Deploying .NET apps on Google Cloud

This article provides an overview of how you can deploy .NET apps on Google Cloud and provides guidance on how to choose the right deployment approach for your app.

Introduction

The Microsoft .NET framework provides a rich set of tools and libraries for app development. With the advent of Docker support on Windows and the ability to run .NET Core apps on Linux (https://docs.microsoft.com/en-us/dotnet/core/linux-prerequisites?), .NET apps are now also able to support a variety of deployment targets.

For development and testing to be efficient, you can automate app deployment and make it part of a continuous integration/continuous delivery (CI/CD) pipeline. But in order to choose the right tooling and to build a CI/CD pipeline, you must first identify how to run the app in production and which approach to deployment you want to take.

There is no single best way to deploy a .NET app on Google Cloud. The best deployment options for you depend on the app and your requirements. For example, if your app requires the full .NET Framework or must run on IIS, your deployment will be based on Windows. On the other hand, if your app can run with the functionality supported by .NET Core, you have the option of deploying under Linux.

This article looks at the various ways you can run .NET apps and deploy them on Google Cloud, including the conditions for when each option is suitable. At the end, your deployment options are summarized in a decision tree to help you decide which Google Cloud components and approaches are best for your .NET app.

Deployment models

There are two basic ways to conduct automated deployment of an app. The deployment package is either *pushed* to the app servers, or the app servers *pull* the app package from a known location. The following sections discuss differences between these two models.

Push-based deployments

In a *push-based deployment*, the deployment artifact—a zip file, a NuGet package, or another artifact—is initially available only to a deployment server. The deployment server can be a dedicated machine or a role that the CI system assumes.

To perform a deployment, a process on the deployment server connects to an app server, copies the deployment artifact, and initiates its installation. If there is more than one app server, this process is repeated in parallel or, more commonly, in sequence so that artifacts are deployed to all app servers.

The following diagram illustrates this flow.

A variety of configuration management tools are available that let you automate deployments this way. Some of these tools follow an imperative approach where the sequence of deployment steps is defined in a script-like manner. Although this approach is intuitive, it's prone to *configuration drift*—that is, after a certain amount of time, the states of multiple machines might not be identical and might not fully reflect your intended state. Many tools therefore let you define the state you want, leaving it to the tool to figure out the steps required to realize this state.

On Windows, commonly used tools for this model of deployment include:

- Microsoft Web Deploy (https://www.iis.net/downloads/microsoft/web-deploy), a free tool designed to remotely deploy web apps to IIS servers.

- Octopus Deploy (https://octopus.com/), commercial software that allows deployments to be orchestrated in a flexible manner across fleets of machines.
- Microsoft Team Foundation Server/Azure Pipelines Agents (https://docs.microsoft.com/en-us/vsts/build-release/concepts/agents/agents?view=vsts), which directly integrate with the release management functionality of TFS/Azure Pipelines.
- Windows PowerShell Desired State Configuration (DSC) (https://docs.microsoft.com/en-us/powershell/scripting/dsc/overview/overview?view=powershell-6), a built-in feature of Windows Server 2012 R2 and later versions.

Popular open source tools include Ansible (https://www.ansible.com/), Chef (https://www.chef.io/chef/), and Puppet (https://puppet.com/). Although these tools primarily target Linux, they are also capable of deploying Windows targets.

Security

For the deployment server to push a deployment to an app server, a back channel must be available. For example, Web Deploy and Octopus Deploy use a custom protocol and port for this task, while Ansible uses SSH.

Regardless of the protocol that the tool uses, it's critical that the communication is secure to help prevent attackers from using the back channel to deploy malicious apps. Most importantly, secure communication requires the deployment server to be able to authenticate with the app server.

SSH can use public key authentication. If you use appropriate Cloud IAM configuration (https://cloud.google.com/compute/docs/access/iam#the_serviceaccountuser_role), you can let Google Cloud automatically take care of distributing the public key used for SSH to the app servers. However, if you're not using Cloud IAM, Google Cloud can't manage the key for you, and you must manage this task yourself.

One option is Active Directory. When both the deployment server and the app server run Windows and are members of an Active Directory domain, authentication is handled using Kerberos (https://msdn.microsoft.com/en-us/library/aa378747(v=vs.85).aspx). However, running a fault-tolerant Active Directory environment (https://cloud.google.com/solutions/deploy-fault-tolerant-active-directory-environment), requires at least two additional VM instances in order to run domain controllers. If your configuration uses autoscaling, all the servers also need to be dynamically joined to the domain, which slows down the process of bringing up a server. Autoscaling can also lead to stale computer objects

accumulating in the directory, calling for additional scavenging logic. If you do use Active Directory in cloud-based environment, you must take these extra factors into account.

In the absence of Active Directory, authentication either needs to be handled using NTLM (https://msdn.microsoft.com/en-us/library/aa378749.aspx) or through other means, such as HTTP Basic authentication. Both approaches require credentials to be kept in sync between the deployment server and app servers and to be securely stored. Both of these tasks can prove challenging.

Whether you are using Linux or Windows, securing the communication between deployment and app servers requires mechanisms that are separate from Cloud IAM. However, using multiple mechanisms to control access to systems increases overall complexity and thereby increases the likeliness of accidental misconfiguration.

Operating system updates

It's important to be able to efficiently deploy new versions of app packages on app servers, but it's also critical to service the underlying operating system on those servers. This means installing security patches. For larger server fleets, you should automate this process in a way that minimizes risk and minimizes the number of servers that are unavailable while being updated.

You can also use a push approach to operating system updates, where the deployment server triggers an OS update on the app servers. On Linux, it's common to use SSH to remotely run update commands for this. On Windows, PowerShell remoting (https://docs.microsoft.com/en-us/powershell/scripting/learn/remoting/running-remote-commands?view=powershell-6) (which relies on WinRM) is a common choice. For both mechanisms, you must be able to securely authenticate and to securely store credentials.

Autoscaling

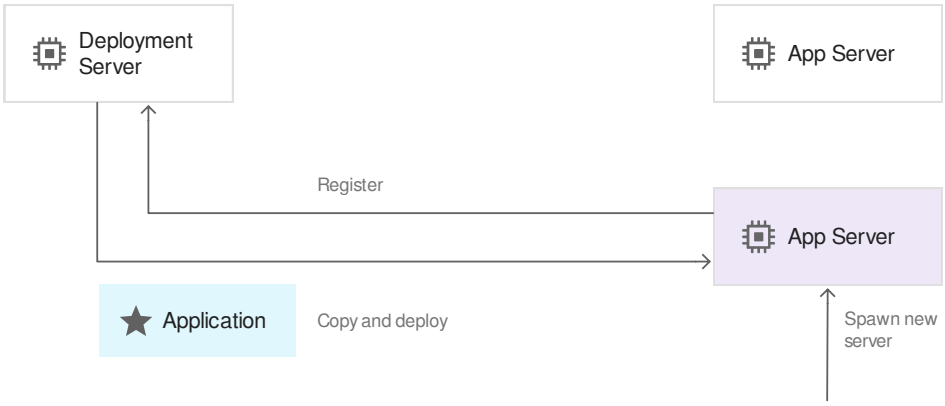
In a static environment where the number of app servers doesn't change, the deployment server knows all deployment targets in advance. In a cloud environment, it's often beneficial to autoscale the number of app servers up and down. This creates two challenges when you're using push-based deployments:

- When a new app server is added, register it with the deployment server to make sure that the new server is included in future deployments.

- The new server needs to receive its initial deployment.

An autoscaling event isn't initiated by the deployment server. Instead, it's initiated by the underlying managed instance group (https://cloud.google.com/compute/docs/instance-groups/), which functions at a level below that of the deployment server.

The new app server instance must register itself with the deployment server and trigger a deployment before the new app server can serve requests. The following diagram illustrates this process.



For this approach to work, it's not sufficient that the deployment server can contact and authenticate with app servers. The app servers also need to contact the deployment server and authenticate with it.

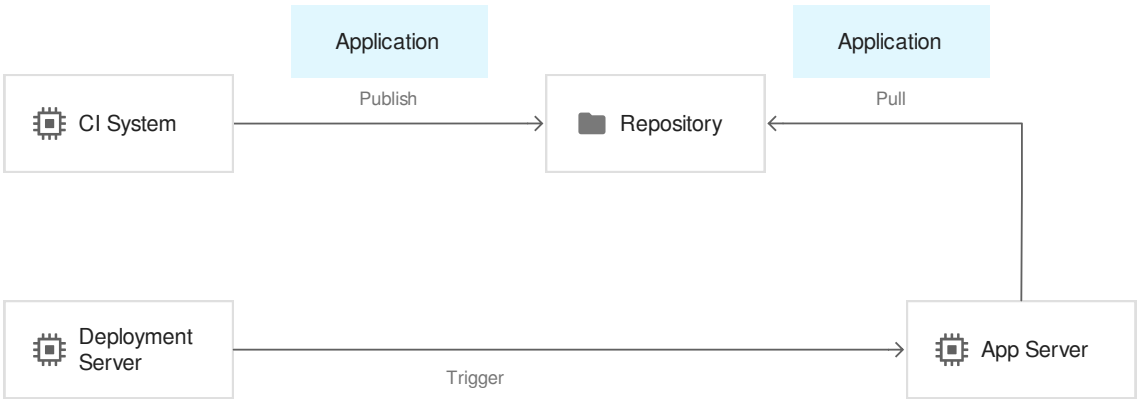
Finally, the newly launched server must also have the latest OS security patches. Initiating an update during the autoscaling process would delay the process significantly. Therefore, the image from which the app server VM is created needs to have the updates installed already. You can manage this in two ways:

- Use the OS images provided by Google Cloud, which are kept up to date by Google. Because these images contain only the OS, you must handle any customizations (your app code, utilities, and OS configurations) using startup scripts (https://cloud.google.com/compute/docs/startupscript) or as part of the app deployment.
- Maintain a custom OS image and keep it up to date. This allows you to apply customizations to the image, but it increases the overall complexity of managing your deployments.

Performing push-based deployments is intuitive, but it can result in substantial complexity when you take into account security, OS updates, and autoscaling. The next section addresses pull-based deployments, which are the more cloud-native way to approach deployments.

Pull-based deployments

In pull-based deployments, deployments are performed in an indirect manner. After the CI system has produced a new version of a deployment artifact, it publishes the artifact to a repository. The following diagram illustrates this flow.



When a deployment is performed—which might be immediately after publishing the artifact or at a later stage—the deployment server triggers the actual deployment. Again, the deployment server might be a separate system or a role that the CI system assumes. Triggering the deployment involves connecting to the app server to have it pull and install the deployment artifact from the central repository.

Although the differences between a push-based model and a pull-based model might initially seem minor, performing a pull-based deployment has a few important implications:

- Triggering an app server to pull a deployment artifact doesn't have to happen at the app or OS level. Instead, the deployment server can trigger the pull operation by having Compute Engine restart or replace the VM. This can avoid the security challenges associated with push-based deployments.
- Rather than merely containing app files, the deployment artifact can be a Docker image or a VM image, which can unify the process of applying app and OS updates.

Security

The deployment server doesn't need to interact with the app server at all for certain kinds of deployments. For example, no interaction is necessary if the deployment artifact is any of the following:

- A VM image.
- A Docker image to be deployed to Google Kubernetes Engine.
- A package to be deployed to App Engine.

Instead, the deployment server just needs to interact with the Google Cloud APIs to initiate the deployment. This in turn means that the deployment process can rely on authentication mechanisms provided by Cloud IAM (https://cloud.google.com/docs/authentication/production), which removes the need to manage keys or credentials.

When you use deployment artifacts such as zip or NuGet packages, which contain only the app files and binaries, you can trigger a deployment in these ways:

- If the server is configured to pull and install the latest deployment artifact when the operating system starts, you can trigger an update by having Google Cloud restart the VM. Although a restart might seem unnecessarily time consuming, this avoids the need to have the deployment server authenticate with the app server.
- As with push-based deployments, the deployment server can remotely trigger the update via a back channel. However, this approach is subject to the same security implications and challenges of managing credentials that apply to push-based deployments.
- The deployment server can run an agent that observes the repository for new deployment artifacts. When a new artifact is detected, the server can apply it automatically. A potential issue is that multiple app servers could end up installing updates concurrently and thus be unavailable. To avoid this, the agent can track server state in the repository and use this server state information to roll out updates in a controlled manner.

In each of these cases, make sure that you control write access to the repository in order to prevent servers from pulling and installing malicious packages.

Operating system updates

When Docker or VM images are used as deployment artifacts, these artifacts combine app files and dependencies. This allows you to use the same deployment mechanism for updating the

operating system and for updating the app. In this case, you should make sure that a new deployment artifact can be built and published for two separate cases. One is when a new app version becomes available. The second is when new security updates to the operating system or other dependencies are released.

In other cases, where the deployment artifact contains only the app files, it's a separate task to keep the operating system up to date. Therefore, the same implications discussed in the context of push-based deployments apply.

Autoscaling

Having app servers pull deployment artifacts aligns well with the idea of autoscaling, and it avoids much of the complexity that arises from combining autoscaling with push-based deployments. Whenever a new app server is launched due to an autoscaling event, the server contacts the repository and pulls and installs the latest deployment package.

If you're using VM or Docker images, the mechanisms for having images pulled are provided by GCP. If you're using other packages such as zip or NuGet archives, you must configure app servers to initiate a deployment after startup. You can do this either by customizing the VM image or by using startup scripts.

Deployment targets

Historically, .NET apps ran only on Windows, and Windows did not support containers. This left you little choice about what environment to run your app in.

With the advent of .NET Core, you can decide between running an app on Windows or on Linux. And because both operating systems support containers, you now have multiple choices about which environment to target.

Operating system

Although Mono has offered a way to deploy .NET apps on platforms other than Windows for many years, it was not until the release of .NET Core that Linux became a fully supported platform for the Microsoft development stack.

.NET Core provides only a subset of the capabilities of the .NET framework. Therefore, targeting .NET Core imposes certain restrictions on apps. More importantly for existing apps, porting from .NET Framework to .NET Core might not always be easy and cost effective; in certain cases, it might not be possible at all.

Therefore, a fundamental question when choosing a deployment model and target is whether to use Linux (which requires .NET Core) or Windows (which supports either .NET Core or the .NET Framework).

The potential benefits of running .NET apps on Linux include the following:

- You can use App Engine flexible environment (https://cloud.google.com/appengine/docs/flexible/dotnet/), a fully managed environment.
- You can use GKE (https://cloud.google.com/kubernetes-engine/), a managed environment that supports container orchestration.
- You can avoid the extra cost of premium (https://cloud.google.com/compute/pricing#premiumimages) Compute Engine (https://cloud.google.com/compute/) images that are associated with Windows licensing.

You must weigh these benefits against the following potential downsides of using .NET Core on Linux:

- The effort required to port an existing .NET app to .NET Core might offset the potential cost savings. Or as noted, it might simply not be possible to port an existing .NET app to .NET Core.
- Linux does not support IIS. Kestrel (https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?view=aspnetcore-2.1&tabs=aspnetcore2x#kestrel), the .NET Core web server, exhibits very good performance, but it does not offer the same feature set as IIS. You might therefore have to use Kestrel in conjunction with a web server like Nginx.
- There is no direct equivalent of a Windows Service on Linux. Although you can usually convert Windows services to Linux console apps that can run as a daemon, this conversion might not always be easy.
- Troubleshooting and debugging .NET Core apps on Linux requires different tools and skills than when you use .NET on Windows. This can prove challenging if your team has limited experience with Linux.

Containers

Containers lend themselves particularly well to apps that run in a single process. Examples include:

- Windows services
- Linux console apps acting as daemons
- Self-hosted WCF services
- Kestrel-hosted ASP.NET MVC or Web API apps

Many .NET apps target IIS. It's commonly used to manage multiple apps (in separate virtual directories and app pools) and might therefore not match the single-process pattern.

When you move an IIS-based setup to a container, you can take different approaches:

- Put IIS, with all virtual directories and pools, into a single Windows-based Docker image using the `microsoft/iis` image as the base. Unless the apps are tightly coupled, this approach is usually not advisable, because it doesn't allow apps to be updated and deployed separately.
- Use separate Windows-based Docker images for each app, each running IIS. This ensures that you can manage apps independently. However, IIS incurs a non-negligible overhead that can become significant if you need to operate a large number of these containers.
- Migrate some or all apps from IIS to Kestrel (https://docs.microsoft.com/en-us/aspnet/core/fundamentals/servers/?view=aspnetcore-2.1&tabs=aspnetcore2x#kestrel). Because Kestrel can be deployed in either a Windows-based container or a Linux-based Docker container, this approach allows you to manage containers individually.

IIS allows multiple web apps to run under a single web site, sharing a single domain name. When you package apps into separate containers, you can get the same functionality by using content-based load balancing (https://cloud.google.com/compute/docs/load-balancing/http/content-based-example). In a similar vein, a Google HTTP load balancer makes it unnecessary to deploy a custom reverse proxy in front of Kestrel servers.

Most apps can be containerized; it's rare to have one that can't be. But some containerization scenarios present challenges:

- For IIS-managed apps, it's common for the app deployment to already be automated. But the steps to configure IIS (creating app pools, bindings, and so on) are carried out manually. When you move to containers, you have to automate all of these initial steps as well.
- Apps that rely on configuration files or on data that's located on disk might require changes. For example, configuration information can be obtained from environment variables, and relevant files and folders can be mounted as a volume. This keeps the image stateless and free of environment-specific configuration.

Finally, if you use Windows-based Docker containers, note that Google Cloud currently doesn't support Hyper-V and doesn't let you run Hyper-V containers. Therefore, you can only Windows Server containers in Google Cloud. Windows Server Containers are more lightweight than Hyper-V containers, but Windows Server containers offer slightly weaker isolation.

Deployment constraints

Certain factors in how your app is built can impose constraints on what deployment approach you use, as discussed in this section.

App architecture

Another factor to consider when choosing the deployment target and model is the architecture of the app. At one end of the spectrum, an app might follow a monolithic architectural pattern, where all app logic is implemented in a single code base and runs in a single process or IIS app pool. At the other end of the spectrum, an app might follow a microservices pattern. In this approach, the app consists of a number of services that run independently in separate processes, in separate IIS app pools, or as separate Windows services.

Finally, you might have multiple, independent apps that are deployed using a uniform deployment strategy, where each app itself might be monolithic. For the purposes of this discussion, this approach can be considered equivalent to the microservices scenario.

In a microservices architecture, you want the app to run cost effectively while keeping services isolated and independently manageable. You can allocate dedicated VMs for each service, which guarantees that services can be managed and deployed individually. But this approach can result in a large number of underutilized VMs, incurring unnecessary cost. For apps like

these, deployment models that allow tighter *packing*—in particular, container-based models—are therefore likely to be more cost effective.

State and statelessness

When you design apps for the cloud, try to keep apps stateless and manage state externally using a GCP-based storage service. Stateless apps offer a number of advantages, including:

- They can be deployed redundantly to increase availability and capacity.
- Requests can be freely distributed among instances.
- They lend themselves well to autoscaling.
- In case of failure, the environment (whether container or VM) can just be recreated without the risk of data loss.

Designing apps to be stateless is not always easy, and many older apps don't follow this practice. Still, it's worthwhile to analyze whether you can make an app stateless.

Session state

ASP.NET and ASP.NET MVC apps commonly use sessions to track user state, making the app stateful. However, there are multiple options to limit the impact of sessions:

- If the amount of session data is small, you can store state in an encrypted or signed cookie instead.
- Rather than using the default `InProc` session state provider, you can use the `SQLServer`provider. However, this requires a SQL Server instance, which incurs additional cost and can impact latency and availability of the app.
- You can take advantage of session affinity (https://cloud.google.com/compute/docs/load-balancing/http/#session_affinity) in Cloud Load Balancing. This feature lets you ensure that all requests from a single client are routed to the same app instance. However, using session affinity can have a negative impact on the fairness of load balancing; that is, certain app instances can end up receiving more requests than others. In addition, if an app instance is terminated for any reason, any sessions handled by the instance will be lost, potentially causing end user impact. Relying on session affinity is therefore not an ideal solution, but it can often be a viable compromise between robustness and cost.

In-memory caches

Apps commonly use in-memory caches to avoid redundant calculations or database lookups. This becomes problematic if multiple instances of the app are running concurrently, because caches can become incoherent.

To avoid incoherencies, use a distributed cache, either directly or by using the `IDistributedCache` interface. Caching servers such as Redis (https://cloud.google.com/memorystore/) or Memcached usually have relatively low resource demands, but they do add complexity to the overall setup.

Storage

Data in the form of images, attachments, or media files is typically stored on disk. Using a persistent disk on a VM for this purpose is usually not an option, because it prevents data from being shared among multiple machines, and it risks data loss if a VM instance is recreated. Instead, you can use one of the following approaches:

- Move the data to a file share server. This minimizes the impact on the app. However, operating a highly available SMB or NFS server implies additional cost and maintenance effort.
- Move the data to Cloud Storage. Although this requires changes to the app, Cloud Storage is highly available, substantially more cost efficient than running a file server, and requires no additional maintenance work.

Deployment strategies

When you deploy a new version of an app, you must minimize risk and end user impact. The three most common strategies to achieve this are *Recreate*, *Blue/Green*, and *rolling deployments*.

Recreate strategy

The idea of the Recreate strategy is to stop the running app on all servers, deploy a new version, and start the app. This strategy has the obvious drawback of causing a service interruption, but

it avoids potential issues that can arise when two different versions of an app temporarily coexist and access common data.

Blue/Green strategy

The idea of the Blue/Green strategy (also referred to as *Red/Black*) is to deploy a new app version on a new set of servers. When the deployment is completed, you switch all traffic from the old to the new set of servers. This approach temporarily requires up to twice the number of servers as you need for production, but it avoids service interruption.

A prerequisite for this strategy is that two versions of an app can temporarily coexist and not interfere with each other. For apps that access databases, this requires that each iteration of changes to database schemas has to be backward compatible to at least the previous version.

Rolling deployments strategy

The idea of a rolling deployment is to update one server after another. As with the Blue/Green strategy, this means that for a certain time, two different versions of an app coexist. Unlike the Blue/Green deployment, however, you shift traffic from the old to the new version gradually. As more servers are updated, more users are routed to the new version until finally, when the last server has been updated, all users use the new version. A key benefit of this approach is that potential issues can be detected early, before all users are affected, which helps to lower the overall risk.

Because rolling deployments require two versions of the app to coexist, this strategy often also requires a load balancer configuration that avoids bouncing users between versions.

Deployment options

Up to now, this article has discussed deployment models, targets, and strategies. The following sections look at specific options for deploying .NET apps on Google Cloud.

App Engine flexible environment (Linux)

App Engine flexible environment provides a platform-as-a-service (PaaS) environment for .NET Core apps. Because App Engine flexible environment is based on Linux, it's useful only for .NET

Core apps.

App Engine flexible environment internally uses containers to run and scale apps, but frees you from having to build or manage container images. Instead, the app binaries can be deployed directly. By using services (https://cloud.google.com/appengine/docs/standard/python/microservices-on-app-engine#app_engine_services_as_microservices), App Engine flexible environment allows you to run apps that are decomposed into a number of smaller microservices.

App Engine flexible environment can automatically scale the number of app instances depending on load, heeding any limits that you have configured for the app. By default, a minimum of two instances is maintained, although you can change (https://cloud.google.com/appengine/docs/flexible/python/reference/app-yaml#automatic_scaling) this.

App Engine flexible environment is most suitable for stateless apps. Although you can disable autoscaling to accommodate stateful apps, doing so means that you're not getting many of the benefits of the managed environment. In addition, although App Engine flexible environment permits disk access, disks are considered ephemeral and are therefore not useful for tracking persistent state.

For each instance of an app, App Engine flexible environment maintains a dedicated VM. Because pricing is based on the number of running VMs, App Engine flexible environment is most cost effective when the app is heavily utilized. However, when apps are accessed infrequently, the underlying VMs might be poorly utilized, which in turn can make App Engine flexible environment less cost effective than other deployment options, particularly GKE.

Pull-based deployment using the gcloud command-line tool

The most common way to deploy to App Engine flexible environment is to use the gcloud command-line tool. The tool is first used to publish deployment artifacts to a repository maintained in Cloud Storage. Whenever an instance is launched, the artifacts are pulled from this repository. Deployments use the Blue/Green strategy by default.

Each deployment is tracked by a version number. In case of problems, a deployment can be reverted to a previous version. Through traffic splitting (https://www.google.com/url?q=/appengine/docs/standard/python/splitting-traffic&sa=D&ust=1523867182934000&usg=AFQjCNG6Fe4WhSoZCN98GmATMnYb_XyP6w), App Engine flexible environment also lets you run multiple versions of the app in parallel and

to direct a certain share of the traffic to either version. This allows you to conduct canary deployments or A/B tests with minimal effort.

GKE (Linux)

GKE provides a fully managed Kubernetes environment. The orchestration capabilities of Kubernetes make GKE particularly well suited for running complex microservices apps that consist of many containers. However, even for apps that don't follow the microservices pattern, GKE lets you run many containers on shared infrastructure in a way that's both resource efficient and simple to maintain.

GKE requires all parts of the app to be packaged as Docker containers. Because GKE currently doesn't support Windows containers, these containers must be Linux based. This in turn requires the use of .NET Core and a Linux-based environment for building containers. Building containers in Linux can prove challenging if your CI system is Windows-based. However, both Azure Pipelines/Team Foundation Server and Cloud Build (https://cloud.google.com/cloud-build/) provide built-in support for building .NET Core apps and for building and publishing Linux-based container images.

GKE offers the most flexibility for apps that are stateless. By using stateful sets and persistent volumes, you can also run certain kinds of stateful apps on GKE.

A GKE cluster includes a number of VM instances, called *nodes*, on which containers are scheduled. In a multi-zone or regional cluster, GKE can spread nodes and workloads over multiple zones to ensure high availability.

Pricing is based on the number of nodes that are running. GKE is therefore most cost effective when nodes are well used. You can run larger workloads on the same cluster or by automatically scaling the number of nodes (https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler)as needed.

Pull-based deployment using kubectl commands

Deploying an app to GKE entails two steps:

1. Publishing Docker images to Container Registry (https://cloud.google.com/container-registry/) or to an external Docker registry using `gcloud docker push` or other means. This step is usually handled by the CI system.

2. Triggering the deployment using `kubectl`. This step can either be handled by the CI system or separately. Because the deployment is initiated remotely, it doesn't matter if `kubectl` is run on Linux or Windows.

GKE has built-in support for the recreate and rolling deployment strategies. Although the primitives to control deployments are flexible enough to allow other deployment strategies, using a different strategy requires additional tooling or scripting.

Pull-based deployment using Spinnaker

If the built-in capabilities of GKE for orchestrating deployments are insufficient for your purpose, you can combine GKE with Spinnaker. Spinnaker has built-in support for GKE and allows you to implement more advanced deployment strategies, including Blue/Green deployments.

Because Spinnaker is not a managed service, you have to deploy and maintain it separately. You can deploy Spinnaker either on separate Linux VM instances (https://console.cloud.google.com/marketplace/details/click-to-deploy-images/spinnaker) or in a GKE cluster (https://cloud.google.com/solutions/continuous-delivery-spinnaker-kubernetes-engine).

Compute Engine (Windows or Linux)

Compute Engine lets you create and manage VM instances. It supports a range of Windows Server versions and Linux distributions, plus sizing, and configuration options. Given this flexibility, you can use Compute Engine VM instances for a wide range of workloads.

To ensure that apps are deployed and maintained individually, deploy only a single app or service for each VM instance. To ensure high availability, run at least two VM instances per app, each located in a different zone. You can therefore assume that you need twice the number of VM instances as the number of apps or services you want to deploy, regardless of the expected load.

Compute Engine provides a simple way to implement autoscaling through managed instance groups. Managed instance groups also provide a way to implement rolling deployments, as discussed later in this article.

Because Compute Engine is priced by VM instance, you can assume that running apps on Compute Engine is most cost effective when apps receive considerable load, which translates into high utilization of the VM instances. In contrast, if the number of services and apps is large

but average utilization is low, other deployment options such as GKE are often more economical, because they allow multiple apps to use common infrastructure without sacrificing workload isolation.

Running Windows VM instances requires you to use premium images (https://cloud.google.com/compute/pricing#premiumimages). These images contain licensed copies of Windows and therefore incur additional fees. As a result, Windows VMs are generally less cost effective than VMs that use Linux distributions such as CentOS or Debian, which do not incur any license fees.

You can use SSH or RDP to manually set up a VM instance, either to deploy an app manually or to handle any initial configuration needed to prepare a machine for a first deployment. However, this can lead to machines that have unique configurations, differing from other VM instances. In the long run, manually setting up a VM instance can become complicated and labor intensive. It's therefore advisable to automate the process in order to make it repeatable.

Automating app deployments on Compute Engine includes these tasks:

1. Provisioning and preparing VM instances for a first app deployment.
2. Performing an app deployment.
3. Servicing the OS (installing security updates).

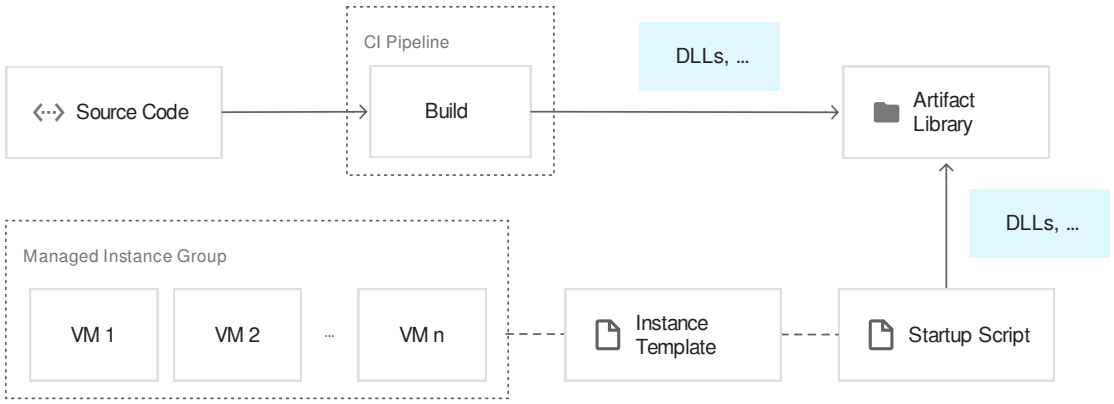
The following two sections describe how you can handle all three steps in a unified fashion using a pull-based deployment approach. While the mechanisms and tools differ for the approaches described in these sections, the general idea is similar to how a container-based app is deployed using GKE.

Pull-based deployment using a managed instance group

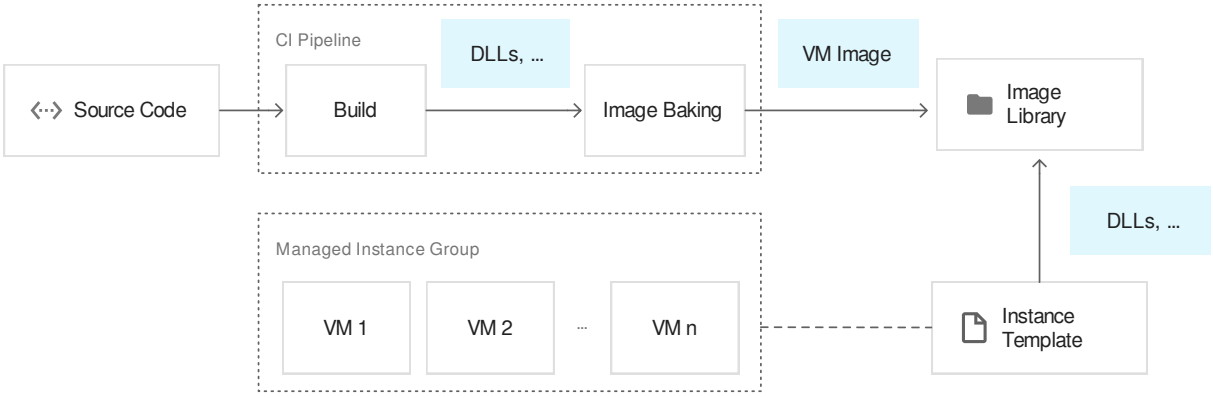
Managed instance groups are most commonly used to implement autoscaling, but they also provide a way to handle rolling deployments. After an instance template has been created that refers to the new version of the app, you can use the rolling replace (https://cloud.google.com/compute/docs/instance-groups/updating-managed-instance-groups?hl=en_US&_ga=2.29217404.-157260409.1512652371#starting_a_basic_rolling_update) functionality to replace VM instances that use the old template with instances that use the new template.

A prerequisite for this approach is that the new version of the app is being made available as an instance template. You can accomplish this in two ways:

- Define an instance template that uses one of the public OS images (https://cloud.google.com/compute/docs/images#os-compute-support). Use a startup script to configure the system and to install the app from a Cloud Storage bucket, a NuGet repository, a Docker registry, or another source. The following diagram illustrates this approach.



- Create a custom VM image as part of the CI/CD process, a process that's often referred to as *image baking*. In this approach, you use one of the public OS images to spawn a new VM instance, install the latest app on it, create a VM image from the instance, and make the image available in the Google Cloud project. The entire process can be fully automated using a tool like Packer (https://www.packer.io/). The resulting image can then be referenced in an instance template. The following diagram illustrates this approach.



A drawback of creating a custom image (the second option) is that image baking is a relatively slow process, often taking several minutes. The approach therefore not only adds complexity to

the CI/CD process, but also slows the CI/CD process down. On the upside, launching new VMs using a custom image is a simple and fast process, which is beneficial when you use autoscaling.

Using startup scripts to deploy the app (the first option) has the opposite tradeoffs. It doesn't incur the overhead of image baking in the CI/CD process, but it slows down the process of creating VM instances. Furthermore, if the startup script is not fully reliable or if the systems that the app binaries are downloaded from are not highly available, this approach can result in lower availability.

The approach that's most suitable to your app depends on the app itself and the complexity of the configuration. In some scenarios, it may even be best to combine both approaches:

- A custom image contains all configuration and dependencies, but not the actual app binaries. A new image is baked when the configuration or any of the dependencies change, but not for every app build. This helps avoid the slowdown of the app CI/CD pipeline.
- The app is installed using a startup script. To minimize risk and slowdown, this process should be as simple as possible.

In a scenario where you want to deploy many different apps or services that have a common base configuration, this hybrid approach can avoid having to build and maintain dozens or hundreds of almost identical images.

You can use managed instance groups to orchestrate deployments for both Linux and Windows workloads. For Linux, using managed instance groups to deploy Docker containers on VM instances is possible and supported by the platform (https://cloud.google.com/compute/docs/containers/deploying-containers). But it's advisable only for heavily utilized apps. In other cases, deploying a single Docker container per VM provides little advantage over using GKE or App Engine flexible environment.

As noted earlier, Windows is currently not supported by GKE. If you use Windows Server containers, follow these guidelines for running the containers using Compute Engine and managed instance groups:

- Use either the public `Windows Server 1709 Datacenter Core for Containers` image or a custom-built image with Docker preinstalled.
- Use a startup script to pull the Docker image and start it as a Windows Server container during VM startup. You can use appropriate port mappings to expose the services that are

running inside the container.

Note that a startup script is not guaranteed to run only after the Docker service has been started. To gracefully handle the case where the script runs before Docker is available, incorporate appropriate retry logic into the script.

When you create Windows-based images in a non-cloud environment, you might rely on Microsoft Deployment Toolkit (MDT) (https://docs.microsoft.com/en-us/sccm/mdt/) or Windows Deployment Services (WDS) (https://msdn.microsoft.com/en-us/library/windows/desktop/dd379586%28v=vs.85%29.aspx?f=255&MSPPError=-2147217396). However, because managing images and creating VM instances based on custom images are core features of Compute Engine, this additional tooling is not necessary. Compute Engine supports not only startup scripts, but also specialization scripts for Windows-based VM instances. Therefore, it's not usually necessary to work with custom `unattend.xml` files. However, it's still important for a Windows installation to be *generalized* using GCESysprep (https://cloud.google.com/compute/docs/instances/windows/creating-windows-os-image) before you create an image.

Pull-based deployment using Spinnaker

Managed instance groups provide a lightweight and robust way to implement rolling deployments, but the capabilities of managed instance groups might be insufficient for certain apps. To implement more sophisticated deployment strategies and pipelines, you can use Spinnaker.

The basic approach taken by Spinnaker to orchestrate deployments on Compute Engine is similar to the one discussed in the previous section—that is, it also relies on image baking. Therefore, the same considerations apply.

Because Spinnaker isn't a managed service, you have to deploy and maintain it separately from the app. You can deploy Spinnaker either on separate Linux VM instances (https://console.cloud.google.com/marketplace/details/click-to-deploy-images/spinnaker?project=jpassing-188008&organizationId=433637338589) or in a GKE cluster.

Push-based remote deployment

The pull-based deployment options discussed in previous sections offer a range of benefits. But they aren't appropriate for every kind of app. In particular, stateful apps often don't lend themselves well to this approach and might be better suited to a push-based approach.

In the push-based approach, the three deployment tasks (provisioning VM instances, performing the app deployment, and servicing the OS) need to be handled individually. It's possible to use the same tooling for all three tasks, but it's not uncommon to use different tools for each task.

You can provision the app server VM instances in the same manner as other infrastructure—common automation tools for this purpose include Deployment Manager (https://cloud.google.com/deployment-manager/) and Terraform. You can use startup or specialization scripts to install the tools that are required to automate the app deployment. For example, if you use Puppet, Chef, or Octopus Deploy, you must make sure that the agent software for these tools is installed.

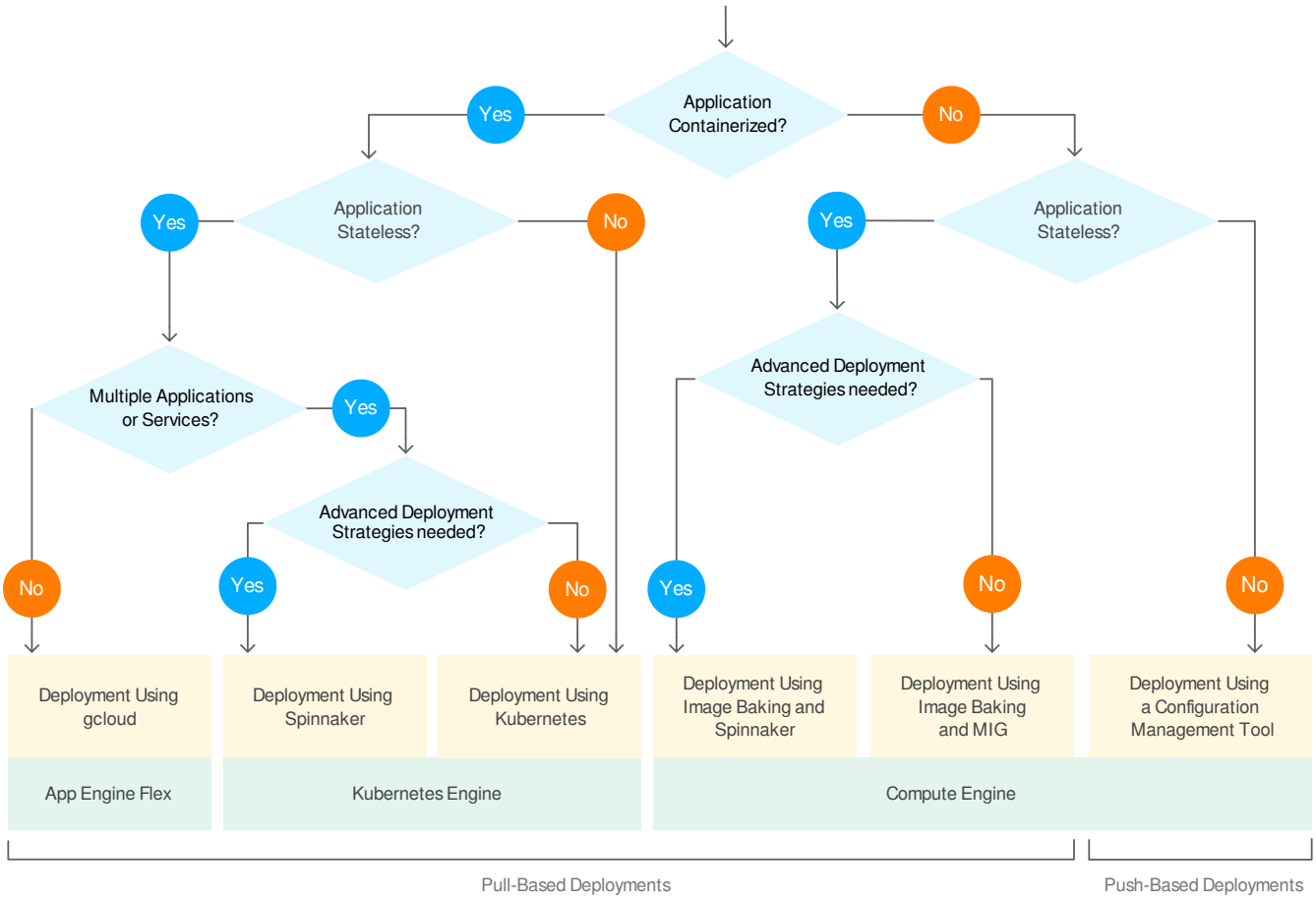
From a security perspective, to reduce the attack surface, ensure that any communication between the deployment server and any agents running on the app server VM instances uses the internal network. In addition, make sure that the ports being used are not exposed to the public internet.

In an environment where autoscaling is not used, joining Windows-based app servers to an Active Directory domain is a viable way to centralize configuration. Using Active Directory also lets you control management tasks such as OS servicing.

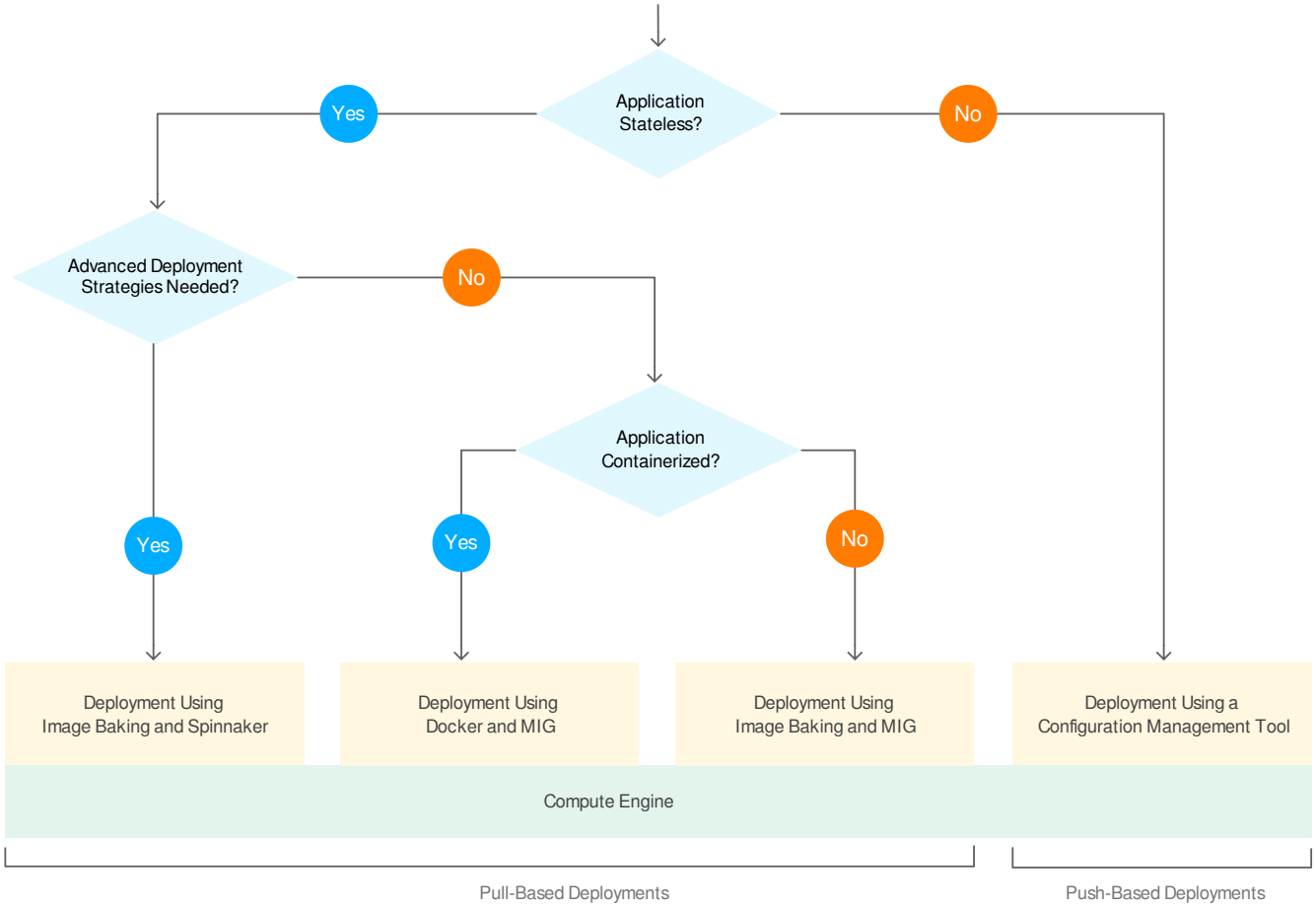
Choosing a deployment option

As noted at the beginning of this article, there is no single best way to deploy a .NET app on Google Cloud. The best deployment options for you depend on the app and your requirements. To pick the right model, one of the first questions is whether to use .NET Core or .NET Framework and, depending on this, whether to deploy on Linux or Windows. After you've identified the target operating system, use the following decision trees to help identify a suitable deployment model.

For deploying .NET Core apps on Linux:



For deploying a .NET Core or .NET Framework app on Windows:



What's next

- Learn how to create a CI/CD pipeline for a .NET Core app with Azure Pipelines and GKE (https://cloud.google.com/solutions/creating-cicd-pipeline-vsts-kubernetes-engine) or how to create a CI/CD pipeline for a .NET Framework app with Azure Pipelines and Compute Engine (https://cloud.google.com/solutions/creating-cicd-pipeline-vsts-compute-engine)
- Read more about .NET on Google Cloud (https://cloud.google.com/dotnet/docs/)
- Install the Tools for Visual Studio (https://cloud.google.com/visual-studio/), which allow you to interact with Google Cloud from within Visual Studio
- Learn more about the App Engine .NET Flexible Environment (https://cloud.google.com/appengine/docs/flexible/dotnet/)
- Try out other Google Cloud Platform features for yourself. Have a look at our tutorials (https://cloud.google.com/docs/tutorials).

Last updated November 21, 2019.