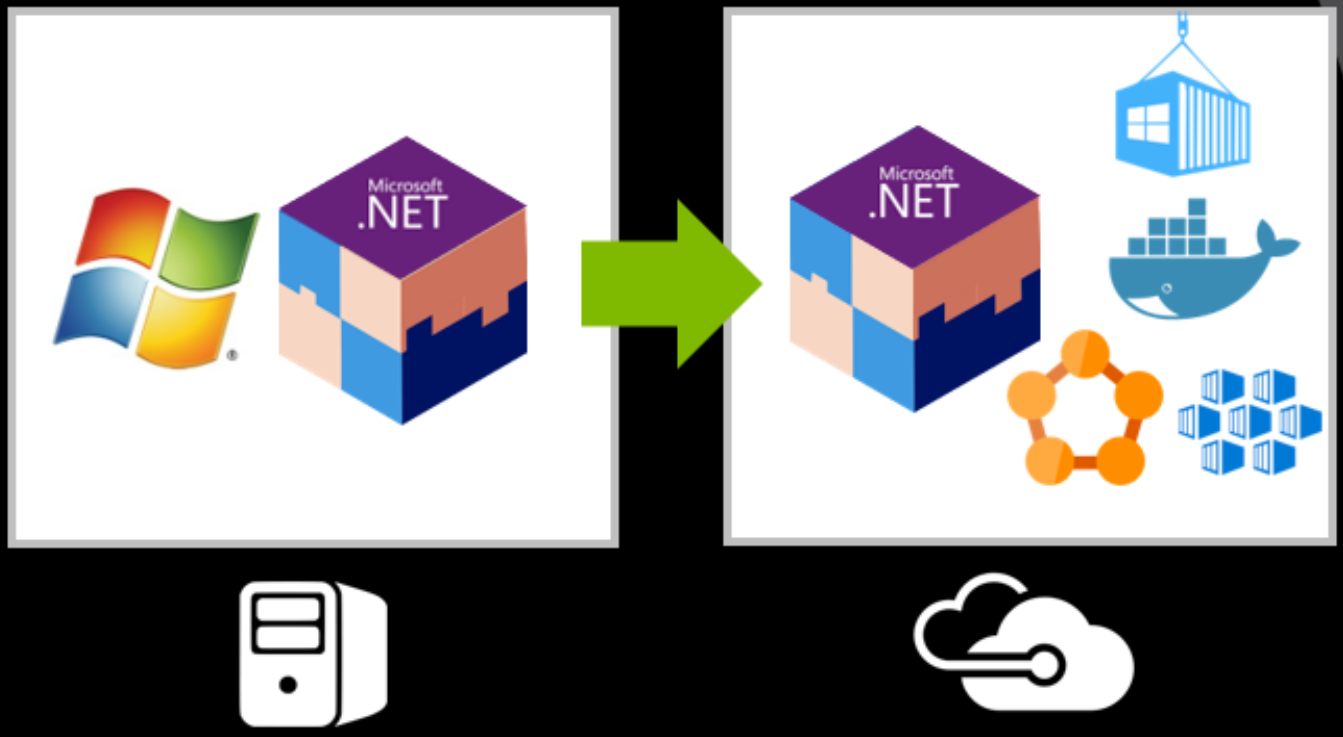
DRAFT

Microsoft

Modernizing existing .NET applications with Azure cloud and Windows Containers



Cesar de la Torre
Microsoft Corp.

Modernize existing .NET applications with Azure and Windows Containers
PUBLISHED BY
Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

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

Author:

Cesar de la Torre, Sr. PM, .NET product team, Microsoft Corp.

Participants and reviewers:

Full name, Title, Team, Company

(Pending to add the reviewers list)

TBD SECTION IN DRAFT

To be completed when finished the review process

Contents

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. Also, depending on the type of application, re-architecting your apps might not be necessary. To optimize the cost-effectiveness of your organization's cloud migration strategy, consider the needs of your business and the app's requirements. 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 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 apps directly to the cloud by modernizing specific areas, without re-architecting or recoding an entire web application.

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.

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. In the cloud, you can set up thousands of servers (virtual machines) 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 applications that essentially run the business and provide differentiated value for the organization, you might benefit from investing in *cloud-optimized* and *cloud-native* application architectures. But for applications that are 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.

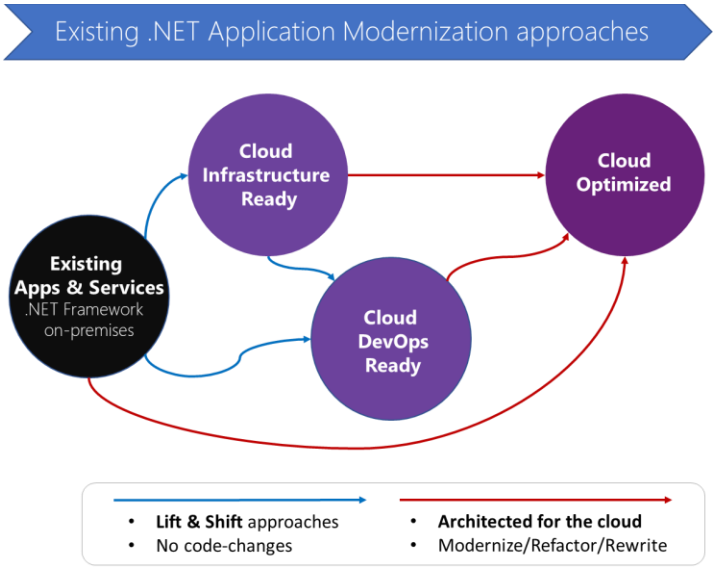


Figure 1–1: Modernization paths for existing .NET applications and services

Each migration approach has different benefits and reasons for using it. You can choose one of these approaches 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.

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

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.
2. **Cloud DevOps-Ready**: In addition to an initial lift and shift, and without needing to re-architect or alter code, you can reap more benefits from the cloud. You can improve the agility of your applications to ship faster by refining your DevOps processes. You achieve this by using base technologies like Windows Containers (based on Docker Engine). Containers remove the friction 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 optional for many apps:

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

Table 1-1 describes the main benefits 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		
Apps deployed to VMs in Azure	Containerized apps deployed to VMs, Azure Service Fabric, or Azure Container Service (or Kubernetes), and so on	Containerized microservices or regular applications based on PaaS (Azure App Service, Azure Service Fabric, Azure Container Service, Kubernetes)
Data target		
SQL or any relational database on a VM	Azure SQL Database Managed Instance	Azure SQL Database, Azure Cosmos Database, or other no-SQL
Advantages		
- No re-architecting, no new code - Least effort for quick migration - Supported least-common denominator in Azure - Basic availability guarantees - After moving to the cloud, it is easier to modernize further	- No re-architecting, no new code - Containers offer small incremental effort above VMs - Improved deployment and DevOps agility to release because of containers - Increased density and lower deployment costs - Portability of apps and dependencies - If you use Azure Container Service (or Kubernetes) and Azure Service Fabric, provides high availability and orchestration - Guest container patching - Nodes/VM patching in Service Fabric - Flexibility of host targets: Azure VMs or VM scale sets, Azure Container Service (or Kubernetes),	- Architect for the cloud, refactor, new code needed - Microservices cloud-native approaches - New web apps, monolithic, n-tier, cloud-resilient, and cloud-optimized - Fully managed services - Automatic patching - Optimized for scale - Optimized for autonomous agility per 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

	Service Fabric, and future container-based choices	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

Figure 1-2 shows the three cloud maturity models from a different perspective, including primary technologies and architecture styles used at each maturity level.

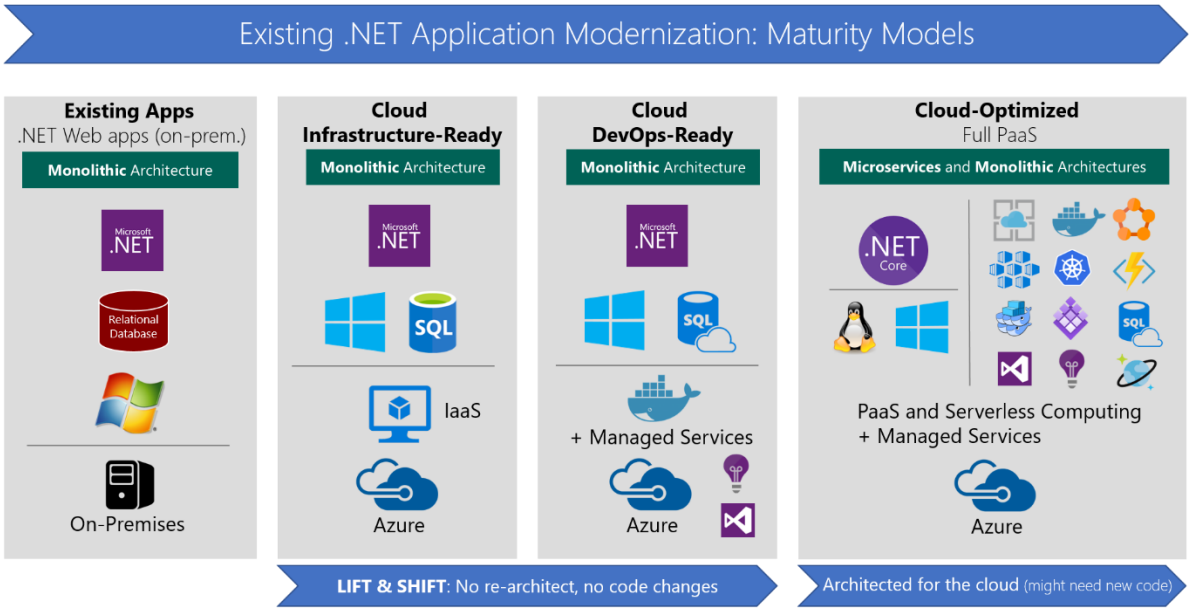


Figure 1–2: Maturity levels for modernizing existing .NET web applications

Note that we highlight the most common scenarios, but many variations are possible when it comes to architecture. (For example, the maturity models apply not only to monolithic architecture for existing web apps, but also to service orientation, n-tier, and other architecture style variants.)

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

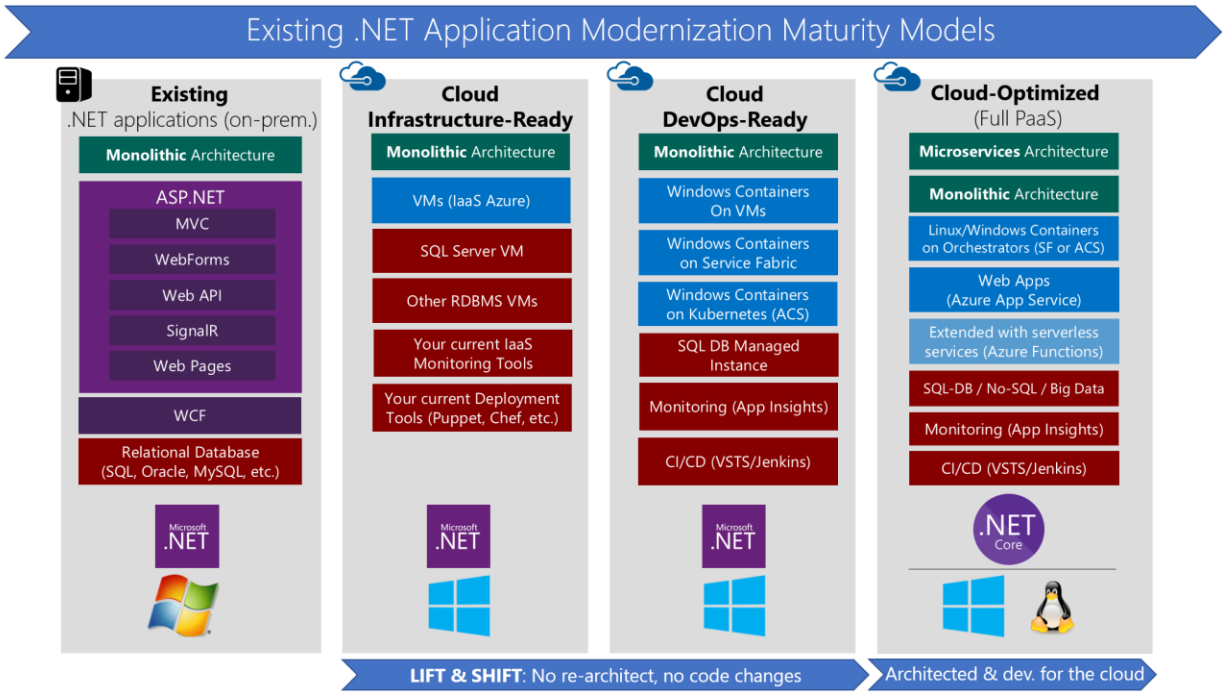


Figure 1–3: Internal technologies for each modernization maturity level

.NET Framework applications initially were built by using the .NET Framework version 1.0, which was released in 2002. These applications ran on Windows Server and Internet Information Server (IIS), and used a relational database, like SQL Server, Oracle, or MySQL.

Most of these existing .NET applications might now be based on .NET Framework 4.x, or even on .NET Framework 3.x, 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 traditional .NET Framework technologies depend on Windows. That dependency is important to consider if you are simply migrating legacy apps, and want to make minimal application infrastructure changes.

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

- **Cloud Infrastructure-Ready** apps (rehost or basic lift and shift): As a first step, many organizations are looking only to quickly execute their migration to the cloud. In these cases, applications are simply rehosted. Most rehosting can be automated with Azure cloud tools like Azure Site Recovery and Azure Database Migration Service. You also can manually set up rehosting, to 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 only minor configuration changes. The networking in this case looks similar to an on-premises environment, especially if you create virtual networks in Azure.
- **Cloud DevOps-Ready** apps (improved lift and shift): This model is about making a few cloud 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, which 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. This improves your agility when deploying to multiple environments. Then, moving to production, you can deploy your Windows Containers to orchestrators like Azure Service Fabric or Azure Container Service (or Kubernetes, DC/OS, or Swarm). During this initial modernization, you also can add assets from the cloud. You can add 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. Also by using Windows Containers, you can easily migrate your data to a database in Azure SQL Database Managed Instance, and monitor with Azure Application Insights, all without changing the core architecture of the application.

- **Cloud-Optimized** apps: 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 (or Kubernetes, DC/OS, or Swarm). These orchestrators are made specifically for microservices and multi-container applications. All of these approaches typically require you to write new code—code that is adapted to specific PaaS platforms, or code that aligns with specific architectures, like microservices.

Lift and shift scenarios

For lift and shift migrations, it's important to note that you can use many different variations in your application scenarios. If you only rehosted your application, you might have a scenario like the one shown in Figure 1-4, where you are using VMs in the cloud only for your application and your database server.

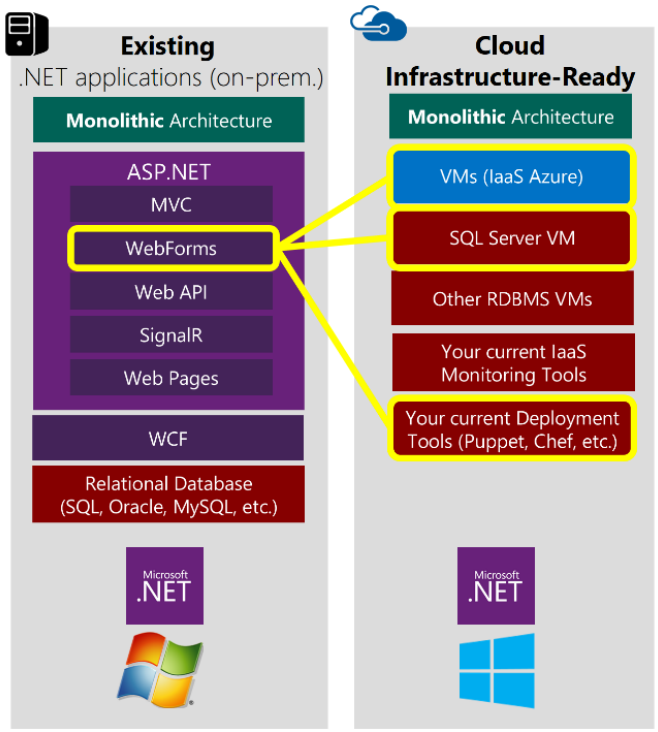


Figure 1–4: Example of a pure IaaS scenario in the cloud

You might have a pure cloud-ready application that uses only elements from that maturity level. Or, you might have an intermediate-state application with some elements from lift and shift, and other elements from cloud-ready (a "pick and choose" or mixed model), like in Figure 1-5.

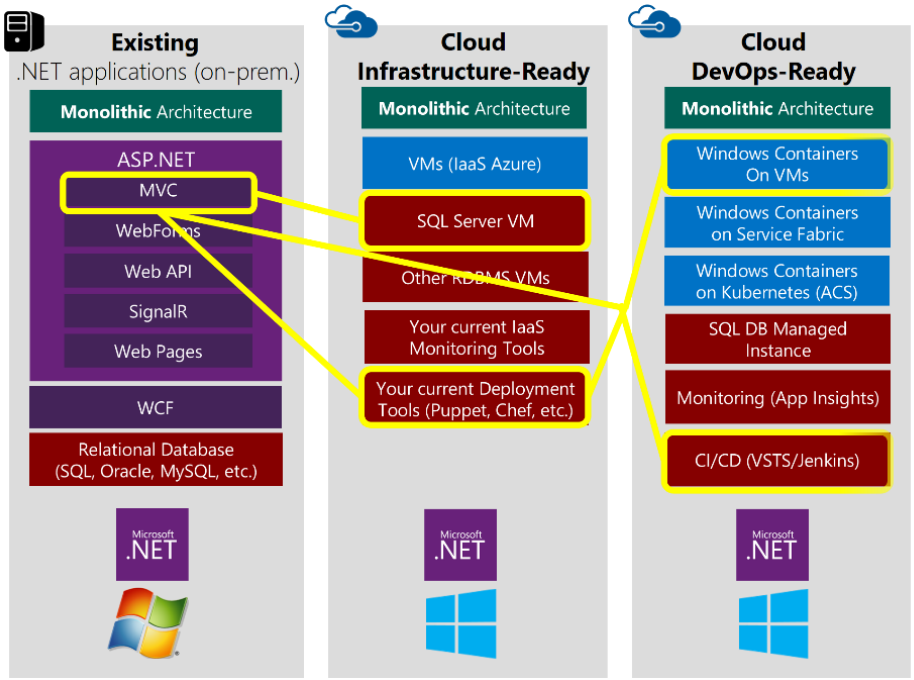


Figure 1–5: Example of a "pick and choose" scenario, with database on IaaS, DevOps, and containerization assets

Then, as the ideal scenario for many applications, 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.

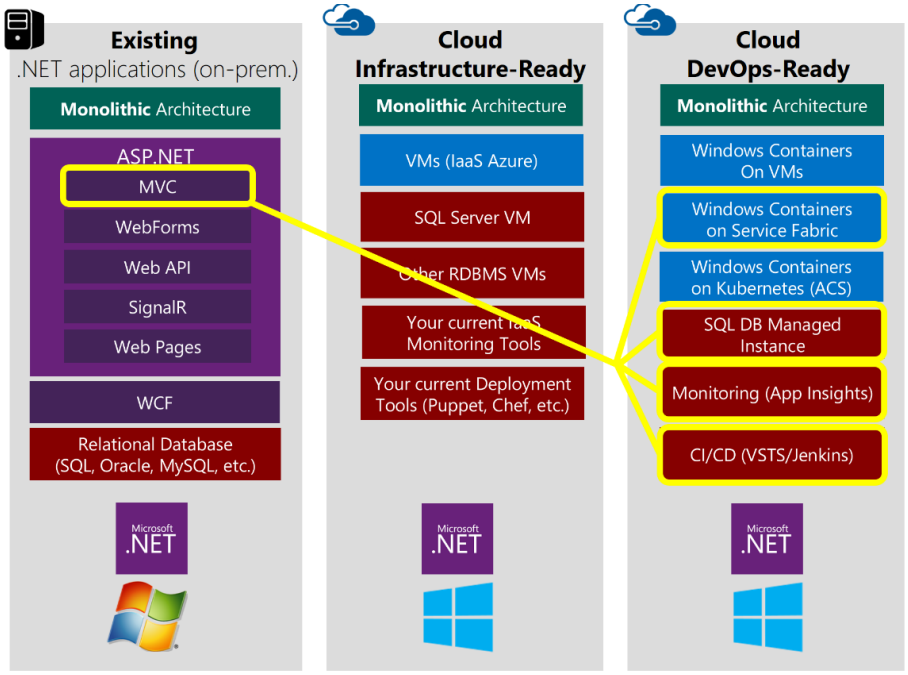


Figure 1–6: Example Cloud DevOps-Ready applications scenario using Windows Containers and managed services

Going even further, as already mentioned, 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.

What this guide does not cover

This guide covers a specific subset of the previous diagrams, 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. All of these are fundamental for deploying applications in the cloud faster, and with agility.

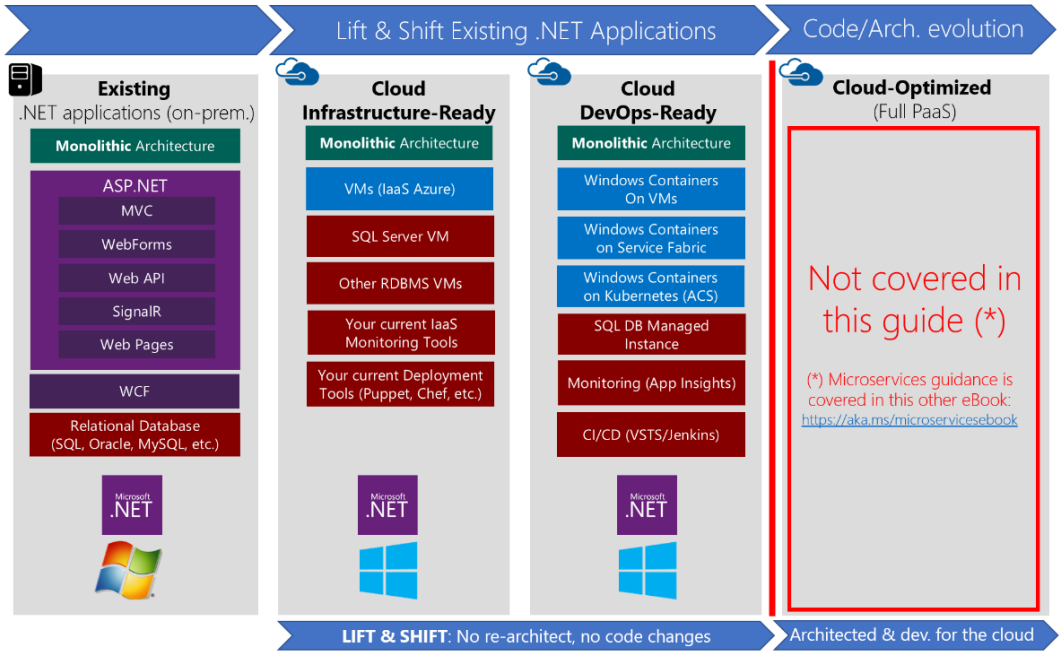


Figure 1–7: Lift and shift and initial modernization to Cloud DevOps-Ready applications

The focus of this guide is specific. We show you the path to achieve a lift and shift of your existing .NET applications, with no re-architecting and no code changes. Ultimately, we show you how to make your application Cloud DevOps-Ready.

This guide doesn't cover how to work with cloud-native applications, such as moving to a microservices architectures. To re-architect your applications or to create brand-new applications that are based on microservices, see .NET Microservices: Architecture for Containerized .NET Applications.

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 when 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 to deploy to the cloud by 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 is designed for architects and technical decision makers who want an overview, but who don't need to focus on implementation and the technical, step-by-step details.

The last two chapters of this guide introduce 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 on 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 at 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 repository 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. The sample apps don't use .NET Core or ASP.NET Core.

Both sample apps have a second version, with modernized code, 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 reference application is evolving. We welcome your feedback! If you have comments about how this guide might be more helpful, send them to dotnet-architecture-ebooks-feedback@service.microsoft.com.

CHAPTER 2

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 Azure infrastructure as a service (IaaS), it's important to analyze the reasons *why* you'd want to migrate directly to IaaS in Azure. Essentially, at this modernization maturity level, you'd start using virtual machines 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.

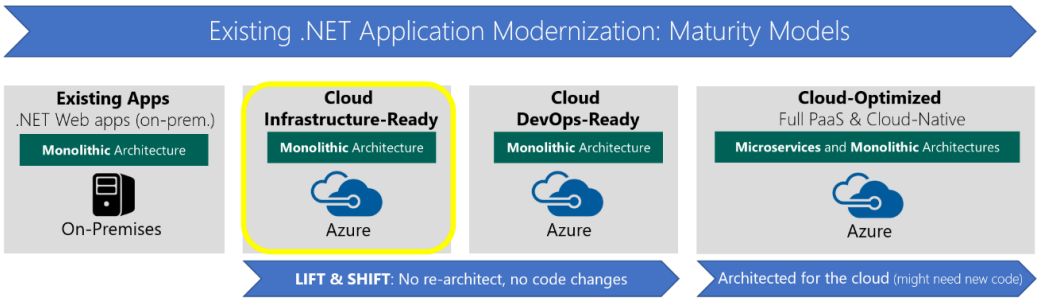


Figure 2–1: Positioning Cloud Infrastructure-Ready applications

Why migrate existing .NET web applications to Azure IaaS

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

After you make the decision to move to the cloud, the main reason that 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, so it's the quickest path to the cloud.

But, taking the quickest path to the cloud doesn't mean that you will see the most benefits from having your applications running in the cloud. An organization will reap the most significant benefits at the Cloud DevOps-Ready and PaaS (Cloud-Optimized) maturity levels.

It's also 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 been achieved. Also, your organization will have gained better skills for working in the cloud, and made the shift to operating in a "cloud culture."

When to migrate to IaaS instead of PaaS

In the next sections, this guide focuses on Cloud DevOps-Ready applications that are mostly based on PaaS platforms and services. They also give you the most benefits from running your apps in the cloud.

If your goal is simply to move existing applications to the cloud, first, identify existing applications that will require substantial modifications to run in Azure App Service. If you don't want to use Windows Containers and orchestrators like Azure Service Fabric or Kubernetes, you can use VMs (IaaS) to simplify the migration to the cloud.

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. App Service and Windows Containers on top of orchestrators like Service Fabric are PaaS.

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

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 replicate them to a secondary on-premises location. Replication to Azure eliminates the cost and complexity of maintaining a secondary datacenter. You can even replicate a workload that's running on a supported Azure VM, on-premises Hyper-V VM, VMware VM, or Windows or Linux physical server.

Site Recovery is 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.

Additional resources

- **Azure Cloud Migration**
  https://azure.microsoft.com/en-us/solutions/cloud-migration/
- **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 Databse, 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.

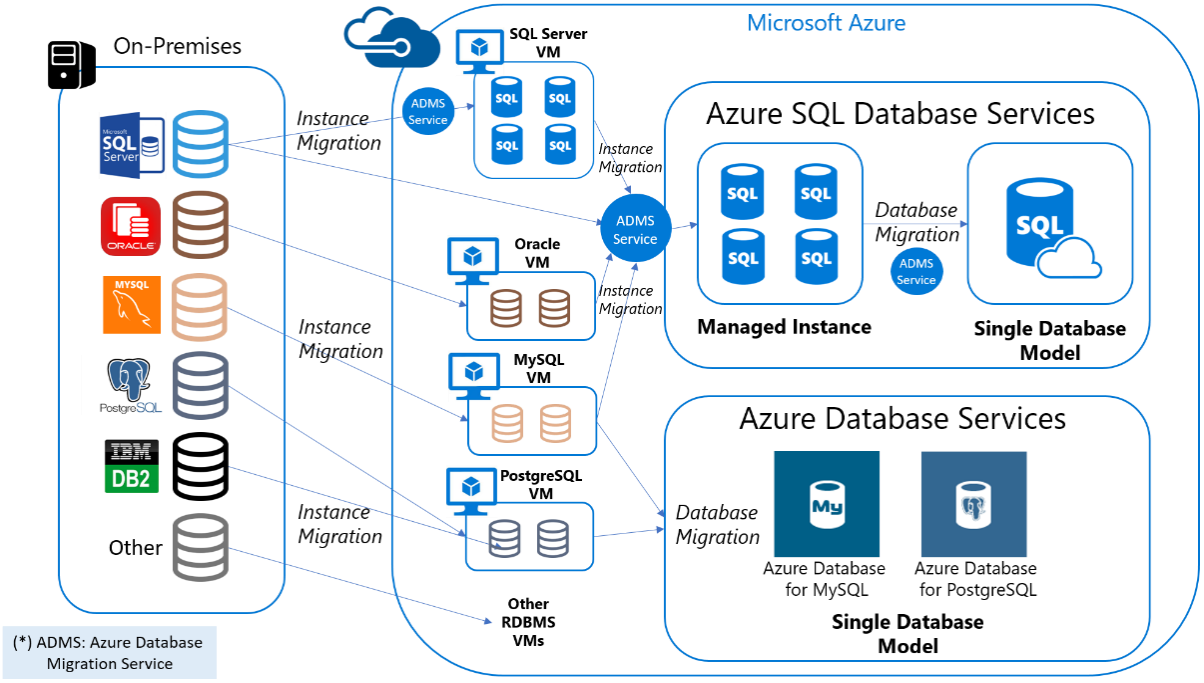


Figure 3–1: Database migrations paths in Azure

When to migrate to Managed Instance in Azure SQL Database

In most cases, Azure SQL Database Managed Instance will be your first 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 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.

For example, an organization that has made deep investments in instance-level SQL Server capabilities would benefit from migrating to a 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 a standard SQL database.

An organization that operates in a highly regulated industry, and which needs to maintain isolation for security purposes, also would benefit from choosing the SQL Database managed instance model.

Managed Instance in Azure SQL Database has the following characteristics:

- Security isolation through Azure Virtual Network
- Application surface compatibility with:
    - 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:
    - 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.

This choice 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

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 regular 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 DB 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 accelerates your time to market. SQL Database features built-in machine learning that quickly learns your app's unique characteristics. It dynamically adapts to your app to maximize performance, reliability, and data protection. 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
- Secure and compliant, to protect sensitive data
- 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 first 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 the best option. It might not be the best way to take advantage of 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 No-SQL 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

There might be just a few cases where you still might need to migrate to SQL Server as a regular VM. For example, if you need to use SQL Server Reporting Services. In most of the cases, though, Azure SQL Database Managed Instance can provide anything you need to migrate from on-premises SQL servers.

Migrate relational databases by using Azure Database Migration Service

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 the 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, Oracle or 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 have additional impact.

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

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.

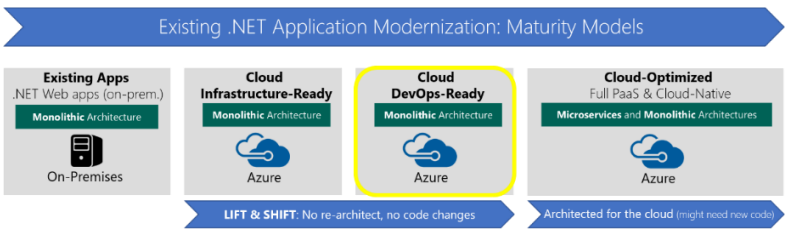


Figure 4–1: Positioning Cloud DevOps-Ready applications

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

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, your application is Cloud DevOps-Ready.

The key word 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 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) are key technologies to achieve 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 them, 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.

- **Managed cloud**: The cloud provides a platform that is managed, elastic, scalable, and resilient. These characteristics are fundamental for you 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 cannot 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 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.

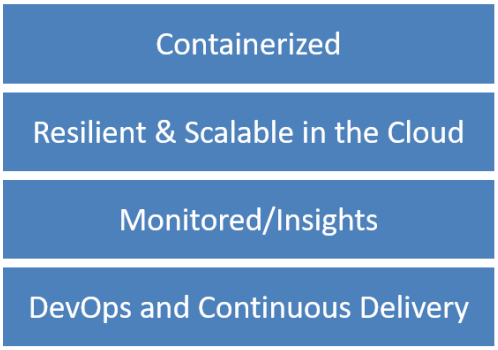


Figure 4–2 Main pillars of a Cloud DevOps-Ready application

To summarize, a question: What is a Cloud DevOps-Ready application all about? It's an approach to building and managing applications that takes advantage of the cloud computing model while using a combination of containers, managed cloud services, resilient application techniques, monitoring, continuous delivery, and DevOps, all without needing to re-architect and recode your existing applications.

Your organization's adoption of these technologies and approaches can be gradual. You don't have to embrace all of them, all at once. You can adopt them gradually or 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-ready application (without re-architecting):

- **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, auto-scale, high availability, not only related to the compute features (VMs

and containers) but also related to the resources in the cloud, like DBaaS, CaaS, and any infrastructure needed by applications.

- **Resilient application and infrastructure**. When using the cloud, you need to embrace transient failures because failures will happen in the cloud and the infrastructure and hardware is "replaceable". However, the inner cloud capabilities plus certain application development techniques to implement resiliency ensure that it is much easier to recover from unexpected failures in the cloud thanks to automated recovering capabilities.
- **Deeper insights into application performance**. Cloud monitoring tooling (like Azure Application Insights) provides visualization for health management, logging and notifications with audit logs making 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) provide the best way to have a no-cloud-locked application. By using containers, Docker hosts and multi-cloud orchestrators you can easily move from one environment/cloud to another, plus containers eliminate typical frictions 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, grouped by subjects, 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, 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 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 this. With 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 products 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 to build applications that run in containers. In this guide, we focus on .NET. But, you could also 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 the applications and containers running in the cloud is critical and fundamental for any cloud-ready application. Azure Application Insights and Microsoft Operations Management Suite are the main Microsoft tools that provide monitoring and auditing for cloud-ready applications.
- **Provisioning**: Automation tools help you provision the infrastructure and deploy the application to multiple environments (production, testing, staging, and so on). 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 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 continuous integration (CI) and continuous delivery (CD) pipelines, including release management.

The following chapters in this guide and the related walkthroughs focus specifically on details about the runtime layer (Windows Containers). They describe possible ways to deploy Windows Containers on Windows Server 2016 (and later versions) virtual machines. They also cover more advanced orchestrator layers (Azure Service Fabric, Kubernetes, and so on). Setting up orchestrator layers is a fundamental requirement for modernizing existing .NET Framework applications (Windows-based) as Cloud DevOps-Ready applications.

Monolithic applications can be Cloud DevOps-Ready applications

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.

And, 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.

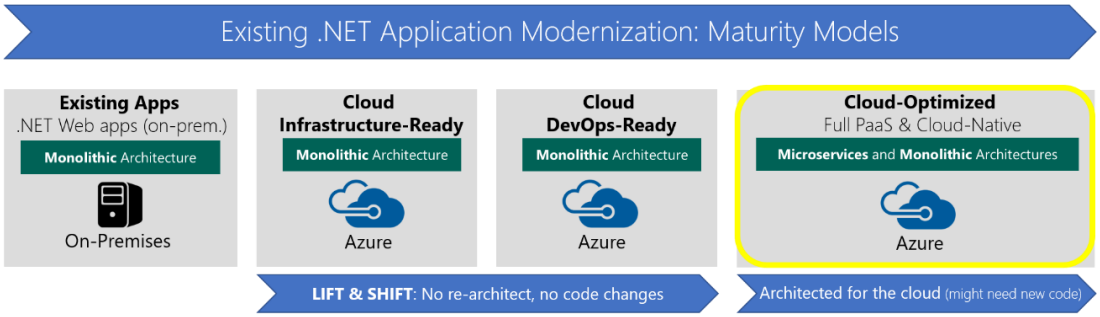


Figure 4–3: Positioning Cloud-Optimized applications

The Cloud-Optimized 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 to orchestrators.

In some cases, you also might create cloud-native applications based on microservices architectures to evolve with agility and scale to limits that would be difficult to achieve in an on-premises environment.

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 for the cloud, especially when 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.

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 apps and cloud-native applications.

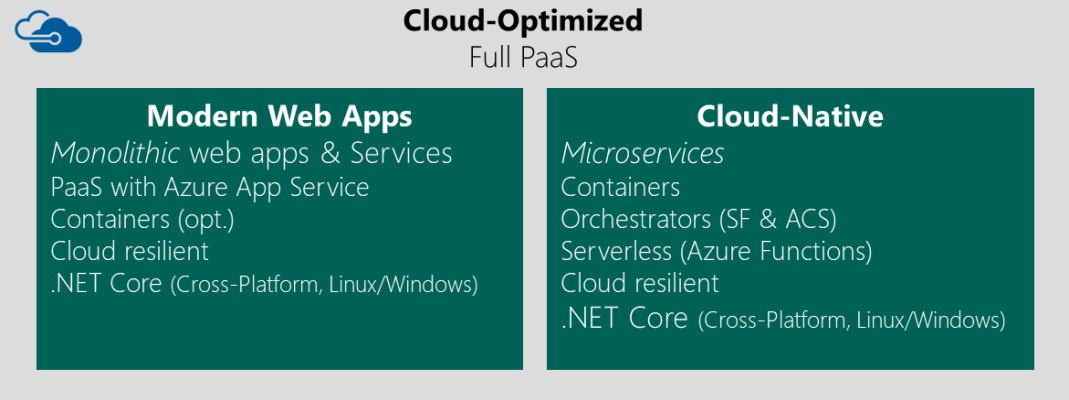


Figure 4–4: App types at the Cloud-Optimized level

You can extend basic modern web apps and cloud-native apps by adding other flavors, like artificial intelligence (AI), machine learning (ML), and IoT services. 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, and specific technologies and platforms, compared to a native .NET application.

But, creating new applications with no microservices also makes sense. There are many new and still modern scenarios where a microservices-based approach is more than you need. You might just want to create a simpler monolithic web application (or coarse-grained services in n-tier apps) by using the latest development technologies (NET Core, cross-platform, and so on). You can still make full use of the cloud PaaS capabilities like the ones offered by Azure App Service. You still reduce your maintenance work to the limit.

Also, because in Cloud-Optimized scenarios you develop new code (for a full application or for partial sub-systems), 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 as .NET Core in a lean framework. You'll get a small memory footprint and fast start in containers, and your applications will be highly performant. This approach perfectly fits with the needs of microservices and containers, and you also 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

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 modernizing existing applications. The key difference between a cloud-native application and a cloud-ready application is the recommendation to use microservices architectures in a cloud-native approach, instead of monolithic design and deployment or traditional n-tier architectures in a cloud-ready approach.

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 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 in 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 for applications that are created from scratch. You can also use microservices to evolve existing applications (either on-premises or cloud DevOps-Ready apps) toward cloud-native applications. You can start adding a few microservices to existing applications to learn about the new microservice paradigms.

But 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 developed, tested, versioned, deployed, and scaled independently. This can include any related, autonomous database per microservice.

For a detailed look at microservices architecture that you can implement by using .NET Core, see the 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. These 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 over 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 those cases, it is worth investing in a more complex software architecture, for increased long-term agility and more efficient application maintenance. But in other, 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 small, new 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 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. Then, you 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 using containers. 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 Linux containers. So, the question arises: "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 way, it is most effective, and your application will have a simpler maintenance path because of the PaaS benefits detailed in the following sections.

However, if your application has server or custom dependencies that are not supported in Azure App Service, you might need to consider other choices based on Windows Containers. Examples of server or custom dependencies include third-party software or an .MSI 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, 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 Server. Depending on the volume of work that would require, you'd have to carefully evaluate whether that is 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. With App Service, you avoid the infrastructure and database management costs associated with upgrading and maintaining web applications 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 also 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 to 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, with 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 use an open-source app from the Azure Web Application Gallery, or you can create a new site by using the framework and tools of your choice. The 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 up and down 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 testing multiple site designs.
- Load balancing and Distributed Denial of Service (DDoS) protection.
- Traffic management to direct users to the closest geographic deployment.

Even if App Service might be the best choice for new web apps, for existing applications, App Service might be the best choice only if your application dependencies are supported in App Service. Some existing .NET Framework applications require substantial modifications to run in App Service because of third-party .msi installations in the server, use of COM and COM+ components, or other additional cases. For those cases, moving to Windows Containers and other Azure deployment environments can be a better choice when migrating than having to re-architect or re-write significant volume of code.

Additional resources

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

Benefits of moving to Windows Containers

The main benefits you get by using Windows Containers is a more reliable and improved deployment experience. 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 main compute environments (in general availability, as of mid-2017) in Azure that support Windows Containers are Azure Service Fabric and basic Windows Containers hosts (Windows Server 2016 VMs). Those environments are the main infrastructure scenarios in this guide.

You also could deploy Windows containers to other orchestrators, like Kubernetes, Docker Swarm, or DC/OS. Currently, those tools are in preview for using Windows Containers.

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. 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 and migrate sites 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.

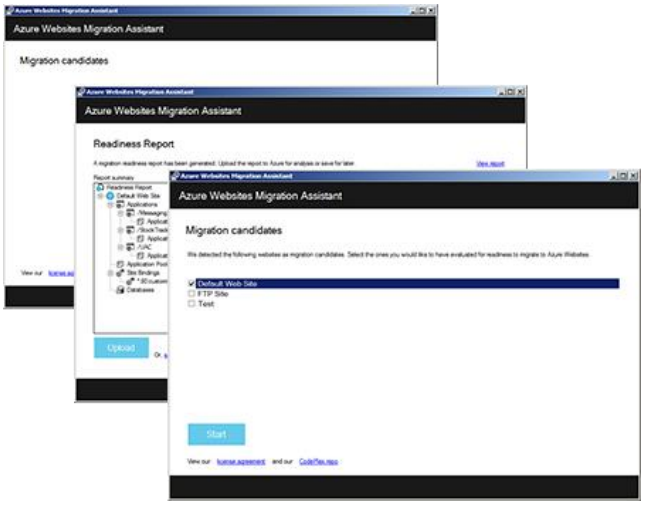


Figure 4–5: Using Azure App Service Migration Assistant

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

After web sites are migrated to Azure App Service, the sites have everything needed to run safely and efficiently. Sites will be set up and running automatically in our cloud PaaS service (Azure App Service).

The App Service migration tool can analyze your websites, report on their compatibility to move to App Service and, if you're happy, it will do the migration of content, data, and settings for you. If a site is not quite compatible, it will tell 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 box. The application in its container is not affected by applications or processes that exist outside of that container. Everything the application depends on to run successfully as a process also is inside this 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 deploy it in a reliable and reproducible way. You could say that Docker is not only a technology, but also a philosophy and a process.

A container is becoming the unit of deployment.

Benefits of containers (Docker Engine on Linux or Windows)

Building applications by using containers (which also could be defined as lightweight building blocks) offers a significant increase in agility to build, ship, and run 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 VMs instead of to containers, you probably already have a method in place for deploying ASP.NET apps to your VMs. It's likely, though, that your method requires a lot of 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, and then testing afterward. All those steps in the deployment add time and risk to deployments, and you will get failures whenever any dependency is not present in the target environment.

In Windows Containers, the process of packaging applications is fully automated. It's based on the Docker platform, on which you can have automatic updates and rollback for container deployments. The important improvement is that you create images that are like snapshots of your application, plus all its dependencies. The images are Docker images (a Windows Container image, in this case), and run ASP.NET apps in containers without going back to source code. The container snapshot becomes the unit of deployment.

Large numbers of companies 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.

- **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 a container level. A container can be secured, isolated, and even limited by setting execution constraint policies. 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 for automating 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.

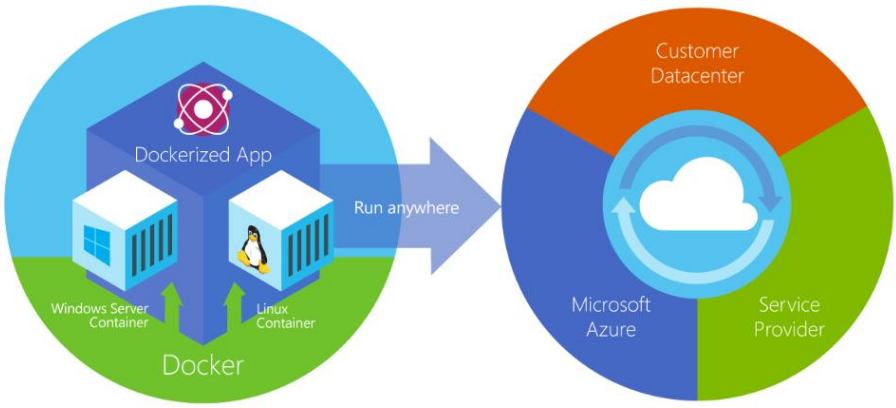


Figure 4–6: Docker deploys containers at all layers of the hybrid cloud

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.

Benefits of Windows Containers for your existing .NET applications

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

However, when we talk about existing .NET applications, we mostly mean traditional applications that were created by using the .NET Framework, like traditional ASP.NET web applications (not using .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.

For these reasons, 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—by containerizing them. 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, and 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 Nano Server. These versions of Windows provide different characteristics (like IIS versus a self-hosted web server like Kestrel) that might be needed by .NET Framework or .NET Core.

For Linux, many distros are available, but only a few of them are supported in official .NET Docker images (like Debian).

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

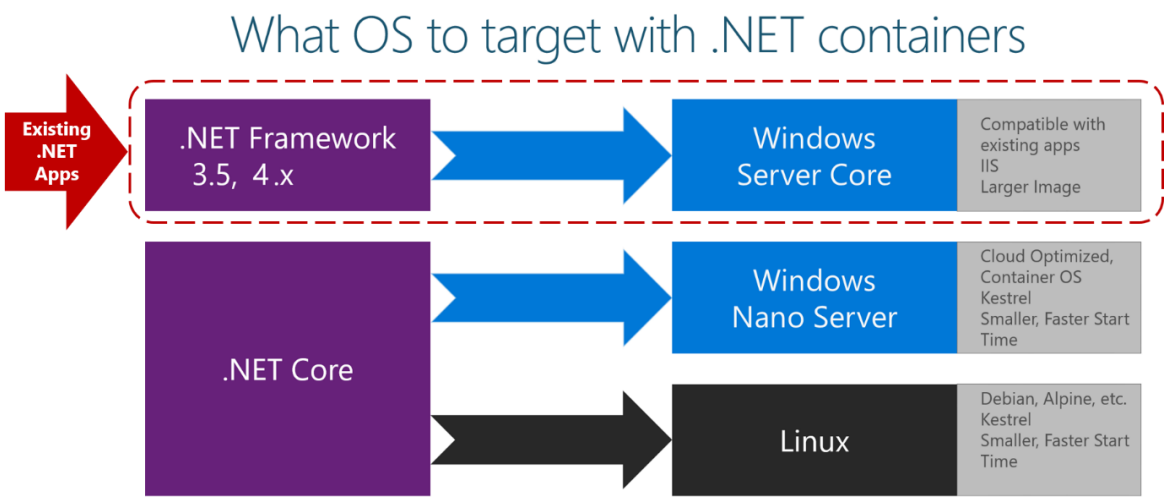


Figure 4-7. Operating systems to target based on .NET Framework version

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

When you add the image name to your Dockerfile file, you can select the operating system and version by using a tag, like 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 the following:

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 mid-2017, you also can use a new feature in Docker called multi-arch images. The .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 **microsoft/dotnet:2.0.0-runtime** tag. 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: This container type 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: This container type expands on the isolation provided by Windows Server Containers by running each container in a highly optimized virtual machine. 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 virtual machine. 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 later, 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 to not deploy to Windows Containers

Some Windows technologies are not supported by Windows Containers or by Azure App Service. 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 Queing (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 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)

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

- 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.
- 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 could continue to use those tools to deploy to Azure VMs. But, 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 soon will need to use platforms that move further away from IaaS VMs. This is so that they can easily achieve improved automated scalability and high scalability, and get significant improvements in a complete management experience that they can use for deployments, upgrades, versioning, rollbacks, and health monitoring. You can achieve those 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 by running applications at a higher density.
- A consistent deployment contract between development and operations.

Cost reductions are 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, but the operations team will only accept 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 only work with certain ones. Essentially, each application is containerized in a self-contained deployment image.

Some organizations will continue modernizing by adding the use of microservices (Cloud-Optimized and cloud-native applications), but many organizations will stop here (Cloud DevOps-Ready). As shown in image 4-8, these organizations won't move to microservices architectures because they might not need it. 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.

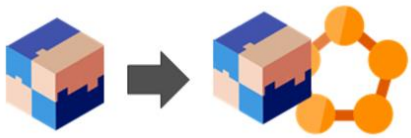


Figure 4-8. Lift and shift an application to Service Fabric

A key approach to Service Fabric is to reuse existing code and simply lift and shift. Or 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 running Windows-based applications and services (it's been running Windows-based services and applications for years, including Tier-1, mission-critical products from Microsoft). 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 not as much as 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 microservice approach. This is where you eventually want to be for certain types of applications, so that you don't have to go through as many costly redesigns. You can start small, scale when needed, deprecate services, add new ones, and evolve with customer use. We know that there are many other problems 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 precisely the sweet spot for Service Fabric.

When to deploy Windows Containers to Azure Container Service (Kubernetes, Swarm)

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, number of hosts, and the orchestrator tools. Azure Container Service handles the infrastructure for you.

If you are already familiar 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 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).

Build resilient cloud-ready apps: Embrace transient failures in the cloud

Resiliency is the ability to recover from failures and continue to function. It 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 will be 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, eventually. You need to design or partially refactor your application if you want to achieve resiliency to those partial failures.

You need to keep your application running in an environment where some sort of failure is certain. Your application must be designed to be resilient. 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 not be available to respond for a short time. Or, a single VM or server can crash. Because clients and services are separate processes, a service might not be able to respond in a timely way 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 period of 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 during that very same moment.

A similar instance 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, as 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. Your applications also should use "circuit breakers," to stop it from trying to access a resource when

it's actually a long-term failure. By using a circuit breaker, the application won't provoke a denial of service to itself.

These techniques can be applied both to HTTP resources and to database resources. In fact, in a monolithic application that uses only a single process (no additional services or microservices) handling transient failures at the database connection level might be enough.

---------- *TBD Image on Retries with Exponential Backoff and circuit breakers* ----------

---

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

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

Additional resources

- **Strategies for handling partial failure**
  https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/implement-resilient-applications/partial-failure-strategies

Why modernize your app's monitoring/telemetry

---------- *TBD SECTION IN DRAFT* ----------

---------- *Content TBD* ----------

Why modernize your app's lifecycle towards DevOps in the cloud

---

---------- *TBD SECTION IN DRAFT* ----------

---------- *Content TBD* ----------

When to migrate to hybrid-cloud scenarios

---------- TBD SECTION IN DRAFT ----------

---------- Content TBD ----------

Single sign-on considerations/guidance

Integration with Windows Active Directory and AAD, custom login, etc.

Networking considerations/guidance (linking to on-premises from Azure, issues, etc.)

Additional resources

- **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 Use Benefit**
  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 following online series 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 involved. You can get the walkthroughs themselves in the eShopModernizing apps GitHub repo wiki at https://github.com/dotnet-architecture/eShopModernizing/wiki.

Technical walkthroughs 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 the initial 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 with a monolithic architecture, that 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.

Both apps can be containerized similar to the way you can containerize a classic Windows Communication Foundation (WCF) application to be consumed as a desktop application, as in this other example (eShopModernizingWCFWinForms).

Goals for this walkthrough

The goal of this walkthrough is just to get familiar with these apps, and with their code and configuration. You can configure the app so that it generates and uses mock data, without using the SQL database, if you want, 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.

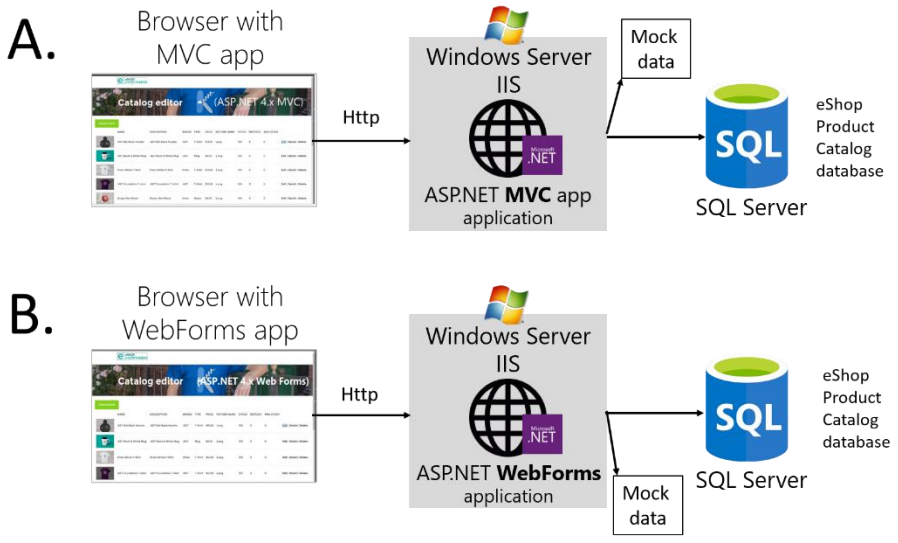


Figure 5-1. Simple architecture scenario of the original legacy applications

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 screenshots.

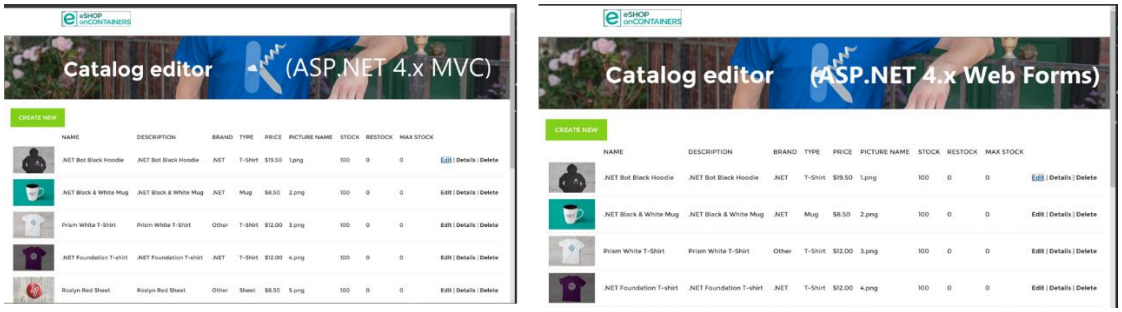


Figure 5-2. ASP.NET MVC and Web Forms applications (existing/legacy technologies)

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. You can see a similar modernization process for apps that were created by using the ASP.NET MVC and ASP.NET Web Forms frameworks.

The dependency taken on ASP.NET 4.x or older (either for Web Forms or MVC) means these applications will not run on .NET Core unless the code is fully re-written and instead uses ASP.NET Core MVC. But this is precisely the point, in this case you don't want to re-architect or rewrite any code but just containerized these existing applications while still using the same .NET technologies and the same code. You will see how you can run applications like these in containers without changes.

Benefits

The benefits of this walkthrough are simple, in this case. Just get familiar with the code and application's configuration based on Dependency Injection, so you can play with it when containerizing and deploying to multiple environments, later on.

Next steps

Go ahead and explore the deeper technical content within the GitHub Wiki page here:

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

This technical walkthrough is written in further details at 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

Windows Containers should be used as a way to improve deployments to production, development and test environments of existing .NET applications based on .NET Framework technologies like MVC, Web Forms or WCF.

Goals for this walkthrough

Show you several alternatives on how to containerize an existing .NET Framework application:

1. Containerized your application through Visual Studio 2017 Tools for Docker (VS 2017 or later).
2. Containerized your application by manually adding the dockerfiles and using the Docker CLI.
3. Containerized your application through Img2Docker tool (Open Source tool from Docker)

This Walkthrough covers the three approaches although it offers further details for the VS 2017 Tools for Docker approach.

Scenario

The diagram below shows the scenario for the containerized eShop legacy applications.

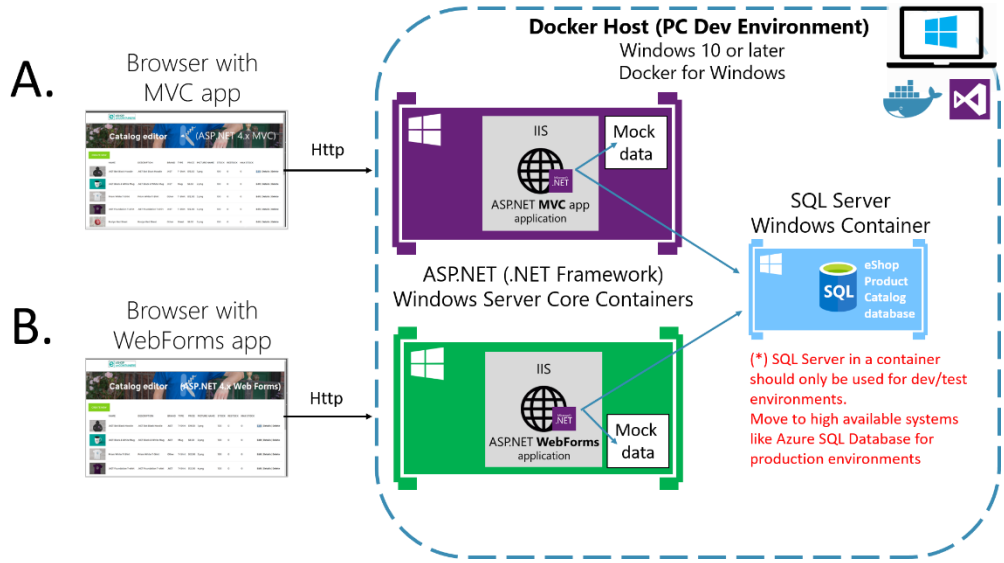


***Figure X-X**. Simplified architecture diagram of containerized applications in development environment*

Benefits of containerizing a monolithic application

There are advantages to running the application in a container. You create an image for the application. From that point forward, 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 included in the image, and is built using the same process. Basically, you control the dependencies of your application within the Docker image and those dependencies travel with the application when deploying the containers.

As additional benefit, developers can all run the application in this consistent environment provided by Windows Containers. Issues that only appear with certain versions will appear immediately for developers rather than surfacing in a staging or production environment. Differences between the development environments among the development team matter less once applications run in containers.

Finally, containerized applications have a flatter scale-out curve. You have learned how containerized apps enable more containers in a VM or more containers in a physical machine. This translates to higher density and fewer required resources, especially when using orchestrators like Kubernetes or Service Fabric.

For all these reasons, consider running existing monolithic ASP.NET web applications or existing services (like WCF or ASP.NET 4.x Web.API services) in Windows Containers.

This containerization operation, in ideal situations, will not need to make any changes to the application code (C#) to run it in a container. In most situations, you will just need the Docker deployment metadata files (dockerfiles and docker-compose files).

Next steps

Go ahead and explore the deeper technical content within the GitHub Wiki page here:

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

This technical walkthrough is written in further details at 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 allows you to have quick dev/test/staging environments where you can run your applications in a common place for testers or business users validating the app. VMs can also be valid for IaaS production environments.

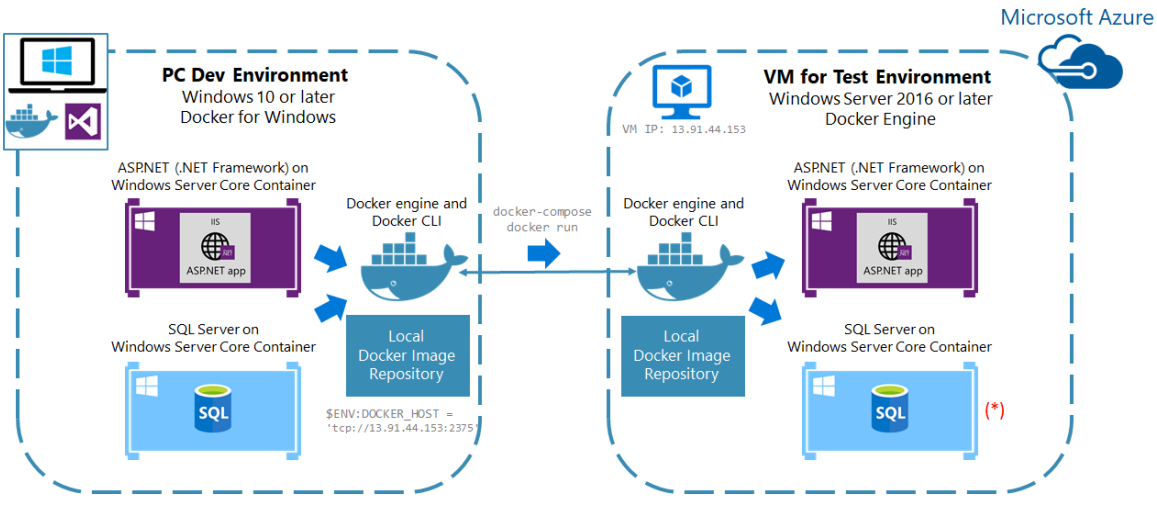
Goals for this walkthrough

This walkthrough shows you the multiple alternatives you can have when deploying Windows Containers to Azure VMs based on Windows Server 2016 or later.

Scenarios

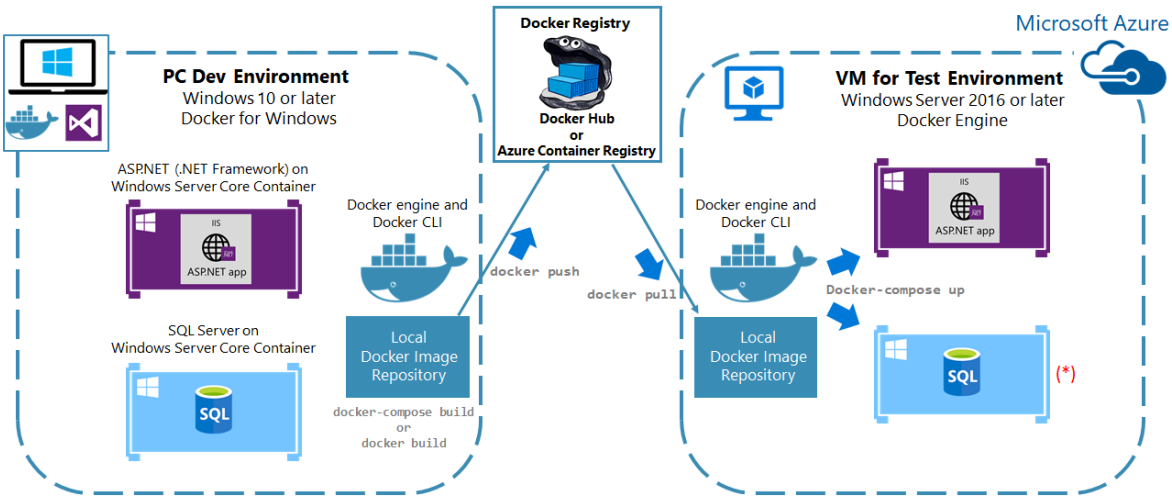
There are several scenarios. The following diagrams show you each of them to be detailed in the page.

A. Deploy to an Azure VM from dev PC through Docker Engine connection



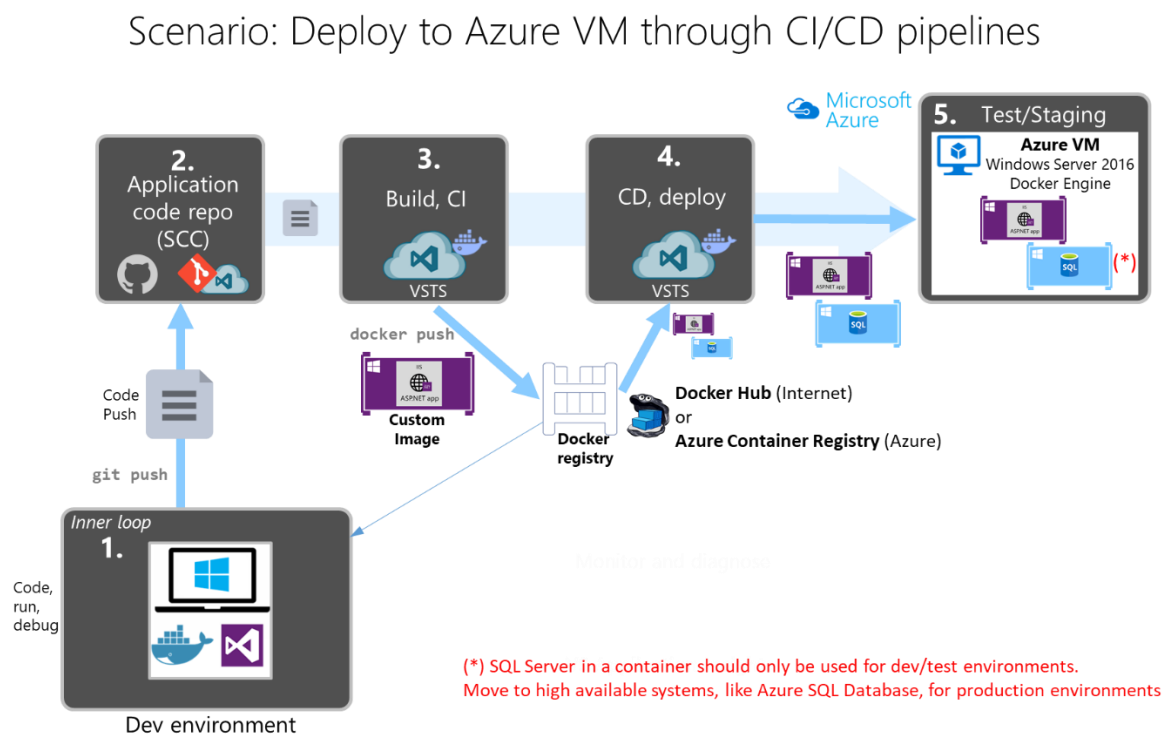
(*) SQL Server in a container should only be used for dev/test environments.
Move to high available systems, like Azure SQL Database, for production environments

B. Deploy to an Azure VM through a Docker Registry



(*) SQL Server in a container should only be used for dev/test environments.
Move to high available systems, like Azure SQL Database, for production environments

C. Deploy to an Azure VM from CI/CD pipelines in VSTS



(*) SQL Server in a container should only be used for dev/test environments.
Move to high available systems, like Azure SQL Database, for production environments

Azure VMs for Windows Containers

Virtual Machines for Windows containers are simply VMs based on Windows Server 2016, Windows 10, or later versions, both with Docker engine setup. Usually, you will use Windows Server2016 in the Azure VMs.

Azure currently provides a VM named "Windows Server 2016 with Containers" which allows you to try the new Windows Server Container feature with both Windows Server Core and Windows Nano Server Container OS Images installed and ready to use with Docker.

Benefits

Although Windows Containers can be deployed into on-premises Windows Server 2016 VMs, however, when deploying into Azure you get a much easier way to get started with ready-to-use Windows Server Container VMs, a common place in the Internet accessible for testers plus automatic scalability of VMs through Azure scale-sets.

Next steps

Go ahead and explore the deeper technical content within the GitHub Wiki page here:

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

This technical walkthrough is written in further details at 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

Applications based on Windows Containers will soon need to use platforms going further from IaaS VMs in order to easily achieve a better automated scalability, high scalability and significant improvement in automated deployments and versioning. You can achieve those goals with the orchestrator Kubernetes, available in Azure Container Services.

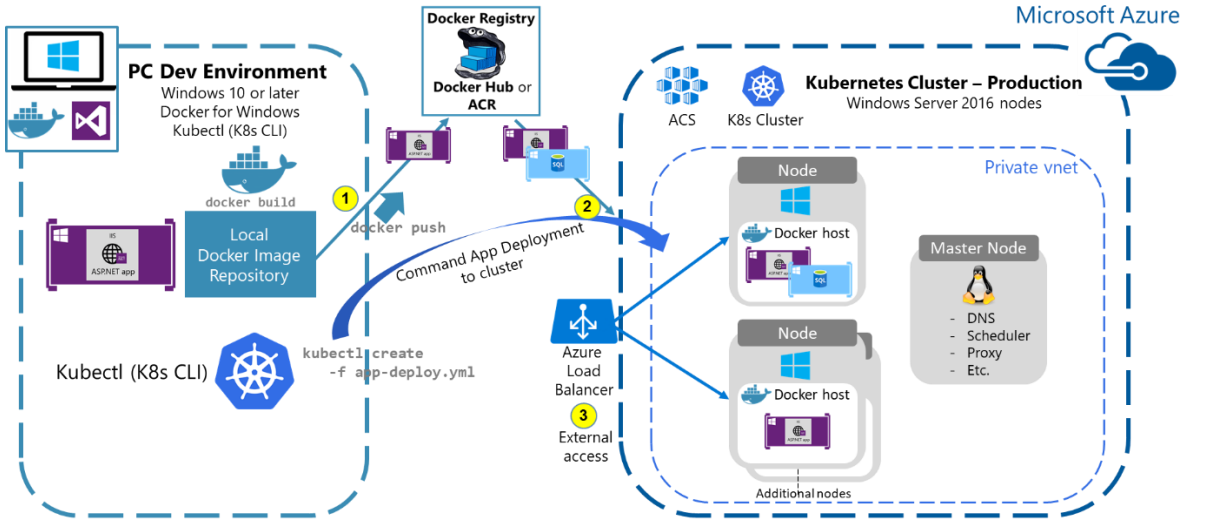
Goals for this walkthrough

Learn how to deploy a Windows Container based application into Kubernetes (K8s) in Azure Container Service (ACS). Deploying to Kubernetes (K8s) from scratch requires a two-step process:

1. Deploy a K8s cluster into ACS
2. Deploy the application and related resources into the K8s cluster

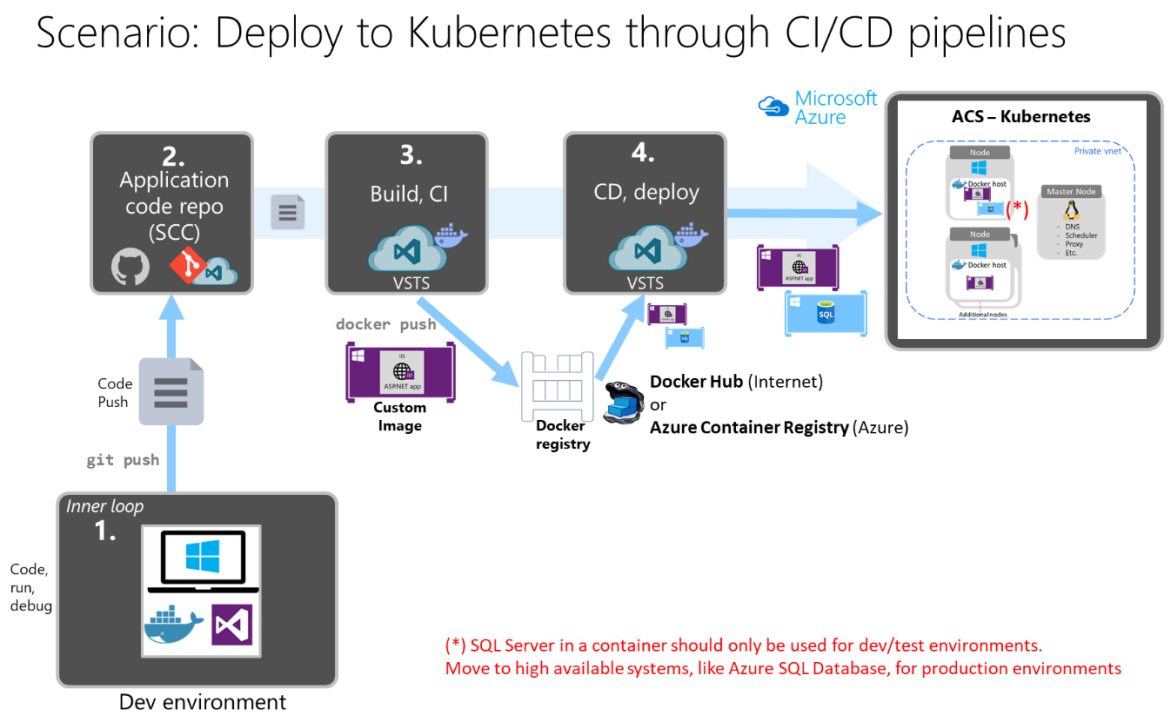
Scenarios

Scenario A: Direct deployment to K8s cluster from development environment



(*) SQL Server in a container should only be used for dev/test environments.
Move to high available systems, like Azure SQL Database, for production environments

Scenario B: Deployment to K8s cluster from CI/CD pipelines in VSTS



Benefits

The benefits of deploying to a cluster in Kubernetes are many. Most of all, this is a production-ready environment that will allow you to scale-out the application based on your desired number of container instances (inner-scalability within the existing nodes) and number of nodes/VMs in the cluster (global scalability of the cluster).

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

Kubernetes allows developers to change your mindset from physical and virtual machines, to a container-centric infrastructure facilitating the following capabilities, among many others:

- Facilitates multi-container based applications
- Replicating container instances and using horizontal autoscaling
- Naming and discovering (internal DNS, etc.)
- Balancing loads
- Rolling updates
- Application health checks

Next steps

Go ahead and explore the deeper technical content within the GitHub Wiki page here:

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

This technical walkthrough is written in further details at 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

Applications based on Windows Containers will soon need to use platforms going further from IaaS VMs in order to easily achieve a better automated scalability, high scalability and significant improvement in automated deployments and versioning. You can achieve those goals with the orchestrator Azure Service Fabric, available in Microsoft Azure cloud but also on-premises or even in any other public cloud you'd like to deploy it.

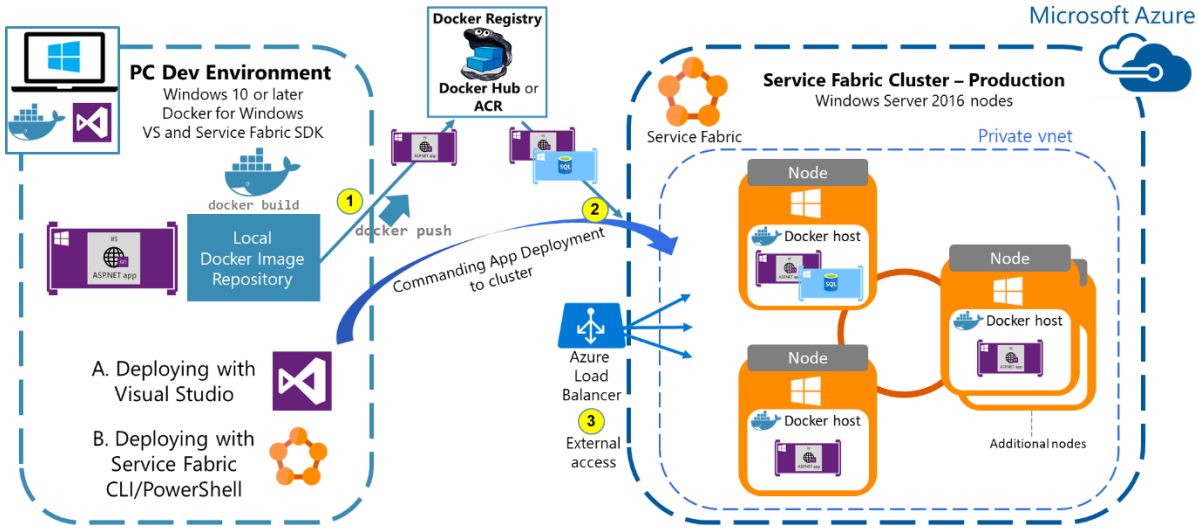
Goals for this walkthrough

Learn how to deploy a Windows Container based application into a Service Fabric cluster in Azure. Deploying to Service Fabric, from scratch, requires a two-step process:

1. Deploy a Service Fabric cluster into Azure (or any other environment)
2. Deploy the application and related resources into the Service Fabric cluster

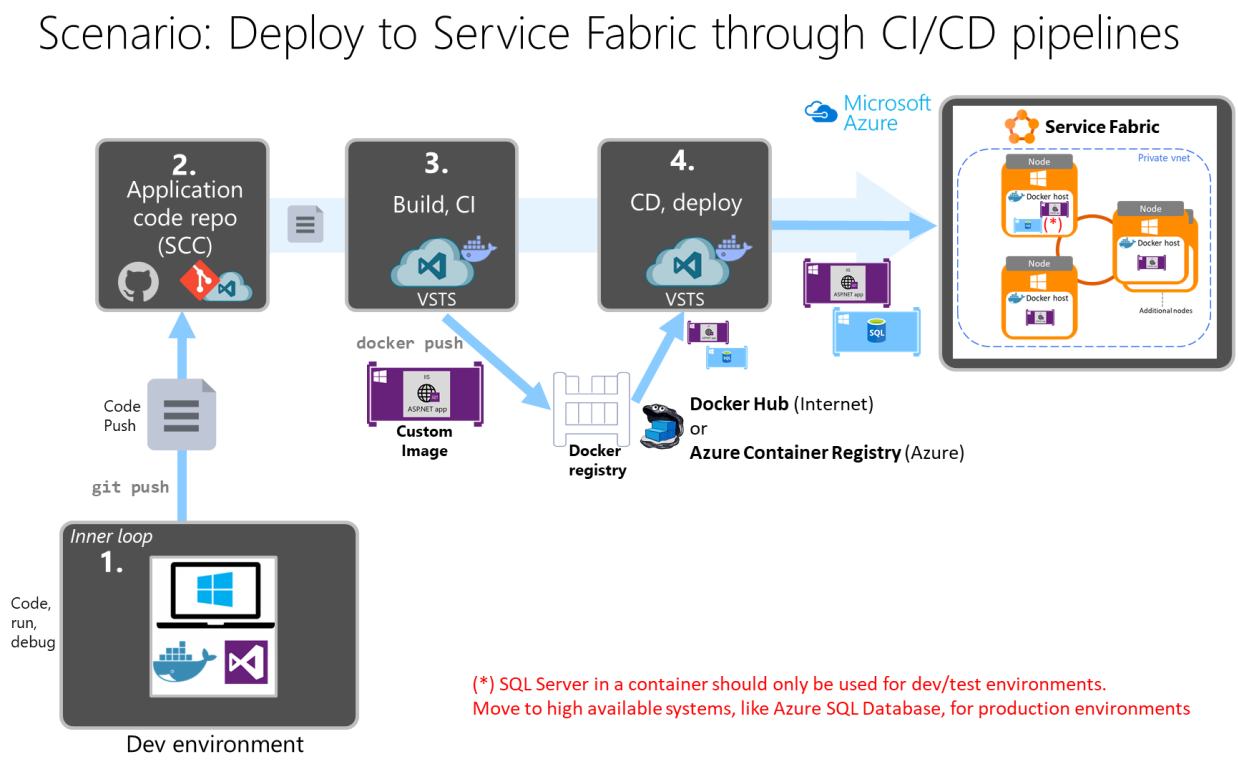
Scenarios

Scenario A: Direct deployment to Service Fabric cluster from development environment



(*) SQL Server in a container should only be used for dev/test environments.
Move to high available systems, like Azure SQL Database, for production environments

Scenario B: Deployment to Service Fabric cluster from CI/CD pipelines in VSTS



Benefits

The benefits of deploying to a cluster in Service Fabric are similar to the ones you get in Kubernetes, although Service Fabric is a very mature production environment for Windows applications compared to Kubernetes which was in Preview until mid-2017 (Kubernetes is a more mature environment for Linux).

Most of all, Service Fabric is a production-ready environment that will allow you to scale-out the application based on your desired number of container instances (inner-scalability within the existing nodes) and number of nodes/VMs in the cluster automatically with VM Scalesets in Azure (global scalability of the cluster).

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

Service Fabric allows developers to change your mindset from physical and virtual machines, to a container-centric infrastructure facilitating the following capabilities, among many others:

- Facilitates multi-container based applications
- Replicating container instances and using horizontal autoscaling
- Naming and discovering (internal DNS, etc.)
- Balancing loads
- Rolling updates

- Distributing secrets
- Application health checks

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

- Stateful Services capability by using the Reliable Services Application Model
- Actors pattern by using the Reliable Actors Application Model
- Can deploy bare-bone processes in addition to Windows or Linux Containers
- Advanced rolling updates and health checks

Next steps

Go ahead and explore the deeper technical content within the GitHub Wiki page here:

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 provide important benefits of cost savings because containers are a solution to deployment problems caused by the lack of dependencies in production environments, therefore, improving DevOps and production operations significantly.
- Docker is becoming the de facto standard in the container industry, supported by the most significant vendors in the Linux and Windows ecosystems, including Microsoft. In the future, Docker will be ubiquitous in any datacenter in the cloud or on-premises.
- A Docker container is becoming the standard unit of deployment for any server-based application or service.
- For production environments, it is recommended to use an orchestrator (like Service Fabric or Kubernetes) for hosting scalable Windows Containers-based applications.

---------- TBD SECTION IN DRAFT ----------

---------- Content TBD ----------
