

**Exhibit 4 to
Intellectual Ventures I LLC and Intellectual Ventures II LLC
Amended Infringement Contentions**

**Amended Infringement Claim Chart for
U.S. Patent No. 7,949,785 (“the ’785 Patent”)**

The Accused Systems and Services include, without limitation, Comerica’s systems that utilize Kubernetes; all past, current and future systems and services that operate in the same or substantially similar manner as the specifically identified systems and services; and all past, current and future Comerica’s systems and services that have the same or substantially similar features as the specifically identified systems and services (“Comerica Count IV Systems and Services”).¹ Upon information and belief, Comerica’s systems and services directly infringe the asserted claims of the ’785 Patent through at least making and using its various systems that utilize Kubernetes as described in more detail below. Comerica also is a direct infringer through use via testing the claimed system. With regards to making the accused system, to the extent that Comerica utilizes a non-licensed public cloud provider, Comerica directs and controls and derives the beneficial use of making and using the claimed system. On information and belief, Comerica also directly infringes the asserted claims of the ’785 Patent by selling and offering to sell the Comerica Systems and Services through Comerica’s various offerings to customers. On information and belief, Comerica also indirectly infringes the ’785 Patent through direct infringement of Comerica customers, employees, and/or vendors and other third parties’ infringement that utilize Comerica’s offerings and systems and services that use Kubernetes. Comerica induces the direct infringement through its customers, employees, and/or vendors and other third parties’ use of Comerica’s Systems and Services. On information and belief, Comerica also contributes to the direct infringement through its vendors and other third parties’ uses of Comerica’s Systems and Services that utilize Kubernetes by providing components relating to Kubernetes that have no substantial non-infringing uses. Comerica has knowledge of the patent at least as of the time of the filing of its complaint and on information and belief provides instructions on how to use its various offerings, the use of which results in the performance of the ’785 Patent claims.

Pursuant to P.R. 3-1(b), IV identifies Kubernetes as used in Comerica’s systems and offerings as the Accused Instrumentality. Information regarding how specifically this Kubernetes is used for what product and offerings is non-public. IV intends to obtain

¹ For the avoidance of doubt, Plaintiffs do not accuse the public clouds of Defendant, to the extent those services are provided by a cloud provider with a license to Plaintiffs’ patents that covers Defendant’s activities.

IV accuses Comerica private clouds that implement Kubernetes and non-licensed public clouds that Comerica uses to support Kubernetes for its systems and services. To the extent any of these licenses are relevant to Defendant’s activities, Plaintiffs will meet and confer with Defendants about the impact of such license(s).



discovery regarding how the Kubernetes is used in Comerica products and services and will update its infringement positions once this discovery is completed. For example, IV served a 30(b)(6) deposition notice on Comerica on June 26, 2024, seeking information relating to Comerica's use and implementation of Kubernetes, including use and implementation of Kubernetes to support services that are offered to Comerica's customers and/or employees. This information requested by IV includes for example technical documents that show the operation of Comerica's use and implementation of Kubernetes, Comerica source code relating to Kubernetes, whether third parties have information relating to Comerica's use and implementation of Kubernetes, and other technical information. *See* Plaintiffs' Initial 30(b)(6) Notice to Defendant (Comerica) at 9-10. IV served an amended 30(b)(6) notice on August 16, 2024, requesting additional relevant information. *See generally* Plaintiffs' Amended 30(b)(6) Notice to Defendant (Comerica). IV has also served 30(b)(1) deposition notices for several technical Comerica employees that likely possess knowledge of Comerica's use and implementation of Kubernetes. To date, Comerica has not offered any deposition dates for either potential corporate witnesses or witnesses having personal knowledge of Comerica's use and implementation of Kubernetes. These amended contentions are based on information available to IV as of the date of service of these contentions, and are served without the benefit of even a single technical deposition to understand how Comerica uses and implements Kubernetes in its systems and services. Further, fact discovery remains open and Plaintiffs intend to serve additional discovery to better understand Comerica's use and implementation of Kubernetes. Accordingly, Plaintiffs reserve all rights to amend these contentions consistent with the local rules. Further, IV has filed a motion to compel seeking discovery including the identification of Comerica software applications that utilize Kubernetes, and reserve the right to supplement these contentions once the Court has resolved this pending dispute.



U.S. Patent No. 7,949,785	
Claim 30	Comerica Count IV Systems and Services
30pre. A virtual network manager, comprising:	Upon information and belief, Comerica’s systems include “a virtual network manager.” Comerica’s systems use Kubernetes to facilitate workloads and services. . The virtual network manager is the Kubernetes network model that includes functionality related to creating DNS records for Services and Pods. Kubernetes is made up of several components where deployment forms a cluster. Kubernetes API communicates with other components, like CoreDNS or kube-dns, which are responsible for handling communication from one pod to another through cluster-scoped virtual IP addresses. The Kubernetes network model consists of nodes that include pods with a unique cluster-wide IP address. Pods run on nodes and can communicate with one another, whether they are running on the same node in the cluster or different nodes. A node, or nodes, each contains an agent, such as the kubelet, that communicates with the pods running on that node.

The Kubernetes network model

The Kubernetes network model is built out of several pieces:

- Each **pod** in a cluster gets its own unique cluster-wide IP address.
 - A pod has its own private network namespace which is shared by all of the containers within the pod. Processes running in different containers in the same pod can communicate with each other over `localhost`.
- The *pod network* (also called a cluster network) handles communication between pods. It ensures that (barring intentional network segmentation):
 - All pods can communicate with all other pods, whether they are on the same **node** or on different nodes. Pods can communicate with each other directly, without the use of proxies or address translation (NAT).

On Windows, this rule does not apply to host-network pods.

 - Agents on a node (such as system daemons, or kubelet) can communicate with all pods on that node.

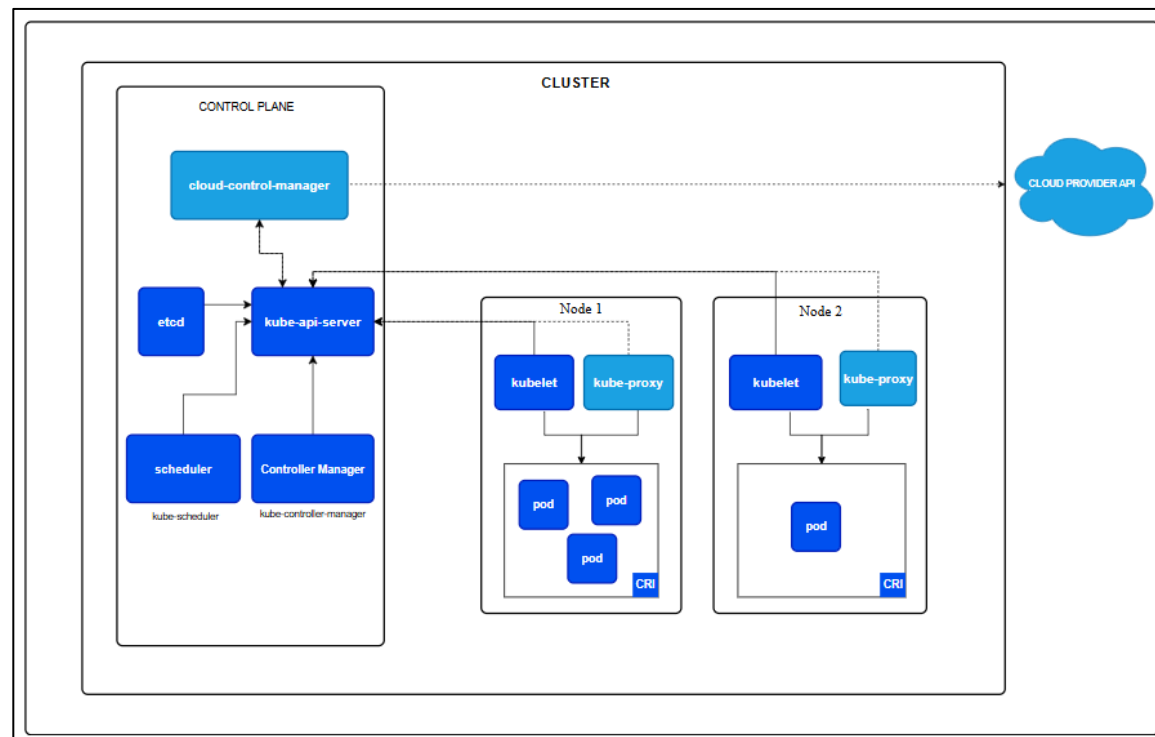
Source: <https://kubernetes.io/docs/concepts/services-networking/> (last accessed October 1, 2024).

For example, Comerica uses Kubernetes containers, ecosystems, and clusters to facilitate workloads and services. A Kubernetes cluster is composed of nodes, containing pods, and a control plane that includes the

Kubernetes API. The Kubernetes API can communicate with other components, like CoreDNS or kube-dns, which are responsible for handling internal communication from one pod to another through cluster-scoped virtual IP addresses.

Kubernetes is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

Source: <https://kubernetes.io/docs/concepts/overview/> (last accessed July 30, 2024).



Source: <https://kubernetes.io/docs/concepts/architecture/> (last accessed July 31, 2024).

[REDACTED]

For example, a pod or several pods running on a node are contained within a Kubernetes Cluster. A Service in Kubernetes defines and exposes a set of pods within the same node or between different nodes. The Service is accessible and exposed by a clusterIP, allowing for internal networking between workloads.

A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service is defined using YAML or JSON, like all Kubernetes object manifests. The set of Pods targeted by a Service is usually determined by a *label selector* (see below for why you might want a Service without including a *selector* in the spec).

Source: <https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/> (last accessed on November 10, 2023)

Kubernetes assigns this Service an IP address (the *cluster IP*), that is used by the virtual IP address mechanism. For more details on that mechanism, read [Virtual IPs and Service Proxies](#).

Source: <https://kubernetes.io/docs/concepts/services-networking/service/> (last accessed on November 10, 2023)

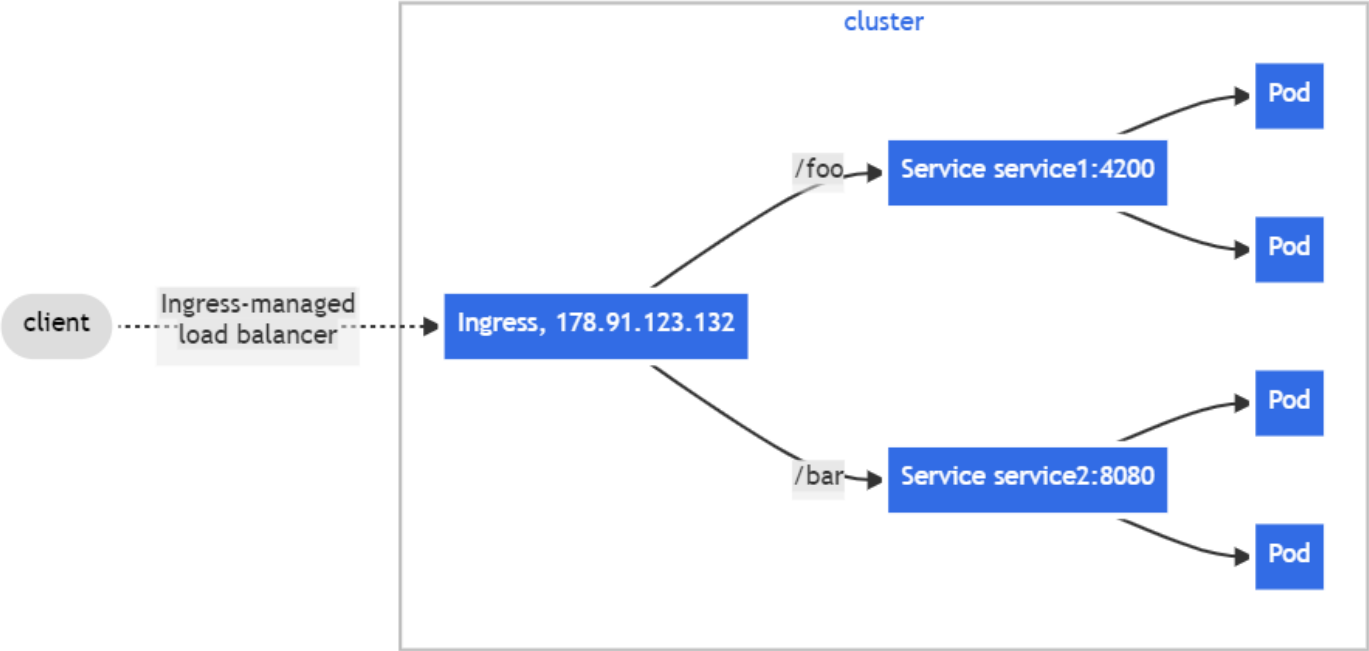
Upon information and belief, Comerica systems include the application BigID by utilizing Kubernetes clusters. Kubernetes clusters include the Kubernetes networking model to help in managing containerized applications across a distributed environment. The clusters reach out to a BigID ASP-hosted solution where traffic routes through the internet but does not use an internet IP for an endpoint. Several computers for the BigID application include the installation of kubernetes-cni, a container network interface package for cluster networking.

[REDACTED]

	See COM000000131. [REDACTED] <i>Id.</i> Comerica’s systems include Kubernetes functionality. <i>See, e.g.,</i> Source: https://careers.comerica.com/job/19419241/principal-engineer-cloud-automation-governance-and-engineering-cage-dallas-or-detroit-metro-auburn-hills-mi/ (last accessed on November 13, 2023). Source Code Evidence [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED]
30a. a network interface configured for data communication via a virtual network that is	Upon information and belief, Comerica systems utilize Kubernetes and include “a network interface configured for data communication via a virtual network that is defined by a domain name having an associated public network address.” For example, a Kubernetes cluster exposes HTTP/HTTPS from outside the cluster (e.g. a virtual network) to services inside the cluster (e.g., configured for data communication) by

defined by a domain name having an associated public network address;

employing an ingress-managed load balancer outside of the cluster (e.g., a network interface). The external load balancer includes a name and IP address (e.g. defined by a domain name having an associated public network address).





An Ingress may be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name-based virtual hosting. An [Ingress controller](#) is responsible for fulfilling the Ingress, usually with a load balancer, though it may also configure your edge router or additional frontends to help handle the traffic.

An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type [Service.Type=NodePort](#) or [Service.Type=LoadBalancer](#).

Source: <https://kubernetes.io/docs/concepts/services-networking/ingress/> (last accessed on October 1, 2024)

For example, a user can create an external load balancer. This requires a name to be associated with the load balancer and provides an externally accessible IP address (e.g., having an associated public network address).

Create an External Load Balancer

This page shows how to create an external load balancer.

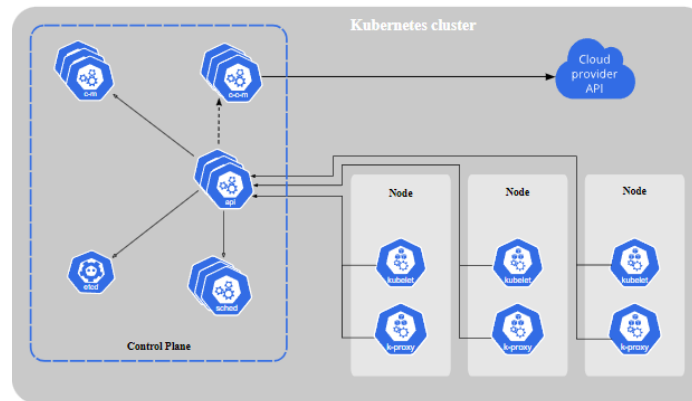
When creating a Service, you have the option of automatically creating a cloud load balancer. This provides an externally-accessible IP address that sends traffic to the correct port on your cluster nodes, *provided your cluster runs in a supported environment and is configured with the correct cloud load balancer provider package.*



	Create a Service from a manifest To create an external load balancer, add the following line to your Service manifest: <pre>type: LoadBalancer</pre>
--	--

Your manifest might then look like:

```
apiVersion: v1
kind: Service
metadata:
  name: example-service
spec:
  selector:
    app: example
  ports:
    - port: 8765
      targetPort: 9376
  type: LoadBalancer
```

Source: <https://kubernetes.io/docs/concepts/overview/components/> (last accessed on October 4, 2024)

DNS

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then all Pods should automatically be able to resolve Services by their DNS name.

External IPs

If there are external IPs that route to one or more cluster nodes, Kubernetes Services can be exposed on those externalIPs. When network traffic arrives into the cluster, with the external IP (as destination IP) and the port matching that Service, rules and routes that Kubernetes has configured ensure that the traffic is routed to one of the endpoints for that Service.

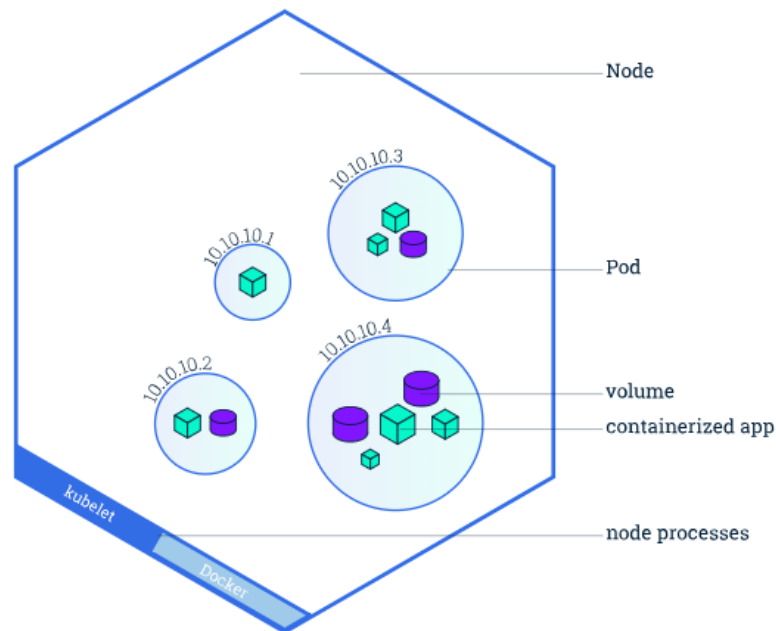
Source: <https://kubernetes.io/docs/concepts/services-networking/service/> (last accessed on November 10, 2023)

Source Code Evidence

[REDACTED]

	[REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED]
30b. a memory and a processor to implement a register module configured to register devices in a virtual network	Upon information and belief, Comerica systems utilize Kubernetes and include “a memory and a processor to implement a register module configured to register devices in a virtual network.” For example, a node can be a physical or virtual machine and includes a memory and processor. Nodes consisting of pod(s) (e.g., devices) are registered by the kubelet with the API Server (e.g., register module configured to register devices in a virtual network).

Node overview



Source: <https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/> (last accessed on October 3, 2024)

Kubernetes runs your workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods.

Source: <https://kubernetes.io/docs/concepts/architecture/nodes> (last accessed on November 10, 2023)

Management

There are two main ways to have Nodes added to the API server:

1. The kubelet on a node self-registers to the control plane
2. You (or another human user) manually add a Node object

After you create a Node object, or the kubelet on a node self-registers, the control plane checks whether the new Node object is valid. For

Source: <https://kubernetes.io/docs/concepts/architecture/nodes/> (last accessed on October 1, 2024)

Each node includes resources, such as memory and CPU(s) (e.g., a memory and a processor).

Resource capacity tracking

Node objects track information about the Node's resource capacity: for example, the amount of memory available and the number of CPUs.

Nodes that **self register** report their capacity during registration. If you **manually** add a Node, then you need to set the node's capacity information when you add it.

Source: <https://kubernetes.io/docs/concepts/architecture/nodes/> (last accessed on October 1, 2024)

The kubelet is an agent on each node that registers a newly created node with the API Server. By default, the self-registration of the node is attempted by the kubelet.

The kubelet is the primary "node agent" that runs on each node. It can register the node with the apiserver using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider.

Source: <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/> (last accessed August 1, 2024).

Self-registration of Nodes

When the kubelet flag `--register-node` is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.

Source: <https://kubernetes.io/docs/concepts/architecture/nodes/> (last accessed on October 1, 2024)

The control plane is a node in Kubernetes that manages the pods and worker nodes. The API Server runs on the control plane.

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. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

Source: <https://kubernetes.io/docs/concepts/overview/components/#node-components> (last accessed July 31, 2024).



Initializing your control-plane node

The control-plane node is the machine where the control plane components run, including etcd (the cluster database) and the API Server (which the kubectl command line tool communicates with).

Source: <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/> (last accessed on November 10, 2023)

Upon information and belief, Comerica systems utilize Kubernetes and include “a memory and a processor to implement a register module configured to register devices in a virtual network.” For example, a node can be a physical or virtual machine and includes a memory and processor. Pods (e.g., devices) run on nodes and use the node’s resources such as memory and CPU. Pods receive an IP address after being scheduled on a node through components such as the kubelet, Container Runtime Interface (CRI), and Container Network Interface (CNI) (e.g., register module configured to register devices in a virtual network).

Each node includes resources, such as memory and CPU(s) (e.g., a memory and a processor).

Resource capacity tracking

Node objects track information about the Node's resource capacity: for example, the amount of memory available and the number of CPUs. Nodes that **self register** report their capacity during registration. If you **manually** add a Node, then you need to set the node's capacity information when you add it.

Source: <https://kubernetes.io/docs/concepts/architecture/nodes/> (last accessed on October 1, 2024)

A Container Network Interface (CNI) is required to be installed to implement the Kubernetes network model.

Network Plugins

Kubernetes (version 1.3 through to the latest 1.31, and likely onwards) lets you use [Container Network Interface](#) (CNI) plugins for cluster networking. You must use a CNI plugin that is compatible with your cluster and that suits your needs. Different plugins are available (both open- and closed- source) in the wider Kubernetes ecosystem.

A CNI plugin is required to implement the [Kubernetes network model](#).

You must use a CNI plugin that is compatible with the [v0.4.0](#) or later releases of the CNI specification. The Kubernetes project recommends using a plugin that is compatible with the [v1.0.0](#) CNI specification (plugins can be compatible with multiple spec versions).

Source: <https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/> (last accessed on October 3, 2024)

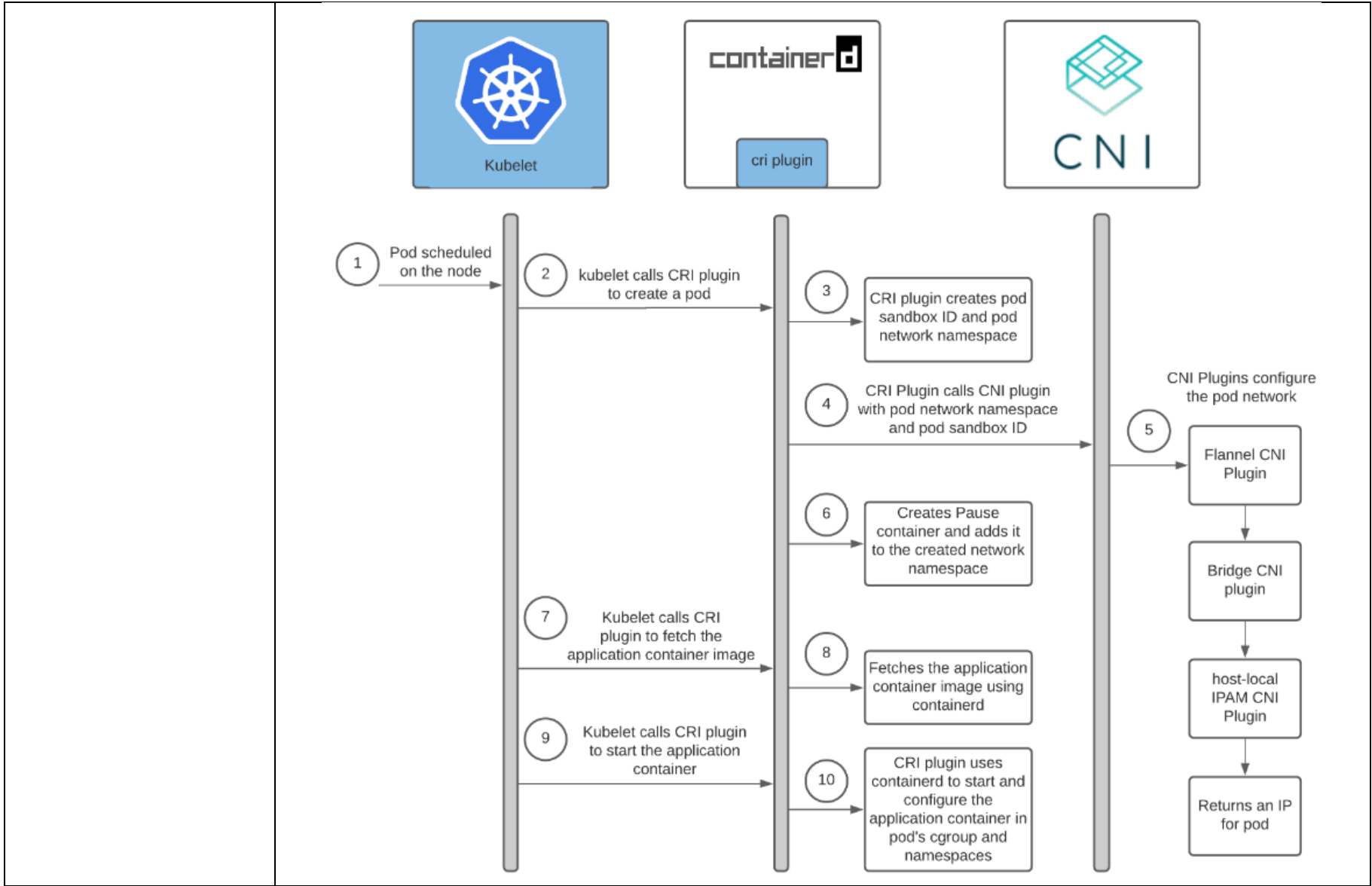
Kubernetes IP address ranges

Kubernetes clusters require to allocate non-overlapping IP addresses for Pods, Services and Nodes, from a range of available addresses configured in the following components:

- The network plugin is configured to assign IP addresses to Pods.
- The kube-apiserver is configured to assign IP addresses to Services.
- The kubelet or the cloud-controller-manager is configured to assign IP addresses to Nodes.

Source: <https://kubernetes.io/docs/concepts/cluster-administration/networking/> (last accessed on October 3, 2024)

The image provided below shows the process of a pod being assigned an IP address (number 5 in the image) by the CNI plugin.



[REDACTED]

	Source: https://ronaknathani.com/blog/2020/08/how-a-kubernetes-pod-gets-an-ip-address/ (last accessed on October 3, 2024) Source Code Evidence [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED] [REDACTED]
30c. the register module further configured to: receive a registration request from an agent associated with a device;	Upon information and belief, Comerica systems include a “register module further configured to receive a registration request from an agent associated with a device.” For example, nodes consisting of pod(s) (e.g., device) are registered by the kubelet (e.g., agent associated with a device) with the API Server (e.g. register module). <u>Kubernetes runs your workload by placing containers into Pods to run on <i>Nodes</i>. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the control plane and contains the services necessary to run Pods.</u> Source: https://kubernetes.io/docs/concepts/architecture/nodes (last accessed on November 10, 2023)

Management

There are two main ways to have Nodes added to the API server:

1. The kubelet on a node self-registers to the control plane
2. You (or another human user) manually add a Node object

After you create a Node object, or the kubelet on a node self-registers, the control plane checks whether the new Node object is valid. For

Source: <https://kubernetes.io/docs/concepts/architecture/nodes/> (last accessed on October 1, 2024)

By default, the kubelet flag parameter is set to true and will attempt self-registration with the API Server.

Self-registration of Nodes [↗](#)

When the kubelet flag `--register-node` is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.

For self-registration, the kubelet is started with the following options:

- `--kubeconfig` - Path to credentials to authenticate itself to the API server.
- `--cloud-provider` - How to talk to a cloud provider to read metadata about itself.
- `--register-node` - Automatically register with the API server.
- `--register-with-taints` - Register the node with the given list of taints (comma separated `<key>=<value>:<effect>`).

No-op if `register-node` is false.

- `--node-ip` - Optional comma-separated list of the IP addresses for the node. You can only specify a single address for each address family. For example, in a single-stack IPv4 cluster, you set this value to be the IPv4 address that the kubelet should use for the node. See [configure IPv4/IPv6 dual stack](#) for details of running a dual-stack cluster.

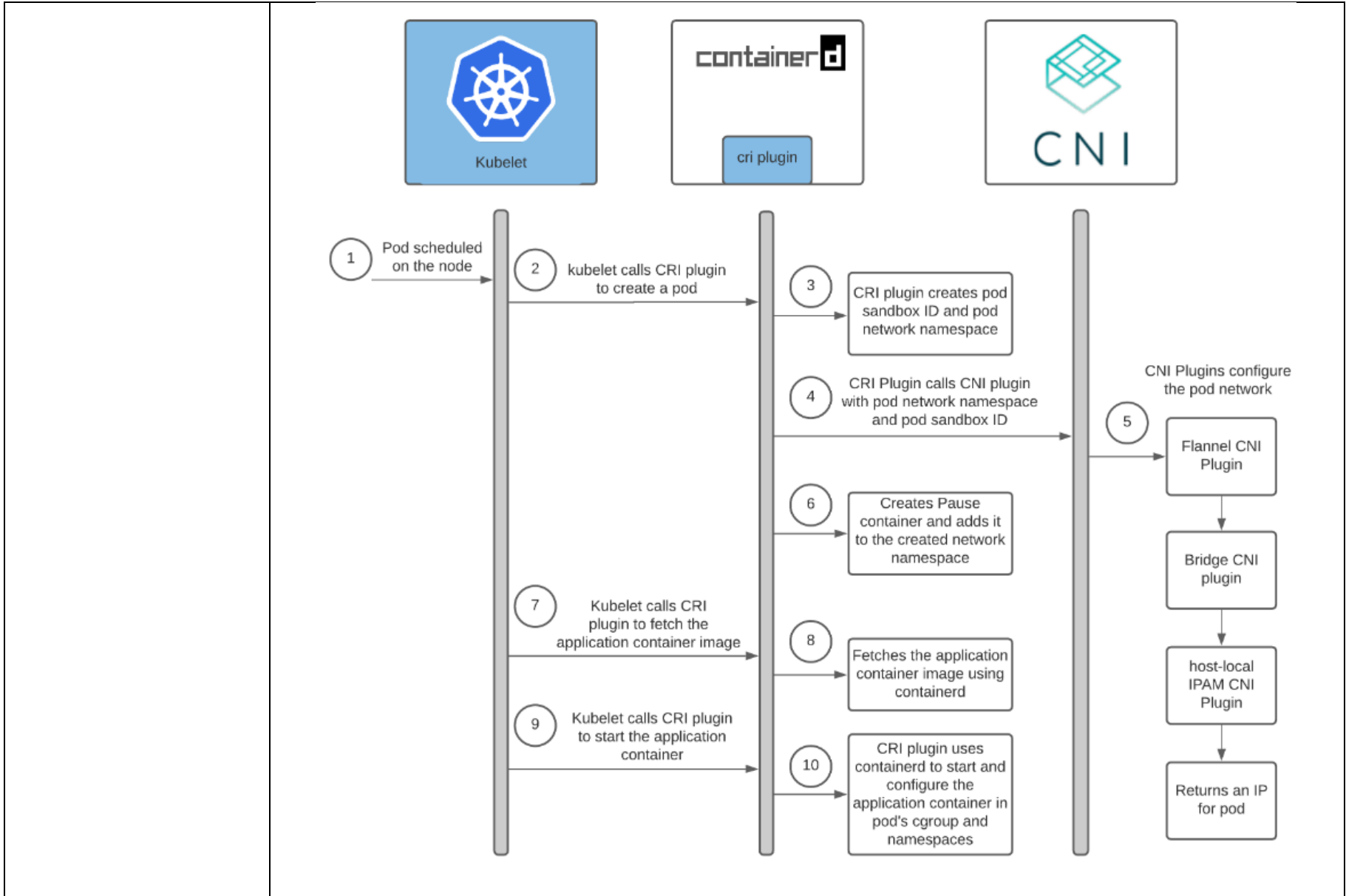
If you don't provide this argument, the kubelet uses the node's default IPv4 address, if any; if the node has no IPv4 addresses then the kubelet uses the node's default IPv6 address.

- `--node-labels` - Labels to add when registering the node in the cluster (see label restrictions enforced by the [NodeRestriction admission plugin](#)).
- `--node-status-update-frequency` - Specifies how often kubelet posts its node status to the API server.

Source: <https://kubernetes.io/docs/concepts/architecture/nodes/> (last accessed on October 1, 2024)



	Upon information and belief, Comerica systems include a “register module further configured to receive a registration request from an agent associated with a device.” For example, the kubelet (e.g., agent associated with the device) calls the Container Runtime Interface (CRI) plugin to create a pod (e.g., receive a registration request) after the pod (e.g., device) is scheduled on a node.
--	---



Source: <https://ronaknathani.com/blog/2020/08/how-a-kubernetes-pod-gets-an-ip-address/> (last accessed on October 3, 2024)

A PodSpec in Kubernetes is a yaml or json file that describes a pod object. The kubelet uses these to ensure pods are healthy and running as described in the respective PodSpec.

The kubelet is the primary "node agent" that runs on each node. It can register the node with the apiserver using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider.

The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

Source: <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/> (last accessed on October 3, 2024)

Source Code Evidence

[REDACTED]

	[Redacted]
30d. distribute a virtual network address to the device when the device is registered in the virtual network, the device being identified to other devices in the virtual network by the virtual network address; and	Upon information and belief, Comerica’s systems include a register module further configured to “distribute a virtual network address to the device when the device is registered in the virtual network, the device being identified to other devices in the virtual network by the virtual network address.” The kubelet on a node assigns an IP address to the said node (e.g., distribute a virtual network address to the device when the device is registered in the virtual network). These IP addresses are required to be non-overlapping so nodes can be identified by other resources in the network for data communication (e.g., the device being identified to other devices in the virtual network by the virtual network address). Kubernetes IP address ranges Kubernetes clusters require to allocate non-overlapping IP addresses for Pods, Services and Nodes, from a range of available addresses configured in the following components: • The network plugin is configured to assign IP addresses to Pods. • The kube-apiserver is configured to assign IP addresses to Services. • The kubelet or the cloud-controller-manager is configured to assign IP addresses to Nodes. Source: https://kubernetes.io/docs/concepts/cluster-administration/networking/ (last accessed on October 3, 2024)



- `--node-ip` - Optional comma-separated list of the IP addresses for the node. You can only specify a single address for each address family. For example, in a single-stack IPv4 cluster, you set this value to be the IPv4 address that the kubelet should use for the node. See [configure IPv4/IPv6 dual stack](#) for details of running a dual-stack cluster.

If you don't provide this argument, the kubelet uses the node's default IPv4 address, if any; if the node has no IPv4 addresses then the kubelet uses the node's default IPv6 address.

Source: <https://kubernetes.io/docs/concepts/architecture/nodes/> (last accessed on October 1, 2024)

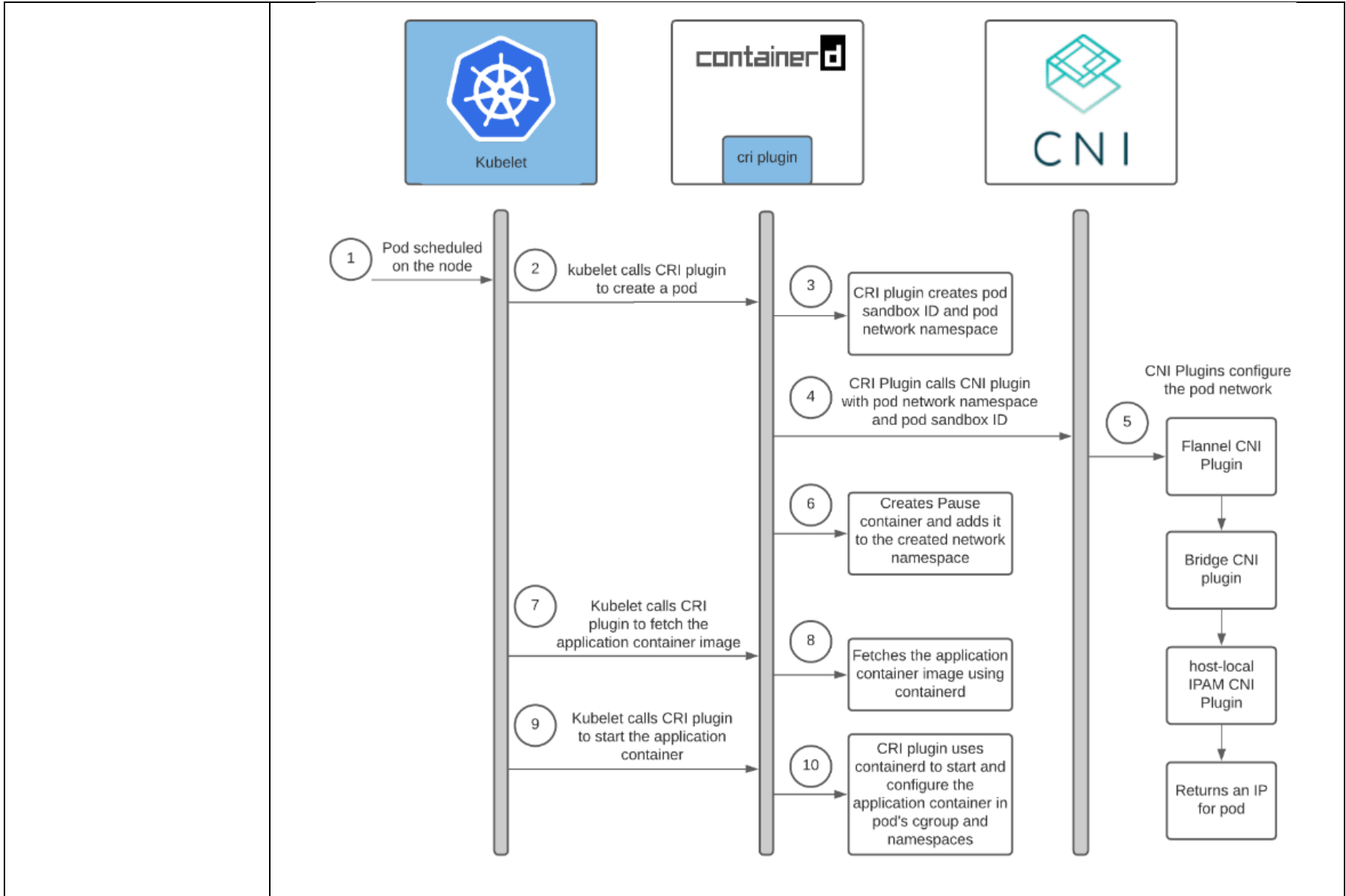
Upon information and belief, Comerica's systems include a register module further configured to "distribute a virtual network address to the device when the device is registered in the virtual network, the device being identified to other devices in the virtual network by the virtual network address." Once a pod is scheduled on a node, it receives an IP address by the Container Network Interface (CNI) (e.g., distribute a virtual network address to the device when the device is registered in the virtual network). These IP addresses are required to be non-overlapping so pods can be identified by other resources in the network for data communication (e.g., the device being identified to other devices in the virtual network by the virtual network address).

Kubernetes IP address ranges

Kubernetes clusters require to allocate non-overlapping IP addresses for Pods, Services and Nodes, from a range of available addresses configured in the following components:

- The network plugin is configured to assign IP addresses to Pods.
- The kube-apiserver is configured to assign IP addresses to Services.
- The kubelet or the cloud-controller-manager is configured to assign IP addresses to Nodes.

Source: <https://kubernetes.io/docs/concepts/cluster-administration/networking/> (last accessed on October 3, 2024)





Source: <https://ronaknathani.com/blog/2020/08/how-a-kubernetes-pod-gets-an-ip-address/> (last accessed on October 3, 2024)

Additionally, a Service in Kubernetes defines a set of pods and policies for accessing them. Services are used to expose pods on a network for data communication.

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Kubernetes gives every pod its own cluster-private IP address, so you do not need to explicitly create links between pods or map container ports to host ports. This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT. The rest of this document elaborates on how you can run reliable services on such a networking model.

Source: <https://kubernetes.io/docs/tutorials/services/connect-applications-service/> (last accessed August 1, 2024).

Services are assigned an IP address when created and allow clients to connect using this virtual IP address.



	Using a Service to Expose Your App Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic. Services can be exposed in different ways by specifying a <code>type</code> in the <code>spec</code> of the Service: • <u>ClusterIP (default)</u> - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster. Source: https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/ (last accessed on November 10, 2023) Service ClusterIP allocation In Kubernetes, <u>Services</u> are an abstract way to expose an application running on a set of Pods. Services can have a cluster-scoped virtual IP address (using a Service of <code>type: clusterIP</code>). Clients can connect using that virtual IP address, and Kubernetes then load-balances traffic to that Service across the different backing Pods. Source: https://kubernetes.io/docs/concepts/services-networking/cluster-ip-allocation/ (last accessed on November 10, 2023) VIP <u>a virtual IP address, such as the one assigned to every Service in Kubernetes</u> Source: https://kubernetes.io/docs/tutorials/services/source-ip/ (last accessed on November 10, 2023)
--	--

30e. a DNS server for the virtual network, the DNS server configured to receive a DNS request from a first device in the virtual network and return a network address associated with a network route director,

Upon information and belief, Comerica's systems include "a DNS server for the virtual network, the DNS server configured to receive a DNS request from a first device in the virtual network." For example, the Kubernetes DNS functionality, which includes CoreDNS (e.g., a DNS server) functions as the extensible DNS server for a given Kubernetes cluster DNS. For example, a client/frontend pod (e.g., a first device) sends a DNS query (e.g., DNS request), which is forwarded to CoreDNS (or, alternatively, Kube-dns). This Kubernetes DNS functionality is implemented, in part, using resolver routines associated with the requesting pod (see e.g., resolve.conf).

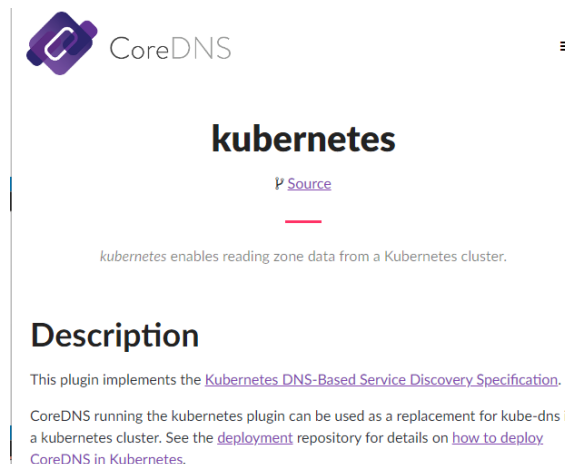
Using CoreDNS for Service Discovery

About CoreDNS

CoreDNS is a flexible, extensible DNS server that can serve as the Kubernetes cluster DNS. Like Kubernetes, the CoreDNS project is hosted by the CNCF.

You can use CoreDNS instead of kube-dns in your cluster by replacing kube-dns in an existing deployment, or by using tools like kubeadm that will deploy and upgrade the cluster for you.

Source: <https://kubernetes.io/docs/tasks/administer-cluster/coredns/> (last accessed on November 10, 2023)



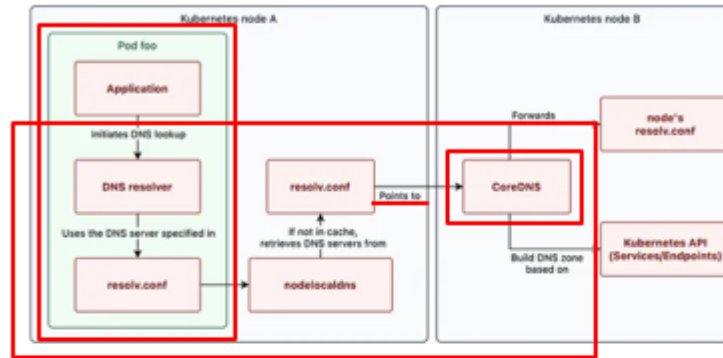


	 CoreDNS Metadata The <code>kubernetes</code> plugin will publish the following metadata, if the <code>metadata</code> plugin is also enabled: • <code>kubernetes/endpoint</code>: the endpoint name in the query • <code>kubernetes/kind</code>: the resource kind (pod or svc) in the query • <code>kubernetes/namespace</code>: the namespace in the query • <code>kubernetes/port-name</code>: the port name in an SRV query • <code>kubernetes/protocol</code>: the protocol in an SRV query • <code>kubernetes/service</code>: the service name in the query • <code>kubernetes/client-namespace</code>: the client pod's namespace (see requirements below) • <code>kubernetes/client-pod-name</code>: the client pod's name (see requirements below) The <code>kubernetes/client-namespace</code> and <code>kubernetes/client-pod-name</code> metadata work by reconciling the client IP address in the DNS request packet to a known pod IP address. Therefore the following is required: • <code>pods-verified</code> mode must be enabled • the remote IP address in the DNS packet received by CoreDNS must be the IP address of the Pod that sent the request. Source: https://coredns.io/plugins/kubernetes/ (last accessed on November 11, 2023). The example provided below shows the process of a DNS query made by a pod in “node A” and resolver routines forwarding the query to the CoreDNS on “node B.”
--	--



	Pod foo When a pod sends an API request to a service within the same Kubernetes cluster, it must first resolve the IP address of the service. To do this, <u>the pod performs a DNS lookup using the DNS server specified in its <code>/etc/resolv.conf</code> configuration file.</u> <u>This file, which is provisioned by the Kubelet, defines the settings for DNS lookups in the pod. It contains a reference to the cluster DNS server.</u> By default, this configuration file looks something like this: <pre>search namespace.svc.cluster.local svc.cluster.local cluster.local nameserver 10.123.0.10 options ndots:5</pre>
--	--

DNS lookups on services



When a pod performs a DNS lookup, the query is first sent to the local DNS resolver in the pod. This resolver uses the `resolv.conf` configuration file. In this file, the `nodeLocaldns` server is set up as the default recursive DNS resolver, which acts as a cache.

If this cache does not contain the IP address for the requested hostname, the query is forwarded to the cluster DNS server (CoreDNS).

This DNS server determines the IP address by consulting the Kubernetes service registry. This registry contains a mapping of service names to their corresponding IP addresses. This allows the cluster DNS server to return the correct IP address to the requesting pod.

Any domains that are queried but are not in the Kubernetes service registry are forwarded to an upstream DNS server.

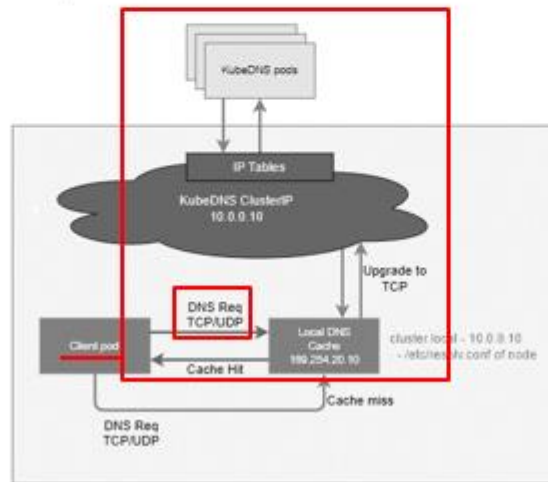
We will go through each of these components in more detail step-by-step.

Source: <https://medium.com/@seifeddinerajhi/connecting-the-dots-understanding-how-pods-talk-in-kubernetes-networks-992fa69fbbeb> (last accessed on November 10, 2023)

The NodeLocal DNSCache is a node-local cache to improve performance for DNS functionality within the cluster.

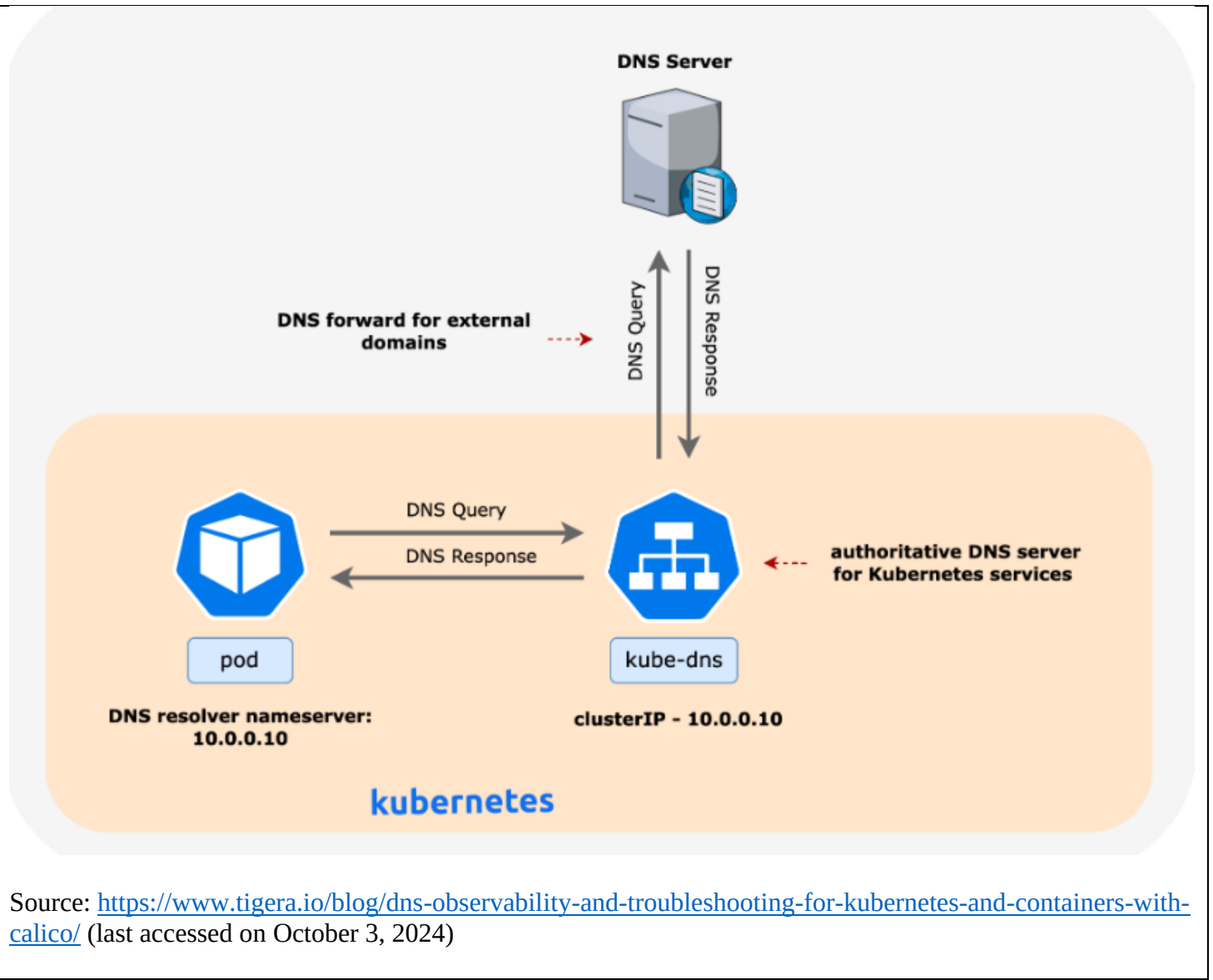
Introduction

NodeLocal DNSCache improves Cluster DNS performance by running a DNS caching agent on cluster nodes as a DaemonSet. In today's architecture, Pods in 'ClusterFirst' DNS mode reach out to a kube-dns serviceIP for DNS queries. This is translated to a kube-dns/CoreDNS endpoint via iptables rules added by kube-proxy. With this new architecture, Pods will reach out to the DNS caching agent running on the same node, thereby avoiding iptables DNAT rules and connection tracking. The local caching agent will query kube-dns service for cache misses of cluster hostnames ("cluster.local" suffix by default).



Source: <https://kubernetes.io/docs/tasks/administer-cluster/nodelocaldns/> (last accessed on November 10, 2023)

	resolv.conf(5)File Formats Manualresolv.conf(5) NAME top resolv.conf - resolver configuration file SYNOPSIS top /etc/resolv.conf DESCRIPTION top The resolver is a set of routines in the C library that provide access to the Internet Domain Name System (DNS). The resolver configuration file contains information that is read by the resolver routines the first time they are invoked by a process. The file is designed to be human readable and contains a list of keywords with values that provide various types of resolver information. The configuration file is considered a trusted source of DNS information; see the <code>trust-ad</code> option below for details. Source: https://man7.org/linux/man-pages/man5/resolv.conf.5.html (last accessed on November 10, 2023) If the DNS query cannot be resolved by CoreDNS, the DNS query (e.g., DNS request) is forwarded to an external upstream DNS server (e.g., network route director).
--	---



In the example shown below, the DNS server's answer to a DNS query (e.g., DNS request) involves a DNS record, which includes an indication of a DNS zone (e.g., a network address associated with a network route director), a namespace, and a cluster or endpoint IP.

Any DNS-based service discovery solution for Kubernetes must provide the resource records (RRs) described below to be considered compliant with this specification.

2.1 - Definitions

In the RR descriptions below, values not in angle brackets, `< >`, are literals. The meaning of the values in angle brackets are defined below or in the description of the specific record.

- `<zone>` = configured cluster domain, e.g. cluster.local
- `<ns>` = a Namespace
- `<ttl>` = the standard DNS time-to-live value for the record

In the RR descriptions below, the following definitions should be used for words in *italics*.

hostname

- In order of precedence, the *hostname* of an endpoint is:
 - The value of the endpoint's `hostname` field.
 - A unique, system-assigned identifier for the endpoint. The exact format and source of this identifier is not prescribed by this specification. However, it must be possible to use this to identify a specific endpoint in the context of a Service. This is used in the event no explicit endpoint hostname is defined.

ready

- An endpoint is considered *ready* if its address is in the `addresses` field of the EndpointSubset object, or the corresponding service has the `service.alpha.kubernetes.io/tolerate-unready-endpoints` annotation set to `true`.

All comparisons between query data and data in Kubernetes are **case-insensitive**.

2.3 - Records for a Service with ClusterIP

Given a Service named `<service>` in Namespace `<ns>` with ClusterIP `<cluster-ip>`, the following records must exist.

2.3.1 - A / AAAA Record

If the `<cluster-ip>` is an IPv4 address, an `A` record of the following form must exist.

- Record Format:
 - `<service>.<ns>.svc.<zone>. <ttl> IN A <cluster-ip>`
- Question Example:
 - `kubernetes.default.svc.cluster.local. IN A`
- Answer Example:
 - `kubernetes.default.svc.cluster.local. 4 IN A 10.3.0.1`

2.4 - Records for a Headless Service

Given a headless Service `<service>` in Namespace `<ns>` (i.e., a Service with no ClusterIP), the following records must exist.

2.4.1 - A / AAAA Records

There must be an `A` record for each *ready* endpoint of the headless Service with IPv4 address `<endpoint-ip>` as shown below. If there are no *ready* endpoints for the headless Service, the answer should be `NXDOMAIN`.

- Record Format:
 - `<service>.<ns>.svc.<zone>. <ttl> IN A <endpoint-ip>`
- Question Example:
 - `headless.default.svc.cluster.local. IN A`
- Answer Example:

```
headless.default.svc.cluster.local. 4 IN A 10.3.0.1
headless.default.svc.cluster.local. 4 IN A 10.3.0.2
headless.default.svc.cluster.local. 4 IN A 10.3.0.3
```



Source: <https://github.com/kubernetes/dns/blob/master/docs/specification.md> (last accessed on October 3, 2024)

A “zone” value is included in the previous example of a DNS record response to a DNS query (e.g., DNS request). The example of the Kubernetes plugin configuration below states that “zones specify the DNS zones for which the Kubernetes plugin is authoritative.” A DNS “zone” can be returned in a DNS response to a DNS query (e.g., return a network address associated with a network route director).

kubernetes: This is an essential plugin for implementing DNS-based service discovery in the Kubernetes cluster. It supports many directives that are worth discussing.

```
kubernetes {cluster.local in-addr.arpa ip6.arpa} {  
  pods insecure ← Mode to handle IP-based pod A records  
  fallthrough in-addr.arpa ip6.arpa ← Pass unhandled queries to next plugin in the chain (for specified zones only)  
  ttl 30 ← Set the custom TTL of 30 seconds for responses  
}
```

CoreDNS Kubernetes plugin configuration

The plugin's syntax follows the pattern *kubernetes [zones]{}*, where *zones* specify the DNS zones for which the kubernetes plugin is authoritative. As per the screenshot, the *cluster.local* zone is for the default cluster domain, while *in-addr.arpa* and *ip6.arpa* are the reverse DNS zones for the IPv4 and IPv6 addresses, respectively. The curly braces include various directives for adjusting the plugin's behavior.

Source: <https://4sysops.com/archives/kubernetes-coredns/> (last accessed on October 3, 2024)



30f. a private network address associated with a second device in the virtual network, and a virtual network address associated with the second device.

Upon information and belief, Comerica’s systems include “a private network address associated with a second device in the virtual network, and a virtual network address associated with the second device.” In the example shown below, the DNS server’s answer to a DNS query involves a DNS record, which includes an indication of the DNS zone, a namespace (e.g., a private network address associated with a second device), and a cluster or endpoint IP (e.g., virtual network address associated with the second device).

Any DNS-based service discovery solution for Kubernetes must provide the resource records (RRs) described below to be considered compliant with this specification.

2.1 - Definitions

In the RR descriptions below, values not in angle brackets, `< >`, are literals. The meaning of the values in angle brackets are defined below or in the description of the specific record.

- `<zone>` = configured cluster domain, e.g. cluster.local
- `<ns>` = a Namespace
- `<ttl>` = the standard DNS time-to-live value for the record

In the RR descriptions below, the following definitions should be used for words in *italics*.

hostname

- In order of precedence, the *hostname* of an endpoint is:
 - The value of the endpoint's `hostname` field.
 - A unique, system-assigned identifier for the endpoint. The exact format and source of this identifier is not prescribed by this specification. However, it must be possible to use this to identify a specific endpoint in the context of a Service. This is used in the event no explicit endpoint hostname is defined.

ready

- An endpoint is considered *ready* if its address is in the `addresses` field of the EndpointSubset object, or the corresponding service has the `service.alpha.kubernetes.io/tolerate-unready-endpoints` annotation set to `true`.

All comparisons between query data and data in Kubernetes are **case-insensitive**.

2.3 - Records for a Service with ClusterIP

Given a Service named `<service>` in Namespace `<ns>` with ClusterIP `<cluster-ip>`, the following records must exist.

2.3.1 - A / AAAA Record

If the `<cluster-ip>` is an IPv4 address, an `A` record of the following form must exist.

- Record Format:
 - `<service>.<ns>.svc.<zone>. <ttl> IN A <cluster-ip>`
- Question Example:
 - `kubernetes.default.svc.cluster.local. IN A`
- Answer Example:
 - `kubernetes.default.svc.cluster.local. 4 IN A 10.3.0.1`

2.4 - Records for a Headless Service

Given a headless Service `<service>` in Namespace `<ns>` (i.e., a Service with no ClusterIP), the following records must exist.

2.4.1 - A / AAAA Records

There must be an `A` record for each *ready* endpoint of the headless Service with IPv4 address `<endpoint-ip>` as shown below. If there are no *ready* endpoints for the headless Service, the answer should be `NXDOMAIN`.

- Record Format:
 - `<service>.<ns>.svc.<zone>. <ttl> IN A <endpoint-ip>`
- Question Example:
 - `headless.default.svc.cluster.local. IN A`
- Answer Example:

```
headless.default.svc.cluster.local. 4 IN A 10.3.0.1
headless.default.svc.cluster.local. 4 IN A 10.3.0.2
headless.default.svc.cluster.local. 4 IN A 10.3.0.3
```



Source: <https://github.com/kubernetes/dns/blob/master/docs/specification.md> (last accessed on October 3, 2024)

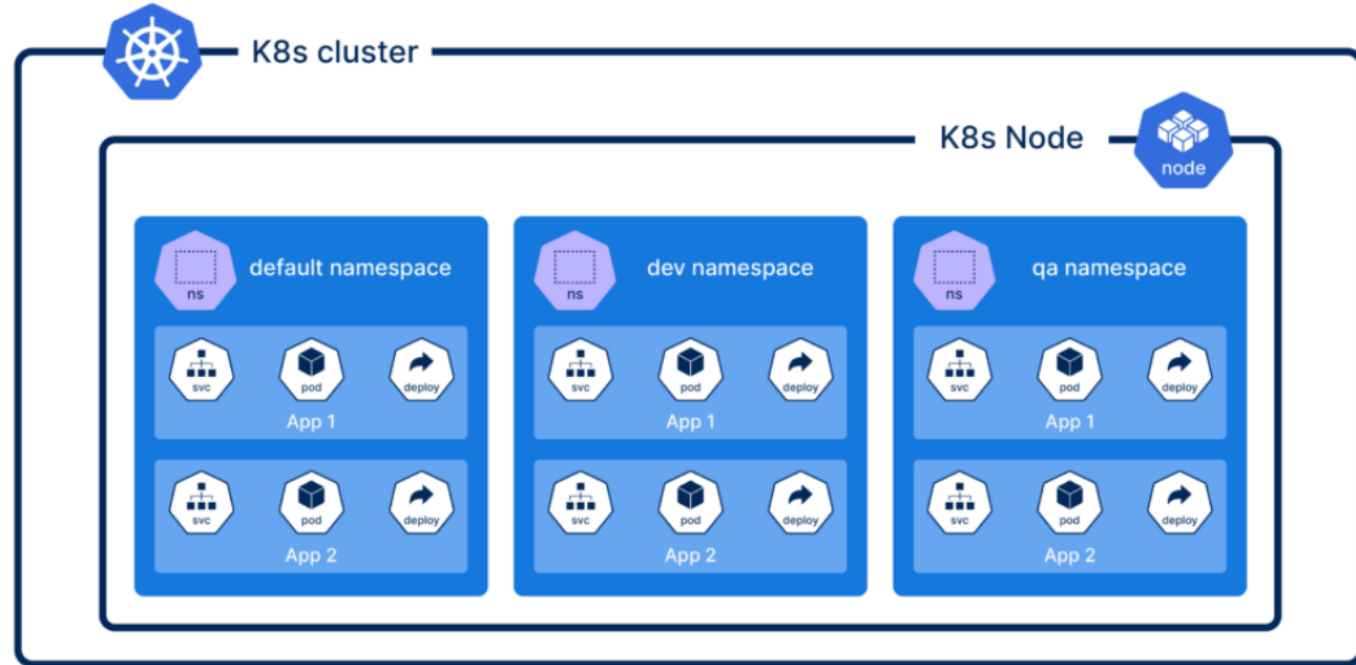
Namespaces can be used to isolate groups of resources within a Kubernetes cluster and the resources within a namespace have unique names.

Namespaces

In Kubernetes, *namespaces* provide a mechanism for isolating groups of resources within a single cluster. Names of resources need to be unique within a namespace, but not across namespaces. Namespace-based scoping is applicable only for namespaced objects (*e.g. Deployments, Services, etc.*) and not for cluster-wide objects (*e.g. StorageClass, Nodes, PersistentVolumes, etc.*).

Source: <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/> (last accessed October 3, 2024)

Kubernetes - Namespaces



Source: <https://www.appvia.io/blog/best-practices-for-kubernetes-namespaces> (last accessed on October 3, 2024)

Services in Kubernetes are used for exposing pods over the network. Like pods, a Service is a Kubernetes object and defines a logical set of pods and policies for accessing said pods. The pods associated with a Service are also referred to as endpoints.

Services in Kubernetes

The Service API, part of Kubernetes, is an abstraction to help you expose groups of Pods over a network. Each Service object defines a logical set of endpoints (usually these endpoints are Pods) along with a policy about how to make those pods accessible.

For example, consider a stateless image-processing backend which is running with 3 replicas. Those replicas are fungible—frontends do not care which backend they use. While the actual Pods that compose the backend set may change, the frontend clients should not need to be aware of that, nor should they need to keep track of the set of backends themselves.

The Service abstraction enables this decoupling.

Source: <https://kubernetes.io/docs/concepts/services-networking/service/> (last accessed on October 10, 2024)

The image below shows details of the Service named “kube-dns.” The Service details include its own ClusterIP and the endpoint IP addresses (pod IP addresses running on nodes).

```
surender@kube-srv1: ~$ kubectl describe svc kube-dns -n kube-system
Name: kube-dns
Namespace: kube-system
Labels: k8s-app=kube-dns
        kubernetes.io/cluster-service=true
        kubernetes.io/name=CoreDNS
Annotations: prometheus.io/port: 9153
              prometheus.io/scrape: true
Selector: k8s-app=kube-dns
Type: ClusterIP
IP Family Policy: SingleStack
IP Families: IPv4
IP: 10.96.0.10 ← IP address of CoreDNS service
IPs: 10.96.0.10
Port: dns 53/UDP
TargetPort: 53/UDP
Endpoints: 10.244.0.62:53,10.244.0.63:53 ← Viewing service endpoints
Port: dns-tcp 53/TCP
TargetPort: 53/TCP
Endpoints: 10.244.0.62:53,10.244.0.63:53
Port: metrics 9153/TCP
TargetPort: 9153/TCP
Endpoints: 10.244.0.62:9153,10.244.0.63:9153
Session Affinity: None
Events: <none>
surender@kube-srv1:~$
surender@kube-srv1:~$ kubectl get pods -n kube-system -l k8s-app=kube-dns -o wide
NAME                                READY  STATUS   RESTARTS  AGE  IP            NODE
coredns-5d78c9869d-7hnb5            1/1    Running  23        130d  10.244.0.62   kube-srv1.test1
ab.local                            <none>  <none>    <none>    <none>
coredns-5d78c9869d-sj75b            1/1    Running  23        130d  10.244.0.63   kube-srv1.test1
ab.local                            <none>  <none>    <none>    <none>
surender@kube-srv1:~$
surender@kube-srv1:~$
```

View kube dns service endpoints and the IP addresses of the CoreDNS Pods



	Source: https://4sysops.com/archives/kubernetes-coredns/ (last accessed on October 3, 2024)
Claim 34	Comerica Count IV System and Services
34. The virtual network manager of claim 30 wherein the register module is further configured to receive the registration request from the agent that is installed on the device for data communication via the virtual network.	Upon information and belief, Comerica’s systems include the “virtual network manager of claim 30 wherein the register module is further configured to receive the registration request from the agent that is installed on the device for data communication via the virtual network.” For example, kube-apiserver which includes the Kubernetes Service API (e.g. register module) receives the registration request from kubelets (e.g. agent that is installed on the device) which operate in each node containing a set of pods (e.g. device) to communicate back to the kube-apiserver. See Claim 30, <i>supra</i>. As an example, on information and belief, the below further describes this limitation. The kubelet is an agent on each node that registers a newly created node with the API Server. By default, the self-registration of the node is attempted by the kubelet. kubelet The kubelet is the primary “node agent” that runs on each node. It can register the node with the apiserver using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider. The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn’t manage containers which were not created by Kubernetes. Source: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/ (last accessed March 15, 2024).

Self-registration of Nodes

When the kubelet flag `--register-node` is true (the default), the kubelet will attempt to register itself with the API server. This is the preferred pattern, used by most distros.

Source: <https://kubernetes.io/docs/concepts/architecture/nodes/> (last accessed on October 1, 2024)

kubelet + kube-apiserver + etcd

Actually, when kubelet talks to kube-apiserver then we shouldn't call it standalone anymore, it is API Server mode now. So instead of creating Pods by creating the POD definition yaml into the kubelet-static-pod directory, we will configure kubelet talks to kube-apiserver and create Pods from there.

Source: <https://medium.com/@ElieXU/kubelet-api-server-8e2bdd6a147e> (last accessed March 15, 2024).

When Pods are scheduled to a node. Kubelet sees the volume in the Pod is a projected type volume, Kubelet handles communicating with the Kubernetes API server to request the bound token and ensures the projected volume contains the token. Kubelet also periodically refreshes the token when its expiry is approaching.

Source: <https://luandy-4171.medium.com/kubernetes-new-service-account-tokens-25adf0d9c164> (last accessed March 15, 2024).



	DNS You can (and almost always should) <u>set up a DNS service for your Kubernetes cluster using an add-on</u>. A cluster-aware DNS server, such as CoreDNS, <u>watches the Kubernetes API for new Services</u> and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then <u>all Pods</u> should automatically be able to resolve Services by their DNS name. For example, if you have a Service called <code>my-service</code> in a Kubernetes namespace <code>my-ns</code>, the control plane and the DNS Service acting together create a DNS record for <code>my-service.my-ns</code>. Pods in the <code>my-ns</code> namespace should be able to find the service by doing a name lookup for <code>my-service</code> (<code>my-service.my-ns</code> would also work). Source: https://kubernetes.io/docs/concepts/services-networking/service/ (last accessed March 15, 2024). Note that, as a further example, the relation between the Kube and API server in accordance with the limitation. The kubelet, for example, transmits the registration request to the Kubernetes API server. kube-apiserver The API server is a component of the Kubernetes <u>control plane</u> that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane. Source: https://kubernetes.io/docs/concepts/overview/components/ (last accessed March 15, 2024).
--	---

35. The virtual network manager of claim 30 further comprising a join module configured to receive a join request from the agent associated with the device to indicate that the device is connected for data communication within the virtual network, the join module further configured to receive a leave request from the agent associated with the device to indicate that the device will be disconnected from data communication within the virtual network.

Upon information and belief, Comerica's systems include the "virtual network manager of claim 30 further comprising a join module configured to receive a join request from the agent associated with the device to indicate that the device is connected for data communication within the virtual network, the join module further configured to receive a leave request from the agent associated with the device to indicate that the device will be disconnected from data communication within the virtual network." For example, the kubelet (e.g. agent associated with the device) located within a node of a set of pods (e.g. device connected for data communication) is responsible for deleting pod functions or processes, thus removing them (e.g. disconnected from data communication) from the kube-apiserver. The kubelet on the unresponsive node starts responding by killing the pod and removing the entry from the apiserver.

See Claim 30, *supra*.

kubelet

The kubelet is the primary "node agent" that runs on each node. It can register the node with the apiserver using one of: the hostname; a flag to override the hostname; or specific logic for a cloud provider.

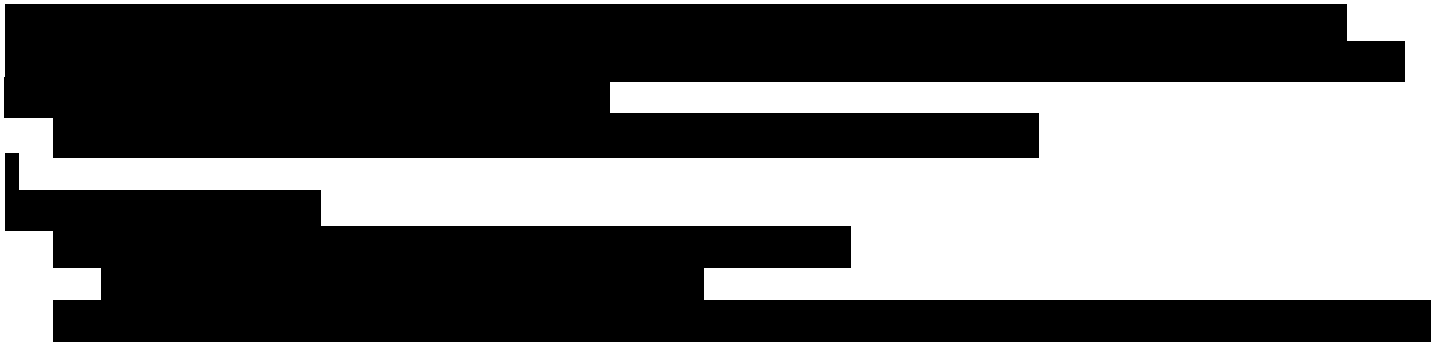
The kubelet works in terms of a PodSpec. A PodSpec is a YAML or JSON object that describes a pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms (primarily through the apiserver) and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

Source: <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/> (last accessed March 15, 2024).



	Delete Pods A Pod is not deleted automatically when a node is unreachable. The Pods running on an unreachable Node enter the 'Terminating' or 'Unknown' state after a <u>timeout</u>. Pods may also enter these states when the user attempts graceful deletion of a Pod on an unreachable Node. The only ways in which a Pod in such a state can be removed from the apiserver are as follows: • The Node object is deleted (either by you, or by the Node Controller). • The kubelet on the unresponsive Node starts responding, kills the Pod and removes the entry from the apiserver. • Force deletion of the Pod by the user. The recommended best practice is to use the first or second approach. If a Node is confirmed to be dead (e.g. permanently disconnected from the network, powered down, etc), then delete the Node object. If the Node is suffering from a network partition, then try to resolve this or wait for it to resolve. When the partition heals, <u>the kubelet will complete the deletion of the Pod</u> and free up its name in the apiserver. Source: https://kubernetes.io/docs/tasks/run-application/force-delete-stateful-set-pod/ (last accessed March 15, 2024).
--	--



	DNS You can (and almost always should) <u>set up a DNS service for your Kubernetes cluster using an add-on.</u> <u>A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one.</u> If DNS has been enabled throughout your cluster then all <u>Pods</u> should automatically be able to resolve Services by their DNS name. Source: https://kubernetes.io/docs/concepts/services-networking/service/ (last accessed March 15, 2024). Source Code Evidence 
Claim 37	Comerica Count IV System and Services
37. The virtual network manager of claim 35 wherein the join module is further configured to maintain data to associate a virtual network address with a device in the virtual network.	Upon information and belief, Comerica’s systems include the “virtual network manager of claim 35 wherein the join module is further configured to maintain data to associate a virtual network address with a device in the virtual network.” <i>See Claim 35, supra.</i> As an example, the Kubernetes DNS service module includes defining a service, and within that service (as explained below) the join module is configured “to maintain data to associate...” by, for example, allowing creating, viewing, or modifying service definitions and watching Kubernetes APIs.

Defining a Service

A Service is an object (the same way that a Pod or a ConfigMap is an object). You can create, view or modify Service definitions using the Kubernetes API. Usually you use a tool such as `kubectl` to make those API calls for you.

DNS

You can (and almost always should) set up a DNS service for your Kubernetes cluster using an add-on.

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then all Pods should automatically be able to resolve Services by their DNS name.

For example, if you have a Service called `my-service` in a Kubernetes namespace `my-ns`, the control plane and the DNS Service acting together create a DNS record for `my-service.my-ns`. Pods in the `my-ns` namespace should be able to find the service by doing a name lookup for `my-service` (`my-service.my-ns` would also work).

Source: <https://kubernetes.io/docs/concepts/services-networking/service/> (last accessed March 15, 2024).

Service ClusterIP allocation

In Kubernetes, Services are an abstract way to expose an application running on a set of Pods. Services can have a cluster-scoped virtual IP address (using a Service of type: `ClusterIP`). Clients can connect using that virtual IP address, and Kubernetes then load-balances traffic to that Service across the different backing Pods.

Source: <https://kubernetes.io/docs/concepts/services-networking/cluster-ip-allocation/> (last accessed March 15, 2024).