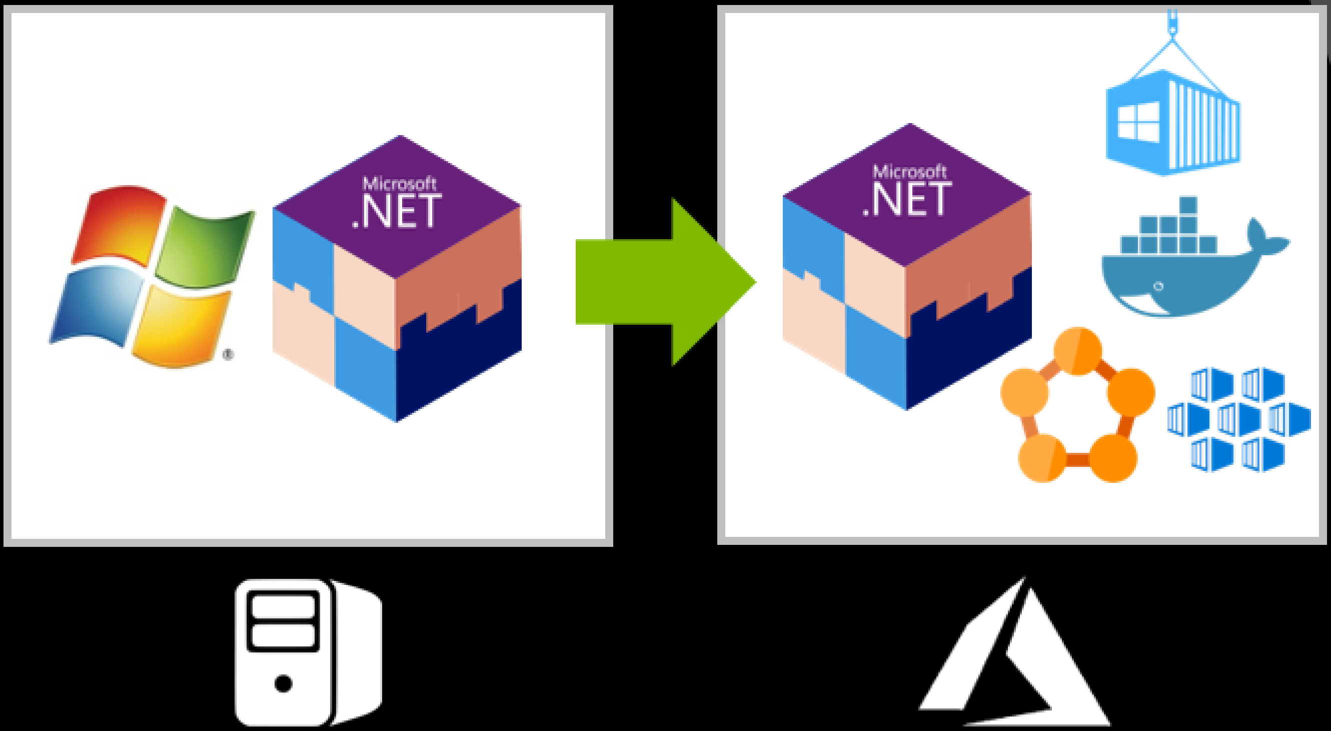
Microsoft

Modernize existing .NET applications with Azure cloud and Windows Containers



Cesar de la Torre
Microsoft Corp.

Modernize existing .NET applications with Azure cloud and Windows Containers (First Edition)
PUBLISHED BY
Microsoft Press and Microsoft DevDiv
Divisions of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Author:

Cesar de la Torre, Sr. PM, .NET Product Team, Microsoft Corp.

Participants and reviewers:

Scott Hunter, Partner Director PM, .NET team, Microsoft

Paul Yuknewicz, Principal PM Manager, Visual Studio Tools team, Microsoft

Lisa Guthrie, Sr. PM, Visual Studio Tools team, Microsoft

Ankit Asthana, Principal PM Manager, .NET team, Microsoft

Unai Zorrilla, Developer Lead, Plain Concepts

Javier Valero, Chief Operating Officer at Grupo Solutio

Contents

Introduction

When you decide to modernize your web applications and move them to the cloud, you don't necessarily have to fully re-architect your apps. Re-architecting an application by using an advanced approach like microservices isn't always an option, because of cost and time restraints. Depending on the type of application, re-architecting an app also might not be necessary. To optimize the cost-effectiveness of your organization's cloud migration strategy, it's important to consider the needs of your business and requirements of your apps. You'll need to determine:

- Which apps require a transformation or re-architecting.
- Which apps need to be only partially modernized.
- Which apps you can "lift and shift" directly to the cloud.

About this guide

This guide focuses primarily on "lift and shift" scenarios, and initial modernization of existing Microsoft .NET Framework web or service-oriented applications. Lift and shift is the action of moving a workload to a newer or more modern environment without altering the application's code and basic architecture.

This guide describes how to move your existing .NET Framework server-applications directly to the cloud by modernizing specific areas, without re-architecting or recoding entire applications.

This guide also highlights the benefits of moving your apps to the cloud and partially modernizing apps by using a specific set of new technologies and approaches, like Windows Containers and orchestrators in Azure.

Path to the cloud for existing .NET applications

Organizations typically choose to move to the cloud for the agility and speed they can get for their applications. You can set up thousands of servers (VMs) in the cloud in minutes, compared to the weeks it typically takes to set up on-premises servers.

There isn't a single, one-size-fits-all strategy for migrating applications to the cloud. The right migration strategy for you will depend on your organization's needs and priorities, and the kind of applications you are migrating. Not all applications warrant the investment of moving to a platform as a service (PaaS) model or developing a cloud-native application model. In many cases, you can take a phased or incremental approach to invest in moving your assets to the cloud, based on your business needs.

For modern applications with the best long-term agility and value for the organization, you might benefit from investing in *cloud-optimized* and *cloud-native* application architectures. However, for applications that are existing or legacy assets, the key is to spend minimal time and money (no re-architecting or code changes) while moving them to the cloud, to realize significant benefits.

Figure 1-1 shows the possible paths you can take when you move existing .NET applications to the cloud in incremental phases.

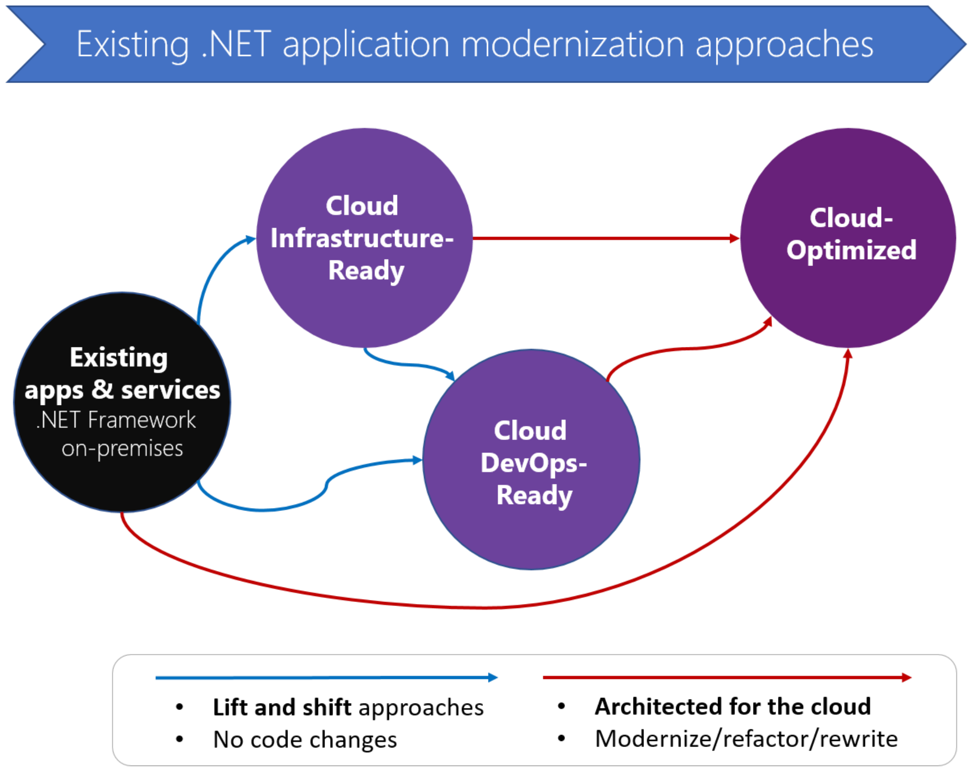

Existing .NET application modernization approaches
Cloud Infrastructure-Ready
Cloud-Optimized
Existing apps & services
.NET Framework on-premises
Cloud DevOps-Ready
Lift and shift approaches
No code changes
Architected for the cloud
Modernize/refactor/rewrite

Figure 1-1. *Modernization paths for existing .NET applications and services*

Each migration approach has different benefits and reasons for using it. You can choose a single approach when you migrate apps to the cloud, or choose certain components from multiple approaches. Individual applications aren't limited to a single approach or maturity state. For instance, a common hybrid approach would have certain on-premises components plus other components in the cloud.

At the first two migration levels, you can just lift and shift your applications:

Level 1: **Cloud Infrastructure-Ready**: In this migration approach, you simply rehost or move your current on-premises applications to an infrastructure as a service (IaaS) platform. Your apps have almost the same composition as before, but now you deploy them to VMs in the cloud.

Level 2: **Cloud DevOps-Ready**: At this level, after an initial lift and shift, and still without re-architecting or altering code, you can gain even more benefits from running your app in the cloud. You improve the agility of your applications to ship faster by refining your enterprise development operations (DevOps) processes. You achieve this by using technologies like Windows Containers, which is based on Docker Engine. Containers remove the friction that's caused by application dependencies when you deploy in multiple stages. Containers also use additional cloud managed services related to data, monitoring, and continuous integration/continuous deployment (CI/CD) pipelines.

The third level of maturity is the ultimate goal in the cloud, but it's optional for many apps and not the main focus of this guide:

Level 3: **Cloud-Optimized**: This migration approach typically is driven by business need, and targets modernizing your mission-critical applications. At this level, you use PaaS services to move your apps to PaaS computing platforms. You implement cloud-native applications and microservices architecture to evolve applications with long-term agility, and to scale to new limits. This type of modernization usually requires architecting specifically for the cloud. New code often must be written, especially when you move to cloud-native application and microservice-based models. This approach can help you gain benefits that are difficult to achieve in your monolithic and on-premises application environment.

Table 1-1 describes the main benefits of and reasons for choosing each migration or modernization approach.

Cloud Infrastructure-Ready	**Cloud DevOps-Ready**	**Cloud-Optimized**
Lift and shift		*Modernize/refactor/rewrite*
Application's compute target		
Applications deployed to VMs in Azure	Containerized monolithic or N-Tier apps deployed to VMs, Azure Service Fabric, or Azure Container Service (i.e. Kubernetes)	Containerized microservices or regular applications based on PaaS on Azure App Service, Azure Service Fabric, Azure Container Service (i.e. Kubernetes)
Data target		
SQL or any relational database on a VM	Azure SQL Database Managed Instance	Azure SQL Database, Azure Cosmos DB, or other NoSQL
Advantages		
• No re-architecting, no new code • Least effort for quick migration • Least-common denominator supported in Azure • Basic availability guarantees	• No re-architecting, no new code • Containers offer small incremental effort over VMs • Improved deployment and DevOps agility to release because of containers	• Architect for the cloud, refactor, new code needed • Microservices cloud-native approaches • New web apps, monolithic, N-Tier, cloud-resilient, and cloud-optimized

• After moving to the cloud, it's easier to modernize even more	• Increased density and lower deployment costs • Portability of apps and dependencies • With Azure Container Service (or Kubernetes) and Azure Service Fabric, provides high availability and orchestration • Nodes/VM patching in Service Fabric • Flexibility of host targets: Azure VMs or VM scale sets, Azure Container Service (or Kubernetes), Service Fabric, and future container-based choices	• Fully managed services • Automatic patching • Optimized for scale • Optimized for autonomous agility by subsystem • Built on deployment and DevOps • Enhanced DevOps, like slots and deployment strategies • PaaS and orchestrator targets: Azure App Service, Azure Container Service (or Kubernetes), Azure Service Fabric, and future container-based PaaS
	Challenges	
• Smaller cloud value, other than shift in operating expense or closing datacenters • Very little is managed: No OS or middleware patching; might require immutable infrastructure solutions, like Terraform, Spinnaker, or Puppet	• Containerizing is an additional step in the learning curve	• Might require significant code refactoring or rewriting (increased time and budget)

***Table 1-1.** Benefits and challenges of modernization paths for existing .NET applications and services*

Key technologies and architectures by maturity level

.NET Framework applications initially started with the .NET Framework version 1.0, which was released in late 2001. Then, companies moved towards newer versions (such as 2.0, 3.5 and .NET 4.x). Most of those applications run on Windows Server and Internet Information Server (IIS), and used a relational database, like SQL Server, Oracle, MySQL or any other RDBMS.

Most existing .NET applications might nowadays be based on .NET Framework 4.x, or even on .NET Framework 3.5, and use web frameworks like ASP.NET MVC, ASP.NET Web Forms, ASP.NET Web API, Windows Communication Foundation (WCF), ASP.NET SignalR, and ASP.NET Web Pages. These established .NET Framework technologies depend on Windows. That dependency is important to consider if you are simply migrating legacy apps, and you want to make minimal changes to your application infrastructure.

Figure 1-2 shows the primary technologies and architecture styles used at each of the three cloud maturity levels:

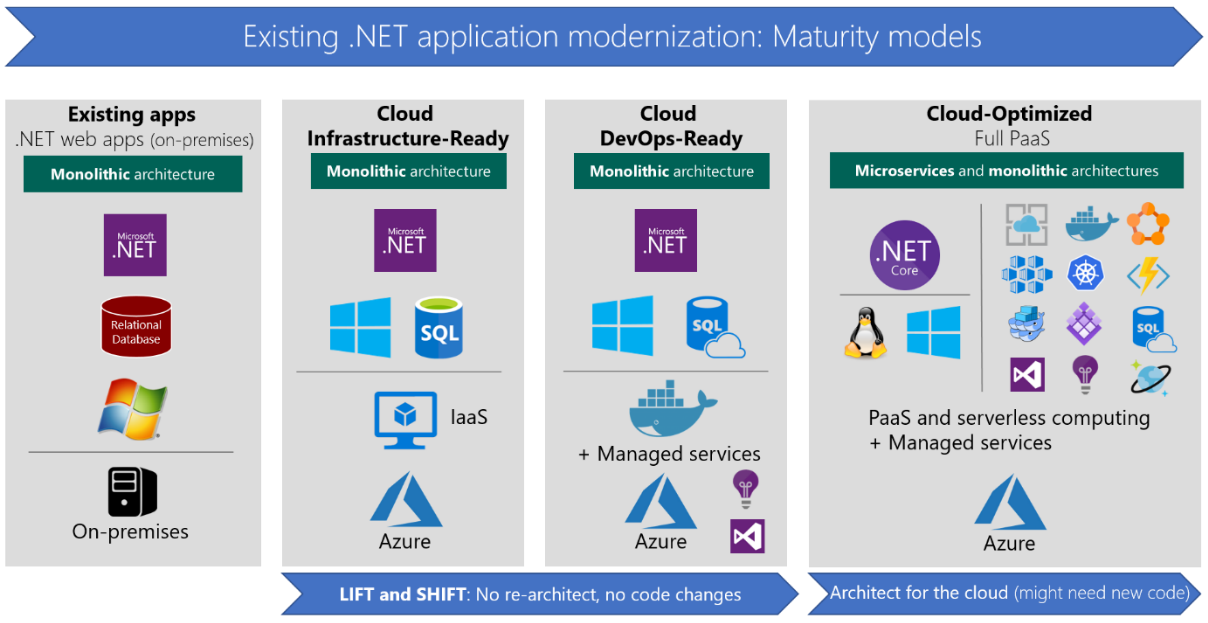

Existing .NET application modernization: Maturity models
Existing apps
.NET web apps (on-premises)
Monolithic architecture
Relational Database
On-premises
Cloud Infrastructure-Ready
Monolithic architecture
SQL
IaaS
Azure
Cloud DevOps-Ready
Monolithic architecture
SQL
+ Managed services
Azure
Cloud-Optimized
Full PaaS
Microservices and monolithic architectures
.NET Core
PaaS and serverless computing
+ Managed services
Azure
LIFT and SHIFT: No re-architect, no code changes
Architect for the cloud (might need new code)

Figure 1-2. *Primary technologies for each maturity level for modernizing existing .NET web applications*

Figure 1-2 highlights the most common scenarios, but many hybrid and mixed variations are possible when it comes to architecture. For example, the maturity models apply not only to monolithic architectures in existing web apps, but also to service orientation, N-Tier, and other architecture style variations.

Each maturity level in the modernization process is associated with the following key technologies and approaches:

- **Cloud Infrastructure-Ready** (rehost or basic lift and shift): As a first step, many organizations want only to quickly execute a cloud-migration strategy. In this case, applications are simply rehosted. Most rehosting can be automated by using Azure Migrate, a service that provides the guidance, insights, and mechanisms needed to assist you in migrating to Azure based on cloud tools like Azure Site Recovery and Azure Database Migration Service. You can also set up rehosting manually, so that you can learn infrastructure details about your assets when you move legacy apps to the cloud. For example, you can move your applications to VMs in Azure with very little modification—probably with only minor configuration changes. The networking in this case is similar to an on-premises environment, especially if you create virtual networks in Azure.
- **Cloud DevOps-Ready** (improved lift and shift): This model is about making a few important deployment optimizations to gain some significant benefits from the cloud, without changing the core architecture of the application. The fundamental step here is to add Windows Containers support to your existing .NET Framework applications. This important step (containerization) doesn't require touching the code, so the overall lift and shift effort is very light. You can use tools like Image2Docker or Visual Studio, with its tools for Docker. Visual Studio automatically chooses smart defaults for ASP.NET applications and Windows Containers images. These tools offer both a rapid inner loop, and a fast path to get the containers to Azure. Your agility is improved when you deploy to multiple environments. Then, moving to production, you can deploy your Windows Containers to orchestrators like

Azure Service Fabric or Azure Container Service (Kubernetes, DC/OS, or Swarm). During this initial modernization, you can also add assets from the cloud, such as monitoring with tools like Azure Application Insights; CI/CD pipelines for your app lifecycles with Visual Studio Team Services; and many more data resource services that are available in Azure. For instance, you can modify a monolithic web app that was originally developed by using traditional ASP.NET Web Forms or ASP.NET MVC, but now you deploy it by using Windows Containers. When you use Windows Containers, you should also migrate your data to a database in Azure SQL Database Managed Instance, all without changing the core architecture of your application.

- **Cloud-Optimized**: As noted, the ultimate goal when you modernize applications in the cloud is basing your system on PaaS platforms like Azure App Service. PaaS platforms focus on modern web applications, and extend your apps with new services based on serverless computing and platforms like Azure Functions. The second and more advanced scenario in this maturity model is about microservices architectures and cloud-native applications, which typically use orchestrators like Azure Service Fabric or Azure Container Service (Kubernetes, DC/OS, or Swarm). These orchestrators are made specifically for microservices and multi-container applications. All these approaches (like microservices and PaaS) typically require you to write new code—code that is adapted to specific PaaS platforms, or code that aligns with specific architectures, like microservices.

Figure 1-3 shows the internal technologies that you can use for each maturity level:

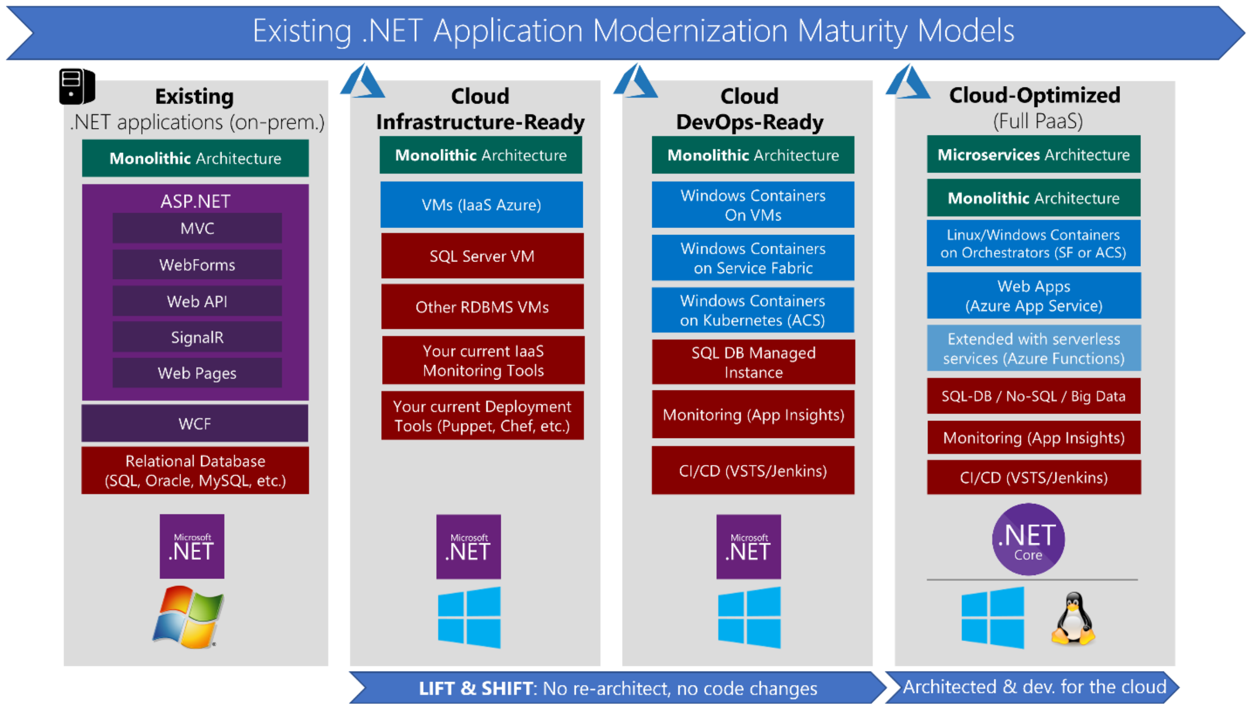

Existing .NET Application Modernization Maturity Models
Existing
.NET applications (on-prem.)
Monolithic Architecture
ASP.NET
MVC
WebForms
Web API
SignalR
Web Pages
WCF
Relational Database (SQL, Oracle, MySQL, etc.)
Microsoft .NET
Cloud Infrastructure-Ready
Monolithic Architecture
VMs (IaaS Azure)
SQL Server VM
Other RDBMS VMs
Your current IaaS Monitoring Tools
Your current Deployment Tools (Puppet, Chef, etc.)
Microsoft .NET
Cloud DevOps-Ready
Monolithic Architecture
Windows Containers On VMs
Windows Containers on Service Fabric
Windows Containers on Kubernetes (ACS)
SQL DB Managed Instance
Monitoring (App Insights)
CI/CD (VSTS/Jenkins)
Microsoft .NET
Cloud-Optimized
(Full PaaS)
Microservices Architecture
Monolithic Architecture
Linux/Windows Containers on Orchestrators (SF or ACS)
Web Apps (Azure App Service)
Extended with serverless services (Azure Functions)
SQL-DB / No-SQL / Big Data
Monitoring (App Insights)
CI/CD (VSTS/Jenkins)
.NET Core
LIFT & SHIFT: No re-architect, no code changes
Architected & dev. for the cloud

Figure 1-3. *Internal technologies for each modernization maturity level*

Lift and shift scenarios

For lift and shift migrations, keep in mind that you can use many different variations of lift and shift in your application scenarios. If you only rehost your application, you might have a scenario like the one shown in Figure 1-4, where you use VMs in the cloud only for your application and for your database server.

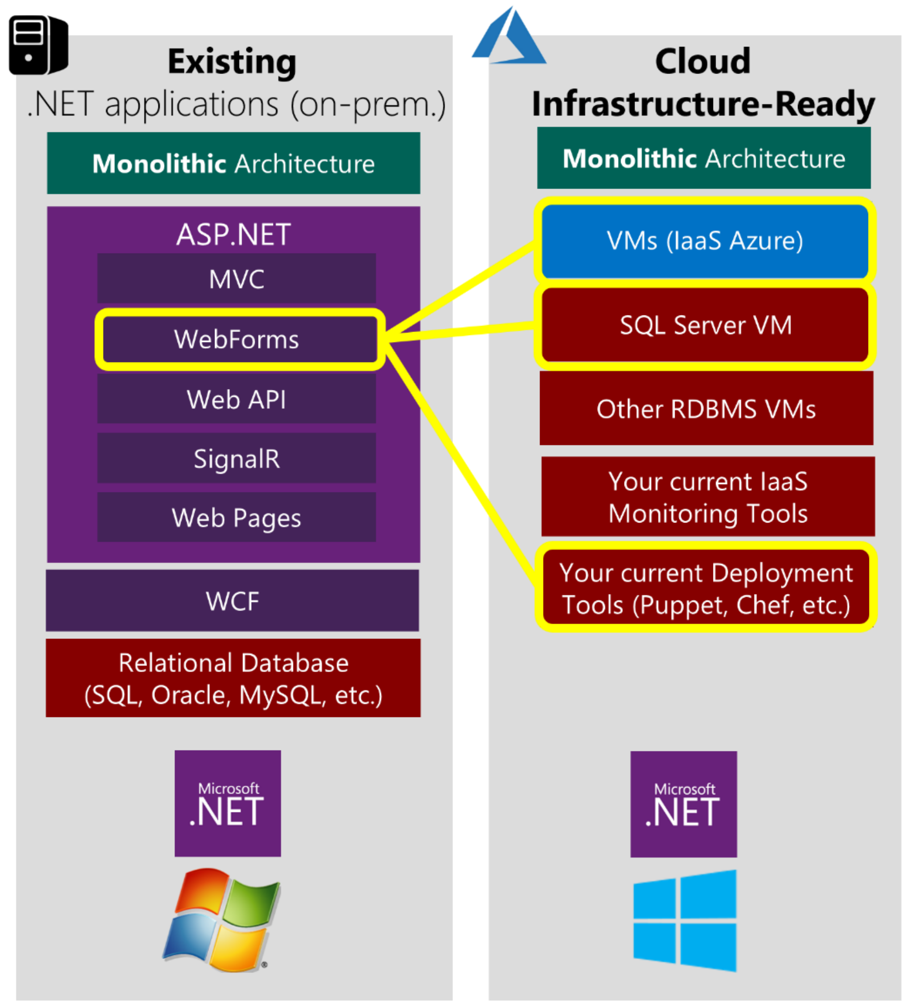


Figure 1-4*. Example of a pure IaaS scenario in the cloud*

Moving forward, you might have a pure Cloud DevOps-ready application that uses elements only from that maturity level. Or, you might have an intermediate-state application with some elements from Cloud Infrastructure-Ready and other elements from Cloud DevOps-Ready (a "pick and choose" or mixed model), like in Figure 1-5.

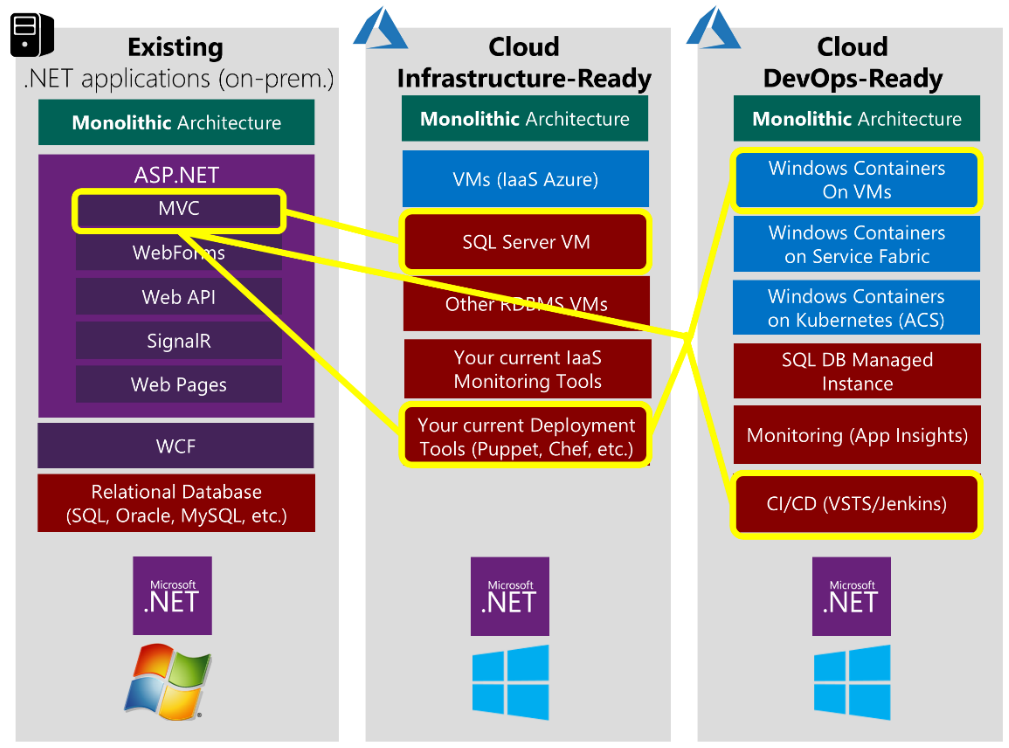

Existing
.NET applications (on-prem.)
Monolithic Architecture
ASP.NET
MVC
WebForms
Web API
SignalR
Web Pages
WCF
Relational Database (SQL, Oracle, MySQL, etc.)
Cloud Infrastructure-Ready
Monolithic Architecture
VMs (IaaS Azure)
SQL Server VM
Other RDBMS VMs
Your current IaaS Monitoring Tools
Your current Deployment Tools (Puppet, Chef, etc.)
Cloud DevOps-Ready
Monolithic Architecture
Windows Containers On VMs
Windows Containers on Service Fabric
Windows Containers on Kubernetes (ACS)
SQL DB Managed Instance
Monitoring (App Insights)
CI/CD (VSTS/Jenkins)
Microsoft .NET

Figure 1-5. *Example "pick and choose" scenario, with database on IaaS, DevOps, and containerization assets*

Next, as the ideal scenario for many existing .NET Framework applications to migrate, you could migrate to a Cloud DevOps-Ready application, to get big benefits from little work. This approach also sets you up for cloud optimization as a possible future step. Figure 1-6 shows an example.

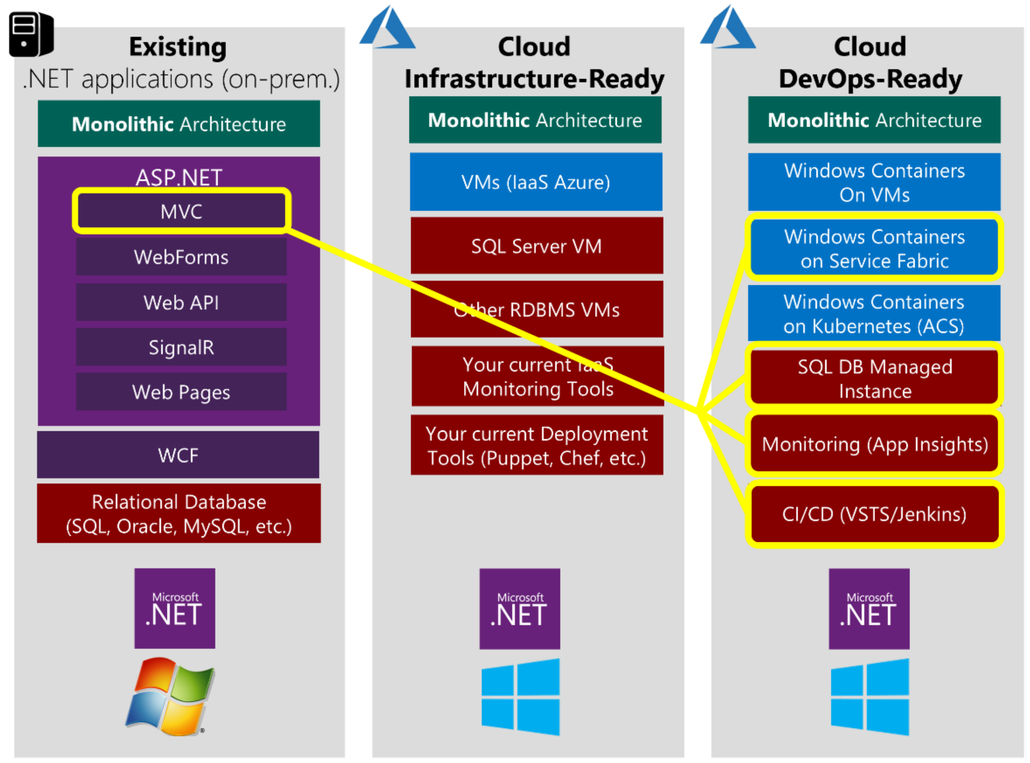
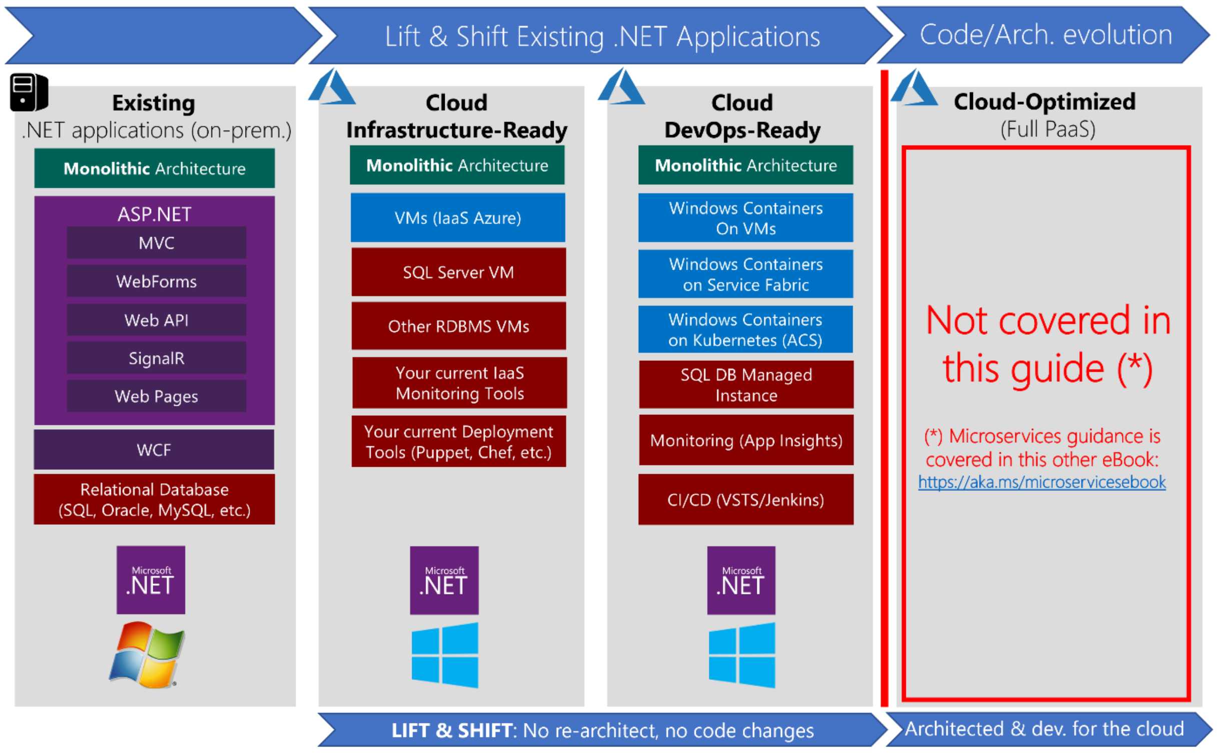

Existing
.NET applications (on-prem.)
Monolithic Architecture
ASP.NET
MVC
WebForms
Web API
SignalR
Web Pages
WCF
Relational Database (SQL, Oracle, MySQL, etc.)
Cloud Infrastructure-Ready
Monolithic Architecture
VMs (IaaS Azure)
SQL Server VM
Other RDBMS VMs
Your current IaaS Monitoring Tools
Your current Deployment Tools (Puppet, Chef, etc.)
Cloud DevOps-Ready
Monolithic Architecture
Windows Containers On VMs
Windows Containers on Service Fabric
Windows Containers on Kubernetes (ACS)
SQL DB Managed Instance
Monitoring (App Insights)
CI/CD (VSTS/Jenkins)
Microsoft .NET

Figure 1-6. *Example Cloud DevOps-Ready apps scenario, with Windows Containers and managed services*

Going even further, you could extend your existing Cloud DevOps-Ready application by adding a few microservices for specific scenarios. This would move you partially to the level of cloud-native in the Cloud-Optimized model, which is not the focus of the present guidance.

What this guide does not cover

This guide covers a specific subset of the example scenarios, as shown in Figure 1-7. This guide focuses only on lift and shift scenarios, and ultimately, on the Cloud DevOps-Ready model. In the Cloud DevOps-Ready model, a .NET Framework application is modernized by using Windows Containers, plus additional components like monitoring and CI/CD pipelines. Each component is fundamental to deploying applications to the cloud, faster, and with agility.



Lift & Shift Existing .NET Applications
Code/Arch. evolution
Existing
.NET applications (on-prem.)
Monolithic Architecture
ASP.NET
MVC
WebForms
Web API
SignalR
Web Pages
WCF
Relational Database (SQL, Oracle, MySQL, etc.)
Cloud Infrastructure-Ready
Monolithic Architecture
VMs (IaaS Azure)
SQL Server VM
Other RDBMS VMs
Your current IaaS Monitoring Tools
Your current Deployment Tools (Puppet, Chef, etc.)
Cloud DevOps-Ready
Monolithic Architecture
Windows Containers On VMs
Windows Containers on Service Fabric
Windows Containers on Kubernetes (ACS)
SQL DB Managed Instance
Monitoring (App Insights)
CI/CD (VSTS/Jenkins)
Cloud-Optimized
(Full PaaS)
Not covered in this guide (*)
(*) Microservices guidance is covered in this other eBook:
https://aka.ms/microservicesebook
Microsoft .NET
LIFT & SHIFT: No re-architect, no code changes
Architected & dev. for the cloud

Figure 1-7. *Lift and shift and initial modernization to Cloud DevOps-Ready applications*

The focus of this guide is specific. We show you the path you can take to achieve a lift and shift of your existing .NET applications, without re-architecting, and with no code changes. Ultimately, we show you how to make your application Cloud DevOps-Ready.

This guide doesn't show you how to work with cloud-native applications, such as how to evolve to a microservices architectures. To re-architect your applications or to create brand-new applications that are based on microservices, see the eBook [.NET Microservices: Architecture for containerized .NET applications](https://aka.ms/microservicesebook).

Additional resources

- **Containerized Docker application lifecycle with Microsoft platform and tools** (downloadable eBook): https://aka.ms/dockerlifecycleebook
- **.NET Microservices: Architecture for containerized .NET applications** (downloadable eBook): https://aka.ms/microservicesebook
- **Architecting modern web applications with ASP.NET Core and Azure** (downloadable eBook): https://aka.ms/webappebook

Who should use this guide

We wrote this guide for developers and solution architects who want to modernize existing ASP.NET applications that are based on the .NET Framework, for improved agility in shipping and releasing applications.

You also might find this guide useful if you are a technical decision maker, such as an enterprise architect or a development lead/director who just wants an overview of the benefits that you can get by using Windows Containers, and by deploying to the cloud when using Microsoft Azure.

How to use this guide

This guide addresses the "why"—why you might want to modernize your existing applications, and the specific benefits you get from using Windows Containers when you move your apps to the cloud. The content in the first few chapters of the guide is designed for architects and technical decision makers who want an overview, but who don't need to focus on implementation and technical, step-by-step details.

The last chapter of this guide introduces multiple walkthroughs that focus on specific deployment scenarios. In this guide, we offer shorter versions of the walkthroughs, to summarize the scenarios and highlight their benefits. The full walkthroughs drill down into setup and implementation details, and are published as a set of wiki posts in the same public GitHub repo where related sample apps reside (discussed in the next section). The last chapter and the step-by-step wiki walkthroughs on GitHub will be of more interest to developers and architects who want to focus on implementation details.

Sample apps for modernizing legacy apps: eShopModernizing

The eShopModernizing repo on GitHub offers two sample applications that simulate legacy monolithic web applications. One web app is developed by using ASP.NET MVC; the second web app is developed by using ASP.NET Web Forms. Both web apps are based on the traditional .NET Framework. These sample apps don't use .NET Core or ASP.NET Core as they are supposed to be existing/legacy .NET Framework applications to be modernized.

Both sample apps have a second version, with modernized code, and which are fairly straightforward. The most important difference between the app versions is that the second versions use Windows Containers as the deployment choice. There also are a few additions to the second versions, like Azure

Storage Blobs for managing images, Azure Active Directory for managing security, and Azure Application Insights for monitoring and auditing the applications.

Send us your feedback!

We wrote this guide to help you understand your options for improving and modernizing existing .NET web applications. The guide and related sample applications are evolving. We welcome your feedback! If you have comments about how this guide might be more helpful, please send them to dotnet-architecture-ebooks-feedback@service.microsoft.com.

Lift and shift existing .NET apps to Azure IaaS (Cloud Infrastructure-Ready)

Vision: As a first step, to reduce your on-premises investment and total cost of hardware and networking maintenance, simply rehost your existing applications in the cloud.

Before getting into *how* to migrate your existing applications to the Azure infrastructure as a service (IaaS) platform, it's important to analyze the reasons *why* you'd want to migrate directly to IaaS in Azure. The scenario at this modernization maturity level essentially is to start using VMs in the cloud, instead of continuing to use your current, on-premises infrastructure.

Another point to analyze is *why* you might want to migrate to pure IaaS cloud instead of just adding more advanced managed services in Azure. You need to determine what cases might require IaaS in the first place.

Figure 2-1 positions Cloud Infrastructure-Ready applications in the modernization maturity levels:

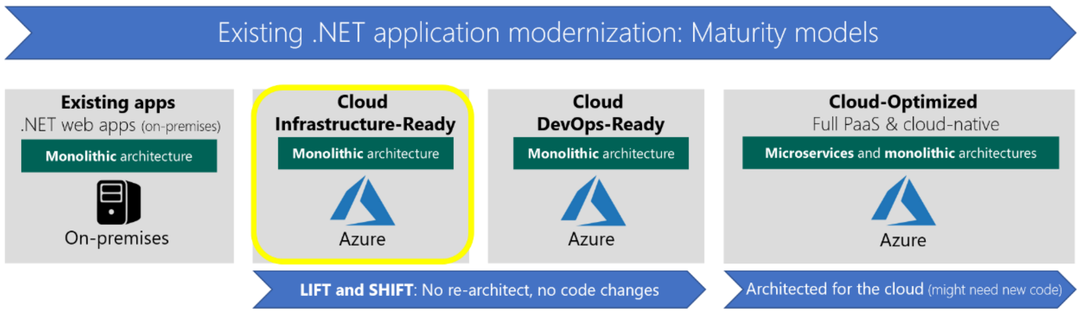

Existing .NET application modernization: Maturity models
Existing apps
.NET web apps (on-premises)
Monolithic architecture
On-premises
Cloud Infrastructure-Ready
Monolithic architecture
Azure
Cloud DevOps-Ready
Monolithic architecture
Azure
Cloud-Optimized
Full PaaS & cloud-native
Microservices and monolithic architectures
Azure
LIFT and SHIFT: No re-architect, no code changes
Architected for the cloud (might need new code)

Figure 2-1. *Positioning Cloud Infrastructure-Ready applications*

Why migrate existing .NET web applications to Azure IaaS

The main reason to migrate to the cloud, even at an initial IaaS level, is to achieve cost reductions. By using more managed infrastructure services, your organization can lower its investment in hardware maintenance, server or VM provisioning and deployment, and infrastructure management.

After you make the decision to move your apps to the cloud, the main reason why you might choose IaaS instead of more advanced options like PaaS is simply that the IaaS environment will be more familiar. Moving to an environment that's similar to your current, on-premises environment offers a lower learning curve, which makes it the quickest path to the cloud.

However, taking the quickest path to the cloud doesn’t mean that you will gain the most benefit from having your applications running in the cloud. Any organization will gain the most significant benefits from a cloud migration at the already introduced Cloud DevOps-Ready and PaaS (Cloud-Optimized) maturity levels.

It also has become evident that applications are easier to modernize and re-architect in the future when they are already running in the cloud, even on IaaS. This is true in part because application data migration has already been achieved. Also, your organization will have gained skills required for working in the cloud, and made the shift to operating in a “cloud culture.”

When to migrate to IaaS instead of to PaaS

In the next sections, we discuss Cloud DevOps-Ready applications that are mostly based on PaaS platforms and services. These apps give you the most benefits from migrating to the cloud.

If your goal is simply to move existing applications to the cloud, first, identify existing applications that will require substantial modification to run in Azure App Service. These apps should be the first candidates.

Then, if you don’t want or still cannot move to Windows Containers and or orchestrators like Azure Service Fabric or Kubernetes, yet, then is when you would use plain VMs (IaaS).

But, keep in mind that correctly configuring, securing, and maintaining VMs requires much more time and IT expertise compared to using PaaS services in Azure. If you are considering Azure Virtual Machines, make sure that you take into account the ongoing maintenance effort required to patch, update, and manage your VM environment. Azure Virtual Machines is IaaS.

Use Azure Migrate to analyze and migrate your existing applications to Azure

Migrating to the cloud doesn't have to be difficult. But many organizations struggle to get started — to get deep visibility into the environment and the tight interdependencies between applications, workloads, and data. Without that visibility, it can be difficult to plan the path forward. Without detailed information on what's required for a successful migration, you can't have the right conversations within your organization. You don't know enough about the potential cost benefits, or whether workloads could just lift-and-shift or would require significant rework to migrate successfully. No wonder many organizations hesitate.

Azure Migrate is a new service that provides the guidance, insights, and mechanisms needed to assist you in migrating to Azure. Azure Migrate provides:

- Discovery and assessment for on-premises virtual machines
- Inbuilt dependency mapping for high-confidence discovery of multi-tier applications
- Intelligent rightsizing to Azure virtual machines
- Compatibility reporting with guidelines for remediating potential issues
- Integration with Azure Database Management Service for database discovery and migration

Azure Migrate gives you confidence that your workloads can migrate with minimal impact to the business and run as expected in Azure. With the right tools and guidance, you can achieve maximum return on investment while assuring that critical performance and reliability needs are met.

Figure 2-2 shows you the built-in dependency mapping for all server and application connections performed by Azure Migrate.

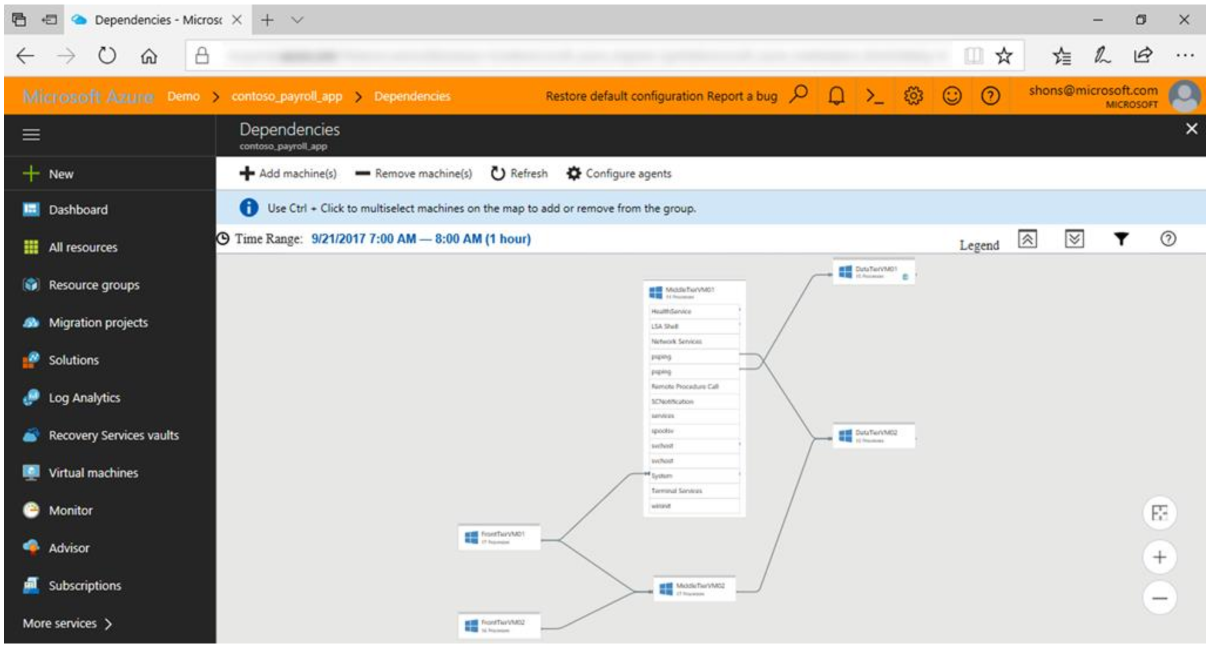


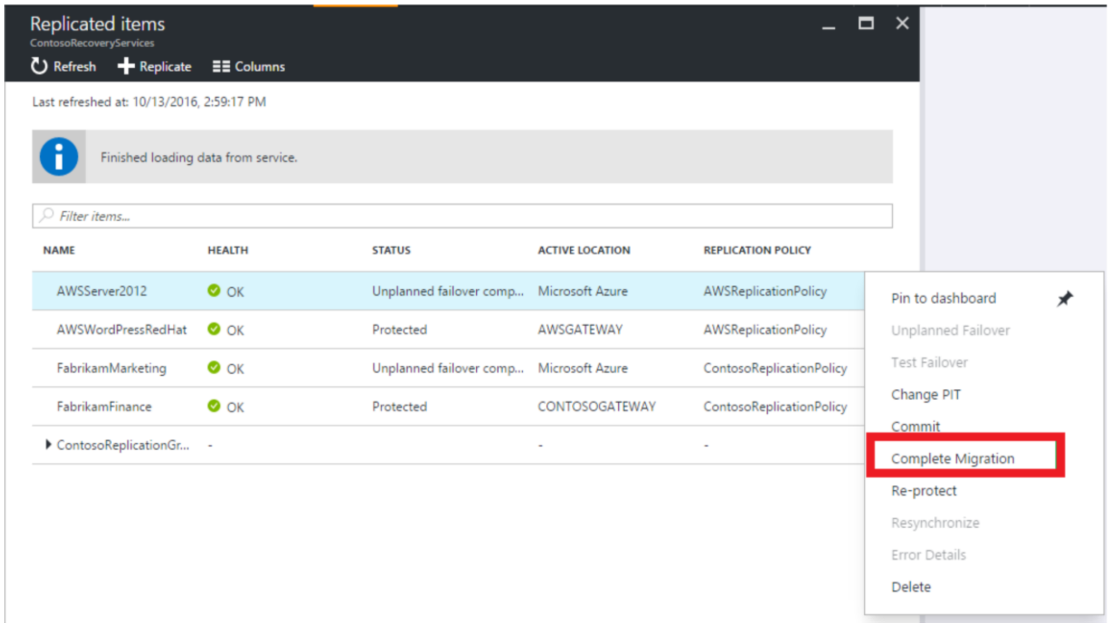
Figure 2-2. *Positioning Cloud Infrastructure-Ready applications*

Use Azure Site Recovery to migrate your existing VMs to Azure VMs

As part of the end-to-end Azure Migrate, Azure Site Recovery is a tool that you can use to easily migrate your web apps to VMs in Azure. You can use Site Recovery to replicate on-premises VMs and physical servers to Azure, or to replicate them to a secondary on-premises location. You can even replicate a workload that's running on a supported Azure VM, on an on-premises *Hyper-V* VM, on a *VMware* VM, or on a Windows or Linux physical server. Replication to Azure eliminates the cost and complexity of maintaining a secondary datacenter.

Site Recovery is also made specifically for hybrid environments that are partly on-premises and partly on Azure. Site Recovery helps ensure business continuity by keeping your apps that are running on VMs and on-premises physical servers available if a site goes down. It replicates workloads that are running on VMs and physical servers so that they remain available in a secondary location if the primary site isn't available. It recovers workloads to the primary site when it's up and running again.

Figure 2-3 shows the execution of multiple VM migrations by using Azure Site Recovery.



Replicated items
ContosoRecoveryServices

Refresh Replicate Columns

Last refreshed at: 10/13/2016, 2:59:17 PM

Finished loading data from service.

Filter items...

NAME	HEALTH	STATUS	ACTIVE LOCATION	REPLICATION POLICY
AWSServer2012	OK	Unplanned failover comp...	Microsoft Azure	AWSReplicationPolicy
AWSWordPressRedHat	OK	Protected	AWSGATEWAY	AWSReplicationPolicy
FabrikamMarketing	OK	Unplanned failover comp...	Microsoft Azure	ContosoReplicationPolicy
FabrikamFinance	OK	Protected	CONTOSOGATEWAY	ContosoReplicationPolicy
ContosoReplicationGr...	-		-	-

Figure 2-3. *Positioning Cloud Infrastructure-Ready applications*

Additional resources

- **Azure Migrate Datasheet**
  https://aka.ms/azuremigration_datasheet
- **Azure Migrate**
  http://azuremigrationcenter.com/
- **Migrate to Azure with Site Recovery**
  https://docs.microsoft.com/en-us/azure/site-recovery/site-recovery-migrate-to-azure
- **Azure Site Recovery service overview**
  https://docs.microsoft.com/en-us/azure/site-recovery/site-recovery-overview
- **Migrating VMs in AWS to Azure VMs**
  https://docs.microsoft.com/en-us/azure/site-recovery/site-recovery-migrate-aws-to-azure

Migrate your relational databases to Azure

Vision: Azure offers the most comprehensive database migration.

In Azure, you can migrate your database servers directly to IaaS VMs (pure lift and shift), or you can migrate to Azure SQL Database, for additional benefits. Azure SQL Database offers managed instance and full database-as-a-service (DBaaS) options. Figure 3-1 shows the multiple relational database migration paths available in Azure.

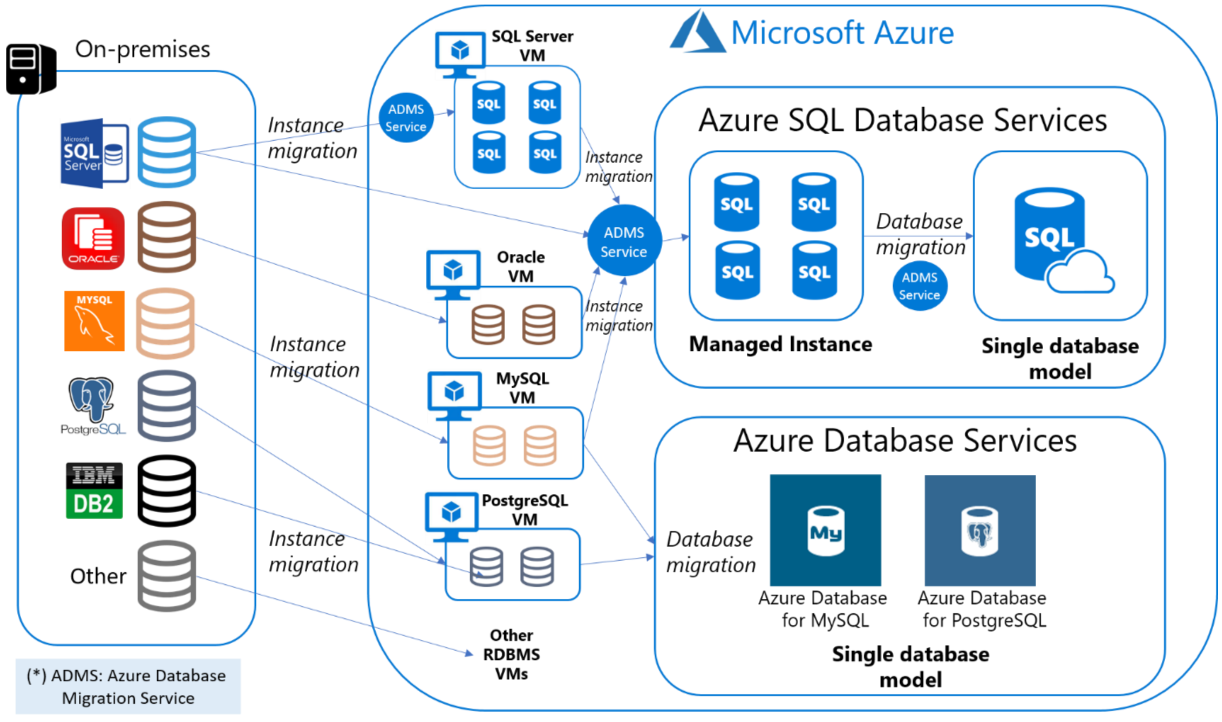


Figure 3-1. *Database migration paths in Azure*

When to migrate to Azure SQL Database Managed Instance

In most cases, Azure SQL Database Managed Instance will be your best option to consider when you migrate your data to Azure. If you are migrating SQL Server databases and need nearly 100% assurance that you won't need to re-architect your application or make changes to your data or data access code, choose the Managed Instance feature of Azure SQL Database.

Azure SQL Database Managed Instance is the best option if you have additional requirements for SQL Server instance-level functionality, or isolation requirements beyond the features provided in a standard Azure SQL Database (single database model). This last one is the most PaaS-oriented choice, but it doesn't offer the same features as that of a traditional SQL server. Migration might surface frictions.

For example, an organization that has made deep investments in instance-level SQL Server capabilities would benefit from migrating to SQL Managed Instance. Examples of instance-level SQL Server capabilities include SQL common language runtime (CLR) integration, SQL Server Agent, and cross-database querying. Support for these features are not available in standard Azure SQL Database (a single-database model).

An organization that operates in a highly regulated industry, and which needs to maintain isolation for security purposes, also might benefit from choosing the SQL Managed Instance model.

Managed Instance in Azure SQL Database has the following characteristics:

- Security isolation through Azure Virtual Network
- Application surface compatibility, with these features:
    - SQL Server Agent and SQL Server Profiler
    - Cross-database references and queries, SQL CLR, replication, change data capture (CDC), and Service Broker
- Database sizes up to 35 TB
- Minimum-downtime migration, with these features:
    - Azure Database Migration Service
    - Native backup and restore, and log shipping

With these capabilities, when you migrate existing application databases to Azure SQL Database, the Managed Instance model offers nearly 100% of the benefits of Paas for SQL Server. Managed Instance is a SQL Server environment where you continue using instance-level capabilities without changing your application design.

Managed Instance is probably the best fit for enterprises that currently are using SQL Server, and which require flexibility in their network security in the cloud. It's like having a private virtual network for your SQL databases.

When to migrate to Azure SQL Database

As mentioned, the standard Azure SQL Database is a fully managed, relational DBaaS. SQL Database currently manages millions of production databases, across 38 datacenters, around the world. It supports a broad range of applications and workloads, from managing straightforward transactional data, to driving the most data-intensive, mission-critical applications that require advanced data processing at a global scale.

Because of its full PaaS features and better pricing—and ultimately lower cost—you should move to the standard Azure SQL Database as your "by-default choice" if you have an application that uses basic, standard SQL databases, and no additional instance features. SQL Server features like SQL CLR integration, SQL Server Agent, and cross-database querying are not supported in the standard Azure SQL Database. Those features are available only in the Azure SQL Database Managed Instance model.

Azure SQL Database is the only intelligent cloud database service that's built for app developers. It's also the only cloud database service that scales on-the-fly, without downtime, to help you efficiently deliver multitenant apps. Ultimately, Azure SQL Database leaves you more time to innovate, and it accelerates your time to market. You can build secure apps, and connect to your SQL database by using the languages and platforms that you prefer.

Azure SQL Database offers the following benefits:

- Built-in intelligence (machine learning) that learns and adapts to your app
- On-demand database provisioning
- A range of offers, for all workloads
- 99.99% availability SLA, zero maintenance
- Geo-replication and restore services for data protection
- Azure SQL Database Point in Time Restore feature
- Compatibility with SQL Server 2016, including hybrid and migration

The standard Azure SQL Database is closer to PaaS than Azure SQL Database Managed Instance. You should try to use it, if possible, because you'll get more benefits from a managed cloud. However, Azure SQL Database has some key differences from regular and on-premises SQL Server instances. Depending on your existing application's database requirements, and your enterprise requirements and policies, it might not be the best choice when you are planning your migration to the cloud.

When to move your original RDBMS to a VM (IaaS)

One of your migration options is to move your original relational database management system (RDBMS), including Oracle, IBM DB2, MySQL, PostgreSQL, or SQL Server, to a similar server that's running on an Azure VM. If you have existing applications that require the fastest migration to the cloud with minimal changes, or no changes at all, a direct migration to IaaS in the cloud might be a fair option. It might not be the best way to take advantage of all the cloud's benefits, but it's probably the fastest initial path.

Currently, Microsoft Azure supports up to 331 different database servers deployed as IaaS VMs. These include popular RDBMSes like SQL Server, Oracle, MySQL, PostgreSQL, and IBM DB2, and many other NoSQL databases like MongoDB, Cassandra, DataStax, MariaDB, and Cloudera.

Note that although moving your RDBMS to an Azure VM might be the fastest way to migrate your data to the cloud (because it is IaaS), this approach requires a significant investment in your IT teams (database administrators and IT pros). Enterprise teams need to be able to set up and manage high availability, disaster recovery, and patching for SQL Server. This context also needs a customized environment, with full administrative rights.

When to migrate to SQL Server as a VM (IaaS)

There might be a few cases where you still need to migrate to SQL Server as a regular VM. An example scenario is if you need to use SQL Server Reporting Services. In most cases, though, Azure SQL Database Managed Instance can provide everything you need to migrate from on-premises SQL servers, so migration to a SQL Server VM should be your last resort to try.

Use Azure Database Migration Service to migrate your relational databases to Azure

You can use Azure Database Migration Service to migrate relational databases like SQL Server, Oracle, and MySQL to Azure, whether your target database is Azure SQL Database, Azure SQL Database Managed Instance, or SQL Server on an Azure VM.

The automated workflow, with assessment reporting, guides you through the changes you need to make before you migrate the database. When you are ready, the service migrates the source database to Azure.

Whenever you change an original RDBMS, you might need to retest. You also might need to change the SQL sentences or Object-Relational Mapping (ORM) code in your application, depending on testing results.

If you have any other database (for example, IBM DB2) and you opt for a lift and shift approach, you might want to continue using those databases as IaaS VMs in Azure, unless you are willing to perform a more complex data migration. A more complex data migration will require additional effort, because you'd be migrating to a different database type with new schema and different programming libraries.

To learn how to migrate databases by using Azure Database Migration Service, see Get to the cloud faster with Azure SQL Database Managed Instance and Azure Database Migration Service.

Additional resources

- **Choose a cloud SQL Server option: Azure SQL Database (PaaS) or SQL Server on Azure VM (IaaS)**
  https://docs.microsoft.com/en-us/azure/sql-database/sql-database-paas-vs-sql-server-iaas
- **Get to the cloud faster with Azure SQL DB Managed Instance and Database Migration Service**
  https://channel9.msdn.com/Events/Build/2017/P4008
- **SQL Server database migration to SQL Database in the cloud**
  https://docs.microsoft.com/en-us/azure/sql-database/sql-database-cloud-migrate
- **Azure SQL Database**
  https://azure.microsoft.com/en-us/services/sql-database/?v=16.50
- **SQL Server on virtual machines**
  https://azure.microsoft.com/en-us/services/virtual-machines/sql-server/

CHAPTER 4

Lift and shift existing .NET apps to Cloud DevOps-Ready applications

Vision: Lift and shift your existing .NET Framework applications to Cloud DevOps-Ready applications to drastically improve your deployment agility, so you can ship faster and lower app delivery costs.

To take advantage of the benefits of the cloud and new technologies like containers, you should at least partially modernize your existing .NET applications. Ultimately, modernizing your enterprise applications will lower your total cost of ownership.

Partially modernizing an app doesn’t necessarily mean a full migration and re-architecture. You can initially modernize your existing applications by using a lift and shift process that's easy and fast. You can maintain your current code base, written in existing .NET Framework versions, with any Windows and IIS dependencies. Figure 4-1 highlights how Cloud DevOps-Ready apps are positioned in Azure application modernization maturity models.

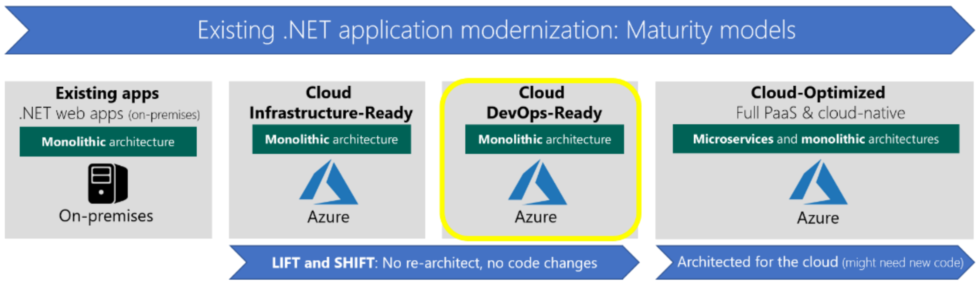

Existing .NET application modernization: Maturity models
Existing apps
.NET web apps (on-premises)
Monolithic architecture
On-premises
Cloud Infrastructure-Ready
Monolithic architecture
Azure
Cloud DevOps-Ready
Monolithic architecture
Azure
Cloud-Optimized
Full PaaS & cloud-native
Microservices and monolithic architectures
Azure
LIFT and SHIFT: No re-architect, no code changes
Architected for the cloud (might need new code)

Figure 4-1. *Positioning Cloud DevOps-Ready applications*

Reasons to lift and shift existing .NET apps to Cloud DevOps-Ready applications

With a Cloud DevOps-Ready application, you can rapidly and repeatedly deliver reliable applications to your customers. You gain essential agility and reliability by deferring much of the operational complexity of your app to the platform.

If you can't get your applications to market quickly, by the time you ship your app, the market you were targeting will have evolved. You might be too late, no matter how well the application was architected or engineered. You might be failing or not reaching your full potential because you can't sync app delivery with the needs of the market.

The need for continuous business innovation pushes development and operations teams to the limit. The only way to achieve the agility you need in continuous business innovation is by modernizing your applications with technologies like containers and specific Cloud DevOps-Ready application principles.

The bottom line is that when an organization builds and manages applications that are Cloud DevOps-Ready, it can put solutions in the hands of customers sooner, and bring new ideas to market when they are relevant.

Cloud DevOps-Ready application principles and tenets

Improvements in the cloud are mostly focused on meeting two goals: Reduce costs, and improve business growth by improving agility. These goals are achieved by simplifying processes and reducing friction when you release and ship applications.

Your application is Cloud DevOps-Ready if you can—in an agile manner—develop your app autonomously from other on-premises apps, and then release, deploy, auto-scale, monitor, and troubleshoot your app in the cloud.

The key is *agility*. You can't ship with agility unless you reduce to an absolute minimum any deployment-to-production issues and dev/test environment issues. Containers (specifically, Docker, as a de facto standard) and managed services were designed specifically for this purpose.

To achieve agility, you also need automated DevOps processes that are based on CI/CD pipelines that release to scalable platforms in the cloud. CI/CD platforms (like Visual Studio Team Services or Jenkins) that deploy to a scalable and resilient cloud platform (like Azure Service Fabric or Kubernetes) are key technologies for achieving agility in the cloud.

The following list describes the main tenets or practices for Cloud DevOps-Ready applications. Note that you can adopt all or only some of these principles, in a progressive or incremental approach:

- **Containers**. Containers give you the ability to include application dependencies with the application itself. Containerization significantly reduces the number of issues you might encounter when you deploy to production environments or test in staging environments. Ultimately, containers improve the agility of application delivery.

- **Resilient and scalable cloud**. The cloud provides a platform that is managed, elastic, scalable, and resilient. These characteristics are fundamental to gain cost improvements and ship highly available and reliable applications in a continuous delivery. Managed services like managed databases, managed cache as a service (CaaS), and managed storage are fundamental pieces in alleviating the maintenance costs of your application.
- **Monitoring**. You can't have a reliable application without having a good way to detect and diagnose exceptions and application performance issues. You need to get actionable insights through application performance management and instant analytics.
- **DevOps culture and continuous delivery**. Adopting Cloud DevOps-Ready practices requires a cultural change in which teams no longer work in independent silos. CI/CD pipelines are possible only when there is an increased collaboration between development and IT operations teams, supported by containers and CI/CD tools.

Figure 4–2 shows the main optional pillars of a Cloud DevOps-Ready application. The more pillars you implement, the readier your application will be to succeed in meeting your customers' expectations.

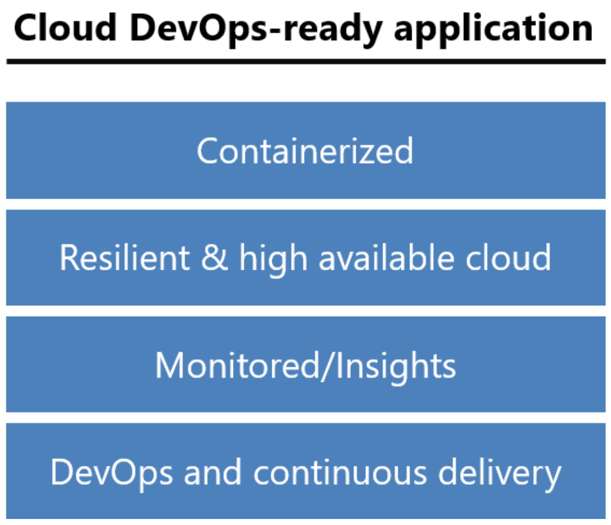

Cloud DevOps-ready application
Containerized
Resilient & high available cloud
Monitored/Insights
DevOps and continuous delivery

Figure 4-2. *Main pillars of a Cloud DevOps-ready application*

To summarize, a Cloud DevOps-Ready application is an approach to building and managing applications that takes advantage of the cloud computing model, while using a combination of containers, managed cloud infrastructure, resilient application techniques, monitoring, continuous delivery, and DevOps, all without the need to re-architect and recode your existing applications.

Your organization can adopt these technologies and approaches gradually. You don't have to embrace all of them, all at once. You can adopt them incrementally, depending on enterprise priorities and user needs.

Benefits of a Cloud DevOps-Ready application

You can get the following benefits by converting an existing application to a Cloud DevOps-ready application (without re-architecting or coding):

- **Lower costs, because the managed infrastructure is handled by the cloud provider**. Cloud DevOps-Ready applications get the benefits of the cloud by using the cloud's out-of-the-box elasticity, autoscale, and high availability. Benefits are related not only to the compute features (VMs and containers), but also depend on resources in the cloud, like DBaaS, CaaS, and any infrastructure an application might needed.
- **Resilient application and infrastructure**. When you migrate to the cloud, you need to embrace transient failures; failures will occur in the cloud. Also, cloud infrastructure and hardware is "replaceable," which increases opportunities for transient downtime. At the same time, inner cloud capabilities and certain application development techniques that implement resiliency and automate recovery make it much easier to recover from unexpected failures in the cloud.
- **Deeper insights into application performance**. Cloud monitoring tools like Azure Application Insights provide visualization for health management, logging, and notifications. Audit logs make applications easy to debug and audit. This is fundamental for a reliable cloud application.
- **Application portability, with agile deployments**. Containers (either Linux or Windows containers based on Docker Engine) offer the best solution to avoiding a cloud-locked application. By using containers, Docker hosts, and multi-cloud orchestrators, you can easily move from one environment or cloud to another. Containers eliminate the friction that typically occurs in deployments to any environment (stage/test/production).

All of these benefits ultimately provide key cost reductions for your end-to-end application lifecycle.

In the following sections, these benefits are explained in more detail, and are linked to specific technologies.

Microsoft technologies in Cloud DevOps-Ready applications

The following list describes the tools, technologies, and solutions that are recognized as requirements for Cloud DevOps-Ready apps. You can adopt Cloud DevOps-Ready apps selectively or gradually, depending on your priorities.

- **Cloud infrastructure**: The infrastructure that provides the compute platform, operating system, network, and storage. Microsoft Azure is positioned at this level.
- **Runtime**: This layer provides the environment for the application to run. If you are using containers, this layer usually is based on Docker Engine, running either on Linux hosts or on Windows hosts. (Windows Containers are supported beginning with Windows Server 2016. Windows Containers is the best choice for existing .NET Framework applications that run on Windows.)
- **Managed cloud**: When you choose a managed cloud option, you can avoid the expense and complexity of managing and supporting the underlying infrastructure, VMs, OS patches, and networking configuration. If you choose to migrate by using IaaS, you are responsible for all

of these tasks, and for associated costs. In a managed cloud option, you manage only the applications and services that you develop. The cloud service provider typically manages everything else. Examples of managed cloud services in Azure include Azure SQL Database, Azure Redis Cache, Azure Cosmos DB, Azure Storage, Azure Database for MySQL, Azure Database for PostgreSQL, Azure Active Directory, and managed compute services like VM scale sets, Azure Service Fabric, Azure App Service, and Azure Container Service.

- **Application development**: You can choose from many languages when you build applications that run in containers. In this guide, we focus on .NET, but, you can develop container-based apps by using other languages, like Node.js, Python, Spring/Java, or GoLang.
- **Monitoring, telemetry, logging, and auditing**: The ability to monitor and audit applications and containers that are running in the cloud is critical for any Cloud DevOps-Ready application. Azure Application Insights and Microsoft Operations Management Suite are the main Microsoft tools that provide monitoring and auditing for Cloud DevOps-Ready apps.
- **Provisioning**: Automation tools help you provision the infrastructure and deploy an application to multiple environments (production, testing, staging). You can use tools like Chef and Puppet to manage an application's configuration and environment. This layer also can be implemented by using simpler and more direct approaches. For example, you can deploy directly by using Azure command-line interface (Azure CLI) tooling, and then use the continuous deployment and release management pipelines in Visual Studio Team Services.
- **Application lifecycle**: Visual Studio Team Services and other tools, like Jenkins, are build automation servers that help you implement CI/CD pipelines, including release management.

The next sections of this chapter, and the related walkthroughs, focus specifically on details about the runtime layer (Windows Containers). The guidance describes the ways you can deploy Windows Containers on Windows Server 2016 (and later versions) VMs. It also covers more advanced orchestrator layers, like Azure Service Fabric, Kubernetes, and Azure Container Service. Setting up orchestrator layers is a fundamental requirement for modernizing existing .NET Framework (Windows-based) applications as Cloud DevOps-Ready applications.

Monolithic applications *can* be Cloud DevOps-Ready

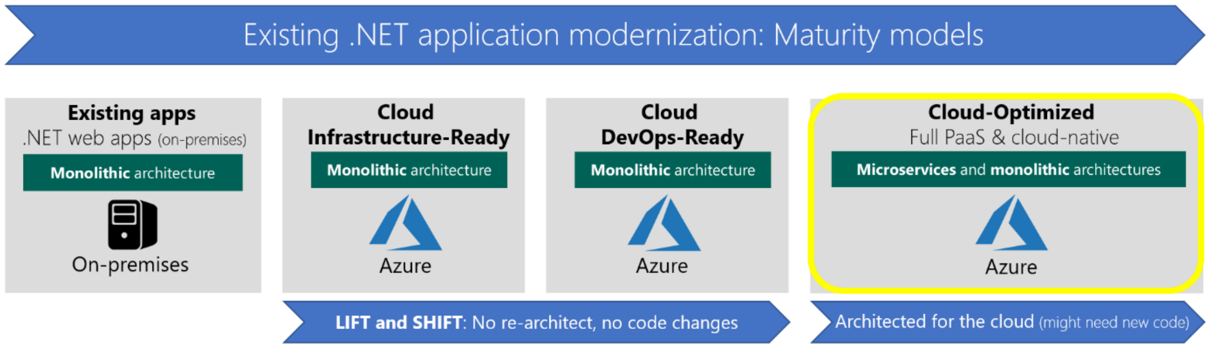
It's important to highlight that monolithic applications (applications that are not based on microservices) *can* be Cloud DevOps-Ready applications. You can build and operate monolithic applications that take advantage of the cloud computing model by using a combination of containers, continuous delivery, and DevOps. If an existing monolithic application is right for your business goals, you can modernize it and make it Cloud DevOps-Ready.

Similarly, if monolithic applications can be Cloud DevOps-Ready applications, other, more complex architectures like N-Tier applications also can be modernized as Cloud DevOps-Ready applications.

What about Cloud-Optimized applications?

Although Cloud-Optimized and cloud-native applications are not the main focus of this guide, it's helpful to have an understanding of this modernization maturity level, and to distinguish it from Cloud DevOps-Ready.

Figure 4-3 positions Cloud-Optimized apps in the application modernization maturity levels:



Existing .NET application modernization: Maturity models
Existing apps
.NET web apps (on-premises)
Monolithic architecture
On-premises
Cloud Infrastructure-Ready
Monolithic architecture
Azure
Cloud DevOps-Ready
Monolithic architecture
Azure
Cloud-Optimized
Full PaaS & cloud-native
Microservices and monolithic architectures
Azure
LIFT and SHIFT: No re-architect, no code changes
Architected for the cloud (might need new code)

Figure 4-3. *Positioning Cloud-Optimized applications*

The Cloud-Optimized modernization maturity level usually requires new development investments. Moving to the Cloud-Optimized level typically is driven by business need to modernize applications as much as possible to lower costs and increase agility and compete advantage. These goals are accomplished by maximizing the use of cloud PaaS. This means not only using PaaS services like DBaaS, CaaS, and storage as a service (STaaS), but also by migrating your own applications and services to a PaaS compute platform like Azure App Service, or using orchestrators.

This type of modernization usually requires refactoring or writing new code that is optimized for cloud PaaS platforms (like for Azure App Service). It might even require architecting specifically for the cloud environment, especially if you are moving to cloud-native application models that are based on microservices. This is a key differentiating factor compared to Cloud DevOps-Ready, which requires no re-architecting and no new code.

In some more advanced cases, you might create cloud-native applications based on microservices architectures, which can evolve with agility and scale to limits that would be difficult to achieve in a monolithic architecture deployed to an on-premises environment.

Figure 4-4 shows the type of applications that you can deploy when you use the Cloud-Optimized model. You have two basic choices—modern web applications and cloud-native applications.

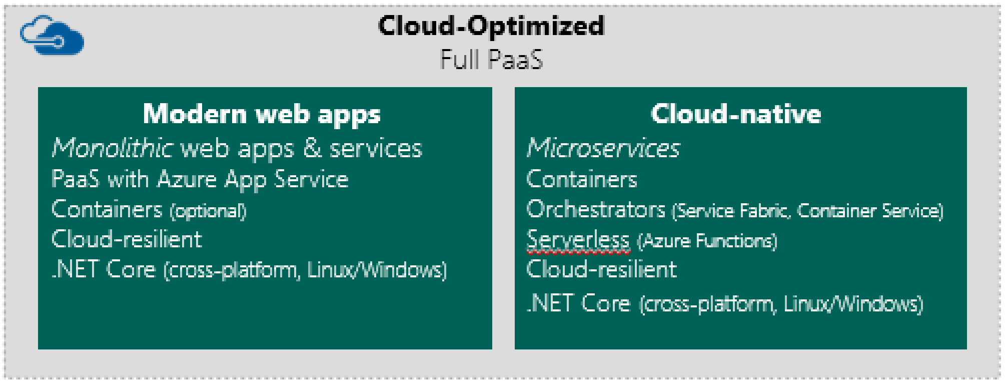

Cloud-Optimized
Full PaaS
Modern web apps
Monolithic web apps & services
PaaS with Azure App Service
Containers (optional)
Cloud-resilient
.NET Core (cross-platform, Linux/Windows)
Cloud-native
Microservices
Containers
Orchestrators (Service Fabric, Container Service)
Serverless (Azure Functions)
Cloud-resilient
.NET Core (cross-platform, Linux/Windows)

Figure 4-4. *App types at the Cloud-Optimized level*

You can extend basic modern web apps and cloud-native apps by adding other services, like artificial intelligence (AI), machine learning (ML), and IoT. You might use any of these services to extend any of the possible Cloud-Optimized approaches.

The fundamental difference in applications at the Cloud-Optimized level is in the application architecture. Cloud-native applications are, by definition, apps that are based on microservices. Cloud-native apps require special architectures, technologies, and platforms, compared to a monolithic web application or traditional N-Tier application.

Creating new applications that don't use microservices also makes sense. There are many new and still modern scenarios in which a microservices-based approach might exceed your needs. In some cases, you might just want to create a simpler monolithic web application, or add coarse-grained services to an N-Tier app. In these cases, you can still make full use of cloud PaaS capabilities like the ones offered by Azure App Service. You still reduce your maintenance work to the limit.

Also, because you develop new code in Cloud-Optimized scenarios (for a full application or for partial subsystems), when you create new code, you should use the newer versions of .NET (.NET Core and ASP.NET Core, in particular). This is especially true if you create microservices and containers because .NET Core is a lean and fast framework. You'll get a small memory footprint and fast start in containers, and your applications will be high-performing. This approach fits perfectly with the requirements of microservices and containers, and you get the benefits of a cross-platform framework—being able to run the same application on Linux, Windows Server, and Mac (Mac for development environments).

Cloud-native applications with Cloud-Optimized applications

Cloud-native is a more advanced or mature state for large and mission-critical applications. Cloud-native applications usually require architecture and design that are created from scratch instead of by modernizing existing applications. The key difference between a cloud-native application and a simpler Cloud-Optimized web app deployed to PaaS is the recommendation to use microservices architectures in a cloud-native approach. Cloud-Optimized apps can also be monolithic web apps or N-Tier apps deployed to a cloud PaaS service like Azure App Service.

The Twelve-Factor App (a collection of patterns that are closely related to microservices approaches) also is considered a requirement for cloud-native application architectures.

The Cloud Native Computing Foundation (CNCF) is a primary promoter of cloud-native principles. Microsoft is a member of the CNCF.

For a sample definition and for more information about the characteristics of cloud-native applications, see the Gartner article How to architect and design cloud-native applications. For specific guidance from Microsoft about how to implement a cloud-native application, see .NET microservices: Architecture for containerized .NET applications.

The most important factor to consider if you migrate a full application to the cloud-native model is that you must re-architect to a microservices-based architecture. This clearly requires a significant investment in development because of the large refactoring process involved. This option usually is chosen for mission-critical applications that need new levels of scalability and long-term agility. But, you could start moving toward cloud-native by adding microservices for just a few new scenarios, and eventually refactor the application fully as microservices. This is an incremental approach that is the best option for some scenarios.

What about microservices?

Understanding microservices and how they work is important when you are considering cloud-native applications for your organization.

The microservices architecture is an advanced approach that you can use for applications that are created from scratch or when you evolve existing applications (either on-premises or cloud DevOps-Ready apps) toward cloud-native applications. You can start by adding a few microservices to existing applications to learn about the new microservices paradigms. But clearly, you need to architect and code, especially for this type of architectural approach.

However, microservices are not mandatory for any new or modern application. Microservices are not a "magic bullet," and they aren't the single, best way to create every application. How and when you use microservices depends on the type of application that you need to build.

The microservices architecture is becoming the preferred approach for distributed and large or complex mission-critical applications that are based on multiple, independent subsystems in the form of autonomous services. In a microservices-based architecture, an application is built as a collection of services that can be independently developed, tested, versioned, deployed, and scaled. This can include any related, autonomous database per microservice.

For a detailed look at a microservices architecture that you can implement by using .NET Core, see the downloadable PDF eBook .NET microservices: Architecture for containerized .NET applications. The guide also is available online.

But even in scenarios in which microservices offer powerful capabilities—independent deployment, strong subsystem boundaries, and technology diversity—they also raise many new challenges. The challenges are related to distributed application development, such as fragmented and independent data models; achieving resilient communication between microservices; the need for eventual consistency; and operational complexity. Microservices introduce a higher level of complexity compared to traditional monolithic applications.

Because of the complexity of a microservices architecture, only specific scenarios and certain application types are suitable for microservice-based applications. These include large and complex applications that have multiple, evolving subsystems. In these cases, it's worth investing in a more complex software architecture, for increased long-term agility and more efficient application maintenance. But for less complex scenarios, it might be better to continue with a monolithic application approach or simpler N-Tier approaches.

As a final note, even at the risk of being repetitive about this concept, you shouldn't look at using microservices in your applications as "all-in or nothing at all." You can extend and evolve existing monolithic applications by adding new, small scenarios based on microservices. You don't need to start from scratch to start working with a microservices architecture approach. In fact, we recommend that you evolve from using an existing monolithic or N-Tier application by adding new scenarios. Eventually, you can break down the application into autonomous components or microservices. You can start evolving your monolithic applications in a microservices direction, step by step.

When to use Azure App Service for modernizing existing .NET apps

When you modernize existing ASP.NET web applications to the Cloud-Optimized maturity level, because your web applications were developed by using the .NET Framework, your main dependencies are on Windows and, most likely, Internet Information Server (IIS). You can use and deploy Windows-based and IIS-based applications either by directly deploying to Azure App Service or by first containerizing your application by using Windows Containers. If containerizing, deploy the applications either to Windows Containers hosts (VM-based) or to an Azure Service Fabric cluster that supports Windows Containers.

When you use Windows Containers, you get all the benefits of containerization. You increase agility in shipping and deploying your app, and reduce friction for environment issues (staging, dev/test, production). In the next few sections, we go into more detail about the benefits you get from using containers.

As of the writing of this guide, Azure App Service does not support Windows Containers. It does support containers for Linux. So, your next question might be, "How do I choose between Azure App Service and Windows Containers?"

Basically, if Azure App Service works for your application and there aren't any server or custom dependencies blocking the path to use App Service, you should migrate your existing .NET web application to App Service. That's the easiest, most effective way to maintain your application. In Azure, your application also will have a simpler maintenance path because of the benefits from PaaS infrastructure, like DBaaS, CaaS, and STaaS.

However, if your application does have server or custom dependencies that are not supported in Azure App Service, you might need to consider options that are based on Windows Containers. Examples of server or custom dependencies include third-party software or an .msi file that needs to be installed on the server, but which is not supported in Azure App Service. Another example is any other server configuration that's not supported in Azure App Service, like using assemblies in the Global Assembly Cache (GAC) or COM/COM+ components. Thanks to Windows container images, you can include your custom dependencies in the same "unit of deployment."

Alternatively, you could refactor the areas of your application that are not supported by Azure App Service. Depending on the volume of work refactoring would require, you'd have to carefully evaluate whether that's worth doing.

Benefits of moving to Azure App Service

Azure App Service is a fully managed PaaS offering that makes it easy to build web apps that are backed by business processes. When you use App Service, you avoid the infrastructure management costs associated with upgrading and maintaining web apps on-premises. Specifically, you cut the hardware and licensing costs of running web apps on-premises.

If your web application is suitable for migrating to Azure App Service, the main benefit is the short amount of time it takes to move the app. App Service offers a very easy environment in which to get started.

Azure App Service is the best choice for most web apps because it's the simplest PaaS in Azure that you can use to run web apps. Deployment and management are integrated into the platform, sites scale quickly to handle high traffic loads, and the built-in load balancing and traffic manager provide high availability.

Even monitoring your web apps is simple, through Azure Application Insights. Application Insights comes free with App Service, and doesn't require writing any special code in your application. Just run your web app in App Service, and you'll get a compelling monitoring system, with no additional work.

With App Service, you can also directly use many open-source apps from the Azure Web Application Gallery (like WordPress or Umbraco), or you can create a new site by using the framework and tools of your choice, like ASP.NET. The App Service WebJobs feature makes it easy to add background job processing to your App Service web app.

Key advantages of migrating your web apps by using the Web Apps feature of Azure App Service include the following:

- Automatic scaling to meet demand during busy times and reduce costs during quiet times.
- Automatic site backups to protect changes and data.
- High availability and resilience on the Azure PaaS platform.
- Deployment slots for development and staging environments, and for testing multiple site designs.
- Load balancing and Distributed Denial of Service (DDoS) protection.
- Traffic management to direct users to the closest geographic deployment.

Although App Service might be the best choice for new web apps, however, for existing applications, App Service might be the best choice only if your application dependencies are supported in App Service.

Additional resources

- **Compatibility analysis for Azure App Service**
  https://www.migratetoazure.net/Resources

Benefits of moving to Windows Containers

The main benefit of using Windows Containers is that you gain a more reliable and improved deployment experience, compared to non-containerized apps. In addition, having your application modernized with containers effectively makes your application ready for many other platforms and clouds that support Windows Containers. The benefits of moving to Windows Containers are covered in more detail in the next sections.

The primary compute environments in Azure (in general availability, as of mid-2017) that support Windows Containers are Azure Service Fabric and basic Windows Containers hosts (Windows Server 2016 VMs). These environments are the main infrastructure scenarios covered in this guide.

You also can deploy Windows Containers to other orchestrators, like Kubernetes, Docker Swarm, or DC/OS. Currently (early fall 2017), these platforms are in preview in Azure Container Service for using Windows Containers.

How to deploy existing .NET apps to Azure App Service

The Web Apps feature of Azure App Service is a fully managed compute platform that is optimized for hosting websites and web apps. This PaaS offering in Microsoft Azure lets you focus on your business logic, while Azure takes care of the infrastructure to run and scale your apps.

Validate sites and migrate to App Service with Azure App Service Migration Assistant

When you create a new application in Visual Studio, moving the app to App Service usually is straightforward. However, if you are planning to migrate an existing application to App Service, first you need to evaluate whether all your application's dependencies are compatible with App Service. This includes dependencies like server OS and any third-party software that's installed on the server.

You can use Azure App Service Migration Assistant to analyze sites and then migrate them from Windows and Linux web servers to App Service. As part of the migration, the tool creates web apps and databases on Azure as needed, publishes content, and publishes your database.

Azure App Service Migration Assistant supports migrating from IIS running on Windows Server to the cloud. App Service supports Windows Server 2003 and later versions.

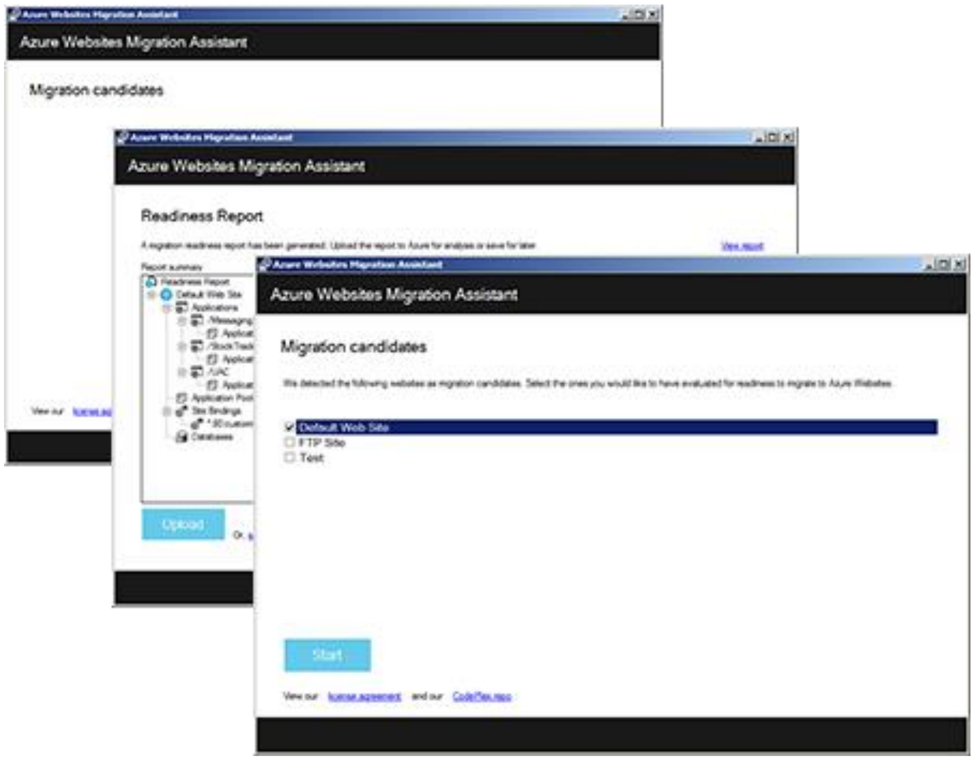


Figure 4-5. Using Azure App Service Migration Assistant

App Service Migration Assistant is a tool that moves your websites from your web servers to the Azure cloud.

After a website is migrated to App Service, the site has everything it needs to run safely and efficiently. Sites are set up and run automatically in the Azure cloud PaaS service (App Service).

The App Service migration tool can analyze your websites and report on their compatibility for moving to App Service. If you're happy with the analysis, you can let App Service Migration Assistant migrate content, data, and settings for you. If a site is not quite compatible, the migration tool tells you what you need to adjust to make it work.

Additional resources

- **Azure App Service Migration Assistant**
  https://www.migratetoazure.net/

Deploy existing .NET apps as Windows containers

Deployments that are based on Windows Containers are applicable to Cloud-Optimized applications, cloud-native applications, and Cloud DevOps-Ready applications.

In this guide, and in the following sections, we focus on using Windows Containers for *Cloud DevOps-Ready* applications, when you lift and shift existing .NET applications.

What are containers? (Linux or Windows)

Containers are a way to wrap up an application into its own isolated package. In its container, the application is not affected by applications or processes that exist outside of the container. Everything the application depends on to run successfully as a process is inside the container. Wherever the container might move, the requirements of the application will always be met, in terms of direct dependencies, because it is bundled with everything that it needs to run (library dependencies, runtimes, and so on).

The main characteristic of a container is that it makes the environment the same across different deployments because the container itself comes with all the dependencies it needs. This means that you can debug the application on your machine, and then deploy it to another machine, with the same environment guaranteed.

A container is an instance of a container image. A container image is a way to package an app or service (like a snapshot), and then deploy it in a reliable and reproducible way. You could say that Docker is not only a technology—it's also a philosophy and a process.

As containers daily become more common, they are becoming an industry-wide "unit of deployment."

Benefits of containers (Docker Engine on Linux or Windows)

Building applications by using containers—which also might be defined as lightweight building blocks—offers a significant increase in agility for building, shipping, and running any application, across any infrastructure.

With containers, you can take any app from development to production with little or no code change, thanks to Docker integration across Microsoft developer tools, operating systems, and cloud.

When you deploy to plain VMs, you probably already have a method in place for deploying ASP.NET apps to your VMs. It's likely, though, that your method involves multiple manual steps or complex automated processes by using a deployment tool like Puppet, or a similar tool. You might need to perform tasks like modifying configuration items, copying application content between servers, and running interactive setup programs based on .msi setups, followed by testing. All those steps in the deployment add time and risk to deployments. You will get failures whenever a dependency is not present in the target environment.

In Windows Containers, the process of packaging applications is fully automated. Windows Containers is based on the Docker platform, which offers automatic updates and rollbacks for container deployments. The main improvement you get from using the Docker engine is that you create images, which are like snapshots of your application, with all its dependencies. The images are Docker images (a Windows container image, in this case). The images run ASP.NET apps in containers, without going back to source code. The container snapshot becomes the unit of deployment.

A large number of organizations are containerizing existing monolithic applications for the following reasons:

- **Release agility through improved deployment**. Containers offer a consistent deployment contract between development and operations. When you use containers, you won't hear developers say, “It works on my machine, why not in production?” They can simply say, “It runs as a container, so it'll run in production.” The packaged application, with all its dependencies, can be executed in any supported container-based environment. It will run the way it was intended to run in all deployment targets (dev, QA, staging, production). Containers eliminate most frictions when they move from one stage to the next, which greatly improves deployment, and you can ship faster.
- **Cost reductions**. Containers lead to lower costs, either by the consolidation and removal of existing hardware, or from running applications at a higher density per unit of hardware.
- **Portability**. Containers are modular and portable. Docker containers are supported on any server operating system (Linux and Windows), in any major public cloud (Microsoft Azure, Amazon AWS, Google, IBM), and in on-premises and private or hybrid cloud environments.
- **Control**. Containers offer a flexible and secure environment that's controlled at the container level. A container can be secured, isolated, and even limited by setting execution constraint policies on the container. As detailed in the section about Windows Containers, Windows Server 2016 and Hyper-V containers offer additional enterprise support options.

Significant improvements in agility, portability, and control ultimately lead to significant cost reductions when you use containers to develop and maintain applications.

What is Docker?

Docker is an open-source project that automates the deployment of applications as portable, self-sufficient containers that can run in the cloud or on-premises. Docker also is a company that promotes and evolves this technology. The company works in collaboration with cloud, Linux, and Windows vendors, including Microsoft.

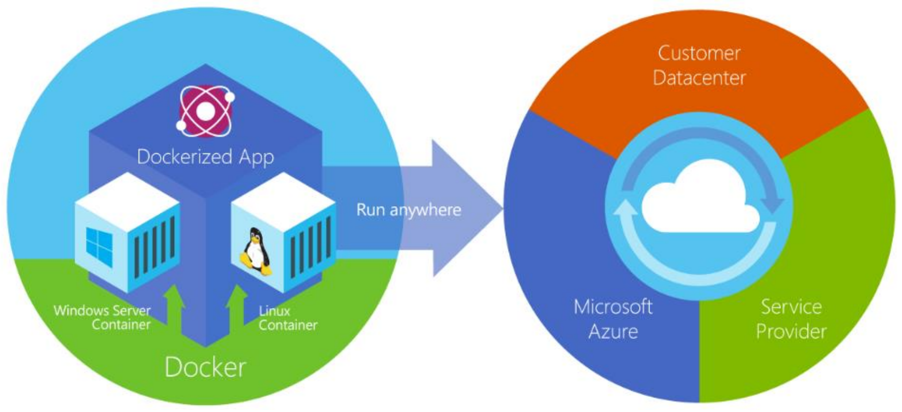


Figure 4-6. Docker deploys containers at all layers of the hybrid cloud

To someone familiar with virtual machines, containers might appear to be remarkably similar. A container runs an operating system, has a file system, and can be accessed over a network, just like a physical or virtual computer system. However, the technology and concepts behind containers are vastly different from virtual machines. From a developer point of view, a container must be treated more like a single process. In fact, a container has a single entry point for one process.

Docker containers (for simplicity, *containers*) can run natively on Linux and Windows. When running regular containers, Windows containers can run only on Windows hosts (a host server or a VM), and Linux containers can run only on Linux hosts. However, in recent versions of Windows Server and Hyper-V containers, a Linux container also can run natively on Windows Server by using the Hyper-V isolation technology that currently is available only in Windows Server Containers.

In the near future, mixed environments that have both Linux and Windows containers will be possible and even common.

Benefits of Windows Containers for your existing .NET applications

The benefits of using Windows Containers are fundamentally the same benefits you get from containers in general. Using Windows Containers is about greatly improving agility, portability, and control.

When we talk about existing .NET applications, we mostly mean traditional applications that were created by using the .NET Framework. For example, they might be traditional ASP.NET web applications—they don't use .NET Core, which is newer and runs cross-platform on Linux, Windows, and MacOS.

The main dependency in the .NET Framework is Windows. It also has secondary dependencies, like IIS, and System.Web in traditional ASP.NET.

A .NET Framework application must run on Windows, period. If you want to containerize existing .NET Framework applications and you can't or don't want to invest in a migration to .NET Core ("If it works properly, don't migrate it"), the only choice you have for containers is to use Windows Containers.

So, one of the main benefits of Windows Containers is that they offer you a way to modernize your existing .NET Framework applications that are running on Windows—through containerization. Ultimately, Windows Containers gets you the benefits that you are looking for by using containers—agility, portability, and better control.

Choose an OS to target with .NET-based containers

Given the diversity of operating systems that are supported by Docker, as well as the differences between .NET Framework and .NET Core, you should target a specific OS and specific versions based on the framework you are using.

For Windows, you can use Windows Server Core or Windows Nano Server. These Windows versions provide different characteristics (like IIS versus a self-hosted web server like Kestrel) that might be needed by .NET Framework or .NET Core applications.

For Linux, multiple distros are available and supported in official .NET Docker images (like Debian).

Figure 4-7 shows OS versions that you can target, depending on the app's version of the .NET Framework.

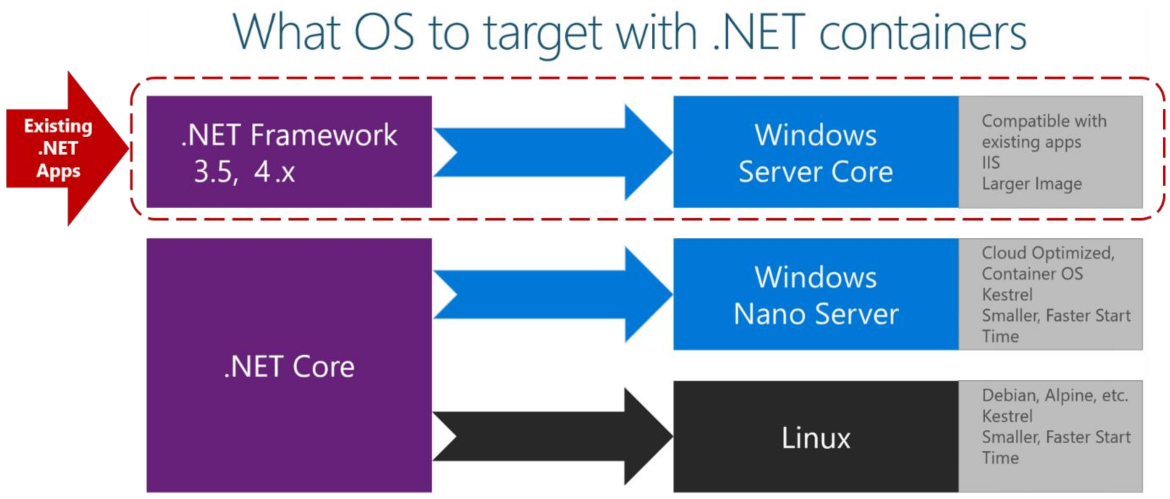

What OS to target with .NET containers
Existing .NET Apps
.NET Framework 3.5, 4.x
Windows Server Core
Compatible with existing apps
IIS
Larger Image
.NET Core
Windows Nano Server
Cloud Optimized, Container OS
Kestrel
Smaller, Faster Start Time
Linux
Debian, Alpine, etc.
Kestrel
Smaller, Faster Start Time

Figure 4-7. *Operating systems to target based on .NET Framework version*

In migration scenarios for existing or legacy applications that are based on .NET Framework applications, the main dependencies are on Windows and IIS. Your only option is to use Docker images based on Windows Server Core and the .NET Framework.

When you add the image name to your Dockerfile file, you can select the operating system and version by using a tag, as in the following examples for .NET Framework-based Windows container images:

Tag	System and version
microsoft/dotnet-framework:4.x-windowsservercore	.NET Framework 4.x on Windows Server Core
microsoft/aspnet:4.x-windowsservercore	.NET Framework 4.x with additional ASP.NET customization, on Windows Server Core

For .NET Core (cross-platform for Linux and Windows), the tags would look like this:

Tag	System and version
microsoft/dotnet:2.0.0-runtime	.NET Core 2.0 runtime-only on Linux

microsoft/dotnet:2.0.0-runtime-nanoserver	.NET Core 2.0 runtime-only on Windows Nano Server

Multi-arch images

Beginning in mid-2017, you also can use a new feature in Docker called multi-arch images. .NET Core Docker images can use multi-arch tags. Your Dockerfile files no longer need to define the operating system that you are targeting. The multi-arch feature allows a single tag to be used across multiple machine configurations. For instance, with multi-arch, you can use one common tag: **microsoft/dotnet:2.0.0-runtime**. If you pull that tag from a Linux container environment, you get the Debian-based image. If you pull that tag from a Windows container environment, you get the Nano Server-based image.

For .NET Framework images, because the traditional .NET Framework supports only Windows, you cannot use the multi-arch feature.

Windows container types

Like Linux containers, Windows Server containers are managed by using Docker Engine. Unlike Linux containers, Windows containers include two different container types, or run times—Windows Server containers and Hyper-V isolation.

Windows Server containers: Provides application isolation through process and namespace isolation technology. A Windows Server container shares a kernel with the container host and all containers that are running on the host. These containers do not provide a hostile security boundary and should not be used to isolate untrusted code. Because of the shared kernel space, these containers require the same kernel version and configuration.

Hyper-V isolation: Expands on the isolation provided by Windows Server Containers by running each container on a highly optimized VM. In this configuration, the kernel of the container host is not shared with other containers on the same host. These containers are designed for hostile multitenant hosting, with the same security assurances of a VM. Because these containers don't share the kernel with the host or other containers on the host, they can run kernels with different versions and configurations (with supported versions). For example, all Windows containers on Windows 10 use Hyper-V isolation to utilize the Windows Server kernel version and configuration.

Running a container on Windows with or without Hyper-V isolation is a run-time decision. You might choose to create the container with Hyper-V isolation initially, and at run time, choose to run it as a Windows Server container instead.

Additional resources

- **Windows Containers documentation**
  https://docs.microsoft.com/en-us/virtualization/windowscontainers/
- **Windows Containers fundamentals**
  https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/
- **Infographic: Microsoft and containers**
  https://info.microsoft.com/rs/157-GQE-382/images/Container%20infographic%201.4.17.pdf

When not to deploy to Windows Containers

Some Windows technologies are not supported by Windows Containers. In those cases, you still need to migrate to standards VMs, usually with just Windows and IIS.

Cases not supported in Windows Containers, as of mid-2017:

- Microsoft Message Queuing (MSMQ) currently is not supported in Windows Containers.
  - UserVoice request forum
  - Discussion forum
- Microsoft Distributed Transaction Coordinator (MSDTC) currently is not supported in Windows Containers
  - GitHub issue
- Microsoft Office currently does not support containers
  - UserVoice request forum

For additional not-supported scenarios and requests from the community, see the UserVoice forum for Windows Containers: https://windowsserver.uservoice.com/forums/304624-containers.

Additional resources

- **Virtual machines and containers in Azure**
  https://docs.microsoft.com/en-us/azure/virtual-machines/windows/containers

When to deploy Windows Containers in your on-premises IaaS VM infrastructure

Deploying Windows Containers in your on-premises infrastructure (VMs or bare-metal servers) is an important choice for several reasons:

- Your organization might not be ready to move to the cloud, or it might simply not be able to move to the cloud for a business reason. But, you can still get the benefits of using Windows Containers in your own datacenters.
- You might have other artifacts that are being used on-premises, and which might slow you down when you try to move to the cloud. For example, security or authentication dependencies with on-premises Windows Server Active Directory, or any other on-premises asset.
- If you start using Windows Containers today, you can make a phased migration to the cloud tomorrow from a much better position. Windows Containers is becoming a unit of deployment for any cloud, with no lock-in.

When to deploy Windows Containers to Azure VMs (IaaS cloud)

If your organization is using Azure VMs, even if you are also using Windows Containers, you are still dealing with IaaS. That means that dealing with infrastructure operations, VM OS patches, and infrastructure complexity for highly scalable applications when you need to deploy to multiple VMs in a load balanced infrastructure. The main scenarios for using Windows Containers in an Azure VM are:

- **Dev/test environment**: A VM in the cloud is perfect for development and testing in the cloud. You can rapidly create or stop the environment depending on your needs.
- **Small and medium scalability needs**: In scenarios where you might need just a couple of VMs for your production environment, managing a small number of VMs might be affordable until you can move to more advanced PaaS environments, like orchestrators.
- **Production environment with existing deployment tools**: You might be moving from an on-premises environment in which you have invested in tools to make complex deployments to VMs or bare-metal servers (like Puppet or similar tools). To move to the cloud with minimal changes to production environment deployment procedures, you might continue to use those tools to deploy to Azure VMs. However, you'll want to use Windows Containers as the unit of deployment to improve the deployment experience.

When to deploy Windows Containers to Service Fabric

Applications that are based on Windows Containers will quickly need to use platforms that move even further away from IaaS VMs. This is for improved automated scalability and high scalability, and to gain significant improvements in a complete management experience for deployments, upgrades, versioning, rollbacks, and health monitoring. You can achieve these goals with the orchestrator Azure Service Fabric, available in the Microsoft Azure cloud, but also on-premises, or even in another cloud.

Many organizations are lifting and shifting existing monolithic applications to containers for two reasons:

- Cost reductions, either due to consolidation and removal of existing hardware, or from running applications at a higher density.
- A consistent deployment contract between development and operations.

Pursuing cost reductions is understandable, and it's likely that all organizations are chasing that goal. Consistent deployment is harder to evaluate, but it's equally as important. A consistent deployment contract says that developers are free to choose to use the technology that suits them, and the operations team gets a single way to deploy and manage applications. This agreement alleviates the pain of having operations deal with the complexity of many different technologies, or forcing developers to work only with certain technologies. Essentially, each application is containerized in a self-contained deployment image.

Some organizations will continue modernizing by adding microservices (Cloud-Optimized and cloud-native applications). Many organizations will stop here (Cloud DevOps-Ready). As shown in Figure 4-8, these organizations won't move to microservices architectures because they might not need to. In any case, they already get the benefits that using containers plus Service Fabric provides—a complete management experience that includes deployment, upgrades, versioning, rollbacks, and health monitoring.

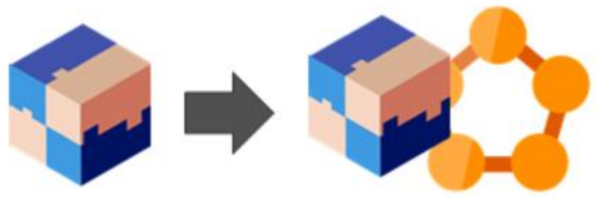


Figure 4-8. *Lift and shift an application to Service Fabric*

A key approach to Service Fabric is to reuse existing code and simply lift and shift. Therefore, you can migrate your current .NET Framework applications, by using Windows Containers, and deploy them to Service Fabric. It will be easier to keep going modernizing, eventually, by adding new microservices.

When comparing Service Fabric to other orchestrators, it's important to highlight that Service Fabric is very mature at running Windows-based applications and services. Service Fabric has been running Windows-based services and applications, including Tier-1, mission-critical products from Microsoft, for years. It was the first orchestrator to have general availability for Windows Containers (May 2017). Other containers, like Kubernetes, DC/OS, and Docker Swarm, are more mature in Linux, but less mature than Service Fabric for Windows-based applications and Windows Containers.

The ultimate goal of Service Fabric is to reduce the complexities of building applications by using a microservices approach. This is where you eventually want to be for certain types of applications, to avoid costly redesigns. You can start small, scale when needed, deprecate services, add new services, and evolve your application with customer use. We know that there are many other problems that are yet to be solved to make microservices more approachable for most developers. If you currently are just lifting and shifting an application with Windows Containers, but you are thinking about adding microservices based on containers in the future, that is the Service Fabric sweet spot.

When to deploy Windows Containers to Azure Container Service (i.e., Kubernetes)

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

If you are already working with open-source orchestrators like Kubernetes, Docker Swarm, or DC/OS, you don't need to change your existing management practices to move container workloads to the cloud. Use the application management tools that you're already familiar with, and connect via the standard API endpoints for the orchestrator of your choice.

All these orchestrators are mature environments if you are using Linux Docker containers, but they also support Windows Containers as of 2017 (some earlier, some more recently, depending on the orchestrator).

For example, in Kubernetes, support for containers is native (first-class citizen), so using Windows Containers on Kubernetes is also very effective and reliable (in preview until early fall 2017).

Build resilient services ready for the cloud: Embrace transient failures in the cloud

Resiliency is the ability to recover from failures and continue to function. Resiliency is not about avoiding failures, but accepting the fact that failures will occur, and then responding to them in a way that avoids downtime or data loss. The goal of resiliency is to return the application to a fully functioning state after a failure.

Your application is ready for the cloud when, at a minimum, it implements a software-based model of resiliency, rather than a hardware-based model. Your cloud application must embrace the partial failures that will certainly occur. You need to design or partially refactor your application if you want to achieve resiliency to expected partial failures. It should be designed to cope with partial failures, like transient network outages and nodes, or VMs crashing in the cloud. Even containers being moved to a different node within an orchestrator cluster can cause intermittent short failures within the application.

Handling partial failure

In a cloud-based application, there's an ever-present risk of partial failure. For instance, a single website instance or a container might fail, or it might be unavailable or unresponsive for a short time. Or, a single VM or server might crash.

Because clients and services are separate processes, a service might not be able to respond in a timely manner to a client’s request. The service might be overloaded and respond extremely slowly to requests, or it might simply not be accessible for a short time because of network issues.

For example, consider a monolithic .NET application that accesses a database in Azure SQL Database. If the Azure SQL database or any other third-party service is unresponsive for a brief time (an Azure SQL database might be moved to a different node or server, and be unresponsive for a few seconds), when the user tries to do any action, the application might crash and show an exception at the very same moment.

A similar scenario might occur in an app that consumes HTTP services. The network or the service itself might not be available in the cloud during a short, transient failure.

A resilient application like the one shown in Figure 4-9 should implement techniques like “retries with exponential backoff” to give the application an opportunity to handle transient failures in resources. You also should use “circuit breakers” in your applications. A circuit breaker stops an application from trying to access a resource when it's actually a long-term failure. By using a circuit breaker, the application avoids provoking a denial of service to itself.

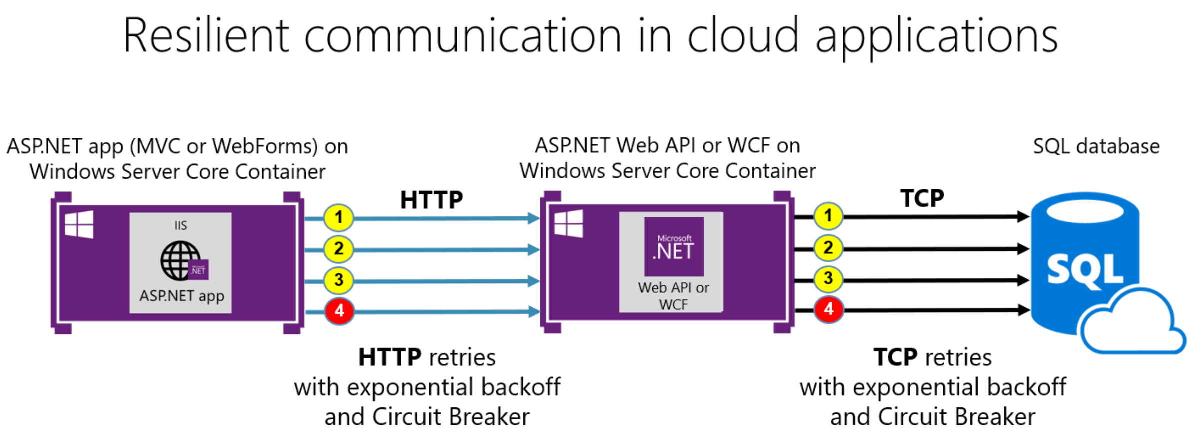

Resilient communication in cloud applications
ASP.NET app (MVC or WebForms) on Windows Server Core Container
IIS
ASP.NET app
HTTP
1
2
3
4
ASP.NET Web API or WCF on Windows Server Core Container
Microsoft .NET
Web API or WCF
TCP
SQL database
SQL
HTTP retries with exponential backoff and Circuit Breaker
TCP retries with exponential backoff and Circuit Breaker

Figure 4-9. *Partial failures handled by retries with exponential backoff*

You can use these techniques both in HTTP resources and in database resources. In Figure 4-9, the application is based on a 3-tier architecture, so you need these techniques at the services level (HTTP) and at the data tier level (TCP). In a monolithic application that uses only a single app tier in addition to the database (no additional services or microservices), handling transient failures at the database connection level might be enough. In that scenario, usually just a particular configuration of the database connection is required.

When implementing resilient communications that access the database, depending on the version of .NET you are using, it can be very straightforward (for example, with Entity Framework 6 or later, it's just a matter of configuring the database connection). Or, you might need to use additional libraries like the Transient Fault Handling Application Block (for earlier versions of .NET), or even implement your own library.

When implementing HTTP retries and circuit breakers, the recommendation for .NET is to use the Polly library, which targets .NET 4.0, .NET 4.5, and .NET Standard 1.1, which includes .NET Core support.

To learn how to implement strategies for handling partial failures in the cloud, see the following references.

Additional resources

- **Implementing resilient communication to handle partial failure**
  https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/implement-resilient-applications/partial-failure-strategies
- **Entity Framework connection resiliency and retry logic (version 6 and later)**
  https://msdn.microsoft.com/en-us/library/dn456835(v=vs.113).aspx
- **The Transient Fault Handling Application Block**
- https://msdn.microsoft.com/en-us/library/hh680934(v=pandp.50).aspx
- **Polly library for resilient HTTP communication**
  https://github.com/App-vNext/Polly

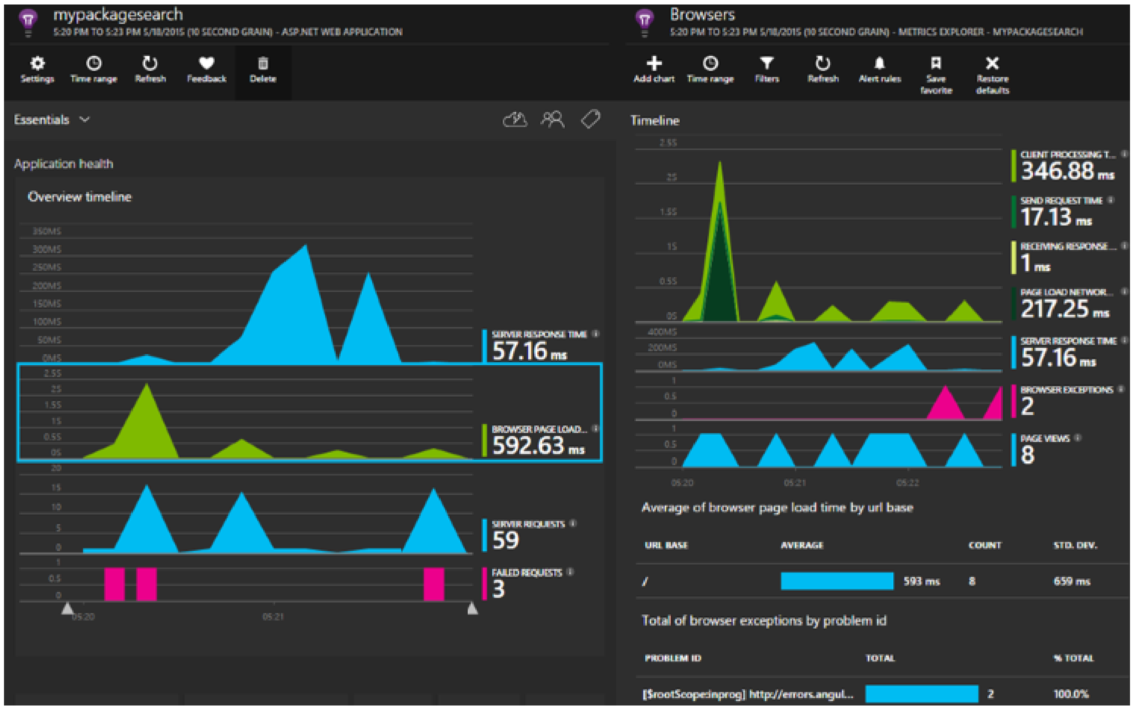
Modernize your apps with monitoring and telemetry

When you run an application in production, it's critical that you have insights about how your application is performing. Is it performing at a high level? Are users getting errors, or is the application stable and reliable? You need rich performance monitoring, powerful alerting, and dashboards to help ensure that your application is available and performing as expected. You also need to be able to see quickly if there's a problem, determine how many customers are affected, and perform a root-cause analysis to find and fix the issue.

Monitor your application with Application Insights

Application Insights is an extensible Application Performance Management (APM) service for web developers who work on multiple platforms. Use it to monitor your live web application. Application Insights automatically detects performance anomalies. It includes powerful analytics tools to help you diagnose issues, and to help you understand what users actually do with your app. Application Insights is designed to help you continuously improve performance and usability. It works for apps on a wide variety of platforms, including .NET, Node.js, and J2EE, whether hosted on-premises or in the cloud. Application Insights integrates with your DevOps processes, and has connection points to a variety of development tools.

Figure 4-10 shows an example of how Application Insights monitors your application and how it surfaces those insights to a dashboard.



Average of browser page load time by url base

URL BASE	AVERAGE	COUNT	STD. DEV.
/	593 ms	8	659 ms

Total of browser exceptions by problem id

PROBLEM ID	TOTAL	% TOTAL
[$rootScope:inprog] http://errors.angul...	2	100.0%

Figure 4-10. *Application Insights monitoring dashboard*

Monitor your Docker infrastructure with Log Analytics and its Container Monitoring solution

Azure Log Analytics is part of the Microsoft Azure overall monitoring solution. It's also a service in Operations Management Suite (OMS). Log Analytics monitors cloud and on-premises environments (OMS for on-premises) to help maintain availability and performance. It collects data generated by resources in your cloud and on-premises environments and from other monitoring tools to provide analysis across multiple sources.

In relation to Azure infrastructure logs, Log Analytics, as an Azure service, ingests log and metric data from other Azure services (via Azure Monitor), Azure VMs, Docker containers, and on-premises or other cloud infrastructures. Log Analytics offers flexible log search and out-of-the box analytics on top of this data. It provides rich tools that you can use to analyze data across sources, it allows complex queries across all logs, and it can proactively alert based on specified conditions. You can even collect custom data in the central Log Analytics repository, where you can query and visualize it. You also can take advantage of the Log Analytics built-in solutions to immediately gain insights into the security and functionality of your infrastructure.

You can access Log Analytics through the OMS portal or the Azure portal, which run in any browser, and provide you with access to configuration settings and multiple tools to analyze and act on collected data.

The Container Monitoring solution in Log Analytics helps you view and manage your Docker and Windows Container hosts in a single location. The solution shows which containers are running, what container image they're running, and where containers are running. You can view detailed audit information, including commands that are being used with containers. You also can troubleshoot containers by viewing and searching centralized logs, without needing to remotely view Docker or Windows hosts. You can find containers that might be noisy and consuming excess resources on a host. Additionally, you can view centralized CPU, memory, storage, and network usage, and performance information, for containers. On computers running Windows, you can centralize and compare logs from Windows Server, Hyper-V, and Docker containers. The solution supports the following container orchestrators:

- Docker Swarm
- DC/OS
- Kubernetes
- Service Fabric
- Red Hat OpenShift

Figure 4-11 shows the relationships between various container hosts and agents and OMS.

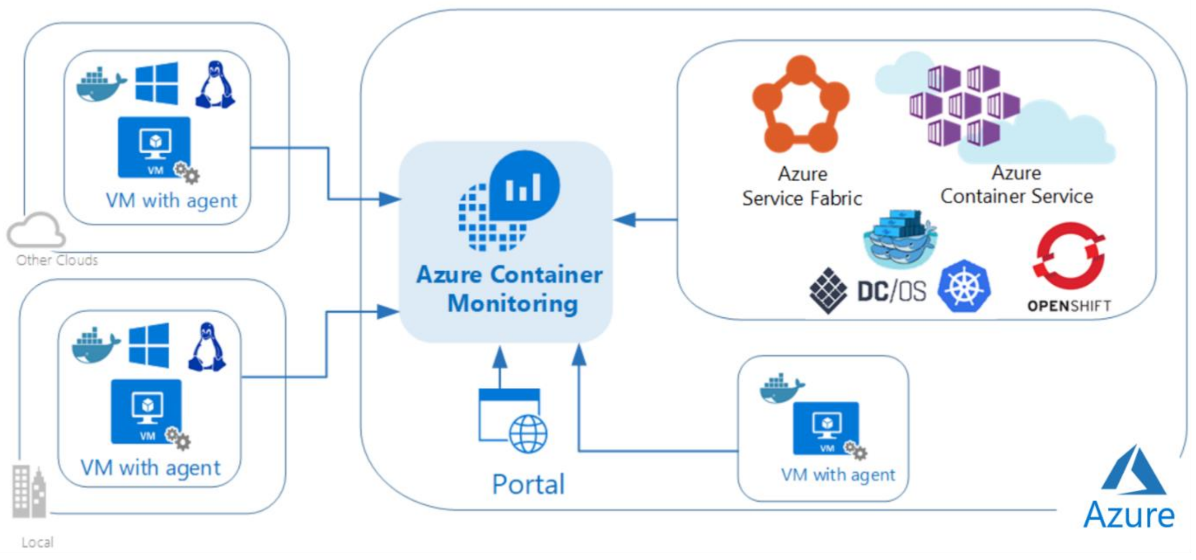


Figure 4-11. *Log Analytics Container Monitoring solution*

You can use the Log Analytics Container Monitoring solution to:

- 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.
- See an audit trail for actions on containers.
- Troubleshoot by viewing and searching centralized logs without remote login to the Docker hosts.
- Find containers that might be "noisy neighbors," and be consuming excess resources on a host.
- View centralized CPU, memory, storage, and network usage, and performance information, for containers.

Additional resources

- **Overview of monitoring in Microsoft Azure**
  https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview
- **What is Application Insights?**
  https://docs.microsoft.com/en-us/azure/application-insights/app-insights-overview
- **What is Log Analytics?**
  https://docs.microsoft.com/en-us/azure/log-analytics/log-analytics-overview
- **Container Monitoring solution in Log Analytics**
  https://docs.microsoft.com/en-us/azure/log-analytics/log-analytics-containers
- **Overview of Azure Monitor**
  https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview-azure-monitor
- **What is Operations Management Suite (OMS)?**
  https://docs.microsoft.com/en-us/azure/operations-management-suite/operations-management-suite-overview
- **Monitoring Windows Server containers in Service Fabric with OMS**
  https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-diagnostics-containers-windowsserver

Modernize your app's lifecycle with CI/CD pipelines and DevOps tools in the cloud

Today's businesses need to innovate at a rapid pace to be competitive in the marketplace. Delivering high-quality, modern applications requires DevOps tools and processes that are critical to implement this constant cycle of innovation. With the right DevOps tools, developers can streamline continuous deployment and get innovative applications into the hands of users more quickly.

Although continuous integration and deployment practices are well established, the introduction of containers introduces new considerations, particularly when you are working with multi-container applications.

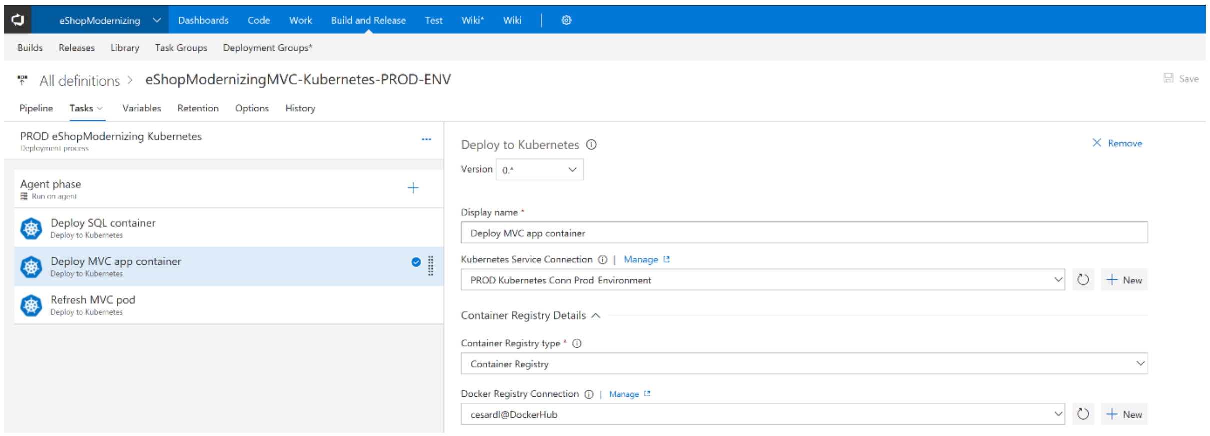
Visual Studio Team Services supports continuous integration and deployment of multi-container applications to a variety of environments through the official Team Services deployment tasks:

- Deploy to standalone Docker Host VM (Linux or Windows Server 2016 or later)
- Deploy to Service Fabric
- Deploy to Azure Container Service – Kubernetes

But you also can deploy to Docker Swarm or DC/OS by using Team Services script-based tasks.

To continue facilitating deployment agility, these tools provide excellent dev-to-test-to-production deployment experiences for container workloads, with a choice of development and CI/CD solutions.

Figure 4-12 shows a continuous deployment pipeline that deploys to a Kubernetes cluster in Azure Container Service.



All definitions > eShopModernizingMVC-Kubernetes-PROD-ENV

PROD eShopModernizing Kubernetes
Deployment process

Agent phase
Run on agent

Deploy SQL container
Deploy to Kubernetes

Deploy MVC app container
Deploy to Kubernetes

Refresh MVC pod
Deploy to Kubernetes

Deploy to Kubernetes

Version: 0.*

Display name *: Deploy MVC app container

Kubernetes Service Connection: PROD Kubernetes Conn Prod Environment

Container Registry Details

Container Registry type *: Container Registry

Docker Registry Connection: cesardl@DockerHub

Figure 4-12. *Visual Studio Team Services continuous deployment pipeline, deploying to a Kubernetes cluster*

Migrate to hybrid cloud scenarios

Some organizations and enterprises can't migrate some of their applications to public clouds like Microsoft Azure or any other public cloud due to regulations or their own policies. However, it's likely that any organization might benefit from having some of their applications in the public cloud and other applications on-premises. But a mixed environment can lead to excessive complexity in environments due to different platforms and technologies used in public clouds versus on-premises environments.

Microsoft provides the best hybrid cloud solution, one in which you can optimize your existing assets on-premises and in the public cloud, while you ensure consistency in an Azure hybrid cloud. You can maximize existing skills, and get a flexible, unified approach to building apps that can run in the cloud or on-premises, thanks to Azure Stack (on-premises) and Azure (public cloud).

When it comes to security, you can centralize management and security across your hybrid cloud. You can get control over all assets, from your datacenter to the cloud, by providing single sign-on to on-premises and cloud apps. You accomplish this by extending Active Directory to a hybrid cloud, and by using identity management.

Finally, you can distribute and analyze data seamlessly, use the same query languages for cloud and on-premises assets, and apply analytics and deep learning in Azure to enrich your data, regardless of its source.

Azure Stack

Azure Stack is a hybrid cloud platform that lets you deliver Azure services from your organization's datacenter. Azure Stack is designed to support new options for your modern applications in key scenarios, like edge and unconnected environments, or meeting specific security and compliance requirements.

Figure 4-13 shows an overview of the true hybrid cloud platform that Microsoft offers.

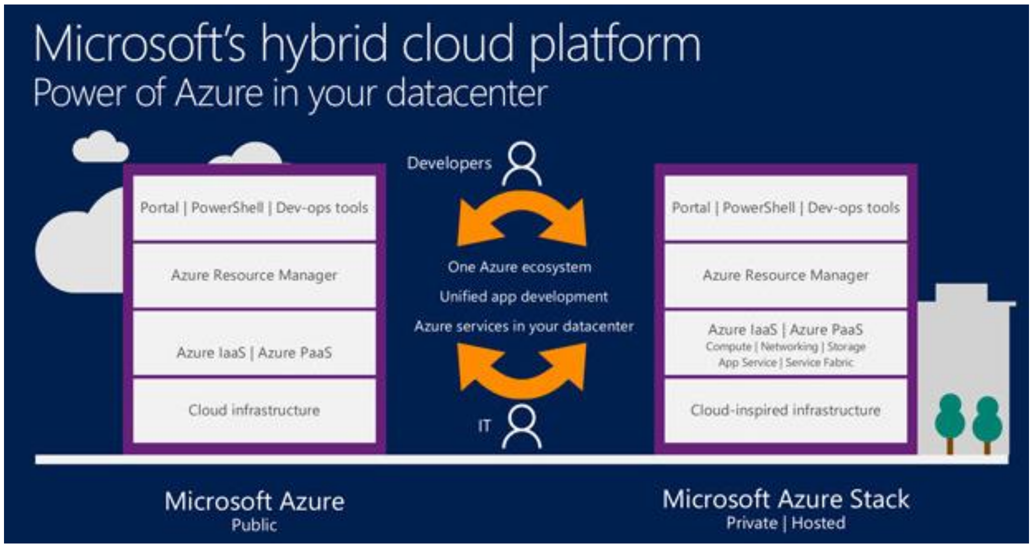

Microsoft's hybrid cloud platform
Power of Azure in your datacenter
Developers
Portal | PowerShell | Dev-ops tools
Azure Resource Manager
Azure IaaS | Azure PaaS
Cloud infrastructure
One Azure ecosystem
Unified app development
Azure services in your datacenter
IT
Portal | PowerShell | Dev-ops tools
Azure Resource Manager
Azure IaaS | Azure PaaS
Compute | Networking | Storage
App Service | Service Fabric
Cloud-inspired infrastructure
Microsoft Azure
Public
Microsoft Azure Stack
Private | Hosted

Figure 4-13. *Microsoft hybrid cloud platform with Azure Stack and Azure*

Azure Stack is offered in two deployment options, to meet your needs:

- Azure Stack integrated systems
- Azure Stack Development Kit

Azure Stack integrated systems

Azure Stack integrated systems are offered through a partnership of Microsoft and hardware partners. The partnership creates a solution that offers cloud-paced innovation that is balanced with simplicity in management. Because Azure Stack is offered as an integrated system of hardware and software, you get the right amount of flexibility and control, while still adopting innovation from the cloud. Azure Stack integrated systems range in size from 4 to 12 nodes, and are jointly supported by the hardware partner and Microsoft. Use Azure Stack integrated systems to implement new scenarios for your production workloads.

Azure Stack Development Kit

Microsoft Azure Stack Development Kit is a single-node deployment of Azure Stack, which you can use to evaluate and learn about Azure Stack. You can also use Azure Stack Development Kit as a developer environment, where you can develop using APIs and tooling that are consistent with Azure. Azure Stack Development Kit is not intended to be used as a production environment.

Additional resources

- **Azure hybrid cloud**
  https://www.microsoft.com/en-us/cloud-platform/hybrid-cloud
- **Azure Stack**
  https://azure.microsoft.com/en-us/overview/azure-stack/
- **Active Directory Service Accounts for Windows Containers**
  https://docs.microsoft.com/en-us/virtualization/windowscontainers/manage-containers/manage-serviceaccounts
- **Create a container with Active Directory support**
  https://blogs.msdn.microsoft.com/containerstuff/2017/01/30/create-a-container-with-active-directory-support/
- **Azure Hybrid Benefit licensing**
  https://azure.microsoft.com/en-us/pricing/hybrid-use-benefit/

Walkthroughs and technical get-started overview

To limit the size of this eBook, we've made additional technical documentation and the full walkthroughs available in a GitHub repo. The online series of walkthroughs that is described in this chapter covers the step-by-step setup of the multiple environments that are based on Windows Containers, and deployment to Azure.

The following sections explain what each walkthrough is about—its objectives, its high-level vision—and provides a diagram of the tasks that are involved. You can get the walkthroughs themselves in the *eShopModernizing* apps GitHub repo wiki at https://github.com/dotnet-architecture/eShopModernizing/wiki.

Technical walkthrough list

The following get-started walkthroughs provide consistent and comprehensive technical guidance for sample apps that you can lift and shift by using containers, and then move by using multiple deployment choices in Azure.

Each of the following walkthroughs uses the new sample eShopLegacy and eShopModernizing apps, which are available on GitHub at https://github.com/dotnet-architecture/eShopModernizing.

- **Tour of eShop legacy apps**
- **Containerize your existing .NET applications with Windows Containers**
- **Deploy your Windows Containers-based app to Azure VMs**
- **Deploy your Windows Containers-based apps to Kubernetes in Azure Container Service**
- **Deploy your Windows Containers-based apps to Azure Service Fabric**

Walkthrough 1: Tour of eShop legacy apps

Technical walkthrough availability

The full technical walkthrough is available in the eShopModernizing GitHub repo wiki:

https://github.com/dotnet-architecture/eShopModernizing/wiki/01.-Tour-on-eShopModernizing-apps-implementation-code

Overview

In this walkthrough, you can explore the initial implementation of two sample legacy applications. Both sample apps have a monolithic architecture, and were created by using classic ASP.NET. One application is based on ASP.NET 4.x MVC; the second application is based on ASP.NET 4.x Web Forms. Both applications are in the eShopModernizing GitHub repo.

You can containerize both sample apps, similar to the way you can containerize a classic Windows Communication Foundation (WCF) application to be consumed as a desktop application. For an example, see eShopModernizingWCFWinForms.

Goals

The main goal of this walkthrough is simply to get familiar with these apps, and with their code and configuration. You can configure the apps so that they generate and use mock data, without using the SQL database, for testing purposes. This optional config is based on dependency injection, in a decoupled way.

Scenario

Figure 5-1 shows the simple scenario of the original legacy applications.

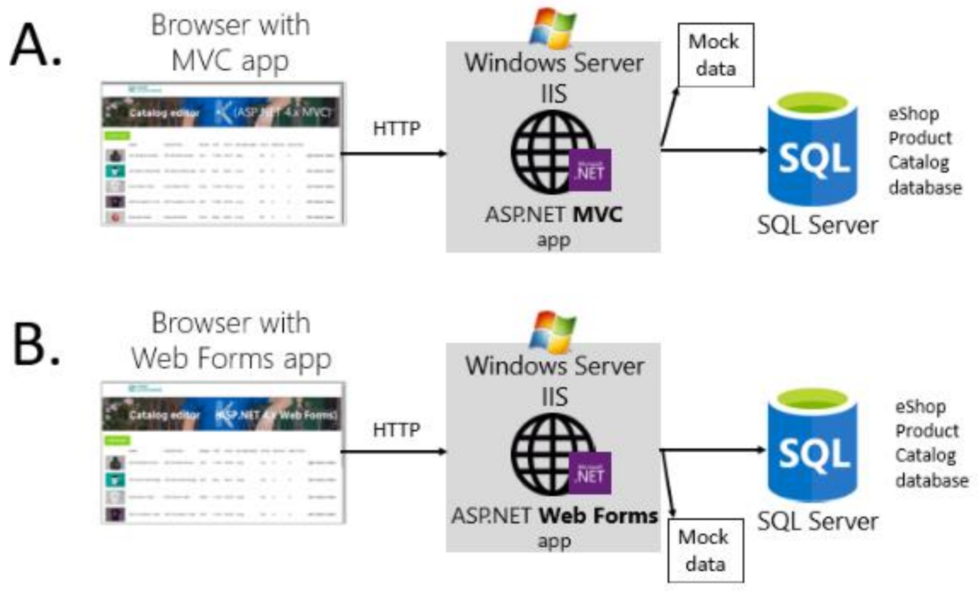
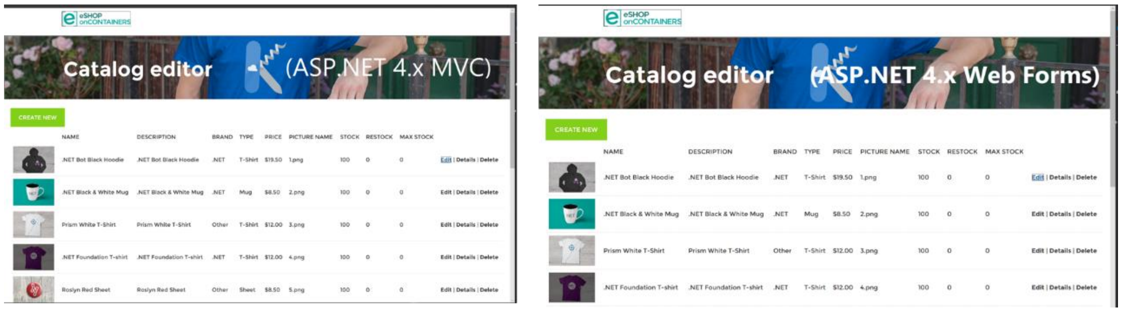


Figure 5-1. *Simple architecture scenario of the original legacy applications*

From a business domain perspective, both apps offer the same catalog management features. Members of the eShop enterprise team would use the app to view and edit the product catalog. Figure 5-2 shows the initial app screenshots.



	NAME	DESCRIPTION	BRAND	TYPE	PRICE	PICTURE NAME	STOCK	RESTOCK	MAX STOCK	
	.NET Bot Black Hoodie	.NET Bot Black Hoodie	.NET	T-Shirt	$19.50	1.png	100	0	0	Edit | Details | Delete
	.NET Black & White Mug	.NET Black & White Mug	.NET	Mug	$8.50	2.png	100	0	0	Edit | Details | Delete
	Prism White T-Shirt	Prism White T-Shirt	Other	T-Shirt	$12.00	3.png	100	0	0	Edit | Details | Delete
	.NET Foundation T-shirt	.NET Foundation T-shirt	.NET	T-Shirt	$12.00	4.png	100	0	0	Edit | Details | Delete
	Roslyn Red Sheet	Roslyn Red Sheet	Other	Sheet	$8.50	5.png	100	0	0	Edit | Details | Delete

	NAME	DESCRIPTION	BRAND	TYPE	PRICE	PICTURE NAME	STOCK	RESTOCK	MAX STOCK	
	.NET Bot Black Hoodie	.NET Bot Black Hoodie	.NET	T-Shirt	$19.50	1.png	100	0	0	Edit | Details | Delete
	.NET Black & White Mug	.NET Black & White Mug	.NET	Mug	$8.50	2.png	100	0	0	Edit | Details | Delete
	Prism White T-Shirt	Prism White T-Shirt	Other	T-Shirt	$12.00	3.png	100	0	0	Edit | Details | Delete
	.NET Foundation T-shirt	.NET Foundation T-shirt	.NET	T-Shirt	$12.00	4.png	100	0	0	Edit | Details | Delete

Figure 5-2. *ASP.NET MVC and ASP.NET Web Forms applications (existing/legacy technologies)*

These are web applications that are used to browse and modify catalog entries. The fact that both apps deliver the same business/functional features is simply for comparison purposes. You can see a similar modernization process for apps that were created by using the ASP.NET MVC and ASP.NET Web Forms frameworks.

Dependencies in ASP.NET 4.x or earlier versions (either for MVC or for Web Forms) means that these applications won't run on .NET Core unless the code is fully rewritten by using ASP.NET Core MVC. This demonstrates the point that if you don't want to re-architect or rewrite code, you can containerize existing applications, and still use the same .NET technologies and the same code. You can see how you can run applications like these in containers, without any changes to legacy code.

Benefits

The benefits of this walkthrough are simple: Just get familiar with the code and application configuration, based on dependency injection. Then, you can experiment with this approach when you containerize and deploy to multiple environments in the future.

Next steps

Explore this content more in-depth on the GitHub wiki:

https://github.com/dotnet-architecture/eShopModernizing/wiki/01.-Tour-on-eShopModernizing-apps-implementation-code

Walkthrough 2: Containerize your existing .NET applications with Windows Containers

Technical walkthrough availability

The full technical walkthrough is available in the eShopModernizing GitHub repo wiki:

https://github.com/dotnet-architecture/eShopModernizing/wiki/02.-How-to-containerized-the-.NET-Framework-web-apps-with-Windows-Containers-and-Docker

Overview

Use Windows Containers to improve deployment of existing .NET applications, like those based on MVC, Web Forms, or WCF, to production, development, and test environments.

Goals

The goal of this walkthrough is to show you several options for containerizing an existing .NET Framework application. You can:

- Containerize your application by using Visual Studio 2017 Tools for Docker (Visual Studio 2017 or later versions).
- Containerize your application by manually adding a Dockerfile, and then using the Docker CLI.
- Containerize your application by using the Img2Docker tool (an open-source tool from Docker).

This walkthrough focuses on the Visual Studio 2017 Tools for Docker approach, but the other two approaches are fairly similar in regards to using Dockerfiles.

Scenario

Figure 5-3 shows the scenario for containerized eShop legacy applications.

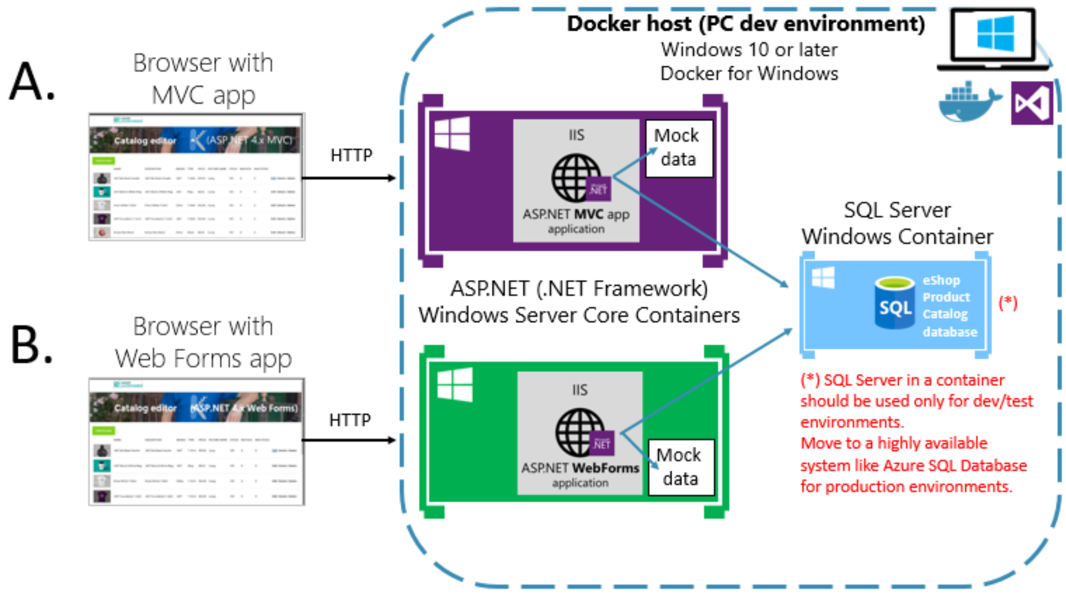

A.
Browser with MVC app
Catalog editor
(ASP.NET 4.x MVC)
HTTP
B.
Browser with Web Forms app
Catalog editor
ASP.NET 4.x Web Forms
HTTP
Docker host (PC dev environment)
Windows 10 or later
Docker for Windows
IIS
Mock data
ASP.NET MVC app application
ASP.NET (.NET Framework) Windows Server Core Containers
IIS
ASP.NET WebForms application
Mock data
SQL Server Windows Container
SQL
eShop Product Catalog database
(*)
(*) SQL Server in a container should be used only for dev/test environments.
Move to a highly available system like Azure SQL Database for production environments.

Figure 5-3. *Simplified architecture diagram of containerized applications in a development environment*

Benefits

There are advantages to running your monolithic application in a container. First, you create an image for the application. From that point on, every deployment runs in the same environment. Every container uses the same OS version, has the same version of dependencies installed, uses the same .NET framework version, and is built by using the same process. Basically, you control the dependencies of your application by using a Docker image. The dependencies travel with the application when you deploy the containers.

An additional benefit is that developers can run the application in the consistent environment that's provided by Windows Containers. Issues that appear only with certain versions can be spotted immediately, instead of surfacing in a staging or production environment. Differences in development environments used by members of the development team matter less when applications run in containers.

Containerized applications also have a flatter scale-out curve. Containerized apps enable you to have more application and service instances (based on containers) in a VM or physical machine compared to regular application deployments per machine. This translates to higher density and fewer required resources, especially when you use orchestrators like Kubernetes or Service Fabric.

Containerization, in ideal situations, does not require making any changes to the application code (C#). In most scenarios, you just need the Docker deployment metadata files (Dockerfiles and Docker Compose files).

Next steps

Explore this content more in-depth on the GitHub wiki:

https://https://github.com/dotnet-architecture/eShopModernizing/wiki/02.-How-to-containerized-the-.NET-Framework-web-apps-with-Windows-Containers-and-Docker

Walkthrough 3: Deploy your Windows Containers-based app to Azure VMs

Technical walkthrough availability

The full technical walkthrough is available in the eShopModernizing GitHub repo wiki:

https://github.com/dotnet-architecture/eShopModernizing/wiki/03.-How-to-deploy-your-Windows-Containers-based-app-into-Azure-VMs-(Including-CI-CD)

Overview

Deploying to a Docker host on a Windows Server 2016 VM in Azure lets you quickly set up dev/test/staging environments. It also gives you a common place for testers or business users to validate the app. VMs also can be valid IaaS production environments.

Goals

The goal of this walkthrough is to show you the multiple alternatives you have when you deploy Windows Containers to Azure VMs that are based on Windows Server 2016 or later versions.

Scenarios

Several scenarios are covered in this walkthrough.

Scenario A: Deploy to an Azure VM from a dev PC through Docker Engine connection

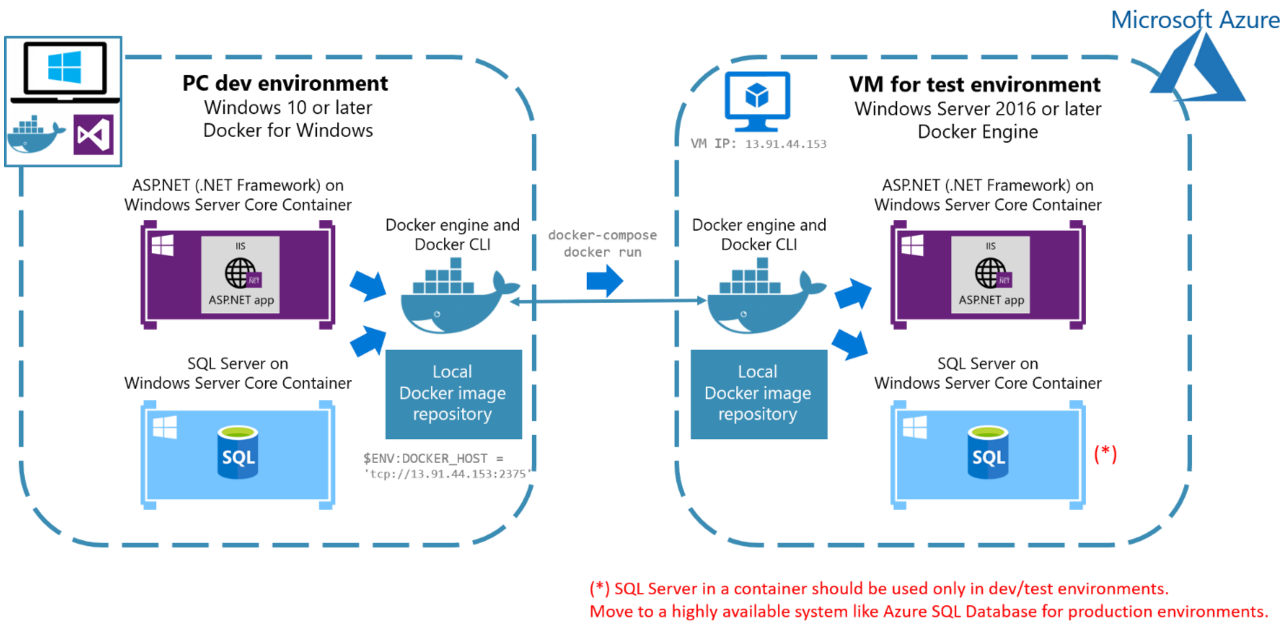

Microsoft Azure
PC dev environment
Windows 10 or later
Docker for Windows
ASP.NET (.NET Framework) on Windows Server Core Container
IIS
ASP.NET app
SQL Server on Windows Server Core Container
SQL
Docker engine and Docker CLI
Local Docker image repository
$ENV:DOCKER_HOST = 'tcp://13.91.44.153:2375'
docker-compose
docker run
VM for test environment
Windows Server 2016 or later
Docker Engine
VM IP: 13.91.44.153
Docker engine and Docker CLI
Local Docker image repository
ASP.NET (.NET Framework) on Windows Server Core Container
IIS
ASP.NET app
SQL Server on Windows Server Core Container
SQL
(*)
(*) SQL Server in a container should be used only in dev/test environments.
Move to a highly available system like Azure SQL Database for production environments.

Figure 5-4. *Deploy to an Azure VM from a dev PC through a Docker Engine connection*

Scenario B: Deploy to an Azure VM through a Docker Registry

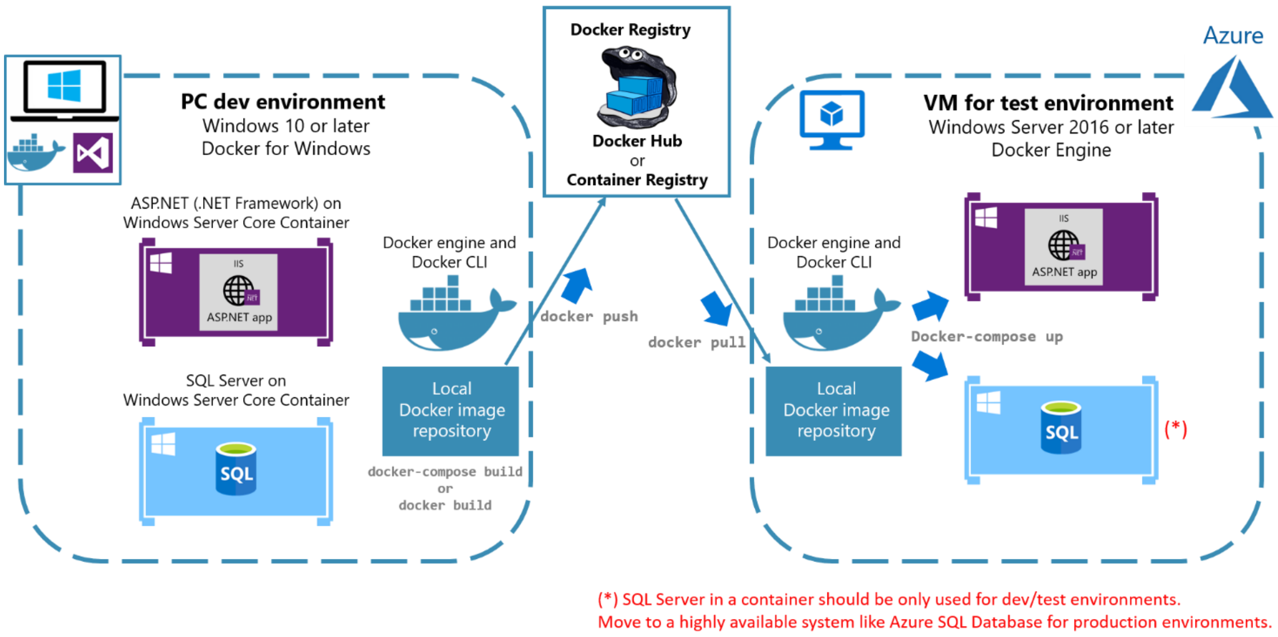


Figure 5-5. *Deploy to an Azure VM through a Docker Registry*

Scenario C: Deploy to an Azure VM from CI/CD pipelines in Visual Studio Team Services

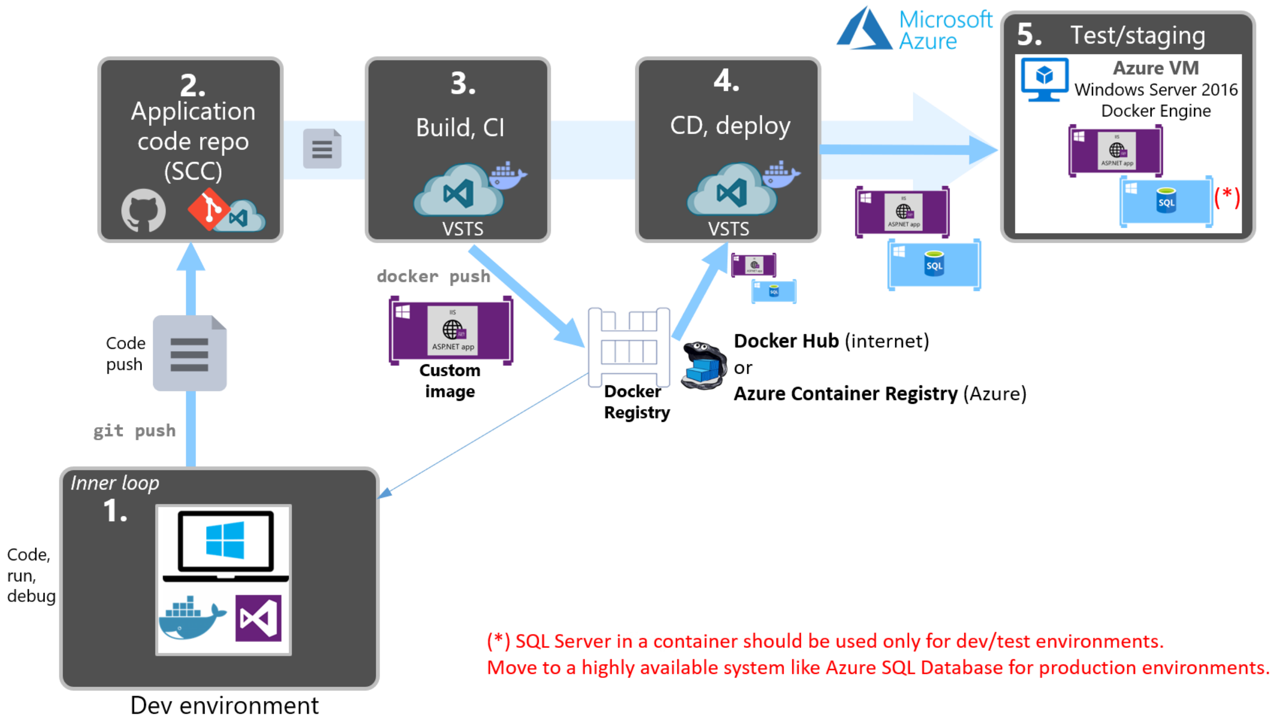


Figure 5-6. *Deploy to an Azure VM from CI/CD pipelines in Visual Studio Team Services*

Azure VMs for Windows Containers

Azure VMs for Windows Containers are simply VMs that are based on Windows Server 2016, Windows 10, or later versions, both with Docker Engine set up. In most cases, you will use Windows Server 2016 in the Azure VMs.

Azure currently provides a VM named **Windows Server 2016 with Containers**. You can use this VM to try the new Windows Server Container feature, with either Windows Server Core or Windows Nano Server. Container OS images are installed, and then the VM is ready to use with Docker.

Benefits

Although Windows Containers can be deployed to on-premises Windows Server 2016 VMs, when you deploy to Azure, you get an easier way to get started, with ready-to-use Windows Server Container VMs. You also get a common online location that's accessible to testers, and automatic scalability through Azure VM scale sets.

Next steps

Explore this content more in-depth on the GitHub wiki:

https://github.com/dotnet-architecture/eShopModernizing/wiki/03.-How-to-deploy-your-Windows-Containers-based-app-into-Azure-VMs-(Including-CI-CD)

Walkthrough 4: Deploy your Windows Containers-based apps to Kubernetes in Azure Container Service

Technical walkthrough availability

The full technical walkthrough is available in the eShopModernizing GitHub repo wiki:

https://github.com/dotnet-architecture/eShopModernizing/wiki/04.-How-to-deploy-your-Windows-Containers-based-apps-into-Kubernetes-in-Azure-Container-Service-(Including-C-CD)

Overview

An application that's based on Windows Containers will quickly need to use platforms, moving even further away from IaaS VMs. This is needed to easily achieve high scalability and better automated scalability, and for a significant improvement in automated deployments and versioning. You can achieve these goals by using the orchestrator Kubernetes, available in Azure Container Services.

Goals

The goal of this walkthrough is to learn how to deploy a Windows Container–based application to Kubernetes (also called *K8s*) in Azure Container Service. Deploying to Kubernetes from scratch is a two-step process:

1. Deploy a Kubernetes cluster to Azure Container Service.
2. Deploy the application and related resources to the Kubernetes cluster.

Scenarios

Scenario A: Deploy directly to a Kubernetes cluster from a dev environment

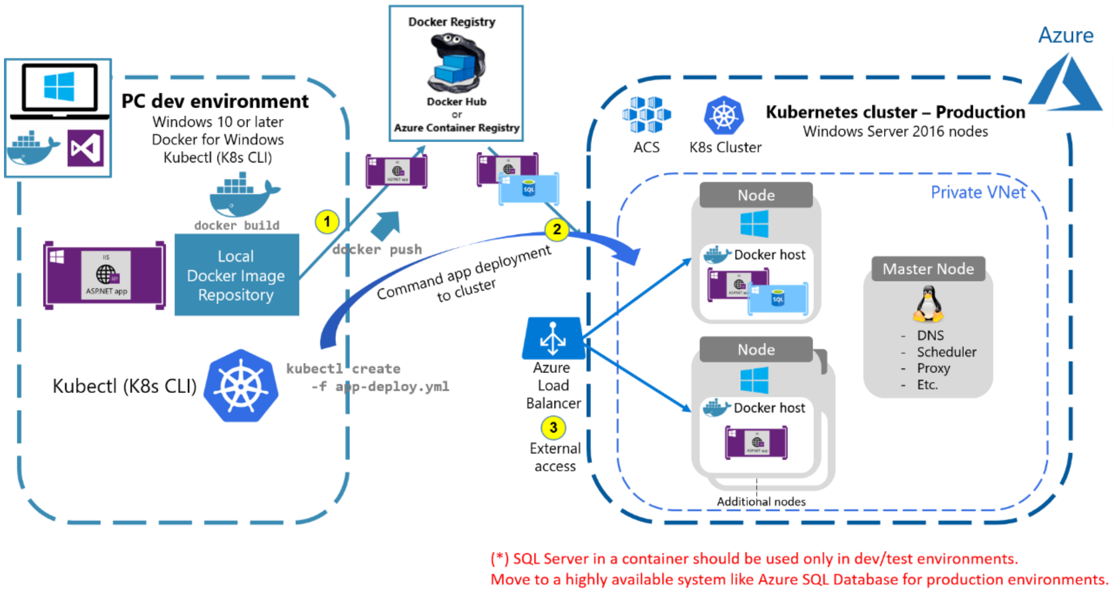
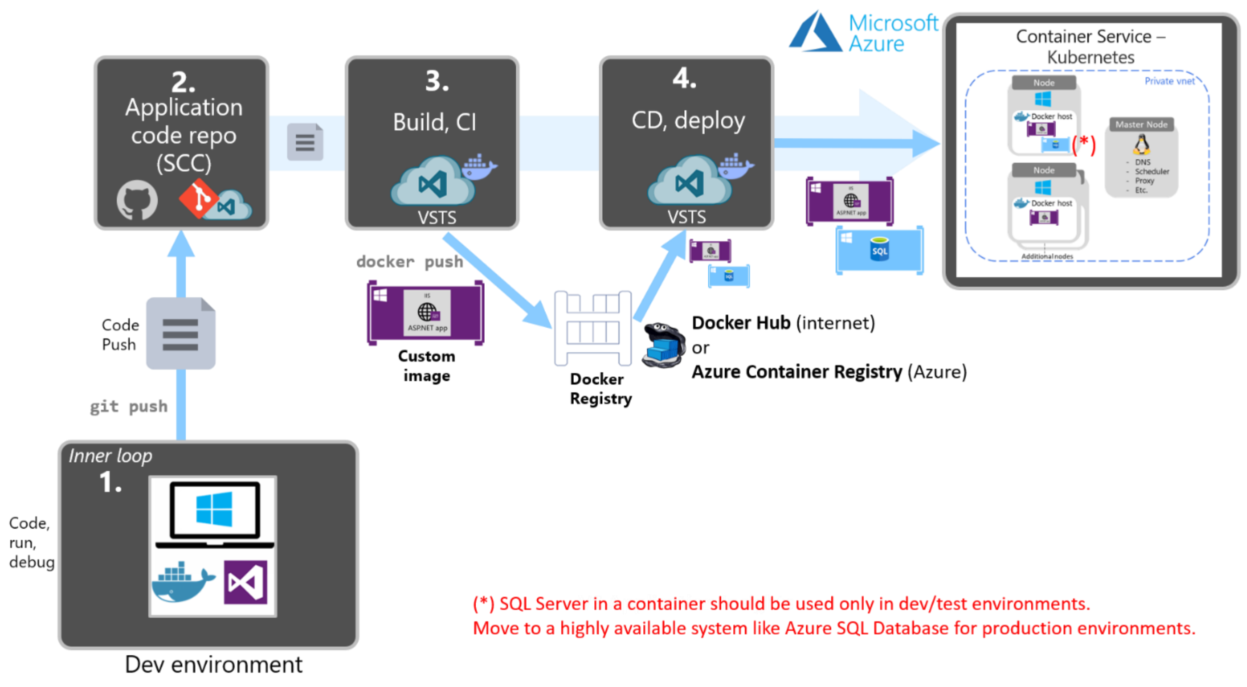


Figure 5-7. *Deploy directly to a Kubernetes cluster from a development environment*

Scenario B: Deploy to a Kubernetes cluster from CI/CD pipelines in Team Services



Microsoft Azure
Container Service – Kubernetes
Private vnet
Node
Docker host
Master Node
DNS
Scheduler
Proxy
Etc.
(*)
Additional nodes
2. Application code repo (SCC)
3. Build, CI
VSTS
4. CD, deploy
VSTS
docker push
Custom image
Docker Registry
Docker Hub (internet)
or
Azure Container Registry (Azure)
Code Push
git push
Inner loop
1.
Code, run, debug
Dev environment
(*) SQL Server in a container should be used only in dev/test environments.
Move to a highly available system like Azure SQL Database for production environments.

Figure 5-8. *Deploy to a Kubernetes cluster from CI/CD pipelines in Team Services*

Benefits

There are many benefits to deploying to a cluster in Kubernetes. The biggest benefit is that you get a production-ready environment in which you can scale-out the application based on the number of container instances you want to use (inner-scalability in the existing nodes), and based on the number of nodes or VMs in the cluster (global scalability of the cluster).

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

With Kubernetes, developers can progress from thinking about physical and virtual machines, to planning a container-centric infrastructure that facilitates the following capabilities, among others:

- Applications based on multiple containers
- Replicating container instances and horizontal autoscaling
- Naming and discovering (for example, internal DNS)
- Balancing loads
- Rolling updates
- Distributing secrets
- Application health checks

Next steps

Explore this content more in-depth on the GitHub wiki: https://github.com/dotnet-architecture/eShopModernizing/wiki/04.-How-to-deploy-your-Windows-Containers-based-apps-into-Kubernetes-in-Azure-Container-Service-(Including-C-CD)

Walkthrough 5: Deploy your Windows Containers-based apps to Azure Service Fabric

Technical walkthrough availability

The full technical walkthrough is available in the eShopModernizing GitHub repo wiki:

https://github.com/dotnet-architecture/eShopModernizing/wiki/05.-How-to-deploy-your-Windows-Containers-based-apps-into-Azure-Service-Fabric-(Including-CI-CD)

Overview

An application that's based on Windows Containers will quickly need to use platforms, moving even further away from IaaS VMs. This is needed to easily achieve high scalability and better automated scalability, and for a significant improvement in automated deployments and versioning. You can achieve these goals by using the orchestrator Azure Service Fabric, which is available in the Azure cloud, but also available to use on-premises, or even in a different public cloud.

Goals

The goal of this walkthrough is to learn how to deploy a Windows Container–based application to a Service Fabric cluster in Azure. Deploying to Service Fabric from scratch is a two-step process:

1. Deploy a Service Fabric cluster to Azure (or to a different environment).
2. Deploy the application and related resources to the Service Fabric cluster.

Scenarios

Scenario A: Deploy directly to a Service Fabric cluster from a dev environment

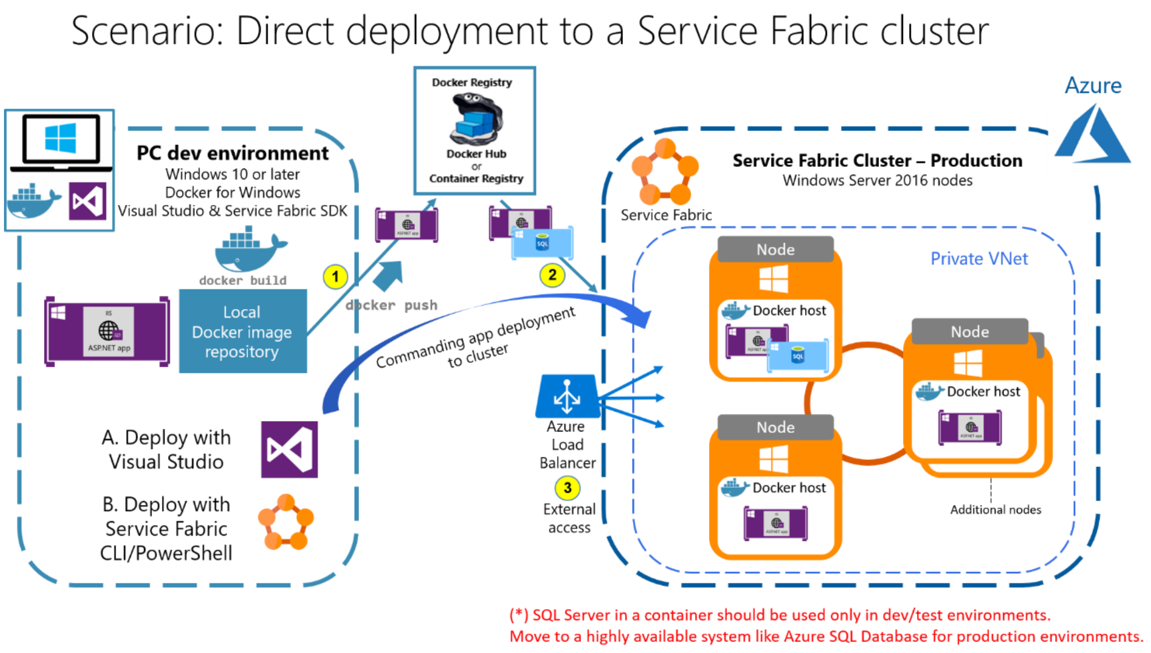

Scenario: Direct deployment to a Service Fabric cluster
PC dev environment
Windows 10 or later
Docker for Windows
Visual Studio & Service Fabric SDK
docker build
Local Docker image repository
A. Deploy with Visual Studio
B. Deploy with Service Fabric CLI/PowerShell
Docker Registry
Docker Hub or Container Registry
1
docker push
2
Commanding app deployment to cluster
Azure
Service Fabric Cluster – Production
Windows Server 2016 nodes
Service Fabric
Private VNet
Node
Docker host
Azure Load Balancer
3
External access
Additional nodes
(*) SQL Server in a container should be used only in dev/test environments.
Move to a highly available system like Azure SQL Database for production environments.

Figure 5-9. *Deploy directly to a Service Fabric cluster from a development environment*

Scenario B: Deploy to a Service Fabric cluster from CI/CD pipelines in Team Services

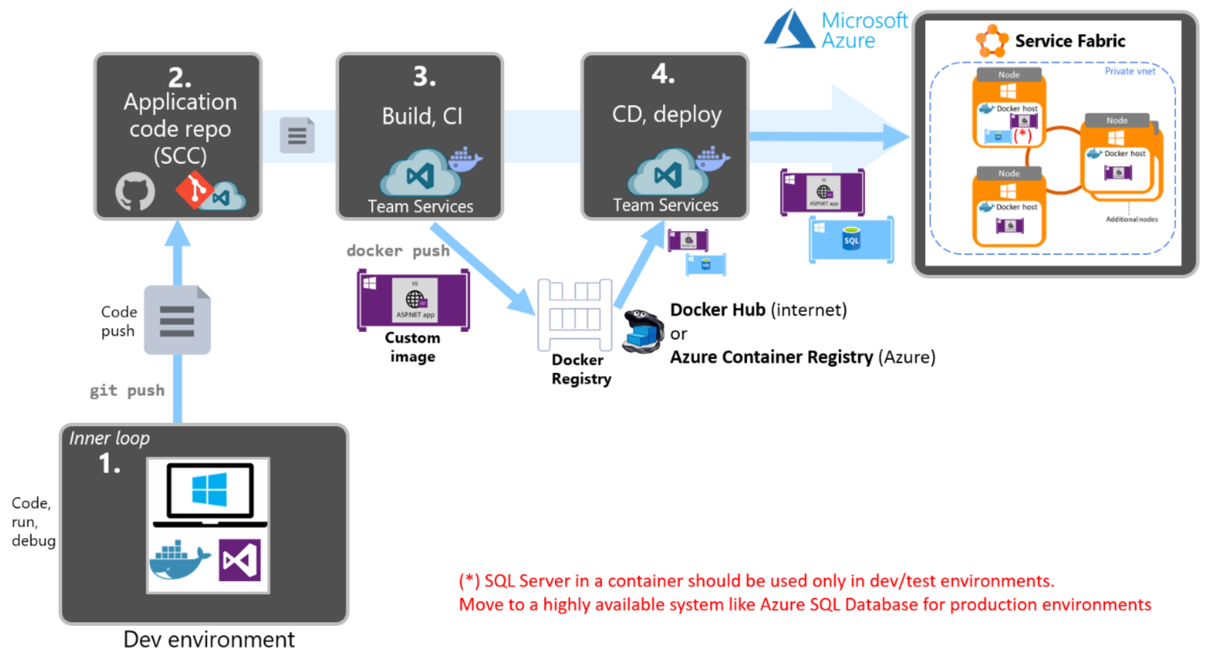

Microsoft Azure
Service Fabric
Private vnet
Node
Docker host
(*)
Additional nodes
2. Application code repo (SCC)
3. Build, CI
Team Services
4. CD, deploy
Team Services
docker push
Custom image
Docker Registry
Docker Hub (internet) or Azure Container Registry (Azure)
Code push
git push
Inner loop
1.
Code, run, debug
Dev environment
(*) SQL Server in a container should be used only in dev/test environments.
Move to a highly available system like Azure SQL Database for production environments

Figure 5-10. *Deploy to a Service Fabric cluster from CI/CD pipelines in Visual Studio Team Services*

Benefits

The benefits of deploying to a cluster in Service Fabric are similar to the benefits of using Kubernetes. One difference, though, is that Service Fabric is a very mature production environment for Windows applications compared to Kubernetes, which was in preview for Windows Containers until early fall of 2017. (Kubernetes is a more mature environment for Linux).

The main benefit of using Azure Service Fabric is that you get a production-ready environment in which you can scale-out the application based on the number of container instances you want to use (inner-scalability in the existing nodes), and based on the number of nodes or VMs in the cluster (global scalability of the cluster).

Azure Service Fabric offers portability both for your containers and for your application configuration. You can have a Service Fabric cluster in Azure, or install it on-premises in your own datacenter. You can even install a Service Fabric cluster in a different cloud, like Amazon AWS.

With Service Fabric, developers can progress from thinking about physical and virtual machines to planning a container-centric infrastructure that facilitates the following capabilities, among others:

- Applications based on multiple containers.
- Replicating container instances and horizontal autoscaling.
- Naming and discovering (for example, internal DNS).
- Balancing loads.
- Rolling updates.
- Distributing secrets.
- Application health checks.

The following capabilities are exclusive in Service Fabric (compared to other orchestrators):

- Stateful services capability, through the Reliable Services application model.
- Actors pattern, through the Reliable Actors application model.
- Deploy bare-bone processes, in addition to Windows or Linux containers.
- Advanced rolling updates and health checks.

Next steps

Explore this content more in-depth on the GitHub wiki:

https://github.com/dotnet-architecture/eShopModernizing/wiki/05.-How-to-deploy-your-Windows-Containers-based-apps-into-Azure-Service-Fabric-(Including-CI-CD)

Conclusions

Key takeaways

- Container-based solutions ultimately provide cost savings benefits. Containers are a solution to deployment problems because they remove the friction caused by lack of dependencies in production environments. Removing those issues, it improves Dev/Test, DevOps and production operations significantly.
- Docker is becoming the de facto standard in the container industry. Docker is supported by the most prominent vendors in the Linux and Windows ecosystems, including Microsoft. In the future, provides the most comprehensive and complete environment to modernize your existing .NET Framework applications with Windows Containers and Azure infrastructure services A Docker container is becoming the standard unit of deployment for any server-based application or service.
- For production environments, you should use an orchestrator (like Service Fabric or Kubernetes) to host scalable Windows Containers–based applications.
- Azure VMs hosting containers are a fast and simple way to create small Dev/Test environments in the cloud.
- Azure SQL Database Managed Instance is the recommended "by default" choice when migrating your relational databases, from existing applications, to Azure
- Visual Studio 2017 and Image2Docker are fundamental tools for you to start modernizing your existing .NET applications with Windows Containers by accelerating the getting started learning curve.
- When establishing containerized applications in production you will always need to settle it down on a DevOps culture and DevOps tools for CI/CD pipelines, like Visual Studio Team Services or Jenkins.
- Microsoft Azure provides the most comprehensive and complete environment to modernize your existing .NET Framework applications with Windows Containers, cloud infrastructure and PaaS services.