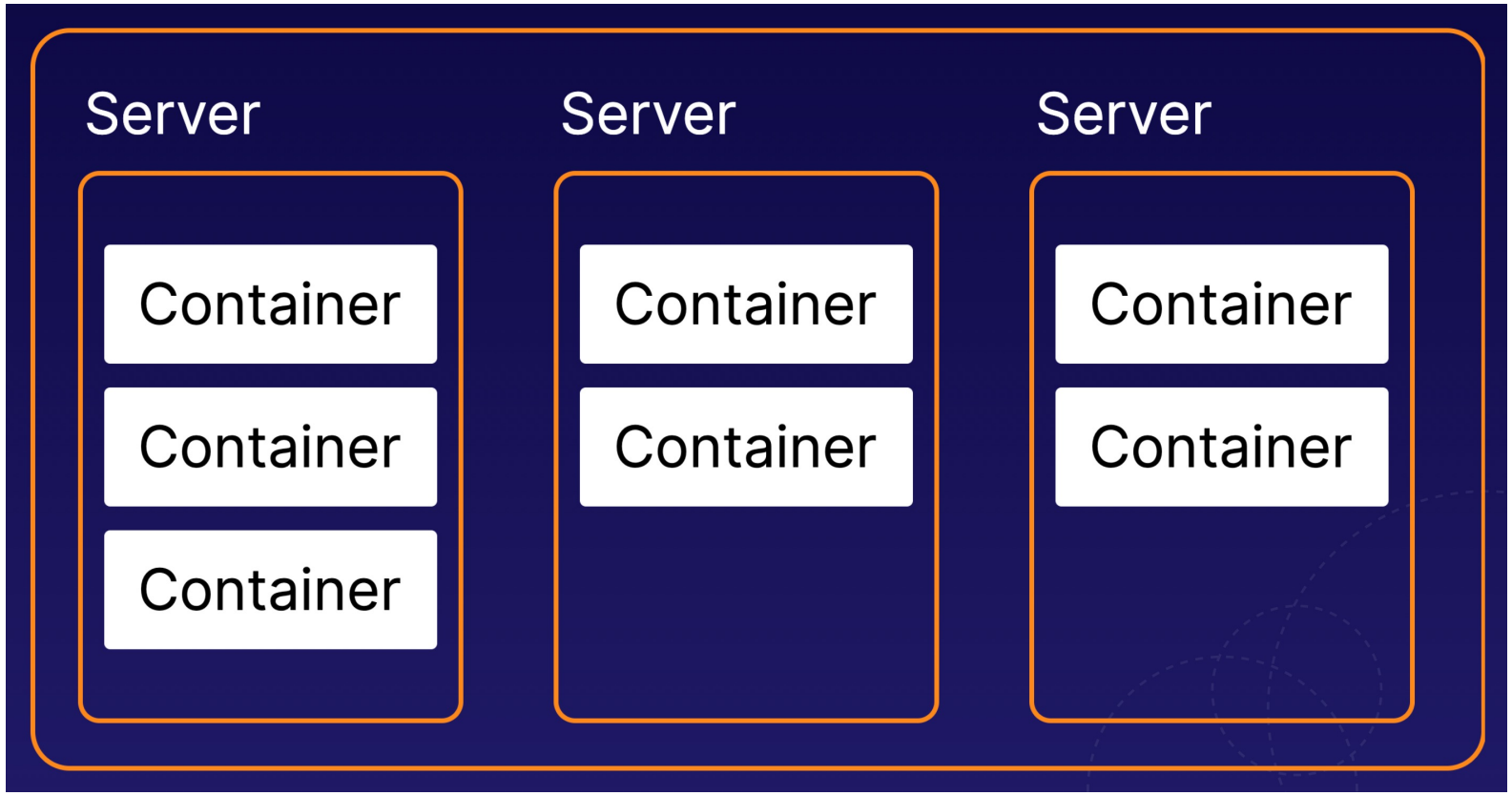

kubernetes

What Does "K8s" Mean?

K8s is simply short for Kubernetes. The 8 represents the 8 letters between K and S!

Introduction


Server
Server
Server
Container
Container
Container
Container
Container
Container
Container
Container

K8s Cluster

K8s Features

Container Orchestration

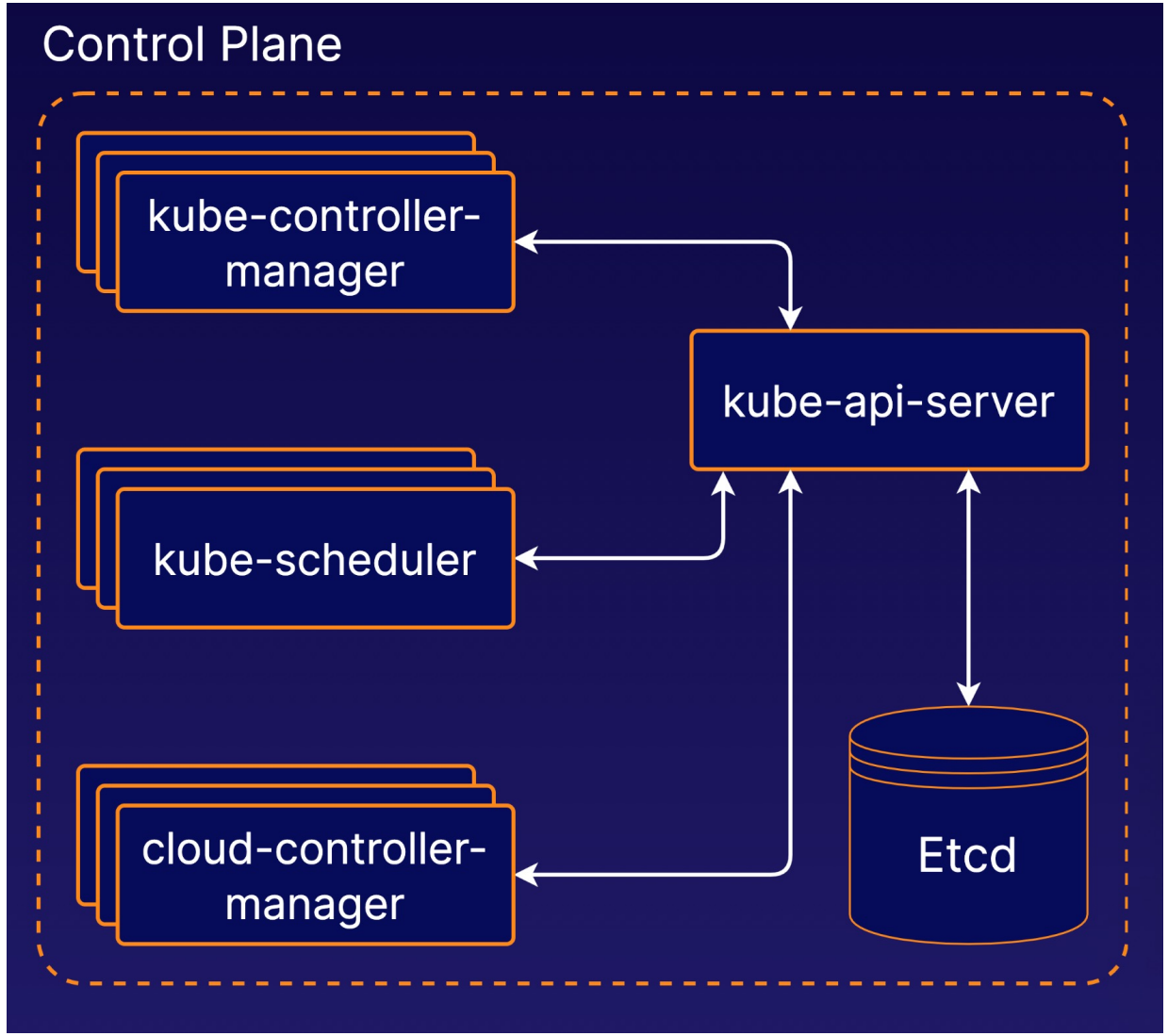
The primary purpose of Kubernetes is to dynamically manage containers across multiple host systems.

Application Reliability

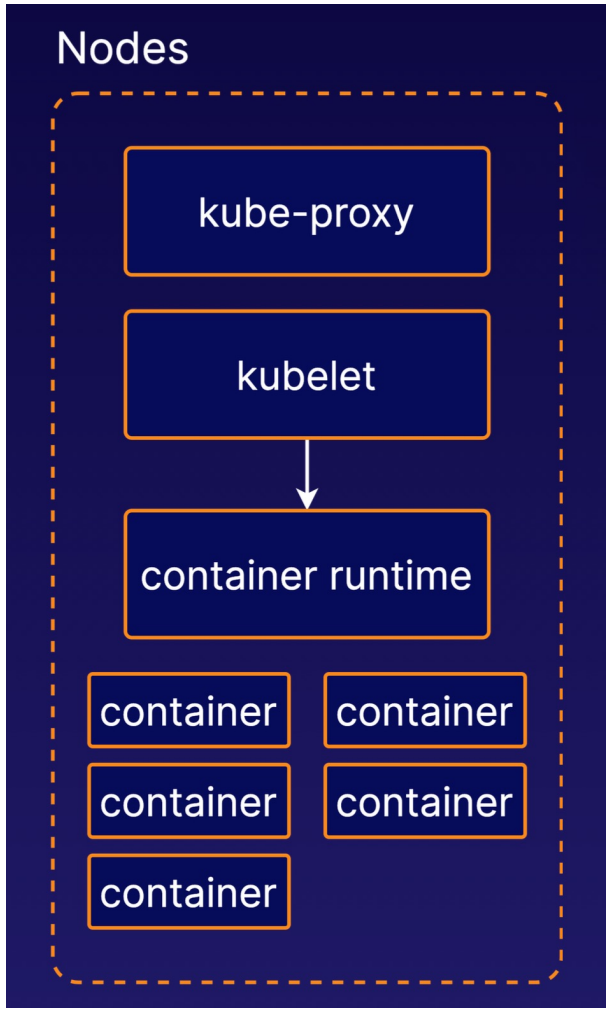
Kubernetes makes it easier to build reliable, self-healing, and scalable applications.

Automation

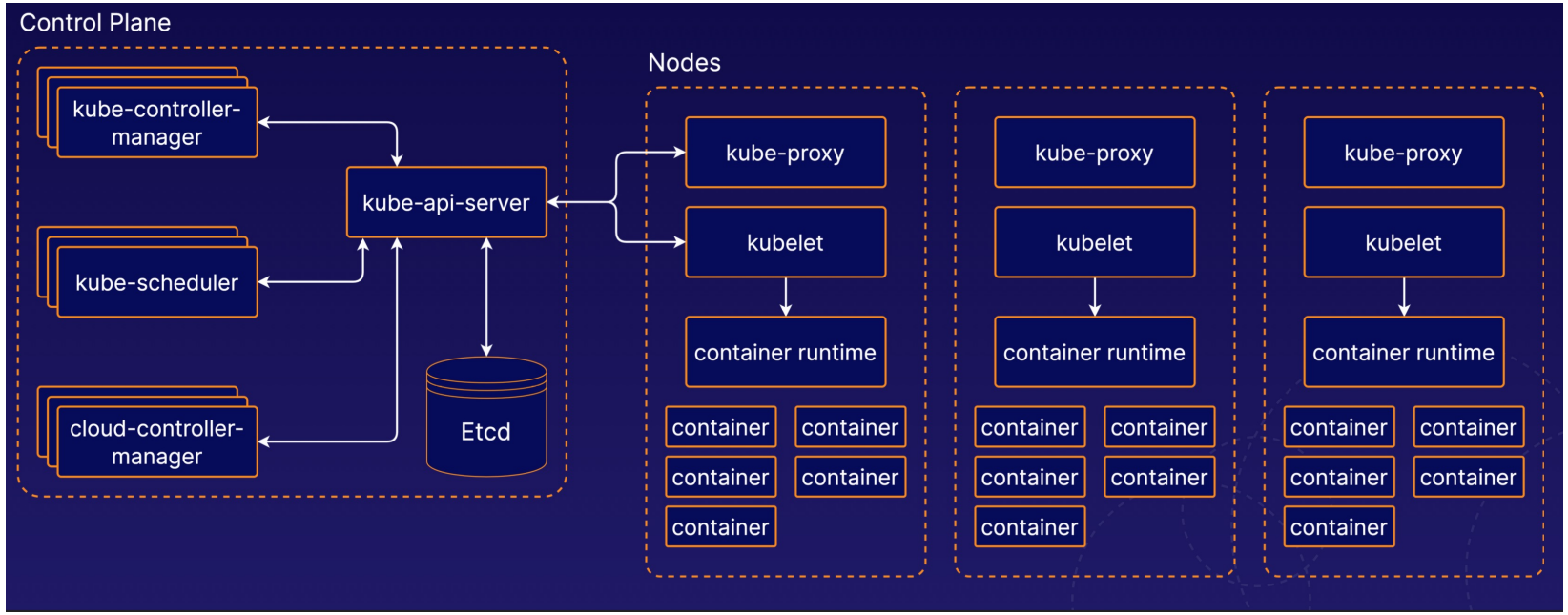
Kubernetes offers a variety of features to help automate the management of your container apps.



K8s Architecture



K8s Architecture



K8s Architecture

Workshop 1

```
❯ k3d cluster create mycluster \
--servers 1 \
--agents 3 \
--image rancher/k3s:latest
INFO[0000] Prep: Network
INFO[0000] Created network 'k3d-mycluster'
INFO[0000] Created image volume k3d-mycluster-images
INFO[0000] Starting new tools node...
INFO[0001] Creating node 'k3d-mycluster-server-0'
INFO[0001] Pulling image 'ghcr.io/k3d-io/k3d-tools:5.6.0'
INFO[0004] Starting Node 'k3d-mycluster-tools'
INFO[0004] Pulling image 'rancher/k3s:latest'
INFO[0011] Creating node 'k3d-mycluster-agent-0'
INFO[0011] Creating node 'k3d-mycluster-agent-1'
INFO[0011] Creating node 'k3d-mycluster-agent-2'
INFO[0011] Creating LoadBalancer 'k3d-mycluster-serverlb'
INFO[0013] Pulling image 'ghcr.io/k3d-io/k3d-proxy:5.6.0'
INFO[0018] Using the k3d-tools node to gather environment information
INFO[0018] Starting new tools node...
INFO[0018] Starting Node 'k3d-mycluster-tools'
INFO[0019] Starting cluster 'mycluster'
INFO[0019] Starting servers...
INFO[0019] Starting Node 'k3d-mycluster-server-0'
INFO[0023] Starting agents...
INFO[0023] Starting Node 'k3d-mycluster-agent-0'
INFO[0023] Starting Node 'k3d-mycluster-agent-1'
INFO[0023] Starting Node 'k3d-mycluster-agent-2'
INFO[0027] Starting helpers...
INFO[0027] Starting Node 'k3d-mycluster-serverlb'
INFO[0033] Injecting records for hostAliases (incl. host.k3d.internal) and for 6 network members into CoreDNS configmap...
INFO[0035] Cluster 'mycluster' created successfully!
INFO[0035] You can now use it like this:
kubectl cluster-info
❯ kubectl get no
NAME                     STATUS   ROLES                  AGE     VERSION
k3d-mycluster-server-0   Ready    control-plane,master   9m54s   v1.25.16+k3s4
k3d-mycluster-agent-0    Ready    <none>                 9m52s   v1.25.16+k3s4
k3d-mycluster-agent-2    Ready    <none>                 9m50s   v1.25.16+k3s4
k3d-mycluster-agent-1    Ready    <none>                 9m50s   v1.25.16+k3s4
```

YAML 101

YAML

```yaml
name: Courses
list:
- name: Go for Beginner
  price: 600
- name: Redis Fundamental
  price: 300
- name: RxJS for Beginner
  price: 500
```

JSON

```json
{
  "name": "Courses",
  "list": [
    {
      "name": "Go for Beginner",
      "price": 600
    },
    {
      "name": "Redis Fundamental",
      "price": 300
    },
    {
      "name": "RxJS for Beginner",
      "price": 500
    }
  ]
}
```

Reference: https://yaml.org

YAML 101

- Simple variables (name = “Kongsak Limpitikeat”)

```
# Simple variables (name = ”Kongsak Limpitikeat")
name: Kongsak Limpitikeat
video_ideas: hundreds
free_time: 0
```

- List

```
# List (books = ["Cryptonomicon", "Snow Crash" ... ])
books:
  - Cryptonomicon
  - Snow Crash
  - The Design and Implementation of the FreeBSD Operating System
  - The Practice of Cloud System Administration
```

- Dictionary

```
# Dictionary ( languages = {"python": "excellent", "ruby": "good",
... })
languages:
  python: excellent
  ruby: good
  clojure: bad
  assembly: wannabe
```

YAML 101

- Multi-line string: with newlines

```
# Multi-line string: with newlines
hobbies: |
  4 GCSEs
  3 A-Levels
  BSc in the Internet of Things
```

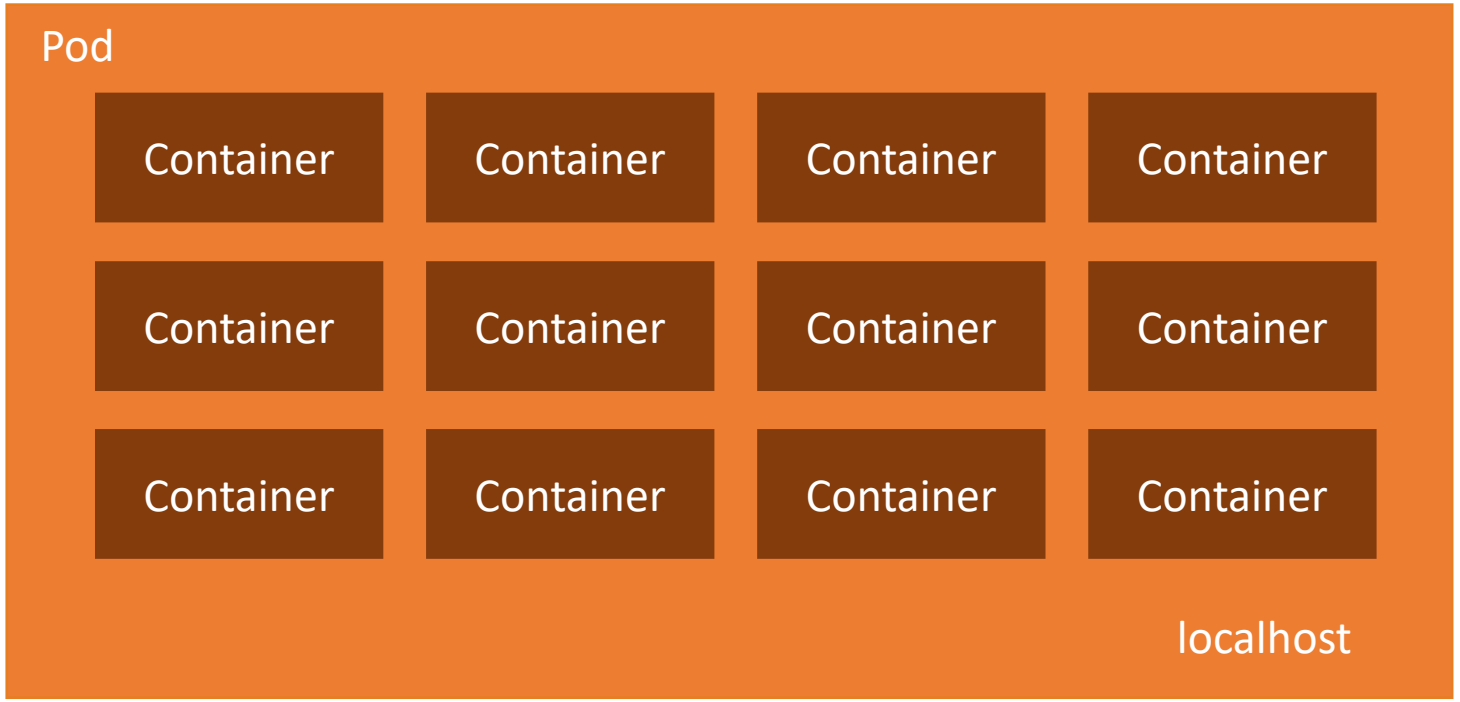
Output: "4 GECSEs\n3 A-Levels\nBsc in the Internet of Things\n"

- Multi-line string: one line

```
# Multi-line string: with newlines
hobbies: >
  4 GCSEs
  3 A-Levels
  BSc in the Internet of Things
```

Output: "4 GECSEs 3 A-Levels Bsc in the Internet of Things\n"

- Pods (po) : a group of one or more containers



POD

00-namespace.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: workshop-02
  labels:
    name: workshop-02
    module: workshop-02
```

01-one-container-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: one-container-pod
  namespace: workshop-02
spec:
  containers:
    - name: curl001
      image: curlimages/curl
      command: ['sh', '-c', 'while true; do sleep 5; done']
```

02-multi-container-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: multi-container-pod
  namespace: workshop-02
spec:
  containers:
    - name: curl002
      image: curlimages/curl
      command: ['sh', '-c', 'while true; do sleep 5; done']
    - name: curl003
      image: curlimages/curl
      command: ['sh', '-c', 'while true; do sleep 5; done']
    - name: curl004
      image: curlimages/curl
      command: ['sh', '-c', 'while true; do sleep 5; done']
```

Workshop 2 Pod and Namespaces

kubectl is a command line tool that allows you to interact with Kubernetes. kubectl uses the Kubernetes API to communicate with the cluster and carry out your commands.

"You can use kubectl to deploy applications, inspect and manage cluster resources, and view logs."

What Is kubectl ?

Use `kubectl get` to list objects in the Kubernetes cluster.

- `-o` — Set output format.
- `--sort-by` — Sort output using a JSONPath expression.
- `--selector` — Filter results by label.

```
$ kubectl get <object type> <object name> -o
<output> --sort-by <JSONPath> --selector
<selector>
```

kubectl get

You can get detailed information about Kubernetes objects using **kubectl describe**.

```
$ kubectl describe <object type> <object name>
```

kubectl describe

Use `kubectl create` to create objects.

Supply a YAML file with `-f` to create an object from a YAML descriptor stored in the file.

If you attempt to create an object that already exists, an error will occur.

```
$ kubectl create -f <file name>
```

kubectl create

`kubectl apply` is similar to `kubectl create`. However, if you use `kubectl apply` on an object that already exists, it will modify the existing object, if possible.

```
$ kubectl apply -f <file name>
```

kubectl apply

Use `kubectl delete` to delete objects from the cluster.

```
$ kubectl delete <object type> <object name>
```

kubectl delete

`kubectl exec` can be used to run commands inside containers. Keep in mind that, in order for a command to succeed, the necessary software must exist within the container to run it.

For pods with multiple containers, specify the container name with `-c`.

```
$ kubectl exec <pod name> -c <container name> --
<command>
```

kubectl exec

Workshop 3
kubectl

00-echo-pod.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: workshop-03
  labels:
    name: workshop-03
    module: workshop-03
---
apiVersion: v1
kind: Pod
metadata:
  name: echo-pod
  namespace: workshop-03
  labels:
    app: echo-pod-app
spec:
  containers:
    - name: echo
      image: ealen/echo-server:latest
    - name: netshoot
      image: nicolaka/netshoot
      command: ['bash', '-c', 'while true; do sleep 5; done']
      securityContext:
        privileged: true
```

ConfigMaps

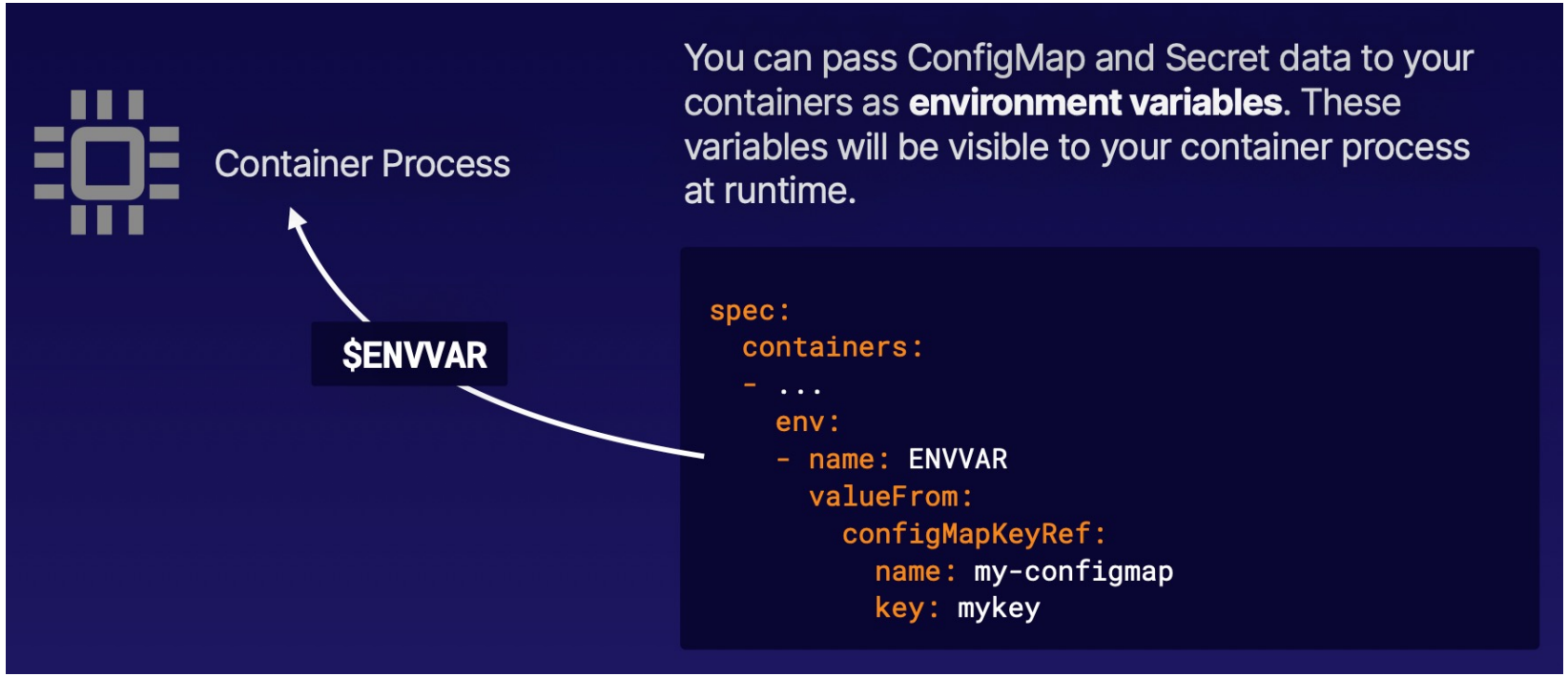
```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  key1: value1
  key2: value2
  key3:
    subkey:
      morekeys: data
      evenmore: some more data
  key4: |
    You can also do
    multi-line
    data.
```

You can store configuration data in Kubernetes using **ConfigMaps**. ConfigMaps store data in the form of a key-value map. ConfigMap data can be passed to your container applications.

Secrets are similar to ConfigMaps but are designed to store sensitive data, such as passwords or API keys, more securely. They are created and used similarly to ConfigMaps.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: user
  password: mypass
```

Secrets



Environment Variables

Configuration Volumes

Configuration data from ConfigMaps and Secrets can also be passed to containers in the form of **mounted volumes**. This will cause the configuration data to appear in files available to the container file system.

Each top-level key in the configuration data will appear as a file containing all keys below that top-level key.

```
...
volumes:
- name: secret-vol
  secret:
    secretName: my-secret
```

01-config-map.yml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
  namespace: workshop-04
data:
  key1: Hello, world!
  key2: |
    Test
    multiple lines
    more lines
  key3: >
    Test
    multiple lines
    one lines
```

00-namespace.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: workshop-04
  labels:
    name: workshop-04
    module: workshop-04
```

02-secret.yml

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
  namespace: workshop-04
type: Opaque
data:
  secretkey1: c2VjcmV0
  secretkey2: YW5vdGhlcnNlY3JldA==
```

Workshop 4
Managing Application Configuration

03-environment-variable.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: env-pod
  namespace: workshop-04
spec:
  containers:
    - name: busybox
      image: busybox
      command: [ '/bin/sh', '-c', 'while true; do sleep 5; echo "configmap: $CONFIGMAPVAR secret: $SECRETVAR"; done']
      env:
        - name: CONFIGMAPVAR
          valueFrom:
            configMapKeyRef:
              name: my-configmap
              key: key1
        - name: SECRETVAR
          valueFrom:
            secretKeyRef:
              name: my-secret
              key: secretkey1
```

Workshop 4
Managing Application Configuration

04-volume-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
  namespace: workshop-04
spec:
  containers:
    - name: busybox
      image: busybox
      command: [ '/bin/sh', '-c', 'while true; do sleep 5; done']
      volumeMounts:
        - name: configmap-volume
          mountPath: /etc/config/configmap
        - name: secret-volume
          mountPath: /etc/config/secret
  volumes:
    - name: configmap-volume
      configMap:
        name: my-configmap
    - name: secret-volume
      secret:
        secretName: my-secret
```

Workshop 4

Managing Application Configuration

05-nginx-pod.yml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: workshop-04
  name: nginx-config
data:
  nginx.conf: |
    user  nginx;
    worker_processes  1;

    error_log  /var/log/nginx/error.log warn;
    pid        /var/run/nginx.pid;

    events {
      worker_connections  1024;
    }
    http {
        server {
            listen       80;
            listen  [::]:80;
            server_name  localhost;
            location / {
                root   /usr/share/nginx/html;
                index  index.html index.htm;
            }
            auth_basic "Secure Site";
            auth_basic_user_file conf/.htpasswd;
        }
    }

---
```

05-nginx-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-04
  name: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
      volumeMounts:
        - name: config-volume
          mountPath: /etc/nginx
        - name: htpasswd-volume
          mountPath: /etc/nginx/conf
  volumes:
    - name: config-volume
      configMap:
        name: nginx-config
    - name: htpasswd-volume
      secret:
        secretName: nginx-htpasswd
---
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-04
  name: curl-pod
spec:
  containers:
    - name: curl-pod
      image: curlimages/curl
      command: [ 'sh', '-c', 'while true; do sleep 5; done' ]
```

Workshop 4
Managing Application Configuration

Kubernetes Metrics Server

In order to view metrics about the resources pods and containers are using, we need an add-on to collect and provide that data. One such add-on is **Kubernetes Metrics Server**.

Reference: https://github.com/kubernetes-sigs/metrics-server

Metrics Server

With `kubectl top`, you can view data about resource usage in your pods and nodes. `kubectl top` also supports flags like `--sort-by` and `--selector`.

```
$ kubectl top pod --sort-by <JSONPATH> --
selector <selector>
```

kubectl top

Resource requests allow you to define an amount of resources (such as CPU or memory) you expect a container to use. The Kubernetes scheduler will use resource requests to avoid scheduling pods on nodes that do not have enough available resources.

Tip: Containers are allowed to use more (or less) than the requested resources. Resource requests only affect scheduling.

Memory is measured in bytes. CPU is measured in CPU units, which are 1/1000 of one CPU.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: busybox
    image: busybox
    resources:
      requests:
        cpu: "250m"
        memory: "128Mi"
```

Resource Requests

Resource Limits

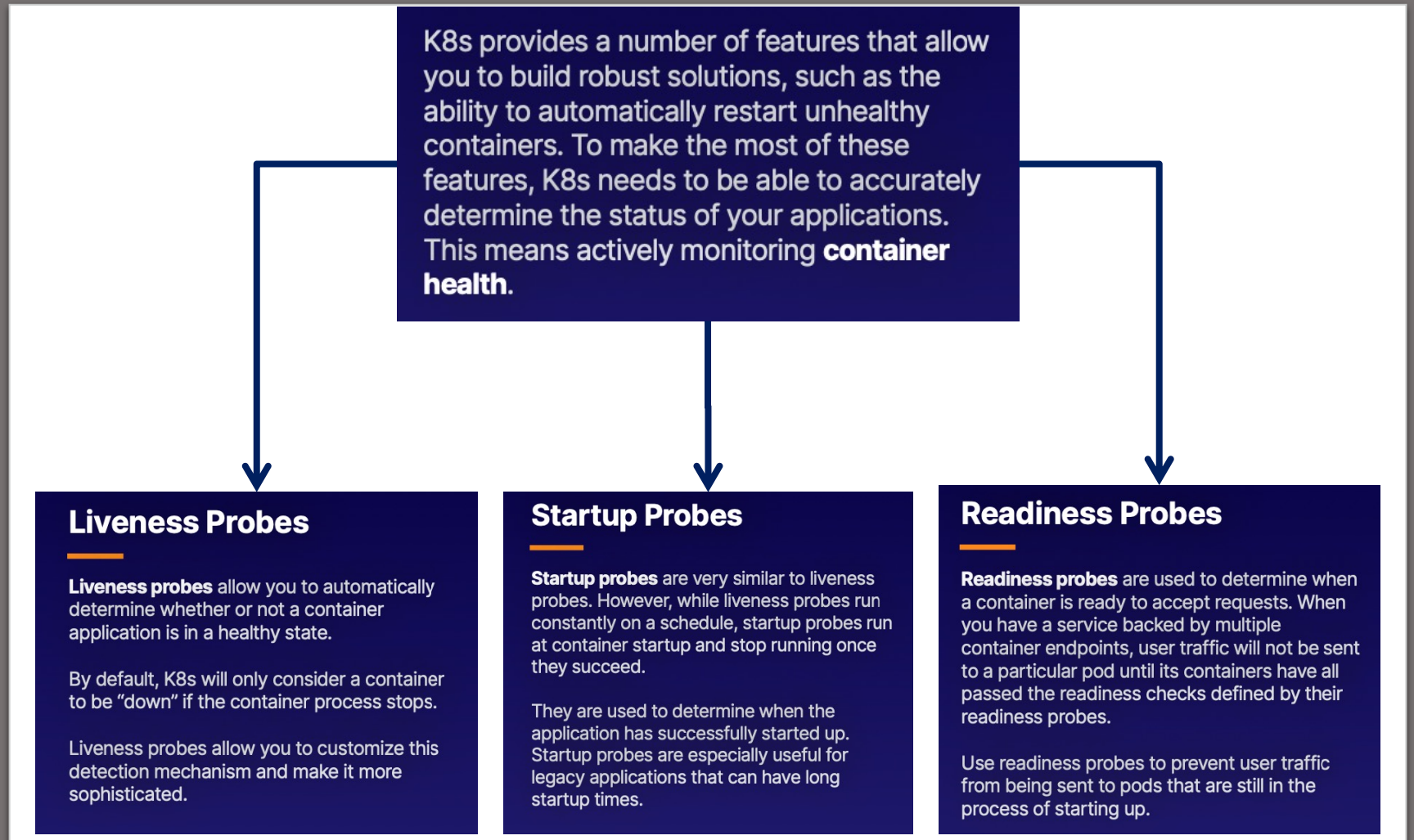
Resource limits provide a way for you to limit the amount of resources your containers can use. The container runtime is responsible for enforcing these limits, and different container runtimes do this differently.

Tip: Some runtimes will enforce these limits by terminating container processes that attempt to use more than the allowed amount of resources.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: busybox
    image: busybox
    resources:
      limits:
        cpu: "250m"
        memory: "128Mi"
```

Reference: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#resource-requests-and-limits-of-pod-and-container

Container Health



Pod Life Cycle: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/
Container Probes: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes

Workshop 5
Managing Container Resources

00-namespace.yml

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: workshop-05
  labels:
    name: workshop-05
    module: workshop-05
```

01-metrics-test.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-05
  name: metrics-test
  labels:
    app: metrics-test
spec:
  containers:
    - name: busybox
      image: busybox
      command: ['sh', '-c', 'while true; do sleep 5; done']
```

02-big-request-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-05
  name: big-request-pod
spec:
  containers:
    - name: busybox
      image: busybox
      command: ['sh','-c','while true; do sleep 5; done']
      resources:
        requests:
          #TODO: wait request cpu and memory
          cpu: '' #example '10000m'
          memory: '' #example '1Gi'
```

03-resource-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-05
  name: resource-pod
spec:
  containers:
    - name: busybox
      image: busybox
      command: ['/bin/sh','-c','while true; do sleep 5; done']
      resources:
        requests:
          cpu: '250m'
          memory: '128Mi'
        limits:
          cpu: '500m'
          memory: '256Mi'
```

Workshop 5
Managing Container Resources

04-liveness-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-05
  name: liveness-pod
spec:
  containers:
    - name: busybox
      image: busybox
      command: ['sh', '-c', 'while true; do sleep 5; done']
      livenessProbe:
        exec:
          command: ['echo', 'Hello, World!']
        initialDelaySeconds: 5
        periodSeconds: 5
```

05-liveness-pod-http.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-05
  name: liveness-pod-http
spec:
  containers:
    - name: nginx
      image: nginx:latest
      livenessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 5
        periodSeconds: 5
```

06-startup-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-05
  name: startup-pod
spec:
  containers:
    - name: nginx
      image: nginx:latest
      startupProbe:
        httpGet:
          path: /
          port: 80
        failureThreshold: 30
        periodSeconds: 10
```

07-readiness-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-05
  name: readiness-pod
spec:
  containers:
    - name: nginx
      image: nginx:latest
      readinessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 5
        periodSeconds: 5
```

Restart Policies

K8s can automatically restart containers when they fail. **Restart policies** allow you to customize this behavior by defining when you want a pod's containers to be automatically restarted.

Restart policies are an important component of self-healing applications, which are automatically repaired when a problem arises.

There are three possible values for a pod's restart policy in K8s: **Always**, **OnFailure**, and **Never**.

Always

Always is the default restart policy in K8s. With this policy, containers will always be restarted if they stop, even if they completed successfully. Use this policy for applications that should always be running.

OnFailure

The **OnFailure** restart policy will restart containers only if the container process exits with an error code or the container is determined to be unhealthy by a liveness probe. Use this policy for applications that need to run successfully and then stop.

Never

The **Never** restart policy will cause the pod's containers to never be restarted, even if the container exits or a liveness probe fails. Use this for applications that should run once and never be automatically restarted.

Restart Policy: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy

Self-Healing Pods with Restart Policies

01-always-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-06
  name: always-pod
spec:
  restartPolicy: Always
  containers:
    - name: busybox
      image: busybox
      command: [ 'sh', '-c', 'sleep 10' ]
```

02-onfailure-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-06
  name: onfailure-pod
spec:
  restartPolicy: OnFailure
  containers:
    - name: busybox
      image: busybox
      command: [ 'sh', '-c', 'sleep 10' ]
```

03-onfailure-pod-with-err.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-06
  name: onfailure-pod-with-err
spec:
  restartPolicy: OnFailure
  containers:
    - name: busybox
      image: busybox
      command: [ 'sh', '-c', 'sleep 10; this is a bad command that will fail']
```

Workshop 6
Managing Container Resources

04-never-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-06
  name: never-pod
spec:
  restartPolicy: Never
  containers:
    - name: busybox
      image: busybox
      command: [ 'sh', '-c', 'sleep 10; this is a bad command that will fail']
```

Workshop 6

Managing Container Resources

Multi-Container Pod

A Kubernetes Pod can have one or more containers. A Pod with more than one container is a **multi-container Pod**.

In a multi-container Pod, the containers share resources such as network and storage. They can interact with one another, working together to provide functionality.

Best practice: Keep containers in separate Pods unless they need to share resources.

Cross-Container Interaction

Containers sharing the same Pod can interact with one another using shared resources.

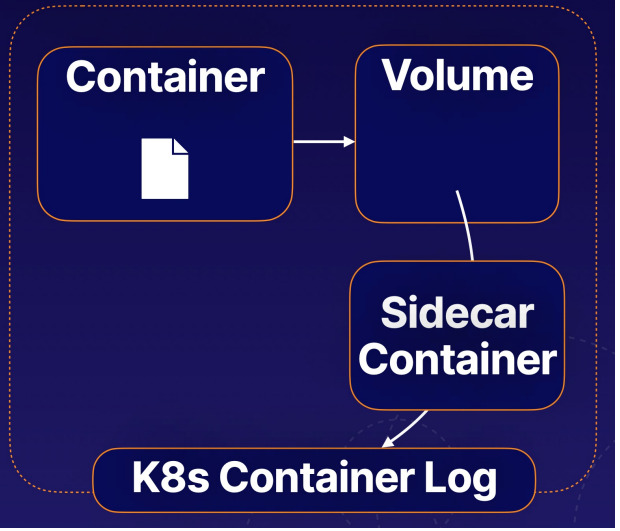
Network
Containers share the same networking namespace and can communicate with one another on any port, even if that port is not exposed to the cluster.

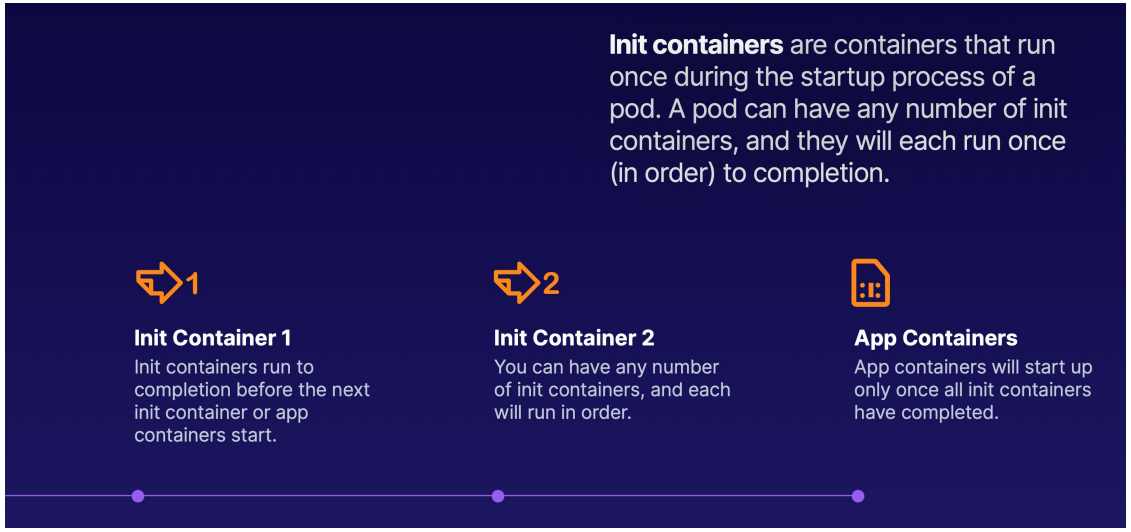
Storage
Containers can use volumes to share data in a Pod.

Example Use case

You have an application that is hard-coded to write log output to a file on disk.

You add a secondary container to the Pod (sometimes called a **sidecar**) that reads the log file from a shared volume and prints it to the console so the log output will appear in the container log.





You can use init containers to perform a variety of startup tasks. They can contain and use software and setup scripts that are not needed by your main containers.

They are often useful in keeping your main containers lighter and more secure by offloading startup tasks to a separate container.

Use Cases for Init Containers

Some sample use cases for init containers:

- Cause a pod to wait for another K8s resource to be created before finishing startup.
- Perform sensitive startup steps securely outside of app containers.
- Populate data into a shared volume at startup.
- Communicate with another service at startup.

Init containers

01-sidecar-pod.yml

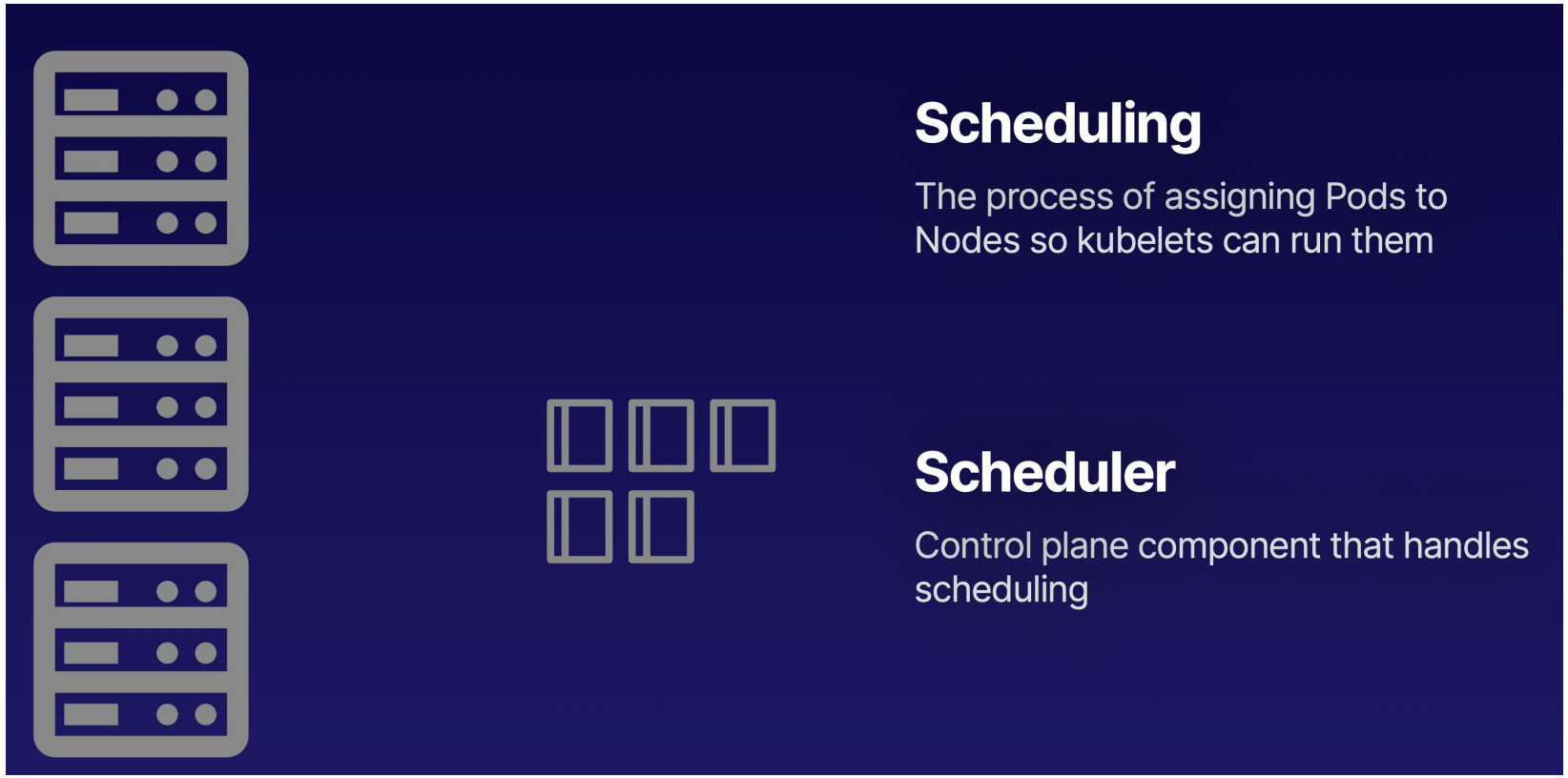
```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-07
  name: sidecar-pod
spec:
  containers:
    - name: busybox
      image: busybox
      command: ['sh', '-c', 'while true; do now=$(date);echo logs time $now >> /logs/output.log; sleep 5; done']
      volumeMounts:
        - name: sharedvol
          mountPath: /logs
    - name: sidecar
      image: busybox
      command: ['sh', '-c', 'tail -f /logs/output.log']
      volumeMounts:
        - name: sharedvol
          mountPath: /logs
  volumes:
    - name: sharedvol
      emptyDir: {}
```

Workshop 7 Multi-Container and Init Containers

02-initial-container.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-07
  name: shipping-backend
  labels:
    app: shipping-svc
spec:
  containers:
    - name: nginx
      image: nginx:latest
---
apiVersion: v1
kind: Service
metadata:
  namespace: workshop-07
  name: shipping-svc
spec:
  selector:
    app: shipping-svc
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
---
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-07
  name: shipping-web
spec:
  containers:
    - name: nginx
      image: nginx:latest
  initContainers:
    - name: shipping-svc-check
      image: curlimages/curl
      command: ['sh', '-c', 'until curl -s -f -o /dev/null "http://shipping-svc"; do echo waiting for shipping-svc; sleep 5; done']
```

Workshop 7 Multi-Container and Init Containers



What Is Scheduling?

Kubernetes Scheduler: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
Assigning Pods to Nodes: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

Scheduling Process

The Kubernetes scheduler selects a suitable Node for each Pod. It takes into account things like:

- Resource requests vs. available node resources
- Various configurations that affect scheduling using node labels

nodeSelector

You can configure a **nodeSelector** for your Pods to limit which Node(s) the Pod can be scheduled on.

Node selectors use node labels to filter suitable nodes.

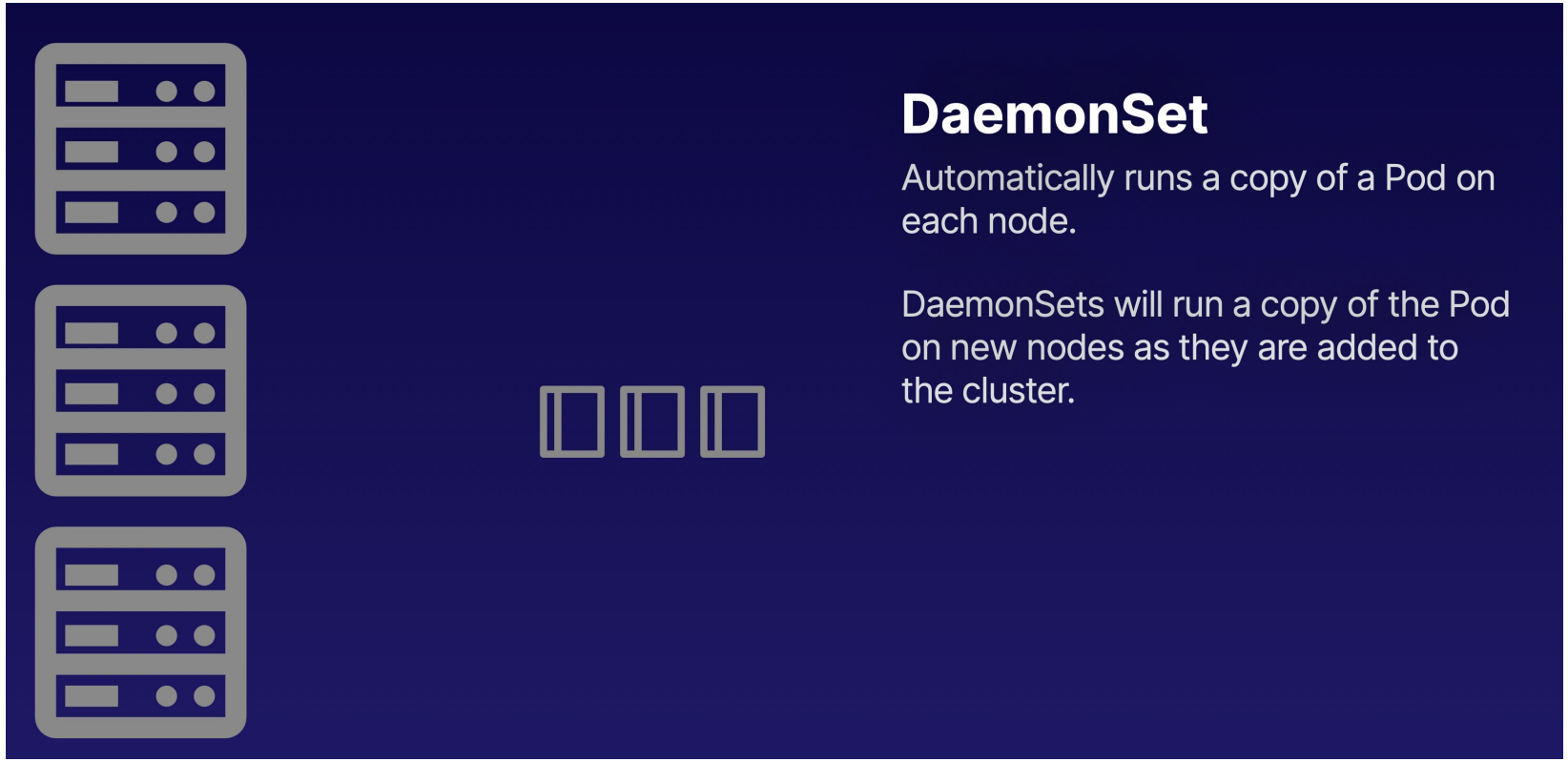
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    mylabel: myvalue
```

nodeName

You can bypass scheduling and assign a Pod to a specific Node by name using **nodeName**.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeName: k8s-worker1
```

Scheduling Process and Node Selector



DaemonSet:
https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/

DaemonSet

Workshop 8 Scheduling and DaemonSets

01-nodeselector-pod.yml

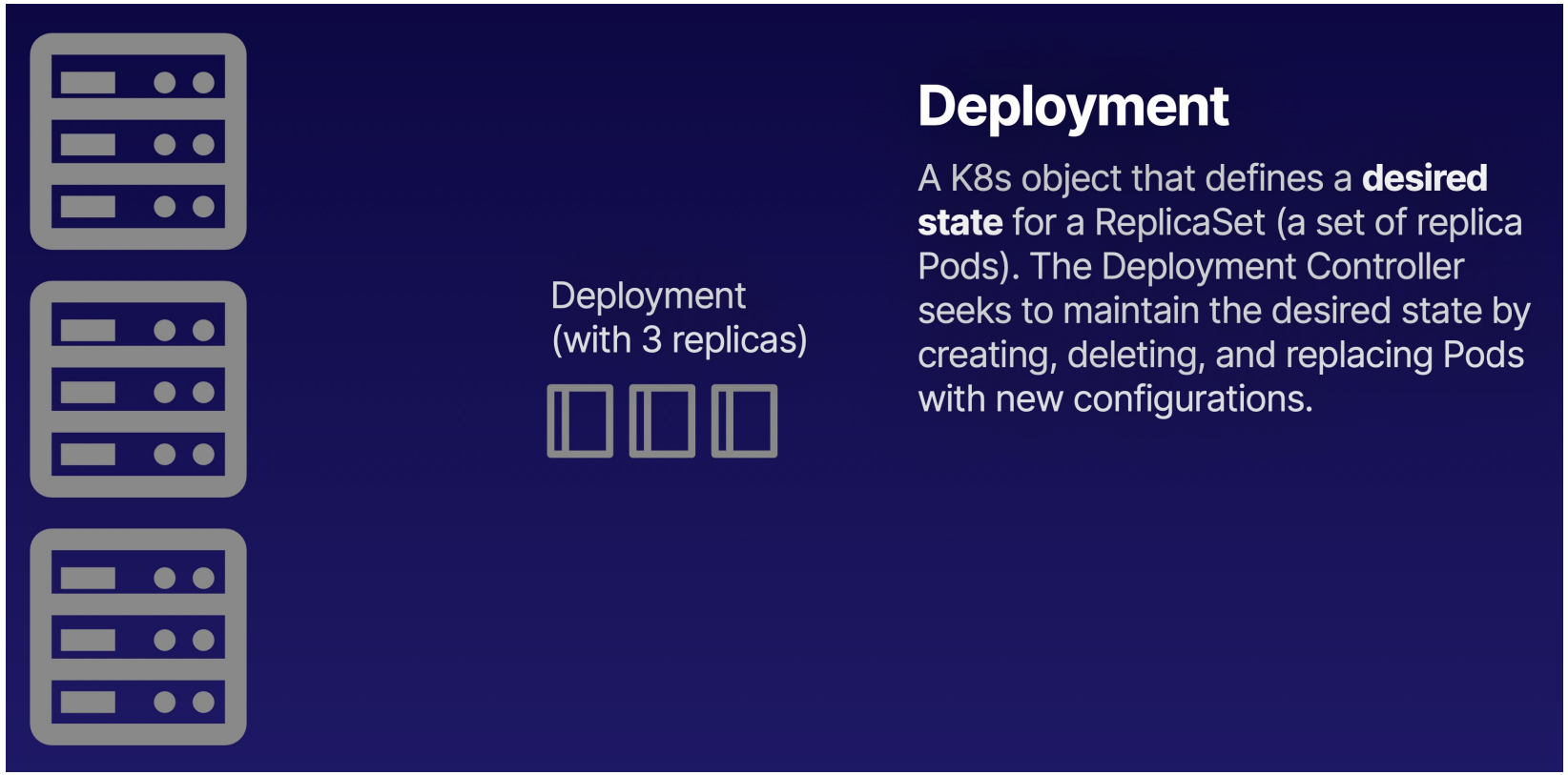
```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-08
  name: frontend-pod
spec:
  nodeSelector:
    worker-type: frontend
  containers:
    - name: nginx
      image: nginx:latest
---
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-08
  name: service-pod
spec:
  nodeSelector:
    worker-type: service
  containers:
    - name: echo
      image: ealen/echo-server:latest
---
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-08
  name: database-pod
spec:
  nodeSelector:
    worker-type: database
  containers:
    - name: redis
      image: redis:latest
```

02-nodename-pod.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-08
  name: nodename-pod
spec:
  nodeName: k3d-mycluster-agent-0
  containers:
    - name: nginx
      image: nginx:latest
```

03-daemonset-pod.yml

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  namespace: workshop-08
  name: my-daemonset
spec:
  selector:
    matchLabels:
      app: my-daemonset
  template:
    metadata:
      labels:
        app: my-daemonset
    spec:
      containers:
        - name: nginx
          image: nginx:latest
```



Deployments:
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Deployment

Deployment Desired State

Scaling refers to dedicating more (or fewer) resources to an application in order to meet changing needs.

K8s Deployments are very useful in **horizontal scaling**, which involves changing the number of containers running an application.

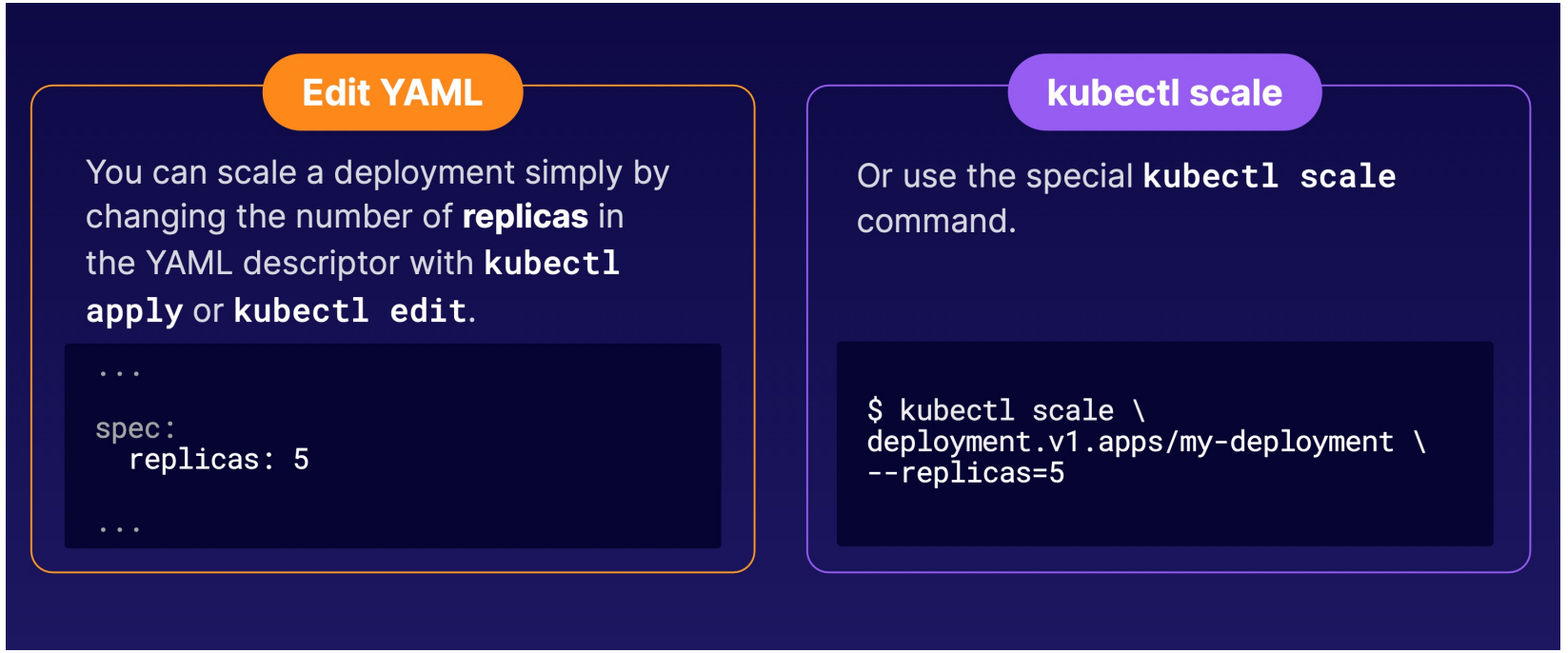
Deployment
Replicas: 3

Deployment Scaling

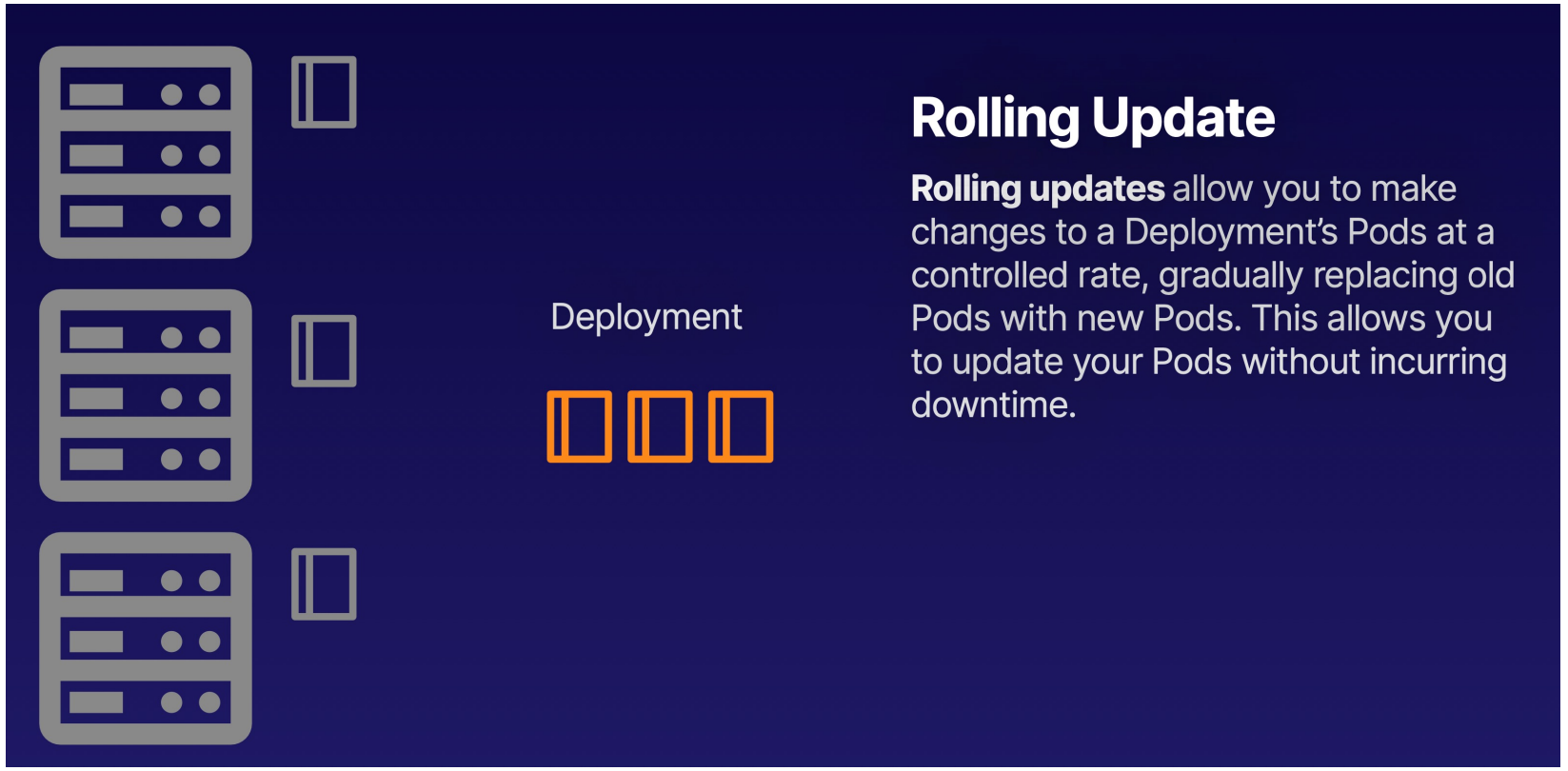
The Deployment's **replicas** setting determines how many replicas are desired in its desired state. If the **replicas** number is changed, replica Pods will be created or deleted to satisfy the new number.

Scaling with Deployments

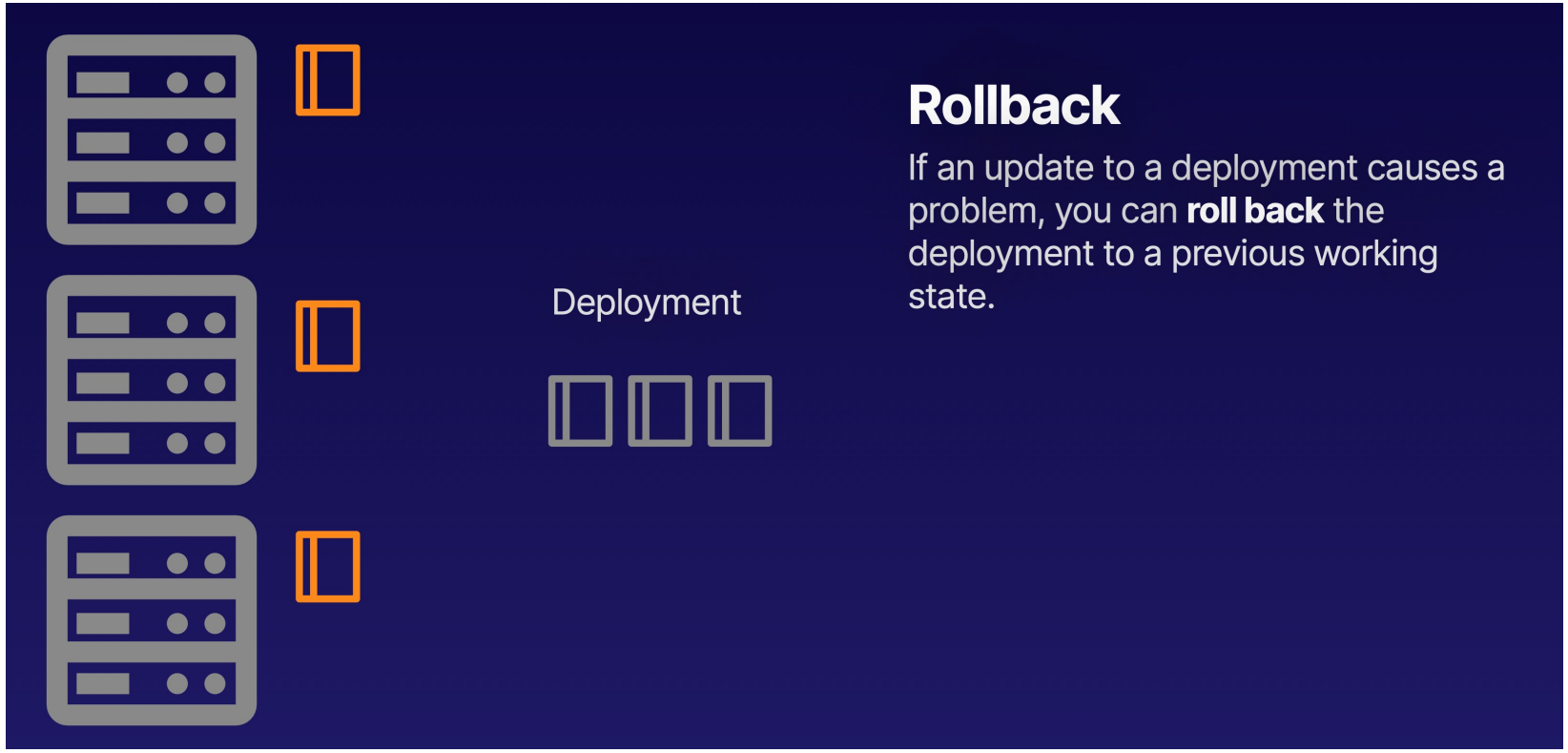
How to Scale



Scaling a Deployment:
https://kubernetes.io/docs/concepts/workloads/controllers/deployment/#scaling-a-deployment



Rolling Update


Rollback
If an update to a deployment causes a problem, you can **roll back** the deployment to a previous working state.
Deployment

Rollback

Workshop 9
Deployment

01-my-deployment.yml

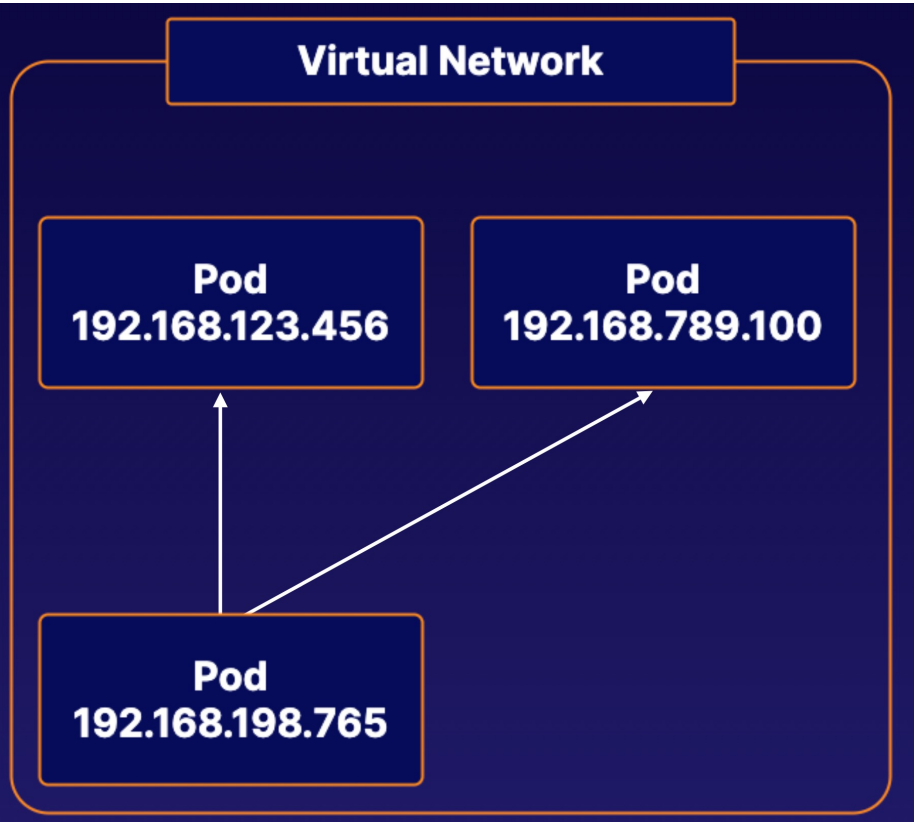
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: workshop-09
  name: my-deployment
spec:
  replicas: 6
  selector:
    matchLabels:
      app: my-deployment
  template:
    metadata:
      labels:
        app: my-deployment
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80

```

Network Model

The Kubernetes network model defines how Pods communicate with each other, regardless of which Node they are running on.

Each Pod has its own **unique IP address** within the cluster.

Any Pod can reach any other Pod using that Pod's IP address. This creates a **virtual network** that allows Pods to easily communicate with each other, regardless of which node they are on.



DNS in K8s

The K8s virtual network uses a **DNS** (Domain Name System) to allow Pods to locate other Pods and Services using domain names instead of IP addresses.

This DNS runs as a Service within the cluster. You can usually find it in the **`kube-system`** namespace.

Kubeadm clusters use **CoreDNS**.

Pod Domain Names

All Pods in our kubeadm cluster are automatically given a domain name of the following form:

```
pod-ip-address.namespace-name.pod.cluster.local
```

A Pod in the **`default`** namespace with the IP address 192.168.10.100 would have a domain name like this:

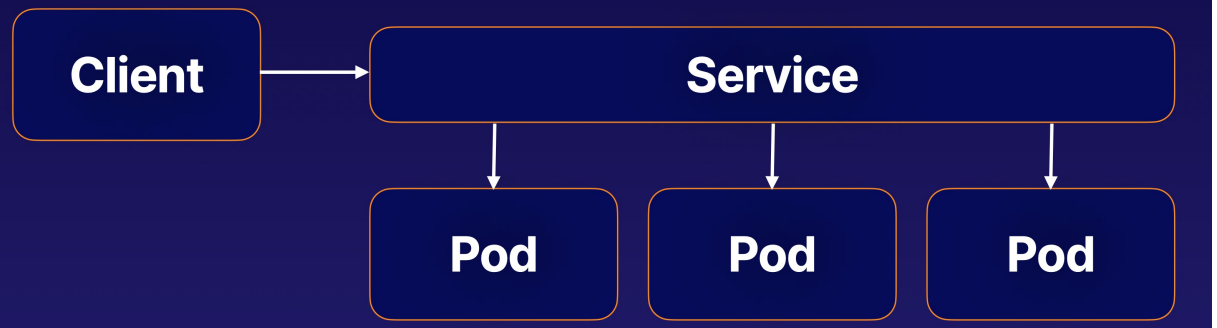
```
192-168-10-100.default.pod.cluster.local
```

DNS for Services and Pods: https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/

Kubernetes **Services** provide a way to expose an application running as a set of Pods.

They provide an abstract way for clients to access applications without needing to be aware of the application's Pods.

Clients make requests to a Service, which **routes** traffic to its Pods in a load-balanced fashion.



Service

Endpoints are the backend entities to which Services route traffic. For a Service that routes traffic to multiple Pods, each Pod will have an endpoint associated with the Service.

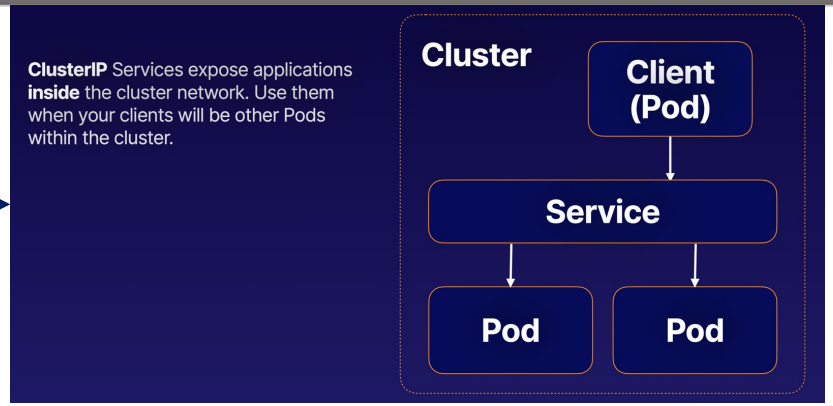
Tip: One way to determine which Pod(s) a Service is routing traffic to is to look at that service's Endpoints.

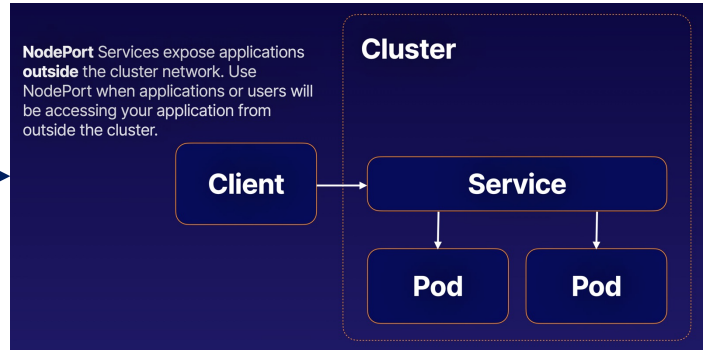
Endpoint

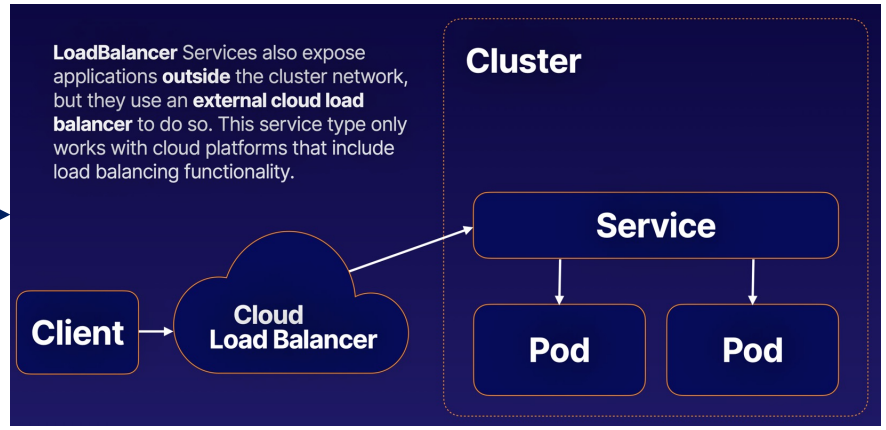
Service Types

Each Service has a type. The **Service type** determines how and where the Service will expose your application. There are four service types:

- ClusterIP
- NodePort
- LoadBalancer
- ExternalName







The Kubernetes DNS (Domain Name System) assigns **DNS names** to Services, allowing applications within the cluster to easily locate them.

A service's fully qualified domain name has the following format:

```
service-name.namespace-name.svc.cluster-domain.example
```

Service DNS and Namespaces

A Service's fully qualified domain name can be used to reach the service from within **any Namespace** in the cluster.

However, Pods within the same Namespace can also simply use the service name.

my-service

Discovering Services With DNS

An **Ingress** is a Kubernetes object that manages external access to Services in the cluster.

An Ingress is capable of providing more functionality than a simple NodePort Service, such as SSL termination, advanced load balancing, or name-based virtual hosting.

Ingress

Routing to a Service

Ingresses define a set of **routing rules**. A routing rule's properties determine to which requests it applies.

Each rule has a set of **paths**, each with a **backend**. Requests matching a path will be routed to its associated backend.

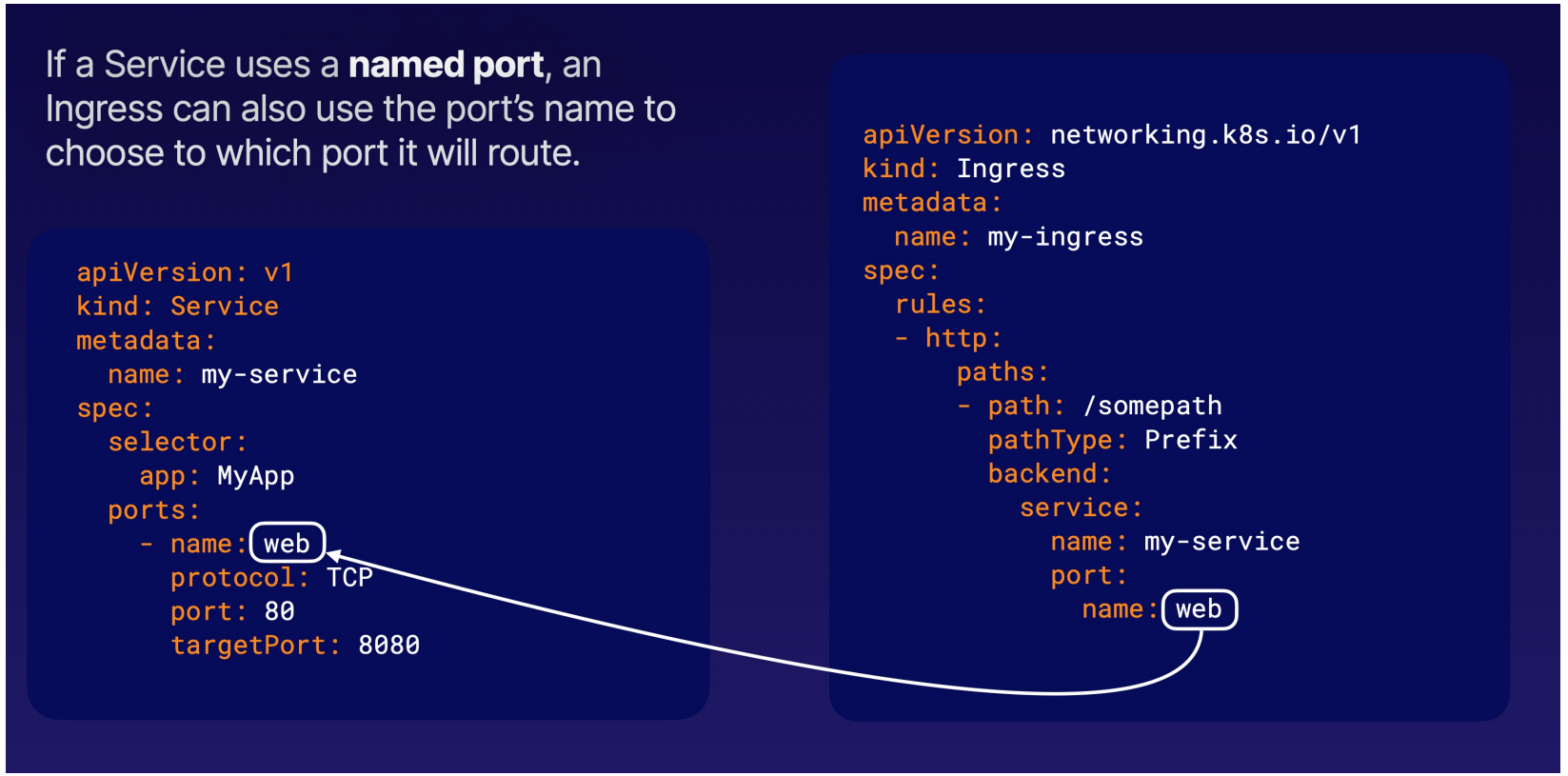
In this example, a request to *http://<some-endpoint>/somepath* would be routed to port 80 on the `my-service` Service.

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
      paths:
      - path: /somepath
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              number: 80
```

If a Service uses a **named port**, an Ingress can also use the port's name to choose to which port it will route.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - name: web
      protocol: TCP
      port: 80
      targetPort: 8080
```

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
spec:
  rules:
  - http:
      paths:
      - path: /somepath
        pathType: Prefix
        backend:
          service:
            name: my-service
            port:
              name: web
```



Routing to a Service

01-dnstest.yml

```yaml
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-10
  name: netshoot
spec:
  containers:
    - name: netshoot
      image: nicolaka/netshoot
      command: ['sh', '-c', 'while true; do sleep 5; done']
      securityContext:
        privileged: true
---
apiVersion: v1
kind: Pod
metadata:
  namespace: workshop-10
  name: nginx-dnstest
spec:
  containers:
    - name: nginx
      image: nginx:latest
      ports:
        - containerPort: 80
```

Workshop 10
Service and Ingress

Workshop 10
Service and Ingress

02-frontend-app.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  namespace: workshop-10
  name: frontend-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: frontend-app
  template:
    metadata:
      labels:
        app: frontend-app
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80

```

03-svc-clusterip.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  namespace: workshop-10
  name: svc-clusterip
spec:
  type: ClusterIP
  selector:
    app: frontend-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

04-svc-nodeport.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  namespace: workshop-10
  name: svc-nodeport
spec:
  type: NodePort
  selector:
    app: frontend-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30080
```

05-svc-external.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  namespace: workshop-10
  name: chula-hospital
spec:
  type: ExternalName
  externalName: chulalongkornhospital.go.th
```

Workshop 10
Service and Ingress

06-my-ingress.yml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  namespace: workshop-10
  name: my-ingress
  annotations:
    ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
    - http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: svc-clusterip
                port:
                  number: 80
```

Workshop 10
Service and Ingress

Container File Systems

The container file system is **ephemeral**. Files on the container's file system exist only as long as the container exists.

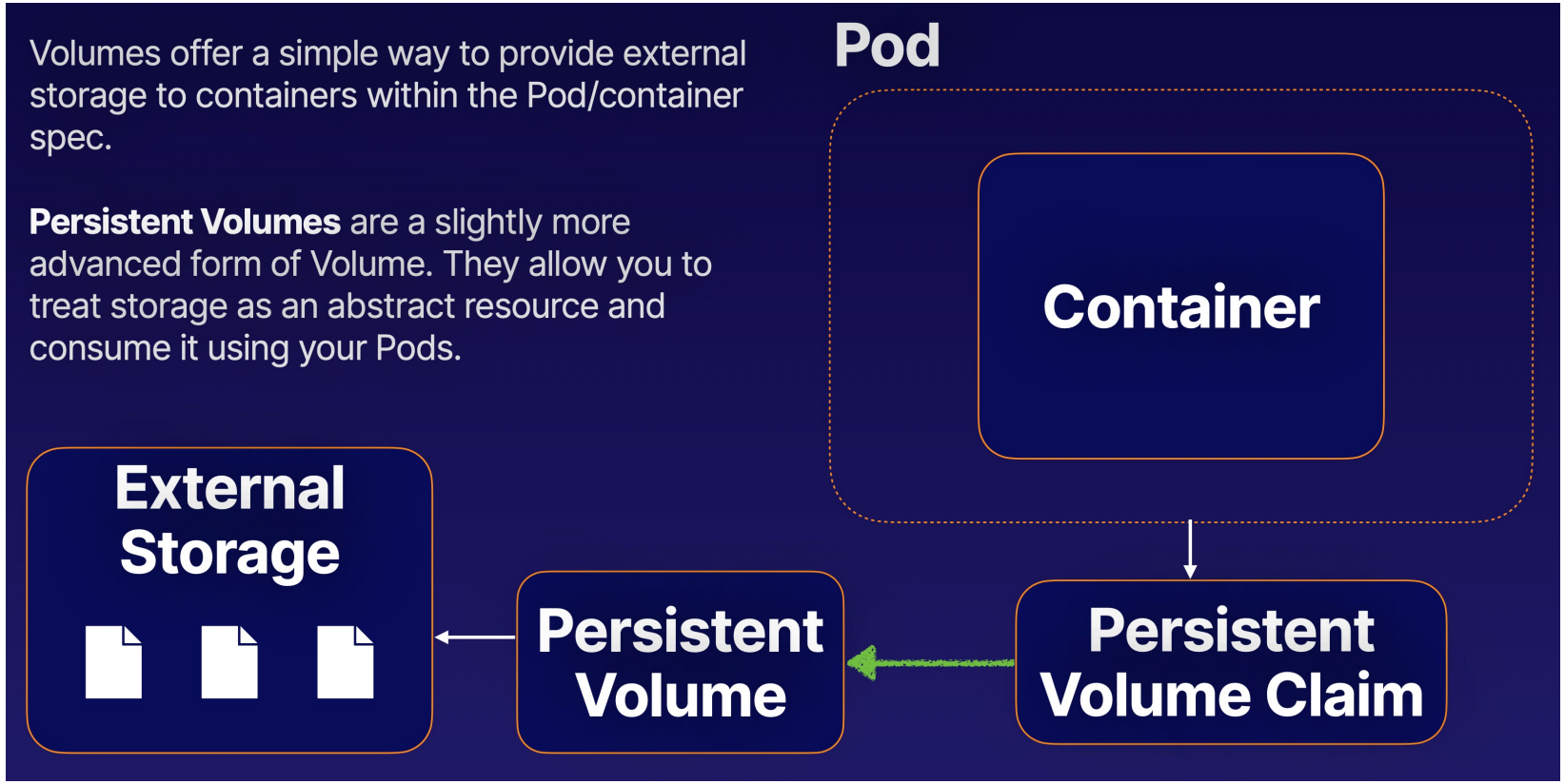
If a container is deleted or re-created in K8s, data stored on the container file system is lost.

Many applications need a more persistent method of data storage.

Volumes allow you to store data outside the container file system while allowing the container to access the data at runtime.

This can allow data to persist beyond the life of the container.

Volumes



Persistent Volumes

Volume Types

Both Volumes and Persistent Volumes each have a **volume type**. The volume type determines how the storage is actually handled.

Various volume types support storage methods such as:

- NFS
- Cloud storage mechanisms (AWS, Azure, GCP)
- ConfigMaps and Secrets
- A simple directory on the K8s node

Volumes and volumeMounts

Regular Volumes can be set up relatively easily within a Pod/container specification.

volumes: In the Pod spec, these specify the storage volumes available to the Pod. They specify the volume type and other data that determines where and how the data is actually stored.

volumeMounts: In the container spec, these reference the volumes in the Pod spec and provide a `mountPath` (the location on the file system where the container process will access the volume data).

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
  - name: busybox
    image: busybox
    volumeMounts:
    - name: my-volume
      mountPath: /output
  volumes:
  - name: my-volume
    hostPath:
      path: /data
```

Sharing Volumes Between Containers

You can use `volumeMounts` to mount the same volume to multiple containers within the same Pod.

This is a powerful way to have multiple containers interact with one another. For example, you could create a secondary sidecar container that processes or transforms output from another container.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: volume-pod
spec:
  containers:
  - name: busybox1
    image: busybox
    volumeMounts:
    - name: my-volume
      mountPath: /output
  - name: busybox2
    image: busybox
    volumeMounts:
    - name: my-volume
      mountPath: /input
  volumes:
  - name: my-volume
    emptyDir: {}
```

Common Volume Types

There are many volume types, but there are two you may want to be especially aware of.

hostPath: Stores data in a specified directory on the K8s node.

emptyDir: Stores data in a dynamically created location on the node. This directory exists only as long as the Pod exists on the node. The directory and the data are deleted when the Pod is removed. This volume type is very useful for simply sharing data between containers in the same Pod.

PersistantVolumes

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: my-pv
spec:
  storageClassName: localdisk
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /var/output
```

PersistentVolumes are K8s objects that allow you to treat storage as an abstract resource to be consumed by Pods, much like K8s treats compute resources such as memory and CPU.

A PersistentVolume uses a set of attributes to describe the underlying storage resource (such as a disk or cloud storage location) which will be used to store data.

Storage Classes

Storage Classes allow K8s administrators to specify the types of storage services they offer on their platform.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: localdisk
provisioner: kubernetes.io/no-provisioner
```

Storage Classes

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/no-provisioner
```

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/no-provisioner
```

For example, an administrator could create a StorageClass called **slow** to describe low-performance but inexpensive storage resources, and another called **fast** for high-performance but more costly resources.

This would allow users to choose storage resources that fit the needs of their applications.

The **allowVolumeExpansion** property of a StorageClass determines whether or not the StorageClass supports the ability to resize volumes after they are created.

If this property is not set to true, attempting to resize a volume that uses this StorageClass will result in an error.

```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: localdisk
provisioner: kubernetes.io/no-provisioner
allowVolumeExpansion: true
```

allowVolumeExpansion

A PersistentVolume's **persistentVolumeReclaimPolicy** determines how the storage resources can be reused when the PersistentVolume's associated PersistentVolumeClaims are deleted.

- **Retain:** Keeps all data. This requires an administrator to manually clean up the data and prepare the storage resource for reuse.
- **Delete:** Deletes the underlying storage resource automatically (only works for cloud storage resources).
- **Recycle:** Automatically deletes all data in the underlying storage resource, allowing the PersistentVolume to be reused.

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: my-pv
spec:
  storageClassName: localdisk
  persistentVolumeReclaimPolicy: Recycle
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /etc/output
```

reclaimPolicies

A **PersistentVolumeClaim** represents a user's request for storage resources. It defines a set of attributes similar to those of a PersistentVolume (StorageClass, etc.).

When a PersistentVolumeClaim is created, it will look for a PersistentVolume that is able to meet the requested criteria. If it finds one, it will automatically be **bound** to the PersistentVolume.


```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: localdisk
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 100Mi
```

PersistentVolumeClaims

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pv-pod
spec:
  containers:
  - name: busybox
    image: busybox
    volumeMounts:
    - name: pv-storage
      mountPath: /output
  volumes:
  - name: pv-storage
    persistentVolumeClaim:
      claimName: my-pvc
```

PersistentVolumeClaims can be **mounted** to a Pod's containers just like any other volume.

If the PersistentVolumeClaim is bound to a PersistentVolume, the containers will use the underlying PersistentVolume storage.

Using a PersistentVolumeClaims in a Pod

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  storageClassName: localdisk
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 200Mi
```

You can **expand** PersistentVolumeClaims without interrupting applications that are using them.

Simply edit the **spec.resources.requests.storage** attribute of an existing PersistentVolumeClaim, increasing its value.

However, the StorageClass must support resizing volumes and must have **allowVolumeExpansion** set to **true**.

Resizing a PersistentVolumeClaims

thankyou.™