OpenShift Primer

2025-04-17

Welcome

Welcome to the OpenShift Application Development Workshop!

Over the next two days, we'll dive deep into building, deploying, and managing applications on OpenShift. Whether you're new to OpenShift or looking to sharpen your skills, this workshop will provide hands-on experience and practical knowledge that you can apply right away.

Workshop Goals

1. Understand the fundamentals of OpenShift and its core components.

2. Learn how to develop, containerize, and deploy applications efficiently.

3. Explore best practices for application scaling.

4. Gain hands-on experience with troubleshooting and maintaining applications in an OpenShift environment.

Workshop Environment

To facilitate hands-on learning, this workshop provides the following tools:

- **A Terminal for CLI Testing**: Direct access to the OpenShift command-line interface (CLI) for efficient management and control of your applications.

- **A Web Console for Visual Interaction**: A user-friendly graphical interface to explore OpenShift projects, resources, and configurations.

These tools ensure a smooth and flexible experience, allowing you to interact with OpenShift both visually and through the command line while coding directly within the environment.

Agenda

Day 1

- **Background**:
  Why you might use OpenShift for application development.

- **What is OpenShift?**:
  Components used to run OpenShift

- **OpenShift vs Kubernetes**:
  How OpenShift and Kubernetes differ.

- **CLI and Web Console**:
  OpenShift's most common interfaces.

- **Projects**:
  OpenShift's workspace resource.

- **Application Deployment**:
  How to create an application image from source code and package it for OpenShift

- **OpenShift Networking**:
  Configuration options for exposing application traffic.

Day 2

- **Container Lifecycle**:
  Handling the image artifact in and out of the OpenShift cluster

- **Managing Configuration**:
  Parameterizing applications with dynamic information

- **Observability**:
  Monitoring your application metrics and logs with OpenShift

- **Scaling Applications**:
  Leveraging OpenShift's robust scaling capabilities

- **Debugging Applications**:
  Common workflows for identifying failure scenarios and fixes

- **Deployment Strategies**:
  Workloads beyond basic deployments

Who Should Attend

This workshop is designed for developers, DevOps engineers, and IT professionals who are either getting started with OpenShift or looking to enhance their application development skills on the platform.

Conclusion

By the end of the workshop, you'll walk away with a solid foundation in OpenShift application development and practical skills to accelerate your projects.

Let's get started!

Day One Agenda

-

-

-

-

-

-

-

Outcomes

By the end of day one, participants will be equipped with foundational knowledge and hands-on experience working with OpenShift. The following learning outcomes are expected:

- **Understand the Components that Comprise OpenShift**: Gain insight into the architecture of OpenShift, including its control plane, worker nodes, developer tools, and platform services.

- **Outline the Differences Between OpenShift and Kubernetes**: Identify how OpenShift builds upon Kubernetes by adding developer-focused tools, enterprise features, and integrated services.

- **Interact with an OpenShift Cluster**: Learn how to navigate and manage resources using both the OpenShift web console and the command-line interface (oc).

- **Manage Projects and Resources**: Understand how to create and manage OpenShift **projects**, and how to organize, monitor, and maintain the resources within them.

- **Create Applications from Source Code**: Use tools like Project Shipwright to securely build container images from source code inside a containerized environment—without requiring a Docker daemon.

- **Expose Applications with Customized Routing**: Learn how to create and configure OpenShift **routes** to expose services externally, including options for custom domains and TLS settings.

Background

How OpenShift and Kubernetes Help Application Developers

OpenShift and Kubernetes provide a powerful platform for building, deploying, and managing containerized applications. Together, they offer a range of capabilities that streamline the developer experience and support modern DevOps practices.

Developer Benefits

- **Simplified Deployment**: With built-in tools like Source-to-Image (S2I) and support for

alternative build tools such as Project Shipwright, OpenShift allows developers to quickly and securely build container images directly from source code—even in environments without a Docker daemon.

- **Consistent Environments**: Kubernetes ensures that applications run the same way across development, testing, and production environments.

- **Parameterized Environments**: ConfigMaps and Secrets allow developers to externalize environment-specific configurations—such as database URLs, feature flags, or credentials—without changing application code, enabling easier reuse and better separation of concerns.

- **Self-Service Access**: Developers can create, deploy, and manage applications independently through OpenShift's CLI, web console, and developer-friendly APIs.

- **Built-in CI/CD Support**: OpenShift Pipelines (based on Tekton) help automate build, test, and deploy workflows directly within the platform.

- **Scalability**: Kubernetes' orchestration engine automatically handles load balancing and scaling, allowing applications to grow with demand.

- **Integrated Monitoring and Logging**: Developers get visibility into application performance and logs through OpenShift's built-in tools, making debugging and optimization easier.

- **Security and Policy Management**: OpenShift enforces security best practices by default, including role-based access control (RBAC), policy, and secure default settings.

Why It Matters

These features reduce the operational burden on developers, allowing them to focus more on writing code and delivering value. OpenShift enhances the Kubernetes developer experience by abstracting complexity and providing a more integrated, opinionated platform tailored for enterprise use.
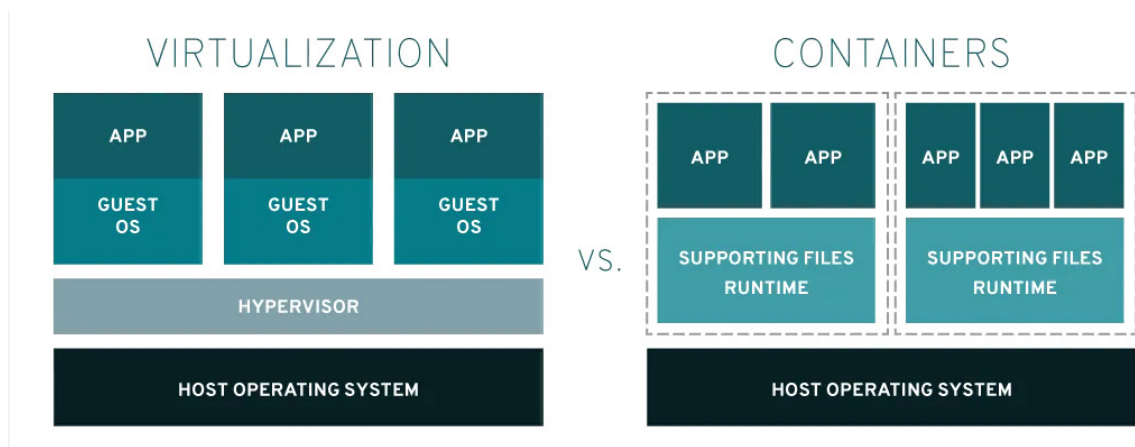
What is OpenShift?

OpenShift is an application platform

In simplest terms, OpenShift is a platform designed to help reliably run applications. It can be deployed on premise, in the cloud, directly on hardware, and also on various hypervisors. While the original focus of the platform was to run applications in linux containers it has since evolved to also run windows application containers and applications hosted by an array of virtualization providers.

> If you can't explain the difference between a virtual machine and a container, look `here`.

OpenShift is an open source application platform

Like the rest of the Red Hat portfolio, OpenShift is open source. While most users won't need to dive into the codebase to identify the source of a behavior, any user certainly can. The operating system (RHEL CoreOS), control components (Kubernetes), and layered software solutions (Operators) are all freely available to read, fork, and patch.

> Being open source has the added advantage of being able to run the platform without a dedicated subscription:
>
> - You can run a local version of OpenShift in a local VM with `Code Ready Containers(CRC)`.
>
> - You can run a distributed version of OpenShift on dedicated hardware or virtual machines using `OpenShift Kubernetes Distribution(OKD)`

OpenShift is an open source application platform built with Kubernetes

The original release of OpenShift actually predates the rise of modern standards like Kubernetes. With the release of OpenShift version 3 however, Kubernetes became the foundation of the platform. The OpenShift and Kubernetes communities thrive to this day for a number of reasons.

Just remember to PURSUE...

- **P**ortability - Kubernetes makes minimal demands on the underlying infrastructure, and can run on diverse sets of systems.

- **U**sability - Kubernetes configuration is API driven, and every change can be implemented with YAML syntax.

- **R**eliability - Kubernetes is self-healing, limiting the effects of service disruption.

- **S**calability - All of the components that comprise Kubernetes are designed with massive scale in mind.

- **U**nivesrality - Kubernetes has a large body of supporting documents from both the maintainers and the community.

- **E**xtensibility - The computing landscape is constantly shifting, and kubernetes was built to accommodate new runtimes, modes of operation, paradigms, etc.
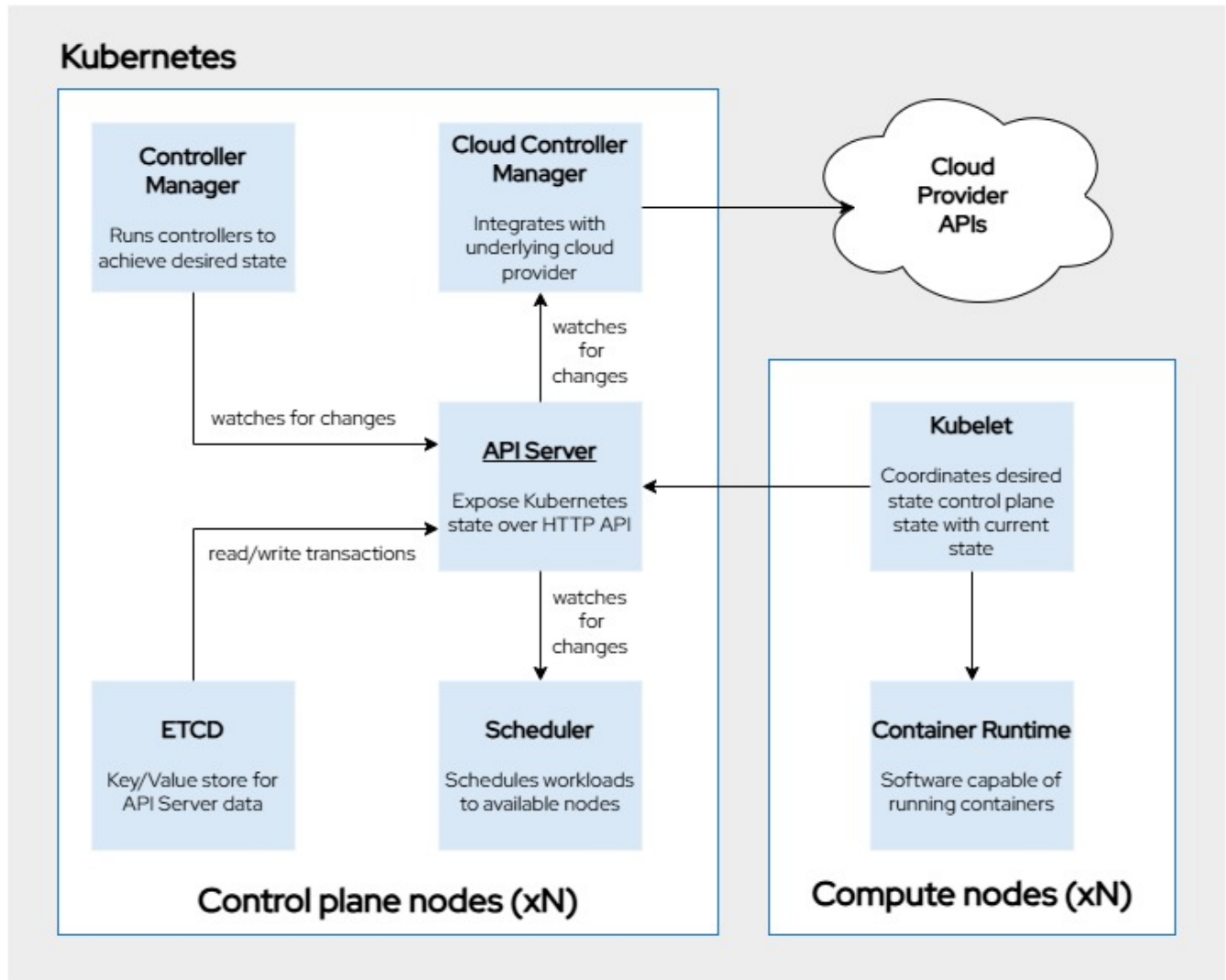
Kubernetes Basics

> The content in this section is a minimal description of Kubernetes. If you are new to Kubernetes please refer to these links for more comprehensive tutorials:
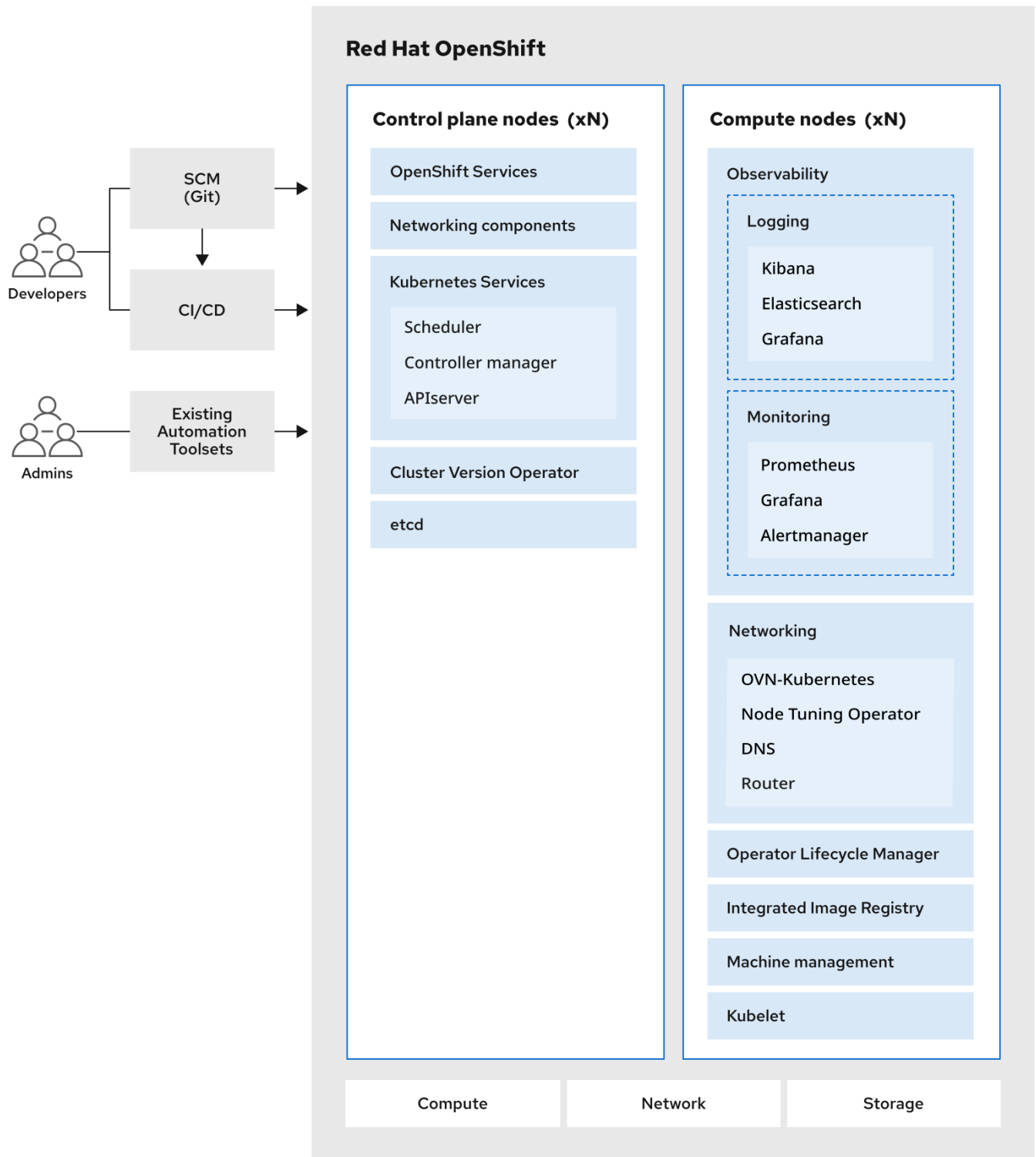>
> - What is Kubernetes?
>
> - Kubernetes 101
>
> - Learn Kubernetes Basics

Kubernetes Components

OpenShift is an open source application platform built with Kubernetes, that includes a suite of additional services to facilitate operations

OpenShift goes well beyond the capabilities of a basic Kubernetes cluster. We can see a few of the additional components in this diagram taken directly from the documentation.

Red Hat OpenShift

Control plane nodes (xN)

OpenShift Services

Networking components

Kubernetes Services

- Scheduler
- Controller manager
- APIserver

Cluster Version Operator

etcd

Compute nodes (xN)

Observability

Logging
- Kibana
- Elasticsearch
- Grafana

Monitoring
- Prometheus
- Grafana
- Alertmanager

Networking

- OVN-Kubernetes
- Node Tuning Operator
- DNS
- Router

Operator Lifecycle Manager

Integrated Image Registry

Machine management

Kubelet

| Compute | Network | Storage |

Developers

SCM (Git)

CI/CD

Admins

Existing Automation Toolsets

525-OpenShift-arch-012025

OpenShift provides "baked in" solutions for Networking, Observability, Machine Configuration, and Container Image Distribution.

OpenShift is an enterprise grade open source application platform built with Kubernetes, that includes a suite of additional services to facilitate operations

The final addition to our definition is "enterprise grade". OpenShift has been validated against a number of robust standards including but not limited to:

- NIST - National Institute of Standards and Technology
- NERC CIP - North American Electric Reliability Corporation Critical Infrastructure Protection
- PCI DSS - Payment Card Industry Data Security Standard
- DISA STIG - Defense Information Systems Agency Security Technical Implementation Guides
- CIS - Center For Internet Security Benchmarks

OpenShift takes much of the guess work out of securing a kubernetes platform with best practice configuration embedded directly into the infrastructure, platform, and runtimes.

References

- OpenShift Architecture Documentation
- Kubernetes Architecture Documentation

Knowledge Check

Can you explain what happens when you create a new resource in Kubernetes?

▼ *Answer*

Deep Dive

1. The API Server receives your request and validates your credentials and the content.
2. The API Server reformats the request and commits the information to ETCD.
3. Several Controllers begin remediating the difference in desired vs actual state.
4. The Scheduler assigns the resource to the most viable node.
5. The Kubelet identifies an update to the colocated node's state.
6. The Kubelet coordinates with the Container Runtime to create the workload.

How many components does OpenShift add to Kubernetes?

▼ *Answer*

From the list above:

- OpenShift Services
- Cluster Version Operator
- Observability
- Networking
- Operator Lifecycle Manager
- Integrated Image Registry
- Machine Management

From the docs

From a fresh install you can also run the following command to identify all of the additional

types associated with OpenShift:

```
oc api-resources -o name | grep -e ".*.openshift.io"
```

OpenShift vs Kubernetes

The relationship between OpenShift and Kubernetes parallels the relationship between YAML and JSON. YAML is considered a superset of JSON, meaning that it provides all of the functionality as well as some additional features. Kubernetes resources will *generally* operate within an OpenShift environment as is, or with some minor adjustments.
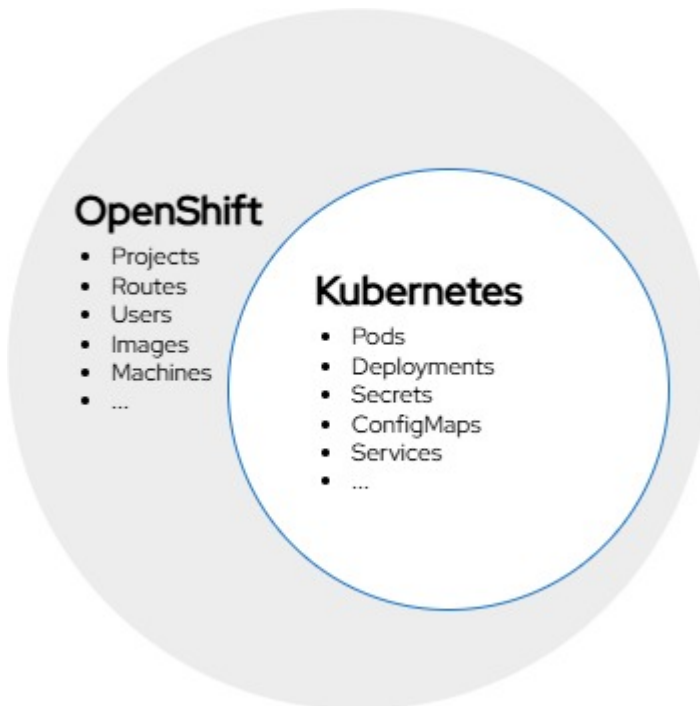
A Kubernetes engineer migrating to OpenShift will do so without much effort, but will not be making effective use of the entire platform.
Whereas an OpenShift engineer migrating to Kubernetes will have a higher barrier to migration, but will not be leaving functionality unused.

The following resources are some of the main resources OpenShift adds to Kubernetes. While they aren't truly essential to operating the platform, a cursory understanding will reduce toil and confusion. (sorted by decreasing correlation)

Projects

Kubernetes and OpenShift are designed to allow for multiple tenants. Taking a cue from the Linux kernel, both platforms provide isolated "namespaces" for a tenant's resources. Resources deployed in one namespace generally do not impact the behavior of resources deployed in another.

The differences between how OpenShift implements these isolated spaces with "Projects" differs from the Kubernetes implementation only slightly. Run the following command to compare the "default" `project` to the "default" `namespace`:

Linux

```
# Compare the YAML definition of each resource
diff <(oc get ns test -o yaml) <(oc get project test -o yaml)
```

Windows

```
# Create a file with the YAML definition of each resource and compare them
oc get namespace test -o yaml > %Temp%\test-namespace.yaml
oc get project test -o yaml > %Temp%\test-project.yaml
FC %Temp%\test-namespace.yaml %Temp%\test-project.yaml
```

OpenShift has been engineered to make the difference in these resources virtually transparent. When a namespace is created, OpenShift will create a corresponding project and vice-versa.

The only perceivable difference between these resources lies within a `project's ability to leverage OpenShift templates. OpenShift allows a privileged user to modify what is created when a user attempts to create a new project. link

Run the following to commands to observe the difference:

Linux

```
# Create a new project template and install it.
oc adm create-bootstrap-project-template -o yaml | oc apply -n openshift-config -f -
oc patch project.config.openshift.io/cluster --type=json -p
'[{"op":"add","path":"/spec/projectRequestTemplate","value":{"name":"project-
request"}}]'

# Confirm that a new project has an additional rolebinding (admin) while the new
namespace does not.
oc new-project example-project && oc create ns example-ns
diff <(oc get rolebindings -n example-project -o name) <(oc get rolebindings -n
example-ns -o name)
```

Windows

```
# Create a new project template and install it.
oc adm create-bootstrap-project-template -o yaml | oc apply -n openshift-config -f -
oc patch project.config.openshift.io/cluster --type=json -p
'[{"op":"add","path":"/spec/projectRequestTemplate","value":{"name":"project-
request"}}]'

# Confirm that a new project has an additional rolebinding (admin) while the new
namespace does not.
oc new-project example-project && oc create ns example-ns
oc get rolebindings -n example-project -o name > %Temp%\rolebindings-project.yaml
oc get rolebindings -n example-ns -o name > %Temp%\rolebindings-namespace.yaml
FC %Temp%\rolebindings-project.yaml %Temp%\rolebindings-namespace.yaml
```

> While it's true that namespaces isolate resources, they do not necessarily isolate the compute/memory/storage/network that underpins the platform. It's quite possible to degrade the performance of another tenant's resources by deploying resources in a separate namespace. Quotas will be introduced later to address this.

Web Console

Like projects, the additional Web Console that OpenShift provides does not directly impact the behavior of workloads or control components in any way. Unlike projects however, the difference between what OpenShift and Kubernetes provide is quite perceivable! Kubernetes by default **does not** include a graphical user interface, but OpenShift does. There are several open source options available for Kubernetes such as dashboard, lens, headlamp, devtron, but onboarding any one of them can be something of a burden when it comes to integration, upgrades, and compliance.

For most audiences, the web console will simply be an interface that allows users to "point and click" through their operations, but it can be modified to include:

- custom branding

- additional views

- quickstarts

- terminal environments

- even an AI assistant!
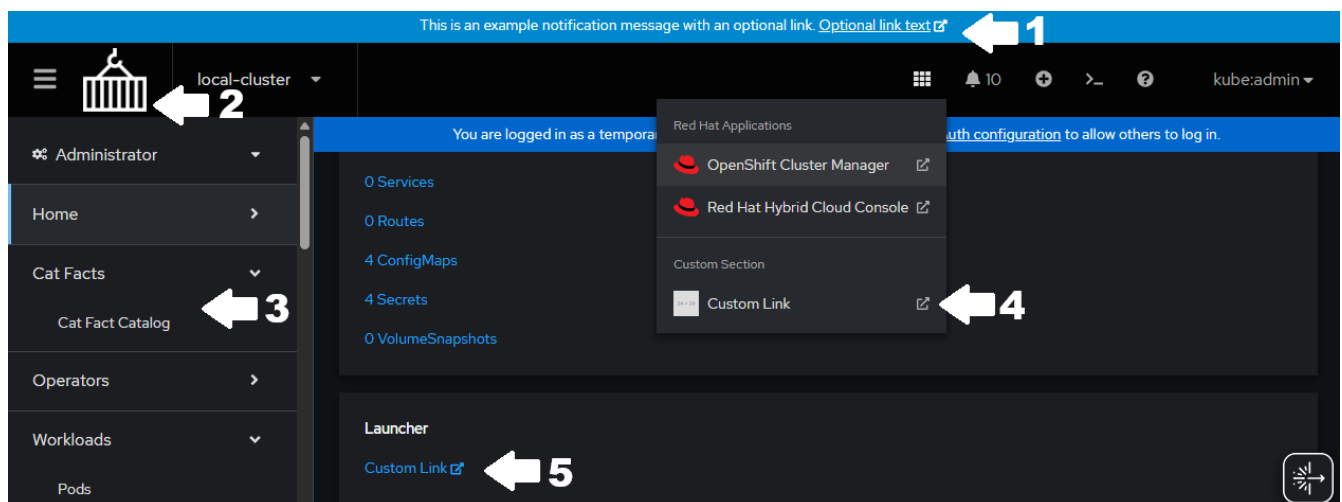
1. Custom Banners

2. Custom Logos

3. Custom Sections

4. Custom Applications

5. Custom Namespace Launchers

> State tuned!
> We will explore the Web Console in greater detail in the next module.

Routes

Routes are a specific implementation for the generic Kubernetes concern of external application exposure. Because applications hosted in Kubernetes often require a mechanism to serve them on extended/public/global networks, Kubernetes provides a "pluggable" system for this with resources like `ingress` and `gateways`.

Historically `routes` provided a simple and stable implementation before the ingress API became generally available. Today, they are the "batteries include" solution, but do not preclude the use of other ingress options.

> OpenShift will create Route objects for Ingresses that do not specify an `IngressClass`. This simplifies the adoption of generic Kubernetes configurations, but requires:
>
> - a `host` that aligns with the templated `*.apps."CLUSTER_NAME"."BASE_DOMAIN"`
> - annotations to configure TLS settings like:
>   - `route.openshift.io/termination` or
>   - `route.openshift.io/destination-ca-certificate-secret`

Open Virtual Network Container Network Interface (OVN)

> The topic of Container Network Interfaces is extensive, and what follows is **not** a comprehensive description.
>
> *For more in depth coverage of the CNI domain:*
> - Brief Overview of CNI
> - CNI Homepage
> - Kubernetes Networking Overview

There are many CNI's available for Kubernetes consumption. `Antrea`,`Calico`,`Cilium`,`NSX-T`,`OVN` are just a few that all implement the same specification, but in wildly different ways (eBPF, BGP, RDMA, Hardware Offloading…). The standard for OpenShift since version 4.12 is OVN. There are many specific features that OVN provides, but for the purposes of this workshop, only it's relationship with `kube-proxy` will be discussed.

OVN Kubernetes DOES NOT leverage `kube-proxy`!

In light of this, troubleshooting network connectivity should follow the procedures outlined here in. Standard `iptables` commands are effectively replaced with `ovn-nbctl` and `ovn-sbctl` from within the `ovnkube-node` workload.

You can confirm this by running:

Linux

```
# View the running configuration that indicates KubeProxy status
```

```
oc get network.operator.openshift.io cluster -o yaml | grep deployKubeProxy
```

Windows

```
# View the running configuration that indicates KubeProxy status
oc get network.operator.openshift.io cluster -o yaml | FINDSTR deployKubeProxy
```

Security

OpenShift takes `security` very seriously. The entire platform, from hardware to runtime, leverages comprehensive security tooling and practices such as encryption, selinux, seccomp, image signatures, system immutability, etc. Kubernetes can be made secure without additional tooling, but OpenShift enforces rather strict configurations by default.

The two primary sources of frustrations for users migrating from Kubernetes to OpenShift are:

- Security Context Constraints
    - Prevent elevated privileges for resources created by specific accounts
    - Enforced at admission on the pod level
- Pod Security Admission
    - Prevent elevated privileges broadly at the namespace level
    - Enforced at admission on the workload level
    - The Pod Security Standards are defined here

The correct approach to resolving issues of either type is to reconfigure the failing workload in order to comply with the default policy. There are several circumstances that might prevent this approach however. When the workload can not be configured in a compliant way:

- An appropriate scc must be added to the account associated with the running workload

```
oc adm policy add-scc-to-user "SCC" -z "SERVICE_ACCOUNT"
```

- Or the level of enforcement at the namespace level must be reduced

```
apiVersion: v1
kind: Namespace
metadata:
  annotations:
    openshift.io/sa.scc.mcs: s0:c31,c10
    openshift.io/sa.scc.supplemental-groups: 1000950000/10000
    openshift.io/sa.scc.uid-range: 1000950000/10000
  labels:
    kubernetes.io/metadata.name: test
    openshift-pipelines.tekton.dev/namespace-reconcile-version: 1.18.0
    pod-security.kubernetes.io/audit: restricted
    pod-security.kubernetes.io/audit-version: latest
```

```
    pod-security.kubernetes.io/warn: restricted
    pod-security.kubernetes.io/warn-version: latest
    # Add and configure the two lines below #
    pod-security.kubernetes.io/enforce: restricted
    pod-security.kubernetes.io/enforce-version: latest
  name: test
...
spec: {}
```

> ℹ️ In the previous namespace sample the three annotations are a security configuration associated with SCC's as well.
>
> These control SELinuxContext, Supplemental Groups, and Runnable UserIDs for workloads in a given namespace.

References

- Projects
- Web Console
- Routes
- Ingress Routes
- Open Virtual Network CNI

Knowledge Check

What project names are reserved for system use only?

▼ *Answer*

`openshift-*` and `kube-\*` are reserved project names.

Notice that there is nothing preventing namespaces with that format however:

```
oc new-project kube-example #Fails
oc create ns kube-example #Succeeds
```

What opensource tool is used to provide `route` support?

▼ *Answer*

HAProxy

You can confirm that with this command

```
oc exec -n openshift-ingress deploy/router-default -- /bin/sh -c "ps -ef | grep haproxy"
```

A workload needs to run with a User and Group ID within a range, how would you accomplish this without hardcoding the value in the container?

▼ *Answer*

You could create a new project template as was shown above, but add an annotation that specifies the proper range:

```
openshift.io/sa.scc.uid-range: XXXXX/10000
```

Command Line Interface and Web Console

Now that we've dispensed with the basics, it's time to start practical hands-on exploration of OpenShift. As we've already discussed and demonstrated, the two major methods for a user to interact with an OpenShift environment is through the `Web Console` or the `oc` cli.

Setup

Web Console

If you have an OpenShift cluster, then you likely already have the console available. The standard address location for the console follows this format:

- `https://` <~ the web console is secured with TLS
- `console-openshift-console` <~ "console" is the name of the workload and "openshift-console" is the name of the project in which it exists
- `.apps` <~ is the standard endpoint that all `route` based traffic is exposed
- `."CLUSTER_NAME"` <~ remember this for when you track multiple OpenShift environments
- `."BASE_DOMAIN"` <~ set at install time

Confirm the web console is healthy like so:

```
curl https://console-openshift-console.apps."CLUSTER_NAME"."BASE_DOMAIN"/health
```

Command Line Client

The `oc` cli requires a little more effort to setup, but nothing extraordinary.

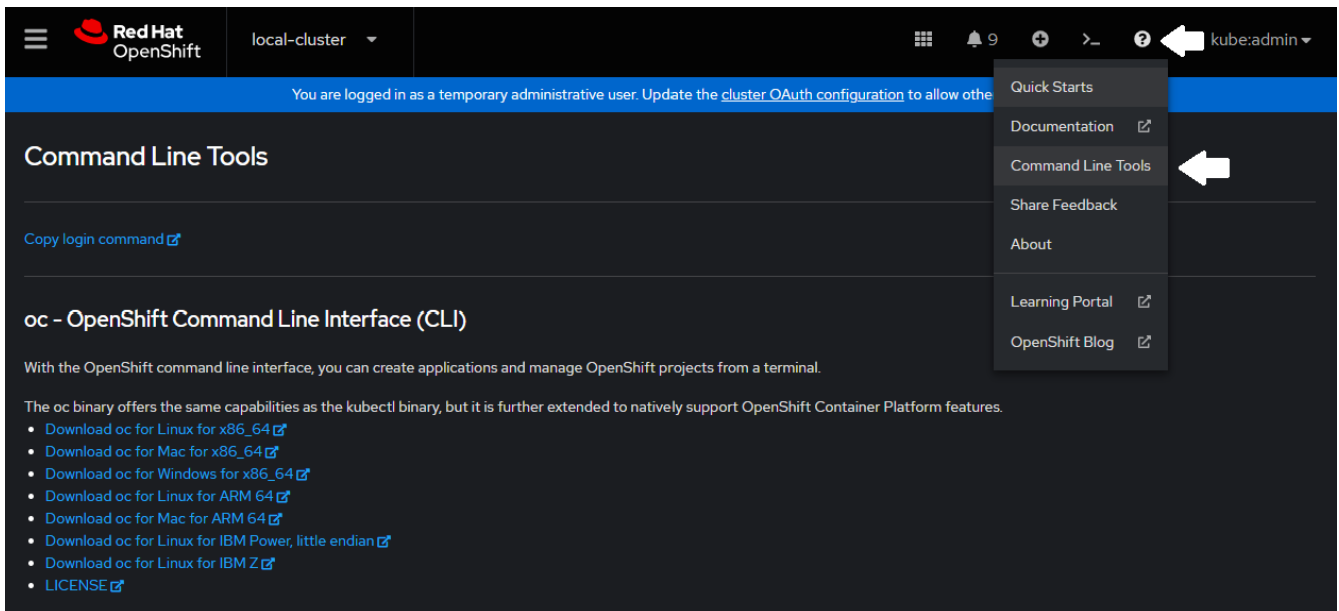You can find the appropriate CLI on a running cluster using the web console:

Figure 1. Download the CLI from the Console

Or you can download with using well defined URL endpoints:

Local Cluster

```
curl -L https://downloads-openshift-
console.apps."CLUSTER_NAME"."BASE_DOMAIN"/"ARCHITECTURE"/"OPERATING_SYSTEM"/oc.tar |
tar -xvf -
```

Public Artifact Store

```
curl -L https://mirror.openshift.com/pub/openshift-
v4/"ARCHITECTURE"/clients/ocp/stable/openshift-client-linux-"RELEASE_VERSION".tar.gz |
tar -zxvf -
```

Locating Resources

Web Console

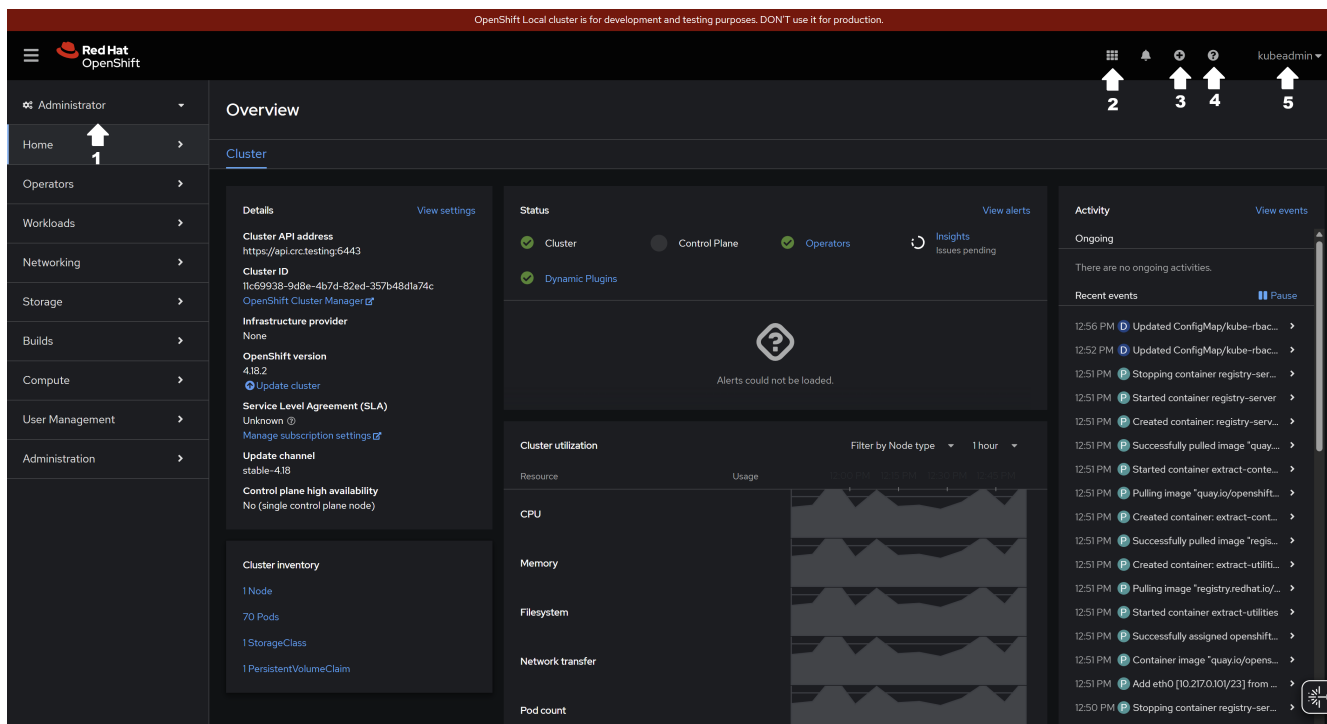The design of the console is intuitive, but here are some of the primary areas of interest:

Figure 2. Administrator Console Basics

1. `Role Dropdown`: Navigate between Administration and Development

2. `Red Hat Applications Selector`: Additional Red Hat applications such as **Cluster Manager** and **Hybrid Cloud Console** can be accessed here

3. `Quick Import Options`: Quickly add new resources to OpenShift with yaml files, git repositories, or container images

4. `References and Artifacts`: Links to CLI downloads, versioned documentation, and guided quickstart tutorials

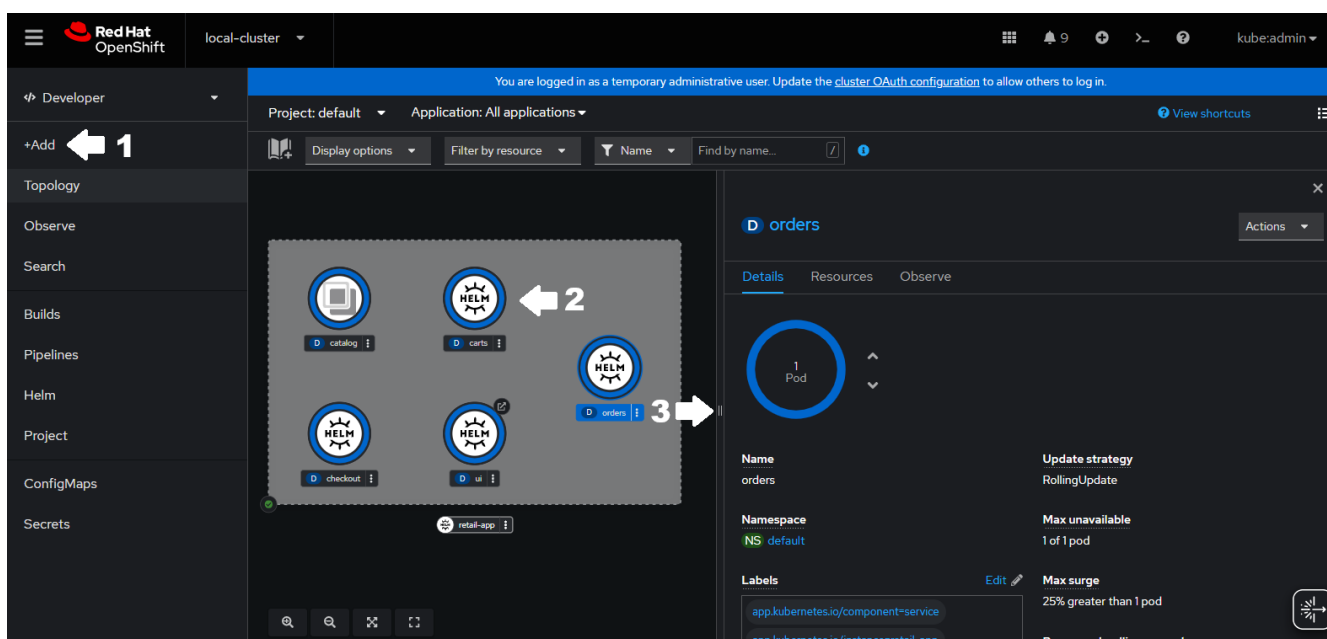5. `User Configuration`: Manage the current user context and generate new time bound cli credentials

Figure 3. Developer Console Basics

1. `Quick Add`: Add new applications from a variety of sources (containers, registries, templates, helm, object storage, etc)

2. `Topology View`: View the relationship of applications within a project

3. `Application Panel`: High level view of resource information including replicas, configuration, services, and routes

Command Line Client

Accessing resources from the cli is simpler, but less pleasant on the eyes. Locating and exploring resources will mostly involve one of the following two commands:

Get (view in native api format)

```
oc get "RESOURCE_TYPE" "RESOURCE_NAME" (-l|-o|-n|-A|...)
# -l: filter by labels
# -o: results format
# -n: scoped to a namespace
# -A: include all namespaces
```

Describe (view in human-friendly format)

```
oc describe "RESOURCE_TYPE" "RESOURCE_NAME"
```

Building Resources

Web Console

The console can be used to simplify the process of constructing resources for OpenShift.

As a gentle introduction, OpenShift provides `forms` for several resource types.

Figure 4. Form View

1. `Populated Fields`: Reduce the available field options to valid resources

2. `Freeform Field`: Allow for generic string input (either required or optional)

3. `Increment with Units`: Scalable numeric input with orders of magnitude

4. `Toggles/Checkboxes`: Hide irrelevant config behind enabled/disabled markers

5. `Radio Selection`: Select from a predefined static set

For more in depth customization there are several tools built into embedded resource editor

Figure 5. Editor View

1. `Editor`: Modify yaml with schema assisted tab complete

2. `YAML`: View and edit the stored configuration for this resource. (Available for all resources)

3. `Schema Sidebar`: Explore the resource schema to identify available keys, values, and their associated types

Command Line Client

Unfortunately the cli does not provide as many user experience benefits. Creating and modifying resources is nearly identical to upstream Kubernetes in these are the most relevant commands:

Create / Apply (submit files previously created/modified in your IDE of choice)

```
oc create/apply -f {$FILENAME}
```

Edit (modify existing cluster resources using the default host IDE)

```
oc edit "RESOURCE_TYPE" "RESOURCE_NAME"
```

Patch (modify existing cluster resources, but without opening an IDE)

```
oc patch "RESOURCE_TYPE" "RESOURCE_NAME" (-p|--patch-file|--type|...)
# -p: apply an inline patch definition
# --patch-file: apply a patch defined in a file
# --type: leverage json, merge, or strategic patch strategies
```

Contexts and Clusters

Our final concern involves operations across multiple tenants and multiple clusters. Any given OpenShift user could manage several applications across several projects that are deployed across several clusters.

Web Console

In this scenario, the console does not provide much additional assistance. Your options are limited to:

- logging in/out to switch users

- opening a browser or browser tab for each cluster's independent console

- leveraging the "Advanced Cluster Management" console plugin to manipulate resources across multiple clusters at once

> `Advanced Cluster Management (ACM)` is an advanced topic that is beyond the scope of this workshop. There is an excellent workshop here for those interested in learning about ACM.

Command Line Client

Unlike the console, the cli provides several user mechanisms to quickly switch between projects

and clusters. In order to understand how the CLI provides this functionality, let's breakdown the terms `Context` and `Cluster`.

A `Context` *is a combination of three things:*

- User: Can be a service account or a user provided by an external identity provider

- Namespace/Project: Interchangeable in this context

- Cluster: An api endpoint in the form 'https://api."CLUSTER"."DOMAIN":6443'

You can see two example contexts in the following KUBECONFIG:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: # Redacted
    server: https://api.crc.testing:6443
  name: api-crc-testing:6443
contexts:
#### CONTEXT 1 #########################
- context:
    cluster: api-crc-testing:6443
    namespace: default
    user: kubeadmin/api-crc-testing:6443
  name: crc-admin
######################################
#### CONTEXT 2 #########################
- context:
    cluster: api-crc-testing:6443
    user: developer/api-crc-testing:6443
  name: crc-developer
######################################
current-context: crc-admin
kind: Config
preferences: {}
users:
- name: developer/api-crc-testing:6443
  user:
    token: sha256~6FxIo-2otdT1-GWq_3XeRxbYtmeN5LJRFeMILQkYfAs
- name: kubeadmin/api-crc-testing:6443
  user:
    token: sha256~LrlOU2OTEVZI7ln7Rh6ZrAvYXCpn8aowmSBmHmZAK_8
```

A user could manually modify this file to switch between any number of contexts, but there are much quicker ways provided by the cli.

Get [Users,Contexts,Clusters]

```
oc config get-users ...
oc config get-contexts ...
oc config get-clusters ...
```

Set [Credentials,Contexts,Clusters]

```
oc config set-credentials ...
oc config set-contexts ...
oc config set-clusters ...
```

Use [Context] (Changes "current-context" to "CONTEXT")

```
oc config use-context "CONTEXT"
```

> ℹ️ There is an asymmetry between the "get" and "set" methods for users.
> This design implies that the KUBECONFIG, and by extension the cluster, doesn't natively handle users.

References

- Web Console
- Command Line Client
- JSON Patch Specification

Knowledge Check

How many separate methods exist in the Developer view's "+Add" panel?

How many projects are in this cluster?

How many projects are "Default Projects"?

What are valid values for a Deployment's "spec.strategy.type" field?

What command line interfaces are available besides `oc` from the local cluster?

Can you switch between two projects/contexts/clusters?

Projects

Before we start deploying workloads, there is one last major concern to be addressed: How to be a responsible tenant in an OpenShift environment? The 'Tragedy of the Commons' shows us that without protections OpenShift resources would be depleted and the platform would likely become unusable.

Although the specifics of how to govern tenants can vary from engineer to engineer or from organization to organization, there several standard tools and practices used to maintain a hospitable multi-tenant OpenShift environment.

This section introduces those tools:
- `Service Accounts` - Provide identities and attribution

- `Role-Based Access Control (RBAC)` - Provide permissions boundaries

- `ResourceQuotas` - Provide namespace level maximums

- `LimitRanges` - Provide resource level maximums

Service Accounts

Service accounts are special user identities automatically created for workloads running in your project. They allow pods to securely interact with the OpenShift API and other services. If you are running a workload on OpenShift, then you are already using at least one service account.

These are the default Service Accounts:

- `default`: Used by pods without a specified service account.

- `builder`: Used by build configurations.

- `deployer`: Used by deployment configurations.

Don't believe me? You can view them with:

```
oc get serviceaccounts
```

Service accounts work in combination with users to provide authentication to the platform. Whereas user authentication comes from an identity provider that is not part of kubernetes, the identity of a service account is provided entirely by the platform itself. The credentials that a service account uses to prove its identity are JSON Web Tokens(JWT).

OpenShift will automatically create and mount these JWTs when a service account is attached to a workload, but you can get a token in a few other ways:

From the command line

```
oc create token "SERVICE_ACCOUNT_NAME" -n "NAMESPACE"
```

From a secret (Linux)

```
cat <<EOF | oc apply -f -
apiVersion: v1
kind: Secret
type: kubernetes.io/service-account-token
metadata:
  name: "SECRET_NAME"
  namespace: default
  annotations:
    kubernetes.io/service-account.name: "SERVICE_ACCOUNT_NAME"
EOF
oc extract secret/"SECRET_NAME" --to=- --keys=token
```

From a secret (Windows)

```
echo
{"apiVersion":"v1","kind":"Secret","metadata":{"name":"SECRET_NAME","annotations":{"ku
bernetes.io/service-
account.name":"SERVICE_ACCOUNT_NAME"}},"type":"kubernetes.io/service-account-token"} |
oc apply -f -
oc extract secret/"SECRET_NAME" --to=- --keys=token
```

If you are familiar with JSON Web Tokens, you'll know that there is a human readable format behind the encoded one. An example token for the `default` service account in the `default` namespace might look like this:

```
{
  // audience
  "aud": [
    "https://kubernetes.default.svc"
  ],
  // expire date
  "exp": 1744690648,
  // issue date
  "iat": 1744687048,
  // issuer
  "iss": "https://kubernetes.default.svc",
  // unique identifier
  "jti": "52c10b4a-1526-4197-9601-3021852011fd",
  // kubernetes identifiers
  "kubernetes.io": {
    "namespace": "default",
    "serviceaccount": {
      "name": "default",
      "uid": "f448c70f-ef06-409a-a782-8d71e4943979"
    }
  },
  // start date
  "nbf": 1744687048,
  // service account identifier (string form)
  "sub": "system:serviceaccount:default:default"
}
```

And finally, if you ever need to get the token for your current user, you can run the following:

```
oc whoami -t
```

Role-Based Access Control (RBAC)

With human and machine identities established, access control can be applied. The standard

mechanism to accomplish this in OpenShift is `Role Based Access Control` or `RBAC`. Effectively, RBAC is when an identity assumes a role which has been given access to perform some set of actions on a given set of resources and inherits all of the policy enforced on the role. This "assumption" and the "given access" are defined in `RoleBinding`, `ClusterRoleBinding`, `Roles`, and `ClusterRoles` resources.

> For this workshop, `ClusterRoles` and `ClusterRoleBindings` are nothing more than the cluster scoped versions of `Roles` and `Rolebindings`. A `ClusterRole` can give the same permissions cluster wide, and a `ClusterRoleBinding` can be associated with any identity cluster-wide. ClusterRoles do have a slight difference in implementation that we will ignore for now.

To create a role based access strategy, you'll need to be able to determine three things: . Which actions do I need from all available actions . Which API resources do I need from all available resources . Which API groups do the resources I need belong to

Role Actions

Thankfully this list is static, and directly correlates with HTTP verbs:

HTTP Verb	Request Verb
POST	create
GET,HEAD	get, list, watch
PUT	update
PATCH	patch
DELETE	delete, deletecollection

Role Resources and Groups

Unfortunately the list of resources and groups is not as short and not entirely static. OpenShift APIs are always improving and new resources are being added each release.

Obtaining the `GROUP` and `RESOURCE` from a running cluster can be done by:

"Explaining the resource"

```
oc explain "RESOURCE"
```

"Getting the entire list of API resources"

```
oc api-resources
```

Creating Roles and Rolebindings

Having verbs, resources, and groups, we can now start making roles and rolebindings.

Create a role

```
oc create role "ROLE_NAME" \
  --verbs "VERB,VERB,VERB" \
  --resource "RESOURCE"."GROUP"
```

Bind the role to a user or service account

```
oc create rolebinding "ROLEBINDING_NAME" \
  --role "ROLE_NAME" \
  (--user "USER" | --serviceaccount "SERVICE_ACCOUNT")
```

> 💡 You can "bind" as many ROLES as you want to an identity, so don't be afraid to create multiple roles that are more human readable or map to business logic.

ResourceQuotas and LimitRanges

The last level of platform protection is targeting resource exhaustion directly. Even though OpenShift can scale to incredible sizes, their may be other constraints limiting the total amount of hardware that can be afforded to the platform (budget, logistics, etc). ResourceQuotas and LimitRanges work in tandem to guarantee that the maximum number of resources used with a given project is not exceeded, and the consumption of those resources is spread evenly among workloads.

ResourceQuotas

Here is a sample ResourceQuota

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: example
  namespace: default
spec:
  hard:
    pods: '4'
    requests.cpu: '1'
    requests.memory: 1Gi
    limits.cpu: '2'
    limits.memory: 2Gi
```

This would mean that the total number of pods in the "default" namespace can not exceed 4, and that a total of 2 cores and 2 Gibibytes is all the system resources that these pods can make use of.

LimitRanges

LimitRanges control how "smooth" resource consumption is. Given the previous ResourceQuota, a

project could still have resources anywhere between one pod using all resources (leaving no room for other workloads) and all four pods evenly sharing the resources.

If our ultimate goal was to have an even distribution of resources, we would pair the previous `ResourceQuota` with a `LimitRange` similar to this

```
apiVersion: v1
kind: LimitRange
metadata:
  name: example
  namespace: default
spec:
  limits:
    - min:
        cpu: .250
        memory: 256Mi
      max:
        cpu: .750
        memory: 768Mi
      type: Pod
```

This would force all pods in this namespace to fall between .250 by 256Mi and .750 by 768Mi. This would also prevent any single pod from consuming all of the available resources.

References

- Service Accounts
- JSON Debugger
- Using RBAC
- ResourceQuotas and LimitRanges

Knowledge Check

Where are service account tokens mounted in a running workload?

▼ *Hint*

Since all workloads leverage a service account, you can find the token using your linux filesystem skills.
`oc exec -it "POD_NAME" — sh` will get you into a pod's context.

How long do service account tokens last?

▼ *Hint*

You can decode one of your own JWTs or identify the difference in the example above.
Once you know ($exp - iat$), you can convert to the correct unit of time.

If a given role is duplicated across several namespaces, how can you reduce the number of

roles that need to be managed?

▼ *Hint*

Do you remember the "NOTE" about `ClusterRoles` and `ClusterRoleBindings`?
Well they can be mixed with `Roles` and `RoleBindings`!
To remove duplicate `Roles` you simply have to make a `ClusterRole` and change the references in any `RoleBindings`

ResourceQuotas and LimitRanges only work when you create them, how would you guarantee that they are created when a project is created?

▼ *Hint*

The solution was introduced in the project section of "OpenShift vs Kubernetes".

CPU and Memory resources are measured with specific units in OpenShift, what are they?

▼ *Answer*

`CPU` can be measured with "fractional count" (i.e. 1.0, 2.5, 1.001), or by "millicpu/millicores". `Memory` can be measured with base ten units (kilobyte, megabyte, gigabyte…) or base two units (kibibyte, mebibyte, gibibyte…)

CPU and Memory resources are measured with specific units in OpenShift, what are they?

▼ *Answer*

`CPU` can be measured with "fractional count" (i.e. 1.0, 2.5, 1.001), or by "millicpu/millicores". `Memory` can be measured with base ten units (kilobyte, megabyte, gigabyte…) or base two units (kibibyte, mebibyte, gibibyte…)

Application Deployment

Now that we've established some "ground rules" on "fair behavior" in the platform we can start leveraging OpenShift as a platform for applications. This module will cover everything needed to take source code, build it, package it, and deploy it on OpenShift.

> The source code referenced in this page is available in the top-right "Links" section.
> It's also available here.

Container Basics

Images are where our application journey begins. Without them, and the standards built around them, Kubernetes and OpenShift would be much different technologies if they even existed at all. In simplest terms, an image is a repeatable recipe for software wrapped into a well defined binary format. The format is has proven to be particularly flexible. Today there are images for several architectures (`arm`,`x86`,`mips`,`power8`,`mainframe`,…), several operating system families (`Windows` and `Linux`), and several operating systems (`rhel`,`ubuntu`,`suse`,…).

Let's dig in to an example Dockerfile:

```
FROM python:3.12
WORKDIR /usr/local/app

# Install the application dependencies
COPY requirements.txt ./
RUN pip install --no-cache-dir -r requirements.txt

# Copy in the source code
COPY src ./src
EXPOSE 5000

# Setup an app user so the container doesn't run as the root user
RUN useradd app
USER app

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8080"]
```

Each (non-empty) line begins with a token in all caps that signifies a specific instruction when building:

- `FROM`: Are we starting from `SCRATCH` or are we building **FROM** another image?

- `WORKDIR`: In our new image, which **WORK**ing **DIR**ectory will be the context for further actions?

- `COPY`: What files do we need to **COPY** into our new images new working directory?

- `RUN`: What do we need to **RUN** in our image to achieve the correct configuration?

- `EXPOSE`: What ports are being **EXPOSE**d by our process?

- `USER`: What **USER** are we running the process as?

- `CMD`: What **C**o**M**man**D** are we running when this image is instantiated?

With the exception of `FROM` (beginning of file) and `CMD|ENTRYPOINT` (end of file), the specification allows near complete freedom on the order and amount of instructions. There are best practices however. Apart from successfully executing an app, readability of the container file, the size of your container file, and the size of the resulting image should be highly prioritized.

💡 Starting off `FROM` the right image can help keeps images small and secure.
Red Hat provides a family of based images called "Universal Base Images" to provide a solid foundation for your applications.

For the most part the use cases for these instructions are self-evident, but there are several that overlap and often cause confusion.

Table 1. Instruction Comparisons

Instructions	Explanation
CMD vs RUN	Both run commands, but at runtime and on build respectively

Instructions	Explanation
CMD vs ENTRYPOINT	Both the runtime process, but CMD is overridable while ENTRYPOINT is not
ADD vs COPY	Both add files, but ADD should only be used for remote/packaged files
ARG vs ENV	Both control parameters, but at runtime and on build respectively

And with all that behind us, we can start building images and creating containers.

Grab your client of choice podman or docker (links below) and try the following commands:

▼ *Podman Image Commands*

Build,tag, and label the image

```
# From the source repository root
podman build --file src/cart/Dockerfile --tag cart:0.0.1 --label build-0.0.1
```

Find the built image

```
# From anywhere
podman images --filter label=build-0.0.1
```

Inspect the image

```
# From anywhere
podman inspect localhost/cart:0.0.1
```

Create a container from the image

```
# From anywhere
podman run --name cart-0.0.1 -d -p 8080:8080 localhost/cart:0.0.1
```

Test the image

```
# From anywhere
curl -I localhost:8080/carts/1
```

Cleanup

```
# From anywhere
podman kill cart-0.0.1
podman rmi localhost/cart:0.0.1
```

Builds

Being able to build locally is a powerful asset, but what if you do not have the correct architecture or a powerful enough machine to handle builds on your own? There are several options provided either out of the box or as an addon.

Native Builds

OpenShift can build natively with the same `Dockerfile` strategy. Assuming that the architecture of your own build environment is the same as your OpenShift nodes, we can effectively build the same artifact the same way as we did previously.

Build Once

```
oc create build cart --strategy Docker --source-git=https://github.com/shpwrck/retail
-store-sample-app --context-dir=./src/cart --dockerfile-path=. --to-image-stream
cart:0.0.1
```

If you run the same command again, it will fail (by design). We need a generative way to produce builds repeatedly. That is where `BuildConfigs` come in.

▼ *OC Image Commands*

Build Repeatably

```
oc new-build https://github.com/shpwrck/retail-store-sample-app --strategy docker
--context-dir=./src/cart --to cart:0.0.1
```

Trigger Build

```
oc start-build retail-store-sample-app
```

Confirm Builds

```
oc get builds
oc get is cart -o yaml
```

> ℹ️ We manually created the builds with the `oc start-build` … command, but there's another option. We can trigger builds when webhooks or code changes are received.
> Look here for more

Builds with Shipwright

`Builds` and `BuildConfigs` have been around for a long time, and as such have less functionality than more modern approaches. Since their creation, projects such as `BuildKit`,`Buildpacks`, `Buildah`, and `Kaniko` have added interesting and useful new spins to the domain of cluster based builds. And

while it's out of scope for this workshop, OpenShift can host a tool that takes the benefits of each technology and wraps them into a singular build platform. That tool is Shipwright or Builds for Red Hat OpenShift, and it is quickly becoming the common standard.

Workload Basics

The next step after accomplishing reliable/repeatable builds is deployment. We can generate the necessary images and we have a full understanding of how they operate. Transitioning to OpenShift from our local environment is a simple undertaking. Sometimes as easy as `oc new-app`!

Create a new build and a running 'Deployment'

```
oc new-app https://github.com/shpwrck/retail-store-sample-app --name test-cart
--context-dir ./src/cart
```

or

Create just a running 'Deployment'

```
# This command borrows the image from our previous builds
oc create deployment test-cart-deployment --image image-registry.openshift-image-
registry.svc:5000/default/test-cart:latest
```

These two commands do greatly oversimplify the capabilities that OpenShift has in regards to running workloads.

Take for instance:

- The many types of workloads `Deployments`, `StatefulSets`, `Daemonsets`, `Jobs`, `CronJobs`
- The `1,000+` keys and values that each resource has
- The `100+` Red Hat operator-based solutions that augment workload behaviors

It's important to approach this domain incrementally. So for now we will leave workloads behind and focus on one final application tool.

Helm

Helm is the last remaining tool in our Application's MVP. Helm allows us to take what we've done up to this point. (Images, Builds, Deployments,…) and package it up just like we packaged our application itself. This affords us the same benefits that containerization did: reliable and repeatable results.

The process for taking what we have and packaging it with helm is more involved than our previous commands. But it can still be done in less than a few lines of execution.

▼ *Basic Helm Steps*

Initialize a helm chart

```
# Create the file structure for helm
```

```
helm create cart
```

Remove Default Templates

```
cd cart
rm -r templates/*
```

Push the generate resources into the "templates" directory

```
# Add "-o yaml" to our previous "new-app" command and dump it to a file
oc new-app https://github.com/shpwrck/retail-store-sample-app --name test-cart
--context-dir ./src/cart -o yaml > templates/resources.yaml
```

Install Helm Chart

```
# From cart/
helm install helm-cart .
```

Start Build

```
oc start-build test-cart
```

And voila! You can take that example and run it in any OpenShift Cluster! There are plenty more things to say about `helm` though.

First and foremost, our MVP would ideally have the following improvements done upon it in the real world:

- Split the resources out of the `List` type, and into individual resources

- Add some parameters to allow for dynamic naming, scaling, or other resources

- Provide RBAC, Resource Limits, and Resource Requests

- Provide some additional documentation as to how the application should operate.

- Recreate meaningful tests

This list could honestly go on for several dozen more bullet points, but this is just an MVP (not **PRODUCTION**). After we've completed both "Day 1" and "Day 2", I highly recommend returning to helm, workloads, and builds to explore what was originally out of scope.

References

- [Dockerfile Reference](#)
- [Docker](#)
- [Podman](#)
- [Buildah](#)
- [Helm](#)

- Building Applications (OpenShift)

- Builds with BuildConfig

- Builds with Shipwright

Knowledge Check

Digging around the source code repository, you may see Dockerfiles with multiple `FROM` instructions, what does this imply?

▼ *Answer*

Multi-stage Builds!

Just as you can chain together `FROM` instructions from file to file, you can chain them inside a single file. This improves readability, consolidates the process of a full build, and helps keep image sizes small.

If you `COPY` a directory, but you'd like to skip specific files, what recourse do you have?

▼ *Answer*

Similar to a "gitignore", `podman` and `docker` both support a "dockerignore" By specifying directories,files, and file-globs in this way, you can keep sensitive files out of images and reduce the overall size of the image.

What images are included in the UBI catalog?

▼ *Answer*

Each RHEL Operating System (7,8,9) have "ubi","ubi-init","ubi-micro", and "ubi-minimal".

Helm is supported both from the CLI and the Web Console, can you create a helm chart from the Console?

▼ *Answer*

Networking

OpenShift provides a powerful networking stack to expose your applications running inside the cluster to external users. This section explains how traffic flows into the cluster, how to control it with routes, and how to secure and configure those routes.

Network Path

When a client accesses an OpenShift-hosted application, the traffic follows this general path:
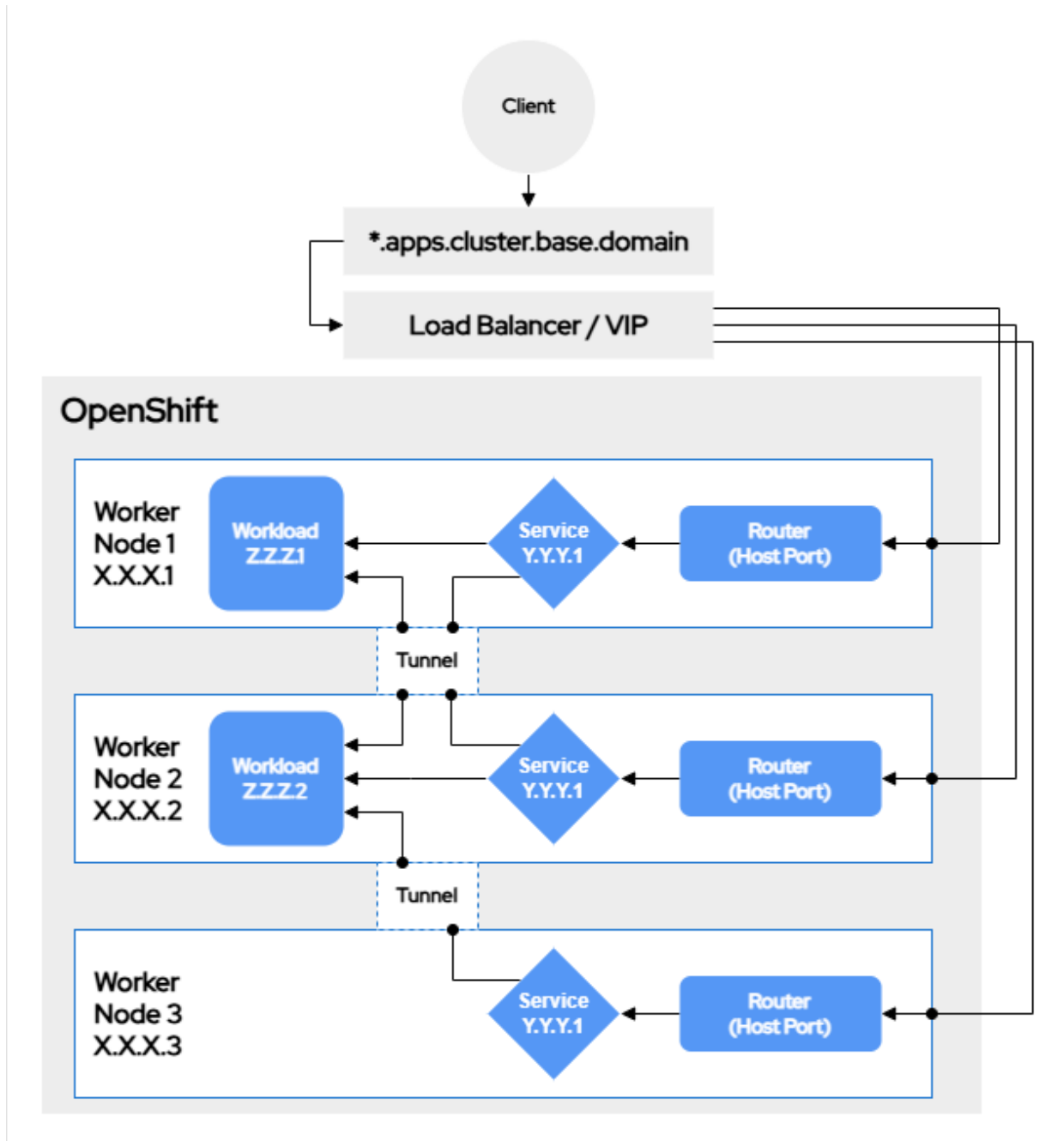

Figure 6. Diagram: Network Path into OpenShift

1. Client sends a request to `*.apps.cluster.base.domain`

2. The is resolved to either a Load Balancer or a Virtual IP (VIP)

3. The traffic is sent to a node based on the load balancing algorithm

4. The node receives the traffic and translates it to the router's network namespace

5. The router forwards the traffic to the correct node given the service state and definition

This is a very **high-level** view. Each bullet point could in fact be expanded into several additional bullet points with a much longer list of relevant technologies. Fortunately for developers, most networking is transparent to developer operations and is more likely to be managed by platform administrators. Developer concerns are normally limited to `Services` and `Routes` definitions.

Services

Services provide a single fully qualified domain name (FQDN) and a single IP on which a set of workloads can all be addressed. This design makes accessing services predictable and reliable.

The FQDN that is associated with any service will follow a common format:
`SERVICE_NAME.SERVICE_NAMESPACE.cluster.local`

> ℹ️ Pods also follow a standard naming convention, but it is less commonly leveraged.
> `POD_IP.POD_NAMESPACE.pod.cluster.local`
> - where the (`.`) in the `POD_IP` are replaced with (`-`)

The single IP address that is associated with a service is referred to as a `ClusterIP`. You'll see that defined on every service.

```
oc get svc kubernetes -o yaml
```

Running this command will reveal several references to `ClusterIP` in fact (`clusterIP`,`clusterIPs`, and `type: ClusterIP`). The first two are simply storing the IP address (`clusterIPs handles ipv4 and ipv6`), but the last `type: ClusterIP` is a reference to something more substantial. The `type` field controls how the service itself is implemented on the network and it can come in any of the following forms:

- `ExternalName`: the service is only a "pointer" record without deeper networking implementation

- `ClusterIP`: limits access to the service to internal traffic

- `NodePort`: the service is available to internal traffic, but also on a high numbered port on the network the hosts belong to

- `LoadBalancer`: does everything a `NodePort` service does, but also facilitates the connection to a hardware or software based load balancer

> ℹ️ The `NodePort` range is between 30000-32767.

In many situations (particularly in cloud based environments) these four are sufficient to expose workloads internally and externally without much additional work. With "on premise" installs however, there is no cloud controller available and thus a different approach is necessary.

Routes

`Routes` are the "batteries included" solution OpenShift provides that allows for more sophisticated

external network exposure. OpenShift routes are effectively a specific implementation of the upstream Kubernetes Ingress. The underlying technology that makes this entire routing solution work is HAProxy.

Notice the output here:

```
oc exec deployment/router-default -n openshift-ingress -- sh -c "ps -ef"
```

Path-Based Routing

You can route multiple applications or services under a single hostname using **path-based routing**. For example:

- https://example.com/api → backend
- https://example.com/web → frontend

This is achieved by setting the `path` on a route:

```yaml
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: frontend
spec:
  host: example.com
  path: /web
  to:
    kind: Service
    name: frontend-svc
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: backend
spec:
  host: example.com
  path: /api
  to:
    kind: Service
    name: backend-svc
...
```

Weighted Routing

You can do the reverse as well! By putting several backends behind a single path you enable more complex deployment strategies. Here's an example of **weighted-routing**

```
---
```

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: two-service-route
spec:
  host: two-service.route
  to:
    kind: Service
    name: service-a
    weight: 10
  alternateBackends:
  - kind: Service
    name: service-b
    weight: 10
...
```

> **ℹ** The weights in **weighted-routing** are not percentages.
> They are defined as "an integer between 0 and 256".

Security (TLS Termination)

One of the most important features `Routes` provide is network security with **TLS termination** strategies. Instead of relying solely on the distribution of TLS certificates within every application, a platform can consolidate all TLS concerns into the router itself.

The three included strategies are:

- `edge` – TLS is terminated at the router; traffic to pods is HTTP.

- `passthrough` – TLS traffic is sent directly to the pod.

- `reencrypt` – TLS is terminated at the router, then re-encrypted and sent to the pod.

```
...
spec:
  tls:
    termination: ("edge"|"passthrough"|"reencrypt")
    insecureEdgeTerminationPolicy: Redirect
...
```

You provide certificates directly in the route with plaintext, or let OpenShift manage them via the more advanced cert-manager project.

```
...
spec:
  tls:
    caCertificate: ...inline cacert...
    certificate: ...inline cert...
```

Configuration via Annotations

There are a number of "advanced options" that `Routes` provide through another mechanism, annotations. The features provided by this method involve concurrency, rate-limiting, timeouts, and more. Listed below are the more commonly used options:

- "haproxy.router.openshift.io/balance":
- "haproxy.router.openshift.io/disable_cookies":
- "haproxy.router.openshift.io/hsts_header":
- "haproxy.router.openshift.io/ip_allowlist":
- "haproxy.router.openshift.io/ip_whitelist":
- "haproxy.router.openshift.io/pod-concurrent-connections":
- "haproxy.router.openshift.io/rate-limit-connections":
- "haproxy.router.openshift.io/rewrite-target":
- "haproxy.router.openshift.io/timeout":

> For a full list of annotations check out "HAProxy Annotations" in the reference list.

You can apply annotations via `oc annotate` or include them in the route YAML:

```
...
metadata:
  annotations:
    haproxy.router.openshift.io/timeout: 10s
    haproxy.router.openshift.io/rate-limit-connections: "5"
...
```

An easy way to add this functionality is to run

```
# To add an annotation
oc annotate route -n "NAMESPACE" "ANNOTATION_KEY"="ANNOTATION_VALUE"
# To remove an annotation
oc annotate route -n "NAMESPACE" "ANNOTATION_KEY"-
```

Ingress Compatibility

`Routes` are an ingress implementation and 100% compatible with standard Kubernetes. You could deploy all of the necessary components in any Kubernetes cluster using the github source. Despite this however, `Routes` are not common outside of OpenShift configuration. The most common solution for Kubernetes networking is Ingress.

The routing implementation has a solution to this discrepancy. If you have not set the `ingressClassName` field to `openshift-default`, fear not! Ingresses will still be adopted by the routing controller if the `host` field matches a domain currently being monitored by the router. (`*.apps."CLUSTER_NAME"."BASE_DOMAIN"`)

> **ℹ** **Additional annotations for `Route` definition are available with this method**
>
> - route.openshift.io/termination: "reencrypt"
>
> - route.openshift.io/destination-ca-certificate-secret: "..."

References

- [Kubernetes Services](#)

- [HAProxy Annotations](#)

- [Routes](#)

- [Ingress Sharding (Advanced)](#)

Knowledge Check

The ClusterIP field of a `Service` is a "string" field with valid values beyond IP addresses. Can you find out what they are, and when they might be used?

▼ *Answer*

The two valid values are:

- "None"

- "" This will create what is called a ["Headless Service"](#).
  These services are helpful when a connection needs to be made to one or many specific pods.

Can you create a route from an `oc` cli command?

▼ *Answer*

Yes! Two separate ways in fact:

- Insecure Routes: `oc expose service "SERVICE_NAME"`

- Secure Routes: `oc create route (edge|passthrough|reencrypt) --service "SERVICE_NAME"` ⋯

What happens when you delete a `Route` that has been created from an `Ingress` resource?

▼ *Answer*

As with most other resources that have been created with a controller, it will be recreated.
The `Ingress` resource sets a declarative configuration that the ingress router controller continually attempts to resolve.

Day Two Agenda

- [Container Lifecycle](#)

- [Managing Configuration](#)

- [Observability](#)

- [Scaling Applications](#)

- [Debugging Applications](#)

- [Deployment Strategies](#)

Outcomes

By the end of day two, participants will have expanded their operational understanding of OpenShift and gained experience with advanced platform capabilities. The following learning outcomes are expected:

- **Manage Image Artifacts Inside and Outside of a Cluster**:
  Learn how to work with container images using internal OpenShift image registries as well as external registries, and understand how image policies impact builds and deployments.

- **Configure Applications with Secrets and ConfigMaps**:
  Use Kubernetes-native resources to securely manage application configuration, environment variables, and sensitive data without hardcoding values.

- **Track Metrics and Logging of User Workloads**:
  Explore observability tools available in OpenShift to view pod-level logs, inspect metrics, and monitor application health using the built-in dashboard and command-line utilities.

- **Scale Applications with Horizontal Pod Autoscalers**:
  Configure and observe autoscaling behavior based on CPU utilization or custom metrics to meet changing demand automatically.

- **Debug Applications with Built-In Tools**:
  Leverage OpenShift's built-in debugging capabilities—such as remote shell access, live container inspection, and diagnostic events—to troubleshoot issues effectively.

- **Understand the Different Workload Strategies for Applications**:
  Learn how OpenShift helps manage modern deployment strategies like Canary Rollouts, Blue/Green Deployments, and A/B Testing.

[Jump to Day One](#)

Container Lifecycle

Managing container images is often an overlooked part of application development and operations. However, understanding the container image lifecycle in OpenShift can lead to significant benefits. Efficient image management can:

- Speed up cold starts on new or rebooted nodes by reducing image pull times.

- Prevent storage exhaustion by pruning unused or outdated images.

- Simplify operations in secure or disconnected environments where external registries may be unavailable.

This page explores how images are brought into OpenShift, how they are managed on the cluster, and how the platform automatically prunes old images to maintain performance and stability.

To supplement the standard image lifecycle, we will introduce four supporting technologies: `kubernetes-image-puller`, `mirror registry`, `image-pruner`, and the included internal `image registry`.

Importing Images

The simplest way to use an image in OpenShift is to reference it directly in a workload manifest. This can be done with any workload type, such as a `Deployment`, `DaemonSet`, or `StatefulSet`.

In this case, the image is typically hosted on an external registry like Quay or Docker Hub, or on a registry hosted on-premise within the organization. There is no need to import the image beforehand — OpenShift will attempt to pull it when the pod is scheduled.

But what specifically happens after the manifest is applied?

- OpenShift schedules the pod onto a node.

- The container runtime checks if the image is already present on the node.

- If not present, the runtime attempts to pull the image from the specified registry.

- If the registry requires authentication, the runtime uses credentials stored in the cluster (typically via a `Secret` linked to a `ServiceAccount`).

- Once pulled, the image may be cached locally on the node for future use.

> Image pulling behavior is controlled by the `imagePullPolicy` field in the container spec. It supports three values:
>
> - `Always` – Always pull the image from the registry, even if it already exists on the node.
>
> - `IfNotPresent` – Pull the image only if it does not already exist on the node (this is the default for tags other than `:latest`).
>
> - `Never` – Do not pull the image. Use only what is already available on the node.
>
> Choosing the right policy can help balance reliability, performance, and security depending on your use case.

It's important to note that image caching is **node-specific**. If a pod is rescheduled to a different node — for example, due to a node reboot or scale event — the image must be pulled again on that new node. This can introduce additional latency if the image is large or the registry is slow or unreachable.

To reduce this impact, the `kubernetes-image-puller` project can be used to proactively pre-fetch commonly used or frequently updated images across all nodes in a cluster.

▼ *ImagePuller*

Install the operator with

```
---
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  labels:
    operators.coreos.com/kubernetes-imagepuller-operator.openshift-operators: ""
  name: kubernetes-imagepuller-operator
  namespace: openshift-operators
spec:
  channel: stable
  installPlanApproval: Automatic
  name: kubernetes-imagepuller-operator
  source: community-operators
  sourceNamespace: openshift-marketplace
  startingCSV: kubernetes-imagepuller-operator.v1.1.0
```

Create an image-puller resource

```
---
kind: KubernetesImagePuller
apiVersion: che.eclipse.org/v1alpha1
metadata:
  name: image-puller
  namespace: openshift-operators
spec:
  daemonsetName: k8s-image-puller
  images: 'ubi9=registry.redhat.io/ubi9:latest'
```

> ❗ The example image-puller resource uses the "latest" tag. Avoid using the latest tag wherever possible, because it does not necessarily provide a consistent image.

Disconnected Installs

In some environments — like high-security data centers or government systems — clusters aren't allowed to reach out to the internet. That means they can't pull images directly from external registries.

So how do you run workloads in a setup like that?

The answer is to mirror the images you need into a local registry that the cluster **can** access. This process usually happens ahead of time, in a connected environment. Once the images are downloaded, they're pushed into a registry that lives inside your network.

Here's the general approach:

- Pull the images you need while connected.

- Push them into a registry that's reachable by the disconnected cluster.

- Update the cluster configuration so that it uses the internal registry instead of reaching out to the public internet.

OpenShift provides tools like `oc mirror` and `oc adm release mirror` to help automate this process, especially for cluster upgrade images and curated image sets.

> In disconnected clusters, that local registry becomes critical. If it's misconfigured or goes down, pods won't start, and cluster upgrades can break. Make sure it's reliable and backed up.

And if you want to go one step further, cluster administrators can use a combination of `ImageContentSourcePolicy` (ICSP), `ImageDigestMirrorSet`, and `ImageTagMirrorSet` objects. These let you rewrite image references at the cluster level — so even core OpenShift components pull from your mirrored registries instead of the internet.

Images on the Cluster

Once an image is pulled, OpenShift doesn't just forget about it. The platform tracks images using its internal registry, metadata, and built-in Kubernetes resources. This gives OpenShift powerful visibility and automation around how images are stored, updated, and referenced across the cluster.

In this section, we'll explore how OpenShift manages images that have already been brought into the cluster, including how it stores them, associates them with workloads, and keeps them up to date.

ImageStreams

In Kubernetes, you typically reference container images by their full registry URL and tag. But in OpenShift, you have the option to use something more flexible: an `ImageStream`. You have already created one in the "Application Deployment" module!

An ImageStream is a resource that acts like a pointer to one or more related images. It lets you manage versions, automate updates, and even trigger builds or deployments when a new image is available.

You can think of an ImageStream as a named alias for a group of images — similar to how a Git branch points to different commits over time.

In OpenShift you can use standard image references or ImageStreams. So why use ImageStreams?

- They decouple your application from the raw image URL.

- They enable automatic redeployments or builds when a new image is pushed.

- They give you visibility into image history, metadata, and source.

A basic ImageStream might look like this:

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: retail-cart
  namespace: default
spec:
  lookupPolicy:
    local: false
status:
  dockerImageRepository: image-registry.openshift-image-
registry.svc:5000/default/retail-cart
  publicDockerImageRepository: default-route-openshift-image-
registry.apps.ms01.k8socp.com/default/retail-cart
  tags:
  - items:
    - created: "2025-04-09T15:43:06Z"
      dockerImageReference: image-registry.openshift-image-
registry.svc:5000/default/retail-
cart@sha256:f14cc636d6f62738da50597f32829920b32495a1f714bea431281bba6f4b37bf
      generation: 1
      image: sha256:f14cc636d6f62738da50597f32829920b32495a1f714bea431281bba6f4b37bf
    tag: latest
```

In this example, the ImageStream is called `retail-cart`, and its `latest` tag points to an external image. Once this is set up, you can reference it in workloads like this `image: image-registry.openshift-image-registry.svc:5000/my-project/my-app@sha256:⋯`. where the `sha256` comes from the ImageStream's tag status.

Behind the scenes, OpenShift handles tracking and pulling the actual image so that your workload always uses the correct version.

ImageStream Facts

- ImageStreams can track images from many sources: external registries, the internal OpenShift registry, or even from builds running inside the cluster.

- Deployments that have ImageStream triggers will get an annotation like `image.openshift.io/triggers`

- ImageStreams are namespace scoped, but you can reference them from other namespaces as long as the service account has the right permissions

  - `oc policy add-role-to-user system:image-puller system:serviceaccount:test:default --namespace=default` would give the default service account in the test namespace access to the ImageStreams in the default namespace

Image Pruning

Over time, clusters accumulate a lot of images — from builds, deployments, and testing. Without cleanup, this can take up valuable storage in the internal registry and slow down operations.

OpenShift includes a built-in pruning mechanism to help manage this. Pruning removes old, unused, or orphaned images from the internal registry, keeping your storage lean and healthy.

You don't need to manually delete images one by one. The platform provides a way to safely automate cleanup while keeping important images intact.

How Pruning Works

Pruning works by checking for images that:

- Are no longer referenced by any ImageStream or workload.
- Are older than a configured threshold.
- Have more revisions than the retention policy allows.

When you run a prune operation, OpenShift analyzes the registry and removes image data that's no longer needed.

Here's an example command:

```
oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m
```

This tells OpenShift to:

- Keep the most recent 3 revisions of each tag.
- Preserve any images that were pushed in the last 60 minutes.
- Delete anything older that isn't referenced.

> **i** Pruning only affects images stored in the **internal** registry. If you'd like to know how images are cleared from nodes directly, look at garbage collection.

You can also run pruning in dry-run mode to preview what would be deleted:

```
oc adm prune images --keep-tag-revisions=3 --keep-younger-than=60m --confirm=false
```

This is especially useful in production clusters where caution is critical.

OpenShift includes an `ImagePruner` resource that allows you to schedule pruning as part of cluster operations.

This resource lets cluster administrators define pruning policies and run schedules directly within the cluster — no need for external cron jobs or scripts.

It's a good idea to monitor registry usage and prune regularly — especially in environments where image builds and CI/CD activity happen often.

References

- [Images](#)
- [Mirroring](#)
- [Image Puller Operator](#)
- [Image Pruner Operator](#)

Knowledge Check

- What happens when a workload references an external image that isn't cached on the node?
- How are images tracked in OpenShift once they're imported?
- What is an ImageStream, and why might you use one instead of a direct image reference?
- In a disconnected environment, what steps are required to run workloads that depend on public images?
- What does the `oc adm prune images` command do?
- How does OpenShift decide which images to prune from the internal registry?
- What role does the `ImagePruner` resource play in managing image lifecycle automation?
- What are the uses and differences between `ImageContentSourcePolicy`, `ImageDigestMirrorSet`, and `ImageTagMirrorSet`?

Managing Configuration

Applications often need more than just code to run — they also need configuration data. In OpenShift, there are multiple ways to supply this configuration to your workloads in a secure, flexible, and dynamic way.

In this module, we'll explore how you can use:

- `ConfigMaps` to inject environment-specific data.
- `Secrets` to safely store sensitive values like API keys or credentials.
- `Persistent Volumes` to manage state and external configuration files.

These tools give you powerful ways to separate configuration from code — which is key for portability, reuse, and secure operations.

ConfigMaps

ConfigMaps are a way to provide non-sensitive configuration data to your applications. Think of them as key-value pairs you can inject into your workloads — without baking that data into your container image.

You can use a ConfigMap in two main ways:

- **As environment variables** — values from the ConfigMap are exposed as environment

variables inside your container.

- **As mounted files** — the key-value pairs are mounted into the container's filesystem, with each key becoming a filename and each value becoming the file's contents.

Using ConfigMaps as Environment Variables

This is one of the most common approaches. For example, you might create a ConfigMap like this:

```
oc create configmap app-config \
   --from-literal=APP_MODE=production \
   --from-literal=LOG_LEVEL=debug
```

And then reference it in a deployment:

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

Inside your container, `APP_MODE` and `LOG_LEVEL` will be available just like any other environment variables.

But there's a catch: **environment variables are baked into the container at startup**. That means if you update the ConfigMap later, running pods won't see the new values. You'll need to redeploy the pod to pick up the change.

Using ConfigMaps as Files

If you mount the ConfigMap as a volume, each key becomes a file. For example:

```
volumeMounts:
  - name: config-volume
    mountPath: /etc/config

volumes:
  - name: config-volume
    configMap:
      name: app-config
```

Inside the container, you'll get files like `/etc/config/APP_MODE` and `/etc/config/LOG_LEVEL`.

In this case, OpenShift **will** update the files on disk if the ConfigMap changes — but not immediately. These updates depend on the **Kubelet's Sync Frequency**, which by default is set to every minute. If the ConfigMap is updated in the cluster, the Kubelet will eventually detect the change and update the mounted files on the node.

You can find the current sync frequency with the following two commands:

In one terminal

```
# Don't forget to turn this off when you are done!
oc proxy --port 8001
```

In a second terminal

```
curl "http://localhost:8001/api/v1/nodes/'NAME_OF_NODE'/proxy/configz"
```

> 🛈   This field can be modified by an administrator using the kubeletConfig resource.

▼ *Here are three resources to test this behavior*

Original ConfigMap

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: foo
  namespace: default
data:
  foo: bar
binaryData: {}
immutable: false
```

Pod with Environment and File mounts

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: example
  labels:
    app: httpd
  namespace: default
spec:
  volumes:
    - name: configmap
      configMap:
        name: foo
  containers:
    - name: httpd
      image: 'image-registry.openshift-image-
registry.svc:5000/openshift/httpd:latest'
      envFrom:
        - configMapRef:
            name: foo
      ports:
        - containerPort: 8080
```

```
      volumeMounts:
        - mountPath: /mnt/configmap
          name: configmap
```

Updated ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: foo
  namespace: default
data:
  foo: bar-bar
binaryData: {}
immutable: false
```

Despite this automatic sync, it's still up to your application to watch for changes and reload them —
OpenShift won't restart the pod automatically.

> When choosing between environment variables and files, consider how your
> application loads and refreshes configuration. For static config, environment
> variables are fine. For dynamic updates, file mounts give you more flexibility — if
> your app can react to changes.

Secrets

Despite the name, Kubernetes `Secrets` aren't as secret as you might think.

By default, a Secret is just a base64-encoded blob stored in etcd — which means it's not encrypted
unless you explicitly enable encryption at rest. Anyone with the right permissions (or enough
access to the cluster) can decode and read its contents.

That said, Secrets are still a useful tool for storing sensitive data like:

- API keys
- Database credentials
- TLS certificates

Just like ConfigMaps, you can mount Secrets into a container in two main ways:

- **As environment variables** — where each key becomes an environment variable.
- **As files via volume mounts** — where each key is exposed as a file in a specified path.

But remember: just mounting a Secret doesn't make it secure — especially if the pod or container
has elevated access.

Better Secret Management

For production workloads, especially in regulated or security-conscious environments, it's common to **avoid storing long-lived secrets directly in the cluster**.

Instead, you can pull secrets from more secure sources like:

- HashiCorp Vault
- AWS Secrets Manager
- Azure Key Vault
- Google Secret Manager

To bridge these external systems with Kubernetes, there are well-supported tools:

- External Secrets Operator — syncs external secrets into Kubernetes Secrets.
- Secrets Store CSI Driver — mounts secrets directly into pods without ever writing them to etcd.

> ℹ️ Using Kubernetes Secrets is fine for many basic use cases, but it's a best practice to layer on a dedicated secrets management solution — especially if you're dealing with sensitive, rotating, or regulated credentials.

Persistent Volumes

Not all configuration fits neatly into key-value pairs.

Sometimes, your application expects to read full files, entire directories, or even structured data that changes over time. In those cases, `Persistent Volumes` (PVs) can provide a more flexible approach.

A Persistent Volume is essentially a chunk of storage that's provisioned for your workload. It can come from many backends — NFS, cloud block storage, local disks, or more — and it stays around even if the pod using it gets deleted.

Configuration Use Cases

While Persistent Volumes are often associated with stateful apps (like databases), they can also be used for configuration. For example:

- Mounting large or structured config files that don't fit into a ConfigMap.
- Mounting shared configuration across multiple pods or nodes.
- Supplying application plugins, rules, or custom scripts packaged in a volume.

> ℹ️ A single ConfigMap has a maximum size of **1 MiB**. If your configuration files are larger than that, or if you need to include binary data, Persistent Volumes are often a better fit.

Unlike ConfigMaps and Secrets, Persistent Volumes are not managed as Kubernetes-native objects by developers — they require additional coordination with the cluster admin or storage provider.

How Volume Claims Work

As a developer, you don't create Persistent Volumes directly. Instead, you create a `PersistentVolumeClaim` (PVC), which describes how much storage you need and how you plan to use it.

Here's a simple example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: config-storage
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

When this claim is created, OpenShift looks for a `StorageClass` that defines how to provision the volume. A built-in controller called the **CSI provisioner** talks to the underlying storage system — like AWS EBS, Ceph, or a local storage driver — and creates a volume that satisfies the request.

A `StorageClass` acts like a blueprint for dynamic volume provisioning. It tells OpenShift:

- Which storage backend to use (e.g., AWS EBS, Ceph, or local disk).
- What parameters to apply (e.g., volume type, IOPS, encryption).
- Whether volumes should be retained or deleted when the PVC is removed.

Different `StorageClass` objects can offer different performance levels, availability zones, or backup policies — making it easy to match the right storage to the right workload.

The volume is then bound to the PVC and can be mounted into your workload like this:

```
volumeMounts:
  - name: config
    mountPath: /etc/app/config

volumes:
  - name: config
    persistentVolumeClaim:
      claimName: config-storage
```

From the container's perspective, this is just another directory. But under the hood, it's backed by real storage, with durability, capacity, and IOPS defined by the backend.

> Unlike Secrets and ConfigMaps, the data inside a Persistent Volume can be modified from within the container. Any changes your application makes to files

on the volume will persist.

Also, depending on the access mode (like `ReadWriteMany`), a single volume can sometimes be mounted by multiple pods at the same time — which is useful for shared config, caches, or collaboration between services.

References

- ConfigMaps
- Secrets
- External Secrets Operator
- Secrets Store CSI Driver
- Storage(Persistent Volumes)

Knowledge Check

- What are the two ways a ConfigMap can be mounted into a pod?
- What happens when you update a ConfigMap that is used as an environment variable?
- How does mounting a ConfigMap as a file behave differently from mounting it as an environment variable?
- Why are Kubernetes Secrets not considered fully secure by default?
- What tools can help integrate external secret management systems with OpenShift?
- When might you use a Persistent Volume instead of a ConfigMap or Secret?
- What is a PersistentVolumeClaim, and how does it get fulfilled in OpenShift?
- What is the role of a StorageClass in dynamic volume provisioning?
- Can data inside a Persistent Volume be modified by the pod? Why or why not?
- Under what circumstances can multiple pods share the same volume?

Observability

As you build and operate applications in OpenShift, observability becomes one of your most valuable tools.

You'll use metrics to scale applications with HPAs, inspect logs to debug crashes, and rely on probes and events to understand how deployments are behaving. Observability isn't just a feature — it's how you'll understand what's happening inside your workloads, at every stage of their lifecycle.

This module takes a closer look at the tools OpenShift provides to help you **monitor and troubleshoot your own applications**, not just the platform.

We'll focus on two key systems:

- **User Workload Monitoring** — for gathering custom application metrics and using them to power dashboards, alerts, and autoscalers

- **User Workload Logging** — for collecting and querying logs from your workloads at scale

By the end of this module, you'll know how to instrument your apps, access metrics and logs from the web console or CLI, and build a more observable system from day one.

User Workload Monitoring

OpenShift includes a built-in monitoring stack powered by **Prometheus**, **Alertmanager**, and **Grafana**. This stack collects platform-level metrics — like node health, etcd performance, and API server activity — and stores them in a highly available fashion.

By default, this stack is focused on monitoring **OpenShift itself**. But what about your own apps?

That's where **User Workload Monitoring** comes in.

What Is It?

User Workload Monitoring is an OpenShift feature that lets you scrape, store, and query **Prometheus-style metrics from applications running in your own projects**.

It gives you access to:

- A dedicated Prometheus instance for user-defined workloads

- A ServiceMonitor or PodMonitor interface to tell Prometheus what to scrape

- The ability to create alerts and visualize metrics using Grafana or the OpenShift web console

How to Enable It

User workload monitoring is disabled by default. You can enable it by modifying the `cluster-monitoring-config` ConfigMap:

```
oc edit configmap cluster-monitoring-config -n openshift-monitoring
```

Add or update the following:

```
data:
  config.yaml: |
    enableUserWorkload: true
```

This enables a second Prometheus stack in the `openshift-user-workload-monitoring` namespace.

> This step usually requires cluster-admin privileges. You'll also need to wait a few minutes for the new Prometheus instance to come online.

How to Expose Your Metrics

To expose metrics from your app, you'll need:

1. An HTTP `/metrics` endpoint (in Prometheus format)

2. A `Service` that selects your pods

3. A `ServiceMonitor` or `PodMonitor` resource that tells Prometheus where to scrape

Example `ServiceMonitor`:

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: my-app-monitor
  labels:
    release: prometheus
spec:
  selector:
    matchLabels:
      app: my-app
  namespaceSelector:
    matchNames:
      - my-project
  endpoints:
    - port: metrics
      interval: 30s
```

This tells Prometheus to scrape metrics from all pods labeled `app=my-app` in the `my-project` namespace, using the `metrics` port every 30 seconds.

Where to View Metrics

Once metrics are flowing, you can query them using:

- The **OpenShift web console** → **Observe** → **Metrics**
- The **Prometheus UI** exposed via `oc -n openshift-user-workload-monitoring port-forward svc/prometheus-user-workload 9090`
- A connected **Grafana instance** (if installed)

You can also use these metrics to power:

- Horizontal Pod Autoscalers (HPAs)
- Custom alerts with Alertmanager
- Dashboards for internal or external users

> User workload metrics are stored separately from platform metrics — and you are responsible for maintaining their signal quality. Avoid high cardinality, unbounded labels, or overly frequent scrapes.

User Workload Logging

Metrics give you the big picture — but logs are how you zoom in on specific events, requests, or failures. In OpenShift, you can enable a logging stack that collects, stores, and makes searchable the logs from your **application containers**.

This is especially useful when:

- You want to debug applications **without exec'ing into a pod**
- You want to **correlate logs across multiple pods or services**
- You need to **store logs** longer than a container's lifetime
- Your developers or support teams need log access without cluster-admin rights

What Is User Workload Logging?

OpenShift's modern logging architecture is built around the **Vector log collector**, which gathers logs from all nodes and routes them to one or more backends, such as:

- **Loki** — a horizontally-scalable log store designed for Kubernetes
- **Elasticsearch** — a traditional full-text log indexing engine
- (Optional) External log stores — S3, Splunk, Kafka, etc.

These logs can be accessed directly through:

- The **OpenShift Console → Observe → Logs**
- The **Grafana Loki UI**, if exposed

Enabling Logging for Workloads

To collect logs from application namespaces, follow these steps:

1. Install the **OpenShift Logging Operator** via **Operators → OperatorHub**
2. Install the **Loki Operator** (required if using Loki as your log store)

Then, deploy a `LokiStack` instance:

```
apiVersion: loki.grafana.com/v1
kind: LokiStack
metadata:
  name: logging-loki
  namespace: openshift-logging
spec:
  size: 1x.extra-small
  storage:
    schemas:
      - version: v12
        effectiveDate: "2022-06-01"
    secret:
```

```
      name: logging-loki-s3
    type: s3
  tenants:
    mode: openshift-logging
```

Finally, configure log forwarding using a `ClusterLogForwarder` resource:

```
apiVersion: logging.openshift.io/v1
kind: ClusterLogForwarder
metadata:
  name: instance
  namespace: openshift-logging
spec:
  inputs:
    - name: application-logs
      application:
        namespaces: ["my-project"]
  outputs:
    - name: default
      type: loki
      url: http://logging-loki-gateway.openshift-logging.svc:8080
  pipelines:
    - inputRefs: [application-logs]
      outputRefs: [default]
```

This setup tells Vector to collect logs from the `my-project` namespace and forward them to your LokiStack instance.

> Log collection is namespace-aware. You must explicitly include namespaces you want to monitor in the `ClusterLogForwarder`. This allows separation between platform logs and developer logs.

Accessing Logs

You can access logs directly from the **OpenShift Console**:

- Go to **Observe → Logs**
- Filter by project, pod, container, or log level
- Search or tail logs in real time without shell access

If desired, you can also expose and use the **Grafana Loki UI** for advanced log querying.

Best Practices

- Use structured logs (JSON is strongly preferred)
- Limit high-volume debug output in production
- Avoid logging sensitive information (secrets, tokens)

- Always log to `stdout` and `stderr` — not to local files

> **ⓘ** Enabling user workload logging gives you durable, searchable insight into your applications — making it easier to diagnose issues, investigate incidents, and support multi-team environments.

Optional: Network Observability Operator

If you want to go beyond metrics for CPU, memory, and custom application data, OpenShift also provides **network-level observability** using the **Network Observability Operator**.

This operator enables **flow-based network monitoring** — giving you visibility into traffic between namespaces, pods, services, external endpoints, and even dropped packets.

It is especially useful for:

- Troubleshooting network latency or traffic anomalies
- Understanding who is talking to whom inside your cluster
- Identifying unexpected or unauthorized external traffic
- Visualizing data flow between workloads

Once installed, it collects network flows using the eBPF-based **Flow Collector** and surfaces them in the **OpenShift Console under Observe → Network Traffic**.

You can:

- Filter flows by namespace, pod, protocol, or direction
- View conversations (source/destination pairs) and traffic rates
- Export flow logs to a remote system for long-term analysis

To install the operator:

1. Go to **Operators → OperatorHub** in the OpenShift Console
2. Search for **Network Observability Operator**
3. Install it into the `netobserv` namespace
4. Accept the default configuration (or customize as needed)

Once installed, navigate to **Observe → Network Traffic** to explore live and historical traffic flows.

> **ⓘ** The Network Observability Operator is powerful for debugging network issues, enforcing policies, and gaining visibility into production traffic patterns. It complements Prometheus-based monitoring by showing **how data moves**, not just how applications behave.

Optional: Distributed Tracing

For workloads that span multiple services, logs and metrics may not be enough. That's where **distributed tracing** comes in.

Tracing lets you see how a single request flows across your entire system — including timing information for each hop. This is invaluable for debugging performance issues, latency bottlenecks, or failed transactions.

In OpenShift, you can deploy **Grafana Tempo**, a lightweight, scalable tracing backend designed for cloud-native environments.

Deploying Tempo via the Tempo Operator

If you're already using **Grafana and Loki**, Tempo integrates seamlessly and allows you to correlate logs and traces using shared trace IDs.

To get started:

1. Install the **Tempo Operator** from OperatorHub

2. Deploy a `TempoStack` resource

3. Configure your applications to send traces using OpenTelemetry-compatible SDKs or exporters

Example:

```
apiVersion: tempo.grafana.com/v1alpha1
kind: TempoStack
metadata:
  name: tempo
  namespace: openshift-monitoring
spec:
  storage:
    type: memory
  replicas: 1
```

This creates a minimal, in-cluster Tempo setup suitable for development and testing.

Instrumenting Applications

Your applications need to be instrumented to generate and export trace data.

This is typically done using:

- **OpenTelemetry SDKs** for languages like Go, Java, Python, JavaScript, etc.
- Libraries or frameworks that support tracing headers and propagation
- Sidecars or ingress layers that forward trace headers (in service mesh scenarios)

Instrumentation includes:

- Creating spans for each operation

- Tagging spans with metadata (e.g., endpoint, status, duration)

- Forwarding trace data to Tempo via OTLP

> ℹ️ Distributed tracing adds critical visibility into the flow of requests across services — but only works if trace data is generated and exported by your workloads. Start with a single service and expand as you gain confidence.

References

- [Monitoring](#)

- [Logging](#)

- [Network Observability](#)

- [Tracing](#)

Knowledge Check

- What is the difference between **platform monitoring** and **user workload monitoring** in OpenShift?

- How do you expose Prometheus metrics from your application to the OpenShift monitoring stack?

- What are `ServiceMonitor` and `PodMonitor`, and when would you use each?

- How can you view user workload metrics from the OpenShift web console?

- What is the role of **Vector** in OpenShift's logging stack?

- How do you configure which application logs are collected by OpenShift?

- Why is it a bad practice to write logs to files inside the container?

- What kinds of issues is the **Network Observability Operator** designed to help identify?

- How does **distributed tracing** differ from traditional logging?

- What are some ways to instrument an application to produce trace data?

Scaling Applications

Applications don't always have a static workload — traffic can spike, data can grow, and demand can fluctuate throughout the day. OpenShift provides several tools that help your applications scale up when needed and scale down to save resources.

In this module, we'll look at how to:

- Use metrics to monitor application performance and behavior.

- Define resource requests and limits to ensure fair scheduling.

- Automatically scale applications with Horizontal Pod Autoscalers (HPAs).

- Use custom metrics to make smarter scaling decisions beyond just CPU and memory.

These tools help you build applications that are elastic, responsive, and efficient — ready to handle whatever comes their way.

Metrics

If you want to scale your application intelligently, you first need visibility into how it's performing. Metrics provide that visibility — they tell you how much CPU or memory your app is using, how many requests it's handling, and whether it's under stress or sitting idle.

In OpenShift, metrics are collected using **Prometheus** and exposed through the Kubernetes **Resource Metrics API**. These metrics are provided **out of the box** as part of the OpenShift platform — no additional setup is required for basic autoscaling and resource monitoring.

This API powers features like the **Horizontal Pod Autoscaler** (HPA), which can automatically adjust the number of running pods based on usage.

By default, the metrics you'll typically see include:

- **CPU usage** (measured in cores or millicores)

- **Memory usage** (measured in bytes)

- **Pod-level metrics** like restarts or container health

You can view these metrics using:

```
oc adm top pods
oc adm top nodes
```

To focus on a specific workload, you can:

- Use `oc adm top pod POD_NAME --namespace=my-app` to check metrics for a specific pod.

- Navigate to your project in the **OpenShift web console**, select **Workloads → Deployments**, and click your deployment. From there, the **Metrics** tab will show real-time CPU and memory usage for your application.

> For metrics-based scaling to work, metrics collection must be enabled in your cluster. In OpenShift, this is handled by the **Cluster Monitoring Operator**, which deploys and manages Prometheus and related services — and it's included out of the box.

Resource Limits and Resource Requests

In OpenShift, every container can declare how much CPU and memory it needs — and how much it's allowed to use. These values are defined using **requests** and **limits**, and they play a key role in how your application is scheduled, scaled, and isolated from noisy neighbors.

What's the Difference?

- A **request** is the amount of CPU or memory a container is **guaranteed**. The scheduler uses this value to decide where to place the pod.

- A **limit** is the **maximum** amount the container is allowed to use. If it tries to go beyond that, it may be throttled (CPU) or killed (memory).

For example:

```
resources:
  requests:
    memory: "256Mi"
    cpu: "250m"
  limits:
    memory: "512Mi"
    cpu: "500m"
```

This container is guaranteed 256Mi of memory and 250 millicores of CPU, but won't be allowed to use more than 512Mi or 500 millicores.

Quality of Service (QoS)

OpenShift uses these values to assign **Quality of Service (QoS)** classes to pods:

- `Guaranteed` — You specify matching requests and limits for every resource.

- `Burstable` — You set both request and limit, but they don't match.

- `BestEffort` — You don't set any request or limit.

QoS affects how pods are treated during resource pressure. `Guaranteed` pods are the last to be evicted; `BestEffort` are the first to go.

Scheduling Scenarios

These values directly impact where and how your pods are scheduled:

- **Pod stuck in Pending** — If your pod requests more memory or CPU than any node can currently provide (based on other workloads), the scheduler won't place it. It will stay in a `Pending` state until resources free up or the request is lowered.

- **Pod scheduled but throttled** — If your pod is placed on a node and it hits its **CPU limit**, the container runtime will throttle it. This won't kill the pod, but it may result in degraded performance.

- **OOMKilled** — If your container exceeds its **memory limit**, the kernel may terminate it. This usually shows up as a `OOMKilled` status in the pod's lifecycle.

These behaviors are designed to protect the node and other workloads, but they can cause confusion if you're not tracking resource usage closely.

> If your project defines a `LimitRange`, OpenShift will automatically apply default requests and limits to pods that don't specify them. This helps ensure consistent scheduling and resource fairness — even if you forget to add them manually.

Horizontal Pod Autoscalers

Horizontal Pod Autoscalers (HPAs) automatically adjust the number of pods in a workload based on real-time metrics — like CPU usage. They help your application scale up during high demand and scale down when things are quiet, so you're only using the resources you actually need.

HPAs are great for stateless workloads where performance depends on the number of running pods, such as web applications, APIs, and job workers.

Deploying an HPA

Let's say you already have a deployment called `my-app` running in your namespace. You can create an HPA like this:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-app-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-app
  minReplicas: 2
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 75
```

This configuration tells OpenShift:

- Always run **at least 2 pods** and never more than **10 pods**.

- Scale the workload so that the **average CPU usage across all pods stays at 75% of each pod's CPU request**. For example, if each pod requests 500 millicores, the HPA will try to maintain usage around 375 millicores per pod.

Once this HPA is applied, OpenShift will continuously evaluate CPU usage (using metrics provided out of the box) and automatically adjust the number of pods to maintain the target utilization.

Understanding `behavior`

The `behavior` field allows you to fine-tune how aggressively the HPA reacts to changes. It helps prevent overreaction (flapping) or sluggish response during rapid demand changes.

Here's an example:

```
behavior:
  scaleUp:
    stabilizationWindowSeconds: 60
    policies:
      - type: Percent
        value: 100
        periodSeconds: 15
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
      - type: Pods
        value: 1
        periodSeconds: 60
```

This configuration does a few things:

- **Scale-up policy**: When scaling up, the HPA can double the number of pods every 15 seconds, but it will **wait 60 seconds** to make sure the spike is stable before acting.

- **Scale-down policy**: When scaling down, it can only reduce the workload by **1 pod every 60 seconds**, and it waits **5 minutes** of low usage before starting to scale down.

These settings help smooth out noisy metrics and ensure the autoscaler behaves in a predictable way.

> If you omit the `behavior` section, the HPA uses conservative defaults:
>
> - **Scale up**: No limit on how fast the workload can grow, but the controller scales gradually to avoid overshooting.
>
> - **Scale down**: The HPA waits 5 minutes of stable usage before reducing the number of pods, and only scales down one pod at a time.
>
> For most production workloads, tuning `behavior` is a good idea — especially for highly dynamic traffic or cost-sensitive environments.

Custom Metrics for Horizontal Pod Autoscalers

CPU and memory usage are great starting points for autoscaling — but they don't always reflect what really matters to your application.

What if you want to scale based on:

- The number of incoming HTTP requests

- Queue length in a job processing system

- Application-specific metrics like database connection saturation or user sessions?

This is where **custom metrics** come in. OpenShift supports Horizontal Pod Autoscalers that use custom application-level metrics — collected via Prometheus and surfaced to the HPA through **KEDA**.

What is KEDA?

KEDA (Kubernetes Event-driven Autoscaling) is an open source project that integrates with the Kubernetes control plane to provide autoscaling based on custom or external metrics. OpenShift includes KEDA as part of its serverless and advanced autoscaling support.

KEDA installs its own controller that listens for `ScaledObject` resources and drives HPA behavior based on metrics from Prometheus, Kafka, Redis, AWS CloudWatch, and more.

In OpenShift, you can use KEDA with Prometheus to drive autoscaling based on metrics your application emits.

Resources Involved (When Using KEDA with Prometheus)

Assuming you've already installed KEDA and **User Workload Monitoring** is enabled, here's what you'll need to make a custom metrics autoscaler work:

1. **An application that exposes Prometheus metrics** Your app must expose metrics in Prometheus format, typically at `/metrics`.

2. **A PodMonitor or ServiceMonitor** This resource tells Prometheus how to scrape your application's metrics. It must match the labels on your pod or service and specify the correct port and path.

3. **A ServiceAccount with credentials to query Prometheus** This is the identity that KEDA will use to fetch metrics. The service account should live in the same namespace as your app.

4. **A Role or ClusterRole that grants access to the Prometheus API** You'll need to bind a role that allows the service account to query metrics from the Prometheus `thanos-querier` endpoint.

5. **A TriggerAuthentication resource** This object references the service account and provides the authentication context for KEDA to talk to Prometheus.

6. **A ScaledObject** Finally, you define a `ScaledObject`, which links your deployment to a custom metric, defines the threshold, and instructs KEDA how to scale based on it.

Once all of these are in place, KEDA handles the rest — scraping the metric from Prometheus, calculating when to scale, and adjusting the number of pods using a built-in HPA. There is a full tutorial here.

Example: Scaling Based on HTTP Request Rate

Let's say you have an application exposing a Prometheus metric called `http_requests_total`. This metric is scraped by Prometheus and made available through the KEDA Prometheus scaler. You want to scale your workload when it receives more than **5 requests per 2 minutes**.

First, ensure your app exposes the metric in Prometheus format, e.g.:

```
# Exposed by the app at /metrics
http_requests_total{route="/api"} 12452
```

Then configure a ScaledObject resource that uses the Prometheus trigger:

```yaml
apiVersion: keda.sh/v1alpha1
kind: ScaledObject
metadata:
  name: prometheus-scaledobject
  namespace: default
spec:
  maxReplicaCount: 10
  minReplicaCount: 1
  scaleTargetRef:
    name: prometheus-example-app
  triggers:
  - authenticationRef:
      name: example-triggerauthentication
    metadata:
      authModes: bearer
      metricName: http_requests_total
      namespace: default
      query: sum(rate(http_requests_total{job="default/prometheus-example-app"}[2m]))
      serverAddress: https://thanos-querier.openshift-
monitoring.svc.cluster.local:9092
      threshold: "5"
    type: prometheus
```

And a corresponding TriggerAuthentication:

```yaml
apiVersion: keda.sh/v1alpha1
kind: TriggerAuthentication
metadata:
  name: example-triggerauthentication
  namespace: default
spec:
  secretTargetRef:
  - key: token
    name: autoscaler-token
    parameter: bearerToken
  - key: ca.crt
    name: autoscaler-token
    parameter: ca
```

This setup ensures KEDA can:

- Connect to Prometheus using a secure service account

- Query your application-specific metric

- Automatically scale your deployment based on the result

> KEDA makes it possible to scale your applications based on external or custom metrics — even those not natively exposed by Kubernetes. It's especially useful when you need to scale based on business logic or throughput, not just CPU or memory.

Autoscaling workloads with HPAs and autoscaling cluster nodes (with Cluster Autoscaler or MachineAutoscaler) at the same time introduces complexity. Both layers are adjusting based on demand, which can create feedback loops or unpredictable scaling patterns. Before adopting custom or CPU-based autoscaling in production, make sure you understand how to observe and debug both layers of scaling. Tools like `oc describe hpa`, Prometheus dashboards, and reviewing KEDA logs are essential when troubleshooting.

References

- [Requests and Limits](#)

- [Autoscaler](#)

- [Keda](#)

- [Custom Metrics Autoscaler](#)

Knowledge Check

- What's the difference between a resource **request** and a **limit**?

- What determines a pod's **Quality of Service (QoS)** class, and why does it matter?

- What happens if a pod exceeds its memory limit? What about CPU limit?

- What does an HPA do with the metric `averageUtilization: 75`?

- How can you tune how aggressively an HPA scales up or down?

- What is KEDA, and how does it enhance autoscaling in OpenShift?

- What components are required to scale an application based on a Prometheus metric using KEDA?

- Why is it important to understand both **workload autoscaling** and **node autoscaling** before enabling them together?

- Which tools can you use to debug scaling behavior in OpenShift?

Debugging Applications

Even with the best CI/CD pipelines and automation, something will eventually go wrong. A container crashes, a pod won't start, or your app is running — but not responding the way you expect.

OpenShift provides several tools to help you dig into these problems, whether the issue is with your app's logic, container runtime, network traffic, or storage.

In this module, we'll walk through techniques for:

- Inspecting pod behavior with the `oc debug` command
- Troubleshooting networking issues — including service discovery, DNS resolution, and external access
- Understanding how to debug PVCs and storage mounts
- Identifying common failure modes like CrashLoopBackOff, OOMKilled, or image pull errors
- Applying general best practices for diagnosing stuck or failing applications

By the end of this section, you should be able to confidently troubleshoot most common issues that show up when working with OpenShift workloads.

Common Issues

Before diving into debugging tools, it helps to recognize the types of problems you're most likely to encounter — and where OpenShift reports them.

Containers can fail in many ways, but the symptoms usually show up in a few predictable places:

Pod Statuses

You can learn a lot just by checking a pod's status:

- `CrashLoopBackOff` — the container starts, fails, and restarts repeatedly.
- `ImagePullBackOff` or `ErrImagePull` — the image couldn't be pulled, often due to authentication or tag errors.
- `OOMKilled` — the container was terminated because it exceeded its memory limit.
- `Pending` — the pod can't be scheduled, often due to insufficient resources or missing PVCs.
- `Completed` — the pod finished its task (expected for jobs and init containers).

Use this command to check:

```
oc get pods
```

Events

Events are how OpenShift reports issues from the scheduler, controller manager, and other core components. They often show why a pod is stuck or failed.

You can see them with:

```
oc describe pod POD_NAME
```

Look for messages like:

- "FailedScheduling" — usually tied to node availability or resource requests
- "FailedMount" — often related to volume issues
- "Back-off restarting failed container" — signals repeated crashes

Logs

Logs are often the most direct way to identify application-level issues. If the pod is running or recently failed, try:

```
oc logs POD_NAME
```

If the pod has multiple containers, add `-c` to target one:

```
oc logs POD_NAME -c CONTAINER_NAME
```

> **ℹ** Most problems show up as a combination of pod status, events, and logs. Understanding how to read those three signals is the first step toward resolving any issue.

Debug Process

When something goes wrong in OpenShift, it can be tempting to jump straight into logs or YAML files — but the most effective debugging starts with a structured process.

Here's a common workflow used by experienced developers and administrators:

1. Identify the Symptom

Start with what you know:

- Is the application unavailable, slow, or crashing?
- Is the pod stuck in a particular state?
- Are users reporting specific errors or timeouts?

Use `oc get pods` to quickly scan for anomalies in pod status.

2. Gather Context

Check the environment around the failure:

- Is it affecting one pod or many?
- Has anything recently changed — deployments, configs, images, or secrets?
- Are any PVCs or services involved?

This helps narrow your investigation.

3. Describe the Resource

Use `oc describe` to get detailed information:

```
oc describe pod POD_NAME
```

Look at the events section, resource limits, volume mounts, and status messages. This will often point directly to the root cause.

4. Check the Logs

If the pod is running or recently failed, pull logs:

```
oc logs POD_NAME
```

If your workload has multiple containers, include `-c CONTAINER_NAME`.

If the pod crashed before writing logs, try:

```
oc logs --previous POD_NAME
```

5. Interactively Debug the Container

If the pod starts but doesn't behave as expected, you can inspect it interactively:

```
oc rsh POD_NAME
```

Or use a clean debug container with OpenShift tooling:

```
oc debug pod/POD_NAME
```

More details on this method are in the **Debug Pod** section below.

This gives you root access to a clone of the environment — great for exploring filesystem layout, testing DNS resolution, or running shell commands.

6. Examine Configuration

If the container is healthy but not working, check mounted config files, environment variables, and injected secrets.

To list environment variables for a specific pod:

```
oc set env pod/POD_NAME --list
```

To list environment variables for all pods in the namespace:

```
oc set env pods --all --list
```

You can also view what volumes are mounted using `oc describe`.

> ℹ️ Debugging isn't just about fixing — it's about learning. If you keep a consistent process, you'll spot patterns faster and resolve issues more confidently.

Debug Pod

Sometimes, a pod isn't working as expected — but it's hard to tell why. Maybe the container isn't starting, or it fails before you can log in. In those cases, OpenShift provides two powerful tools to help you investigate: the `oc debug` command and **ephemeral containers**.

Both tools let you interactively explore what's happening inside a pod — without needing to modify your original deployment.

Option 1: Using `oc debug`

The `oc debug` command is one of the most versatile ways to troubleshoot pod behavior.

When you run `oc debug pod/POD_NAME`, OpenShift:

- Creates a **temporary pod** based on the original
- Replaces the container image with a known-good one (usually a Red Hat UBI image)
- Mounts the same volumes, secrets, and configs
- Gives you a shell with root privileges

This is ideal when:

- The container exits before you can run `oc rsh`
- You need to inspect mounted files, config, or secret data
- You want a clean environment with troubleshooting tools installed

Example:

```
oc debug pod/POD_NAME
```

Inside the debug shell, you can:

```
ls /etc/secrets          # Explore secrets
```

```
cat /etc/resolv.conf      # Check DNS settings
env                       # View environment variables
```

You can also change the image used for debugging:

```
oc debug pod/POD_NAME --image=registry.access.redhat.com/ubi8/ubi
```

Or override the container's entrypoint:

```
oc debug pod/POD_NAME -- /bin/bash
```

Option 2: Attaching Ephemeral Containers

An **ephemeral container** is a lightweight debugging container that you can temporarily inject into a running pod — even if the pod's containers don't have a shell or crash on startup.

Unlike `oc debug`, ephemeral containers:

- Attach directly to the **live** pod without creating a copy
- Run side-by-side with existing containers
- Are not restarted automatically and don't affect the pod's lifecycle

To add one, use `kubectl` (ephemeral containers are not yet supported directly by `oc`):

```
kubectl debug --target=CONTAINER_NAME pod/POD_NAME
```

This attaches a troubleshooting container (based on UBI by default) to the running pod and targets a specific container (useful for matching volumes or namespaces).

You can verify the container was added by describing the pod:

```
oc get pod POD_NAME -o yaml
```

Look for the `ephemeralContainers` section in the output.

> ℹ️ Use ephemeral containers when the pod is still running but doesn't offer a direct shell or logs aren't telling the full story. Use `oc debug` when the pod crashes immediately or you need a clean shell environment with debugging tools.

Debug Network

Networking issues can be some of the most frustrating to troubleshoot — because they might not look like networking issues at first. A service times out, a connection is dropped, or an app just "hangs" with no clear reason.

OpenShift networking includes multiple layers: DNS, Services, Routes, Ingress, firewall rules, and pod-to-pod communication — so a structured approach is essential.

Start with Context

The most effective way to debug networking issues is to pick a starting point — either:

- The **container/pod that's trying to make a connection**, or
- The **client that's trying to reach a service**

Then work your way **step by step** through the network path. This helps you isolate where the failure is happening — and rule out parts that are working.

Common Problem Areas

Networking problems can stem from many sources. Here are some of the most common:

- **DNS resolution** — The pod can't resolve a service name to an IP.
- **Port issues** — The wrong port is exposed, or the container isn't listening.
- **Protocol mismatches** — The app expects HTTPS, but the service sends plain HTTP (or vice versa).
- **Service or endpoint misconfiguration** — A service has no healthy endpoints to forward traffic to.
- **Node locality issues** — A `Service` with `internalTrafficPolicy: Local` can't reach a backend if there are no matching pods on the same node.
- **NetworkPolicy blocking traffic** — A policy is in place that prevents traffic between pods or namespaces.
- **Pod readiness problems** — A pod isn't passing its readiness probe, so the service won't send traffic to it.

Tools to Help

Here are some OpenShift-native tools and techniques to help you investigate:

- Use `oc rsh` or `oc debug` to get a shell inside a pod:

```
oc rsh POD_NAME
curl http://SERVICE_NAME:PORT
```

- Check DNS resolution inside the pod:

```
getent hosts SERVICE_NAME
cat /etc/resolv.conf
```

- List endpoints for a service (to verify it's routing to the right pods):

```
oc get endpoints SERVICE_NAME
```

- Examine the service definition and target port:

```
oc describe svc SERVICE_NAME
```

- Test across namespaces:

```
curl http://SERVICE_NAME.NAMESPACE.svc.cluster.local:PORT
```

- Use `oc exec` or `oc debug` with `tcpdump`, `netstat`, or `ss` for deeper TCP-level debugging (you may need to install these tools in a debug pod).

> 🛈 When debugging network issues, assume nothing. A single missing port, label, or probe can silently break connectivity. Step-by-step checks help you validate each layer — and prevent wild goose chases.

Debug Storage

Storage issues can be subtle — your application may fail to start, hang on I/O, or crash unexpectedly. In OpenShift, Persistent Volume Claims (PVCs) are used to bind workloads to storage. If that binding fails or the volume misbehaves, it can block the entire pod from running.

The good news is: most storage problems can be debugged with a consistent set of steps.

What Can Go Wrong?

Some common storage-related problems include:

- A pod is stuck in `Pending` because its PVC hasn't been bound
- A PVC is created, but no `PersistentVolume` is available to match it
- The pod fails with a `FailedMount` event or hangs during startup
- The container logs show read/write errors or permission issues
- The pod is running, but the expected files aren't present

Step-by-Step Debugging

Here's a methodical process to follow:

1. Check the PVC status

```
oc get pvc
```

You should see the `STATUS` column as `Bound`. If it's `Pending`, the cluster couldn't find or create a

matching volume.

To see more detail:

```
oc describe pvc PVC_NAME
```

Look for messages about why the binding failed (e.g., no matching StorageClass, size too large, etc.).

2. Check the pod's event logs

Use `oc describe pod` to look for `FailedMount` or `MountVolume.SetUp` errors:

```
oc describe pod POD_NAME
```

Common issues include:

- AccessMode mismatch (`ReadWriteOnce` vs `ReadOnlyMany`)
- Volume not attaching or not mounting
- File system formatting errors

3. Enter the pod and inspect the mount

If the pod is running, you can inspect the mount inside the container:

```
oc rsh POD_NAME
mount | grep /mnt
df -h
ls -l /mnt/data    # Replace with your actual mount path
```

Make sure the volume is there, readable, and writable.

4. Check file permissions and user IDs

If the files are present but inaccessible, the problem might be with UID/GID mismatches or restrictive securityContext settings.

- Make sure the volume contents are owned by the correct UID
- Check if the container is running as non-root

You can also inspect the SCC (Security Context Constraint) in use:

```
oc get pod POD_NAME -o yaml | grep -i security
```

5. Look at the StorageClass

If PVC provisioning is failing, check what StorageClass is being used:

```
oc get storageclass
```

Then describe it:

```
oc describe storageclass CLASS_NAME
```

This can reveal issues like reclaim policy, volumeBindingMode, or provisioner errors.

> ℹ️ Storage issues can block pods silently. If your app won't start and no logs are printed, always check for missing volumes or failed mounts.

> ❗ Don't forget that Red Hat Support is just an email away!

References

- Investigating Pod Issues
- How OC Debug Works
- Ephemeral Containers

Knowledge Check

- What are some of the most common pod status values, and what do they indicate?
- How can you view recent events related to a pod's failure?
- What's the difference between using `oc rsh` and `oc debug`?
- What are ephemeral containers, and when would you use one over `oc debug`?
- What step-by-step approach can help you diagnose a failed network connection inside a pod?
- How can `internalTrafficPolicy` affect pod-to-pod connectivity?
- What commands can you use to check that a PVC is correctly bound and mounted?
- How can incorrect file permissions or security contexts cause read/write errors on a mounted volume?
- Why is it important to understand both the pod definition and the underlying `StorageClass` when debugging storage issues?

Deployment Strategies

Deploying an application isn't just about pushing new code — it's about doing so **safely** and **predictably**, with minimal risk and downtime.

OpenShift gives you several strategies to control how updates roll out, from simple approaches like replacing old pods with new ones, to more advanced workflows like canary and blue-green deployments.

In this module, we'll walk through:

- The basic rollout strategies: `Recreate` and `Rolling`

- More advanced models: `Blue-Green`, `Canary`, and `A/B testing`

- The critical role of health checks in making any of this work

It's important to understand that **none of these strategies work reliably without health checks**. OpenShift relies on three types of **probes** to know whether a pod is ready, healthy, or even safe to start in the first place:

- **Readiness Probe** — Controls whether a pod is eligible to receive traffic

- **Liveness Probe** — Detects when a pod needs to be restarted

- **Startup Probe** — Helps slow-starting containers avoid being killed prematurely

We'll revisit these probes throughout the examples.

By the end of this module, you'll understand how to choose the right strategy for your workload — and how to build deployments that are robust, observable, and easy to roll back when something goes wrong.

Recreate and Rolling Strategies

When you're deploying a new version of your application, OpenShift provides a couple of built-in deployment strategies to manage how pods are replaced.

These two core strategies — **Recreate** and **Rolling** — serve very different purposes depending on the needs of your application.

Recreate Strategy

The **Recreate** strategy is the simplest approach: it stops all the old pods **before** starting any new ones.

This means there is a brief period of downtime while the old version is removed and the new version is brought up.

Use this when:

- Your application **can't tolerate having two versions running at once** (e.g., when there's a shared database schema change)

- You're running a **single-instance** workload that doesn't need high availability

- You want to keep deployment logic as simple as possible

> ℹ️ Recreate is risky for production workloads unless carefully planned. Users will experience downtime unless external routing or failover is used. If your workload cannot tolerate overlap or requires strict startup ordering, consider using a **StatefulSet** instead of a Deployment.

You can specify it in a Deployment like this:

```
spec:
  strategy:
    type: Recreate
```

Rolling Strategy

The **Rolling** strategy is the default in OpenShift. It replaces old pods with new ones **gradually**, so there's always at least part of your application available.

OpenShift spins up a few new pods, waits for them to become ready, and then terminates the old ones in batches.

This works well when:

- Your application can run multiple versions side-by-side (even briefly)
- You want **zero-downtime** deployments
- You have health checks configured (readiness and liveness)

You can customize how fast the rollout happens using `maxUnavailable` and `maxSurge`:

```
spec:
  strategy:
    type: Rolling
    rollingParams:
      maxUnavailable: 25%
      maxSurge: 25%
```

This means:

- No more than 25% of the pods can be **unavailable** at a time
- OpenShift can create **up to 25% more pods** than the desired count during rollout

> ℹ️ The Rolling strategy relies heavily on **readiness probes**. If a new pod never becomes ready, OpenShift will pause the deployment and report the issue — giving you time to fix things before bad code reaches users.

Blue-Green Deployment Strategy

The **Blue-Green** strategy involves maintaining two separate environments — one "live" (Blue), and one "staged" (Green). When you're ready to release, you simply switch traffic from Blue to Green.

This strategy minimizes downtime and risk by keeping the new version completely isolated until it's proven to be working.

How It Works

1. Your live version (Blue) is running and receiving traffic.

2. You deploy the new version (Green) as a separate deployment — often with a different name, label, or route.

3. You run tests against the Green version while it's isolated.

4. When you're ready, you update a **Service** or **Route** to point to Green.

5. If anything goes wrong, you can roll back by switching traffic back to Blue.

This strategy requires you to **manually manage routing or labels**, but it gives you maximum control.

Example Setup

You might have two deployments:

```
oc get deployment
NAME            READY   UP-TO-DATE   AVAILABLE
my-app-blue     3/3     3            3
my-app-green    3/3     3            3
```

And a service that initially points to Blue:

```
selector:
  app: my-app
  color: blue
```

To switch traffic to Green, you change the selector:

```
selector:
  app: my-app
  color: green
```

Alternatively, if you're using **Routes**, you can assign the Route to the Green deployment instead of Blue, then roll back if needed.

When to Use It

- You need **zero-downtime** upgrades with a fast rollback option

- You want to validate the new version **in production**, but without exposing it to users immediately

- Your application allows two versions to run in parallel

If you don't want to manage route switching manually, you can automate blue-green deployment workflows using the **OpenShift GitOps Operator**, which includes support for **Argo Rollouts**. Argo

enables declarative progressive delivery strategies like blue-green with built-in traffic switching, automated analysis, and rollbacks — all controlled through GitOps.

> The OpenShift GitOps Operator (powered by Argo CD and Argo Rollouts) provides native support for automating blue-green and canary rollouts. This can simplify your deployment pipelines by taking care of service and route switching for you.

Canary Deployment Strategy

The **Canary** strategy is a progressive rollout model where you deploy a small percentage of traffic to a new version of your application and gradually increase it as confidence builds.

This reduces risk by limiting the blast radius of any bugs or regressions. If something goes wrong, you can stop the rollout early — before exposing the issue to all users.

How It Works

In a canary deployment:

1. You deploy a new version of the application alongside the stable one.
2. A **small percentage of traffic** (e.g., 5–10%) is routed to the new version.
3. You monitor metrics, logs, or user feedback to validate the release.
4. If all goes well, you gradually increase traffic to the new version until it reaches 100%.
5. If problems arise, you roll back and send all traffic back to the stable version.

Canary deployments often rely on **advanced traffic routing** to split requests between versions — typically using service mesh, ingress controllers, or route weights.

Canary + Blue-Green

Canary is often used **in combination with blue-green** deployments to smooth the transition:

- The "green" environment is deployed with the new version
- A small percentage of traffic is routed to Green as a **canary**
- As confidence grows, routing is shifted fully from Blue to Green

This approach gives you the **safe isolation of blue-green** with the **gradual exposure of canary**, resulting in a highly controlled release strategy.

Example with Manual Routing

You can implement a basic canary strategy using OpenShift Routes by assigning weights to backends:

```
kind: Route
spec:
  to:
```

```
    kind: Service
    name: my-app-v1
    weight: 90
  alternateBackends:
    - kind: Service
      name: my-app-v2
      weight: 10
```

This sends 90% of the traffic to the stable version and 10% to the canary.

> ℹ️ Traffic shaping via Route weights is a simple way to implement canary logic in OpenShift. For more sophisticated control, you can use OpenShift Service Mesh (Istio) or the GitOps Operator with Argo Rollouts.

Automating with GitOps and Argo

The **OpenShift GitOps Operator** (powered by Argo CD) supports `Canary` rollout steps as part of the `Argo Rollouts` API.

With this setup, you can define a rollout with traffic steps, analysis, and even automatic rollback:

```
strategy:
  canary:
    steps:
      - setWeight: 10
      - pause:
          duration: 1m
      - setWeight: 50
      - pause:
          duration: 5m
```

Argo Rollouts can pause between steps, evaluate metrics, and give you a chance to observe behavior before proceeding.

> ℹ️ Canary deployments provide a controlled release path, but they work best with **automated observability** (logs, metrics, and alerts) in place. Without feedback signals, a canary is just a slower rollout — not a safer one.

A/B Testing Strategy

A/B testing is a deployment strategy focused on **running two or more versions of an application in parallel** and directing different types of users or requests to each version. The goal isn't just safe rollout — it's **experimentation**.

This approach is often used to test new features, UI changes, or algorithm tweaks on a subset of users and compare behavior, conversion rates, or performance metrics.

How It Works

In an A/B setup:

1. Multiple versions of the application (A and B) are deployed side-by-side.

2. Traffic is split **intentionally**, based on request headers, cookies, user identity, or other rules.

3. Observability tools are used to compare user behavior between versions.

4. Based on results, you may promote B to 100%, roll it back, or continue iterating.

Unlike canary deployments, which focus on safe progressive rollout, A/B is about testing **variation** — and both versions may stay live for a long time.

Implementing A/B in OpenShift

There's no built-in A/B testing primitive in Kubernetes or OpenShift, but you can implement it using tools like:

- **OpenShift Routes** with custom middleware (e.g., HAProxy header-based routing)

- **OpenShift Service Mesh (Istio)** to direct traffic based on headers or cookies

- **Argo Rollouts** with **analysis templates** that measure business metrics

A/B routing usually relies on **something in the incoming request** — like a custom HTTP header, a cookie, or a user-specific token.

Users might receive these headers:

- From a **feature flag service** or experimentation platform

- Through **custom headers added by a mobile or frontend client**

- From an **authentication gateway** or reverse proxy that injects group or role data

- Via **browser cookies** set after opting into a beta program

For example, using Service Mesh:

```
match:
  - headers:
      x-user-group:
        exact: beta
    route:
      - destination:
          host: my-app-v2
  - route:
      - destination:
          host: my-app-v1
```

This sends beta users to version B and everyone else to version A.

When to Use It

- You want to test features on a **subset of users**
- You want to compare **business impact** between versions
- You're experimenting with UI changes, recommendations, or pricing models

> 🛈 A/B testing is about data-driven decisions. It requires coordination between your platform, observability stack, and product team to be effective. It's less about "rolling out" and more about "learning fast."

Health Checks and Probes

We've referenced probes several times at this point, but we haven't given a clear definition. Let's fix that.

Before you can safely deploy or scale applications, OpenShift needs a way to know whether your containers are behaving correctly. This is where **probes** come in.

OpenShift supports three types of probes:

- **Readiness Probe** — tells OpenShift when a pod is ready to receive traffic
- **Liveness Probe** — tells OpenShift when a pod should be restarted
- **Startup Probe** — tells OpenShift how long to wait for an app to start before applying liveness checks

Each of these serves a different purpose during the container lifecycle.

Probe Types: Execution vs Network

All probes fall into one of two test categories:

- **Execution-based probes** run a command **inside the container**. This is useful for checking application state, presence of lock files, or internal flags.
- **Network-based probes** check application connectivity over HTTP, TCP, or gRPC. These are typically used to verify that a service is responsive and accepting traffic.

OpenShift supports the following probe types:

- `exec` — runs a command inside the container
- `httpGet` — sends an HTTP GET request to a specific path and port
- `tcpSocket` — opens a TCP connection to a given port
- `grpc` — performs a gRPC health check using the standard gRPC health protocol

Choosing the right type depends on what your application is doing and how you want to measure its health.

Now let's look at how each probe type fits into the container lifecycle.

Readiness Probes

OpenShift uses readiness probes to decide when a container is eligible to receive traffic from Services. If the readiness check fails, the container is temporarily removed from the pool of backends.

This is especially important during rolling deployments — OpenShift won't scale down old pods until new ones are **ready**.

Example: HTTP readiness check

```
readinessProbe:
  httpGet:
    path: /healthz
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10
```

Liveness Probes

Liveness probes detect **long-term failure**. If the probe fails consistently, OpenShift will restart the container. This is useful for recovering from deadlocks, hung processes, or stuck threads.

Example: TCP liveness check

```
livenessProbe:
  tcpSocket:
    port: 3306
  initialDelaySeconds: 15
  periodSeconds: 20
```

Example: Exec liveness check

```
livenessProbe:
  exec:
    command:
      - cat
      - /tmp/app.lock
  initialDelaySeconds: 10
  periodSeconds: 5
```

Startup Probes

Startup probes are designed for applications that take a long time to initialize. They delay the start of liveness checks so the container has a chance to fully boot before being restarted prematurely.

If a startup probe is defined, **liveness probes are disabled** until the startup probe succeeds once.

Example: Long boot time

```
startupProbe:
  httpGet:
    path: /startup
    port: 8080
  failureThreshold: 30
  periodSeconds: 10
```

This gives the container up to **5 minutes** (30 × 10 seconds) to start up before OpenShift intervenes.

Combining Probes

You can use all three probes in a single container to cover:

- Startup time (via `startupProbe`)

- Runtime health (via `livenessProbe`)

- Traffic readiness (via `readinessProbe`)

Example: Combined probe setup

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 10

livenessProbe:
  httpGet:
    path: /live
    port: 8080
  initialDelaySeconds: 15
  periodSeconds: 20

startupProbe:
  httpGet:
    path: /startup
    port: 8080
  failureThreshold: 30
  periodSeconds: 10
```

> 🛈 Readiness and liveness probes are essential for reliable deployments, scaling, and self-healing. Without them, OpenShift has no way to safely replace or recover containers.

References

- [Deployment Strategies](#)
- [GitOps Operator](#)
- [Health Checks](#)

Knowledge Check

- What are the key differences between the **Recreate** and **Rolling** deployment strategies?
- Why are **readiness probes** critical for safe rolling deployments?
- When would you consider using a **StatefulSet** over a standard Deployment?
- What is a **blue-green deployment**, and how do you switch traffic between environments?
- How can OpenShift Routes be used to implement basic **blue-green or canary** deployments?
- What is the purpose of a **canary rollout**, and how is it different from blue-green?
- How does **Argo Rollouts** enhance deployment strategies like canary or blue-green?
- What is the main goal of an **A/B testing** deployment, and how does it differ from a traditional rollout?
- What kinds of request data (e.g., headers or cookies) might be used to route traffic in an A/B test?
- Which tools in OpenShift can help automate or control progressive delivery strategies?