OpenShift Primer

2025-04-16

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
- **Scaling Applications**:
  Leveraging OpenShift's robust scaling capabilities
- **Debugging Applications**:
  Common workflows for identifying failure scenarios and fixes
- **Deployment Strategies**:
  Workloads beyond basic deployments
- **Observabiliity**:
  Monitoring your application metrics and logs with OpenShift

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.

Jump to Day Two

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

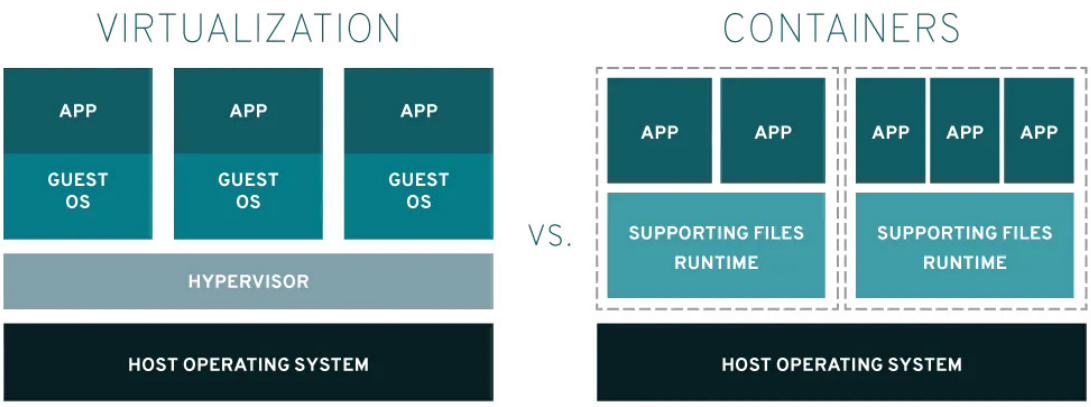
These features reduce the operational burden on developers, allowing them to focus more on writing code and delivering value. OpenShift enhances the Kubernetes developer experience by abstracting complexity and providing a more integrated, opinionated platform tailored for enterprise use.

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...

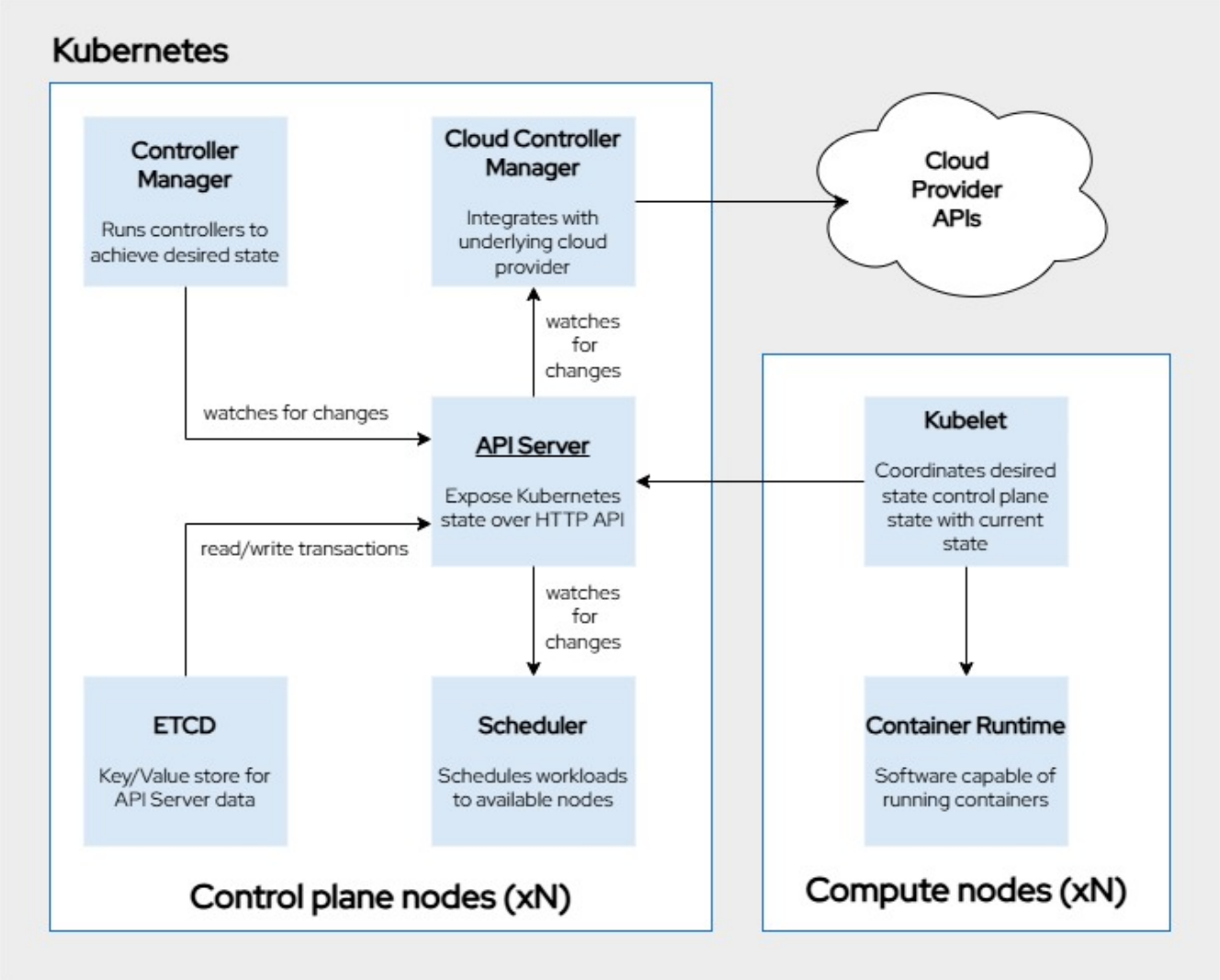
- **P**ortability - Kubernetes makes minimal demands on the underlying infrastructure, and can run on diverse sets of systems.
- **U**sability - Kubernetes configuration is API driven, and every change can be implemented with YAML syntax.
- **R**eliability - Kubernetes is self-healing, limiting the effects 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:

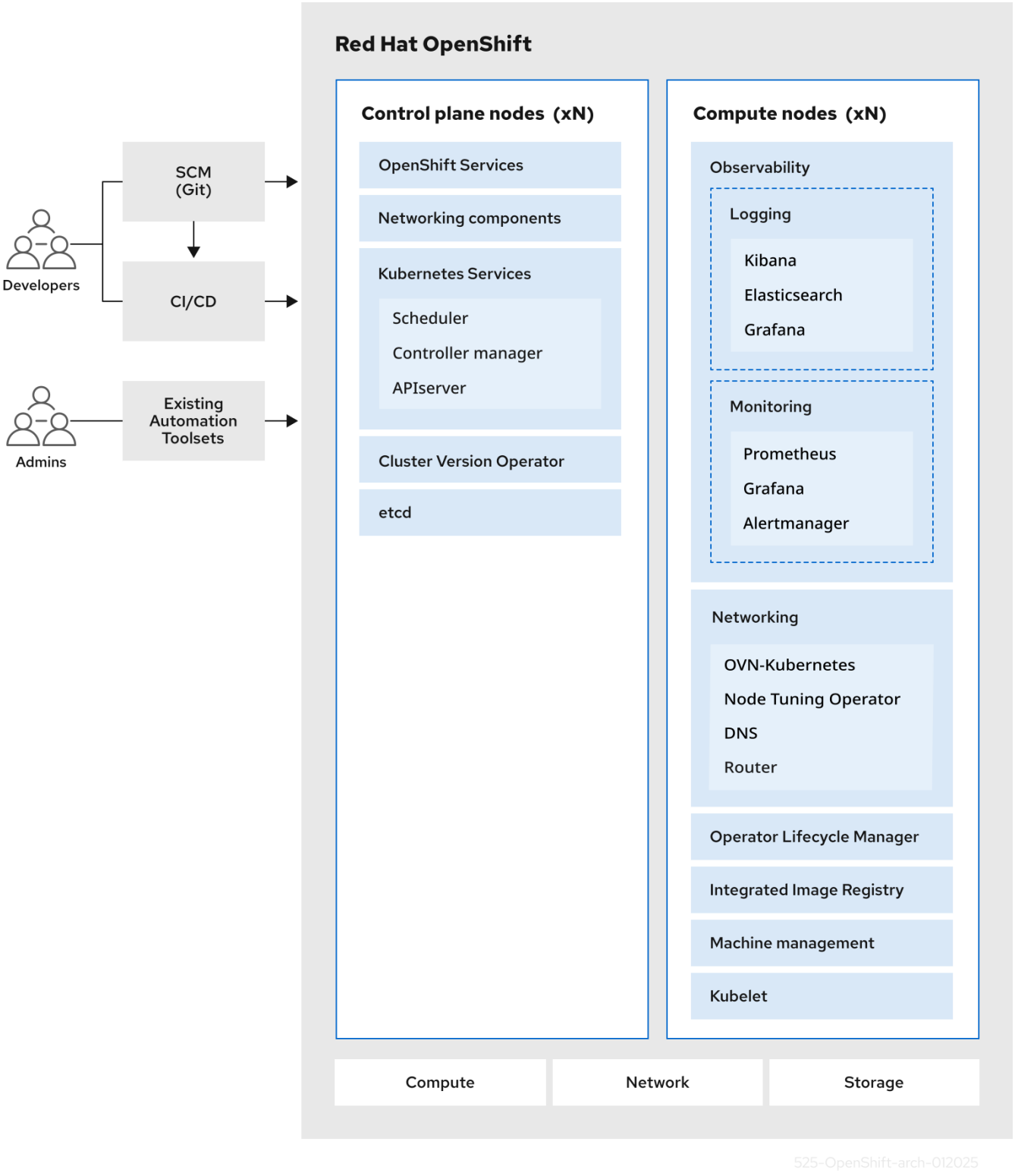
- What is Kubernetes?
- Kubernetes 101
- Learn Kubernetes Basics

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.



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

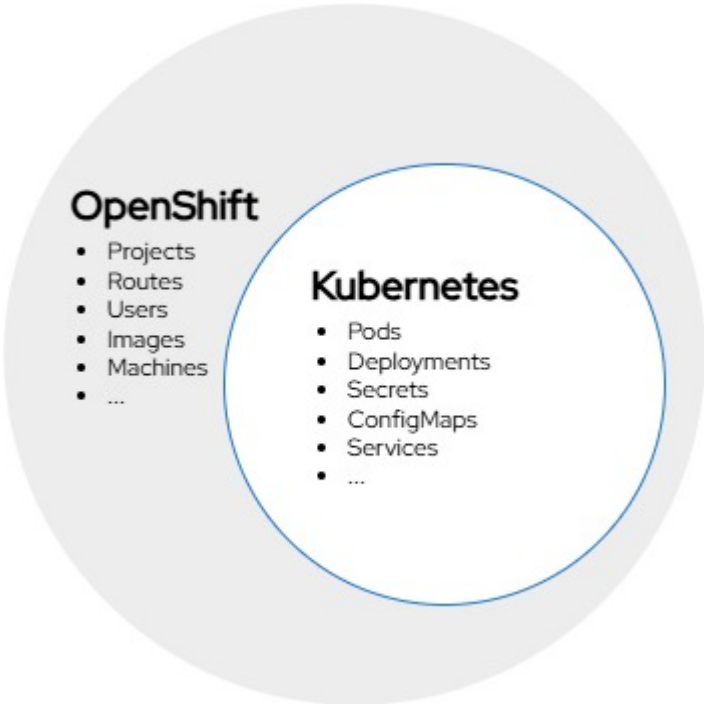
From a fresh install you can also run the following command to identify all of the additional

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
```

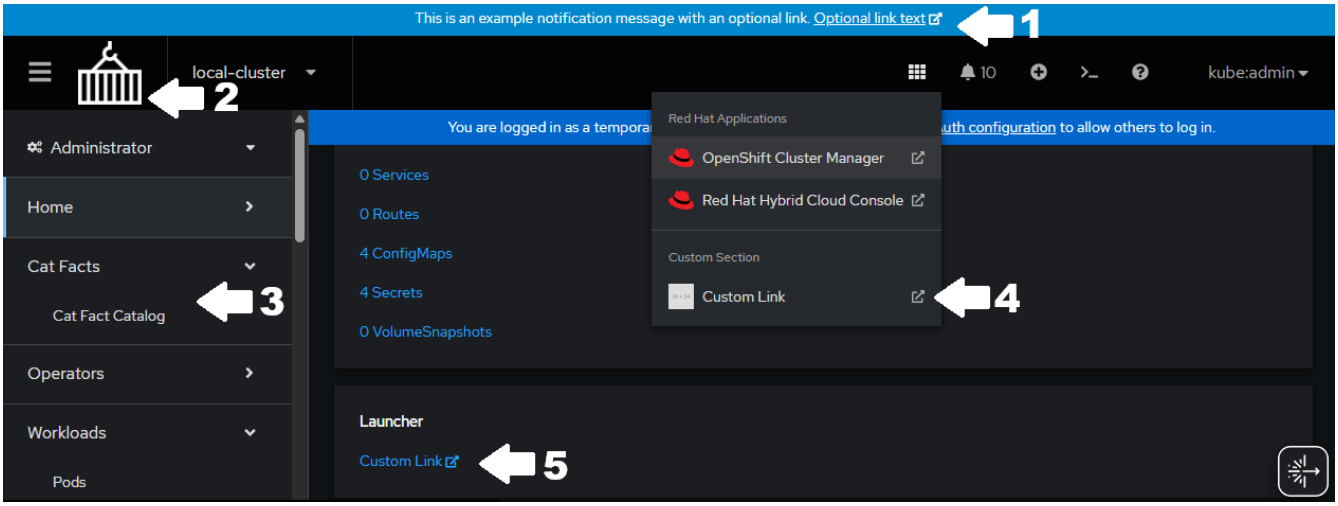
While it's true that namespaces isolate resources, they do not necessarily isolate the compute/memory/storage/network that underpins the platform. It's quite possible to degrade the performance of another tenant's resources by deploying resources in a separate namespace. `Quotas` will be introduced later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

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:

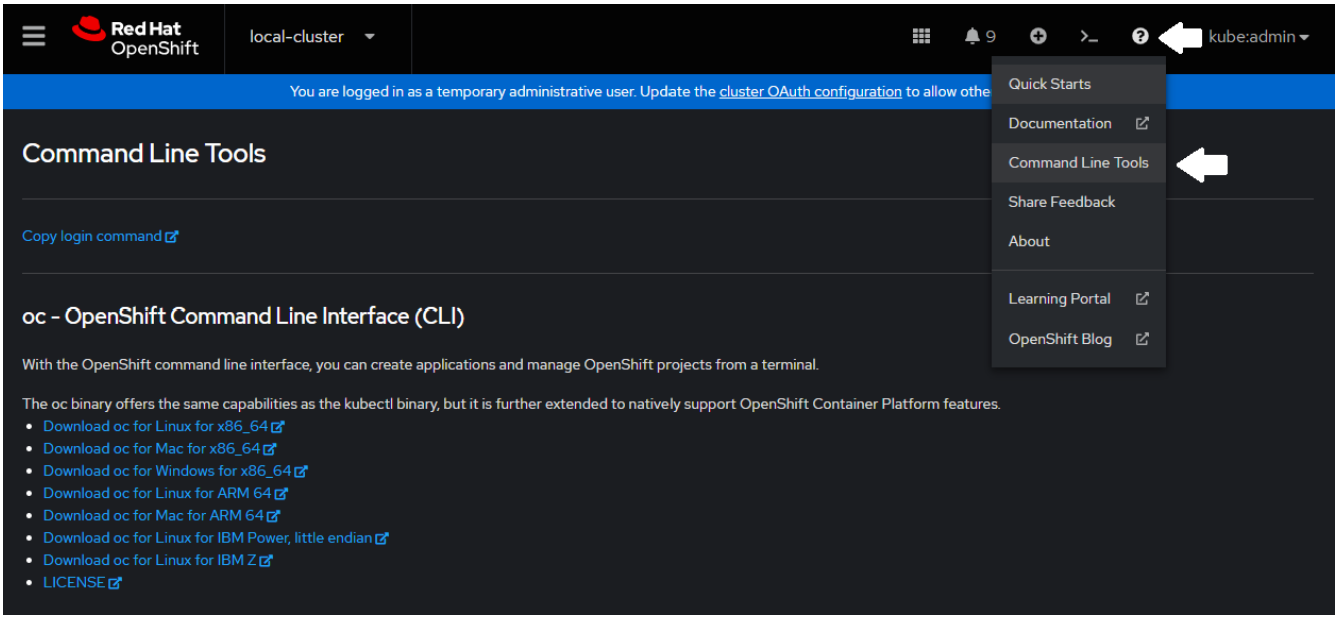


Figure 1. Download the CLI from the Console

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:

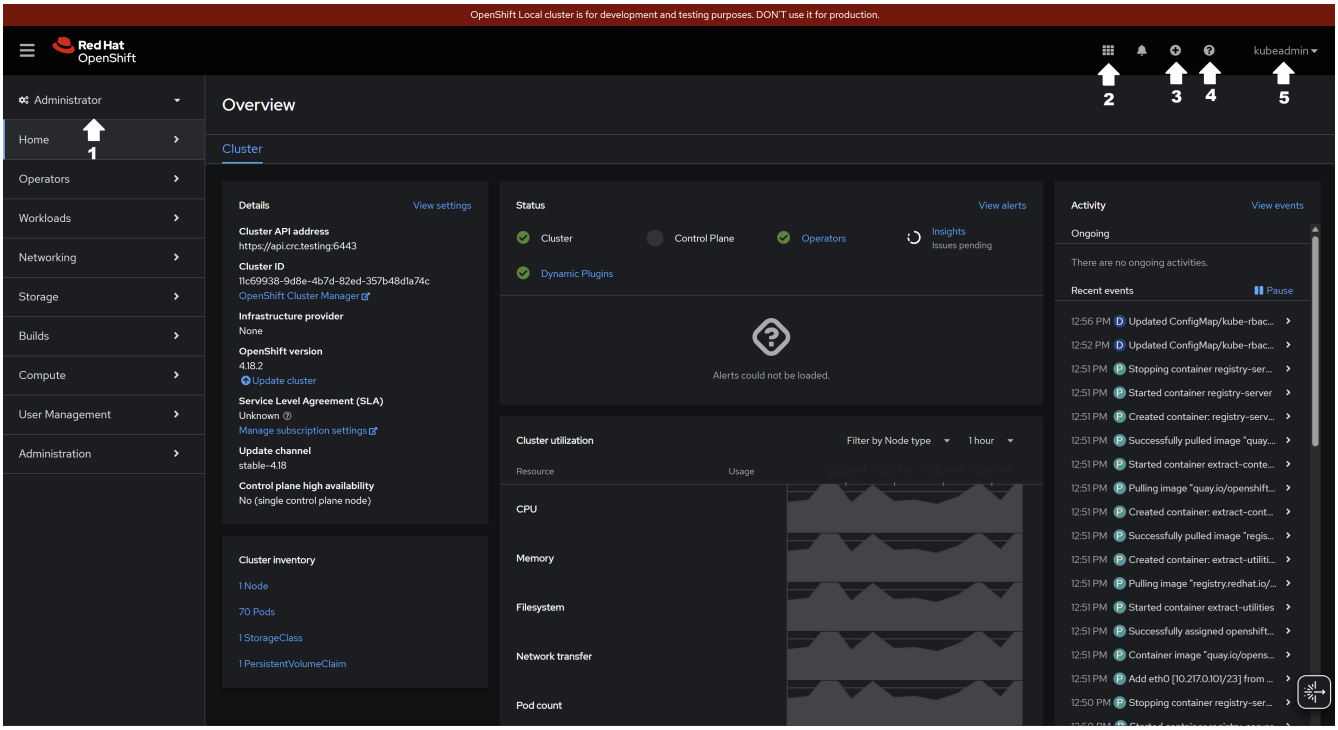


Figure 2. Administrator Console Basics

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

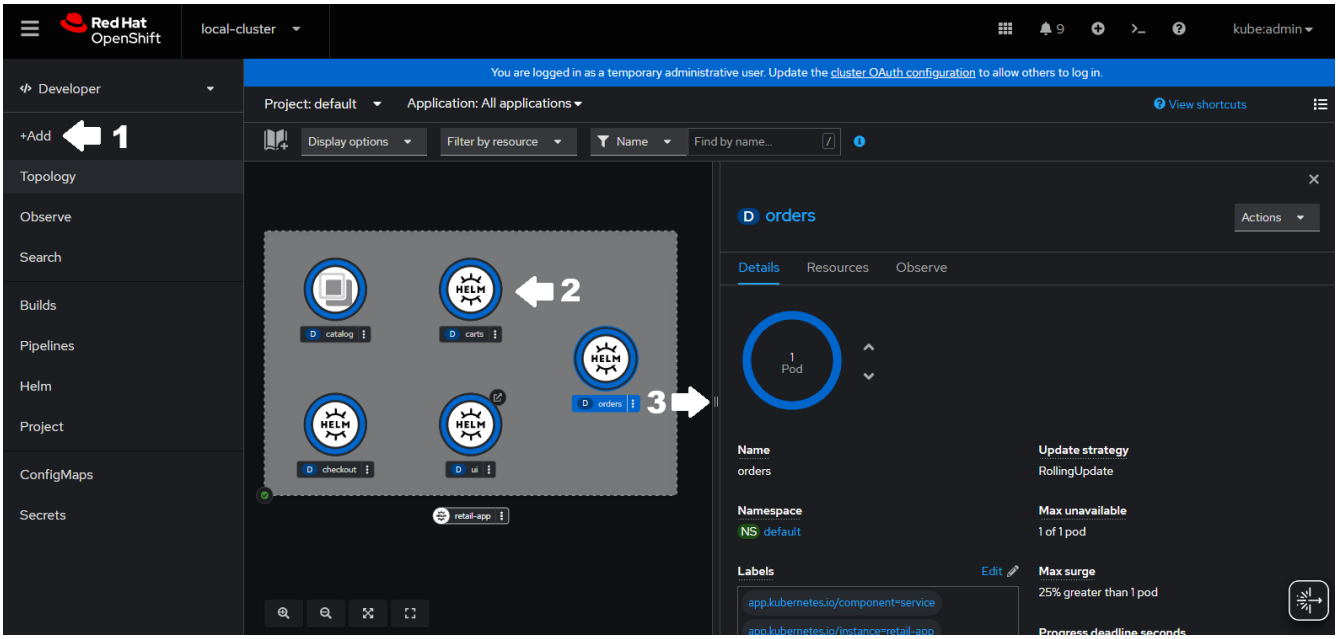


Figure 3. Developer Console Basics

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.

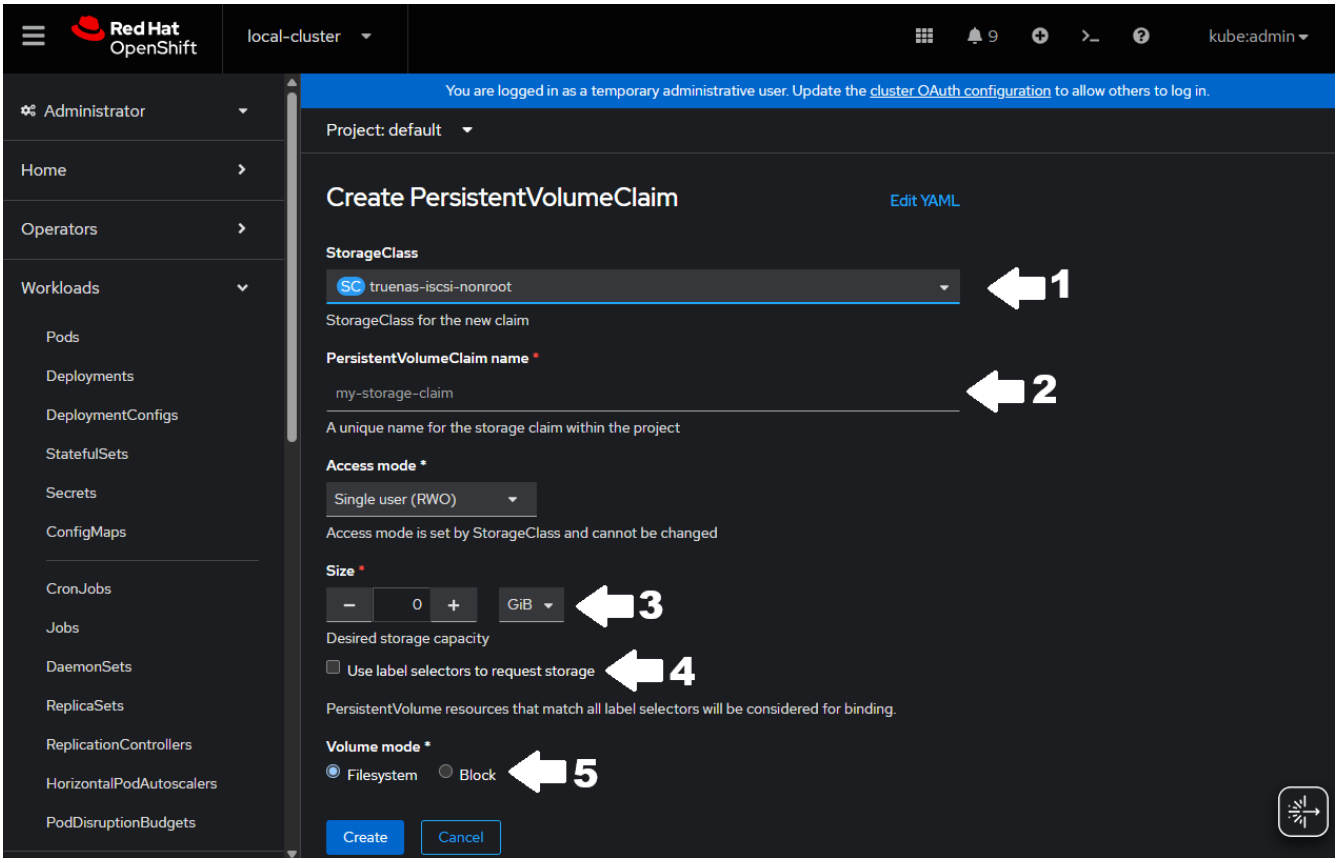


Figure 4. Form View

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

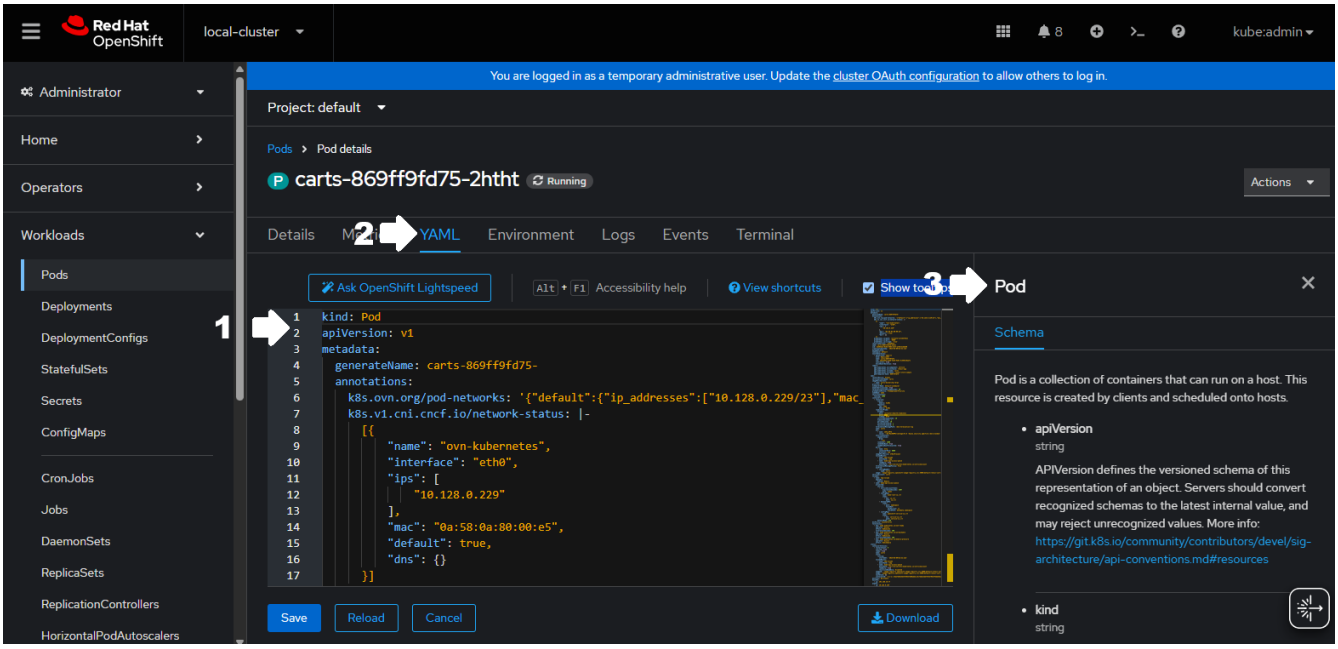


Figure 5. Editor View

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"
```

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
```

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.

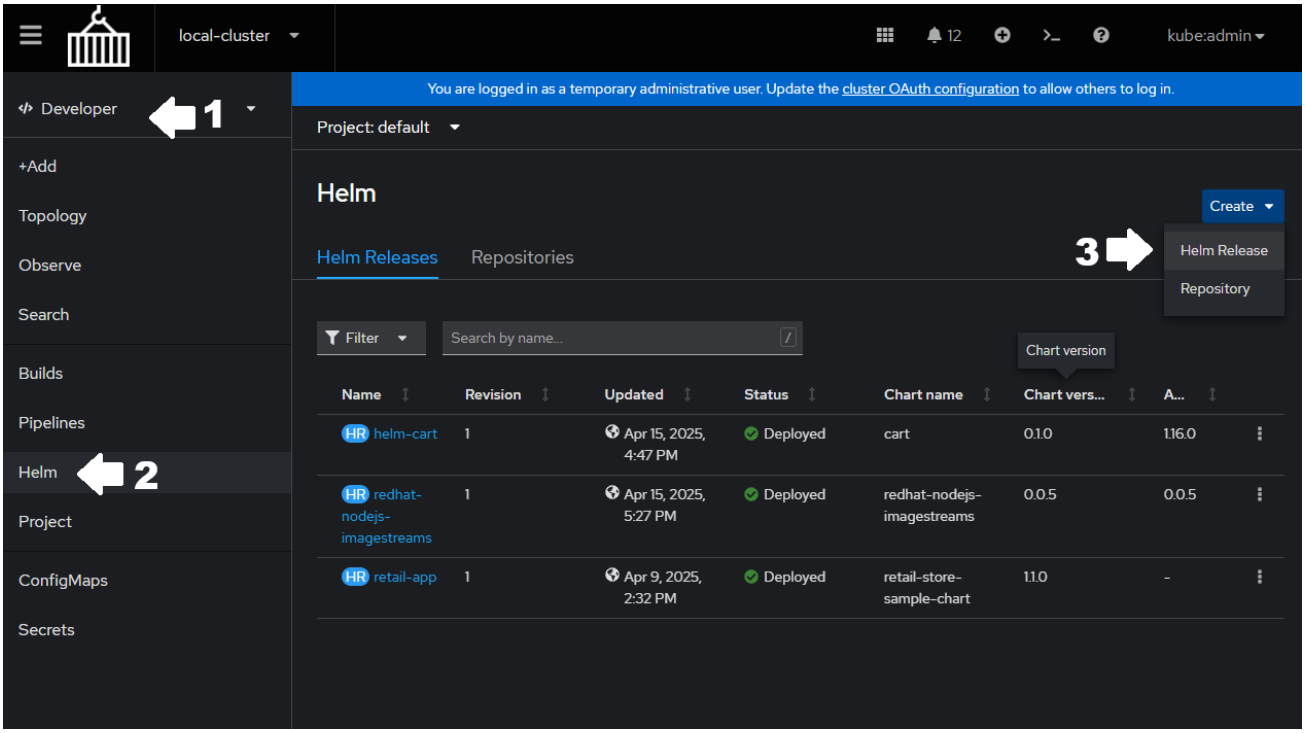
What images are included in the UBI catalog?

▼ *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:

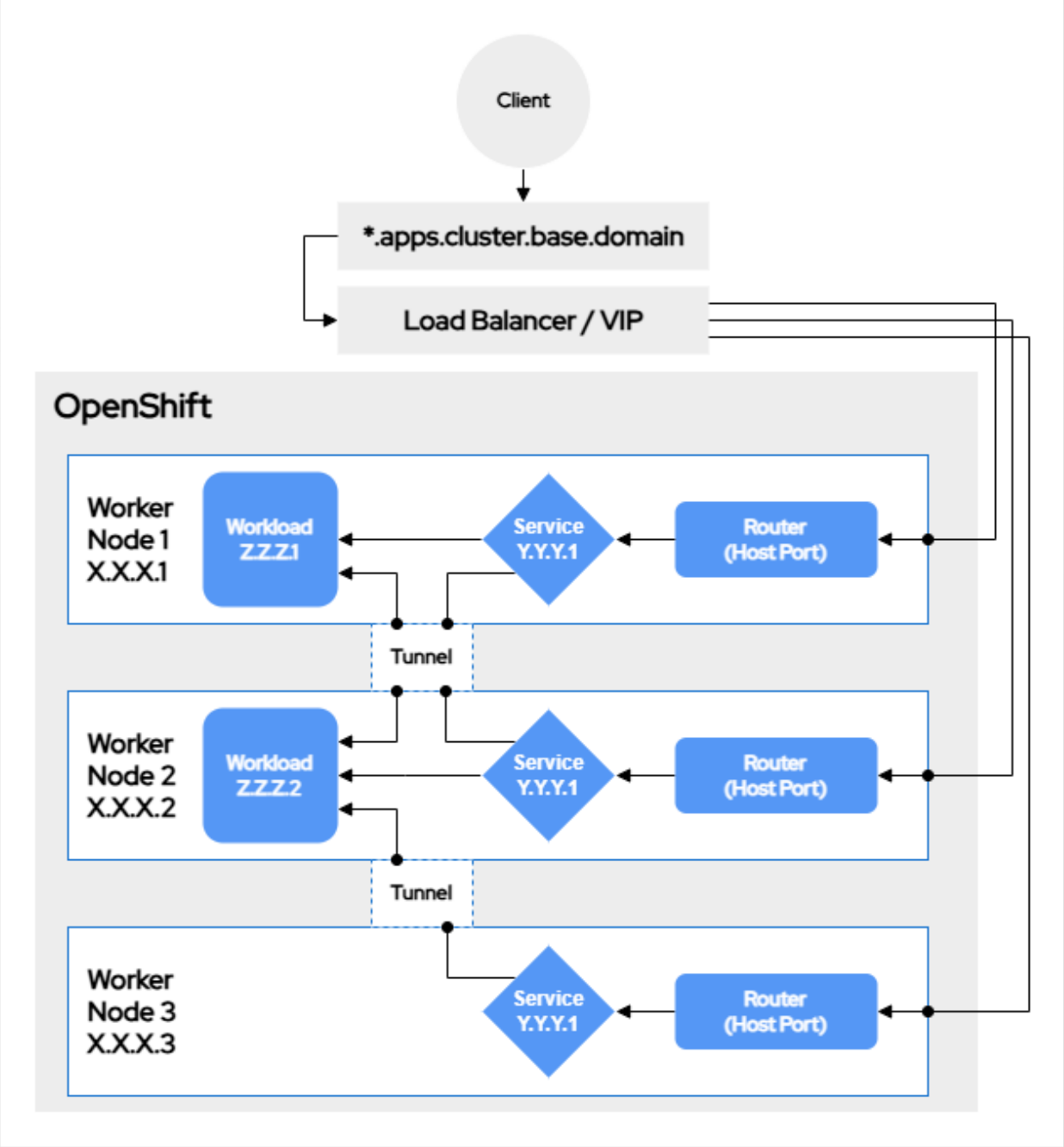


Figure 6. Diagram: Network Path into OpenShift

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`

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

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:

```
---
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
...
```

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"`)

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

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.
- **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 Options for Applications**:
  Distinguish between Deployments, DaemonSets, StatefulSets, Jobs, and CronJobs, and know when to use each based on workload behavior and lifecycle needs.
- **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.

Jump to Day One

Container Lifecycle

OpenShift manages the lifecycle of container images from initial pull to eventual removal. Understanding how images are handled helps ensure consistency, performance, and security in your applications.

Image Policy

OpenShift controls how and when images are pulled using its **image pull policy**, which is defined in the pod or deployment specification.

Common Policies

- `Always` – Pull the image every time the pod starts (default for `:latest` tag).
- `IfNotPresent` – Pull the image only if it's not already cached on the node.
- `Never` – Never pull the image; use only what's already present on the node.

Example:

```
spec:
  containers:
  - name: app
    image: myregistry.io/myimage:latest
    imagePullPolicy: Always
```

For predictable deployments, avoid `:latest` and prefer fixed tags or digests.

Exercise: Test Image Pull Behavior

1. Deploy a pod with `imagePullPolicy: Always` and update the image tag in the registry.
2. Re-deploy the pod and confirm the image is re-pulled.
3. Repeat using `IfNotPresent` and observe the difference.

External Registries

By default, OpenShift pulls from public or authenticated registries like Docker Hub or Quay.io. Pulling from **private or self-hosted registries** requires extra configuration.

Steps to Use an External Registry

1. Create a Kubernetes `Secret` with your registry credentials:

   ```
   oc create secret docker-registry my-registry-secret \
     --docker-server=registry.example.com \
     --docker-username=myuser \
     --docker-password=mypassword \
     --docker-email=user@example.com
   ```

2. Link the secret to your service account:

   ```
   oc secrets link default my-registry-secret --for=pull
   ```

Exercise: Use a Private Image

1. Set up a secret for an external registry.
2. Create a pod that pulls an image from the registry.
3. Confirm the image is pulled and the pod starts successfully.

Image Streams

OpenShift provides **ImageStreams**, a Kubernetes-compatible abstraction to track changes to images over time.

Pros

- Automatically detect new image versions and trigger builds/deployments.
- Integrates with BuildConfigs and DeploymentConfigs.
- Provides consistent naming and referencing.

Cons

- Adds complexity if you're already using standard Kubernetes tooling.
- Requires additional OpenShift-specific resources.

```
apiVersion: image.openshift.io/v1
kind: ImageStream
metadata:
  name: myapp
```

Exercise: Create and Use an ImageStream

1. Define an `ImageStream` and link it to a DeploymentConfig.
2. Push a new version of the image to the source registry.
3. Watch the DeploymentConfig auto-trigger a new rollout.

Tags and Digests

Images can be referenced by:

- **Tags** – human-readable labels like `:v1.0`, `:latest`
- **Digests** – immutable SHA256 hashes

Example digest reference:

```
image: myimage@sha256:ab1234...
```

Why Use Digests?

- Digests guarantee **immutability** — the image will never change.
- Tags can point to different versions over time, which may lead to drift or bugs.

Use digests for production deployments to ensure consistency.

Process: Image Lifecycle in OpenShift

The full lifecycle of an image inside OpenShift includes:

Figure 7. Diagram: Image Lifecycle

1. **Pull** – The image is downloaded to the node from a registry.
2. **Use** – The image is run inside pods.
3. **Cache** – The image is cached on the node for future reuse.
4. **Prune (Garbage Collection)** – Images not in use can be cleaned up:

```
oc adm prune images
```

You can configure image pruning based on age, usage, or tag history limits.

Exercise: View and Prune Images

1. Run:

```
oc adm top images
```

to view storage usage.

2. Simulate a prune operation:

```
oc adm prune images --confirm
```

Pruning should only be done by cluster admins with caution.

Managing Configuration

OpenShift applications often require configuration data that is external to the container image. This page explains how to use ConfigMaps and Secrets to provide environment-specific settings and how changes can be automatically reflected in running pods.

PersistentVolumes and PersistentVolumeClaims

ConfigMaps

A **ConfigMap** stores non-sensitive configuration data as key-value pairs. You can use ConfigMaps to:

- Set environment variables
- Mount configuration files into containers
- Decouple config from application code

Example ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: app-config
data:
  APP_MODE: production
  LOG_LEVEL: debug
```

Mounting the config as environment variables:

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

Or as files in a volume:

```
volumeMounts:
- name: config-volume
  mountPath: /etc/config
volumes:
- name: config-volume
  configMap:
    name: app-config
```

Exercise: Use a ConfigMap

1. Create a ConfigMap:

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

2. Create a pod that uses it as environment variables.
3. Use `oc exec` to shell into the pod and check the env values.

Secrets

A **Secret** is used to store sensitive information such as passwords, tokens, or keys. Like ConfigMaps, Secrets can be used as:

- Environment variables
- Volume-mounted files

Secrets are base64-encoded, not encrypted by default (encryption at rest must be enabled cluster-wide).

Example:

```
apiVersion: v1
kind: Secret
metadata:
  name: db-secret
type: Opaque
data:
  username: YWRtaW4=   # admin
  password: cGFzc3dvcmQ= # password
```

Referencing the secret:

```
env:
- name: DB_USER
  valueFrom:
    secretKeyRef:
      name: db-secret
      key: username
```

Exercise: Use a Secret

1. Create a secret with base64-encoded values or use the CLI:

```
oc create secret generic db-secret --from-literal=username=admin --from
```

```
-literal=password=password
```

2. Deploy a pod that uses the secret as env vars.
3. Confirm secret values are available inside the container using `echo $DB_USER`.

Automation: Updating Pods with New Configurations

By default, changes to ConfigMaps or Secrets **do not** automatically restart or reload running pods.

However, OpenShift supports **automatic update propagation** via volume mounts:

- When mounted as volumes, updated values are **eventually** reflected in the container (typically within a minute).
- Environment variable usage does **not** support automatic updates.

For changes to take effect immediately in env vars, pods must be manually restarted.

Best Practices

- Use volume mounts if you want in-place updates.
- Use `DeploymentConfig` triggers with ImageStreams or ConfigMap changes for full redeployment.
- Use `oc rollout restart` for manual restarts.

```
oc rollout restart deployment my-app
```

Exercise: Observe ConfigMap Update Propagation

1. Create a pod with a ConfigMap mounted as a volume.
2. Inside the pod, `cat` the config file.
3. Update the ConfigMap using:

```
oc create configmap app-config --from-literal=APP_MODE=debug -o yaml --dry
-run=client | oc apply -f -
```

4. Wait ~1 minute, then re-check the mounted file in the pod.
5. Optional: restart the pod and confirm the update happens immediately.

Scaling Applications

OpenShift enables you to scale your applications based on demand using built-in metrics and autoscaling tools. This section covers the fundamentals of application scaling, how resources are requested and limited, and how to automate scaling with metrics and autoscalers.

Metrics

OpenShift uses the **metrics server** to collect CPU and memory usage data from running pods. This information powers features like the web console's usage graphs and Horizontal Pod Autoscalers (HPAs).

You can view metrics using:

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

These commands show real-time resource consumption for pods and nodes.

Exercise: View Pod Metrics

1. Deploy a simple app (e.g., `oc new-app nginx`).
2. Wait for metrics collection to initialize (up to a few minutes).
3. Run:

   ```
   oc adm top pods
   ```

4. Generate traffic to the app using `curl` or a browser.
5. Run the command again and observe usage changes.

Resource Limits and Resource Requests

Every container in OpenShift should declare **resource requests and limits** to guide scheduling and ensure fair usage of cluster resources.

- **Requests**: Minimum guaranteed resources for the container.
- **Limits**: Maximum resources the container can consume.

Example:

```
resources:
  requests:
    memory: "64Mi"
    cpu: "250m"
  limits:
    memory: "128Mi"
    cpu: "500m"
```

If a pod exceeds its memory limit, it may be terminated. CPU limits, if exceeded, result in throttling.

Exercise: Set Resource Requests and Limits

1. Create a deployment with the `resources:` block defined as above.
2. Use `oc describe pod` to verify the requests/limits.
3. Generate load and observe behavior using `oc adm top pods`.

Horizontal Pod Autoscalers

OpenShift supports **Horizontal Pod Autoscalers (HPAs)** to automatically increase or decrease the number of pod replicas based on metrics.

Basic HPA configuration scales based on average CPU utilization.

Example:

```
oc autoscale deployment my-app --min=1 --max=5 --cpu-percent=50
```

This command creates an HPA that keeps CPU usage per pod around 50%.

Check the HPA status with:

```
oc get hpa
```

Exercise: Enable HPA

1. Deploy a sample app:

   ```
   oc new-app --name=hpa-demo quay.io/openshifttest/hello-openshift
   ```

2. Expose it and enable autoscaling:

   ```
   oc expose svc/hpa-demo
   oc autoscale deployment hpa-demo --min=1 --max=5 --cpu-percent=10
   ```

3. Generate load (e.g., using `curl` in a loop).
4. Watch for scaling:

   ```
   watch oc get pods
   ```

Custom Metrics for Horizontal Pod Autoscalers

In addition to CPU and memory, OpenShift supports **custom metrics** through integration with tools like Prometheus.

You can configure HPA to use:

- Application-specific metrics (e.g., request count, queue depth)
- External metrics via adapters (e.g., Prometheus Adapter)

This allows scaling based on what really matters to your app, not just resource usage.

Custom metrics HPA example:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: queue-autoscaler
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-queue-app
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Pods
    pods:
      metric:
        name: queue_length
      target:
        type: AverageValue
        averageValue: "10"
```

Exercise (Advanced): Explore Custom Metrics

1. Confirm Prometheus Adapter is available (your instructor or admin may have set it up).
2. Check what custom metrics are exposed:

   ```
   kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1" | jq .
   ```

3. Review an existing custom-metric HPA YAML and discuss how it works.

Debugging Applications

OpenShift provides several tools and techniques to help developers and administrators identify and resolve issues with running applications. This section walks through the general debugging approach and then dives into tools for debugging processes, networking, and storage.

Debug Process

When something goes wrong with an application running on OpenShift, follow a systematic

debugging approach:

1. **Check the Pod status**:

```
oc get pods
```

2. **Inspect Events** for failures or restarts:

```
oc describe pod "pod-name"
```

3. **Check Logs**:

```
oc logs "pod-name"
```

4. **Open a Remote Shell** to investigate interactively:

```
oc rsh "pod-name"
```

These steps help isolate whether the issue is in the application, container, environment, or platform.

Exercise: Basic Debug Workflow

1. Deploy a known working app (e.g., `nginx`).
2. Break it by misconfiguring the image or command.
3. Use the steps above to investigate the failure.

Debug Pod

OpenShift includes a powerful feature: the `oc debug` command. This launches a new **debug pod** based on the original pod's configuration, giving you root access and debugging tools.

```
oc debug pod/"pod-name"
```

By default, this creates a new pod with a debugging image (like `tools`), mounts the same volumes, and starts a shell.

You can also debug a node:

```
oc debug node/"node-name"
```

This launches a privileged container running on the target node.

Exercise: Use a Debug Pod

1. Create a pod (e.g., `oc run mypod --image=nginx`).
2. Use:

```
oc debug pod/mypod
```

3. From the debug shell, list files, test networking, or inspect mounts:

```
ls /etc
env
ping google.com
```

Debug Network

Network issues are common in distributed systems. OpenShift offers several tools for network debugging:

- Use `oc rsh` or `oc debug` to enter the container and run tools like `curl`, `ping`, or `dig`.
- Check `NetworkPolicies` that might block traffic.
- Validate DNS resolution:

```
dig myservice.myproject.svc.cluster.local
```

Example: Test connectivity between pods:

```
curl http://myservice:8080
```

Use `tcpdump` in a debug pod for deep packet inspection (requires tools image).

Exercise: Network Debug

1. Deploy two pods in the same project.
2. From one pod, try to reach the other using its service name and IP.
3. If using NetworkPolicies, add one that blocks access and test again.
4. Debug with `oc exec`, `curl`, and `dig`.

Debug Storage

Storage issues often manifest as read/write errors, stuck pods, or failed mounts.

Checklist:

- Inspect pod events for mount errors:

```
oc describe pod "pod-name"
```

- Use `oc debug` to inspect mounted volumes:

```
mount | grep pvc
ls /mnt/data
```

- Look for file permission errors or full volumes.
- If using PVCs, check their status:

```
oc get pvc
```

Some storage issues require checking backend logs (e.g., NFS, Ceph, etc.)—consult your storage admin if needed.

Exercise: Debug a PVC

1. Create a pod that mounts a PVC.
2. Use `oc debug` to explore the mounted volume.
3. Simulate an error by trying to write to a read-only volume.
4. Check permissions and volume status.

Deployment Strategies

OpenShift supports a variety of workload types and deployment techniques to suit different application needs. This section introduces the most common workload types and explains how to implement and tune health checks for reliable operation.

Workload Types

OpenShift supports several Kubernetes workload types, each suited to different deployment scenarios:

Deployment

- The most common workload type.
- Supports rolling updates, rollbacks, and scaling.
- Ideal for **stateless web apps, APIs**, and microservices.

```
oc create deployment my-app --image=nginx
```

StatefulSet

- Designed for **stateful applications** that need stable network identity and persistent storage (e.g., databases).
- Pods are created in a strict order and maintain their names.

```
kind: StatefulSet
metadata:
  name: my-db
```

DaemonSet

- Ensures a pod runs on **every node** (or a subset).
- Ideal for **log collectors, monitoring agents**, or infrastructure tools.

```
kind: DaemonSet
metadata:
  name: node-exporter
```

Job / CronJob

- Run once (`Job`) or on a schedule (`CronJob`).
- Ideal for **batch tasks**, **backups**, or **data processing**.

```
oc create job --image=busybox run-once -- echo "hello world"
```

Health Checks

OpenShift uses **probes** to monitor application health and availability. These allow the platform to take automated action when a container becomes unhealthy.

Liveness Probe

- Checks if the app is **still running**.
- If it fails, the container is restarted.

```
livenessProbe:
  httpGet:
    path: /health
    port: 8080
  initialDelaySeconds: 10
  periodSeconds: 5
```

Readiness Probe

- Checks if the app is **ready to serve traffic**.
- If it fails, the pod is removed from the service endpoint list.

```
readinessProbe:
  httpGet:
    path: /ready
    port: 8080
  initialDelaySeconds: 5
  periodSeconds: 3
```

Startup Probe

- Used for **slow-starting applications**.
- Overrides liveness probe until it succeeds.

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

Exercise: Configure Probes

1. Deploy a sample app (or reuse an existing one).
2. Add `livenessProbe`, `readinessProbe`, and `startupProbe` fields to its deployment spec.
3. Use `oc describe pod` to see probe status.
4. Simulate failure by modifying the app to return errors and observe pod behavior.

Observability

Observability is the ability to measure the internal state of a system based on the data it produces—such as metrics and logs. OpenShift provides built-in support for monitoring and logging user workloads through the OpenShift Console and CLI.

User Workload Monitoring

OpenShift provides Prometheus-based monitoring for user-deployed applications in addition to platform components. This feature is **opt-in** and must be enabled explicitly.

Enabling Monitoring

To enable user workload monitoring:

1. Create or modify the `cluster-monitoring-config` ConfigMap in the `openshift-monitoring` namespace:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-config
  namespace: openshift-monitoring
data:
  config.yaml: |
    enableUserWorkload: true
```

1. Apply the config:

```
oc apply -f cluster-monitoring-config.yaml
```

2. Wait a few moments for the `openshift-user-workload-monitoring` namespace to appear.

Exposing Metrics in a Pod

To expose metrics:

1. Your application must serve Prometheus-formatted metrics on an HTTP endpoint (e.g., `/metrics`).
2. Annotate the `Service` to be scraped:

```
metadata:
  annotations:
    prometheus.io/scrape: 'true'
    prometheus.io/port: '8080'
    prometheus.io/path: '/metrics'
```

Exercise: Monitor a Sample App

1. Deploy an app that exposes `/metrics` (e.g., Prometheus example apps).
2. Annotate the service.
3. Visit **Observe** → **Metrics** in the OpenShift web console.
4. Use the query bar to explore your metrics.

User Workload Logging

OpenShift can collect logs from containers and forward them to a central logging stack. Logs are collected using the **Collector** (e.g., Fluentd or Vector), stored in **Loki** or **Elasticsearch**, and visualized through **Kibana** or **Grafana Loki UI**.

This is typically enabled through the **OpenShift Logging Operator**.

Enabling Logging

1. Install the **OpenShift Logging** and **Elasticsearch Operators** (or Loki Operator for Loki stack).
2. Create a `ClusterLogging` custom resource to define the log collection pipeline:

```
apiVersion: "logging.openshift.io/v1"
kind: ClusterLogging
metadata:
  name: instance
  namespace: openshift-logging
spec:
  collection:
    logs:
      type: fluentd
  logStore:
    type: loki
  visualization:
    type: grafana
```

3. Logs for user workloads in the cluster are collected automatically once configured.

Viewing Logs

You can view logs in two main ways:

- **Web Console**: Go to **Observe → Logs**
- **CLI**: View logs per pod using:

  ```
  oc logs "pod-name"
  ```

Exercise: View Logs

1. Deploy any pod (e.g., `nginx`).
2. Use `oc logs` to view its output.
3. Navigate to the **Logs** section in the web console and filter by project or container.
4. Optionally, write a custom message to stdout and confirm it appears in the logs.