Containers, Docker, Images, and Registries

What is a Container?

A container is a lightweight, executable unit of software that packages application code together with all the libraries and dependencies required to run it. The key idea behind containers is consistency: the same container can run unchanged on a developer's laptop, an on-premises server, or a cloud environment.
Containers achieve this by using **operating system–level virtualization**. Instead of virtualizing hardware, containers share the host operating system kernel while isolating application processes from one another.
On Linux systems, this isolation is enabled by:

- **Namespaces**, which isolate processes, networking, and filesystems
- **Control groups (cgroups)**, which control CPU, memory, and disk usage

Because containers reuse the host OS kernel, they avoid the overhead of running a full operating system per application.

Why Containers Are Lightweight and Portable

Containers are small, fast, and portable primarily because they do not include a guest operating system. This results in:

- Smaller image sizes (often a few megabytes)
- Faster startup times (seconds or less)
- Lower CPU and memory overhead

Portability comes from packaging everything an application needs to run inside the container itself. Once built, a container behaves the same across environments, eliminating the classic "works on my machine" problem.

Containers vs Virtual Machines

Virtual machines and containers both provide isolation, but they do so at different layers.

Virtual machines virtualize hardware. Each VM runs its own operating system on top of a hypervisor, along with application binaries and libraries. This provides strong isolation but results in larger images and slower startup times.
Containers virtualize the operating system. A single host OS is shared across all containers, and a container runtime manages isolated execution environments for applications.
In practice:

- Virtual machines are heavier but support multiple operating systems
- Containers are lighter and better suited for microservices and rapid scaling

---

Why Containers Matter in Modern Development

Containers align naturally with modern software practices:

- Microservice architectures, where applications are split into small, independent services
- DevOps workflows that require frequent, repeatable deployments
- Hybrid and multi-cloud setups where portability is essential
- Application modernization efforts that start by containerizing existing workloads

---

What "Docker" Means in Practice

The word *Docker* is used in multiple, valid ways, which often causes confusion.

In practical terms, Docker refers to:

- A **platform** for building and running containers
- A **command-line interface (CLI)** used to work with containers and images
- A **container runtime** that executes containers
- A set of tools used to create **container images**

Docker did not invent containers, but it made them easy to use and widely adopted by standardizing tooling and workflows.

Docker as a Container Runtime

A container runtime is the software responsible for running containers. It sits above the operating system and below the application.

A typical container stack looks like this:

- Physical infrastructure
- Host operating system
- Container runtime (for example, Docker or containerd)
- Containers running applications

The runtime handles tasks such as creating container processes, applying resource limits, and managing isolation.

Images and Containers: Understanding the Difference
Although often used interchangeably in casual conversation, **images** and **containers** are different concepts.
Image

- Immutable and read-only
- Contains application code, libraries, and dependencies
- Built once and reused
- Serves as a template

Container

- A running instance of an image
- Includes a writable layer on top of the image
- Executes application logic
- Typically short-lived

Image → class
Container → object (instance of that class)

The Container Development Flow

The container development process follows three fundamental steps:

1. Write a Dockerfile
2. Build an image from the Dockerfile
3. Run the image as a container

Supporting steps such as tagging and pushing images allow those images to be reused and deployed elsewhere.

Dockerfiles and Image Layers

A Dockerfile is a plain text file that describes how to build a container image. It contains a sequence of instructions that Docker executes in order. Each instruction creates a new, read-only layer in the image.

Because layers are immutable and shareable:

- Images can reuse common layers
- Disk usage is reduced
- Image downloads are faster

Common Dockerfile Instructions

Some instructions appear in almost every Dockerfile:

- `FROM` defines the base image and must be the first instruction
- `RUN` executes commands during image build
- `COPY` copies files from the local filesystem into the image
- `ADD` copies files and can also fetch remote URLs
- `ENV` sets environment variables
- `EXPOSE` documents which port the container listens on
- `CMD` defines the default command executed when the container starts

Only one `CMD` instruction is effective; if multiple are present, the last one wins.

Building an Image

Once a Dockerfile is written, an image is built using the Docker CLI:

```
docker build -t myimage:v1 .
```

The -t flag assigns a name and tag to the image. The final argument (.) specifies the build context, which is the directory whose contents are sent to the Docker daemon.

After the build completes, local images can be inspected with:

```
docker images
```

Running Containers

An image is executed as a container using docker run.

```
docker run myimage:v1
```

To run a container in the background and map ports:

```
docker run -dp 8080:8080 myimage:v1
```

Running containers can be listed with:

```
docker ps
```

Stopped containers can be viewed with:

```
docker ps -a
```

Tagging Images

Tagging allows the same image to be referenced by multiple names or versions without duplicating data.

```
docker tag myimage:v1 another-name:v1
```

Both tags point to the same image ID and underlying layers.

Container Registries

A container registry is a service used to store and distribute container images. While images can exist locally, registries are essential for collaboration, deployment, and automation.

Registries can be:

- Public or private

- Hosted or self-managed

Images stored in registries follow a standard naming format:

```
<registry>/<repository>:<tag>
```

For example:

```
docker.io/ubuntu:18.04
```

Pushing and Pulling Images

Images are uploaded to registries using a push operation and retrieved using a pull operation.

```
docker pull hello-world
docker run hello-world
```

```
docker push <registry>/<repository>:<tag>
```

During a push, Docker uploads image layers individually, skipping layers that already exist in the registry.

Container Orchestration and Kubernetes Fundamentals

What is Container Orchestration?

Container orchestration is the process of **automatically managing the lifecycle of containers** in environments where applications consist of many containers running across multiple machines. Orchestration becomes necessary as soon as systems move beyond a single container. As applications grow, containers must be deployed, restarted, scaled, distributed across infrastructure, and kept healthy. Doing this manually does not scale.

At a high level, container orchestration handles:

- Deployment and provisioning of containers
- Scheduling containers onto available infrastructure
- Scaling containers up and down based on demand
- Load balancing traffic across container instances
- Monitoring container health and replacing failed instances

Without orchestration, managing a large container environment quickly becomes error-prone and chaotic.

Why Kubernetes?

Kubernetes is the most widely adopted container orchestration platform. It was designed specifically to manage containerized workloads at scale and has become the de facto standard for orchestration. Kubernetes is:

- **Open source**, with a large global contributor base
- **Declarative**, allowing users to describe desired state instead of issuing step-by-step commands
- **Extensible**, with a broad ecosystem of tools and integrations
- **Platform-agnostic**, running on many types of infrastructure

Rather than being an opinionated platform that dictates tooling choices, Kubernetes provides a flexible foundation. It does not mandate CI/CD tools, logging systems, or monitoring solutions. As long as an application can run in a container, Kubernetes can run it.

Declarative Management: The Core Kubernetes Idea

One of Kubernetes' defining characteristics is declarative management.
Instead of telling the system *how* to perform each action, users describe *what the final state should be*.
Kubernetes continuously works to make the actual state match the desired state.
For example, if a configuration specifies that three replicas of an application should be running, Kubernetes ensures that exactly three are running. If one fails, Kubernetes creates another without user intervention.

This reconciliation loop is fundamental to how Kubernetes operates.

Kubernetes Cluster Architecture

A Kubernetes deployment is called a **cluster**. A cluster consists of two major parts:

- The **control plane**, which makes global decisions
- **Worker nodes**, which run user workloads

Control Plane Components

The control plane manages the overall state of the cluster and responds to changes.
Key components include:

- **API Server**
  The central access point for the cluster. All commands and internal communication go through the API server.
- **etcd**
  A distributed key-value store that holds the entire cluster state. Desired configurations and current state are stored here.
- **Scheduler**
  Assigns newly created Pods to nodes based on resource availability and scheduling rules.
- **Controller Manager**
  Runs controllers that monitor cluster state and take corrective action when actual state diverges from desired state.
- **Cloud Controller Manager**
  Integrates Kubernetes with underlying infrastructure providers while keeping Kubernetes itself cloud-agnostic.

Worker Nodes and Their Components

Worker nodes are the machines where application workloads run. They can be physical or virtual.
Each node includes:

- **kubelet**
  The primary node agent. It communicates with the API server, ensures Pods are running as specified, and reports health information.
- **Container Runtime**
  Responsible for pulling images and running containers. Kubernetes supports multiple runtimes through a standard interface.
- **kube-proxy**
  Manages network rules that enable communication to and between Pods.

Controllers and Control Loops
Controllers are processes that continuously monitor Kubernetes objects and take action to reconcile state.

A controller works like a thermostat:

- Desired state is defined (for example, 3 replicas)
- Actual state is observed
- Actions are taken to eliminate differences

Controllers never stop running. This constant reconciliation is what makes Kubernetes self-healing.

Kubernetes Objects: Defining Cluster State

Kubernetes objects are **persistent entities** that represent the desired state of the cluster. Once created, Kubernetes ensures these objects exist until they are modified or deleted.
Objects are manipulated via the Kubernetes API, typically using the `kubectl` CLI.

Every Kubernetes object has two key sections:

- **spec**: user-defined desired state
- **status**: system-reported current state

Kubernetes continuously works to align status with spec.

Namespaces

Namespaces provide **logical isolation** within a cluster. They allow a single cluster to behave like multiple virtual clusters.
Namespaces are useful for:

- Separating teams
- Isolating projects
- Preventing naming collisions

Object names must be unique **within a namespace**, but the same name can be reused across namespaces or across different resource types.

Labels and Selectors

Labels are key-value pairs attached to objects. They are not unique identifiers; instead, they describe attributes.
Selectors use labels to group objects. Controllers rely heavily on selectors to determine which objects they manage.
This labeling system enables flexible grouping without rigid hierarchy.

Pods: The Smallest Deployable Unit

A Pod is the simplest object in Kubernetes. It represents a single instance of an application.

Key characteristics:

- A Pod usually wraps one container
- Multiple containers are possible but typically tightly coupled

- Pods share networking and storage within the Pod

Pods are not designed to be created and managed manually at scale.

ReplicaSets: Scaling Pods

A ReplicaSet ensures that a specified number of identical Pods are running at all times.
Its responsibilities include:

- Creating Pods when replicas are missing
- Deleting Pods when there are too many
- Tracking Pods using label selectors

ReplicaSets add horizontal scaling capability to Pods.

Deployments: Managing Updates and Scaling

Deployments are higher-level objects that manage ReplicaSets and Pods.
Deployments are typically used for stateless applications and provide:

- Replica management
- Rolling updates
- Version control for workloads

A Deployment creates and owns ReplicaSets, which in turn manage Pods. Users usually interact with Deployments rather than ReplicaSets directly.

Rolling updates are a key feature. When an application version changes, Kubernetes gradually scales up the new version while scaling down the old one, minimizing downtime.

---

kubectl: The Kubernetes CLI

`kubectl` is the primary tool for interacting with a Kubernetes cluster. It communicates with the API server and manages objects.
Before using kubectl, it must be configured to target a specific cluster and namespace.
Common inspection commands include:

```
kubectl get pods

kubectl get deployments

kubectl describe pod <name>

kubectl describe deployment <name>
```

Imperative vs Declarative Commands

Imperative Commands

Imperative commands explicitly state what action to take.
Example:

```
kubectl run nginx --image nginx
```

They are:

- Easy to use
- Fast for experimentation
- Poor for reproducibility
- Unsuitable for production

Imperative Object Configuration

This approach uses configuration files but still specifies actions.
Example:

```
kubectl create -f pod.yaml
```

It improves repeatability but still requires users to know which operation to perform.

Declarative Commands

Declarative commands define desired state and let Kubernetes decide what actions are needed.
Example:

```
kubectl apply -f deployment.yaml

kubectl apply -f ./configs/
```

This approach:

- Maintains a single source of truth
- Handles both creation and updates
- Works well at scale
- Is preferred for production systems

Creating and Inspecting Resources

The typical workflow when deploying resources is:

1. Apply configuration
2. List resources
3. Inspect status

Common commands:

```
kubectl apply -f deployment.yaml

kubectl get deployments
kubectl get pods
```

```
kubectl describe deployment <name>
```

Services and Load Balancing

When multiple replicas of an application are running, Kubernetes can distribute traffic across them using a **Service**.

A Service:

- Provides a stable network endpoint
- Load balances requests across matching Pods
- Uses labels to determine target Pods

Creating a Service for a Deployment:

```
kubectl expose deployment/hello-world
```

This creates a ClusterIP service accessible inside the cluster.

Observing Load Balancing Behavior

When multiple replicas exist, repeated requests may be handled by different Pods. This demonstrates Kubernetes' built-in load balancing.

Pods can be terminated manually, and Kubernetes will automatically create replacements to maintain the desired replica count.

Cleaning Up Resources

Multiple resources can be deleted in a single command:

```
kubectl delete deployment/hello-world service/hello-world
```

This removes both the application and its network endpoint.

Scaling, Updating, and Configuring Kubernetes Applications

ReplicaSets and Application Scaling

What a ReplicaSet Does

A ReplicaSet is responsible for **maintaining a stable number of identical Pods** running at any given time. Its job is simple but critical: if the number of Pods drops below the desired count, it creates new Pods; if the number exceeds the desired count, it removes some.

ReplicaSets do not *own* Pods in a strict sense. Instead, they use **label selectors** to decide which Pods they manage. This loose coupling is intentional and aligns with Kubernetes' design philosophy of independent, composable objects.

In practice, ReplicaSets are rarely created directly. When you create a Deployment, Kubernetes automatically creates and manages a ReplicaSet on your behalf.

Observing ReplicaSets Created by Deployments

When a Deployment is created, a corresponding ReplicaSet appears automatically.

```
kubectl get rs
```

The ReplicaSet name typically includes the Deployment name followed by a hash. If you inspect a Pod created by the Deployment, you'll see it is *controlled by* that ReplicaSet.
This ReplicaSet ensures the number of running Pods always matches the desired replica count.

Scaling Applications with ReplicaSets

Scaling an application means changing the number of replicas managed by the ReplicaSet. This is usually done at the Deployment level.

```
kubectl scale deployment hello-world --replicas=3
```

After scaling:

```
kubectl get pods
```

You should see three Pods running. If you delete one of them manually, the ReplicaSet immediately creates a replacement Pod to restore the desired count. This behavior demonstrates Kubernetes' self-healing nature.
Scaling down works the same way:

```
kubectl scale deployment hello-world --replicas=1
```

The ReplicaSet deletes excess Pods until only one remains.

Autoscaling with Horizontal Pod Autoscaler (HPA)

Why Autoscaling Is Needed

Fixed replica counts are often inefficient. Traffic can fluctuate over time, and running too many Pods wastes resources while running too few can degrade performance.
Horizontal Pod Autoscaler (HPA) enables **dynamic scaling** based on observed metrics such as CPU utilization.

How Autoscaling Works

Autoscaling does not replace ReplicaSets. Instead, HPA **updates the replicas field** of the Deployment or ReplicaSet based on metrics. The control plane periodically checks Pod metrics and adjusts replicas to match the defined target.

Enabling Autoscaling from the CLI

Autoscaling can be enabled directly using the CLI:

```
kubectl autoscale deployment hello-world \
  --min=1 \
  --max=5 \
  --cpu-percent=10
```

This creates a HorizontalPodAutoscaler behind the scenes. The low CPU percentage here is intentional for demonstration purposes. You can verify autoscaling behavior by inspecting the Deployment and ReplicaSet replica counts.

Autoscaling Using YAML

Autoscaling can also be configured declaratively by creating a `HorizontalPodAutoscaler` object in YAML. This approach offers more control but is functionally equivalent to the CLI method. In practice, the CLI method is often sufficient for simple use cases.

Rolling Updates and Zero-Downtime Deployments

What Rolling Updates Solve

Updating an application by stopping all existing Pods and starting new ones causes downtime. Rolling updates avoid this by **gradually replacing old Pods with new ones**. Rolling updates are supported by Deployments and rely on ReplicaSets to manage transitions between versions.

Rolling Update Strategy

A Deployment can define how updates are performed using a rolling update strategy. Common parameters include:

- **maxUnavailable**: maximum number of Pods that can be unavailable during the update
- **maxSurge**: maximum number of extra Pods allowed during rollout
- **minReadySeconds**: minimum time a Pod must be ready before proceeding

These settings allow fine-grained control over availability during updates.

Performing a Rolling Update

A rolling update is triggered by changing the container image in a Deployment:

```
kubectl set image deployment/hello-world \
  hello-world=us.icr.io/<namespace>/hello-world:2
```

You can monitor progress with:

```
kubectl rollout status deployment/hello-world
```

Once the rollout completes, all Pods run the new image version without service interruption.

Rolling Back a Deployment

If a new version introduces issues, Kubernetes can roll back to the previous version:

```
kubectl rollout undo deployment/hello-world
```

Kubernetes terminates the new Pods and restores Pods from the earlier ReplicaSet. This rollback mechanism makes Deployments safe for production updates.

Configuration Management with ConfigMaps

Why ConfigMaps Exist

Hard-coding configuration values in application code is brittle and unsafe. Configuration often changes across environments, while code should not.
ConfigMaps provide a way to **externalize non-sensitive configuration** and inject it into Pods at runtime.

Creating ConfigMaps

ConfigMaps can be created in several ways.
Using string literals:

```
kubectl create configmap app-config \
  --from-literal=MESSAGE="This message came from a ConfigMap!"
```

Using a file containing key-value pairs:

```
kubectl create configmap app-config \
  --from-file=my.properties
```

Using a YAML descriptor:

```
kubectl apply -f my-config.yaml
```

Consuming ConfigMaps in Deployments

ConfigMaps can be consumed as:

- Environment variables
- Mounted files

Example using environment variables:

```
envFrom:
- configMapRef:
    name: app-config
```

Inside the application, values are accessed like any other environment variable.
Because configuration is separate from the image, updates to ConfigMaps can change application behavior without rebuilding images. A rollout restart is typically required to reload environment variables:

```
kubectl rollout restart deployment hello-world
```

Managing Sensitive Data with Secrets

What Secrets Are

Secrets are similar to ConfigMaps but are intended for **sensitive data** such as API keys, credentials, and tokens.
Kubernetes stores Secret values in encoded form and avoids displaying them in plain text by default.

Creating Secrets

Using string literals:

```
kubectl create secret generic api-creds \

  --from-literal=API_KEY=abcdef
```

You can verify creation:

```
kubectl get secrets
kubectl describe secret api-creds
```

Using Secrets in Applications
Secrets can be injected:

- As environment variables
- As mounted files

Environment variable example:

```
env:

- name: API_KEY
  valueFrom:
    secretKeyRef:
      name: api-creds
      key: API_KEY
```

Volume mount example:

```
volumes:
- name: secret-volume
  secret:
    secretName: api-creds
```

Mounted Secrets appear as files, allowing applications to read them securely at runtime.

Service Binding and External Services

What Service Binding Means

Service binding is the process of **connecting an external service to a Kubernetes application** by automatically providing credentials through Secrets.
Instead of manually injecting credentials, the binding process creates Secrets that the application can consume.

How Binding Works Conceptually
When a service is bound:

- Credentials are generated
- A Secret is created in the cluster
- The Secret is made available to Pods
- Applications consume credentials via environment variables or mounted files

This approach keeps credentials out of source code and container images.

Consuming Bound Services
Bound service credentials typically appear as environment variables such as:

- `binding.apikey`
- `binding.username`
- `binding.password`

Applications read these values and authenticate with the external service without embedding credentials directly in code.

Kubernetes Ecosystem, OpenShift, Operators, and Service Mesh

Why Kubernetes Needs an Ecosystem

Kubernetes intentionally focuses on **container orchestration**, not on the entire software delivery lifecycle. This design choice avoids locking users into a single opinionated workflow and instead enables flexibility.
Running real-world cloud-native applications requires more than orchestration alone. You need tools for:

- Building container images
- Storing and distributing images
- CI/CD automation
- Observability (logging, monitoring, tracing)
- Security and networking
- Traffic control between services

Rather than bundling all of this into Kubernetes itself, these concerns are handled by a **broader ecosystem of tools** that integrate cleanly with Kubernetes.

This ecosystem-first philosophy is a major reason Kubernetes has scaled so successfully.

CNCF and the Kubernetes Ecosystem

The **Cloud Native Computing Foundation** (CNCF) hosts Kubernetes and many other cloud-native projects. Its role is not just governance, but also **ecosystem navigation**.
Two CNCF resources are particularly useful conceptually:

- **Cloud-Native Trail Map**
  Shows the full cloud-native journey: containerization, CI/CD, orchestration, observability, security, and more. Kubernetes appears only in the orchestration layer, highlighting that many additional tools are required.
- **CNCF Landscape**
  A comprehensive map of open-source and commercial tools that work with Kubernetes. Its sheer size reflects how broad the ecosystem has become.

The takeaway is simple: Kubernetes is the core, not the whole platform.

Where OpenShift Fits

Red Hat OpenShift is not a Kubernetes alternative; it is a **Kubernetes distribution and platform built on top of Kubernetes**.
Where Kubernetes provides orchestration primitives, OpenShift layers additional capabilities to form a more complete application platform.

OpenShift Architecture at a High Level

At its core, OpenShift still runs Kubernetes. The difference lies in what surrounds it. Conceptually, the stack looks like this:

- Infrastructure (on-prem or cloud)
- Kubernetes control plane and worker nodes
- Integrated cluster services (networking, registry, monitoring)
- Platform services (CI/CD, logging, service mesh)
- Developer tools (CLI, web console, IDE integrations)

This structure turns Kubernetes into a **ready-to-use application platform**, especially for teams that prefer strong defaults and integrated workflows.

Kubernetes vs OpenShift: Key Differences

Kubernetes and OpenShift share a large common core, but differ in intent and experience.
Kubernetes

- Open-source project
- Flexible, unopinionated
- Requires users to assemble tooling
- Uses `kubectl` CLI
- Dashboard is optional and external

OpenShift

- Product built on Kubernetes
- Opinionated and integrated
- Provides CI/CD, registry, logging out of the box
- Uses `oc` CLI (superset of kubectl)
- Includes a first-class web console

A useful analogy is Linux:

- Kubernetes ≈ Linux kernel
- OpenShift ≈ a Linux distribution with tooling, defaults, and support

---

The oc CLI

The `oc` CLI is OpenShift’s command-line tool. It includes **all kubectl functionality** plus OpenShift-specific features.

Examples:

```
oc get pods
oc get buildconfigs

oc project
```

Because `oc` embeds `kubectl`, Kubernetes commands work unchanged while additional OpenShift objects become accessible.

Builds in OpenShift

OpenShift introduces **builds** as first-class objects. A build is the process of transforming input (usually source code) into an output (often a container image).
A **BuildConfig** defines *how* a build should run. A build is simply an execution of that configuration. This tightly integrates CI-like behavior directly into the platform.

Build Strategies

OpenShift supports multiple build strategies depending on how images are produced.
Docker Build Strategy

This strategy mirrors what you already know:

- Input: source code + Dockerfile
- Process: `docker build`
- Output: container image

OpenShift runs the build and pushes the resulting image to its internal registry automatically.

Source-to-Image (s2i)

Source-to-image removes the need to write a Dockerfile.

Instead:

- A **builder image** (for Node.js, Python, Java, etc.) is used
- Application source is injected into the builder image
- The result is a runnable container image

This approach significantly simplifies developer workflows and speeds up onboarding.

Custom Build Strategy

Custom builds allow complete flexibility:

- You define a custom builder image
- That image contains the logic to transform inputs into outputs
- Outputs don't have to be container images (could be artifacts like JARs)

This strategy is used for advanced or non-standard pipelines.

Build Triggers

Builds can be triggered automatically. OpenShift supports three trigger types:

- **Webhook triggers**
  Commonly used with Git repositories to trigger builds on commits or merges.
- **Image change triggers**
  Automatically rebuild applications when a base image is updated (useful for security patches).
- **Configuration change triggers**
  Trigger builds when the BuildConfig itself changes.

Together, these enable fully automated build pipelines without external CI tools.

ImageStreams

ImageStreams are an OpenShift abstraction for tracking images over time.

They do **not** store image data. Instead, they:

- Track immutable image digests
- Decouple Deployments from mutable tags like `latest`
- Provide controlled image updates

This prevents unintended rollouts when tags are reused and improves deployment stability.

Operators and the Operator Pattern

Kubernetes allows extending its API through **Custom Resource Definitions (CRDs)**. CRDs define new resource types, but on their own they only store data.
To make CRDs useful, you need **custom controllers** that interpret those resources and act on them.
The combination of:

- Custom resources (CRDs)
- Custom controllers (control loops)

is called the **Operator pattern**.
An **operator** encodes human operational knowledge into software.

What Operators Enable

Operators can:

- Deploy entire applications using a single custom resource
- Manage complex dependencies (Pods, Services, Secrets, storage)
- Automate operational tasks like backups, restores, and upgrades
- Continuously reconcile application state

Instead of manually managing dozens of Kubernetes objects, users interact with a single high-level resource.

OperatorHub
OpenShift provides **OperatorHub**, a curated catalog of operators that can be installed with minimal effort.
Operators fall into several categories:

- Platform-provided operators
- Certified third-party operators
- Community operators
- Custom, user-defined operators

This makes extending cluster capabilities both discoverable and standardized.

Microservices and Their Challenges

Microservices decompose applications into smaller, independently deployable services. This enables:

- Independent scaling
- Faster deployments
- Technology diversity across teams

However, microservices also introduce complexity:

- Secure service-to-service communication
- Traffic routing and versioning
- Failure isolation
- Observability across many services

These challenges are not solved by Kubernetes alone.

Service Mesh and Istio

A **service mesh** adds a dedicated layer for managing communication between services.
Istio is a widely used service mesh that integrates well with Kubernetes.
Istio does not replace Kubernetes networking; it augments it.

What Istio Provides

Istio’s capabilities fall into three main areas.

Traffic Management

- Traffic splitting and shifting
- Canary deployments
- A/B testing
- Intelligent request routing

Security

- Mutual TLS between services
- Authentication and authorization
- Fine-grained access control

Observability

- Request metrics
- Latency tracking
- Error rates
- Distributed tracing

These features operate transparently without requiring application code changes.

Why Istio Matters for Microservices

As the number of services grows, managing communication becomes more difficult than managing deployments themselves.
Istio centralizes:

- Traffic policies
- Security enforcement
- Service visibility

This allows teams to focus on business logic rather than networking concerns.
