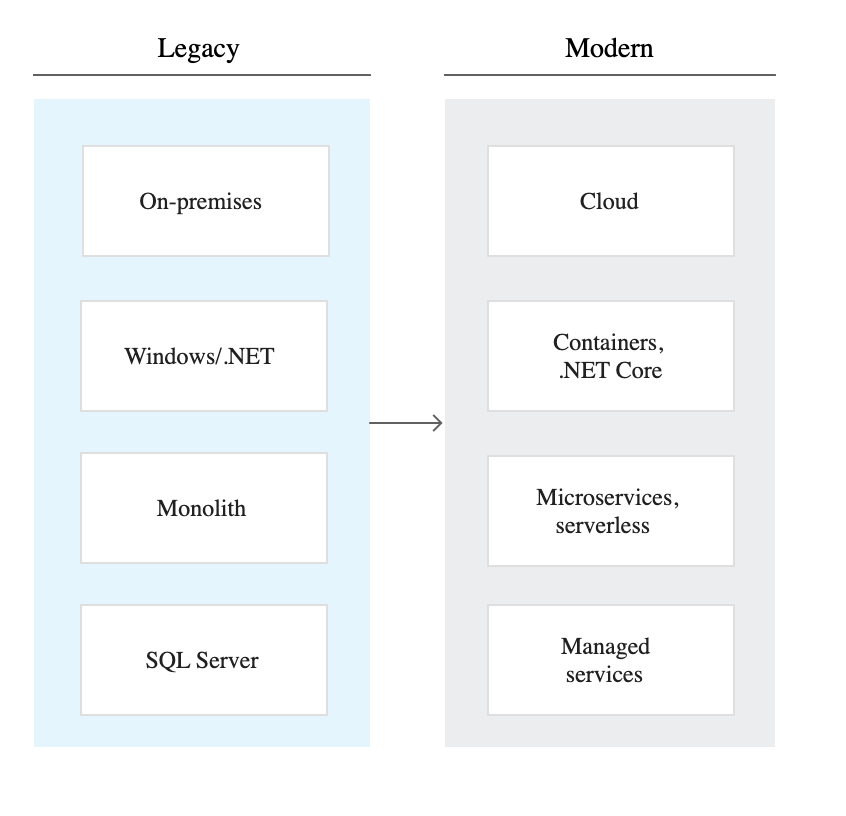
# Workloads on Google Cloud

**Legacy versus modern applications: Why modernize?**

Modernizing legacy applications is a journey. Where you begin and end that journey, and what benefits you gain, largely depend on the state of your applications and the time and effort you can invest to modernize.

In the context of .NET applications, what is *legacy* and what is *modern*? This question is not easy to answer exhaustively or definitively. Every application has distinct legacy and modernization needs. However, legacy applications share some common limitations.

The following diagram summarizes the characteristics of legacy applications and modern cloud-based applications.

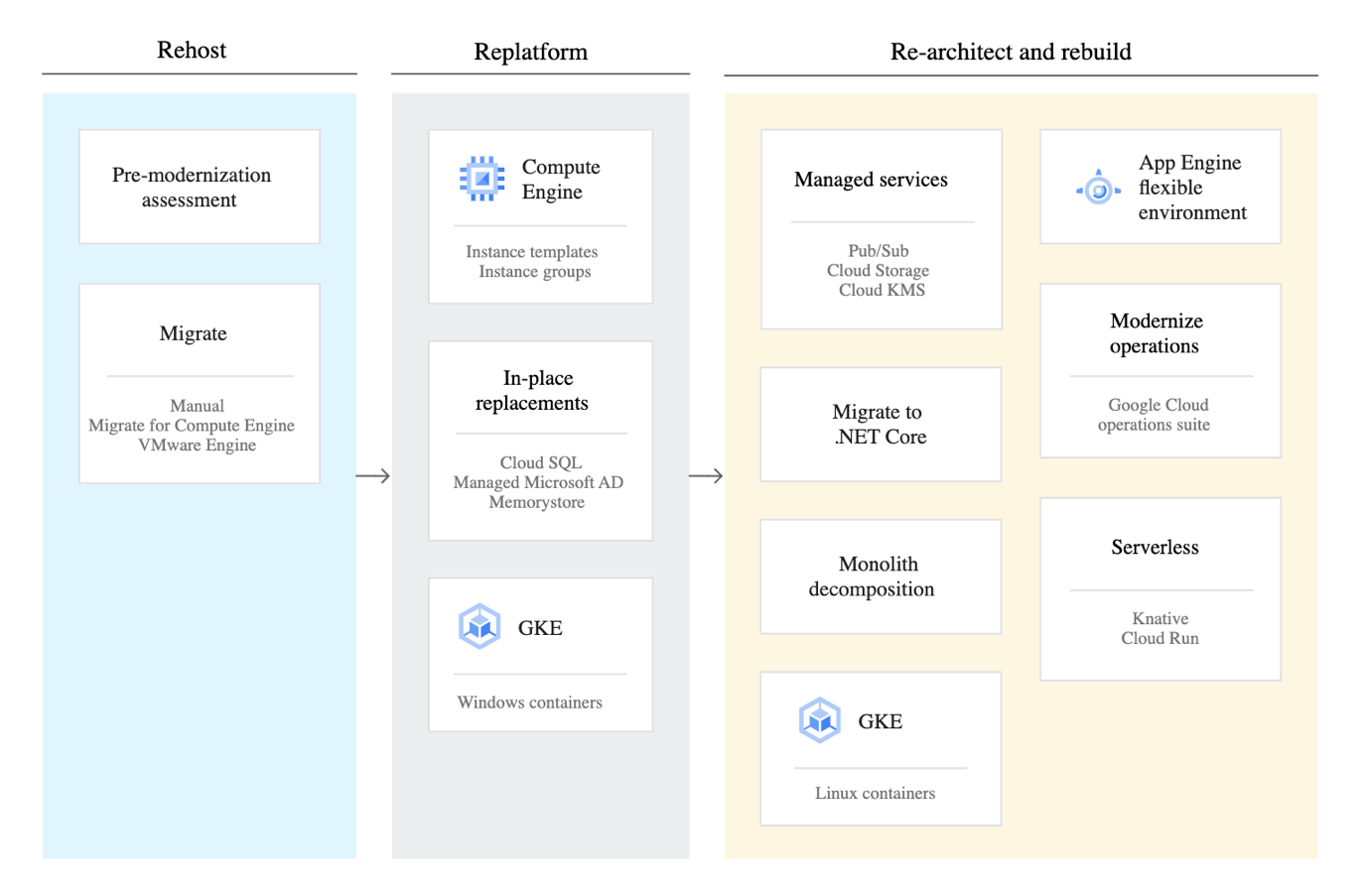


A legacy application is typically a [monolith](https://wikipedia.org/wiki/Monolithic_application) that is built on the .NET Framework, hosted on an on-premises Microsoft Windows server, and connected to an on-premises server running Microsoft SQL Server. The details of your architecture might differ from these general characteristics, but most monolithic applications have the following limitations:

Compared to the conventional on-premises approach, a cloud architecture offers a more cost-effective, efficient, and resilient way to run your applications. In a cloud-based approach, you have more flexibility to choose where and when to deploy your applications.

**Modernization path**

While the benefits of a cloud-based architecture are clear, the path to the cloud might not be. Modernization from a legacy .NET architecture to a cloud-based architecture does not follow a single, one-size-fits-all pattern. As the following diagram shows, modernization involves a series of steps, where each step removes a limitation, elevates application capabilities, and opens up opportunities for later phases of modernization.



The steps to modernization are grouped into three phases:

1. Rehost in the cloud (also known as *lift and shift*)
2. Replatform
3. Re-architect and rebuild

**Pre-modernization assessment and learning**

After you identify what applications to modernize, you can begin migrating your conventional applications to the cloud as-is or with minimal changes to the application code or configuration.

**Phase 1: Rehost in the cloud**

The primary goal of this first phase is to transfer the burden of server management from your on-premises resources to the cloud infrastructure. In this phase, you ensure that your infrastructure is cloud-ready so that you can optimize it for the cloud in later phases.

**Manual migration versus tool-based migration**

A lift and shift of Windows-based .NET applications typically starts by moving on-premises Windows Server and SQL Server instances to Compute Engine virtual machine (VM) instances. You can perform this process manually or automate it with the help of a migration tool.

In a manual migration, you can use Compute Engine [Windows Server images](https://cloud.google.com/compute/docs/instances/windows#windows_server) to start instances. [Google Cloud Marketplace](https://cloud.google.com/marketplace) also has solutions ready to deploy to Compute Engine, such as the [ASP.NET Framework solution](https://console.cloud.google.com/marketplace/details/click-to-deploy-images/aspnet) to get a Windows Server VM that includes IIS, SQL Express, and ASP.NET.

Similarly, you can start SQL Server instances from [SQL Server images](https://cloud.google.com/compute/docs/instances/windows#sql_server), or you can go directly to a more managed solution—[Cloud SQL for SQL Server](https://cloud.google.com/sql).

Google Cloud also offers migration tools such as [Migrate to Virtual Machines](https://cloud.google.com/migrate/compute-engine) or [VMware Engine](https://cloud.google.com/vmware-engine) to help you move on-premises VMware VMs to a VMware environment in Google Cloud.

After you configure the VMs, you typically create custom VM images so that you can recreate new instances on demand. This step is also important for instance templates, which are discussed [later in this document](https://cloud.google.com/architecture/modernization-path-dotnet-applications-google-cloud#take_advantage_of_compute_engine).

If you need domain services in the cloud, you can deploy a [Fault-tolerant Microsoft Active Directory environment](https://cloud.google.com/compute/docs/tutorials/setup-active-directory) on Compute Engine with a Virtual Private Cloud (VPC) or directly go to [Managed Service for Microsoft Active Directory](https://cloud.google.com/managed-microsoft-ad/docs/overview).

**On-premises and cloud connectivity**

As you migrate VMs to the cloud, it's not uncommon to keep some workloads on-premises—for example, when you have applications that require legacy hardware or software, or when you need to meet compliance and local regulatory requirements. You need a VPN or an interconnect solution to securely connect on-premises and cloud resources. For various ways to create and manage this connection, as well as other implications of running hybrid-cloud and on-premises workloads, see [Migration to Google Cloud: Building your foundation](https://cloud.google.com/solutions/migration-to-google-cloud-building-your-foundation).

**Initial benefits**

At the end of Phase 1, you have basic infrastructure running in the cloud, which provides benefits such as the following:

* **Cost optimizations.** You can create a [custom machine type](https://cloud.google.com/compute/docs/instances/creating-instance-with-custom-machine-type) (CPU, memory, and storage) and pay for what you use; start and stop VMs and disaster recovery environments at will and only pay when they are running; and get [rightsizing recommendations](https://cloud.google.com/migrate/compute-engine/docs/4.9/concepts/planning-a-migration/cloud-instance-rightsizing) before migration.
* **Increased operational efficiency.** You can attach persistent disks to VMs and create snapshots for simplified backup and restore.
* **Increased reliability.** You no longer need to schedule maintenance windows because of the [live migration](https://cloud.google.com/compute/docs/instances/live-migration) feature.

These initial benefits are useful, but more benefits are unlocked when you start optimizing for the cloud.

**Phase 2: Replatform**

When you replatform, you optimize your application by upgrading parts of the application's components—such as its database, caching layer, or storage system—without changing the application's architecture and with minimal changes to the codebase. The goal of Phase 2 is to start using cloud features for better management, resilience, scalability, and elasticity of your application without significantly restructuring it or leaving the VM environment.

**Take advantage of Compute Engine**

Compute Engine provides some standard features that are useful to explore. For example, you can use [instance templates](https://cloud.google.com/compute/docs/instance-templates) in Compute Engine to create templates from existing VM configurations. [Instance groups](https://cloud.google.com/compute/docs/instance-groups) are a fleet of identical VMs that lets you efficiently scale your application's performance and redundancy. Beyond simple load-balancing and redundancy, managed instance groups have scalability features like autoscaling, high availability features like [autohealing](https://cloud.google.com/compute/docs/instance-groups/autohealing-instances-in-migs), [regional deployments](https://cloud.google.com/compute/docs/instance-groups/distributing-instances-with-regional-instance-groups), and safety features like [auto-updating](https://cloud.google.com/compute/docs/images#os-details).

With these features, you can stay in the VM world but increase the resiliency, redundancy, and availability of your applications without needing to restructure your application completely.

**Look for in-place replacements**

As you move your application to the cloud, you need to look for opportunities to replace your hosted infrastructure with managed cloud options from Google and third-party partners on [Cloud Marketplace](https://cloud.google.com/marketplace), including the following:

* [Cloud SQL](https://cloud.google.com/sql) instead of self-hosted SQL Server, MySQL, or Postgres. Cloud SQL lets you focus on managing the database instead of managing the infrastructure (such as patching database VMs for security or managing backups) with the added benefit of removing the requirement for a Windows license.
* [Managed Service for Microsoft Active Directory](https://cloud.google.com/managed-microsoft-ad/docs/overview) instead of self-hosted Active Directory.
* [Memorystore](https://cloud.google.com/memorystore) instead of self-hosted Redis instances.

These replacements should require no code changes and only minimal configuration changes, and they have the advantages of minimal management, enhanced security, and scalability.

**First steps with Windows containers**

After you optimize basic functions for the cloud, you can begin the move from VMs to containers.

A container is a lightweight package that contains an application and all its dependencies. Compared to running your application directly on a VM, containers let you run your applications in various environments and in a more consistent, predictable, and efficient way (especially when you run multiple containers on the same host). The ecosystem around containers (such as Kubernetes, Istio, and Knative) also provides a number of management, resilience, and monitoring features that can further accelerate your application's transformation from a single monolith to a set of focused microservices.

For some time, containerization was a Linux-only feature. Windows applications could not benefit from containers. That changed with Windows containers and their subsequent [support in Kubernetes](https://kubernetes.io/docs/setup/production-environment/windows/intro-windows-in-kubernetes/) and [Google Kubernetes Engine (GKE)](https://cloud.google.com/kubernetes-engine/docs/how-to/creating-a-cluster-windows).

Windows containers are an option if you don't want to migrate .NET Framework applications to .NET Core but still want the benefits of containers (such as agility, portability, and control). You need to [choose the right OS to target](https://docs.microsoft.com/en-us/dotnet/architecture/modernize-with-azure-containers/modernize-existing-apps-to-cloud-optimized/deploy-existing-net-apps-as-windows-containers#choose-an-os-to-target-with-net-based-containers) depending on the .NET Framework version, and you need to remember that [not all the Windows stack is supported](https://docs.microsoft.com/en-us/dotnet/architecture/modernize-with-azure-containers/modernize-existing-apps-to-cloud-optimized/when-not-to-deploy-to-windows-containers) on Windows containers. For the limitations of this approach and alternatives, see [.NET Core and Linux containers](https://cloud.google.com/architecture/modernization-path-dotnet-applications-google-cloud#dotnet-core-linux-containers) later in this document.

After you containerize your .NET Framework application into a Windows container, we recommend that you run it in a Kubernetes cluster. Kubernetes provides standard features such as detecting when a container Pod is down and recreating it, autoscaling Pods, automated rollouts or rollbacks, and health checks. GKE adds features such as cluster autoscaling, regional clusters, highly available control planes, and hybrid and multi-cloud support with [Anthos](https://cloud.google.com/anthos). If you decide to use GKE or Anthos, you can use [Migrate to Containers](https://cloud.google.com/migrate/anthos) to simplify and accelerate the migration of Windows VMs to containers. Migrate to Containers automates the extraction of applications from VMs into containers without requiring you to rewrite or re-architect applications.

Although you can realize many benefits by using the right features in Compute Engine, moving to containers and GKE helps you fully use your VMs by packing multiple Pods onto the same VMs. This strategy potentially results in fewer VMs and lower Windows licensing costs.

Managing both Windows and Linux containers declaratively with Kubernetes and GKE can also streamline your infrastructure management. With containerization in place, your team is prepared for the next phase of modernization.

**Phase 3: Re-architect and rebuild**

Replatforming is only the start toward fully benefiting from the cloud. Transforming your architecture to a cloud-based platform offers several advantages, such as the following:

* **Increasing cost optimizations** by using Kubernetes and Linux containers instead of Windows containers and moving workloads to run on [serverless computing](https://cloud.google.com/serverless).
* **Increasing reliability and availability** by replacing self-managed VMs with managed services that provide multiregional availability, stronger service level agreements (SLAs), and better security.
* **Speeding up product delivery** by improving your developer productivity and software quality with [continuous integration tools](https://cloud.google.com/solutions/continuous-integration). You can use these tools to create automated builds, run tests, provision environments, and scan artifacts for security vulnerabilities—all within minutes.
* **Unblocking capabilities** for your applications with [petabyte-scale data warehousing](https://cloud.google.com/bigquery), [globally scalable databases for strong consistency](https://cloud.google.com/spanner), and [multiregional and highly available storage solutions](https://cloud.google.com/storage).

**The move toward managed services**

As you start rewriting parts of your application, we recommend that you begin to move from hosted services to managed services. For example, you might use the following:

* [Cloud Storage](https://cloud.google.com/storage) instead of relying on a network attached storage (NAS).
* [Pub/Sub](https://cloud.google.com/pubsub) instead of self-hosting MSMQ, RabbitMQ, or Kafka.
* [Identity-Aware Proxy (IAP)](https://cloud.google.com/iap) instead of coding the authentication layer into your application.
* [Cloud Key Management Service (Cloud KMS)](https://cloud.google.com/kms) and [Secret Manager](https://cloud.google.com/secret-manager) instead of storing your secrets and encryption keys locally with your application.

Although you need additional code to integrate your application with these services, it's a worthwhile investment, because you are shifting the burden of platform management to Google Cloud. [Google Cloud .NET Client libraries](https://github.com/googleapis/google-cloud-dotnet), [Tools for Visual Studio](https://cloud.google.com/visual-studio), and [Cloud Code](https://cloud.google.com/code) for Visual Studio Code can help you stay in the .NET ecosystem and tools while you integrate with these services.

Managed services can also support operations for your application. You can store your application logs in Cloud Logging and send your application metrics to Cloud Monitoring, where you can build dashboards with server and application metrics. Google Cloud offers .NET client libraries for [Cloud Logging](https://cloud.google.com/logging/docs/reference/libraries#client-libraries-install-csharp), [Cloud Monitoring](https://cloud.google.com/monitoring/docs/reference/libraries#client-libraries-install-csharp), and [Cloud Trace](https://cloud.google.com/trace/docs/client-libraries).

**.NET Core and Linux containers**

If your application is a legacy .NET Framework application that runs only on Windows, you might be tempted to keep it running on a Windows server on Compute Engine or in a Windows Server container on GKE. Although this approach might work in the short term, it can severely limit you in the long term. Windows comes with licensing fees and an overall larger resource footprint than Linux, and these factors can result in a higher cost of ownership in the long term.

.NET Core is the modern and modular version of .NET Framework. Microsoft [guidance](https://devblogs.microsoft.com/dotnet/net-core-3-for-windows-desktop/) states that *.NET Core is the future of .NET*. Although Microsoft plans to support .NET Framework, any new features will be added only to .NET Core (and eventually .NET 5). Even if you still want to run on Windows, any new development should occur on .NET Core.

One of the most important aspects of .NET Core is that it's multi-platform. You can containerize a .NET Core application into a Linux container. Linux containers are more lightweight than Windows containers, and they run on more platforms more efficiently. This factor creates deployment options for .NET applications and lets you break free from dependency on Windows and the licensing costs associated with it.

**Porting .NET Framework applications to .NET Core**

A good way to begin moving toward .NET Core is to read [Overview of porting from .NET Framework to .NET Core](https://docs.microsoft.com/en-us/dotnet/core/porting/). Tools such as [.NET Portability Analyzer](https://docs.microsoft.com/dotnet/standard/analyzers/portability-analyzer) and [.NET API Analyzer](https://docs.microsoft.com/dotnet/standard/analyzers/api-analyzer) can help you determine if assemblies and APIs are portable. Other porting tools such as [dotnet try-convert](https://github.com/dotnet/try-convert) can be helpful.

External tools can help you identify compatibility issues and decide what components to migrate first. Eventually, you need to create .NET Core projects, gradually move your .NET Framework code to the new project, and fix any incompatibilities along the way. Before you port your code, it's crucial to put tests in place and then test your functionality after porting. We recommend that you use [A/B testing](https://wikipedia.org/wiki/A/B_testing) to test old and new code. With A/B testing, you can keep your legacy application running while directing some of your users to the new application. This approach lets you test the outputs, scalability, and resilience of the new application. To assist with A/B testing, Google Cloud offers load-balancing solutions such as [Traffic Director](https://cloud.google.com/traffic-director/docs/traffic-control).

**Cultural transformation**

The transformation from .NET Framework and Windows servers to .NET Core and Linux containers is not merely technical. This shift requires a cultural transformation in your organization. Staff who might be used to Windows-only environments need to adapt to multi-platform environments. This cultural transformation requires time and budget to train in .NET Core, Linux, and container tools such as Docker and Kubernetes. However, a transformation from a Windows-only to a multi-platform organization lets you access a larger set of tools and skills.

**Monolith decomposition**

The move from .NET Framework to .NET Core can raise several questions, including the following:

* Do you rewrite your whole application in .NET Core?
* Do you break your application into smaller services and write those in .NET Core?
* Do you only write new services in .NET Core?

How you decide these questions needs to account for the benefits, time, and cost associated with each approach. It's good to have a balanced approach where you don't rewrite everything all at once. Instead, you can write new services in. NET Core and break your existing monolith into smaller services in .NET Core as opportunity arises. The following whitepapers can help as you plan:

* [Migrating a monolithic application to microservices on GKE](https://cloud.google.com/solutions/migrating-a-monolithic-app-to-microservices-gke)
* [Whitepaper: Embark on a journey from monoliths to microservices](https://cloud.google.com/blog/products/gcp/whitepaper-embark-on-journey-from-monoliths-to-microservices)

**Deployment options for .NET Core containers**

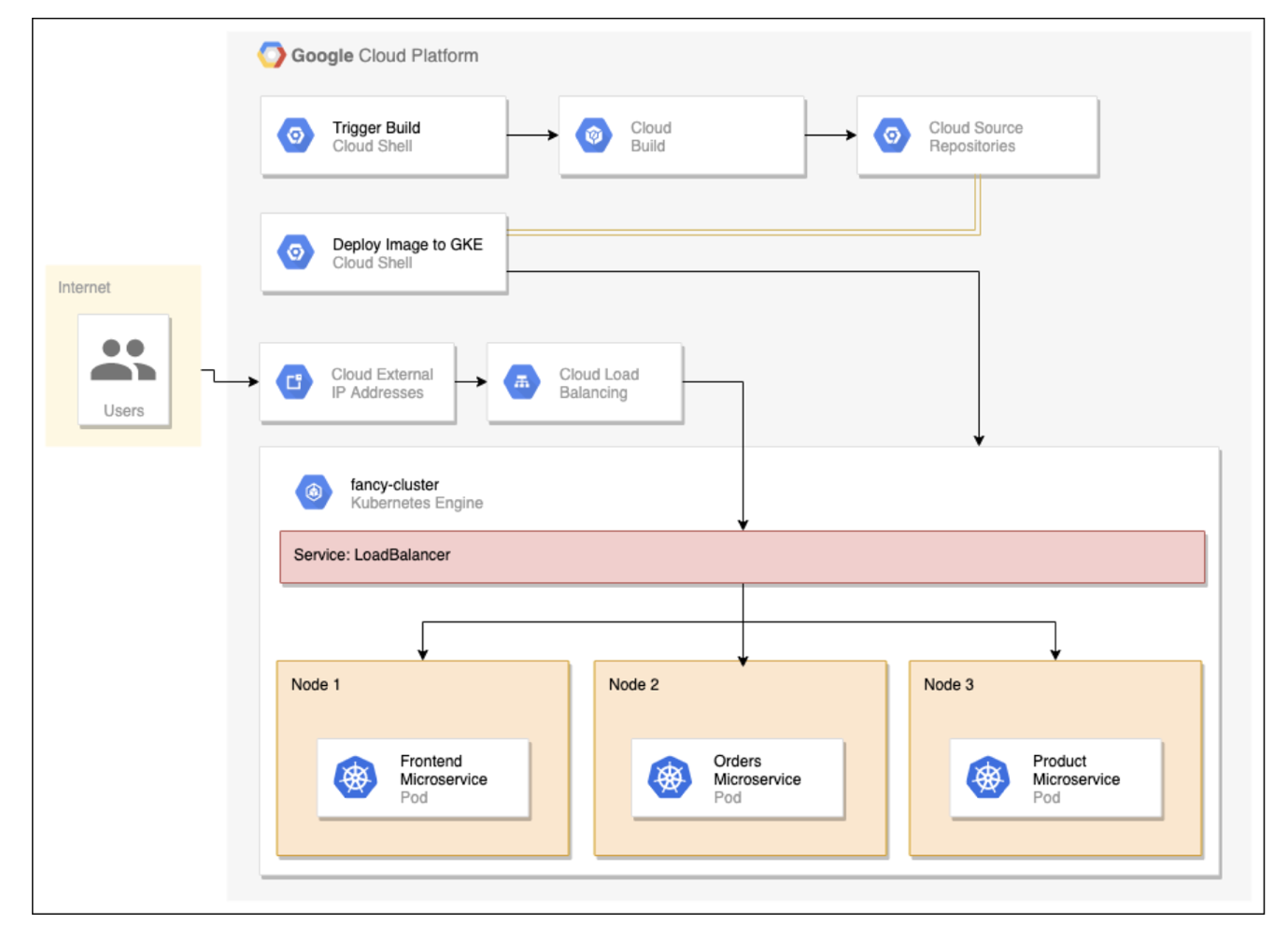
As [Deploying .NET apps on Google Cloud](https://cloud.google.com/solutions/deploy-dotnet-applications) states, you have different options for deploying .NET Core containers on Google Cloud. As you deconstruct your monolithic application to microservices, you might decide to use more than one hosting solution, depending on the architecture and design of your microservices.

Answering the following questions can help you decide the best hosting strategy:

* **What triggers your application?** All hosting solutions are suitable for standard HTTP(S), but if your protocol is TCP/UDP or a proprietary protocol, GKE might be your only option.
* **Does your application require specific hardware?** Cloud Run offers a reasonable but limited amount of RAM and CPU for each request. Cloud Run for Anthos offers further customization options such as GPU, more memory, and disk space.
* **What are your scaling expectations?** If your application has periods of inactivity, serverless solutions such as Cloud Run can offer the option to scale down to zero.
* **How important is latency and how tolerant is your application to** [**cold starts**](https://wikipedia.org/wiki/Cold_start_(computing))**?** If tolerance to cold starts is low, you need to consider using a minimum number of instances on App Engine flexible environment or GKE with autoscaling.
* **Does your service communicate with other services?** If your application requires access to your VPC, or to a peered VPC, consider hosting your application in App Engine flexible environment, Cloud Run on Anthos, or GKE.

We recommend that you read the documentation for each hosting environment to get familiar with its capabilities, strengths and weaknesses, and pricing model.

As a general rule, if you want to create microservices that serve HTTP requests, you need to deploy to Cloud Run when possible and fall back to GKE if you want to stay in the Kubernetes ecosystem or require more customization options. GKE is also the default choice if you have a long-running process, such as a process that listens on a queue, or an application that uses protocols other than HTTP(S). We recommend that you use App Engine flexible environment only if you cannot use Cloud Run because of some limitation—for example, with WebSocket support or connecting to a VPC network—or if your application doesn't justify the complexity of Kubernetes.



**App Engine flexible environment**

[App Engine flexible environment](https://cloud.google.com/appengine/docs/flexible) is a platform that can directly run web applications written in many languages, including .NET Core. App Engine installs and runs your .NET Core application inside Docker containers that have the .NET runtime. If you want to customize this runtime, you can [provide App Engine with your own Dockerfile](https://cloud.google.com/appengine/docs/flexible/dotnet/runtime#extending_the_runtime). App Engine still uses VM-based Compute Engine instances, but App Engine manages those VMs. The flexible environment comes with some features like autoscaling, revisions, and traffic splitting. The flexible environment also allows multiple services in a single application, a capability that is useful for microservices architecture.

If you want to deploy a web frontend or a small number of similar containers, App Engine is an option. However, with App Engine, deployment times are generally slower than other options, and you're still charged for each VM, even if you don't use the application.

**Kubernetes and GKE**

If you want to run in a container-optimized environment, that approach likely involves Kubernetes and its managed version, [GKE](https://cloud.google.com/kubernetes-engine). Kubernetes and GKE are especially applicable if you plan to deploy many containers with different requirements and want fine-grained control on how each one is deployed and managed.

Kubernetes is designed to run containers at scale and provides building blocks such as Pods, Services, Deployments, and replica sets. Properly understanding and using these constructs can be challenging, but they let you shift most of the management burden of containers to Kubernetes. They're also well suited for microservices architecture where a microservice is a Deployment with a set of load-balanced Pods behind a Service.

In addition to Kubernetes, GKE offers features such as cluster autoscaling, auto-repairing, and [auto-upgrading](https://cloud.google.com/kubernetes-engine/docs/how-to/node-auto-upgrades) to simplify Kubernetes management, and security features such as container isolation and private registries. Although you are charged for each node in the cluster in GKE, GKE supports [preemptible VMs](https://cloud.google.com/kubernetes-engine/docs/how-to/preemptible-vms) to reduce costs.

GKE can manage both Windows and Linux containers. This capability is useful if you want to maintain a single hybrid environment for both Windows-based and modern Linux-based applications.

**Knative and Cloud Run**

For your stateless .NET Core containers, [Knative](https://knative.dev/) and its managed version, [Cloud Run](https://cloud.google.com/run), provide a serverless container environment. Serverless containers offer benefits such as provisioning, autoscaling, and load balancing without the infrastructure management overhead.

For deploying containers in a Kubernetes cluster, Knative provides an API surface that is higher level and smaller than Kubernetes. Knative can thus help you avoid the complexities of Kubernetes, making your container deployment easier.

Cloud Run follows the Knative API but runs on Google infrastructure, thus removing the need for Kubernetes clusters. Cloud Run provides a serverless option for containers. By default in Cloud Run, containers are autoscaled and billed for the duration of the request. The deployment time is in seconds. Cloud Run also inherits useful features from App Engine, such as revisions and traffic splitting.

[Cloud Run for Anthos](https://cloud.google.com/anthos/run) is the more flexible version of Cloud Run for those customers who want the simplicity of Knative and Cloud Run with the operational flexibility of Kubernetes. For example, Cloud Run on Anthos lets you add GPUs to underlying instances running your containers, or lets you access other VMs or GKE clusters in the same VPC.

Cloud Run integrates with other services such as Pub/Sub, Cloud Scheduler, Cloud Tasks, and backends such as Cloud SQL. It can be used for both autoscaled web frontends or internal microservices triggered by events.

**Modernization of operations**

Modernizing is not only about your application code. It applies to the whole lifecycle of your application—how it is built, tested, deployed, and monitored. Therefore, when you think about modernization, you need to consider operations.

[Cloud Build](https://cloud.google.com/build) can help you modernize and automate your application's build-test-deploy cycle. Not only does Cloud Build provide [builders for .NET Core](https://github.com/GoogleCloudPlatform/cloud-builders/tree/master/dotnet), Cloud Build also integrates with Container Registry's [Vulnerability Scanner](https://cloud.google.com/container-registry/docs/container-analysis) and with [Binary Authorization](https://cloud.google.com/binary-authorization/docs/cloud-build) to prevent images that are built from unknown source code or insecure repositories from running in your deployment environment.

[Google Cloud's operations suite](https://cloud.google.com/products/operations) (formerly Stackdriver) offers several services that let you modernize the observability of your application:

* [Cloud Logging](https://cloud.google.com/logging) for application and system logs
* [Cloud Monitoring](https://cloud.google.com/monitoring) for performance monitoring
* [Cloud Trace](https://cloud.google.com/trace) and [Cloud Debugger](https://cloud.google.com/debugger) for application performance management

