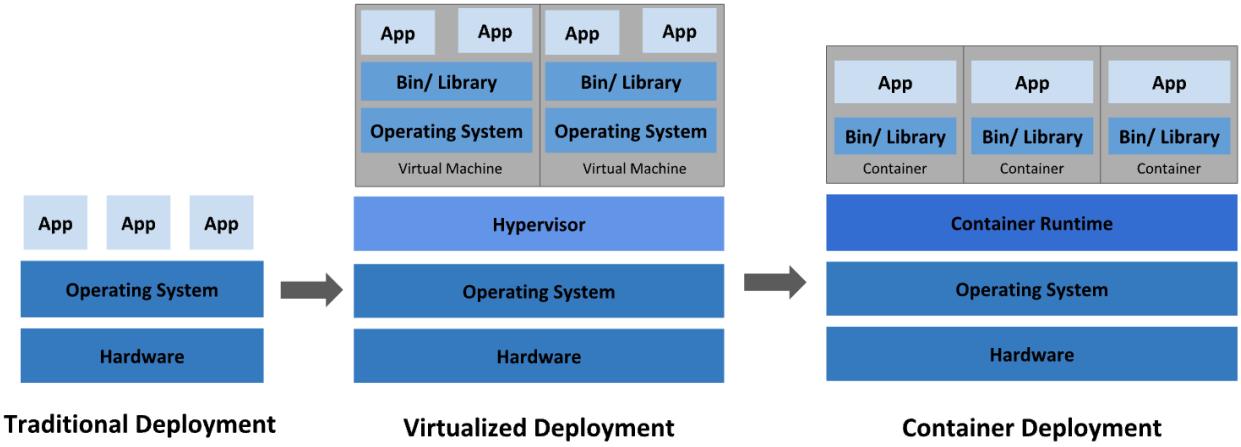
Containers



Containers are similar to VMs, but they have relaxed isolation properties to share the Operating System (OS) among the applications. Therefore, containers are considered lightweight. Similar to a VM, a container has its own filesystem, share of CPU, memory, process space, and more. As they are decoupled from the underlying infrastructure, they are portable across clouds and OS distributions.

Containers benefits:

- Agile application creation and deployment: increased ease and efficiency of container image creation compared to VM image use.
- Continuous development, integration, and deployment: provides for reliable and frequent container image build and deployment with quick and efficient rollbacks (due to **image immutability**).
- Dev and Ops separation of concerns: create application container images at build/release time rather than deployment time, thereby decoupling applications from infrastructure.
- Observability: not only surfaces OS-level information and metrics, but also application health and other signals.
- **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.
- 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.
- Resource isolation: predictable application performance.
- Resource utilization: high efficiency and density.

Docker

Docker-compose

Kubernetes

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 provides you with:

- **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**: You can describe the desired state for your deployed containers using Kubernetes, and it can change the actual state to the desired state at a controlled rate. For example, you can automate Kubernetes to create new containers for your deployment, remove existing containers and adopt all their resources to the new container.
- **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 is not

Kubernetes is not a traditional, all-inclusive PaaS (Platform as a Service) system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

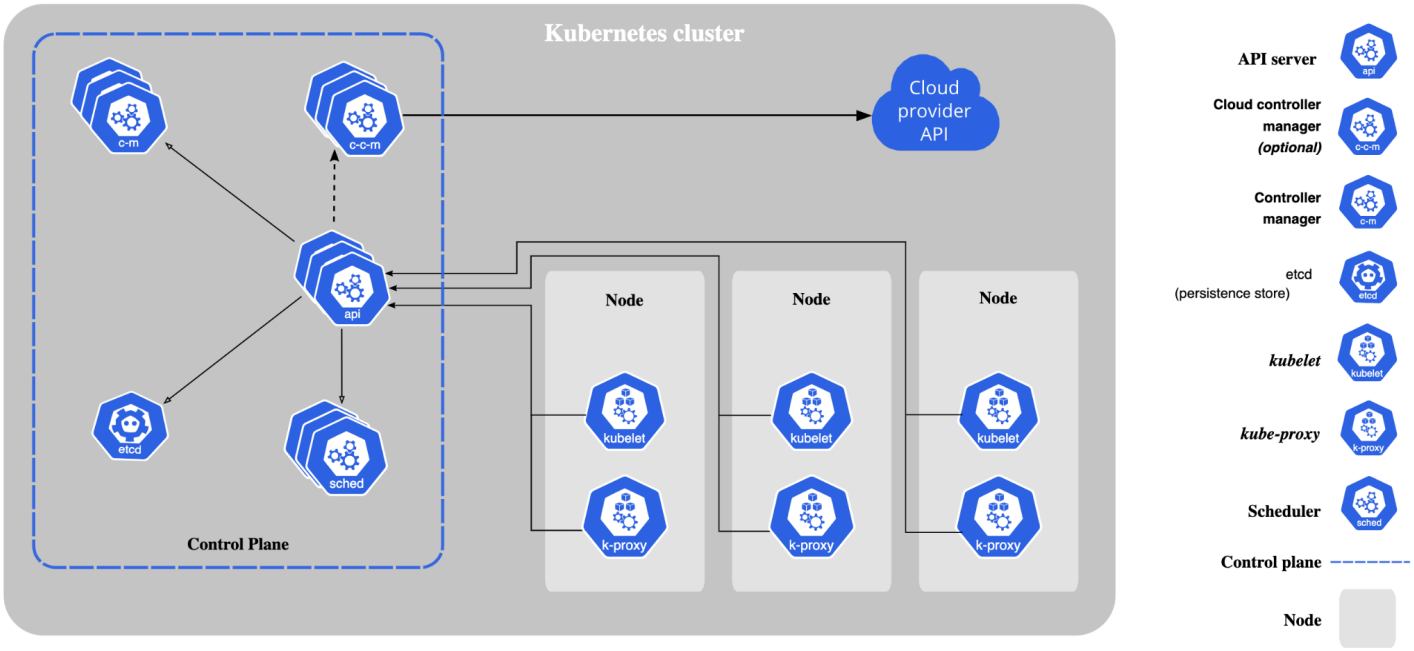
- Does not limit the types of applications supported.
- Does not deploy source code and does not build your application
- Does not provide application-level services, such as middleware (for example, message buses), data-processing frameworks (for example, Spark), databases (for example, MySQL), caches, nor cluster storage systems (for example, Ceph) as built-in services.
- Does not dictate logging, monitoring, or alerting solutions.
- Does not provide nor mandate a configuration language/system (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
- Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.
- Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

Kubernetes Components

When you deploy Kubernetes, you get a **cluster**.

A Kubernetes cluster consists of a set of **worker machines, called nodes**, that run containerized applications. Every cluster has at least one worker node.

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.



Control plane components

The **control plane's** components make **global decisions about the cluster** (for example, scheduling), as well as **detecting and responding to cluster events** (for example, starting up a new pod when a deployment's replicas field is unsatisfied).

Control plane components can be run on any machine in the cluster. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine. See Creating Highly Available clusters with kubeadm for an example control plane setup that runs across multiple VMs.

kube-apiserver

The API server is a component of the Kubernetes control plane that exposes the Kubernetes API. The API server is the **front end for the Kubernetes control plane**.

The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to **scale horizontally** - that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

etcd

Consistent and highly-available **key value store** used as Kubernetes' **backing store for all cluster data**.

If your Kubernetes cluster uses etcd as its backing store, make sure you have a back up plan for that data.

You can find in-depth information about etcd in the official documentation.

kube-scheduler

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

kube-controller-manager

Control plane component that **runs controller processes**.

Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some types of these controllers are:

- Node controller: Responsible for noticing and responding when nodes go down.
- Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.

- Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
- Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.

cloud-controller-manager

A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

Node components

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

kubelet

An **agent** that runs on each node in the cluster. It makes sure that containers are running in a Pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms 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.

kube-proxy

kube-proxy is a **network proxy** that runs on each node in your cluster, implementing part of the Kubernetes **Service concept**.

kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster.

kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

Container runtime

The container runtime is the software that is responsible for running containers.

Kubernetes supports several container runtimes: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

Addons

Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features. Because these are providing cluster-level features, namespaced resources for addons belong within the kube-system namespace.

DNS

While the other addons are not strictly required, **all Kubernetes clusters should have cluster DNS**, as many examples rely on it.

Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which **serves DNS records for Kubernetes services**.

Containers started by Kubernetes automatically include this DNS server in their DNS searches.

Web UI (Dashboard)

Dashboard is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

Container Resource Monitoring

Container Resource Monitoring records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

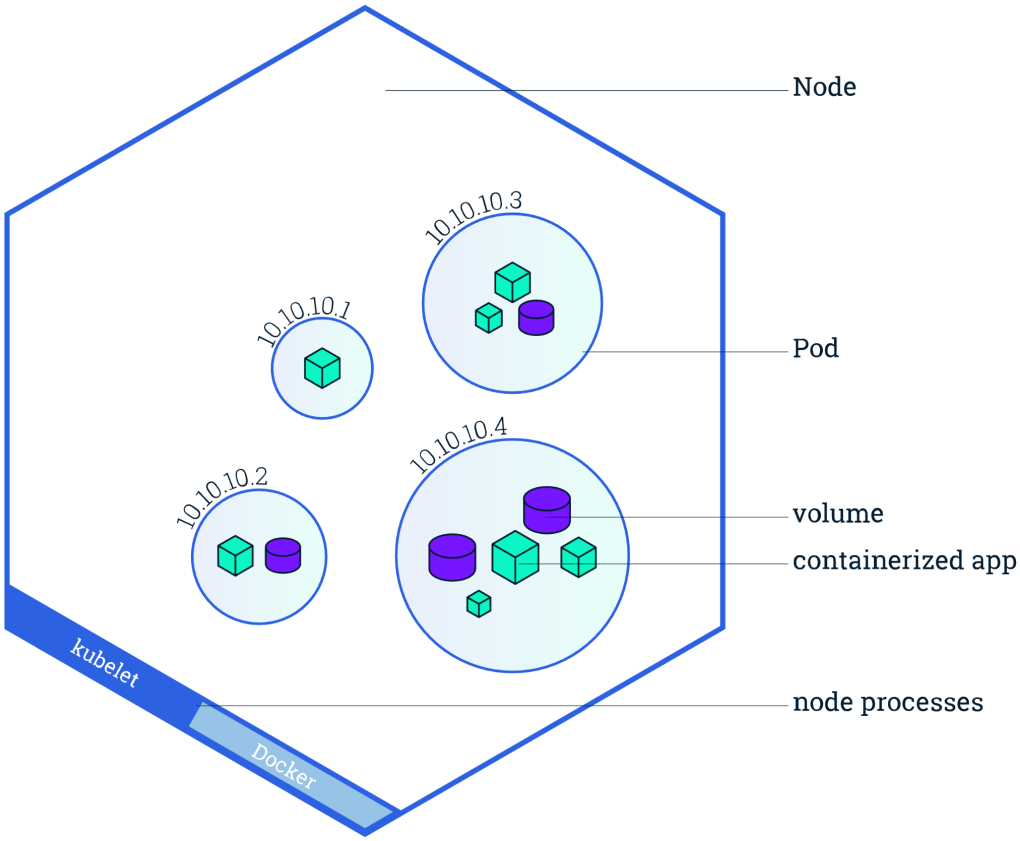
Cluster-level Logging

A cluster-level logging mechanism is responsible for saving container logs to a central log store with search/browsing interface.

Cluster architecture

Nodes

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.



Typically you have several nodes in a cluster; in a learning or resource-limited environment, you might have only one node.

The components on a node include the kubelet, a container runtime, and the kube-proxy.

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

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

```
{
  "kind": "Node",
  "apiVersion": "v1",
  "metadata": {
```

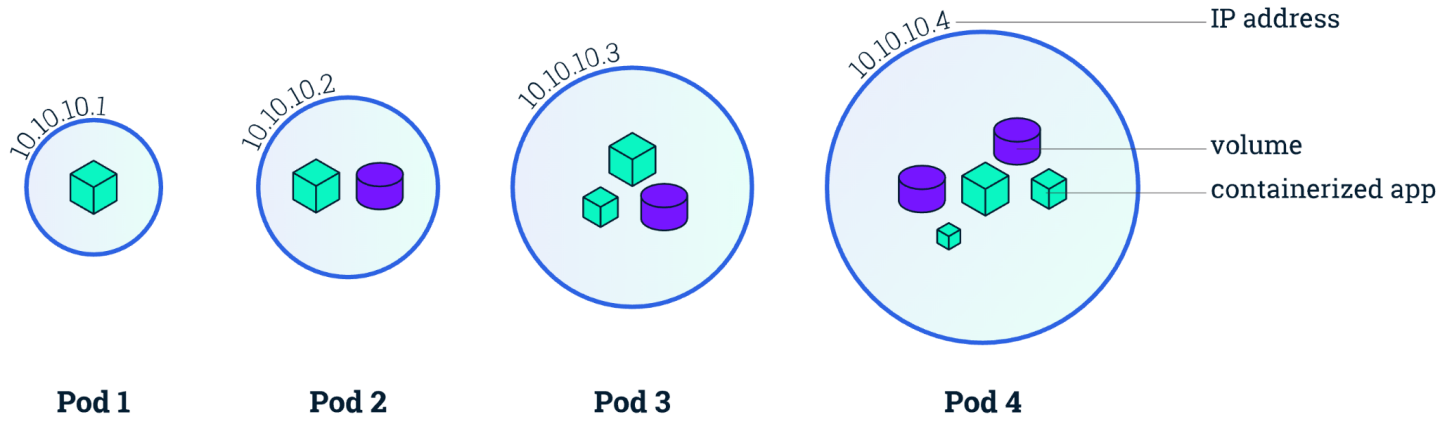
```
    "name": "10.240.79.157",
    "labels": {
      "name": "my-first-k8s-node"
    }
  }
}
```

Kubernetes creates a Node object internally (the representation). Kubernetes checks that a kubelet has registered to the API server that matches the metadata.name field of the Node. If the node is healthy (i.e. all necessary services are running), then it is eligible to run a Pod. Otherwise, that node is ignored for any cluster activity until it becomes healthy.

The name identifies a Node. Two Nodes cannot have the same name at the same time. Kubernetes also assumes that a resource with the same name is the same object. In case of a Node, it is implicitly assumed that an instance using the same name will have the same state (e.g. network settings, root disk contents). This may lead to inconsistencies if an instance was modified without changing its name. If the Node needs to be replaced or updated significantly, the existing Node object needs to be removed from API server first and re-added after the update.

Pods

A Pod is a **group of one or more application containers** (such as Docker or rkt) that includes **shared storage** (volumes), a **unique cluster IP address** and information about how to run them (like container image version or specific ports). **Containers within a Pod share an IP Address and port space**. Containers of the same Pod are always co-located and co-scheduled, and run in a shared context on the same node. A Pod models an application-specific "**logical host**" and contains one or more application containers which are relatively tightly coupled. An example of container that would fit on the same Pod with our the NodeJS app, would be a side container that feeds the data published by the webserver. In a pre-container world, they would have run on the same physical or virtual machine.



Pods are tied to the Node where they are deployed and remain there until termination (according to restart policy) or deletion. In case of a Node failure, new identical Pods will be deployed on other available Nodes. **The Pod is the atomic deployment unit on the Kubernetes platform**. When we trigger a Deployment on Kubernetes, it will create Pods with containers inside them, not containers directly.

Benefits of a pod: when pods contain multiple containers, communications, and data sharing between them is simplified. Since all containers in a pod **share the same network namespace**, they can locate each other and communicate via localhost. Pods can communicate with each other by using another pods IP address or by referencing a resource that resides in another pod.

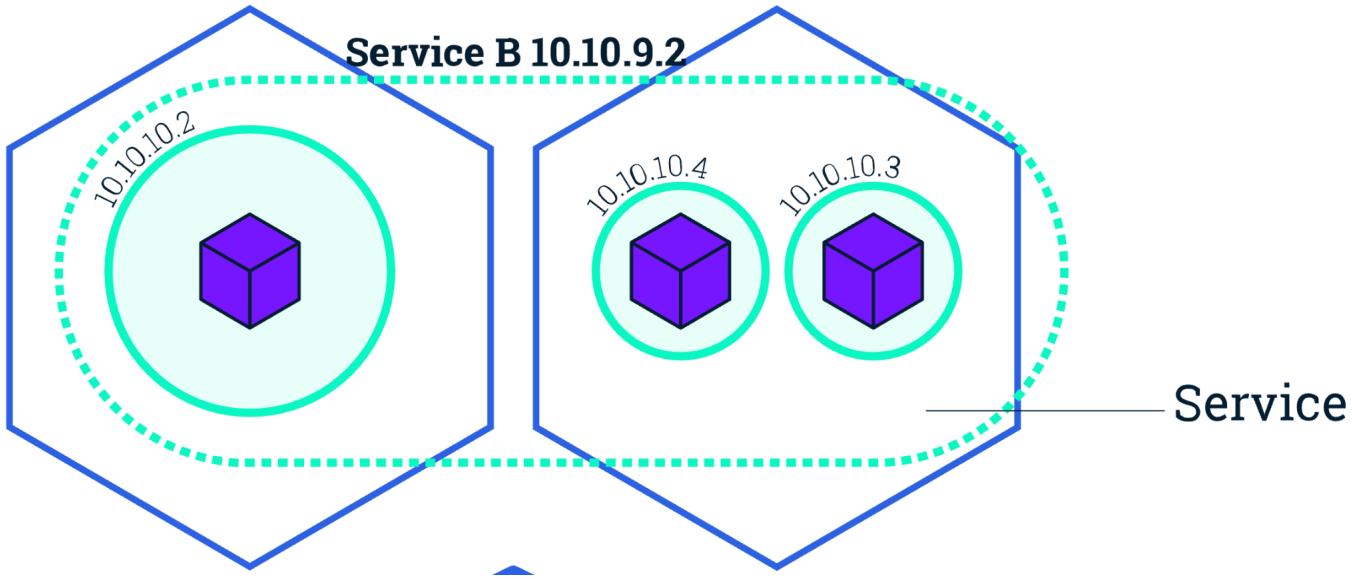
Pods can include containers that run when the pod is started, say to perform initiation required before the application containers run. Additionally, pods simplify scalability, enabling replica pods to be created and shut down automatically based on changes in demand.

Services

An abstract way to expose an application running on a set of Pods as a network service.

While Pods do have their own unique IP across the cluster, those IP's **are not exposed outside Kubernetes**. Taking into account that over time Pods may be terminated, deleted or replaced by other Pods, we need a way to let other Pods and applications automatically **discover each other**. Kubernetes addresses this by grouping Pods in Services. A Kubernetes Service is an abstraction layer which defines a **logical set of Pods** and enables **external traffic exposure, load balancing and service discovery** for those Pods.

With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.



This abstraction will allow us to expose Pods to traffic originating from outside the cluster. Services have their own unique cluster-private IP address and expose a port to receive traffic. If you choose to expose the service outside the cluster, the options are:

- **LoadBalancer** - provides a public IP address (what you would typically use when you run Kubernetes on GKE or AWS)
- **NodePort** - exposes the Service on the same port on each Node of the cluster using NAT (available on all Kubernetes clusters, and in Minikube)

A Service in Kubernetes is a REST object, similar to a Pod:

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
```

```
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

This specification creates a new Service object named "my-service", which targets TCP port 9376 on any Pod with the app=MyApp label.