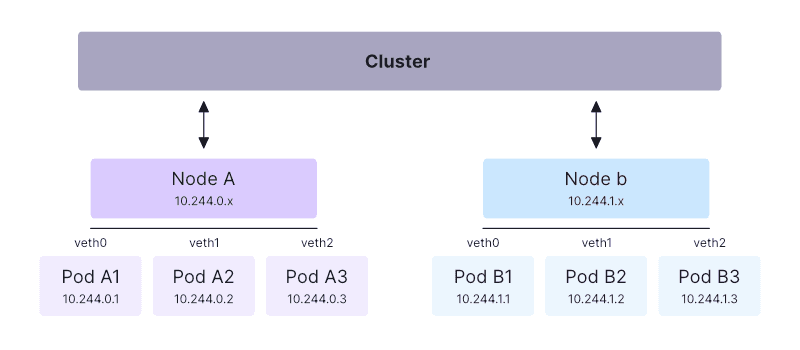
**What is Kubernetes networking?**

Kubernetes networking is the mechanism by which different resources within and outside your cluster are able to communicate with each other. Networking handles several different scenarios which we’ll explore below, but some key ones include communication between Pods, [communication between Kubernetes Services](https://spacelift.io/blog/kubernetes-service), and handling external traffic to the cluster.

Because Kubernetes is a distributed system, the network plane spans across your cluster’s physical Nodes. It uses a virtual overlay network that provides a flat structure for your cluster resources to connect to.

Below is an example of a Kubernetes networking diagram:



The Kubernetes networking implementation allocates IP addresses, assigns DNS names, and maps ports to your Pods and Services. This process is generally automatic—when using Kubernetes, you won’t normally have to manage these tasks on your network infrastructure or Node hosts.

At a high level, the Kubernetes network model works by allocating each Pod a unique IP address that resolves within your cluster. Pods can then communicate using their IP addresses, without requiring [NAT](https://www.cisco.com/c/en/us/products/routers/network-address-translation.html) or any other configuration.

This basic architecture is enhanced by the [Service](https://kubernetes.io/docs/concepts/services-networking/service) model, which allows traffic to route to any one of a set of Pods, as well as control methods, including[network policies](https://kubernetes.io/docs/concepts/services-networking/network-policies) that prevent undesirable Pod-to-Pod communications.

**What is the difference between physical/VM networking and Kubernetes networking?**

Kubernetes networking takes familiar networking principles and applies them to Kubernetes cluster environments. Kubernetes networking is simpler, more consistent, and more automated when compared to traditional networking models used for physical devices and VMs.

Whereas you’d previously have to manually configure new endpoints with IP addresses, firewall port openings, and DNS routes, Kubernetes provides all this functionality for your cluster’s workloads.

Developers and operators don’t need to understand how the network is implemented to successfully deploy resources and make them accessible to others. This simplifies setup, maintenance, and continual enforcement of security requirements by allowing all management to be performed within Kubernetes itself.

**What is the difference between Docker networking and Kubernetes networking?**

Kubernetes uses a flat networking model that’s designed to accommodate distributed systems. All Pods can communicate with each other, even when they’re deployed to different physical Nodes.

As a single-host containerization solution, [Docker takes a different approach to networking](https://spacelift.io/blog/docker-networking). It defaults to joining all your containers into a bridge network that connects to your host. You can create other networks for your containers using a variety of network types, including bridge, host (direct sharing of your host’s network stack), and overlay (distributed networking across Nodes, required for [Swarm](https://docs.docker.com/engine/swarm) environments).

Once they’re in a shared network, Docker containers can communicate with each other. Each container is assigned a network-internal IP address and DNS name that allows other network members to reach it. However, Docker does not automatically create port mappings from your host to your containers—you must [configure these](https://docs.docker.com/network/#published-ports) when you start your containers.

In summary, **Docker and Kubernetes networking have similarities, but each is adapted to its use case**. Docker is primarily concerned with single-node networking, which the bridged mode helps to simplify, whereas Kubernetes is a naturally distributed system that requires overlay networking.

This difference is apparent in how you prevent containers from communicating with each other: to stop Docker containers from interacting, you must ensure they’re in different networks. This contrasts with Kubernetes, where all Pods are automatically part of one overlay network, and traffic through the network is controlled using policy-based methods.

**Kubernetes networking architecture**

As we’ve mentioned, Kubernetes networking has a fundamentally flat structure with the following characteristics:

* All Pods are assigned their own IP addresses.
* Nodes run a root network namespace that bridges between the Pod interfaces. This allows all Pods to communicate with each other using their IP addresses, regardless of the Node they’re scheduled to.
* Communication does not depend on Network Address Translation (NAT), reducing complexity and improving portability.
* Pods are assigned their own network namespaces and interfaces. All communications with Pods go through their assigned interfaces.
* The cluster-level network layer maps the Node-level namespaces, allowing traffic to be correctly routed across Nodes.
* There’s no need to manually bind Pod ports to Nodes, although this is possible when required by assigning Pods a [hostPort](https://kubernetes.io/docs/concepts/configuration/overview/" \l "services" \t "_blank).

These concepts make Kubernetes networking predictable and consistent for both cluster users and administrators. The model imposed by Kubernetes ensures that all Pods can reliably access network connectivity without requiring any manual configuration.

**How Kubernetes allocates pod IP addresses**

Kubernetes allocates IP addresses to Pods using the [Classless Inter-Domain Routing (CIDR)](https://aws.amazon.com/what-is/cidr) system. This notation defines the subnet of IP addresses that will be available for use by your Pods. Each Pod is allocated an address from the CIDR range that applies to your cluster. You’ll need to specify the permitted CIDR range when you configure a new cluster’s networking layer.

Many Kubernetes networking plugins also support [IP Address Management (IPAM)](https://en.wikipedia.org/wiki/IP_address_management) functions, so you can manually assign IP addresses, prefixes, and pools. This facilitates advanced management of IP addresses in more complex networking scenarios

**How DNS works in Kubernetes clusters**

Kubernetes clusters include built-in DNS support. [CoreDNS](https://kubernetes.io/docs/tasks/administer-cluster/coredns" \t "_blank) is one of the most popular Kubernetes DNS providers; it comes enabled by default in many[Kubernetes distributions](https://spacelift.io/blog/install-kubernetes).

Kubernetes automatically assigns DNS names to Pods and Services in the following format:

* **Pod** – pod-ip-address.pod-namespace-name.pod.cluster-domain.example (e.g. 10.244.0.1.my-app.svc.cluster.local)
* **Service** – service-name.service-namespace-name.svc.cluster-domain.example (e.g. database.my-app.svc.cluster.local)

The applications running in your Pods should usually be configured to communicate with Services using their DNS names. Names are predictable, whereas a Service’s IP address will change if the Service is deleted and then replaced.

**Kubernetes network isolation with Network Policies**

Kubernetes defaults to allowing all Pods to communicate with each other. This is a security risk for clusters used for several independent apps, environments, teams, or customers.

[Kubernetes Network Policies](https://spacelift.io/blog/kubernetes-network-policy) are API objects that let you define the permitted ingress and egress routes for your Pods. The following simple example defines a policy that blocks traffic to Pods labeled app-component: database, unless the origin Pod is labeled app-component: api:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: database-network-policy

namespace: default

spec:

podSelector:

matchLabels:

app-component: database

policyTypes:

- Ingress

ingress:

- from:

- podSelector:

matchLabels:

app-component: api

Creating network policies for all your Pods [is good Kubernetes practice](https://spacelift.io/blog/kubernetes-best-practices). They prevent compromised Pods from directing malicious traffic to neighbors in your cluster.

**Types of Kubernetes networking and examples**

Kubernetes clusters need to handle several types of network access:

* **Pod-to-Pod communication**
* **Service-to-Pod communication**
* **External communication to services**

Each scenario uses a slightly different process to resolve the destination Node and Pod.

**1. Pod-to-Pod (same Node)**

Communication between two Pods running on the same Node is the simplest situation.

The Pod that initiates the network communication uses its default network interface to make a request to the target Pod’s IP address. The interface will be a virtual ethernet connection provided by Kubernetes, usually called eth0 on the Pod side and veth0 on the Node side. The second Pod on the Node will have veth1, the third Pod veth2, and so on:

* **Pod 0** — 10.244.0.1, veth0
* **Pod 1** — 10.244.0.1, veth1
* **Pod 2** — 10.244.0.2, veth2

The Node side of the connection acts as a network bridge. Upon receiving the request for the target Pod’s IP, the bridge checks if any of the devices attached to it (which are the Pod network interfaces veth0, veth1, and veth2, etc.) have the requested IP address:

Incoming Request: 10.244.0.1

Devices on the bridge:

Pod 0 – 10.244.0.1, veth0

Pod 1 – 10.244.0.1, veth1

Pod 2 – 10.244.0.2, veth2

Matching device: veth1

If there’s a match, then the data is forwarded to that network interface, which will belong to the correct Pod.

**2. Pod-to-Pod (different Nodes)**

Communication between Pods on different Nodes isn’t much more complex.

First, the previous Pod-to-Pod flow is initiated, but this will fail when the bridge finds none of its devices have the correct IP address. At this point, the resolution process will fall back to the default gateway on the Node, which will resolve to the cluster-level network layer.

Each Node in the cluster is assigned a unique IP address range; this could look like the following:

* **Node 1** – All Pods have IP addresses in the range 10.244.0.x
* **Node 2** – All Pods have IP addresses in the range 10.244.1.x
* **Node 3** – All Pods have IP addresses in the range 10.244.2.x

Thanks to these known ranges, the cluster can establish which Node is running the Pod and forward the network request on. The destination Node then follows the rest of the Pod-to-Pod routing procedure to select the target Pod’s network interface.

(Node 1) Incoming Request: 10.244.1.1

(Node 1) Devices on the bridge:

Pod 0 – 10.244.0.1, veth0

Pod 1 – 10.244.0.1, veth1

Pod 2 – 10.244.0.2, veth2

(Node 1) No matching interface, fallback to cluster-level network layer

(Cluster) Node IP ranges:

Node 1 – 10.244.0.x

Node 2 – 10.244.1.x

Node 3 – 10.244.2.x

(Cluster) Matching Node: Node 1; forward request

(Node 1) Devices on the bridge:

Pod 0 – 10.244.1.1, veth0

Pod 1 – 10.244.1.1, veth1

Pod 2 – 10.244.1.2, veth2

(Node 1) Matching device: veth1

The network connection is established to the correct Pod network interface on the remote Node. It’s notable that no NAT, proxying, or direct opening of ports was required for the communication, because all Pods in the cluster ultimately share a flat IP address space.

**3. Service-to-Pod**

Service networking results in multiple Pods being used to handle requests to one IP address or DNS name. The Service is assigned a virtual IP address that resolves to one of the available Pods.

Several different [service types](https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types) are supported, giving you options for a variety of use cases:

* **ClusterIP** – ClusterIP services expose the service on an IP address that’s only accessible within the cluster. Use these services for internal components such as databases, where the service is exclusively used by other Pods.
* **NodePort** – The service is exposed on a specified port on each Node in the cluster. Only one service can use a given NodePort at a time.
* **LoadBalancer** – Exposes the service externally using a Load Balancer that’s provisioned in your cloud provider account. (This service type is discussed in more depth below.)

Requests to services are proxied to the available Pods. The proxying is implemented by [kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy" \t "_blank), a Node-level control plane process that runs on each Node. Three different [proxy modes](https://kubernetes.io/docs/reference/networking/virtual-ips/#proxy-modes) are supported to change how the request is forwarded:

* **iptables** – Forwarding is configured using [iptables](https://linux.die.net/man/8/iptables) rules.
* **ipvs** – [Netlink](https://man7.org/linux/man-pages/man7/netlink.7.html" \t "_blank) is used to configure IPVS forwarding rules. Compared to iptables, this provides more traffic balancing options, such as selecting the Pod with the fewest connections or shortest queue.
* **kernelspace** – This option is used on Windows Nodes; it configures packet filtering rules for the Windows Virtual Filtering Platform (VFP), which is comparable to Linux’s iptables.

Once kube-proxy has forwarded the request, the network communication to the Pod proceeds as for a regular Pod-to-Pod request. The proxy step is only required to select a candidate Pod from those available in the Service.

**4. External-to-Service**

External traffic to Kubernetes clusters terminates at Services, within the cluster-level network layer. Direct internet access to Nodes isn’t possible by default.

Services are exposed by assigning them one or more [externalIPs](https://kubernetes.io/docs/tutorials/stateless-application/expose-external-ip-address" \t "_blank) (an IP that’s publicly accessible outside the cluster), or by using the [LoadBalancer service type](https://spacelift.io/blog/kubernetes-load-balancer" \t "_blank). The latter is the preferred approach; it uses your cloud provider’s API to provision a new load balancer infrastructure resource that routes external requests into your cluster.

When a load balancer is used, the load balancer’s IP address will map to the service that created it. When traffic enters the cluster, Kubernetes selects the matching service, then uses the Service-to-Pod flow described above to proxy the network request to a suitable Pod.

Most real-world external access is handled using [Ingress objects](https://spacelift.io/blog/kubernetes-ingress). These use one service to route HTTP traffic to other services in your cluster. A single Load Balancer service receives all the traffic, evaluates your Ingress rules, and then directs the request to the correct application service based on characteristics such as the HTTP URL, method, and host.

**How Kubernetes networking is implemented**

Kubernetes defines the networking functionality that a cluster requires, but it doesn’t include a built-in implementation. All the features discussed above are provided by [Container Network Interface (CNI)](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins) plugins. You have to manually install a CNI plugin when you set up a new Kubernetes cluster from scratch [using Kubeadm](https://spacelift.io/blog/install-kubernetes#how-to-set-up-kubernetes-cluster-with-kubeadm).

The CNI model improves modularity in the Kubernetes ecosystem. Different plugins offer unique combinations of features to accommodate a wide range of use cases and environments. Any CNI plugin will provide the standard set of Kubernetes networking features but can also expand on them, such as by integrating with other network technologies and services.

Some of the most popular CNI plugin options include:

[**Flannel**](https://github.com/flannel-io/flannel)

Flannel is developed by CoreOS. It aims to provide a simple CNI implementation with the fundamental features required by Kubernetes. Flannel is lightweight and easy to install and configure, making it a great option for local clusters and testing environments. However, Network Policies aren’t supported, so Flannel is unsuitable for production use unless you add a separate network policy controller.

[**Calico**](https://docs.tigera.io/calico/latest/getting-started/kubernetes)

Calico is a popular networking solution with support for several environments, including in Kubernetes as a CNI plugin. It’s widely adopted, proven, and capable of offering datacenter-level performance so it’s ready to support large-scale Kubernetes deployments.

[**Cilium**](https://cilium.io/)

Cilium is developed by the Linux kernel team. It uses [eBPF filtering rules](https://ebpf.io/" \t "_blank) to configure traffic flows on your hosts. Cilium supports all CNI features and can integrate with other CNI plugins. There’s also support for networking multiple independent clusters together.

[**Weave Net**](https://www.weave.works/oss/net)

Weaveworks’ Weave Net is a simple networking solution with support for several infrastructure types, including Kubernetes clusters. It prioritizes simplicity, providing a zero-configuration setup experience for quick and easy deployments.

In addition to these general-purpose options, specialized CNI implementations are also available for more specific situations. You can find a non-exhaustive list of currently known plugins in the [CNI documentation](https://www.cni.dev/docs).

**Key Points**

Kubernetes networking consists of a flat overlay network that all Pods automatically join. Pods can communicate with each other using their auto-assigned in-cluster addresses and DNS names.

For practical use, Pods that run network services should be exposed using a Kubernetes Service, which provides a single IP address and DNS name to route traffic to multiple Pods. This ensures the service can be scaled up with additional replicas while also providing the possibility of external access.

In this article, we’ve covered the fundamental Kubernetes networking architecture, how it’s implemented, and how the main cluster networking scenarios are achieved. You should now understand more about how Kubernetes networking works and what happens behind the scenes when you create a new Pod or Service.

Need a way to visualize your Kubernetes networking services and enforce guard rails? Check out [Spacelift](https://spacelift.io/" \t "_blank), our CI/CD-driven IaC management platform with Kubernetes environment support. Spacelift lets you apply policies, approval flows, and rules that prevent infrastructure misconfigurations, such as by requiring that all Kubernetes Pods are network-isolated using a Network Policy.