OKD: Kubernetes and OpenShift at a Glance

Issue/Rev. No.: 1.0
Date: 18/5/2026
Author: Massimo Costantini
Affiliation: INAF - Istituto Nazionale di Astrofisica
Osservatorio Astronomico di Trieste
Gruppo IA2
Approved by: Cristina Knapic

Abstract

This Technical Report documents the end-to-end deployment and operational setup of an OKD 4.21 cluster using the UPI (User Provisioned Infrastructure) method on Proxmox VE.

The first part of the document describes the installation process, including target architecture, infrastructure prerequisites, network topology, DNS configuration, HAProxy load balancing, ignition file generation and delivery, bootstrap sequencing, Machine Config Server routing, control-plane and worker node provisioning, certificate lifecycle considerations, and Proxmox-specific requirements.

The second part extends the installation report into a practical operations vademecum. It documents the post-installation configuration of the cluster, including dedicated worker networking for external Ceph connectivity, Ceph RBD integration through Ceph-CSI, StorageClass creation, persistent volume validation, HTPasswd-based authentication, project and group management, tenant onboarding, bastion-based user access, and backup procedures for Ceph-CSI manifests, authentication objects, tenant project resources, application manifests, and selected pre-update cluster state.

The report also provides practical application deployment examples, including a Python-based web application using persistent Ceph-backed storage, HTTPS exposure through OpenShift Routes, non-admin user workflows, parametrized application templates for multiple users, and validation procedures for persistence, routing, and project-level permissions.

A significant part of the document is intended as an operational command reference. Common administrative and user tasks are presented with both OpenShift commands and their Kubernetes equivalents where applicable. These include checking pods, nodes, services, routes, Kubernetes Ingress equivalents, persistent volumes, logs, deployments, rollouts, user permissions, scheduling behavior, node failures, workload relocation, and cluster update procedures.

Particular attention is given to troubleshooting and selected operational recovery scenarios encountered during and after installation, including bootstrap API failures, ignition retrieval problems, HAProxy and MCS routing issues, certificate expiration, CoreOS installer usage, Ceph-CSI adjustments, PVC binding behavior, OpenShift SCC requirements, tenant RBAC validation, and the expected behavior of workloads when a worker node becomes unavailable.

The report is intended as:

- A deployment guide for future OKD UPI installations on Proxmox VE
- An operations handbook for administrators and users of the deployed OKD cluster
- A troubleshooting and recovery reference for installation, storage, authentication, application deployment, and routine cluster operations

Contents

Part 1

Part 2

Part 3

Part 1

1 Introduction

OKD [1] is the community distribution of OpenShift [2]. It provides a Kubernetes-based [3] container platform enriched with additional components for cluster lifecycle management, integrated routing, authentication, security constraints, machine configuration, operator-based administration, and developer-oriented workflows.

Unlike compact Kubernetes distributions, OKD relies on a sophisticated bootstrap and lifecycle process involving:

- Temporary bootstrap control planes
- Ignition-driven node provisioning
- Machine Config Server[1] infrastructure
- HAProxy-based[2] API load balancing
- DNS-based[3] service discovery
- Cluster operators
- Integrated Ingress routing[4]
- OpenShift-specific authentication and authorization components

[1] The Machine Config Server (MCS) is the OpenShift component that serves machine configuration data to nodes during installation and machine configuration operations.

[2] HAProxy is a reverse-proxy offering high availability, load balancing, and proxying for TCP and HTTP-based applications.

[3] The Domain Name System (DNS) is the distributed naming service that translates human-readable hostnames into network addresses, enabling clients to locate services independently of their physical network location.

[4] Ingress routing manages external traffic access to services using HAProxy-based Ingress Controllers, which are managed by the Ingress Operator. It enables routing through native OKD Routes and standard Kubernetes Ingress resources, supporting HTTP/HTTPS traffic, wildcard DNS, and advanced features such as TLS passthrough and split traffic.

Deploying OKD using the UPI[5] model on Proxmox VE [4] requires careful coordination between:

- DNS
- HAProxy
- Ignition delivery
- Bootstrap lifecycle
- Certificate validity windows
- Node sequencing
- Proxmox virtual machine configuration
- Post-bootstrap load balancer reconfiguration

This report first describes a complete OKD 4.21 UPI deployment process validated on Proxmox VE using CentOS Stream CoreOS (SCOS)[6]. It documents the architecture, required virtual machines, DNS records, HAProxy configuration, ignition generation and delivery, bootstrap process, control-plane installation, worker node installation, validation steps, and troubleshooting procedures.

[5] User provisioned infrastructure (UPI) is an installation method for OKD where the user, rather than the installer, manually creates and manages the underlying infrastructure (virtual machines, networking, load balancers, and storage) before installing the cluster.

[6] CentOS Stream CoreOS (SCOS) is an immutable, container-focused Linux operating system built from CentOS Stream packages.

The scope of the document, however, is broader than the initial installation. After the cluster deployment, the report continues as an operations handbook for the resulting OKD environment. It documents how the installed system was configured and made usable for administrators and tenant users, including:

- Dedicated worker networking for external Ceph [5] connectivity
- Ceph RBD[7] integration through Ceph-CSI[8]
- Persistent storage validation
- HTPasswd-based OpenShift authentication
- Project, group, and user onboarding
- Bastion-based user access
- Tenant isolation through OpenShift groups and RBAC[9]
- Deployment of a sample Python web application with persistent storage
- HTTPS exposure through OpenShift Routes[10]
- Backup procedures for cluster, storage, authentication, and application manifests

The document also provides practical operating procedures and command references for routine cluster usage. Common tasks are shown with both OpenShift commands and Kubernetes equivalents where applicable, so that the reader can understand which operations are standard Kubernetes operations and which are OpenShift-specific extensions.

[7] Ceph RBD (RADOS Block Device) is a high-performance, distributed block storage component of the Ceph storage system that provides virtual block devices (like hard drives) to clients.

[8] Ceph-CSI is the official Container Storage Interface (CSI) driver that enables Kubernetes to use Ceph distributed storage systems (RBD block devices and CephFS file systems) for persistent storage.

[9] Role-Based Access Control (RBAC) is a security method that restricts system access to authorized users based on their job function or role within an organization. Instead of assigning permissions to individuals, administrators assign permissions to roles, and users are assigned to those roles.

[10] OpenShift Routes are a native way to securely expose applications running inside an OpenShift cluster to external users by providing a reachable hostname, such as www.example.com. Through the OpenShift ingress router, usually based on HAProxy, they route HTTP/HTTPS traffic from outside the cluster to specific internal Services and pods.

Examples include checking whether an application is running, identifying the node hosting a pod[11], inspecting logs and events, validating PVCs[12] OpenShift Routes and Kubernetes Ingress equivalents, restarting or scaling deployments, moving workloads away from a node, understanding what happens when a worker node fails, and performing cluster update checks and upgrades.

This report is therefore intended to serve three complementary purposes:

- A reproducible deployment guide for OKD 4.21 UPI installations on Proxmox VE
- A technical record of the validated installation and configuration work
- A practical vademecum for administrators and users operating the deployed OKD cluster

The procedures described in this document were validated on:

- Proxmox VE
- CentOS Stream CoreOS (SCOS)
- OKD 4.21
- HAProxy-based external load balancing
- External Ceph RBD storage integrated through Ceph-CSI

[11] A pod is the smallest, most basic deployable unit in a Kubernetes cluster, representing a single instance of a running process. It encapsulates one or more containers (like Docker), shared storage, shared networking (a unique IP address), and specifications on how to run the containers.

[12] PersistentVolumeClaims (PVCs) are API objects that allow developers and administrators to request persistent storage for applications running in containers or virtual machines (VMs).

2 Architecture Overview

The infrastructure components are:

Host	Role
bastion	Bastion host[13]
infra-1	HAProxy load balancer
bootstrap	Temporary bootstrap node
cp-1	Control plane[14]
cp-2	Control plane
cp-3	Control plane
wk-1	Worker[15]
wk-2	Worker
wk-3	Worker

The bastion host provides:

- DNS
- *`openshift-install`* execution
- Ignition file hosting via Python HTTP server

The bastion and infra-1 hosts are not Kubernetes cluster nodes and do not run cluster workloads, the bootstrap host is also not part of the final cluster, but it temporarily runs bootstrap services required to initialize the control plane during installation.

[13] Bastion host is a heavily secured, hardened server that acts as the single, controlled entry point into an OKD cluster, typically deployed in private, isolated networks.

[14] Control plane is a machine (virtual or physical) in an OKD cluster that runs the core Kubernetes services needed to manage the cluster's state, scheduling, and API. These "master" nodes run components like the API server, etcd, and schedulers, separating management tasks from user workloads.

[15] Worker is a machine (virtual or physical) in an OKD cluster responsible for running application containers, grouped as pods. Unlike control plane nodes that manage the cluster, worker nodes provide the compute power and resources for user applications, hosting containerized workloads.

HAProxy runs on infra-1 and provides:

- Kubernetes API[16] load balancing
- Machine Config Server routing
- Ingress routing

3 Infrastructure Requirements

The suggested VM sizing is:

Bootstrap, control plane and worker nodes

CPU	4 vCPU
RAM	16 GB
Disk	120 GB

In this deployment, infra-1 was sized similarly to the other cluster-support VMs for operational simplicity, although HAProxy itself has lower baseline resource requirements.

The host infra-1 is a dedicated infrastructure VM used to run HAProxy. It is not an OKD cluster node and does not run Kubernetes workloads. The -1 suffix was chosen to keep the naming scheme extensible: a future highly available load-balancer setup could introduce an additional node, such as infra-2, using Keepalived/VRRP[17] or another redundancy mechanism. In this validated deployment, only one HAProxy node was used.

On Proxmox VE, all VMs should use:

CPU Type
host

Using generic CPU models such as kvm64[18] may lead to kernel panics or unsupported instruction set failures.

[16] Kubernetes API is a resource-based (RESTful) programmatic interface provided via HTTP. It supports retrieving, creating, updating, and deleting primary resources via the standard HTTP verbs (POST, PUT, PATCH, DELETE, GET).

[17] Keepalived is a lightweight Linux software daemon that provides high availability (HA) and load balancing by implementing the Virtual Router Redundancy Protocol (VRRP).

[18] kvm64 is a default, generic virtual CPU model used in QEMU/KVM virtualization to ensure maximum compatibility for virtual machine migration across different physical hosts. It emulates a basic 64-bit x86 processor (similar to an older Pentium 4), presenting a limited set of instructions to the guest operating system.

4 DNS Configuration

The following records are required:

1. API endpoints

api.okd.ia2.inaf.it	A	<HAPROXY_IP>[19]
api-int.okd.ia2.inaf.it	A	<HAPROXY_IP>

Both records must point to the HAProxy host infra-1.

2. Application wildcard endpoint

*.apps.okd.ia2.inaf.it	A	<HAPROXY_IP>

This wildcard record must point to the HAProxy host infra-1 and is required by OpenShift Routes. It allows applications exposed through Routes to receive hostnames such as:

test-web-test.apps.okd.ia2.inaf.it

After creating the DNS records, verify name resolution from the bastion host or from any machine using the same DNS resolver as the OKD nodes:

```
dig api.okd.ia2.inaf.it

dig api-int.okd.ia2.inaf.it

dig test.apps.okd.ia2.inaf.it
```

The first two records must resolve to the HAProxy host used for the Kubernetes API and internal API endpoints. The third record is only a validation name. It does not need to exist as an individual DNS record. It should resolve through the wildcard record:

*.apps.okd.ia2.inaf.it

If the wildcard record is correctly configured, any hostname under *apps.okd.ia2.inaf.it*, such as *test.apps.okd.ia2.inaf.it* or *test-web-test.apps.okd.ia2.inaf.it*, should resolve to the HAProxy host.

[19] <HAPROXY_IP> represents the IP address of the HAProxy host infra-1. In a recommended production design, this address should belong to a private infrastructure network and should be reachable by the bastion host and OKD nodes. Public exposure should be avoided unless explicitly required and protected by appropriate firewalling and access controls.

3. Node records

bootstrap.okd.ia2.inaf.it	A	<BOOTSTRAP_IP>[20]
cp-1.okd.ia2.inaf.it	A	<CP_1_IP>[21]
cp-2.okd.ia2.inaf.it	A	<CP_2_IP>
cp-3.okd.ia2.inaf.it	A	<CP_3_IP>
wk-1.okd.ia2.inaf.it	A	<WK_1_IP>[22]
wk-2.okd.ia2.inaf.it	A	<WK_2_IP>
wk-3.okd.ia2.inaf.it	A	<WK_3_IP>

All cluster node IP addresses should normally be private. Control-plane nodes, worker nodes, the bootstrap node, HAProxy backends, and storage interfaces should remain reachable only from the internal infrastructure networks. Only the bastion host should be placed in the DMZ[23], providing controlled administrative access to the OKD environment.

In older OKD 4 UPI examples, etcd[24] DNS SRV records were commonly defined for the control-plane nodes. For OKD 4.4 and later, these records are no longer required by the official installation documentation. If they are used for compatibility with an existing DNS layout or operational convention, they should point to the three control-plane nodes on port 2380:

[20] <BOOTSTRAP_IP> represents the IP address assigned to the temporary bootstrap node. The bootstrap node is required only during the initial installation phase to initialize the control plane. After openshift-install wait-for bootstrap-complete succeeds, this address must be removed from the HAProxy backends and the bootstrap VM can be powered off or deleted.

[21] <CP_N_IP> represents the IP address assigned to a control-plane node, for example cp-1, cp-2, or cp-3. These addresses should preferably belong to a private infrastructure network and must be reachable by HAProxy, the bastion host, and the other OKD nodes. Control-plane nodes provide the Kubernetes API, etcd, and other core cluster services.

[22] <WK_N_IP> represents the IP address assigned to a worker node, for example wk-1, wk-2, or wk-3. Worker node addresses should preferably belong to a private infrastructure network and must be reachable by HAProxy, the bastion host, and the control-plane nodes. Worker nodes run application workloads and receive ingress traffic from HAProxy on ports 80 and 443.

[23] DMZ (Demilitarized Zone), or perimeter network, is a physical or logical subnetwork that acts as a secure buffer between a trusted internal network (LAN) and an untrusted external network, usually the internet. It exposes external-facing services, like web or email servers, while protecting the private network behind firewalls.

[24] etcd is the distributed key-value database used by Kubernetes and OpenShift to store the cluster state. The Kubernetes API server uses etcd to persist objects such as nodes, pods, deployments, services, secrets, configmaps, persistent volume claims, and RBAC resources. In an OKD cluster, etcd runs on the control-plane nodes and requires quorum to remain available.

_etcd-server-ssl._tcp.okd.ia2.inaf.it.	IN SRV 0 10 2380	cp-1.okd.ia2.inaf.it.
_etcd-server-ssl._tcp.okd.ia2.inaf.it.	IN SRV 0 10 2380	cp-2.okd.ia2.inaf.it.
_etcd-server-ssl._tcp.okd.ia2.inaf.it.	IN SRV 0 10 2380	cp-3.okd.ia2.inaf.it.

Each SRV record uses priority 0, weight 10, port 2380, and one control-plane node as target. The SRV targets must resolve through A records.

5 HAProxy Architecture

In this UPI deployment, HAProxy runs on the dedicated infrastructure host infra-1 and provides the external load-balancing layer required by the OKD cluster.

The load balancer exposes stable endpoints for the Kubernetes API, the Machine Config Server, and application ingress traffic. DNS records such as *api.okd.ia2.inaf.it*, *api-int.okd.ia2.inaf.it*, and **.apps.okd.ia2.inaf.it* point to the HAProxy host, while HAProxy forwards traffic to the appropriate backend nodes depending on the requested port and installation phase.

The ports used are:

Port	**Purpose**
6443	Kubernetes API
22623	Machine Config Server
80	HTTP ingress
443	HTTPS ingress

Port 6443 is used by clients, cluster components, and nodes to reach the Kubernetes API. During bootstrap, this backend includes the bootstrap node and the control-plane nodes. After bootstrap completion, the bootstrap node is removed and API traffic is forwarded only to the control-plane nodes.

Port 22623 is used by the Machine Config Server. This port is especially important during node provisioning because control-plane and worker nodes retrieve machine configuration data through this endpoint. In this deployment, during bootstrap, port 22623 was routed to the bootstrap node. After bootstrap completion, it was routed to the control-plane nodes.

Ports 80 and 443 are used for application ingress traffic. They forward HTTP and HTTPS traffic to the worker nodes where the OpenShift ingress router pods run. These ports are used by OpenShift Routes and provide external access to applications under the **.apps.okd.ia2.inaf.it* wildcard domain.

The HAProxy configuration must therefore be updated after bootstrap completion. At that point, bootstrap backend entries must be removed, and the Machine Config Server backend must point to the real control-plane nodes. Incorrect load-balancer routing, especially on port 22623, is one of the most common causes of failed UPI installations.

6 Preparing the Bastion Host

The bastion host is the administrative entry point used to prepare and operate the OKD installation. It is not an OKD cluster node and does not run Kubernetes workloads.

In this deployment, the bastion host is used to:

- Download the OKD installer, the OpenShift client, and the SCOS installation ISO
- Prepare the installation directory and the *install-config.yaml* file
- Run *openshift-install* to generate manifests and ignition files
- Host the generated ignition files through a temporary HTTP server
- Run administrative *oc* commands during and after installation
- Store installation artifacts, backup manifests, and operational procedures

The bastion host must have network connectivity to the HAProxy host, the bootstrap node, the control-plane nodes, the worker nodes, and any internal services required during installation. It should also be the controlled administrative access point to the environment, especially if the cluster nodes use private addresses.

The following command installs the required packages:

```
sudo dnf install -y podman jq curl wget bind-utils tar gzip
```

podman is useful for inspecting bootstrap containers, *jq* for processing JSON output, *cur*l and *wge*t for endpoint validation and downloads, *bind-utils* for DNS checks such as *dig*, and *tar/gzip* for extracting OKD client and installer archives.

7 Downloading OKD Components

The OKD installer, the OpenShift client tools, and the CentOS Stream CoreOS installation ISO were downloaded on the bastion host.

The installer archive provides the *openshift-install* binary, which is used to generate manifests and ignition files and to monitor the bootstrap and installation phases.

The client archive provides the *oc* and *kubectl* command-line tools used to interact with the cluster during and after installation.

The SCOS ISO is used to boot the Proxmox VMs before installing CentOS Stream CoreOS on the VM disks with the appropriate ignition file.

For this deployment, OKD 4.21.0 on SCOS was used:

```
export OKD_VERSION=4.21.0-okd-scos.0

mkdir -p /opt/okd/downloads

cd /opt/okd/downloads

wget
"https://github.com/okd-project/okd/releases/download/${OKD_VERSION}/op
enshift-install-linux.tar.gz"

wget
"https://github.com/okd-project/okd/releases/download/${OKD_VERSION}/op
enshift-client-linux.tar.gz"

wget
"https://github.com/okd-project/okd/releases/download/${OKD_VERSION}/ok
d-scos-4.21.iso"
```

After downloading the archives, extract the installer and client tools, then install the *openshift-install*, *oc*, and *kubectl* binaries in */usr/local/bin*:

```
tar -xzf openshift-install-linux.tar.gz
```

```
tar -xzf openshift-client-linux.tar.gz
```

```
sudo install -m 0755 openshift-install /usr/local/bin/openshift-install
```

```
sudo install -m 0755 oc /usr/local/bin/oc
```

```
sudo install -m 0755 kubectl /usr/local/bin/kubectl
```

To verify the installed binaries:

```
openshift-install version
```

```
oc version --client
```

```
kubectl version --client
```

The *openshift-install* binary is release-specific. Do not mix an installer from one OKD release with release images or installation assets from another release.

8 Creating install-config.yaml

The *install-config.yaml* file is the main input file used by *openshift-install* to define the desired cluster configuration before ignition files are generated. It describes the cluster identity, base DNS domain, control-plane and worker node layout, network ranges, platform type, pull secret[25], and SSH key[26] that will be embedded into the generated ignition configuration.

In a UPI installation, this file does not create virtual machines automatically. Instead, it defines the logical OKD cluster configuration, while the administrator remains responsible for provisioning the infrastructure manually on Proxmox VE, configuring DNS, setting up HAProxy, and booting each node with the appropriate ignition file.

After the manifests and ignition files are generated, *install-config.yaml* is consumed by the installer and should be considered part of the installation backup material. For reproducibility, a sanitized copy should be stored securely, especially if it contains sensitive fields such as the pull secret or SSH key.

[25] The pullSecret field contains the registry authentication information used by OKD to pull the container images required during installation and cluster operation. It is provided as a JSON object and may include credentials for registries such as quay.io. Because it contains authentication material, it must be treated as sensitive information and should not be committed to Git or exposed in documentation.

[26] The sshKey field contains the public SSH key that is embedded into the generated ignition files and installed on the OKD nodes. It allows administrators to access the CentOS Stream CoreOS nodes, usually as the core user, for troubleshooting and recovery operations. Only the public key should be included in install-config.yaml; the corresponding private key must be stored securely and never committed to Git or included in documentation.

```
apiVersion: v1
baseDomain: ia2.inaf.it
metadata:
  name: okd
compute:
- architecture: amd64
  hyperthreading: Enabled
  name: worker
  replicas: 0
  platform: {}
controlPlane:
  architecture: amd64
  hyperthreading: Enabled
  name: master
  replicas: 3
  platform: {}
networking:
  networkType: OVNKubernetes[27]
  clusterNetwork[28]:
  - cidr: 10.128.0.0/14
    hostPrefix: 23
  serviceNetwork:
  - 172.30.0.0/16
```

[27] OVNKubernetes is the OpenShift network plugin based on OVN, Open Virtual Network. It provides pod networking, service connectivity, network policy enforcement, and the virtual networking layer used by the OKD cluster.

[28] clusterNetwork defines the internal CIDR range used for pod IP addresses. serviceNetwork defines the internal CIDR range used for Kubernetes Service ClusterIP addresses. hostPrefix controls the size of the pod subnet assigned to each node. These networks are internal to Kubernetes/OKD and must not overlap with the physical infrastructure, Proxmox, Ceph, or management networks.

```
platform:
  none: {}
fips: false
pullSecret:
'<REDACTED_PULL_SECRET_JSON> '
sshKey: '<SSH_PUBLIC_KEY>'
```

9 Generating Manifests and Ignition Files

After *install-config.yaml* has been prepared, the next step is to let the OKD installer generate the Kubernetes manifests and the ignition files required to bootstrap the cluster.

The manifests represent the initial cluster configuration that will be applied during installation. They include cluster-level objects, machine configuration definitions, networking configuration, operator configuration, and other resources required by the bootstrap process.

The ignition files are the machine initialization files used by CentOS Stream CoreOS during the first boot. Each node role receives a different ignition file:

- *bootstrap.ign* is used only by the temporary bootstrap node
- *master.ign* is used by the control-plane nodes
- *worker.ign* is used by the worker nodes

In a UPI installation, these ignition files are not automatically injected by a cloud provider. They must be manually provided to the corresponding Proxmox virtual machines during the CoreOS installation process:

```
mkdir -p okd-install
```

```
cp install-config.yaml okd-install/
```

To create the manifests:

```
openshift-install --dir okd-install create manifests
```

To generate the ignition files:

```
openshift-install --dir okd-install create ignition-configs
```

Expected output:

```
bootstrap.ign
master.ign
worker.ign
```

10 Hosting Ignition Files

In a UPI installation, ignition files are not automatically injected by an infrastructure provider. They must be made reachable by the nodes during the CentOS Stream CoreOS installation process.

In this deployment, the ignition files generated by *`openshift-install`* were temporarily served from the bastion host using the Python built-in HTTP server. This method is simple and sufficient for a controlled installation environment, provided that the bootstrap, control-plane, and worker nodes can reach the bastion host over the network.

From the installation directory containing *`bootstrap.ign`*, *`master.ign`*, and *`worker.ign`*, start the temporary HTTP server:

```
cd /path/to/okd-install
```

```
python3 -m http.server 8080
```

This serves ignition files over HTTP.

11 Installing the Bootstrap Node

The bootstrap node is a temporary machine used only during the initial OKD installation phase.

Its purpose is to start the first control-plane services, expose the temporary Kubernetes API, provide the initial Machine Config Server workflow, and coordinate the creation of the real control-plane nodes.

The bootstrap node is not part of the final cluster. After the bootstrap process completes successfully, it must be removed from the HAProxy backends and can be powered off and deleted.

In a UPI installation, the bootstrap VM is not created automatically by the OKD installer. It must be created manually in the Proxmox VE environment before the SCOS installation can begin.

Create a new VM in Proxmox VE for the bootstrap node using the sizing defined in the infrastructure requirements section. Attach the SCOS ISO to the VM as a virtual CD/DVD device, configure the VM to boot from the ISO, and start it from the Proxmox console.

At this stage, the VM is still only a generic Proxmox virtual machine booted into the SCOS live environment. It becomes the OKD bootstrap node only after CentOS Stream CoreOS is installed on its disk using the *`bootstrap.ign`* ignition file.

Attach the ISO:

```
okd-scos-4.21.iso
```

From the bootstrap console download the ignition file:

```
curl http://<BASTION_IP>[29]:8080/bootstrap.ign -o /tmp/bootstrap.ign
```

To install CentOS Stream CoreOS:

```
sudo coreos-installer install /dev/sda
    -i /tmp/bootstrap.ign --copy-network
```

[29] <BASTION_IP> represents the IP address of the bastion host. In a recommended deployment, the bastion is the only host exposed for controlled administrative access, while OKD nodes and internal services remain on private infrastructure networks. It must be able to reach HAProxy, the bootstrap node, control-plane nodes, worker nodes, DNS, and any required internal services.

After installation, reboot the VM:

```
reboot
```

12 Bootstrap Validation

After the bootstrap VM has been installed and rebooted from disk, the bootstrap services should start automatically on the node. During this phase, the bootstrap node temporarily hosts the components required to initialize the real control plane.

The goal of this validation step is to confirm that the bootstrap containers are running, that the Kubernetes API is reachable through HAProxy, and that the installer can observe the bootstrap process until completion.

Check running containers:

```
sudo podman ps
```

Expected result:

```
cluster-bootstrap
```

Check API availability from the bastion host:

```
curl -k https://api.okd.ia2.inaf.it:6443/version
```

If the API is reachable, the command should return Kubernetes version information in JSON format.

Wait for bootstrap completion from the bastion host, using the same installation directory used to generate the manifests and ignition files:

```
openshift-install --dir okd-install wait-for bootstrap-complete
```

Expected result:

```
It is now safe to remove the bootstrap resources
```

Bootstrap completion means that the temporary bootstrap node has finished its role and that the real control-plane nodes have taken over the required control-plane services. At this point, the HAProxy configuration can be updated to remove the bootstrap node from the API backend and to route the Machine Config Server endpoint to the control-plane nodes.

13 Installing Control Plane Nodes

After the bootstrap node is running and the API and Machine Config Server endpoints are reachable through HAProxy, the control-plane nodes can be installed.

In a UPI installation, the control-plane VMs are not created by the OKD installer. They must be created manually in Proxmox VE using the sizing defined in the infrastructure requirements section. In this deployment, three control-plane VMs were created:

- cp-1
- cp-2
- cp-3

Each VM is booted from the SCOS ISO and installed to disk using the same *master.ign* ignition file. The role of the node is determined by the ignition file, so all control-plane nodes use *master.ign*.

From the SCOS live environment of each control-plane VM, download the ignition file from the bastion host:

```
curl http://<BASTION_IP>:8080/master.ign -o /tmp/master.ign
```

Install CentOS Stream CoreOS on the VM disk:

```
sudo coreos-installer install /dev/sda
    -i /tmp/master.ign
    --copy-network
```

The *--copy-network* option preserves the network configuration from the live environment and copies it into the installed system. This is important in a Proxmox UPI installation because the node must keep network connectivity after reboot in order to reach the API and Machine Config Server endpoints.

After installation, reboot the VM:

```
reboot
```

Repeat the same procedure for cp-1, cp-2, and cp-3.

After reboot, the control-plane nodes should contact the Machine Config Server through *api-int.okd.ia2.inaf.it:22623*, retrieve their machine configuration, start kubelet, and join the forming control plane.

14 Machine Config Server (MCS)

The Machine Config Server is the service that provides machine configuration and ignition-related data to OKD nodes during installation and during later machine configuration operations.
In an OKD installation, nodes do not simply boot as generic Linux machines. They must receive role-specific configuration that tells them how to join the cluster as control-plane or worker nodes. This configuration includes machine config data, certificates, cluster connection information, kubelet configuration, and other bootstrap material required for the node to become part of the cluster.

The MCS is exposed on TCP port 22623.

During the bootstrap phase, the MCS endpoint must point to the temporary bootstrap node:

```
22623 -> bootstrap
```

This is because, at the beginning of the installation, the real control-plane nodes are not yet fully available. The bootstrap node temporarily provides the services needed by the first control-plane machines to initialize correctly.

After the bootstrap process completes, the bootstrap node is no longer needed. At that point, the MCS endpoint must be moved away from the bootstrap node and must point to the real control-plane nodes:

```
22623 -> control plane nodes
```

In this UPI deployment, this routing is handled manually through HAProxy.

This means that HAProxy must be configured in two distinct phases:

1. Bootstrap phase

Port 22623 must include the bootstrap node, because during the early installation phase the bootstrap node provides the Machine Config Server endpoint required by the first control-plane nodes.

In this deployment, the MCS backend was intentionally routed only to the bootstrap node during the bootstrap phase.

Some OKD UPI configurations may also include the control-plane nodes in the 22623 backend during this phase, but the bootstrap node must be present and reachable.

2. Post-bootstrap phase

After bootstrap completion, the bootstrap node must be removed from the HAProxy configuration, and port 22623 must route to the real control-plane nodes.

At this point, the bootstrap node is no longer required and can be powered off and removed.

Incorrect MCS routing is one of the most common causes of failed OKD UPI installations. If the control-plane or worker nodes cannot reach the MCS endpoint, they may fail to retrieve their configuration, fail to start kubelet correctly, or remain stuck during the join process.

A quick validation command from the bastion or another machine with network access is:

```
curl -k https://api-int.okd.ia2.inaf.it:22623/config/master
```

During the appropriate installation phase, this command should return machine configuration data rather than timing out or failing with a connection error.
For worker configuration, the equivalent endpoint is:

```
curl -k https://api-int.okd.ia2.inaf.it:22623/config/worker
```

If these checks fail, verify:

- DNS resolution for *api-int.okd.ia2.inaf.it*
- HAProxy frontend binding on port 22623
- HAProxy backend target for the current installation phase
- Firewall rules between nodes and HAProxy
- Bootstrap node availability during the bootstrap phase
- Control-plane node availability after bootstrap completion

15 HAProxy Configuration for Bootstrap Phase

During the bootstrap phase, HAProxy is responsible for exposing the OKD API and Machine Config Server endpoints through stable DNS names.

In this deployment, both *api.okd.ia2.inaf.it* and *api-int.okd.ia2.inaf.it* resolve to the HAProxy host.
HAProxy then forwards traffic to the correct backend nodes depending on the requested port.

The two critical ports during installation are:

- 6443 for Kubernetes API, bootstrap and control-plane nodes
- 22623 for Machine Config Server, bootstrap node only

During the early installation phase, the bootstrap node temporarily provides services required to initialize the real control-plane nodes. In this deployment, port 22623 for the Machine Config Server was routed only to the bootstrap node during bootstrap. Other UPI configurations may include different backend combinations during this phase, depending on the documented load-balancer pattern and the installation stage. The critical requirement is that the Machine Config Server endpoint remains reachable by the nodes during the appropriate phase, and that bootstrap backend entries are removed after bootstrap completion.

This configuration must be changed after bootstrap completion, as described in the following sections.

```
global
    log /dev/log local0
    maxconn 2000
defaults
    log global
    mode tcp
    option dontlognull
    timeout connect 10s
    timeout client 1m
    timeout server 1m
frontend okd-api
    bind *:6443
    default_backend okd-api
backend okd-api
    balance roundrobin
    option tcp-check
    server bootstrap <BOOTSTRAP_IP>:6443 check
    server cp-1 <CP_1_IP>:6443 check
    server cp-2 <CP_2_IP>:6443 check
    server cp-3 <CP_3_IP>:6443 check
frontend okd-mcs
    bind *:22623
    default_backend okd-masters-mcs
backend okd-masters-mcs
    balance roundrobin
    option tcp-check
```

```
    server bootstrap <BOOTSTRAP_IP>:22623 check
frontend okd-ingress-http
    bind *:80
    default_backend okd-ingress-http
```

The ingress frontends can also be defined at this stage. They are used later by the OpenShift router to expose applications through Routes. In this deployment, ports 80 and 443 are forwarded to the worker nodes where the ingress router pods run.

```
backend okd-ingress-http
    balance source
    option tcp-check
    server wk-1 <WK_1_IP>:80 check
    server wk-2 <WK_2_IP>:80 check
    server wk-3 <WK_3_IP>:80 check
frontend okd-ingress-https
    bind *:443
    default_backend okd-ingress-https
backend okd-ingress-https
    balance source
    option tcp-check
    server wk-1 <WK_1_IP>:443 check
    server wk-2 <WK_2_IP>:443 check
    server wk-3 <WK_3_IP>:443 check
```

16 Bootstrap Completion

Bootstrap completion must be checked from the bastion host using the installation directory created earlier.

The bootstrap phase is complete when the temporary bootstrap node has successfully initialized the real control plane and the control-plane nodes are able to continue without it.

Wait for bootstrap completion:

```
openshift-install --dir okd-install wait-for bootstrap-complete
```

Expected output:

```
It is now safe to remove the bootstrap resources
```

This message means that the bootstrap node has completed its role and can be removed from the load-balancer backends. It does not mean that the entire cluster installation is complete. Worker nodes still need to be installed, pending CSRs[30] may need to be approved, and the final installation completion check must still be performed.

At this stage, the next required step is to update HAProxy:

- Remove the bootstrap node from the API backend on port 6443
- Move the Machine Config Server backend on port 22623 to the control-plane nodes
- Reload HAProxy and verify API and MCS reachability

Only after the load balancer has been updated should the worker nodes be installed.

Do not power off or delete the bootstrap VM before `wait-for bootstrap-complete` has completed successfully. Removing it too early can interrupt control-plane initialization.

[30] CertificateSigningRequest (CSR) is a Kubernetes object used by nodes and components to request certificates from the cluster certificate authority. During node bootstrap, new nodes may create CSRs that must be approved before the node can fully join the cluster.

17 Updating HAProxy After Bootstrap

After *openshift-install wait-for bootstrap-complete* reports that bootstrap is complete, the temporary bootstrap node has finished its role. At this point, the real control-plane nodes have taken over the required control-plane services, and the bootstrap node must be removed from the load-balancer configuration.

This HAProxy update is mandatory before continuing with the final installation steps and before relying on the cluster for normal operation.

The post-bootstrap changes are:

- Remove the bootstrap node from the Kubernetes API backend on port 6443
- Move the Machine Config Server backend on port 22623 from the bootstrap node to the control-plane nodes
- Reload or restart HAProxy after validating the configuration
- Power off or delete the bootstrap VM only after confirming that the load-balancer update is correct

The API backend should no longer include the bootstrap node:

```
backend okd-api

      balance roundrobin

      option tcp-check

      server cp-1 <CP_1_IP>:6443 check

      server cp-2 <CP_2_IP>:6443 check

      server cp-3 <CP_3_IP>:6443 check
```

The Machine Config Server backend should now point to the control-plane nodes:

```
backend okd-masters-mcs
      balance roundrobin
      option tcp-check
      server cp-1 <CP_1_IP>:22623 check
      server cp-2 <CP_2_IP>:22623 check
      server cp-3 <CP_3_IP>:22623 check
```

Before reloading HAProxy, validate the configuration file:

```
      sudo haproxy -c -f /etc/haproxy/haproxy.cfg
```

If the configuration is valid, reload HAProxy:

```
      sudo systemctl reload haproxy
```

If reload is not available in the environment:

```
      sudo systemctl restart haproxy
```

After the reload, verify that the API is still reachable:

```
      curl -k https://api.okd.ia2.inaf.it:6443/version
```

Verify that the Machine Config Server endpoint is reachable through the internal API name:

```
curl -k https://api-int.okd.ia2.inaf.it:22623/config/master
```

For worker node provisioning, also verify:

```
curl -k https://api-int.okd.ia2.inaf.it:22623/config/worker
```

Only after these checks succeed should the bootstrap VM be powered off or removed.

Incorrect post-bootstrap HAProxy routing can prevent worker nodes from joining the cluster or can break access to the Machine Config Server. For this reason, the load-balancer transition should be treated as a required installation step, not as a cleanup task.

After this step, the worker nodes can be installed using *`worker.ign`*.

18 Installing Worker Nodes

After bootstrap has completed and HAProxy has been updated so that port 22623 points to the control-plane nodes, install the worker nodes.
Repeat the same operations for:

- wk-1
- wk-2
- wk-3

Download the worker ignition file:

```
curl http://<BASTION_IP>:8080/worker.ign -o /tmp/worker.ign
```

Install CentOS Stream CoreOS on the worker disk:

```
sudo coreos-installer install /dev/sda
    -i /tmp/worker.ign
    --copy-network
```

Reboot the worker node:

```
reboot
```

After all worker nodes have been installed, any pending node CSRs have been approved, and the nodes have joined the cluster, wait for the installation to complete:

```
openshift-install --dir okd-install wait-for install-complete
```

Expected output:

```
INFO Waiting up to 40m0s for the cluster at https://api.okd.ia2.inaf.it:6443
to initialize...
INFO Checking to see if there is a route at openshift-console/console...
INFO Install complete!
```

19 Certificate Expiration Considerations

The ignition files generated by *openshift-install* contain bootstrap material and short-lived certificates used during the initial installation process. These certificates have a limited validity window, commonly described as 24 hours from ignition generation.

For this reason, ignition files should be generated shortly before the actual deployment and used without long delays. In practice, it is safer to complete the bootstrap and control-plane installation within a much shorter window, for example within 12 hours, to avoid failures related to certificate expiration or rotation during installation.

If ignition files are generated too early and used after their validity window, nodes may fail to retrieve configuration, fail to start kubelet correctly, fail to join the cluster, or remain stuck during bootstrap/control-plane initialization.

Recommended practices:

- Generate ignition files immediately before deployment
- Avoid long pauses between ignition generation, bootstrap installation, and control-plane installation
- Keep the temporary ignition HTTP server reachable until all required nodes have downloaded their ignition files
- Do not regenerate ignition files after the bootstrap process has started
- If the installation must be restarted from scratch because ignition material expired, recreate the installation directory and repeat the installation cleanly

Regenerating ignition files after bootstrap activation is not a safe recovery procedure, because the generated files are tied to the installation state and cluster bootstrap material. Doing so may invalidate assumptions already used by the bootstrap process and can leave the cluster in an inconsistent state.

If the deployment cannot continue within the validity window, the safest approach is usually to discard the partially used installation artifacts and restart the installation procedure from a clean installation directory, rather than mixing newly generated ignition files with an already-started bootstrap process.

20 Proxmox-Specific Notes

Although OKD is installed on CentOS Stream CoreOS, the virtual machines are created and managed manually in Proxmox VE. For this reason, some Proxmox-specific settings are important for a reliable installation.

The most important VM setting is the CPU type. All OKD VMs should use:

CPU Type
host

Using *host* allows the guest operating system to see the CPU features exposed by the physical Proxmox host. Generic CPU models such as kvm64 may hide required CPU instructions and can lead to boot problems, kernel panics, or unsupported instruction set errors.

After CentOS Stream CoreOS has been installed on the VM disk, the SCOS ISO should be detached or removed from the virtual CD/DVD drive. The boot order should then be changed so that the VM boots from the installed disk only.

Recommended post-installation boot configuration:

- remove ISO
- boot from disk only

If the ISO remains attached and first in the boot order, the VM may boot again into the live environment instead of the installed CoreOS system.

Networking must also be handled carefully. During installation, the SCOS live environment is used to download the appropriate ignition file and install CoreOS onto the VM disk. The working network configuration from the live environment should be copied into the installed system by using:

```
sudo coreos-installer install /dev/sda

    -i /tmp/<role>.ign[31]

    --copy-network
```

The *--copy-network* option is especially important in a Proxmox UPI installation, because after reboot the node must immediately be able to reach DNS, the HAProxy API endpoint, and the Machine Config Server. If the installed system loses its network configuration, the node may fail to join the cluster.

For each VM, verify that the expected virtual network interface is connected to the correct Proxmox bridge and VLAN/network segment before installation. The OKD node network must allow access to the API and Machine Config Server endpoints, while any additional storage network interfaces, such as the dedicated Ceph interface on workers, should be configured separately after the base cluster installation.

[31] <role>.ign must be replaced with the appropriate ignition file prefix for the node being installed: bootstrap for the bootstrap node, master for control-plane nodes, and worker for worker nodes. For example, the resulting ignition file path should be /tmp/bootstrap.ign, /tmp/master.ign, or /tmp/worker.ign.

In summary, the key Proxmox-specific requirements are:

- Use CPU type host
- Attach the SCOS ISO only for the installation phase
- Remove or detach the ISO after CoreOS installation
- Boot from the installed disk after installation
- Use `--copy-network` during coreos-installer install
- Verify that VM network interfaces are attached to the correct Proxmox bridges
- Ensure that node IP addresses, DNS, HAProxy, and routing are consistent before bootstrapping the cluster

21 Troubleshooting

During an OKD UPI installation, most failures are caused by one of the external dependencies required by the bootstrap process: DNS resolution, HAProxy routing, ignition file delivery, Machine Config Server reachability, node networking, or expired bootstrap material.

Troubleshooting should therefore start from the outside and move progressively toward the node internals.

First, verify that the Kubernetes API endpoint is reachable through HAProxy:

```
curl -k https://api.okd.ia2.inaf.it:6443/version
```

If the command returns Kubernetes version information in JSON format, DNS resolution, HAProxy routing on port 6443, and at least one API backend are working.

If the API is not reachable, check:

- DNS resolution for `api.okd.ia2.inaf.it`
- HAProxy frontend binding on port 6443
- HAProxy backend targets for the current installation phase
- Bootstrap node availability during the bootstrap phase
- Control-plane node availability after bootstrap completion
- Firewall or routing rules between the bastion, HAProxy, and cluster nodes

Verify DNS resolution:

```
dig api.okd.ia2.inaf.it
dig api-int.okd.ia2.inaf.it
```

The API and internal API names should resolve to the HAProxy host.

If nodes fail to retrieve their configuration or do not join the cluster, verify the Machine Config Server endpoint:

```
curl -k https://api-int.okd.ia2.inaf.it:22623/config/master
```

For worker nodes, use:

```
curl -k https://api-int.okd.ia2.inaf.it:22623/config/worker
```

A successful response confirms that the Machine Config Server endpoint is reachable. If these commands fail, check the HAProxy backend for port 22623. During bootstrap, this endpoint must be reachable according to the bootstrap-phase configuration. After bootstrap completion, it must be routed to the control-plane nodes.

On the bootstrap node, inspect the running containers:

```
sudo podman ps
```

The bootstrap containers should be running while the temporary bootstrap services are active.

Check the kubelet status on a node:

```
systemctl status kubelet
```

Inspect kubelet logs:

```
journalctl -u kubelet -b --no-pager
```

During bootstrap, inspect the bootkube service logs on the bootstrap node:

```
journalctl -b -f -u bootkube.service
```

Check listening ports on the relevant node or HAProxy host:

```
sudo ss -lntp
```

This is useful to confirm that expected services are listening on ports such as 6443, 22623, 80, or 443.

If ignition download fails from the SCOS live environment, verify that the temporary HTTP server is still running on the bastion and that the file can be retrieved:

```
curl -I http://<BASTION_IP>:8080/bootstrap.ign
```

```
curl -I http://<BASTION_IP>:8080/master.ign
```

```
curl -I http://<BASTION_IP>:8080/worker.ign
```

Also verify that the correct ignition file is used for each node role:

```
bootstrap node bootstrap.ign

control-plane nodes master.ign

worker nodes worker.ign
```

If installation appears to stall for a long time, also consider the ignition certificate validity window. Ignition files should be generated shortly before installation and should not be regenerated after bootstrap has started.

In general, the most common troubleshooting path is:

- DNS resolution
- HAProxy frontend/backend routing
- Ignition HTTP availability
- Machine Config Server reachability
- Bootstrap container status
- Kubelet and bootkube logs
- Node network configuration
- Certificate validity window

22 Final Cluster Validation

After the bootstrap phase has completed, HAProxy has been updated, and all control-plane and worker nodes have been installed, the final installation validation can be performed from the bastion host.

The goal of this step is to confirm that the cluster API is reachable, that all expected nodes have joined the cluster, and that the OpenShift cluster operators have reconciled successfully.

Set the kubeconfig generated by the installer:

```
export KUBECONFIG=auth/kubeconfig
```

Check the registered nodes:

```
oc get nodes
```

All control-plane and worker nodes should appear in Ready state. For this deployment, the expected nodes are:

- cp-1
- cp-2
- cp-3
- wk-1
- wk-2
- wk-3

Check the cluster operators:

```
oc get co
```

A healthy cluster should generally report:

```
AVAILABLE=True
DEGRADED=False
PROGRESSING=False
```

Some operators may remain temporarily in *`Progressing=True`* immediately after installation, while the cluster finishes reconciling. This should settle after the installation completes and all required components are available.

It is also useful to check the ClusterVersion object:

```
oc get clusterversion
```

And verify that no unexpected pods are failing:

```
oc get pods -A | grep -v Running
```

The command above may still show completed pods or transient initialization pods. These should be interpreted together with the ClusterOperator[32] status.

The installation can be considered successful when all expected nodes are Ready, the cluster operators are available and not degraded, and openshift-install wait-for install-complete reports a completed installation.

[32] ClusterOperator is an OpenShift status object representing the health and reconciliation state of a major cluster component, such as authentication, ingress, networking, storage, or the API server.

23 Installation Conclusions

The OKD 4.21 UPI installation on Proxmox VE confirmed that the most complex part of the deployment is not the installation command itself, but the correct preparation and sequencing of the external infrastructure around the cluster.

A UPI deployment requires the administrator to manually provide the virtual machines, DNS records, load-balancer configuration, ignition delivery mechanism, and network connectivity required by the bootstrap, control-plane, and worker nodes. Each of these components must be available at the right time during the installation process.

The installation validated the following core elements:

- Ignition delivery
- Bootstrap sequencing
- HAProxy routing
- Machine Config Server availability
- DNS correctness
- Certificate validity windows

The most critical aspects identified during deployment were:

- Correct HAProxy routing for port 22623
- Maintaining valid bootstrap certificates
- Ensuring bootstrap availability before control-plane provisioning
- Correct use of `--copy-network`
- Proper Proxmox CPU configuration (`host`)

Once the bootstrap and Machine Config Server routing are correctly configured, the remainder of the deployment proceeds reliably.

The most critical operational point was the routing of the Machine Config Server on port 22623. During installation, nodes depend on this endpoint to retrieve machine configuration data and complete their initialization. Incorrect MCS routing can prevent control-plane or worker nodes from joining the cluster, even when DNS and API access appear to be working.

The bootstrap phase also proved to be time-sensitive. The generated ignition files contain short-lived certificate material, so they should be used soon after generation and should not be regenerated after the bootstrap process has started. Long pauses between ignition generation, bootstrap installation, and control-plane provisioning increase the risk of certificate-related installation failures.

Several Proxmox-specific details were also important. The VM CPU type should be set to *host* to avoid unsupported instruction set or kernel-level issues. The SCOS installation should preserve working network configuration by using *--copy-network*, especially when static addressing or manually configured live-environment networking is used.

After bootstrap completion, the HAProxy configuration must be updated carefully. Bootstrap backend entries should be removed, API traffic should be routed only to the real control-plane nodes, and the Machine Config Server endpoint should be routed to the control-plane nodes. This transition marks the point where the temporary bootstrap node is no longer part of the installation path.

Overall, the installation showed that OKD on Proxmox VE is reproducible when the external dependencies are explicitly documented and validated. Once DNS, ignition delivery, HAProxy routing, MCS reachability, certificate timing, and Proxmox VM settings are correct, the remaining installation steps proceed predictably and the cluster can be brought to a healthy operational state.

Part 2

24 Dedicated Ceph Network on Worker Nodes

After the base OKD cluster was installed, a dedicated Ceph storage network was configured on the worker nodes through the additional interface *ens19*[33].

The goal was to provide direct worker-to-Ceph connectivity on a dedicated private storage network, represented here as *<CEPH_STORAGE_CIDR>*[34], while keeping the existing OKD external/API network on *ens18*/*br-ex* unchanged.

[33] ens18, ens19, and similar names are Linux network interface names inside the OKD virtual machines. In this deployment, ens18 is the main node interface associated with the OKD external/API network and OpenShift br-ex, which is the external Linux bridge used by OVN-Kubernetes/OpenShift node networking. The ens19 interface is an additional worker interface configured for the dedicated Ceph storage network. Interface names are environment-specific and should be verified on each node with commands such as ip -br a or nmcli device status.

[34] <CEPH_STORAGE_CIDR> represents the CIDR block of the dedicated Ceph storage network, for example a private subnet used only for storage traffic between OKD worker nodes and the external Ceph cluster. It should not overlap with the OKD pod network, service network, Proxmox management network, API/ingress network, or any other routed infrastructure network.

Worker Ceph Network Addressing:

Worker	**Ceph Interface**	**Ceph IP**
wk-1.okd.ia2.inaf.it	ens19	<WK1_CEPH_IP>/<PREFIX>[35]
wk-2.okd.ia2.inaf.it	ens19	<WK2_CEPH_IP>/<PREFIX>
wk-3.okd.ia2.inaf.it	ens19	<WK3_CEPH_IP>/<PREFIX>

For the MachineConfig-Based[36] network persistence, the final approach used NetworkManager[37] profiles managed by MachineConfig-delivered systemd units.

For each dedicated worker MachineConfigPool, a MachineConfig installed a small script under `/usr/local/bin` and enabled a systemd oneshot unit.

[35] <WKN_CEPH_IP>/<PREFIX> represents the IP address and subnet prefix assigned to a worker node on the dedicated Ceph storage network. For example, <WK1_CEPH_IP>/<PREFIX> is the Ceph-network address configured on wk-1 through the additional storage interface, such as ens19. This address is used for worker-to-Ceph communication and should belong to <CEPH_STORAGE_CIDR>.

[36] MachineConfig is an OpenShift object used to manage operating-system-level configuration on cluster nodes, such as files, systemd units, kernel arguments, or networking scripts. A MachineConfigPool groups nodes with the same role and applies the matching MachineConfigs to them. This mechanism is OpenShift-specific and is part of the Machine Config Operator workflow.

[37] The NetworkManager is the Linux service used to manage network interfaces and connection profiles. nmcli is its command-line tool. In this report it is used to create and persist the dedicated Ceph network profile on the worker nodes.

Example for *wk-1*:

```
cat >/root/ceph-ens19-wk1.sh <<'EOF'
#!/usr/bin/bash
set -euxo pipefail
if /usr/bin/nmcli -t -f NAME con show | grep -qx ceph-net; then
  /usr/bin/nmcli con mod ceph-net \
    connection.id ceph-net \
    connection.interface-name ens19 \
    connection.autoconnect yes \
    ipv4.method manual \
    ipv4.addresses <WK1_CEPH_IP>/<PREFIX> \
    ipv4.never-default yes \
    ipv6.method ignore
else
  /usr/bin/nmcli con add type ethernet ifname ens19 con-name ceph-net \
    ipv4.method manual \
    ipv4.addresses <WK1_CEPH_IP>/<PREFIX> \
    ipv4.never-default yes \
    ipv6.method ignore \
    connection.autoconnect yes
fi
/usr/bin/nmcli con up ceph-net
EOF
chmod 755 /root/ceph-ens19-wk1.sh
SCRIPT_B64=$(base64 -w0 /root/ceph-ens19-wk1.sh)
cat <<EOF | oc apply -f -
```

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  name: 99-worker-wk1-ceph-ens19
  labels:
    machineconfiguration.openshift.io/role: worker-wk1
spec:
  config:
    ignition:
      version: 3.4.0
    storage:
      files:
        - path: /usr/local/bin/ceph-ens19-wk1.sh
          mode: 0755
          overwrite: true
          contents:
            source: data:text/plain;charset=utf-8;base64,${SCRIPT_B64}
    systemd:
      units:
        - name: ceph-ens19-wk1.service
          enabled: true
          contents: |
            [Unit]
            Description=Configure Ceph network on ens19 for wk-1
            After=network-online.target
            Wants=network-online.target
            ConditionPathExists=/sys/class/net/ens19
            [Service]
```

```
            Type=oneshot
            ExecStart=/usr/local/bin/ceph-ens19-wk1.sh
            RemainAfterExit=yes
            [Install]
            WantedBy=multi-user.target
EOF
```

Equivalent commands were applied for *wk-2* and *wk-3*, changing only the MachineConfig name, script name, MachineConfigPool role label, systemd unit name, and IP address.

The following validation commands check the dedicated Ceph network.

OpenShift command:

```
oc get mcp

oc get nodes
```

Kubernetes equivalent:

```
kubectl get nodes
```

There is no Kubernetes equivalent for *MachineConfigPool*, it is OpenShift-specific.

Validate the NetworkManager profile and interface address:

```
oc debug node/wk-1.okd.ia2.inaf.it
    -- chroot /host nmcli -t -f NAME,DEVICE,TYPE con show

oc debug node/wk-1.okd.ia2.inaf.it -- chroot /host ip -br a
```

Kubernetes equivalent:

```
kubectl debug node/wk-1.okd.ia2.inaf.it -it
    --image=registry.access.redhat.com/ubi9/ubi
    -- chroot /host ip -br a
```

Validate Ceph monitor reachability from each worker:

```
oc debug node/wk-1.okd.ia2.inaf.it -- chroot /host ping -c 2
    <CEPH_MON_1_IP>[38]

oc debug node/wk-1.okd.ia2.inaf.it -- chroot /host nc -vz -w 3
    <CEPH_MON_1_IP> 3300

oc debug node/wk-1.okd.ia2.inaf.it -- chroot /host nc -vz -w 3
    <CEPH_MON_1_IP> 6789
```

The same checks were performed successfully on *wk-2* and *wk-3*.

[38] <CEPH_MON_N_IP> represents the IP address of a Ceph monitor reachable from the dedicated Ceph storage network. For example, <CEPH_MON_1_IP> is the first Ceph monitor endpoint used by OKD worker nodes and Ceph-CSI to reach the external Ceph cluster. The same pattern applies to <CEPH_MON_2_IP>, <CEPH_MON_3_IP>, and any additional monitor endpoints.

25 External Ceph RBD Integration with Ceph-CSI

A dedicated Ceph RBD pool named *okd-rbd* was created on the external Ceph cluster and exposed to OKD through the Ceph-CSI RBD driver.

The Ceph-side cluster configuration information was saved under:

```
/opt/okd/backups/ceph-csi/
```

Important files:

File	Purpose
fsid.txt	Ceph cluster FSID[39]
mon-dump.txt	Monitor endpoints
client-okd.txt	CephX[40] client definition
client-okd.key	CephX key only
pools.txt	Pool information

Ceph-side commands:

```
ceph osd pool create okd-rbd

rbd pool init okd-rbd

ceph auth get-or-create client.okd
    mon 'profile rbd'
    osd 'profile rbd pool=okd-rbd'
    mgr 'profile rbd pool=okd-rbd'
```

[39] Ceph FSID is the unique identifier of a Ceph cluster. It is used by Ceph clients and CSI configuration to identify the target Ceph cluster unambiguously.

[40] CephX is the native Ceph authentication mechanism. A CephX client identity, such as client.okd, uses a secret key and assigned capabilities to authenticate to the Ceph cluster and access only the permitted pools and operations.

The resulting key was stored securely and should not be committed to Git or exposed in documentation.

The following commands implement the OKD namespace and the base ConfigMaps.

OpenShift command:

```
oc new-project ceph-csi-rbd
```

Kubernetes equivalent:

```
kubectl create namespace ceph-csi-rbd
```

The Ceph-CSI configuration was created as a ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: ceph-csi-config
  namespace: ceph-csi-rbd
data:
  config.json: |-
    [
      {
        "clusterID": "b8503354-d5dd-4e10-823f-8bc615cb5ed3",
        "monitors": [
          "<CEPH_MON_1_IP>:6789",
          "<CEPH_MON_2_IP>:6789",
          "<CEPH_MON_3_IP>:6789",
```

```
        "<CEPH_MON_4_IP>:6789"
      ]
    }
  ]
```

The KMS[41] configuration was intentionally empty:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: ceph-csi-encryption-kms-config
  namespace: ceph-csi-rbd
data:
  config.json: "{}"
```

[41] Key Management Service (KMS), in Ceph-CSI, is used when encrypted volumes require integration with an external key management system. In this deployment it was left empty because RBD volume encryption through an external KMS was not configured.

The Ceph client configuration was created as:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: ceph-config
  namespace: ceph-csi-rbd
data:
  ceph.conf: |
    [global]
    auth_cluster_required = cephx
    auth_service_required = cephx
    auth_client_required = cephx
  keyring: |
```

The RBD secret used a placeholder in this report:

```
apiVersion: v1
kind: Secret
metadata:
  name: csi-rbd-secret
  namespace: ceph-csi-rbd
stringData:
  userID: okd
  userKey: <CLIENT_OKD_KEY>[42]
```

OpenShift command:

```
oc apply -f <file>.yaml
```

Kubernetes equivalent:

```
kubectl apply -f <file>.yaml
```

The following commands install the Ceph-CSI RBD manifests.

[42] <CLIENT_OKD_KEY> represents the CephX secret key associated with the client.okd Ceph user. It is used by Ceph-CSI to authenticate to the external Ceph cluster and provision RBD volumes. This value is sensitive and must not be committed to Git, exposed in documentation, or stored with world-readable permissions.

The Ceph-CSI RBD Kubernetes manifests were downloaded from the Ceph-CSI repository using version v3.16.0.

```
export CEPHCSI_VER=v3.16.0

mkdir -p /opt/okd/backups/ceph-csi/manifests/upstream

cd /opt/okd/backups/ceph-csi/manifests/upstream

curl -LO
https://raw.githubusercontent.com/ceph/ceph-csi/${CEPHCSI_VER}/deploy/r
bd/kubernetes/csi-provisioner-rbac.yaml

curl -LO
https://raw.githubusercontent.com/ceph/ceph-csi/${CEPHCSI_VER}/deploy/r
bd/kubernetes/csi-nodeplugin-rbac.yaml

curl -LO
https://raw.githubusercontent.com/ceph/ceph-csi/${CEPHCSI_VER}/deploy/r
bd/kubernetes/csi-rbdplugin-provisioner.yaml

curl -LO
https://raw.githubusercontent.com/ceph/ceph-csi/${CEPHCSI_VER}/deploy/r
bd/kubernetes/csi-rbdplugin.yaml
```

The manifests default to the `default` namespace, so they were rewritten for *`ceph-csi-rbd`*:

```
sed -i 's/namespace: default/namespace: ceph-csi-rbd/g'
    csi-provisioner-rbac.yaml

sed -i 's/namespace: default/namespace: ceph-csi-rbd/g'
    csi-nodeplugin-rbac.yaml

sed -i 's/namespace: default/namespace: ceph-csi-rbd/g'
    csi-rbdplugin-provisioner.yaml

sed -i 's/namespace: default/namespace: ceph-csi-rbd/g'
    csi-rbdplugin.yaml
```

OpenShift command:

```
oc apply -f csi-provisioner-rbac.yaml

oc apply -f csi-nodeplugin-rbac.yaml

oc apply -f csi-rbdplugin-provisioner.yaml

oc apply -f csi-rbdplugin.yaml
```

Kubernetes equivalent:

```
kubectl apply -f csi-provisioner-rbac.yaml

kubectl apply -f csi-nodeplugin-rbac.yaml

kubectl apply -f csi-rbdplugin-provisioner.yaml

kubectl apply -f csi-rbdplugin.yaml
```

There are some OpenShift SCC[43] requirements to consider.

Ceph-CSI requires privileged access for node-level RBD operations.

OpenShift command:

```
oc adm policy add-scc-to-user privileged -z rbd-csi-nodeplugin -n
    ceph-csi-rbd

oc adm policy add-scc-to-user privileged -z rbd-csi-provisioner -n
    ceph-csi-rbd
```

There is no direct Kubernetes equivalent to OpenShift SCC. In Kubernetes, this is normally handled through Pod Security Admission namespace labels, Pod Security Standards, and/or distribution-specific admission policies.

After applying the Ceph-CSI manifests, some deployment-specific adjustments were required to make the RBD provisioner work correctly in this OKD environment.

[43] OpenShift Security Context Constraints (SCCs) are OpenShift-specific security policies that control how pods are allowed to run. They define constraints such as whether containers may run as root, which user IDs they may use, which Linux capabilities are allowed, whether host networking or host paths can be used, and which volume types are permitted. SCCs have no direct upstream Kubernetes equivalent; in standard Kubernetes, similar concerns are handled through securityContext settings, Pod Security Standards, Pod Security Admission, and distribution-specific admission policies.

The final working Ceph-CSI controller deployment required the following production-specific adjustments:

- Run the RBD provisioner deployment with `hostNetwork: true`[44]
- Set `dnsPolicy: ClusterFirstWithHostNet`
- Use a single RBD provisioner replica
- Disable topology handling in the external CSI provisioner by using one combined feature-gate[45] argument:
  `--feature-gates=HonorPVReclaimPolicy=true,Topology=false`
- Do not pass `--feature-gates=Topology=false` to the `cephcsi` container itself; that flag is not valid for the Ceph-CSI binary
- Change the controller liveness metrics port from `8680` to `8681`, because with `hostNetwork: true` the provisioner pod could conflict with the nodeplugin metrics listener already using the same host port

[44] hostNetwork: true makes a pod use the node network namespace instead of the normal pod network. When this is enabled, dnsPolicy: ClusterFirstWithHostNet is commonly used so that the pod can still resolve cluster DNS names correctly while using host networking.

[45] Feature gates are command-line switches used by Kubernetes components or sidecars to enable or disable specific optional behaviors. In this report, the feature-gate setting applies to the external CSI provisioner sidecar, not to the Ceph-CSI driver binary itself.

Final JSON patch logic:

```
oc get deployment -n ceph-csi-rbd csi-rbdplugin-provisioner -o json
| jq '
  .spec.replicas = 1 |
  .spec.strategy = {
    "type": "RollingUpdate",
    "rollingUpdate": {"maxSurge": 0, "maxUnavailable": 1}
  } |
  .spec.template.spec.hostNetwork = true |
  .spec.template.spec.dnsPolicy = "ClusterFirstWithHostNet" |
  .spec.template.spec.containers |= map(
    if .name == "csi-rbdplugin" then
      .args = [.args[] | select(. != "--feature-gates=Topology=false")]
    elif .name == "csi-provisioner" then
      .args = ([.args[]
        | select(. != "--feature-gates=HonorPVReclaimPolicy=true")
        | select(. != "--feature-gates=Topology=false")
        | select(. !=
"--feature-gates=HonorPVReclaimPolicy=true,Topology=false")
      ] + ["--feature-gates=HonorPVReclaimPolicy=true,Topology=false"])
    elif .name == "liveness-prometheus" then
      .args = ([.args[] | select(. != "--metricsport=8680")] +
["--metricsport=8681"])
      | .ports = ([.ports[] | if .name == "http-metrics" then .containerPort
= 8681 else . end])
    else .
    end
  )
```

```
' > /tmp/csi-rbdplugin-provisioner.final.json
```

To apply:

```
oc apply -f /tmp/csi-rbdplugin-provisioner.final.json

oc rollout restart -n ceph-csi-rbd deployment/csi-rbdplugin-provisioner

oc rollout status -n ceph-csi-rbd deployment/csi-rbdplugin-provisioner
```

Kubernetes equivalent:

```
kubectl apply -f /tmp/csi-rbdplugin-provisioner.final.json

kubectl rollout restart -n ceph-csi-rbd
    deployment/csi-rbdplugin-provisioner

kubectl rollout status -n ceph-csi-rbd
    deployment/csi-rbdplugin-provisioner
```

The final StorageClass was:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: okd-rbd
provisioner: rbd.csi.ceph.com
parameters:
  clusterID: <CEPH_FSID>
  pool: okd-rbd
  imageFeatures: layering
  csi.storage.k8s.io/provisioner-secret-name: csi-rbd-secret
  csi.storage.k8s.io/provisioner-secret-namespace: ceph-csi-rbd
  csi.storage.k8s.io/controller-expand-secret-name: csi-rbd-secret
  csi.storage.k8s.io/controller-expand-secret-namespace: ceph-csi-rbd
  csi.storage.k8s.io/node-stage-secret-name: csi-rbd-secret
  csi.storage.k8s.io/node-stage-secret-namespace: ceph-csi-rbd
reclaimPolicy: Delete
allowVolumeExpansion: true
volumeBindingMode: WaitForFirstConsumer[46]
```

OpenShift command:

```
oc apply -f storageclass-okd-rbd.yaml

oc get sc
```

[46] WaitForFirstConsumer is a Kubernetes volume binding mode where a PVC is not bound immediately when created. Binding is delayed until a pod that uses the PVC is scheduled, so the storage system can take the selected node and scheduling constraints into account.

Kubernetes equivalent:

```
kubectl apply -f storageclass-okd-rbd.yaml

kubectl get storageclass
```

To validate Ceph-CSI, a test PVC and pod confirmed end-to-end persistence:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-rbd-pvc
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce[47]
  resources:
    requests:
      storage: 2Gi
  storageClassName: okd-rbd
---
apiVersion: v1
kind: Pod
metadata:
  name: test-rbd-pod
  namespace: default
```

[47] ReadWriteOnce means that a volume can normally be mounted read-write by a single node at a time. ReadWriteMany means that a volume can be mounted read-write by multiple nodes at the same time, if the storage backend supports it.

```
spec:
  containers:
    - name: test
      image: registry.access.redhat.com/ubi9/ubi-minimal
      command: ["/bin/sh","-c"]
      args:
        - "echo hello-from-ceph > /data/hello.txt && sleep 3600"
      volumeMounts:
        - name: data
          mountPath: /data
  volumes:
    - name: data
      persistentVolumeClaim:
        claimName: test-rbd-pvc
```

OpenShift command:

```
oc get pvc,pv,pod -n default -o wide

oc exec -n default test-rbd-pod -- cat /data/hello.txt
```

Kubernetes equivalent:

```
kubectl get pvc,pv,pod -n default -o wide

kubectl exec -n default test-rbd-pod -- cat /data/hello.txt
```

Expected output:

```
hello-from-ceph
```

26 Backup Strategy for Ceph-CSI and Cluster Manifests

Final working manifests and runtime status were saved under:

```
/opt/okd/backups/ceph-csi/
```

Recommended backup commands:

```
mkdir -p /opt/okd/backups/ceph-csi
oc get namespace ceph-csi-rbd -o yaml >
/opt/okd/backups/ceph-csi/00-namespace.ceph-csi-rbd.final.yaml

oc get configmap -n ceph-csi-rbd ceph-csi-config -o yaml >
/opt/okd/backups/ceph-csi/01-configmap.ceph-csi-config.final.yaml

oc get configmap -n ceph-csi-rbd ceph-csi-encryption-kms-config -o yaml
>
/opt/okd/backups/ceph-csi/02-configmap.ceph-csi-encryption-kms-config.f
inal.yaml

oc get configmap -n ceph-csi-rbd ceph-config -o yaml >
/opt/okd/backups/ceph-csi/03-configmap.ceph-config.final.yaml

oc get secret -n ceph-csi-rbd csi-rbd-secret -o yaml >
/opt/okd/backups/ceph-csi/04-secret.csi-rbd-secret.final.yaml

oc get serviceaccount -n ceph-csi-rbd rbd-csi-provisioner -o yaml >
/opt/okd/backups/ceph-csi/05-sa.rbd-csi-provisioner.final.yaml

oc get serviceaccount -n ceph-csi-rbd rbd-csi-nodeplugin -o yaml >
/opt/okd/backups/ceph-csi/06-sa.rbd-csi-nodeplugin.final.yaml

oc get deployment -n ceph-csi-rbd csi-rbdplugin-provisioner -o yaml >
/opt/okd/backups/ceph-csi/07-deployment.csi-rbdplugin-provisioner.final
.yaml

oc get daemonset -n ceph-csi-rbd csi-rbdplugin -o yaml >
```

```
/opt/okd/backups/ceph-csi/08-daemonset.csi-rbdplugin.final.yaml

oc get storageclass okd-rbd -o yaml >
/opt/okd/backups/ceph-csi/11-storageclass.okd-rbd.final.yaml

oc get all -n ceph-csi-rbd -o wide >
/opt/okd/backups/ceph-csi/22-runtime-status.ceph-csi-rbd.txt
```

Kubernetes equivalent: replace *oc* with *kubectl* for standard Kubernetes objects. There is no Kubernetes equivalent for OpenShift SCC commands.

Sensitive files, especially CephX keys and Kubernetes secrets, must be protected with restrictive permissions.

27 OpenShift Authentication with HTPasswd

A local HTPasswd identity provider was added to allow local users to log in to OKD.

To install the *htpasswd* utility on RHEL/CentOS-compatible systems:

```
sudo dnf install -y httpd-tools
```

The *htpasswd* binary is provided by the *httpd-tools* package.

To create or update the *htpasswd* file, for example:

```
mkdir -p /opt/okd/backups/htpasswd

htpasswd -c /opt/okd/backups/htpasswd/htpasswd massimo.costantini
```

For additional users, omit `-c`:

```
htpasswd /opt/okd/backups/htpasswd/htpasswd giacomo.coran

htpasswd /opt/okd/backups/htpasswd/htpasswd andrea.bignamini

htpasswd /opt/okd/backups/htpasswd/htpasswd giovanni.lacopo
```

To create or update the OpenShift secret:

```
oc create secret generic htpasswd-secret
    --from-file=htpasswd=/opt/okd/backups/htpasswd/htpasswd
    -n openshift-config
    --dry-run=client -o yaml | oc apply -f -
```

This command is OpenShift-specific because the secret is referenced by the OpenShift OAuth operator.

To configure the OAuth identity provider:

```
cat <<'EOF' | oc apply -f -
apiVersion: config.openshift.io/v1
kind: OAuth
metadata:
  name: cluster
spec:
  identityProviders:
  - name: localusers
    mappingMethod: claim
    type: HTPasswd
    htpasswd:
      fileData:
        name: htpasswd-secret
EOF
```

There is no direct Kubernetes equivalent to OpenShift OAuth configuration. Kubernetes authentication is normally provided externally through the API server configuration, OIDC, certificates, or a distribution-specific authentication stack.

The following commands validate the authentication operator:

```
oc get pods -n openshift-authentication
```

```
oc get co authentication
```

The watch form is useful during rollouts but intentionally keeps the terminal attached until interrupted with *Ctrl+C*:

```
oc get pods -n openshift-authentication -w
```

To switch back to the admin context, check the current user:

```
oc whoami
```

List contexts:

```
oc config get-contexts
```

Switch back to an existing admin context:

```
oc config use-context admin
```

or, depending on the context name:

```
oc config use-context test/api-okd-ia2-inaf-it:6443/system:admin
```

Kubernetes equivalent:

```
kubectl config get-contexts

kubectl config use-context <context-name>
```

28 Projects, Groups, Users, and Tenant Onboarding

OpenShift Projects were used to represent tenant namespaces.

To create a test project:

```
oc new-project test --display-name="Test" --description="Test Project"
```

Kubernetes equivalent:

```
kubectl create namespace test
```

An OpenShift *`Project`* is a Kubernetes *`Namespace`* with OpenShift-specific metadata and project lifecycle behavior.

Two additional production tenant projects were created:

```
oc new-project yabi --display-name="Yabi Project" --description="Yabi
    Project"

oc new-project nadir --display-name="Nadir Project"
    --description="Nadir Project"
```

Kubernetes equivalent:

```
kubectl create namespace yabi

kubectl create namespace nadir
```

OpenShift groups were created with the same names as the projects:

```
oc adm groups new test || true

oc adm groups new yabi || true

oc adm groups new nadir || true
```

Users were added to groups:

```
oc adm groups add-users test massimo.costantini

oc adm groups add-users test giovanni.lacopo

oc adm groups add-users yabi andrea.bignamini

oc adm groups add-users nadir giacomo.coran
```

The *edit* role was assigned to each group in its project:

```
oc adm policy add-role-to-group edit test -n test

oc adm policy add-role-to-group edit yabi -n yabi

oc adm policy add-role-to-group edit nadir -n nadir
```

Kubernetes equivalent RBAC example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: edit-to-test-group
  namespace: test
subjects:
- kind: Group
  name: test
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
```

Apply with:

```
kubectl apply -f rolebinding.yaml
```

The following commands validate user permissions.

OpenShift command:

```
oc login https://api.okd.ia2.inaf.it:6443 -u massimo.costantini

oc project test

oc auth can-i create deployment -n test

oc auth can-i create pvc -n test
```

Kubernetes equivalent:

```
kubectl auth can-i create deployment -n test

kubectl auth can-i create pvc -n test
```

Expected:

```
yes
yes
```

Naming recommendation for future identity providers, usernames were created in *`firstname.lastname`* form, for example:

```
massimo.costantini
giacomo.coran
andrea.bignamini
giovanni.lacopo
```

This naming convention is suitable for future migration to an external identity provider such as Keycloak, provided the future identity provider exposes the same username claim.

Part 3

29 Application Deployment Example with Persistent Ceph RBD Storage

A complete test application was deployed in the *test* project to validate daily cluster usage by a non-admin user.

The application exposes a simple web page with a text field. Submitted strings are appended to a file stored on a Ceph-backed PVC. This provides a simple end-to-end validation of application deployment, internal service exposure, external HTTPS routing, dynamic storage provisioning, and persistent data writes.

The example manifests are stored under:

```
/opt/okd/example
```

The application is created by applying the following manifests:

```
oc apply -f 01-configmap-test-web-app.yaml

oc apply -f 02-pvc-test-web-pvc.yaml

oc apply -f 03-deployment-test-web.yaml

oc apply -f 04-service-test-web.yaml

oc apply -f 05-route-test-web-https.yaml
```

The files have the following purpose:

01-configmap-test-web-app.yaml

Contains the Python source code of the web application, stored inside a Kubernetes/OpenShift ConfigMap[48].

02-pvc-test-web-pvc.yaml

Contains the PersistentVolumeClaim definition, which requests persistent disk space for the web application.

03-deployment-test-web.yaml

Contains the Deployment definition, which tells OpenShift to start and keep the application container running.

04-service-test-web.yaml

Contains the Kubernetes/OpenShift Service, which exposes the application internally inside the cluster.

05-route-test-web-https.yaml

Contains the OpenShift Route, which exposes the test-web Service outside the cluster through HTTPS.

After the resources have been created, the application can be accessed from a browser through the HTTPS Route:

```
https://test-web-test.apps.okd.ia2.inaf.it
```

[48] ConfigMap is a Kubernetes object used to store non-sensitive configuration data as key-value pairs or files. In this example, the Python source file app.py is stored inside a ConfigMap and mounted into the application pod as a file. This avoids building a custom container image for the demo application, but for production workloads the application code should normally be packaged into a versioned container image.

After submitting a message through the web form, verify that the message has been written to the persistent file:

```
oc exec -n test deploy/test-web -- cat /data/messages.txt
```

The expected output is the list of messages submitted through the web interface. Because `/data/messages.txt` is stored on a Ceph-backed PVC, the content should persist across pod deletion and recreation.

The following subsections describe the five manifest files used to deploy the demo application. Each file creates one Kubernetes/OpenShift resource and represents one step of the deployment workflow.

The resources are presented in the same order in which they are normally applied: first the ConfigMap containing the application code, then the PersistentVolumeClaim requesting persistent storage, then the Deployment that runs the container, followed by the Service for internal cluster exposure and the OpenShift Route for external HTTPS access.

For comparison, the external exposure step is also shown with a standard Kubernetes Ingress manifest, which represents the closest Kubernetes equivalent to the OpenShift Route.

1. 01-configmap-test-web-app.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: test-web-app
  namespace: test
data:
  app.py: |
    import os
    from http.server import BaseHTTPRequestHandler, HTTPServer
    from urllib.parse import parse_qs
    DATA_DIR = "/data"
    DATA_FILE = os.path.join(DATA_DIR, "messages.txt")
    os.makedirs(DATA_DIR, exist_ok=True)
    class Handler(BaseHTTPRequestHandler):
        def _send_html(self, body, code=200):
            self.send_response(code)
            self.send_header("Content-Type", "text/html; charset=utf-8")
            self.end_headers()
            self.wfile.write(body.encode("utf-8"))
        def do_GET(self):
            messages = ""
            if os.path.exists(DATA_FILE):
                with open(DATA_FILE, "r", encoding="utf-8") as f:
                    messages = f.read()
            body = f"""<!doctype html>
    <html>
      <head>
```

```
    <meta charset="utf-8">
    <title>Test Ceph Write</title>
</head>
<body>
  <h1>Test Ceph Write</h1>
  <form method="POST">
    <input type="text" name="message" style="width: 400px;"
      placeholder="Write a string" />
      <button type="submit">Save</button>
  </form>
  <h2>File content</h2>
  <pre>{messages}</pre>
</body>
</html>
"""
        self._send_html(body)
    def do_POST(self):
        length = int(self.headers.get("Content-Length", "0"))
        raw = self.rfile.read(length).decode("utf-8")
        data = parse_qs(raw)
        message = data.get("message", [""])[0].strip()
        if message:
            with open(DATA_FILE, "a", encoding="utf-8") as f:
                f.write(message + "\n")
        self.send_response(303)
        self.send_header("Location", "/")
        self.end_headers()
if __name__ == "__main__":
    server = HTTPServer(("0.0.0.0", 8080), Handler)
    server.serve_forever()
```

To apply:

```
oc apply -f 01-configmap-test-web-app.yaml
```

Kubernetes equivalent:

```
kubectl apply -f 01-configmap-test-web-app.yaml
```

The final working deployment used the OpenShift-compatible UBI[49] Python image instead of the generic *`python:3.11-slim`* image.

The UBI Python image provides a Python runtime based on Red Hat Universal Base Image and is better aligned with OpenShift container security expectations. In particular, it works well with non-root execution, arbitrary user IDs, and the default filesystem layout used by OpenShift-compatible application images.

This avoided issues encountered with the generic *`python:3.11-slim`* image and allowed the demo application to run without building a custom container image.

[49] Universal Base Image (UBI) is a Red Hat base container image family designed for enterprise Linux container workloads. UBI images are commonly used in OpenShift environments because they support non-root execution patterns and are maintained with security updates from the Red Hat ecosystem.

2. 02-pvc-test-web-pvc.yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-web-pvc
  namespace: test
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
  storageClassName: okd-rbd
```

OpenShift command:

```
oc apply -f 02-pvc-test-web-pvc.yaml
```

Kubernetes equivalent:

```
kubectl apply -f 02-pvc-test-web-pvc.yaml
```

Because the *okd-rbd* StorageClass uses WaitForFirstConsumer, the PVC remains *Pending* until a pod using it is scheduled.

3. 03-deployment-test-web.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-web
  namespace: test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test-web
  template:
    metadata:
      labels:
        app: test-web
    spec:
      containers:
      - name: web
        image: registry.access.redhat.com/ubi9/python-311:latest
        imagePullPolicy: IfNotPresent
        command: ["python", "/opt/app-root/src/app.py"]
        ports:
        - containerPort: 8080
        securityContext:
          runAsNonRoot: true
          allowPrivilegeEscalation: false
          capabilities:
            drop:
```

```
        - ALL
      seccompProfile:
        type: RuntimeDefault
    volumeMounts:
    - name: app-code
      mountPath: /opt/app-root/src
    - name: data
      mountPath: /data
  volumes:
  - name: app-code
    configMap:
      name: test-web-app
  - name: data
    persistentVolumeClaim:
      claimName: test-web-pvc
```

OpenShift command:

```
oc apply -f 03-deployment-test-web.yaml
```

Kubernetes equivalent:

```
kubectl apply -f 03-deployment-test-web.yaml
```

The Service provides a stable internal endpoint for the application inside the cluster. Pods can be recreated with different IP addresses, so the Route and other cluster components should not connect directly to the pod IP. The Service selects the application pod through the *`app: test-web`* label and forwards traffic to container port 8080.

The OpenShift Route defined in the next step exposes this Service externally.

4. 04-service-test-web.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: test-web
  namespace: test
spec:
  selector:
    app: test-web
  ports:
  - name: http
    port: 8080
    targetPort: 8080
```

OpenShift command:

```
oc apply -f 04-service-test-web.yaml
```

Kubernetes equivalent:

```
kubectl apply -f 04-service-test-web.yaml
```

The OpenShift Route exposes the internal *test-web* Service outside the cluster through the OpenShift ingress router. While the Service provides connectivity only inside the cluster, the Route creates an externally reachable hostname under the **.apps.okd.ia2.inaf.it* wildcard domain.

In this example, the Route points to the *test-web* Service and forwards traffic to the Service port named *http*. With edge TLS termination, HTTPS is terminated by the OpenShift router, while traffic from the router to the application Service remains HTTP on port 8080.

5. 05-route-test-web-https.yaml

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: test-web
  namespace: test
spec:
  to:
    kind: Service
    name: test-web
  port:
    targetPort: http
```

OpenShift command:

```
oc apply -f 05-route-test-web-https.yaml
```

The resulting external hostname is:

```
https://test-web-test.apps.okd.ia2.inaf.it
```

There is no direct Kubernetes kubectl equivalent for OpenShift Route, because Route is an OpenShift-specific API object.

A standard Kubernetes equivalent would normally be an Ingress, assuming an Ingress controller is installed:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-web
  namespace: test
spec:
  rules:
  - host: test-web-test.apps.okd.ia2.inaf.it
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: test-web
            port:
              number: 8080
```

Kubernetes command:

```
kubectl apply -f 05-ingress-test-web.yaml
```

During testing, using hostUsers: false caused the pod to fail with:

```
container create failed: setgroups: Invalid argument
```

For the demo workload, the working solution was to avoid *hostUsers: false* and allow the project service account to use an appropriate OpenShift Security Context Constraint.

OpenShift command:

```
oc adm policy add-scc-to-user anyuid -z default -n test
```

There is no direct Kubernetes equivalent to OpenShift SCC. In Kubernetes this is normally handled through Pod Security Admission labels, securityContext settings, or distribution-specific admission policies.

Check the pod, service, and route:

```
oc get pod,pvc,svc,route -n test
```

Kubernetes equivalent:

```
kubectl get pod,pvc,svc,ingress -n test
```

Open the application through the OpenShift route:

```
oc get route test-web -n test
```

Example route:

```
https://test-web-test.apps.okd.ia2.inaf.it
```

After submitting a string from the web form, verify that the value was written to the persistent volume:

```
oc exec -n test deploy/test-web -- cat /data/messages.txt
```

Kubernetes equivalent:

```
kubectl exec -n test deploy/test-web -- cat /data/messages.txt
```

To verify persistence across pod recreation:

```
oc delete pod -n test -l app=test-web
```

```
oc get pods -n test -w
```

```
oc exec -n test deploy/test-web -- cat /data/messages.txt
```

Kubernetes equivalent:

```
kubectl delete pod -n test -l app=test-web
```

```
kubectl get pods -n test -w
```

```
kubectl exec -n test deploy/test-web -- cat /data/messages.txt
```

This confirms that the application writes data into */data/messages.txt*, where */data* is backed by the Ceph RBD volume dynamically provisioned through the *okd-rbd* StorageClass.

30 Backup Strategy for Tenant Projects and Test Application

The final `test` project manifests were saved under:

```
/opt/okd/backups/test/
```

Recommended backup commands:

```
mkdir -p /opt/okd/backups/test

oc get project test -o yaml >
/opt/okd/backups/test/00-project.test.yaml

oc get group test -o yaml > /opt/okd/backups/test/01-group.test.yaml

oc get pvc -n test test-web-pvc -o yaml >
/opt/okd/backups/test/02-pvc.test-web-pvc.yaml

oc get configmap -n test test-web-app -o yaml >
/opt/okd/backups/test/03-configmap.test-web-app.yaml

oc get deployment -n test test-web -o yaml >
/opt/okd/backups/test/04-deployment.test-web.yaml

oc get service -n test test-web -o yaml >
/opt/okd/backups/test/05-service.test-web.yaml

oc get route -n test test-web -o yaml >
/opt/okd/backups/test/06-route.test-web.yaml

oc get all -n test -o wide >
/opt/okd/backups/test/07-runtime-status.test.txt

oc get pvc -n test -o wide >
/opt/okd/backups/test/08-runtime-pvc.test.txt

oc describe pvc -n test test-web-pvc >
/opt/okd/backups/test/09-describe-pvc.test-web-pvc.txt
```

```
oc describe pod -n test -l app=test-web >
/opt/okd/backups/test/10-describe-pod.test-web.txt
```

Also keep the clean source manifests:

```
cp -a /tmp/test-web /opt/okd/backups/test/manifests
```

Back up tenant projects and groups:

```
mkdir -p /opt/okd/backups/projects

oc get project yabi -o yaml >
/opt/okd/backups/projects/01-project.yabi.yaml

oc get project nadir -o yaml >
/opt/okd/backups/projects/02-project.nadir.yaml

oc get group yabi -o yaml >
/opt/okd/backups/projects/03-group.yabi.yaml

oc get group nadir -o yaml >
/opt/okd/backups/projects/04-group.nadir.yaml
```

Back up authentication-related objects:

```
mkdir -p /opt/okd/backups/auth

oc get oauth cluster -o yaml >
/opt/okd/backups/auth/01-oauth.cluster.yaml

cp /opt/okd/backups/htpasswd/htpasswd
/opt/okd/backups/auth/02-htpasswd.users

oc get group nadir -o yaml > /opt/okd/backups/auth/03-group.nadir.yaml

oc get group yabi -o yaml > /opt/okd/backups/auth/04-group.yabi.yaml

oc get group test -o yaml > /opt/okd/backups/auth/05-group.test.yaml
```

The files generated by *`oc get -o yaml`* include runtime fields such as *`uid`*, *`resourceVersion`*, and *`status`*. These are useful for backup and documentation, but clean source manifests should also be kept for redeployment.

31 Daily Operations Cheat Sheet: OpenShift and Kubernetes Equivalents

This section provides a compact reference for common day-to-day operations on the OKD cluster. Each task is shown with the OpenShift command and, where possible, the closest Kubernetes equivalent.

Some commands operate on standard Kubernetes resources and are nearly identical with *`oc`* and *`kubectl`*. Others rely on OpenShift-specific features such as Projects, Routes, Security Context Constraints, OpenShift groups, OAuth login, or *`oc adm`* helpers. In those cases, the Kubernetes column shows the closest equivalent or indicates that no direct upstream Kubernetes equivalent exists.

Task	OpenShift	Kubernetes
List nodes	oc get nodes	kubectl get nodes
List pods	oc get pods -n <ns>	kubectl get pods -n <ns>
Apply manifest	oc apply -f file.yaml	kubectl apply -f file.yaml
Create project/namespace	oc new-project <name>	kubectl create namespace <name>
Switch project	oc project <name>	kubectl config set-context --current --namespace=<name>
Check permissions	oc auth can-i <verb> <resource> -n <ns>	kubectl auth can-i <verb> <resource> -n <ns>
Create route	oc expose service/<svc> or Route YAML	Ingress YAML
View route	oc get route -n <ns>	kubectl get ingress -n <ns>
Add SCC	oc adm policy add-scc-to-user ...	No direct equivalent
Add group user	oc adm groups add-users ...	No direct equivalent CLI
Debug node	oc debug node/<node>	kubectl debug node/<node>
Watch pods	oc get pods -n <ns> -w	kubectl get pods -n <ns> -w

32 Updated Conclusions

The original OKD UPI installation on Proxmox was extended with fully working storage, authentication, tenant, and application deployment workflows.

The validated final state includes:

- OKD 4.21 installed on Proxmox using UPI
- Dedicated Ceph network connectivity on worker nodes through *`ens19`*
- External Ceph RBD pool *`okd-rbd`* integrated through Ceph-CSI
- StorageClass *`okd-rbd`* available to applications
- Successful end-to-end PVC provisioning, pod attachment, file write, and persistence validation
- HTPasswd-based local authentication
- User/group/project onboarding for *`test`*, *`yabi`*, and *`nadir`* tenants
- A working web application demonstrating persistent storage on Ceph RBD
- Backup procedures for Ceph-CSI, authentication, projects, groups, and application manifests

The cluster is now usable not only as an installed OKD platform, but also as a practical multi-tenant environment where users can deploy applications and request persistent Ceph-backed storage through normal Kubernetes/OpenShift objects.

33 Rebuilding the Test Application as a Non-Admin User

A later operational validation was performed to ensure that the demo application could be deleted and recreated by a normal project user rather than by *system:admin*.

The user used for the validation was:

```
massimo.costantini
```

The goal was to confirm that a tenant user belonging to the appropriate OpenShift group could manage application resources in the *test* project without cluster-admin privileges.

OpenShift command:

```
oc whoami

oc project
```

Expected result after switching to the normal user context:

```
massimo.costantini
Already on project "test" on server "https://api.okd.ia2.inaf.it:6443"
```

Kubernetes equivalent:

```
kubectl config current-context

kubectl config view --minify
```

oc whoami is OpenShift-specific, but *kubectl config view --minify* can be used to inspect the current Kubernetes user context.

Before recreating the application, the normal user verified that the required operations were permitted in the test namespace, checking Project-Level permissions.

OpenShift command:

```
oc auth can-i create deployment -n test

oc auth can-i create route -n test

oc auth can-i create pvc -n test
```

Expected output:

```
yes
yes
yes
```

Kubernetes equivalent:

```
kubectl auth can-i create deployment -n test

kubectl auth can-i create pvc -n test
```

There is no Kubernetes equivalent for OpenShift Route; the equivalent external exposure object in Kubernetes is usually *Ingress*:

```
kubectl auth can-i create ingress -n test
```

The application resources were deleted in dependency order. The PVC was also deleted to force a full re-creation of the persistent volume claim.

OpenShift command:

```
oc project test

oc delete route test-web --ignore-not-found

oc delete svc test-web --ignore-not-found

oc delete deployment test-web --ignore-not-found

oc delete configmap test-web-app --ignore-not-found

oc delete pvc test-web-pvc --ignore-not-found
```

Kubernetes equivalent:

```
kubectl config set-context --current --namespace=test

kubectl delete ingress test-web --ignore-not-found

kubectl delete svc test-web --ignore-not-found

kubectl delete deployment test-web --ignore-not-found

kubectl delete configmap test-web-app --ignore-not-found

kubectl delete pvc test-web-pvc --ignore-not-found
```

Route is OpenShift-specific. In Kubernetes, use Ingress only if the cluster has an ingress controller configured.

A clean working directory was used for the manifest set:

```
mkdir -p /tmp/test-web
```

```
cd /tmp/test-web
```

The final clean manifest set consisted of:

```
01-configmap-test-web-app.yaml
```

```
02-pvc-test-web-pvc.yaml
```

```
03-deployment-test-web.yaml
```

```
04-service-test-web.yaml
```

```
05-route-test-web-https.yaml
```

The preferred permanent location for the tested manifests is:

```
/opt/okd/example/
```

For backup purposes:

```
/opt/okd/backups/test/manifests-https/
```

To recreate the demo application the clean manifests were applied:

```
oc apply -f 01-configmap-test-web-app.yaml

oc apply -f 02-pvc-test-web-pvc.yaml

oc apply -f 03-deployment-test-web.yaml

oc apply -f 04-service-test-web.yaml

oc apply -f 05-route-test-web-https.yaml
```

Kubernetes equivalent for standard resources:

```
kubectl apply -f 01-configmap-test-web-app.yaml

kubectl apply -f 02-pvc-test-web-pvc.yaml

kubectl apply -f 03-deployment-test-web.yaml

kubectl apply -f 04-service-test-web.yaml
```

For the OpenShift Route, Kubernetes requires an Ingress manifest instead of *`05-route-test-web-https.yaml`*.

During validation, the PVC remained temporarily in `Pending` with this event:

```
WaitForFirstConsumer: waiting for first consumer to be created before binding
```

This was expected because the *`okd-rbd`* StorageClass uses:

```
volumeBindingMode: WaitForFirstConsumer
```

The PVC is bound only after a pod that uses it is created and scheduled.

OpenShift command:

```
oc get pvc test-web-pvc -n test

oc describe pvc test-web-pvc -n test

oc get storageclass
```

Kubernetes equivalent:

```
kubectl get pvc test-web-pvc -n test

kubectl describe pvc test-web-pvc -n test

kubectl get storageclass
```

The correct operational sequence is therefore:

- Create the PVC
- Create the Deployment that consumes it
- Wait for the PVC and pod to become ready

OpenShift command:

```
oc get pvc,pods -n test -w
```

Kubernetes equivalent:

```
kubectl get pvc,pods -n test -w
```

For a functional validation, the OpenShift commands are:

```
oc get pods -n test

oc get pvc -n test

oc get svc -n test

oc get route -n test
```

Kubernetes equivalent:

```
kubectl get pods -n test

kubectl get pvc -n test

kubectl get svc -n test

kubectl get ingress -n test
```

Again, the application was accessed through the HTTPS route:

```
https://test-web-test.apps.okd.ia2.inaf.it
```

After submitting a message through the web form, persistence was verified with:

```
oc exec -n test deploy/test-web -- cat /data/messages.txt
```

Kubernetes equivalent:

```
kubectl exec -n test deploy/test-web -- cat /data/messages.txt
```

34 HTTPS Exposure with OpenShift Routes

The original demo application served HTTP on port `8080` inside the pod. HTTPS exposure was implemented at the OpenShift Router level, not inside the container.

To route the TLS edge termination a final HTTPS route manifest was applied:

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: test-web
  namespace: test
spec:
  to:
    kind: Service
    name: test-web
  port:
    targetPort: http
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: Redirect
```

OpenShift command:

```
oc apply -f 05-route-test-web-https.yaml
```

The same Route can also be created imperatively with:

```
oc create route edge test-web
    --service=test-web
    --port=http
    --insecure-policy=Redirect
    -n test
```

The declarative YAML manifest is preferred for this report because it is explicit, versionable, and includes the full desired Route configuration, including TLS edge termination and HTTP-to-HTTPS redirection.

The Route can also be recreated from scratch:

```
oc delete route test-web -n test --ignore-not-found
```

```
oc create route edge test-web
    --service=test-web
    --port=http
    --insecure-policy=Redirect
    -n test
```

Why is Port 443 not specified?

The user-facing TLS endpoint is provided by the OpenShift ingress router. Therefore:

```
Browser -> HTTPS 443 -> OpenShift Router -> HTTP 8080 -> Service -> Pod
```

The application container still listens on port `8080` and does not need certificates or a local HTTPS listener, the *`--port=http`* or *`targetPort`* refers to the Service port, not to the external router port.

OpenShift Route has no direct Kubernetes equivalent. The usual equivalent is an Ingress object, assuming that an ingress controller and TLS configuration are available.

Example with Kubernetes Ingress:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-web
  namespace: test
spec:
  tls:
  - hosts:
    - test-web-test.apps.okd.ia2.inaf.it
    secretName: test-web-tls
  rules:
  - host: test-web-test.apps.okd.ia2.inaf.it
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: test-web
            port:
              number: 8080
```

To apply:

```
kubectl apply -f ingress.yaml
```

Unlike OpenShift Routes, Kubernetes Ingress requires an ingress controller and usually a TLS secret or certificate manager.

35 Bastion-Based User Workflow

A decision was made to let users access the OKD environment from the bastion host instead of installing `oc` locally on their workstations.

This simplifies user onboarding because:

- *`oc`* is already installed on the bastion
- Users do not need to configure local clients
- Examples and templates can be stored centrally under */opt/okd/example*
- Access can be controlled through Linux users and OpenShift users separately

Users connect to the bastion via SSH:

```
ssh <username>@bastion.okd.ia2.inaf.it
```

They should immediately change the Linux password on first access:

```
passwd
```

Users then log in to OKD:

```
oc login https://api.okd.ia2.inaf.it:6443
    --username=<username>
    --insecure-skip-tls-verify=true
```

The password is requested interactively.

The *--insecure-skip-tls-verify=true* option bypasses local certificate validation on the bastion user environment. It is operationally convenient but less secure than distributing the cluster CA correctly.

Recommended future improvement:

- Install the OKD API CA in a system/user trust store
- Or provide a preconfigured kubeconfig with the CA embedded

To select the project, the OpenShift command is:

```
oc project <project-name>
```

Kubernetes equivalent:

```
kubectl config set-context --current --namespace=<namespace-name>
```

A user cannot run *oc config use-context* before logging in at least once. The context is created by *oc login*.

Correct first-time workflow:

```
oc login https://api.okd.ia2.inaf.it:6443
    --username=<username>
    --insecure-skip-tls-verify=true

oc project <project-name>
```

After a successful login, contexts can be listed with:

```
oc config get-contexts
```

Kubernetes equivalent:

```
kubectl config get-contexts
```

36 Managing HTPasswd Users, Passwords, and Backups

The cluster uses the OpenShift HTPasswd identity provider named *localusers* backed by the secret *htpasswd-secret* in the *openshift-config* namespace.

To inspect the OAuth configuration, the OpenShift command is:

```
oc get oauth cluster -o yaml
```

Expected relevant configuration:

```
spec:
  identityProviders:
  - htpasswd:
      fileData:
        name: htpasswd-secret
    mappingMethod: claim
    name: localusers
    type: HTPasswd
```

There is no direct Kubernetes equivalent for OpenShift OAuth configuration.

To inspect the active HTPasswd secret, the OpenShift command is:

```
oc get secret -n openshift-config htpasswd-secret
    -o jsonpath='{.data.htpasswd}' | base64 -d
```

This verifies the file actually used by the cluster. It may differ from a local backup copy.

Kubernetes equivalent for decoding a generic secret:

```
kubectl get secret -n openshift-config htpasswd-secret
    -o jsonpath='{.data.htpasswd}' | base64 -d
```

The namespace and meaning of the secret are OpenShift-specific.

The canonical local HTPasswd source file is:

```
/opt/okd/backups/htpasswd/htpasswd
```

To create or update a user password:

```
htpasswd -Bb /opt/okd/backups/htpasswd/htpasswd cristiano.urban
'<TEMP_PASSWORD>'
```

For interactive password entry:

```
htpasswd -B /opt/okd/backups/htpasswd/htpasswd cristiano.urban
```

Then update the OpenShift OAuth secret:

```
oc create secret generic htpasswd-secret
    --from-file=htpasswd=/opt/okd/backups/htpasswd/htpasswd
    -n openshift-config
    --dry-run=client -o yaml | oc apply -f -
```

To verify:

```
oc get secret -n openshift-config htpasswd-secret
    -o jsonpath='{.data.htpasswd}' | base64 -d | grep cristiano.urban
```

To test the user login:

```
oc login https://api.okd.ia2.inaf.it:6443
    --username=cristiano.urban
    --insecure-skip-tls-verify=true

oc whoami
```

Expected:

```
cristiano.urban
```

Example: add a user to the common `test` group:

```
oc adm groups add-users test cristiano.urban
```

To verify:

```
oc get group test -o yaml

oc describe group test
```

There is no Kubernetes equivalent to *`oc adm groups add-users`* for local OpenShift groups. In Kubernetes, group membership is normally asserted by the external identity provider and consumed by RBAC.

After any HTPasswd or group change, update the authentication backup files:

```
mkdir -p /opt/okd/backups/auth

cp /opt/okd/backups/htpasswd/htpasswd
/opt/okd/backups/auth/02-htpasswd.users

oc get secret htpasswd-secret -n openshift-config -o yaml >
/opt/okd/backups/auth/02-secret.htpasswd-secret.yaml

oc get oauth cluster -o yaml >
/opt/okd/backups/auth/01-oauth.cluster.yaml

oc get group test -o yaml > /opt/okd/backups/auth/05-group.test.yaml
```

For additional groups:

```
oc get group ai -o yaml > /opt/okd/backups/auth/06-group.ai.yaml

oc get group nadir -o yaml > /opt/okd/backups/auth/03-group.nadir.yaml

oc get group yabi -o yaml > /opt/okd/backups/auth/04-group.yabi.yaml
```

Secrets and password hash files must be protected with restrictive permissions.

37 Linux Users on the Bastion Host

Linux accounts on the bastion are separate from OpenShift HTPasswd users. The same username convention is used for convenience, but the two identities are technically independent.

To create a local bastion user, for example:

```
useradd -m -s /bin/bash cristiano.urban

passwd cristiano.urban
```

To verify:

```
id cristiano.urban

getent passwd cristiano.urban
```

For an optional SSH Key-Based Access, create *.ssh*:

```
mkdir -p /home/cristiano.urban/.ssh

chmod 700 /home/cristiano.urban/.ssh

touch /home/cristiano.urban/.ssh/authorized_keys

chmod 600 /home/cristiano.urban/.ssh/authorized_keys

chown -R cristiano.urban:cristiano.urban /home/cristiano.urban/.ssh
```

Add the user public key:

```
echo 'ssh-ed25519 AAAA... user-key-comment' >>
/home/cristiano.urban/.ssh/authorized_keys

chown cristiano.urban:cristiano.urban
    /home/cristiano.urban/.ssh/authorized_keys
```

For an optional bastion access group:

```
groupadd bastionusers

usermod -aG bastionusers cristiano.urban
```

In */etc/ssh/sshd_config*:

```
AllowGroups bastionusers
```

To reload SSH:

```
systemctl reload sshd
```

38 Project and Group Access Model

The cluster uses OpenShift groups mapped to project-level role bindings, this is the final intended access model:

Project	Group	Members	Role
test	test	all enabled local users	edit
ai	ai	giovanni.lacopo	edit
nadir	nadir	giacomo.coran	edit
yabi	yabi	andrea.bignamini	edit

system:admin remains cluster-admin and therefore can access all projects independently of tenant RBAC.

To create project and group for ai, the OpenShift commands are:

```
oc adm groups new ai || true
```

```
oc adm groups add-users ai giovanni.lacopo
```

```
oc new-project ai --display-name="AI Project" --description="Project
    for AI workloads"
```

```
oc adm policy add-role-to-group edit ai -n ai
```

Kubernetes equivalent:

```
kubectl create namespace ai
```

RBAC RoleBinding equivalent:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: edit-to-ai-group
  namespace: ai
subjects:
- kind: Group
  name: ai
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: edit
  apiGroup: rbac.authorization.k8s.io
```

To apply:

```
kubectl apply -f rolebinding-ai.yaml
```

Group membership itself is external to Kubernetes unless provided by the authentication system.

To verify RoleBinding in a project, the OpenShift command is:

```
oc describe rolebinding edit -n test
```

Expected:

```
Role:
  Kind:  ClusterRole
  Name:  edit
Subjects:
  Kind   Name
  Group  test
```

Kubernetes equivalent:

```
kubectl describe rolebinding edit -n test
```

The rolebinding object is a standard Kubernetes RBAC object, although OpenShift creates and manages some bindings through *`oc adm`* helpers.

When using impersonation, *`--as=<user>`* alone is not enough if permissions are granted through groups. The group must also be supplied.

Example:

```
oc auth can-i get pods -n test --as=giovanni.lacopo --as-group=test
```

Kubernetes equivalent:

```
kubectl auth can-i get pods -n test --as=giovanni.lacopo
    --as-group=test
```

Validated test command set:

```
for u in massimo.costantini giovanni.lacopo cristiano.urban
         giacomo.coran andrea.bignamini; do
    echo -n "test $u: "
    oc auth can-i get pods -n test --as="$u" --as-group=test
done
echo -n "ai giovanni.lacopo: "

oc auth can-i get pods -n ai --as=giovanni.lacopo --as-group=ai

echo -n "ai massimo.costantini: "

oc auth can-i get pods -n ai --as=massimo.costantini --as-group=test

echo -n "ai giacomo.coran: "

oc auth can-i get pods -n ai --as=giacomo.coran --as-group=nadir

echo -n "ai andrea.bignamini: "

oc auth can-i get pods -n ai --as=andrea.bignamini --as-group=yabi
echo -n "nadir giacomo.coran: "

oc auth can-i get pods -n nadir --as=giacomo.coran --as-group=nadir

echo -n "nadir massimo.costantini: "

oc auth can-i get pods -n nadir --as=massimo.costantini --as-group=test

echo -n "nadir giovanni.lacopo: "

oc auth can-i get pods -n nadir --as=giovanni.lacopo --as-group=ai
```

```
echo -n "nadir andrea.bignamini: "

oc auth can-i get pods -n nadir --as=andrea.bignamini --as-group=yabi
echo -n "yabi andrea.bignamini: "

oc auth can-i get pods -n yabi --as=andrea.bignamini --as-group=yabi

echo -n "yabi massimo.costantini: "

oc auth can-i get pods -n yabi --as=massimo.costantini --as-group=test

echo -n "yabi giovanni.lacopo: "

oc auth can-i get pods -n yabi --as=giovanni.lacopo --as-group=ai

echo -n "yabi giacomo.coran: "

oc auth can-i get pods -n yabi --as=giacomo.coran --as-group=nadir
```

Expected final result:

```
test massimo.costantini: yes
test giovanni.lacopo: yes
test cristiano.urban: yes
test giacomo.coran: yes
test andrea.bignamini: yes
ai giovanni.lacopo: yes
ai massimo.costantini: no
ai giacomo.coran: no
ai andrea.bignamini: no
nadir giacomo.coran: yes
nadir massimo.costantini: no
```

```
nadir giovanni.lacopo: no

nadir andrea.bignamini: no

yabi andrea.bignamini: yes

yabi massimo.costantini: no

yabi giovanni.lacopo: no

yabi giacomo.coran: no
```

If for example *ai giovanni.lacopo* returns *no*, add the missing binding:

```
oc adm policy add-role-to-group edit ai -n ai
```

The `edit` role allows a user to operate normally inside a project, including:

- Creating and modifying Deployments
- Creating Services
- Creating Routes, in OpenShift
- Creating ConfigMaps and Secrets
- Creating PVCs
- Viewing pods and logs
- Scaling and deleting application workloads

The *admin role* is only needed if the user must also manage access delegation and project-level RBAC.

39 Parametrized Application Example for Multiple Users

The fixed-name manifest set is useful as a technical example, but it should not be applied unchanged by multiple users in the shared *test* project because all users would attempt to create the same object names.

The recommended multi-user approach is to provide an OpenShift Template with an *APP_NAME* parameter.

Why not use shell variables directly in YAML?

Kubernetes and OpenShift YAML manifests do not automatically expand shell variables such as *${APP_NAME}* when used with:

```
oc apply -f file.yaml
```

For variable substitution, use one of:

- OpenShift Templates with *oc process*
- *envsubst*
- Kustomize
- Helm

For this environment, the OpenShift Template is the simplest native solution.

To create an OpenShift template file, the command is:

```
/opt/okd/example/test-web-template.yaml
```

Deploy users with:

```
oc project test

oc process -f /opt/okd/example/test-web-template.yaml

    -p APP_NAME=<username>-web

    -p NAMESPACE=test

| oc apply -f -
```

Example:

```
oc process -f /opt/okd/example/test-web-template.yaml
    -p APP_NAME=test-web
    -p NAMESPACE=test
| oc apply -f -
```

There is no OpenShift Template equivalent in Kubernetes. Typical alternatives are:

1. Envsubst

   ```
   envsubst < manifest.yaml | kubectl apply -f -
   ```

2. Kubectl

   ```
   kubectl apply -k <kustomize-directory>
   ```

3. Helm

   ```
   helm install <release-name> <chart-directory> --namespace <namespace>
   ```

The recommended parameters are:

Parameter	Example	Purpose
APP_NAME	test-web	Prefix/name for ConfigMap, PVC, Deployment, Service, Route
NAMESPACE	test	Target OpenShift project
STORAGE_SIZE	2Gi	PVC size
STORAGE_CLASS	okd-rbd	Ceph-backed StorageClass

For resource naming pattern, an *APP_NAME* of *test-web*, the template should create:

Resource	Name
ConfigMap	test-web-app
PVC	test-web-pvc
Deployment	test-web
Service	test-web
Route	test-web

This avoids conflicts in the shared *test* project.

To retrieve the route:

```
oc get route <APP_NAME>
```

To verify the persistent file:

```
oc exec deploy/<APP_NAME> -- cat /data/messages.txt
```

Kubernetes equivalent for an Ingress-based deployment:

```
kubectl get ingress <APP_NAME> -n <namespace>

kubectl exec -n <namespace> deploy/<APP_NAME> -- cat /data/messages.txt
```

40 End-User Operating Procedure

The following short procedure is suitable for users who access the cluster through the bastion host.

1. Login to bastion

```
ssh <username>@bastion.okd.ia2.inaf.it
```

2. Change the Linux password on first login

```
Passwd
```

3. Login to OKD

```
oc login https://api.okd.ia2.inaf.it:6443 --username=<username>
    --insecure-skip-tls-verify=true
```

4. Check the authenticated OpenShift user

```
oc whoami
```

5. Select a project

```
oc project <project-name>
```

Examples:

```
oc project test

oc project ai

oc project nadir

oc project yabi
```

The example manifests are stored centrally:

```
cd /opt/okd/example

ls -l
```

They are examples and should not be applied unchanged by multiple users in the same shared project.

Recommended shared-project deployment:

```
oc process -f /opt/okd/example/test-web-template.yaml
    -p APP_NAME=<username>-web
    -p NAMESPACE=<project-name>
| oc apply -f -
```

Validate Resources:

```
oc get pods,svc,route,pvc
```

Kubernetes equivalent:

```
kubectl get pods,svc,ingress,pvc -n <namespace>
```

Open the application, get the HTTPS route:

```
oc get route <APP_NAME>
```

Open the route host in a browser.

Validate Persistent Write:

```
oc exec deploy/<APP_NAME> -- cat /data/messages.txt
```

Kubernetes equivalent:

```
kubectl exec -n <namespace> deploy/<APP_NAME> -- cat /data/messages.txt
```

41 Daily Operations Cheat Sheet Addendum

This addendum complements the previous operations cheat sheet with commands related to login, context management, authentication, group-based access, RBAC validation, HTTPS exposure, and common workload operations.

Where possible, each OpenShift command is paired with the closest Kubernetes equivalent. Some operations, such as *`oc login`*, OpenShift OAuth/HTPasswd management, OpenShift groups, Routes, and SCC-related configuration, are specific to OpenShift and do not have a direct Kubernetes equivalent.

Task	OpenShift	Kubernetes
Login to cluster	oc login https://api... --username=<user>	Depends on kubeconfig/auth plugin
Show current user	oc whoami	kubectl config view --minify
Switch to project	oc project <project>	kubectl config set-context --current --namespace=<ns>
List contexts	oc config get-contexts	kubectl config get-contexts
Use context	oc config use-context <ctx>	kubectl config use-context <ctx>
Create HTPasswd user	htpasswd ... + update OAuth secret	No direct equivalent
Update HTPasswd secret	oc create secret ... -n openshift-config ... | oc apply -f -	kubectl create secret ... | kubectl apply -f -
Add user to OpenShift group	oc adm groups add-users <group> <user>	No direct equivalent
Bind role to group	oc adm policy add-role-to-group edit <group> -n <ns>	kubectl create rolebinding ... --clusterrole=edit --group=<group> -n <ns>
Verify access through group	oc auth can-i get pods -n <ns> --as=<user> --as-group=<group>	Same with kubectl auth can-i ...
Create HTTPS endpoint	Route with tls.termination=edge	Ingress with TLS
Execute command in deployment pod	oc exec deploy/<name> -- <cmd>	kubectl exec deploy/<name> -- <cmd>
Restart deployment	oc rollout restart deployment/<name>	kubectl rollout restart deployment/<name>

42 Application Operations and Workload Management

The following chapters close the operational loop of the OKD-on-Proxmox installation by collecting practical day-to-day commands for administrators and tenant users.

The intent is to answer common operational questions such as:

- Is my application running?
- On which node is it running?
- How do I see logs, events, routes, services, and storage?
- How do I restart, scale[50], or redeploy it?
- How do I move a workload away from a node?
- What happens if a worker node dies?
- What should an administrator do during node failure recovery?
- How is OKD updated safely?

Where possible, commands are shown both in OpenShift form and in Kubernetes form.

Most commands below assume one of the following contexts.

Administrator context:

```
export KUBECONFIG=/path/to/okd-install/auth/kubeconfig

oc whoami
```

Tenant user context from the bastion:

```
oc login https://api.okd.ia2.inaf.it:6443
    --username=<username>
    --insecure-skip-tls-verify=true

oc project <project-name>
```

[50] In Kubernetes and OpenShift, scaling a Deployment means changing the desired number of pod replicas managed by that Deployment. Scaling up increases the number of running application instances and can improve capacity or resilience for stateless applications. Scaling down reduces the number of instances and can reduce resource usage. In this report, the demo application uses a ReadWriteOnce Ceph RBD PVC, so scaling above one replica is shown as an operational command but is not a recommended pattern for multiple pods writing to the same persistent storage.

Kubernetes equivalent:

```
kubectl config current-context

kubectl config set-context --current --namespace=<namespace>
```

For examples, the demo application is assumed to be:

```
Namespace/Project:   test
Deployment:          test-web
Label:               app=test-web
Service:             test-web
Route:               test-web
PVC:                 test-web-pvc
StorageClass:        okd-rbd
```

For parametrized user deployments, replace test-web with the chosen APP_NAME.

The following commands provide a first health snapshot of the cluster, covering cluster version, operator health, node readiness, MachineConfigPool status, and non-running pods.

OpenShift commands:

```
oc get clusterversion

oc get co

oc get nodes -o wide

oc get mcp

oc get pods -A | grep -v Running
```

Kubernetes equivalent:

```
kubectl get nodes -o wide

kubectl get pods -A | grep -v Running
```

Notes:

- *`ClusterVersion`*, *`ClusterOperator`*, and *`MachineConfigPool`* are OpenShift-specific.
- Standard Kubernetes does not have direct equivalents for *`oc get co`*, *`oc get clusterversion`*, or *`oc get mcp`*.

A healthy OKD cluster should generally show:

```
ClusterOperators: AVAILABLE=True, DEGRADED=False

Nodes:            Ready

MCPs:             UPDATED=True, DEGRADED=False
```

To verify that an application is running the OpenShift commands are:

```
oc project test

oc get deploy test-web

oc get pods -l app=test-web -o wide

oc get svc test-web

oc get route test-web

oc get pvc test-web-pvc
```

Kubernetes equivalent:

```
kubectl get deployment test-web -n test

kubectl get pods -n test -l app=test-web -o wide

kubectl get svc test-web -n test

kubectl get ingress test-web -n test

kubectl get pvc test-web-pvc -n test
```

Useful compact view:

```
oc get deploy,pods,svc,route,pvc -n test
```

Kubernetes equivalent:

```
kubectl get deploy,pods,svc,ingress,pvc -n test
```

Interpretation:

Object	**What it confirms**
Deployment	Desired and available replicas
Pod	Actual running container instance
Service	Internal stable network endpoint
Route/Ingress	External HTTP/HTTPS exposure
PVC	Persistent storage claim state

To find the node where a pod is running, the OpenShift command is:

```
oc get pods -n test -l app=test-web -o wide
```

Kubernetes equivalent:

```
kubectl get pods -n test -l app=test-web -o wide
```

The *NODE* column shows the worker node.

To print only pod name and node:

```
oc get pods -n test -l app=test-web
    -o custom-columns='POD:.metadata.name,
                       NODE:.spec.nodeName,
                       STATUS:.status.phase'
```

Kubernetes equivalent:

```
kubectl get pods -n test -l app=test-web \
    -o custom-columns='POD:.metadata.name,
                       NODE:.spec.nodeName,
                       STATUS:.status.phase'
```

To show all pods running on a specific node:

```
oc get pods -A -o wide
    --field-selector spec.nodeName=wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl get pods -A -o wide
    --field-selector spec.nodeName=wk-1.okd.ia2.inaf.it
```

To inspect logs, events, and runtime details:

```
oc logs -n test deploy/test-web
```

Kubernetes equivalent:

```
kubectl logs -n test deploy/test-web
```

To follow the logs:

```
oc logs -n test deploy/test-web -f
```

Kubernetes equivalent:

```
kubectl logs -n test deploy/test-web -f
```

To describe[51] the deployment:

```
oc describe deployment test-web -n test
```

Kubernetes equivalent:

```
kubectl describe deployment test-web -n test
```

To describe the pod:

```
POD=$(oc get pod -n test -l app=test-web
    -o jsonpath='{.items[0].metadata.name}')
oc describe pod "$POD" -n test
```

Kubernetes equivalent:

```
POD=$(kubectl get pod -n test -l app=test-web
    -o jsonpath='{.items[0].metadata.name}')
kubectl describe pod "$POD" -n test
```

[51] In Kubernetes and OpenShift, the describe command displays a detailed human-readable view of a resource. Unlike get, which mainly shows a compact summary, describe includes configuration details, status information, conditions, related objects, and recent events. It is useful for troubleshooting Deployments, Pods, PVCs, Nodes, and other resources.

Recent namespace events:

```
oc get events -n test --sort-by=.lastTimestamp
```

Kubernetes equivalent:

```
kubectl get events -n test --sort-by=.lastTimestamp
```

Cluster-wide non-running pods:

```
oc get pods -A --field-selector=status.phase!=Running
```

Kubernetes equivalent:

```
kubectl get pods -A --field-selector=status.phase!=Running
```

To verify external access to the application through the OpenShift Route:

```
oc get route test-web -n test
```

To get only the route host:

```
oc get route test-web -n test -o jsonpath='{.spec.host}{"\n"}'
```

To test the HTTPS route from the bastion:

```
ROUTE_HOST=$(oc get route test-web -n test -o jsonpath='{.spec.host}')

curl -k https://${ROUTE_HOST}/
```

Kubernetes equivalent with Ingress:

```
kubectl get ingress test-web -n test

kubectl get ingress test-web -n test
    -o jsonpath='{.spec.rules[0].host}{"\n"}'
```

To test the internal Service DNS from a temporary pod:

```
oc run curl-test -n test --rm -it --restart=Never
    --image=registry.access.redhat.com/ubi9/ubi-minimal
    -- curl -s http://test-web:8080/
```

Kubernetes equivalent:

```
kubectl run curl-test -n test --rm -it --restart=Never
    --image=curlimages/curl
    -- curl -s http://test-web:8080/
```

To verify persistent storage and data, check PVC and PV:

```
oc get pvc -n test test-web-pvc -o wide

oc get pv

oc describe pvc -n test test-web-pvc
```

Kubernetes equivalent:

```
kubectl get pvc -n test test-web-pvc -o wide

kubectl get pv

kubectl describe pvc -n test test-web-pvc
```

To read the persisted file from the demo application:

```
oc exec -n test deploy/test-web -- cat /data/messages.txt
```

Kubernetes equivalent:

```
kubectl exec -n test deploy/test-web -- cat /data/messages.txt
```

To confirm persistence across pod recreation:

```
oc delete pod -n test -l app=test-web

oc get pods -n test -w

oc exec -n test deploy/test-web -- cat /data/messages.txt
```

Kubernetes equivalent:

```
kubectl delete pod -n test -l app=test-web

kubectl get pods -n test -w

kubectl exec -n test deploy/test-web -- cat /data/messages.txt
```

For Ceph-CSI troubleshooting:

```
oc get pods -n ceph-csi-rbd -o wide

oc logs -n ceph-csi-rbd deployment/csi-rbdplugin-provisioner
    -c csi-provisioner

oc logs -n ceph-csi-rbd daemonset/csi-rbdplugin -c csi-rbdplugin
```

Kubernetes equivalent:

```
kubectl get pods -n ceph-csi-rbd -o wide

kubectl logs -n ceph-csi-rbd deployment/csi-rbdplugin-provisioner
    -c csi-provisioner

kubectl logs -n ceph-csi-rbd daemonset/csi-rbdplugin -c csi-rbdplugin
```

To inspect volume attachments:

```
oc get volumeattachments[52]
```

[52] VolumeAttachment is a Kubernetes storage object that represents the attachment of a persistent volume to a specific node. It is useful for troubleshooting CSI attach/detach problems, especially with block storage such as Ceph RBD.

Kubernetes equivalent:

```
kubectl get volumeattachments
```

To restart an application:

```
oc rollout restart deployment/test-web -n test

oc rollout status deployment/test-web -n test
```

Kubernetes equivalent:

```
kubectl rollout restart deployment/test-web -n test

kubectl rollout status deployment/test-web -n test
```

To scale up:

```
oc scale deployment/test-web -n test --replicas=2
```

Kubernetes equivalent:

```
kubectl scale deployment/test-web -n test --replicas=2
```

To scale down:

```
oc scale deployment/test-web -n test --replicas=1
```

Kubernetes equivalent:

```
kubectl scale deployment/test-web -n test --replicas=1
```

To view rollout history:

```
oc rollout history deployment/test-web -n test
```

Kubernetes equivalent:

```
kubectl rollout history deployment/test-web -n test
```

To undo the last rollout:

```
oc rollout undo deployment/test-web -n test
```

Kubernetes equivalent:

```
kubectl rollout undo deployment/test-web -n test
```

Important storage notes:

- The demo application uses a *`ReadWriteOnce`* Ceph RBD PVC
- A *`ReadWriteOnce`* PVC can normally be mounted read-write by only one node at a time
- Scaling the demo deployment above one replica may fail or may schedule only one usable replica unless the storage backend and access mode support multi-writer behavior
- For multiple replicas that all write shared files, use a *`ReadWriteMany`* storage backend such as CephFS, not RBD

To update the container image used by the Deployment:

```
oc set image deployment/test-web
    web=registry.access.redhat.com/ubi9/python-311:latest -n test

oc rollout status deployment/test-web -n test
```

Kubernetes equivalent:

```
kubectl set image deployment/test-web
    web=registry.access.redhat.com/ubi9/python-311:latest -n test

kubectl rollout status deployment/test-web -n test
```

To edit a deployment interactively[53]:

```
oc edit deployment/test-web -n test
```

Kubernetes equivalent:

```
kubectl edit deployment/test-web -n test
```

To re-apply clean manifests, first distinguish between static Kubernetes/OpenShift manifests and OpenShift Templates.

[53] To edit a Deployment interactively means to open the live Deployment definition in a terminal editor, modify its YAML configuration, and save it back to the cluster. This is useful for quick administrative changes or troubleshooting, but for reproducible operations it is better to update the source manifest and re-apply it with oc apply.

If the application is stored as static manifests, re-apply the manifest directory or the individual YAML files:

```
oc apply -f /opt/okd/example/test-web/
```

Kubernetes equivalent:

```
kubectl apply -f /opt/okd/example/test-web/ -n test
```

If the application is stored as an OpenShift Template, applying the template file directly only creates or updates the Template object. It does not instantiate the application resources.

To recreate or update the application resources from the template, process the template first and pipe the generated manifests to `oc apply`:

```
oc process -f /opt/okd/example/test-web-template.yaml
   -p APP_NAME=test-web
   -p NAMESPACE=test
| oc apply -f -
```

There is no direct Kubernetes equivalent for OpenShift Templates. With Kubernetes, use static manifests, Kustomize, Helm, or envsubst-based rendering.

How to “move” an application to another node?

In Kubernetes and OKD, a pod is not normally moved live like a VM.

The usual model is:

- make the old pod disappear or become unschedulable
- let the Deployment/ReplicaSet create a replacement pod
- allow the scheduler to place the replacement on an eligible node

1. Delete the pod and let the scheduler recreate it

OpenShift commands:

```
oc delete pod -n test -l app=test-web

oc get pods -n test -l app=test-web -o wide -w
```

Kubernetes equivalent:

```
kubectl delete pod -n test -l app=test-web

kubectl get pods -n test -l app=test-web -o wide -w
```

This does not guarantee a specific target node. It only asks the controller to recreate the pod.

2. Temporarily cordon[54] a node

Prevent new pods from being scheduled on a node:

```
oc adm cordon wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl cordon wk-1.okd.ia2.inaf.it
```

[54] To cordon a node means to mark it as unschedulable. Existing pods already running on the node are not removed, but the scheduler will not place new pods on that node. This is useful before maintenance or when testing workload relocation.

Then recreate the pod:

```
oc delete pod -n test -l app=test-web

oc get pods -n test -l app=test-web -o wide -w
```

Kubernetes equivalent:

```
kubectl delete pod -n test -l app=test-web

kubectl get pods -n test -l app=test-web -o wide -w
```

Re-enable scheduling later:

```
oc adm uncordon wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl uncordon wk-1.okd.ia2.inaf.it
```

3. Drain[55] a node for maintenance

Drain evacuates eligible pods from a node:

```
oc adm drain wk-1.okd.ia2.inaf.it
    --ignore-daemonsets
    --delete-emptydir-data
```

Kubernetes equivalent:

```
kubectl drain wk-1.okd.ia2.inaf.it
    --ignore-daemonsets
    --delete-emptydir-data
```

After maintenance:

```
oc adm uncordon wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl uncordon wk-1.okd.ia2.inaf.it
```

[55] To drain a node means to evict eligible pods from that node so that the scheduler and workload controllers can recreate them on other available nodes. This is commonly used before maintenance or controlled node shutdown. Draining does not move pods live like virtual machines; it terminates pods and relies on controllers such as Deployments, ReplicaSets, StatefulSets, or DaemonSets to recreate them according to their rules. DaemonSet pods are normally ignored, and pods using local emptyDir data lose that temporary data when they are removed.

Important notes:

- Draining does not move standalone pods that are not managed by a controller
- Pods managed by Deployments, ReplicaSets, ReplicationControllers, StatefulSets, Jobs, or similar controllers can be recreated elsewhere, subject to scheduling and storage constraints
- DaemonSet[56] pods are ignored because each node is expected to run its own DaemonSet copy
- Pods using local *`emptyDir`*[57] data lose that data when recreated
- Pods using RBD *`ReadWriteOnce`* storage may need time for detach/reattach

4. Pin[58] a workload to a specific node

This is usually not recommended for general workloads, but it is useful for tests.

Label a node:

```
oc label node wk-2.okd.ia2.inaf.it test-web-target=true
```

Kubernetes equivalent:

```
kubectl label node wk-2.okd.ia2.inaf.it test-web-target=true
```

[56] Kubernetes and OpenShift provide several workload controllers that manage pods according to different operational models. A Deployment is commonly used for stateless applications and manages interchangeable pod replicas through ReplicaSets. A DaemonSet runs one pod on each eligible node, typically for node-level services such as monitoring, logging, networking, or storage plugins. A StatefulSet manages stateful applications that require stable pod identities, ordered startup or shutdown, and persistent storage associated with each replica. A Job runs pods until a task completes successfully, while a CronJob runs Jobs on a schedule. In normal application deployment, users rarely create bare pods directly; they usually create one of these controllers and let it manage pod lifecycle, replacement, and scheduling.

[57] emptyDir is temporary pod-local storage created when a pod starts and deleted when the pod is removed. Data stored in emptyDir does not survive pod deletion or rescheduling.

[58] To pin a workload to a specific node means to constrain the scheduler so that the workload can run only on a selected node or set of nodes. This is usually done by labeling the target node and adding a matching nodeSelector to the Deployment. It is useful for tests, troubleshooting, or special workloads, but it should be avoided for general applications because it reduces scheduling flexibility and can make the workload less resilient to node failures.

Patch the deployment with a node selector:

```
oc patch deployment test-web -n test --type=merge -p '
    {
      "spec": {
        "template": {
          "spec": {
            "nodeSelector": {
              "test-web-target": "true"
            }
          }
        }
      }
    }'
```

Kubernetes equivalent:

```
kubectl patch deployment test-web -n test --type=merge -p '
    {
      "spec": {
        "template": {
          "spec": {
            "nodeSelector": {
              "test-web-target": "true"
            }
          }
        }
      }
```

```
    }'
```

Watch the rollout:

```
oc rollout status deployment/test-web -n test

oc get pods -n test -l app=test-web -o wide
```

Kubernetes equivalent:

```
kubectl rollout status deployment/test-web -n test

kubectl get pods -n test -l app=test-web -o wide
```

Remove the node selector:

```
oc patch deployment test-web -n test --type=json
    -p='[{"op":"remove","path":"/spec/template/spec/nodeSelector"}]'
```

Kubernetes equivalent:

```
kubectl patch deployment test-web -n test --type=json
    -p='[{"op":"remove","path":"/spec/template/spec/nodeSelector"}]'
```

Remove the node label:

```
oc label node wk-2.okd.ia2.inaf.it test-web-target-
```

Kubernetes equivalent:

```
kubectl label node wk-2.okd.ia2.inaf.it test-web-target-
```

43 Failure Recovery

If a worker node dies, several things happen.

For a Deployment-managed application:

- The node eventually becomes `NotReady`
- Pods bound to that node become unavailable
- Kubernetes marks pods on the lost node as failed or terminating according to node lifecycle behavior
- The Deployment/ReplicaSet tries to maintain the desired replica count
- Replacement pods can be scheduled on surviving eligible nodes
- If the pod uses a `ReadWriteOnce` PVC, the storage system must detach the volume from the dead node and attach it to the replacement node

For a standalone pod:

- There is no higher-level controller to recreate it
- If the pod is deleted or lost, it is simply gone
- This is why applications should normally be deployed through Deployments, StatefulSets, Jobs, or other controllers, not as bare pods

For a DaemonSet:

- One pod normally runs on every eligible node
- If a node dies, its DaemonSet pod disappears with that node
- No extra DaemonSet pod is created on another node, because DaemonSet semantics are one-per-node

For a StatefulSet:

- Replacement behavior is more conservative
- Stateful identity and storage attachment must be handled carefully
- RBD `ReadWriteOnce` detach/attach delays are especially relevant

The following commands are an operator checklist in case of a worker node failure.

Confirm Node State:

```
oc get nodes -o wide

oc describe node wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl get nodes -o wide

kubectl describe node wk-1.okd.ia2.inaf.it
```

Look for:

```
Ready=False
Ready=Unknown
NotReady
NetworkUnavailable
DiskPressure
MemoryPressure
PIDPressure
```

List pods that were running on the failed node:

```
oc get pods -A -o wide
    --field-selector spec.nodeName=wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl get pods -A -o wide
    --field-selector spec.nodeName=wk-1.okd.ia2.inaf.it
```

Check whether the application has been recreated elsewhere:

```
oc get pods -n test -l app=test-web -o wide

oc get deployment test-web -n test

oc describe deployment test-web -n test
```

Kubernetes equivalent:

```
kubectl get pods -n test -l app=test-web -o wide

kubectl get deployment test-web -n test

kubectl describe deployment test-web -n test
```

Check storage attachment if the pod uses a PVC:

```
oc get pvc -n test

oc get pv

oc get volumeattachments
```

Kubernetes equivalent:

```
kubectl get pvc -n test

kubectl get pv

kubectl get volumeattachments
```

If a replacement pod remains stuck in *`ContainerCreating`*, describe it:

```
POD=$(oc get pod -n test -l app=test-web
    -o jsonpath='{.items[0].metadata.name}')

oc describe pod "$POD" -n test
```

Kubernetes equivalent:

```
POD=$(kubectl get pod -n test -l app=test-web
    -o jsonpath='{.items[0].metadata.name}')

kubectl describe pod "$POD" -n test
```

Look for attach/mount errors such as:

```
Multi-Attach error
Unable to attach or mount volumes
timed out waiting for the condition
```

If the node is still reachable, drain it cleanly:

```
oc adm cordon wk-1.okd.ia2.inaf.it

oc adm drain wk-1.okd.ia2.inaf.it
    --ignore-daemonsets
      --delete-emptydir-data
```

Kubernetes equivalent:

```
kubectl cordon wk-1.okd.ia2.inaf.it

kubectl drain wk-1.okd.ia2.inaf.it
    --ignore-daemonsets
    --delete-emptydir-data
```

After repair:

```
oc adm uncordon wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl uncordon wk-1.okd.ia2.inaf.it
```

If the node is permanently dead, only after confirming that the VM is powered off and will not return unexpectedly:

```
oc adm cordon wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl cordon wk-1.okd.ia2.inaf.it
```

If pods remain stuck on the dead node and block replacement, force deletion may be required:

```
oc delete pod -n test <pod-name> --grace-period=0 --force
```

Kubernetes equivalent:

```
kubectl delete pod -n test <pod-name> --grace-period=0 --force
```

Important notes:

- Force deletion removes the API object immediately
- It does not guarantee that the process stopped on the old node
  use it only when the old node is confirmed dead or isolated
- This matters especially for stateful workloads and RBD volumes

If the node will be replaced rather than recovered, remove the Kubernetes node object:

```
oc delete node wk-1.okd.ia2.inaf.it
```

Kubernetes equivalent:

```
kubectl delete node wk-1.okd.ia2.inaf.it
```

Then provision a replacement worker VM using the documented worker ignition workflow.

If a replacement worker is added, after booting a new worker, check CSRs:

```
oc get csr
```

Approve pending node CSRs if required:

```
oc adm certificate approve <csr-name>
```

There is no exact *kubectl* equivalent for *oc adm certificate approve*; Kubernetes can approve CSRs by patching the CSR approval subresource, but OpenShift provides the operational helper.

To validate:

```
oc get nodes -o wide

oc get mcp

oc get pods -A | grep -v Running
```

Kubernetes equivalent:

```
kubectl get nodes -o wide

kubectl get pods -A | grep -v Running
```

In case of control plane node failure, a three-node control plane tolerates the loss of one control-plane node while preserving etcd quorum.

Immediate checks:

```
oc get nodes -l node-role.kubernetes.io/master -o wide

oc get co

oc get pods -n openshift-etcd -o wide

oc get pods -n openshift-kube-apiserver -o wide
```

Kubernetes equivalent for standard objects:

```
kubectl get nodes -l node-role.kubernetes.io/master -o wide

kubectl get pods -n openshift-etcd -o wide

kubectl get pods -n openshift-kube-apiserver -o wide
```

OpenShift-specific health:

```
oc get clusterversion

oc get co etcd kube-apiserver kube-controller-manager kube-scheduler
```

If one control-plane VM is temporarily down:

- Recover the Proxmox VM if possible
- Confirm the node returns *`Ready`*
- Verify *`oc get co`*
- Do not remove etcd members unless a proper control-plane recovery procedure is being followed

If two control-plane nodes are lost at the same time, etcd quorum is lost and the cluster is in a serious disaster-recovery scenario. At that point, do not improvise destructive actions; use the official etcd/control-plane recovery documentation and existing backups.

For Proxmox-level checks during node failure, on the Proxmox side, verify VM status:

```
qm list

qm status <vmid>
```

Check VM console or recent logs:

```
qm terminal <vmid>

journalctl -b
```

If the VM is running but the node is *NotReady*, check from the node console:

```
systemctl status kubelet

journalctl -u kubelet -b --no-pager

crictl ps

ip -br a

ip route
```

From the OKD side:

```
oc debug node/wk-1.okd.ia2.inaf.it
    -- chroot /host systemctl status kubelet
```

```
oc debug node/wk-1.okd.ia2.inaf.it
    -- chroot /host journalctl -u kubelet -b --no-pager | tail -100
```

Kubernetes equivalent:

```
kubectl debug node/wk-1.okd.ia2.inaf.it -it
    --image=registry.access.redhat.com/ubi9/ubi
    -- chroot /host systemctl status kubelet
```

44 Cluster Update

OKD cluster updates are managed by the Cluster Version Operator (CVO). The CVO tracks the desired cluster version through the ClusterVersion object, evaluates available update targets, applies the selected release payload, and coordinates the reconciliation of cluster operators during the update process.

The update is not a single package upgrade on individual nodes. It is a cluster-level operation involving the OpenShift release payload, cluster operators, machine configuration, node updates, and workload reconciliation. For this reason, update readiness must be checked before proceeding: ClusterOperators should not be degraded, nodes should be Ready, MachineConfigPools should be updated, and key services such as ingress and storage should be functional.

Updates can be initiated either from the command line with *`oc adm upgrade`* or from the OpenShift Web Console in the Administrator perspective, under:

Administration → Cluster Settings → Details

Cluster update management is OpenShift-specific. There is no direct Kubernetes equivalent to *`oc adm upgrade`*, because standard Kubernetes does not provide a built-in Cluster Version Operator or an integrated cluster-wide update workflow.

1. Pre-Update health checks

```
oc get clusterversion

oc get co

oc get nodes -o wide

oc get mcp

oc get pods -A | grep -v Running

oc get pvc -A
```

Check whether the cluster is upgradeable:

```
oc get clusterversion version
    -o jsonpath='{range .status.conditions[*]}
                {.type}={.status} {" "}{.message}{"\n"}{end}'
```

Look specifically for:

```
Upgradeable=True
Available=True
Progressing=False
Degraded=False
```

Check current channel and available updates:

```
oc adm upgrade
```

If supported by the installed OKD version, run the recommendation/precheck command:

```
oc adm upgrade recommend
```

2. Back up critical state before updating

At minimum:

```
mkdir -p /opt/okd/backups/pre-update-$(date +%F)

oc get clusterversion version -o yaml >
/opt/okd/backups/pre-update-$(date +%F)/clusterversion.yaml

oc get co -o yaml > /opt/okd/backups/pre-update-$(date
+%F)/clusteroperators.yaml

oc get nodes -o wide > /opt/okd/backups/pre-update-$(date
+%F)/nodes.txt

oc get mcp -o yaml > /opt/okd/backups/pre-update-$(date +%F)/mcp.yaml

oc get storageclass -o yaml > /opt/okd/backups/pre-update-$(date
+%F)/storageclasses.yaml

oc get pv -o yaml > /opt/okd/backups/pre-update-$(date +%F)/pvs.yaml
```

Also refresh the backups already documented in this report:

`/opt/okd/backups/ceph-csi/`

`/opt/okd/backups/auth/`

`/opt/okd/backups/projects/`

`/opt/okd/backups/test/`

For production, perform an etcd backup before updating.

Typical OpenShift etcd backup entry point:

```
oc debug node/<control-plane-node>
    -- chroot /host /usr/local/bin/cluster-backup.sh /home/core/backup
```

Then copy the generated backup files from the control-plane node to protected external storage.

3. Start the update

Update to the latest recommended version:

```
oc adm upgrade --to-latest=true
```

Or update to a specific version listed by `oc adm upgrade`:

```
oc adm upgrade --to=<version>
```

Example:

```
oc adm upgrade --to=4.21.0-okd-scos.ec.10
```

Use only versions appropriate for the cluster stream and architecture.

4. Monitor the update

```
watch -n 30 'oc get clusterversion; echo; oc get co; echo; oc get mcp;
echo; oc get nodes'
```

Non-watch form:

```
oc get clusterversion

oc describe clusterversion version

oc get co

oc get mcp

oc get nodes -o wide
```

Check update history:

```
oc describe clusterversion version
```

Watch MachineConfigPool progress:

```
oc get mcp -w
```

Watch nodes rebooting one at a time:

```
oc get nodes -w
```

5. Post-update validation

```
oc get clusterversion

oc get co

oc get mcp

oc get nodes -o wide

oc get pods -A | grep -v Running
```

Validate storage:

```
oc get pods -n ceph-csi-rbd -o wide

oc get pvc -A
```

Validate the demo application:

```
oc get deploy,pods,svc,route,pvc -n test

ROUTE_HOST=$(oc get route test-web -n test -o jsonpath='{.spec.host}')

curl -k https://${ROUTE_HOST}/

oc exec -n test deploy/test-web -- cat /data/messages.txt
```

Show all non-running pods:

```
oc get pods -A --field-selector=status.phase!=Running
```

Kubernetes equivalent:

```
kubectl get pods -A --field-selector=status.phase!=Running
```

Show pods sorted by node:

```
oc get pods -A -o wide --sort-by=.spec.nodeName
```

Kubernetes equivalent:

```
kubectl get pods -A -o wide --sort-by=.spec.nodeName
```

Show pods sorted by creation time:

```
oc get pods -A --sort-by=.metadata.creationTimestamp
```

Kubernetes equivalent:

```
kubectl get pods -A --sort-by=.metadata.creationTimestamp
```

Show images used in a namespace:

```
oc get pods -n test
    -o jsonpath='{range .items[*]}{.metadata.name}{" -> "}
                 {range .spec.containers[*]}{.image}
                 {" "}{end}{"\n"}{end}'
```

Kubernetes equivalent:

```
kubectl get pods
    -n test -o jsonpath='{range .items[*]}
                         {.metadata.name}{" -> "}
                         {range .spec.containers[*]}
                         {.image}{" "}{end}{"\n"}{end}'
```

Show resource requests and limits:

```
oc describe node wk-1.okd.ia2.inaf.it
| sed -n '/Allocated resources:/,/Events:/p'
```

Kubernetes equivalent:

```
kubectl describe node wk-1.okd.ia2.inaf.it
| sed -n '/Allocated resources:/,/Events:/p'
```

Show current user and namespace:

```
oc whoami

oc project
```

Kubernetes equivalent:

```
kubectl config view --minify
    -o jsonpath='user={.contexts[0].context.user}
       namespace={.contexts[0].context.namespace}{"\n"}'
```

Show all routes:

```
oc get route -A
```

Kubernetes equivalent:

```
kubectl get ingress -A
```

Show all PVCs and their StorageClass:

```
oc get pvc -A -o
custom-columns='NAMESPACE:.metadata.namespace,NAME:.metadata.name,STATU
S:.status.phase,VOLUME:.spec.volumeName,SC:.spec.storageClassName,SIZE:
.spec.resources.requests.storage'
```

Kubernetes equivalent:

```
kubectl get pvc -A -o

custom-columns='NAMESPACE:.metadata.namespace,NAME:.metadata.name,STATU

S:.status.phase,VOLUME:.spec.volumeName,SC:.spec.storageClassName,SIZE:

.spec.resources.requests.storage'
```

Practical rules of thumb:

- Do not deploy important applications as bare pods. Use Deployments, StatefulSets, Jobs, or other controllers
- A pod is disposable. The application state must live in a database, object store, or persistent volume
- A Deployment recreates pods. It does not guarantee that a replacement pod lands on a specific node unless scheduling rules are set
- A node can be cordoned before maintenance. Cordon prevents new pods from landing there
- A node can be drained before maintenance. Drain evicts eligible pods so they can be recreated elsewhere
- RBD is good for single-writer persistent volumes. Do not assume it behaves like shared multi-writer storage
- If a node dies, do not immediately force-delete stateful pods unless the old node is confirmed off or isolated. This avoids split-brain and storage corruption risks
- Before an OKD update, fix degraded operators first. Updating a degraded cluster increases risk
- Before an OKD update, take an etcd backup and refresh manifest backups. Cluster updates are designed to be online, but recovery requires backups
- After an OKD update, validate the cluster, storage, ingress, and a real application. A green `ClusterVersion` alone is not a complete application-level validation

45 Next Steps

The deployment and operational procedures documented in this report provide a validated foundation for running OKD on Proxmox VE with external Ceph RBD storage. Future iterations of the platform documentation and implementation could extend this work in the following areas:

1. Architectural Diagrams

The current report is primarily command-oriented and procedural. A useful next step would be to add architectural diagrams to make the infrastructure easier to understand at a glance.

Recommended diagrams include:

- Network topology, showing the bastion host, HAProxy, DNS records, OKD nodes, application ingress, and external Ceph connectivity
- Ceph-CSI integration, showing the relationship between worker nodes, the dedicated Ceph network, Ceph monitors, Ceph-CSI controller components, Ceph-CSI node plugins, StorageClass, PVC, PV, and application pods

2. Automation

Several procedures in this report are intentionally documented as explicit manual commands for clarity and auditability. Future work could convert selected procedures into automation, for example through Ansible playbooks or shell scripts.

Candidate automation areas include:

- Initial HAProxy configuration generation from a node inventory
- DNS record generation or validation, depending on the DNS management system available in the target environment
- Bastion preparation, including required packages, directory layout, client installation, and backup folders
- Ceph-CSI deployment, namespace creation, ConfigMaps, Secrets, SCC bindings, manifest patching, and StorageClass creation
- Tenant onboarding, including project creation, OpenShift group creation, RBAC binding, and backup refresh
- Application template deployment and validation

Automation should be introduced carefully. The most critical installation phases, especially ignition generation, bootstrap sequencing, HAProxy phase transitions, and certificate validity windows, should remain explicit and well validated before being fully automated.

3. Monitoring, Logging, and Alerting

The current document includes operational checks based on `oc`, `kubectl`, logs, events, ClusterOperators, MachineConfigPools, PVCs, and workload status. A future extension should describe a more complete observability model.

Topics to document include:

- Cluster monitoring through the OpenShift monitoring stack
- Prometheus-based metrics for nodes, pods, storage, ingress, and cluster operators
- Grafana dashboards, if an external or additional Grafana instance is used
- Centralized logging for application and infrastructure logs, for example through Loki, EFK, or the OpenShift logging stack
- Alerting rules for important operational conditions, such as nodes becoming `NotReady`, degraded ClusterOperators, pending PVCs, Ceph-CSI failures, certificate expiration, storage attachment errors, and API or ingress unavailability

4. Scalability and High Availability

The validated deployment uses three control-plane nodes and multiple worker nodes, but the HAProxy layer currently remains a single point of failure. Future work should therefore address high availability and scaling in more detail.

Recommended topics include:

- Redundant HAProxy design using Keepalived/VRRP or another virtual IP mechanism
- Failure behavior of the load-balancer layer and operational tests for HAProxy failover
- Additional details on etcd quorum, control-plane failure tolerance, and safe recovery boundaries
- Worker node scaling procedures, including adding and removing worker nodes
- MachineConfigPool behavior during node expansion
- Possible use of cluster autoscaling or machine management approaches, if suitable for the Proxmox UPI environment
- Capacity planning for CPU, memory, storage, ingress traffic, and Ceph-backed persistent workloads

5. Advanced Security

The current report covers HTPasswd authentication, OpenShift groups, RBAC, SCC requirements, bastion-based access, and certificate trust considerations. Future work should expand the security model.

Recommended topics include:

- NetworkPolicy examples to restrict traffic between projects, pods, and application tiers
- More detailed comparison between OpenShift SCC and Kubernetes Pod Security Admission
- Least-privilege service account design for applications
- Custom TLS certificates for OpenShift Routes
- Integration with cert-manager or Let's Encrypt where appropriate
- Replacement or integration of HTPasswd authentication with a central identity provider such as Keycloak, LDAP, or OIDC
- Hardening of bastion access, SSH policies, audit logging, and user lifecycle management
- Secret management practices for CephX keys, pull secrets, application secrets, and backup material

6. CI/CD

The application deployment example in this report is intentionally simple and manifest-based. A future improvement would be to add a CI/CD workflow that builds, tests, and deploys applications automatically.

Possible topics include:

- Tekton Pipelines on OpenShift
- Source-to-image or container image build workflows
- Git-based manifest deployment
- Promotion from test to production projects
- Automated application rollout and rollback
- Image update strategies and image scanning
- Separation between application source repositories, deployment manifests, and cluster configuration repositories

7. Cluster Updates and Lifecycle Management

This report includes a safe update procedure based on the Cluster Version Operator, pre-update checks, backups, update execution, monitoring, and post-update validation. Future work could expand this into a full lifecycle management chapter.

Recommended additions include:

- A detailed example of an OKD minor update, for example from OKD 4.21 to a later supported release
- Channel selection and available update path validation
- Pre-update etcd backup procedure and restore testing
- Update risk checklist for degraded operators, MachineConfigPools, Ceph-CSI, ingress, authentication, and application workloads
- Post-update validation matrix covering cluster health, storage, routes, authentication, and tenant workloads
- Rollback and recovery considerations, clearly distinguishing supported recovery procedures from unsafe downgrade assumptions

8. Overall Direction

The next phase of the documentation should move from a validated manual installation and operations handbook toward a more complete production-readiness guide. The most important priorities are high availability for HAProxy, stronger observability, automated repeatability, improved identity integration, security hardening, and tested backup and recovery procedures.

46 Final Conclusions

This work transformed a manually deployed OKD 4.21 UPI cluster on Proxmox VE into a documented and operationally usable platform.

The initial installation validated the core infrastructure required by OKD in a UPI environment: DNS resolution, HAProxy-based load balancing, bootstrap sequencing, Machine Config Server reachability, control-plane formation, worker node provisioning, and post-bootstrap reconfiguration. Particular attention was given to the aspects that are most likely to cause installation failures, such as port 22623 routing, ignition file validity, certificate expiration windows, Proxmox CPU settings, and correct network preservation during CoreOS installation.

After the base cluster was installed, the environment was extended beyond a simple proof of installation. Dedicated worker networking was configured for external Ceph access, Ceph RBD storage was integrated through Ceph-CSI, and a working StorageClass was validated through dynamic PVC provisioning, pod attachment, file writes, and persistence checks. This confirmed that the cluster can support stateful workloads backed by external storage.

The platform was then prepared for multi-user operation. Local authentication was configured through the OpenShift HTPasswd identity provider, tenant projects were created, users were mapped to OpenShift groups, and project-level RBAC was validated. This provided a practical access model in which users can work independently inside assigned namespaces without requiring cluster-admin privileges.

The bastion-based workflow further simplifies user access. Users can connect to a controlled administrative host, authenticate to OKD with `oc login`, select their project, and deploy applications without installing local tooling on their workstations. This model also keeps operational examples, templates, and backup material in a central location.

Application-level validation was performed with a small Python web application using Ceph-backed persistent storage and HTTPS exposure through OpenShift Routes. The application was rebuilt and operated as a non-admin user, confirming that the tenant workflow is functional and that common application operations can be performed with normal project-level permissions.

The report also documents daily operational procedures for both administrators and users. These include checking cluster and application health, inspecting pods, services, routes, PVCs, logs and events, restarting and scaling workloads, understanding scheduling behavior, handling worker node failures, validating storage attachment, and preparing for cluster updates. Where possible, OpenShift commands are shown alongside Kubernetes equivalents to distinguish standard Kubernetes operations from OpenShift-specific features.

The resulting environment is therefore not only an installed OKD cluster, but a practical multi-tenant platform with documented installation, storage, authentication, application deployment, troubleshooting, backup, and operational procedures.

Some limitations remain. The current HAProxy deployment uses a single load-balancer node, which represents a single point of failure for API access and application ingress. Authentication is based on a local HTPasswd provider, which is suitable for a controlled environment but should eventually be replaced or integrated with a central identity provider such as Keycloak, LDAP, or OIDC. The demo application uses Ceph RBD with ReadWriteOnce volumes, which is appropriate for single-writer workloads but not for shared multi-writer application patterns. For production environments, additional hardening should also include proper certificate trust distribution, stronger backup automation, documented etcd backup/restore testing, and a redundant load-balancer design.

Overall, the deployment demonstrates that OKD on Proxmox VE can be operated as a usable and extensible platform when the surrounding infrastructure is carefully prepared and documented. The most important operational lessons are that UPI installations depend heavily on correct external services, that storage and authentication should be validated with real workloads, and that day-to-day procedures are as important as the initial cluster installation itself.

47 References

[1] *Origin Kubernetes Distribution (OKD)* is the community distribution of Kubernetes optimized for continuous application development and multi-tenant deployment:
https://okd.io/

[2] *Red Hat OpenShift* is an enterprise-grade, hybrid cloud Platform-as-a-Service (PaaS) built on Kubernetes.
https://www.redhat.com/it/technologies/cloud-computing/openshift

[3] *Kubernetes*, also known as K8s, is an open source system for automating deployment, scaling and management of containerized applications:
https://kubernetes.io

[4] *Proxmox Virtual Environment (VE)* is an open-source, enterprise-grade platform for server virtualization:
https://www.proxmox.com/en/

[5] *Ceph* is an open-source, software-defined, distributed storage system designed for high performance, reliability, and scalability to the exabyte level:
https://ceph.io/en/