
# IoT Project

## Project Report and Principles

Metropolia University of Applied Sciences

19 March 2025

# Abstract

Authors: Elham Rastighahfarokhi, Mehdi Nourivahid, Mostafa Sharghi

Title: Pod Deployment and Management in MicroShift

Number of Pages: 102 pages

Date: 19 March 2025

Degree: Bachelor of Engineering

Degree Programme: Information Technology

Specialisation option: Smart IoT Systems and Networking

Instructors: Markku Niiranen (Project Manager), Marko Uusitalo (Principal Lecturer), Tapio Wikström ((Principal Lecturer)

This project explores the deployment and management of pods in a MicroShift environment, a lightweight Kubernetes distribution designed for edge computing. The objective was to analyze the process of deploying containerized applications using YAML manifests and assess the operational aspects, including scheduling, networking, monitoring, and troubleshooting in MicroShift. The study was conducted within the context of container orchestration for enterprise and edge computing.

The study involved deploying a Python-based web server using the Red Hat Python 3.9 container image. YAML manifest files were used to define pod configurations, and the deployment process was carried out using OpenShift commands. The implementation covered key aspects such as container image retrieval, pod scheduling, networking setup, readiness and liveness probes, and status monitoring using relevant OpenShift CLI commands.

The findings of the study highlight the structured deployment process of pods in MicroShift and the critical role of YAML manifests in defining and managing containerized applications. The study also provides insights into monitoring techniques for maintaining pod health and reliability.

Additionally, it outlines common troubleshooting steps for addressing pod failures, ensuring seamless operation within a MicroShift cluster.

The outcome of this study contributes to a deeper understanding of containerized application management in lightweight Kubernetes environments. These insights can help organizations streamline their MicroShift deployments, improve reliability, and enhance troubleshooting strategies. Future work could explore automation techniques for deployment and scaling within edge computing environments.

**Keywords:** MicroShift, Kubernetes, Pod Deployment, Container Orchestration, OpenShift

# Table of Content

# 1. Introduction

This documentation presents a comprehensive exploration of the deployment, management, and optimization of MicroShift on Red Hat Enterprise Linux (RHEL) 9.5, with the goal of providing an efficient and scalable edge computing solution through containerized environments. Edge computing is becoming increasingly important as businesses and organizations seek to manage data and applications closer to the source, reducing latency and improving performance. The project focuses on addressing the challenges of managing Kubernetes-based containerized applications in resource-constrained edge environments, which are typically characterized by limited processing power, storage, and network capabilities.

The primary objective of this report is to offer a detailed guide on how to effectively deploy, configure, and manage MicroShift on RHEL 9.5. This includes an analysis of hardware selection, the installation of the operating system, and the deployment of containerized applications through MicroShift. The document also explores the impact of upgrading and downgrading RHEL and MicroShift, particularly concerning time, CPU, and memory usage, and the behavior of pods during these processes. The overall goal is to provide administrators with practical insights and best practices for managing containerized applications in edge environments, ensuring minimal service disruption during system upgrades and reboots.

This project aims to answer key research questions related to the deployment of MicroShift in edge computing environments. Specifically, it seeks to understand how MicroShift can be deployed and managed on RHEL 9.5 for optimal edge computing performance and what best practices can be implemented to ensure pod stability and continuity during system upgrades and reboots. The findings of this report are expected to be valuable for organizations that rely on edge devices and containerized applications, enabling them to improve the efficiency and reliability of their operations while minimizing downtime.

The value provided by this documentation lies in its ability to offer actionable recommendations and technical guidance for integrating MicroShift with RHEL, a combination that can significantly enhance the performance of edge computing systems. By providing a clear framework for the setup and management of this solution, this work contributes to the successful deployment of containerized applications on edge devices. This can lead to cost savings, improved operational efficiency, and enhanced system reliability for organizations in diverse industries, ranging from IoT to telecommunications.

The structure of this report is organized as follows:

The Theory section introduces fundamental concepts such as containers, Kubernetes, OpenShift, and MicroShift, providing the necessary background for understanding the deployment process. The Methods and Material section discusses the hardware selection, and the installation challenges faced during the transition from Raspberry Pi 4 to Dell PowerEdge R630, as well as the configuration of RHEL 9.5. The Proposed Solution/Implementation section offers a detailed walkthrough of the installation and management process for MicroShift, covering key tasks such as pod creation, management, and troubleshooting. In the Project Specifications (Upgrading) section, the report examines the effects of upgrading and downgrading RHEL and MicroShift, with a focus on maintaining pod stability during these processes. Finally, the Summary and Conclusions section highlights the key findings and provides recommendations for further optimization.

This report is intended to serve as a practical guide for organizations looking to implement MicroShift in edge environments, offering insights into how to ensure efficient and reliable operation of containerized applications in such resource-constrained settings.

# 2. Theory

## 2.1. Container

A container is a lightweight, portable, and standalone executable package that encapsulates an application and its dependencies, including libraries, runtime, and configurations, ensuring consistent execution across different environments (Red Hat, Podman Documentation). Containers provide process isolation using Linux namespaces and cgroups, enabling them to run independently of the underlying host system (The Linux Foundation, Kubernetes Documentation). Unlike virtual machines (VMs), containers share the host OS kernel, making them more efficient in terms of resource utilization.

## 2.2. Image

A container image is an immutable, pre-built package containing the application code, dependencies, configurations, and runtime required to create a running container (Red Hat, Podman Documentation). Images use a union file system, allowing efficient storage and reuse. They are stored in container registries such as Docker Hub, Quay.io, or Red Hat Container Registry and are pulled when deploying a container (Red Hat, OpenShift Documentation).

## 2.3. Container Runtime

A container runtime is the software responsible for executing and managing containers. It handles low-level container operations such as pulling images, creating and running containers, and managing networking and security (Red Hat, CRI-O Documentation). Popular container runtimes include Docker, which is a full containerization platform providing both runtime and management tools (Red Hat, Podman Documentation); containerd, a lightweight container runtime that manages the lifecycle of containers (The Linux Foundation, Kubernetes Documentation); and CRI-O, a Kubernetes-native runtime optimized for running OCI (Open Container Initiative) images efficiently in OpenShift and Kubernetes clusters (Red Hat, CRI-O Documentation).

## 2.4. Pod

A pod is the smallest deployable unit in Kubernetes, consisting of one or more containers that share network resources and storage (The Linux Foundation, Kubernetes Documentation). Containers inside a pod communicate using a shared network namespace, allowing seamless interaction between them. Kubernetes schedules and manages pods instead of individual containers to provide better orchestration and scalability (Red Hat, OpenShift Documentation).

## 2.5. Node

A node is a worker machine in a Kubernetes cluster that runs pods. Nodes can be either physical servers or virtual machines and are managed by the Kubernetes control plane (The Linux Foundation, Kubernetes Documentation). Each node runs key components, including the Kubelet, which manages the pod lifecycle on the node (The Linux Foundation, Kubernetes Documentation). The container runtime runs containers inside pods (Red Hat, CRI-O Documentation), while the Kube-proxy handles networking and service discovery within the node (The Linux Foundation, Kubernetes Documentation).

## 2.6. Kubernetes

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications (The Linux Foundation, Kubernetes Documentation). It abstracts infrastructure complexities and provides features such as automatic scaling, which adjusts the number of running containers based on demand (The Linux Foundation, Kubernetes Documentation). Kubernetes also includes self-healing, a feature that restarts failed containers and reschedules pods on healthy nodes to ensure application availability (Red Hat,

OpenShift Documentation). Additionally, it supports service discovery and load balancing, ensuring seamless communication between microservices (The Linux Foundation, Kubernetes Documentation).

## 2.7. OpenShift

OpenShift is an enterprise-grade Kubernetes platform developed by Red Hat, extending Kubernetes with additional security features, developer tools, and built-in container management capabilities (Red Hat, OpenShift Documentation). Key features include integrated CI/CD pipelines, which automate application builds and deployments (Red Hat, OpenShift Documentation). OpenShift also offers enhanced security, enforcing stricter policies and role-based access control (RBAC) to ensure secure operations (Red Hat, OpenShift Documentation). Additionally, it provides Source-to-Image (S2I) builds, enabling developers to create container images from source code automatically (Red Hat, OpenShift Documentation).

## 2.8. MicroShift

MicroShift is a lightweight Kubernetes distribution optimized for edge computing and IoT (Internet of Things) devices, developed as a minimal version of OpenShift (Red Hat, MicroShift Documentation). It offers a low resource footprint, making it suitable for constrained environments (Red Hat, MicroShift Documentation). MicroShift also supports offline and disconnected operations, allowing deployments in environments with limited network connectivity (Red Hat, MicroShift Documentation). Furthermore, it enables seamless integration with OpenShift, allowing containerized workloads to run at the edge while benefiting from OpenShift's enterprise features (Red Hat, OpenShift Documentation).

## 2.9. Cluster

A cluster is a group of interconnected nodes that work together as a single system in Kubernetes (The Linux Foundation, Kubernetes Documentation). A Kubernetes cluster consists of the control plane, which manages the cluster, schedules workloads, and maintains the desired application state (The Linux Foundation, Kubernetes Documentation). The worker nodes execute containerized applications inside pods (The Linux Foundation, Kubernetes Documentation).

## 2.10. API (Application Programming Interface)

An API is a set of rules and protocols that allow different software components to communicate (Red Hat, OpenShift CLI Documentation). In Kubernetes, the Kubernetes API Server is the primary interface that enables users, controllers, and automation scripts to manage cluster resources through RESTful API calls (The Linux Foundation, Kubernetes Documentation).

## 2.11. Operator

An Operator is a Kubernetes extension that automates the lifecycle management of complex applications by encapsulating operational knowledge into Kubernetes-native controllers (Red Hat, OpenShift Documentation). Operators simplify tasks such as automated application deployment and scaling, ensuring that applications are deployed and scaled according to the desired state without manual intervention (Red Hat, OpenShift Documentation). They also enable self-healing and configuration management, allowing applications to automatically recover from failures and maintain the desired configuration (Red Hat, OpenShift Documentation). Additionally, Operators facilitate the use of custom resource definition (CRD) extensions, enabling the management of application-specific resources in a declarative manner (Red Hat, OpenShift Documentation).

# 3. Methods and Material

## 3.1. Hardware Selection and Installation Challenges

### 3.1.1. Raspberry Pi 4 as the Initial Edge Device

At the beginning of this project, the Raspberry Pi 4 was selected as the edge device due to its compact size, low power consumption, and affordability. The goal was to deploy Red Hat Enterprise Linux (RHEL) 9.5 to evaluate its feasibility on an ARM-based platform.

### 3.1.2. Challenges in RHEL Installation

Upon consulting Red Hat Support, it was clarified that Raspberry Pi is not an officially supported hardware platform for RHEL. Despite this limitation, multiple installation attempts were made, which involved identifying compatible bootloader configurations to ensure system initialization,

troubleshooting kernel compatibility issues to enable hardware support, and adjusting firmware and system settings to optimize performance and stability. Through extensive debugging and manual configurations, RHEL 9.5 was successfully installed on the Raspberry Pi.

#### 3.1.3. Documentation and Reference

The entire installation process, including troubleshooting steps, modifications, and workarounds, was meticulously documented and is available for reference at:

https://github.com/itnetworking2022

#### 3.1.4. Transition to Dell Server

Despite the successful installation, hardware limitations such as processing power, memory constraints, and storage capacity posed challenges for long-term usability. To ensure optimal performance and compatibility with enterprise workloads, a transition to a Dell server was later recommended, leading to a shift in the hardware platform.

### 3.2. Server Hardware Used

#### 3.2.1. Dell PowerEdge R630 Selection

The Dell PowerEdge R630 was selected as the primary hardware for this project due to its enterprise-grade performance, reliability, and scalability. This server was essential for handling the complex demands of edge computing and supporting the deployment of resource-intensive workloads such as containerization, virtual environments, and real-time data processing. The PowerEdge R630 offered the necessary computing power, memory, and network capabilities, surpassing the limitations of the initially intended Raspberry Pi 4, and providing a robust solution for the project's requirements.

#### 3.2.2. Key Features of Dell PowerEdge R630

The Dell PowerEdge R630 server is equipped with dual Intel Xeon E5-2600 v4 series processors, delivering significant computational power. These processors enable the efficient execution of parallel tasks and support demanding enterprise-level applications, including data analytics, cloud computing, and virtualization.

With 1TB of DDR4 RAM, the system is capable of processing large datasets and handling memory-intensive applications. This substantial memory capacity enhances efficiency in tasks such as data analysis, machine learning, and virtual machine hosting. Additionally, the ample memory contributes to improved multitasking and system responsiveness, particularly in high-demand environments.

The storage configuration of the R630 offers a combination of SSD and HDD drives, providing both high-speed data access and ample storage capacity. The inclusion of SSDs ensures fast boot times and quick data retrieval, while HDDs serve as a cost-effective solution for large-scale storage needs. Furthermore, the server supports RAID configurations, enhancing data safety and reliability through redundancy.

Networking capabilities are integrated with both 10GbE and 1GbE network adapters, ensuring fast and reliable communication. These features are essential for applications requiring high bandwidth and low latency, such as real-time data transfer, distributed computing, and cloud-based services.

Designed with scalability in mind, the Dell PowerEdge R630 allows for seamless expansion by accommodating additional storage, memory, and network interfaces. This flexibility ensures that the server can adapt to the growing demands of the project, supporting future enhancements and increased computational requirements without necessitating a complete system overhaul.

### 3.2.3. Transition from Raspberry Pi 4 to Dell PowerEdge R630

The transition from the Raspberry Pi 4 to the Dell PowerEdge R630 was prompted by the hardware limitations encountered during the project's initial stages. While the Raspberry Pi 4 was sufficient for basic IoT tasks, it lacked the necessary processing power, memory capacity, and network performance to handle more demanding workloads and enterprise-level applications. The PowerEdge R630's superior performance, scalability, and networking capabilities provided the required platform to ensure the project's long-term success, supporting real-time data processing, container orchestration, and large-scale data storage.

## 3.3. Operating System: Linux RHEL 9.5

For this project, Red Hat Enterprise Linux (RHEL) 9.5 was selected as the operating system due to its robust security features, optimized support for containers and Kubernetes, and its seamless integration with the Red Hat ecosystem. The choice of RHEL 9.5 ensures a stable and secure

environment for deploying MicroShift, a lightweight Kubernetes distribution designed specifically for edge environments. Below are the key technical reasons for selecting RHEL 9.5:

### 3.3.1. Enterprise-Grade Stability and Security

Security and Patching: RHEL 9.5 provides a comprehensive security framework, incorporating SELinux (Security-Enhanced Linux), which enforces mandatory access controls and enhances security for containerized workloads. RHEL's integrated vulnerability scanning tools offer proactive protection, ensuring that security gaps are identified and patched quickly. This is vital for ensuring the stability and security of MicroShift in production environments.

Long-Term Support: RHEL 9.5 offers long-term support for up to 10 years, guaranteeing a stable environment for ongoing projects. This long support lifecycle ensures that the operating system remains secure, updated, and compatible with evolving technologies, minimizing the need for frequent upgrades and reducing maintenance overhead. This longevity makes it a reliable choice for edge and IoT deployments, where system stability is essential.

### 3.3.2. Container and Kubernetes Optimization

Optimized Kernel for Containers: RHEL 9.5 features an optimized kernel tailored for the efficient execution of containerized workloads. This optimization is critical for edge computing, where resources are often limited, and performance efficiency is essential. The kernel enhancements directly benefit the performance of MicroShift, enabling smooth and effective container orchestration.

CRI-O Container Runtime: RHEL 9.5 supports CRI-O, the container runtime recommended for Kubernetes and MicroShift. CRI-O is a lightweight, high-performance runtime that allows Kubernetes and MicroShift to run containers efficiently while consuming minimal system resources. This support ensures optimal integration and high-performance container orchestration, essential for edge applications.

### 3.3.3. Compatibility with OpenShift and Kubernetes

OpenShift Compatibility: As MicroShift is based on OpenShift, RHEL 9.5 ensures full compatibility with OpenShift tools, services, and features. This compatibility makes scaling from MicroShift to OpenShift straightforward when additional enterprise-level features and scalability

are needed. Additionally, this compatibility enables seamless migration between MicroShift and OpenShift depending on evolving needs, ensuring a flexible and adaptable platform.

Kubernetes Integration: RHEL 9.5 integrates smoothly with Kubernetes, offering out-of-the-box support for essential Kubernetes tools such as kubelet, kubeadm, and kubectl. This ensures that MicroShift functions as intended by leveraging Kubernetes' robust container orchestration capabilities, which include automated deployment, scaling, and management of applications across a distributed edge network.

### 3.3.4. Red Hat Ecosystem and Support

Certified Ecosystem: RHEL 9.5 is fully certified for integration with Red Hat's ecosystem of tools, including Podman, OpenShift, and Red Hat Universal Base Images (UBIs). This certification guarantees that MicroShift operates seamlessly within the Red Hat ecosystem, benefiting from Red Hat's comprehensive suite of tools designed for containerized environments. The ecosystem integration ensures that all components work together efficiently, from deployment to monitoring.

Official Support: RHEL 9.5 includes official support from Red Hat, which provides access to critical updates, security patches, and expert technical guidance. This support is crucial for maintaining the reliability, security, and overall health of the MicroShift environment, particularly in production settings where uptime and performance are paramount. Red Hat's support network ensures that any challenges encountered can be addressed promptly.

### 3.3.5. Performance and Resource Efficiency

Optimized for Edge Computing: RHEL 9.5 is designed to operate efficiently in edge environments, which often have limited resources such as CPU, memory, and storage. The operating system's optimization ensures that MicroShift can run effectively in these constrained environments, making it ideal for deployments in remote locations, smart cities, and IoT applications, where the computing power may be limited but performance cannot be compromised.

Minimal Footprint: The RHEL for Edge variant of RHEL 9.5 is specifically designed for smaller, lightweight deployments, allowing for minimal system overhead while maintaining critical functionality. This version of the operating system is perfect for scenarios where resources are constrained, such as remote field locations or small-scale edge computing environments. It enables the deployment of MicroShift efficiently, optimizing resource use while ensuring the necessary system capabilities are met.

### 3.3.6. Integration with Red Hat Tools

Red Hat Insights: RHEL 9.5 integrates seamlessly with Red Hat Insights, a predictive analytics tool that offers insights into system health, performance, and security. With this integration, proactive management of the MicroShift environment becomes possible, allowing administrators to detect and address issues before they escalate, ensuring smooth operations.

Automation Tools: RHEL 9.5 works in conjunction with Red Hat Ansible and Red Hat OpenShift Automation tools, enabling automation of deployment, configuration, and management tasks. These tools are essential for maintaining large-scale deployments of MicroShift at the edge, reducing manual intervention, ensuring consistency, and enhancing operational efficiency. Automation improves the overall lifecycle management of edge applications, particularly when scaling across numerous devices or locations.

### 3.3.7. Conclusion and Justification for RHEL 9.5 Selection

The selection of RHEL 9.5 for this project was based on its comprehensive security features, optimized support for containers and Kubernetes, and seamless integration with the Red Hat ecosystem. With its long-term support, performance optimizations for edge environments, and official Red Hat support, RHEL 9.5 offers a stable, secure, and efficient foundation for deploying and managing MicroShift. The system ensures that the project is equipped with a reliable operating system capable of handling the complex demands of edge computing while offering flexibility and scalability for future expansion.

## 3.4. MicroShift

MicroShift is a lightweight, Kubernetes-based platform designed specifically for edge and IoT (Internet of Things) environments. It is optimized for running containerized applications on resource-constrained devices, making it ideal for deployment on edge devices with limited resources. MicroShift is based on the Kubernetes project but is tailored to meet the specific needs of edge computing and IoT use cases, providing the scalability and flexibility of Kubernetes without the overhead typically associated with a full-scale deployment.

### 3.4.1. Key Features of MicroShift

MicroShift is a lightweight Kubernetes distribution designed to operate efficiently in edge environments with limited resources. By minimizing unnecessary components and optimizing core

functionalities, MicroShift ensures that Kubernetes capabilities can be utilized without a heavy footprint. This lightweight design enables faster startup times, more efficient resource allocation, and reduced operational overhead, making it well-suited for edge computing.

Optimized for edge and IoT environments, MicroShift facilitates seamless deployment and management of containerized applications on devices such as Raspberry Pi, edge servers, and other low-resource hardware platforms. Its efficient resource utilization ensures smooth operation even in distributed or remote deployments. Additionally, MicroShift supports the decentralized nature of edge computing, where devices are often located far from centralized cloud infrastructures. This approach enhances responsiveness, reduces latency, and improves resilience for edge applications.

MicroShift follows a container-oriented architecture, allowing applications to be packaged and deployed using containers. This approach simplifies application management, scalability, and security in edge environments. The ability to run multiple containers in isolated environments ensures that applications can be independently managed, upgraded, and scaled as needed. This flexibility is particularly beneficial for edge applications that require adaptability and agility.

To cater to resource-constrained edge environments, MicroShift simplifies core Kubernetes features while maintaining essential container orchestration functionalities such as automatic scaling and application management. By removing non-essential components, it reduces resource consumption, making it more efficient for devices with limited CPU, memory, and storage capacity. Unlike full Kubernetes distributions, MicroShift does not include an extensive suite of networking and storage components, further optimizing its footprint.

As part of the Red Hat ecosystem, MicroShift seamlessly integrates with Red Hat tools and services. This includes compatibility with OpenShift, enabling smooth migration between MicroShift and OpenShift environments when scalability or advanced features are required. Integration with Red Hat Ansible allows for automated deployment, management, and orchestration of MicroShift clusters, simplifying operations in large-scale edge deployments.

MicroShift is optimized for container runtimes, using CRI-O as its default runtime. CRI-O is specifically designed to work efficiently with Kubernetes and other container orchestration tools, reducing the overhead commonly associated with container runtimes in edge environments. This optimization ensures high performance, allowing containerized applications to function effectively even with limited resources.

With a minimal footprint tailored for edge deployments, MicroShift reduces operating system and resource overhead. This design is particularly advantageous for IoT and edge devices that have

restricted CPU, memory, and storage capacity. The reduced footprint contributes to faster boot times and lower maintenance needs, making MicroShift suitable for autonomous operations or deployments in environments with unreliable connectivity.

Built using Kubernetes components, MicroShift maintains full compatibility with Kubernetes and OpenShift environments. This ensures that workloads can be migrated seamlessly between edge devices, cloud platforms, and on-premises data centers. Organizations can deploy MicroShift on edge devices and later transition to OpenShift for more extensive and complex cloud or data center deployments as their requirements evolve.

### 3.4.2. Conclusion and Justification for MicroShift Selection

MicroShift is an optimized Kubernetes distribution tailored for edge and IoT environments. Its lightweight nature, combined with container orchestration, makes it an ideal choice for running applications in resource-constrained environments. Built on Kubernetes and fully integrated with the Red Hat ecosystem, MicroShift enables efficient management of containerized applications at the edge, ensuring scalability and flexibility for various edge computing and IoT use cases. The platform's minimal footprint, compatibility with Red Hat tools, and seamless integration with Kubernetes and OpenShift further justify its selection for this project, ensuring both operational efficiency and long-term sustainability.

## 3.5. Podman

Podman is an open-source container management tool that allows users to build, manage, and run containers and containerized applications. Unlike traditional container runtimes, Podman is daemonless, meaning it does not rely on a long-running background process or service to manage containers. This feature makes Podman lightweight, secure, and suitable for both individual and large-scale container management in various environments, including edge and IoT. Podman's architecture enables it to run in environments where resources are constrained, and security is a top priority.

### 3.5.1. Key Features of Podman

Podman features a daemonless architecture, meaning it does not rely on a centralized service to manage containers. Unlike Docker, which requires a background daemon, Podman runs containers as independent processes, improving security and reducing system resource consumption. This

design eliminates a single point of failure and makes Podman highly efficient for environments like IoT and edge computing, where lightweight and secure container management is essential.

Podman maintains full compatibility with Docker's command-line interface (CLI), allowing users to execute familiar commands without modifying workflows or rewriting scripts. Common Docker commands, such as podman run, podman build, and podman push, function similarly, ensuring a smooth transition from Docker to Podman without disrupting container management processes.

Security is enhanced through Podman's support for rootless containers, enabling containers to run without requiring root privileges. This reduces the risk of privilege escalation and enhances container security in restricted environments, such as shared systems or IoT devices. By allowing non-root users to manage containers, Podman provides a safer alternative for containerized applications that operate in security-sensitive environments.

Podman introduces the concept of "pods," similar to Kubernetes, which allows multiple containers to be grouped and managed together. This functionality is particularly beneficial for microservices architectures and multi-container applications that require tightly coupled components to function as a single unit. Pods simplify container lifecycle management, ensuring efficient deployment and synchronization of related containers.

Comprehensive container image management is another key feature of Podman. It allows users to build, pull, push, and inspect container images while integrating seamlessly with popular container registries like Docker Hub and Quay. This ensures efficient distribution and version control of container images, making it easier to manage containerized applications across various environments, including edge computing projects.

Podman facilitates integration with Kubernetes by allowing users to generate Kubernetes YAML files directly from running containers. This feature simplifies the transition to Kubernetes or MicroShift, enabling users to deploy locally built containers in cloud-native environments with minimal effort. It streamlines containerized application deployment across Kubernetes clusters, particularly in edge scenarios where scaling is required.

Security is a core focus of Podman, with built-in features such as SELinux integration, AppArmor, and seccomp support. These security mechanisms enforce mandatory access controls and container isolation, preventing unauthorized access to the host system. By incorporating multiple layers of security, Podman ensures a robust container runtime environment suitable for enterprise and IoT applications with strict security requirements.

Podman's lightweight and daemonless design contributes to enhanced performance and efficiency. It consumes fewer system resources than traditional container engines that rely on a long-running service, making it ideal for resource-constrained environments like edge devices. By eliminating the need for a daemon, Podman enables optimized use of CPU and memory, ensuring efficient container execution on low-power devices.

As a key component of the Red Hat ecosystem, Podman integrates seamlessly with OpenShift and MicroShift. It is fully supported on RHEL 9.5, making it a suitable container management tool for enterprise and edge computing projects. Podman's compatibility with Red Hat Satellite and other Red Hat management tools simplifies large-scale deployment, ensuring consistent container management across cloud, edge, and on-premises environments.

### 3.5.2. Conclusion and Justification for Podman Selection

Podman is a powerful, secure, and efficient container management tool that is well-suited for edge and IoT environments. Its daemonless architecture, Docker CLI compatibility, rootless operation, and security features make it an ideal choice for managing containers in resource-constrained settings. Podman's ability to work seamlessly with Kubernetes and OpenShift tools further enhances its suitability for deployment in edge computing environments, including those using MicroShift. Its lightweight nature and integration with the Red Hat ecosystem make it an excellent choice for this project, ensuring both performance efficiency and strong security in diverse edge computing applications.

## 3.6. OpenShift CLI (oc)

The OpenShift CLI (oc) is a command-line tool used for interacting with and managing OpenShift clusters. It enables users to perform a variety of administrative and operational tasks, including deploying applications, managing resources, and troubleshooting issues within an OpenShift environment. The oc tool integrates with OpenShift's Kubernetes-based platform, providing a convenient interface for users and administrators to manage and automate containerized workloads, offering a simplified yet powerful method for cluster and application management.

### 3.6.1. Key Features of OpenShift CLI (oc)

The OpenShift CLI (oc) is a powerful command-line tool that enables users to interact with OpenShift clusters efficiently. It provides essential functionalities for managing cluster resources, deploying applications, automating builds, and ensuring security and monitoring capabilities.

The oc CLI facilitates cluster management by allowing users to create, modify, and delete OpenShift resources, including pods, services, deployments, and routes. This ensures that applications and infrastructure components are configured and managed seamlessly.

Application deployment and management are streamlined through commands such as oc new app, oc expose, and oc apply, which enable users to deploy containerized applications, configure services, and expose applications to external traffic. These commands simplify the application lifecycle by handling deployment, scaling, and service configuration within the OpenShift environment.

Namespace and resource management is another key feature of oc. It allows users to manage resources within specific namespaces, making it ideal for multi-tenant environments. Commands like oc get, oc describe, and oc delete enable users to list, inspect, and remove resources as needed, ensuring efficient organization and access control.

Since OpenShift is built on Kubernetes, the oc CLI includes many of the same commands as kubectl, while also providing additional OpenShift-specific capabilities. This ensures compatibility with Kubernetes workflows while enhancing OpenShift's functionality for complex application management.

The oc CLI supports build and deployment automation, allowing users to streamline continuous delivery workflows. The oc new-build command initiates automated builds from source code repositories, while oc rollout manages deployment strategies, including rolling updates and rollbacks. These features ensure smooth and efficient application deployment.

Security and access control are integrated within oc, enabling robust authentication and authorization management. Commands such as oc policy and oc adm allow fine-grained control over roles, permissions, and security contexts, ensuring that only authorized users can access and modify cluster resources.

For monitoring and troubleshooting, the oc CLI provides commands like oc logs, oc describe pod, and oc exec, which allow users to inspect container logs, analyze resource utilization, and execute commands inside running containers. These tools help diagnose and resolve issues quickly within the OpenShift environment.

As a core component of the Red Hat ecosystem, oc integrates seamlessly with tools such as Podman, OpenShift Operator, and Red Hat Ansible. This integration facilitates automation,

container orchestration, and streamlined management across multiple environments, enhancing efficiency in enterprise and edge computing use cases.

The oc CLI also supports custom resources and operators, allowing users to extend OpenShift's capabilities. Commands like oc create and oc apply enable the management of custom Kubernetes resources and operators, automating operational tasks and simplifying the lifecycle management of complex applications.

#### 3.6.2. Conclusion and Justification for OpenShift CLI (oc) Selection

The OpenShift CLI (oc) is a vital tool for managing and interacting with OpenShift clusters. Its comprehensive set of commands supports application deployment, resource management, security configuration, and troubleshooting. By integrating deeply with Kubernetes and the Red Hat tools suite, oc simplifies the management of containerized applications, making it indispensable for OpenShift and Kubernetes environments, particularly in production environments where automation and efficient management are critical.

# 4. Proposed Solution/Implementation

## 4.1. Installing and Configuring MicroShift on Red Hat Enterprise Linux (RHEL) 9

MicroShift installation and configuration on RHEL 9 involve several key steps to ensure proper operation. These steps include enabling repositories, installing necessary packages, and configuring the system for MicroShift.

### 4.1.1. Enabling Repositories

Before installing MicroShift, it is essential to enable the required repositories. This ensures that all necessary packages for OpenShift and Fast Data Path (FDP) are available. The following command enables these repositories based on the system's architecture.

| Command | Description |
|---|---|
| sudo subscription-manager repos --enable rhocp-4.16-for-rhel-9-$(uname -m)-rpms --enable fast-datapath-for-rhel-9-$(uname -m)-rpms | Enables the OpenShift Container Platform (OCP) 4.16 repository and the Fast Data Path repository for RHEL 9, ensuring access to required packages for MicroShift installation. |

*Table1: Enabling Required Repositories for MicroShift Installation*

By executing this command, administrators ensure that the system can download and install MicroShift and its dependencies seamlessly.

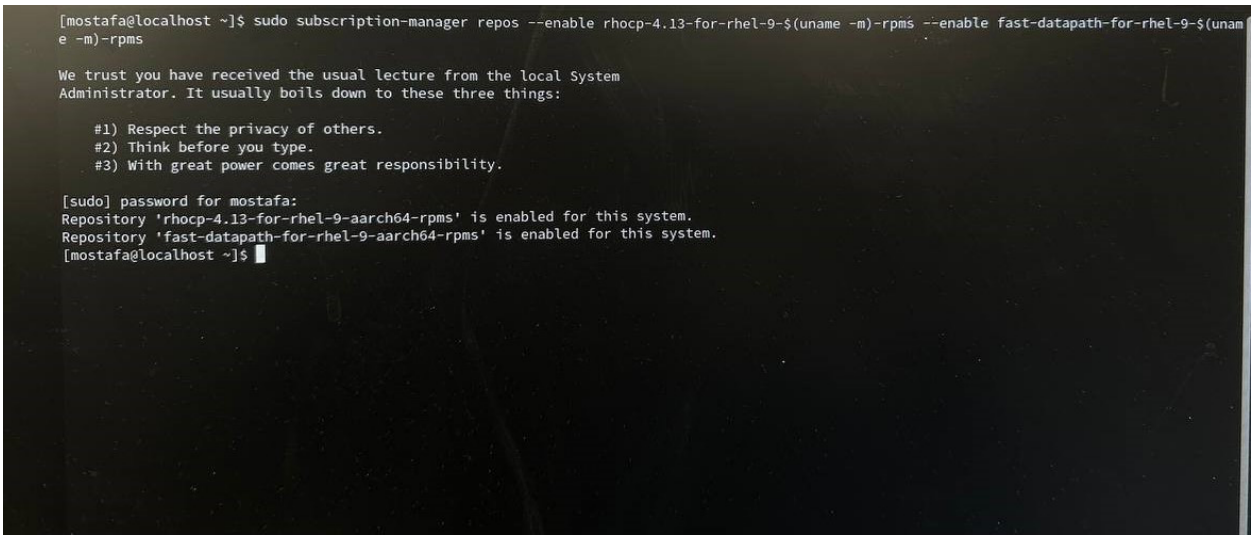



*Figure 1: Enabling Required Repositories for MicroShift Installation*

### 4.1.2. Installing MicroShift

Once the required repositories are enabled, MicroShift can be installed using the dnf package manager. The following command ensures that MicroShift and its dependencies are installed:

| Command | Description |
|---|---|
| sudo dnf install -y microshift | Installs MicroShift along with all required dependencies. The -y flag automatically confirms the installation, avoiding manual prompts. |

*Table 2: MicroShift Installation*

This command ensures that the MicroShift container runtime and other necessary components are set up on the system.

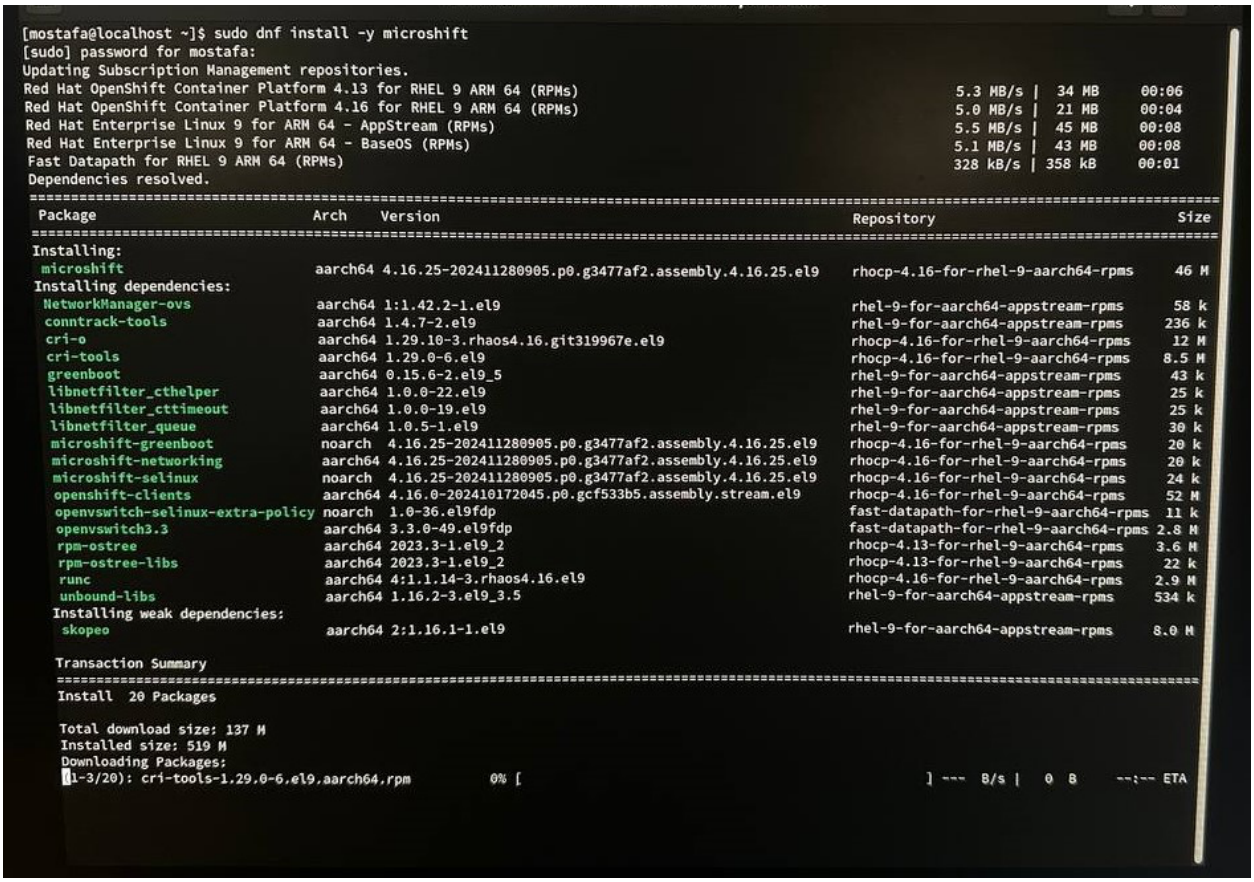



*Figure 2: MicroShift Installation*

### 4.1.3. Using the Pull Secret

To enable MicroShift to pull container images, a pull secret is required. This secret should be obtained from the OpenShift account under the "Pull Secrets" section and saved locally. The following commands ensure the pull secret is properly configured:

| Command | Description |
|---|---|
| sudo cp path/to/pull_secret /etc/crio/openshift-pull-secret | Copies the pull secret to the correct directory for CRI-O. |
| sudo chown root:root /etc/crio/openshift-pull-secret | Sets the file ownership to root, ensuring only the root user has access. |
| sudo chmod 600 /etc/crio/openshift-pull-secret | Restricts file permissions, allowing only the root user to read and write the file. |

*Table 3: Using the Pull Secret*

These steps ensure that MicroShift has the necessary credentials to pull container images securely.

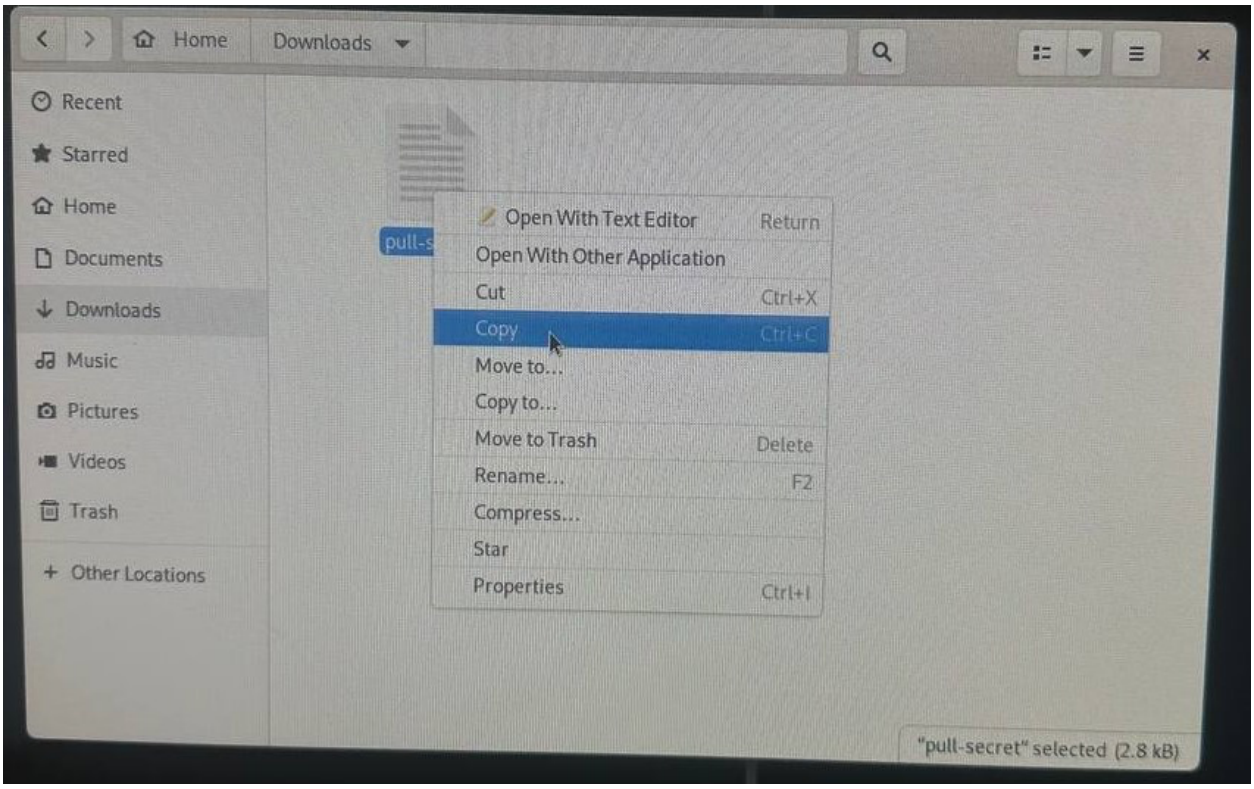



*Figure 3: Copying the pull secret*


*Figure 4: Sets the file ownership to root*


*Figure 5: Restricts file permissions*

### 4.1.4. Configuring the Firewall

MicroShift requires specific network configurations to function properly. The following IP ranges must be added to the trusted firewall zone to allow internal communication and metadata access.

| Command | Description |
|---|---|
| sudo firewall-cmd --permanent --zone=trusted --add-source=10.42.0.0/16 | Adds the 10.42.0.0/16 subnet to the trusted zone for internal networking. |
| sudo firewall-cmd --permanent --zone=trusted --add-source=169.254.169.1 | Adds 169.254.169.1 to the trusted zone for metadata access. |
| sudo firewall-cmd --reload | Reloads the firewall to apply the changes. |

*Table 4: Configuring the Firewall*

These steps ensure that MicroShift can communicate properly within the cluster and access necessary metadata.

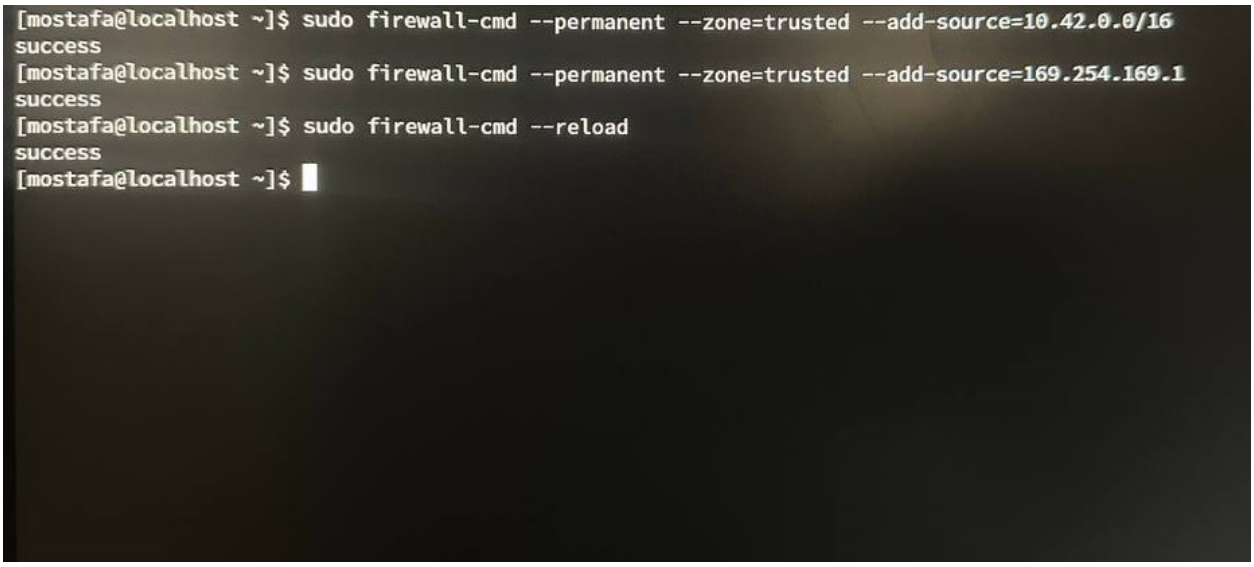



*Figure 6: Configuring the Firewall*

### 4.1.5. Enabling and Starting MicroShift

After completing the configuration, the MicroShift service should be enabled and started to ensure it runs immediately and restarts on system boot.

| Command | Description |
|---|---|
| sudo systemctl enable --now microshift | Enables and starts the MicroShift service immediately, ensuring it auto-starts on boot. |
| sudo systemctl status microshift | Checks and displays the status of the MicroShift service to confirm it is active and running. |

*Table 5: Enabling and Starting MicroShift*

These steps ensure that MicroShift is properly initialized and will persist across reboots.


*Figure 7: Enabling and Starting MicroShift*

### 4.1.6. Verifying Installation

To confirm the success of the installation, you can check the installed version of MicroShift:

| Command | Description |
|---|---|
| microshift version | Displays the installed version of MicroShift, confirming a successful installation. |

*Table 6: Verifying Installation MicroShift*

*Running this command will show the version, helping to verify that MicroShift is installed correctly.*

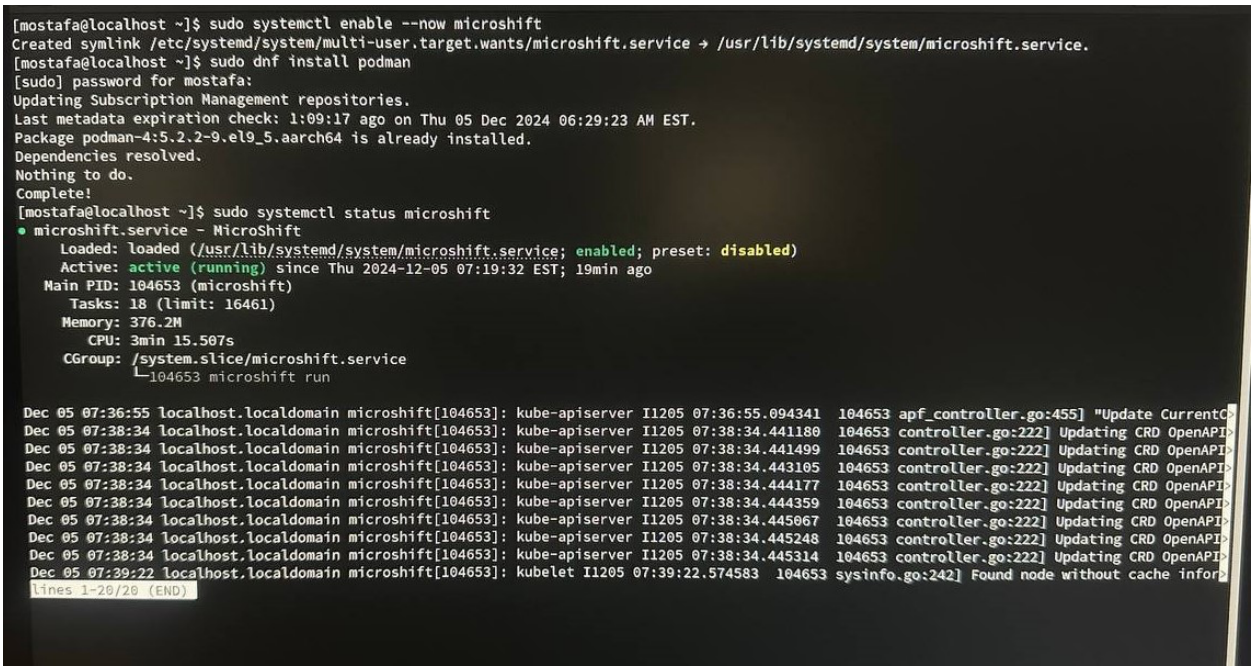



*Figure 8: Enabling and Starting MicroShift*

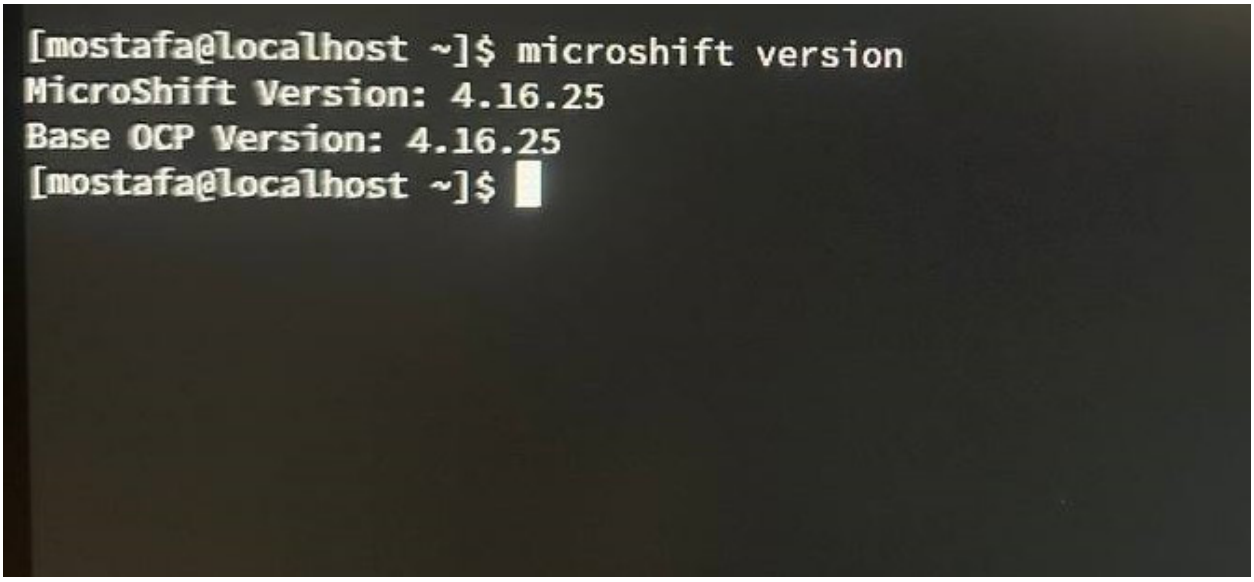



*Figure 9: Verifying Installation*

### 4.1.7. Installing Podman

To install and manage Podman, the following steps should be taken:

| Command | Description |
|---|---|
| sudo dnf install podman | Installs Podman, a container management tool for MicroShift. |
| sudo systemctl enable podman | Enables Podman to start automatically on system boot. |
| sudo systemctl start podman | Starts the Podman service immediately. |
| sudo systemctl status podman | Checks the status of the Podman service to confirm if it is active and running. |

*Table 7: Installing Podman*

These commands ensure that Podman is installed, started, and configured to run on boot.

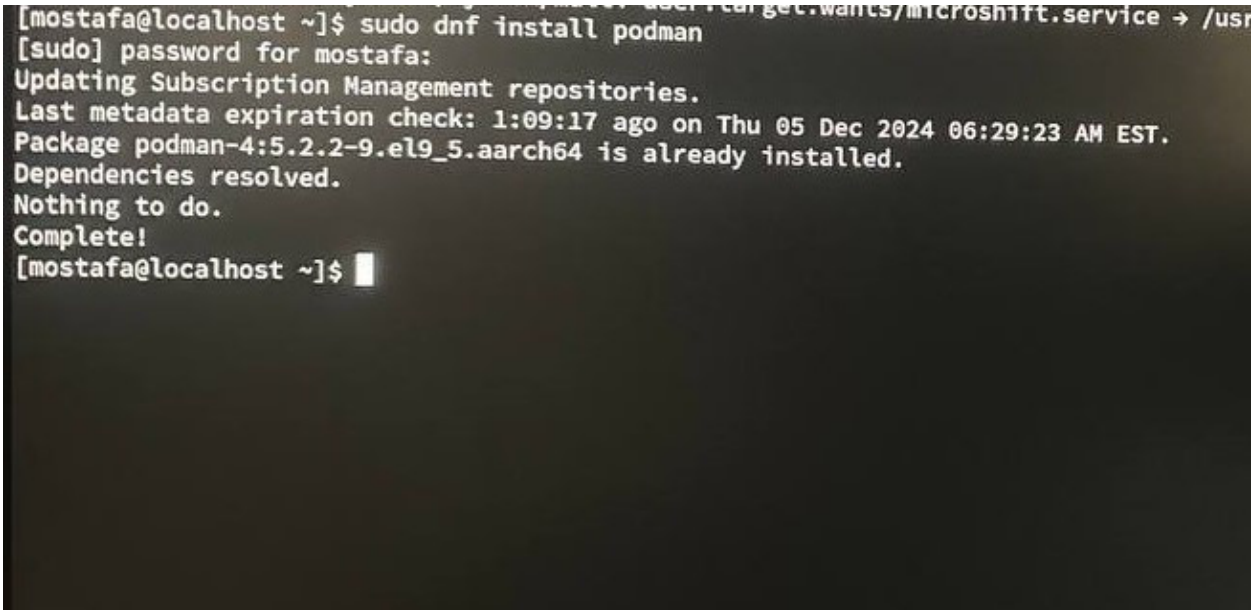



*Figure 10: Installing Podman*

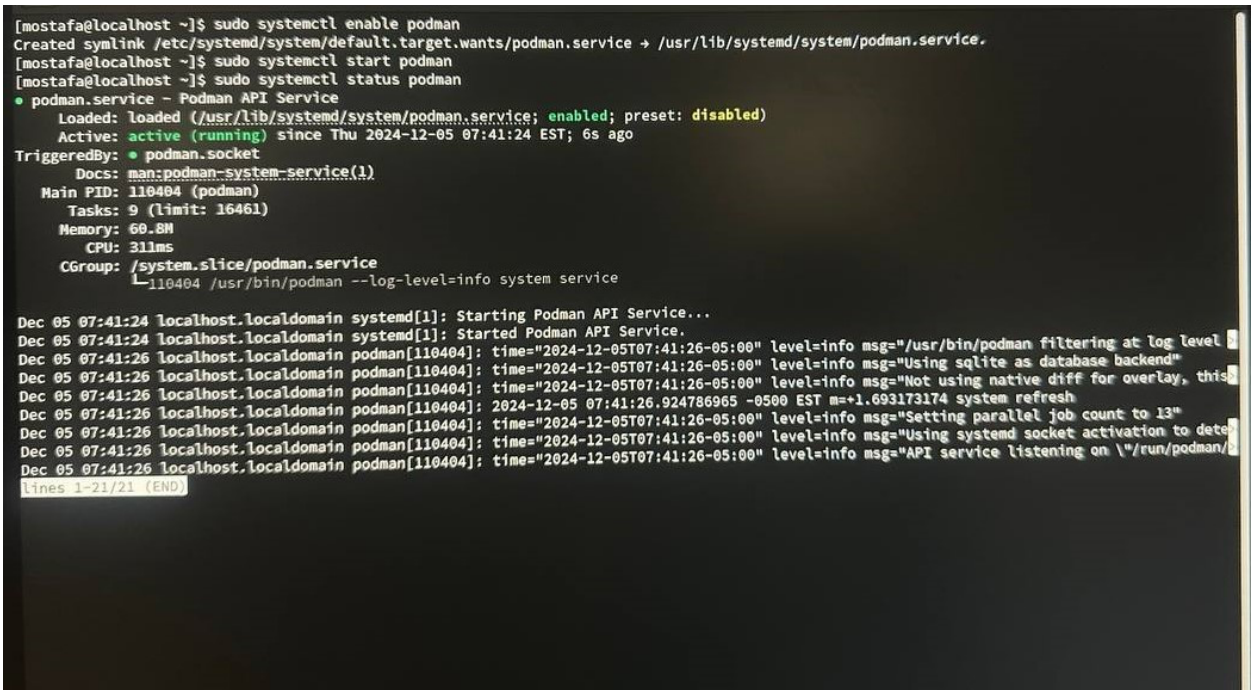



*Figure 11: Enabling podman*

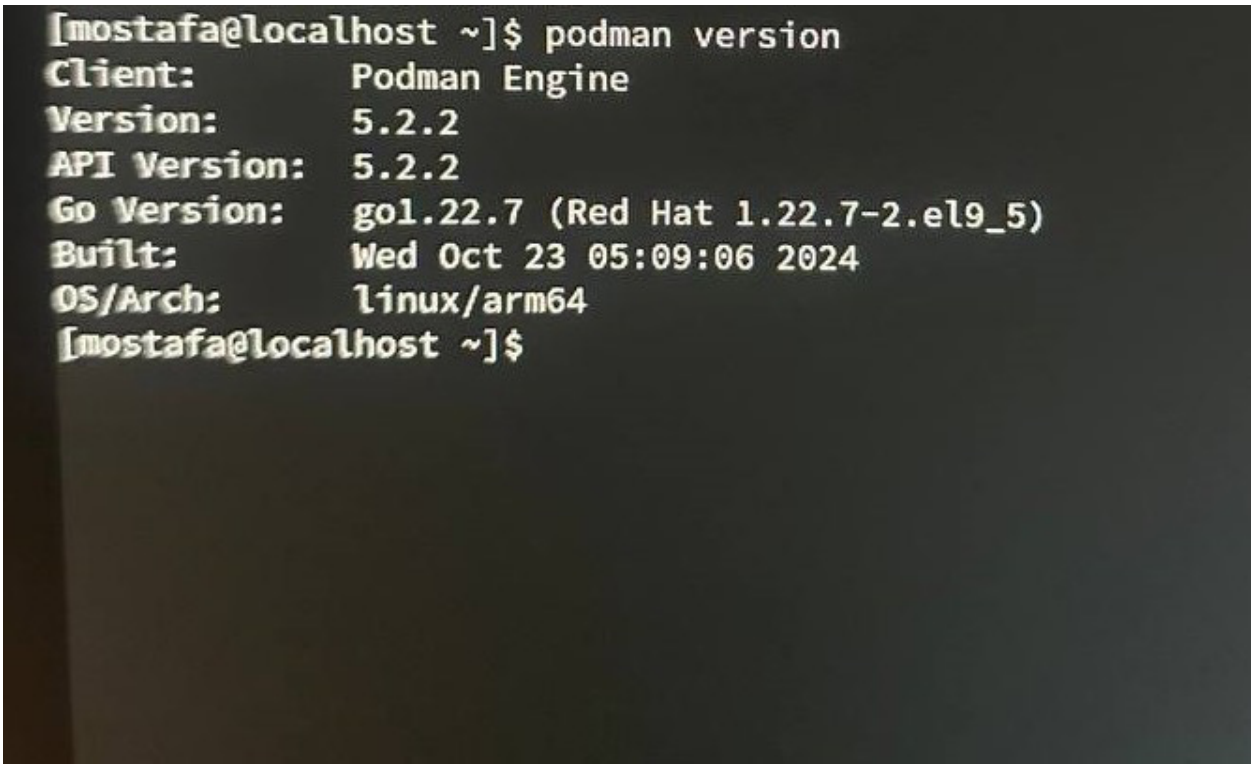



*Figure 12: Verifying Podman Installation*

### 4.1.8. Installing OpenShift CLI (oc)

Before starting the setup process for MicroShift and OpenShift CLI on Red Hat Enterprise Linux (RHEL), several prerequisites need to be met.

First, the system must be registered with Red Hat Subscription Management (RHSM). This ensures access to Red Hat repositories, software, and updates. The system's registration status should be verified, and if it is not registered, the registration process must be completed. This registration allows the system to access all required packages.

Next, the system must be up-to-date with the latest packages and security patches. Ensuring that the system is fully updated reduces the risk of vulnerabilities and ensures compatibility with the MicroShift and OpenShift CLI components. Regular updates are essential to maintaining system security and stability.

Finally, the system needs to have access to the required repositories for OpenShift and MicroShift components. Enabling the necessary repositories will allow the installation of these tools and

dependencies needed to run OpenShift and MicroShift on the RHEL system. Once the repositories are enabled, the system will be ready for installation.

After these prerequisites have been verified and the system is properly prepared, you can proceed with the installation of the MicroShift and OpenShift CLI components.

| Task | Command |
|---|---|
| Check RHEL system registration status | sudo subscription-manager status |
| Register RHEL system (if not already done) | sudo subscription-manager register |
| Update all installed packages | sudo dnf update -y |
| Enable required RHEL repositories | sudo subscription-manager repos --enable=rhel-8-for-x86_64-appstream-rpms<br>sudo subscription-manager repos --enable=rhel-8-for-x86_64-baseos-rpms<br>sudo subscription-manager repos --enable=openshift-4.8-for-rhel-8-x86_64-rpms |
| Install OpenShift CLI tools | sudo dnf install -y openshift-clients |
| Download MicroShift package | sudo dnf install -y microshift |
| Start the MicroShift service | sudo systemctl start microshift |
| Enable the MicroShift service on boot | sudo systemctl enable microshift |

*Table 8: Installing OpenShift CLI (oc)*

These commands go through the necessary setup steps, ensuring that the RHEL system is ready to run MicroShift and OpenShift CLI.

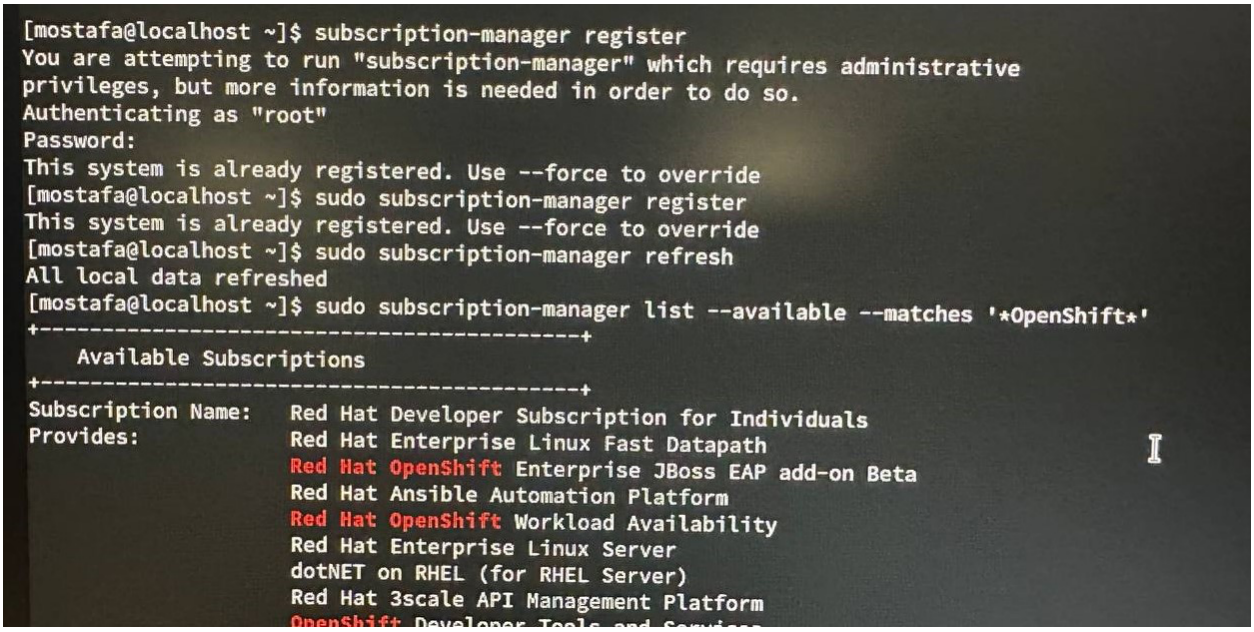



*Figure 13: Register RHEL system*

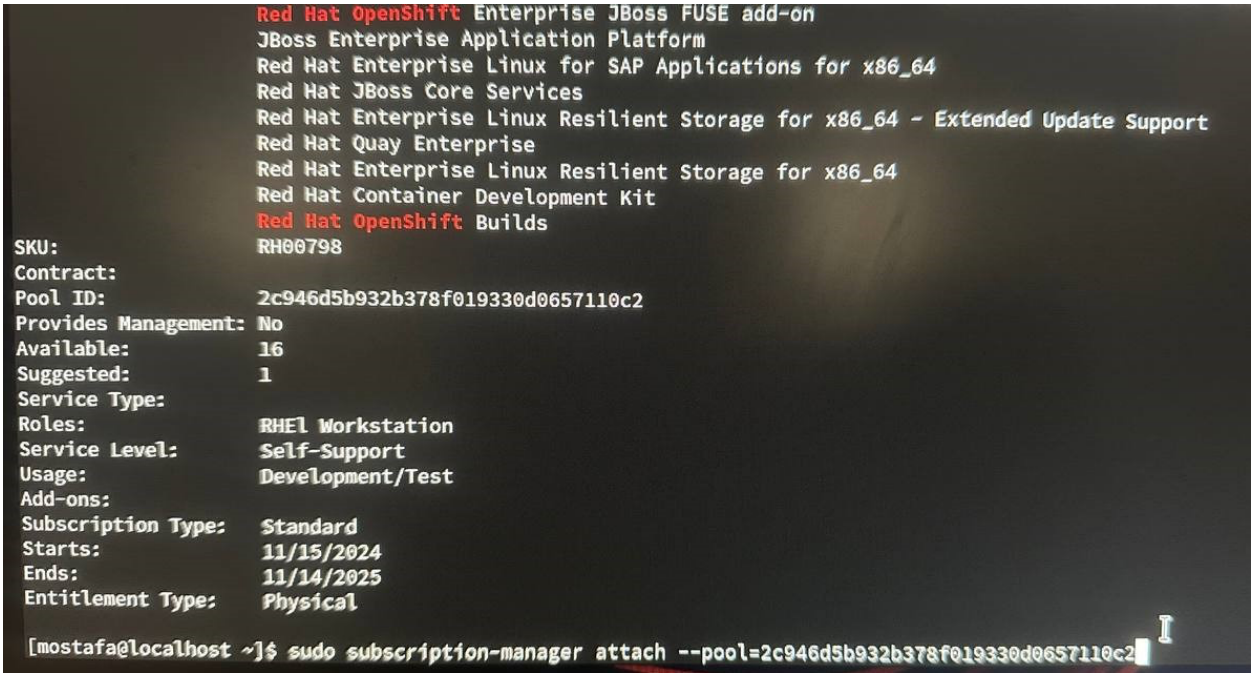



*Figure 14: Register RHEL system*

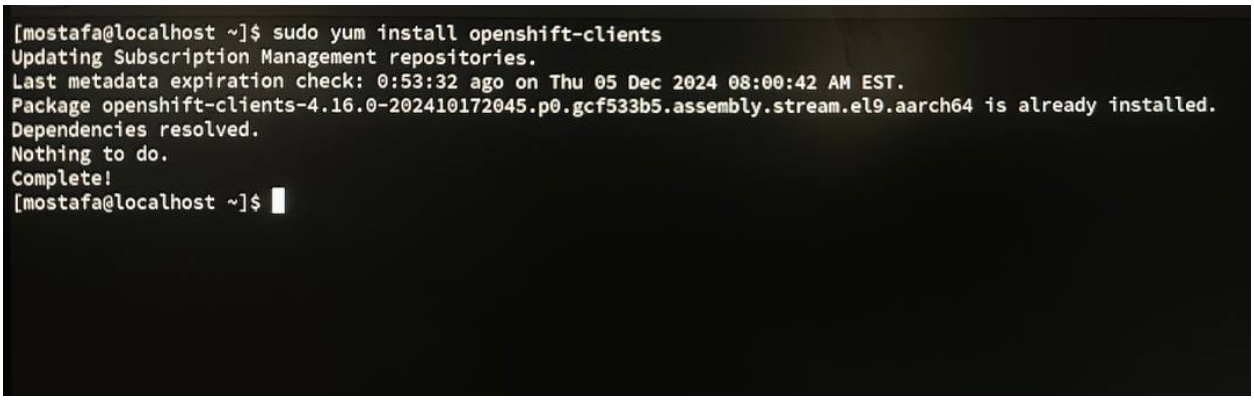



*Figure 15: Install OpenShift CLI tools*


*Figure 16: Verifying oc Installation*

### 4.1.9. Enabling Tab Completion for oc

To enable tab completion for oc commands, follow these steps:

| Command | Description |
|---|---|
| oc completion bash > oc_bash_completion | Generates the oc bash completion script and saves it to a file named oc_bash_completion. |
| sudo cp oc_bash_completion /etc/bash_completion.d/ | Copies the generated completion script to the appropriate directory for bash completion. |

*Table 9: Enabling Tab Completion for OpenShift CLI (oc)*

This will enable tab completion for oc commands, making it easier to use in the terminal.


*Figure 17: Enabling Tab Completion for oc*

### 4.1.10. Configuring Kubernetes Context

#### 4.1.10.1. Creating Kubernetes Config Directory

The Kubernetes configuration directory, which is essential for storing the kubeconfig file required for accessing the MicroShift/OpenShift cluster, should be created. This directory should be located in the user's home directory. If it does not already exist, the directory will be created.

#### 4.1.10.2. Retrieving and Storing the KubeConfig File

Once the configuration directory has been created, the kubeconfig file, which contains the credentials and connection details for accessing the cluster, is copied to this directory. Storing the kubeconfig file in the correct directory allows access to the cluster and ensures that the necessary credentials are in place.

#### 4.1.10.3. Setting Correct File Permissions

To protect the kubeconfig file and ensure that it is secure, the file permissions are set correctly. By removing read permissions for others, only the file owner can access it, thereby enhancing the security of the credentials.

| Task | Command | Description |
|---|---|---|
| Enable and start MicroShift service | sudo systemctl enable --now microshift | The MicroShift service is enabled and started immediately, ensuring it auto-starts on boot. |
| Check the status of MicroShift service | sudo systemctl status microshift | The status of the MicroShift service is displayed to verify that it is active and running. |
| Create Kubernetes configuration directory | mkdir -p ~/.kube/ | The .kube directory is created in the home directory if it does not already exist. |
| Copy the kubeconfig file to the Kubernetes directory | sudo cat /var/lib/microshift/resources/kubeadmin/kubeconfig > ~/.kube/config | The kubeconfig file is copied to the .kube/config directory for access to the MicroShift/OpenShift cluster. |
| Set correct permissions for the kubeconfig file | sudo chmod go-r ~/.kube/config | File permissions are set to protect the kubeconfig file from unauthorized access. |

*Table 10: Setting Correct File Permissions*

By following these steps, the Kubernetes context is configured properly, the kubeconfig file is securely stored, and MicroShift is initialized and accessible for use.

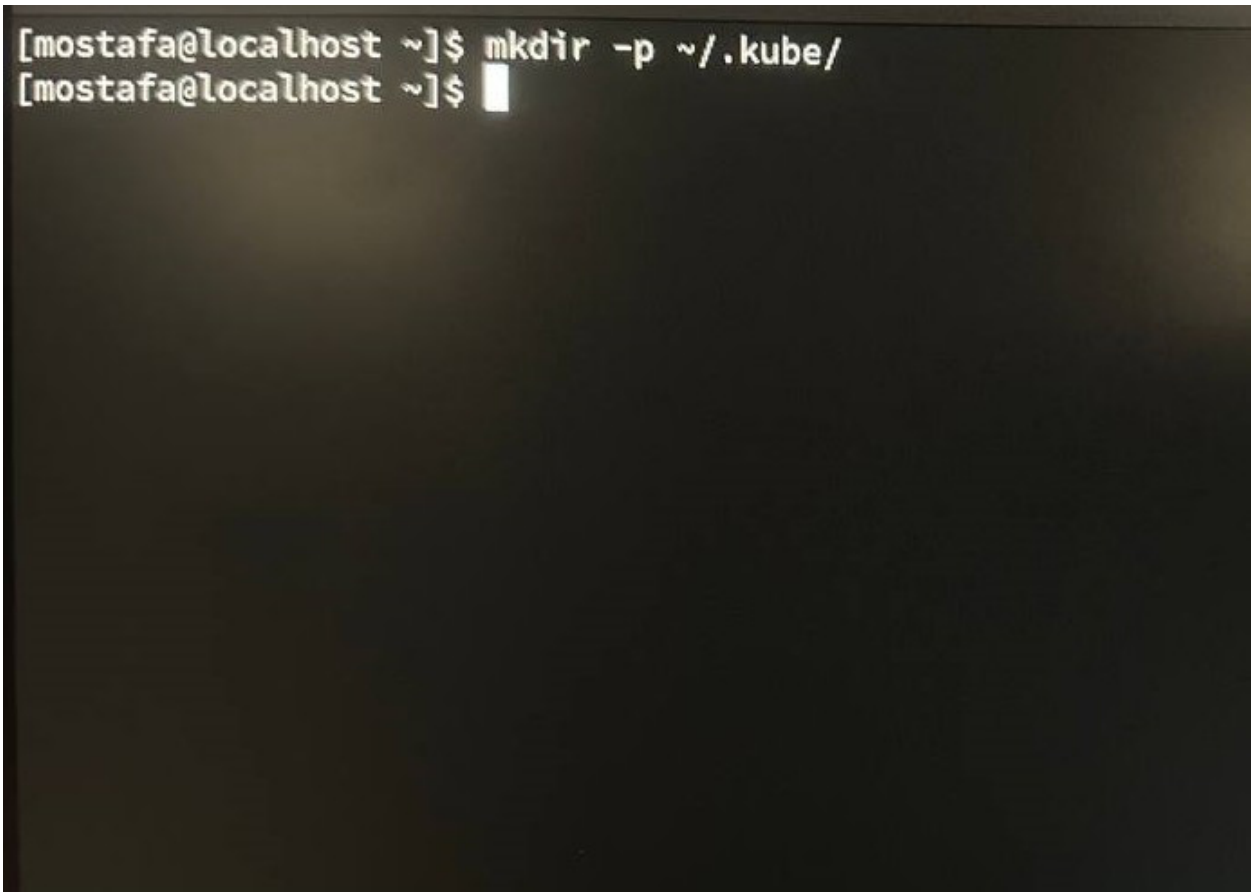



*Figure 18: Create Kubernetes configuration directory*


*Figure 19: Set correct permissions for the kubeconfig file*

### 4.1.11. Managing MicroShift Cluster with oc

Once the configuration is complete, oc commands can be used to interact with the cluster.

| Command | Description |
|---|---|
| oc get all -A | Displays all resources across all namespaces in the cluster, providing a comprehensive overview of the cluster's state. |

*Table 11: Managing MicroShift Cluster with oc*

This allows the user to verify the overall status and availability of resources within the MicroShift/OpenShift cluster.

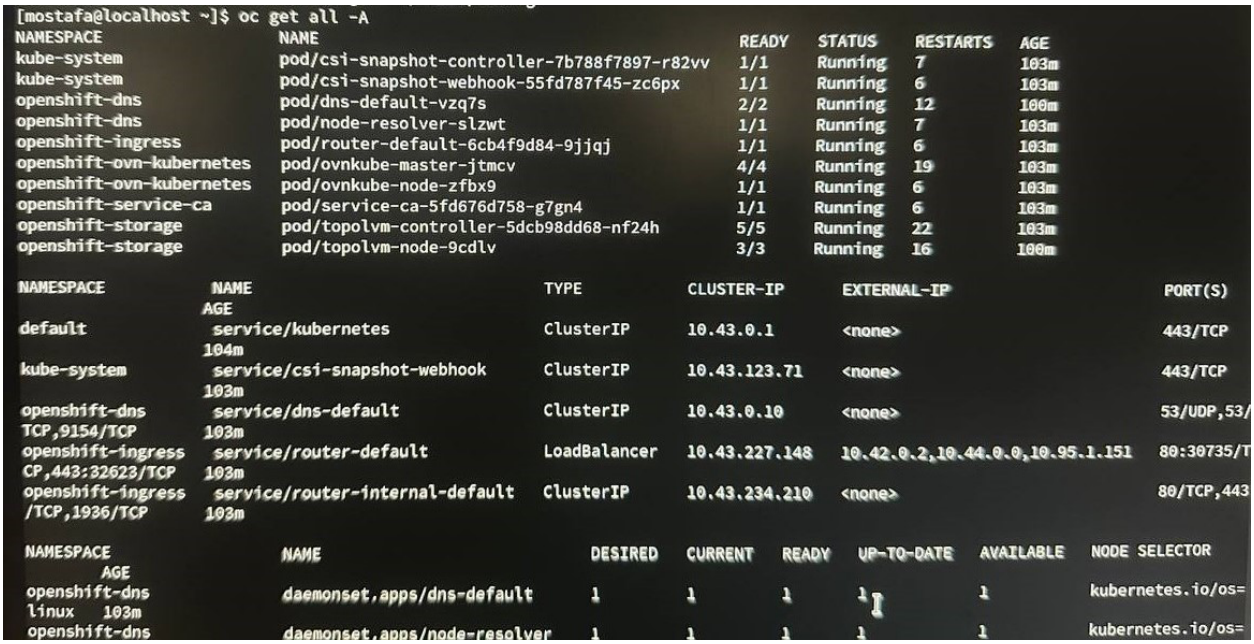



*Figure 20: Managing MicroShift Cluster with oc*

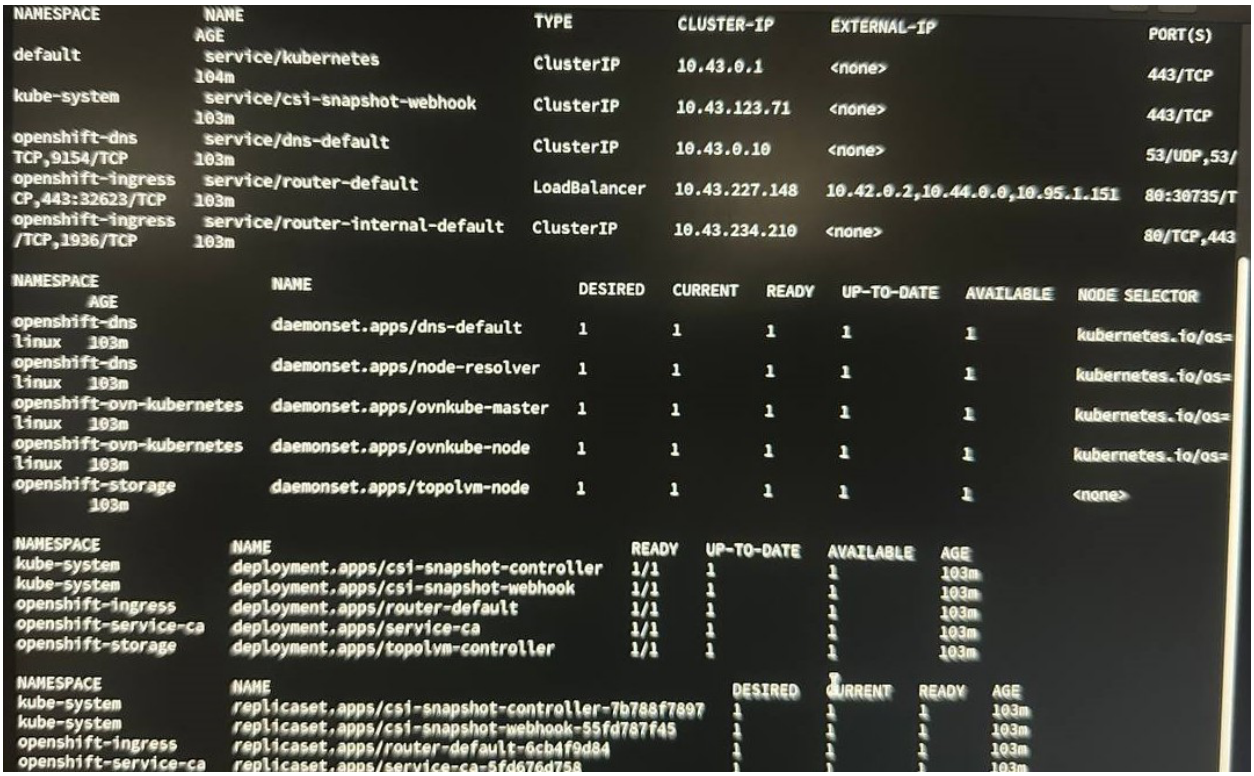



*Figure 21: Managing MicroShift Cluster with oc*

### 4.1.12. Troubleshooting

In case of issues, system logs can be reviewed to diagnose problems.

| Command | Description |
|---|---|
| sudo journalctl -u microshift | Displays the system logs for the MicroShift service, providing insights into errors or issues related to its operation. |
| sudo journalctl -u podman | Displays the system logs for the Podman service, helping diagnose issues related to container management within MicroShift. |

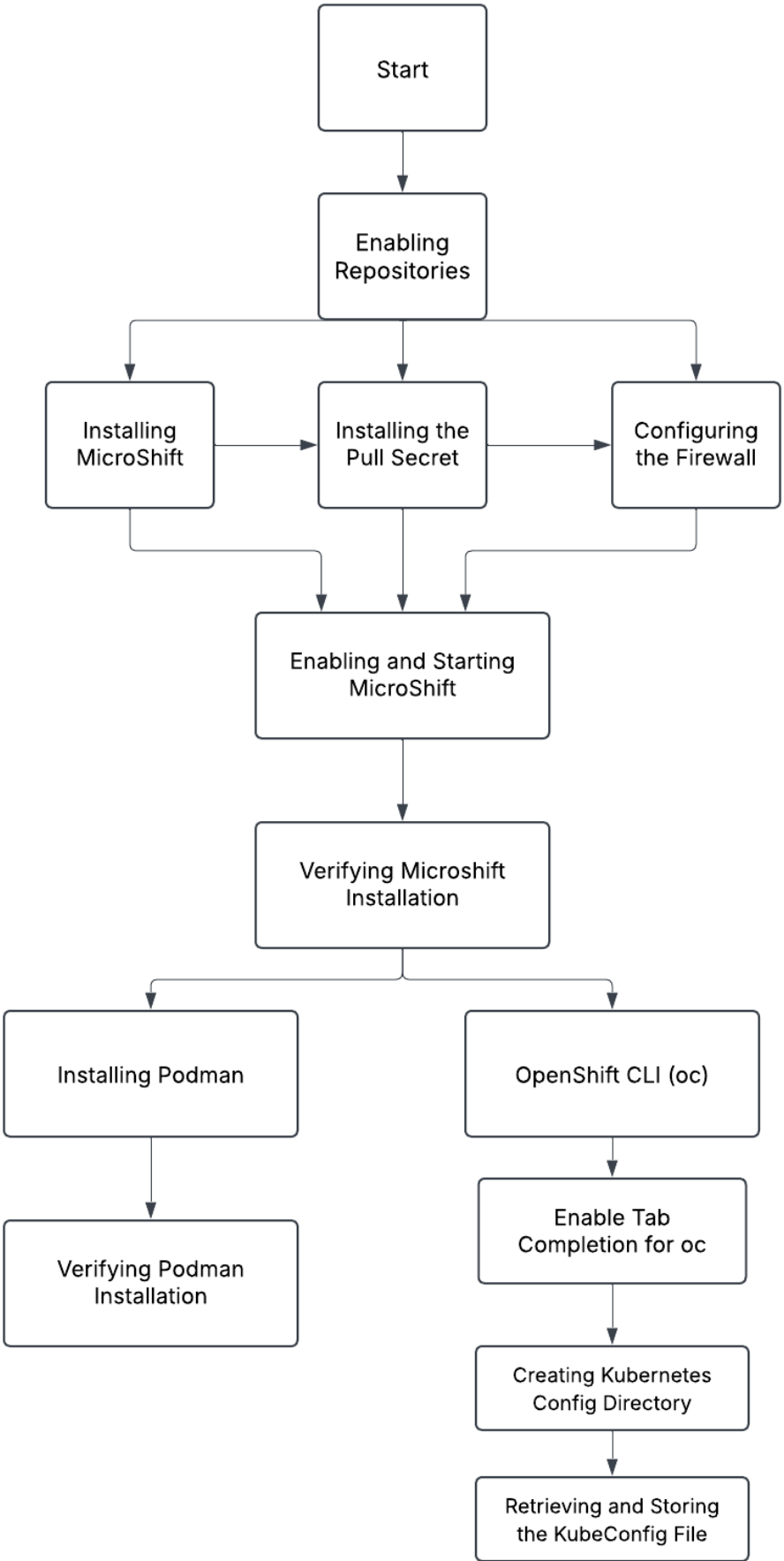
*Table 12: Troubleshooting*

These logs are essential for troubleshooting and identifying any errors or misconfigurations during installation or service startup.

### 4.1.13. Key Considerations

MicroShift and OpenShift CLI should be kept up to date by regularly using dnf update to install the latest versions. It is also crucial to securely store the kubeconfig file, ensuring that its permissions prevent unauthorized access.

Finally, monitoring resource usage is essential for ensuring that MicroShift runs effectively in resource-constrained environments. System performance, including CPU and memory usage, should be regularly monitored to ensure the optimal operation of the MicroShift cluster.


Start
Enabling Repositories
Installing MicroShift
Installing the Pull Secret
Configuring the Firewall
Enabling and Starting MicroShift
Verifying Microshift Installation
Installing Podman
Verifying Podman Installation
OpenShift CLI (oc)
Enable Tab Completion for oc
Creating Kubernetes Config Directory
Retrieving and Storing the KubeConfig File

## 4.2. Implementation of Pod Creation and Management in MicroShift

MicroShift, a lightweight Kubernetes distribution by Red Hat, facilitates containerized application deployment through multiple methods. The creation and management of pods can be executed using Podman, OpenShift CLI (oc), or Kubernetes YAML manifests. This section provides a detailed implementation of pod deployment and management using these tools.

### 4.2.1 Methods for Pod Creation

#### 4.2.1.1. Using Podman

Podman, a daemonless container management tool, enables the creation and management of containers and pods efficiently. Unlike Docker, Podman does not require a daemon process to run containers, making it a lightweight alternative suitable for local development and testing.

##### 4.2.1.1.1. Pod Creation with Podman

Podman allows users to create, manage, and run containers and pods in a simple, efficient manner. It’s often used as an alternative to Docker, especially in environments where a lightweight and secure container engine is required. In the context of MicroShift, which is a lightweight Kubernetes distribution designed to run on resource-constrained environments like IoT devices, Podman can be used to create and manage pods locally before deploying them into MicroShift.

#### 4.2.1.2. Using OpenShift CLI (oc)

The OpenShift CLI (oc) provides an efficient interface for managing OpenShift clusters, including the creation and deployment of pods.

##### 4.2.1.2.1. Pod Creation with oc

Pods can be created using various methods such as oc run, oc new-app, or by applying Kubernetes YAML manifests. Each method offers flexibility depending on the user's needs and the complexity of the deployment.

OpenShift ensures efficient resource scheduling, networking, and security policy enforcement, optimizing the operation of containerized applications across the cluster.

Although standalone pods can be created directly, it is generally preferred to use a Deployment for pod management. Deployments offer several advantages, including scalability, automatic restarts, and version management. These features ensure the availability and resilience of applications running within the cluster.

#### 4.2.1.3. Using Kubernetes YAML Manifests

Kubernetes environments, including MicroShift, primarily utilize YAML manifests for defining pod configurations.

##### 4.2.1.3.1. Defining a Pod in YAML

Pod specifications in YAML include essential details such as container images, environment variables, resource limits, and networking configurations. These specifications are critical for managing containerized applications within a Kubernetes-based environment.

The manifest can be applied using the kubectl apply -f <name>.yaml or oc apply -f <name>.yaml commands in MicroShift environments. These commands ensure that the pod configuration is successfully deployed to the cluster.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: multi-pod
  namespace: test-namespace
spec:
  replicas: 100
  selector:
    matchLabels:
      app: multi-container-app
  template:
    metadata:
      labels:
        app: multi-container-app
    spec:
      # Security settings for the whole pod
      securityContext:
        runAsNonRoot: true           # Ensure containers don't run as root
        seccompProfile:
          type: RuntimeDefault      # Apply default seccomp profile for improved security
      containers:
        - name: container-1
          image: registry.redhat.io/ubi8/python-39:latest
          command: ["python3", "-m", "http.server", "8080"]
          securityContext:
            allowPrivilegeEscalation: false  # Prevent privilege escalation
            capabilities:
              drop:
                - ALL  # Drop all capabilities to limit container privileges
        - name: container-2
          image: registry.redhat.io/ubi8/python-39:latest
          command: ["python3", "-m", "http.server", "8081"]
          securityContext:
            allowPrivilegeEscalation: false
            capabilities:
              drop:
                - ALL
        - name: container-3
```

*Figure 22: YAML Deployment*

## 4.3. Implementation: Pod Deployment in MicroShift

For this case study, a Python-based web server was deployed using the official Red Hat Python 3.9 image obtained from registry.redhat.io. The following steps outline the process of pod deployment and management.

### 4.3.1. Selection of the Container Image

The Python 3.9 container image from Red Hat's registry was chosen due to its pre-configured environment, ensuring compatibility and security.

### 4.3.2. Pod Creation Using OpenShift CLI

The OpenShift CLI (oc) was utilized to create and manage the pod efficiently. This approach enabled automated scheduling, networking, and resource allocation.

### 4.3.3. Pod Execution Workflow

The container runtime (CRI-O) retrieved the Python image from the Red Hat registry, ensuring the necessary image for execution was available.

The pod executed a Python-based web server, which was designed to listen for HTTP requests and handle incoming traffic efficiently.

The pod's network namespace facilitated seamless communication between containers within the same pod, enabling them to share networking resources and interact with each other effectively.

### 4.3.4. Container Management in MicroShift

Several of OpenShift's built-in functionalities were leveraged to enhance the management of the pod. Continuous monitoring of the pod's health and resource usage was implemented to ensure optimal performance and quickly detect any potential issues. In the event of a failure, automatic container restarts were enabled, which ensured high availability and minimized downtime. Additionally, the scaling capabilities were utilized to handle increased load, allowing the system to automatically adjust resources in response to varying traffic demands.

### 4.3.5. Exposing the Application

To ensure the web server was accessible from external networks, an OpenShift Service and Route were created. The Service was responsible for exposing the pod internally within the OpenShift cluster, allowing it to be discovered and accessed by other services within the cluster. The Route, on the other hand, facilitated the management of external ingress traffic, directing incoming requests to the appropriate pod, thereby enabling external users to access the web server.

## 4.4. Deploying Pods on MicroShift

Pods were deployed using a YAML manifest file (m.yaml), specifying the desired pod configuration, including containers, resources, networking, and storage settings.

### 4.4.1. Applying the Manifest File

The following command was executed to apply the YAML configuration and create the pod:

| Command | Description |
|---|---|
| oc apply -f m.yaml | Applies the YAML configuration file (m.yaml) to create the pod. |

*Table 13: Applying the Manifest File*

### 4.4.2. Pod Deployment Process

#### 4.4.2.1. Manifest Processing

The YAML file (m.yaml) was parsed to determine the necessary resource creation steps. If the resource did not exist, it was created; otherwise, an update was applied to match the new configuration.

#### 4.4.2.2. Pod Creation and Scheduling

The MicroShift API server registered the pod definition. Based on available resources, the Scheduler selected an appropriate node for the pod to run on.

#### 4.4.2.3. Container Image Retrieval

The CRI-O container runtime fetched the container image from registry.redhat.io. If the image was already present, a cached version was used to accelerate the deployment process.

#### 4.4.2.4. Container Initialization

The Kubelet initiated the pod and its containers. If an initContainer was defined, it executed first before the main containers started. Following this, the container entrypoint was executed, launching the application inside the container.

#### 4.4.2.5. Networking Configuration

An IP address was assigned to the pod within the MicroShift network. All containers within the same pod shared the same network namespace. If a Service was defined for the pod, an internal DNS entry was created to facilitate communication between pods.

#### 4.4.2.6. Readiness and Liveness Probes

A readiness check was performed to ensure the pod was ready to handle traffic. A liveness probe monitored the health of the pod, restarting failed instances when necessary to maintain reliability.

#### 4.4.2.7. Pod Transition to Running State

After the successful initialization of the container, the pod transitioned to the Running state, signaling that it was fully operational.

## 4.5. Monitoring Pod Status in MicroShift

After deployment, the pod status was monitored using the following commands:

| Action | Command | Description |
|---|---|---|
| List All Pods | oc get pods | Lists all pods in the current namespace. |
| Describe Pod Details | oc describe pod my-pod | Displays detailed information about a specific pod. |
| View Pod Logs | oc logs -f pod/my-pod | Follows and displays the logs of the specified pod. |
| Retrieve Pod Events | oc get events --sort-by=.metadata.creationTimestamp | Lists all events in the cluster, sorted by their creation timestamp. |

*Table 14: Monitoring Pod Status*

## 4.6. Common Pod Statuses

| Status | Description |
|---|---|
| ContainerCreating | Image is being pulled and container started |
| Running | Successfully running |
| CrashLoopBackOff | Container is repeatedly crashing |
| ErrImagePull | that there was an error while attempting to pull the container image |
| ImagePullBackOff | indicates that Kubernetes has failed to pull the container image multiple times and is now backing off from further attempts, |

*Table 15: Applying the Manifest File*

## 4.7. Overview of Pod Deployment and Management in MicroShift

This implementation demonstrated pod deployment in MicroShift using oc apply -f m.yaml. The deployment process included pod scheduling, image retrieval, container startup, and networking configuration. Efficient monitoring and troubleshooting methods were also outlined, ensuring seamless management of pods in a MicroShift environment.

# 5. Project Specifications (Upgrading)

## 5.1. Downgrading and Upgrading RHEL

The process began with downgrading RHEL to version 9.4. The following command was used to perform the downgrade:

| Operation | Command |
|---|---|
| Downgrade to RHEL 9.4 | /usr/bin/time -v sudo dnf -y downgrade redhat-release |

*Table 16: Downgrading RHEL*

After executing the command, the transaction steps were verified, including preparation, upgrading, cleanup, and final verification. Subsequently, RHEL was upgraded to version 9.5 using the command:

| Operation | Command |
|---|---|
| Upgrade to RHEL 9.5 | /usr/bin/time -v sudo dnf -y update redhat-release |

*Table 17: Upgrading RHEL*

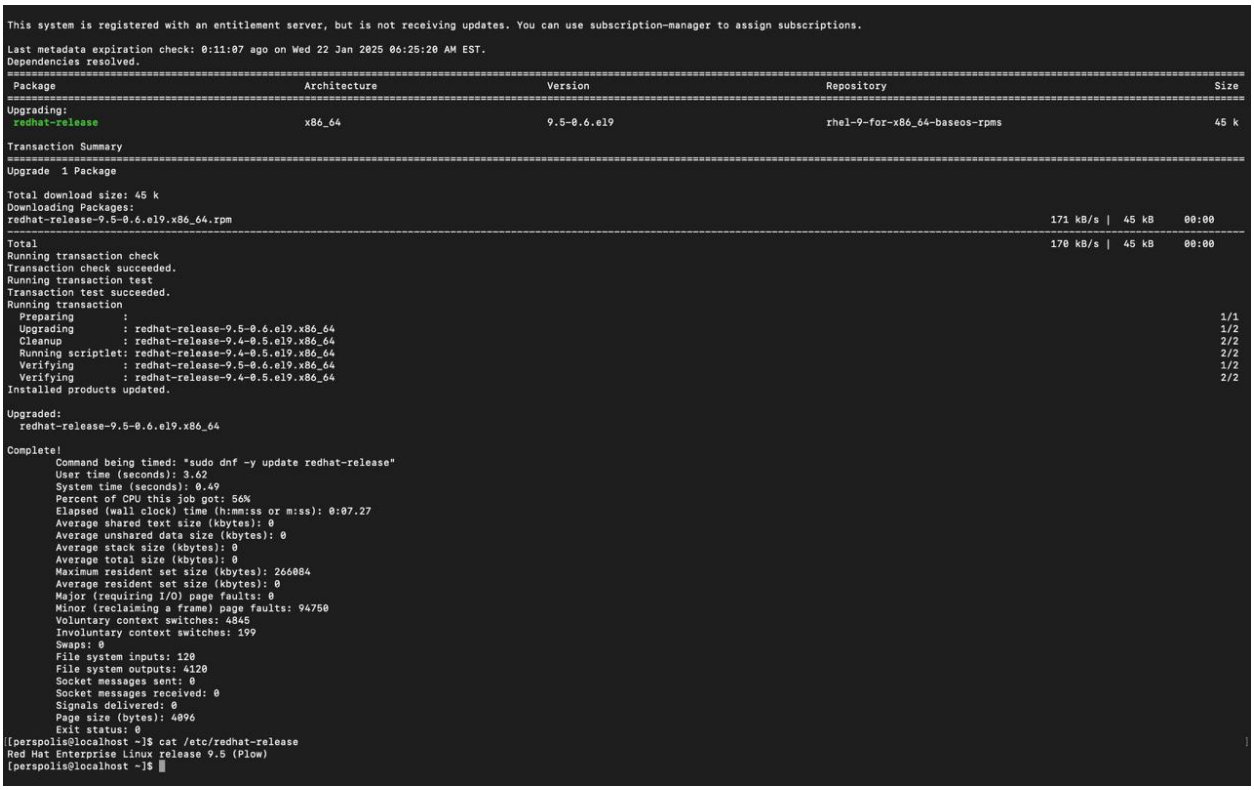



*Figure 23: Upgrade to RHEL 9.5*

The upgrade resulted in the installation of redhat-release-9.5-0.6.619.x86_64, with the transaction check, transaction test, and running transaction steps completing successfully. The resource usage measurements during the upgrade included a user time of 3.62 seconds, a system time of 0.49 seconds, a CPU usage of 56 percent, and a memory consumption of 266 megabytes.

| Measurement | Upgrade to RHEL 9.5 |
|---|---|
| User Time (s) | 3.62 |
| System Time (s) | 0.49 |
| CPU Usage (%) | 56 |
| Memory Consumption (MB) | 266 |

*Table 18: Measurements(Upgrade to RHEL 9.5)*

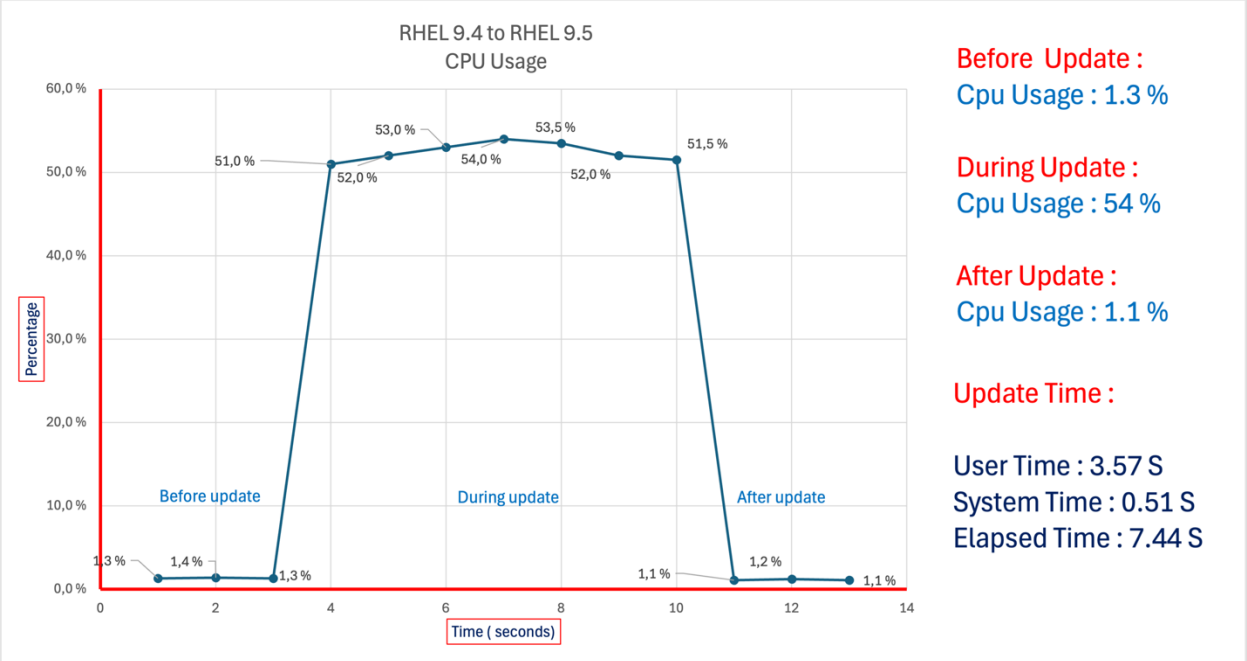



*Figure 24: Upgrade to RHEL 9.5(CPU Usage)*

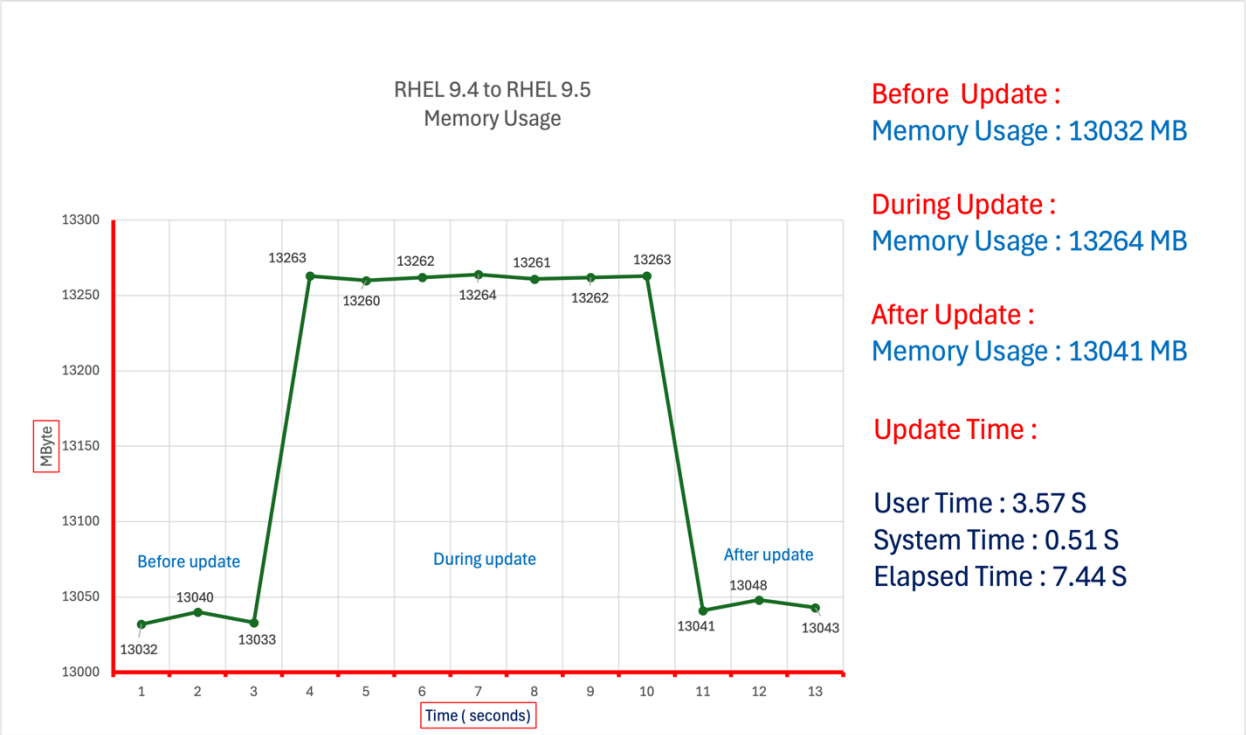



*Figure 25: Upgrade to RHEL 9.5(Memory Usage)*

## 5.2. Managing MicroShift

### 5.2.1. Downgrading Microshift

The MicroShift downgrade process involved moving from version 4.14.45 to version 4.13.21 using the following command:

| Operation | Command |
|---|---|
| Downgrade to MicroShift 4.13.21 | sudo dnf -y install microshift-4.13.21 |
| Time and resources measurement | /usr/bin/time -v sudo dnf -y install microshift-4.13.21 |

*Table 19: Downgrading Microshift*

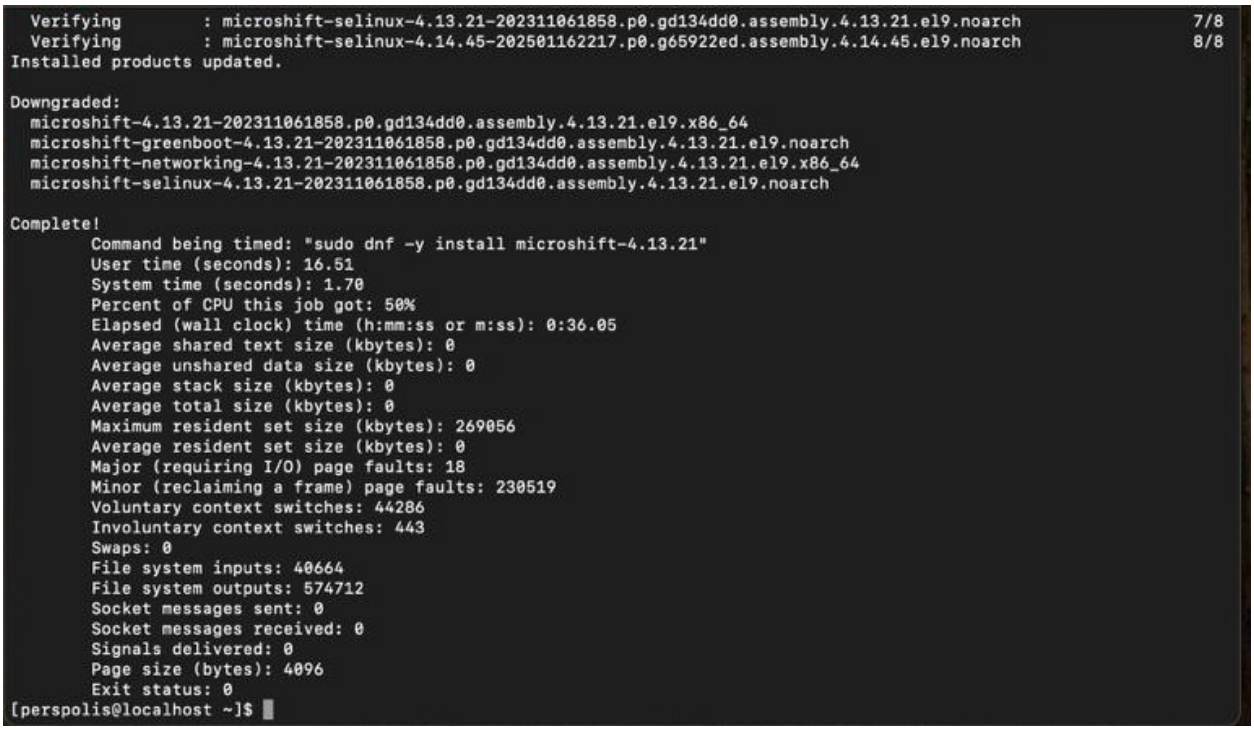



*Figure 26: Downgrade to MicroShift 4.13.21*

The downgrade installed the packages microshift-4.13.21, microshift-greenboot-4.13.21, microshift-networking-4.13.21, and microshift-selinux-4.13.21. The system recorded a user time of 16.51 seconds, a system time of 1.70 seconds, a CPU usage of 50 percent, and a memory consumption of 269 megabytes.

| Measurement | Downgrade to MicroShift 4.13.21 |
|---|---|
| User Time (s) | 16.51 |
| System Time (s) | 1.70 |
| CPU Usage (%) | 50 |
| Memory Consumption (MB) | 269 |

*Table 20: Measurements(Downgrade to MicroShift 4.13.21)*

### 5.2.2. Upgrading Microshift

The upgrade process proceeded in two steps. The first upgrade from version 4.13.21 to version 4.14.44 was executed using the following command:

| Operation | Command |
|---|---|
| Upgrade to MicroShift 4.14.44 | sudo dnf -y install microshift-4.14.44 |
| Time and resources measurement | /usr/bin/time -v sudo dnf -y install microshift-4.13.21 |

*Table 21: Upgrade to MicroShift 4.14.44(Minor Update)*

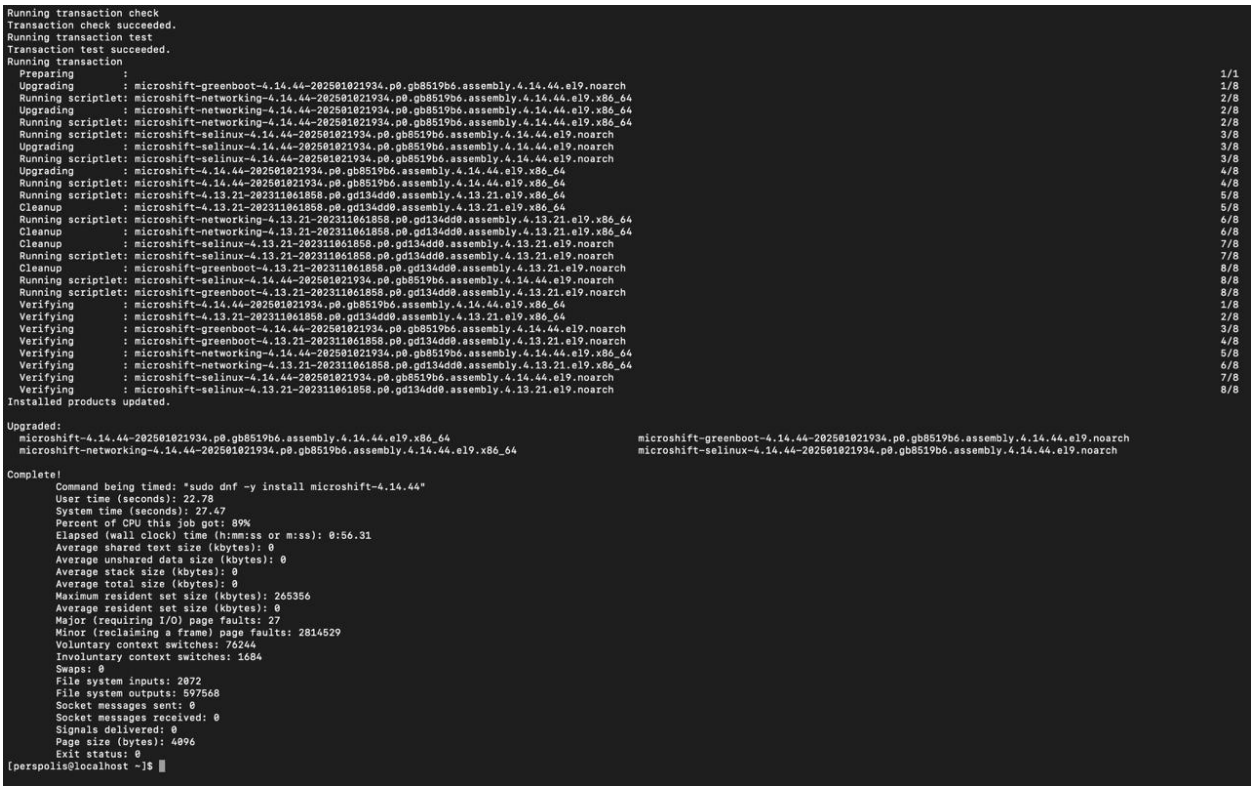



*Figure 27: Upgrade to MicroShift 4.14.44*

This upgrade installed microshift-4.14.44 and microshift-networking-4.14.44. The recorded resource usage included a user time of 22.78 seconds, a system time of 27.47 seconds, a CPU usage of 89 percent, and a memory consumption of 265 megabytes.

| Measurement | Upgrade to MicroShift 4.14.44 |
|---|---|
| User Time (s) | 22.78 |
| System Time (s) | 27.47 |
| CPU Usage (%) | 89 |
| Memory Consumption (MB) | 265 |

*Table 22: Measurements(Upgrade to MicroShift 4.14.44)*

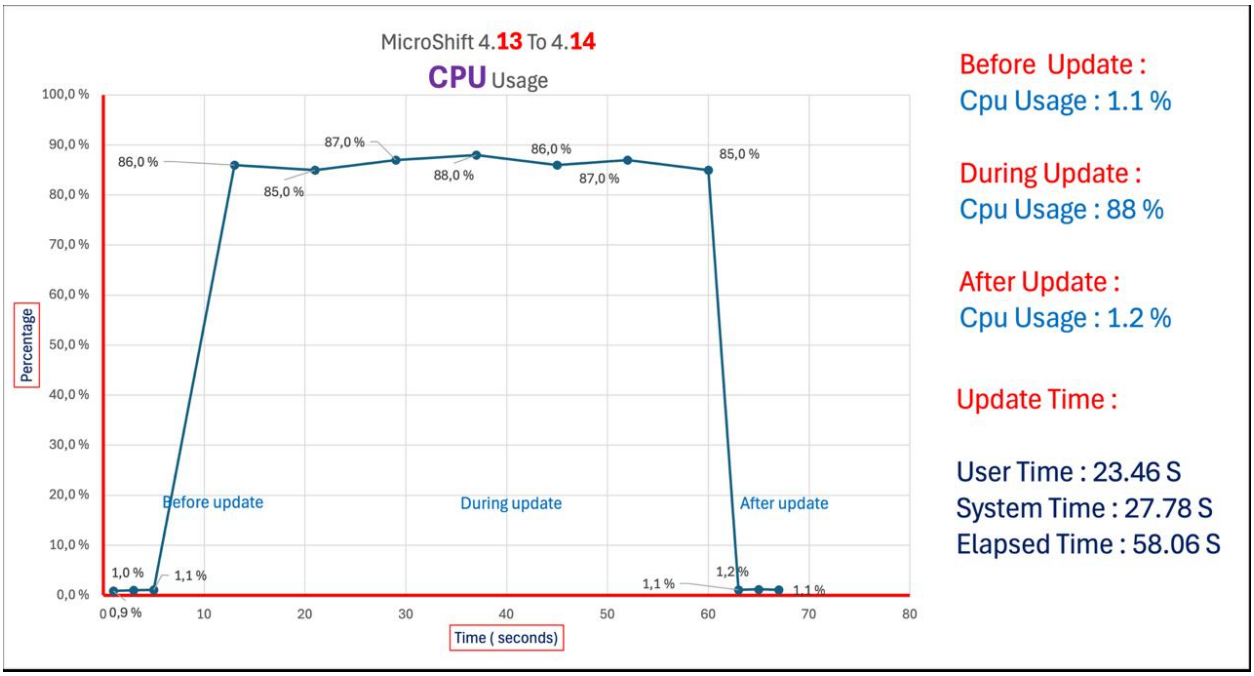



*Figure 28: Upgrade to MicroShift 4.14.44(CPU Usage)*

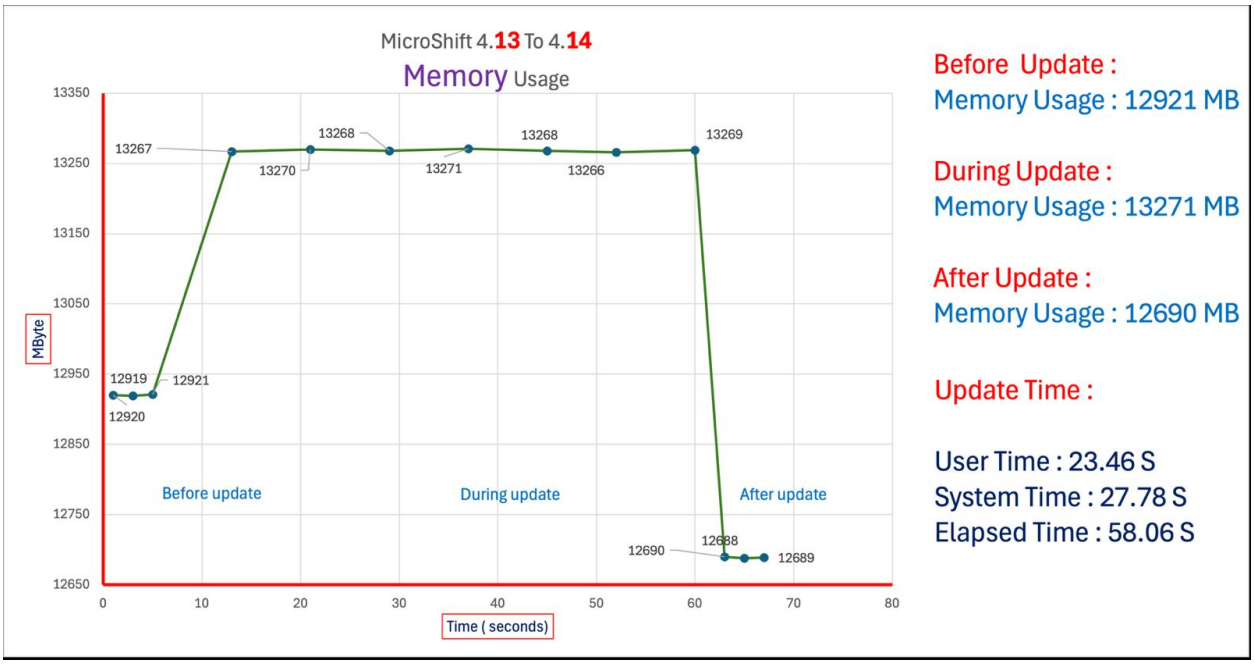



*Figure 29: Upgrade to MicroShift 4.14.44(Memory Usage)*

The second upgrade step moved from version 4.14.44 to version 4.14.45 using the command:

| Operation | Command |
|---|---|
| Upgrade to MicroShift 4.14.45 | sudo dnf -y install microshift-4.14.45 |
| Time and resources measurement | /usr/bin/time -v sudo dnf -y install microshift-4.13.21 |

*Table 23: Upgrade to MicroShift 4.14.45(Patch Update)*

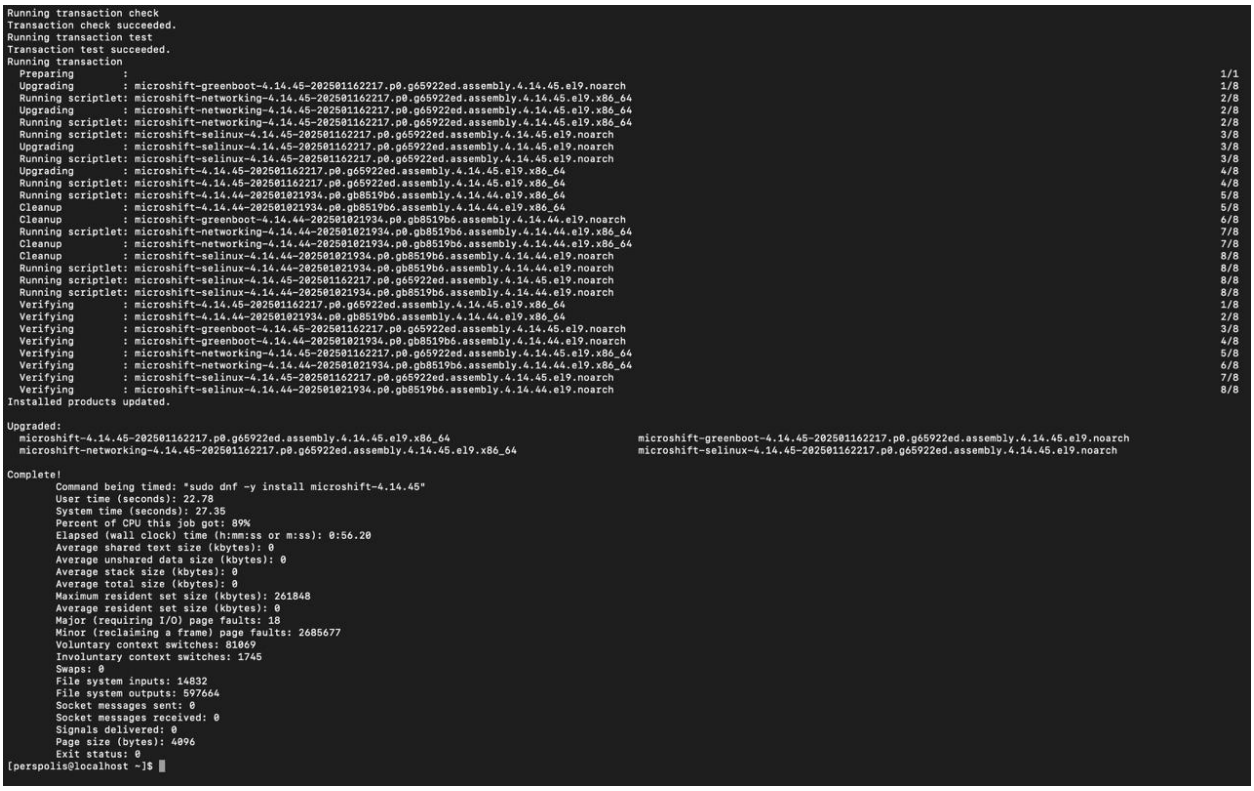



*Figure 30: Upgrade to MicroShift 4.14.45(Patch Update)*

This operation installed microshift-4.14.45 and microshift-networking-4.14.45. The resource usage measurements included a user time of 22.78 seconds, a system time of 27.36 seconds, a CPU usage of 89 percent, and a memory consumption of 261 megabytes.

| Measurement | Upgrade to MicroShift 4.14.45 |
|---|---|
| User Time (s) | 22.78 |
| System Time (s) | 27.36 |
| CPU Usage (%) | 89 |
| Memory Consumption (MB) | 261 |

*Table 24: Measurements(Upgrade to MicroShift 4.14.45)*

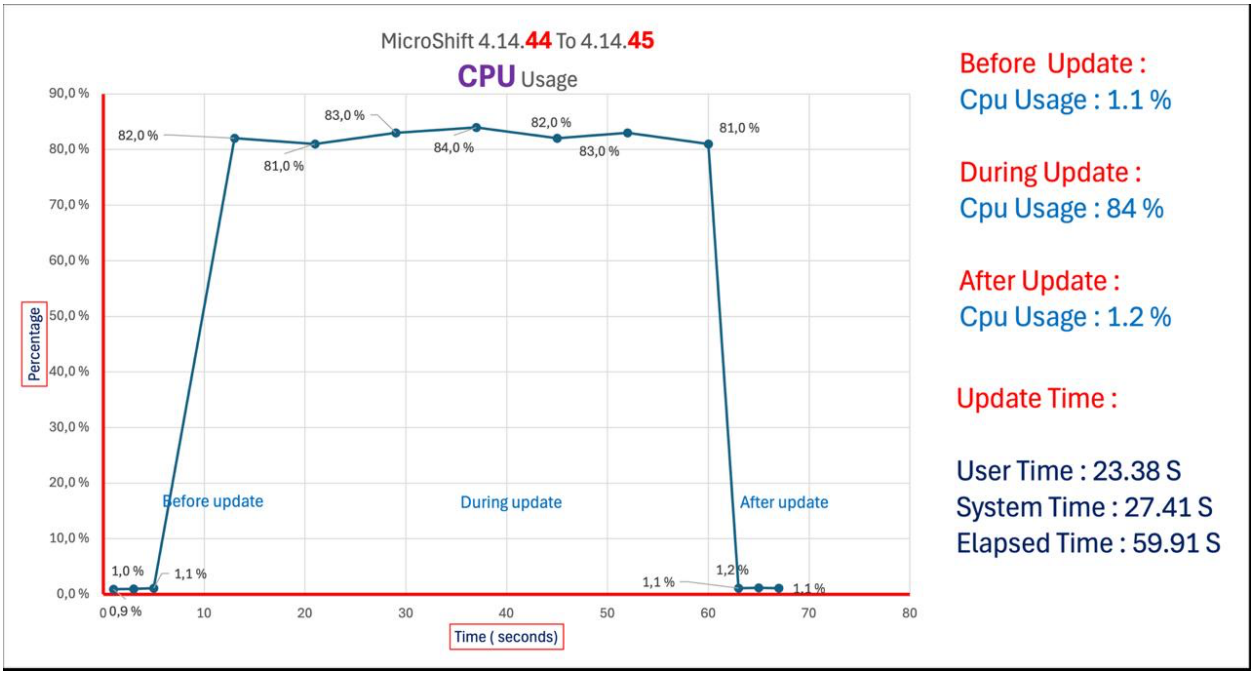



*Figure 31: Upgrade to MicroShift 4.14.45(CPU Usage)*

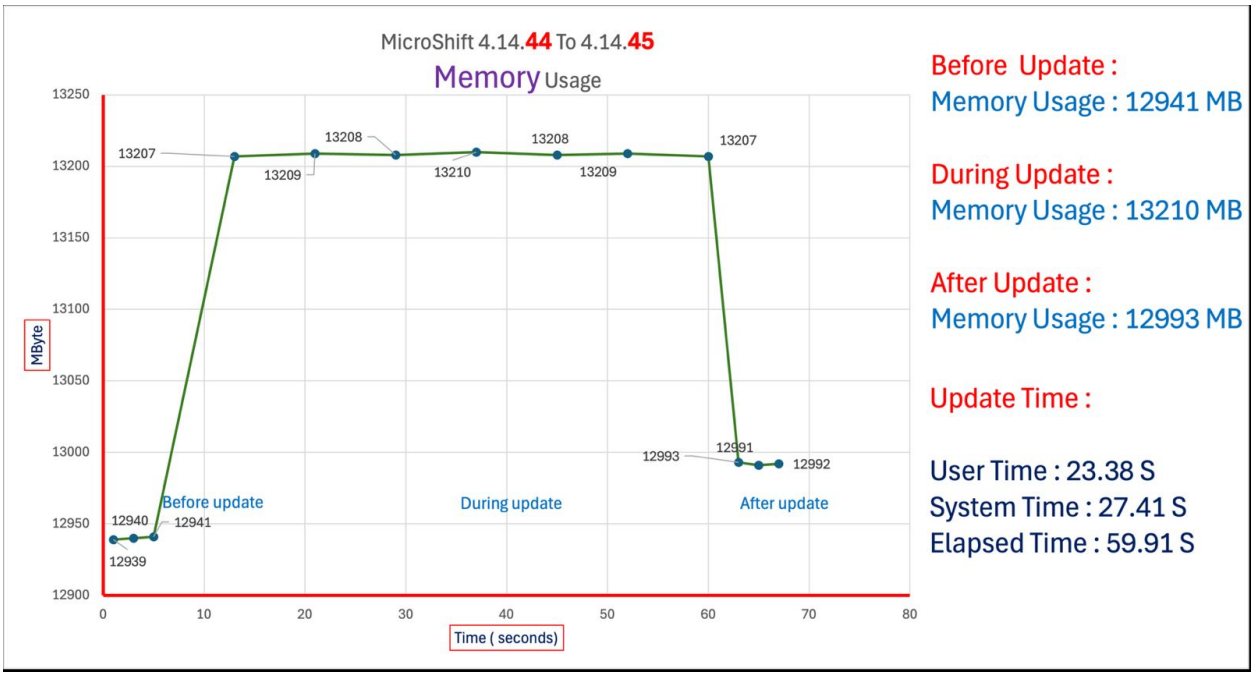



*Figure 32: Upgrade to MicroShift 4.14.45(Memory Usage)*

## 5.3. Pod Continuity During MicroShift and RHEL Updates

When updating both MicroShift and RHEL on the server, the running pods were not interrupted. This behavior can be explained by analyzing the update processes separately for MicroShift and RHEL, considering that CRI-O is utilized as the container runtime.

However, in certain cases, pods may be restarted during a MicroShift update due to specific changes in the deployment configuration or the behavior of the applications running inside the containers. This section also addresses scenarios where interruptions may occur.

### 5.3.1 When Pods Restart During a MicroShift Update

Although pods typically remain running during MicroShift updates, certain conditions may cause them to restart. One such condition is when a MicroShift update modifies deployment specifications, such as pod definitions, security policies, or environment variables. In these instances, a pod restart may be triggered to apply the new configuration. Additionally, if

modifications are made to critical components like CRI-O, Kubelet, or the networking stack during the update, running pods may experience temporary connectivity loss or be restarted.

Some applications inside containers may also terminate unexpectedly if the control plane briefly disconnects, depending on how they handle lost API server communication. While most containers will continue running, those that have strict dependencies on Kubernetes API calls may exit and require a restart. Another scenario that may lead to pod restarts is if an update affects storage-related configurations or Container Storage Interface (CSI) drivers. In such cases, pods that rely on persistent volumes may need to restart in order to re-establish storage connections. Finally, if a MicroShift update requires a system reset or clears certain configurations, pods may be re-deployed, leading to a restart.

### 5.3.2. MicroShift Update: No Pod Interruption

MicroShift is designed as a lightweight platform optimized for edge computing, ensuring minimal disruption during updates. Several factors contribute to the continuity of pods during MicroShift updates. First, MicroShift updates are performed incrementally, allowing core services to restart individually without affecting running pods. Unlike full cluster restarts, MicroShift updates are designed to replace components seamlessly. The MicroShift control plane components, including the API server, scheduler, and controller manager, operate independently from the workloads running inside the pods. During an update, only the control plane services are restarted, leaving the pods managed by CRI-O unaffected.

Additionally, since CRI-O directly manages the containers, it continues running existing pods even if MicroShift's control plane undergoes a temporary restart. The runtime does not terminate containers unless explicitly instructed. The Kubelet, which maintains pod states, ensures that running workloads remain active. Even if the control plane components restart, the Kubelet will re-establish communication but will not stop the running pods.

### 5.3.3. RHEL Update: No Pod Interruption

The RHEL operating system update also did not interrupt running pods due to several reasons. If live kernel patching (kpatch) is utilized, certain kernel updates can be applied without requiring a reboot, which prevents disruptions to running processes, including MicroShift and CRI-O.

Furthermore, the CRI-O container runtime operates as a user-space application, independent of the underlying OS updates. As long as CRI-O remains running, all containers within pods continue functioning normally.

Standard RHEL updates do not force immediate reboots or restart critical services. Therefore, MicroShift and CRI-O remain operational unless explicitly restarted. Since MicroShift, CRI-O, and other critical services are managed as systemd services, updates to other OS components, such as libraries and utilities, do not directly impact running containers. Additionally, if a reboot is not performed immediately after the RHEL update, all services, including MicroShift and CRI-O, remain in their pre-update state, ensuring uninterrupted pod operation.

### 5.3.4. Observations and Insights

#### 5.3..4.1. MicroShift Update Behavior

In most cases, MicroShift updates do not lead to pod restarts. This is due to the independent execution of the control plane and the stability of the CRI-O runtime, which ensures that updates can be performed without interrupting running pods. The Kubelet also plays a key role in maintaining the continuity of running workloads by ensuring they remain active during updates.

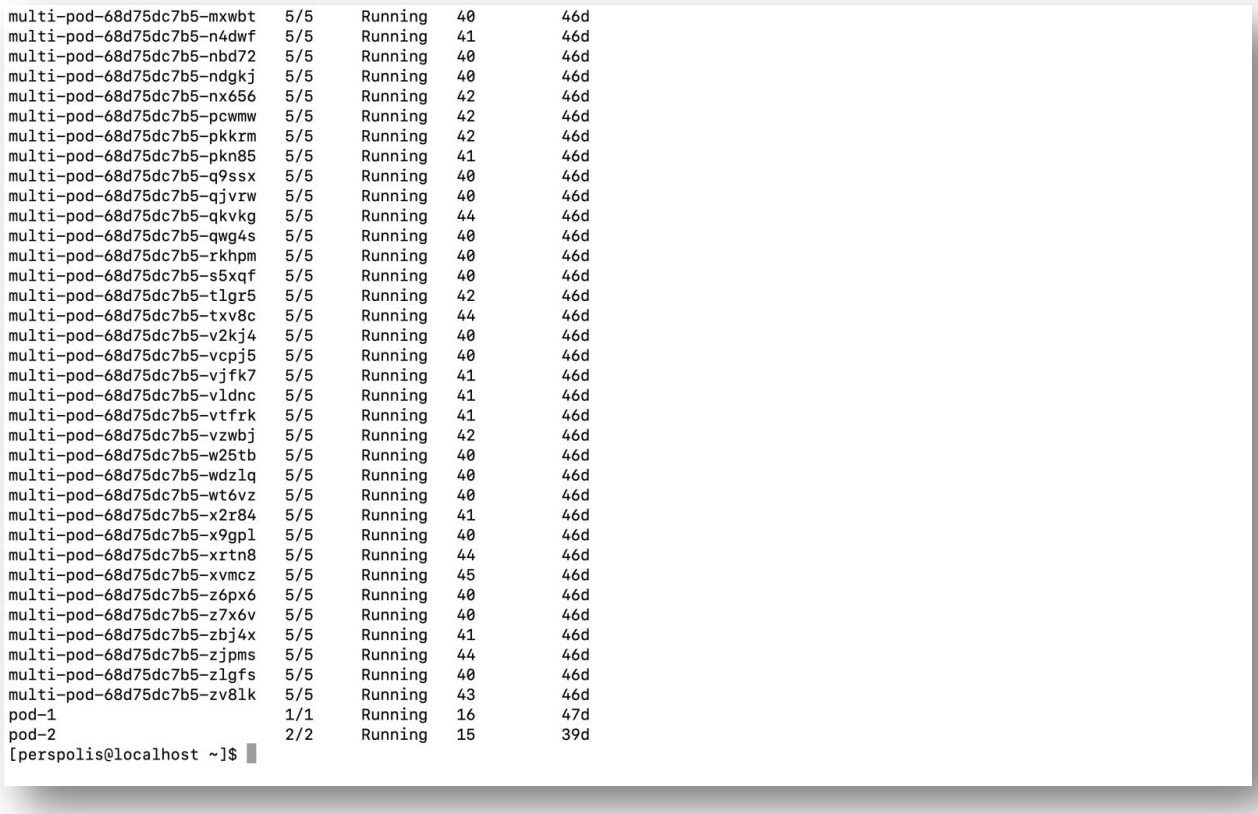
However, there are scenarios where pods may restart. If deployment configurations are modified, critical components are updated, or if an application inside a container does not handle API disconnections properly, pods may need to be restarted. Additionally, persistent storage dependencies and MicroShift reset actions could trigger restarts.

#### 5.3.4.2. RHEL Update Behavior

In the case of RHEL updates, pods remained uninterrupted as long as no reboot was performed. The stability of CRI-O and its independence from the underlying OS updates allowed containers to continue running without disruption. As long as a reboot is avoided, CRI-O remains operational and pods continue functioning smoothly.

#### 5.3.4.3. Implications for Administrators

By understanding these behaviors, administrators can strategically plan updates to minimize potential disruptions. This ensures that system availability is maintained and that updates can be applied with minimal impact on running workloads.



```
multi-pod-68d75dc7b5-mxwbt   5/5     Running   40          46d
multi-pod-68d75dc7b5-n4dwf   5/5     Running   41          46d
multi-pod-68d75dc7b5-nbd72   5/5     Running   40          46d
multi-pod-68d75dc7b5-ndgkj   5/5     Running   40          46d
multi-pod-68d75dc7b5-nx656   5/5     Running   42          46d
multi-pod-68d75dc7b5-pcwmw   5/5     Running   42          46d
multi-pod-68d75dc7b5-pkkrm   5/5     Running   42          46d
multi-pod-68d75dc7b5-pkn85   5/5     Running   41          46d
multi-pod-68d75dc7b5-q9ssx   5/5     Running   40          46d
multi-pod-68d75dc7b5-qjvrw   5/5     Running   40          46d
multi-pod-68d75dc7b5-qkvkg   5/5     Running   44          46d
multi-pod-68d75dc7b5-qwg4s   5/5     Running   40          46d
multi-pod-68d75dc7b5-rkhpm   5/5     Running   40          46d
multi-pod-68d75dc7b5-s5xqf   5/5     Running   40          46d
multi-pod-68d75dc7b5-tlgr5   5/5     Running   42          46d
multi-pod-68d75dc7b5-txv8c   5/5     Running   44          46d
multi-pod-68d75dc7b5-v2kj4   5/5     Running   40          46d
multi-pod-68d75dc7b5-vcpj5   5/5     Running   40          46d
multi-pod-68d75dc7b5-vjfk7   5/5     Running   41          46d
multi-pod-68d75dc7b5-vldnc   5/5     Running   41          46d
multi-pod-68d75dc7b5-vtfrk   5/5     Running   41          46d
multi-pod-68d75dc7b5-vzwbj   5/5     Running   42          46d
multi-pod-68d75dc7b5-w25tb   5/5     Running   40          46d
multi-pod-68d75dc7b5-wdzlq   5/5     Running   40          46d
multi-pod-68d75dc7b5-wt6vz   5/5     Running   40          46d
multi-pod-68d75dc7b5-x2r84   5/5     Running   41          46d
multi-pod-68d75dc7b5-x9gpl   5/5     Running   40          46d
multi-pod-68d75dc7b5-xrtn8   5/5     Running   44          46d
multi-pod-68d75dc7b5-xvmcz   5/5     Running   45          46d
multi-pod-68d75dc7b5-z6px6   5/5     Running   40          46d
multi-pod-68d75dc7b5-z7x6v   5/5     Running   40          46d
multi-pod-68d75dc7b5-zbj4x   5/5     Running   41          46d
multi-pod-68d75dc7b5-zjpms   5/5     Running   44          46d
multi-pod-68d75dc7b5-zlgfs   5/5     Running   40          46d
multi-pod-68d75dc7b5-zv8lk   5/5     Running   43          46d
pod-1                        1/1     Running   16          47d
pod-2                        2/2     Running   15          39d
[perspolis@localhost ~]$
```

*Figure 33: RHEL Update Behavior*

## 5.4. Pod Restart Behavior After Server Reboot in MicroShift

### 5.4.1. Overview

When a server running MicroShift is rebooted, all previously running pods are restarted as part of the system and container orchestration recovery process. This ensures that workloads resume operation automatically after system downtime. This document explains what happens during the reboot sequence at both the system and MicroShift levels, including startup timings and pod recovery.

### 5.4.2. System and MicroShift Recovery Process

#### 5.4.2.1. System Boot Sequence

When the server is powered on, the following sequence occurs. The system firmware (BIOS/UEFI) initializes hardware components, and the bootloader (GRUB) loads the Linux kernel into memory. After the kernel is initialized, it starts core system functions such as CPU, memory, and I/O subsystems. Following this, Systemd, the service manager, orchestrates the startup of system services. Essential system services, such as networking, logging, and SSH, are started, and persistent storage devices are mounted. At this stage, MicroShift and its dependencies are queued for startup. The total reboot duration, from power on to full operational status, was measured at 6 minutes and 26 seconds (as determined by systemd-analyze).

#### 5.4.2.2. MicroShift Initialization and CRI-O Startup

MicroShift, being a lightweight Kubernetes distribution, relies on several services to restore the cluster state after a reboot. The MicroShift service is started by Systemd, which reads the stored state, including previous deployments and pod configurations. Following this, CRI-O, the container runtime interface for MicroShift, is started to manage containers. CRI-O loads container images from local storage, if they are cached, or pulls missing images from the registry. CRI-O was activated 3 minutes and 26 seconds after the reboot.

#### 5.4.2.3. Explanation of CRI-O Activation Delay During Server Reboot

During the server reboot, the system reached full operational status in 6 minutes and 26 seconds. However, CRI-O was activated 3 minutes and 26 seconds after the reboot. This delay in the activation of CRI-O was caused by the specific order of dependencies and services it requires to be initialized during the boot process.

CRI-O depends on a minimal set of critical system services for proper initialization, and these services must be initialized first. While CRI-O starts early in the boot process, it cannot proceed until these essential services are available. Networking is one of the services CRI-O depends on, as it is required for communication between containers. Networking services are typically

initialized after basic system and storage services, which explains the delay. Additionally, CRI-O needs access to storage for pulling container images and storing data, meaning it must wait until storage services, such as the root filesystem and container image stores, are fully mounted and accessible. Furthermore, CRI-O requires certain kernel features, such as Namespaces, Cgroups, and SELinux/AppArmor, which are loaded early in the boot process. Once these services and features are ready, CRI-O can begin its initialization and function as expected.

| Component | Time Taken | Description |
|---|---|---|
| System Reboot | 6 minutes 26 seconds | Time taken for the server to fully boot up and reach an operational state. |
| CRI-O Activation | 3 minutes 26 seconds | Time taken after reboot for CRI-O to initialize, due to dependencies on networking, storage, and kernel features. |

*Table 25: Explanation of CRI-O Activation Delay During Server Reboot*

#### 5.4.2.4. Pod Recovery Process

Once MicroShift and CRI-O are running, the pod recovery process begins. MicroShift retrieves the last known state of running workloads and schedules previously running pods for restart. The first pod to be scheduled and successfully transition to a "Running" state is deployed. This typically happens for system-critical or low-resource pods. The first pod transitioned to a running state at 3 minutes and 27 seconds after the reboot. Subsequently, the remaining pods are initialized based on

resource availability. If images are already available in CRI-O's local cache, startup is faster. If images need to be pulled from a registry, additional network time is required. The final pod reaches the "Running" state, marking the completion of full cluster recovery. This occurred at 7 minutes and 49 seconds after the reboot.

### 5.4.3. Post-Reboot Verification Commands

After a system reboot, administrators should verify the status of MicroShift and its associated components to ensure smooth operation. The following checks help confirm that the MicroShift service, CRI-O runtime, and application pods are running correctly.

| Command | Description |
|---|---|
| systemctl status microshift | Checks whether the MicroShift service is active and running. If the status shows "active (running)," MicroShift is functioning correctly. If it is inactive or failed, further troubleshooting is required. |
| systemctl status crio | Verifies the status of CRI-O, the container runtime used by MicroShift. Ensuring that CRI-O is running is crucial because it is responsible for managing container lifecycles. |
| oc get pods -A | Lists all running pods across all namespaces. This command helps confirm whether deployed workloads have restarted properly after the reboot. |
| oc get events --sort-by=.metadata.creationTimestamp | Displays pod startup events in chronological order. This is useful for tracking the startup sequence and identifying any errors or delays in pod initialization. |

*Table 26: Post-Reboot Verification Commands*

These commands provide a comprehensive post-reboot verification process, ensuring that the MicroShift environment is fully operational.

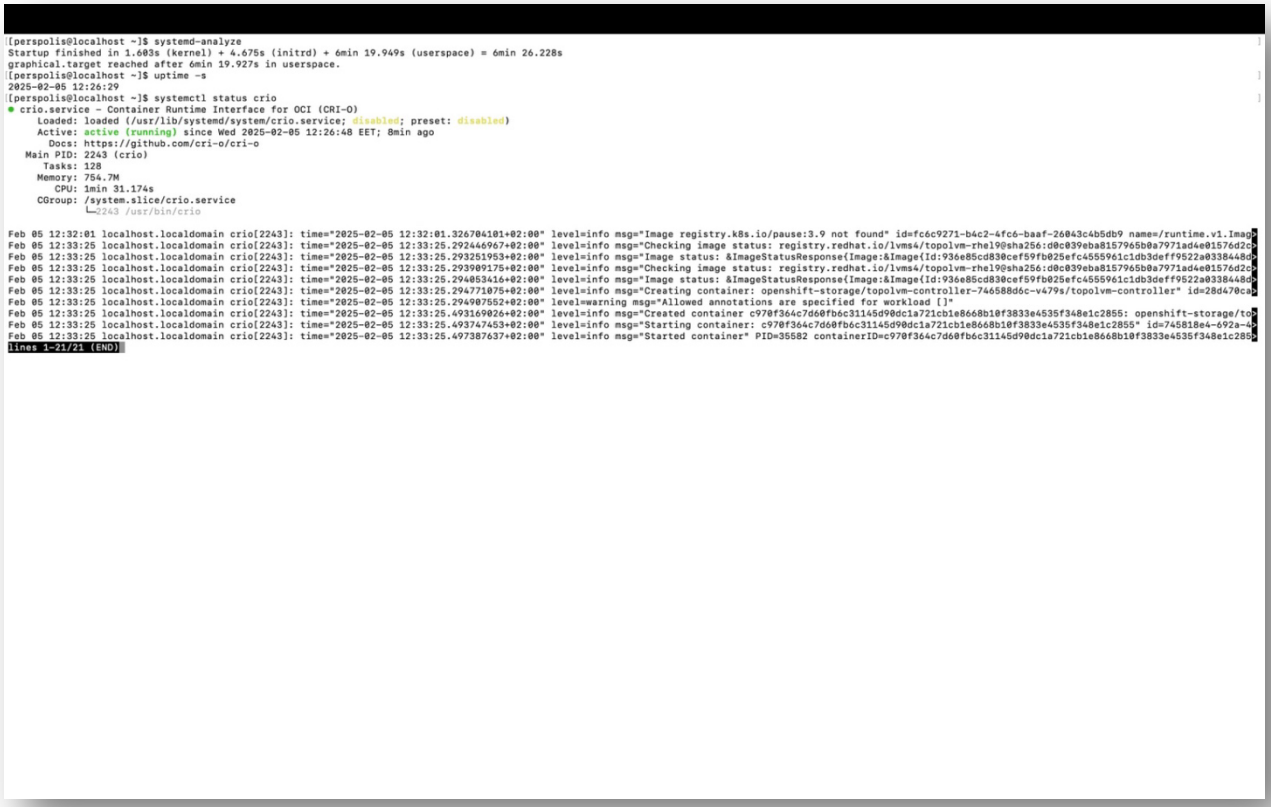
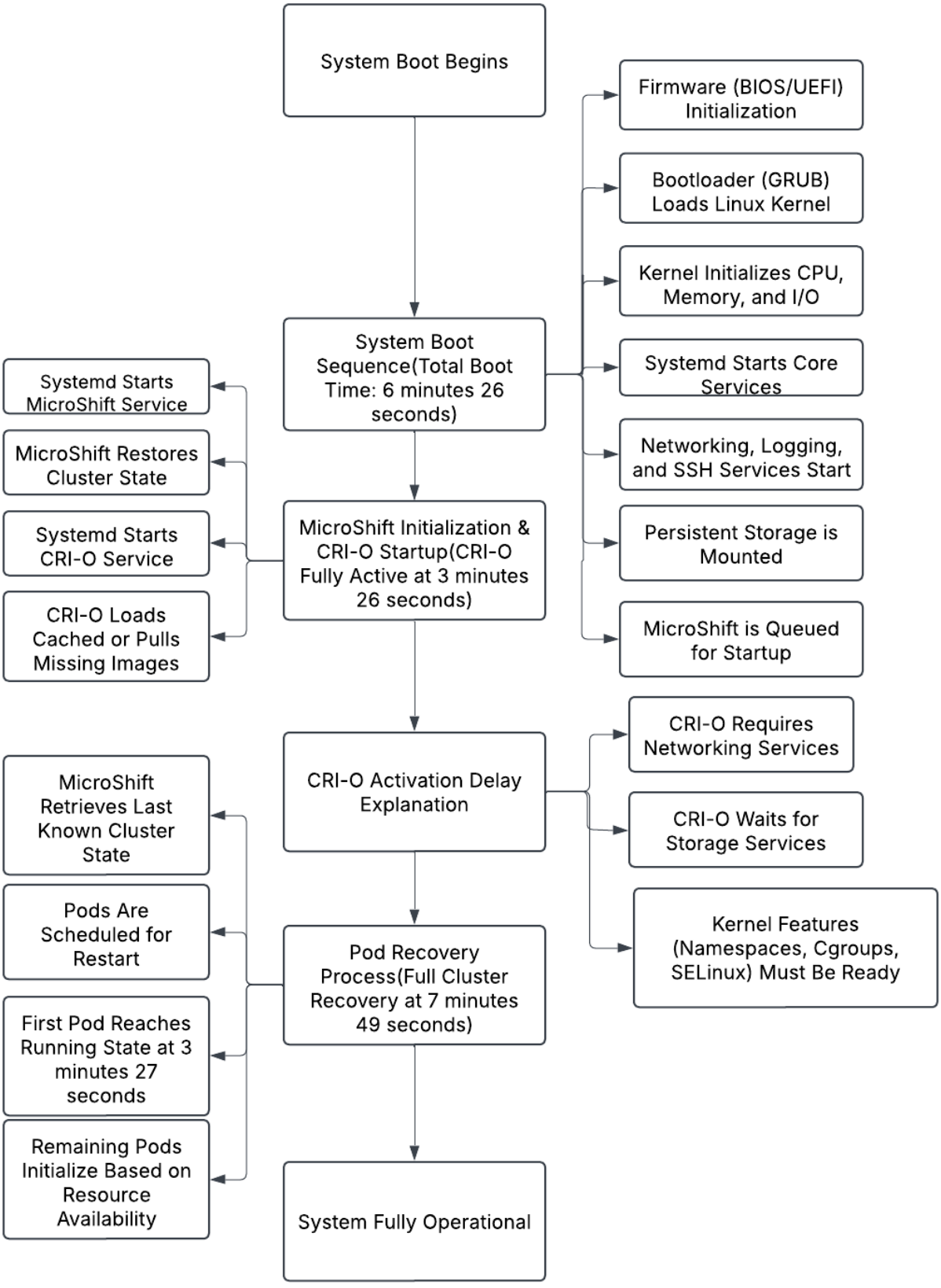



*Figure 34: Pod Recovery Process*

### 5.4.4. Findings

When the server reboots, MicroShift restores the last known state of the cluster and reinitializes pods. CRI-O starts at 3 minutes and 26 seconds, enabling container management. The first pod is running by 3 minutes and 27 seconds, indicating early service recovery. Full pod recovery is complete at 7 minutes and 49 seconds, marking the cluster as fully operational. System administrators can monitor the recovery process using oc get pods, oc get events, and systemctl

status microshift. This structured recovery ensures that MicroShift-based deployments resume operations automatically after a server reboot, minimizing downtime and maintaining service availability.


System Boot Begins
Firmware (BIOS/UEFI) Initialization
Bootloader (GRUB) Loads Linux Kernel
Kernel Initializes CPU, Memory, and I/O
System Boot Sequence(Total Boot Time: 6 minutes 26 seconds)
Systemd Starts Core Services
Networking, Logging, and SSH Services Start
Persistent Storage is Mounted
MicroShift is Queued for Startup
Systemd Starts MicroShift Service
MicroShift Restores Cluster State
Systemd Starts CRI-O Service
CRI-O Loads Cached or Pulls Missing Images
MicroShift Initialization & CRI-O Startup(CRI-O Fully Active at 3 minutes 26 seconds)
CRI-O Activation Delay Explanation
CRI-O Requires Networking Services
CRI-O Waits for Storage Services
Kernel Features (Namespaces, Cgroups, SELinux) Must Be Ready
MicroShift Retrieves Last Known Cluster State
Pods Are Scheduled for Restart
First Pod Reaches Running State at 3 minutes 27 seconds
Remaining Pods Initialize Based on Resource Availability
Pod Recovery Process(Full Cluster Recovery at 7 minutes 49 seconds)
System Fully Operational

# 6. Key Findings

## 6.1. Differences Between OpenShift and MicroShift

A comparative analysis was conducted to evaluate the differences between MicroShift and OpenShift Single Node across six key areas: pod behavior during updates, update duration, installation size, APIs, operator availability, and versioning. This analysis aimed to assess how both platforms perform in terms of efficiency, reliability, and overall performance, specifically in scenarios where a lightweight, single-node deployment is required, such as edge computing environments. The focus was on understanding the trade-offs between OpenShift's full-featured capabilities and MicroShift's streamlined approach tailored for minimal resource usage.

### 6.1.1. Pod Behavior

A notable difference was observed between MicroShift and OpenShift Single Node in terms of pod behavior during updates. Testing was conducted with 100 pods and a total of 500 containers, ensuring identical conditions for both platforms. It was found that a seamless update experience was provided by MicroShift, with all pods remaining operational throughout the update process. No interruptions, downtime, or disruptions were experienced by the pods, highlighting MicroShift's capability to maintain high availability during updates.

In contrast, a temporary interruption was experienced by OpenShift Single Node during the update process. Specifically, an average downtime of approximately two minutes occurred, during which the pods were not operational before normal functioning was resumed. This brief downtime highlights a key difference in how updates are handled between the two platforms.

The uninterrupted pod operation in MicroShift makes it particularly suitable for environments where high availability and minimal service disruption are essential, such as edge computing or single-node deployments.

### 6.1.2. Time (Deployment and Operational Overhead)

The time required for updates was identified as another significant differentiator between MicroShift and OpenShift Single Node. Measurements revealed a stark contrast in update durations.

For MicroShift, minor and patch updates were typically completed within approximately one minute. This rapid update process is attributed to MicroShift's lightweight and streamlined architecture, enabling efficient software upgrades with minimal overhead.

In contrast, OpenShift Single Node required significantly more time for updates. Findings indicated that a typical update took approximately 74 minutes to complete. This prolonged update duration reflects OpenShift's more complex architecture and resource-intensive design, which necessitate a more extensive update process.

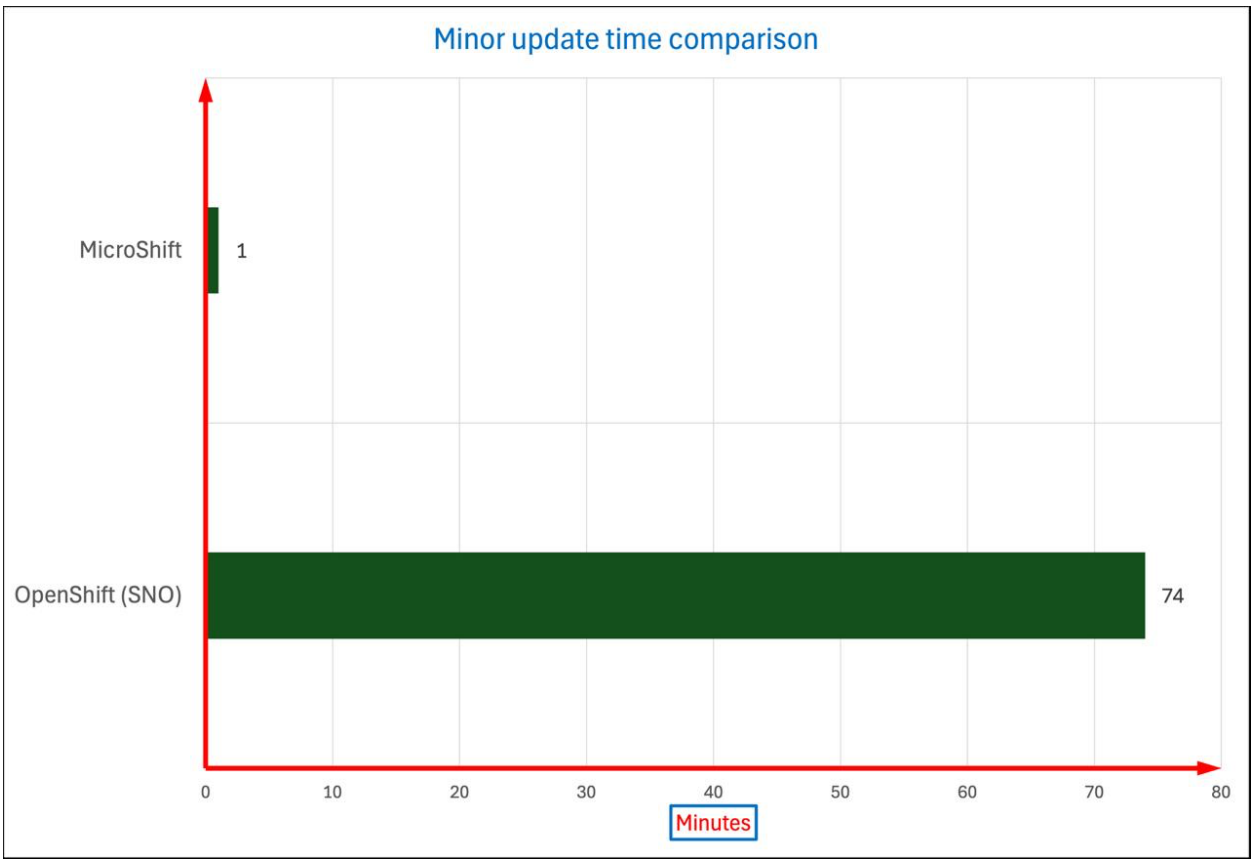



*Figure 35: Minor Update Time Comparison*

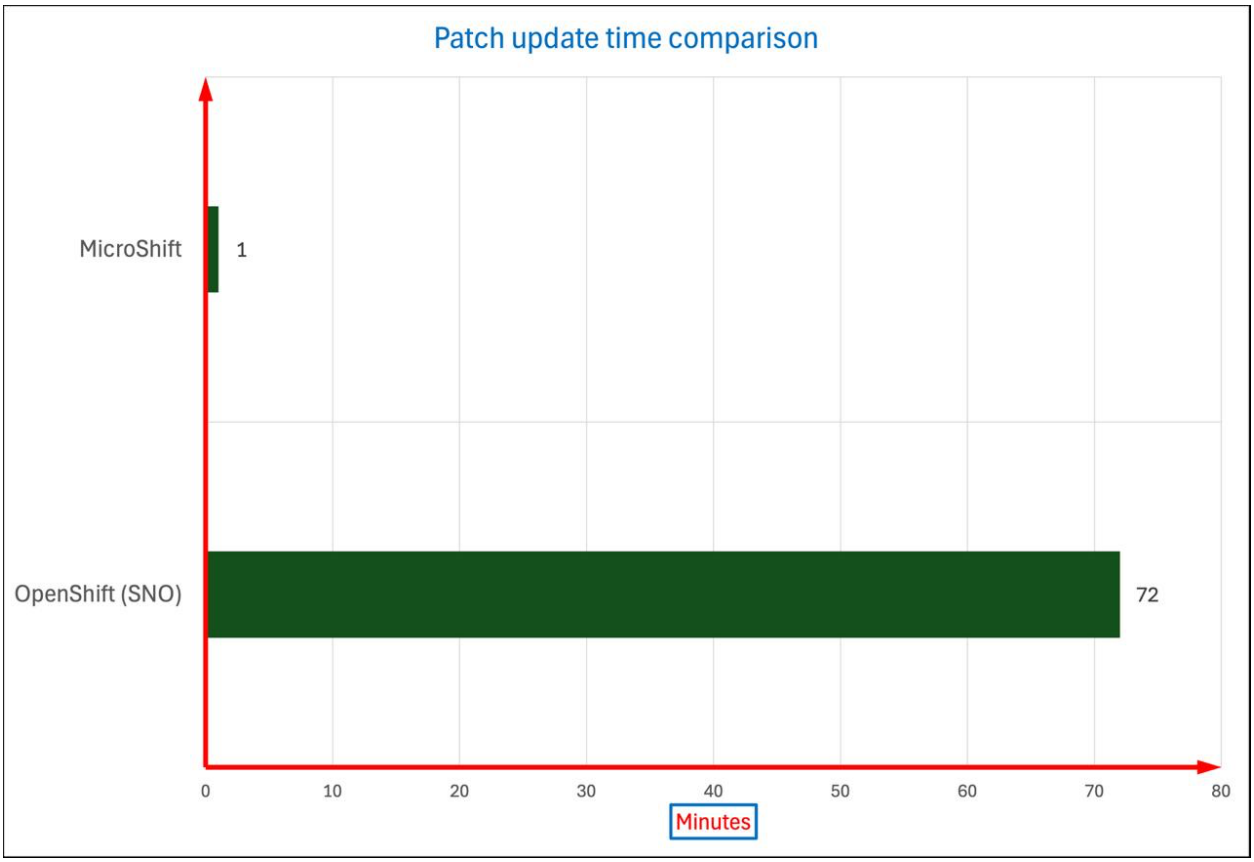



*Figure 36: Patch Update Time Comparison*

### 6.1.3. Installing Size

When comparing the installation sizes of MicroShift and Single Node OpenShift (SNO), their distinct architectures and deployment methods must be considered. MicroShift, a lightweight Kubernetes distribution optimized for edge computing, is installed via an RPM package, requiring approximately 2 GB of root storage. In contrast, SNO, which consolidates a full OpenShift deployment into a single node, is installed using a discovery ISO. The minimal image size for SNO has been reported as approximately 1.5 GB; however, this figure only accounts for the installation media and does not reflect the total disk space required for a functional deployment. A minimum of 500 GB of storage is recommended by Red Hat to accommodate pre-cached images and ensure stability. Although the ISO size appears like MicroShift's RPM, a significantly larger operational footprint is required due to SNO's more complex architecture. Therefore, storage requirements must be carefully evaluated to ensure adequate resources for stable operation.

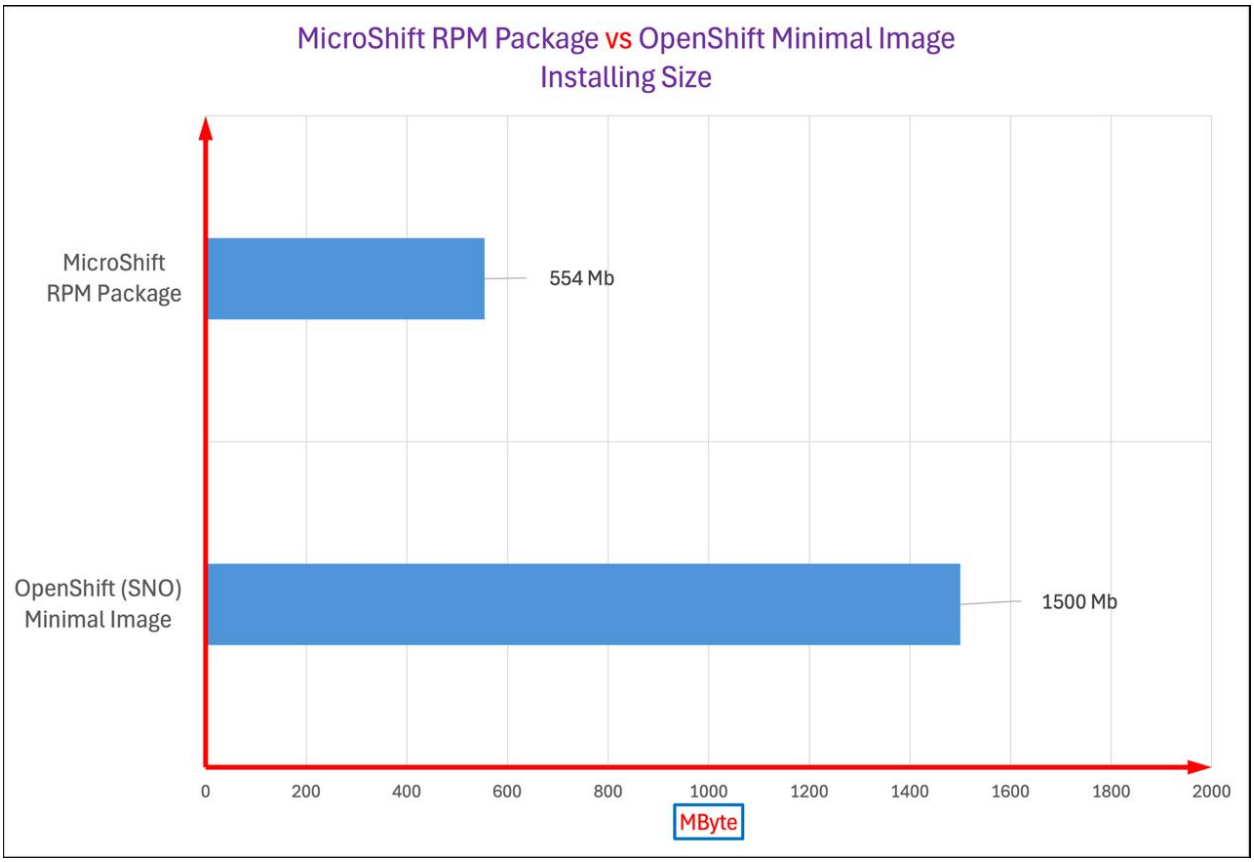



*Figure 37: Installing Size*

### 6.1.4. APIs

APIs play a crucial role in both OpenShift and MicroShift, providing the interface for users, applications, and system components to interact within the Kubernetes ecosystem. While both platforms utilize the core Kubernetes APIs, their scope and functionality differ due to the contrasting goals of OpenShift and MicroShift.

#### 6.1.4.1. The Role of APIs in Kubernetes-Based Platforms

APIs exist in Kubernetes-based platforms like OpenShift and MicroShift to facilitate interaction between users, applications, and system components. They enable the deployment and management of workloads, such as pods, services, and deployments. Additionally, APIs allow applications to interact with storage and networking resources, ensuring seamless communication

within the platform. They also play a crucial role in enforcing security policies, handling authentication, and managing authorization to control access to system resources. Furthermore, APIs automate system management tasks by supporting Operators, which streamline the deployment, scaling, and maintenance of applications.

#### 6.1.4.2. OpenShift APIs

In OpenShift, a comprehensive suite of APIs is available to manage not just the fundamental Kubernetes resources, but also advanced features tailored for enterprise-scale applications. These APIs cover a wide range of functionalities including:

- Build APIs: OpenShift includes the Build API, which facilitates the automation of application builds using source code, image streams, and pipelines. This is an essential feature for continuous integration (CI) and continuous deployment (CD).
- Image Stream APIs: These allow OpenShift to manage container images in a more efficient way than standard Kubernetes, supporting advanced use cases like image promotion and integration into CI/CD workflows.
- CI/CD APIs: OpenShift offers comprehensive tools for managing CI/CD pipelines, automating the entire software development lifecycle. This includes operators like Tekton, Jenkins, and OpenShift Pipelines, which integrate directly with OpenShift to provide full DevOps capabilities.
- Security and Monitoring APIs: OpenShift includes specialized APIs for managing enterprise security, monitoring tools, and service meshes. This includes integrations for advanced security policies, as well as monitoring APIs for tools like Prometheus and Grafana.

| API Name | Use Case |
|---|---|
| Build API | Automates building container images from source code. |
| Image API | Manages container images, including build and distribution. |
| Cluster Version API | Handles OpenShift cluster updates and upgrades. |
| Cluster-wide Route API | Enables advanced ingress and route management for exposing services. |
| Machine API | Manages worker nodes in OpenShift clusters (not relevant in single-node MicroShift). |
| Monitoring APIs | OpenShift has built-in monitoring (Prometheus, Grafana) which is not present in MicroShift. |
| Multitenancy APIs | Features like OpenShift's Project API for namespace-based isolation are limited in MicroShift. |

*Table 27: OpenShift APIs*

#### 6.1.4.3. MicroShift APIs

MicroShift does not introduce exclusive APIs that do not exist in OpenShift. Instead, it removes unnecessary APIs to keep a small footprint. All APIs in MicroShift are a subset of OpenShift APIs, meaning there are no APIs in MicroShift that OpenShift lacks.

However, MicroShift adapts existing APIs to suit edge computing:

| Modification | Purpose |
|---|---|
| Trimmed OpenShift APIs | Focuses on single-node workloads and removes cluster-scale management APIs. |
| Modified Networking | Supports lightweight deployments with essential networking capabilities. |

*Table 28: MicroShift APIs*

In a nutshell, in both OpenShift and MicroShift, common APIs include Kubernetes Core APIs, Security APIs such as Role-Based Access Control (RBAC) and Security Context Constraints (SCCs), Operator APIs, and some Networking APIs. However, MicroShift lacks several advanced OpenShift APIs, including those for image builds, cluster upgrades, machine management, monitoring, and multitenancy. OpenShift does not miss any exclusive APIs that exist only in MicroShift; however, MicroShift modifies certain APIs to better suit the constraints and requirements of edge computing environments.

| API Type | OpenShift | MicroShift | Description |
|---|---|---|---|
| Kubernetes Core APIs | Yes | Yes | Standard Kubernetes APIs (Pods, Services, Deployments, ConfigMaps, Secrets, etc.). |
| OpenShift APIs | Yes | Partial | Some OpenShift-specific APIs are supported in MicroShift, but not all. |
| Security APIs | Yes | Yes (Limited) | RBAC and SCCs (Security Context Constraints) exist in both but may be limited in MicroShift. |
| Operator APIs | Yes | Yes (Limited) | Both support Operators, but MicroShift has fewer due to single-node limitations. |
| Networking APIs | Yes | Yes (Limited) | OpenShift has a full networking stack, while MicroShift provides minimal networking. |

*Table 29: Common APIs*

### 6.1.5. Operators

#### 6.1.5.1. Definition of an Operator

An Operator is a method for packaging, deploying, and managing Kubernetes-native applications. Operators use Kubernetes APIs and kubectl/oc commands to manage the lifecycle of complex, stateful applications, ensuring that the application operates as expected. Operators automate tasks such as:

- Application deployment
- Configuration management
- Scaling

- Backup and restore
- Monitoring and health checks
- Updates and rollbacks

Operators extend Kubernetes to manage custom applications beyond basic stateless services, turning day-2 operations into automated, repeatable tasks.

#### 6.1.5.2. Purpose of Operators

Operators help reduce manual intervention by automating operational tasks. Without Operators, users would need to manually handle application updates, scaling, backups, and other critical maintenance activities. Operators ensure that these tasks are automated and performed in a reliable and predictable way, which is particularly useful for complex applications that require state management (e.g., databases, message queues).

They are often built for specific applications to encapsulate knowledge of that application's operational needs. For example, a database operator knows how to manage database upgrades, scaling, and backups, while a storage operator can handle the provisioning and scaling of persistent storage volumes.

#### 6.1.5.3. Types of Operators in MicroShift and OpenShift

Since MicroShift is a lightweight version of OpenShift, it supports only a subset of the Operators available in full OpenShift. Below are the types of Operators in both environments:

##### 6.1.5.3.1. Common Operators in Both MicroShift and OpenShift

- Deployment Operator: The Deployment Operator automates the deployment and management of Kubernetes applications. It ensures that the desired number of pods and deployments are always running and correctly configured as per the specified deployment settings.
- StatefulSet Operator: The StatefulSet Operator is responsible for managing stateful applications, which require stable network identities and persistent storage. This is commonly used for applications such as databases that need to maintain their state across pod restarts.

- DaemonSet Operator: The DaemonSet Operator ensures that a specific set of pods runs on every node in the Kubernetes cluster. This operator is typically used for running system-level services, such as logging or monitoring agents, that need to operate on all nodes.
- ReplicaSet Operator: The ReplicaSet Operator maintains a specific number of pod replicas, ensuring that the desired count of pods is running at all times. If a pod fails, the operator automatically creates a new one to replace it and maintain the required number.
- Prometheus Operator: The Prometheus Operator manages the deployment and lifecycle of Prometheus, as well as related monitoring resources such as metrics and alerts. While it is available in both MicroShift and OpenShift, its functionality in MicroShift is limited due to resource constraints.
- Cert-Manager Operator: The Cert-Manager Operator automates the management of TLS certificates for securing communications between applications. It handles tasks such as certificate creation, renewal, and revocation, although its features in MicroShift may be more limited.

| Operator | Purpose | Availability |
|---|---|---|
| Deployment Operator | Manages the lifecycle of Kubernetes applications (pods, deployments). | Available in both MicroShift and OpenShift. |
| StatefulSet Operator | Handles stateful applications that require persistent storage and stable identities (e.g., databases). | Available in both MicroShift and OpenShift. |
| DaemonSet Operator | Ensures that specific pods run on every node in the cluster. | Available in both MicroShift and OpenShift. |
| ReplicaSet Operator | Ensures a specific number of pod replicas are running at any given time. | Available in both MicroShift and OpenShift. |
| Prometheus Operator | Manages Prometheus and monitoring resources (metrics, alerts). | Available in both MicroShift and OpenShift (limited in MicroShift). |
| Cert-Manager Operator | Manages the lifecycle of TLS certificates for applications. | Available in both MicroShift and OpenShift (limited in MicroShift). |

*Table 30: Common Operators*

#### 6.1.5.3.2. Operators in OpenShift but not in MicroShift

MicroShift, as a resource-constrained, lightweight Kubernetes distribution for edge computing, does not support many of OpenShift's advanced Operators. These Operators are not part of MicroShift because they require additional resources or multi-node configurations that MicroShift does not support.

- OpenShift Container Storage Operator: The OpenShift Container Storage Operator manages storage solutions, such as Ceph, for OpenShift clusters. It enables the dynamic provisioning of persistent volumes, allowing applications to have scalable and reliable storage. MicroShift does not include this operator because it is designed for smaller, single-node environments and does not require such a complex storage solution.
- OpenShift CI/CD Operators (e.g., OpenShift Pipelines Operator): The OpenShift CI/CD Operators automate continuous integration (CI) and continuous delivery (CD) pipelines within OpenShift. This allows for streamlined development workflows and automated deployment processes. MicroShift does not support these operators because it is aimed at edge applications and lacks the advanced CI/CD automation typically found in OpenShift.
- Service Mesh Operator (e.g., OpenShift Service Mesh): The Service Mesh Operator enables microservices networking, including features like traffic management, observability, and security between microservices in OpenShift. MicroShift does not support this operator because it lacks the multi-node setup necessary for service meshes, which are designed for more complex environments like OpenShift.
- Cluster Autoscaler Operator: The Cluster Autoscaler Operator automatically adjusts the size of the cluster by adding or removing nodes based on resource demand. However, since MicroShift is a single-node deployment, it does not require a node-scaling feature and therefore does not include this operator.
- Machine API Operator: The Machine API Operator manages machine resources in OpenShift clusters, provisioning nodes and managing their lifecycle. MicroShift, being a single-node system, does not need machine management and thus does not include this operator.
- Build Operator (e.g., OpenShift BuildConfig Operator): The Build Operator automates the build and deployment of container images directly from source code, integrated with OpenShift's CI/CD pipeline. Since MicroShift does not have the full CI/CD pipeline or build automation capabilities, this operator is not available in MicroShift.

| Operator | Use Case | Not in MicroShift |
|---|---|---|
| OpenShift Container Storage Operator | Manages storage solutions (e.g., Ceph) for OpenShift clusters, enabling dynamic provisioning of persistent volumes. | MicroShift is designed for smaller, single-node environments and does not require complex storage solutions. |
| OpenShift CI/CD Operators (e.g., OpenShift Pipelines Operator) | Automates continuous integration and continuous delivery pipelines within OpenShift. | MicroShift is intended for edge applications and lacks the advanced CI/CD automation found in OpenShift. |
| Service Mesh Operator (e.g., OpenShift Service Mesh) | Provides microservices networking, including traffic management, observability, and security between microservices. | MicroShift lacks the multi-node setup required for service meshes, which are meant for more complex environments. |
| Cluster Autoscaler Operator | Automatically adjusts the cluster size by adding or removing nodes based on resource demand. | MicroShift is a single-node deployment, so scaling nodes is not applicable. |
| Machine API Operator | Manages machine resources in OpenShift clusters, provisioning and managing node lifecycle. | MicroShift runs on a single node and does not require machine management. |
| Build Operator (e.g., OpenShift BuildConfig Operator) | Automates the build and deployment of container images directly from source code in OpenShift. | MicroShift does not have full CI/CD pipelines or build automation that OpenShift provides. |

*Table 31: Operators in OpenShift*

#### 6.1.5.4. Operators Summary

- Common Operators: Both OpenShift and MicroShift support core Kubernetes Operators (e.g., Deployments, StatefulSets, DaemonSets), but the full set of Operators in OpenShift, such as CI/CD, storage, and service meshes, is unavailable in MicroShift due to its focus on lightweight, edge computing.
- Missing in MicroShift: Advanced Operators in OpenShift like the Machine API, OpenShift Pipelines, Service Mesh, and Container Storage are not available in MicroShift because they are designed for large-scale, multi-node environments.
- MicroShift's Approach: MicroShift does not introduce exclusive Operators that do not exist in OpenShift. Instead, it removes unnecessary Operators to maintain a small footprint. All Operators in MicroShift are a subset of OpenShift Operators, meaning there are no Operators in MicroShift that OpenShift lacks.

### 6.1.6. Versioning

#### 6.1.6.1. OpenShift vs. MicroShift Versioning and Release Patterns

The versioning relationship between OpenShift and MicroShift is based on the alignment of core features but differs in scope due to the design goals of MicroShift, which is optimized for edge environments.

##### 6.1.6.1.1. Version Alignment

OpenShift releases (e.g., 4.17, 4.18) follow a regular schedule, introducing new features, security patches, and updates to core components. MicroShift versions are typically aligned with OpenShift versions, although MicroShift excludes or modifies certain features to fit its lightweight, single-node edge computing use case.

For example, OpenShift 4.17 and MicroShift 4.17 share core components like Kubernetes APIs and container runtimes but differ in advanced features. MicroShift does not include features such as machine management, multi-cluster management, and complex networking solutions that are part of OpenShift.

When a new OpenShift version, like 4.18, is released, MicroShift 4.18 would follow shortly after, including core features such as Kubernetes, container runtimes, and basic networking, but without enterprise features like multi-node support or advanced storage.

6.1.6.1.2. Core Similarities

- Kubernetes Features
  Both OpenShift and MicroShift include Kubernetes APIs such as Pods, Deployments, StatefulSets, and Services. Basic container management with CRI-O as the container runtime is available in both environments.
- Networking
  Basic networking features such as Network Policies and service discovery are supported in both, ensuring pod-to-pod communication works similarly. However, OpenShift supports more complex networking features like advanced ingress management and multi-cluster networking, which are absent in MicroShift.
- Security
  Both platforms apply security patches for Kubernetes vulnerabilities and container runtimes. MicroShift, however, may lack advanced security features found in OpenShift, like Advanced Network Policies or PodSecurityPolicies.
- Operators
  While both support basic operators like those for StatefulSets and Deployments, OpenShift provides additional operators for complex services like multi-cluster management, storage, and networking, which MicroShift excludes due to its single-node nature.

6.1.6.1.3. Key Differences

- Enterprise Features
  OpenShift provides enterprise-grade features such as multi-node clusters, advanced storage management, and integrated monitoring (Prometheus, Grafana). MicroShift is designed for edge and lightweight environments, focusing on basic container orchestration and monitoring but lacking advanced features.
- Machine and Node Management
  OpenShift supports machine management and multi-node clusters, with full

operator support for the Machine API. In contrast, MicroShift is built for single-node setups and does not require or support complex machine management.

- User Interface
  OpenShift offers a web-based management console for cluster management, while MicroShift typically uses a CLI-based management approach due to its lightweight nature.
- Storage
  OpenShift integrates enterprise storage solutions such as OpenShift Container Storage (OCS) for persistent volumes and multi-node storage clusters. MicroShift uses local or external lightweight storage options but lacks advanced storage orchestration.

#### 6.1.6.1.4. Patches and Security Updates

- Security Patches
  Both OpenShift and MicroShift receive security patches for shared components like Kubernetes and CRI-O. However, OpenShift's patches are more comprehensive, covering enterprise-specific features not supported by MicroShift.
- Networking Updates
  OpenShift's networking updates include advanced features like OpenShift SDN, NetworkPolicy, and routing. MicroShift, on the other hand, focuses on core Kubernetes networking with limited capabilities for advanced networking.
- Bug Fixes and Features
  OpenShift's updates include bug fixes for a wide range of features, including Operators, Service Mesh, and more. MicroShift updates are focused on maintaining core Kubernetes components and bug fixes related to the features it supports.
- Performance Optimizations
  OpenShift optimizes for large-scale deployments, ensuring high availability and multi-node scaling. MicroShift is optimized for edge deployments with a focus on minimizing memory and CPU usage, making it suitable for smaller environments.

The following table delineates the features and components of OpenShift 4.17 and MicroShift 4.17, facilitating a comparative analysis between the two platforms:

| Feature/Component | OpenShift 4.17 | MicroShift 4.17 |
|---|---|---|
| **Cluster Type** | Multi-node, enterprise-grade | Single-node, edge-optimized |
| **Storage** | Supports complex storage solutions, including OpenShift Container Storage (OCS) | Utilizes local or lightweight storage options |
| **Networking** | Offers advanced Software-Defined Networking (SDN), routing, and service meshes | Provides basic networking with simple ingress configurations |
| **Machine Management** | Comprehensive support for multi-node clusters and machine API | Designed for single-node deployments without machine management capabilities |
| **Security** | Implements advanced Role-Based Access Control (RBAC), Security Policies, and Network Policies | Includes essential security features suitable for edge environments |
| **Operator Support** | Extensive set of enterprise operators, such as those for Storage, Continuous Integration/Continuous Deployment (CI/CD), and Service Mesh | Limited set of operators tailored for lightweight deployments |
| **Console** | Features a full web-based management console | Lacks a full console; management is primarily command-line interface (CLI)-based |
| **Patches and Security Updates** | Provides comprehensive patches covering all features | Focuses patches on core Kubernetes components |

*Table 32: The Features and components of OpenShift 4.17 and MicroShift 4.17*

Note: The information presented is based on the latest available data up to OpenShift 4.17 and MicroShift 4.17.

# 7. Next Steps

## 7.1. Advanced Cluster Management (ACM) Integration

The integration of MicroShift with Advanced Cluster Management (ACM) was attempted to enable remote management capabilities. The ACM team outlined three potential methods for establishing a connection: utilizing a token, configuring a DNS-based connection, or providing a YAML URL. To facilitate the process, the kubeconfig file was shared with the ACM team, firewall rules were adjusted to allow necessary port access, and appropriate permissions were granted. Despite these efforts, the integration was not successful.

Upon further investigation, it was determined that the initial configuration should have been initiated by the ACM group. If this prerequisite had been met, the connection could have been successfully established. Given the unresolved nature of this issue, the next step involves identifying a viable solution to ensure seamless ACM integration with MicroShift. Continued collaboration with the ACM team will be necessary to implement a configuration that meets the requirements for remote cluster management.

## 7.2. API Analysis in OpenShift and MicroShift

A comprehensive analysis of the APIs utilized in both OpenShift and MicroShift is essential to understand their architectural differences and operational implications. This analysis will provide insights into how these APIs interact with the system and how they can be leveraged for optimized performance. Identifying discrepancies in API structures and functionalities will enable more effective management and operational efficiency.

By evaluating API compatibility and performance characteristics, it will be possible to determine whether modifications or adaptations are required to improve interoperability between OpenShift and MicroShift. The objective is to optimize API utilization for seamless integration, improved resource management, and enhanced performance across both platforms.

## 7.3. Development and Implementation of Operators for MicroShift

Operators in Kubernetes-based environments serve as automation tools that streamline cluster management by encapsulating complex operational logic. The focus of this phase will be on assessing whether existing OpenShift Operators can be deployed within MicroShift or if custom Operators need to be developed to address specific requirements.

Given that MicroShift is a lightweight version of OpenShift, certain Operators may not be directly compatible due to differences in resource constraints and system architecture. The investigation will involve analyzing the feasibility of deploying standard OpenShift Operators within MicroShift and, if necessary, designing and implementing custom Operators tailored to the unique characteristics of MicroShift. The outcome of this analysis will determine the best approach to achieving efficient automation and resource management within the MicroShift environment.

## 7.4. Version Analysis of OpenShift and MicroShift

A comparative evaluation of different versions of OpenShift and MicroShift is critical to understanding their feature sets, compatibility, and performance variations. This analysis will facilitate informed decision-making regarding version selection, ensuring that deployments align with both functional and operational requirements.

The primary objectives of this version analysis include identifying performance enhancements, compatibility issues, and key differences between successive releases. By systematically examining various versions, it will be possible to determine the optimal combination of OpenShift and MicroShift versions that balance stability, feature availability, and resource efficiency.

## 7.5. MicroShift in Edge Computing Environments

MicroShift is specifically designed for edge computing environments, offering a lightweight and efficient platform for resource-constrained devices such as Raspberry Pi and IoT gateways. One of its primary advantages is its rapid deployment capability, enabling users to set up and manage applications with minimal overhead.

Additionally, its modular architecture provides flexibility, allowing configurations to be adapted to meet specific requirements. This adaptability is particularly beneficial in environments where power consumption and resource efficiency must be carefully managed. While MicroShift seamlessly integrates with OpenShift and Kubernetes, it has limitations, including the need for additional customization and a reduced set of prebuilt automation features compared to full-scale OpenShift deployments. Addressing these challenges through configuration optimizations and enhanced automation strategies will ensure that MicroShift remains a viable and high-performing solution for edge computing use cases.

| Focus Area | Objective | Challenges | Next Steps |
|---|---|---|---|
| ACM Integration | Enable remote management of MicroShift through ACM. | Initial configuration required ACM involvement. | Establish a stable connection. |
| API Analysis | Evaluate API differences between OpenShift and MicroShift. | Architectural differences impact API interoperability. | Conduct a comparative analysis and optimize API utilization. |
| Operator Implementation | Assess the feasibility of deploying OpenShift Operators in MicroShift. | Potential incompatibility with MicroShift's architecture. | Determine whether custom Operators are necessary and proceed with development. |
| Version Analysis | Compare different versions of OpenShift and MicroShift. | Performance, stability, and feature variations across versions. | Review release notes, benchmarks, and compatibility for optimal selection. |
| MicroShift in Edge Computing | Optimize deployment on IoT and edge devices. | Requires additional customization and lacks full automation support. | Enhance configuration strategies to improve efficiency and resource utilization. |

*Table 33: Next Steps*

# 8. Summary and Conclusions

The purpose of this project was to investigate the deployment, management, and optimization of MicroShift on Red Hat Enterprise Linux (RHEL) 9.5 for edge computing environments. Specifically, the project aimed to provide a detailed guide on how MicroShift can be effectively deployed in such environments, how the performance of containerized applications can be optimized, and how stability can be ensured during system upgrades and reboots. The research focused on understanding how resource-constrained devices, typical of edge computing, can be handled, and how the associated challenges can be managed effectively.

This project was based on the principles of containerization and edge computing. Key technologies like MicroShift, Kubernetes, and RHEL were utilized to explore how these tools could be integrated into an edge computing infrastructure. The research involved the installation and configuration of MicroShift on RHEL 9.5, the deployment of containerized applications, and the analysis of the impact of upgrading and downgrading these systems, with particular emphasis on time, CPU, and memory usage, as well as the stability of pods.

The outcome of the project revealed several important findings. It was demonstrated that MicroShift can be deployed on RHEL 9.5 as an effective solution for edge computing environments, with the ability to handle containerized applications with resource constraints while maintaining performance and stability. Furthermore, it was shown that the upgrade and downgrade processes of both RHEL and MicroShift had a notable impact on system resources, and strategies for minimizing service disruptions during these processes were identified. The stability of pods during upgrades was a key focus, and recommendations for best practices in maintaining continuous service were provided.

The objectives of the project were largely achieved, as it was shown that MicroShift can be deployed, configured, and managed efficiently and scalably for edge computing environments.

Additionally, the challenges of system upgrades were addressed, and the stability of pods was ensured. Without this work, the specifics of managing MicroShift deployments in edge environments and the effects of upgrades on performance and stability would not have been as clearly understood.

Based on the findings, it is recommended that organizations implementing MicroShift on RHEL 9.5 for edge computing ensure that proper strategies are in place for upgrading and downgrading systems. System resources should be regularly monitored during these processes, and techniques to maintain pod stability should be implemented. Best practices for configuring MicroShift in resource-constrained environments should be adopted to maximize performance and minimize downtime.

While this research covered several aspects of MicroShift deployment and management, some areas were left unexplored. For example, further studies could investigate more deeply into the long-term performance impacts of upgrading and downgrading RHEL and MicroShift. Additionally, the research could be extended to a wider variety of edge devices, assessing how different hardware configurations affect the performance of MicroShift.

Future research could focus on expanding the scalability of MicroShift in larger edge computing environments and evaluating its performance in highly distributed systems. Moreover, the integration of AI-driven monitoring tools to predict system failures or optimize resource allocation could also be explored.

This project can be used by organizations considering the deployment of MicroShift in edge computing settings, particularly those using RHEL as their base operating system. The findings and recommendations will guide organizations in optimizing their edge computing infrastructures for better efficiency and minimal downtime.

The reliability and trustworthiness of this study are considered to be high, as the research was conducted based on practical implementation, thorough testing, and analysis of system resources. The methodologies used were well-suited to the research objectives and provided valuable insights into the challenges and solutions for edge computing with MicroShift.

On an important note, valuable insights were gained into edge computing and containerization, particularly in how these technologies can be optimized for real-world applications. The experience of deploying and managing MicroShift on RHEL has been invaluable in understanding system management in resource-constrained environments, which will serve as a strong foundation for future work in this field.

# 9. References

[1] The Linux Foundation, Kubernetes Documentation. Available: https://kubernetes.io/docs/. Referred on 29/10/2024.

[2] Red Hat, OpenShift Documentation. Available: https://docs.openshift.com/. Referred on 25/10/2024.

[3] Red Hat, Podman Documentation. Available: https://podman.io/docs/. Referred on 03/11/2024.

[4] Red Hat, MicroShift Documentation. Available: https://docs.openshift.com/microshift/latest/. Referred on 28/10/2024.

[5] Red Hat, CRI-O Documentation. Available: https://github.com/cri-o/cri-o. Referred on 27/01/2025.

[6] Red Hat, OpenShift CLI Documentation (oc). Available: https://docs.openshift.com/container-platform/latest/cli_reference/index.html. Referred on 23/11/2024.

[7] The Kubernetes Authors, "Extend the Kubernetes API with CustomResourceDefinitions." Kubernetes Documentation. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. Referred on 12/02/2025.

[8] Red Hat, "Operator SDK Documentation." Red Hat OpenShift. Available: https://sdk.operatorframework.io/. Referred on 27/02/2025.

[9] Operator Framework, "What is an Operator?" OperatorHub.io. Available: https://operatorhub.io/. Referred on 27/02/2025.