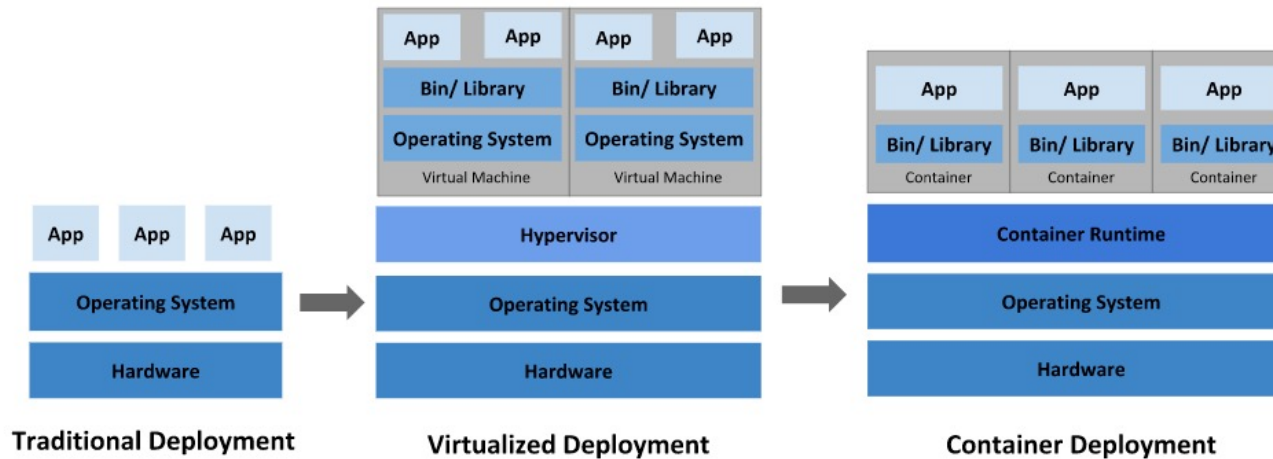


- History
- Open Source K8S Engine
- Google Kubernetes Engine
- GKE Networking
- GKE deployments





Why Containers?

- Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
- Loosely coupled, distributed, elastic, liberated micro-services: applications are broken into smaller, independent pieces and can be deployed and managed dynamically – not a monolithic stack running on one big single-purpose machine.
- Continuous development, integration, and deployment
- Environmental consistency across development, testing, and production: Runs the same on a laptop as it does in the cloud.
- Cloud and OS distribution portability: Runs on Ubuntu, RHEL, CoreOS, on-premises, on major public clouds, and anywhere else.
- Application-centric management: Raises the level of abstraction from running an OS on virtual hardware to running an application on an OS using logical resources.
- Resource utilization: high efficiency and density.

Why Kubernetes:

- **Service discovery and load balancing** Kubernetes can expose a container using the DNS name or using their own IP address. If traffic to a container is high, Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks**
- **Automatic bin packing** You provide Kubernetes with a cluster of nodes that it can use to run containerized tasks. You tell Kubernetes how much CPU and memory (RAM) each container needs. Kubernetes can fit containers onto your nodes to make the best use of your resources.
- **Self-healing** Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.
- **Secret and configuration management** Kubernetes lets you store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys. You can deploy and update secrets and application configuration without rebuilding your container images, and without exposing secrets in your stack configuration

Kubernetes playgrounds:

- [Katacoda](https://www.katacoda.com/courses/kubernetes/playground) - <https://www.katacoda.com/courses/kubernetes/playground>
- [Play with Kubernetes](https://labs.play-with-k8s.com/) - <https://labs.play-with-k8s.com/>

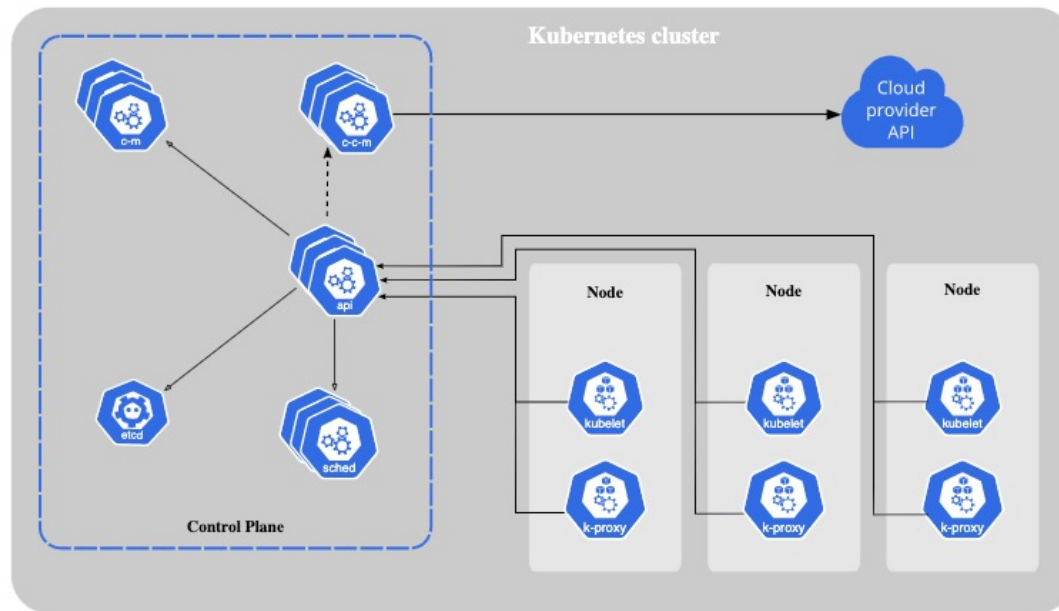


Kubernetes: Architecture & Components










Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. A Kubernetes cluster consists of a set of worker machines, called [nodes](#), that run containerized applications. The worker node(s) host the [Pods](#) that are the components of the application workload. The [control plane](#) manages the worker nodes and the Pods in the cluster.

Open Source Kubernetes architecture

- The core of Kubernetes' control plane is the API server. The API server exposes an HTTP API that lets end users communicate with the cluster.
- kubectl - command-line interface or other command-line tools, such as kubeadm
- In Kubernetes, a Pod is the most basic deployable unit within a Kubernetes cluster. A Pod runs one or more containers. Zero or more Pods run on a node. Each node in the cluster is part of a node pool.



Kubernetes Components:

API server		Front end and inter cluster communicator for the Kubernetes Control Plane. Designed to scale horizontally.
Cloud controller manager (optional)		A control plane component that embeds cloud-specific control logic (Node, Route, Service Controllers)
Controller manager		Abstract Components to reduce complexity (Node, Replica, Endpoints, Service Account & Token Controllers)
etcd (persistence store)		Distributed key-value store used to store state information
kubelet		Runs on every node to make sure container is running.
kube-proxy		A Network proxy to maintain network rules on pods.
Scheduler		Low level computer abstractions to run pods.
Control plane		Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.
Node		<ul style="list-style-type: none">• Kubelet - Runs on every node to make sure container is running.• Kube-proxy - A Network proxy to maintain network rules on pods.

Kubernetes provides several built-in workload resources:

- [Deployment](#) and [ReplicaSet](#). Deployment is a good fit for managing a stateless application workload on the cluster, where any Pod in the Deployment is interchangeable and can be replaced if needed.
- [StatefulSet](#) If your workload records data persistently, you can run a StatefulSet that matches each Pod with a [PersistentVolume](#).
- [DaemonSet](#) defines Pods that provide node-local facilities. These might be fundamental to the operation of your cluster, such as a networking helper tool, or be part of an add-on.
- [Job](#) and [CronJob](#) define tasks that run to completion and then stop. Jobs represent one-off tasks, whereas CronJobs recur according to a schedule.



Kubernetes: Google Kubernetes Engine(GKE)

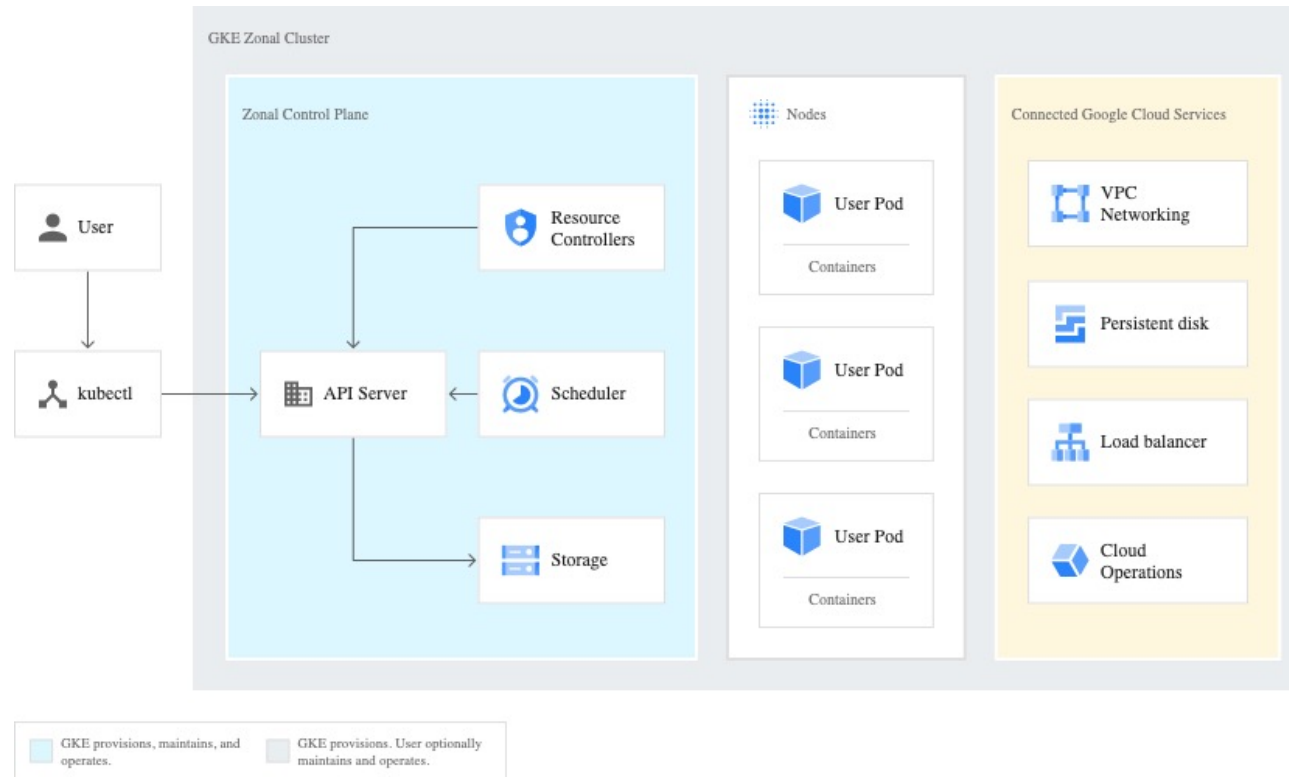
Google Kubernetes Engine (GKE) provides a managed environment for Kubernetes using Google infrastructure. Kubernetes draws on the same design principles that run popular Google services and provides the same benefits: automatic management, monitoring and liveness probes for application containers, automatic scaling, rolling updates, and more. GKE cluster control planes are automatically upgraded to run new versions of Kubernetes as those versions become stable to take advantage of newer features from the open source Kubernetes project.

Kubernetes on Google Cloud

When you run a GKE cluster, you gain the benefit of advanced cluster management features that Google Cloud provides. These include:

- Google Cloud's [load-balancing](#) for Compute Engine instances
- [Node pools](#) to designate subsets of nodes within a cluster for additional flexibility
- [Automatic scaling](#) of cluster's node instance count
- [Automatic upgrades](#) for cluster's node software
- [Node auto-repair](#) to maintain node health and availability
- [Logging and monitoring](#) with Google Cloud's operations suite for visibility into your cluster

GKE Cluster architecture



Modes of operation

GKE clusters have two modes of operation to choose from:

- **Autopilot:** Manages the entire cluster and node infrastructure for you. Autopilot provides a hands-off Kubernetes experience so that you can focus on your workloads and only pay for the resources required to run your applications. Autopilot clusters are pre-configured with an optimized cluster configuration that is ready for production workloads.
- **Standard:** Provides you with node configuration flexibility and full control over managing your clusters and node infrastructure. For clusters created using the Standard mode, you determine the configurations needed for your production workloads, and you pay for the nodes that you use.



Some of the clusters configuration:

Node pools: A *node pool* is a group of nodes within a cluster that all have the same configuration. Node pools use a NodeConfig specification. Each node in the pool has a Kubernetes node label, `cloud.google.com/gke-nodepool`, which has the node pool's name as its value.

Node images: When you create a GKE cluster or node pool, you can choose the operating system image that runs on each node. GKE Autopilot clusters use only the `cos_containerd` node image. The Container-Optimized OS from Google node images are based on a recent version of the Linux kernel and are optimized to enhance node security. Container-Optimized OS images are backed by a team at Google that can quickly patch images for security and iterate on features. The Container-Optimized OS images provides better support, security, and stability than other images.

OS	Node images
Container-Optimized OS	<ul style="list-style-type: none">Container-Optimized OS with Containerd (<code>cos_containerd</code>)Container-Optimized OS with Docker (<code>cos</code>)
Ubuntu	<ul style="list-style-type: none">Ubuntu with Containerd (<code>ubuntu_containerd</code>)Ubuntu with Docker (<code>ubuntu</code>)
Windows Server	<ul style="list-style-type: none">Windows Server LTSC (<code>windows_ltsc</code>)Windows Server SAC (<code>windows_sac</code>)

Cluster autoscaler: GKE's cluster autoscaler automatically resizes the number of nodes in a given node pool, based on the demands of workloads. You don't need to manually add or remove nodes or over-provision your node pools. Instead, you specify a minimum and maximum size for the node pool, and the rest is automatic.

- If Pods are unschedulable because there are not enough nodes in the node pool, cluster autoscaler adds nodes, up to the maximum size of the node pool.
- If nodes are under-utilized, and all Pods could be scheduled even with fewer nodes in the node pool, Cluster autoscaler removes nodes, down to the minimum size of the node pool. If the node cannot be drained gracefully after a timeout period (currently 10 minutes), the node is forcibly terminated. The grace period is not configurable for GKE clusters.

Horizontal Pod Autoscaling: HPA changes the shape of your Kubernetes workload by automatically increasing or decreasing the number of Pods in response to the workload's CPU or memory consumption, or in response to custom metrics

Vertical Pod Autoscaling frees you from having to think about what values to specify for a container's CPU requests and limits and memory requests and limits. The autoscaler can recommend values for CPU and memory requests and limits, or it can automatically update the values.



The Kubernetes networking model relies heavily on IP addresses. Services, Pods, containers, and nodes communicate using IP addresses and ports.

Kubernetes provides different types of load balancing to direct traffic to the correct Pods.

- **ClusterIP:** The IP address assigned to a Service. In other documents, it may be called the "Cluster IP". This address is stable for the lifetime of the Service.
- **Pod IP:** The IP address assigned to a given Pod, which is ephemeral.
- **Node IP:** The IP address assigned to a given node.

IP allocation:

Kubernetes uses various IP ranges to assign IP addresses to nodes, Pods, and Services.

- Each node has an IP address assigned from the cluster's Virtual Private Cloud (VPC) network.
- Each node has a pool of IP addresses that GKE assigns Pods running on that node (a /24 CIDR block by default). Each Pod has a single IP address assigned from the Pod CIDR range of its node. This IP address is shared by all containers running within the Pod, and connects them to other Pods running in the cluster.
- Each Service has an IP address, called the ClusterIP, assigned from the cluster's VPC network.

Services:

In Kubernetes, you can assign arbitrary key-value pairs called labels to any Kubernetes resource. Kubernetes uses labels to group multiple related Pods into a logical unit called a Service. A Service has a stable IP address and ports, and provides load balancing among the set of Pods whose labels match all the labels you define in the label selector when you create the Service.

Kube-Proxy:

- Kubernetes manages connectivity among Pods and Services using the kube-proxy component. This is deployed as a static Pod on each node by default.
- kube-proxy, which is **not** an in-line proxy, but an egress-based load- balancing controller, watches the Kubernetes API server and continually maps the ClusterIP to healthy Pods by adding and removing destination NAT (DNAT) rules to the node's iptables subsystem. When a container running in a Pod sends traffic to a Service's ClusterIP, the node selects a Pod at random and routes the traffic to that Pod.

Networking outside the cluster:

- **External load balancers** manage traffic coming from outside the cluster and outside your Google Cloud Virtual Private Cloud (VPC) network. They use forwarding rules associated with the Google Cloud network to route traffic to a Kubernetes node.
- **Internal load balancers** manage traffic coming from within the same VPC network. Like external load balancers, they use forwarding rules associated with the Google Cloud network to route traffic to a Kubernetes node.
- **HTTP(S) load balancers** are specialized external load balancers used for HTTP(S) traffic. They use an Ingress resource rather than a forwarding rule to route traffic to a Kubernetes node.



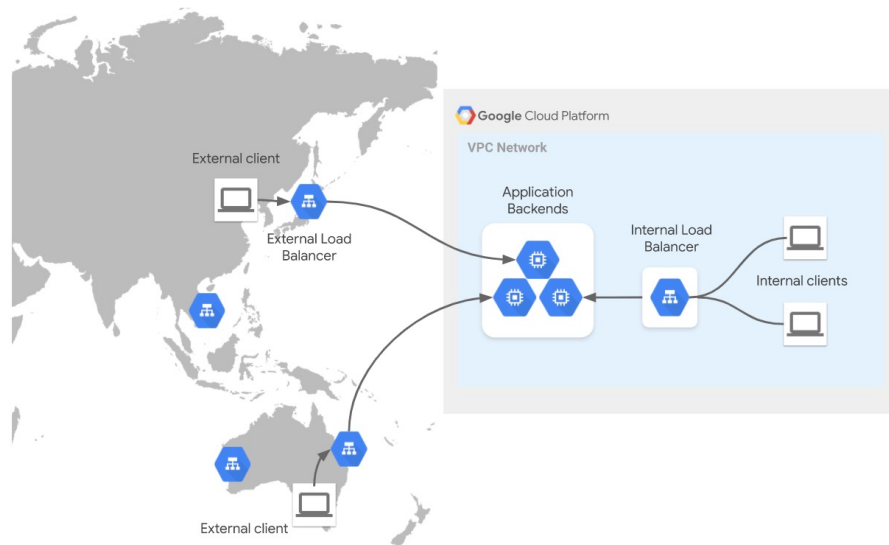
Kubernetes: GKE Load Balancing

Service networking is the publishing of applications in a way that abstracts the underlying ownership, implementation, or environment of the application that is being consumed by clients.

Exposing an application to clients involves three key elements of a Service:

1. **Frontend**
2. **Routing and load balancing**
3. **Backends**

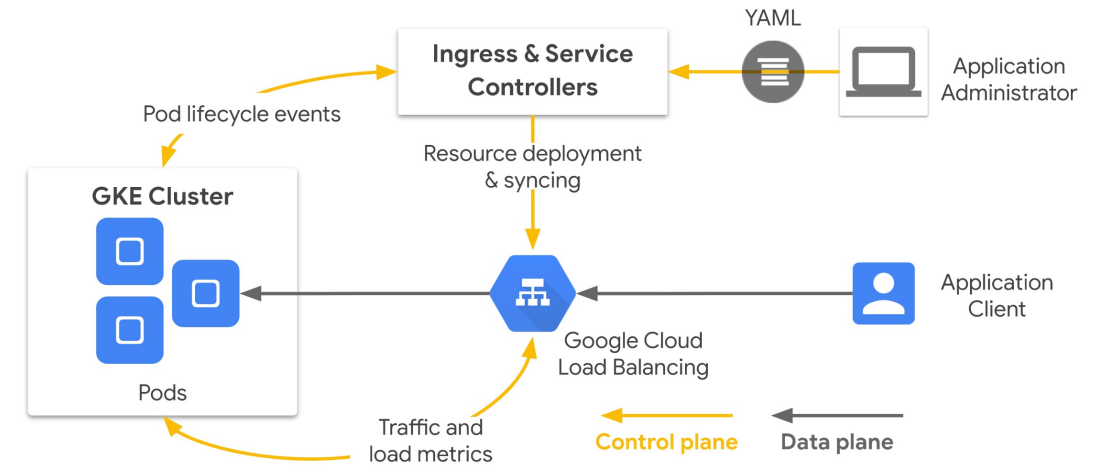
The following diagram illustrates these concepts for internal and external traffic flows in Google Cloud:



In this diagram, the External HTTP(S) Load Balancer is listening for traffic on the public internet through hundreds of Google points of presence around the world. This global frontend allows traffic to be terminated at the edge, close to clients, before it load balances the traffic to its backends in a Google data center.

The Internal HTTP(S) load balancer listens within the scope of your VPC network, allowing private communications to take place internally. These load balancer properties make them suited for different kinds of application use cases.

The following diagram illustrates how the GKE network controllers automate the creation of load balancers:



- As displayed in the diagram, an infrastructure or app admin deploys a declarative manifest against their GKE cluster.
- Ingress and Service controllers watch for GKE networking resources (such as Ingress or MultiClusterIngress objects) and deploy Google Cloud load balancers (plus IP addressing, firewall rules, and so on) based on the manifest.
- The controller continues managing the load balancer and backends based on environmental and traffic changes. Because of this, GKE load balancing becomes a dynamic and self-sustaining load balancer with a simple and developer-oriented interface.



Kubernetes: Deployments

Kubernetes objects can be created, updated, and deleted by storing multiple object configuration files in a directory and using `kubectl` apply to recursively create and update those objects as needed. A *Deployment* provides declarative updates for Pods and ReplicaSets. You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

Example Manifest File: `controllers/nginx-deployment.yaml`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

-- Kubernetes API version is used to create the object

-- The Object type

-- Details to identify the object

-- A Deployment named nginx-deployment is created, indicated by the `.metadata.name` field.

-- The Deployment creates three replicated Pods, indicated by the `.spec.replicas` field.

-- The `.spec.selector` field defines how the Deployment finds which Pods to manage.

-- The Pods are labeled `app: nginx` using the `.metadata.labels` field.

-- Pods run one container, `nginx`, which runs the [nginx Docker Hub](https://hub.docker.com/_/nginx/) image at version 1.14.2.

-- Create one container and name it `nginx` using the `.spec.template.spec.containers[0].name` field.