Containerized Docker Application Lifecycle with Microsoft Platform and Tools
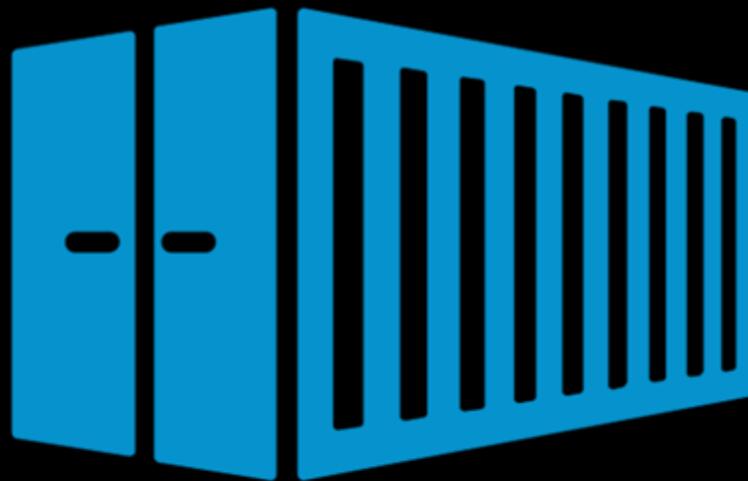
Cesar de la Torre

Microsoft Corp.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

This book is provided "as-is" and expresses the author's views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at http://www.microsoft.com on the "Trademarks" webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Contents

Introduction

This ebook is one of many offered by Microsoft Press, and is part of a library that spans the broad range of Microsoft products, services, and technologies. We hope that you find it useful and invite you to explore the many other titles available. To find out more about this library and to learn how you can communicate with us, click the links in the subsections that follow.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: http://twitter.com/MicrosoftPress.

Introduction to containers and Docker

Containerization is an approach to software development in which an application or service, its dependencies, and its configuration (abstracted as deployment manifest files) are packaged together as a container image. You then can test the containerized application as a unit and deploy it as a container image instance to the host operating system.

Just as the shipping industry uses standardized containers to move goods by ship, train, or truck, regardless of the cargo within them, software containers act as a standard unit of software that can contain different code and dependencies. Placing software into containers makes it possible for developers and IT professionals to deploy those containers across environments with little or no modification.

Containers also isolate applications from one another on a shared operating system (OS). Containerized applications run on top of a container host, which in turn runs on the OS (Linux or Windows). Thus, containers have a significantly smaller footprint than virtual machine (VM) images.

Each container can run an entire web application or a service, as shown in Figure 1-1.
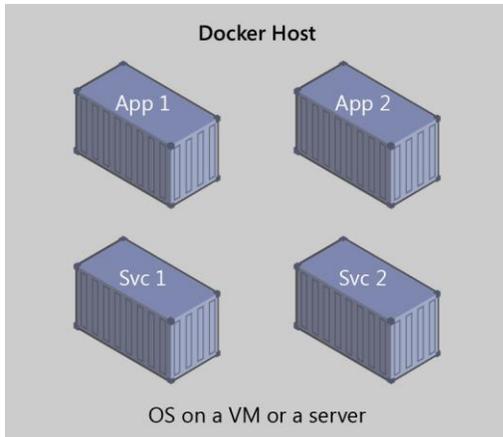
Figure 1-1: Multiple containers running on a container host

In this example, Docker Host is a container host, and App 1, App 2, Svc 1, and Svc 2 are containerized applications or services.

Another benefit you can derive from containerization is scalability. You can scale-out quickly by creating new containers for short-term tasks. From an application point of view, *instantiating an image* (creating a container) is similar to instantiating a process like a service or web app. For reliability, however, when you run multiple instances of the same image across multiple host servers, you typically want each container (image instance) to run in a different host server or VM in different fault domains.

In short, containers offer the benefits of isolation, portability, agility, scalability, and control across the entire application life cycle workflow. The most important benefit is the isolation provided between Dev and Ops.

What is Docker?

Docker is an open-source project for automating the deployment of applications as portable, self-sufficient containers that can run in the cloud or on-premises (see Figure 1-2). Docker is also a company that promotes and develops this technology, working in collaboration with cloud, Linux, and Windows vendors, including Microsoft.
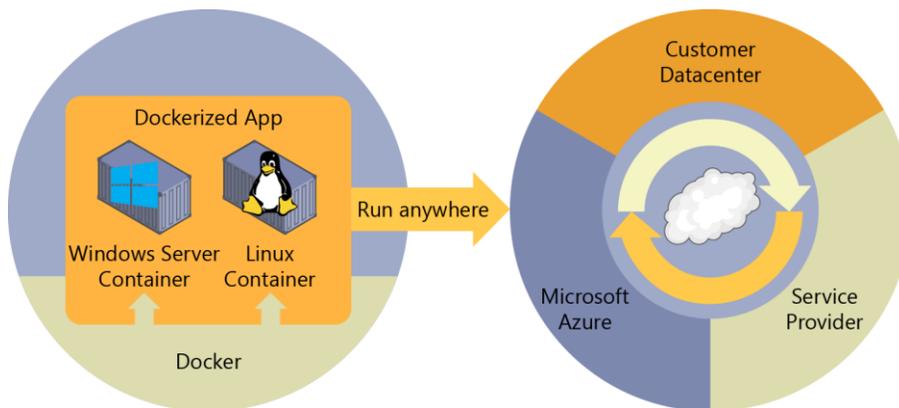
Figure 1-2: Docker deploys containers at all layers of the hybrid cloud

Docker image containers can run natively on Linux and Windows. However, Windows images can run only on Windows hosts and Linux images can run only on Linux hosts, meaning a host server or a VM.

Developers can use development environments on Windows, Linux, or macOS. On the development computer, the developer runs a Docker host to which Docker images are deployed, including the app and its dependencies. Developers who work on Linux or on the Mac use a Docker host that is Linux based, and they can create images only for Linux containers. (Developers working on the Mac can edit code or run the Docker command-line interface [CLI] from macOS, but, as of this writing, containers do not run directly on macOS.) Developers who work on Windows can create images for either Linux or Windows Containers.

To host containers in development environments and provide additional developer tools, Docker ships Docker Community Edition (CE) for Windows or for macOS. These products install the necessary VM (the Docker host) to host the containers. Docker also makes available Docker Enterprise Edition (EE), which is designed for enterprise development and is used by IT teams who build, ship, and run large business-critical applications in production.

To run Windows Containers, there are two types of runtimes:

- **Windows Server Container**   This runtime provides application isolation through process and namespace isolation technology. A Windows Server Container shares a kernel with the container host and with all containers running on the host.

- **Hyper-V Container**   This expands on the isolation provided by Windows Server Containers by running each container in a highly optimized VM. In this configuration, the kernel of the container host is not shared with the Hyper-V Containers, providing better isolation.

The images for these containers are created in the same way and function the same. The difference is in how the container is created from the image—running a Hyper-V Container requires an extra parameter. For details, see Hyper-V Containers.

Comparing Docker containers with VMs

Figure 1-3 shows a comparison between VMs and Docker containers.

Because containers require far fewer resources (for example, they do not need a full OS), they are easy to deploy and they start fast. This makes it possible for you to have higher density, meaning that you can run more services on the same hardware unit, thereby reducing costs.

As a side effect of running on the same kernel, you achieve less isolation than VMs.

The main goal of an image is that it makes the environment (dependencies) the same across different deployments. This means that you can debug it on your machine and then deploy it to another machine with the same environment guaranteed.

A container image is a way to package an app or service and deploy it in a reliable and reproducible manner. In this respect, Docker is not only a technology, it's also a philosophy and a process.

When using Docker, you will not hear developers say, "It works on my machine, why not in production?" They can simply say, "It runs on Docker," because the packaged Docker application can be run on any supported Docker environment, and it will run the way it was intended to on all deployment destinations (Dev, QA, staging, production, etc.).
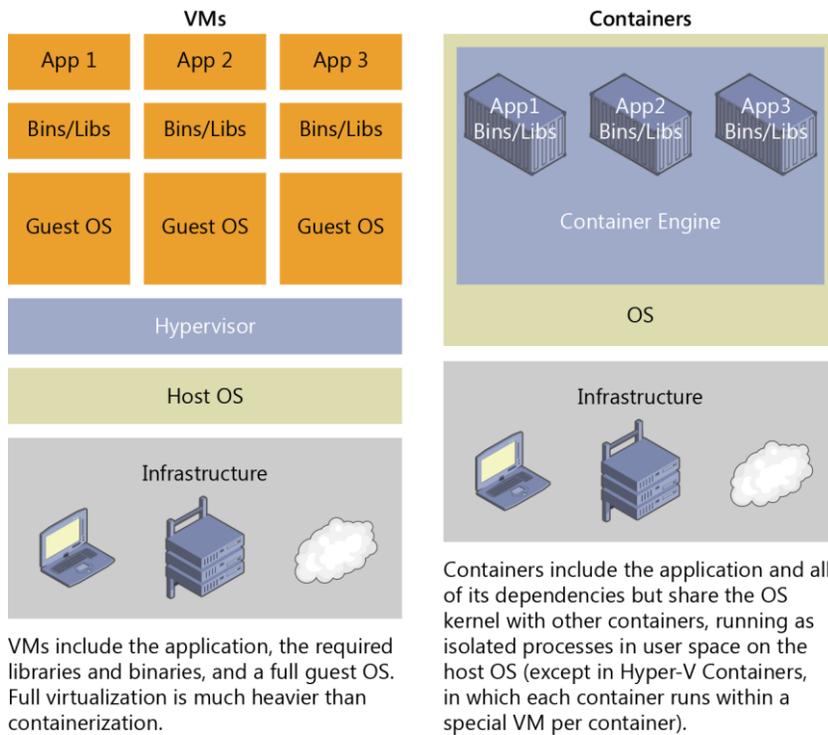
Figure 1-3: Comparison of traditional VMs to Docker containers

Docker terminology

This section lists terms and definitions with which you should become familiar with before delving deeper into Docker (for further definitions, see the extensive [glossary](https://docs.docker.com/v1.11/engine/reference/glossary/) provided by Docker at https://docs.docker.com/v1.11/engine/reference/glossary/):

- **Container image**   A package with all of the dependencies and information needed to create a container. An image includes all of the dependencies (such as frameworks) plus deployment and configuration to be used by a container runtime. Usually, an image derives from multiple base images that are layers stacked one atop the other to form the container's file system. An image is immutable after it has been created.

- **Container**   An instance of a Docker image. A container represents a runtime for a single application, process, or service. It consists of the contents of a Docker image, a runtime environment, and a standard set of instructions. When scaling a service, you create multiple instances of a container from the same image. Or, a batch job can create multiple containers from the same image, passing different parameters to each instance.

- **Tag**   A mark or label that you can apply to images so that different images or versions of the same image (depending on the version number or the destination environment) can be identified.

- **Dockerfile**   A text file that contains instructions for how to build a Docker image.

- **Build**   The action of building a container image based on the information and context provided by its Dockerfile as well as additional files in the folder where the image is built. You can build images by using the Docker `docker build` command.

- **Repository (aka repo)**   A collection of related Docker images labeled with a tag that indicates the image version. Some repositories contain multiple variants of a specific image, such as an image containing SDKs (heavier), an image containing only runtimes (lighter), and so on. Those variants can be marked with tags. A single repository can contain platform variants, such as a Linux image and a Windows image.

- **Registry**   A service that provides access to repositories. The default registry for most public images is [Docker Hub](#) (owned by Docker as an organization). A registry usually contains repositories from multiple teams. Companies often have private registries to store and manage images that they've created. *Azure Container Registry* is another example.

- **Docker Hub**   A public registry to upload images and work with them. Docker Hub provides Docker image hosting, public or private registries, build triggers and web hooks, and integration with GitHub and Bitbucket.

- **Azure Container Registry**   A public resource for working with Docker images and its components in Azure. This provides a registry that is close to your deployments in Azure and that gives you control over access, making it possible to use your Azure Active Directory groups and permissions.

- **Docker Trusted Registry (DTR)**   A Docker registry service (from Docker) that you can install on-premises so that it resides within the organization's datacenter and network. It is convenient for private images that should be managed within the enterprise. Docker Trusted Registry is included as part of the Docker Datacenter product. For more information, go to https://docs.docker.com/docker-trusted-registry/overview/.

- **Docker Community Edition (CE)**   Development tools for Windows and macOS for building, running, and testing containers locally. Docker CE for Windows provides development environments for both Linux and Windows Containers. The Linux Docker host on Windows is based on a [Hyper-V](#) VM. The host for Windows Containers is directly based on Windows. Docker CE for Mac is based on the Apple Hypervisor framework and the [xhyve hypervisor](#), which provides a Linux Docker host VM on Mac OS X. Docker CE for Windows and for Mac replaces Docker Toolbox, which was based on Oracle VirtualBox.

- **Docker Enterprise Edition (EE)**   An enterprise-scale version of Docker tools for Linux and Windows development.

- **Compose**   A command-line tool and YAML file format with metadata for defining and running multicontainer applications. You define a single application based on multiple images with one or more .yml files that can override values depending on the environment. After you have created the definitions, you can deploy the entire multicontainer application by using a single command (`docker-compose up`) that creates a container per image on the Docker host.

- **Cluster**   A collection of Docker hosts exposed as if they were a single virtual Docker host so that the application can scale to multiple instances of the services spread across multiple hosts within the cluster. You can create Docker clusters by using Docker Swarm, Mesosphere DC/OS, Kubernetes, and Azure Service Fabric. (If you use Docker Swarm for managing a cluster, you typically refer to the cluster as a *swarm* instead of a cluster.)

- **Orchestrator**   A tool that simplifies management of clusters and Docker hosts. Using orchestrators, you can manage their images, containers, and hosts through a CLI or a graphical user interface. You can manage container networking, configurations, load balancing, service discovery, high availability, Docker host configuration, and more. An orchestrator is responsible for running, distributing, scaling, and healing workloads across a collection of nodes. Typically, orchestrator products are the same products that provide cluster infrastructure, like Mesosphere DC/OS, Kubernetes, Docker Swarm, and Azure Service Fabric.

Docker containers, images, and registries

When using Docker, you create an app or service and package it and its dependencies into a container image. An image is a static representation of the app or service and its configuration and dependencies.

To run the app or service, the app's image is instantiated to create a container, which will be running on the Docker host. Containers are initially tested in a development environment or PC.

You store images in a registry, which acts as a library of images. You need a registry when deploying to production orchestrators. Docker maintains a public registry via Docker Hub; other vendors provide registries for different collections of images. Alternatively, enterprises can have a private registry on-premises for their own Docker images.

Figure 1-4 shows how images and registries in Docker relate to other components. It also shows the multiple registry offerings from vendors.
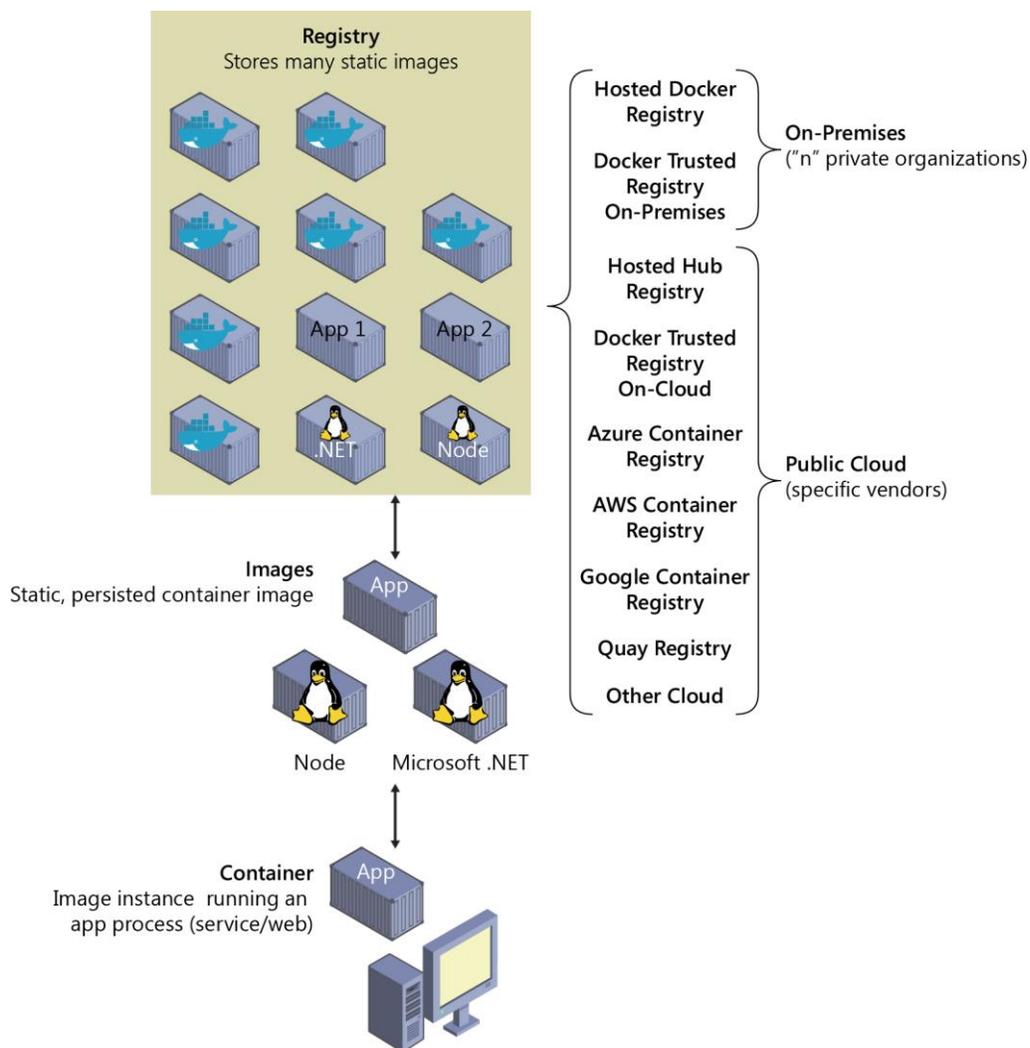
Figure 1-4: Taxonomy of Docker terms and concepts

By putting images in a registry, you can store static and immutable application bits, including all of their dependencies, at a framework level. You then can version and deploy images in multiple environments and thus provide a consistent deployment unit.

Private image registries, either hosted on-premises or in the cloud, are recommended for the following situations:

- Your images must not be shared publicly due to confidentiality.

- You want to have minimum network latency between your images and your chosen deployment environment. For example, if your production environment is Azure, you probably want to store your images in Azure Container Registry so that network latency will be minimal. In a similar way, if your production environment is on-premises, you might want to have an on-premises Docker Trusted Registry available within the same local network.

Road to modern applications based on containers.

Probably, you are reading this book because you are planning the development of new applications and you are analyzing the impact in your Company of using Docker, Containers and new approaches like Microservices.

The adoption of new development paradigms must be measured before starting a project to know the impact on your dev teams, your budget or your infrastructure.

Microsoft has been working in a rich guidance, sample applications and a suite of eBooks, that can help you to take the right decision and guide your team for a successful development, deployment and operations of your new applications.

This book belongs to a Microsoft's suite of guides that cover many of the needs you can have during the process of developing new modern applications based on containers.

The additional Microsoft eBooks related to Docker containers are the following:

> **.NET Microservices: Architecture for Containerized .NET Applications**
> https://aka.ms/MicroservicesEbook
>
> **Modernize existing .NET applications with Azure cloud and Windows Containers**
> https://t.co/xw5ilGAJmY

Introduction to the Docker application life cycle

The life cycle of containerized applications is a journey that begins with the developer. The developer chooses to implement containers and Docker because it eliminates frictions in deployments and IT operations, which ultimately helps everyone to be more agile, more productive end-to-end, and faster.

Containers as the foundation for DevOps collaboration

By the very nature of the containers and Docker technology, developers can share their software and dependencies easily with IT operations and production environments while eliminating the typical "it works on my machine" excuse. Containers solve application conflicts between different environments. Indirectly, containers and Docker bring developers and IT operations closer together, making it easier for them to collaborate effectively. Adopting the container workflow provides many customers with the DevOps continuity they've sought but previously had to implement via more complex configuration for release and build pipelines. Containers simplify the build/test/deploy pipelines in DevOps.
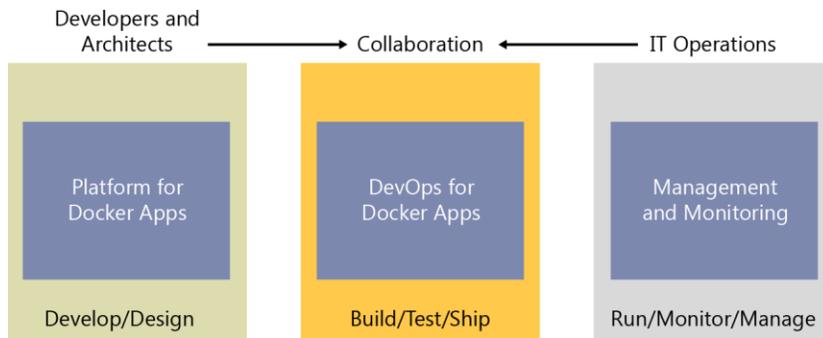
Figure 2-1: Main workloads per "personas" in the life cycle for containerized Docker applications

With Docker containers, developers own what's within the container (application and service, and dependencies to frameworks and components) and how the containers and services behave together as an application composed by a collection of services. The interdependencies of the multiple containers are defined in a docker-compose.yml file, or what could be called a *deployment manifest*. Meanwhile, IT operations teams (IT professionals and management) can focus on the management of production environments; infrastructure; scalability; monitoring; and, ultimately, ensuring that the applications are delivering properly for the end users, without having to know the contents of the various containers. Hence, the name "container," recalling the analogy to real-world shipping containers. Thus, the owners of a container's content need not concern themselves with how the container will be shipped, and the shipping company transports a container from its point of origin to its destination without knowing or caring about the contents. In a similar manner, developers can create and own the contents within a Docker container without the need to concern themselves with the "transport" mechanisms.

In the pillar on the left side of Figure 2-1, developers write and run code locally in Docker containers by using Docker for Windows or Mac. They define the operating environment for the code by using a Dockerfile that specifies the base operating system to run as well as the build steps for building their code into a Docker image. The developers define how the one or more images will interoperate using the aforementioned docker-compose.yml file deployment manifest. As they complete their local development, they push their application code plus the Docker configuration files to the code repository of their choice (i.e., Git repository).

The DevOps pillar defines the build–Continuous Integration (CI) pipelines using the Dockerfile provided in the code repository. The CI system pulls the base container images from the selected Docker registry and builds the custom Docker images for the application. The images then are validated and pushed to the Docker registry used for the deployments to multiple environments.

In the pillar on the right, operations teams manage deployed applications and infrastructure in production while monitoring the environment and applications so that they can provide feedback and insights to the development team about how the application might be improved. Container apps are typically run in production using container orchestrators.

The two teams are collaborating through a foundational platform (Docker containers) that provides a separation of concerns as a contract, while greatly improving the two teams' collaboration in the application life cycle. The developers own the container contents, its operating environment, and the container interdependencies, whereas the operations teams take the built images along with the manifest and runs them in their orchestration system.

Challenges in application life cycle when using Docker.

There are many reasons that will increase the number of containerized applications in the upcoming years, and one of these reasons is the creation of applications based on microservices.

During the last fifteen years the use of web services have been the base of thousands of applications, and probably, after a few years, we will find the same situation with microservice-based applications running on Docker containers.

It is also worth to mention that you can also use Docker containers for monolithic applications and you still get most of the benefits of Docker. Containers are not targeting only microservices.

The use of Docker containerization and microservices causes new challenges in the development process of your organizations and therefore, you need a solid strategy to maintain many containers and microservices running on production systems. Eventually, enterprise applications will have hundreds or thousands of containers/instances running in production.

These challenges create new demands when using DevOps tools. You need to define new processes in your DevOps activities and answers for questions like the following:

- Which tools can I use for development, for CI/CD, management and operations?
- How can my company manage errors in containers when running in production?
- How can we change pieces of our software in production with minimum downtime?
- How can we scale and how can we monitor our production system?
- How can we include testing and deployment of containers in our release pipeline?
- How can we use Open Source tools/platforms for containers in Microsoft Azure?

If you are able to answer to all those questions you will be better prepared to move your applications (existing or new apps) to Docker containers.

Introduction to a generic end-to-end Docker application life cycle workflow

Figure 2-2 presents a more detailed workflow for a Docker application life cycle, focusing in this instance on specific DevOps activities and assets.
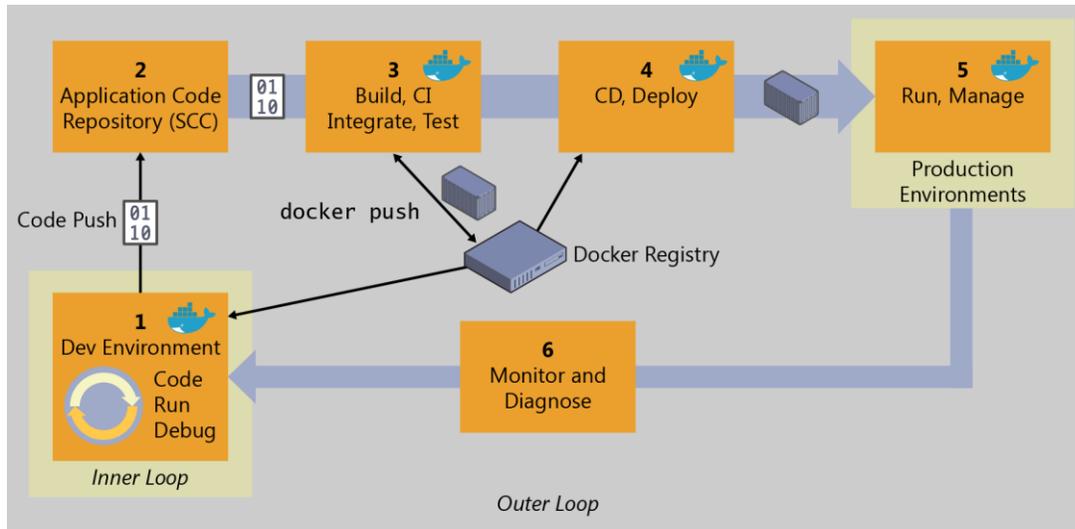
Figure 2-2: High-level workflow for the Docker containerized application life cycle

Everything begins with the developer, who starts writing code in the inner-loop workflow. The inner-loop stage is where developers define everything that happens before pushing code into the code repository (e.g., a source control system such as Git). After it is committed, the repository triggers Continuous Integration (CI) and the rest of the workflow.

The inner loop basically consists of typical steps like "code," "run," "test," and "debug," plus additional steps directly before running the app locally. This is when the developer runs and tests the app as a Docker container. The inner-loop workflow will be explained in the sections that follow.

Taking a step back to look at the end-to end workflow, the DevOps workflow is more than a technology or a tool set: it's a mindset that requires cultural evolution. It is people, processes, and the appropriate tools to make your application life cycle faster and more predictable. Enterprises that adopt a containerized workflow typically restructure their organizations to represent people and processes that match the containerized workflow.

Practicing DevOps can help teams respond faster together to competitive pressures by replacing error-prone manual processes with automation, which results in improved traceability and repeatable workflows. Organizations also can manage environments more efficiently and realize cost savings with a combination of on-premises and cloud resources as well as tightly integrated tooling.

When implementing your DevOps workflow for Docker applications, you'll see that Docker's technologies are present in almost every stage of the workflow, from your development box while working in the inner loop (code, run, debug), to the build-test-CI phase, and, of course, at the production and staging environments and when deploying your containers to those environments.

Improvement of quality practices helps to identify defects early in the development cycle, which reduces the cost of fixing them. By including the environment and dependencies in the image and adopting a philosophy of deploying the same image across multiple environments, you promote a discipline of extracting the environment-specific configurations making deployments more reliable.

Rich data obtained through effective instrumentation (monitoring and diagnostics) provides insight into performance issues and user behavior to guide future priorities and investments.

DevOps should be considered a journey, not a destination. It should be implemented incrementally through appropriately scoped projects from which you can demonstrate success, learn, and evolve.

Benefits of DevOps for containerized applications

Here are some of the most important benefits provided by a solid DevOps workflow:

- Deliver better-quality software, faster and with better compliance

- Drive continuous improvement and adjustments earlier and more economically

- Increase transparency and collaboration among stakeholders involved in delivering and operating software

- Control costs and utilize provisioned resources more effectively while minimizing security risks

- Plug and play well with many of your existing DevOps investments, including investments in open source

Introduction to the Microsoft platform and tools for containerized apps

Vision: Create an adaptable, enterprise-grade, containerized application life cycle that spans your development, IT operations, and production management.

Figure 3-1 shows the main pillars in the life cycle of Docker apps classified by the type of work delivered by multiple teams (app-development, DevOps infrastructure processes, and IT management and operations). Usually, in the enterprise, the profiles of "the persona" responsible for each area are different. So are their skills.
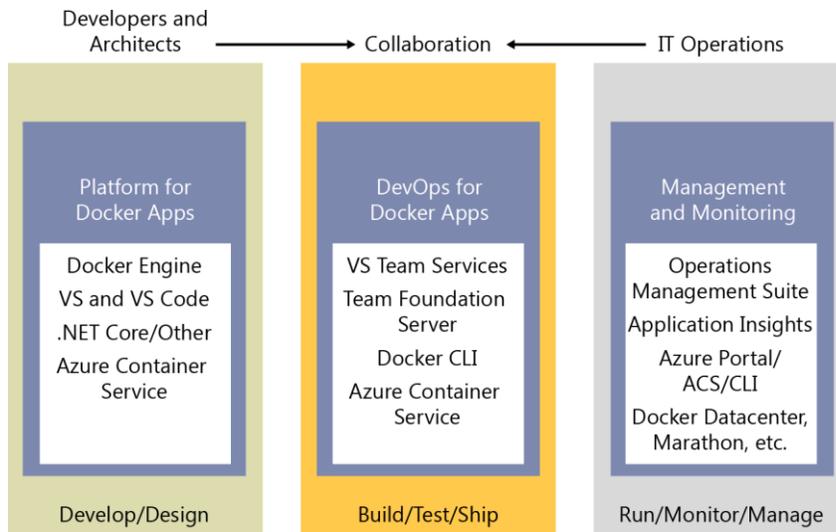
Figure 3-1: Main pillars in the life cycle for containerized Docker applications with Microsoft platform and tools

A containerized Docker life cycle workflow can be initially prescriptive based on "by-default product choices," making it easier for developers to get started faster, but it is fundamental that under the hood there must be an open framework so that it will be a flexible workflow capable of adjusting to the different contexts from each organization or enterprise. The workflow infrastructure (components and products) must be flexible enough to cover the environment that each company will have in the future, even being capable of swapping development or DevOps products to others. This flexibility, openness, and broad choice of technologies in the platform and infrastructure are precisely the Microsoft priorities for containerized Docker applications, as explained in the chapters that follow.

Table 3-1 demonstrates that the intention of the Microsoft DevOps for containerized Docker applications is to provide an open DevOps workflow so that you can choose what products to use for each phase (Microsoft or third party) while providing a simplified workflow that provides "by-default-products" already connected; thus, you can quickly get started with your enterprise-level DevOps workflow for Docker apps.

Table 3-1: DevOps workflow open to any technology

Host	Microsoft technologies	Third-party—Azure pluggable
Platform for Docker apps	• Visual Studio, Visual Studio Code & Visual Studio for Mac • .NET Framework and .NET Core • Azure Container Service & AKS • Azure Service Fabric • Azure Container Registry	• Any code editor (i.e. Sublime Text) • Any language (Node.js, Java, Go, etc.) • Any orchestrator and scheduler • Any Docker registry such as Docker Hub
DevOps for Docker apps	• Visual Studio Team Services • Microsoft Team Foundation Server • Azure Container Service & AKS • Azure Service Fabric • Azure Container Registry	• GitHub, Git, Subversion, etc. • Jenkins, Chef, Puppet, Velocity, CircleCI, TravisCI, etc. • On-premises Docker Datacenter, Docker Swarm, Mesos DC/OS, Kubernetes, etc.
Management and monitoring	• Operations Management Suite • Applications Insights	• Marathon, Chronos, etc.

The Microsoft platform and tools for containerized Docker apps, as defined in Table 3-1, comprise the following components:

- **Platform for Docker Apps development** The development of a service, or collection of services that make up an "app." The development platform provides all the work developers requires prior to pushing their code to a shared code repository. Developing services, deployed as containers, are very similar to the development of the same apps or services without Docker. You continue to use your preferred language (.NET, Node.js, Go, etc.) and preferred editor or IDE like Visual Studio or Visual Studio Code. However, rather than consider Docker a deployment destination, you develop your services in the Docker environment. You build, run, test, and debug your code in containers locally, providing the destination environment at development time. By providing the destination environment locally, Docker containers set up what will drastically help you improve your DevOps life cycle. Visual Studio and Visual Studio Code have extensions to integrate Docker containers within your development process.

- **DevOps for Docker Apps** Developers creating Docker applications can use Visual Studio Team Services (VSTS) or any other third-party product, like Jenkins, to build out a comprehensive automated application life cycle management (ALM).

  With VSTS, developers can create container-focused DevOps for a fast, iterative process that covers source-code control from anywhere (VSTS-Git, GitHub, any remote Git repository, or Subversion), Continuous Integration (CI), internal unit tests, inter container/service integration tests, Continuous Delivery (CD), and release management (RM). Developers also can automate their Docker application releases into Azure Container Service, from development to staging and production environments.

- IT production management and monitoring.

  Management IT can manage production applications and services in several ways:

  - **Azure portal** If you're using open-source orchestrators, Azure Container Service (ACS) plus cluster management tools like Docker Datacenter and Mesosphere Marathon help you to set up and maintain your Docker environments. If you're using Azure Service Fabric, the Service Fabric Explorer tool makes it possible for you to visualize and configure your cluster.

  - **Docker tools** You can manage your container applications using familiar tools. There's no need to change your existing Docker management practices to move container workloads to the cloud. Use the application management tools you're already familiar with and connect via the standard API endpoints for the orchestrator of your choice. You also can use other third-party tools to manage your Docker applications, such as Docker Datacenter or even CLI Docker tools. Even if you are familiarized with Linux commands you can manage your container applications using Microsoft Windows and PowerShell with a Linux Subsystem command-line and the products (Docker, Kubernetes...) clients running on this Linux Subsystem capabilities. You will learn more about using these tools under Linux Subsystem using your favorite Microsoft Windows OS later in this book.

  - **Open-source tools** Because ACS exposes the standard API endpoints for the orchestration engine, the most popular tools are compatible with ACS and, in most cases, will work out of the box—including visualizers, monitoring, command-line tools, and even future tools as they become available.

  Monitoring While running production environments, you can monitor every angle by using the following:

  - **Operations Management Suite (OMS)** The "OMS Container Solution" can manage and monitor Docker hosts and containers by showing information about where your containers and container hosts are, which containers are running or failed, and Docker daemon and

container logs. It also shows performance metrics such as CPU, memory, network, and storage for the container and hosts to help you troubleshoot and find noisy neighbor containers.

- **Application Insights**   You can monitor production Docker applications by simply setting up its SDK into your services so that you can get telemetry data from the applications.

Thus, Microsoft offers a complete foundation for an end-to-end containerized Docker application life cycle. However, it is *a collection of products and technologies that allow you to optionally select and integrate with existing tools and processes*. The flexibility in a broad approach along with the strength in the depth of capabilities place Microsoft in a strong position for containerized Docker application development.

Designing and developing containerized apps using Docker and Microsoft Azure

Vision: Design and develop scalable solutions with Docker in mind.

There are many great-fit use cases for containers, not just for microservices-oriented architectures, but also when you simply have regular services or web applications to run and you want to reduce frictions between development and production environment deployments.

Designing Docker applications

Chapter 1 introduced the fundamental concepts regarding containers and Docker. That information is the basic level of information you need to get started. But, enterprise applications can be complex and composed of multiple services instead of a single service or container. For those optional use cases, you need to know additional approaches to design, such as Service-Oriented Architecture (SOA) and the more advanced microservices concepts and container orchestration concepts. The scope of this document is not limited to microservices but to any Docker application life cycle, therefore, it does not explore microservices architecture in depth because you also can use containers and Docker with regular SAO, background tasks or jobs, or even with monolithic application deployment approaches.

> **More info** To learn more about enterprise applications and microservices architecture in depth, read the that NET Microservices: Architecture for Containerized .NET Applications that you can download from here https://aka.ms/MicroservicesEbook

However, before we get into the application life cycle and DevOps, it is important to know how you are going to design and construct your application and what are your design choices.

Common container design principles

Ahead of getting into the development process there are a few basic concepts worth mentioning with regard to how you use containers.

Container "equals" a process

In the container development model, a container represents a single process. By defining a container as a process boundary, you begin to create the primitives used to scale, or batch-off, processes. When you run a Docker container, you'll see an ENTRYPOINT definition. This defines the process and the lifetime of the container. When the process completes, the container life cycle ends. There are long-running processes, such as web servers, and short-lived processes, such as batch jobs, which might have been implemented as Microsoft Azure WebJobs. If the process fails, the container ends, and the orchestrator takes over. If the orchestrator was instructed to keep five instances running and one fails, the orchestrator will create another container to replace the failed process. In a batch job, the process is started with parameters. When the process completes, the work is complete.

You might find a scenario in which you want multiple processes running in a single container. In any architecture document, there's never a "never," nor is there always an "always." For scenarios requiring multiple processes, a common pattern is to use Supervisor.

Monolithic applications

In this scenario, you are building a single and monolithic web application or service and deploying it as a container. Within the application, the structure might not be monolithic; it might comprise several libraries, components, or even layers (application layer, domain layer, data access layer, etc.). Externally, it is a single container, like a single process, single web application, or single service.

To manage this model, you deploy a single container to represent the application. To scale it, just add a few more copies with a load balancer in front. The simplicity comes from managing a single deployment in a single container or virtual machine (VM).

Following the principal that a container does one thing only, and does it in one process, the monolithic pattern is in conflict. You can include multiple components/libraries or internal layers within each container, as illustrated in Figure 4-1.
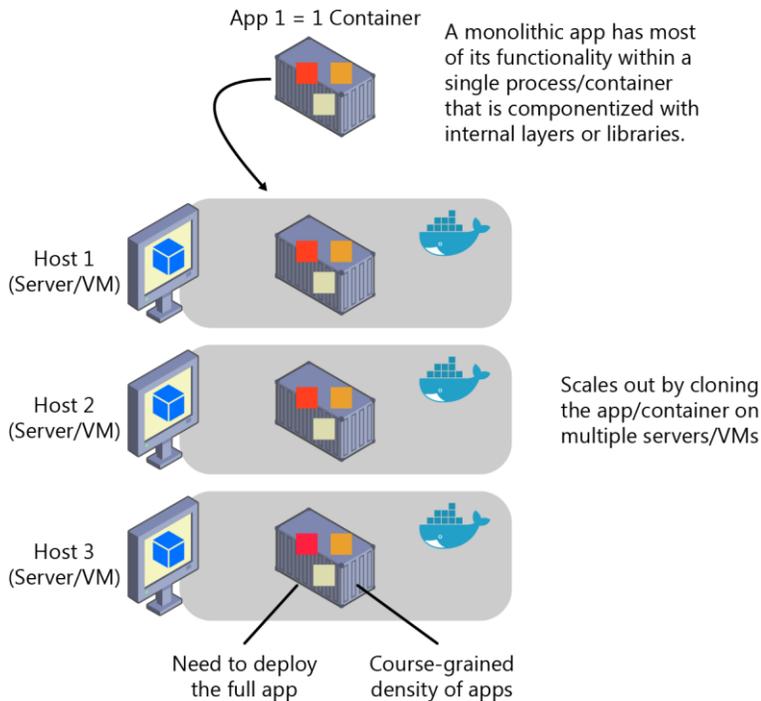
Figure 4-1: An example of monolithic application architecture

The downside to this approach comes if or when the application grows, requiring it to scale. If the entire application scaled, it's not really a problem. However, in most cases, a few parts of the application are the choke points that require scaling, whereas other components are used less.

Using the typical e-commerce example, what you likely need is to scale the product information component. Many more customers browse products than purchase them. More customers use their basket than use the payment pipeline. Fewer customers add comments or view their purchase history. And you likely have only a handful of employees, in a single region, that need to manage the content and marketing campaigns. By scaling the monolithic design, all of the code is deployed multiple times.

In addition to the "scale-everything" problem, changes to a single component require complete retesting of the entire application as well as a complete redeployment of all the instances.

The monolithic approach is common, and many organizations are developing with this architectural method. Many enjoy good enough results, whereas others encounter limits. Many designed their applications in this model because the tools and infrastructure were too difficult to build SOAs, and they didn't see the need—until the app grew.

From an infrastructure perspective, each server can run many applications within the same host and have an acceptable ratio of efficiency in your resources usage, as shown in Figure 4-2.
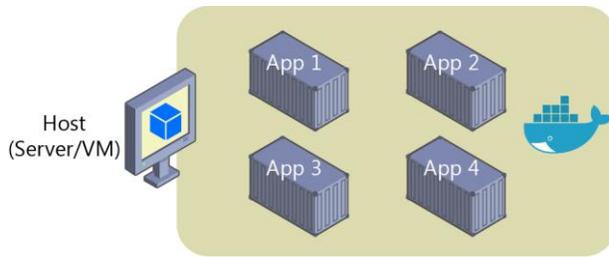
Figure 4-2: A host running multiple apps/containers

Finally, from availability perspective monolithic applications must be deployed completely, that means that in the case that you must *stop and start*, all the functionality and users will be affected during the deployment slot window. In certain situations, the use of Azure and containers can minimize these cases and reduce the probability of downtime of your application as you can imagine viewing the Figure 4-3.

You can deploy monolithic applications in Azure by using dedicated VMs for each instance. Using Azure VM Scale Sets, you can scale the VMs easily.

You can also use Azure App Services to run monolithic applications and easily scale instances without having to manage the infrastructure of the VMs. Since 2016, Azure App Services can run single instances of Docker Linux containers, simplifying the deployment.

You can deploy multiple VM as Docker hosts and run any number of containers per VM. Then, by using the Azure balancer, as illustrated in the Figure 4-3, you can manage scaling.
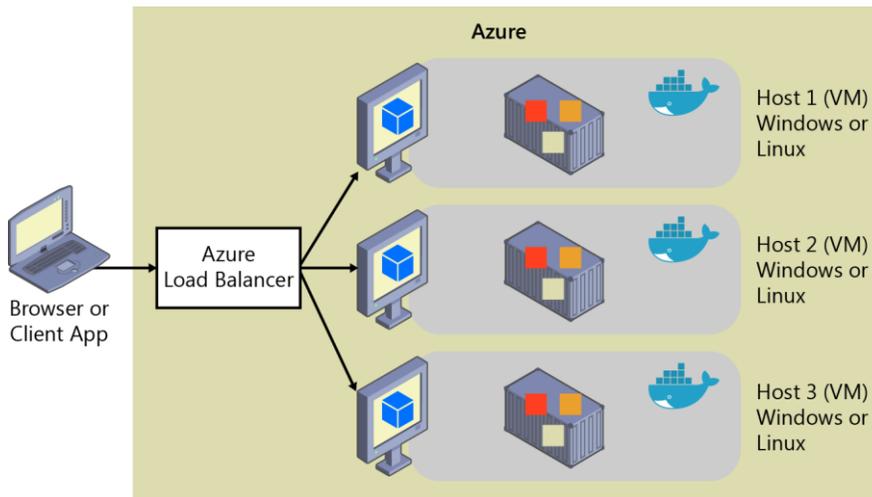
Figure 4-3: Multiple hosts scaling-out a single Docker application apps/containers

You can manage the deployment of the hosts themselves via traditional deployment techniques.

You can manually manage Docker containers deployment by using commands like `docker run` and `docker-compose up`, and you can also automate it in Continuous Delivery (CD) pipelines and deploy to Docker hosts from VSTS releases, for instance.

Monolithic application deployed as a container

There are benefits to using containers to manage monolithic deployments. Scaling the instances of containers is far faster and easier than deploying additional VMs.

Deploying updates as Docker images is far faster and network efficient. Docker containers typically start in seconds, speeding rollouts. Tearing down a Docker container is as easy as invoking the `docker stop` command, typically completing in less than a second.

Because containers are inherently immutable, by design, you never need to worry about corrupted VMs because an update script forgot to account for some specific configuration or file left on disk.

Although monolithic apps can benefit from Docker, we're touching on only the tips of the benefits. The larger benefits of managing containers comes from deploying with container orchestrators that manage the various instances and life cycle of each container instance. Breaking up the monolithic application into subsystems that can be scaled, developed, and deployed individually is your entry point into the realm of microservices.

> To learn about how to "lift and shift" monolithic applications with containers and how you can modernize your applications, you can read this additional Microsoft book named **Modernize existing .NET applications with Azure cloud and Windows Containers,** that you can download from here https://t.co/xw5ilGAJmY

Publishing a single Docker container app to Azure App Service

Either because you want to get a quick validation of a container deployed to Azure or because the app is simply a single-container app, Azure App Services provides a great way to provide scalable single-container services (As of 2017, only Linux containers support in Azure App Service).

Using Azure App Service is intuitive and you can get up and running quickly because it provides great Git integration to take your code, build it in Microsoft Visual Studio, and directly deploy it to Azure. But, traditionally (with no Docker), if you needed other capabilities, frameworks, or dependencies that aren't supported in App Services, you needed to wait for it until the Azure team updates those dependencies in App Service or switched to other services like Service Fabric, Cloud Services, or even plain VMs, for which you have further control and can install a required component or framework for your application.

Now, however, (announced at Microsoft Connect 2016 in November 2016) and as shown in Figure 4-4, when using Visual Studio 2017, container support in Azure App Service gives you the ability to include whatever you want in your app environment. If you added a dependency to your app, because you are running it in a container, you get the capability of including those dependencies in your Dockerfile or Docker image.
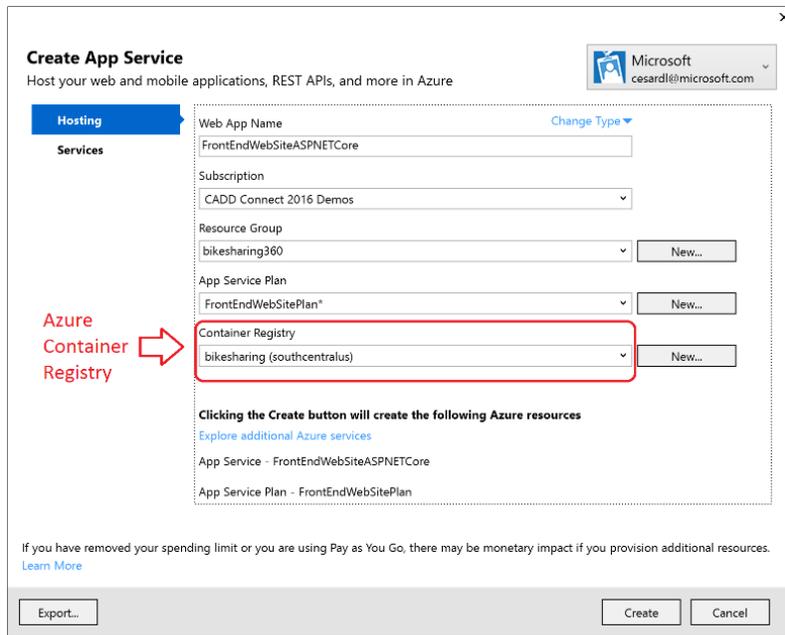
Figure 4-4: Publishing a container to Azure App Service from Visual Studio apps/containers

Figure 4-4 also shows that the publish flow pushes an image through a Container Registry, which can be the Azure Container Registry (a registry near to your deployments in Azure and secured by Azure Active Directory groups and accounts) or any other Docker Registry like Docker Hub or on-premises registries.

State and data in Docker applications

A primitive of containers is immutability. When compared to a VM, containers don't disappear as a common occurrence. A VM might fail in various forms from dead processes, overloaded CPU, or a full or failed disk. Yet, we expect the VM to be available and RAID drives are commonplace to assure drive failures maintain data.

However, containers are thought to be instances of processes. A process doesn't maintain durable state. Even though a container can write to its local storage, assuming that that instance will be around indefinitely would be equivalent to assuming a single-copy memory will be durable. You should assume that containers, like processes, are duplicated, killed, or, when managed with a container orchestrator, they might be moved.

Docker uses a feature known as an *overlay file system* to implement a copy-on-write process that stores any updated information to the root file system of a container, compared to the original image on which it is based. These changes are lost if the container is subsequently deleted from the system. A container, therefore, does not have persistent storage by default. Although it's possible to save the state of a container, designing a system around this would be in conflict with the principle of container architecture.

To manage persistent data in Docker applications, there are common solutions:

- **Data volumes**   These mount to the host, as just noted.

- **Data volume containers**   These provide shared storage across containers, using an external container that can cycle.

- **Volume Plugins**   These mount volumes to remote locations, providing long-term persistence.

- **Remote data sources**   Examples include SQL and NO-SQL databases or cache services like Redis.

- **Azure Storage**   This provides geo distributable platform as a service (PaaS) storage, providing the best of containers as long-term persistence.

Data volumes are specially designated directories within one or more containers that bypass the Union File System. Data volumes are designed to maintain data, independent of the container's life cycle. Docker therefore never automatically deletes volumes when you remove a container, nor will it "garbage collect" volumes that are no longer referenced by a container. The host operating system can browse and edit the data in any volume freely, which is just another reason to use data volumes sparingly.

A data volume container is an improvement over regular data volumes. It is essentially a dormant container that has one or more data volumes created within it (as described earlier). The data volume container provides access to containers from a central mount point. The benefit of this method of access is that it abstracts the location of the original data, making the data container a logical mount point. It also allows "application" containers accessing the data container volumes to be created and destroyed while keeping the data persistent in a dedicated container.

Figure 4-5 shows that regular Docker volumes can be placed on storage out of the containers themselves but within the host server/VM physical boundaries. *Docker volumes don't have the ability to use a volume from one host server/VM to another*.
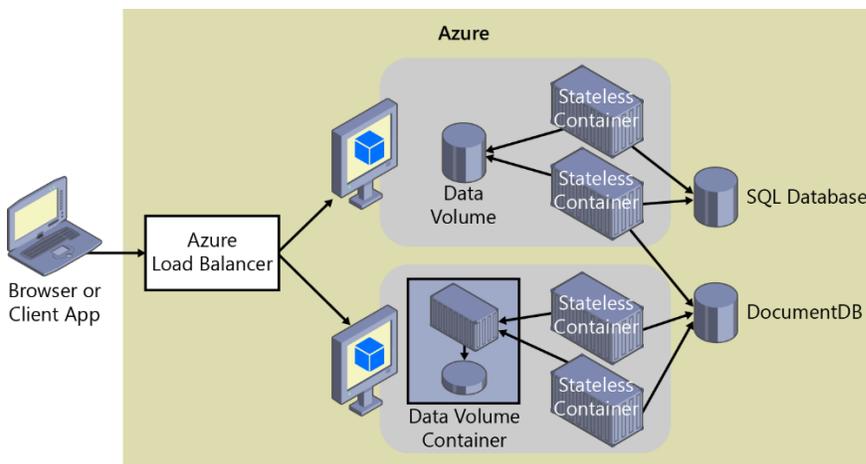
Figure 4-5: Data volumes and external data sources for containers apps/containers

Due to the inability to manage data shared between containers that run on separate physical hosts, it is recommended that you not use volumes for business data unless the Docker host is a fixed host/VM, because when using Docker containers in an orchestrator, containers are expected to be moved from one to another host, depending on the optimizations to be performed by the cluster.

Therefore, regular data volumes are a good mechanism to work with trace files, temporal files, or any similar concept that won't affect the business data consistency if or when your containers are moved across multiple hosts.

Volume plug-ins like Flocker provide data across all hosts in a cluster. Although not all volume plug-ins are created equally, volume plug-ins typically provide externalized persistent reliable storage from the immutable containers.

Remote data sources and caches like SQL Database, DocumentDB, or a remote cache like Redis would be the same as developing without containers. This is one of the preferred, and proven, ways to store business application data.

SOA applications

SOA was an overused term and meant so many different things to different people. But at a minimum and as a common denominator, SOA, or service orientation, mean that you structure the architecture of your application by decomposing it in multiple services (most commonly as HTTP services) that can be classified in different types like subsystems or, in other cases, as tiers.

Today, you can deploy those services as Docker containers, which solves deployment-related issues because all of the dependencies are included within the container image. However, when you need to scale-out SOAs, you might encounter challenges if you are deploying based on single instances. This is where a Docker clustering software or orchestrator will help you. We'll look at this in greater detail in the next section when we examine microservices approaches.

At the end of the day, the container clustering solutions are useful for both a traditional SOA architecture or for a more advanced microservices architecture in which each microservice owns its data model. And, thanks to multiple databases, you also can scale-out the data tier instead of working with monolithic databases shared by the SOA services. However, the discussion about splitting the data is purely about architecture and design.

Orchestrating microservices and multicontainer applications for high scalability and availability

Using orchestrators for production-ready applications is essential if your application is based on microservices or simply split across multiple containers. As introduced previously, in a microservice-based approach, each microservice owns its model and data so that it will be autonomous from a development and deployment point of view. But even if you have a more traditional application that is composed of multiple services (like SOA), you also will have multiple containers or services comprising a single business application that need to be deployed as a distributed system. These kinds of systems are complex to scale out and manage; therefore, you absolutely need an orchestrator if you want to have a production-ready and scalable multicontainer application.

Figure 4-6 illustrates deployment into a cluster of an application composed of multiple microservices (containers).

- For each service instance, you use *one* container
- Docker images/containers are units of deployment
- A container is an *instance* of a docker image
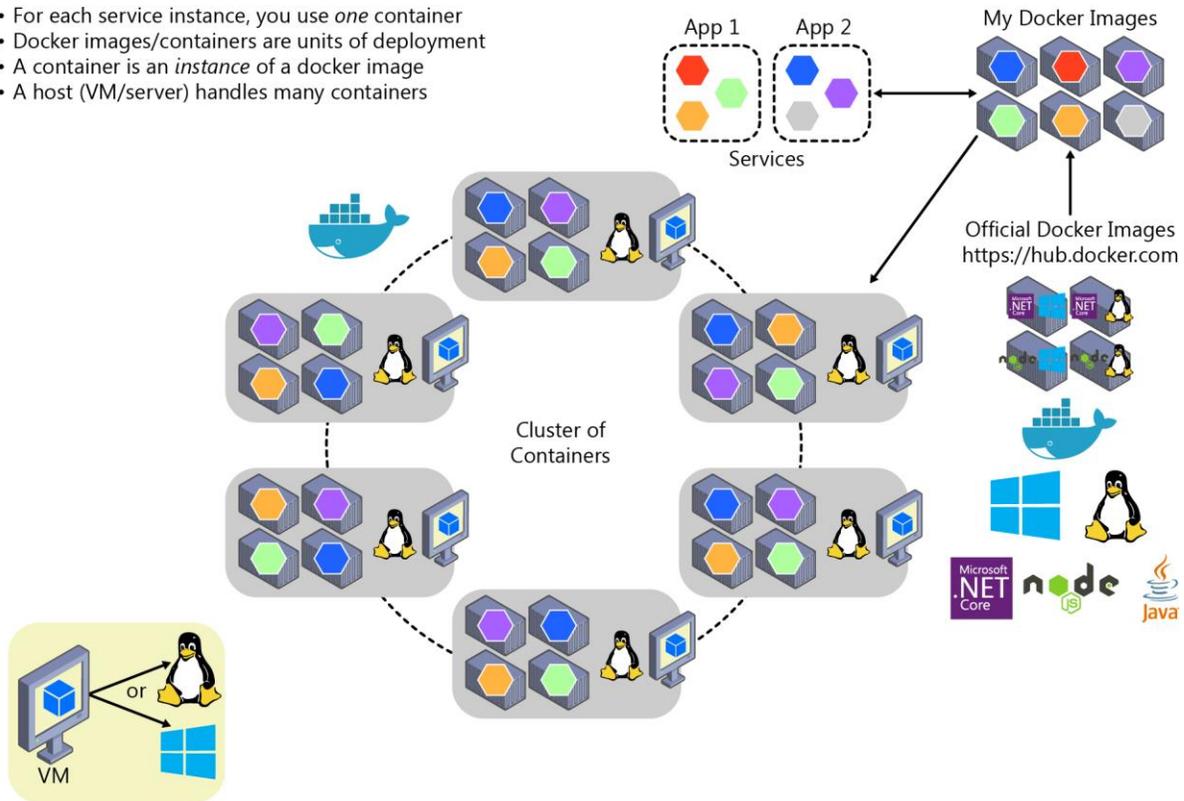- A host (VM/server) handles many containers

Figure 4-6: A cluster of containers

It looks like a logical approach. But how are you handling load balancing, routing, and orchestrating these composed applications?

The Docker command-line interface (CLI) meets the needs of managing one container on one host, but it falls short when it comes to managing multiple containers deployed on multiple hosts for more complex distributed applications. In most cases, you need a management platform that will automatically start containers, suspend them, or shut them down when needed, and ideally also control how they access resources like the network and data storage.

To go beyond the management of individual containers or very simple composed apps and move toward larger enterprise applications with microservices, you must turn to orchestration and clustering platforms.

From an architecture and development point of view, if you are building large, enterprise, microservices-based, applications, it is important to understand the following platforms and products that support advanced scenarios:

- **Clusters and orchestrators**   When you need to scale-out applications across many Docker hosts, such as with a large microservices-based application, it is critical to be able to manage all of those hosts as a single cluster by abstracting the complexity of the underlying platform. That is what the container clusters and orchestrators provide. Examples of orchestrators are Docker Swarm, Mesosphere DC/OS, Kubernetes (the first three available through Azure Container Service and Kubernetes together with the others through Azure Container Service AKS that will replace Azure Container Service in the next future), and Azure Service Fabric.

> **More info** Microsoft has released a preview version of Azure Container Service (AKS) on Oct 24th 2017. AKS makes it easy to run Kubernetes at large scales, with management and maintenance tools. You will know more about AKS later in this book.

- **Schedulers** *Scheduling* means to have the capability for an administrator to launch containers in a cluster so that they also provide a user interface. A cluster scheduler has several responsibilities: to use the cluster's resources efficiently, to set the constraints provided by the user, to efficiently load-balance containers across nodes or hosts, and to be robust against errors while providing high availability.

The concepts of a cluster and a scheduler are closely related, so the products provided by different vendors often provide both sets of capabilities. Table 4-1 lists the most important platform and software choices you have for clusters and schedulers. These clusters are generally offered in public clouds like Azure.

Platform	Description
Google Kubernetes	Kubernetes is an open-source product that provides functionality that ranges from cluster infrastructure and container scheduling to orchestrating capabilities. With it, you can automate deployment, scaling, and operations of application containers across clusters of hosts. Kubernetes provides a container-centric infrastructure that groups application containers into logical units for easy management and discovery.
Azure Service Fabric	Service Fabric is a Microsoft microservices platform for building applications. It is an orchestrator of services and creates clusters of machines. By default, Service Fabric deploys and activates services as processes, but Service Fabric can deploy services in Docker container images. More important, you can mix services in processes with services in containers in the same application. You can develop Service Fabric services in many ways, from using the Service Fabric programming models to deploying guest executables as well as containers. Service Fabric supports prescriptive application models like stateful services and Reliable Actors.
Docker Swarm	Docker Swarm gives you the ability to cluster and schedule Docker containers. By using Swarm, you can turn a pool of Docker hosts into a single, virtual Docker host. Clients can make API requests to Swarm in the same way that they do to hosts, meaning that Swarm makes it easy for applications to scale to multiple hosts. Docker Swarm is a product from Docker, the company. Docker v1.12 or later can run native and built-in Swarm Mode.
Mesosphere DC/OS	Mesosphere Enterprise DC/OS (based on Apache Mesos) is a production-ready platform for running containers and distributed applications. DC/OS works by abstracting a collection of the resources available in the cluster and making those resources available to components built on top of it. Marathon is usually used as a scheduler integrated with DC/OS.

Table 4-1: Software platforms for container clustering, orchestration, and scheduling

Using container-based orchestrators in Azure

Several cloud vendors offer Docker containers support plus Docker clusters and orchestration support, including Azure, Amazon EC2 Container Service, and Google Container Engine. Azure provides Docker cluster and orchestrator support through Azure Container Service, as explained in the next section.

Another choice is to use Azure Service Fabric, which also supports Docker based on Linux and Windows Containers. Service Fabric runs on Azure or any other cloud as well as on-premises.

Using Azure Container Service

A Docker cluster pools multiple Docker hosts and exposes them as a single virtual Docker host, so you can deploy multiple containers into the cluster. The cluster will handle all the complex management plumbing such as scalability and health. Figure 4-7 represents how a Docker cluster for composed applications maps to Container Service.

Container Service provides a way to simplify the creation, configuration, and management of a cluster of VMs that are preconfigured to run containerized applications. Using an optimized configuration of popular open-source scheduling and orchestration tools, Container Service gives you the ability to use your existing skills or draw on a large and growing body of community expertise to deploy and manage container-based applications in Azure.

Container Service optimizes the configuration of popular Docker clustering open-source tools and technologies specifically for Azure. You get an open solution that offers portability for both your containers and your application configuration. You select the size, the number of hosts, and the orchestrator tools, and Container Service handles everything else.

Container Service uses Docker images to ensure that your application containers are fully portable. It supports your choice of open-source orchestration platforms like DC/OS, Kubernetes, and Docker Swarm to ensure that these applications can scale to thousands or even tens of thousands of containers.

With Azure Container Service, you can take advantage of the enterprise-grade features of Azure while still maintaining application portability, including at the orchestration layers.
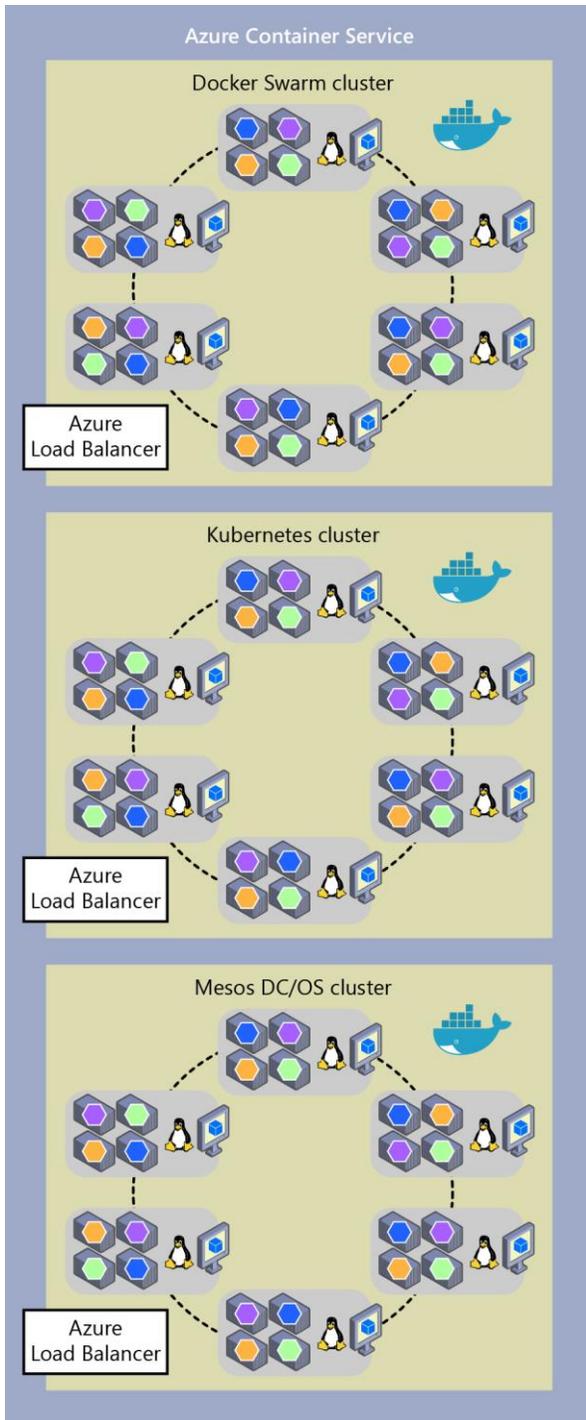
Figure 4-7: Clustering choices in Azure Container Service

Azure Container Service is simply the infrastructure provided by Azure in order to deploy clusters of DC/OS, Kubernetes, or Docker Swarm, but it does not implement any additional orchestrator. Therefore, Azure Container Service is not an orchestrator, as such; it is only an infrastructure that takes advantage of existing open-source orchestrators for containers, as shown in Figure 4-8.
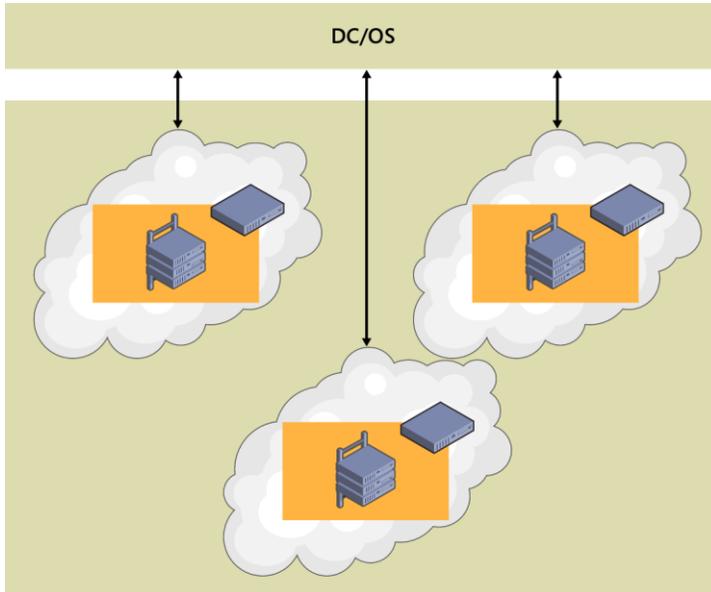
Figure 4-8: Azure Container Service provides the infrastructure for open-source orchestrators

From a usage perspective, the goal of Container Service is to provide a container hosting environment by using popular open-source tools and technologies. To this end, it exposes the standard API endpoints for your chosen orchestrator. By using these endpoints, you can use any software that can communicate to those endpoints. For example, in the case of the Docker Swarm endpoint, you might choose to use the Docker CLI. For DC/OS, you might choose to use the DC/OS CLI.

Getting started with Azure Container Service

To begin using Container Service, you deploy a Container Service cluster from the Azure portal by using an Azure Resource Manager template or the CLI. Available templates include Docker Swarm, Kubernetes, and DC/OS. You can modify the quickstart templates to include additional or advanced Azure configuration.

> **More info** To learn more about deploying a Container Service cluster, on the Azure website, read Deploy an Azure Container Service cluster.

There are no fees for any of the software installed by default as part of ACS. All default options are implemented with open-source software.

Container Service is currently available for Standard A, D, DS, G, and GS series Linux VMs in Azure. You are charged only for the compute instances you choose as well as the other underlying infrastructure resources consumed, such as storage and networking. There are no incremental charges for Container Service itself.

Additional resources

Following are locations where you can find additional information:

- Introduction to Docker container hosting solutions with Container Service:
  https://azure.microsoft.com/documentation/articles/container-service-intro/

- Docker Swarm overview:
  https://docs.docker.com/swarm/overview/

- Swarm mode overview:
  https://docs.docker.com/engine/swarm/

- Mesosphere DC/OS Overview:
  https://docs.mesosphere.com/1.7/overview/

- Kubernetes (the official site):
  http://kubernetes.io/

Using Azure Container Service (AKS)

Azure Container Service has been available since 2015 with support for multiple container orchestrators as described before. During these years Kubernetes has emerged as the open source standard for container orchestration. Microsoft has been working in a new version of Azure Container Service (AKS) that contains a lot of improvements and full coverage of Kubernetes.

Using Azure Container Service AKS, you can take the advantage of the enterprise-grade features of Azure, while still maintaining application portability through Kubernetes and Docker.

Later you will learn through some examples how you can start using AKS.

You can read more about AKS here:

- Azure Container Service (AKS) documentation, quick starts and tutorials

  https://docs.microsoft.com/en-us/azure/aks/

- Azure Container Service (AKS) product information:

  https://azure.microsoft.com/en-us/services/container-service/

If you have your current applications based on Docker Containers on any clouds or on-premises, you can bring these applications directly to Azure through AKS with Kubernetes thanks to the portability that containers provide.

AKS and Kubernetes

The goal with AKS is to provide a container hosting environment by using open-source tools and technologies that are popular among customers. To this end, AKS exposes the standard Kubernetes API endpoints. By using these standard endpoints, you can leverage any tool that is capable of talking to a Kubernetes cluster. For example, you might choose kubectl, helm, or draft.

As mentioned before, Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of host. It was originally created by Google and has been used for a long time in their own systems, that means that is a proved solution in real systems and it wat created and conceptualized using the experience of the needs of these real systems that usually have an important workload.

The main characteristics of Kubernetes are:

- Intelligent distribution of containers across nodes.
- Easy way to determine the scaling.
- Workload management.
- Easy way to manage a big quantity of services and applications.
- High availability.
- Modular and flexible.
- Has been designed to maintain active the same number of containers that means that Kubernetes always tries to maintain the capacity of your system creating new containers in the case of fails.
- Automated rollouts and rollbacks in case of fail.
- Batch capabilities that are very useful in our current online world based even more when these capabilities are based on containers.

In the next chapters you will learn more about Kubernetes and how you use it locally, from your development machine and in Microsoft Azure using Azure Container Service (AKS).

Using Service Fabric

Service Fabric arose from Microsoft's transition from delivering "box" products, which were typically monolithic in style, to delivering services. The experience of building and operating large services at scale, such as Azure SQL Database, Azure Document DB, Azure Service Bus, or Cortana's Backend, shaped Service Fabric. The platform evolved over time as more and more services adopted it. Importantly, Service Fabric had to run not only in Azure but also in standalone Windows Server deployments.

The aim of Service Fabric is to solve the difficult problems of building and running a service and utilizing infrastructure resources efficiently so that teams can solve business problems using a microservices approach.

Service Fabric provides two broad areas to help you build applications that use a microservices approach:

- A platform that provides system services to deploy, scale, upgrade, detect, and restart failed services, discover service location, manage state, and monitor health. These system services in effect provide many of the characteristics of microservices described previously.

- Programming APIs, or frameworks, to help you build applications as microservices: reliable actors and reliable services. Of course, you can choose any code to build your microservice, but these APIs make the job more straightforward, and they integrate with the platform at a deeper level. This way you can get health and diagnostics information, or you can take advantage of reliable state management.

Service Fabric is agnostic with respect to how you build your service, and you can use any technology. However, it provides built-in programming APIs that make it easier to build microservices.

Figure 4-9 demonstrates how you can create and run microservices in Service Fabric either as simple processes or as Docker containers. It is also possible to mix container-based microservices with process-based microservices within the same Service Fabric cluster.
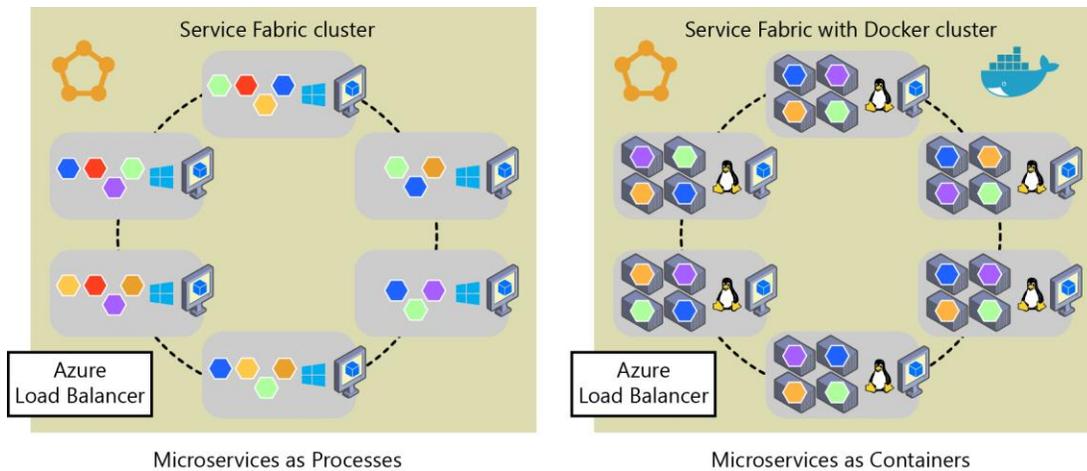
Figure 4-9: Deploying microservices as processes or as containers in Azure Service Fabric

Service Fabric clusters based on Linux and Windows hosts can run Docker Linux containers and Windows Containers.

> **More info** For up-to-date information about containers support in Service Fabric, on the Azure website, read Service Fabric and containers.

Service Fabric is a good example of a platform with which you can define a different logical architecture (business microservices or Bounded Contexts) than the physical implementation. For example, if you implement Stateful Reliable Services in Azure Service Fabric, which are introduced in the next section, "Stateless versus stateful microservices," you have a business microservice concept with multiple physical services.

In Service Fabric, you can group and deploy groups of services as a Service Fabric Application, which is the unit of packaging and deployment for the orchestrator or cluster. Therefore, the Service Fabric Application could be mapped to this autonomous business and logical microservice boundary or Bounded Context, as well.

Service Fabric and containers

With regard to containers in Service Fabric, you also can deploy services in container images within a Service Fabric cluster. Figure 4-11 illustrates that most of the time there will be only one container per service.
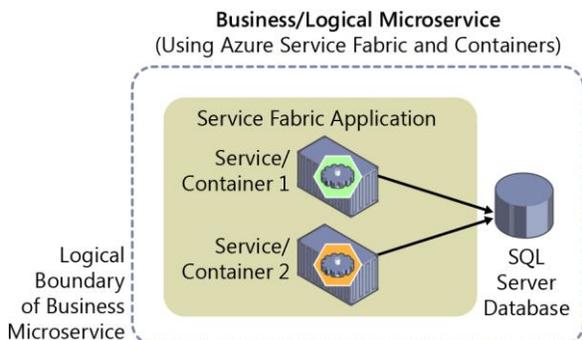

Figure 4-10: Business microservice with several services (containers) in Service Fabric

However, so-called "sidecar" containers (two containers that must be deployed together as part of a logical service) are also possible in Azure Service Fabric. The important thing is that a business

microservice is the logical boundary around several cohesive elements. In many cases, it might be a single service with a single data model, but in some other cases you might have physical several services, as well.

As of this writing (late 2017), in Service Fabric you cannot deploy SF Reliable Stateful Services on containers—you can deploy only guest containers, stateless services, or actor services in containers. But note that you can mix services in processes and services in containers in the same Service Fabric application, as shown in Figure 4-11.
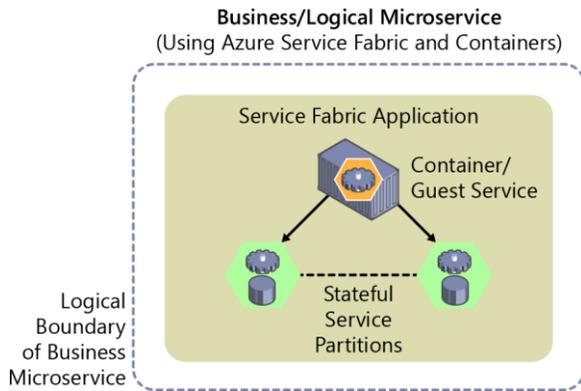

Figure 4-11: Business microservice mapped to a Service Fabric application with containers and stateful services

Support is also different depending on whether you are using Docker containers on Linux or Windows Containers. Support for containers in Service Fabric will be expanding in upcoming releases. For up-to-date news about container support in Service Fabric, on the Azure website, read Service Fabric and containers.

Stateless versus stateful microservices

As mentioned earlier, each microservice (logical Bounded Context) must own its domain model (data and logic). In the case of stateless microservices, the databases will be external, employing relational options like SQL Server, or NoSQL options like MongoDB or Azure DocumentDB.

But the services themselves also can be stateful, which means that the data resides within the microservice. This data might exist not just on the same server, but within the microservice process, in memory, and persisted on drives and replicated to other nodes. Figure 4-12 shows the different approaches.
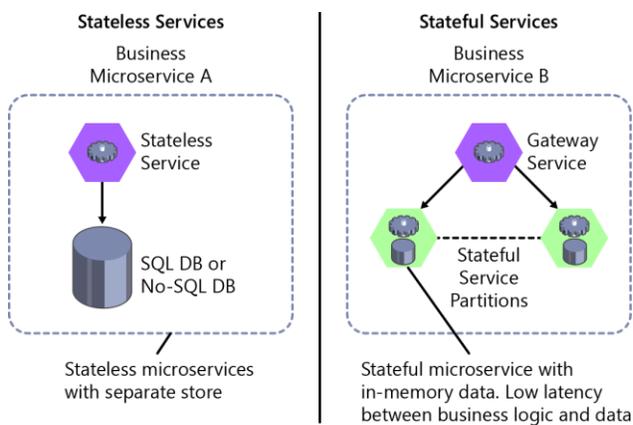

Figure 4-12: Stateless versus stateful microservices

A stateless approach is perfectly valid and is easier to implement than stateful microservices because the approach is similar to traditional and well-known patterns. But stateless microservices impose latency between the process and data sources. They also involve more moving pieces when you are trying to improve performance with additional cache and queues. The result is that you can end up with complex architectures that have too many tiers.

In contrast, stateful microservices can excel in advanced scenarios because there is no latency between the domain logic and data. Heavy data processing, gaming back-ends, databases as a service, and other low-latency scenarios all benefit from stateful services, which provide local state for faster access.

Stateless and stateful services are complementary. For instance, a stateful service could be split into multiple partitions. To access those partitions, you might need a stateless service acting as a gateway service that knows how to address each partition based on partition keys.

Stateful services do have drawbacks. They impose a level of complexity that allows them to scale out. Functionality that would usually be implemented by external database systems must be addressed for tasks such as data replication across stateful microservices and data partitioning. However, this is one of the areas where an orchestrator like Service Fabric with its stateful reliable services can help the most—by simplifying the development and lifecycle of stateful microservices using the Reliable Services API and Reliable Actors.

Other microservice frameworks that allow stateful services, that support the Actor pattern, and that improve fault tolerance and latency between business logic and data are Microsoft Orleans, from Microsoft Research, and Akka.NET. Both frameworks are currently improving their support for Docker.

Note that Docker containers are themselves stateless. If you want to implement a stateful service, you need one of the additional prescriptive and higher-level frameworks noted earlier. However, as of this writing, stateful services in Service Fabric are not supported as containers, only as plain microservices. Reliable services support in containers will be available in upcoming versions of Service Fabric.

Using Azure Container Service (AKS)

You can interact with AKS using your preferred client operating system, here we show how you can do this using Microsoft Windows and using an embedded version of Ubuntu Linux in our Windows, showing how it works using Bash commands.

Prerequisites to have before using AKS are:

- Linux or Mac development machine

- Windows development machine

    o **Developer Mode enabled on Windows**

    o **Windows Subsystem for Linux**

- **Azure-CLI** installed on Windows, Mac or Linux

> **More info** To find complete information about:
>
> Azure-CLI: https://docs.microsoft.com/ cli/azure/overview?view=azure-cli-latest
>
> Windows Subsystem for Linux: https://msdn.microsoft.com/ commandline/wsl/about

Creating the AKS environment in Azure

Creating AKS Environment

There are several ways to create the AKS Environment. It can be done by using Azure-CLI commands or by using the UI in Azure's portal.

Here you can explore some examples using the Azure-CLI to create the cluster and the Azure's portal UI to review the results. There are other tools that you need to install in the development machine, like Kubectl and obviously Docker.

Create the AKS cluster

As of the moment of writing this eBook, AKS was in preview, so if you want to use it when in Preview, you have to activate it in your subscription. To enable AKS you have to use the following command:

```
az provider register -n Microsoft.ContainerService
```

After enabling this preview you can then create the AKS cluster using the following command:

```
az aks create --resource-group MSSampleResourceGroup --name MSSampleClusterK801 --agent-count 1 --generate-ssh-keys --location westus2
```


Figure 4-13: AKS Creation Command

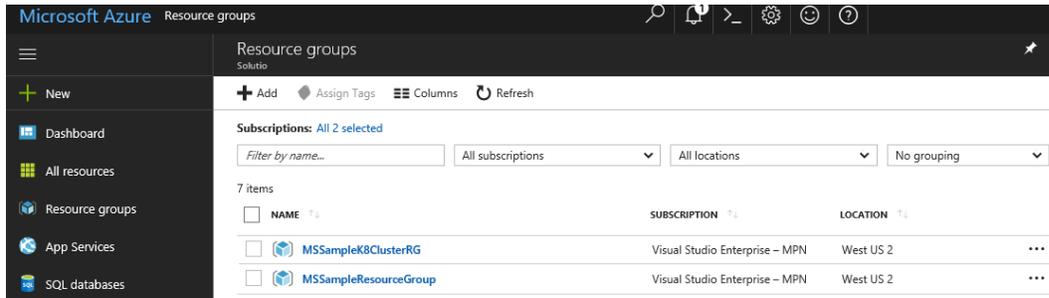After the creation job finish you can see the AKS created in the Azure UI:

The resource group:

Figure 4-14: AKS Resource Group view from Azure.
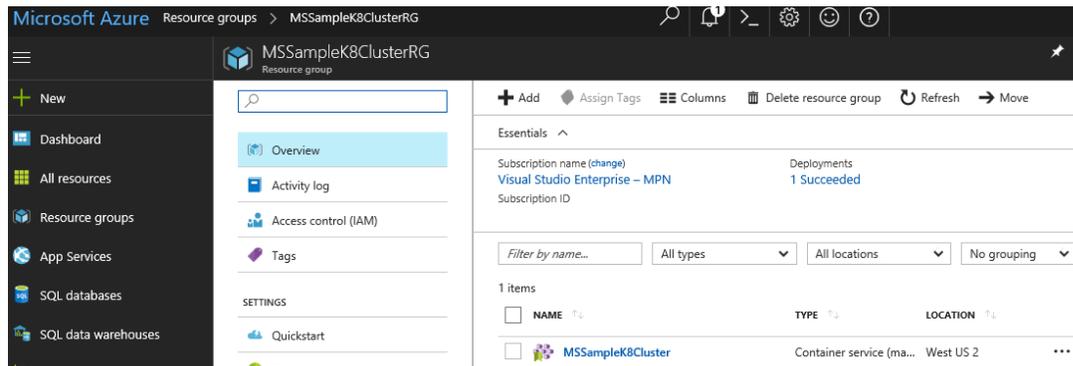
The AKS cluster:

Figure 4-15: AKS view from Azure.

Also, you can view the node created using **Azure-CLI** and **Kubectl,** getting the credentials and invoking the command get nodes from Kubectl:

```
# az aks get-credentials --resource-group MSSampleK8ClusterRG --name MSSampleK8Cluster

# kubectl get nodes
```

Here you can see the commands under the Linux subsystem and the result:

Figure 4-16: View of nodes.

Development environment for Docker apps

Development tools choices: IDE or editor

No matter if you prefer a full and powerful IDE or a lightweight and agile editor, Microsoft has you covered when it comes to developing Docker applications.

Visual Studio Code and Docker CLI (cross-platform tools for Mac, Linux, and Windows)

If you prefer a lightweight, cross-platform editor supporting any development language, you can use Visual Studio Code and Docker CLI. These products provide a simple yet robust experience, which is critical for streamlining the developer workflow. By installing "Docker for Mac" or "Docker for Windows" (development environment), Docker developers can use a single Docker CLI to build apps for both Windows or Linux (runtime environment). Plus, Visual Studio Code supports extensions for Docker with IntelliSense for Dockerfiles and shortcut-tasks to run Docker commands from the editor.

> **Note** To download Visual Studio Code, go to https://code.visualstudio.com/download.
>
> To download Docker for Mac and Windows, go to http://www.docker.com/products/docker.

Visual Studio with Docker Tools (Windows development machine)

We recommend to use Visual Studio 2017 (or later) with Docker Tools enabled that comes built-in. With Visual Studio you can develop, run, and validate your applications directly in the chosen Docker environment. F5 your application (single container or multiple containers) directly into a Docker host with debugging, or press Ctrl+F5 to edit and refresh your app without having to rebuild the container. This is the simplest and more powerful choice for Windows developers creating Docker containers for Linux or Windows.

Visual Studio for Mac (Mac development machine)

You can use Visual Studio for Mac when developing Docker-based applications. Visual Studio for Mac offers a richer IDE for Mac compared to Visual Code for Mac which is a plain code editor.

Language and framework choices

You can develop Docker applications and Microsoft tools with most modern languages. The following is an initial list, but you are not limited to it:

- **.**NET Core and ASP.NET Core
- Node.js
- Golang
- Java
- Ruby
- Python

Basically, you can use any modern language supported by Docker in Linux or Windows.

Inner-loop development workflow for Docker apps

Before triggering the outer-loop workflow spanning the entire DevOps cycle, it all begins on each developer's machine, coding the app itself, using his preferred languages or platforms, and testing it locally (Figure 4-33). But in every case, you will have a very important point in common, no matter what language, framework, or platforms you choose. In this specific workflow, you are always developing and testing Docker containers, but locally.
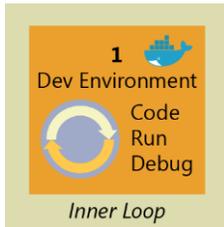

Figure 4-17: Inner-loop development context

The container or instance of a Docker image will contain these components:

- An operating system selection (e.g., a Linux distribution or Windows)

- Files added by the developer (e.g., app binaries)

- Configuration (e.g., environment settings and dependencies)

- Instructions for what processes to run by Docker

You can set up the inner-loop development workflow that utilizes Docker as the process (described in the next section). Take into account that the initial steps to set up the environment is not included, because you need to do that just once.

Building a single app within a Docker container using Visual Studio Code and Docker CLI

Apps are made up from your own services plus additional libraries (dependencies).

Figure 4-18 shows the basic steps that you usually need to carry out when building a Docker app, followed by detailed descriptions of each step.
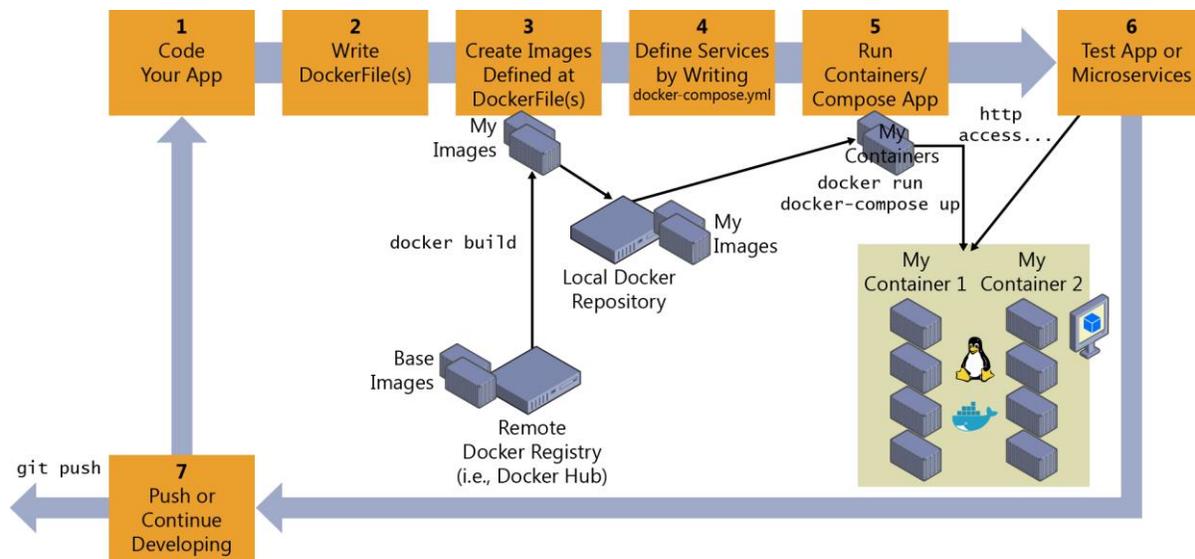
Figure 4-18: High-level workflow for the life cycle for Docker containerized applications using Docker CLI

Step 1: Start coding in Visual Studio Code and create your initial app/service baseline

The way you develop your application is pretty similar to the way you do it without Docker. The difference is that while developing, you are deploying and testing your application or services running within Docker containers placed in your local environment (like a Linux VM or Windows).

Setting up your local environment

With the latest versions of Docker for Mac and Windows, it's easier than ever to develop Docker applications, and the setup is straightforward.

> **More info** For instructions on setting up Docker for Windows, go to https://docs.docker.com/docker-for-windows/.
>
> For instructions on setting up Docker for Mac, go to https://docs.docker.com/docker-for-mac/.

In addition, you'll need a code editor so that you can actually develop your application while using Docker CLI.

Microsoft provides Visual Studio Code, which is a lightweight code editor that is supported on Mac, Windows, and Linux, and provides IntelliSense with support for many languages (JavaScript, .NET, Go, Java, Ruby, Python, and most modern languages), debugging, integration with Git and extensions support. This editor is a great fit for Mac and Linux developers. In Windows, you also can use the full Visual Studio application.

> **More info** For instructions on installing Visual Studio Code for Windows, Mac, or Linux, go to http://code.visualstudio.com/docs/setup/setup-overview/https://docs.docker.com/docker-for-mac/.

You can work with Docker CLI and write your code using any code editor, but if you use Visual Studio Code, it makes it easy to author Dockerfile and docker-compose.yml files in your workspace. Plus, you can run Visual Studio Code tasks from the IDE that will prompt scripts that can be running elaborated operations using Docker CLI underneath.

The Docker extension for VS Code provides the following features:

- Automatic **Dockerfile** and **docker-compose.yml** file generation
- Syntax highlighting and hover tips for **docker-compose.yml** and **Dockerfile** files
- IntelliSense (completions) for **Dockerfile** and **docker-compose.yml** files
- Linting (errors and warnings) for **Dockerfile** files
- Command Palette (F1) integration for the most common Docker commands
- Explorer integration for managing Images and Containers
- Deploy images from DockerHub and Azure Container Registries to Azure App Service

To install the Docker extension press Ctrl+Shift+P, type **ext install**, and then run the Extensions: Install Extension command to bring up the Marketplace extension list. Next, type **docker** to filter the results, and then select the Docker Support extension, as depicted in Figure 4-19.
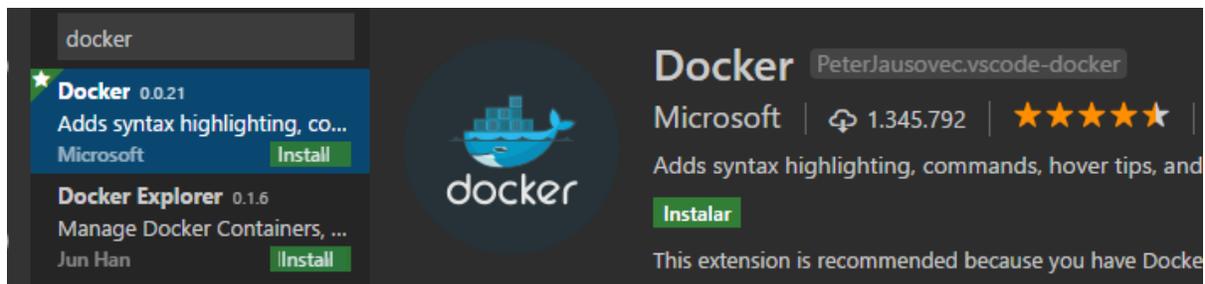

Figure 4-19: Installing the Docker Extension in Visual Studio Code

Step 2: Create a DockerFile related to an existing image (plain OS or dev environments like .NET Core, Node.js, and Ruby)

You will need a DockerFile per custom image to be built and per container to be deployed, therefore, if your app is made up of a single custom service, you will need a single DockerFile. But, if your app is composed of multiple services (as in a microservices architecture), you'll need one Dockerfile per service.

The DockerFile is usually placed within the root folder of your app or service and contains the required commands so that Docker knows how to set up and run that app or service. You can create your DockerFile and add it to your project along with your code (node.js, .NET Core, etc.), or, if you are new to the environment, take a look at the following Tip.

> **Tip** You can use the Docker extension to guide you when using the Dockers files and docker-compose.yml files related to your Docker containers. Eventually, you will probably write this kind of files without this tools, but using the Docker extension is a good starting point that will accelerate your learning curve.

In figure 4-37 you can see how a docker-compose file is added by using the Docker Extension for VS Code.
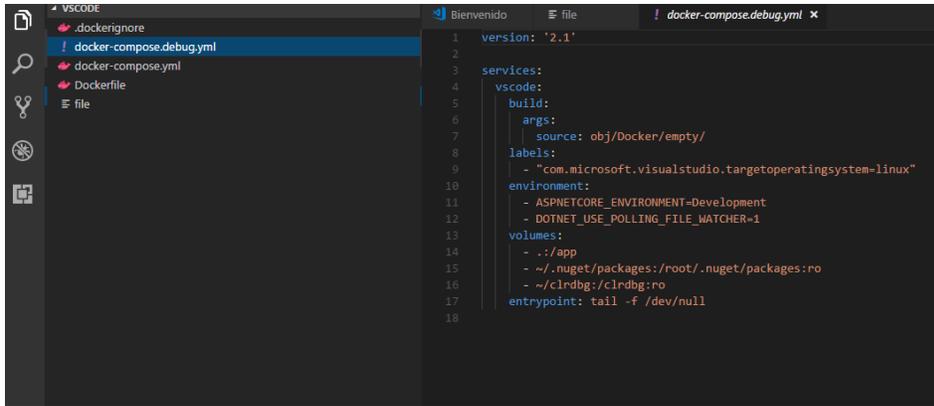
Figure 4-20: Docker files added using the Add Docker files to Workspace command

When you add DockerFile, you specify what base Docker image you'll be using (like using "FROM Microsoft/aspnetcore"). You usually will build your custom image on top of a base image that you can get from any official repository at the [Docker Hub registry](#) (like an [image for .NET Core](#) or one [for Node.js](#)).

Option A: Use an existing official Docker image

Using an official repository of a language stack with a version number ensures that the same language features are available on all machines (including development, testing, and production).

Following is a sample DockerFile for a .NET Core container:

```
# Base Docker image to use
FROM microsoft/aspnetcore:2.0

# Set the Working Directory and files to be copied to the image
ARG source
WORKDIR /app
COPY ${source:-bin/Release/PublishOutput} .

# Configure the listening port to 80 (Internal/Secured port within Docker host)
EXPOSE 80

# Application entry point
ENTRYPOINT ["dotnet", "MyCustomMicroservice.dll"]
```

In this case, it is using the version 2.0 of the official ASP.NET Core Docker image for Linux named `microsoft/aspnetcore:2.0`. For further details, consult the [ASP.NET Core Docker Image page](#) and the [.NET Core Docker Image page](#). You also could be using another comparable image like `node:4-onbuild` for Node.js, or many other preconfigured images for development languages, which are available at [Docker Hub](#).

In the DockerFile, you can instruct Docker to listen to the TCP port that you will use at runtime (such as port 80).

There are other lines of configuration that you can add in the DockerFile depending on the language/framework you are using, so Docker knows how to run the app. For instance, you need the ENTRYPOINT line with `["dotnet", "MyCustomMicroservice.dll"]` to run a .NET Core app, although you can have multiple variants depending on the approach to build and run your service. If you're using the SDK and dotnet CLI to build and run the .NET app, it would be slightly different. The bottom line is that the ENTRYPOINT line plus additional lines will be different depending on the language/platform you choose for your application.

> **More info** For information about building Docker images for .NET Core applications, go to https://docs.microsoft.com/dotnet/articles/core/docker/building-net-docker-images.
>
> To learn more about building your own images, go to https://docs.docker.com/engine/tutorials/dockerimages/.

Multiplatform/Multi-Architecture image repositories

As Windows containers become more prevalent, a single image repository can contain platform variants, such as a Linux and Windows image. This is a new feature in Docker (since 2017) that makes it possible for vendors to use a single image repository to cover multiple platforms, such as microsoft/aspdotnetcore repository, which is available at DockerHub registry.

When pulling the microsoft/aspdotnetcore image from a Windows host it will pull the Windows variant, whereas when pulling the same image name from a Linux host it pulls the Linux variant.

Option B: Create your base image from scratch

You can create your own Docker base image from scratch as explained in this article from Docker. This is a scenario that is probably not best for you if you are just starting with Docker, but if you want to set the specific bits of your own base image, you can do it.

Step 3: Create your custom Docker images embedding your service in it

For each custom service that comprises your app, you'll need to create a related image. If your app is made up of a single service or web app, you'll need just a single image.

> **Note** When taking into account the "outer-loop DevOps workflow," the images will be created by an automated build process whenever you push your source code to a Git repository (Continuous Integration) so the images will be created in that global environment from your source code.
>
> But, before we consider going to that outer-loop route, we need to ensure that the Docker application is actually working properly so that they don't push code that might not work properly to the source control system (Git, etc.).
>
> Therefore, each developer first needs to do the entire inner-loop process to test locally and continue developing until they want to push a complete feature or change to the source control system.

To create an image in your local environment and using the DockerFile, you can use the `docker build` command, as demonstrated in Figure 4-21 (you also can run `docker-compose up --build` for applications composed by several containers/services).

```
PS C:\dev\netcore-webapi-microservice-docker> docker build -t cesardl/netcore-webapi-microservice-docker:first .
Sending build context to Docker daemon 1.148 MB
Step 1 : FROM microsoft/dotnet:latest
latest: Pulling from microsoft/dotnet
5c90d4a2d1a8: Downloading [=================>                                 ] 18.34 MB/51.35 MB
ab30c63719b1: Downloading [================================================> ] 18.48 MB/18.55 MB
c6072700a242: Downloading [====================>                              ] 18.34 MB/42.53 MB
121d7eef6c20: Waiting
eb57cf4f29ee: Waiting
b2c5ae2d325b: Waiting
```

Figure 4-21: Running `docker build`

Optionally, instead of directly running `docker build` from the project's folder, you first can generate a deployable folder with the .NET libraries needed by using the `run dotnet publish` command, and then run `docker build`.

In this example, this creates a Docker image with the name `cesardl/netcore-webapi-microservice-docker:first` (`:first` is a tag, like a specific version). You can take this step for each custom image you need to create for your composed Docker application with several containers.

You can find the existing images in your local repository (your development machine) by using the `docker images` command, as illustrated in Figure 4-22.

```
PS C:\dev\netcore-webapi-microservice-docker> docker images
REPOSITORY                                  TAG       IMAGE ID        CREATED          SIZE
cesardl/netcore-webapi-microservice-docker  first     384c4ac1809b    4 minutes ago    579.8 MB
microsoft/dotnet                            latest    49aaf5daa850    30 hours ago     548.6 MB
ubuntu                                      latest    cf62323fa025    5 days ago       125 MB
hello-world                                 latest    c54a2cc56cbb    12 days ago      1.848 kB
```

Figure 4-22: Viewing existing images using `docker images`

Step 4: Define your services in docker-compose.yml when building a composed Docker app with multiple services

With the docker-compose.yml file you can define a set of related services to be deployed as a composed application with the deployment commands explained in the next step section.

You need to create that file in your main or root solution folder; it should have a similar content to the following docker-compose.yml file:

```yaml
version: '2'
services:
  web:
    build: .
    ports:
      - "81:80"
    volumes:
      - .:/code
    depends_on:
      - redis
  redis:
    image: redis
```

In this particular case, this file defines two services: the `web` service (your custom service) and the `redis` service (a popular cache service). Each service will be deployed as a container, so we need to use a concrete Docker image for each. For this particular web service, the image will need to do the following:

- Build from the DockerFile in the current directory

- Forward the exposed port 80 on the container to port 81 on the host machine

- Mount the project directory on the host to /code within the container, making it possible for you to modify the code without having to rebuild the image

- Link the web service to the redis service

The `redis` service uses the [latest public redis image](#) pulled from the Docker Hub registry. [redis](#) is a very popular cache system for server-side applications.

Step 5: Build and run your Docker app

If your app has only a single container, you just need to run it by deploying it to your Docker Host (VM or physical server). However, if your app is made up of multiple services, you need to *compose it*, too. Let's see the different options.

Option A: Run a single container or service

You can run the Docker image by using the `docker run` command, as shown here:

docker run –t -d -p **80:5000** cesardl/netcore-webapi-microservice-docker:first

Note that for this particular deployment, we'll be redirecting requests sent to port 80 to the internal port 5000. Now, the application is listening on the external port 80 at the host level.

Option B: Compose and run a multiple-container application

In most enterprise scenarios, a Docker application will be composed of multiple services. For these cases, you can run the command `docker-compose up` (Figure 4-23), which will use the docker-compose.yml file that you might have created previously. Running this command deploys a composed application with all of its related containers.

```
PS C:\Dev\WebApplication> docker-compose up
Recreating webapplication_webapplication_1
Attaching to webapplication_webapplication_1
webapplication_1  |  Hosting environment: Production
webapplication_1  |  Content root path: /app
webapplication_1  |  Now listening on: http://*:80
webapplication_1  |  Application started. Press Ctrl+C to shut down.
```

Figure 4-23: Results of running the "docker-compose up" command

After you run `docker-compose up`, you deploy your application and its related container(s) into your Docker Host, as illustrated in Figure 4-24, in the VM representation.
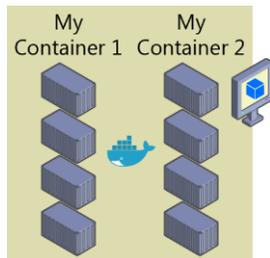

Figure 4-24: VM with Docker containers deployed

Step 6: Test your Docker application (locally, in your local CD VM)

This step will vary depending on what your app is doing.

In a very simple .NET Core Web API "Hello World" deployed as a single container or service, you'd just need to access the service by providing the TCP port specified in the DockerFile.

If `localhost` is not turned on, to navigate to your service, find the IP address for the machine by using this command:

```
docker-machine ip your-container-name
```

On the Docker host, open a browser and navigate to that site; you should see your app/service running, as demonstrated in Figure 4-25.

Figure 4-25: Testing your Docker application locally using localhost

Note that it is using port 80, but internally it was being redirected to port 5000, because that's how it was deployed with `docker run`, as explained earlier.

You can test this by using CURL from the terminal. In a Docker installation on Windows, the default IP is 10.0.75.1, as depicted in Figure 4-26.

Figure 4-26: Testing a Docker application locally by using CURL

Debugging a container running on Docker

Visual Studio Code supports debugging Docker if you're using Node.js and other platforms like .NET Core containers.

You also can debug .NET Core or .NET Framework containers in Docker when using Visual Studio for Windows or Mac, as described in the next section.

> **More info:** To learn more about debugging Node.js Docker containers, go to
> https://blog.docker.com/2016/07/live-debugging-docker/ and https://blogs.msdn.microsoft.com/
> user_ed/2016/02/27/visual-studio-code-new-features-13-big-debugging-updates-rich-object-
> hover-conditional-breakpoints-node-js-mono-more/.

Using Visual Studio Tools for Docker (Visual Studio on Windows)

The developer workflow when using Visual Studio Tools for Docker is similar to the workflow when using Visual Studio Code and Docker CLI (in fact, it is based on the same Docker CLI), but it is easier to get started, simplifies the process, and provides greater productivity for the build, run, and compose tasks. It's also able to execute and debug your containers via simple actions like F5 and Ctrl+F5 from Visual Studio. Even more, with Visual Studio 2017, in addition to being able to run and debug a single container, you also can run and debug a group of containers (a whole solution) at the same time if they are defined in the same docker-compose.yml file at the solution level.

Configuring your local environment

With the latest versions of Docker for Windows, it is easier than ever to develop Docker applications because the setup is straightforward, as explained in the following references.

> **More info:** To learn more about installing Docker for Windows, go to
> https://docs.docker.com/docker-for-windows/.

Using Docker Tools in Visual Studio 2017

When you add Docker support to a service project in your solution (see Figure 4-27), Visual Studio is not just adding a DockerFile file to your project, it also is adding a service section in your solution's docker-compose.yml files (or creating the files if they didn't exist). It's an easy way to begin composing your multicontainer solution; you then can open the docker-compose.yml files and update them with additional features.
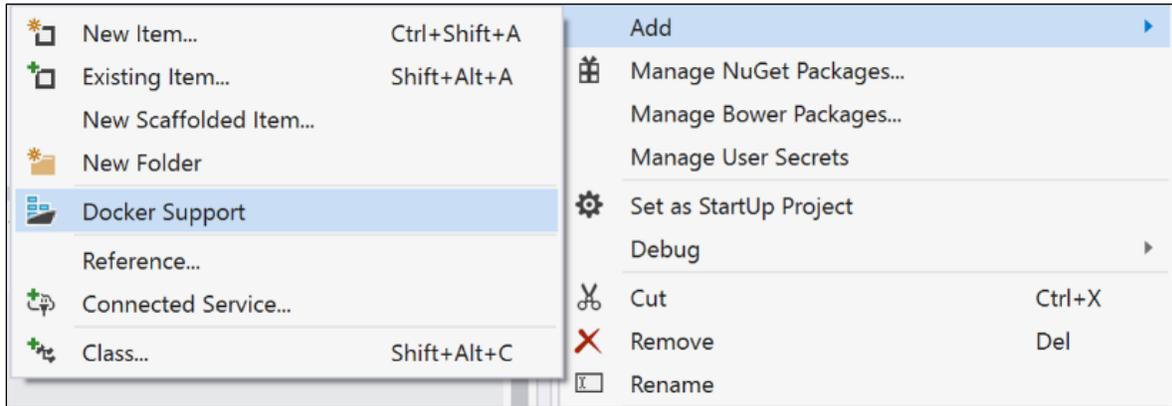

Figure 4-27: Turning on Docker Solution support in a Visual Studio 2017 project

This action not only adds the DockerFile to your project, it also adds the required configuration lines of code to a global docker-compose.yml set at the solution level.

You also can turn on Docker support when creating an ASP.NET Core project in Visual Studio 2017, as shown in Figure 4-28.
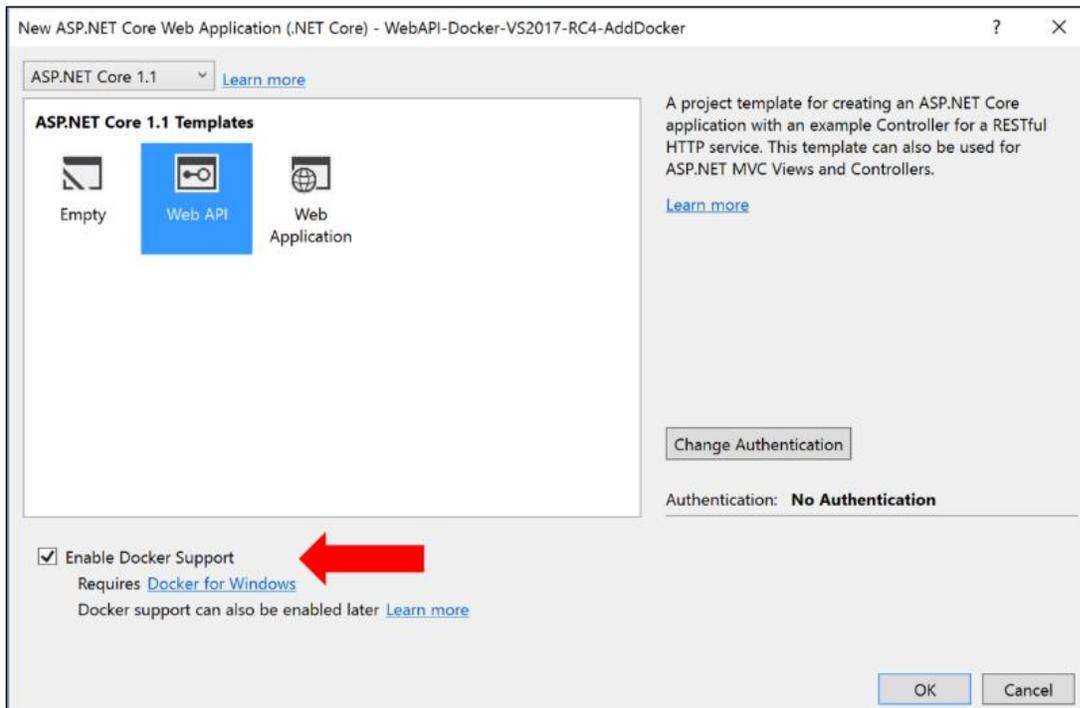

Figure 4-28: Turning on Docker support when creating a project

After you add Docker support to your solution in Visual Studio, you also will see a new node tree in Solution Explorer with the added docker-compose.yml files, as depicted in Figure 4-29.
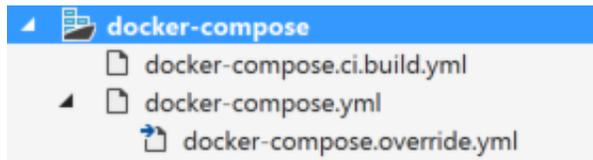

Figure 4-29: docker-compose.yml files in VS Solution Explorer

You could deploy a multicontainer application by using a single docker-compose.yml file when you run `docker-compose up`; however, Visual Studio adds a group of them, so you can override values depending on the environment (development versus production) and the execution type (release versus debug). This capability will be better explained in later chapters.

> **More info:** For further details on the services implementation and use of Visual Studio Tools for Docker, read the following articles:
>
> Build, debug, update, and refresh apps in a local Docker container: https://azure.microsoft.com/documentation/articles/vs-azure-tools-docker-edit-and-refresh/
>
> Deploy an ASP.NET container to a remote Docker host: https://azure.microsoft.com/documentation/articles/vs-azure-tools-docker-hosting-web-apps-in-docker/

Using Windows PowerShell commands in a DockerFile to set up Windows Containers (Docker standard based)

With Windows Containers, you can convert your existing Windows applications to Docker images and deploy them with the same tools as the rest of the Docker ecosystem.

To use Windows Containers, you just need to write Windows PowerShell commands in the DockerFile, as demonstrated in the following example:

```
FROM microsoft/windowsservercore
LABEL Description="IIS" Vendor="Microsoft" Version="10"
RUN powershell -Command Add-WindowsFeature Web-Server
CMD [ "ping", "localhost", "-t" ]
```

In this case, we're using Windows PowerShell to install a Windows Server Core base image as well as IIS.

In a similar way, you also could use Windows PowerShell commands to set up additional components like the traditional ASP.NET 4.x and .NET 4.6 or any other Windows software, as shown here:

```
RUN powershell add-windowsfeature web-asp-net45
```

Building.NET Core 2.0 applications deployed as Linux containers into AKS/Kubernetes clusters

In late 2017, Microsoft released a preview version of a new managed Kubernetes service, **Azure Container Service (AKS)**, which is a separate product from the existing ACS that offers support for multiple orchestrators Kubernetes, Mesos DC/OS and Docker Swarm.

> **More info:** To know more about the Microsoft AKS announce you can go to the ACS docs:
>
> https://docs.microsoft.com/en-us/azure/container-service/kubernetes/container-service-intro-kubernetes

The **new features in AKS:** An Azure-hosted control plane, automated upgrades, self-healing, user configurable scaling, and a simpler user experience for both developers and cluster operators. At launch, AKS will default to using Kubernetes 1.7.7, the software's latest stable release, and customers can opt into using the new 1.8 beta if they choose to do so.

In the following examples you can see the creation of an application, using Visual Studio 2017 with .NET Core 2.0, that runs on Linux and deploys to an AKS Cluster in Azure.

Creating the .NET Core 2.0 Project using Visual Studio 2017

.NET Core is a general-purpose development platform maintained by Microsoft and the .NET community on GitHub. It is cross-platform, supporting Windows, macOS and Linux, and can be used in device, cloud, and embedded/IoT scenarios.

For this example, we use a simple project based on a Visual Studio Web API template, so you don´t need any additional knowledge to create the sample, you only have to create a project using a standard template that include all the elements to run a small project with a REST API under Microsoft .NET Core 2.0 technology.

To create the sample project, you have to select New->Project on Visual Studio and ASP.NET Core Web Application
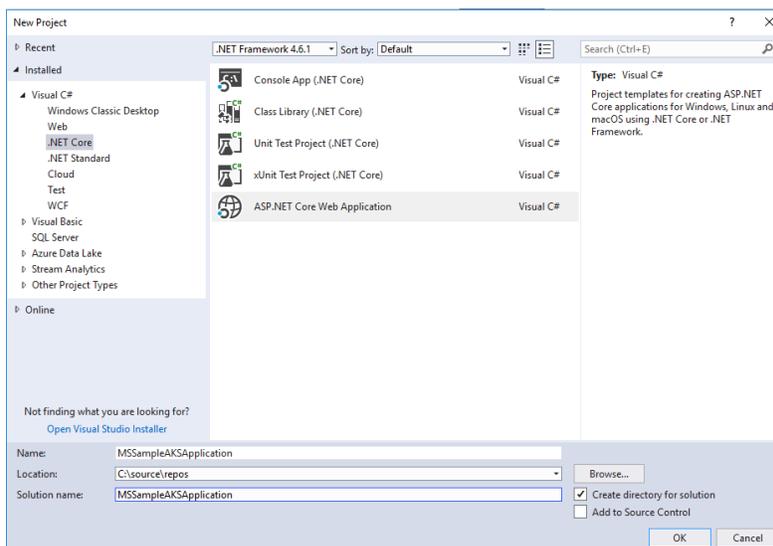

Figure 4-30: Creating ASP.NET Core Application

Visual Studio will show a list with Templates for Web Projects. For our example you must select **Web API,** that means you will create an ASP.NET Web API Application.

Verify that you have selected ASP.NET Core 2.0 as the framework. The .NET Core 2.0 is included in the last version of Visual Studio 2017 and is automatically installed and configured for you when you install Visual Studio 2017.

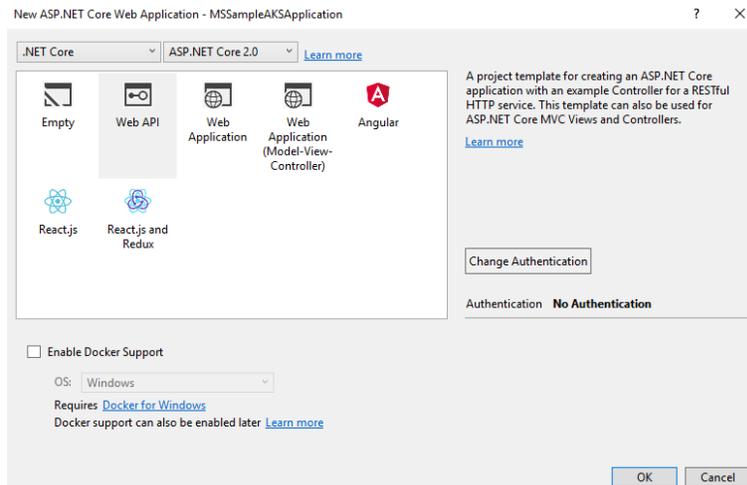If you have any previous version of .NET Core you can download and install the 2.0 version from https://www.microsoft.com/net/download/core#/sdk

Figure 4-31: Selecting .NET CORE 2.0 and Web API project type

You can enable Docker support at the moment of the project creation in the previous step, or later that means that you can Dockerize your project in any moment. To enable the Docker support after the project creation you must right click on the solution file and selecting on the context menu **add - >Docker support**
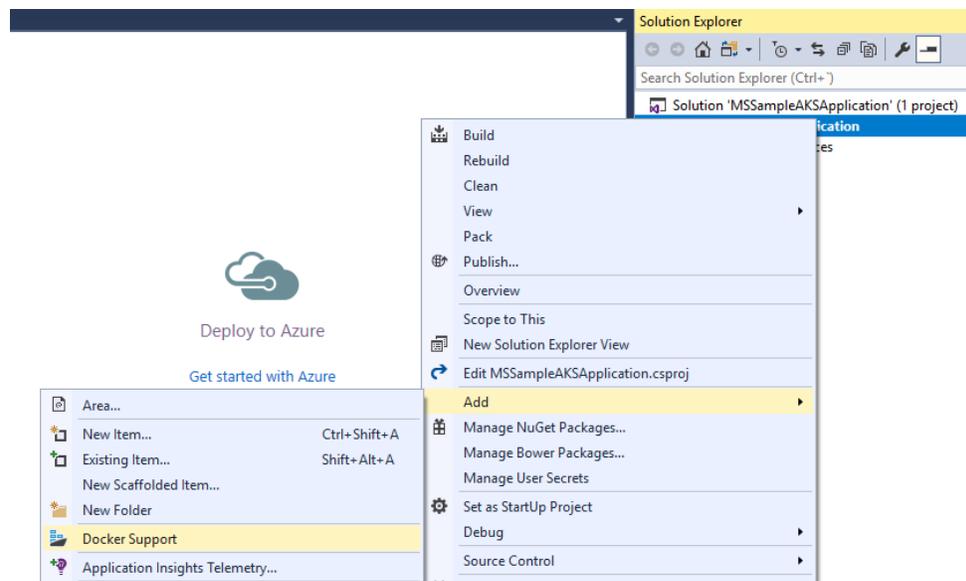

Figure 4-32: Enabling Docker support

You can select Windows or Linux, in our case you must select **Linux,** because as of late 2017, AKS still doesn't support Windows Containers.
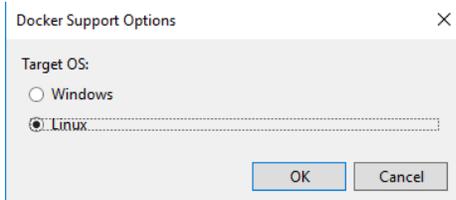

Figure 4-33: Selecting Linux Containers.

With these simple steps you will have your application based on .NET Core 2.0 running on a Linux Container.

As you can see the integration between Visual Studio 2017 and Docker is totally oriented to the developer's productivity.

Now you can run your application with the key **F5** or using the Play Button

After running the project, you can check the images using the Docker images command and in the list, you will see the mssampleapplication image that has been create with the execution an automatic deploy of our project using Visual Studio 2017.

```
# Docker images
```


Figure 4-34: View of docker images

Register the Solution in the Azure Container Registry

We have to upload the image to any Docker registry, like Azure Container Registry (ACR) or Docker Hub so the images will be deployed to the AKS cluster from that image registry. In this case, we're uploading the image to Azure Container Registry.

Create the image in Release

We will create the image in Release Mode (ready for production) changing to Release as shown here and executing the application like you have done before
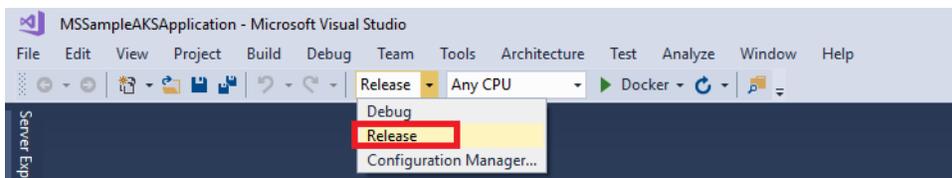

Figure 4-35: Selecting Release Mode

If you execute the Docker Image commad you will see the both images created Debug and Relase Mode.

Create a new Tag for the Image

Each container image needs to be tagged with the loginServer name of the registry. This tag is used for routing when pushing container images to an image registry.

You can view the name of the loginserver in two ways, accesing to Acure UI and taking the information from the Azure ACR Registry
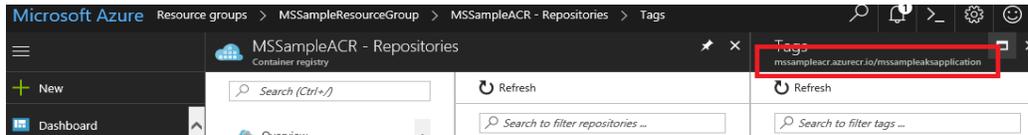

Figure 4-36: View of the name of the Registry

Or running the command

```
# az acr list --resource-group MSSampleResourceGroup --query "[].{acrLoginServer:loginServer}" --output table
```


Figure 4-37: List the name of the registry using powershell

In both cases you will obtain the name, in our example **mssampleacr.azurecr.io**

Now you can Tag the image, taking the latest image (Release image) using the command:

```
# docker tag mssampleaksapplication:latest mssampleacr.azurecr.io/mssampleaksapplication:v1
```


Figure 4-38: Apply Tag Command

After run the Tag Command, if you use the Docker images command to see the images, you will see how the image has changed to the new Tag.

Figure 4-39: View of images

Push the image into the Azure ACR

Now you can Push the image into the Azure ACR, for that you have to execute the command:

```
# docker push mssampleacr.azurecr.io/mssampleaksapplication:v1
```

This command starts uploading and spend a little time always showing you the progress.

Figure 4-40: Uploading the image to the ACR

When the upload has finished you will see the next information:

```
root@DESKTOP-B8UOGOK:/mnt/c/Windows/System32# docker push mssampleacr.azurecr.io/mssampleaksapplication:v1
The push refers to a repository [mssampleacr.azurecr.io/mssampleaksapplication]
4ec82b575b8f: Pushed
74317ca8b71c: Pushed
bf69e01727ff: Pushed
edc89bc130b6: Pushed
f38d5d54bab8: Pushed
3bda74a84fb6: Pushed
a75caa09eb1f: Pushed
v1: digest: sha256:6155b34d672cdf406a3b3b6d7d39c6a94eca53a8c60ea93651faae6d7e8e289c size: 1790
```

Figure 4-41: View of nodes

The next step is deploy your container into your AKS Kubernetes cluster, for that you need a file (**.yml deploy file**), that four our example will contain:

```
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: mssamplesbook
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: mssample-kub-app
    spec:
      containers:
        - name: mssample-services-app
          image: mssampleacr.azurecr.io/mssampleaksapplication:v1
          ports:
            - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: mssample-kub-app
spec:
  ports:
    - name: http-port
      port: 80
      targetPort: 80
  selector:
    app: mssample-kub-app
  type: LoadBalancer
```

> **More info:** You can find more information about yml files here:
>
> https://kubernetes.io/docs/user-guide/kubectl-cheatsheet/

Now you can deploy using **Kubectl,** first you must get the credentials to the AKS Cluster, you can see how create the AKS Cluster that we are using in this sample in **Azure Kubernetes Managed Service** later on this book.

```
# az aks get-credentials --resource-group MSSampleResourceGroupAKS --name
mssampleclusterk801
```

```
root@DESKTOP-B8UOGOK:/mnt/c/Windows/System32# az aks get-credentials --resource-group MSSampleK8ClusterRG --name MSSampl
eK8Cluster
Merged "MSSampleK8Cluster" as current context in /root/.kube/config
```

Figure 4-42: getting credentials

Then you can use the Kubectl create command to launch the deployment.

Figure 4-43: Deploy in Kubernetes

When the Deployment has been finished you can access to the Kubernetes console using a local proxy that you can temporally access using this command:

`# az aks browse --resource-group MSSampleResourceGroupAKS --name mssampleclusterk801`

and accessing to the url http://127.0.0.1:8001
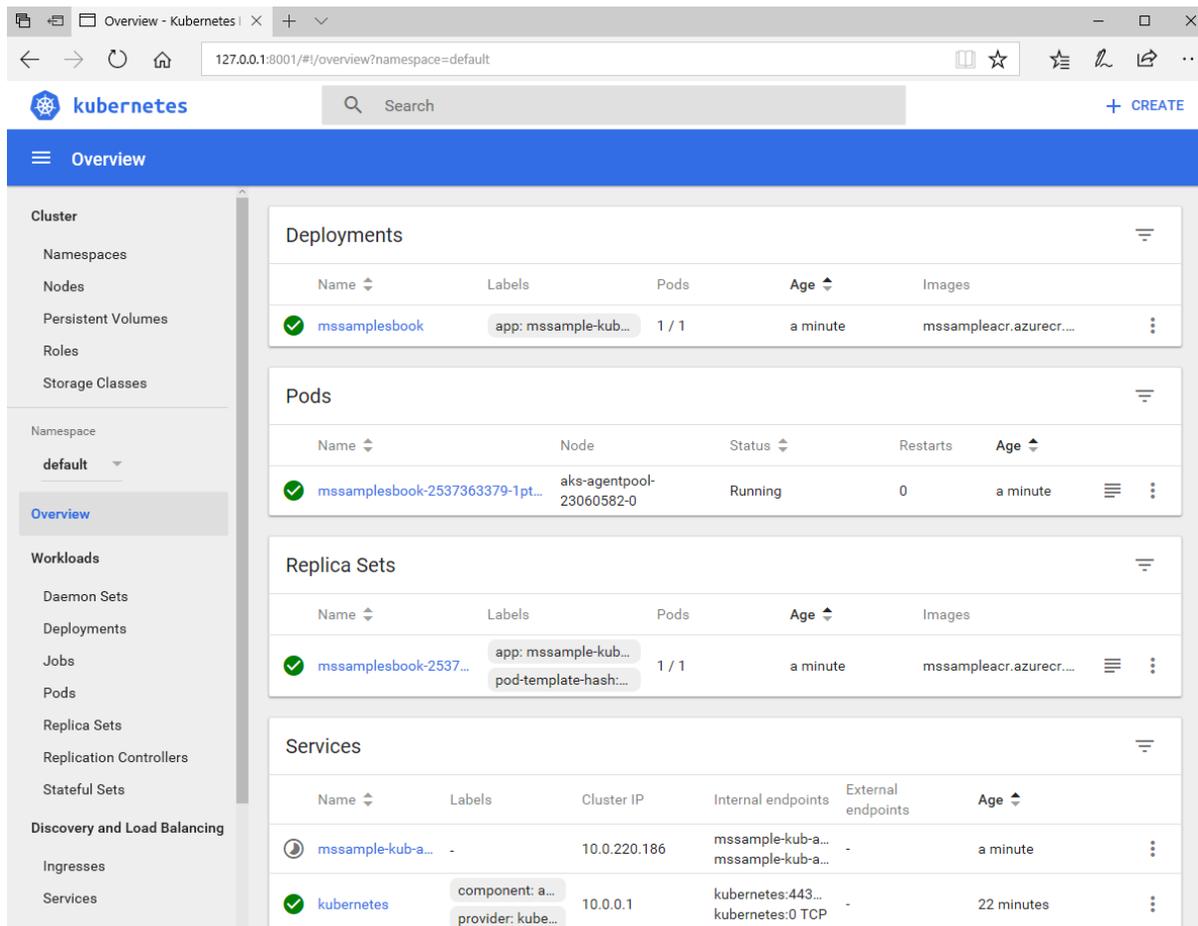
Figure 4-44: View Kubernetes cluster information

Now you have your application deployed on Azure using a Linux Container and a AKS Kubernetes Cluster and you can access it using the public IP of your Service specified in the Services information.

Docker application DevOps workflow with Microsoft tools

Microsoft Visual Studio, Visual Studio Team Services, Team Foundation Server, and Application Insights provide a comprehensive ecosystem for development and IT operations that give your team the tools to manage projects and rapidly build, test, and deploy containerized applications.

With Visual Studio and Visual Studio Team Services in the cloud, along with Team Foundation Server on-premises, development teams can productively build, test, and release containerized applications directed toward any platform (Windows or Linux).

Microsoft tools can automate the pipeline for specific implementations of containerized applications—Docker, .NET Core, or any combination with other platforms—from global builds and Continuous Integration (CI) and tests with Visual Studio Team Services or Team Foundation Server, to Continuous Deployment (CD) to Docker environments (Development, Staging, Production), and to

transmit analytics information about the services to the development team through Application Insights. Every code commit can initiate a build (CI) and automatically deploy the services to specific containerized environments (CD).

Developers and testers can easily and quickly provision production-like development and test environments based on Docker by using templates in Microsoft Azure.

The complexity of containerized application development increases steadily depending on the business complexity and scalability needs. A good example of this are applications based on microservices architectures. To succeed in such an environment, your project must automate the entire life cycle—not only the build and deployment, but it also must manage versions along with the collection of telemetry. Visual Studio Team Services and Azure offer the following capabilities:

- Visual Studio Team Services/Team Foundation Server source code management (based on Git or Team Foundation Version Control), Agile planning (Agile, Scrum, and CMMI are supported), CI, release management, and other tools for Agile teams.

- Visual Studio Team Services/Team Foundation Server include a powerful and growing ecosystem of first- and third-party extensions with which you easily can construct a CI, build, test, delivery, and release management pipeline for microservices.

- Run automated tests as part of your build pipeline in Visual Studio Team Services.

- Visual Studio Team Services can tighten the DevOps life cycle with delivery to multiple environments, not just for production environments, but also for testing, including A/B experimentation, canary releases, and so on.

- Organizations easily can provision Docker containers from private images stored in Azure Container Registry along with any dependency on Azure components (Data, PaaS, etc.) using Azure Resource Manager templates with tools with which they are already comfortable working.

Steps in the outer-loop DevOps workflow for a Docker application

Figure 5-1 presents an end-to-end depiction of the steps comprising the DevOps outer-loop workflow.
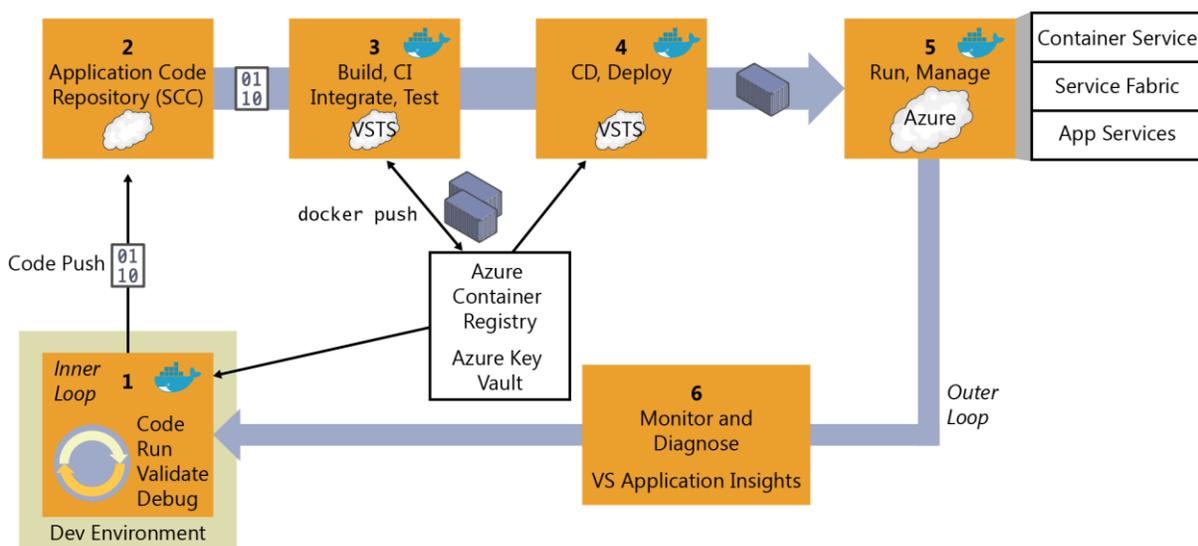

Figure 5-1: DevOps outer-loop workflow for Docker applications with Microsoft tools

Now, let's examine each of these steps in greater detail.

Step 1: Inner-loop development workflow

This step is explained in detail in Chapter 4, but, to recap, here is where the outer-loop begins, the moment at which a developer pushes code to the source control management system (like Git) initiating CI pipeline actions.

Step 2: Source-Code Control integration and management with Visual Studio Team Services and Git

At this step, you need to have a version-control system to gather a consolidated version of all the code coming from the different developers in the team.

Even though source-code control (SCC) and source-code management might seem second-nature to most developers, when creating Docker applications in a DevOps life cycle, it is critical to emphasize that you must not submit the Docker images with the application directly to the global Docker Registry (like Azure Container Registry or Docker Hub) from the developer's machine. On the contrary, the Docker images to be released and deployed to production environments must be created solely on the source code that is being integrated in your global build or CI pipeline based on your source-code repository (like Git).

The local images generated by the developers themselves should be used just by the developers when testing within their own machines. This is why it is critical to have the DevOps pipeline activated from the SCC code.

Visual Studio Team Services and Team Foundation Server support Git and Team Foundation Version Control. You can choose between them and use it for an end-to-end Microsoft experience. However, you also can manage your code in external repositories (like GitHub, on-premises Git repositories, or Subversion) and still be able to connect to it and get the code as the starting point for your DevOps CI pipeline.

Step 3: Build, CI, Integrate, and Test with Visual Studio Team Services and Docker

CI has emerged as a standard for modern software testing and delivery. The Docker solution maintains a clear separation of concerns between the development and operations teams. The immutability of Docker images ensures a repeatable deployment between what's developed, tested through CI, and run in production. Docker Engine deployed across the developer laptops and test infrastructure makes the containers portable across environments.

At this point, after you have a version-control system with the correct code submitted, you need a *build service* to pick up the code and run the global build and tests.

The internal workflow for this step (CI, build, test) is about the construction of a CI pipeline consisting of your code repository (Git, etc.), your build server (Visual Studio Team Services), Docker Engine, and a Docker Registry.

You can use Visual Studio Team Services as the foundation for building your applications and setting your CI pipeline, and for publishing the built "artifacts" to an "artifacts repository," which is explained in the next step.

When using Docker for the deployment, the "final artifacts" to be deployed are Docker images with your application or services embedded within them. Those images are pushed or published to a

Docker Registry (a private repository like the ones you can have in Azure Container Registry, or a public one like Docker Hub Registry, which is commonly used for official base images).

Here is the basic concept: The CI pipeline will be kicked-off by a commit to an SCC repository like Git. The commit will cause Visual Studio Team Services to run a build job within a Docker container and, upon successful completion of that job, push a Docker image to the Docker Registry, as illustrated in Figure 5-2.
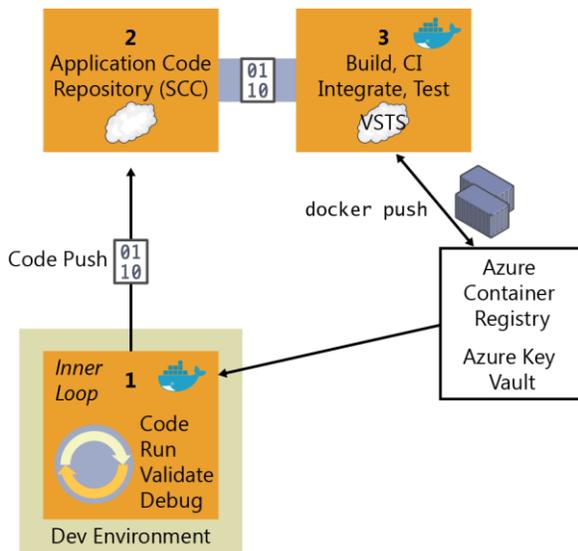
Figure 5-2: The steps involved in CI

Here are the basic CI workflow steps with Docker and Visual Studio Team Services:

1. The developer pushes a commit to an SCC repository (Git/Visual Studio Team Services, GitHub, etc.).

2. If you're using Visual Studio Team Services or Git, CI is built in, which means that it is as simple as selecting a check box in Visual Studio Team Services. If you're using an external SCC (like GitHub), a *webhook* will notify Visual Studio Team Services of the update or push to Git/GitHub.
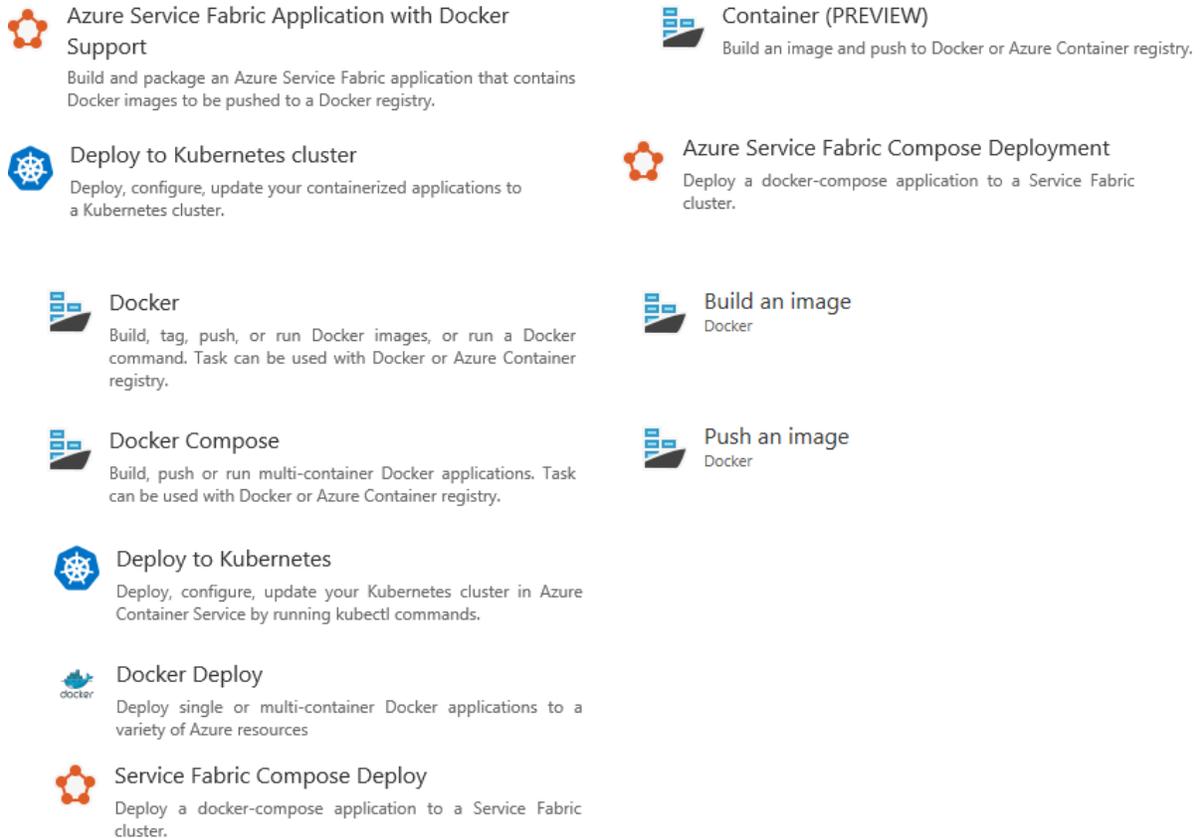
3. Visual Studio Team Services pulls the SCC repository, including the DockerFile describing the image as well as the application and test code.

4. Visual Studio Team Services builds a Docker image and labels it with a build number.

5. Visual Studio Team Services instantiates the Docker container within the provisioned Docker Host, and runs the appropriate tests.

6. If the tests are successful, the image is first relabeled to a meaningful name so that you know it is a "blessed build" (like "/1.0.0" or any other label), and then pushed up to your Docker Registry (Docker Hub, Azure Container Registry, DTR, etc.)

Implementing the CI pipeline with Visual Studio Team Services and the Docker extension for Visual Studio Team Services

Visual Studio Team System VSTS contains Build & Release Templates that you can use in your CI/CD pipeline with which you can build Docker images, push Docker images to an authenticated Docker registry, run Docker images, or run other operations offered by the Docker CLI. It also adds a Docker Compose task that you can use to build, push, and run multicontainer Docker applications, or run other operations offered by the Docker Compose CLI, as shown in Figure 5-3.

Azure Service Fabric Application with Docker Support
Build and package an Azure Service Fabric application that contains Docker images to be pushed to a Docker registry.

Container (PREVIEW)
Build an image and push to Docker or Azure Container registry.

Deploy to Kubernetes cluster
Deploy, configure, update your containerized applications to a Kubernetes cluster.

Azure Service Fabric Compose Deployment
Deploy a docker-compose application to a Service Fabric cluster.

Docker
Build, tag, push, or run Docker images, or run a Docker command. Task can be used with Docker or Azure Container registry.

Build an image
Docker

Docker Compose
Build, push or run multi-container Docker applications. Task can be used with Docker or Azure Container registry.

Push an image
Docker

Deploy to Kubernetes
Deploy, configure, update your Kubernetes cluster in Azure Container Service by running kubectl commands.

Docker Deploy
Deploy single or multi-container Docker applications to a variety of Azure resources

Service Fabric Compose Deploy
Deploy a docker-compose application to a Service Fabric cluster.

Figure 5-3: The Docker CI pipeline in Visual Studio Team Services including Build & Release Templates and associated tasks.

You can use these templates and tasks to construct your CI/CD artifacts to Build / Test and Deploy in Azure Service Fabric, Azure Container Service, etc.

With these Visual Studio Team Services tasks, a build Linux-Docker Host/VM provisioned in Azure and your preferred Docker registry (Azure Container Registry, Docker Hub, private Docker DTR, or any other Docker registry) you can assemble your Docker CI pipeline in a very consistent way.

Requirements:

- Visual Studio Team Services, or for on-premises installations, Team Foundation Server 2015 Update 3 or later.

- A Visual Studio Team Services agent that has the Docker binaries.

  An easy way to create one of these is to use Docker to run a container based on the Visual Studio Team Services agent Docker image.

> **More info** To read more about assembling a Visual Studio Team Services Docker CI pipeline and to view walkthroughs, visit the following sites:
>
> Running a Visual Studio Team Services agent as a Docker container: https://hub.docker.com/r/microsoft/vsts-agent/
>
> Building .NET Core Linux Docker images with Visual Studio Team Services: https://blogs.msdn.microsoft.com/stevelasker/2016/06/13/building-net-core-linux-docker-images-with-visual-studio-team-services/

Integrate, test, and validate multicontainer Docker applications

Typically, most Docker applications are composed of multiple containers rather than a single container. A good example is a microservices-oriented application for which you would have one container per microservice. But, even without strictly following the microservices approach patterns, it is very probable that your Docker application would be composed of multiple containers or services.

Therefore, after building the application containers in the CI pipeline, you also need to deploy, integrate, and test the application as a whole with all of its containers within an integration Docker host or even into a test cluster to which your containers are distributed.

If you're using a single host, you can use Docker commands such as `docker-compose` to build and deploy related containers to test and validate the Docker environment in a single VM. But, if you are working with an orchestrator cluster like DC/OS, Kubernetes, or Docker Swarm, you need to deploy your containers through a different mechanism or orchestrator, depending on your selected cluster/scheduler.

Following are several types of tests that you can run against Docker containers:

- Unit tests for Docker containers

- Testing groups of interrelated applications or microservices

- Test in production and "canary" releases

The important point is that when running integration and functional tests, you must run those tests from outside of the containers. Tests must not be defined and run within the containers that you are deploying, because the containers are based on static images that should be exactly like those that you will be deploying into production.

A very feasible option when testing more advanced scenarios like testing several clusters (test cluster, staging cluster, and production cluster) is to publish the images to a registry to test in various clusters.

Push the custom application Docker image into your global Docker Registry

After the Docker images have been tested and validated, you'll want to tag and publish them to your Docker registry. The Docker registry is a critical piece in the Docker application life cycle because it is the central place where you store your custom test (aka "blessed images") to be deployed into QA and production environments.

Similar to how the application code stored in your SCC repository (Git, etc.) is your "source of truth," the Docker registry is your "source of truth" for your binary application or bits to be deployed to the QA or production environments.

Typically, you might want to have your private repositories for your custom images either in a private repository in Azure Container Registry or in an on-premises registry like Docker Trusted Registry, or in a public-cloud registry with restricted access (like Docker Hub), although in this last case if your code is not open source, you must trust the vendor's security. Either way, the method by which you do this is pretty similar and ultimately based on the `docker push` command, as depicted in Figure 5-4.
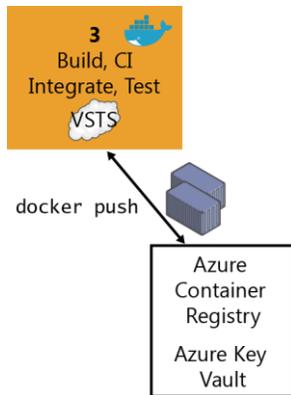
Figure 5-4: Publishing custom images to Docker Registry

There are multiple offerings of Docker registries from cloud vendors like Azure Container Registry, Amazon Web Services Container Registry, Google Container Registry, Quay Registry, and so on.

Using the Docker tasks, you can push a set of service images defined by a docker-compose.yml file, with multiple tags, to an authenticated Docker registry (like Azure Container Registry), as shown in Figure 5-5.
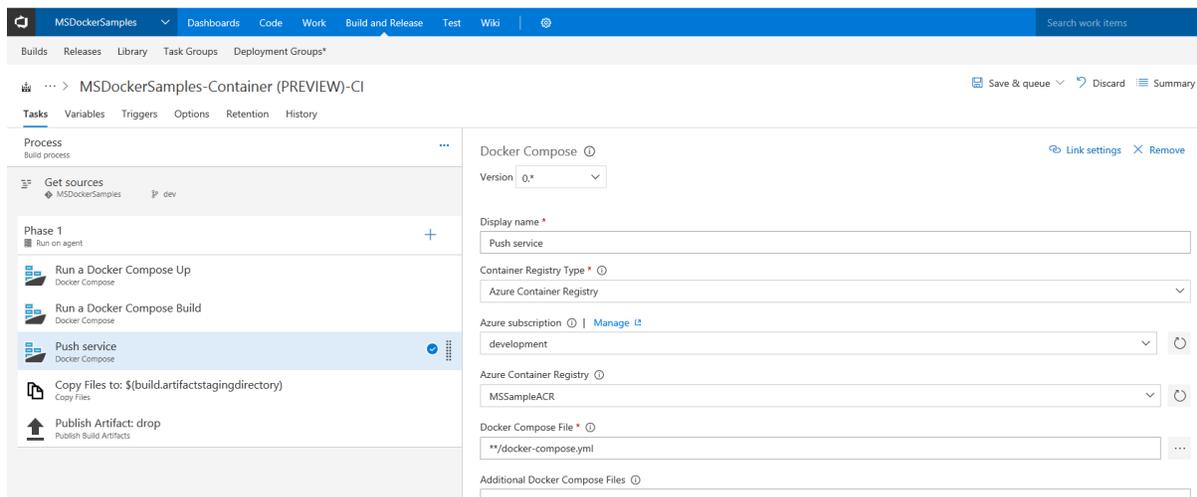

Figure 5-5: Using Visual Studio Team Services to publishing custom images to a Docker Registry

> **More info** To read more about the Docker extension for Visual Studio Team Services, go to https://aka.ms/vstsdockerextension. To learn more about Azure Container Registry, go to https://aka.ms/azurecontainerregistry.

Step 4: CD, Deploy

The immutability of Docker images ensures a repeatable deployment with what's developed, tested through CI, and run in production. After you have the application Docker images published in your Docker registry (either private or public), you can deploy them to the several environments that you might have (production, QA, staging, etc.) from your CD pipeline by using Visual Studio Team Services pipeline tasks or Visual Studio Team Services Release Management.

However, at this point it depends on what kind of Docker application you are deploying. Deploying a simple application (from a composition and deployment point of view) like a monolithic application comprising a few containers or services and deployed to a few servers or VMs is very different from

deploying a more complex application like a microservices-oriented application with hyperscale capabilities. These two scenarios are explained in the following sections.

Deploying composed Docker applications to multiple Docker environments

Let's look first at the less-complex scenario: deploying to simple Docker hosts (VMs or servers) in a single environment or multiple environments (QA, staging, and production). In this scenario, internally your CD pipeline can use `docker-compose` (from your Visual Studio Team Services deployment tasks) to deploy the Docker applications with its related set of containers or services, as illustrated in Figure 5-6.
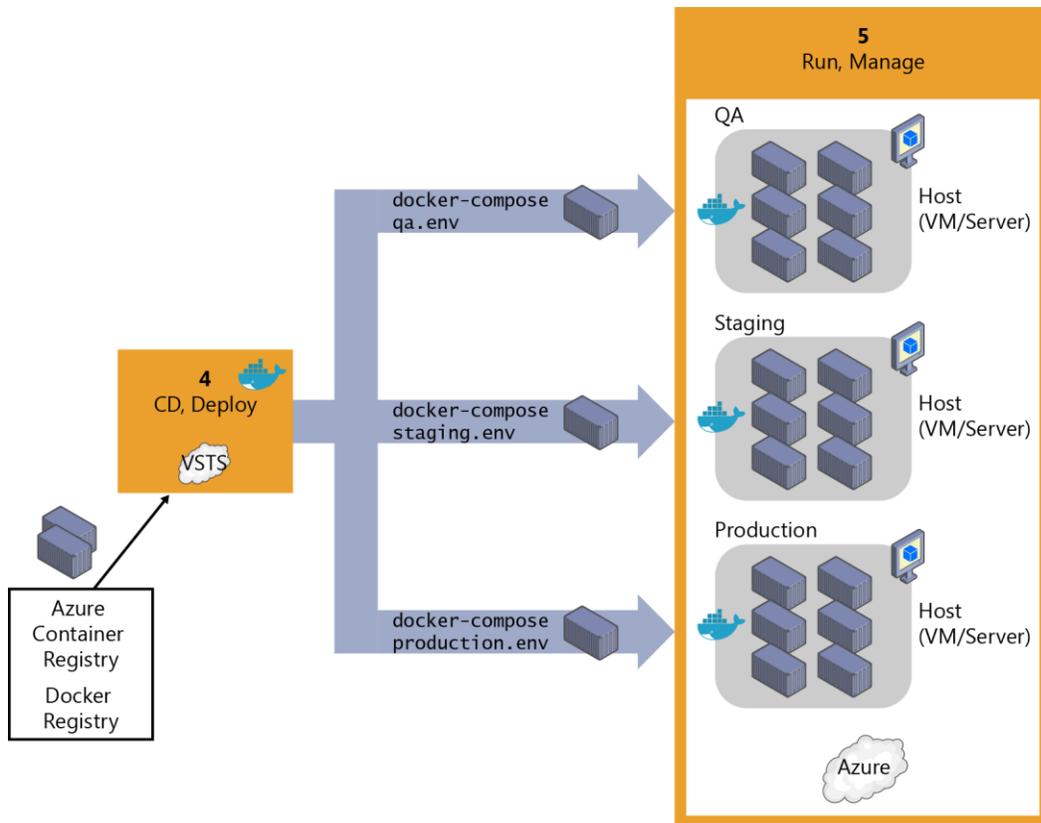

Figure 5-6: Deploying application containers to simple Docker host environments registry

Figure 5-7 highlights how you can connect your build CI to QA/test environments via Visual Studio Team Services by clicking Docker Compose in the Add Task dialog box. However, when deploying to staging or production environments, you would usually use Release Management features handling multiple environments (like QA, staging, and production). If you're deploying to single Docker hosts, it is using the Visual Studio Team Services "Docker Compose" task (which is invoking the `docker-compose up` command under the hood). If you're deploying to Azure Container Service, it uses the Docker Deployment task, as explained in the section that follows.
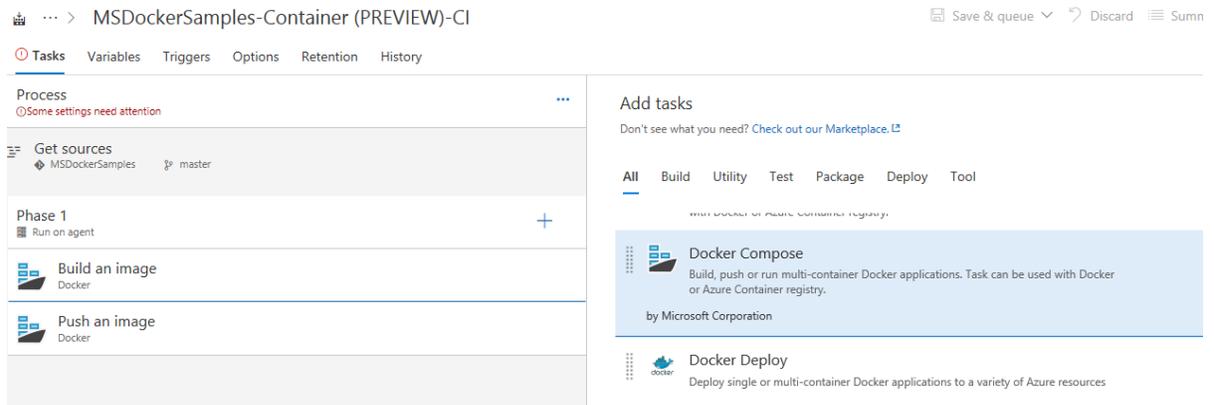
Figure 5-7: Adding a Docker Compose task in a Visual Studio Team Services pipeline

When you create a release in Visual Studio Team Services, it takes a set of input artifacts. These are intended to be immutable throughout the lifetime of the release across multiple environments. When you introduce containers, the input artifacts identify images in a registry to deploy. Depending on how these are identified, they are not guaranteed to remain the same throughout the duration of the release, the most obvious case being when you reference "myimage:latest" from a docker-compose file.

The Docker extension for Visual Studio Team Services gives you the ability to generate build artifacts that contain specific registry image digests that are guaranteed to uniquely identify the same image binary. These are what you really want to use as input to a release.

Managing releases to Docker environments by using Visual Studio Team Services Release Management

Through the Visual Studio Team Services extensions, you can build a new image, publish it to a Docker registry, run it on Linux or Windows hosts, and use commands such as `docker-compose` to deploy multiple containers as an entire application, all through the Visual Studio Team Services Release Management capabilities intended for multiple environments, as shown in Figure 5-8.
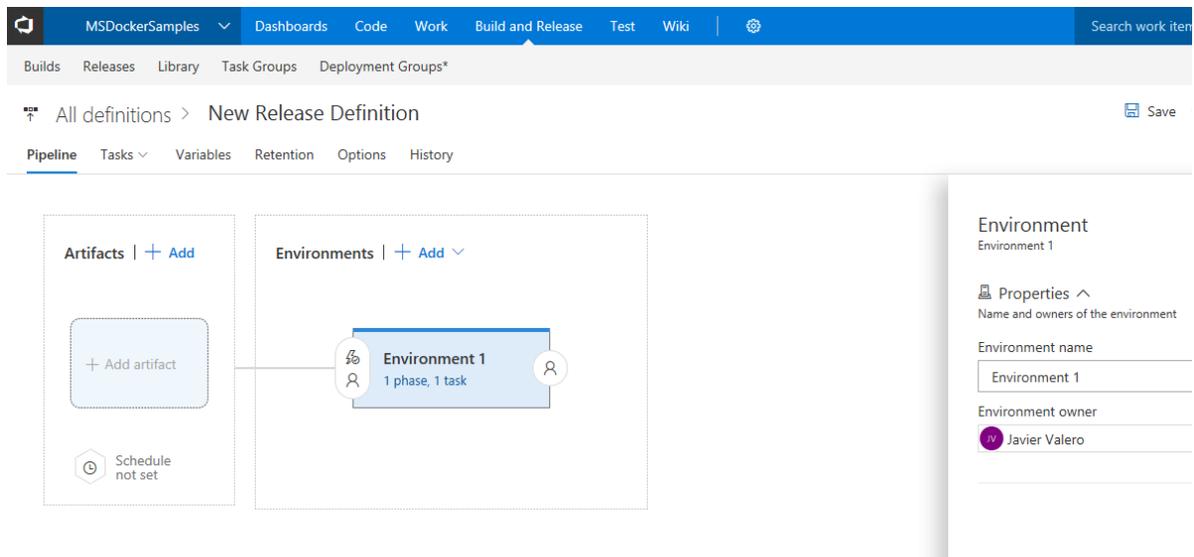
Figure 5-8: Configuring Visual Studio Team Services Docker Compose tasks from Visual Studio Team Services Release Management

However, keep in mind that the scenario shown in Figure 5-6 and implemented in Figure 5-8 is pretty basic (it is deploying to simple Docker hosts and VMs, and there will be a single container or instance per image) and probably should be used only for development or test scenarios. In most enterprise production scenarios, you would want to have High Availability (HA) and easy-to-manage scalability by load balancing across multiple nodes, servers, and VMs, plus "intelligent failovers" so that if a server or node fails, its services and containers will be moved to another host server or VM. In that case, you need more advanced technologies like container clusters, orchestrators, and schedulers. Thus, the way to deploy to those clusters is precisely through the advanced scenarios explained in the next section.

Deploying Docker applications to Docker clusters (Kubernetes, DC/OS and Docker Swarm)

The nature of distributed applications requires compute resources that are also distributed. To have production-scale capabilities, you need to have clustering capabilities that provide high scalability and HA based on pooled resources.

You could deploy containers manually to those clusters from a CLI tool such as Docker Swarm (like using docker service create) or a web UI such as Mesosphere Marathon for DC/OS clusters, but you should reserve that only for punctual deployment testing or for management purposes like scaling-out or monitoring purposes.

From a CD point of view, and Visual Studio Team Services specifically, you can run specially made deployment tasks from your Visual Studio Team Services Release Management environments which will deploy your containerized applications to distributed clusters in Container Service, as illustrated in Figure 5-9.
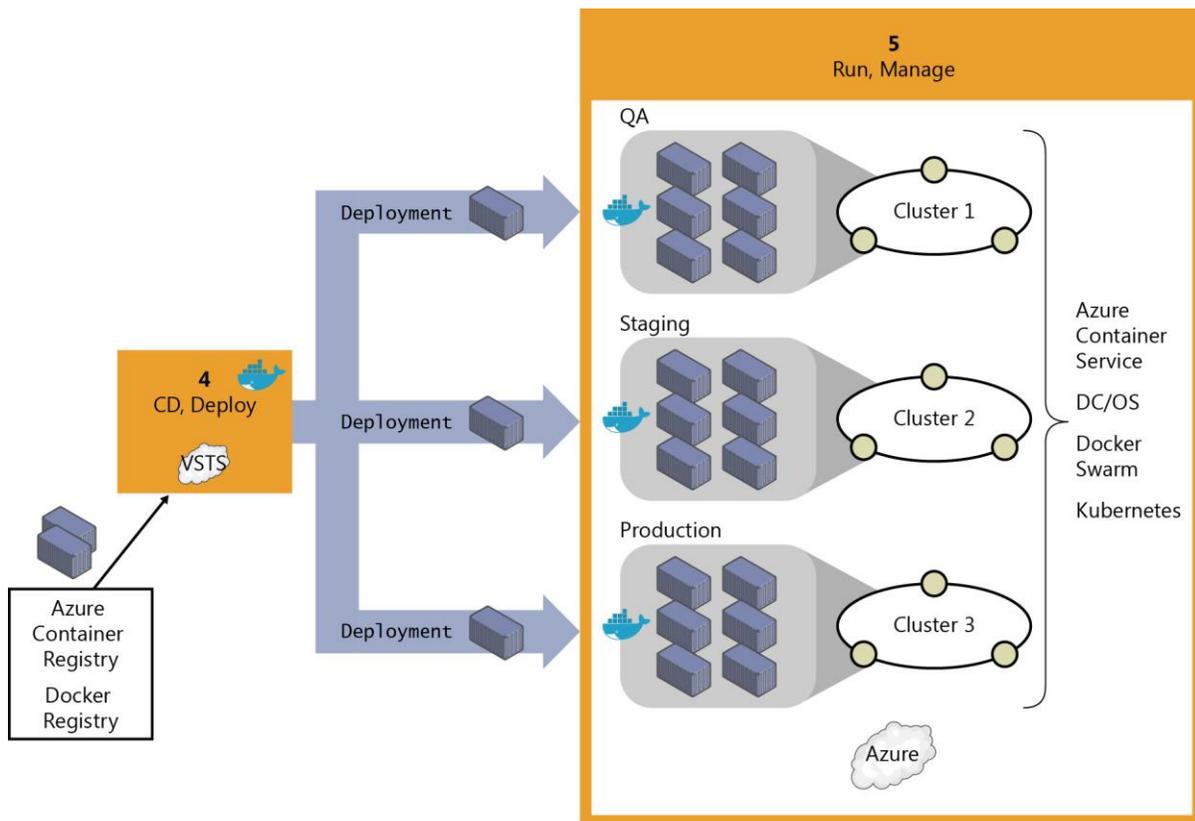

Figure 5-9: Deploying distributed applications to Container Service

Initially, when deploying to certain clusters or orchestrators, you would traditionally use specific deployment scripts and mechanisms per each orchestrator (i.e., Mesosphere DC/OS or Kubernetes have different deployment mechanisms than Docker and Docker Swarm) instead of the simpler and easy-to-use docker-compose tool based on the docker-compose.yml definition file. However, thanks to the Microsoft Visual Studio Team Services Docker Deploy task, shown in Figure 5-10, you now also can deploy to DC/OS by just using your familiar docker-compose.yml file because Microsoft performs that "translation" for you (from your docker-compose.yml file to other formats needed by DC/OS).
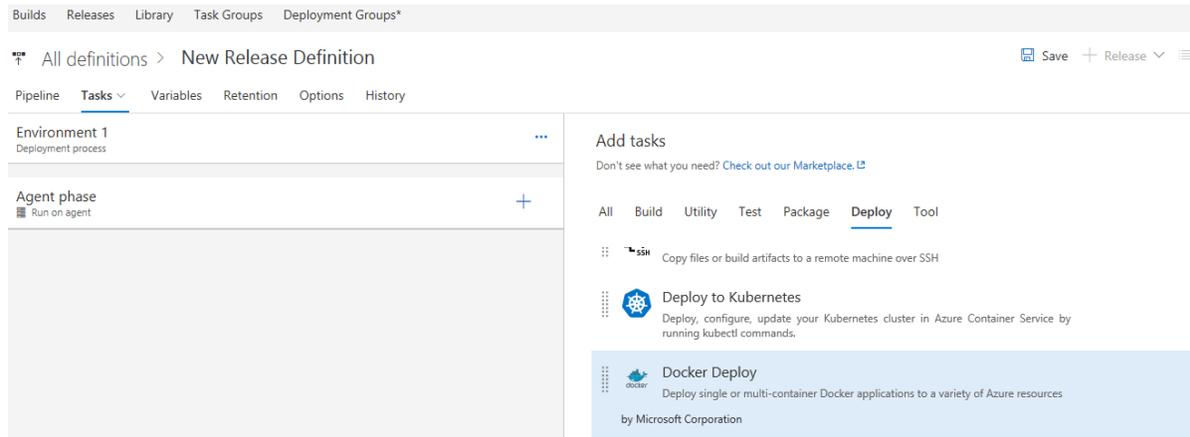

Figure 5-10: Adding the Docker Deploy task to your Environment RM

Figure 5-11 demonstrates how you can edit the Docker Deploy task and specify the Target Type (Azure Container Service DC/OS, in this case), your Docker Compose File, and the Docker Registry connection (like Azure Container Registry or Docker Hub). This is where the task will retrieve your ready-to-use custom Docker images to be deployed as containers in the DC/OS cluster.
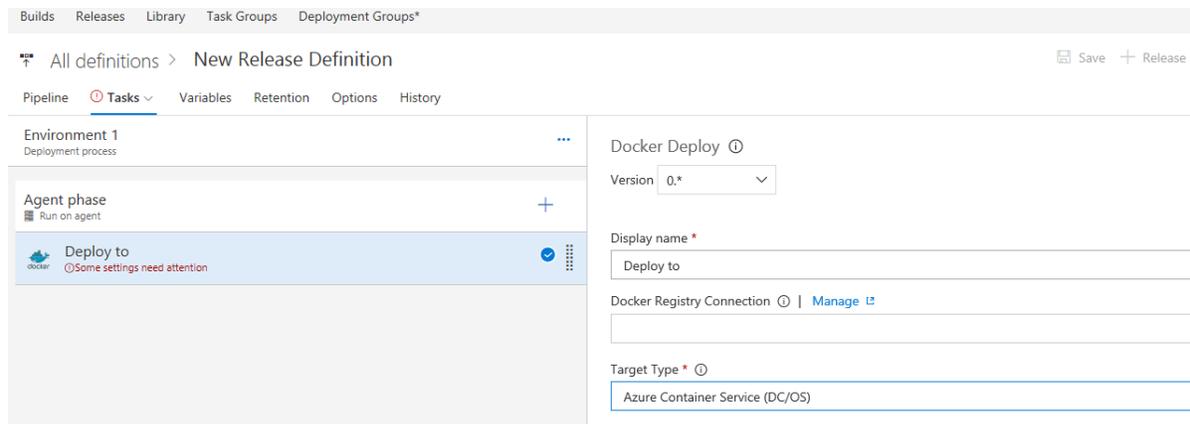
Figure 5-11: Docker Deploy task definition deploying to Azure Container Service DC/OS

> **More info** To read more about the CD pipeline with Visual Studio Team Services and Docker, visit the following sites:
>
> Visual Studio Team Services extension for Docker and Azure Container Service: https://aka.ms/vstsdockerextension
>
> Azure Container Service: https://aka.ms/azurecontainerservice
>
> Mesosphere DC/OS: https://mesosphere.com/product/

Step 5: Run and manage

Because running and managing applications at enterprise-production level is a major subject in and of itself, and due to the type of operations and people working at that level (IT operations) as well as the large scope of this area, we have devoted the entire next chapter to explaining it.

Step 6: Monitor and diagnose

This topic also is covered in the next chapter as part of the tasks that IT operations performs in production systems; however, is important to highlight that the insights obtained in this step must feed back to the development team so that the application is constantly improved. From that point of view, it is also part of DevOps, although the tasks and operations are usually performed by IT.

Only when monitoring and diagnostics are 100 percent within the realm of DevOps are the monitoring processes and analytics performed by the development team against testing or beta environments. This is done either by performing load testing or simply by monitoring beta or QA environments, where beta testers are trying the new versions.

Creating CI/CD pipelines in VSTS for a .NET Core 2.0 application on Containers and deploying to a Kubernetes cluster

In Figure 5-12 you can see the end-to-end DevOps scenario covering the code management, code compilation, Docker images build, Docker images push to a Docker registry and finally the deployment to a Kubernetes cluster in Azure.
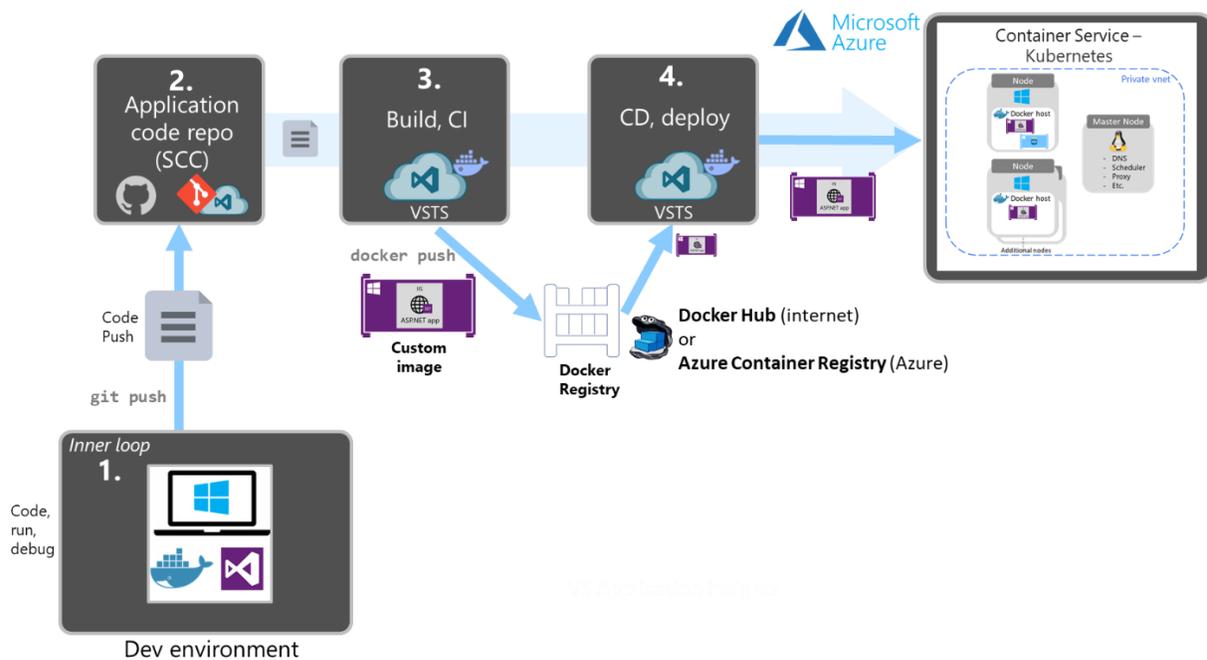

Figure 5-12: CI/CD scenario creating Docker images and deploying to a Kubernetes cluster in Azure

It is important to highlight that the two different pipelines build/CI and release/CD are connected through the Docker Registry (such as Docker Hub or Azure Container Registry). The Docker registry is one of the main differences compared to a traditional CI/CD process with no Docker.

As shown in Figure 5-13, the first phase is the build/CI pipeline. In VSTS you can create build/CD pipelines that will compile the code, create the Docker images and push it to a Docker Registry like Docker Hub or Azure Container Registry.
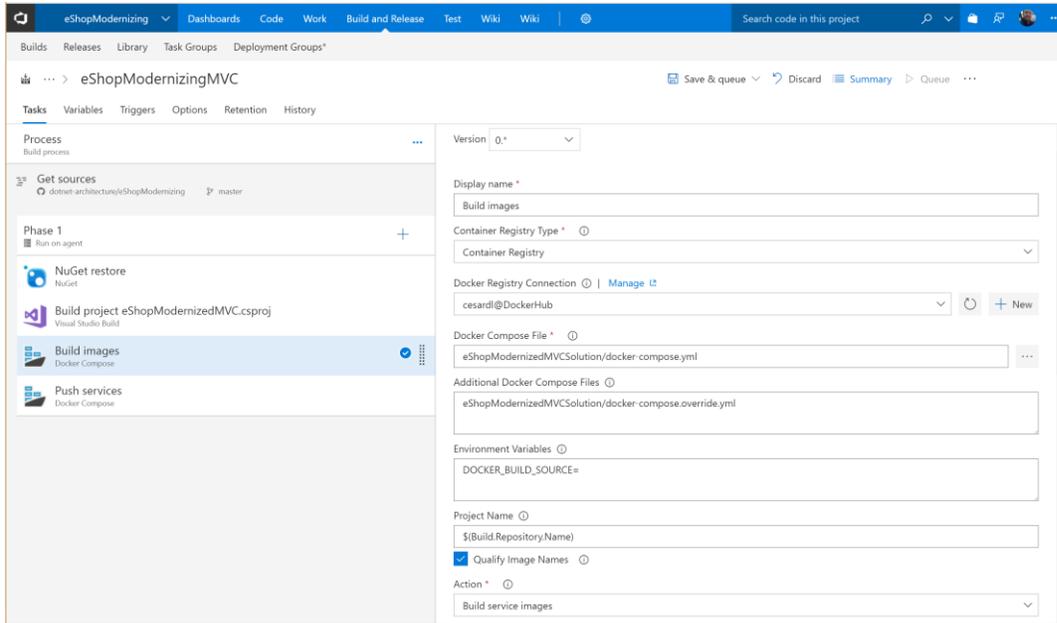
Figure 5-13: Build/CI pipeline in VSTS building Docker images and pushing images to a Docker registry

The second phase is to create a deployment/release pipeline. In VSTS you can easily create a deployment pipeline targeting a Kubernetes cluster by using the Kubernetes tasks for VSTS, like shown in Figure 5-14.
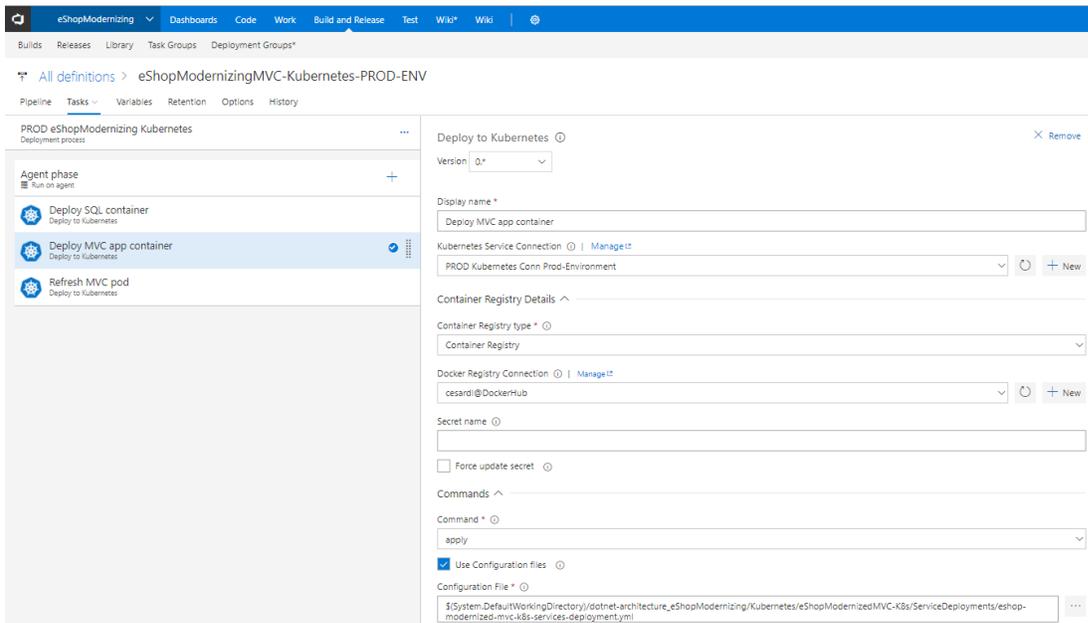
Figure 5-14: Release/CD pipeline in VSTS deploying to a Kubernetes cluster

Walkthrough: Deploying eShopModernized to Kubernetes:
In order to explore a detailed walkthrough of VSTS CI/CD pipelines deploying to Kubernetes, check the following on-line post:
https://github.com/dotnet-architecture/eShopModernizing/wiki/03.-How-to-deploy-your-Windows-Containers-based-app-into-Azure-VMs-(Including-CI-CD)

Running, managing, and monitoring Docker production environments

Vision: Enterprise applications need to run with high availability and high scalability; IT operations need to be able to manage and monitor the environments and the applications themselves.

This last pillar in the containerized Docker applications life cycle is centered on how you can run, manage, and monitor your applications in scalable, high availability (HA) production environments.

How you run your containerized applications in production (infrastructure architecture and platform technologies) is also very much related and completely founded on the chosen architecture and development platforms that we looked at in the Chapter 1 of this ebook. This chapter examines specific products and technologies from Microsoft and other vendors that you can use to effectively run highly scalable, HA distributed applications plus how you can manage and monitor them from the IT perspective.

Running composed and microservices-based applications in production environments

Applications composed by multiple microservices do need to be deployed into orchestrator clusters in order to simplify the complexity of deployment and make it viable from an IT point of view. Without an orchestrator cluster, it would be very difficult to deploy and scale-out a complex microservices application.

Introduction to orchestrators, schedulers, and container clusters

Earlier in this ebook, we introduced *clusters* and *schedulers* as part of the discussion on software architecture and development. Examples of Docker clusters are Docker Swarm and Mesosphere Datacenter Operating System (DC/OS). Both of these can run as a part of the infrastructure provided by Microsoft Azure Container Service.

When applications are scaled-out across multiple host systems, the ability to manage each host system and abstract away the complexity of the underlying platform becomes attractive. That is precisely what orchestrators and schedulers provide. Let's take a brief look at them here:

- **Schedulers**   "Scheduling" refers to the ability for an administrator to load a service file onto a host system that establishes how to run a specific container. Launching containers in a Docker cluster tends to be known as scheduling. Although scheduling refers to the specific act of loading the service definition, in a more general sense, schedulers are responsible for hooking into a host's init system to manage services in whatever capacity needed.

  A cluster scheduler has multiple goals: using the cluster's resources efficiently, working with user-supplied placement constraints, scheduling applications rapidly to not leave them in a pending state, having a degree of "fairness," being robust to errors, and always be available.

- **Orchestration**   Platforms extend life-cycle management capabilities to complex, multicontainer workloads deployed on a cluster of hosts. By abstracting the host infrastructure, orchestration tools give users a way to treat the entire cluster as a single deployment target.

  The process of orchestration involves tooling and a platform that can automate all aspects of application management from initial placement or deployment per container; moving containers to different hosts depending on its host's health or performance; versioning and rolling updates and health monitoring functions that support scaling and failover; and many more.

  Orchestration is a broad term that refers to container scheduling, cluster management, and possibly the provisioning of additional hosts.

The capabilities provided by orchestrators and schedulers are very complex to develop and create from scratch, and therefore you usually would want to make use of orchestration solutions offered by vendors.

Managing production Docker environments

Cluster management and orchestration is the process of controlling a group of hosts. This can involve adding and removing hosts from a cluster, getting information about the current state of hosts and containers, and starting and stopping processes. Cluster management and orchestration are closely tied to scheduling because the scheduler must have access to each host in the cluster in order to schedule services. For this reason, the same tool is often used for both purposes.

Container Service and management tools

Container Service provides rapid deployment of popular open-source container clustering and orchestration solutions. It uses Docker images to ensure that your application containers are fully portable. By using Container Service, you can deploy DC/OS (powered by Mesosphere and Apache Mesos) and Docker Swarm clusters with Azure Resource Manager templates or the Azure portal to ensure that you can scale these applications to thousands—even tens of thousands—of containers.

You deploy these clusters by using Azure Virtual Machine Scale Sets, and the clusters take advantage of Azure networking and storage offerings. To access Container Service, you need an Azure subscription. With Container Service, you can take advantage of the enterprise-grade features of Azure while still maintaining application portability, including at the orchestration layers.

Table 6-1 lists common management tools related to their orchestrators, schedulers, and clustering platform.

Table 6-1: Docker management tools

Management tools	Description	Related orchestrators
Azure Container Service (UI management in Azure portal)	Container Service provides an easy to get started way to deploy a container-cluster in Azure based on popular orchestrators like Mesosphere DC/OS, Kubernetes and Docker Swarm. Container Service optimizes the configuration of those platforms. You just need to select the size, the number of hosts, and choice of orchestrator tools, and Container Service handles everything else.	Mesosphere DC/OS Kubernetes Docker Swarm
Docker Universal Control Plane (on-premises or cloud)	Docker Universal Control Plane is the enterprise-grade cluster management solution from Docker. It helps you manage your entire cluster from a single place. Docker Universal Control Plane is included as part of the commercial product named Docker Datacenter which provides Docker	Docker Swarm

	Swarm, Docker Universal Control Plane and Docker Trusted Registry. Docker Datacenter can be installed on-premises or provisioned from a public cloud like Azure.	
Docker Cloud (aka Tutum; cloud SaaS)	Docker Cloud is a hosted management service (SaaS) that provides orchestration capabilities and a Docker registry with build and testing facilities for Dockerized application images, tools to help you set up and manage your host infrastructure, and deployment features to help you automate deploying your images to your concrete infrastructure. You can connect your SaaS Docker Cloud account to your infrastructure in Container Service running a Docker Swarm cluster.	Docker Swarm
Mesosphere Marathon (on-premises or cloud)	Marathon is a production-grade container orchestration and scheduler platform for Mesosphere's DC/OS and Apache Mesos. It works with Mesos (DC/OS is based on Apache Mesos) to control long-running services and provides a web UI for process and container management. It provides a web UI management tool	Mesosphere DC/OS (Based on Apache Mesos; supported by Container Service)
Kubernetes	Kubernetes spans orchestrating, scheduling, and cluster infrastructure. It is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.	Kubernetes Supported by Azure Container Service (AKS)

Azure Service Fabric

Another choice for cluster-deployment and management is Azure Service Fabric. Service Fabric is a Microsoft microservices platform that includes container orchestration as well as developer programming models to build highly-scalable microservices applications. Service Fabric supports Docker in current Linux preview versions, as in the Service Fabric preview on Linux, and for Windows Containers in the next release.

Following are Service Fabric management tools:

- [Azure portal for Service Fabric](#) cluster-related operations (create/update/delete) a cluster or configure its infrastructure (VMs, load balancer, networking, etc.)

- [Azure Service Fabric Explorer](#) is a specialized web UI tool that provides insights and certain operations on the Service Fabric cluster from the nodes/VMs point of view and from the application and services point of view.

Monitoring containerized application services

It is critical for applications split into multiple containers and microservices to have a way to monitor and analyze the behavior of the application.

Microsoft Application Insights

[Application Insights](#) is an extensible analytics service that monitors your live application. It helps you to detect and diagnose performance issues and to understand what users actually do with your app. It's designed for developers, with the intent of helping you to continuously improve the performance and usability of your services or applications. Application Insights works with both web/services and standalone apps on a wide variety of platforms like .NET, Java, Node.js and many other platforms, hosted on-premises or in the cloud.

Analyzing Docker apps in QA environments using Application Insights

As it pertains to Docker, you can chart life-cycle events and performance counters from Docker containers on Application Insights. You just need to run the [Application Insights Docker image](#) as a container in your host, and it will display performance counters for the host as well as for the other Docker images. This Application Insights Docker image (Figure 6-1) helps you to monitor your containerized applications by collecting telemetry about the performance and activity of your Docker host (i.e., your Linux VMs), Docker containers and the applications running within them.
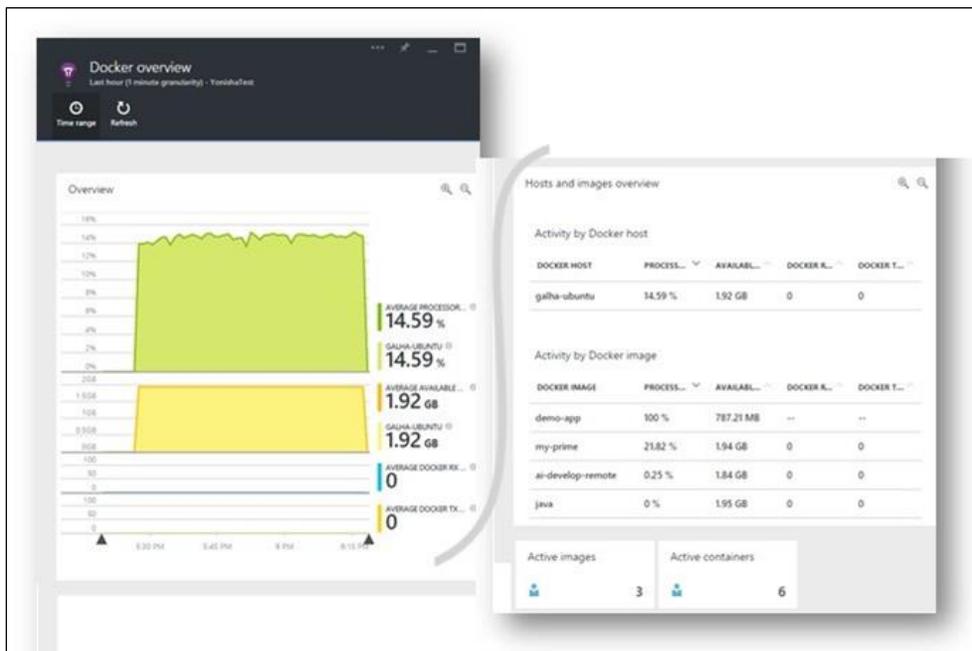

Figure 6-1: Application Insights monitoring Docker hosts and containers

When you run the Application Insights Docker image on your Docker host, you benefit from the following:

- Life-cycle telemetry about all the containers running on the host—start, stop, and so on.

- Performance counters for all the containers: CPU, memory, network usage, and more.

- If you also installed Application Insights SDK in the apps running in the containers, all the telemetry of those apps will have additional properties identifying the container and host machine. So, for example, if you have instances of an app running in more than one host, you'll easily be able to filter your app telemetry by host.

Setting up Application Insights to monitor Docker applications and Docker hosts

To create an Application Insights resource, follow the instructions in the articles presented in the list that follows. Azure Portal will create the necessary script for you.

- **Monitor Docker applications in Application Insights:** https://azure.microsoft.com/documentation/articles/app-insights-docker/

- **Application Insights Docker image at Docker Hub and Github:** https://hub.docker.com/r/microsoft/applicationinsights/ and https://github.com/Microsoft/ApplicationInsights-Docker

- **Set up Application Insights for ASP.NET:** https://azure.microsoft.com/documentation/articles/app-insights-asp-net/

- **Application Insights for web pages:** https://azure.microsoft.com/documentation/articles/app-insights-javascript/

Microsoft Operations Management Suite

Operations Management Suite is a simplified IT management solution that provides log analytics, automation, backup, and site recovery. Based on queries in Operations Management Suite, you can raise alerts and set remediation via Azure Automation. It also seamlessly integrates with your existing management solutions to provide a single pane-of-glass view. Operations Management Suite helps you to manage and protect your on-premises and cloud infrastructure.

Operations Management Suite Container Solution for Docker

In addition to providing valuable services on its own, the Operations Management Suite Container Solution can manage and monitor Docker hosts and containers by showing information about where your containers and container hosts are, which containers are running or failed, and Docker daemon and container logs sent to *stdout* and *stderr*. It also shows performance metrics such as CPU, memory, network, and storage for the container and hosts to help you troubleshoot and find noisy neighbor containers.
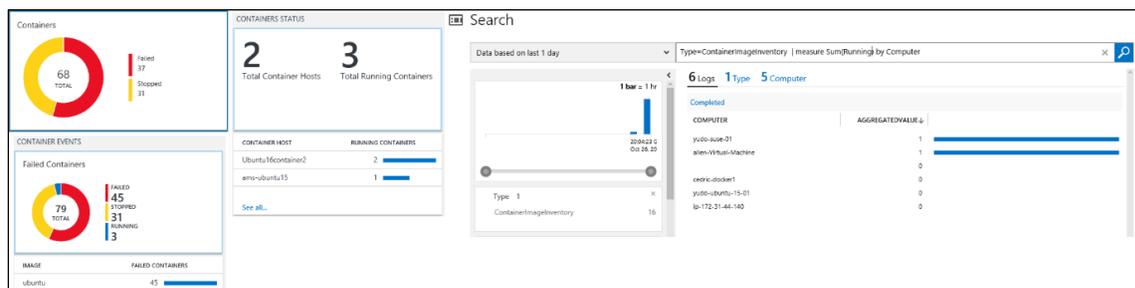

Figure 6-2: Information about Docker containers shown by Operations Management Suite

Application Insights and Operations Management Suite both focus on monitoring activities; however, Application Insights focuses more on monitoring the apps themselves thanks to its SDK running within the app. However, Operations Management Suite focuses much more on the infrastructure around the hosts, plus it offers deep analysis on logs at scale while providing a very flexible data-driven search/query system.

Because Operations Management Suite is implemented as a cloud-based service, you can have it up and running quickly with minimal investment in infrastructure services. New features are delivered automatically, saving you from ongoing maintenance and upgrade costs.

Using Operations Management Suite Container Solution, you can do the following:

- Centralize and correlate millions of logs from Docker containers at scale
- See information about all container hosts in a single location
- Know which containers are running, what image they're running, and where they're running
- Quickly diagnose "noisy neighbor" containers that can cause problems on container hosts
- See an audit trail for actions on containers
- Troubleshoot by viewing and searching centralized logs without remoting to the Docker hosts
- Find containers that might be "noisy neighbors" and consuming excess resources on a host
- View centralized CPU, memory, storage, and network usage and performance information for containers
- Generate test Docker containers with Azure Automation

You can see performance information by running queries like `Type=Perf`, as shown in Figure 6-3.
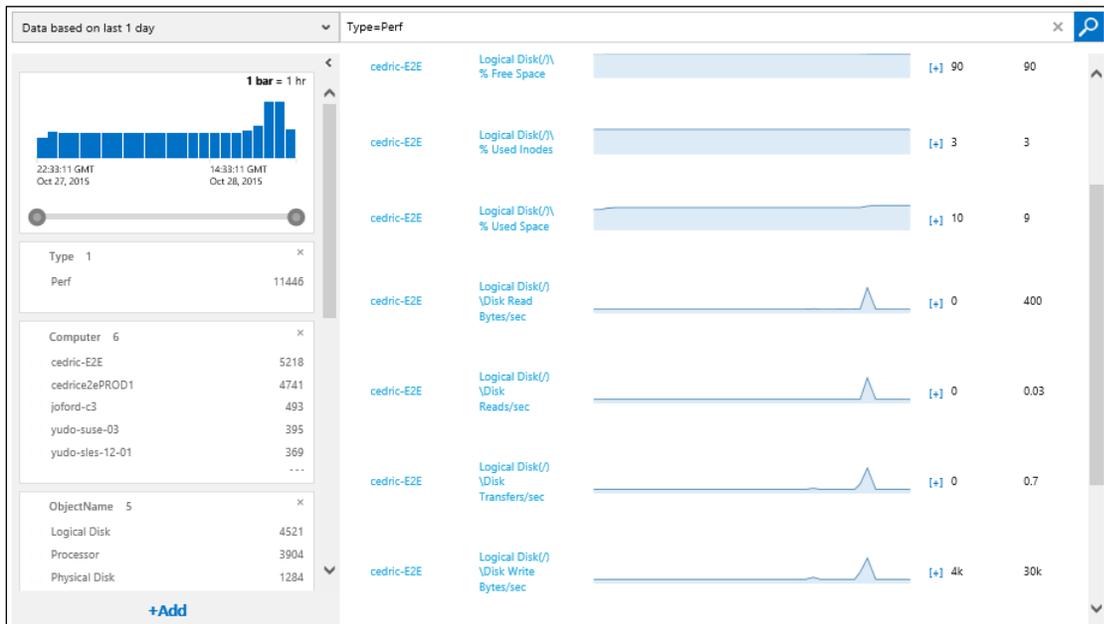
Figure 6-3: Performance metrics of Docker hosts shown by Operations Management Suite

Saving queries is also a standard feature in Operations Management Suite and can help you keep queries you've found useful and discover trends in your system.

> **More info** To find information on installing and configuring the Docker container solution in Operations Management Suite, go to https://azure.microsoft.com/documentation/articles/log-analytics-containers/.

Conclusions

Key takeaways

- Container-based solutions provide important benefits of cost savings because containers are a solution to deployment problems caused by the lack of dependencies in production environments, therefore, improving DevOps and production operations significantly.

- Docker is becoming the de facto standard in the container industry, supported by the most significant vendors in the Linux and Windows ecosystems, including Microsoft. In the future, Docker will be ubiquitous in any datacenter in the cloud or on-premises.

- A Docker container is becoming the standard unit of deployment for any server-based application or service.

- Docker orchestrators like the ones provided in Azure Container Service (AKS) and Azure Service Fabric are fundamental and indispensable for any microservices-based or multicontainer application with significant complexity and scalability needs.

- An end-to-end DevOps environment supporting Continuous Integration/Continuous Deployment connecting to the production Docker environments provides agility and ultimately improves the time to market of your applications.

Visual Studio Team Services greatly simplifies your DevOps environment designated to Docker environments from your Continuous Deployment pipelines, including simple Docker environments or more advanced microservice and container orchestrators based on Azure.