This article describes common patterns for creating production-level, heterogeneous deployments using Kubernetes (https://kubernetes.io/). The article reviews three use cases and describes the architectural details you might use to create deployments for them. The architectural descriptions include Kubernetes in general and Google Kubernetes Engine (GKE) (/kubernetes-engine/) in particular.

Heterogeneous deployments typically involve connecting two or more distinct infrastructure environments or regions to address a specific technical or operational need. Heterogeneous deployments are called "hybrid", "multi-cloud", or "public-private", depending upon the specifics of the deployment. For the purposes of this article, heterogeneous deployments include those that span regions in a single cloud environment, multiple public cloud environments (multi-cloud), or a combination of on-premises and public cloud environments (hybrid or public-private).

Various business and technical challenges can arise in deployments that are limited to a single environment or region:

- **Maxed out resources**: In any single environment, particularly in on-premises environments, you might not have the compute, networking, and storage resources to meet your production needs.
- **Limited geographic reach**: Deployments in a single environment require people who are geographically distant from one another to access one deployment. Their traffic might travel around the world to a central location.
- **Limited availability**: Web-scale traffic patterns challenge apps to remain fault-tolerant and resilient.
- **Vendor lock-in**: Vendor-level platform and infrastructure abstractions can prevent you from porting apps.
- **Inflexible resources**: Your resources might be limited to a particular set of compute, storage, or networking offerings.

Heterogeneous deployments can help address these challenges, but they must be architected using programmatic and deterministic processes and procedures. One-off or ad-hoc deployment procedures can cause deployments or processes to be brittle and intolerant of failures. Ad-hoc processes can lose data or drop traffic. Good deployment processes must be repeatable and use proven approaches for managing provisioning, configuration, and maintenance.

Three common scenarios for heterogeneous deployment are multi-cloud deployments, fronting on-premises data, and continuous integration/continuous delivery (CI/CD) processes.

The following sections describe some common use cases for heterogeneous deployments, along with well-architected approaches using Kubernetes and other infrastructure resources.

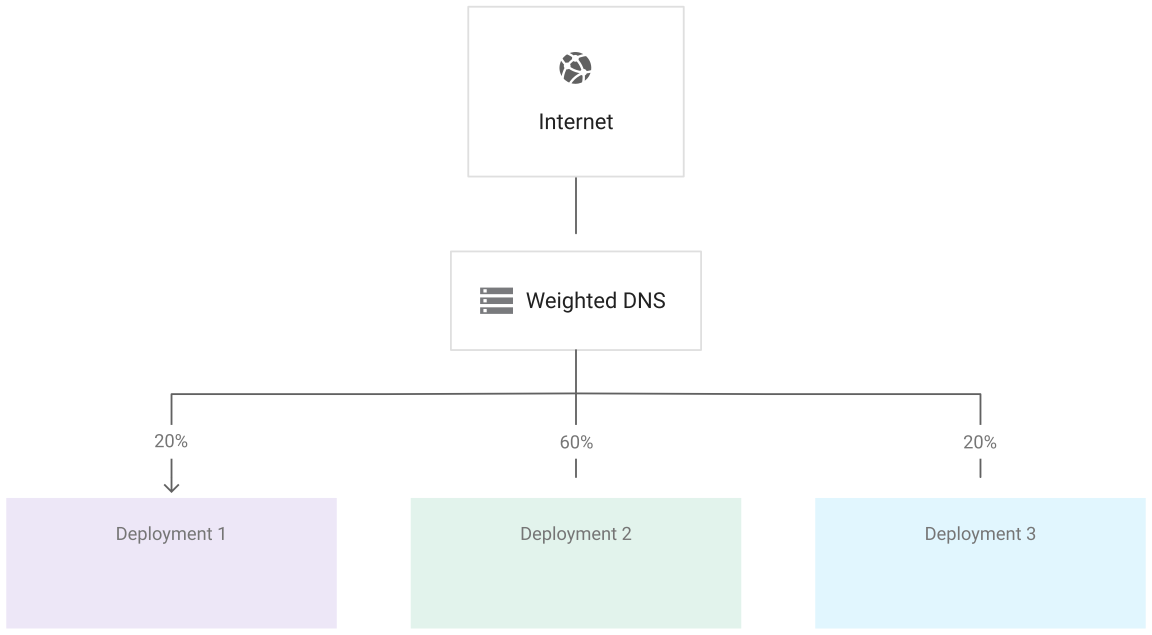
Multi-cloud deployments, in which all deployments are relatively similar, are some of the most common heterogeneous deployment patterns used with Kubernetes.

One use for multi-cloud deployments involves choosing how traffic is directed. In the simplest deployments, you might choose to send specific percentages of inbound traffic to specific deployments. In deployments built from on-premises and public cloud infrastructure, you might send more traffic to cloud infrastructure, either to facilitate a longer term migration or to get around constrained, on-premises resources.

Another common use for multi-cloud deployments is to configure a highly available deployment, able to withstand the failure of any single environment. In these scenarios, you can orchestrate the same Kubernetes deployment into each of the desired environments. Each deployment should be capable of scaling up to meet the needs of all traffic, should any single environment fail.

Finally, you can use multi-cloud deployments to create deployments that are physically closer to users. Placing deployments as near as possible to users can minimize request and response latency.

A robust multi-cloud deployment uses a DNS traffic-management service to resolve DNS queries to the individual deployments. For general traffic-splitting use cases, you can configure DNS traffic-management mechanisms to split traffic, by percentages, across individual deployments.



For highly available deployments, you can configure the DNS mechanism by using custom health checks from each environment. When an environment becomes unhealthy, it stops sending healthy status updates, and the DNS mechanism can shift traffic to deployments that are still healthy.

When latency to the user is critical, DNS mechanisms can use an inbound request's IP address to determine its approximate location, and then direct traffic to the deployment closest to that geographical region. You can use DNS infrastructure service providers, such as NS1 (https://ns1.com/), Dyn (https://dyn.com/), Akamai (https://www.akamai.com/), and others, to direct traffic across multiple deployments.

As DNS directs traffic to particular deployments, a load balancer should receive incoming requests and then direct them to a Kubernetes cluster. Kubernetes offers two mechanisms to expose pods (https://kubernetes.io/docs/user-guide/pods/) to incoming traffic: Services (https://kubernetes.io/docs/user-guide/services/) and Ingress (https://kubernetes.io/docs/user-guide/ingress/).

When pods are deployed in a Kubernetes cluster, they are not easily accessible to other apps or systems inside or outside of the cluster. To make pods accessible, you must first create a service. By default, a service automatically accepts connections from within the cluster, but it can also be configured to accept connections from outside the cluster.

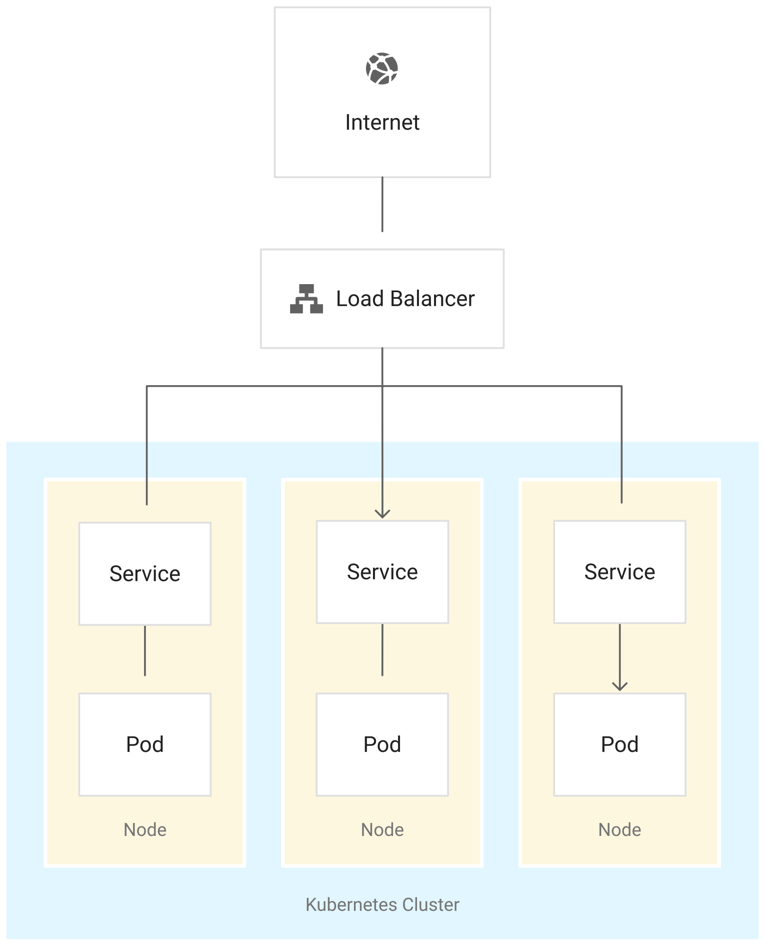
When configuring a service to accept external requests, you configure the service type as either `NodePort` or `LoadBalancer`.

Setting the service type to `NodePort` exposes a unique port for each service on all nodes in the Kubernetes cluster. This unique port allows each node to accept connections and proxy load balance incoming requests to the appropriate pods.

The `LoadBalancer` service type is a superset of NodePort. In addition to providing the port configuration on each node, setting the type to `LoadBalancer` automatically provisions an external load balancer and configures it to direct traffic to the cluster and into the subsequent pods.

Automatic creation and configuration is available only in supported cloud environments, such as Google Cloud.

Services in Kubernetes are a Layer 4 (https://wikipedia.org/wiki/Transport_layer) construct, meaning they can support accessibility only by using IP addresses and port numbers. Ingress, an HTTP(S) (Layer 7 (https://wikipedia.org/wiki/Application_layer)) load-balancing mechanism, is a collection of rules that allow inbound connections to reach backend Kubernetes cluster services. You can configure the Ingress mechanism to give Services additional app layer functionality such as externally reachable URLs, inbound-traffic load balancing, SSL termination, or name-based virtual hosting. Inbound traffic can be directed to pods, exposed through services, by using HTTP host headers or HTTP URL paths.



When you run multi-cloud deployments, you might need to run one or more shared services, such as a database, in each deployment. The communication between shared services needs to be low-latency and secure.

For low-latency, high-bandwidth connectivity, you can connect the underlying networks in each deployment by using direct peering or third-party managed network interconnects. Google Cloud offers direct connectivity through peering (/interconnect/direct-peering) directly with Google at any one of the available network edge locations. To facilitate direct peering, there are a number of technical requirements (https://peering.google.com/#/options/peering). When it is not possible to meet those requirements, another option is Cloud Interconnect (/interconnect/). You can use Cloud Interconnect service providers (/interconnect/docs/how-to/carrier-peering#service_providers) to connect to Google's network edge with enterprise-grade connectivity.

After connectivity is in place, the next step is to secure the link between each environment by using VPN. In each deployment, you need a VPN gateway to secure traffic between the deployments. In Google Cloud, Cloud VPN (/vpn/docs/concepts/overview) secures traffic by using an IPSec VPN connection. Cloud VPN supports multiple VPN tunnels to aggregate bandwidth, and it supports static or dynamic routing by using Cloud Router (/router/docs/).

In your multi-cloud deployment, you might want to administer and control each Kubernetes cluster as an individual entity. To do this, you must manually create the pods and services in each cluster. The benefit here is that, while each deployment might have some shared apps and services, it might also have apps and services suited only for itself.

For example, you might need your deployments to be geographically distributed, but there might be country- or region-specific services that are not appropriate for all geographies.

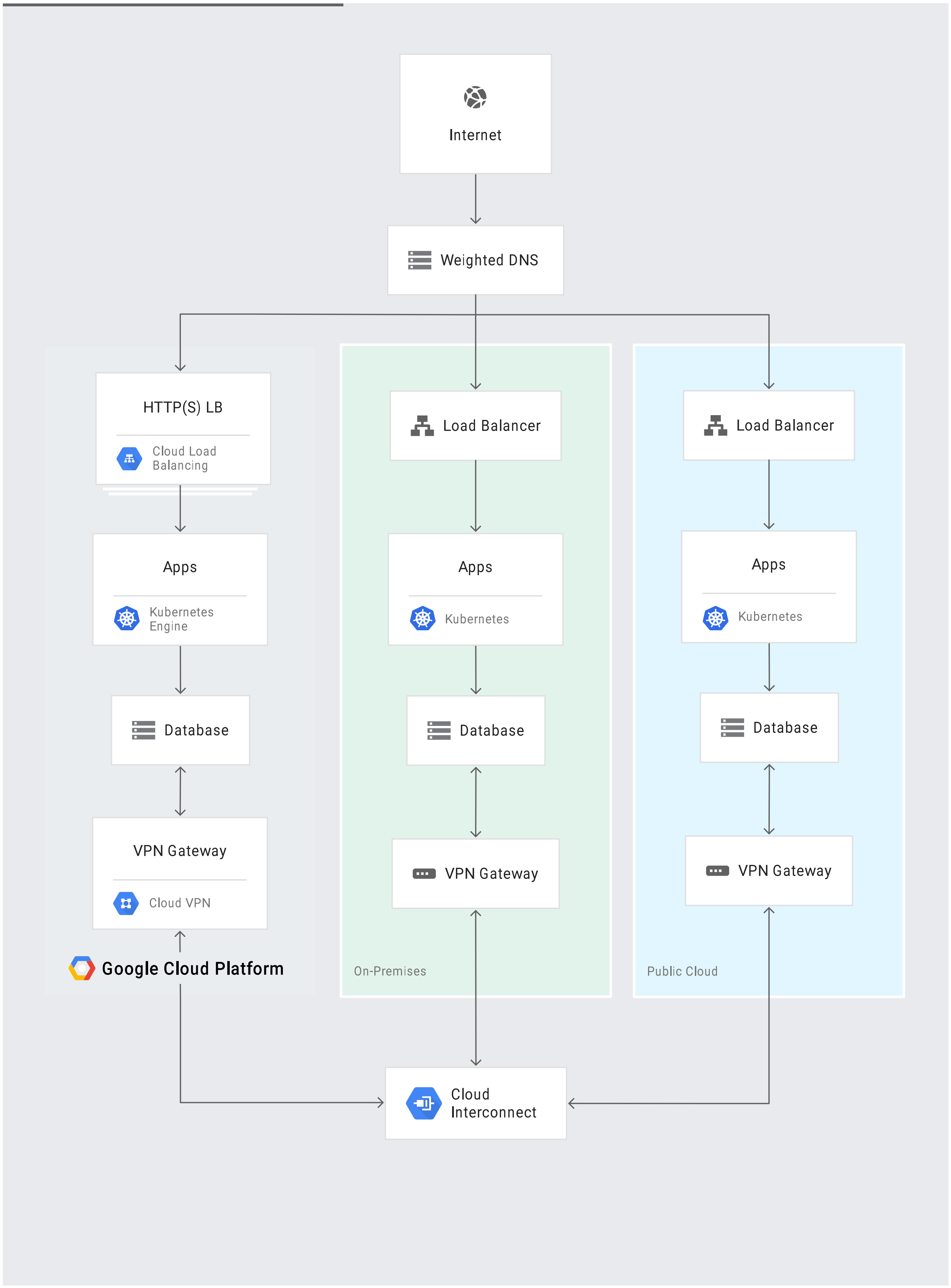
Deployments in which traffic splitting is unevenly distributed might need to deploy pods and services on a per-cluster basis, to ensure incoming traffic and requests are handled appropriately.

Multi-cloud deployments should use service discovery to ensure that different apps and services can easily find one another at run time. Service discovery eliminates the need for complex or brittle naming schemes or conventions that might have to be known prior to deployment. Service discovery mechanisms need to be transparent to the source and destination apps. In multi-cloud deployments, these mechanisms must enable apps and services to discover services running locally and remotely in other clusters.

In deployments architected and deployed as standalone individual clusters, service discovery mechanisms need to be *data center aware*. That is, they must be able to operate as coordinated distributed systems, with the ability to send requests to the appropriate cluster based on the incoming request, local availability, and load capacity. Third-party systems such as Consul (https://www.consul.io/), Linkerd (https://linkerd.io/), and others, can facilitate cross-cluster and cross-environment service discovery with multi-cloud Kubernetes deployments.

Kubernetes provides support for resolving private DNS zones in the Kubernetes cluster. This support is useful in hybrid or multi-cloud scenarios in which the fully qualified domain names of the other clusters are known and traffic can be easily routed to them. You can set up access to private "stub" domains by using `ConfigMap`. You can use Kubernetes support for resolving private DNS zones as a standalone mechanism to send requests to specific Kubernetes clusters, or in conjunction with external systems such as Consul.

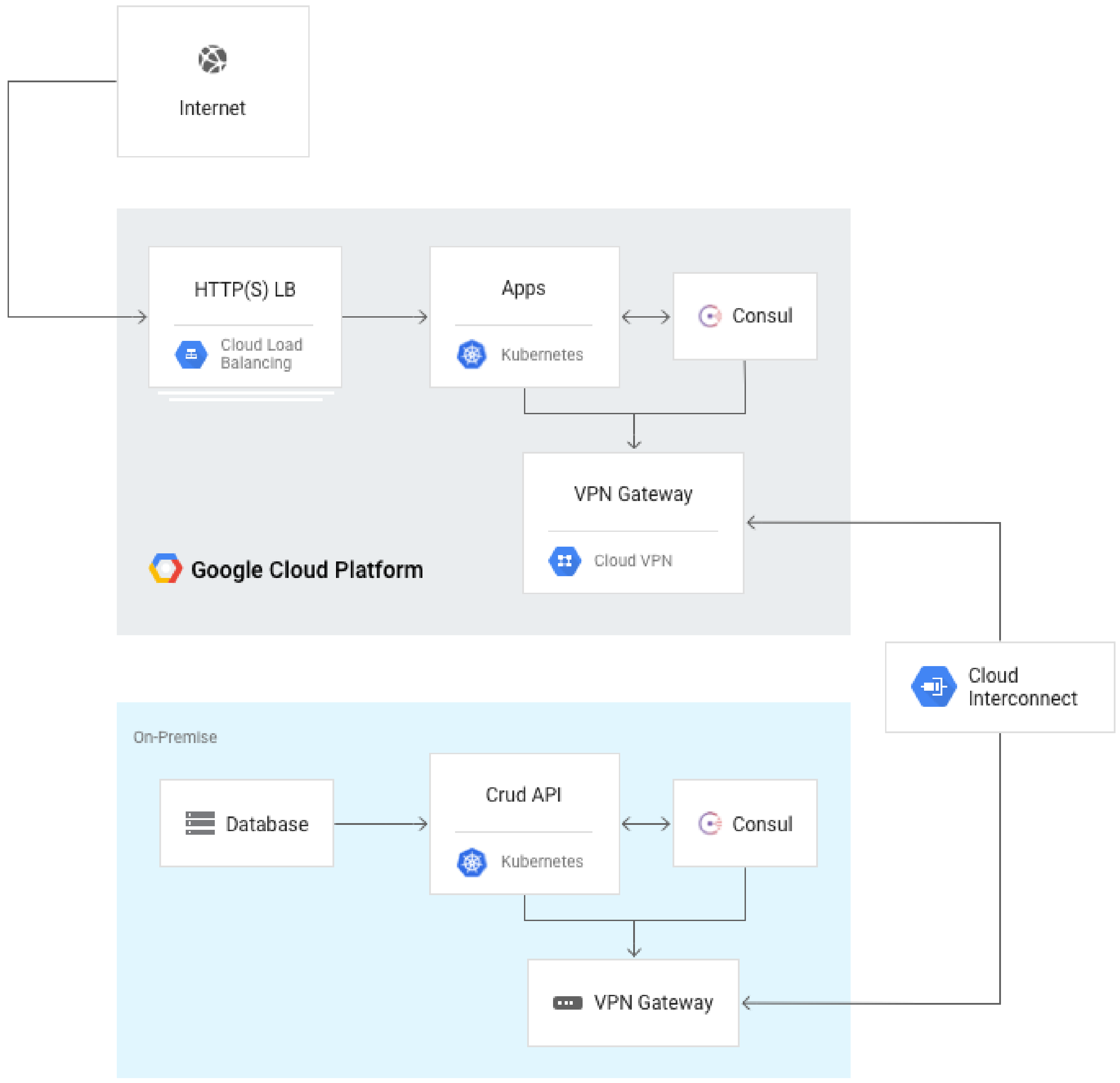
The following diagram shows an example multi-cloud deployment architecture.


Internet
Weighted DNS
HTTP(S) LB
Cloud Load Balancing
Apps
Kubernetes Engine
Database
VPN Gateway
Cloud VPN
Google Cloud Platform
Load Balancer
Apps
Kubernetes
Database
VPN Gateway
On-Premises
Load Balancer
Apps
Kubernetes
Database
VPN Gateway
Public Cloud
Cloud Interconnect

You can architect your cloud deployments to extend capabilities beyond what is available in your private or on-premises deployments. For example, you can architect and deploy cloud-based apps that can access private data systems or infrastructure.

Recall that private or on-premises deployments might be limited in availability, portability, or resource flexibility. Migrating to cloud-based deployments can address these limitations, but might not be possible due to legacy architectures, security compliance, or other software requirements. In such scenarios, you can build and deploy new apps to cloud environments that have greater flexibility or capability than private or on-premises environments.

The following diagram shows an example architecture that demonstrates cloud apps fronting on-premises data infrastructure.



For deployments in which cloud apps are fronting on-premises data infrastructure, you need secure, low-latency connectivity to minimize the overall app response time for your users. You can use Cloud Interconnect (/interconnect/) or direct peering (/interconnect/direct-peering) to minimize latency and maximize available bandwidth between your on-premises and cloud environments. After connectivity is established, each deployment must have a VPN gateway to secure traffic between the deployments. In Google Cloud, Cloud VPN (/vpn/docs/concepts/overview) secures traffic with an IPSec VPN connection. Cloud VPN supports multiple VPN tunnels to aggregate bandwidth, and it supports static or dynamic routing using Cloud Router (/router/docs/). Because security is a critical concern when fronting on-premises data infrastructure, you should configure routes and on-premises firewalls to allow traffic from only specific sets of Google Cloud instances.

In the cloud portion of such hybrid deployments, the architectural components must include an appropriate load balancer, app hosting infrastructure, a VPN gateway, and a service discovery mechanism. The choice of load balancer will depend on the requirements for end-user facing apps. For deployments on Google Cloud, Cloud Load Balancing (/load-balancing/) offers support for HTTP(S), TCP, and UDP with a single, globally available, anycast IP.

In hybrid scenarios such as these, you can deploy pods and services on GKE (/kubernetes-engine/) , a managed Kubernetes deployment available in Google Cloud. GKE directs outbound traffic to the on-premises infrastructure. GKE uses Cloud VPN to secure the traffic, and it uses Cloud Router to configure static or dynamic routes. This configuration ensures that only traffic from the GKE cluster traverses the VPN connection.

The on-premises portion of this deployment contains the data infrastructure backing the cloud apps. For most deployments of this nature, deploying a CRUD API (https://wikipedia.org/wiki/Create,_read,_update_and_delete) in front of the data systems offers several benefits.

If the data systems require high levels of security or compliance, a CRUD API can be useful to help audit and log inbound connections and queries. If the data infrastructure runs on a legacy system, a CRUD API can help provide more-modern connectivity options for newer apps.

In both cases, the CRUD API can help decouple built-in database authentication and authorization mechanisms from those needed by apps, and provide only as much CRUD functionality as the apps require. Specifically, if only a subset of data needs to be exposed to downstream apps, an API can be useful to manage access to the data.

Through auditing connections and queries, the API can also help define the long- term migration strategy of the underlying data. If only a subset of data is needed, and it doesn't fall under stringent security or compliance policies, that data could be migrated to cloud platforms.

The architecture diagram above shows the CRUD API hosted in Kubernetes running on-premises. Using Kubernetes on premises is not technically required, but it offers advantages: as more teams consider Kubernetes as a deployment target in cloud infrastructure, they can benefit from developing additional expertise in using and operating the system.

The on-premises infrastructure must configure the VPN gateway and any firewalls to allow traffic to reach the CRUD API from only known sources, to minimize potential security issues.

In hybrid deployment scenarios, you should use service discovery to ensure that different apps and services can easily connect to one another at run time. Over time, additional cloud apps might be deployed that leverage different components of the on-premises CRUD API. The CRUD API might add additional functionality over time, such as task-specific APIs or APIs to front additional on-premises data infrastructure. In these kinds of deployment scenarios, the release cycle of cloud apps versus on-premises CRUD functionality might differ significantly. Using an external service discovery mechanism, such as Consul (https://www.consul.io/) or Linkerd (https://linkerd.io/), can provide loose coupling of resources and versions, so that they might iterate independently in each environment.

If you plan to deploy cloud apps only in GKE or Kubernetes, you can configure the internal Kubernetes DNS mechanism `kube-dns` to resolve private DNS domains to private IP addresses in the on-premises environment. In that configuration, pods running in the cloud environment can use standard DNS queries to easily access services running in the on-premises environment. For more information, refer to Configuring Private DNS Zones and Upstream Nameservers in Kubernetes (https://kubernetes.io/blog/2017/04/configuring-private-dns-zones-upstream-nameservers-kubernetes/).

Multi-cloud workloads that start from existing on-premises deployments can benefit from migrating more-focused workloads to cloud environments.

Continuous integration (CI) workloads are good candidates for migration, because the ability of cloud environments to automatically scale compute resources can help reduce the time from code completion to built artifacts.

Continuous delivery (CD) workloads can also benefit from running in cloud environments, which enable easier provisioning and deployment of test environments. Migration can increase the number of parallel build processes for unit testing. Another potential benefit is increasing the number of test deployments for end-to-end integration testing, and other automated testing.

Cloud-based, container-centric, CI/CD workloads typically use the following high-level process:

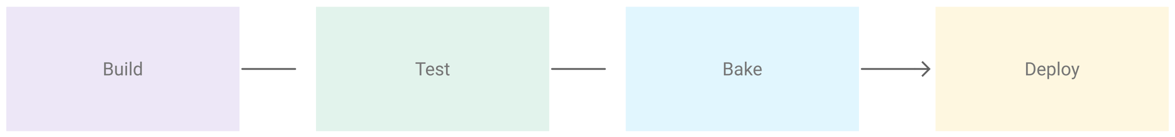
- **Develop.** Developers commit and push code changes to local or remote-hosted source repositories.
- **Build.** A build service continually polls the source code repository. Upon seeing new changes, the service starts the build process.
- **Unit test.** The process builds source, executes unit tests, and builds a resulting container image.
- **Integration test.** The process creates a test cluster, deploys the container image with its associated artifacts, and executes integration tests.
- **Bake.** Upon successful completion, container images are tagged with release version metadata.
- **Deploy.** Optionally, developers or admins can deploy new artifacts to production.

The CI/CD tools most commonly used with Google Cloud are Jenkins (https://jenkins.io/) and Spinnaker (http://www.spinnaker.io/). Jenkins is a popular open source CI/CD system that can be deployed on standalone compute instances or as a series of pods and services in Kubernetes. Spinnaker is an open-source CD system capable of orchestrating and automating software delivery to multiple targets. Spinnaker can leverage CI systems such as Jenkins or other tools such as Cloud Build (/cloud-build/).

For CI/CD workloads using Jenkins with Kubernetes, refer to the following documentation that reviews best practices, common configuration patterns, and orchestration of continuous delivery:

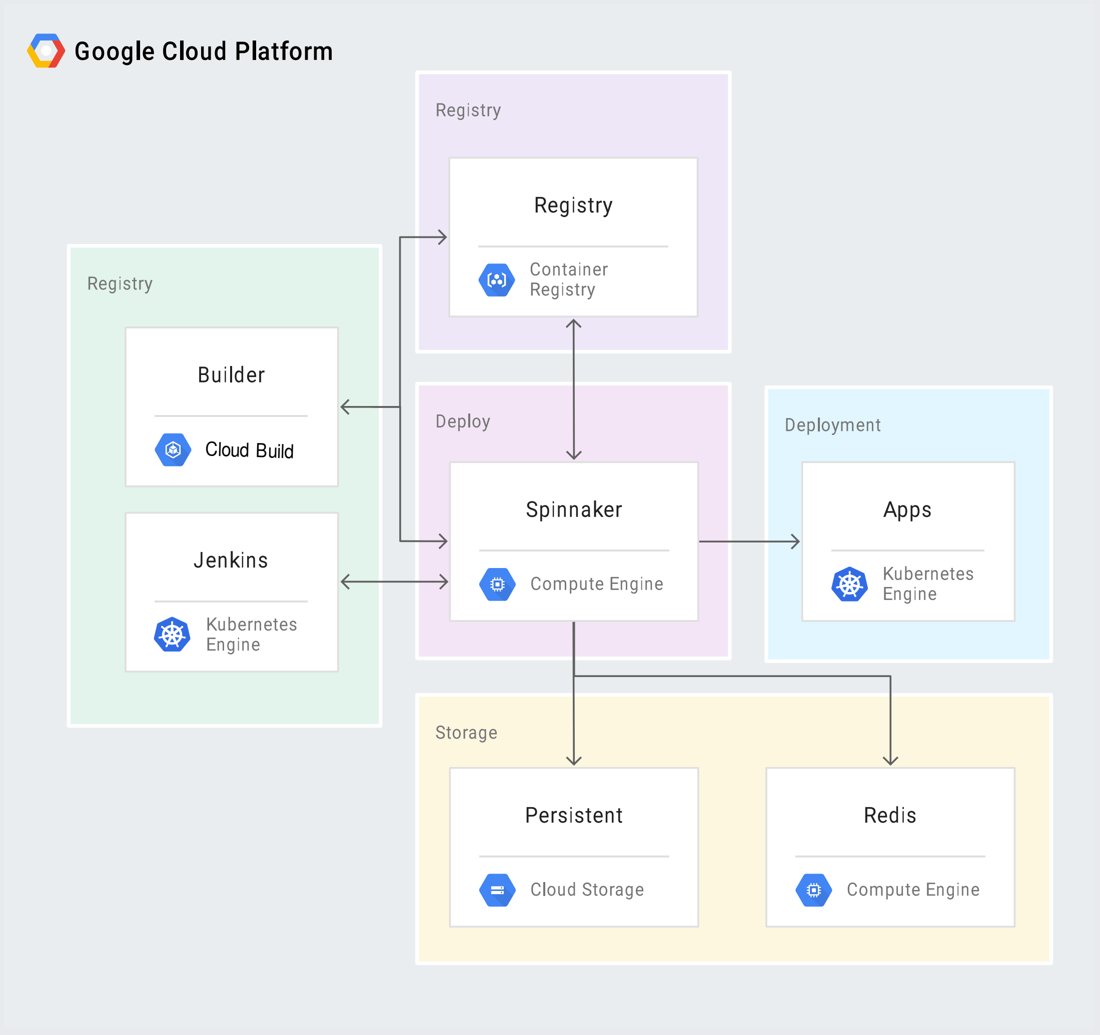
- Best Practices for Running Jenkins on GKE (/solutions/jenkins-on-kubernetes-engine)
- Setting up Jenkins on GKE (/solutions/jenkins-on-kubernetes-engine-tutorial)
- Configuring Jenkins for GKE (/solutions/configuring-jenkins-kubernetes-engine)
- Continuous Delivery in GKE using Jenkins (/solutions/continuous-delivery-jenkins-kubernetes-engine)

Spinnaker is an open source, multi-cloud CD platform that can orchestrate software deployment workflows and cluster management. Spinnaker's cluster management features provide the ability to provision and control cloud resources such as instance groups, instances, firewalls, and load balancers. The software deployment workflows consist of pipelines, each of which consists of a sequence of actions, called *stages*. One stage in a Spinnaker pipeline can pass parameters to the next stage. Pipelines can be started manually or through automatic triggers, such as external CI systems, `cron` scripts, or other pipelines. Spinnaker ships with several pre-packaged stages for baking images, deploying images, working with instance groups, and requesting user input. The following image describes how Spinnaker pipelines release software.



Software is built and then tested. If all tests pass, an immutable image is baked and made available in the cloud. After the image is available, it can be deployed to clusters to update their running software.

When working with container deployments, Spinnaker leverages external CI systems such as Jenkins or Cloud Build to execute the build, test, and bake steps. Spinnaker then orchestrates the target deployment using standard pipeline stages. The following image shows the architecture of such a system.



For information on deploying Spinnaker on Google Cloud, see Running Spinnaker on Compute Engine (/solutions/spinnaker-on-compute-engine).

Jenkins works well with Spinnaker's support for triggers to start pipelines. You can use Jenkins builds to automatically trigger Spinnaker pipelines. In a Spinnaker pipeline, Jenkins Script stages can execute the test and bake stages of the release process. Spinnaker's built-in cluster management stages can orchestrate the target deployment. For more information, see the Spinnaker Hello Deployment (https://www.spinnaker.io/docs/hello-spinnaker) example.

Cloud Build is a fully-managed Google Cloud service that builds container images in a fast, consistent, and reliable environment. Cloud Build integrates directly with Cloud Source Repositories (/source-repositories/) to automatically trigger builds based on changes to sources or repository tags. Cloud Build executes builds in Docker containers and can support any custom build step that can be packaged in a container. Builds in Cloud Build are deeply customizable, with support for step sequencing and concurrency. State changes in the build process are automatically published to a Pub/Sub (/pubsub/) topic.

While Spinnaker does not directly support Cloud Build, it does provide support for Container Registry (/container-registry/) , the container registry automatically used by Cloud Build. You can configure Spinnaker to poll Container Registry and start the pipeline based on detecting updated container image versions or tags. In such scenarios, you should configure Cloud Build to execute the build, test, and bake stages of the release process. You can read about details for configuring Spinnaker to leverage Container Registry in the Spinnaker documentation (https://www.spinnaker.io/setup/install/halyard/#install-halyard-on-docker).

Spinnaker's built-in cluster management mechanisms support Kubernetes. Server Groups and Load Balancers in Spinnaker correspond to Replica Sets and Services within Kubernetes, respectively.

The following documentation reviews the steps necessary for configuring Spinnaker to deploy pods and services to Kubernetes:

- Kubernetes provider setup (https://www.spinnaker.io/setup/providers/kubernetes/)
- Kubernetes provider reference (https://www.spinnaker.io/reference/providers/kubernetes/)
- Kubernetes Source to Prod (https://www.spinnaker.io/guides/tutorials/codelabs/kubernetes-v2-source-to-prod/)

- Learn about Cloud Interconnect (/interconnect/)
- Deploy Spinnaker on Compute Engine (/solutions/spinnaker-on-compute-engine)
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).