/vote-back-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: azure-vote-back
spec:
  ports:
    - port: 6379
  selector:
    app: azure-vote-back
---
```

The output from this command could be piped into `kubectl` to validate the content.

```
$ helm template .\iaks | kubectl apply --dry-run --validate -f -
```

The output could be scanned for errors to determine if any of the Kubernetes definitions are invalid.

Once a chart has been validated according to requirements, it can be further tested to validate that the application is functioning properly. Some of the tests can be embedded in the chart using the Helm test functionality mentioned earlier in this chapter. Most automation pipelines will have additional tests to perform on the application that are beyond the basic testing performed by Helm. The tests included in the Helm chart are meant to test basic functionality, and not more advanced scenarios.

There is an open source project related to Helm that is specifically centered around testing Helm charts. The project is called *Chart Testing* and can be found on GitHub (<https://github.com/helm/chart-testing>). The *Chart Testing* software is capable of performing the linting, template validation, and even running the Helm tests contained in the chart against a release.

Unattended Helm Chart Installs

Installing Helm charts on a Kubernetes cluster can be performed in an automated fashion. The values used to customize a given release can be generated as artifacts in a CI/CD pipeline and then passed to the Helm client either as a file or through the `--set` flag.

The same process can be used for performing upgrades of existing releases, including running the Helm tests afterwards and preparing to run a rollback operation if the tests fail or other issues are identified. The pipeline should record the current revision number of the Helm release prior to upgrade to assist in the rollback process should it be deemed necessary.

Integrating Helm with Azure DevOps

Azure DevOps (ADO) has a number of services that integrate with Helm to assist in automating a development pipeline. *Repos* in ADO can serve as the source control for Helm charts as they are developed. *Artifacts* can be used to store the generated values for a Helm release. *Pipelines* can be used to set up a CI/CD pipeline that uses Helm to deploy releases on an AKS cluster.

The *Pipelines* portion of Azure DevOps includes multiple tasks that enable the use of Helm with the AKS service. The *Helm tool installer* task will install the Helm binary on the agent machine running the job. The *Package and deploy Helm charts* task allows the running of basically any Helm client command, including options for using TLS authentication. The task targets AKS clusters, making the installation of a Helm chart on an AKS cluster relatively straightforward. Listing 8-43 shows an example *yaml* Build pipeline in Azure DevOps.

Listing 8-43. ADO Pipeline definition

```
# Helm deployment pipeline

trigger:
- master

pool:
  vmImage: 'ubuntu-latest'

steps:
- task: HelmInstaller@1
  inputs:
    helmVersionToInstall: 'latest'
- task: DownloadSecureFile@1
  inputs:
    secureFile: 'ca.cert.pem'
- task: DownloadSecureFile@1
  inputs:
    secureFile: 'helm.cert.pem'
- task: DownloadSecureFile@1
  inputs:
    secureFile: 'helm.key.pem'
- task: HelmDeploy@0
  inputs:
    connectionType: 'Azure Resource Manager'
    azureSubscription: 'MAS(4d8e572a-3214-40e9-a26f-8f71ecd24e0d)'
    azureResourceGroup: 'iaks'
```



```
kubernetesCluster: 'iaks1'  
namespace: 'iaks'  
command: 'install'  
chartType: 'FilePath'  
chartPath: 'Helm/aks/iaks'  
releaseName: '${releaseName}'  
enableTls: true  
caCert: 'ca.cert.pem'  
certificate: 'helm.cert.pem'  
privatekey: 'helm.key.pem'
```

The pipeline pulls the TLS files from the Secure Files section of the build library, installs Helm on the agent machine, and then runs `helm install` on the named AKS cluster using the chart found at the path `Helm/aks/iaks`.

Summary

Helm is a tool to assist with the management of applications on Kubernetes. By providing charts, Helm increases the reusability of applications and allows for customization of an application to different environments through well-defined configuration values. Helm is CLI based and easily fits into existing automation pipelines and source control. This chapter serves as an introduction to Helm and a guide for getting started with Helm on AKS.

In this chapter, you learned about what Helm is and how it provides benefits over `kubectl` and traditional application deployment on Kubernetes. We reviewed the process of preparing your AKS cluster to use Helm for application management. Then, we went over the Helm chart structure and what is included in a functioning Helm chart. Armed with a functional chart, you learned the process for deploying and maintaining Helm releases on a Kubernetes cluster. Lastly, we briefly went over how Helm fits into the world of automation and CI/CD.

CHAPTER 9

CI/CD with Azure Kubernetes Service

Software development has been steadily moving toward a model of Continuous Integration and Continuous Delivery. A key enabler along the way has been the introduction of cloud-native applications. The Cloud Native Computing Foundation defines cloud-native as

An open source software stack to deploy applications as microservices, packaging each part into its own container, and dynamically orchestrating those containers to optimize resource utilization.

Hopefully, some of those terms are starting to look a bit familiar to you. Each part of the application is packaged up as a container, most likely using Docker images. The microservices are deployed and maintained by an orchestrator, such as Kubernetes. As we'll see in this chapter, the process of continuously integrating and delivering software can be more efficient if that software is packaged using containers and deployed in a way that is consistent and repeatable. Kubernetes in tandem with a CI/CD pipeline tool, like Azure DevOps Pipelines, empowers developers to iterate faster and create more reliable applications.

In this chapter, we are going to break apart the mysterious CI/CD abbreviation and dissect the components of both CI and CD. You'll see what it means to continually integrate software and how that is accomplished using a build pipeline. Then we'll look at continuous delivery and deployment and how to accomplish each using a release pipeline. Finally, we'll review a few best practices when it comes to using CI/CD with the Azure Kubernetes Service.

We are going to be using the IAKS Voting Application from previous chapters to help you apply the abstract concepts of integration, delivery, and deployment to a real-world scenario. The voting application is a container-based application composed of a web front-end running node.js and a storage backend running Redis as shown in Figure 9-1.

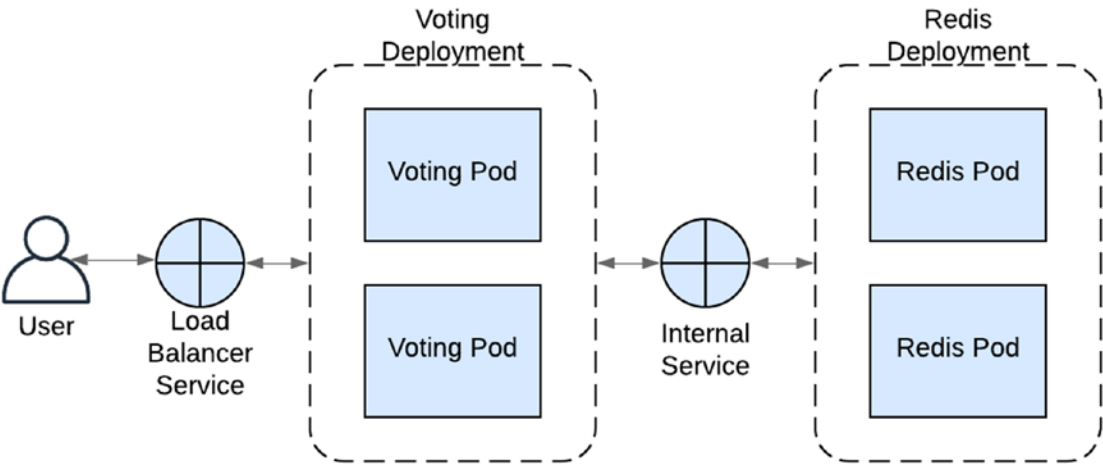


Figure 9-1. Voting App

The project has manifest files for deployment to Kubernetes that have been packaged up in a Helm chart. We are going to follow the process of updating the application and rolling that update out to a development environment and then to Production.

CI/CD Overview

You’ve probably seen the abbreviation CI/CD before and maybe wondered what it means. It’s a weird-looking abbreviation, and marketing folks like to sprinkle it onto products liberally as if it were some magic incantation. But CI/CD does actually stand for something, or more specifically it stands for two things. The CI stands for Continuous Integration, which is the idea that as developers write code, they should be checking their code into a shared mainline several times a day. We’ll expand more on that thought in a moment.

The CD stands for either Continuous Delivery or Continuous Deployment. The primary idea is that an up-to-date build of the software should be available and ready to deploy to Production at any given time. If the build is ready, but not running in

Production, then we can say it has been delivered. If there is an automated process than moves a delivered build into Production, then we can say it has been deployed. Both options conveniently abbreviate to CD, and so the common abbreviation CI/CD works in either case. Which “D” – Deployment or Delivery – is being used can be inferred through context.

Continuous Integration

Continuous Integration (CI) is a software development practice that has a few key principles as illustrated in Figure 9-2. There should be a common shared code base that developers are working from. Developers should check out the most recent version of code, make their changes, and then merge their updated code back into the mainline. Before check-in, developers will get the latest version of the mainline and integrate any changes into their local version. By committing to the mainline often, developers will never be too far off from the mainline version of the code, and therefore the integration process will be simpler and less likely to require refactoring. The constant process of integrating a local code copy with the mainline copy is what is known as Continuous Integration.

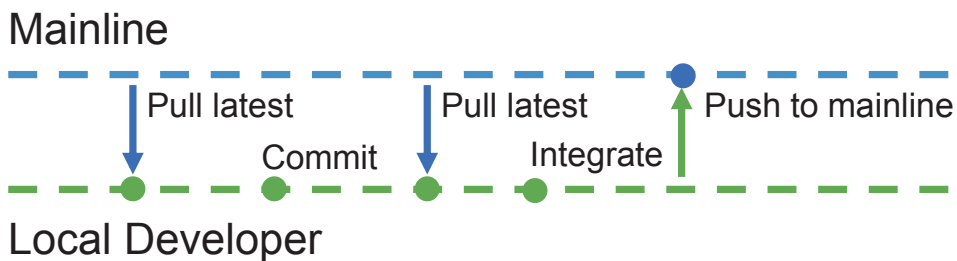


Figure 9-2. *Continuous Integration process*

While CI is a good concept in theory, it wasn’t until supporting toolsets arrived that the concept was turned into a reality. The primary components that enabled the process were Source Control Management (SCM) software, build servers, and automated pipelines. For our example of the IAKS Voting Application, we are going to use Azure DevOps to provide these CI components. Azure Repos will provide a git-based SCM. Azure Pipelines will provide the automation and build servers. Azure Container Registry will be the target to store completed build artifacts.

Shared Repository

When developers are collaborating on code, there are often several copies of the code floating around. Each developer will have a local copy on their workstation that they will use to develop a new feature or function. There is also a shared repository of code, usually in a SCM service, that developers will commit their changes to and get the most recent version of the code from. The most common SCM in use today is Git, although there are others such as Subversion, CVS, and TFS.

Note A full discussion of Git and version control for software development is way outside the scope of this little chapter. We will assume that you are somewhat familiar with the idea of source control and branches. If that seems totally alien to you, then we recommend checking out the hello-world activity on GitHub (<https://guides.github.com/activities/hello-world/>) as a primer.

The IAKS Voting Application is using Azure Repos to host the shared copy of its code as seen in Figure 9-3. The source control mechanism being used to manage the code is Git. The application has a *master* branch for production usage and a *feature* branch used to develop new features. Once a new feature has been tested and approved, the branch will be merged into *master*.

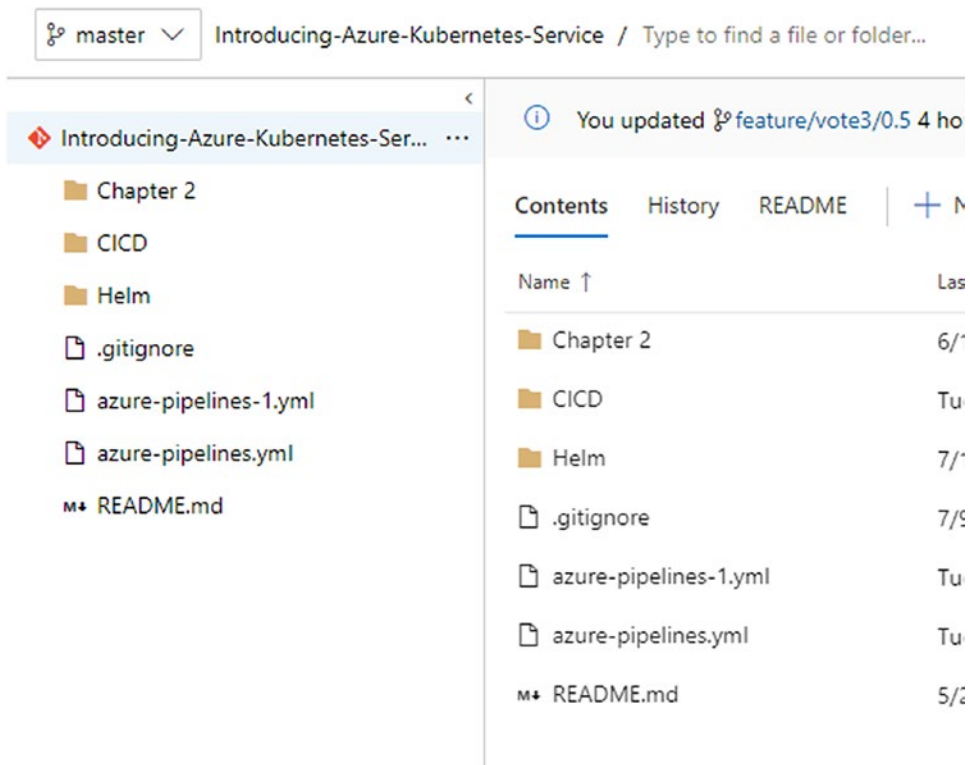


Figure 9-3. Voting App git repository

Build Pipeline

The build pipeline enables the integration of committed code with the larger code base. The build pipeline is the portion of Continuous Integration that occurs after a developer performs a push to the mainline. There are many different products that will run a build pipeline for you. While the terminology may change slightly from product to product, the core concepts are consistent.

A build pipeline is composed of a series of steps that are carried out by build agents in a serial or parallel fashion. The output of a build pipeline is a set of artifacts that represent the functional application. A release pipeline should be able to take those artifacts and deploy the application into target environments.

The build agents are usually virtual machines with agent software running and listening for new requests to come in. When the build pipeline is ready to execute one of the tasks in a step, it will look for an available build agent that meets the task requirements and have that build agent execute the task. The results of the task will be

reported back to the build pipeline, whether the task is successful or not. At that point, the pipeline may continue or fail depending on the task settings.

The IAKS Voting Application will be using Azure Build Pipelines shown in Figure 9-4 as part of its CI process. Azure Build Pipelines express their configuration using `yaml`. The pipeline definition can live in source control along with the rest of the application, ensuring that changes to the pipeline are tracked and versioned along with the rest of the code.

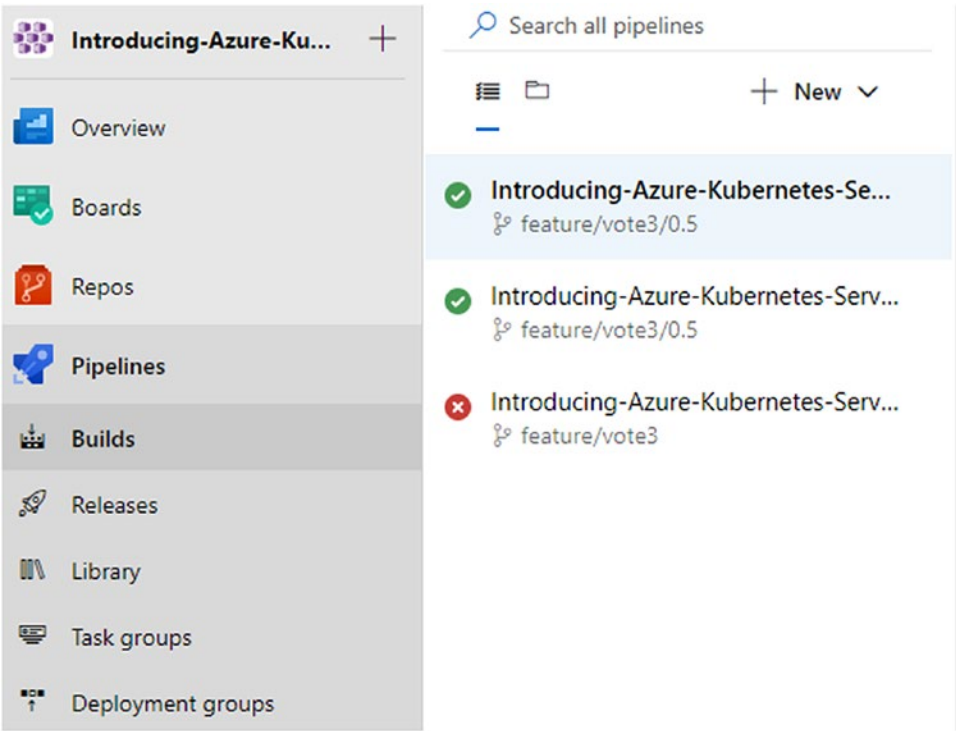


Figure 9-4. Continuous Integration build pipeline

Triggers

The trigger in a pipeline defines the conditions under which the pipeline will execute. A trigger is often scoped to a specific branch, tag, or feature so that the steps in the pipeline will only execute when something in-scope is committed. For a large code base, scoping the trigger is important to ensure that only the components that have changed run through the build process.

The IAKS Voting Application uses a code branch called *features* to develop new features for the application. The current version of the application only supports two

voting options. There is a feature in development to add a third voting option. The code to add that feature is using the branch *feature/vote3/0.5*. Listing 9-1 shows the code in the pipeline file that contains the trigger conditions.

Listing 9-1. Trigger conditions

```
trigger:
  branches:
    include:
      - master
      - feature/*
```

The pipeline will only kick off if a commit is made to *master* or a branch under *feature*.

Variables

A build pipeline won't have all values being used by each task hardcoded into the file. There will dynamic properties, secrets, and calculated values used as part of the pipeline. For instance, a pipeline may need a database password or API key. That value should not be stored in plaintext in a pipeline definition file; instead, it can be stored as a secret that is injected at build time. Properties such as the build number, build branch, and build date are also dynamic and can be used when naming artifacts. Listing 9-2 shows the definition of variables for the IAKS Voting Application build pipeline.

Listing 9-2. Build pipeline variables

```
variables:
  versionNumber: $[format('{0}.{1}', variables['Build.BuildNumber'],
    variables['Build.BuildId'])]
  repositoryName: 'iaks'
```

While these values are being defined within the pipeline file, it is also possible to override these values at runtime.

Steps

A build pipeline is composed of steps to take as the pipeline progresses. A common set of steps is shown in Figure 9-5.

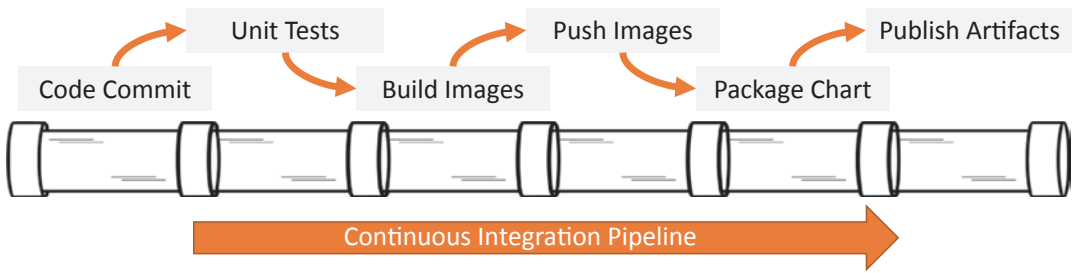


Figure 9-5. *Continuous Integration pipeline*

The build agent used to execute each step can be defined per step or at the beginning of a pipeline. More complicated steps may require a build agent with a specific operating system, specialized application, or geographic location. In the IAKS Voting Application, the build agent in Listing 9-3 is defined as a hosted Ubuntu agent with the latest version of the operating system.

Listing 9-3. Build pipeline agent definition

```
pool:
  vmImage: 'ubuntu-latest'
```

Azure DevOps offers hosted Windows and Linux agents that are allocated on demand. It is also possible to set up dedicated pools of build agents either in Azure or another location.

The code will be tested, packaged, and placed in a designated location as a collection of artifacts. The artifacts will be used by the release pipeline to deploy the application.

The IAKS Voting Application first builds the front-end web application container image as shown in Listing 9-4.

Listing 9-4. Build pipeline docker task

```
- task: Docker@2
  inputs:
    containerRegistry: 'iaks'
    repository: 'azure-voting-app'
    command: 'buildAndPush'
    tags: '$(versionNumber)'
    Dockerfile: '**/CICD/azure-vote/Dockerfile'
```

The task builds the image on the hosted agent and then tags and pushes the image up to an Azure Container Registry (ACR) using the `versionNumber` variable we defined earlier as a tag to differentiate between multiple builds.

The next step shown in Listing 9-5 is to take the Helm chart being used to deploy the application and package it up into a Helm archive file. The packaging process will also update the `Chart.yaml` values with the submitted version and application version numbers.

Listing 9-5. Build pipeline Helm install

```
- task: HelmInstaller@1
  inputs:
    helmVersionToInstall: 'latest'
- task: HelmDeploy@0
  inputs:
    command: 'package'
    chartPath: '**/Helm/aks/iaks'
    chartVersion: '$(versionNumber)'
    arguments: '--app-version $(versionNumber)'
```

The build agent does not have Helm installed by default, so the first task installs the latest version of Helm, and the second task creates the Helm package locally using the `versionNumber` variable for both the chart version and the application version.

We are going to store the packaged Helm chart in the same ACR as the web front-end container image. An Azure CLI command is used to push the package since there is no built-in task that will push a Helm package to ACR as shown in Listing 9-6.

Listing 9-6. Build pipeline ACR task

```
- task: AzureCLI@1
  inputs:
    azureSubscription: $(AzureSubscriptionId)
    scriptLocation: 'inlineScript'
    inlineScript: 'az acr helm push $(System.ArtifactsDirectory)/$(repositoryName)-$(versionNumber).tgz --name $(AzureContainerRegistry);'
```

The last two tasks shown in Listing 9-7 place the version number for this build into a text file and publish that text file as an artifact. The version number was defined in the variables as a combination of the `BuildNumber` and the `BuildId`. The `versionNumber`

variable was used to tag the container image and the Helm chart. The release pipeline will need the proper version number when it runs to find the correct image and chart.

Listing 9-7. Build pipeline bash script

```
- task: Bash@3
  inputs:
    targetType: 'inline'
    script: 'sudo echo $(versionNumber) > $(System.
      DefaultWorkingDirectory)/versionNumber.txt'

- task: PublishPipelineArtifact@1
  inputs:
    targetPath: '$(System.DefaultWorkingDirectory)/versionNumber.txt'
    artifact: 'versionNumber'
```

Notifications

When a build pipeline completes, whether it is successful or not, a notification should be sent out. There are many options when it comes to notification, the most common being email, chat, or webhook.

Notifications for Azure DevOps are handled outside of the pipeline, defined in the project settings. Third-party apps, such as Slack, can subscribe directly to a build or release and provide notifications to a specific channel. The IAKS Voting Application is being monitored on a Slack channel called *iaks*. When a build pipeline completes, the following notification in Figure 9-6 appears in the channel.

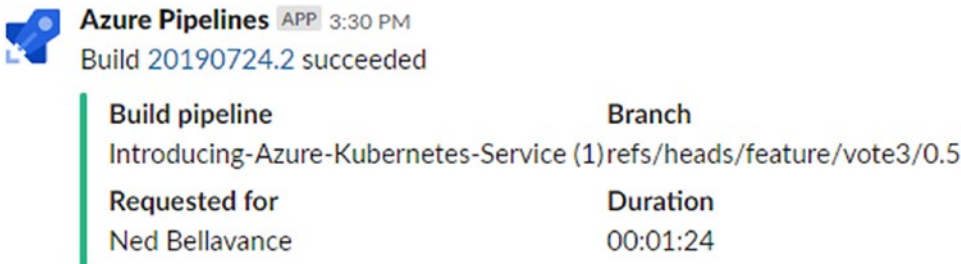


Figure 9-6. Azure Pipelines notification

Artifacts

A successful build pipeline will produce artifacts that can be used by the release pipeline to deliver and possibly deploy the application. The artifacts being produced by the build should be used consistently for any acceptance testing that happens in lower environments as well as Production. That guarantees whatever code is deployed in staging or QA will match what is deployed in Production. Tinkering with artifacts outside of the build pipeline is strongly discouraged, as in don't do it ever.

Why shouldn't you tinker with the artifacts? Let's say that you have built a new version of the application and deployed it into QA and staging. There's a small issue in the staging environment, but you find that you can fix it by tweaking a setting in a configuration file.

You make the change and the testing now passes. Since you tweaked the artifact in staging, the code being deployed no longer matches what is in source control. On a subsequent build, your change will be missing, and the thing you fixed is now broken again.

At best, you broke staging. At worst, the missing change makes it to Production and breaks there. Morale of the story? Don't alter artifacts. Make the change in code and run a new build.

The IAKS Voting Application creates three artifacts. The web front-end container image, the Helm chart, and the `versionNumber` text file. The container image and chart are stored in an Azure Container Registry instance. The text file is published as an artifact from the build, which makes it available to any release pipelines. As we move into the release pipeline section, we'll see how the artifacts from the build are ingested and used by the release.

Continuous Delivery/Deployment

As mentioned in the beginning of this chapter, CD can stand for Continuous Delivery or Deployment. The primary difference is at what point the automated process halts. In a continuous delivery environment, the end of the CD pipeline is a production-ready release. There is a manual step to formally deploy the release into production. A continuous deployment environment automates that last step.

Note For the purposes of this section, we will be primarily looking at setting up a continuous delivery pipeline. Continuous deployment will be explicitly noted when applicable.

Release Pipeline

The end result of a build pipeline is a set of artifacts that make up the application. These artifacts should be in a deployable state. It is the job of a release pipeline to take those artifacts and deploy the application to one or more environments, as well as run tests to validate the functionality of the application. A common example of a release pipeline is shown in Figure 9-7.

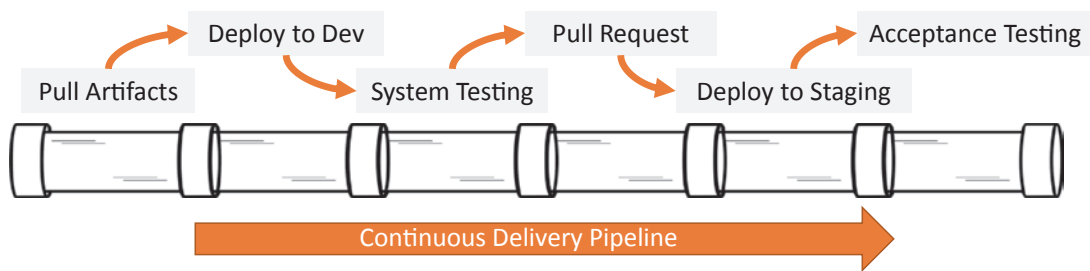


Figure 9-7. *Continuous Delivery pipeline*

The pipeline will pull the artifacts created by a specific build. Then, it will deploy the artifacts as an application to a development environment and run the system tests. Assuming those system tests pass, the pipeline will create a pull request to merge the feature branch into the master branch. Then, the pipeline will deploy the same artifacts to a staging environment where acceptance testing will occur. If acceptance testing is successful, the artifacts can be tagged as production ready or moved to a production-only repository. The movement of artifacts to a production repository could serve as a trigger to kick off another stage in the pipeline that deploys those artifacts to Production.

The IAKS Voting Application is using two release pipelines from Azure Pipelines to handle Continuous Delivery. Code built from the *feature* branch will trigger the *helm-dev-release* pipeline. That pipeline will take the artifacts from the feature build and deploy them to the *development* namespace on the AKS cluster. Rather than using a

single development namespace, it is also possible to enable the Dev Spaces feature on the AKS cluster. Dev Spaces creates a dedicated development environment that can be shared by multiple developers, with the ability to create per-user spaces within the larger development environment to test new features in real time.

When code is merged to the *master* branch through a pull request, it will trigger the *helm-qa-release* pipeline. That pipeline will take the artifacts from the last successful build and deploy them to the *qa* namespace on the AKS cluster. Once acceptance testing has been performed, a second phase will be invoked that takes the artifacts from the build and copies them to a production repository on ACR. There is a third manual phase that will deploy the production-tagged artifacts to the production AKS cluster.

Azure release pipelines are expressed using a graphical UI rather than through *yaml* files. It is likely that this will change soon to make the interface consistent across build and release pipelines. The release pipeline can be exported or imported using JSON files. For the examples in this section, a screenshot will be used to display the relevant portion of the release pipeline configuration.

Triggers

A common trigger for a CD pipeline is the successful completion of a CI pipeline. There may also be situations where the trigger is time-based and fires off on a daily schedule or is based on a pull request from another branch.

The IAKS Voting Application is triggered by a successful build of code from the *feature* branch or a pull request on the *master* branch. The configuration for the feature branch trigger on the *helm-dev-release* pipeline is shown in Figure 9-8.

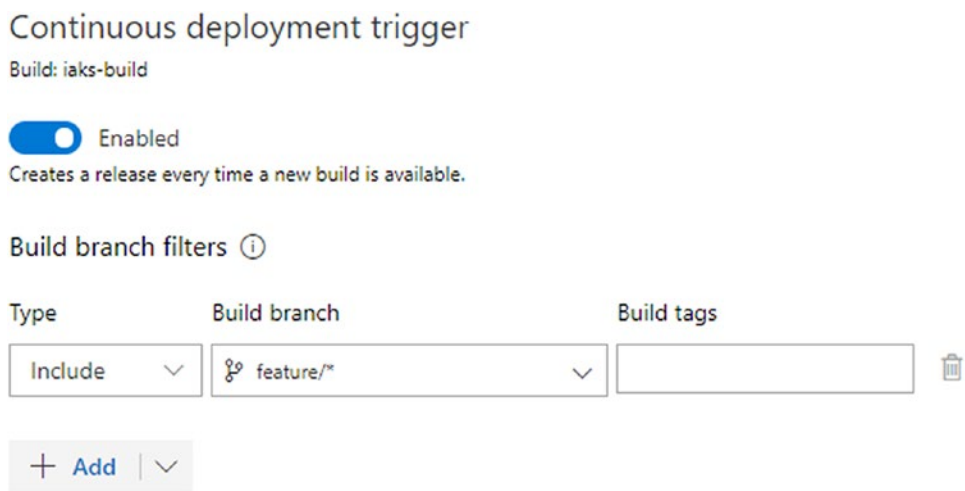


Figure 9-8. Continuous Deployment trigger

Each time a new build from a branch on the *feature* path is available, this pipeline will execute. The configuration for the *helm-qa-release* shown in Figure 9-9 is identical, except that the build branch is set to *master* and there is a pull request trigger added.

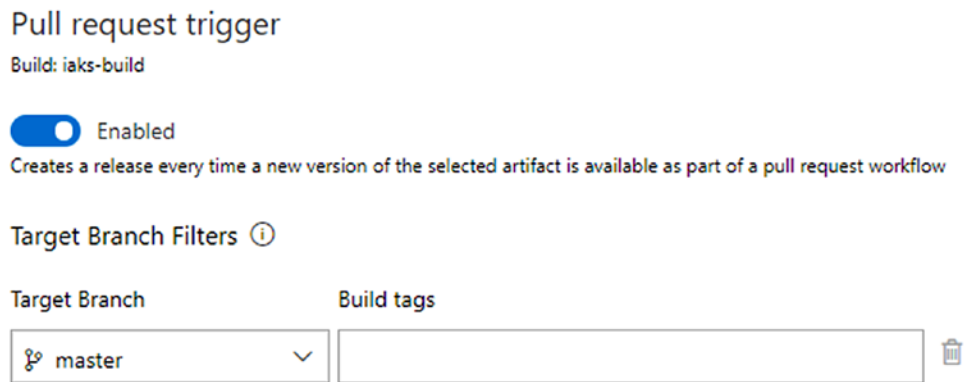


Figure 9-9. Azure Pipelines QA Release

Stages

The terminology for the stages in a CD pipeline depends on the software, but broadly there will be multiple stages in a CD pipeline. Each stage can be composed of one or more jobs, and each job is composed of one or more tasks. The stages and jobs can be sequential or run in parallel. The tasks within a job are usually run serially.

The helm-dev-release pipeline in Figure 9-10 is composed of a single stage, with one job.

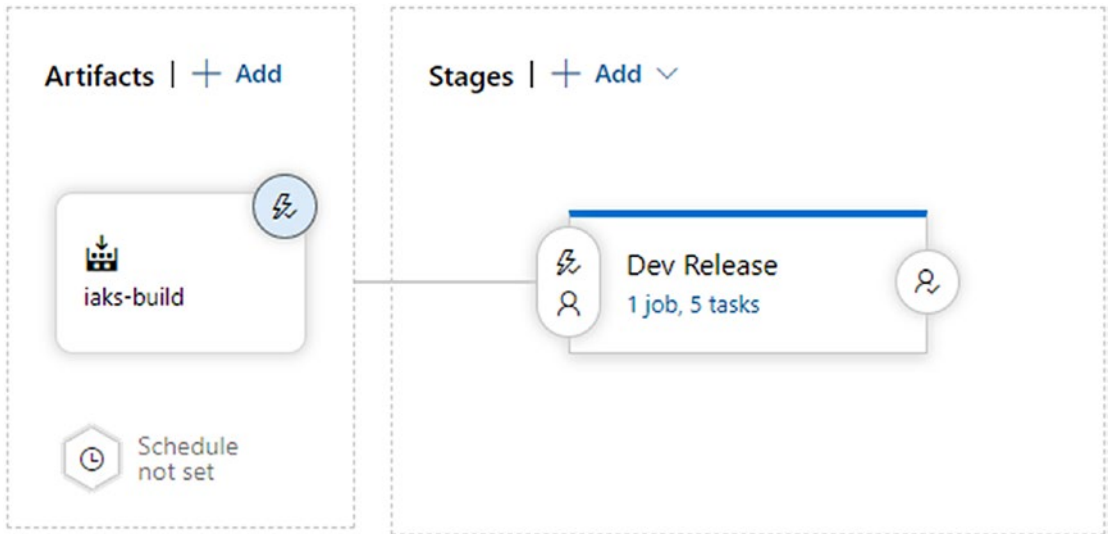


Figure 9-10. Azure Pipelines Dev Release

The artifacts for the stage are sourced from the build process. It is possible to have more than one source for artifacts. In a large application with several microservices, the artifacts might be sourced from the successful build of each microservice pipeline.

The helm-qa-release in Figure 9-11 is composed of three stages, each with one job.

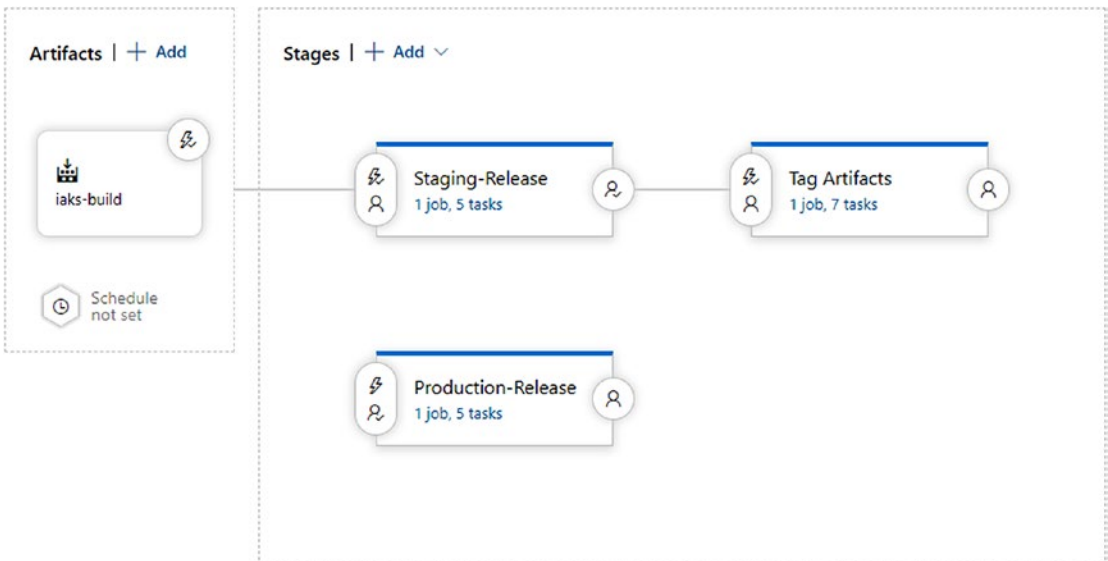


Figure 9-11. Azure Pipelines Staging Release

The jobs for all the stages run on a release agent, in the same way that build tasks are executed on a build agent. The Dev Release job is shown in Figure 9-12 for reference.

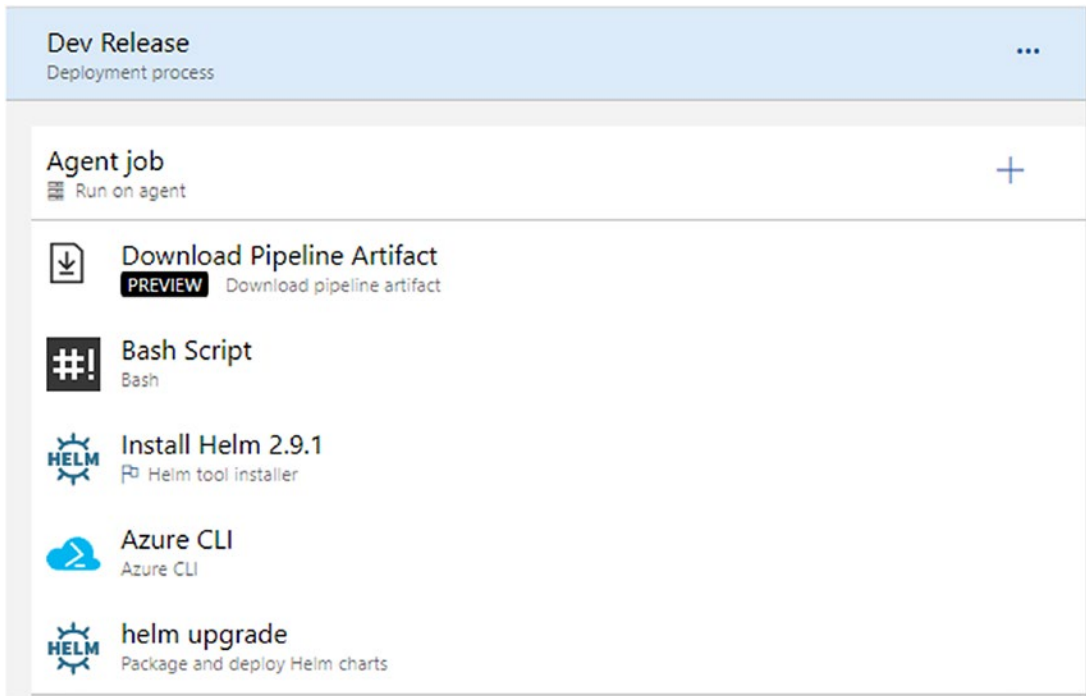



Figure 9-12. Azure Pipelines agent configuration

In Figure 9-12, you can see that the agent first downloads the pipeline artifact that contains the `versionNumber.txt` file. The following task is a bash script that gets the version number out of the text file and sets a release pipeline variable with that version number. Then the agent installs Helm and uses the Azure CLI to add the ACR repository where the Helm chart for the deployment resides. Finally, in Figure 9-13 Helm runs an upgrade on the AKS cluster in the *development* namespace using the Helm chart and container image that were part of the artifacts from the build pipeline.

Display name *

Kubernetes Cluster ^

Connection Type * ⓘ

Azure subscription * ⓘ | [Manage](#) 

ⓘ Scoped to resource group 'iaks'

Resource group * ⓘ

Kubernetes cluster * ⓘ

Namespace ⓘ

Commands ^

Command * ⓘ

Chart Type * ⓘ

Figure 9-13. *Azure Pipelines Helm deployment*

Chart Name * ⓘ

\$(acrName)/iaks

Release Name ⓘ

iaks-\$(Build.BuildId)

Set Values ⓘ

voteFront.image.repository=\$(imagePath),voteFront.image.tag=\$(versionNumber)

Value File ⓘ

☒ Install if release not present. ⓘ

☐ Recreate Pods. ⓘ

☐ Reset Values. ⓘ

☐ Force ⓘ

☒ Wait ⓘ

Figure 9-13. (continued)

When the deployment is finished, the *development* namespace in the AKS cluster has a new deployment of the IAKS Voting Application running as shown in Listing 9-8.

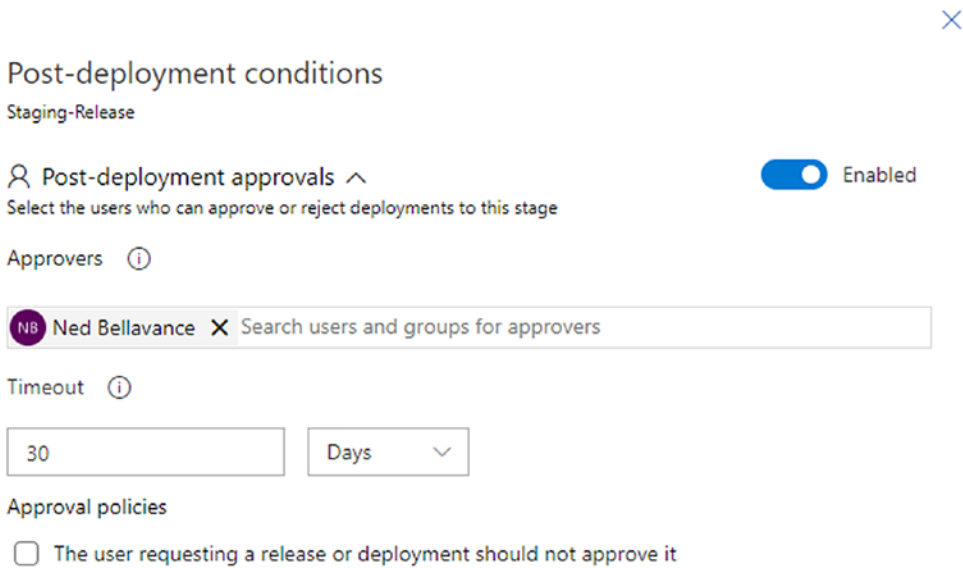
Listing 9-8. Helm deployment listing

```
helm ls --tls --namespace development
```

| NAME | STATUS | CHART | APP VERSION |
|---------|----------|--------------------|---------------|
| iaks-45 | DEPLOYED | iaks-20190725.4.45 | 20190725.4.45 |
| iaks-46 | DEPLOYED | iaks-20190725.4.46 | 20190725.4.46 |

The naming of each release includes the BuildId property, so each successful build will be created as its own deployment on the AKS cluster. The helm-qa-release uses the same tasks to deploy a copy of the application to the *qa* namespace, but it does not use the BuildId property to name the helm release. Therefore, when the helm-qa-release pipeline is run, it upgrades any existing application in place.

At the end of the deployment task in `helm-qa-release`, Figure 9-14 shows there is an approval condition. Someone from the QA department needs to review the deployment and verify that it has passed all the acceptance criteria.



Post-deployment conditions

Staging-Release

Post-deployment approvals [^] Enabled

Select the users who can approve or reject deployments to this stage

Approvers ⓘ

NB Ned Bellavance X Search users and groups for approvers

Timeout ⓘ

30 Days

Approval policies

☐ The user requesting a release or deployment should not approve it

Figure 9-14. Azure Pipelines post-deployment

By approving the Staging-Release stage, the next Tag Artifacts stage in the pipeline as seen in Figure 9-15 is executed, which involves the movement of build artifacts to the production repository. In Azure Container Registry, there is a separate registry instance called *acriaksprod*. The goal of the Tag Artifacts stage is to take the artifacts stored in *acriaks* and move them to *acriaksprod*.

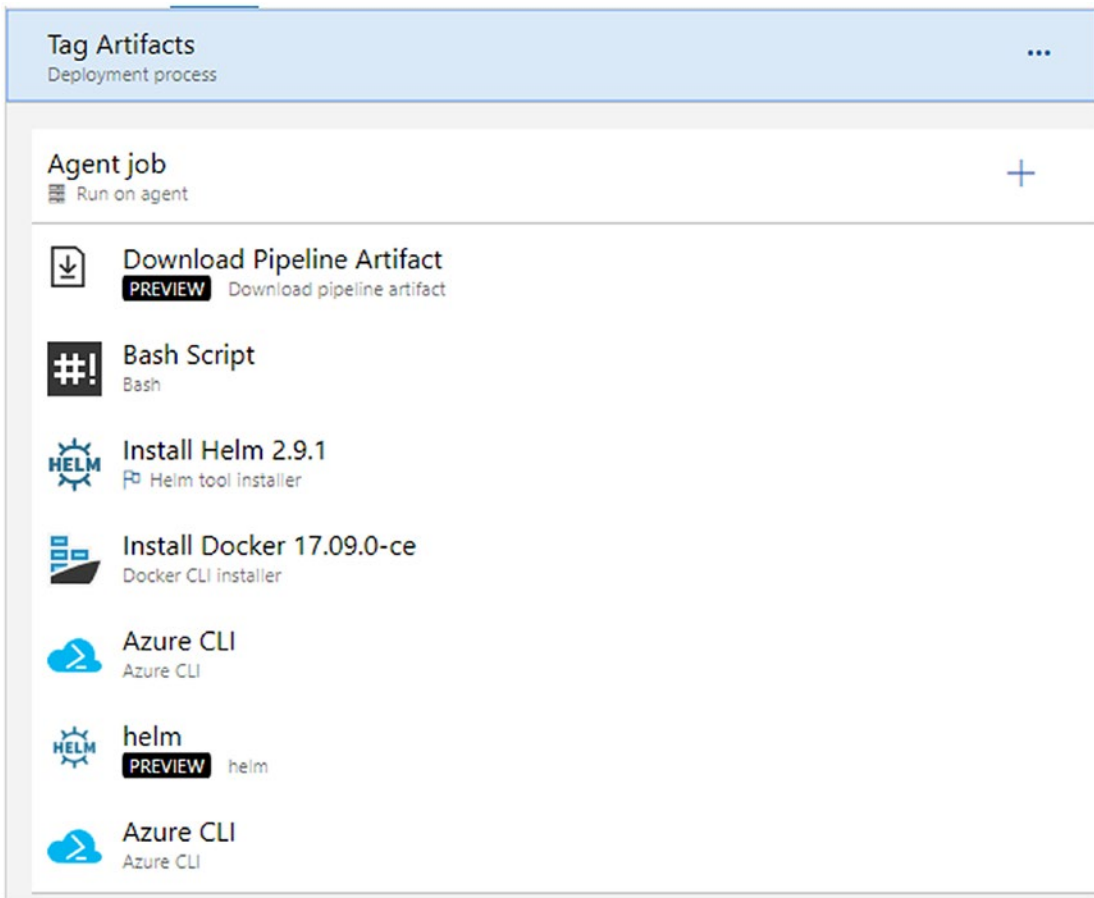


Figure 9-15. Azure Pipelines Tag Artifacts

The deployment process downloads the `versionNumber.txt` artifact and imports it as a variable. Then, it installs both the Helm and Docker clients. The first Azure CLI task adds the *acriaks* ACR as a Helm repository.

```
az acr helm repo add -n $(acrName)
```

Then, the Helm task gets the *iaks* chart from the *acriaks* repository and saves it locally.

```
helm fetch $(acrName)/iaks
```

In the last task shown in Listing 9-9, the Azure CLI first pushes the Helm chart to the *acriaksprod* repository. Then, it logs into both *acriaks* and *acriaksprod*. Using the docker CLI, it pulls the current web front-end image from *acriaks*, retags it for *acriaksprod*, and pushes the image to *acriaksprod*.

Listing 9-9. Azure CLI tasks

```

az acr helm push iaks-${versionNumber}.tgz --name $(acrName)prod
az acr login -n $(acrName)prod
az acr login -n $(acrName)
docker pull $(imagePath):$(versionNumber)
docker tag $(imagePath):$(versionNumber) $(prodImagePath):$(versionNumber)
docker push $(prodImagePath):$(versionNumber)

```

The Helm chart and web front-end image are now both stored in the production repository and ready for deployment to Production. The Production environment runs on a separate AKS cluster. We can configure an admission policy using the ImagePolicyWebhook to allow only images stored on the *acriaksprod* registry to be deployed to the Production cluster.

In *helm-qa-release* there is a separate stage shown in Figure 9-16 that can be manually run to deploy to Production.

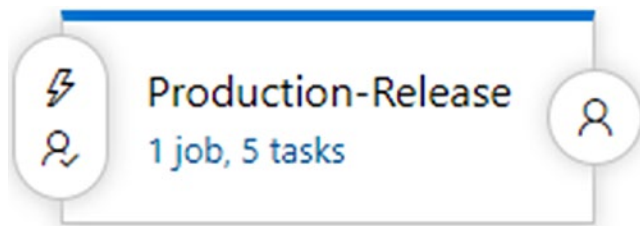
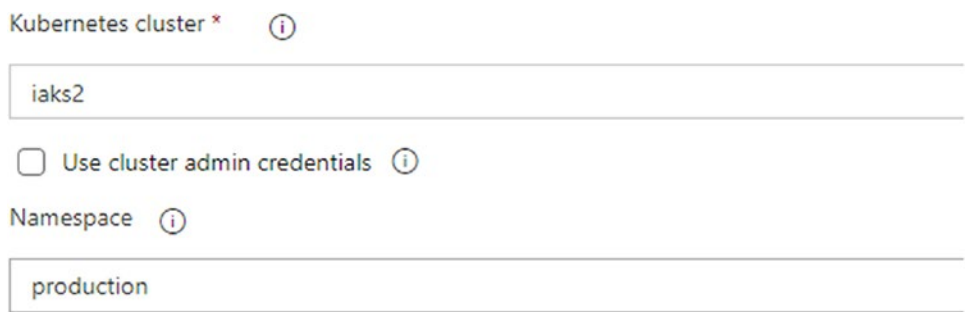


Figure 9-16. Azure Pipelines Production Release

The tasks in the Production-Release stage mirror the tasks in the Staging-Release stage, but some of the values have been changed as seen in Figure 9-17 to use the correct ACR instance, *acriaksprod*, and to deploy to a separate AKS cluster reserved for production use.



The screenshot shows the configuration for the 'Kubernetes cluster' task in Azure Pipelines. The task name is 'Kubernetes cluster *' with an information icon. Below it is a text input field containing 'iaks2'. Underneath is a checkbox labeled 'Use cluster admin credentials' with an information icon, which is currently unchecked. Below that is a 'Namespace' label with an information icon, followed by a text input field containing 'production'.

Figure 9-17. *Azure Pipelines Production Helm Task*

Although the Production-Release stage is currently set to be triggered manually – making it a continuous delivery pipeline – it could be updated to trigger after the Tag Artifacts stage is completed and approved by a select group of people.

Testing

An integral part to the entire CI/CD process is proper testing. There are different stages at which testing will occur, each test building off the last. Ultimately, the goal of testing is to produce reliable software that meets technical and business requirements. A portion of the testing will occur on the developer’s workstation prior to code being pushed to the shared repository. Once code is pushed to the shared repository, a CI pipeline will begin execution. The steps of the CI pipeline will include automated and sometimes manual testing, with the goal of producing stable software that is ready to be deployed and tested in a QA or staging environment. The last set of tests in the CI/CD context will be executed during the CD pipeline, hopefully culminating in an acceptable release that can be deployed in Production.

Depending on the deployment model, testing doesn’t necessarily end with deployment to Production. Code may be gradually rolled out to a subset of users or to certain geographic locations. Telemetry gathered during the rollout can be used to further validate that the software is working as expected.

In the following subsections, we will briefly look at some of the more common testing phases and where they sit within the larger context of the CI/CD process.

Unit Testing

When a new feature or function is being developed, there are requirements that it must meet. For a given set of inputs, it should produce certain outputs. For instance, let's say you were developing a function that adds two integers together. The unit tests would validate that given two integers – 1 and 2 – the function produces the expected out of 3. The unit test would have several cases to test with, including invalid input like a floating-point number or text instead of an integer. We might know that peanut butter and chocolate equals awesome, but our adding function should probably throw an error.

Unit testing will occur either on the developer's local workstation or during the early stages of the CI pipeline.

Integration Testing

After a feature or function is validated by itself, it then needs to be tested against the other portions of the application it interacts with. In our example, there is a good chance that portions of the application are already using a function to add integers together. When you replace that function with your new one, you must test those portions of the application to ensure that they are still working properly. In other words, you are testing the integration of your code with the components it interacts with.

Integration testing will occur as a stage in the CI pipeline before the build is released for deployment in the next testing stage.

System Testing

A cloud-native application is made up of microservices, each providing a service to other components of the application or to an external source. Although not all components will interact directly with the portions of the code being updated, there may still be unforeseen collateral effects. System testing is performed on the application as a whole, and not only on those components that directly interact with the update. In our example, this would be a standard suite of tests that apply to the entire application, and not just pieces that are using your new and totally awesome adding function.

System testing will occur after the release is built, typically during a CD pipeline that tests the release in a development or QA environment.

Acceptance Testing

System testing determines if the system is working as intended from a technical perspective, but there are other criteria for validation. The security, compliance, and QA teams may want to run their automated or manual test suites to validate that the application meets their business requirements. Sometimes a group of users will also be included in the testing to make sure that the user community is happy with the functionality of the application.

Acceptance testing will occur after the CI pipeline is complete and before the application is deployed to Production. There may be several rounds of acceptance testing by different teams included in a CD pipeline which ultimately leads to a release that is ready for Production rollout.

Dev Spaces

Within AKS, there is a feature called Dev Spaces that is currently in preview. The intention behind Dev Spaces is to make the development and testing process simpler for individual developers. Essentially, a parent Dev Space is created within the AKS cluster with one or more child spaces for each developer on a team as displayed in [Figure 9-18](#). In the parent space, a complete version of the application is deployed. As a developer makes updates to their code, the resulting build is deployed in their unique space, but it can interact with the parent space running the rest of the application.

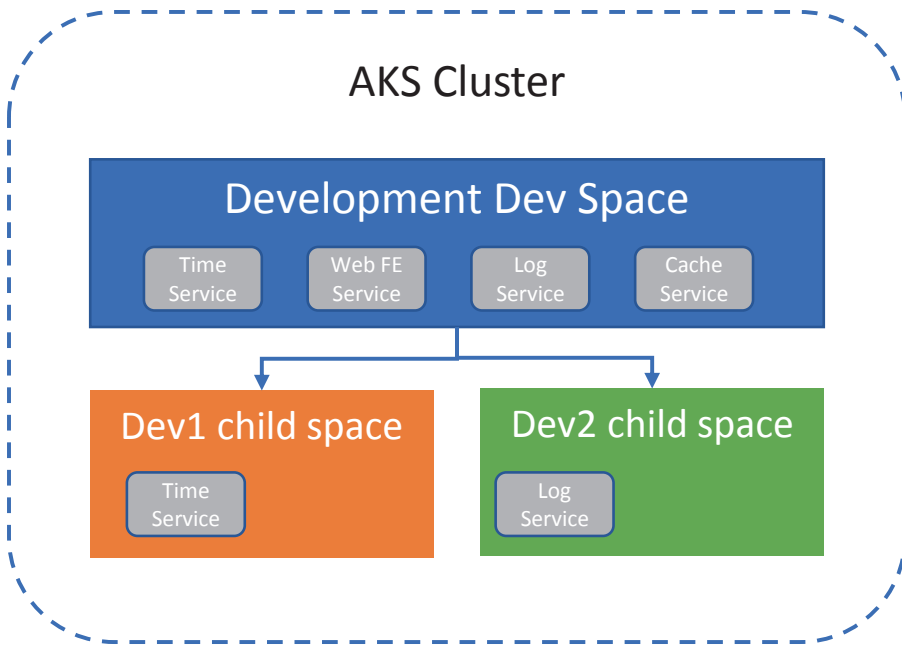


Figure 9-18. *Dev Spaces layout*

For instance, let's think about a microservices application that contains a time tracking service. You may be working on the time tracking service, and that service interacts with many of the other services that make up the application. Rather than trying to run the entire application on your workstation, instead an instance of the application is running on AKS in a Dev Space called *Development*. When you commit a new version of your time tracking service, Dev Spaces can deploy the resulting pods in your personal space, a child space called *Development/Dev1*. For testing purposes, you can now use the application in *Development*, but have your application endpoint use the newer time tracking service in *Development/Dev1*. Once you have successfully performed your unit and integration tests, you could merge your code into the development branch and the application in the *Development* space would be updated.

Dev Spaces does require preparing code to use it, creating a Dev Spaces configuration file, and using some client-side tooling to interact with it. The primary benefit is that developers don't have to try and maintain a local copy of the application for testing, and everyone on the team is using a common environment already running in Kubernetes to perform testing and debugging.

CI/CD Best Practices with AKS

There are many best practices when it comes to CI/CD, and we won't try to summarize them here. Instead, we would like to focus on applying CI/CD to the Azure Kubernetes Service. The best practices can be broken into two distinct areas, cluster operators and application developers.

Cluster Operators

Cluster operators are the folks responsible for managing and maintaining the AKS clusters. They handle configuring the security of the cluster, updating the version of Kubernetes, configuring scaling and node pools, and much more. You might be in this category if you are often deploying and making changes to AKS clusters. Cluster operators are primarily concerned with the Continuous Delivery portion of CI/CD, although they may become involved in assisting developers with Dev Spaces or carving out a permissioned namespace for integration testing.

Separate Environments

When it comes to delivery and deployment, there are going to be several environments that a release moves through: development, QA, staging, production, and so on. These environments will require some level of separation, and there are two primary ways to accomplish this, namespaces and clusters.

Namespaces in Kubernetes are a logical construct to separate workloads. Namespaces can be used in RBAC rules, restricting which namespaces an account could deploy to. Namespaces also have their own DNS space for name resolution, for example, a service in the development namespace might have the address `app.development.svc.cluster.local`. Namespaces may have resource quotas assigned to them, allowing you to limit the amount of resources a namespace can use within the cluster.

For environments that do not need hard separation from each other, namespaces make logical sense. Extra attention needs to be paid to firewall rules and restrictions, since namespaces do not provide network separation or segmentation between each other. A common best practice would be using namespaces to separate multiple development environments, as well as other nonproduction environments.

Clusters in AKS provide a harder separation between environments. Each cluster is completely separate from others from the perspective of Kubernetes administration.

Creating a separate cluster provides an additional level of segmentation from a networking standpoint that may be desirable for certain environments like Production. Multiple clusters increase the administrative burden, as now the cluster operator is responsible for managing and maintaining them. A common best practice would be using a separate cluster for the Production environment or other environments that might require a higher level of separation.

There is a third potential option, which would be the use of node pools in AKS. Node pools are in preview for AKS but should be generally available in the not too distant future. A cluster can have multiple node pools, with each pool comprised of the same node size and type. By using node tainting and selectors, it is possible to separate environments into different node pools, which would increase the isolation without requiring another AKS cluster to manage.

Restrict Access

With a proper release pipeline in place, developers should not be deploying directly to AKS. They should be following the process of committing their code and letting the CI and CD pipelines do the rest. Except for Dev Spaces, developers should not have permissions to deploy directly to any of the environments in AKS.

Build and release pipelines can be associated with service principals in Azure, and those service principals can be granted the necessary rights to deploy to AKS. Each environment should be permissioned with its own service principal, and access to edit pipelines should be restricted as well.

Restricting the developers access, as well as your own access, forces everyone to follow the CI/CD process and stops them from making changes outside the process that might be forgotten or overlooked in future deployments.

Admission Controls

Admission controls on a cluster help determine whether a deployment will be accepted or if anything about the deployment needs to be altered prior to acceptance. This can include requiring specific labels, resource limits, or image registries. To protect against possible tampering within a release pipeline, it is a best practice to create admission policies that prevent certain deployments from being successful. AKS has several admission control plug-ins enabled depending on what version of Kubernetes you are running on your AKS cluster.

Azure Monitor for Deployment

Azure Monitor is the combination of multiple monitoring services in Microsoft Azure. A big part of running an effective CI/CD operation is collecting feedback. Cluster operators are primarily concerned about the health of applications after a deployment or cluster maintenance. Azure Monitor can be configured to collect information from AKS for operators to use in alerting or trend analysis.

Application Developers

Cluster operators are concerned with the proper functioning and maintenance of the AKS clusters in their purview. Application developers are focused on the health and performance of their applications. You might be in this category if you are writing code that will run on an AKS cluster. Application developers are concerned with both the Continuous Integration and Continuous Delivery components of CI/CD.

Debug on Kubernetes

Applications may function differently depending on what environment they are running in. To that end, it makes sense to test and debug applications in an environment that most closely mirrors Production conditions. AKS provides the ability to debug directly on Kubernetes, either through a dedicated development environment for the whole team or through Dev Spaces. Using Dev Spaces in particular provides a common baseline environment for the whole team while still allowing individual development and debugging for a specific component of the application.

Store Credentials Externally

Credentials and other secret information should not be baked into code, deployment files, or CI/CD pipelines. They should be stored in a secure vault that is accessible at application runtime with proper permissions and security controls.

AKS and Azure DevOps are able to make use of Azure Key Vault. Both AKS and Azure DevOps use Azure Active Directory to authenticate against Key Vault and get access to the secrets, certificates, and keys stored there. AKS can use both Managed Security Identities for its nodes and pod-level Azure AD authentication to access Key Vault.

Application developers should take advantage of the Azure Key Vault integration to move any credentials, secrets, and certificates out of their code.

Azure Monitor for Development

Azure Monitor is the combination of multiple monitoring services in Microsoft Azure, including App Insights and Log Analytics. Logs, traces, and debugging information can all be sent to Azure Monitor for alerting and analysis. While the cluster operator is concerned about the health of their AKS clusters, developers are more focused on the health of their applications, especially when a new version of the application is deployed or if there is a sudden spike in traffic. The CI/CD pipeline can also send information to Azure Monitor as each stage executes, making that data available for alerting and analysis.

Application developers should add hooks into their code and pipelines to enable additional Azure Monitor integration.

Summary

Continuous Integration and Continuous Delivery are massive topics within the larger DevOps world. While we have just touched on some of the core concepts behind CI/CD, we hope that you can see how AKS ties into automating the build and release of software. Kubernetes, generally, and AKS, in particular, provide a consistent and stable environment for developing and deploying cloud-native applications. Working in tandem with CI/CD principles, it is possible to iterate rapidly and produce stable applications for the end user.

In this chapter, you learned what CI/CD is and some of the fundamentals behind source control, application builds, and release pipelines. We reviewed a build pipeline in Azure DevOps and saw how a code commit results in usable artifacts for delivery. Then, we looked at a release pipeline and how the same artifacts could be used to deploy the application in multiple environments. Finally, we reviewed some of the best practices around using AKS with CI/CD.

Index

A, B

Apiserver, [39](#)

Azure Active Directory (AAD), [123](#), [124](#)

Azure Container Instances (ACI), [17](#), [106](#),
[107](#), [114](#)

Azure Container Registry (ACR), [17](#)

- AKS integration, [33](#)

- formats, [27](#)

- overview, [27](#)

- permissions, [29](#), [30](#)

- registry creation, [27](#)

- security, [28](#), [29](#)

- tasks and automation, [30](#)

 - mutli-step tasks, [31](#), [32](#)

 - simple tasks, [30](#), [31](#)

 - webhooks, [32](#), [33](#)

Azure DevOps (ADO), [187–189](#)

Azure Kubernetes Service (AKS), [33](#), [63](#)

- access and identity options, [122](#)

 - AAD integration, [123](#)

 - ClusterRole, [123](#)

 - ClusterRoleBinding, [124](#)

 - RBAC clusters, [123](#)

 - RoleBinding, [124](#)

 - roles, [123](#)

 - service account, [122](#)

- ARM (*see* Azure Resource

 - Manager (ARM))

- business continuity and disaster
recovery, [145](#)

- datacenter failures, [147](#)

- disaster recovery, [146](#)

- master node failures, [147](#)

- regional failures, [148](#), [149](#)

- replication and protection
level, [146](#)

- worker nodes, [147](#)

cluster page

- authentication section, [67](#)

- cluster dashboard, [70](#)

- configuration, [64](#)

- connection, [76](#)

- creation, [65](#)

- networking section, [68](#)

- validation section, [69](#)

command line tool, [70–72](#)

container monitoring (*see* Container
monitoring services)

control deployments

- add-on, [125](#), [126](#)

- definitions, [126](#)

- GateKeeper logs, [124](#), [127](#)

- policy preview, [124](#), [125](#)

- prerequisites, [125](#), [126](#)

- validation, [127](#)

deployment

- Azure Portal, [63–69](#)

- overview, [63](#)

Helm (*see* Helm charts)

master component logs

Azure Kubernetes Service (AKS) (*cont.*)

- configuration, [144](#)
 - diagnostic settings, [143](#)
 - KQL queries-retrieve, [145](#)
 - networking concepts
 - CNI model, [120–122](#)
 - kubenet, [120](#)
 - security groups and policies, [122](#)
 - scaling concepts, [110](#)
 - ACI connection, [114](#)
 - automatically scale pods or nodes, [111](#)
 - cluster autoscaler, [113](#)
 - horizontal pod autoscaler, [111](#)
 - pods and nodes, [110](#)
 - virtual kubelet and nodes, [114](#)
 - security concepts, [128](#)
 - cluster upgrade process, [130](#)
 - Kubernetes Secret, [130](#)
 - master components, [128](#)
 - nodes, [128, 129](#)
 - storage options
 - architecture, [115, 116](#)
 - classes, [117](#)
 - PersistentVolumeClaim classes, [118–120](#)
 - PV, [116](#)
 - volumeMount, [118](#)
 - volumes, [116, 117](#)
 - Terraform, [76](#)
- Azure Resource Manager (ARM)
- QuickStart template, [73–76](#)
 - service principal, [72](#)
 - SSH key pair creation, [72](#)

C, D

- Cloud Native Application Bundle (CNAB), [155](#)
- Command line interface (CLI), [2, 70–72](#)
- Community Edition (CE), [3](#)
- Container Networking Interface (CNI), [119](#)
 - advantages and disadvantages of, [121](#)
 - clusters, [121](#)
 - kubenet, [120](#)
 - kubenet *vs.* Azure CNI, [120](#)
- Container monitoring services, [131](#)
 - alert rule creation, [139](#)
 - analytics view, [135](#)
 - architecture, [132](#)
 - cluster tab, [134](#)
 - component logs, [142–145](#)
 - containers, [134–136](#)
 - controllers, [134](#)
 - enable option, [132](#)
 - insights, [133](#)
 - Kubelet logs, [142](#)
 - log analytics, [138–143](#)
 - metrics, [136, 137](#)
 - nodes tab, [134](#)
 - overview, [131–133](#)
- Container registries
 - ACR (*see* Azure Container Registry (ACR))
 - definition, [17](#)
 - differences, [18](#)
 - Docker Hub web site, [26](#)
 - images, [18](#)
 - operations, [20](#)
 - image tags, [25–27](#)
 - login, [20, 21](#)
 - pull option, [22](#)

- push, 23, 24
 - search command, 21
- principle, 17
- private and public registry, 18–20
- repositories
 - differences, 18
 - private and public
 - repositories, 19–21
- Containers
 - definition, 1
- Docker, 2, 3
 - build command, 13
 - command cheat sheet, 10–12
 - compose, 13–15
 - Dockerfile instruction, 12, 13
 - installation, 8
 - login screen, 10
 - management command
 - structure, 11
 - networking, 6, 7
 - orchestration systems, 16, 17
 - requirements, 9
 - running option, 15
 - steps of, 9
 - storage options, 8, 9
 - WordPress application, 14
- images, 6
- value of, 2
- vs. virtual machines, 3–5
- Continuous delivery/deployment (CD)
 - pipeline
 - IAKS voting application, 202
 - master branch, 203
 - stages, 204–212
 - triggers, 203, 204
 - stages
 - agent configuration, 206
 - CLI tasks, 211
 - composed of, 205
 - deployment, 206, 207
 - dev release, 205
 - development, 208
 - helm-qa-release, 205
 - helm task, 212
 - post-deployment, 209
 - production release, 211
 - tag artifacts, 210
- Continuous integration and
 - continuous delivery
 - (CI/CD), 185
 - application developers, 218
 - credentials, 218
 - debug, 218
 - development, 219
 - automating deployments, 185
 - CD (*see* Continuous delivery/
 - deployment (CD))
 - CI (*see* Continuous
 - integration (CI))
 - cloud-native application, 191
 - cluster operators, 216
 - admission controls, 217
 - deployment, 218
 - restrict access, 217
 - separate environments, 216
- DevOps, 187–189
- installation, 187–189
- overview, 192
- testing, 186, 187
 - acceptance, 214
 - definition, 212
 - Dev Spaces, 214, 215
 - integration, 213
 - system, 213
 - unit testing, 213
- voting application, 192

INDEX

Continuous integration (CI)

- build pipeline, 195
 - artifacts, 201
 - bash script, 200
 - notifications, 200
 - steps, 197–200
 - triggers, 196
 - variables, 197
- constant process, 193
- shared repository, 194, 195

E, F, G

Enterprise edition (EE), 3

H, I

Helm charts

- advantages, 152
- chart contents
 - charts directory, 169
 - Chart.yaml file, 164, 165
 - deployment process, 171–173
 - license, 167
 - local repository, 170
 - myvalues.yaml file, 172
 - README.md, 167, 168
 - repositories, 170, 171
 - requirements.yaml, 168, 169
 - stable repository, 170
 - standard file and folder
 - structure, 163
 - templates directory, 169
 - values.yaml file, 165–167, 172
- chart tests, 176
- CI/CD integrations, 185
 - automating deployments, 185
 - Azure DevOps, 187–189

installation, 187

testing, 186, 187

CNAB project, 155

create command, 173, 174

deployment, 178

installation, 178–180

release, 181–183

remove option, 184, 185

status, 180

web page, 180

init (installation), 159

copying files, 163

pod and service details, 160–162

testing, 162

tiller service account, 159

key components, 153

client, 153, 154

repository, 155

Tiller, 154

overview, 151

package command, 176–178

primary use cases, 152

RBAC and service account, 156, 157

requirements, 156

template functions, 174–176

TLS considerations, 157–159

Horizontal pod autoscaler (HPA), 111, 112

J

JavaScript Object Notation (JSON) files,
32, 72, 82, 203

K, L

Kubectl commands, 51

categories, 52

formatting output, 54

- kubeconfig files, [51](#)
- operations, [58](#)
 - apply, [59](#)
 - delete, [61](#)
 - describe, [60](#)
 - exec operation, [61](#)
 - get, [59](#)
 - logs, [62](#)
- resources
 - debugging containers, [57, 58](#)
 - deployments, [54, 55](#)
 - generate config, [56, 57](#)
 - remote and local config, [55](#)
 - viewing pods associate, [57](#)
- syntax, [53](#)

Kubenet (Basic) networking, [120](#)

Kubernetes

- annotations, [42](#)
- architecture, [37](#)
- cloud-controller-manager, [39](#)
- ConfigMaps, [45, 46](#)
- DaemonSets, [44](#)
- dashboard, [38](#)
- deployments, [45](#)
- Docker runtime, [38](#)
- features of, [36](#)
- functions, [35](#)
- interfaces, [37, 38](#)
- jobs, [44](#)
- Kube-controller-manager, [39](#)
- Kube-scheduler, [39](#)
- labels and annotations, [41, 42](#)
- master node, [39, 40](#)
- namespaces, [40, 41](#)
- networking, [47](#)
- orchestration platform, [35](#)
- pods, [43, 44](#)
- replicasets, [43](#)

- secrets, [47](#)
- services, [44, 45](#)
- storage, [48–50](#)
- worker node, [40](#)

M

Macvlan, [7](#)

N

Network address translation
(NAT), [7, 120](#)

O

Open container initiative (OCI), [17](#)

Open Policy Agent (OPA), [124](#)

Operational process, [101](#)

- clusters
 - delete, [105](#)
 - Kubelet architecture, [108](#)
 - Kubernetes dashboard, [110–113](#)
 - nodes, [102, 103](#)
 - scale command, [102](#)
 - show command, [102](#)
 - upgrades commands, [103–105](#)
 - virtual nodes, [105–107](#)

Orchestration systems, [16, 17](#)

P

Port mapping, [7](#) *See also* Network address
translation (NAT)

Public Key Infrastructure (PKI), [157](#)

Q

QuickStart template, [73–76](#)

R

Rancher

- administration, [80](#)
- AKS deployment, [92](#)
 - account access settings, [93](#)
 - cluster name, [94](#)
 - components metrics, [98](#)
 - dashboard, [97](#)
 - DNS prefix, [94](#)
 - Grafana UI, [99](#)
 - Kubernetes provider, [93](#)
 - metrics, [98](#)
 - provision, [95](#)
 - resource group and SSH key, [95](#)
 - resources, [96](#)
- authentication, [90–92](#)
- deployment and management, [80](#)
- deployment option
 - ARM template code, [82–89](#)
 - parameters, [88](#)
 - password setting, [89](#)
 - resources, [88](#)
 - save URL, [90](#)

Kubernetes, [80, 81](#)

overview, [79](#)

RancherNode.JSON, [82–89](#)

Role-based access control
(RBAC), [29, 123, 156, 157](#)

S

Service-level objectives and agreements
(SLOs and SLAs), [145](#)

Service principal name (SPN),
[72, 90, 92](#)

Source control management
(SCM), [193, 194](#)

T, U

Terraform, [76, 77, 132](#)

V, W, X, Y, Z

Virtual machines (VMs), [3–5,](#)
[81, 128, 129, 146](#)