SUSE Telco Cloud Documentation

SUSE Telco Cloud Documentation

Publication Date: 2026-05-27

https://documentation.suse.com

Contents

SUSE Telco Cloud 3.7 Documentation xix

1 What is SUSE Telco Cloud? xix

2 Design Philosophy xix

3 High Level Architecture xx

Components used in SUSE Telco Cloud xxi • Connectivity xxiv

4 Common Edge Deployment Patterns xxv

Directed network provisioning xxvi • Image-based provisioning xxvi

5 SUSE Telco Cloud Stack Validation xxvii

6 Full Component List xxvii

I CONCEPT & ARCHITECTURE 1

1 SUSE Telco Cloud Architecture 2

2 Components 3

3 Deployment Model 4

3.1 Why This Deployment Model? 4

3.2 Deployment of Management Cluster 4

3.3 Deployment of a Single-Node Downstream Cluster with Telco Profiles 5

3.4 Deployment of a Highly-Available Downstream Cluster 6

II QUICK STARTS 8

4 BMC automated deployments with Metal^3 9

4.1 Why use this method 9

4.2 High-level architecture 10

4.3 Prerequisites 11

4.4 Deployment 11
Setup Management Cluster 11 • Installing Metal^3 dependencies 12 • Installing cluster API provider dependencies 14 • Prepare downstream cluster image 14 • Adding BareMetalHost inventory 18 • Creating downstream clusters 22 • Control plane deployment 22 • Worker/Compute deployment 27 • Cluster deprovisioning 30

4.5 Known issues 31

4.6 Planned changes 31

4.7 Additional resources 31
Single-node configuration 31 • Disabling TLS for virtualmedia ISO attachment 32 • Storage configuration 32

5 Standalone clusters with Edge Image Builder 33

5.1 Prerequisites 33
Getting the EIB Image 34

5.2 Creating the image configuration directory 34

5.3 Creating the image definition file 35
Configuring Operating System (OS) 35 • Configuring OS Users 37 • Configuring OS time 38 • Adding certificates 38 • Adding Operating System Files 39 • Configuring RPM packages 39 • Configuring Kubernetes cluster and user workloads 41 • Configuring the network 43

5.4 Building the image 45

5.5 Debugging the image build process 48

5.6 Testing your newly built image 49

III COMPONENTS 50

6 Rancher 51

6.1 Key Features of Rancher 51

6.2 Rancher's use in SUSE Telco Cloud 51

Centralized Kubernetes management 51 • Simplified cluster deployment 52 • Application deployment and management 52 • Security and policy enforcement 52

6.3 Best practices 52

GitOps 52 • Observability 52

6.4 Installing with Edge Image Builder 52

6.5 Additional Resources 53

7 Rancher Dashboard Extensions 54

7.1 Installation 54

Installing with Rancher Dashboard UI 54 • Installing with Helm 56 • Installing with Fleet 56

7.2 KubeVirt Dashboard Extension 58

8 Rancher Turtles 59

8.1 Key Features of Rancher Turtles 59

8.2 Rancher Turtles use in SUSE Telco Cloud 59

8.3 Installing Rancher Turtles 59

8.4 Additional Resources 60

9 Fleet 61

9.1 Installing Fleet with Helm 61

9.2 Using Fleet with Rancher 61

9.3 Accessing Fleet in the Rancher UI 61

Dashboard 62 • Git repos 62 • Clusters 62 • Cluster groups 62 • Advanced 63

9.4 Example of installing KubeVirt with Rancher and Fleet using Rancher dashboard 63

9.5 Debugging and troubleshooting 65

9.6 Fleet examples 66

10 SUSE Linux Micro 67

10.1 How does SUSE Telco Cloud use SUSE Linux Micro? 67

10.2 Best practices 67
Installation media 67 • Local administration 67

10.3 Known issues 68

11 Metal^3 69

11.1 How does SUSE Telco Cloud use Metal^3? 69

11.2 Known issues 69

12 Edge Image Builder 70

12.1 How does SUSE Telco Cloud use Edge Image Builder? 70

12.2 Getting started 71

12.3 Known issues 71

13 Edge Networking 72

13.1 Overview of NetworkManager 72

13.2 Overview of nmstate 72

13.3 Enter: NetworkManager Configurator (nmc) 72

13.4 How does SUSE Telco Cloud use NetworkManager Configurator? 73

13.5 Configuring with Edge Image Builder 73
Prerequisites 73 • Getting the Edge Image Builder container image 73 • Creating the image configuration directory 74 • Creating the image definition file 74 • Defining the network configurations 75 • Building the OS image 80 • Provisioning the

edge nodes 81 • Unified node configurations 87 • Custom network configurations 91

14 RKE2 95

14.1 RKE2 vs K3s 95

14.2 How does SUSE Telco Cloud use RKE2? 95

14.3 Best practices 96
Installation 96 • High availability 96 • Networking 97 • Storage 97

15 SUSE Storage 98

15.1 Prerequisites 98

15.2 Manual installation of SUSE Storage 98
Installing Open-iSCSI 98 • Installing SUSE Storage 99

15.3 Creating SUSE Storage volumes 100

15.4 Accessing the UI 103

15.5 Installing with Edge Image Builder 103

16 SUSE Security 107

16.1 How does SUSE Telco Cloud use SUSE Security? 108

16.2 Important notes 108

16.3 Installing with Edge Image Builder 108

17 MetalLB 109

17.1 How does SUSE Telco Cloud use MetalLB? 109

17.2 Best practices 110

17.3 Known issues 110

18 Endpoint Copier Operator 111

18.1 How does SUSE Telco Cloud use Endpoint Copier Operator? 111

18.2 Best Practices 111

18.3 Known issues 111

19 Edge Virtualization 112

19.1 KubeVirt overview 112

19.2 Prerequisites 113

19.3 Manual installation of Edge Virtualization 113

19.4 Deploying virtual machines 116

19.5 Using virtctl 119

19.6 Simple ingress networking 122

19.7 Using the Rancher UI extension 124
Installation 124 • Using KubeVirt Rancher Dashboard Extension 124

19.8 Installing with Edge Image Builder 126

20 System Upgrade Controller 127

20.1 How does SUSE Telco Cloud use System Upgrade Controller? 127

20.2 Installing the System Upgrade Controller 127
System Upgrade Controller Fleet installation 128 • System Upgrade Controller Helm installation 133

20.3 Monitoring System Upgrade Controller Plans 134
Monitoring System Upgrade Controller Plans - Rancher UI 134 • Monitoring System Upgrade Controller Plans - Manual 135

21 Upgrade Controller 136

21.1 How does SUSE Telco Cloud use Upgrade Controller? 136

21.2 Upgrade Controller vs System Upgrade Controller 137

21.3 Installing the Upgrade Controller 137
Prerequisites 137 • Steps 137

21.4 Installing the Upgrade Controller via Edge Image Builder 138

21.5 How does the Upgrade Controller work? 139
Operating System upgrade 139 • Kubernetes upgrade 140 • Additional components upgrades 141
21.6 Kubernetes API extensions 141
UpgradePlan 141 • ReleaseManifest 143
21.7 Tracking the upgrade process 143
General 144 • Helm Controller 148
21.8 Known Limitations 149

IV REQUIREMENTS & ASSUMPTIONS 150
22 Hardware 151
23 Network 152
24 Port requirements 154
24.1 Management Nodes 154
24.2 Downstream Nodes 157
24.3 CNI specific port requirements 160
25 Services (DHCP, DNS, etc.) 161
26 Disabling systemd services 162

V SETTING UP THE MANAGEMENT CLUSTER 164
27 Introduction 165
28 Steps to set up the management cluster 166
29 Image preparation for connected environments 169
29.1 Directory structure 169
29.2 Management cluster definition file 170

29.3 Custom folder 176

29.4 Kubernetes folder 183

29.5 Networking folder 190

30 Image preparation for air-gap environments 192

30.1 Modifications in the definition file 192

30.2 Modifications in the custom folder 197

30.3 Modifications in the kubernetes folder 197

31 Image creation 201

32 Provision the management cluster 202

33 Dual-stack considerations and configuration 203

VI TELCO FEATURES CONFIGURATION 207

34 Kernel image for real time 210

35 Kernel arguments for low latency and high performance 211

36 CPU Pinning on Host 215

36.1 Isolating CPUs via TuneD 215

36.2 Isolating CPUs via kernel arguments 215

37 CPU Pinning on Kubernetes 219

37.1 RKE2 Versions < v1.32 219

37.2 RKE2 Versions >= v1.32 219

37.3 Deploy Workloads Leveraging Pinned CPUs 220

38 Network Parameters 222

38.1 Core Network Parameters 222

38.2 UDP-Specific Parameters 223

38.3 Real-Time Scheduling 223

39 CNI Configuration 224

39.1 Cilium 224

39.2 Calico 224

39.3 Bond CNI 225

Bond CNI with MACVLAN 225 • Bond CNI with Host Device 228 • Bond CNI with SR-IOV 230

40 SR-IOV 231

40.1 Option 1: SR-IOV Network Device Plugin Daemonset and configMap 231

40.2 Option 2 (Recommended): SR-IOV Network Operator 237

41 DPDK 243

42 vRAN Acceleration (`Intel ACC100/VRB1/VRB2`) 246

42.1 Kernel parameters 246

42.2 Configure SR-IOV on FEC Accelerators 247

42.3 Configure Kubernetes for FEC Acceleration 249

43 Huge pages 250

44 NUMA-aware scheduling 253

44.1 Identifying NUMA nodes 253

45 Metal LB 254

46 Private registry configuration 256

47 Precision Time Protocol 258

47.1 Introduction 258

47.2 Prerequisites 263

47.3 Scenario 1a - Ordinary Clock (PTP-only) 265

47.4 Scenario 1b - Ordinary Clock (Hybrid mode: PTP & SyncE) 272

47.5 Scenario 2a - Boundary Clock (PTP-only) with Multiport Single NIC including propagation to Ordinary Clock server 272

47.6 Scenario 2b - Boundary Clock (Hybrid mode: PTP & SyncE) with Multiport Single NIC including propagation to Ordinary Clock server 278

47.7 Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based) 278

47.8 Scenario 3b - Boundary Clock (Hybrid mode: PTP & SyncE) Multi-NIC with internal timing module (DPLL GNR-D based) 293

48 SCTP - Stream Control Transmission Protocol 294

VII FULLY AUTOMATED DIRECTED NETWORK PROVISIONING 296

49 Introduction 297

50 Prepare downstream cluster image for connected scenarios 299

50.1 Prerequisites for connected scenarios 299

50.2 Image configuration for connected scenarios 299

Downstream cluster image definition file 301 • Growfs script 302 • Performance script 302 • SR-IOV script 303 • Additional configuration for Telco workloads 303 • Additional script for Advanced Network Configuration 305 • Telco required kernel modules 306

50.3 Image creation 307

51 Prepare downstream cluster image for air-gap scenarios 308

51.1 Prerequisites for air-gap scenarios 308

51.2 Image configuration for air-gap scenarios 308
Downstream cluster image definition file 310 • Growfs script 310 • Air-gap script 311 • Performance script 311 • SR-IOV script 311 • Telco required kernel modules 312 • Preparing the air-gap artifacts 312
51.3 Image creation for air-gap scenarios 314
52 Downstream cluster provisioning with Directed network provisioning (single-node) 315
53 Downstream cluster provisioning with Directed network provisioning (multi-node) 324
54 Advanced Network Configuration 339
55 Telco features (DPDK, SR-IOV, CPU isolation, huge pages, NUMA, etc.) 344
55.1 Precision Time Protocol (PTP) 352
Standard scenarios (single ptp4l + phc2sys) 352 • GNR-D multi-NIC scenarios (DPLL) 355
56 Private registry 361
57 Downstream cluster provisioning in air-gapped scenarios 364
57.1 Requirements for air-gapped scenarios 364
57.2 Enroll the bare-metal hosts in air-gap scenarios 364
57.3 Provision the downstream cluster in air-gap scenarios 365
VIII DAY 2 OPERATIONS 372
58 Edge 3.7 migration 373
58.1 Management Cluster 373
Prerequisites 374 • Upgrade Controller 374 • Fleet 375

58.2 Downstream Clusters 376
Fleet 376

59 Management Cluster 377

59.1 Upgrade Controller 377
Prerequisites 377 • Upgrade 378 • Post-Upgrade Steps 379

59.2 Fleet 381
Components 381 • Determine your use-case 382 • Day 2 workflow 383 • OS upgrade 383 • Kubernetes version upgrade 396 • Helm chart upgrade 409

60 Lifecycle actions 432

60.1 Load Balancer Exclusion 432

60.2 Management cluster upgrades 432

60.3 Downstream cluster upgrades 433

60.4 Longhorn Engine Image Cleanup 440
Problem statement 440 • When to implement cleanup 440 • Solution architecture 441 • Implementation 441 • Operational procedures 445

IX HOW-TO GUIDES 448

61 MetalLB on K3s (using Layer 2 Mode) 449

61.1 Why use MetalLB 449

61.2 MetalLB on K3s (using L2) 449

61.3 Prerequisites 450

61.4 Deployment 450

61.5 Configuration 451
Traefik and MetalLB 452

61.6 Usage 452
Ingress with MetalLB 455

62 MetalLB on K3s (using Layer 3 Mode) 458
62.1 Why use MetalLB 458
62.2 MetalLB on K3s (using L3) 458
62.3 Prerequisites 459
62.4 Configuration to Advertise Service IP Addresses 459
62.5 Deployment 459
62.6 Configuration 460
62.7 Usage 461
63 MetalLB in front of the Kubernetes API server 464
63.1 Prerequisites 464
63.2 Installing RKE2/K3s 464
63.3 Configuring an existing cluster 466
63.4 Installing MetalLB 467
63.5 Installing the Endpoint Copier Operator 468
63.6 Adding control-plane nodes 469
64 Air-gapped deployments with Edge Image Builder 471
64.1 Intro 471
64.2 Prerequisites 471
64.3 Libvirt Network Configuration 472
64.4 Base Directory Configuration 472
64.5 Base Definition File 474
64.6 Rancher Installation 475
64.7 SUSE Security Installation 480

64.8 SUSE Storage Installation 483

64.9 KubeVirt and CDI Installation 488

64.10 SUSE Private Registry Installation 491

64.11 Metal^3 Installation 495

64.12 Troubleshooting 497

65 Building Updated SUSE Linux Micro Images with Kiwi 498

65.1 Prerequisites 499

65.2 Getting Started 499

65.3 Building the Default Image 500

65.4 Building images with other profiles 501

65.5 Building images with large sector sizes 501

65.6 Using a custom Kiwi image definition file 502

66 SUSE Storage V2 Data Engine 503

66.1 Prerequisites 503

66.2 Configure V2 for new volumes 503

66.3 Migrate a V1 volume to V2 505

66.4 Usage limitations 507

67 Using clusterclass to deploy downstream clusters 509

67.1 Introduction 509

67.2 What is ClusterClass? 509

67.3 Example of current CAPI provisioning file 510

67.4 Transforming the CAPI provisioning file to ClusterClass 516

ClusterClass definition 516 • Cluster instance definition 525

68 Longhorn Volume Encryption 528
68.1 Introduction 528
68.2 Why implement volume encryption 528
68.3 Architecture overview 529
68.4 Prerequisites 529
68.5 Implementation 530
68.6 Encryption verification summary 536
68.7 Test environment cleanup 537
68.8 CAPI integration 537

X TIPS AND TRICKS 539
69 Edge Image Builder 540
69.1 Common 540
69.2 SUSE Linux Micro 540
69.3 Kubernetes 541
70 Metal³ 542
70.1 `BareMetalHost` selection and Cluster association 542
70.2 Clean up old EFI boot entries 544
70.3 Custom network configuration using the two-secrets approach 545
Example of interface renaming for VLANs 545 • Prerequisites: 546 • Configuring the two secrets 547

XI TROUBLESHOOTING 550

71 General Troubleshooting Principles 551

72 Troubleshooting Kiwi 552

73 Troubleshooting Edge Image Builder (EIB) 554

74 Troubleshooting Edge Networking (NMC) 556

75 Troubleshooting Directed-network provisioning 558

76 Troubleshooting Other components 563

77 Collecting Diagnostics for Support 564

XII APPENDIX 567

78 Release Notes 568

78.1 Abstract 568

78.2 About 568

78.3 Release 3.7.0 569

New Features 569 • Bug & Security Fixes 570 • Known Issues 571 • Component Versions 571

78.4 Removed features 581

78.5 Technology Previews 581

78.6 Component Verification 581

78.7 Upgrade Steps 583

78.8 Product Support Lifecycle 583

78.9 Obtaining source code 584

78.10 Legal notices 584

SUSE Telco Cloud 3.7 Documentation

Welcome to the SUSE Telco Cloud documentation. You will find the high level architectural overview, quick start guides, validated designs, guidance on using components, third-party integrations, and best practices for managing your edge computing infrastructure and workloads.

1 What is SUSE Telco Cloud?

SUSE Telco Cloud is a telecom-optimized computing platform that enables telecom operators and telecom network vendors to innovate and accelerate the modernization of their networks.

SUSE Telco Cloud is a complete Telco-enabled cloud native stack for hosting CNFs, covering all telecom domains: Packet Core, RAN, OSS and BSS.

- Automates zero-touch rollout and lifecycle management of complex edge stack configurations at Telco scale.
- Continuously assures quality on Telco-grade hardware, using Telco-specific configurations and workloads.
- Consists of components that are purpose-built for the edge and hence have smaller footprint and higher performance per Watt.
- Maintains a flexible platform strategy with vendor-neutral APIs and 100% open source.

2 Design Philosophy

The solution is designed with the notion that there is no "one-size-fits-all" edge platform due to customers' widely varying requirements and expectations. Edge deployments push us to solve, and continually evolve, some of the most challenging problems, including massive scalability, restricted network availability, physical space constraints, new security threats and attack vectors, variations in hardware architecture and system resources, the requirement to deploy and interface with legacy infrastructure and applications, and customer solutions that have extended lifespans. Since many of these challenges are different from traditional ways of thinking, e.g. deployment of infrastructure and applications within data centers or in the public cloud, we have to look into the design in much more granular detail, and rethinking many common assumptions.

For example, we find value in minimalism, modularity, and ease of operations. Minimalism is important for edge environments since the more complex a system is, the more likely it is to break. When looking at hundreds of locations, up to hundreds of thousands, complex systems will break in complex ways. Modularity in our solution allows for more user choice while removing unneeded complexity in the deployed platform. We also need to balance these with the ease of operations. Humans may make mistakes when repeating a process thousands of times, so the platform should make sure any potential mistakes are recoverable, eliminating the need for on-site technician visits, but also strive for consistency and standardization.

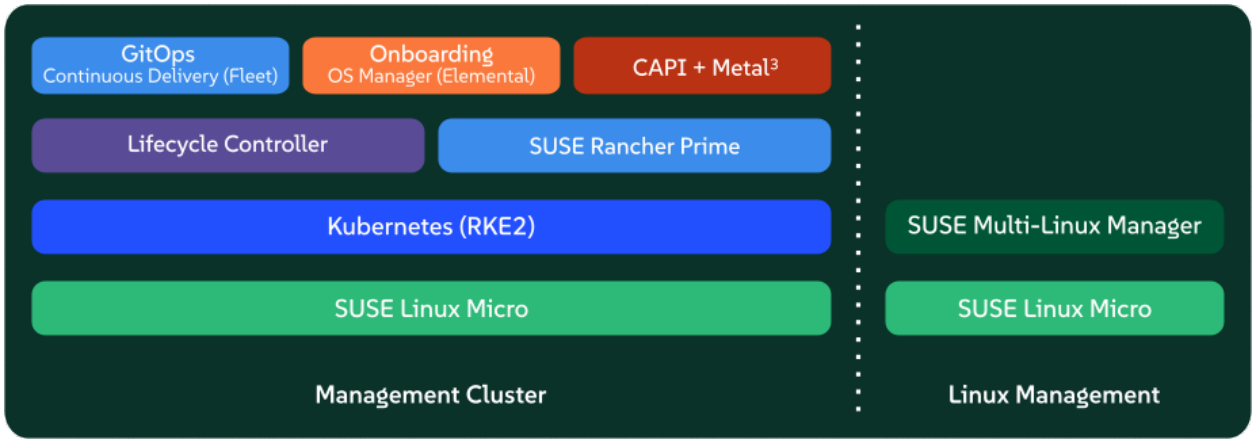
3 High Level Architecture

The high level system architecture of SUSE Telco Cloud is broken into two core categories, namely "management" and "downstream" clusters. The management cluster is responsible for remote management of one or more downstream clusters, although it’s recognized that in certain circumstances, downstream clusters need to operate without remote management, e.g. in situations where an edge site has no external connectivity and needs to operate independently. In SUSE Telco Cloud, the technical components that are utilized for the operation of both the management and downstream clusters are largely common, although likely differentiate in both the system specifications and the applications that reside on-top, i.e. the management cluster would run applications that enable systems management and lifecycle operations, whereas the downstream clusters fulfil the requirements for serving user applications.

3.1 Components used in SUSE Telco Cloud

SUSE Telco Cloud is comprised of both existing SUSE and Rancher components along with additional features and components built by the Edge team to enable us to address the constraints and intricacies required in edge computing. The components used within both the management and downstream clusters are explained below, with a simplified high-level architecture diagram, noting that this isn't an exhaustive list:

3.1.1 Management Cluster



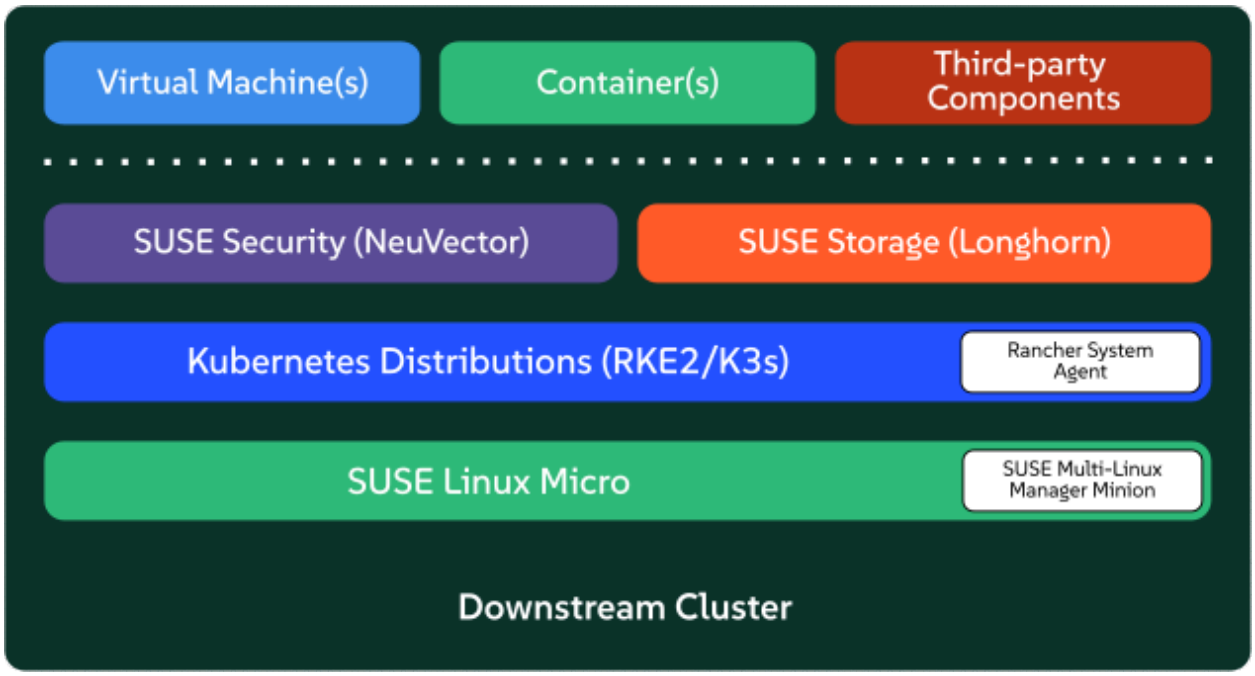
- **Management**: This is the centralized part of SUSE Telco Cloud that is used to manage the provisioning and lifecycle of connected downstream clusters. The management cluster typically includes the following components:

  - Multi-cluster management with Rancher Prime (*Chapter 6, Rancher*), enabling a common dashboard for downstream cluster onboarding and ongoing lifecycle management of infrastructure and applications, also providing comprehensive tenant isolation and `IDP` (Identity Provider) integrations, a large marketplace of third-party integrations and extensions, and a vendor-neutral API.

  - Linux systems management with SUSE Multi-Linux Manager, enabling automated Linux patch and configuration management of the underlying Linux operating system (*SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*)) that runs on the downstream clusters. Note that while this component is containerized, it currently needs to run on a separate system to the rest of the management components, hence labelled as "Linux Management" in the diagram above.

- A dedicated Lifecycle Management (*Chapter 21, Upgrade Controller*) controller that handles management cluster component upgrades to a given SUSE Telco Cloud release.
- An optional full bare-metal lifecycle and management support with Metal3 (*Chapter 11, Metal^3*), MetalLB (*Chapter 17, MetalLB*), and `CAPI` (Cluster API) infrastructure providers, enabling the full end-to-end provisioning of baremetal systems that have remote management capabilities.
- An optional GitOps engine called Fleet (*Chapter 9, Fleet*) for managing the provisioning and lifecycle of downstream clusters and applications that reside on them.
- Underpinning the management cluster itself is SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*) as the base operating system and RKE2 (*Chapter 14, RKE2*) as the Kubernetes distribution supporting the management cluster applications.

3.1.2 Downstream Clusters

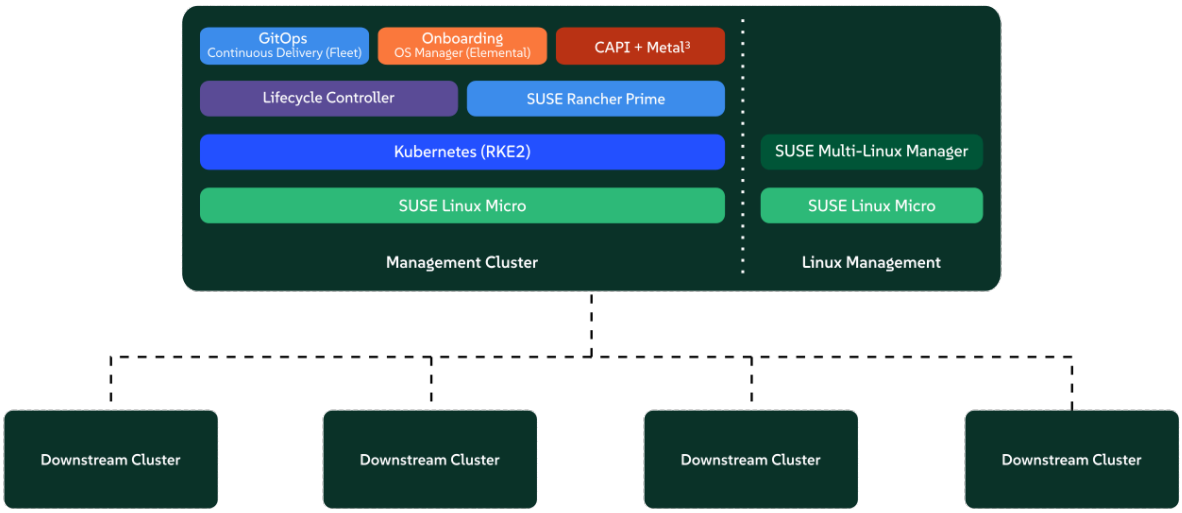


- **Downstream**: This is the distributed part of SUSE Telco Cloud that is used to run the user workloads at the Edge, i.e. the software that is running at the edge location itself, and is typically comprised of the following components:
  - RKE2, a hardened and certified Kubernetes distribution, optimized for usage in government and regulated industries
  - SUSE Security (*Chapter 16, SUSE Security*) to enable security features like image vulnerability scanning, deep packet inspection, and real-time threat and vulnerability protection.
  - Software block storage with SUSE Storage (*Chapter 15, SUSE Storage*) to enable lightweight persistent, resilient, and scalable block-storage.

- A lightweight, container-optimized, hardened Linux operating system with SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*), providing an immutable and highly resilient OS for running containers and virtual machines at the edge. SUSE Linux Micro is available for both AArch64 and AMD64/Intel 64 architectures, and it also supports `Real-Time Kernel` for latency sensitive applications (e.g. telco use-cases).

- For connected clusters (i.e. those that do have connectivity to the management cluster) two agents are deployed, namely Rancher System Agent for managing the connectivity to Rancher Prime, and venv-salt-minion for taking instructions from SUSE Multi-Linux Manager for applying Linux software updates. These agents are not required for management of disconnected clusters.

3.2 Connectivity



The above image provides a high-level architectural overview for **connected** downstream clusters and their attachment to the management cluster. The management cluster can be deployed on a wide variety of underlying infrastructure platforms, in both on-premises and cloud capacities, depending on networking availability between the downstream clusters and the target management cluster. The only requirement for this to function are API and callback URL's to be accessible over the network that connects downstream cluster nodes to the management infrastructure.

It’s important to recognize that there are distinct mechanisms in which this connectivity is established relative to the mechanism of downstream cluster deployment. The details of this are explained in much more depth in the next section, but to set a baseline understanding, there are three primary mechanisms for connected downstream clusters to be established as a "managed" cluster:

1. The downstream clusters are deployed in a "disconnected" capacity at first (e.g. via Edge Image Builder (*Chapter 12, Edge Image Builder*)), and are then imported into the management cluster if/when connectivity allows.

2. The downstream clusters are configured to use the built-in onboarding mechanism and they automatically register into the management cluster at first-boot, allowing for late-binding of the cluster configuration.

3. The downstream clusters have been provisioned with the baremetal management capabilities (CAPI + Metal[3]), and they’re automatically imported into the management cluster once the cluster has been deployed and configured (via the Rancher Turtles operator).

Note

It’s recommended that multiple management clusters are implemented to accommodate the scale of large deployments, optimize for bandwidth and latency concerns in geographically dispersed environments, and to minimize the disruption in the event of an outage or management cluster upgrade. You can find the current management cluster scalability limits and system requirements here (https://ranchermanager.docs.rancher.com/v2.15/getting-started/installation-and-upgrade/installation-requirements).

4 Common Edge Deployment Patterns

Due to the varying set of operating environments and lifecycle requirements, we’ve implemented support for a number of distinct deployment patterns that loosely align to the market segments and use-cases that SUSE Telco Cloud operates in. We have documented a quickstart guide for each of these deployment patterns to help you get familiar with the SUSE Telco Cloud platform based around your needs. The three deployment patterns that we support today are described below, with a link to the respective quickstart page.

4.1 Directed network provisioning

Directed network provisioning is where you know the details of the hardware you wish to deploy to and have direct access to the out-of-band management interface to orchestrate and automate the entire provisioning process. In this scenario, our customers expect a solution to be able to provision edge sites fully automated from a centralized location, going much further than the creation of a boot image by minimizing the manual operations at the edge location; simply rack, power, and attach the required networks to the physical hardware, and the automation process powers up the machine via the out-of-band management (e.g. via the Redfish API) and handles the provisioning, onboarding, and deployment of infrastructure without user intervention. The key for this to work is that the systems are known to the administrators; they know which hardware is in which location, and that deployment is expected to be handled centrally.

This solution is the most robust since you are directly interacting with the hardware's management interface, are dealing with known hardware, and have fewer constraints on network availability. Functionality wise, this solution extensively uses Cluster API and Metal^3 for automated provisioning from bare-metal, through operating system, Kubernetes, and layered applications, and provides the ability to link into the rest of the common lifecycle management capabilities of SUSE Telco Cloud post-deployment. The quick-start for this solution can be found in *Chapter 4, BMC automated deployments with Metal^3*.

4.2 Image-based provisioning

For customers that need to operate in standalone, air-gapped, or network limited environments, SUSE Telco Cloud provides a solution that enables customers to generate fully customized installation media that contains all of the required deployment artifacts to enable both single-node and multi-node highly-available Kubernetes clusters at the edge, including any workloads or additional layered components required, all without any network connectivity to the outside world, and without the intervention of a centralized management platform. The user-experience follows closely to the "phone home" solution in that installation media is provided to the target systems, but the solution will "bootstrap in-place". In this scenario, it's possible to attach the resulting clusters into Rancher for ongoing management (i.e. going from a "disconnected" to "connected" mode of operation without major reconfiguration or redeployment), or can continue to operate in isolation. Note that in both cases the same consistent mechanism for automating lifecycle operations can be applied.

Furthermore, this solution can be used to quickly create management clusters that may host the centralized infrastructure that supports both the "directed network provisioning" and "phone home network provisioning" models as it can be the quickest and most simple way to provision all types of Edge infrastruc-

ture. This solution heavily utilizes the capabilities of SUSE Telco Cloud Image Builder to create fully customized and unattended installation media; the quickstart can be found in *Chapter 5, Standalone clusters with Edge Image Builder*.

5 SUSE Telco Cloud Stack Validation

All SUSE Telco Cloud releases comprise of tightly integrated and thoroughly validated components that are versioned as one. As part of the continuous integration and stack validation efforts that not only test the integration between components but ensure that the system performs as expected under forced failure scenarios, the SUSE Telco Cloud team publishes all of the test runs and the results to the public. The results along with all input parameters can be found at ci.edge.suse.com (https://ci.edge.suse.com).

6 Full Component List

The full list of components, along with a link to a high-level description of each and how it's used in SUSE Telco Cloud can be found below:

- Rancher (*Chapter 6, Rancher*)
- Rancher Dashboard Extensions (*Chapter 7, Rancher Dashboard Extensions*)
- Rancher Turtles (*Chapter 8, Rancher Turtles*)
- SUSE Multi-Linux Manager
- Fleet (*Chapter 9, Fleet*)
- SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*)
- Metal³ (*Chapter 11, Metal³*)
- Edge Image Builder (*Chapter 12, Edge Image Builder*)
- NetworkManager Configurator (*Chapter 13, Edge Networking*)
- RKE2 (*Chapter 14, RKE2*)
- SUSE Storage (*Chapter 15, SUSE Storage*)
- SUSE Security (*Chapter 16, SUSE Security*)

- MetalLB (*Chapter 17, MetalLB*)
- KubeVirt (*Chapter 19, Edge Virtualization*)
- System Upgrade Controller (*Chapter 20, System Upgrade Controller*)
- Upgrade Controller (*Chapter 21, Upgrade Controller*)

I Concept & Architecture

1 SUSE Telco Cloud Architecture 2

2 Components 3

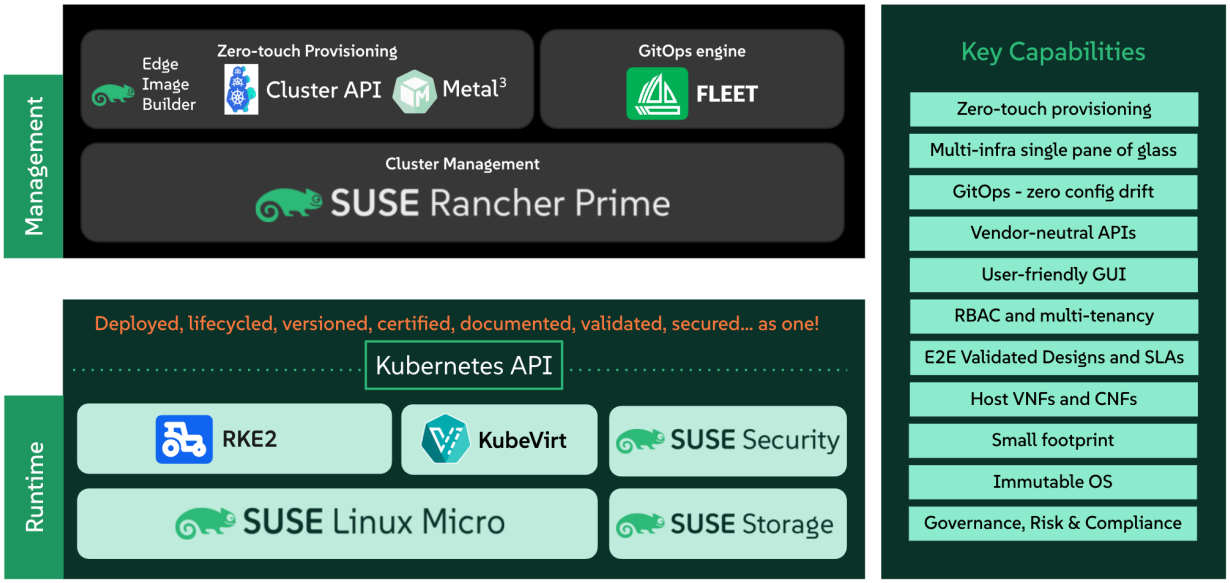
3 Deployment Model 4

SUSE Telco Cloud is a platform designed for hosting modern, cloud native, Telco applications at scale from core to edge.

This page explains the architecture and components used in SUSE Telco Cloud.

1 SUSE Telco Cloud Architecture

The following diagram shows the high-level architecture of SUSE Telco Cloud:



2 Components

There are two different blocks, the management stack and the runtime stack:

- **Management stack**: This is the part of SUSE Telco Cloud that is used to manage the provision and lifecycle of the runtime stacks. It includes the following components:
  - Multi-cluster management in public and private cloud environments with Rancher (*Chapter 6, Rancher*)
  - Bare-metal support with Metal3 (*Chapter 11, Metal^3*), MetalLB (*Chapter 17, MetalLB*) and `CAPI` (Cluster API) infrastructure providers
  - Comprehensive tenant isolation and `IDP` (Identity Provider) integrations
  - Large marketplace of third-party integrations and extensions
  - Vendor-neutral API and rich ecosystem of providers
  - Control the SUSE Linux Micro transactional updates
  - GitOps Engine for managing the lifecycle of the clusters using Git repositories with Fleet (*Chapter 9, Fleet*)
- **Runtime stack**: This is the part of SUSE Telco Cloud that is used to run the workloads.
  - RKE2 (*Chapter 14, RKE2*) serves as the security-hardened, lightweight Kubernetes distribution, optimized for edge and compliance-focused telecom environments.
  - (Optional) SUSE Security (*Chapter 16, SUSE Security*) to enable security features like image vulnerability scanning, deep packet inspection and automatic intra-cluster traffic control.
  - (Optional) Block Storage with SUSE Storage (*Chapter 15, SUSE Storage*) to enable a simple and easy way to use a cloud native storage solution.
  - Optimized Operating System with SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*) to enable a secure, lightweight and immutable (transactional file system) OS for running containers. SUSE Linux Micro is available on AArch64 and AMD64/Intel 64 architectures, and it also supports `Real-Time Kernel` for Telco and edge use cases.

3 Deployment Model

SUSE Telco Cloud follows a two-stage deployment model: the management cluster is deployed using an image generated by Edge Image Builder (*Chapter 12, Edge Image Builder*), and downstream clusters are provisioned via Directed Network Provisioning. This chapter gives an overview of this deployment model, as it is the recommended and supported approach for SUSE Telco Cloud environments.

3.1 Why This Deployment Model?

The management cluster is a one-time, single-site deployment. Image-based Provisioning bundles all the components into a single bootable image. This way, the management cluster will bootstrap itself, requiring minimal operational complexity. It is the simplest and most straightforward way to get it up and running.

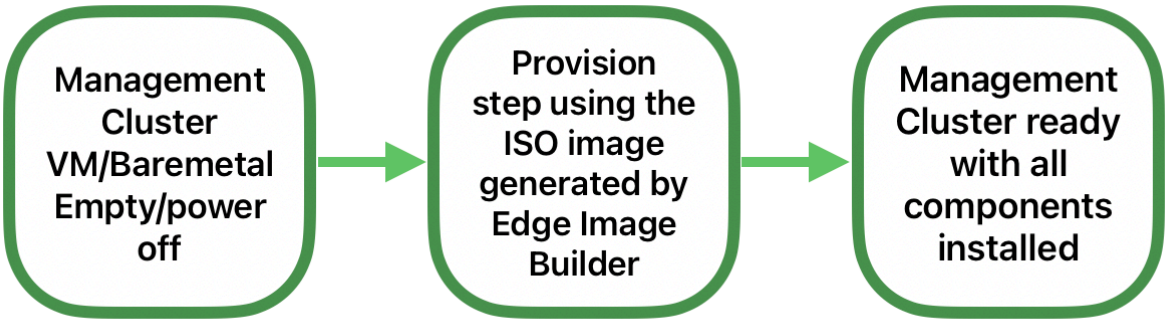
Downstream clusters, however, are a different story. In telco environments, they are deployed at scale across many data centers and edge sites, often with no on-site technical expertise available. Directed Network Provisioning is designed for this: it requires that target servers support an out-of-band management interface such as Redfish, through which the management cluster remotely powers on, inspects, and provisions bare-metal nodes without any on-site intervention.

Baking all cluster-specific configuration into a boot image, as Image-based Provisioning would require, means every site variation demands a different image, and any configuration change requires rebuilding and redistributing images across all sites. Directed Network Provisioning solves this by keeping the OS image generic and driving all cluster-specific configuration, including networking, Kubernetes, and Telco profiles, from the management cluster at provisioning time. Operators simply rack, power, and connect the hardware and the management cluster handles the rest.

This separation also unlocks full GitOps integration. Since the entire downstream cluster definition is expressed as Cluster API manifests, it can be stored in Git and reconciled by Fleet across all sites, making provisioning, configuration, and lifecycle operations fully auditable and repeatable without manual intervention.

3.2 Deployment of Management Cluster

Using the Edge Image Builder (*Chapter 12, Edge Image Builder*) to create a new `ISO` image with the management stack included. You can then use this `ISO` image to install a new management cluster on VMs or bare-metal.



Note

For more information about how to deploy a new management cluster, see the SUSE Telco Cloud Management Cluster guide (*Part V, "Setting up the management cluster"*).

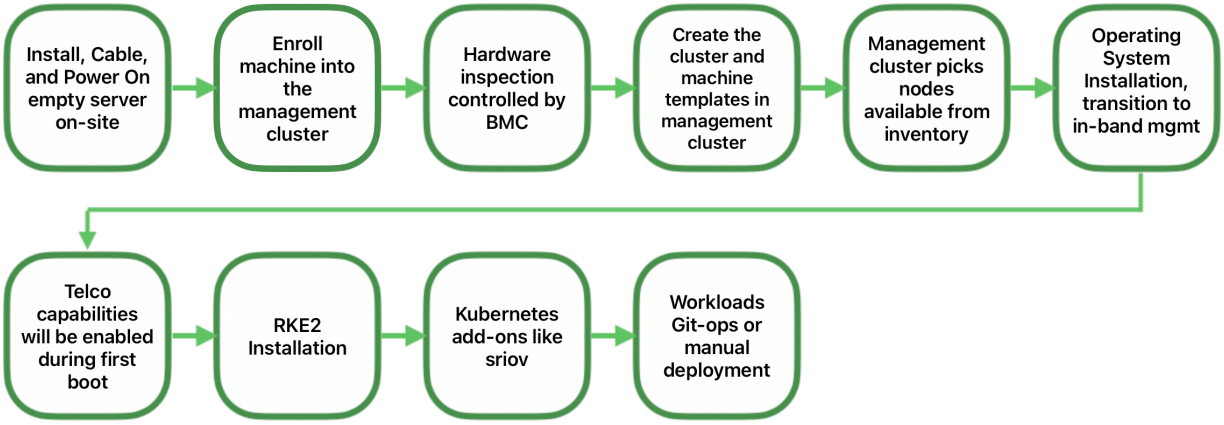
Note

For more information about how to use the Edge Image Builder, see the Edge Image Builder guide (*Chapter 5, Standalone clusters with Edge Image Builder*).

3.3 Deployment of a Single-Node Downstream Cluster with Telco Profiles

Once we have the management cluster up and running, we can use it to deploy a single-node downstream cluster with all Telco capabilities enabled and configured using the directed network provisioning workflow.

The following diagram shows the high-level workflow to deploy it:



Note

For more information about how to deploy a downstream cluster, see the SUSE Telco Cloud Automated Provisioning guide. (*Part VII, "Fully automated directed network provisioning"*)

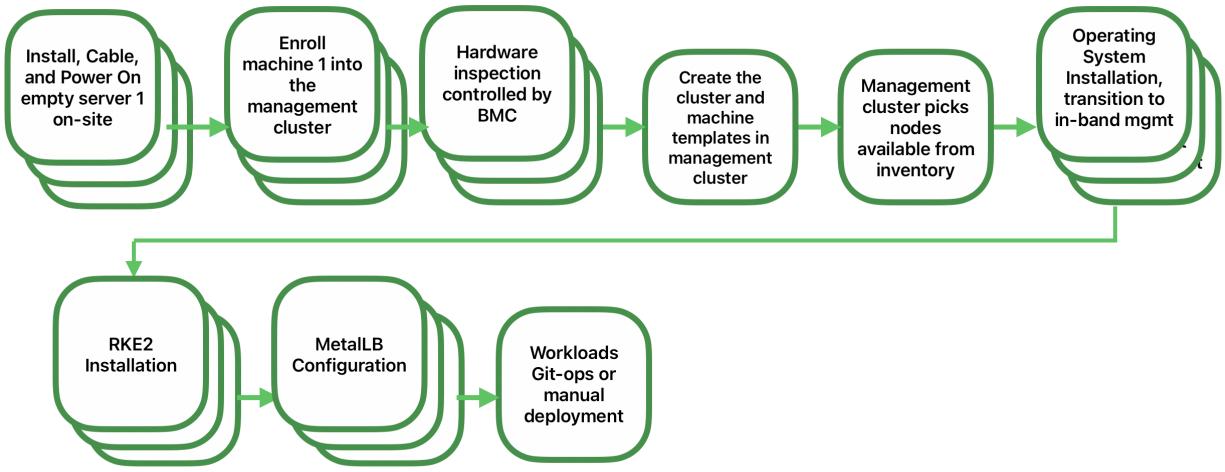
Note

For more information about Telco features, see the SUSE Telco Cloud Telco Features guide. (*Part VI, "Telco features configuration"*)

3.4 Deployment of a Highly-Available Downstream Cluster

Once we have the management cluster up and running, we can use it to deploy a high availability downstream cluster with `MetalLB` as a load balancer using the directed network provisioning workflow.

The following diagram shows the high-level workflow to deploy it:



Note

For more information about how to deploy a downstream cluster, see the SUSE Telco Cloud Automated Provisioning guide. (*Part VII, "Fully automated directed network provisioning"*)

Note

For more information about `MetalLB`, see here: (*Chapter 17, MetalLB*)

II Quick Starts

4 BMC automated deployments with Metal^3 9

5 Standalone clusters with Edge Image Builder 33

Quick Starts here

4 BMC automated deployments with Metal^3

Metal^3 is a CNCF project (https://metal3.io/) which provides bare-metal infrastructure management capabilities for Kubernetes.

Metal^3 provides Kubernetes-native resources to manage the lifecycle of bare-metal servers which support management via out-of-band protocols such as Redfish (https://www.dmtf.org/standards/redfish).

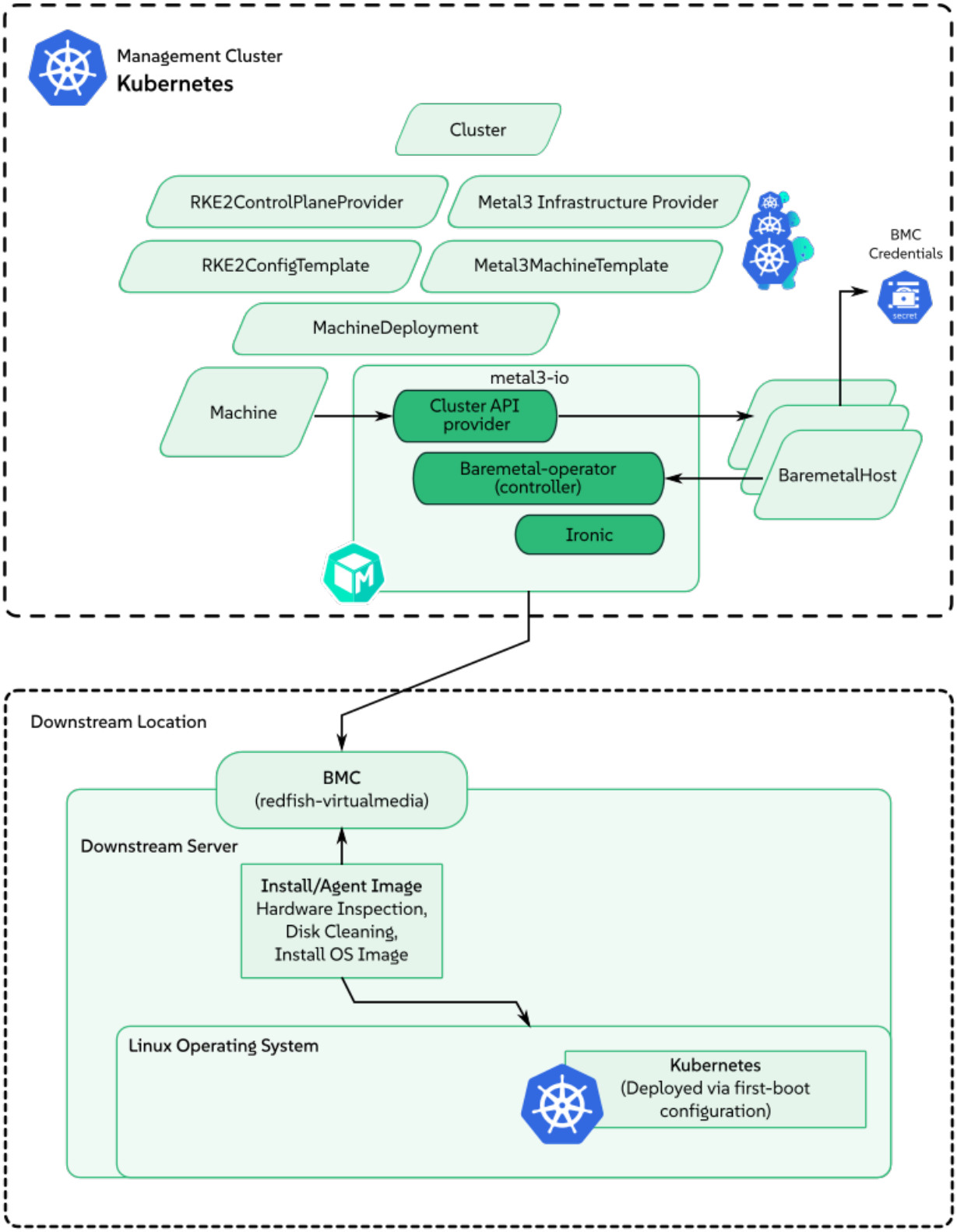
It also has mature support for Cluster API (CAPI) (https://cluster-api.sigs.k8s.io/) which enables management of infrastructure resources across multiple infrastructure providers via broadly adopted vendor-neutral APIs.

4.1 Why use this method

This method is useful for scenarios where the target hardware supports out-of-band management, and a fully automated infrastructure management flow is desired.

A management cluster is configured to provide declarative APIs that enable inventory and state management of downstream cluster bare-metal servers, including automated inspection, cleaning and provisioning/deprovisioning.

4.2 High-level architecture



4.3 Prerequisites

There are some specific constraints related to the downstream cluster server hardware and networking:

- Management cluster
  - Must have network connectivity to the target server management/BMC API
  - Must have network connectivity to the target server control plane network
  - For multi-node management clusters, an additional reserved IP address is required
- Hosts to be controlled
  - Must support out-of-band management via Redfish, iDRAC or iLO interfaces
  - Must support deployment via virtual media (PXE is not currently supported)
  - Must have network connectivity to the management cluster for access to the Metal^3 provisioning APIs

Some tools are required, these can be installed either on the management cluster, or on a host which can access it.

- Kubectl (https://kubernetes.io/docs/reference/kubectl/kubectl/), Helm (https://helm.sh) and Clusterctl (https://cluster-api.sigs.k8s.io/user/quick-start.html#install-clusterctl)
- A container runtime such as Podman (https://podman.io) or Rancher Desktop (https://rancherdesktop.io)
- A SUSE Linux Micro 6.2 raw image created using the Kiwi Builder (*Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi*)

4.4 Deployment

4.4.1 Setup Management Cluster

The basic steps to install a management cluster and use Metal^3 are:

1. Install an RKE2 management cluster
2. Install Rancher

3. Install a storage provider (optional)
4. Install the Metal^3 dependencies
5. Install CAPI provider dependencies
6. Build a SLEMicro OS image for downstream cluster hosts
7. Register BareMetalHost CRs to define the bare-metal inventory
8. Create a downstream cluster by defining CAPI resources

This guide assumes an existing RKE2 cluster and Rancher (including cert-manager) has been installed, for example by using Edge Image Builder (*Chapter 12, Edge Image Builder*).

Tip

The steps here can also be fully automated as described in the Management Cluster Documentation (*Part V, "Setting up the management cluster"*).

4.4.2 Installing Metal^3 dependencies

If not already installed as part of the Rancher installation, cert-manager must be installed and running.

An additional IP is required, which is managed by MetalLB (https://metallb.universe.tf/) to provide a consistent endpoint for the Metal^3 management services. This IP must be part of the control plane subnet and reserved for static configuration (not part of any DHCP pool).

Tip

If the management cluster is a single node, the requirement for an additional floating IP managed via MetalLB can be avoided, see *Section 4.7.1, "Single-node configuration"*

1. First, we install MetalLB:

```
helm install \
  metallb oci://registry.suse.com/edge/charts/metallb \
  --namespace metallb-system \
```

```
  --create-namespace
```

2. Then we define an `IPAddressPool` and `L2Advertisement` using the reserved IP, defined as `STATIC_IRONIC_IP` below:

```
export STATIC_IRONIC_IP=<STATIC_IRONIC_IP>

cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: ironic-ip-pool
  namespace: metallb-system
spec:
  addresses:
  - ${STATIC_IRONIC_IP}/32
  serviceAllocation:
    priority: 100
    serviceSelectors:
    - matchExpressions:
      - {key: app.kubernetes.io/name, operator: In, values: [metal3-ironic]}
EOF
```

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ironic-ip-pool-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
  - ironic-ip-pool
EOF
```

3. Now Metal^3 can be installed:

```
helm install \
  metal3 oci://registry.suse.com/edge/charts/metal3 \
  --namespace metal3-system \
  --create-namespace \
  --set global.ironicIP="$STATIC_IRONIC_IP"
```

4. It can take around two minutes for the init container to run on this deployment, so ensure the pods are all running before proceeding:

```
kubectl get pods -n metal3-system
```

```
NAME                                                      READY   STATUS    RESTARTS
 AGE
baremetal-operator-controller-manager-85756794b-fz98d   2/2     Running   0
 15m
metal3-metal3-ironic-677bc5c8cc-55shd                   4/4     Running   0
 15m
metal3-metal3-mariadb-7c7d6fdbd8-64c7l                  1/1     Running   0
 15m
```

Warning

Do not proceed to the following steps until all pods in the `metal3-system` namespace are running.

4.4.3 Installing cluster API provider dependencies

Cluster API provider dependencies are managed via the Rancher Turtles Providers Helm chart:

```
helm install \
  rancher-turtles oci://registry.suse.com/edge/charts/rancher-turtles-providers \
  --namespace cattle-turtles-system \
  --create-namespace
```

After some time, the controller pods should be running in the `cattle-capi-system`, `capm3-system`, `rke2-bootstrap-system` and `rke2-control-plane-system` namespaces.

4.4.4 Prepare downstream cluster image

Kiwi (*Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi*) and Edge Image Builder (*Chapter 12, Edge Image Builder*) are used to prepare a modified SLEMicro base image which is provisioned on downstream cluster hosts.

In this guide, we cover the minimal configuration necessary to deploy the downstream cluster.

4.4.4.1 Image configuration

Note

Please follow *Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi* first to build a fresh image as the first step required to create clusters.

When running Edge Image Builder, a directory is mounted from the host, so it is necessary to create a directory structure to store the configuration files used to define the target image.

- `downstream-cluster-config.yaml` is the image definition file, see *Chapter 5, Standalone clusters with Edge Image Builder* for more details.
- Copy the Kiwi Builder created base image to the `base-images` directory.
- The `network` folder is optional, see *Section 4.4.5.1.1, "Additional script for static network configuration"* for more details.
- The custom/scripts directory contains scripts to be run on first-boot; currently a `01-fix-growfs.sh` script is required to resize the OS root partition on deployment

```
├── downstream-cluster-config.yaml
├── base-images/
│   └ SL-Micro.x86_64-6.2-Base-GM.raw
├── network/
│   └ configure-network.sh
└── custom/
    └ scripts/
        └ 01-fix-growfs.sh
```

4.4.4.1.1 Downstream cluster image definition file

The `downstream-cluster-config.yaml` file is the main configuration file for the downstream cluster image. The following is a minimal example for deployment via Metal[3]:

```
apiVersion: 1.3
image:
  imageType: raw
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.2-Base-GM.raw
  outputImageName: SLE-Micro-eib-output.raw
operatingSystem:
  time:
    timezone: Europe/London
    ntp:
      forceWait: true
      pools:
        - 2.suse.pool.ntp.org
      servers:
        - 10.0.0.1
        - 10.0.0.2
```

```
  kernelArgs:
    - ignition.platform.id=openstack
  systemd:
    disable:
      - rebootmgr
      - transactional-update.timer
      - transactional-update-cleanup.timer
  users:
    - username: root
      encryptedPassword: $ROOT_PASSWORD
      sshKeys:
      - $USERKEY1
      createHomeDir: true
  packages:
    packageList:
      - jq
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

Where `$SCC_REGISTRATION_CODE` is the registration code copied from SUSE Customer Center (https://scc.suse.com/), and the package list contains `jq` which is required.

`$ROOT_PASSWORD` is the encrypted password for the root user, which can be useful for test/debugging. It can be generated with the `openssl passwd -6 PASSWORD` command.

For the production environments, it is recommended to use the SSH keys that can be added to the users block replacing the `$USERKEY1` with the real SSH keys.

Note

Note that `ignition.platform.id=openstack` is mandatory - without this argument SUSE Linux Micro configuration via ignition will fail in the Metal[3] automated flow.

The `time` section is optional but it is highly recommended to be configured to avoid potential issues with certificates and clock skew. The values provided in this example are for illustrative purposes only. Please adjust them to fit your specific requirements.

4.4.4.1.2 Growfs script

Currently, a custom script (`custom/scripts/01-fix-growfs.sh`) is required to grow the file system to match the disk size on first-boot after provisioning. The `01-fix-growfs.sh` script contains the following information:

```
#!/bin/bash
```

```
growfs() {
  mnt="$1"
  dev="$(findmnt --fstab --target ${mnt} --evaluate --real --output SOURCE --noheadings)"
  # /dev/sda3 -> /dev/sda, /dev/nvme0n1p3 -> /dev/nvme0n1
  parent_dev="/dev/$(lsblk --nodeps -rno PKNAME "${dev}")"
  # Last number in the device name: /dev/nvme0n1p42 -> 42
  partnum="$(echo "${dev}" | sed 's/^.*[^0-9]\([0-9]\+\)$/\1/')"
  ret=0
  growpart "$parent_dev" "$partnum" || ret=$?
  [ $ret -eq 0 ] || [ $ret -eq 1 ] || exit 1
  /usr/lib/systemd/systemd-growfs "$mnt"
}
growfs /
```

Note

Add your own custom scripts to be executed during the provisioning process using the same approach. For more information, see *Chapter 5, Standalone clusters with Edge Image Builder*.

4.4.4.2 Image creation

Once the directory structure is prepared following the previous sections, run the following command to build the image:

```
podman run --rm --privileged -it -v $PWD:/eib \
 registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
 build --definition-file downstream-cluster-config.yaml
```

This creates the output image file named `SLE-Micro-eib-output.raw`, based on the definition described above.

The output image must then be made available via a webserver, either the media-server container enabled via the Metal3 chart (*Note*) or some other locally accessible server. In the examples below, we refer to this server as `imagecache.local:8080`

Note

When deploying EIB images to downstream clusters, it is required also to include the sha256 sum of the image on the `Metal3MachineTemplate` object. It can be generated as:

```
sha256sum <image_file> > <image_file>.sha256
# On this example:
```

```
sha256sum SLE-Micro-eib-output.raw > SLE-Micro-eib-output.raw.sha256
```

4.4.5 Adding BareMetalHost inventory

Registering bare-metal servers for automated deployment requires creating two resources: a Secret storing BMC access credentials and a Metal[3] BareMetalHost resource defining the BMC connection and other details:

```
apiVersion: v1
kind: Secret
metadata:
  name: controlplane-0-credentials
type: Opaque
data:
  username: YWRtaW4=
  password: cGFzc3dvcmQ=
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: controlplane-0
  labels:
    cluster-role: control-plane
spec:
  architecture: x86_64
  online: true
  bootMACAddress: "00:f3:65:8a:a3:b0"
  bmc:
    address: redfish-virtualmedia://192.168.125.1:8000/redfish/v1/Systems/68bd0fb6-
d124-4d17-a904-cdf33efe83ab
    disableCertificateVerification: true
    credentialsName: controlplane-0-credentials
```

Note the following:

- The Secret username/password must be base64 encoded. Note this should not include any trailing newlines (for example, use `echo -n`, not just `echo`!)
- The `cluster-role` label may be set now or later on cluster creation. In the example below, we expect `control-plane` or `worker`
- `bootMACAddress` must be a valid MAC that matches the control plane NIC of the host

- The `bmc` address is the connection to the BMC management API, the following are supported:
  - `redfish-virtualmedia://<IP ADDRESS>/redfish/v1/Systems/<SYSTEM ID>`: Redfish virtual media, for example, SuperMicro
  - `idrac-virtualmedia://<IP ADDRESS>/redfish/v1/Systems/System.Embedded.1`: Dell iDRAC
- See the Upstream API docs (https://github.com/metal3-io/baremetal-operator/blob/main/docs/api.md) for more details on the BareMetalHost API

4.4.5.1 Configuring Static IPs

The BareMetalHost example above assumes DHCP provides the controlplane network configuration, but for scenarios where manual configuration is needed such as static IPs it is possible to provide additional configuration, as described below.

4.4.5.1.1 Additional script for static network configuration

When creating the base image with Edge Image Builder, in the `network` folder, create the following `configure-network.sh` file.

This consumes configuration drive data on first-boot, and configures the host networking using the NM Configurator tool (https://github.com/suse-edge/nm-configurator).

```
#!/bin/bash

set -eux

# Attempt to statically configure a NIC in the case where we find a network_data.json
# In a configuration drive

CONFIG_DRIVE=$(blkid --label config-2 || true)
if [ -z "${CONFIG_DRIVE}" ]; then
  echo "No config-2 device found, skipping network configuration"
  exit 0
fi

mount -o ro $CONFIG_DRIVE /mnt

META_DATA_FILE="/mnt/openstack/latest/meta_data.json"
if [ ! -f "${META_DATA_FILE}" ]; then
```

```
  umount /mnt
  echo "No meta_data.json found, skipping hostname configuration"
  exit 0
fi

DESIRED_HOSTNAME=$(cat /mnt/openstack/latest/meta_data.json | tr ',{}' '\n' | grep
 '"metal3-name"' | sed 's/.*\"metal3-name\": \"\(.*\)\"/\1/')
echo "${DESIRED_HOSTNAME}" > /etc/hostname

NETWORK_DATA_FILE="/mnt/openstack/latest/network_data.json"

if [ ! -f "${NETWORK_DATA_FILE}" ]; then
  umount /mnt
  echo "No network_data.json found, skipping network configuration"
  exit 0
fi

mkdir -p /tmp/nmc/{desired,generated}
cp ${NETWORK_DATA_FILE} /tmp/nmc/desired/_all.yaml
umount /mnt

./nmc generate --config-dir /tmp/nmc/desired --output-dir /tmp/nmc/generated
./nmc apply --config-dir /tmp/nmc/generated
```

4.4.5.1.2 Additional secret with host network configuration

An additional secret containing data in the nmstate (https://nmstate.io/) format supported by NM Configurator (*Chapter 13, Edge Networking*) can be defined for each host.

The secret is then referenced in the `BareMetalHost` resource via the `preprovisioningNetwork-DataName` spec field.

```
apiVersion: v1
kind: Secret
metadata:
  name: controlplane-0-networkdata
type: Opaque
stringData:
  networkData: |
    interfaces:
    - name: enp1s0
      type: ethernet
      state: up
      mac-address: "00:f3:65:8a:a3:b0"
      ipv4:
        address:
```

```
        - ip:  192.168.125.200
          prefix-length: 24
        enabled: true
        dhcp: false
    dns-resolver:
      config:
        server:
        - 192.168.125.1
    routes:
      config:
      - destination: 0.0.0.0/0
        next-hop-address: 192.168.125.1
        next-hop-interface: enp1s0
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: controlplane-0
  labels:
    cluster-role: control-plane
spec:
  preprovisioningNetworkDataName: controlplane-0-networkdata
# Remaining content as in previous example
```

Note

In some circumstances the MAC address may be omitted. See *Section 13.5.8, "Unified node configurations"* for additional details.

4.4.5.2 BareMetalHost preparation

After creating the BareMetalHost resource and associated secrets as described above, a host preparation workflow is triggered:

- A ramdisk image is booted by virtualmedia attachment to the target host BMC
- The ramdisk inspects hardware details, and prepares the host for provisioning (for example by cleaning disks of previous data)
- On completion of this process, hardware details in the BareMetalHost `status.hardware` field are updated and can be verified

This process can take several minutes, but when completed you should see the BareMetalHost state become `available`:

```
% kubectl get baremetalhost
NAME              STATE       CONSUMER   ONLINE   ERROR   AGE
controlplane-0    available              true             9m44s
worker-0          available              true             9m44s
```

4.4.6 Creating downstream clusters

We now create Cluster API resources which define the downstream cluster, and Machine resources which will cause the BareMetalHost resources to be provisioned, then bootstrapped to form an RKE2 cluster.

4.4.7 Control plane deployment

To deploy the controlplane we define a yaml manifest similar to the one below, which contains the following resources:

- Cluster resource defines the cluster name, networks, and type of controlplane/infrastructure provider (in this case RKE2/Metal3)
- Metal3Cluster defines the controlplane endpoint (host IP for single-node, LoadBalancer endpoint for multi-node, this example assumes single-node)
- RKE2ControlPlane defines the RKE2 version and any additional configuration needed during cluster bootstrapping
- Metal3MachineTemplate defines the OS Image to be applied to the BareMetalHost resources, and the hostSelector defines which BareMetalHosts to consume
- Metal3DataTemplate defines additional metaData to be passed to the BareMetalHost (note networkData is not currently supported in the Edge solution)

Note

For simplicity this example assumes a single-node control plane where the BareMetalHost is configured with an IP of `192.168.125.200`. For more advanced multi-node examples, please see *Part VII, "Fully automated directed network provisioning"*.

```
apiVersion: cluster.x-k8s.io/v1beta2
```

```yaml
kind: Cluster
metadata:
  name: sample-cluster
  namespace: default
  labels:
    cluster-api.cattle.io/rancher-auto-import: "true"
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
    services:
      cidrBlocks:
        - 10.96.0.0/12
  controlPlaneRef:
    apiGroup: controlplane.cluster.x-k8s.io
    kind: RKE2ControlPlane
    name: sample-cluster
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3Cluster
    name: sample-cluster
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3Cluster
metadata:
  name: sample-cluster
  namespace: default
spec:
  controlPlaneEndpoint:
    host: 192.168.125.200
    port: 6443
  cloudProviderEnabled: false
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: sample-cluster
  namespace: default
spec:
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: sample-cluster-controlplane
  replicas: 1
  version: v1.36.3+rke2r1
  rolloutStrategy:
```

```
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  agentConfig:
    format: ignition
    kubelet:
      extraArgs:
      - provider-id=metal3://PROVIDERID
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
          - name: rke2-preinstall.service
            enabled: true
            contents: |
              [Unit]
              Description=rke2-preinstall
              Wants=network-online.target
              Before=rke2-install.service
              ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
              [Service]
              Type=oneshot
              User=root
              ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
              # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
              ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
              ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStartPost=/bin/sh -c "umount /mnt"
              [Install]
              WantedBy=multi-user.target
        storage:
          directories:
          - path: /var/lib/rancher/rke2/server/manifests
            overwrite: true
          files:
          - path: /var/lib/rancher/rke2/server/manifests/rke2-traefik-config.yaml
            overwrite: true
            contents:
```

```
              inline: |
                apiVersion: helm.cattle.io/v1
                kind: HelmChartConfig
                metadata:
                  name: rke2-traefik
                  namespace: kube-system
                spec:
                  valuesContent: |-
                    ingressClass:
                      isDefaultClass: true
                    ports:
                      web:
                        hostPort: null    # disallow hostPort
                        exposedPort: 80
                      websecure:
                        hostPort: null    # disallow hostPort
                        exposedPort: 443
                    service:
                      enabled: true
                      spec:
                        type: LoadBalancer
                        externalTrafficPolicy: Local
                        allocateLoadBalancerNodePorts: false
            mode: 0644
            user:
              name: root
            group:
              name: root
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: sample-cluster-controlplane
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: sample-cluster-controlplane-template
      hostSelector:
        matchLabels:
          cluster-role: control-plane
      image:
        checksum: http://imagecache.local:8080/SLE-Micro-eib-output.raw.sha256
        checksumType: sha256
        diskFormat: raw
        url: http://imagecache.local:8080/SLE-Micro-eib-output.raw
```

```
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3DataTemplate
metadata:
  name: sample-cluster-controlplane-template
  namespace: default
spec:
  clusterName: sample-cluster
  metaData:
    objectNames:
    - key: name
      object: machine
    - key: local-hostname
      object: machine
    - key: local_hostname
      object: machine
```

Note

Adding the label `cluster-api.cattle.io/rancher-auto-import: "true"` to the `cluster.x-k8s.io` objects will import the cluster into Rancher (by creating a corresponding `clusters.management.cattle.io` object). See the Cluster API documentation (https://documentation.suse.com/cloudnative/cluster-api/latest/en/tutorials/first-cluster.html#_mark_namespace_for_auto_import) for more information.

Once adapted to your environment, you can apply the example via `kubectl` and then monitor the cluster status via `clusterctl`.

```
% kubectl apply -f rke2-control-plane.yaml

# Wait for the cluster to be provisioned
% clusterctl describe cluster sample-cluster
NAME                                                    READY  SEVERITY  REASON  SINCE
 MESSAGE
Cluster/sample-cluster                                  True                      22m
├─ClusterInfrastructure - Metal3Cluster/sample-cluster  True                      27m
├─ControlPlane - RKE2ControlPlane/sample-cluster        True                      22m
│ └─Machine/sample-cluster-chflc                        True                      23m
```

4.4.8 Worker/Compute deployment

Similar to the control plane deployment, we define a YAML manifest which contains the following resources:

- MachineDeployment defines the number of replicas (hosts) and the bootstrap/infrastructure provider (in this case RKE2/Metal3)
- RKE2ConfigTemplate describes the RKE2 version and first-boot configuration for agent host bootstrapping
- Metal3MachineTemplate defines the OS Image to be applied to the BareMetalHost resources, and the host selector defines which BareMetalHosts to consume
- Metal3DataTemplate defines additional metadata to be passed to the BareMetalHost (note that `networkData` is not currently supported)

```
apiVersion: cluster.x-k8s.io/v1beta2
kind: MachineDeployment
metadata:
  labels:
    cluster.x-k8s.io/cluster-name: sample-cluster
  name: sample-cluster
  namespace: default
spec:
  clusterName: sample-cluster
  replicas: 1
  selector:
    matchLabels:
      cluster.x-k8s.io/cluster-name: sample-cluster
  template:
    metadata:
      labels:
        cluster.x-k8s.io/cluster-name: sample-cluster
    spec:
      bootstrap:
        configRef:
          apiGroup: bootstrap.cluster.x-k8s.io
          kind: RKE2ConfigTemplate
          name: sample-cluster-workers
      clusterName: sample-cluster
      infrastructureRef:
        apiGroup: infrastructure.cluster.x-k8s.io
        kind: Metal3MachineTemplate
        name: sample-cluster-workers
      deletion:
```

```
        nodeDrainTimeoutSeconds: 0
      version: v1.36.3+rke2r1
---
apiVersion: bootstrap.cluster.x-k8s.io/v1beta2
kind: RKE2ConfigTemplate
metadata:
  name: sample-cluster-workers
  namespace: default
spec:
  template:
    spec:
      agentConfig:
        format: ignition
        version: v1.36.3+rke2r1
        kubelet:
          extraArgs:
          - provider-id=metal3://PROVIDERID
        additionalUserData:
          config: |
            variant: fcos
            version: 1.4.0
            systemd:
              units:
              - name: rke2-preinstall.service
                enabled: true
                contents: |
                  [Unit]
                  Description=rke2-preinstall
                  Wants=network-online.target
                  Before=rke2-install.service
                  ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                  [Service]
                  Type=oneshot
                  User=root
                  ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                  # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
                  ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
                  ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                  ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
                  ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                  ExecStartPost=/bin/sh -c "umount /mnt"
                  [Install]
```

```
                  WantedBy=multi-user.target
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: sample-cluster-workers
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: sample-cluster-workers-template
      hostSelector:
        matchLabels:
          cluster-role: worker
      image:
        checksum: http://imagecache.local:8080/SLE-Micro-eib-output.raw.sha256
        checksumType: sha256
        diskFormat: raw
        url: http://imagecache.local:8080/SLE-Micro-eib-output.raw
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3DataTemplate
metadata:
  name: sample-cluster-workers-template
  namespace: default
spec:
  clusterName: sample-cluster
  metaData:
    objectNames:
    - key: name
      object: machine
    - key: local-hostname
      object: machine
    - key: local_hostname
      object: machine
```

When the example above has been copied and adapted to suit your environment, it can be applied via `kubectl` then the cluster status can be monitored with `clusterctl`

```
% kubectl apply -f rke2-agent.yaml

# Wait for the worker nodes to be provisioned
% clusterctl describe cluster sample-cluster
NAME                                                   READY  SEVERITY  REASON  SINCE
 MESSAGE
Cluster/sample-cluster                                 True                     25m
```

```
├─ClusterInfrastructure - Metal3Cluster/sample-cluster  True                    30m
├─ControlPlane - RKE2ControlPlane/sample-cluster        True                    25m
│ └─Machine/sample-cluster-chflc                        True                    27m
└─Workers
  └─MachineDeployment/sample-cluster                    True                    22m
    └─Machine/sample-cluster-56df5b4499-zfljj           True                    23m
```

4.4.9 Cluster deprovisioning

The downstream cluster may be deprovisioned by deleting the resources applied in the creation steps above:

```
% kubectl delete -f rke2-agent.yaml
% kubectl delete -f rke2-control-plane.yaml
```

This triggers deprovisioning of the BareMetalHost resources, which may take several minutes, after which they should be in available state again:

```
% kubectl get bmh
NAME             STATE            CONSUMER                           ONLINE   ERROR
 AGE
controlplane-0   deprovisioning   sample-cluster-controlplane-vlrt6   false
 10m
worker-0         deprovisioning   sample-cluster-workers-785x5        false
 10m

...

% kubectl get bmh
NAME             STATE       CONSUMER   ONLINE   ERROR   AGE
controlplane-0   available              false            15m
worker-0         available              false            15m
```

4.5 Known issues

- The upstream IP Address Management controller (https://github.com/metal3-io/ip-address-manager) is currently not supported, because it's not yet compatible with our choice of network configuration tooling and first-boot toolchain in SLEMicro.
- Relatedly, the IPAM resources and Metal3DataTemplate networkData fields are not currently supported.
- Only deployment via redfish-virtualmedia is currently supported.

4.6 Planned changes

- Enable support of the IPAM resources and configuration via networkData fields

4.7 Additional resources

The SUSE Telco Cloud Management Cluster Documentation (*Part V, "Setting up the management cluster"*) has examples of more advanced usage of Metal^3 for telco use-cases.

4.7.1 Single-node configuration

For test/PoC environments where the management cluster is a single node, it is possible to avoid the requirement for an additional floating IP managed via MetalLB.

In this mode, the endpoint for the management cluster APIs is the IP of the management cluster, therefore it should be reserved when using DHCP or statically configured to ensure the management cluster IP does not change - referred to as `<MANAGEMENT_CLUSTER_IP>` below.

To enable this scenario, the Metal^3 chart values required are as follows:

```
global:
  ironicIP: <MANAGEMENT_CLUSTER_IP>
metal3-ironic:
  service:
    type: NodePort
```

4.7.2 Disabling TLS for virtualmedia ISO attachment

Some server vendors verify the SSL connection when attaching virtual-media ISO images to the BMC, which can cause a problem because the generated certificates for the Metal^3 deployment are self-signed, to work around this issue it's possible to disable TLS only for the virtualmedia disk attachment with Metal^3 chart values as follows:

```
global:
  enable_vmedia_tls: false
```

An alternative solution is to configure the BMCs with the CA cert - in this case you can read the certificates from the cluster using `kubectl`:

```
kubectl get secret -n metal3-system ironic-vmedia-cert -o yaml
```

The certificate can then be configured on the server BMC console, although the process for that is vendor specific (and not possible for all vendors, in which case the `enable_vmedia_tls` flag may be required).

4.7.3 Storage configuration

For test/PoC environments where the management cluster is a single node, no persistent storage is required, but for production use-cases it is recommended to install SUSE Storage (Longhorn) on the management cluster so that images related to Metal^3 can be persisted during a pod restart/reschedule.

To enable this persistent storage, the Metal^3 chart values required are as follows:

```
metal3-ironic:
  persistence:
    ironic:
      size: "5Gi"
```

The SUSE Telco Cloud Management Cluster Documentation (*Part V, "Setting up the management cluster"*) has more details on how to configure a management cluster with persistent storage.

5 Standalone clusters with Edge Image Builder

Edge Image Builder (EIB) is a tool that streamlines the process of generating Customized, Ready-to-Boot (CRB) disk images for bootstrapping machines, even in fully air-gapped scenarios. EIB is used to create deployment images for use in all three of the SUSE Telco Cloud deployment footprints, as it's flexible enough to offer the smallest customizations, e.g. adding a user or setting the timezone, through offering a comprehensively configured image that sets up, for example, complex networking configurations, deploys multi-node Kubernetes clusters, deploys customer workloads, and registers to the centralized management platform via Rancher/Elemental and SUSE Multi-Linux Manager. EIB runs as in a container image, making it incredibly portable across platforms and ensuring that all of the required dependencies are self-contained, having a very minimal impact on the installed packages of the system that's being used to operate the tool.

Note

For multi-node scenarios, EIB automatically deploys MetalLB and Endpoint Copier Operator in order for hosts provisioned using the same built image to automatically join a Kubernetes cluster.

For more information, read the Edge Image Builder Introduction (*Chapter 12, Edge Image Builder*).

Warning

Edge Image Builder 1.3.4 supports customizing SUSE Linux Micro 6.2 images. Older versions, such as SUSE Linux Enterprise Micro 5.5, or 6.0 are not supported.

5.1 Prerequisites

- An AMD64/Intel 64 build host machine (physical or virtual) running SLES 15 SP6.
- The Podman container engine
- A SUSE Linux Micro 6.2 SelfInstall ISO image created using the Kiwi Builder procedure (*Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi*)

Note

For non-production purposes, openSUSE Leap 15.6, or openSUSE Tumbleweed may be used as a build host machine. Other operating systems may function, so long as a compatible container runtime is available.

5.1.1 Getting the EIB Image

The EIB container image is publicly available and can be downloaded from the SUSE Telco Cloud registry by running the following command on your image build host:

```
podman pull registry.suse.com/edge/3.7/edge-image-builder:1.3.4
```

5.2 Creating the image configuration directory

As EIB runs within a container, we need to mount a configuration directory from the host, enabling you to specify your desired configuration, and during the build process EIB has access to any required input files and supporting artifacts. This directory must follow a specific structure. Let’s create it, assuming that this directory will exist in your home directory, and called "eib":

```
export CONFIG_DIR=$HOME/eib
mkdir -p $CONFIG_DIR/base-images
```

In the previous step we created a "base-images" directory that will host the SUSE Linux Micro 6.2 input image, let’s ensure that the image is copied over to the configuration directory:

```
cp /path/to/SL-Micro.x86_64-6.2-Base-SelfInstall-GM.install.iso $CONFIG_DIR/base-images/
slemicro.iso
```

Note

During the EIB run, the original base image is **not** modified; a new and customized version is created with the desired configuration in the root of the EIB config directory.

The configuration directory at this point should look like the following:

```
└── base-images/
    └── slemicro.iso
```

5.3 Creating the image definition file

The definition file describes the majority of configurable options that the Edge Image Builder supports, a full example of options can be found here (https://github.com/suse-edge/edge-image-builder/blob/release-1.3/pkg/image/testdata/full-valid-example.yaml), and we would recommend that you take a look at the upstream building images guide (https://github.com/suse-edge/edge-image-builder/blob/release-1.3/docs/building-images.md) for more comprehensive examples than the one we're going to run through below. Let's start with a very basic definition file for our OS image:

```
cat << EOF > $CONFIG_DIR/iso-definition.yaml
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
EOF
```

This definition specifies that we are generating an output image for an AMD64/Intel 64 based system. The image that will be used as the base for further modification is an `iso` image named `slemicro.iso`, expected to be located at `$CONFIG_DIR/base-images/slemicro.iso`. It also outlines that after EIB finishes modifying the image, the output image will be named `eib-image.iso`, and by default will reside in `$CONFIG_DIR`.

Now our directory structure should look like:

```
├── iso-definition.yaml
└── base-images/
    └── slemicro.iso
```

In the following sections we'll walk through a few examples of common operations:

5.3.1 Configuring Operating System (OS)

The EIB `operatingSystem` section is intended to configure where the operating system is going to be installed, the image size, etc. It is an optional section and should not be included unless one or more customizations are being applied.

```
apiVersion: 1.3
image:
  imageType: iso
```

```
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-Base-RT-SelfInstall.iso
operatingSystem:
  isoConfiguration:
    installDevice: /dev/disk/by-id/ata-QEMU_HARDDISK_111-disk1 # first defined disk
```

Type-specific Configuration. Depending on the type of image being customized, one of the following optional sections may be included.

- `isoConfiguration` - Optional; configuration in this section only applies to ISO images.
- `installDevice` - Optional; specifies the disk that should be used as the install device. This needs to be a block device, and will default to automatically wipe any data found on the disk. Additionally, specifying this attribute triggers a GRUB override to automatically install the operating system rather than prompting user to begin the installation, allowing for a fully unattended and automated installation. If omitted, the user is prompted to select the "Install" option from the GRUB menu, as well as having to select the installation disk and confirm that the device will be wiped in the process.

> **Note**
>
> The device being used on the `installDevice` section can be specified as `/dev/sda` or using the `/dev/disk/by-id`, `/dev/disk/by-path` naming to ensure the proper device is being used. If using libvirt VMs, the `serial` attribute value can be specified when creating a disk for the VM (e.g., `serial=111-disk1`) so it can be used on the `installDevice` value with the `by-id` naming as for example `/dev/disk/by-id/ata-QEMU_HARDDISK_111-disk1` if using ATA devices (`libvirt` automatically prefixes the ID with `ata-QEMU_HARDDISK_` for ATA devices, or `virtio-` for virtio devices, see #17670 virtio issue (https://github.com/systemd/systemd/issues/17670#issuecomment-731261739) for more information).

- `rawConfiguration` - Optional; configuration in this section only applies to RAW images.
- `diskSize` - Optional; sets the desired raw disk image size that EIB will resize the resulting image to. This is important to ensure that your disk image is large enough to accommodate any artifacts being embedded in the image. It is advised to set this to slightly smaller than your SD card size (or block device if writing directly to a disk) as the system will automatically expand at boot time to fill the size of the block device. This is optional, but highly recommended. Specify as an integer with either "M" (Megabyte), "G" (Gigabyte), or "T" (Terabyte) as a suffix (e.g. "32G").

- `luksKey` - Required for encrypted images; the given LUKS key for an encrypted raw image which is necessary for EIB to be able to complete the build process.
- `expandEncryptedPartition` - Optional; disabled by default, when enabled, automatically expands the encrypted partition to its maximum size. E.g. if `diskSize` is `25G` and this field is `true`, EIB will expand the encrypted partition to `25G` during the build process.

5.3.2 Configuring OS Users

EIB allows you to preconfigure users with login information, such as passwords or SSH keys, including setting a fixed root password. As part of this example we're going to fix the root password, and the first step is to use `OpenSSL` to create a one-way encrypted password:

```
openssl passwd -6 SecurePassword
```

This will output something similar to:

```
$6$G392FCbxVgn[...]Y7zTXnC1
```

We can then add a section in the definition file called `operatingSystem` with a `users` array inside it. The resulting file should look like:

```
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$G392FCbxVgn[...]Y7zTXnC1
```

Note

It's also possible to add additional users, create the home directories, set user-id's, add ssh-key authentication, and modify group information. Please refer to the upstream building images guide (https://github.com/suse-edge/edge-image-builder/blob/release-1.3/docs/building-images.md) for further examples.

5.3.3 Configuring OS time

The `time` section is optional but it is highly recommended to be configured to avoid potential issues with certificates and clock skew. EIB will configure chronyd and `/etc/localtime` depending on the parameters here.

```
operatingSystem:
  time:
    timezone: Europe/London
    ntp:
      forceWait: true
      pools:
        - 2.suse.pool.ntp.org
      servers:
        - 10.0.0.1
        - 10.0.0.2
```

- The `timezone` specifies the timezone in the format of "Region/Locality" (e.g. "Europe/London"). The full list may be found by running `timedatectl list-timezones` on a Linux system.
- ntp - Defines attributes related to configuring NTP (using chronyd):
- forceWait - Requests that chronyd attempts to synchronize timesources before starting other services, with a 180s timeout.
- pools - Specifies a list of pools that chronyd will use as data sources (using `iburst` to improve the time taken for initial synchronization).
- servers - Specifies a list of servers that chronyd will use as data sources (using `iburst` to improve the time taken for initial synchronization).

Note

The values provided in this example are for illustrative purposes only. Please adjust them to fit your specific requirements.

5.3.4 Adding certificates

Certificate files with the extension ".pem" or ".crt" stored in the `certificates` directory will be installed in the node system-wide certificate store:

```
.
```

```
├── definition.yaml
└── certificates
    ├── my-ca.pem
    └── my-ca.crt
```

See the "Securing Communication with TLS Certificate" guide (https://documentation.suse.com/smart/security/html/tls-certificates/index.html#tls-adding-new-certificates) for more information.

5.3.5 Adding Operating System Files

The files placed in the `os-files` directory in the image configuration directory are automatically copied into the filesystem of the built image. The exact directory will be retained when they are copied. For example, if a file exists in a subdirectory named `os-files/etc`, it is placed in the `/etc` directory of the built image.

Note

If the `os-files` directory exists, it cannot be empty.

```
.
├── definition.yaml
└── os-files
    └── etc
        └── ssh
            └── sshd_config
```

5.3.6 Configuring RPM packages

One of the major features of EIB is to provide a mechanism to add additional software packages to the image, so when the installation completes the system is able to leverage the installed packages right away. EIB permits users to specify the following:

- Packages by their name within a list in the image definition
- Network repositories to search for these packages in
- SUSE Customer Center (SCC) credentials to search official SUSE repositories for the listed packages

- Via an `$CONFIG_DIR/rpms` directory, side-load custom RPM's that don't exist in network repositories
- Via the same directory (`$CONFIG_DIR/rpms/gpg-keys`), GPG-keys to enable validation of third party packages

EIB will then run through a package resolution process at image build time, taking the base image as the input, and attempts to pull and install all supplied packages, either specified via the list or provided locally. EIB downloads all of the packages, including any dependencies into a repository that exists within the output image and instructs the system to install these during the first boot process. Doing this process during the image build guarantees that the packages will successfully install during first-boot on the desired platform, e.g. the node at the edge. This is also advantageous in environments where you want to bake the additional packages into the image rather than pull them over the network when in operation, e.g. for air-gapped or restricted network environments.

As a simple example to demonstrate this, we are going to install the `nvidia-container-toolkit` RPM package found in the third party vendor-supported NVIDIA repository:

```
  packages:
    packageList:
      - nvidia-container-toolkit
    additionalRepos:
      - url: https://nvidia.github.io/libnvidia-container/stable/rpm/x86_64
```

The resulting definition file looks like the following:

```
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$G392FCbxVgn[...]Y7zTXnC1
  packages:
    packageList:
      - nvidia-container-toolkit
    additionalRepos:
      - url: https://nvidia.github.io/libnvidia-container/stable/rpm/x86_64
```

The above is a simple example, but for completeness, download the NVIDIA package signing key before running the image generation:

```
$ mkdir -p $CONFIG_DIR/rpms/gpg-keys
$ curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey > $CONFIG_DIR/rpms/gpg-keys/nvidia.gpg
```

Warning

Adding in additional RPM's via this method is meant for the addition of supported third party components or user-supplied (and maintained) packages; this mechanism should not be used to add packages that would not usually be supported on SUSE Linux Micro. If this mechanism is used to add components from openSUSE repositories (which are not supported), including from newer releases or service packs, you may end up with an unsupported configuration, especially when dependency resolution results in core parts of the operating system being replaced, even though the resulting system may appear to function as expected. If you're unsure, contact your SUSE representative for assistance in determining the supportability of your desired configuration.

Note

A more comprehensive guide with additional examples can be found in the upstream installing packages guide (https://github.com/suse-edge/edge-image-builder/blob/release-1.3/docs/installing-packages.md).

5.3.7 Configuring Kubernetes cluster and user workloads

Another feature of EIB is the ability to use it to automate the deployment of both single-node and multi-node highly-available Kubernetes clusters that "bootstrap in place" (i.e. don't require any form of centralized management infrastructure to coordinate). The primary driver behind this approach is for air-gapped deployments, or network restricted environments, but it also serves as a way of quickly bootstrapping standalone clusters, even if full and unrestricted network access is available.

This method enables not only the deployment of the customized operating system, but also the ability to specify Kubernetes configuration, any additional layered components via Helm charts, and any user workloads via supplied Kubernetes manifests. However, the design principle behind using this method is that we default to assuming that the user is wanting to air-gap. Therefore, any items specified in the

image definition will be pulled into the image, which includes user-supplied workloads. EIB ensures that any discovered images that are required by definitions are copied locally and are served by the embedded image registry in the resulting deployed system.

In this next example, we're going to take our existing image definition and will specify a Kubernetes configuration (in this example it doesn't list the systems and their roles, so we default to assuming single-node), which will instruct EIB to provision a single-node RKE2 Kubernetes cluster. To show the automation of both the deployment of both user-supplied workloads (via manifest) and layered components (via Helm), we are going to install KubeVirt via the SUSE Telco Cloud Helm chart, as well as NGINX via a Kubernetes manifest. The additional configuration we need to append to the existing image definition is as follows:

```
kubernetes:
  version: v1.36.3+rke2r1
  manifests:
    urls:
      - https://k8s.io/examples/application/nginx-app.yaml
  helm:
    charts:
      - name: kubevirt
        version: 307.0.3+up0.8.0
        repositoryName: suse-edge
    repositories:
      - name: suse-edge
        url: oci://registry.suse.com/edge/charts
```

The resulting full definition file should now look like:

```
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$G392FCbxVgn[...]Y7zTXnC1
  packages:
    packageList:
      - nvidia-container-toolkit
    additionalRepos:
      - url: https://nvidia.github.io/libnvidia-container/stable/rpm/x86_64
kubernetes:
  version: v1.36.3+k3s1
  manifests:
```

```
    urls:
      - https://k8s.io/examples/application/nginx-app.yaml
  helm:
    charts:
      - name: kubevirt
        version: 307.0.3+up0.8.0
        repositoryName: suse-edge
    repositories:
      - name: suse-edge
        url: oci://registry.suse.com/edge/charts
```

Note

Further examples of options such as multi-node deployments, custom networking, and Helm chart options/values can be found in the upstream documentation (https://github.com/suse-edge/edge-image-builder/blob/release-1.3/docs/building-images.md).

5.3.8 Configuring the network

In the last example in this quickstart, let's configure the network that will be brought up when a system is provisioned with the image generated by EIB. It's important to understand that unless a network configuration is supplied, the default model is that DHCP will be used on all interfaces discovered at boot time. However, this is not always a desirable configuration, especially if DHCP is not available and you need to provide static configurations, or you need to set up more complex networking constructs, e.g. bonds, LACP, and VLAN's, or need to override certain parameters, e.g. hostnames, DNS servers, and routes.

EIB provides the ability to provide either per-node configurations (where the system in question is uniquely identified by its MAC address), or an override for supplying an identical configuration to each machine, which is more useful when the system MAC addresses aren't known. An additional tool is used by EIB called Network Manager Configurator, or `nmc` for short, which is a tool built by the SUSE Telco Cloud team to allow custom networking configurations to be applied based on the nmstate.io (https://nmstate.io/) declarative network schema, and at boot time will identify the node it's booting on and will apply the desired network configuration prior to any services coming up.

We'll now apply a static network configuration for a system with a single interface by describing the desired network state in a node-specific file (based on the desired hostname) in the required `network` directory:

```
mkdir $CONFIG_DIR/network

cat << EOF > $CONFIG_DIR/network/host1.local.yaml
```

```
routes:
  config:
  - destination: 0.0.0.0/0
    metric: 100
    next-hop-address: 192.168.122.1
    next-hop-interface: eth0
    table-id: 254
  - destination: 192.168.122.0/24
    metric: 100
    next-hop-address: 192.168.122.1
    next-hop-interface: eth0
    table-id: 254
dns-resolver:
  config:
    server:
    - 192.168.122.1
    - 8.8.8.8
interfaces:
- name: eth0
  type: ethernet
  state: up
  mac-address: 34:8A:B1:4B:16:E7
  ipv4:
    address:
    - ip: 192.168.122.50
      prefix-length: 24
    dhcp: false
    enabled: true
  ipv6:
    enabled: false
EOF
```

Warning

The above example is set up for the default `192.168.122.0/24` subnet assuming that testing is being executed on a virtual machine, please adapt to suit your environment, not forgetting the MAC address. As the same image can be used to provision multiple nodes, networking configured by EIB (via `nmc`) is dependent on it being able to uniquely identify the node by its MAC address, and hence during boot `nmc` will apply the correct networking configuration to each machine. This means that you'll need to know the MAC addresses of the systems you want to install onto. Alternatively, the default behavior is to rely on DHCP, but you can utilize the `configure-network.sh` hook to apply a common configuration to all nodes - see the networking guide (*Chapter 13, Edge Networking*) for further details.

The resulting file structure should look like:

```
├── iso-definition.yaml
├── base-images/
│   └── slemicro.iso
└── network/
    └── host1.local.yaml
```

The network configuration we just created will be parsed and the necessary NetworkManager connection files will be automatically generated and inserted into the new installation image that EIB will create. These files will be applied during the provisioning of the host, resulting in a complete network configuration.

Note

Please refer to the Edge Networking component (*Chapter 13, Edge Networking*) for a more comprehensive explanation of the above configuration and examples of this feature.

5.4 Building the image

Now that we've got a base image and an image definition for EIB to consume, let's go ahead and build the image. For this, we simply use `podman` to call the EIB container with the "build" command, specifying the definition file:

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
build --definition-file iso-definition.yaml
```

The output of the command should be similar to:

```
Setting up Podman API listener...
Downloading file: dl-manifest-1.yaml 100% (498/498 B, 9.5 MB/s)
Pulling selected Helm charts... 100% (1/1, 43 it/min)
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Resolving package dependencies...
Rpm .......................... [SUCCESS]
Os Files ..................... [SKIPPED]
```

```
Systemd ...................... [SKIPPED]
Fips ......................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Populating Embedded Artifact Registry... 100% (3/3, 10 it/min)
Embedded Artifact Registry ... [SUCCESS]
Keymap ....................... [SUCCESS]
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Downloading file: rke2-images-core.linux-amd64.tar.zst 100% (657/657 MB, 48 MB/s)
Downloading file: rke2-images-cilium.linux-amd64.tar.zst 100% (368/368 MB, 48 MB/s)
Downloading file: rke2.linux-amd64.tar.gz 100% (35/35 MB, 50 MB/s)
Downloading file: sha256sum-amd64.txt 100% (4.3/4.3 kB, 6.2 MB/s)
Kubernetes ................... [SUCCESS]
Certificates ................. [SKIPPED]
Cleanup ...................... [SKIPPED]
Building ISO image...
Kernel Params ................ [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

The built ISO image is stored at `$CONFIG_DIR/eib-image.iso`:

```
├── iso-definition.yaml
├── eib-image.iso
├── _build
│   └── cache/
│       └── ...
│   └── build-<timestamp>/
│       └── ...
├── base-images/
│   └── slemicro.iso
└── network/
    └── host1.local.yaml
```

Each build creates a time-stamped folder in `$CONFIG_DIR/_build/` that includes the logs of the build, the artifacts used during the build, and the `combustion` and `artefacts` directories which contain all the scripts and artifacts that are added to the CRB image.

The contents of this directory should look like:

```
├── build-<timestamp>/
│   │── combustion/
│   │   ├── 05-configure-network.sh
│   │   ├── 10-rpm-install.sh
│   │   ├── 12-keymap-setup.sh
│   │   ├── 13b-add-users.sh
│   │   ├── 20-k8s-install.sh
```

```
│   │   ├── 26-embedded-registry.sh
│   │   ├── 48-message.sh
│   │   ├── network/
│   │   │   ├── host1.local/
│   │   │   │   └── eth0.nmconnection
│   │   │   └── host_config.yaml
│   │   ├── nmc
│   │   └── script
│   ├── artefacts/
│   │   ├── registry/
│   │   │   ├── hauler
│   │   │   ├── nginx:<version>-registry.tar.zst
│   │   │   ├── rancher_kubectl:<version>-registry.tar.zst
│   │   │   └── registry.suse.com_suse_sles_15.6_virt-operator:<version>-registry.tar.zst
│   │   ├── rpms/
│   │   │   └── rpm-repo
│   │   │       ├── addrepo0
│   │   │       │   ├── nvidia-container-toolkit-<version>.rpm
│   │   │       │   ├── nvidia-container-toolkit-base-<version>.rpm
│   │   │       │   ├── libnvidia-container1-<version>.rpm
│   │   │       │   └── libnvidia-container-tools-<version>.rpm
│   │   │       ├── repodata
│   │   │       │   ├── ...
│   │   │       └── zypper-success
│   │   └── kubernetes/
│   │       ├── rke2_installer.sh
│   │       ├── registries.yaml
│   │       ├── server.yaml
│   │       ├── images/
│   │       │   ├── rke2-images-cilium.linux-amd64.tar.zst
│   │       │   └── rke2-images-core.linux-amd64.tar.zst
│   │       ├── install/
│   │       │   ├── rke2.linux-amd64.tar.gz
│   │       │   └── sha256sum-amd64.txt
│   │       └── manifests/
│   │           ├── dl-manifest-1.yaml
│   │           └── kubevirt.yaml
│   ├── createrepo.log
│   ├── eib-build.log
│   ├── embedded-registry.log
│   ├── helm
│   │   └── kubevirt
│   │       └── kubevirt-0.4.0.tgz
│   ├── helm-pull.log
│   ├── helm-template.log
│   ├── iso-build.log
│   ├── iso-build.sh
```

```
│   ├── iso-extract
│   │   └── ...
│   ├── iso-extract.log
│   ├── iso-extract.sh
│   ├── modify-raw-image.sh
│   ├── network-config.log
│   ├── podman-image-build.log
│   ├── podman-system-service.log
│   ├── prepare-resolver-base-tarball-image.log
│   ├── prepare-resolver-base-tarball-image.sh
│   ├── raw-build.log
│   ├── raw-extract
│   │   └── ...
│   └── resolver-image-build
│       └──...
└── cache
    └── ...
```

If the build fails, `eib-build.log` is the first log that contains information. From there, it will direct you to the component that failed for debugging.

At this point, you should have a ready-to-use image that will:

1. Deploy SUSE Linux Micro 6.2
2. Configure the root password
3. Install the `nvidia-container-toolkit` package
4. Configure an embedded container registry to serve content locally
5. Install single-node RKE2
6. Configure static networking
7. Install KubeVirt
8. Deploy a user-supplied manifest

5.5 Debugging the image build process

If the image build process fails, refer to the upstream debugging guide (https://github.com/suse-edge/edge-image-builder/blob/release-1.3/docs/debugging.md).

5.6 Testing your newly built image

For instructions on how to test the newly built CRB image, refer to the upstream image testing guide (https://github.com/suse-edge/edge-image-builder/blob/release-1.3/docs/testing-guide.md).

III Components

6 Rancher 51

7 Rancher Dashboard Extensions 54

8 Rancher Turtles 59

9 Fleet 61

10 SUSE Linux Micro 67

11 Metal^3 69

12 Edge Image Builder 70

13 Edge Networking 72

14 RKE2 95

15 SUSE Storage 98

16 SUSE Security 107

17 MetalLB 109

18 Endpoint Copier Operator 111

19 Edge Virtualization 112

20 System Upgrade Controller 127

21 Upgrade Controller 136

List of components for Edge

6 Rancher

See Rancher documentation at https://ranchermanager.docs.rancher.com/v2.15.

> Rancher is a powerful open-source Kubernetes management platform that streamlines the deployment, operations and monitoring of Kubernetes clusters across multiple environments. Whether you manage clusters on premises, in the cloud, or at the edge, Rancher provides a unified and centralized platform for all your Kubernetes needs.

6.1 Key Features of Rancher

- **Multi-cluster management:** Rancher's intuitive interface lets you manage Kubernetes clusters from anywhere—public clouds, private data centers and edge locations.
- **Security and compliance:** Rancher enforces security policies, role-based access control (RBAC), and compliance standards across your Kubernetes landscape.
- **Simplified cluster operations:** Rancher automates cluster provisioning, upgrades and troubleshooting, simplifying Kubernetes operations for teams of all sizes.
- **Centralized application catalog:** The Rancher application catalog offers a diverse range of Helm charts and Kubernetes Operators, making it easy to deploy and manage containerized applications.
- **Continuous delivery:** Rancher supports GitOps and CI/CD pipelines, enabling automated and streamlined application delivery processes.

6.2 Rancher's use in SUSE Telco Cloud

Rancher provides several core functionalities to the SUSE Telco Cloud stack:

6.2.1 Centralized Kubernetes management

In typical edge deployments with numerous distributed clusters, Rancher acts as a central control plane for managing these Kubernetes clusters. It offers a unified interface for provisioning, upgrading, monitoring, and troubleshooting, simplifying operations, and ensuring consistency.

6.2.2 Simplified cluster deployment

Rancher streamlines Kubernetes cluster creation on the lightweight SUSE Linux Micro operating system, easing the rollout of edge infrastructure with robust Kubernetes capabilities.

6.2.3 Application deployment and management

The integrated Rancher application catalog can simplify deploying and managing containerized applications across SUSE Telco Cloud clusters, enabling seamless edge workload deployment.

6.2.4 Security and policy enforcement

Rancher provides policy-based governance tools, role-based access control (RBAC), and integration with external authentication providers. This helps SUSE Telco Cloud deployments maintain security and compliance, critical in distributed environments.

6.3 Best practices

6.3.1 GitOps

Rancher includes Fleet as a built-in component to allow manage cluster configurations and application deployments with code stored in git.

6.3.2 Observability

Rancher includes built-in monitoring and logging tools like Prometheus and Grafana for comprehensive insights into your cluster health and performance.

6.4 Installing with Edge Image Builder

SUSE Telco Cloud is using *Chapter 12, Edge Image Builder* in order to customize base SUSE Linux Micro OS images. Follow *Section 64.6, "Rancher Installation"* for an air-gapped installation of Rancher on top of Kubernetes clusters provisioned by EIB.

6.5 Additional Resources

- Rancher Documentation (https://rancher.com/docs/)
- Rancher Academy (https://www.rancher.academy/)
- Rancher Community (https://rancher.com/community/)
- Helm Charts (https://helm.sh/)
- Kubernetes Operators (https://operatorhub.io/)

7 Rancher Dashboard Extensions

Extensions allow users, developers, partners, and customers to extend and enhance the Rancher UI. SUSE Telco Cloud provides KubeVirt dashboard extensions.

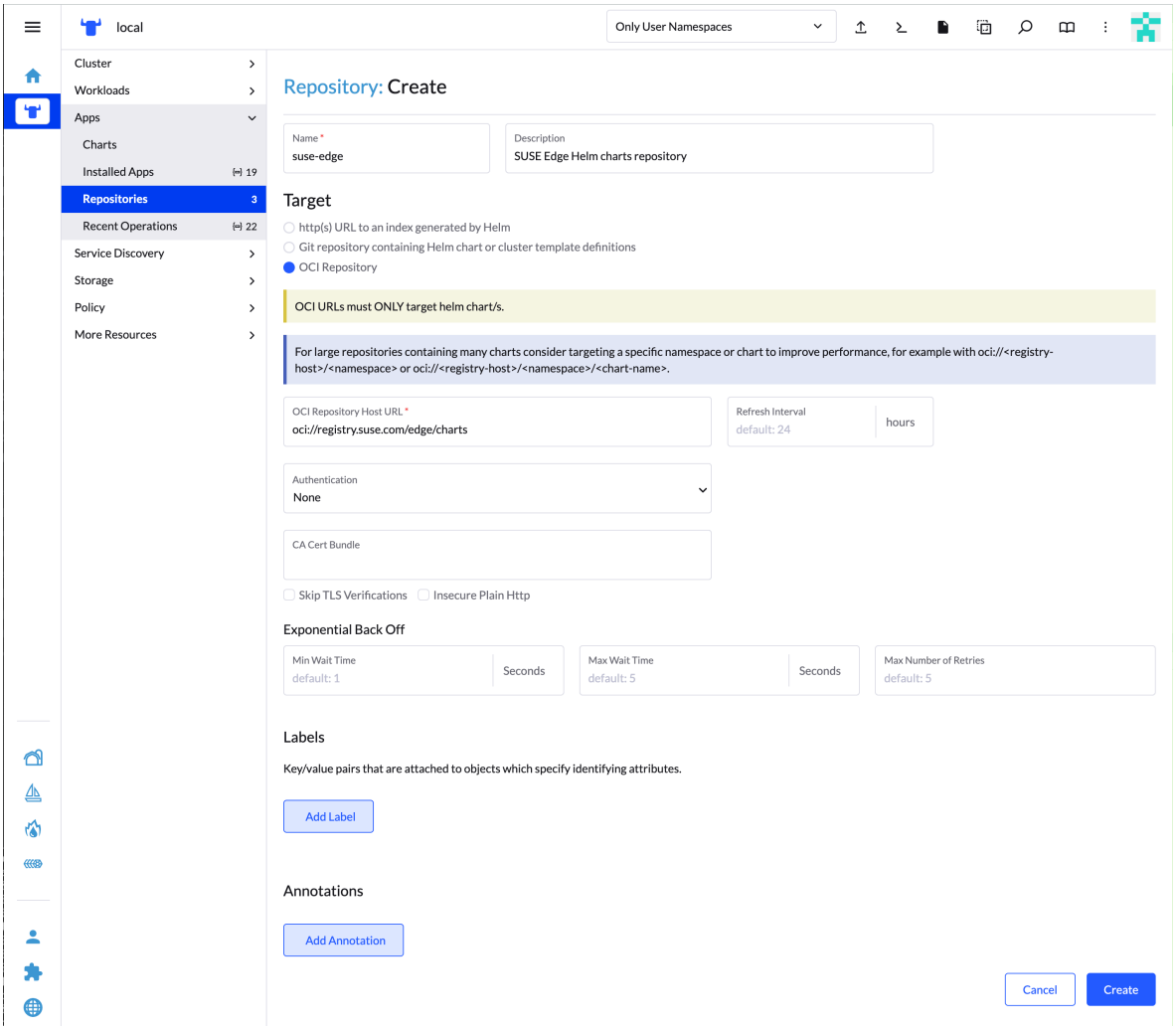
See `Rancher documentation` for general information about Rancher Dashboard Extensions.

7.1 Installation

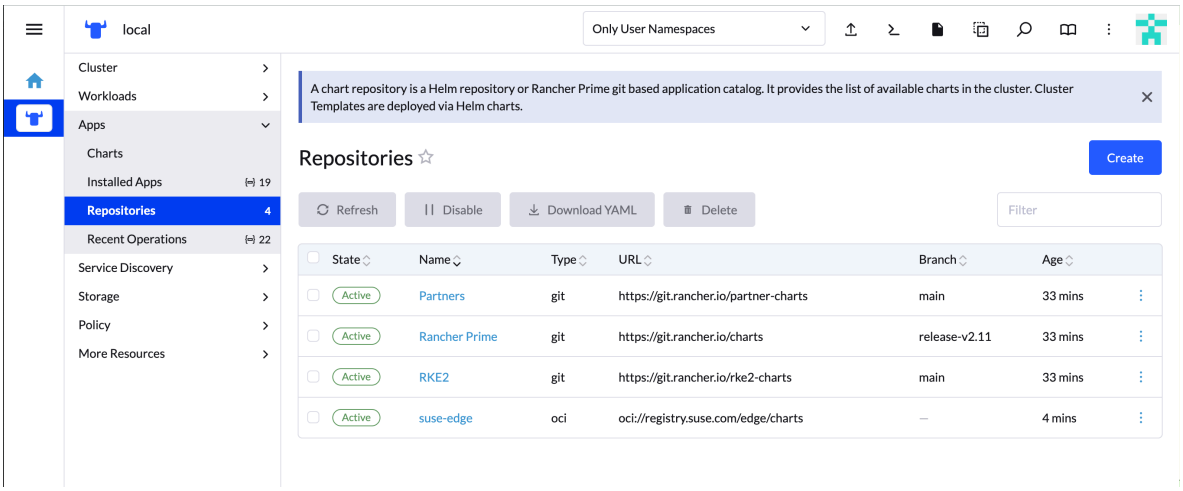
All of the SUSE Telco Cloud 3.7 components, including dashboard extensions, are distributed as OCI artifacts. To install SUSE Telco Cloud Extensions you can use Rancher Dashboard UI, Helm or Fleet:

7.1.1 Installing with Rancher Dashboard UI

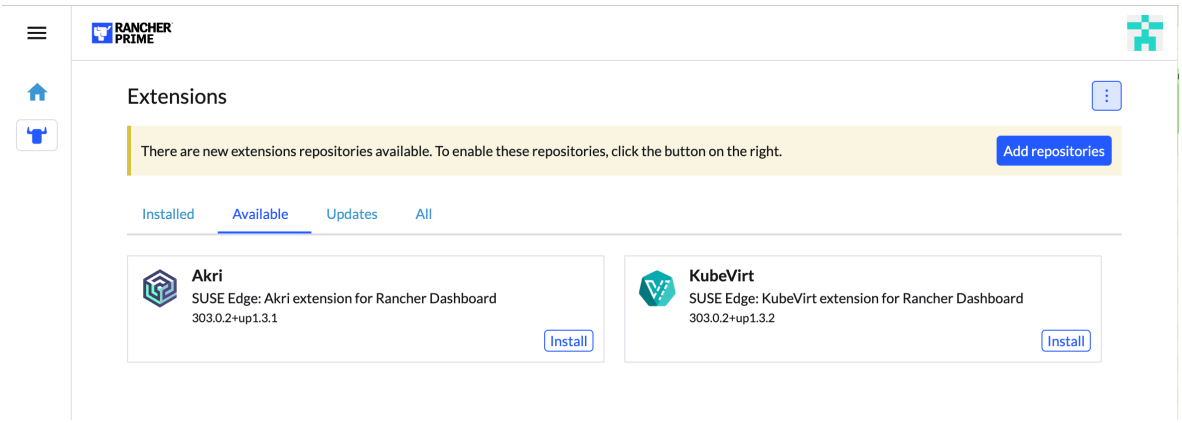
1. Click **Extensions** in the **Configuration** section of the navigation sidebar.
2. On the Extensions page, click the three dot menu at the top right and select **Manage Repositories**. Each extension is distributed via its own OCI artifact. They are available from the SUSE Telco Cloud Helm charts repository.
3. On the **Repositories page**, click `Create`.
4. In the form, specify the repository name and URL, and click `Create`.
   SUSE Telco Cloud Helm charts repository URL: `oci://registry.suse.com/edge/charts`



5. You can see that the extension repository is added to the list and is in `Active` state.



6. Navigate back to the **Extensions** in the **Configuration** section of the navigation sidebar.
   In the **Available** tab you can see the extensions available for installation.



7. On the extension card click `Install` and confirm the installation.
   Once the extension is installed Rancher UI prompts to reload the page as described in the `Installing Extensions Rancher documentation page`.

7.1.2 Installing with Helm

```
# KubeVirt extension
helm install kubevirt-dashboard-extension oci://registry.suse.com/edge/charts/kubevirt-dashboard-extension --version 307.0.5+up1.4.2 --namespace cattle-ui-plugin-system
```

Note

The extensions need to be installed in `cattle-ui-plugin-system` namespace.

Note

After an extension is installed, Rancher Dashboard UI needs to be reloaded.

7.1.3 Installing with Fleet

Installing Dashboard Extensions with Fleet requires defining a `gitRepo` resource which points to a Git repository with custom `fleet.yaml` bundle configuration file(s).

```
# KubeVirt extension fleet.yaml
```

```
defaultNamespace: cattle-ui-plugin-system
helm:
  releaseName: kubevirt-dashboard-extension
  chart: oci://registry.suse.com/edge/charts/kubevirt-dashboard-extension
  version: "307.0.5+up1.4.2"
```

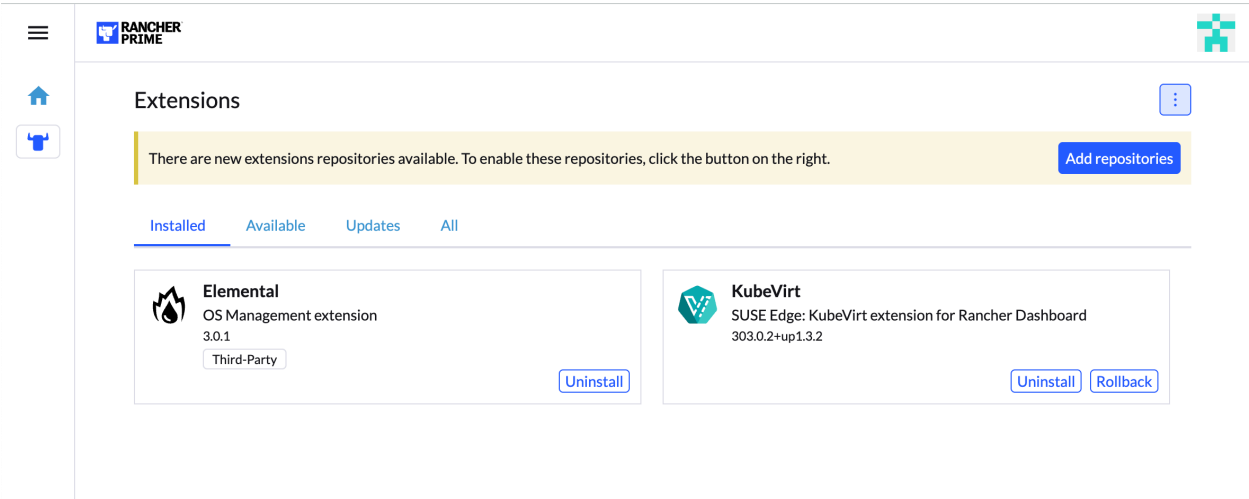
Note

The `releaseName` property is required and needs to match the extension name to get the extension correctly installed.

```
cat <<- EOF | kubectl apply -f -
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: edge-dashboard-extensions
  namespace: fleet-local
spec:
  repo: https://github.com/suse-edge/fleet-examples.git
  branch: main
  paths:
  - fleets/kubevirt-dashboard-extension/
EOF
```

For more information, see *Chapter 9, Fleet* and the `fleet-examples` repository.

Once the Extensions are installed they are listed in **Extensions** section under **Installed** tabs. Since they are not installed via Apps/Marketplace, they are marked with `Third-Party` label.



7.2 KubeVirt Dashboard Extension

KubeVirt Extension provides basic virtual machine management for Rancher dashboard UI. Its capabilities are described in *Section 19.7.2, "Using KubeVirt Rancher Dashboard Extension"*.

8 Rancher Turtles

See Rancher Turtles documentation at https://documentation.suse.com/cloudnative/cluster-api/

> Rancher Turtles is a Kubernetes Operator that provides integration between Rancher Manager and Cluster API (CAPI) with the aim of bringing full CAPI support to Rancher

8.1 Key Features of Rancher Turtles

- Automatically import CAPI clusters into Rancher, by installing the Rancher Cluster Agent in CAPI provisioned clusters.
- Install and configure CAPI controller dependencies via the CAPI Operator (https://cluster-api-operator.sigs.k8s.io/).
- Manage installed CAPI provider dependencies via the `CAPIProvider` API

8.2 Rancher Turtles use in SUSE Telco Cloud

The SUSE Telco Cloud stack provides a `rancher-turtles-providers` helm chart that enables certain CAPI providers via the `CAPIProvider` API:

- RKE2 Control Plane and Bootstrap provider
- Metal3 (*Chapter 11, Metal*[3]) infrastructure provider
- Metal3 (*Chapter 11, Metal*[3]) IPAM provider (currently not supported)
- Fleet addon provider

Only the default providers installed via the wrapper chart are supported - alternative Control Plane, Bootstrap and Infrastructure providers are not currently supported as part of the SUSE Telco Cloud stack.

8.3 Installing Rancher Turtles

Since Rancher 2.13, Rancher Turtles is enabled by default when installing Rancher.

Rancher Turtles Providers may be installed by following the Metal3 Quickstart (*Chapter 4, BMC automated deployments with Metal*[3]) guide, or the Management Cluster (*Part V, "Setting up the management cluster"*) documentation.

8.4 Additional Resources

- Rancher Documentation (https://rancher.com/docs/)
- Cluster API Book (https://cluster-api.sigs.k8s.io/)

9 Fleet

Fleet (https://fleet.rancher.io) is a container management and deployment engine designed to offer users more control on the local cluster and constant monitoring through GitOps. Fleet focuses not only on the ability to scale, but it also gives users a high degree of control and visibility to monitor exactly what is installed on the cluster.

Fleet can manage deployments from Git of raw Kubernetes YAML, Helm charts, Kustomize, or any combination of the three. Regardless of the source, all resources are dynamically turned into Helm charts, and Helm is used as the engine to deploy all resources in the cluster. As a result, users can enjoy a high degree of control, consistency and auditability of their clusters.

For information about how Fleet works, see Fleet Architecture (https://ranchermanager.docs.rancher.com/v2.15/integrations-in-rancher/fleet/architecture).

9.1 Installing Fleet with Helm

Fleet comes built-in to Rancher, but it can be also installed (https://fleet.rancher.io/installation) as a standalone application on any Kubernetes cluster using Helm.

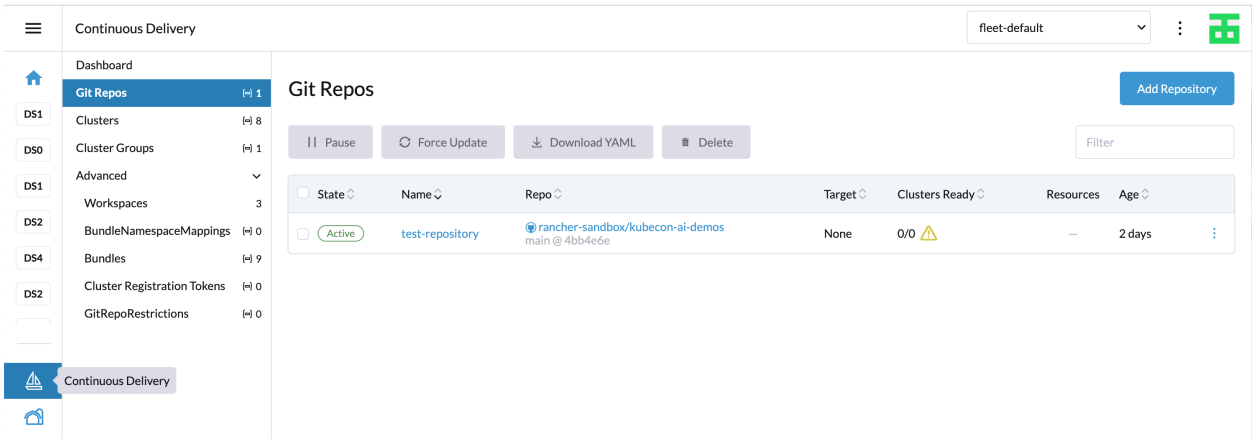
9.2 Using Fleet with Rancher

Rancher uses Fleet to deploy applications across managed clusters. Continuous delivery with Fleet introduces GitOps at scale, designed to manage applications running on large numbers of clusters.

Fleet shines as an integrated part of Rancher. Clusters managed with Rancher automatically get the Fleet agent deployed as part of the installation/import process and the cluster is immediately available to be managed by Fleet.

9.3 Accessing Fleet in the Rancher UI

Fleet comes preinstalled in Rancher and is managed by the **Continuous Delivery** option in the Rancher UI.



Continuous Delivery section consists of following items:

9.3.1 Dashboard

An overview page of all GitOps repositories across all workspaces. Only the workspaces with repositories are displayed.

9.3.2 Git repos

A list of GitOps repositories in the selected workspace. Select the active workspace using the dropdown list at the top of the page.

9.3.3 Clusters

A list of managed clusters. By default, all Rancher-managed clusters are added to the `fleet-default` workspace. `fleet-local` workspace includes the local (management) cluster. From here, it is possible to `Pause` or `Force update` the clusters or move the cluster into another workspace. Editing the cluster allows to update labels and annotations used for grouping the clusters.

9.3.4 Cluster groups

This section allows custom grouping of the clusters within the workspace using selectors.

9.3.5 Advanced

The "Advanced" section allows to manage workspaces and other related Fleet resources.

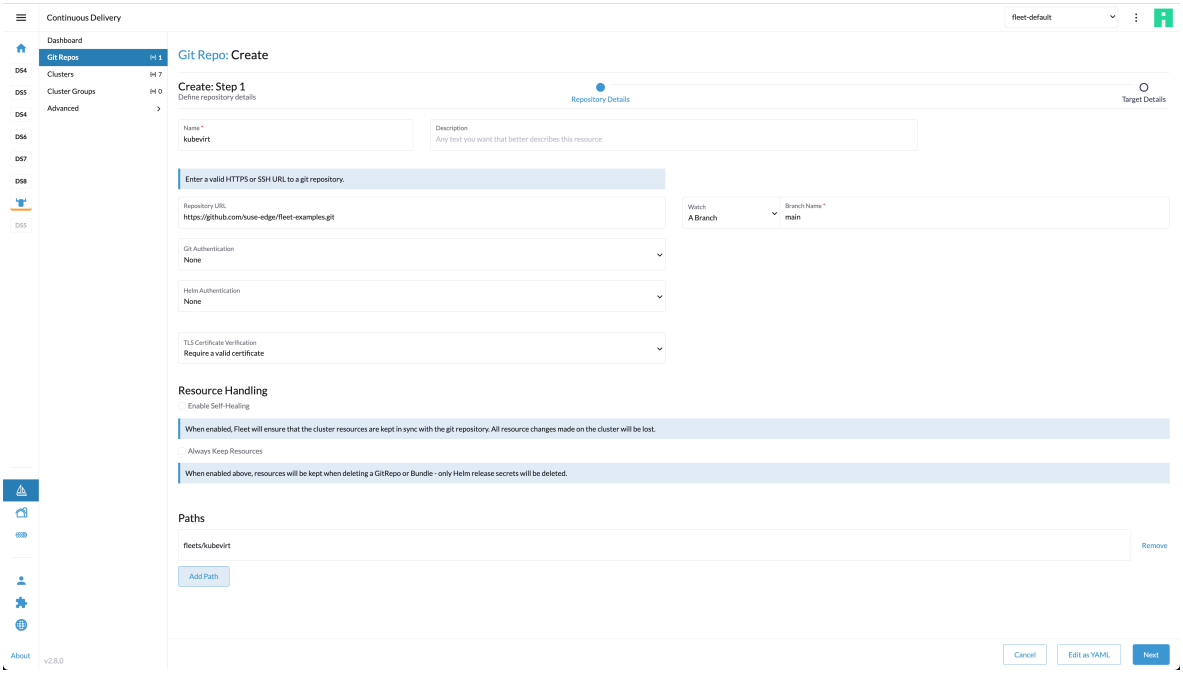
9.4 Example of installing KubeVirt with Rancher and Fleet using Rancher dashboard

1. Create a Git repository containing the `fleet.yaml` file:

```
defaultNamespace: kubevirt
helm:
  chart: "oci://registry.suse.com/edge/charts/kubevirt"
  version: "307.0.3+up0.8.0"
  # kubevirt namespace is created by kubevirt as well, we need to take ownership of
 it
  takeOwnership: true
```

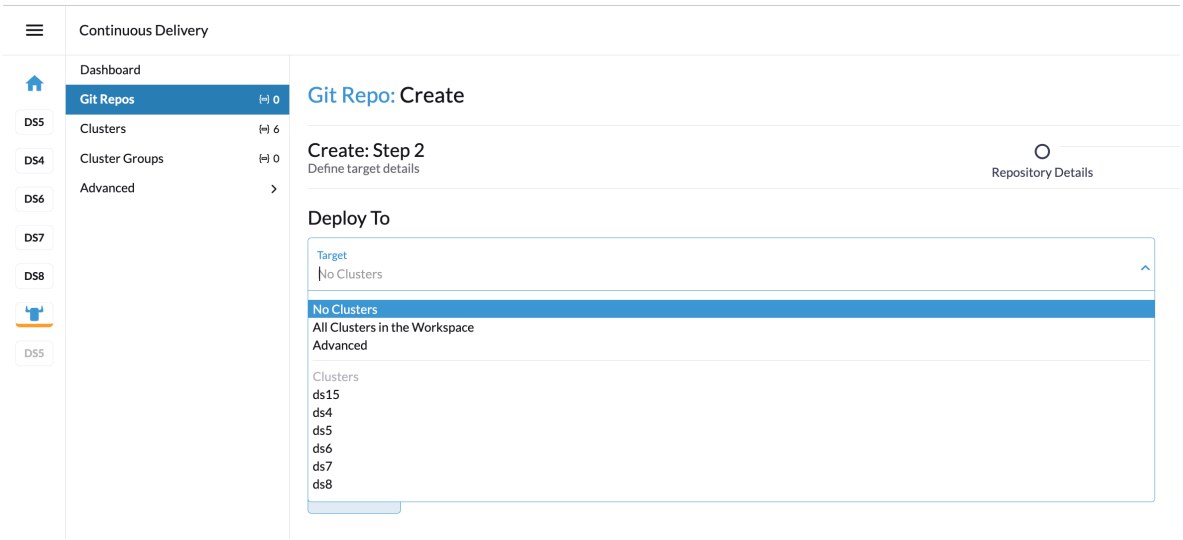
2. In the Rancher dashboard, navigate to # > **Continuous Delivery** > **Git Repos** and click `Add Repository`.

3. The Repository creation wizard guides through creation of the Git repo. Provide **Name**, **Repository URL** (referencing the Git repository created in the previous step) and select the appropriate branch or revision. In the case of a more complex repository, specify **Paths** to use multiple directories in a single repository.



4. Click `Next`.

5. In the next step, you can define where the workloads will get deployed. Cluster selection offers several basic options: you can select no clusters, all clusters, or directly choose a specific managed cluster or cluster group (if defined). The "Advanced" option allows to directly edit the selectors via YAML.

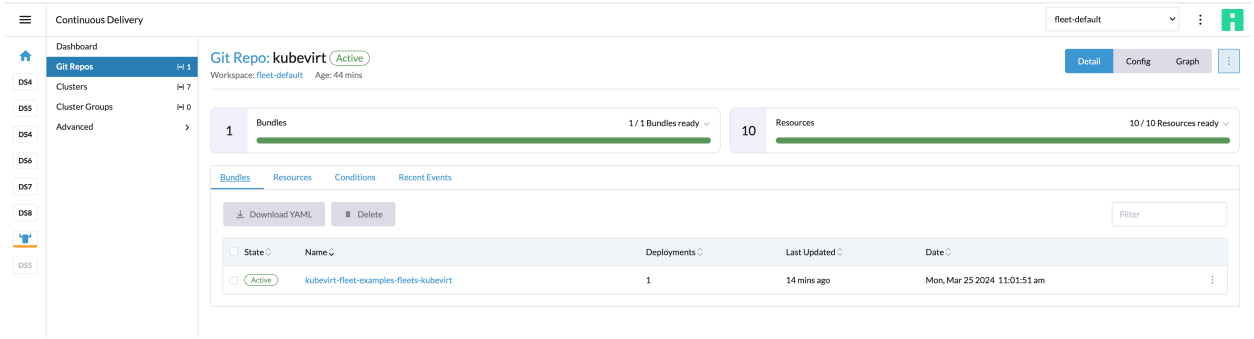


6. Click `Create`. The repository gets created. From now on, the workloads are installed and kept in sync on the clusters matching the repository definition.

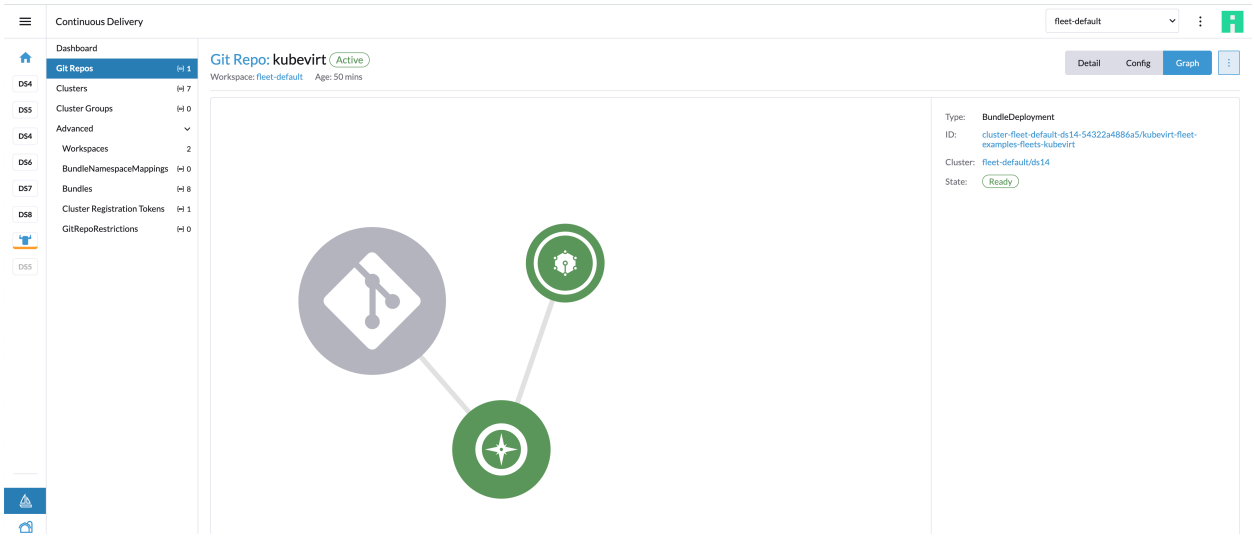
9.5 Debugging and troubleshooting

The "Advanced" navigation section provides overviews of lower-level Fleet resources. A bundle (https://fleet.rancher.io/ref-bundle-stages) is an internal resource used for the orchestration of resources from Git. When a Git repo is scanned, it produces one or more bundles.

To find bundles relevant to a specific repository, go to the Git repo detail page and click the `Bundles` tab.



For each cluster, the bundle is applied to a BundleDeployment resource that is created. To view BundleDeployment details, click the `Graph` button in the upper right of the Git repo detail page. A graph of **Repo > Bundles > BundleDeployments** is loaded. Click the BundleDeployment in the graph to see its details and click the `Id` to view the BundleDeployment YAML.



For additional information on Fleet troubleshooting tips, refer here (https://fleet.rancher.io/troubleshooting).

9.6 Fleet examples

The Edge team maintains a repository (https://github.com/suse-edge/fleet-examples) with examples of installing Edge projects with Fleet.

The Fleet project includes a fleet-examples (https://github.com/rancher/fleet-examples) repository that covers all use cases for Git repository structure (https://fleet.rancher.io/gitrepo-content).

10 SUSE Linux Micro

See SUSE Linux Micro official documentation (https://documentation.suse.com/sle-micro/6.2/)

> SUSE Linux Micro is a lightweight and secure operating system for the edge. It merges the enterprise-hardened components of SUSE Linux Enterprise with the features that developers want in a modern, immutable operating system. As a result, you get a reliable infrastructure platform with best-in-class compliance that is also simple to use.

10.1 How does SUSE Telco Cloud use SUSE Linux Micro?

We use SUSE Linux Micro as the base operating system for our platform stack. This provides us with a secure, stable and minimal base for building upon.

SUSE Linux Micro is unique in its use of file system (Btrfs) snapshots to allow for easy rollbacks in case something goes wrong with an upgrade. This allows for secure remote upgrades for the entire platform even without physical access in case of issues.

10.2 Best practices

10.2.1 Installation media

SUSE Telco Cloud uses the Edge Image Builder (*Chapter 12, Edge Image Builder*) to preconfigure the SUSE Linux Micro self-install installation image.

10.2.2 Local administration

SUSE Linux Micro comes with Cockpit to allow the local management of the host through a Web application.

This service is disabled by default but can be started by enabling the systemd service `cockpit.socket`. As cockpit forbids root login by default, the creation of a user with administrative privileges is recommended, refer to the SUSE Linux Micro official documentation (https://documentation.suse.com/sle-micro/6.2/) for more information.

10.3 Known issues

- There is no desktop environment available in SUSE Linux Micro at the moment but a containerized solution is in development.

11 Metal^3

Metal^3 (https://metal3.io/) is a CNCF project which provides bare-metal infrastructure management capabilities for Kubernetes.

Metal^3 provides Kubernetes-native resources to manage the lifecycle of bare-metal servers which support management via out-of-band protocols such as Redfish (https://www.dmtf.org/standards/redfish).

It also has mature support for Cluster API (CAPI) (https://cluster-api.sigs.k8s.io/). This enables management of hardware resources across multiple infrastructure providers via broadly adopted vendor-neutral APIs. Cluster API uses Metal^3 as an infrastructure backend for Machine objects.

11.1 How does SUSE Telco Cloud use Metal^3?

SUSE Telco Cloud uses Metal^3 to manage the lifecycle of physical hardware, such as servers. The hardware needs to support an out-of-band management protocol that is supported by Metal^3 (e.g. Redfish). When a SUSE Telco Cloud management cluster provisions or deprovisions downstream clusters, Metal^3 will interact with a server's BMC via Redfish. The following actions are typically part of this interaction:

- Mount and umount Virtual Media.
- Power on, power off, and reset servers.

This approach is useful for scenarios where the target hardware supports out-of-band management, and a fully automated infrastructure management flow is desired.

Metal^3 and CAPI provide declarative APIs that enable inventory and state management of bare-metal servers, including automated inspection, cleaning, and provisioning/deprovisioning.

11.2 Known issues

- The upstream IP Address Management controller (https://github.com/metal3-io/ip-address-manager) is currently not supported, because it is not yet compatible with our choice of network configuration tooling. However the `ipam-controller-manager` pod hosted on the `metal3-ipam-system` namespace is needed as CAPM3 requires the `ipam` CRDs to exist.
- Relatedly, the IPAM resources and Metal3DataTemplate networkData fields are not supported.
- Only deployment via redfish-virtualmedia is currently supported.

12 Edge Image Builder

See the Official Repository (https://github.com/suse-edge/edge-image-builder).

Edge Image Builder (EIB) is a tool that streamlines the generation of Customized, Ready-to-Boot (CRB) disk images for bootstrapping machines. These images enable the end-to-end deployment of the entire SUSE software stack with a single image.

Whilst EIB can create CRB images for all provisioning scenarios, EIB demonstrates a tremendous value in air-gapped deployments with limited or completely isolated networks.

12.1 How does SUSE Telco Cloud use Edge Image Builder?

SUSE Telco Cloud uses EIB for the simplified and quick configuration of customized SUSE Linux Micro images for a variety of scenarios. These scenarios include the bootstrapping of virtual and bare-metal machines with:

- Fully air-gapped deployments of K3s/RKE2 Kubernetes (single & multi-node)
- Fully air-gapped Helm chart and Kubernetes manifest deployments
- Registration to Rancher via Elemental API
- Metal[3]
- Customized networking (for example, static IP, host name, VLAN's, bonding, etc.)
- Customized operating system configurations (for example, users, groups, passwords, SSH keys, proxies, NTP, custom SSL certificates, etc.)
- Air-gapped installation of host-level and side-loaded RPM packages (including dependency resolution)
- Registration to SUSE Multi-Linux Manager for OS management
- Embedded container images
- Kernel command-line arguments
- Systemd units to be enabled/disabled at boot time
- Custom scripts and files for any manual tasks

12.2 Getting started

Comprehensive documentation for the usage and testing of Edge Image Builder can be found here (https://github.com/suse-edge/edge-image-builder/tree/release-1.3/docs).

Additionally, see *Chapter 5, Standalone clusters with Edge Image Builder* covering a basic deployment scenario.

Once you are familiar with this tool, please find some more useful information on our EIB Tips and Tricks section (*Part X, "Tips and Tricks"*) page.

12.3 Known issues

- EIB air-gaps Helm charts through templating the Helm charts and parsing all the images within the template. If a Helm chart does not keep all of its images within the template and instead side-loads the images, EIB will not be able to air-gap those images automatically. The solution to this is to manually add any undetected images to the `embeddedArtifactRegistry` section of the definition file.

13 Edge Networking

This section describes the approach to network configuration in the SUSE Telco Cloud solution. We will show how to configure NetworkManager on SUSE Linux Micro in a declarative manner, and explain how the related tools are integrated.

13.1 Overview of NetworkManager

NetworkManager is a tool that manages the primary network connection and other connection interfaces.

NetworkManager stores network configurations as connection files that contain the desired state. These connections are stored as files in the `/etc/NetworkManager/system-connections/` directory.

Details about NetworkManager can be found in the SUSE Linux Micro documentation (https://documentation.suse.com/sle-micro/6.2/html/Micro-network-configuration/index.html).

13.2 Overview of nmstate

nmstate is a widely adopted library (with an accompanying CLI tool) which offers a declarative API for network configurations via a predefined schema.

Details about nmstate can be found in the upstream documentation (https://nmstate.io/).

13.3 Enter: NetworkManager Configurator (nmc)

The network customization options available in SUSE Telco Cloud are achieved via a CLI tool called NetworkManager Configurator or *nmc* for short. It is leveraging the functionality provided by the nmstate library and, as such, it is fully capable of configuring static IP addresses, DNS servers, VLANs, bonding, bridges, etc. This tool allows us to generate network configurations from predefined desired states and to apply those across many different nodes in an automated fashion.

Details about the NetworkManager Configurator (nmc) can be found in the upstream repository (https://github.com/suse-edge/nm-configurator).

13.4 How does SUSE Telco Cloud use NetworkManager Configurator?

SUSE Telco Cloud utilizes *nmc* for the network customizations in the various different provisioning models: * Custom network configurations in the Directed Network Provisioning scenarios (*Chapter 4, BMC automated deployments with Metal*[3]) * Declarative static configurations in the Image Based Provisioning scenarios (*Chapter 5, Standalone clusters with Edge Image Builder*)

13.5 Configuring with Edge Image Builder

Edge Image Builder (EIB) is a tool which enables configuring multiple hosts with a single OS image. In this section we'll show how you can use a declarative approach to describe the desired network states, how those are converted to the respective NetworkManager connections, and are then applied during the provisioning process.

13.5.1 Prerequisites

If you're following this guide, it's assumed that you've got the following already available:

- An AMD64/Intel 64 physical host (or virtual machine) running SLES 15 SP6 or openSUSE Leap 15.6
- An available container runtime (e.g. Podman)
- A copy of the SUSE Linux Micro 6.2 RAW image found here (https://www.suse.com/download/sle-micro/)

13.5.2 Getting the Edge Image Builder container image

The EIB container image is publicly available and can be downloaded from the SUSE Telco Cloud registry by running:

```
podman pull registry.suse.com/edge/3.7/edge-image-builder:1.3.4
```

13.5.3 Creating the image configuration directory

Let's start with creating the configuration directory:

```
export CONFIG_DIR=$HOME/eib
mkdir -p $CONFIG_DIR/base-images
```

We will now ensure that the downloaded base image copy is moved over to the configuration directory:

```
mv /path/to/downloads/SL-Micro.x86_64-6.2-Base-GM.raw $CONFIG_DIR/base-images/
```

> **Note**
>
> EIB is never going to modify the base image input. It will create a new image with its modifications.

The configuration directory at this point should look like the following:

```
└── base-images/
    └── SL-Micro.x86_64-6.2-Base-GM.raw
```

13.5.4 Creating the image definition file

The definition file describes the majority of configurable options that the Edge Image Builder supports.

Let's start with a very basic definition file for our OS image:

```
cat << EOF > $CONFIG_DIR/definition.yaml
apiVersion: 1.3
image:
  arch: x86_64
  imageType: raw
  baseImage: SL-Micro.x86_64-6.2-Base-GM.raw
  outputImageName: modified-image.raw
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/zb4r8EmnrrNCF.P/
EOF
```

The `image` section is required, and it specifies the input image, its architecture and type, as well as what the output image will be called. The `operatingSystem` section is optional, and contains configuration to enable login on the provisioned systems with the `root/eib` username/password.

Note

Feel free to use your own encrypted password by running `openssl passwd -6 <password>`.

The configuration directory at this point should look like the following:

```
├── definition.yaml
└── base-images/
    └── SL-Micro.x86_64-6.2-Base-GM.raw
```

13.5.5 Defining the network configurations

The desired network configurations are not part of the image definition file that we just created. We'll now populate those under the special `network/` directory. Let's create it:

```
mkdir -p $CONFIG_DIR/network
```

As previously mentioned, the NetworkManager Configurator (*nmc*) tool expects an input in the form of predefined schema. You can find how to set up a wide variety of different networking options in the upstream NMState examples documentation (https://nmstate.io/examples.html).

This guide will explain how to configure the networking on three different nodes:

- A node which uses two Ethernet interfaces
- A node which uses network bonding
- A node which uses a network bridge

Warning

Using completely different network setups is not recommended in production builds, especially if configuring Kubernetes clusters. Networking configurations should generally be homogeneous amongst nodes or at least amongst roles within a given cluster. This guide is including various different options only to serve as an example reference.

Note

The following assumes a default `libvirt` network with an IP address range `192.168.122.1/24`. Adjust accordingly if this differs in your environment.

Let’s create the desired states for the first node which we will call `node1.suse.com`:

```
cat << EOF > $CONFIG_DIR/network/node1.suse.com.yaml
routes:
  config:
    - destination: 0.0.0.0/0
      metric: 100
      next-hop-address: 192.168.122.1
      next-hop-interface: eth0
      table-id: 254
    - destination: 192.168.122.0/24
      metric: 100
      next-hop-address: 192.168.122.1
      next-hop-interface: eth0
      table-id: 254
dns-resolver:
  config:
    server:
      - 192.168.122.1
      - 8.8.8.8
interfaces:
  - name: eth0
    type: ethernet
    state: up
    mac-address: 34:8A:B1:4B:16:E1
    ipv4:
      address:
        - ip: 192.168.122.50
          prefix-length: 24
      dhcp: false
      enabled: true
    ipv6:
      enabled: false
  - name: eth3
    type: ethernet
    state: down
    mac-address: 34:8A:B1:4B:16:E2
    ipv4:
      address:
        - ip: 192.168.122.55
          prefix-length: 24
      dhcp: false
      enabled: true
    ipv6:
      enabled: false
EOF
```

In this example we define a desired state of two Ethernet interfaces (eth0 and eth3), their requested IP addresses, routing, and DNS resolution.

Warning

You must ensure that the MAC addresses of all Ethernet interfaces are listed. Those are used during the provisioning process as the identifiers of the nodes and serve to determine which configurations should be applied. This is how we are able to configure multiple nodes using a single ISO or RAW image.

Next up is the second node which we will call `node2.suse.com` and which will use network bonding:

```
cat << EOF > $CONFIG_DIR/network/node2.suse.com.yaml
routes:
  config:
    - destination: 0.0.0.0/0
      metric: 100
      next-hop-address: 192.168.122.1
      next-hop-interface: bond99
      table-id: 254
    - destination: 192.168.122.0/24
      metric: 100
      next-hop-address: 192.168.122.1
      next-hop-interface: bond99
      table-id: 254
dns-resolver:
  config:
    server:
      - 192.168.122.1
      - 8.8.8.8
interfaces:
  - name: bond99
    type: bond
    state: up
    ipv4:
      address:
        - ip: 192.168.122.60
          prefix-length: 24
      enabled: true
    link-aggregation:
      mode: balance-rr
      options:
        miimon: '140'
      port:
```

```
        - eth0
        - eth1
  - name: eth0
    type: ethernet
    state: up
    mac-address: 34:8A:B1:4B:16:E3
    ipv4:
      enabled: false
    ipv6:
      enabled: false
  - name: eth1
    type: ethernet
    state: up
    mac-address: 34:8A:B1:4B:16:E4
    ipv4:
      enabled: false
    ipv6:
      enabled: false
EOF
```

In this example we define a desired state of two Ethernet interfaces (eth0 and eth1) which are not enabling IP addressing, as well as a bond with a round-robin policy and its respective address which is going to be used to forward the network traffic.

Lastly, we'll create the third and final desired state file which will be utilizing a network bridge and which we'll call `node3.suse.com`:

```
cat << EOF > $CONFIG_DIR/network/node3.suse.com.yaml
routes:
  config:
    - destination: 0.0.0.0/0
      metric: 100
      next-hop-address: 192.168.122.1
      next-hop-interface: linux-br0
      table-id: 254
    - destination: 192.168.122.0/24
      metric: 100
      next-hop-address: 192.168.122.1
      next-hop-interface: linux-br0
      table-id: 254
dns-resolver:
  config:
    server:
      - 192.168.122.1
      - 8.8.8.8
interfaces:
  - name: eth0
```

```
    type: ethernet
    state: up
    mac-address: 34:8A:B1:4B:16:E5
    ipv4:
      enabled: false
    ipv6:
      enabled: false
  - name: linux-br0
    type: linux-bridge
    state: up
    ipv4:
      address:
        - ip: 192.168.122.70
          prefix-length: 24
      dhcp: false
      enabled: true
    bridge:
      options:
        group-forward-mask: 0
        mac-ageing-time: 300
        multicast-snooping: true
        stp:
          enabled: true
          forward-delay: 15
          hello-time: 2
          max-age: 20
          priority: 32768
      port:
        - name: eth0
          stp-hairpin-mode: false
          stp-path-cost: 100
          stp-priority: 32
EOF
```

The configuration directory at this point should look like the following:

```
├── definition.yaml
├── network/
│   │── node1.suse.com.yaml
│   │── node2.suse.com.yaml
│   └── node3.suse.com.yaml
└── base-images/
    └── SL-Micro.x86_64-6.2-Base-GM.raw
```

> **Note**
>
> The names of the files under the `network/` directory are intentional. They correspond to the hostnames which will be set during the provisioning process.

13.5.6 Building the OS image

Now that all the necessary configurations are in place, we can build the image by simply running:

```
podman run --rm -it -v $CONFIG_DIR:/eib registry.suse.com/edge/3.7/edge-image-
builder:1.3.4 build --definition-file definition.yaml
```

The output should be similar to the following:

```
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Rpm .......................... [SKIPPED]
Systemd ...................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Embedded Artifact Registry ... [SKIPPED]
Keymap ....................... [SUCCESS]
Kubernetes ................... [SKIPPED]
Certificates ................. [SKIPPED]
Building RAW image...
Kernel Params ................ [SKIPPED]
Image build complete!
```

The snippet above tells us that the `Network` component has successfully been configured, and we can proceed with provisioning our edge nodes.

> **Note**
>
> A log file (`network-config.log`) and the respective NetworkManager connection files can be inspected in the resulting `_build` directory under a timestamped directory for the image run.

13.5.7 Provisioning the edge nodes

Let's copy the resulting RAW image:

```
mkdir edge-nodes && cd edge-nodes
for i in {1..4}; do cp $CONFIG_DIR/modified-image.raw node$i.raw; done
```

You will notice that we copied the built image four times but only specified the network configurations for three nodes. This is because we also want to showcase what will happen if we provision a node which does not match any of the desired configurations.

> **Note**
>
> This guide will use virtualization for the node provisioning examples. Ensure the necessary extensions are enabled in the BIOS (see here (https://documentation.suse.com/sles/15-SP6/html/SLES-all/cha-virt-support.html#sec-kvm-requires-hardware) for details).

We will be using `virt-install` to create virtual machines using the copied raw disks. Each virtual machine will be using 10 GB of RAM and 6 vCPUs.

13.5.7.1 Provisioning the first node

Let's create the virtual machine:

```
virt-install --name node1 --ram 10000 --vcpus 6 --disk path=node1.raw,format=raw --osinfo
 detect=on,name=sle-unknown --graphics none --console pty,target_type=serial --network
 default,mac=34:8A:B1:4B:16:E1 --network default,mac=34:8A:B1:4B:16:E2 --virt-type kvm --
import
```

> **Note**
>
> It is important that we create the network interfaces with the same MAC addresses as the ones in the desired state we described above.

Once the operation is complete, we will see something similar to the following:

```
Starting install...
Creating domain...
```

```
Running text console command: virsh --connect qemu:///system console node1
Connected to domain 'node1'
Escape character is ^] (Ctrl + ])


Welcome to SUSE Linux Micro 6.0 (x86_64) - Kernel 6.4.0-18-default (tty1).

SSH host key: SHA256:XN/R5Tw43reG+QsOw480LxCnhkc/1uqMdwlI6KUBY70 (RSA)
SSH host key: SHA256:/96yGrPGKlhn04f1rb9cXv/2WJt4TtrIN5yEcN66r3s (DSA)
SSH host key: SHA256:Dy/YjBQ7LwjZGaaVcMhTWZNSOstxXBsPsvgJTJq5t00 (ECDSA)
SSH host key: SHA256:TNGqY1LRddpxD/jn/8dkT/9YmVl9hiwulqmayP+wOWQ (ED25519)
eth0: 192.168.122.50
eth1:


Configured with the Edge Image Builder
Activate the web console with: systemctl enable --now cockpit.socket

node1 login:
```

We're now able to log in with the `root:eib` credentials pair. We're also able to SSH into the host if we prefer that over the `virsh console` we're presented with here.

Once logged in, let's confirm that all the settings are in place.

Verify that the hostname is properly set:

```
node1:~ # hostnamectl
 Static hostname: node1.suse.com
 ...
```

Verify that the routing is properly configured:

```
node1:~ # ip r
default via 192.168.122.1 dev eth0 proto static metric 100
192.168.122.0/24 dev eth0 proto static scope link metric 100
192.168.122.0/24 dev eth0 proto kernel scope link src 192.168.122.50 metric 100
```

Verify that Internet connection is available:

```
node1:~ # ping google.com
PING google.com (142.250.72.78) 56(84) bytes of data.
64 bytes from den16s09-in-f14.1e100.net (142.250.72.78): icmp_seq=1 ttl=56 time=13.2 ms
64 bytes from den16s09-in-f14.1e100.net (142.250.72.78): icmp_seq=2 ttl=56 time=13.4 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
```

```
rtt min/avg/max/mdev = 13.248/13.304/13.361/0.056 ms
```

Verify that exactly two Ethernet interfaces are configured and only one of those is active:

```
node1:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
 default qlen 1000
    link/ether 34:8a:b1:4b:16:e1 brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
    inet 192.168.122.50/24 brd 192.168.122.255 scope global noprefixroute eth0
       valid_lft forever preferred_lft forever
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
 default qlen 1000
    link/ether 34:8a:b1:4b:16:e2 brd ff:ff:ff:ff:ff:ff
    altname enp0s3
    altname ens3

node1:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME  UUID                                   TYPE      DEVICE  FILENAME
eth0  dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0    /etc/NetworkManager/system-
connections/eth0.nmconnection
eth1  7e211aea-3d14-59cf-a4fa-be91dac5dbba  ethernet  --      /etc/NetworkManager/system-
connections/eth1.nmconnection
```

You'll notice that the second interface is `eth1` instead of the predefined `eth3` in our desired networking state. This is the case because the NetworkManager Configurator (*nmc*) is able to detect that the OS has given a different name for the NIC with MAC address `34:8a:b1:4b:16:e2` and it adjusts its settings accordingly.

Verify this has indeed happened by inspecting the Combustion phase of the provisioning:

```
node1:~ # journalctl -u combustion | grep nmc
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z INFO
 nmc::apply_conf] Identified host: node1.suse.com
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z INFO
 nmc::apply_conf] Set hostname: node1.suse.com
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z INFO
 nmc::apply_conf] Processing interface 'eth0'...
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z INFO
 nmc::apply_conf] Processing interface 'eth3'...
```

```
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z INFO
 nmc::apply_conf] Using interface name 'eth1' instead of the preconfigured 'eth3'
Apr 23 09:20:19 localhost.localdomain combustion[1360]: [2024-04-23T09:20:19Z INFO  nmc]
 Successfully applied config
```

We will now provision the rest of the nodes, but we will only show the differences in the final configuration. Feel free to apply any or all of the above checks for all nodes you are about to provision.

13.5.7.2 Provisioning the second node

Let's create the virtual machine:

```
virt-install --name node2 --ram 10000 --vcpus 6 --disk path=node2.raw,format=raw --osinfo
 detect=on,name=sle-unknown --graphics none --console pty,target_type=serial --network
 default,mac=34:8A:B1:4B:16:E3 --network default,mac=34:8A:B1:4B:16:E4 --virt-type kvm --
import
```

Once the virtual machine is up and running, we can confirm that this node is using bonded interfaces:

```
node2:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master bond99
 state UP group default qlen 1000
    link/ether 34:8a:b1:4b:16:e3 brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
3: eth1: <BROADCAST,MULTICAST,SLAVE,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master bond99
 state UP group default qlen 1000
    link/ether 34:8a:b1:4b:16:e3 brd ff:ff:ff:ff:ff:ff permaddr 34:8a:b1:4b:16:e4
    altname enp0s3
    altname ens3
4: bond99: <BROADCAST,MULTICAST,MASTER,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
 default qlen 1000
    link/ether 34:8a:b1:4b:16:e3 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.60/24 brd 192.168.122.255 scope global noprefixroute bond99
       valid_lft forever preferred_lft forever
```

Confirm that the routing is using the bond:

```
node2:~ # ip r
default via 192.168.122.1 dev bond99 proto static metric 100
```

```
192.168.122.0/24 dev bond99 proto static scope link metric 100
192.168.122.0/24 dev bond99 proto kernel scope link src 192.168.122.60 metric 300
```

Ensure that the static connection files are properly utilized:

```
node2:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME    UUID                                  TYPE      DEVICE  FILENAME
bond99  4a920503-4862-5505-80fd-4738d07f44c6  bond      bond99  /etc/NetworkManager/
system-connections/bond99.nmconnection
eth0    dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0    /etc/NetworkManager/
system-connections/eth0.nmconnection
eth1    0523c0a1-5f5e-5603-bcf2-68155d5d322e  ethernet  eth1    /etc/NetworkManager/
system-connections/eth1.nmconnection
```

13.5.7.3 Provisioning the third node

Let’s create the virtual machine:

```
virt-install --name node3 --ram 10000 --vcpus 6 --disk path=node3.raw,format=raw --osinfo
 detect=on,name=sle-unknown --graphics none --console pty,target_type=serial --network
 default,mac=34:8A:B1:4B:16:E5 --virt-type kvm --import
```

Once the virtual machine is up and running, we can confirm that this node is using a network bridge:

```
node3:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast master linux-br0
 state UP group default qlen 1000
    link/ether 34:8a:b1:4b:16:e5 brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
3: linux-br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
 default qlen 1000
    link/ether 34:8a:b1:4b:16:e5 brd ff:ff:ff:ff:ff:ff
    inet 192.168.122.70/24 brd 192.168.122.255 scope global noprefixroute linux-br0
       valid_lft forever preferred_lft forever
```

Confirm that the routing is using the bridge:

```
node3:~ # ip r
default via 192.168.122.1 dev linux-br0 proto static metric 100
```

```
192.168.122.0/24 dev linux-br0 proto static scope link metric 100
192.168.122.0/24 dev linux-br0 proto kernel scope link src 192.168.122.70 metric 425
```

Ensure that the static connection files are properly utilized:

```
node3:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME       UUID                                  TYPE      DEVICE     FILENAME
linux-br0  1f8f1469-ed20-5f2c-bacb-a6767bee9bc0  bridge    linux-br0  /etc/
NetworkManager/system-connections/linux-br0.nmconnection
eth0       dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0       /etc/
NetworkManager/system-connections/eth0.nmconnection
```

13.5.7.4 Provisioning the fourth node

Lastly, we will provision a node which will not match any of the predefined configurations by a MAC address. In these cases, we will default to DHCP to configure the network interfaces.

Let's create the virtual machine:

```
virt-install --name node4 --ram 10000 --vcpus 6 --disk path=node4.raw,format=raw --osinfo
 detect=on,name=sle-unknown --graphics none --console pty,target_type=serial --network
 default --virt-type kvm --import
```

Once the virtual machine is up and running, we can confirm that this node is using a random IP address for its network interface:

```
localhost:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
 default qlen 1000
    link/ether 52:54:00:56:63:71 brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
    inet 192.168.122.86/24 brd 192.168.122.255 scope global dynamic noprefixroute eth0
       valid_lft 3542sec preferred_lft 3542sec
    inet6 fe80::5054:ff:fe56:6371/64 scope link noprefixroute
       valid_lft forever preferred_lft forever
```

Verify that nmc failed to apply static configurations for this node:

```
localhost:~ # journalctl -u combustion | grep nmc
```

```
Apr 23 12:15:45 localhost.localdomain combustion[1357]: [2024-04-23T12:15:45Z ERROR nmc]
 Applying config failed: None of the preconfigured hosts match local NICs
```

Verify that the Ethernet interface was configured via DHCP:

```
localhost:~ # journalctl | grep eth0
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>  [1713874529.7801]
 manager: (eth0): new Ethernet device (/org/freedesktop/NetworkManager/Devices/2)
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>  [1713874529.7802]
 device (eth0): state change: unmanaged -> unavailable (reason 'managed', sys-iface-
state: 'external')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>  [1713874529.7929]
 device (eth0): carrier: link connected
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>  [1713874529.7931]
 device (eth0): state change: unavailable -> disconnected (reason 'carrier-changed', sys-
iface-state: 'managed')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>
 [1713874529.7944] device (eth0): Activation: starting connection 'Wired
 Connection' (300ed658-08d4-4281-9f8c-d1b8882d29b9)
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>  [1713874529.7945]
 device (eth0): state change: disconnected -> prepare (reason 'none', sys-iface-state:
 'managed')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>  [1713874529.7947]
 device (eth0): state change: prepare -> config (reason 'none', sys-iface-state:
 'managed')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>  [1713874529.7953]
 device (eth0): state change: config -> ip-config (reason 'none', sys-iface-state:
 'managed')
Apr 23 12:15:29 localhost.localdomain NetworkManager[704]: <info>  [1713874529.7964]
 dhcp4 (eth0): activation: beginning transaction (timeout in 90 seconds)
Apr 23 12:15:33 localhost.localdomain NetworkManager[704]: <info>  [1713874533.1272]
 dhcp4 (eth0): state changed new lease, address=192.168.122.86

localhost:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME              UUID                                  TYPE      DEVICE  FILENAME
Wired Connection  300ed658-08d4-4281-9f8c-d1b8882d29b9  ethernet  eth0    /var/run/
NetworkManager/system-connections/default_connection.nmconnection
```

13.5.8 Unified node configurations

There are occasions where relying on known MAC addresses is not an option. In these cases we can opt for the so-called *unified configuration* which allows us to specify settings in an `_all.yaml` file which will then be applied across all provisioned nodes.

We will build and provision an edge node using different configuration structure. Follow all steps starting from *Section 13.5.3, "Creating the image configuration directory"* up until *Section 13.5.5, "Defining the network configurations"*.

In this example we define a desired state of two Ethernet interfaces (eth0 and eth1) - one using DHCP, and one assigned a static IP address.

```
mkdir -p $CONFIG_DIR/network

cat <<- EOF > $CONFIG_DIR/network/_all.yaml
interfaces:
- name: eth0
  type: ethernet
  state: up
  ipv4:
    dhcp: true
    enabled: true
  ipv6:
    enabled: false
- name: eth1
  type: ethernet
  state: up
  ipv4:
    address:
    - ip: 10.0.0.1
      prefix-length: 24
    enabled: true
    dhcp: false
  ipv6:
    enabled: false
EOF
```

Let's build the image:

```
podman run --rm -it -v $CONFIG_DIR:/eib registry.suse.com/edge/3.7/edge-image-
builder:1.3.4 build --definition-file definition.yaml
```

Once the image is successfully built, let's create a virtual machine using it:

```
virt-install --name node1 --ram 10000 --vcpus 6 --disk path=$CONFIG_DIR/modified-
image.raw,format=raw --osinfo detect=on,name=sle-unknown --graphics none --console
 pty,target_type=serial --network default --network default --virt-type kvm --import
```

The provisioning process might take a few minutes. Once it's finished, log in to the system with the provided credentials.

Verify that the routing is properly configured:

```
localhost:~ # ip r
default via 192.168.122.1 dev eth0 proto dhcp src 192.168.122.100 metric 100
10.0.0.0/24 dev eth1 proto kernel scope link src 10.0.0.1 metric 101
192.168.122.0/24 dev eth0 proto kernel scope link src 192.168.122.100 metric 100
```

Verify that Internet connection is available:

```
localhost:~ # ping google.com
PING google.com (142.250.72.46) 56(84) bytes of data.
64 bytes from den16s08-in-f14.1e100.net (142.250.72.46): icmp_seq=1 ttl=56 time=14.3 ms
64 bytes from den16s08-in-f14.1e100.net (142.250.72.46): icmp_seq=2 ttl=56 time=14.2 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 14.196/14.260/14.324/0.064 ms
```

Verify that the Ethernet interfaces are configured and active:

```
localhost:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
 default qlen 1000
    link/ether 52:54:00:26:44:7a brd ff:ff:ff:ff:ff:ff
    altname enp1s0
    inet 192.168.122.100/24 brd 192.168.122.255 scope global dynamic noprefixroute eth0
       valid_lft 3505sec preferred_lft 3505sec
3: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
 default qlen 1000
    link/ether 52:54:00:ec:57:9e brd ff:ff:ff:ff:ff:ff
    altname enp7s0
    inet 10.0.0.1/24 brd 10.0.0.255 scope global noprefixroute eth1
       valid_lft forever preferred_lft forever

localhost:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME  UUID                                  TYPE      DEVICE  FILENAME
eth0  dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0    /etc/NetworkManager/system-
connections/eth0.nmconnection
eth1  0523c0a1-5f5e-5603-bcf2-68155d5d322e  ethernet  eth1    /etc/NetworkManager/system-
connections/eth1.nmconnection
```

```
localhost:~ # cat /etc/NetworkManager/system-connections/eth0.nmconnection
[connection]
autoconnect=true
autoconnect-slaves=-1
id=eth0
interface-name=eth0
type=802-3-ethernet
uuid=dfd202f5-562f-5f07-8f2a-a7717756fb70

[ipv4]
dhcp-client-id=mac
dhcp-send-hostname=true
dhcp-timeout=2147483647
ignore-auto-dns=false
ignore-auto-routes=false
method=auto
never-default=false

[ipv6]
addr-gen-mode=0
dhcp-timeout=2147483647
method=disabled

localhost:~ # cat /etc/NetworkManager/system-connections/eth1.nmconnection
[connection]
autoconnect=true
autoconnect-slaves=-1
id=eth1
interface-name=eth1
type=802-3-ethernet
uuid=0523c0a1-5f5e-5603-bcf2-68155d5d322e

[ipv4]
address0=10.0.0.1/24
dhcp-timeout=2147483647
method=manual

[ipv6]
addr-gen-mode=0
dhcp-timeout=2147483647
method=disabled
```

13.5.9 Custom network configurations

We have already covered the default network configuration for Edge Image Builder which relies on the NetworkManager Configurator. However, there is also the option to modify it via a custom script. Whilst this option is very flexible and is also not MAC address dependant, its limitation stems from the fact that using it is much less convenient when bootstrapping multiple nodes with a single image.

> **Note**
>
> It is recommended to use the default network configuration via files describing the desired network states under the `/network` directory. Only opt for custom scripting when that behaviour is not applicable to your use case.

We will build and provision an edge node using different configuration structure. Follow all steps starting from *Section 13.5.3, "Creating the image configuration directory"* up until *Section 13.5.5, "Defining the network configurations"*.

In this example, we will create a custom script which applies static configuration for the `eth0` interface on all provisioned nodes, as well as removing and disabling the automatically created wired connections by NetworkManager. This is beneficial in situations where you want to make sure that every node in your cluster has an identical networking configuration, and as such you do not need to be concerned with the MAC address of each node prior to image creation.

Let's start by storing the connection file in the `/custom/files` directory:

```
mkdir -p $CONFIG_DIR/custom/files

cat << EOF > $CONFIG_DIR/custom/files/eth0.nmconnection
[connection]
autoconnect=true
autoconnect-slaves=-1
autoconnect-retries=1
id=eth0
interface-name=eth0
type=802-3-ethernet
uuid=dfd202f5-562f-5f07-8f2a-a7717756fb70
wait-device-timeout=60000

[ipv4]
dhcp-timeout=2147483647
method=auto

[ipv6]
```

```
addr-gen-mode=eui64
dhcp-timeout=2147483647
method=disabled
EOF
```

Now that the static configuration is created, we will also create our custom network script:

```
mkdir -p $CONFIG_DIR/network

cat << EOF > $CONFIG_DIR/network/configure-network.sh
#!/bin/bash
set -eux

# Remove and disable wired connections
mkdir -p /etc/NetworkManager/conf.d/
printf "[main]\nno-auto-default=*\n" > /etc/NetworkManager/conf.d/no-auto-default.conf
rm -f /var/run/NetworkManager/system-connections/* || true

# Copy pre-configured network configuration files into NetworkManager
mkdir -p /etc/NetworkManager/system-connections/
cp eth0.nmconnection /etc/NetworkManager/system-connections/
chmod 600 /etc/NetworkManager/system-connections/*.nmconnection
EOF

chmod a+x $CONFIG_DIR/network/configure-network.sh
```

Note

The nmc binary will still be included by default, so it can also be used in the `configure-network.sh` script if necessary.

Warning

The custom script must always be provided under `/network/configure-network.sh` in the configuration directory. If present, all other files will be ignored. It is NOT possible to configure a network by working with both static configurations in YAML format and a custom script simultaneously.

The configuration directory at this point should look like the following:

```
├── definition.yaml
├── custom/
│   └── files/
```

```
│       └── eth0.nmconnection
├── network/
│   └── configure-network.sh
└── base-images/
    └── SL-Micro.x86_64-6.2-Base-GM.raw
```

Let’s build the image:

```
podman run --rm -it -v $CONFIG_DIR:/eib registry.suse.com/edge/3.7/edge-image-
builder:1.3.4 build --definition-file definition.yaml
```

Once the image is successfully built, let’s create a virtual machine using it:

```
virt-install --name node1 --ram 10000 --vcpus 6 --disk path=$CONFIG_DIR/modified-
image.raw,format=raw --osinfo detect=on,name=sle-unknown --graphics none --console
 pty,target_type=serial --network default --virt-type kvm --import
```

The provisioning process might take a few minutes. Once it’s finished, log in to the system with the provided credentials.

Verify that the routing is properly configured:

```
localhost:~ # ip r
default via 192.168.122.1 dev eth0 proto dhcp src 192.168.122.185 metric 100
192.168.122.0/24 dev eth0 proto kernel scope link src 192.168.122.185 metric 100
```

Verify that Internet connection is available:

```
localhost:~ # ping google.com
PING google.com (142.250.72.78) 56(84) bytes of data.
64 bytes from den16s09-in-f14.1e100.net (142.250.72.78): icmp_seq=1 ttl=56 time=13.6 ms
64 bytes from den16s09-in-f14.1e100.net (142.250.72.78): icmp_seq=2 ttl=56 time=13.6 ms
^C
--- google.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 13.592/13.599/13.606/0.007 ms
```

Verify that an Ethernet interface is statically configured using our connection file and is active:

```
localhost:~ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen
 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
```

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group
 default qlen 1000
    link/ether 52:54:00:31:d0:1b brd ff:ff:ff:ff:ff:ff
    altname enp0s2
    altname ens2
    inet 192.168.122.185/24 brd 192.168.122.255 scope global dynamic noprefixroute eth0

localhost:~ # nmcli -f NAME,UUID,TYPE,DEVICE,FILENAME con show
NAME  UUID                                  TYPE      DEVICE  FILENAME
eth0  dfd202f5-562f-5f07-8f2a-a7717756fb70  ethernet  eth0    /etc/NetworkManager/system-
connections/eth0.nmconnection

localhost:~ # cat  /etc/NetworkManager/system-connections/eth0.nmconnection
[connection]
autoconnect=true
autoconnect-slaves=-1
autoconnect-retries=1
id=eth0
interface-name=eth0
type=802-3-ethernet
uuid=dfd202f5-562f-5f07-8f2a-a7717756fb70
wait-device-timeout=60000

[ipv4]
dhcp-timeout=2147483647
method=auto

[ipv6]
addr-gen-mode=eui64
dhcp-timeout=2147483647
method=disabled
```

14 RKE2

See RKE2 official documentation (https://docs.rke2.io/).

RKE2 is a fully conformant Kubernetes distribution that focuses on security and compliance by:

- Providing defaults and configuration options that allow clusters to pass the CIS Kubernetes Benchmark v1.6 or v1.23 with minimal operator intervention
- Enabling FIPS 140-2 compliance
- Regularly scanning components for CVEs using trivy (https://trivy.dev) in the RKE2 build pipeline

RKE2 launches control plane components as static pods, managed by kubelet. The embedded container runtime is containerd.

Note: RKE2 is also known as RKE Government in order to convey another use case and sector it currently targets.

14.1 RKE2 vs K3s

K3s is a fully compliant and lightweight Kubernetes distribution focused on Edge, IoT, ARM - optimized for ease of use and resource constrained environments.

RKE2 combines the best of both worlds from the 1.x version of RKE (hereafter referred to as RKE1) and K3s.

From K3s, it inherits the usability, ease of operation and deployment model.

From RKE1, it inherits close alignment with upstream Kubernetes. In places, K3s has diverged from upstream Kubernetes in order to optimize for edge deployments, but RKE1 and RKE2 can stay closely aligned with upstream.

14.2 How does SUSE Telco Cloud use RKE2?

RKE2 is a fundamental piece of the SUSE Telco Cloud stack. It sits on top of SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*), providing a standard Kubernetes interface required to deploy Edge workloads.

14.3 Best practices

14.3.1 Installation

The recommended way of installing RKE2 as part of the SUSE Telco Cloud stack is by using Edge Image Builder (EIB). See the EIB documentation (*Chapter 12, Edge Image Builder*) for more details on how to configure it to deploy RKE2.

EIB is flexible enough to support any parameter required by RKE2, such as specifying the RKE2 version, the servers (https://docs.rke2.io/reference/server_config) or the agents (https://docs.rke2.io/reference/linux_agent_config) configuration, covering all the Edge use cases.

For other use cases involving Metal^3, RKE2 is also being used and installed. In those particular cases, the Cluster API Provider RKE2 (https://github.com/rancher-sandbox/cluster-api-provider-rke2) automatically deploys RKE2 on clusters being provisioned with Metal^3 using the Edge Stack.

In those cases, the RKE2 configuration must be applied on the different CRDs involved. An example of how to provide a different CNI using the `RKE2ControlPlane` CRD looks like:

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  serverConfig:
    cni: calico
    cniMultusEnable: true
...
```

For more information about the Metal^3 use cases, see *Chapter 11, Metal^3*.

14.3.2 High availability

For HA deployments, EIB automatically deploys and configures MetalLB (*Chapter 17, MetalLB*) and the Endpoint Copier Operator (*Chapter 18, Endpoint Copier Operator*) to expose the RKE2 API endpoint externally.

14.3.3 Networking

SUSE Telco Cloud Stack supports Cilium (https://docs.cilium.io/en/stable/), Calico (https://docs.tigera.io/calico/latest/about/), with Cilium as its default CNI. Multus (https://github.com/k8snetworkplumbingwg/multus-cni) meta-plugin can also be used when pods require multiple network interfaces. RKE2 standalone supports a wider range of CNI options (https://docs.rke2.io/install/network_options).

14.3.4 Storage

RKE2 does not provide any kind of persistent storage class or operators. For clusters spanning over multiple nodes, it is recommended to use SUSE Storage (*Chapter 15, SUSE Storage*).

15 SUSE Storage

SUSE Storage is a lightweight, reliable, and user-friendly distributed block storage system designed for Kubernetes. It is a product based on Longhorn, an open-source project initially developed by Rancher Labs and currently incubated under the CNCF.

15.1 Prerequisites

If you are following this guide, it assumes that you have the following already available:

- At least one host with SUSE Linux Micro 6.2 installed; this can be physical or virtual
- A Kubernetes cluster installed; either K3s or RKE2
- Helm

15.2 Manual installation of SUSE Storage

15.2.1 Installing Open-iSCSI

A core requirement of deploying and using SUSE Storage is the installation of the `open-iscsi` package and the `iscsid` daemon running on all Kubernetes nodes. This is necessary, since Longhorn relies on `iscsiadm` on the host to provide persistent volumes to Kubernetes.

Let's install it:

```
transactional-update pkg install open-iscsi
```

It is important to note that once the operation is completed, the package is only installed into a new snapshot as SUSE Linux Micro is an immutable operating system. In order to load it and for the `iscsid` daemon to start running, we must reboot into that new snapshot that we just created. Issue the reboot command when you are ready:

```
reboot
```

> **Tip**
>
> For additional help installing open-iscsi, refer to the official Longhorn documentation (https://longhorn.io/docs/1.12.1/deploy/install/#installing-open-iscsi).

15.2.2 Installing SUSE Storage

There are several ways to install SUSE Storage on your Kubernetes clusters. This guide will follow through the Helm installation, however feel free to follow the official documentation (https://longhorn.io/docs/1.12.1/deploy/install/) if another approach is desired.

Starting with SUSE Telco Cloud 3.7, the V2 Data Engine is recommended for new volumes. Before enabling it, prepare the required block devices and review *Chapter 66, SUSE Storage V2 Data Engine*.

1. Log into the Rancher Application Collection:

```
helm registry login dp.apps.rancher.io --username $APPS.RANCHER.IO_USERNAME --
password $APPS.RANCHER.IO_ACCESS_TOKEN
```

2. Install SUSE Storage in the `longhorn-system` namespace and add your container registry credentials:

```
helm install longhorn oci://dp.apps.rancher.io/charts/suse-storage \
  --version 1.12.1 \
  --namespace longhorn-system \
  --create-namespace \
  --set privateRegistry.createSecret=true \
  --set privateRegistry.registryUrl=dp.apps.rancher.io \
  --set privateRegistry.registryUser=$APPS.RANCHER.IO_USERNAME \
  --set privateRegistry.registryPasswd=$APPS.RANCHER.IO_ACCESS_TOKEN \
  --set privateRegistry.registrySecret=application-collection \
  --set defaultSettings.v1DataEngine=false \
  --set defaultSettings.v2DataEngine=true \
  --set persistence.dataEngine=v2
```

3. Confirm that the deployment succeeded:

```
kubectl -n longhorn-system get pods
```

```
localhost:~ # kubectl -n longhorn-system get pods
NAME                                                READY   STATUS    RESTARTS
  AGE
```

```
csi-attacher-7656559cf4-pkhh6                       1/1     Running   0
  103s
csi-attacher-7656559cf4-pnzw5                       1/1     Running   0
  103s
csi-attacher-7656559cf4-z94mm                       1/1     Running   0
  103s
csi-provisioner-6d9cf6456d-kcwtq                    1/1     Running   0
  103s
csi-provisioner-6d9cf6456d-mvvml                    1/1     Running   0
  103s
csi-provisioner-6d9cf6456d-q4f88                    1/1     Running   0
  103s
csi-resizer-f587cd467-clr2n                         1/1     Running   0
  103s
csi-resizer-f587cd467-z28v4                         1/1     Running   0
  103s
csi-resizer-f587cd467-zxmtx                         1/1     Running   0
  103s
csi-snapshotter-6dcdf78684-757mg                    1/1     Running   0
  103s
csi-snapshotter-6dcdf78684-8ktgc                    1/1     Running   0
  103s
csi-snapshotter-6dcdf78684-ffsqr                    1/1     Running   0
  103s
engine-image-ei-099f845a-lvdtr                      1/1     Running   0
  2m21s
instance-manager-4adffddaffe02374cd5635b8a6113de7   1/1     Running   0
  111s
longhorn-csi-plugin-w7pwr                           3/3     Running   0
  103s
longhorn-driver-deployer-6886fb84bc-wm9h6           1/1     Running   2 (2m32s ago)
  2m45s
longhorn-manager-zblbl                              2/2     Running   0
  2m45s
longhorn-ui-6bcc65d4bd-mcn6r                        1/1     Running   0
  2m45s
longhorn-ui-6bcc65d4bd-rwf97                        1/1     Running   0
  2m45s
```

15.3 Creating SUSE Storage volumes

SUSE Storage utilizes Kubernetes resources called `StorageClass` in order to automatically provision `PersistentVolume` objects for pods. Think of `StorageClass` as a way for administrators to describe the *classes* or *profiles* of storage they offer.

The following example creates a V2 `StorageClass`. The V2 Data Engine and a schedulable block disk must already be configured as described in *Chapter 66, SUSE Storage V2 Data Engine*.

```
kubectl apply -f - <<EOF
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: longhorn-v2-example
provisioner: driver.longhorn.io
allowVolumeExpansion: true
parameters:
  dataEngine: "v2"
  numberOfReplicas: "3"
  staleReplicaTimeout: "2880" # 48 hours in minutes
  fromBackup: ""
  fsType: "ext4"
EOF
```

Now that we have our `StorageClass` in place, we need a `PersistentVolumeClaim` referencing it. A `PersistentVolumeClaim` (PVC) is a request for storage by a user. PVCs consume `PersistentVolume` resources. Claims can request specific sizes and access modes (e.g., they can be mounted once read/write or many times read-only).

Let's create a `PersistentVolumeClaim`:

```
kubectl apply -f - <<EOF
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: longhorn-volv-pvc
  namespace: longhorn-system
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: longhorn-v2-example
  resources:
    requests:
      storage: 2Gi
EOF
```

That's it! Once we have the `PersistentVolumeClaim` created, we can proceed with attaching it to a `Pod`. When the `Pod` is deployed, Kubernetes creates the Longhorn volume and binds it to the `Pod` if storage is available.

```
kubectl apply -f - <<EOF
```

```
apiVersion: v1
kind: Pod
metadata:
  name: volume-test
  namespace: longhorn-system
spec:
  containers:
  - name: volume-test
    image: nginx:stable-alpine
    imagePullPolicy: IfNotPresent
    volumeMounts:
    - name: volv
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: volv
    persistentVolumeClaim:
      claimName: longhorn-volv-pvc
EOF
```

Tip

The concept of storage in Kubernetes is a complex, but important topic. We briefly mentioned some of the most common Kubernetes resources, however, we suggest to familiarize yourself with the terminology documentation (https://longhorn.io/docs/1.12.1/terminology/) that Longhorn offers.

In this example, the result should look something like this:

```
localhost:~ # kubectl get storageclass
NAME                 PROVISIONER          RECLAIMPOLICY   VOLUMEBINDINGMODE
 ALLOWVOLUMEEXPANSION   AGE
longhorn (default)   driver.longhorn.io   Delete          Immediate           true
            12m
longhorn-v2-example  driver.longhorn.io   Delete          Immediate           true
            24s

localhost:~ # kubectl get pvc -n longhorn-system
NAME                STATUS   VOLUME                                     CAPACITY   ACCESS
 MODES   STORAGECLASS       AGE
longhorn-volv-pvc   Bound    pvc-f663a92e-ac32-49ae-b8e5-8a6cc29a7d1e   2Gi        RWO
         longhorn-v2-example   54s

localhost:~ # kubectl get pods -n longhorn-system
```

```
NAME                                                READY   STATUS    RESTARTS      AGE
csi-attacher-5c4bfdcf59-qmjtz                       1/1     Running   0             14m
csi-attacher-5c4bfdcf59-s7n65                       1/1     Running   0             14m
csi-attacher-5c4bfdcf59-w9xgs                       1/1     Running   0             14m
csi-provisioner-667796df57-fmz2d                    1/1     Running   0             14m
csi-provisioner-667796df57-p7rjr                    1/1     Running   0             14m
csi-provisioner-667796df57-w9fdq                    1/1     Running   0             14m
csi-resizer-694f8f5f64-2rb8v                        1/1     Running   0             14m
csi-resizer-694f8f5f64-z9v9x                        1/1     Running   0             14m
csi-resizer-694f8f5f64-zlncz                        1/1     Running   0             14m
csi-snapshotter-959b69d4b-5dpvj                     1/1     Running   0             14m
csi-snapshotter-959b69d4b-lwwkv                     1/1     Running   0             14m
csi-snapshotter-959b69d4b-tzhwc                     1/1     Running   0             14m
engine-image-ei-5cefaf2b-hvdv5                      1/1     Running   0             14m
instance-manager-0ee452a2e9583753e35ad00602250c5b   1/1     Running   0             14m
longhorn-csi-plugin-gd2jx                           3/3     Running   0             14m
longhorn-driver-deployer-9f4fc86-j6h2b              1/1     Running   0             15m
longhorn-manager-z4lnl                              1/1     Running   0             15m
longhorn-ui-5f4b7bbf69-bln7h                        1/1     Running   3 (14m ago)   15m
longhorn-ui-5f4b7bbf69-lh97n                        1/1     Running   3 (14m ago)   15m
volume-test                                         1/1     Running   0             26s
```

15.4 Accessing the UI

If you installed SUSE Storage with kubectl or Helm, you need to set up an Ingress controller to allow external traffic into the cluster. Authentication is not enabled by default. If the Rancher catalog app was used, Rancher automatically created an Ingress controller with access control (the rancher-proxy).

1. Get the Longhorn's external service IP address:

   ```
   kubectl -n longhorn-system get svc
   ```

2. Once you have retrieved the `longhorn-frontend` IP address, you can start using the UI by navigating to it in your browser.

15.5 Installing with Edge Image Builder

SUSE Telco Cloud is using *Chapter 12, Edge Image Builder* in order to customize base SUSE Linux Micro OS images. We are going to demonstrate how to do so for provisioning an RKE2 cluster with SUSE Storage on top of it.

Let’s create the definition file:

```
export CONFIG_DIR=$HOME/eib
mkdir -p $CONFIG_DIR

cat << EOF > $CONFIG_DIR/iso-definition.yaml
apiVersion: 1.3
image:
  imageType: iso
  baseImage: SL-Micro.x86_64-6.2-Base-SelfInstall-GM.install.iso
  arch: x86_64
  outputImageName: eib-image.iso
kubernetes:
  version: v1.36.3+rke2r1
  helm:
    charts:
      - name: suse-storage
        releaseName: longhorn
        version: 1.12.1
        repositoryName: rancher-application-collection
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
    repositories:
      - name: rancher-application-collection
        url: oci://dp.apps.rancher.io/charts
        authentication:
          username: $APPS.RANCHER.IO_USERNAME
          password: $APPS.RANCHER.IO_ACCESS_TOKEN
embeddedArtifactRegistry:
  registries:
    - uri: dp.apps.rancher.io
      authentication:
        username: $APPS.RANCHER.IO_USERNAME
        password: $APPS.RANCHER.IO_ACCESS_TOKEN
  images:
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-attacher:4.12.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-provisioner:5.3.0-17.8
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-resizer:2.2.1-10.8
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-snapshotter:8.6.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-livenessprobe:2.19.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-node-driver-
registrar:2.17.0-17.6
    - name: dp.apps.rancher.io/containers/longhorn-backing-image-manager:1.12.1-1.5
    - name: dp.apps.rancher.io/containers/longhorn-engine:1.12.1-1.6
    - name: dp.apps.rancher.io/containers/longhorn-instance-manager:1.12.1-1.7
    - name: dp.apps.rancher.io/containers/longhorn-manager:1.12.1-1.5
```

```
    - name: dp.apps.rancher.io/containers/longhorn-share-manager:1.12.1-1.5
    - name: dp.apps.rancher.io/containers/longhorn-ui:1.12.1-1.3
    - name: dp.apps.rancher.io/containers/rancher-support-bundle-kit:0.0.92-13.12
operatingSystem:
  packages:
    sccRegistrationCode: <reg-code>
    packageList:
      - open-iscsi
  users:
  - username: root
    encryptedPassword: \$6\$jHugJNNd3HElGsUZ\
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/zb4r8EmnrrNCF.P/
EOF
```

Note

Customizing any of the Helm chart values is possible via a separate file provided under `helm.charts[].valuesFile`. Refer to the upstream documentation (https://github.com/suse-edge/edge-image-builder/blob/release-1.3/docs/building-images.md#kubernetes) for details. To use the V2 Data Engine, include the V2 values and ensure that the required raw block devices are prepared and registered as described in *Chapter 66, SUSE Storage V2 Data Engine*.

Let's build the image:

```
podman run --rm --privileged -it -v $CONFIG_DIR:/eib registry.suse.com/edge/3.7/edge-
image-builder:1.3.4 build --definition-file $CONFIG_DIR/iso-definition.yaml
```

After the image is built, you can use it to install your OS on a physical or virtual host. Once the provisioning is complete, you are able to log in to the system using the `root:eib` credentials pair.

Ensure that SUSE Storage has been successfully deployed:

```
localhost:~ # /var/lib/rancher/rke2/bin/kubectl --kubeconfig /etc/rancher/rke2/rke2.yaml
 -n longhorn-system get pods
NAME                                                READY   STATUS    RESTARTS        AGE
csi-attacher-5c4bfdcf59-qmjtz                       1/1     Running   0
 103s
csi-attacher-5c4bfdcf59-s7n65                       1/1     Running   0
 103s
csi-attacher-5c4bfdcf59-w9xgs                       1/1     Running   0
 103s
csi-provisioner-667796df57-fmz2d                    1/1     Running   0
 103s
```

```
csi-provisioner-667796df57-p7rjr                    1/1     Running   0
 103s
csi-provisioner-667796df57-w9fdq                    1/1     Running   0
 103s
csi-resizer-694f8f5f64-2rb8v                        1/1     Running   0
 103s
csi-resizer-694f8f5f64-z9v9x                        1/1     Running   0
 103s
csi-resizer-694f8f5f64-zlncz                        1/1     Running   0
 103s
csi-snapshotter-959b69d4b-5dpvj                     1/1     Running   0
 103s
csi-snapshotter-959b69d4b-lwwkv                     1/1     Running   0
 103s
csi-snapshotter-959b69d4b-tzhwc                     1/1     Running   0
 103s
engine-image-ei-5cefaf2b-hvdv5                      1/1     Running   0
 109s
instance-manager-0ee452a2e9583753e35ad00602250c5b   1/1     Running   0
 109s
longhorn-csi-plugin-gd2jx                           3/3     Running   0
 103s
longhorn-driver-deployer-9f4fc86-j6h2b              1/1     Running   0
 2m28s
longhorn-manager-z4lnl                              1/1     Running   0
 2m28s
longhorn-ui-5f4b7bbf69-bln7h                        1/1     Running   3 (2m7s ago)
 2m28s
longhorn-ui-5f4b7bbf69-lh97n                        1/1     Running   3 (2m10s ago)
 2m28s
```

Note

This installation will not work for completely air-gapped environments. In those cases, please refer to *Section 64.8, "SUSE Storage Installation"*.

16 SUSE Security

SUSE Security is a security solution for Kubernetes that provides L7 network security, runtime security, supply chain security, and compliance checks in a cohesive package.

SUSE Security is a product that is deployed as a platform of multiple containers, each communicating over various ports and interfaces. Under the hood, it uses NeuVector as its underlying container security component. The following containers make up the SUSE Security platform:

- Manager. A stateless container which presents the Web-based console. Typically, only one is needed and this can run anywhere. Failure of the Manager does not affect any of the operations of the controller or enforcer. However, certain notifications (events) and recent connection data are cached in memory by the Manager so viewing of these would be affected.
- Controller. The 'control plane' for SUSE Security must be deployed in an HA configuration, so configuration is not lost in a node failure. These can run anywhere, although customers often choose to place these on 'management', master or infra nodes because of their criticality.
- Enforcer. This container is deployed as a DaemonSet so one Enforcer is on every node to be protected. Typically deploys to every worker node but scheduling can be enabled for master and infra nodes to deploy there as well. Note: If the Enforcer is not on a cluster node and connections come from a pod on that node, SUSE Security labels them as 'unmanaged' workloads.
- Scanner. Performs the vulnerability scanning using the built-in CVE database, as directed by the Controller. Multiple scanners can be deployed to increase scanning capacity. Scanners can run anywhere but are often run on the nodes where the controllers run. See below for sizing considerations of scanner nodes. A scanner can also be invoked independently when used for build-phase scanning, for example, within a pipeline that triggers a scan, retrieves the results, and stops the scanner. The scanner contains the latest CVE database so should be updated daily.
- Updater. The updater triggers an update of the scanner through a Kubernetes cron job when an update of the CVE database is desired. Please be sure to configure this for your environment.

A more in-depth SUSE Security onboarding and best practices documentation can be found here (https://open-docs.neuvector.com/).

16.1 How does SUSE Telco Cloud use SUSE Security?

SUSE Telco Cloud provides a leaner configuration of SUSE Security as a starting point for edge deployments.

16.2 Important notes

- The `Scanner` container must have enough memory to pull the image to be scanned into memory and expand it. To scan images exceeding 1 GB, increase the scanner's memory to slightly above the largest expected image size.
- High network connections expected in Protect mode. The `Enforcer` requires CPU and memory when in Protect (inline firewall blocking) mode to hold and inspect connections and possible payload (DLP). Increasing memory and dedicating a CPU core to the `Enforcer` can ensure adequate packet filtering capacity.

16.3 Installing with Edge Image Builder

SUSE Telco Cloud is using *Chapter 12, Edge Image Builder* in order to customize base SUSE Linux Micro OS images. Follow *Section 64.7, "SUSE Security Installation"* for an air-gapped installation of SUSE Security on top of Kubernetes clusters provisioned by EIB.

17 MetalLB

See MetalLB official documentation (https://metallb.universe.tf/).

MetalLB is a load-balancer implementation for bare-metal Kubernetes clusters, using standard routing protocols.

In bare-metal environments, setting up network load balancers is notably more complex than in cloud environments. Unlike the straightforward API calls in cloud setups, bare-metal requires either dedicated network appliances or a combination of load balancers and Virtual IP (VIP) configurations to manage High Availability (HA) or address the potential Single Point of Failure (SPOF) inherent in a single node load balancer. These configurations are not easily automated, posing challenges in Kubernetes deployments where components dynamically scale up and down.

MetalLB addresses these challenges by harnessing the Kubernetes model to create Load-Balancer type services as if they were operating in a cloud environment, even on bare-metal setups.

There are two different approaches, via L2 mode (https://metallb.universe.tf/concepts/layer2/) (using ARP *tricks*) or via BGP (https://metallb.universe.tf/concepts/bgp/). Mainly L2 does not need any special network gear but BGP is in general better. It depends on the use cases.

17.1 How does SUSE Telco Cloud use MetalLB?

SUSE Telco Cloud uses MetalLB in three key ways:

- As a Load Balancer Solution: MetalLB serves as the Load Balancer solution for bare-metal machines.
- For an HA K3s/RKE2 Setup: MetalLB allows for load balancing the Kubernetes API using a Virtual IP address.
- As an L3 BGP solution where MetalLB advertises routes to the service IPs to nearby routers.

Note

In order to be able to expose the API, the Endpoint Copier Operator (*Chapter 18, Endpoint Copier Operator*) is used to keep in sync the K8s API endpoints from the `kubernetes` service to a `kubernetes-vip` LoadBalancer service.

17.2 Best practices

Installation of MetalLB in L2 mode is described in *Chapter 61, MetalLB on K3s (using Layer 2 Mode)* and for L3 mode in *Chapter 62, MetalLB on K3s (using Layer 3 Mode)*.

A guide on installing MetalLB in front of the `kube-api-server` to achieve high-availability topology can be found in *Chapter 63, MetalLB in front of the Kubernetes API server*.

17.3 Known issues

- K3s comes with its Load Balancer solution called `Klipper`. To use MetalLB, `Klipper` must be disabled. This can be done by starting the K3s server with the `--disable servicelb` option, as described in the K3s documentation (https://docs.k3s.io/networking).

18 Endpoint Copier Operator

Endpoint Copier Operator (https://github.com/suse-edge/endpoint-copier-operator) is a Kubernetes operator whose purpose is to create a copy of a Kubernetes Service and Endpoint and to keep them synced.

18.1 How does SUSE Telco Cloud use Endpoint Copier Operator?

At SUSE Telco Cloud, the Endpoint Copier Operator plays a crucial role in achieving High Availability (HA) setup for K3s/RKE2 clusters. This is accomplished by creating a `kubernetes-vip` service of type `LoadBalancer`, ensuring its Endpoint remains in constant synchronization with the kubernetes Endpoint. MetalLB (*Chapter 17, MetalLB*) is leveraged to manage the `kubernetes-vip` service, as the exposed IP address is used from other nodes to join the cluster.

18.2 Best Practices

Comprehensive documentation for using the Endpoint Copier Operator can be found here (https://github.com/suse-edge/endpoint-copier-operator/blob/main/README.md).

Additionally, refer to our guide (*Chapter 61, MetalLB on K3s (using Layer 2 Mode)*) on achieving a K3s/RKE2 HA setup using the Endpoint Copier Operator and MetalLB.

18.3 Known issues

Presently, the Endpoint Copier Operator is limited to working with only one Service/Endpoint. Enhancements to support multiple Services/Endpoints are planned for the future.

19 Edge Virtualization

This section describes how you can use Edge Virtualization to run virtual machines on your edge nodes. Edge Virtualization is designed for lightweight virtualization use-cases, where it is expected that a common workflow for the deployment and management of both virtualized and containerized applications will be utilized.

SUSE Telco Cloud Virtualization supports two methods of running virtual machines:

1. Deploying the virtual machines manually via libvirt+qemu-kvm at the host level (where Kubernetes is not involved)
2. Deploying the KubeVirt operator for Kubernetes-based management of virtual machines

Both options are valid, but only the second one is covered below. If you want to use the standard out-of-the box virtualization mechanisms provided by SUSE Linux Micro, a comprehensive guide can be found here (https://documentation.suse.com/sles/15-SP6/html/SLES-all/chap-virtualization-introduction.html), and whilst it was primarily written for SUSE Linux Enterprise Server, the concepts are almost identical.

This guide initially explains how to deploy the additional virtualization components onto a system that has already been pre-deployed, but follows with a section that describes how to embed this configuration in the initial deployment via Edge Image Builder. If you do not want to run through the basics and set things up manually, skip right ahead to that section.

19.1 KubeVirt overview

KubeVirt allows for managing Virtual Machines with Kubernetes alongside the rest of your containerized workloads. It does this by running the user space portion of the Linux virtualization stack in a container. This minimizes the requirements on the host system, allowing for easier setup and management.

Details about KubeVirt's architecture can be found in the upstream documentation. (https://kubevirt.io/user-guide/architecture/)

19.2 Prerequisites

If you are following this guide, we assume you have the following already available:

- At least one physical host with SUSE Linux Micro 6.2 installed, and with virtualization extensions enabled in the BIOS (see here (https://documentation.suse.com/sles/15-SP6/html/SLES-all/cha-virt-support.html#sec-kvm-requires-hardware) for details).
- Across your nodes, a K3s/RKE2 Kubernetes cluster already deployed and with an appropriate `kubeconfig` that enables superuser access to the cluster.
- Access to the root user — these instructions assume you are the root user, and *not* escalating your privileges via `sudo`.
- You have Helm (https://helm.sh/docs/intro/install/) available locally with an adequate network connection to be able to push configurations to your Kubernetes cluster and download the required images.

19.3 Manual installation of Edge Virtualization

This guide will not walk you through the deployment of Kubernetes, but it assumes that you have either installed the SUSE Telco Cloud-appropriate version of K3s (https://k3s.io/) or RKE2 (https://docs.rke2.io/install/quickstart) and that you have your kubeconfig configured accordingly so that standard `kubectl` commands can be executed as the superuser. We assume your node forms a single-node cluster, although there are no significant differences expected for multi-node deployments.

SUSE Telco Cloud Virtualization is deployed via three separate Helm charts, specifically:

- **KubeVirt**: The core virtualization components, that is, Kubernetes CRDs, operators and other components required for enabling Kubernetes to deploy and manage virtual machines.
- **KubeVirt Dashboard Extension**: An optional Rancher UI extension that allows basic virtual machine management, for example, starting/stopping of virtual machines as well as accessing the console.
- **Containerized Data Importer (CDI)**: An additional component that enables persistent-storage integration for KubeVirt, providing capabilities for virtual machines to use existing Kubernetes storage back-ends for data, but also allowing users to import or clone data volumes for virtual machines.

Each of these Helm charts is versioned according to the SUSE Telco Cloud release you are currently using. For production/supported usage, employ the artifacts that can be found in the SUSE Registry.

First, ensure that your `kubectl` access is working:

```
$ kubectl get nodes
```

This should show something similar to the following:

```
NAME                    STATUS   ROLES                       AGE     VERSION
node1.edge.rdo.wales    Ready    control-plane,etcd,master   4h20m   v1.30.5+rke2r1
node2.edge.rdo.wales    Ready    control-plane,etcd,master   4h15m   v1.30.5+rke2r1
node3.edge.rdo.wales    Ready    control-plane,etcd,master   4h15m   v1.30.5+rke2r1
```

Now you can proceed to install the **KubeVirt** and **Containerized Data Importer (CDI)** Helm charts:

```
$ helm install kubevirt oci://registry.suse.com/edge/charts/kubevirt --namespace
 kubevirt-system --create-namespace
$ helm install cdi oci://registry.suse.com/edge/charts/cdi --namespace cdi-system --
create-namespace
```

In a few minutes, you should have all KubeVirt and CDI components deployed. You can validate this by checking all the deployed resources in the `kubevirt-system` and `cdi-system` namespace.

Verify KubeVirt resources:

```
$ kubectl get all -n kubevirt-system
```

This should show something similar to the following:

```
NAME                                  READY   STATUS    RESTARTS      AGE
pod/virt-operator-5fbcf48d58-p7xpm    1/1     Running   0             2m24s
pod/virt-operator-5fbcf48d58-wnf6s    1/1     Running   0             2m24s
pod/virt-handler-t594x                1/1     Running   0             93s
pod/virt-controller-5f84c69884-cwjvd  1/1     Running   1 (64s ago)   93s
pod/virt-controller-5f84c69884-xxw6q  1/1     Running   1 (64s ago)   93s
pod/virt-api-7dfc54cf95-v8kcl         1/1     Running   1 (59s ago)   118s

NAME                                 TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
 AGE
service/kubevirt-prometheus-metrics  ClusterIP   None            <none>        443/TCP
 2m1s
service/virt-api                     ClusterIP   10.43.56.140    <none>        443/TCP
 2m1s
service/kubevirt-operator-webhook    ClusterIP   10.43.201.121   <none>        443/TCP
 2m1s
service/virt-exportproxy             ClusterIP   10.43.83.23     <none>        443/TCP
 2m1s
```

```
NAME                          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE
 SELECTOR            AGE
daemonset.apps/virt-handler   1         1         1       1            1
 kubernetes.io/os=linux   93s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/virt-operator       2/2     2            2           2m24s
deployment.apps/virt-controller     2/2     2            2           93s
deployment.apps/virt-api            1/1     1            1           118s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/virt-operator-5fbcf48d58      2         2         2       2m24s
replicaset.apps/virt-controller-5f84c69884    2         2         2       93s
replicaset.apps/virt-api-7dfc54cf95           1         1         1       118s

NAME                              AGE     PHASE
kubevirt.kubevirt.io/kubevirt     2m24s   Deployed
```

Verify CDI resources:

```
$ kubectl get all -n cdi-system
```

This should show something similar to the following:

```
NAME                                   READY   STATUS    RESTARTS   AGE
pod/cdi-operator-55c74f4b86-692xb      1/1     Running   0          2m24s
pod/cdi-apiserver-db465b888-62lvr      1/1     Running   0          2m21s
pod/cdi-deployment-56c7d74995-mgkfn    1/1     Running   0          2m21s
pod/cdi-uploadproxy-7d7b94b968-6kxc2   1/1     Running   0          2m22s

NAME                              TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/cdi-uploadproxy           ClusterIP   10.43.117.7    <none>        443/TCP
 2m22s
service/cdi-api                   ClusterIP   10.43.20.101   <none>        443/TCP
 2m22s
service/cdi-prometheus-metrics    ClusterIP   10.43.39.153   <none>        8080/TCP
 2m21s

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/cdi-operator        1/1     1            1           2m24s
deployment.apps/cdi-apiserver       1/1     1            1           2m22s
deployment.apps/cdi-deployment      1/1     1            1           2m21s
deployment.apps/cdi-uploadproxy     1/1     1            1           2m22s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/cdi-operator-55c74f4b86       1         1         1       2m24s
replicaset.apps/cdi-apiserver-db465b888       1         1         1       2m21s
replicaset.apps/cdi-deployment-56c7d74995     1         1         1       2m21s
```

```
replicaset.apps/cdi-uploadproxy-7d7b94b968   1         1         1       2m22s
```

To verify that the `VirtualMachine` custom resource definitions (CRDs) are deployed, you can validate with:

```
$ kubectl explain virtualmachine
```

This should print out the definition of the `VirtualMachine` object, which should print as follows:

```
GROUP:      kubevirt.io
KIND:       VirtualMachine
VERSION:    v1

DESCRIPTION:
    VirtualMachine handles the VirtualMachines that are not running or are in a
    stopped state The VirtualMachine contains the template to create the
    VirtualMachineInstance. It also mirrors the running state of the created
    VirtualMachineInstance in its status.
(snip)
```

19.4 Deploying virtual machines

Now that KubeVirt and CDI are deployed, let us define a simple virtual machine based on openSUSE Tumbleweed (https://get.opensuse.org/tumbleweed/). This virtual machine has the most simple of configurations, using standard "pod networking" for a networking configuration identical to any other pod. It also employs non-persistent storage, ensuring the storage is ephemeral, just like in any container that does not have a PVC (https://kubernetes.io/docs/concepts/storage/persistent-volumes/).

```
$ cat <<EOF > user-data.yaml
#cloud-config
disable_root: false
ssh_pwauth: True
users:
  - default
  - name: suse
    groups: sudo
    shell: /bin/bash
    sudo:  ALL=(ALL) NOPASSWD:ALL
    lock_passwd: False
    plain_text_passwd: 'suse'
EOF
$ kubectl apply -f - <<EOF
apiVersion: kubevirt.io/v1
kind: VirtualMachine
```

```
metadata:
  name: tumbleweed
  namespace: default
spec:
  runStrategy: Always
  template:
    spec:
      domain:
        devices: {}
        machine:
          type: q35
        memory:
          guest: 2Gi
        resources: {}
      volumes:
      - containerDisk:
          image: quay.io/containerdisks/opensuse-tumbleweed:1.0.0
        name: tumbleweed-containerdisk-0
      - cloudInitNoCloud:
          userDataBase64: $(cat user-data.yaml | base64 -w 0)
        name: cloudinitdisk
EOF
```

This should print that a `VirtualMachine` was created:

```
virtualmachine.kubevirt.io/tumbleweed created
```

This `VirtualMachine` definition is minimal, specifying little about the configuration. It simply outlines that it is a machine type "q35 (https://wiki.qemu.org/Features/Q35)" with 2 GB of memory that uses a disk image based on an ephemeral `containerDisk` (that is, a disk image that is stored in a container image from a remote image repository), and specifies a base64 encoded cloudInit disk, which we only use for user creation and password enforcement at boot time (use `base64 -d` to decode it).

Note

This virtual machine image is only for testing. The image is not officially supported and is only meant as a documentation example.

This machine takes a few minutes to boot as it needs to download the openSUSE Tumbleweed disk image, but once it has done so, you can view further details about the virtual machine by checking the virtual machine information:

```
$ kubectl get vmi
```

This should print the node that the virtual machine was started on, and the IP address of the virtual machine. Remember, since it uses pod networking, the reported IP address will be just like any other pod, and routable as such:

```
NAME          AGE     PHASE     IP           NODENAME                READY
tumbleweed    4m24s   Running   10.42.2.98   node3.edge.rdo.wales    True
```

When running these commands on the Kubernetes cluster nodes themselves, with a CNI that routes traffic directly to pods (for example, Cilium), you should be able to `ssh` directly to the machine itself. Substitute the following IP address with the one that was assigned to your virtual machine:

```
$ ssh suse@10.42.2.98
(password is "suse")
```

Once you are in this virtual machine, you can play around, but remember that it is limited in terms of resources, and only has 1 GB disk space. When you are finished, `Ctrl-D` or `exit` to disconnect from the SSH session.

The virtual machine process is still wrapped in a standard Kubernetes pod. The `VirtualMachine` CRD is a representation of the desired virtual machine, but the process in which the virtual machine is actually started is via the `virt-launcher` pod, a standard Kubernetes pod, just like any other application. For every virtual machine started, you can see there is a `virt-launcher` pod:

```
$ kubectl get pods
```

This should then show the one `virt-launcher` pod for the Tumbleweed machine that we have defined:

```
NAME                             READY   STATUS    RESTARTS   AGE
virt-launcher-tumbleweed-8gcn4   3/3     Running   0          10m
```

If we take a look into this `virt-launcher` pod, you see it is executing `libvirt` and `qemu-kvm` processes. We can enter the pod itself and have a look under the covers, noting that you need to adapt the following command for your pod name:

```
$ kubectl exec -it virt-launcher-tumbleweed-8gcn4 -- bash
```

Once you are in the pod, try running `virsh` commands along with looking at the processes. You will see the `qemu-system-x86_64` binary running, along with certain processes for monitoring the virtual machine. You will also see the location of the disk image and how the networking is plugged (as a tap device):

```
qemu@tumbleweed:/> ps ax
  PID TTY      STAT   TIME COMMAND
    1 ?        Ssl    0:00 /usr/bin/virt-launcher-monitor --qemu-timeout 269s --name
 tumbleweed --uid b9655c11-38f7-4fa8-8f5d-bfe987dab42c --namespace default --kubevirt-
share-dir /var/run/kubevirt --ephemeral-disk-dir /var/run/kubevirt-ephemeral-disks --
container-disk-dir /var/run/kube
```

```
   12 ?        Sl     0:01 /usr/bin/virt-launcher --qemu-timeout 269s --name tumbleweed
 --uid b9655c11-38f7-4fa8-8f5d-bfe987dab42c --namespace default --kubevirt-share-dir /
var/run/kubevirt --ephemeral-disk-dir /var/run/kubevirt-ephemeral-disks --container-disk-
dir /var/run/kubevirt/con
   24 ?        Sl     0:00 /usr/sbin/virtlogd -f /etc/libvirt/virtlogd.conf
   25 ?        Sl     0:01 /usr/sbin/virtqemud -f /var/run/libvirt/virtqemud.conf
   83 ?        Sl     0:31 /usr/bin/qemu-system-x86_64 -name
 guest=default_tumbleweed,debug-threads=on -S -object {"qom-
type":"secret","id":"masterKey0","format":"raw","file":"/var/run/kubevirt-private/
libvirt/qemu/lib/domain-1-default_tumbleweed/master-key.aes"} -machine pc-q35-7.1,usb
  286 pts/0    Ss     0:00 bash
  320 pts/0    R+     0:00 ps ax

qemu@tumbleweed:/> virsh list --all
 Id   Name                 State
-----------------------------------
 1    default_tumbleweed   running

qemu@tumbleweed:/> virsh domblklist 1
 Target   Source
-------------------------------------------------------------------------------------------
 sda      /var/run/kubevirt-ephemeral-disks/disk-data/tumbleweed-containerdisk-0/
disk.qcow2
 sdb      /var/run/kubevirt-ephemeral-disks/cloud-init-data/default/tumbleweed/
noCloud.iso

qemu@tumbleweed:/> virsh domiflist 1
 Interface   Type       Source   Model                     MAC
-----------------------------------------------------------------------------
 tap0        ethernet   -        virtio-non-transitional   e6:e9:1a:05:c0:92

qemu@tumbleweed:/> exit
exit
```

Finally, let us delete this virtual machine to clean up:

```
$ kubectl delete vm/tumbleweed
virtualmachine.kubevirt.io "tumbleweed" deleted
```

19.5 Using virtctl

Along with the standard Kubernetes CLI tooling, that is, `kubectl`, KubeVirt comes with an accompanying CLI utility that allows you to interface with your cluster in a way that bridges some gaps between the virtualization world and the world that Kubernetes was designed for. For example, the `virtctl` tool

provides the capability of managing the lifecycle of virtual machines (starting, stopping, restarting, etc.), providing access to the virtual consoles, uploading virtual machine images, as well as interfacing with Kubernetes constructs such as services, without using the API or CRDs directly.

Let us download the latest stable version of the `virtctl` tool:

```
$ export VERSION=v1.8.3
$ wget https://github.com/kubevirt/kubevirt/releases/download/$VERSION/virtctl-$VERSION-
linux-amd64
```

If you are using a different architecture or a non-Linux machine, you can find other releases here (https://github.com/kubevirt/kubevirt/releases). You need to make this executable before proceeding, and it may be useful to move it to a location within your `$PATH`:

```
$ mv virtctl-$VERSION-linux-amd64 /usr/local/bin/virtctl
$ chmod a+x /usr/local/bin/virtctl
```

You can then use the `virtctl` command-line tool to create virtual machines. Let us replicate our previous virtual machine, noting that we are piping the output directly into `kubectl apply`:

```
$ cat <<EOF >  user-data.yaml
#cloud-config
disable_root: false
ssh_pwauth: True
users:
  - default
  - name: suse
    groups: sudo
    shell: /bin/bash
    sudo:  ALL=(ALL) NOPASSWD:ALL
    lock_passwd: False
    plain_text_passwd: 'suse'
EOF
$ alias virtctl=echo
$ virtctl create vm --name virtctl-example --memory=1Gi \
    --volume-containerdisk=src:quay.io/containerdisks/opensuse-tumbleweed:1.0.0 \
    --cloud-init-user-data "$(cat user-data.yaml | base64 -w 0)"
```

This should then show the virtual machine running (it should start a lot quicker this time given that the container image will be cached):

```
$ kubectl get vmi
NAME              AGE   PHASE     IP           NODENAME               READY
virtctl-example   52s   Running   10.42.2.29   node3.edge.rdo.wales   True
```

Now we can use `virtctl` to connect directly to the virtual machine:

```
$ virtctl ssh suse@virtctl-example
```

```
(password is "suse" - Ctrl-D to exit)
```

There are plenty of other commands that can be used by `virtctl`. For example, `virtctl console` can give you access to the serial console if networking is not working, and you can use `virtctl guestosinfo` to get comprehensive OS information, subject to the guest having the `qemu-guest-agent` installed and running.

Finally, let us pause and resume the virtual machine:

```
$ virtctl pause vm virtctl-example
VMI virtctl-example was scheduled to pause
```

You find that the `VirtualMachine` object shows as **Paused** and the `VirtualMachineInstance` object shows as **Running** but **READY=False**:

```
$ kubectl get vm
NAME              AGE     STATUS   READY
virtctl-example   8m14s   Paused   False

$ kubectl get vmi
NAME              AGE     PHASE     IP           NODENAME               READY
virtctl-example   8m15s   Running   10.42.2.29   node3.edge.rdo.wales   False
```

You also find that you can no longer connect to the virtual machine:

```
$ virtctl ssh suse@virtctl-example
can't access VMI virtctl-example: Operation cannot be fulfilled on
 virtualmachineinstance.kubevirt.io "virtctl-example": VMI is paused
```

Let us resume the virtual machine and try again:

```
$ virtctl unpause vm virtctl-example
VMI virtctl-example was scheduled to unpause
```

Now we should be able to re-establish a connection:

```
$ virtctl ssh suse@virtctl-example
suse@vmi/virtctl-example.default's password:
suse@virtctl-example:~> exit
logout
```

Finally, let us remove the virtual machine:

```
$ kubectl delete vm/virtctl-example
virtualmachine.kubevirt.io "virtctl-example" deleted
```

19.6 Simple ingress networking

In this section, we show how you can expose virtual machines as standard Kubernetes services and make them available via the Kubernetes ingress service, for example, Traefik with RKE2 (https://docs.rke2.io/networking/networking_services#ingress-controller) or Traefik with K3s (https://docs.k3s.io/networking/networking-services#traefik-ingress-controller). This document assumes that these components are already configured appropriately and that you have an appropriate DNS pointer, for example, via a wild card, to point at your Kubernetes server nodes or your ingress virtual IP for proper ingress resolution.

> **Note**
>
> In SUSE Telco Cloud 3.1+, if you are using K3s in a multi-server node configuration, you might have needed to configure a MetalLB-based VIP for Ingress; this is not required for RKE2.

In the example environment, another openSUSE Tumbleweed virtual machine is deployed, cloud-init is used to install NGINX as a simple Web server at boot time, and a simple message is configured to be returned to verify that it works as expected when a call is made. To see how this is done, simply `base64 -d` the cloud-init section in the output below.

Let us create this virtual machine now:

```
$ cat <<EOF > user-data.yaml
#cloud-config
disable_root: false
ssh_pwauth: True
users:
  - default
  - name: suse
    groups: sudo
    shell: /bin/bash
    sudo:  ALL=(ALL) NOPASSWD:ALL
    lock_passwd: False
    plain_text_passwd: 'suse'
EOF
$ kubectl apply -f - <<EOF
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: ingress-example
  namespace: default
```

```
spec:
  runStrategy: Always
  template:
    metadata:
      labels:
        app: nginx
    spec:
      domain:
        devices: {}
        machine:
          type: q35
        memory:
          guest: 2Gi
        resources: {}
      volumes:
      - containerDisk:
          image: quay.io/containerdisks/opensuse-tumbleweed:1.0.0
        name: tumbleweed-containerdisk-0
      - cloudInitNoCloud:
          userDataBase64: $(cat user-data.yaml | base64 -w 0)
        name: cloudinitdisk
EOF
```

When this virtual machine has successfully started, we can use the `virtctl` command to expose the `VirtualMachineInstance` with an external port of `8080` and a target port of `80` (where NGINX listens by default). We use the `virtctl` command here as it understands the mapping between the virtual machine object and the pod. This creates a new service for us:

```
$ virtctl expose vmi ingress-example --port=8080 --target-port=80 --name=ingress-example
Service ingress-example successfully exposed for vmi ingress-example
```

We will then have an appropriate service automatically created:

```
$ kubectl get svc/ingress-example
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP       PORT(S)
          AGE
ingress-example   ClusterIP      10.43.217.19    <none>            8080/TCP
          9s
```

Next, if you then use `kubectl create ingress`, we can create an ingress object that points to this service. Adapt the URL (known as the "host" in the ingress (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_create/kubectl_create_ingress/) object) here to match your DNS configuration and ensure that you point it to port `8080`:

```
$ kubectl create ingress ingress-example --rule=ingress-example.suse.local/=ingress-
example:8080
```

With DNS being configured correctly, you should be able to curl the URL immediately:

```
$ curl ingress-example.suse.local
It works!
```

Let us clean up by removing this virtual machine and its service and ingress resources:

```
$ kubectl delete vm/ingress-example svc/ingress-example ingress/ingress-example
virtualmachine.kubevirt.io "ingress-example" deleted
service "ingress-example" deleted
ingress.networking.k8s.io "ingress-example" deleted
```

19.7 Using the Rancher UI extension

SUSE Telco Cloud Virtualization provides a UI extension for Rancher Manager, enabling basic virtual machine management using the Rancher dashboard UI.

19.7.1 Installation

See Rancher Dashboard Extensions (*Chapter 7, Rancher Dashboard Extensions*) for installation guidance.

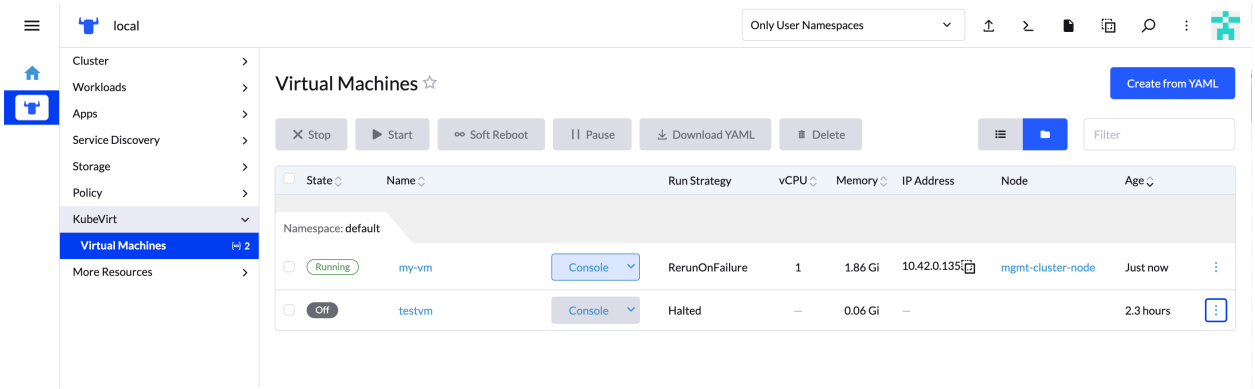
19.7.2 Using KubeVirt Rancher Dashboard Extension

The extension introduces a new **KubeVirt** section to the Cluster Explorer. This section is added to any managed cluster which has KubeVirt installed.

The extension allows you to directly interact with KubeVirt Virtual Machine resources to manage Virtual Machines lifecycle.

19.7.2.1 Creating a virtual machine

1. Navigate to **Cluster Explorer** clicking KubeVirt-enabled managed cluster in the left navigation.
2. Navigate to **KubeVirt** > **Virtual Machines** page and click `Create from YAML` in the upper right of the screen.
3. Fill in or paste a virtual machine definition and press `Create`. Use virtual machine definition from Deploying Virtual Machines section as an inspiration.



19.7.2.2 Virtual Machine Actions

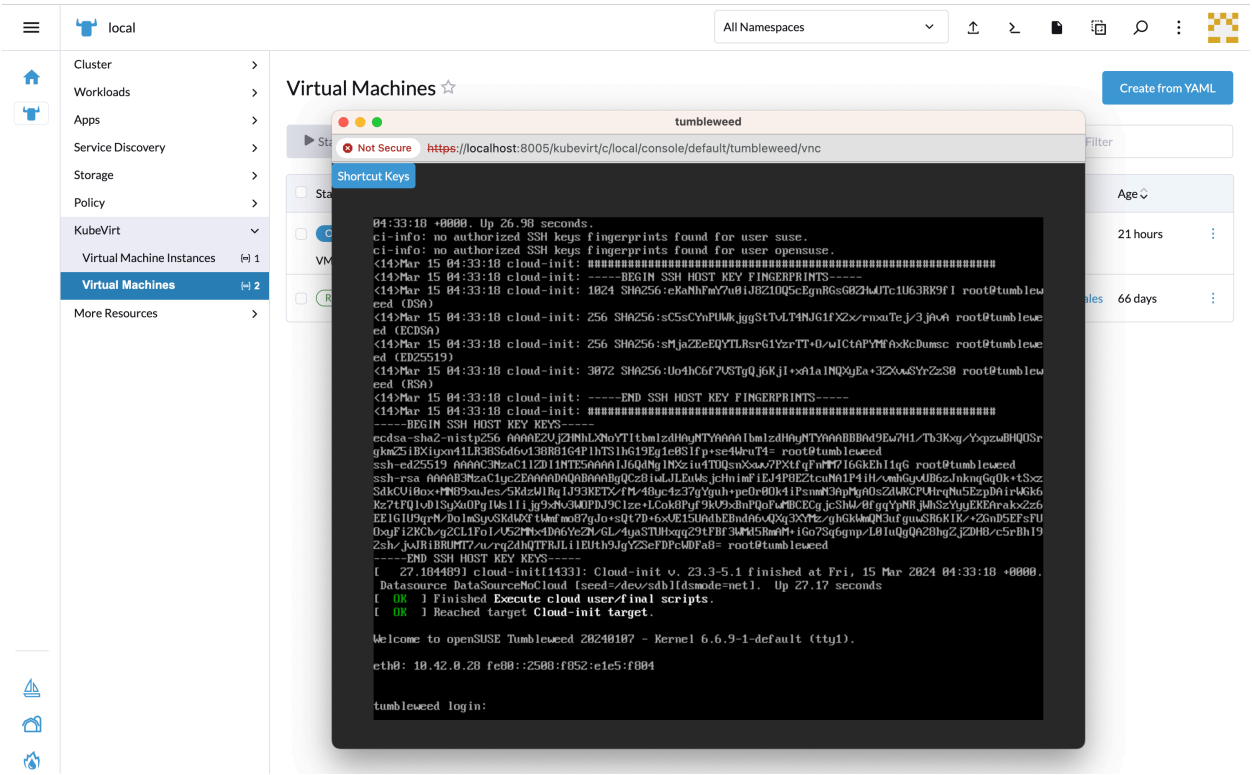
You can use the action menu accessed from the # drop-down list to the right of each virtual machine to perform start, stop, pause or soft reboot actions. Alternatively you can also use group actions at the top of the list by selecting virtual machines to perform the action on.

Performing the actions may have an effect on Virtual Machine Run Strategy. See the table in KubeVirt documentation (https://kubevirt.io/user-guide/compute/run_strategies/#virtctl) for more details.

19.7.2.3 Accessing virtual machine console

The "Virtual machines" list provides a `Console` drop-down list that allows to connect to the machine using **VNC or Serial Console**. This action is only available to running machines.

In some cases, it takes a short while before the console is accessible on a freshly started virtual machine.



19.8 Installing with Edge Image Builder

SUSE Telco Cloud is using *Chapter 12, Edge Image Builder* in order to customize base SUSE Linux Micro OS images. Follow *Section 64.9, "KubeVirt and CDI Installation"* for an air-gapped installation of both KubeVirt and CDI on top of Kubernetes clusters provisioned by EIB.

20 System Upgrade Controller

See the System Upgrade Controller documentation (https://github.com/rancher/system-upgrade-controller).

> The System Upgrade Controller (SUC) aims to provide a general-purpose, Kubernetes-native upgrade controller (for nodes). It introduces a new CRD, the Plan, for defining any and all of your upgrade policies/requirements. A Plan is an outstanding intent to mutate nodes in your cluster.

20.1 How does SUSE Telco Cloud use System Upgrade Controller?

SUSE Telco Cloud uses `SUC` to facilitate various "Day 2" operations related to OS and Kubernetes version upgrades on management and downstream clusters.

"Day 2" operations are defined through `SUC Plans`. Based on these plans, `SUC` deploys workloads on each node to execute the respective "Day 2" operation.

`SUC` is also used within the *Chapter 21, Upgrade Controller*. To learn more about the key differences between SUC and the Upgrade Controller, see *Section 21.2, "Upgrade Controller vs System Upgrade Controller"*.

20.2 Installing the System Upgrade Controller

Important

Starting with Rancher v2.10.0 (https://github.com/rancher/rancher/releases/tag/v2.10.0), the `System Upgrade Controller` is installed automatically.

Follow the steps below **only** if your environment is **not** managed by Rancher, or if your Rancher version is lesser than `v2.10.0`.

We recommend that you install SUC through Fleet (*Chapter 9, Fleet*) located in the suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) repository.

Note

The resources offered by the `suse-edge/fleet-examples` repository **must** always be used from a valid fleet-examples release (https://github.com/suse-edge/fleet-examples/releases). To determine which release you need to use, refer to the Release Notes (*Chapter 78, Release Notes*).

If you are unable to use Fleet for the installation of SUC, you can install it through Rancher's Helm chart repository, or incorporate the Rancher's Helm chart in your own third-party GitOps workflow.

This section covers:

- Fleet installation (*Section 20.2.1, "System Upgrade Controller Fleet installation"*)
- Helm installation (*Section 20.2.2, "System Upgrade Controller Helm installation"*)

20.2.1 System Upgrade Controller Fleet installation

Using Fleet, there are two possible resources that can be used to deploy SUC:

- GitRepo (https://fleet.rancher.io/ref-gitrepo) resource - for use cases where an external/local Git server is available. For installation instructions, see System Upgrade Controller installation - GitRepo (*Section 20.2.1.1, "System Upgrade Controller installation - GitRepo"*).
- Bundle (https://fleet.rancher.io/bundle-add) resource - for air-gapped use cases that do not support a local Git server option. For installation instructions, see System Upgrade Controller installation - Bundle (*Section 20.2.1.2, "System Upgrade Controller installation - Bundle"*).

20.2.1.1 System Upgrade Controller installation - GitRepo

Note

This process can also be done through the Rancher UI, if such is available. For more information, see Accessing Fleet in the Rancher UI (https://ranchermanager.docs.rancher.com/v2.15/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui).

In your **management** cluster:

1. Determine on which clusters you want to deploy SUC. This is done by deploying a SUC `GitRepo` resource in the correct Fleet workspace on your **management** cluster. By default, Fleet has two workspaces:

   - `fleet-local` - for resources that need to be deployed on the **management** cluster.
   - `fleet-default` - for resources that need to be deployed on **downstream** clusters.

     For more information on Fleet workspaces, see the upstream (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups) documentation.

2. Deploy the `GitRepo` resource:

   - To deploy SUC on your management cluster:

     ```
     kubectl apply -n fleet-local -f - <<EOF
     apiVersion: fleet.cattle.io/v1alpha1
     kind: GitRepo
     metadata:
       name: system-upgrade-controller
     spec:
       revision: release-3.7.0
       paths:
       - fleets/day2/system-upgrade-controller
       repo: https://github.com/suse-edge/fleet-examples.git
     EOF
     ```

   - To deploy SUC on your downstream clusters:

     > **Note**
     >
     > Before deploying the resource below, you **must** provide a valid `targets` configuration, so that Fleet knows on which downstream clusters to deploy your resource. For information on how to map to downstream clusters, see Mapping to Downstream Clusters (https://fleet.rancher.io/gitrepo-targets).

     ```
     kubectl apply -n fleet-default -f - <<EOF
     apiVersion: fleet.cattle.io/v1alpha1
     kind: GitRepo
     metadata:
       name: system-upgrade-controller
     ```

```
spec:
  revision: release-3.7.0
  paths:
  - fleets/day2/system-upgrade-controller
  repo: https://github.com/suse-edge/fleet-examples.git
  targets:
  - clusterSelector: CHANGEME
  # Example matching all clusters:
  # targets:
  # - clusterSelector: {}
EOF
```

3. Validate that the `GitRepo` resource is deployed:

```
# Namespace will vary based on where you want to deploy SUC
kubectl get gitrepo system-upgrade-controller -n <fleet-local/fleet-default>

NAME                        REPO                                                COMMIT
          BUNDLEDEPLOYMENTS-READY   STATUS
system-upgrade-controller   https://github.com/suse-edge/fleet-examples.git
 release-3.7.0   1/1
```

4. Validate the System Upgrade Controller deployment:

```
kubectl get deployment system-upgrade-controller -n cattle-system
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
system-upgrade-controller   1/1     1            1           2m20s
```

20.2.1.2 System Upgrade Controller installation - Bundle

This section illustrates how to build and deploy a `Bundle` resource from a standard Fleet configuration using the fleet-cli (https://fleet.rancher.io/cli/fleet-cli/fleet).

1. On a machine with network access download the `fleet-cli`:

> **Note**
>
> Make sure that the version of the fleet-cli you download matches the version of Fleet that has been deployed on your cluster.

- For Mac users there is a fleet-cli (https://formulae.brew.sh/formula/fleet-cli) Homebrew Formulae.
- For Linux and Windows users the binaries are present as **assets** to each Fleet release (https://github.com/rancher/fleet/releases).
  - Linux AMD:

    ```
    curl -L -o fleet-cli https://github.com/rancher/fleet/releases/download/v0.16.1/fleet-linux-amd64
    ```

  - Linux ARM:

    ```
    curl -L -o fleet-cli https://github.com/rancher/fleet/releases/download/v0.16.1/fleet-linux-arm64
    ```

2. Make `fleet-cli` executable:

   ```
   chmod +x fleet-cli
   ```

3. Clone the `suse-edge/fleet-examples` release (https://github.com/suse-edge/fleet-examples/releases) that you wish to use:

   ```
   git clone -b release-3.7.0 https://github.com/suse-edge/fleet-examples.git
   ```

4. Navigate to the SUC fleet, located in the `fleet-examples` repo:

   ```
   cd fleet-examples/fleets/day2/system-upgrade-controller
   ```

5. Determine on which clusters you want to deploy SUC. This is done by deploying the SUC Bundle in the correct Fleet workspace inside your management cluster. By default, Fleet has two workspaces:
   - `fleet-local` - for resources that need to be deployed on the **management** cluster.
   - `fleet-default` - for resources that need to be deployed on **downstream** clusters.
     For more information on Fleet workspaces, see the upstream (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups) documentation.

6. If you intend to deploy SUC only on downstream clusters, create a `targets.yaml` file that matches the specific clusters:

   ```
   cat > targets.yaml <<EOF
   ```

```
targets:
- clusterSelector: CHANGEME
EOF
```

For information on how to map to downstream clusters, see Mapping to Downstream Clusters (https://fleet.rancher.io/gitrepo-targets)

7. Proceed to building the Bundle:

> **Note**
>
> Make sure you did **not** download the fleet-cli in the `fleet-examples/fleets/day2/system-upgrade-controller` directory, otherwise it will be packaged with the Bundle, which is not advised.

- To deploy SUC on your management cluster, execute:

```
fleet-cli apply --compress -n fleet-local -o - system-upgrade-controller . >
 system-upgrade-controller-bundle.yaml
```

- To deploy SUC on your downstream clusters, execute:

```
fleet-cli apply --compress --targets-file=targets.yaml -n fleet-default -o -
 system-upgrade-controller . > system-upgrade-controller-bundle.yaml
```

For more information about this process, see Convert a Helm Chart into a Bundle (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle).
For more information about the `fleet-cli apply` command, see fleet apply (https://fleet.rancher.io/cli/fleet-cli/fleet_apply).

8. Transfer the `system-upgrade-controller-bundle.yaml` bundle to your management cluster machine:

```
scp system-upgrade-controller-bundle.yaml <machine-address>:<filesystem-path>
```

9. On your management cluster, deploy the `system-upgrade-controller-bundle.yaml` Bundle:

```
kubectl apply -f system-upgrade-controller-bundle.yaml
```

10. On your management cluster, validate that the Bundle is deployed:

```
# Namespace will vary based on where you want to deploy SUC
```

```
kubectl get bundle system-upgrade-controller -n <fleet-local/fleet-default>

NAME                        BUNDLEDEPLOYMENTS-READY   STATUS
system-upgrade-controller   1/1
```

11. Based on the Fleet workspace that you deployed your Bundle to, navigate to the cluster and validate the SUC deployment:

> **Note**
>
> SUC is always deployed in the **cattle-system** namespace.

```
kubectl get deployment system-upgrade-controller -n cattle-system
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
system-upgrade-controller   1/1     1            1           111s
```

20.2.2 System Upgrade Controller Helm installation

1. Add the Rancher chart repository:

```
helm repo add rancher-charts https://charts.rancher.io/
```

2. Deploy the SUC chart:

```
helm install system-upgrade-controller rancher-charts/system-upgrade-controller --
version 110.0.0 --set global.cattle.psp.enabled=false -n cattle-system --create-
namespace
```

This will install SUC version v0.20.1 which is needed by the Edge 3.7 platform.

3. Validate the SUC deployment:

```
kubectl get deployment system-upgrade-controller -n cattle-system
NAME                        READY   UP-TO-DATE   AVAILABLE   AGE
system-upgrade-controller   1/1     1            1           37s
```

20.3 Monitoring System Upgrade Controller Plans

SUC Plans can be viewed in the following ways:

- Through the Rancher UI (*Section 20.3.1, "Monitoring System Upgrade Controller Plans - Rancher UI"*).
- Through manual monitoring (*Section 20.3.2, "Monitoring System Upgrade Controller Plans - Manual"*) inside of the cluster.

Important

Pods deployed for SUC Plans are kept alive **15** minutes after a successful execution. After that they are removed by the corresponding Job that created them. To have access to the Pod's logs after this time period, you should enable logging for your cluster. For information on how to do this in Rancher, see Rancher Integration with Logging Services (https://ranchermanager.docs.rancher.com/v2.15/integrations-in-rancher/logging).

20.3.1 Monitoring System Upgrade Controller Plans - Rancher UI

To check Pod logs for the specific SUC plan:

1. In the upper left corner, **#** → **<your-cluster-name>**
2. Select Workloads → Pods
3. Select the `Only User Namespaces` drop down menu and add the `cattle-system` namespace
4. In the Pod filter bar, write the name for your SUC Plan Pod. The name will be in the following template format: `apply-<plan_name>-on-<node_name>`

   #### Note

   There may be both `Completed` and `Unknown` Pods for a specific SUC Plan. This is expected and happens due to the nature of some of the upgrades.

5. Select the pod that you want to review the logs of and navigate to **#** → **View Logs**

20.3.2 Monitoring System Upgrade Controller Plans - Manual

Note

The below steps assume that `kubectl` has been configured to connect to the cluster where the **SUC Plans** have been deployed to.

1. List deployed **SUC** Plans:

```
kubectl get plans -n cattle-system
```

2. Get Pod for **SUC** Plan:

```
kubectl get pods -l upgrade.cattle.io/plan=<plan_name> -n cattle-system
```

Note

There may be both `Completed` and `Unknown` Pods for a specific SUC Plan. This is expected and happens due to the nature of some of the upgrades.

3. Get logs for the Pod:

```
kubectl logs <pod_name> -n cattle-system
```

21 Upgrade Controller

A Kubernetes controller capable of performing upgrades over the following SUSE Telco Cloud platform components:

- Operating System (SUSE Linux Micro)
- Kubernetes (K3s & RKE2)
- Additional components (Rancher, Elemental, SUSE Security, etc.)

The Upgrade Controller (https://github.com/suse-edge/upgrade-controller) streamlines the upgrade process for the components mentioned above by encapsulating their complexities within a single `user-facing` resource that serves as a **trigger** for the upgrade. Users only need to configure this resource and the `Upgrade Controller` takes care of the rest.

> **Note**
>
> The `Upgrade Controller` currently supports SUSE Telco Cloud platform upgrades only for **non air-gapped management** clusters. Refer to the *Section 21.8, "Known Limitations"* section for more information.

21.1 How does SUSE Telco Cloud use Upgrade Controller?

The **Upgrade Controller** is essential in automating the (formerly manual) "Day 2" operations required to upgrade management clusters from one SUSE Telco Cloud release version to the next.

To achieve this automation, the Upgrade Controller utilizes tools such as the System Upgrade Controller (*Chapter 20, System Upgrade Controller*) and the Helm Controller (https://github.com/k3s-io/helm-controller/).

For further details on how the Upgrade Controller works, see *Section 21.5, "How does the Upgrade Controller work?"*.

For known limitations that the Upgrade Controller has, see *Section 21.8, "Known Limitations"*.

For information on the difference between the Upgrade Controller and the System Upgrade Controller, see *Section 21.2, "Upgrade Controller vs System Upgrade Controller"*.

21.2 Upgrade Controller vs System Upgrade Controller

The System Upgrade Controller (SUC) (*Chapter 20, System Upgrade Controller*) is a general-purpose tool that propagates upgrade instructions to specific Kubernetes nodes.

While it supports some "Day 2" operations for the SUSE Telco Cloud platform, it **does not** cover all of them. Moreover, even for supported operations, users have to manually configure, maintain, and deploy multiple `SUC Plans` — an error-prone process that can lead to unexpected issues.

This led to the need for a tool that **automates** and **abstracts** the complexity of managing various "Day 2" operations for the SUSE Telco Cloud platform. Thus, the `Upgrade Controller` was developed. It simplifies the upgrade process by introducing a single `user-facing resource` that drives the upgrade. Users only need to manage this resource, while the `Upgrade Controller` takes care of the rest.

21.3 Installing the Upgrade Controller

21.3.1 Prerequisites

- Helm (https://helm.sh/docs/intro/install/)
- cert-manager (https://cert-manager.io/docs/installation/helm/)
- System Upgrade Controller (*Section 20.2, "Installing the System Upgrade Controller"*)
- A Kubernetes cluster; either K3s or RKE2

21.3.2 Steps

1. Install the Upgrade Controller Helm chart on your management cluster:

```
helm install upgrade-controller oci://registry.suse.com/edge/charts/upgrade-
controller --version 307.0.4+up0.1.3 --create-namespace --namespace upgrade-
controller-system
```

2. Validate the Upgrade Controller deployment:

```
kubectl get deployment -n upgrade-controller-system
```

3. Validate the Upgrade Controller pod:

```
kubectl get pods -n upgrade-controller-system
```

4. Validate the Upgrade Controller pod logs:

```
kubectl logs <pod_name> -n upgrade-controller-system
```

21.4 Installing the Upgrade Controller via Edge Image Builder

As an alternative to the manual installation described above, it is possible to install the upgrade controller as part of the initial deployment orchestrated by Edge Image Builder (*Chapter 12, Edge Image Builder*).

In this case it is necessary to add the following helm chart configuration to the EIB configuration file:

```
kubernetes:
  helm:
    charts:
      - name: cert-manager
        repositoryName: jetstack
        version: {version-cert-manager}
        targetNamespace: cert-manager
        valuesFile: certmanager-values.yaml
        createNamespace: true
        installationNamespace: kube-system
      - name: upgrade-controller
        version: {version-upgrade-controller-chart}
        repositoryName: suse-edge-charts
        targetNamespace: upgrade-controller-system
        createNamespace: true
        installationNamespace: kube-system
```

21.5 How does the Upgrade Controller work?

In order to perform an Edge release upgrade, the Upgrade Controller introduces two new Kubernetes custom resources (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/):

- UpgradePlan (*Section 21.6.1, "UpgradePlan"*) - created by the user; holds configurations regarding an Edge release upgrade.
- ReleaseManifest (*Section 21.6.2, "ReleaseManifest"*) - created by the Upgrade Controller; holds component versions specific to a particular Edge release version. **This file must not be edited by users.**

The Upgrade Controller proceeds to create a `ReleaseManifest` resource that holds the component data for the Edge release version specified by the user under the `releaseVersion` property in the `UpgradePlan` resource.

Using the component data from the `ReleaseManifest`, the Upgrade Controller proceeds to upgrade the Edge release components in the following order:

1. Operating System (OS) (*Section 21.5.1, "Operating System upgrade"*).
2. Kubernetes (*Section 21.5.2, "Kubernetes upgrade"*).
3. Additional components (*Section 21.5.3, "Additional components upgrades"*).

Note

During the upgrade process, the Upgrade Controller continually outputs upgrade information to the created `UpgradePlan`. For more information on how to track the upgrade process, see Tracking the upgrade process (*Section 21.7, "Tracking the upgrade process"*).

21.5.1 Operating System upgrade

To upgrade the operating system, the Upgrade Controller creates SUC (*Chapter 20, System Upgrade Controller*) Plans that have the following naming template:

- For SUC Plans related to control plane node OS upgrades - `control-plane-<os-name>-<os-version>-<suffix>`.
- For SUC Plans related to worker node OS upgrades - `workers-<os-name>-<os-version>-<suffix>`.

Based on these plans, SUC proceeds to create workloads on each node of the cluster that perform the actual OS upgrade.

Depending on the `ReleaseManifest`, the OS upgrade may include:

- Package only updates - for use-cases where the OS version does not change between Edge releases.
- Full OS migration - for use-cases where the OS version changes between Edge releases.

The upgrade is executed **one** node at a time starting with the control plane nodes first. Only if the control-plane node upgrade finishes will the worker nodes begin to be upgraded.

Note

The Upgrade Controller configures the OS SUC Plans to do perform a drain (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_drain/) of the cluster nodes if the cluster has more than **one** node of the specified type.

For clusters where the control plane nodes are **greater than** one and there is **only one** worker node, a drain will be performed only for the control plane nodes and vice versa.

For information on how to disable node drains altogether, see the UpgradePlan (*Section 21.6.1, "UpgradePlan"*) section.

21.5.2 Kubernetes upgrade

To upgrade the Kubernetes distribution of a cluster, the Upgrade Controller creates SUC (*Chapter 20, System Upgrade Controller*) Plans that have the following naming template:

- For SUC Plans related to control plane node Kubernetes upgrades - `control-plane-<k8s-version>-<suffix>`.
- For SUC Plans related to worker node Kubernetes upgrades - `workers-<k8s-version>-<suffix>`.

Based on these plans, SUC proceeds to create workloads on each node of the cluster that perform the actual Kubernetes upgrade.

The Kubernetes upgrade will happen **one** node at a time starting with the control plane nodes first. Only if the control plane node upgrade finishes will the worker nodes begin to be upgraded.

Note

The Upgrade Controller configures the Kubernetes SUC Plans to perform a drain (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_drain/) of the cluster nodes if the cluster has more than **one** node of the specified type.

For clusters where the control plane nodes are **greater than** one and there is **only one** worker node, a drain will be performed only for the control plane nodes and vice versa.

For information on how to disable node drains altogether, see *Section 21.6.1, "UpgradePlan"*.

21.5.3 Additional components upgrades

Currently, all additional components are installed via Helm charts. For a full list of the components for a specific release, refer to the Release Notes (*Chapter 78, Release Notes*).

For Helm charts deployed through EIB (*Chapter 12, Edge Image Builder*), the Upgrade Controller updates the existing HelmChart CR (https://docs.rke2.io/helm#using-the-helm-crd) of each component.

For Helm charts deployed outside of EIB, the Upgrade Controller creates a `HelmChart` resource for each component.

After the creation/update of the `HelmChart` resource, the Upgrade Controller relies on the helm-controller (https://github.com/k3s-io/helm-controller/) to pick up this change and proceed with the actual component upgrade.

Charts will be upgraded sequentially based on their order in the `ReleaseManifest`. Additional values can also be passed through the `UpgradePlan`. If a chart's version remains unchanged in the new SUSE Telco Cloud release, it will not be upgraded. For more information about this, refer to *Section 21.6.1, "UpgradePlan"*.

21.6 Kubernetes API extensions

Extensions to the Kubernetes API introduced by the Upgrade Controller.

21.6.1 UpgradePlan

The Upgrade Controller introduces a new Kubernetes custom resource (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) called an `UpgradePlan`.

The `UpgradePlan` serves as an instruction mechanism for the Upgrade Controller and it supports the following configurations:

- `releaseVersion` - Edge release version to which the cluster should be upgraded to. The release version must follow semantic (https://semver.org) versioning and should be retrieved from the Release Notes (*Chapter 78, Release Notes*).
- `disableDrain` - **Optional**; instructs the Upgrade Controller on whether to disable node drains (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_drain/). Useful for when you have workloads with Disruption Budgets (https://kubernetes.io/docs/tasks/run-application/configure-pdb/).
  - Example for control plane node drain disablement:

    ```
    spec:
      disableDrain:
        controlPlane: true
    ```

  - Example for control plane and worker node drain disablement:

    ```
    spec:
      disableDrain:
        controlPlane: true
        worker: true
    ```

- `helm` - **Optional**; specifies additional values for components installed via Helm.

> **Warning**
>
> It is only advised to use this field for values that are critical for upgrades. Standard chart value updates should be performed after the respective charts have been upgraded to the next version.

  - Example:

    ```
    spec:
      helm:
      - chart: foo
        values:
    ```

```
      bar: baz
```

21.6.2 ReleaseManifest

The Upgrade Controller introduces a new Kubernetes custom resource (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) called a `ReleaseManifest`.

The `ReleaseManifest` resource is created by the Upgrade Controller and holds component data for **one** specific Edge release version. This means that each Edge release version upgrade will be represented by a different `ReleaseManifest` resource.

> **Warning**
>
> The Release Manifest should always be created by the Upgrade Controller.
>
> It is not advisable to manually create or edit the `ReleaseManifest` resources. Users that decide to do so should do this **at their own risk**.

Component data that the Release Manifest ships include, but is not limited to:

- Operating System data - version, supported architectures, additional upgrade data, etc.
- Kubernetes distribution data - RKE2 (https://docs.rke2.io)/K3s (https://k3s.io) supported versions
- Additional components data - SUSE Helm chart data (location, version, name, etc.)

For an example of how a Release Manifest can look, refer to the upstream (https://github.com/suse-edge/upgrade-controller/blob/main/config/samples/lifecycle_v1alpha1_releasemanifest.yaml) documentation. *Please note that this is just an example and it is not intended to be created as a valid* `ReleaseManifest` *resource.*

21.7 Tracking the upgrade process

This section serves as means to track and debug the upgrade process that the Upgrade Controller initiates once the user creates an `UpgradePlan` resource.

21.7.1 General

General information about the state of the upgrade process can be viewed in the Upgrade Plan's status conditions.

The Upgrade Plan resource's status can be viewed in the following way:

```
kubectl get upgradeplan <upgradeplan_name> -n upgrade-controller-system -o yaml
```

EXAMPLE 21.1: RUNNING UPGRADE PLAN EXAMPLE:

```
apiVersion: lifecycle.suse.com/v1alpha1
kind: UpgradePlan
metadata:
  name: upgrade-plan-mgmt
  namespace: upgrade-controller-system
spec:
  releaseVersion: 3.7
status:
  conditions:
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Control plane nodes are being upgraded
    reason: InProgress
    status: "False"
    type: OSUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Kubernetes upgrade is not yet started
    reason: Pending
    status: Unknown
    type: KubernetesUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Rancher upgrade is not yet started
    reason: Pending
    status: Unknown
    type: RancherUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Longhorn upgrade is not yet started
    reason: Pending
    status: Unknown
    type: LonghornUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: MetalLB upgrade is not yet started
    reason: Pending
    status: Unknown
    type: MetalLBUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: CDI upgrade is not yet started
    reason: Pending
```

```
    status: Unknown
    type: CDIUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: KubeVirt upgrade is not yet started
    reason: Pending
    status: Unknown
    type: KubeVirtUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: NeuVector upgrade is not yet started
    reason: Pending
    status: Unknown
    type: NeuVectorUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: EndpointCopierOperator upgrade is not yet started
    reason: Pending
    status: Unknown
    type: EndpointCopierOperatorUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Elemental upgrade is not yet started
    reason: Pending
    status: Unknown
    type: ElementalUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: SRIOV upgrade is not yet started
    reason: Pending
    status: Unknown
    type: SRIOVUpgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: Metal3 upgrade is not yet started
    reason: Pending
    status: Unknown
    type: Metal3Upgraded
  - lastTransitionTime: "2024-10-01T06:26:27Z"
    message: RancherTurtles upgrade is not yet started
    reason: Pending
    status: Unknown
    type: RancherTurtlesUpgraded
  observedGeneration: 1
  sucNameSuffix: 90315a2b6d
```

Here you can view every component that the Upgrade Controller will try to schedule an upgrade for. Each condition follows the below template:

- `lastTransitionTime` - the last time that this component condition has transitioned from one status to another.
- `message` - message that indicates the current upgrade state of the specific component condition.

- `reason` - the current upgrade state of the specific component condition. Possible `reasons` include:
  - `Succeeded` - upgrade of the specific component is successful.
  - `Failed` - upgrade of the specific component has failed.
  - `InProgress` - upgrade of the specific component is currently in progress.
  - `Pending` - upgrade of the specific component is not yet scheduled.
  - `Skipped` - specific component is not found on the cluster, so its upgrade will be skipped.
  - `Error` - specific component has encountered a transient error.
- `status` - status of the current condition `type`, one of `True`, `False`, `Unknown`.
- `type` - indicator for the currently upgraded component.

The Upgrade Controller creates SUC Plans for component conditions of type `OSUpgraded` and `KubernetesUpgraded`. To further track the SUC Plans created for these components, refer to *Section 20.3, "Monitoring System Upgrade Controller Plans"*.

All other component condition types can be further tracked by viewing the resources created for them by the helm-controller (https://github.com/k3s-io/helm-controller/). For more information, see *Section 21.7.2, "Helm Controller"*.

An Upgrade Plan scheduled by the Upgrade Controller can be marked as `successful` once:

1. There are no `Pending` or `InProgress` component conditions.
2. The `lastSuccessfulReleaseVersion` property points to the `releaseVersion` that is specified in the Upgrade Plan's configuration. *This property is added to the Upgrade Plan's status by the Upgrade Controller once the upgrade process is successful.*

EXAMPLE 21.2: SUCCESSFUL `UpgradePlan` EXAMPLE:

```
apiVersion: lifecycle.suse.com/v1alpha1
kind: UpgradePlan
metadata:
  name: upgrade-plan-mgmt
  namespace: upgrade-controller-system
spec:
  releaseVersion: 3.7
status:
  conditions:
  - lastTransitionTime: "2024-10-01T06:26:48Z"
    message: All cluster nodes are upgraded
```

```
    reason: Succeeded
    status: "True"
    type: OSUpgraded
  - lastTransitionTime: "2024-10-01T06:26:59Z"
    message: All cluster nodes are upgraded
    reason: Succeeded
    status: "True"
    type: KubernetesUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart rancher upgrade succeeded
    reason: Succeeded
    status: "True"
    type: RancherUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart longhorn is not installed
    reason: Skipped
    status: "False"
    type: LonghornUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Specified version of chart metallb is already installed
    reason: Skipped
    status: "False"
    type: MetalLBUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart cdi is not installed
    reason: Skipped
    status: "False"
    type: CDIUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart kubevirt is not installed
    reason: Skipped
    status: "False"
    type: KubeVirtUpgraded
  - lastTransitionTime: "2024-10-01T06:27:13Z"
    message: Chart neuvector-crd is not installed
    reason: Skipped
    status: "False"
    type: NeuVectorUpgraded
  - lastTransitionTime: "2024-10-01T06:27:14Z"
    message: Specified version of chart endpoint-copier-operator is already installed
    reason: Skipped
    status: "False"
    type: EndpointCopierOperatorUpgraded
  - lastTransitionTime: "2024-10-01T06:27:14Z"
    message: Chart elemental-operator upgrade succeeded
    reason: Succeeded
    status: "True"
```

```
    type: ElementalUpgraded
  - lastTransitionTime: "2024-10-01T06:27:15Z"
    message: Chart sriov-crd is not installed
    reason: Skipped
    status: "False"
    type: SRIOVUpgraded
  - lastTransitionTime: "2024-10-01T06:27:19Z"
    message: Chart metal3 is not installed
    reason: Skipped
    status: "False"
    type: Metal3Upgraded
  - lastTransitionTime: "2024-10-01T06:27:27Z"
    message: Chart rancher-turtles is not installed
    reason: Skipped
    status: "False"
    type: RancherTurtlesUpgraded
  lastSuccessfulReleaseVersion: 3.7
  observedGeneration: 1
  sucNameSuffix: 90315a2b6d
```

21.7.2 Helm Controller

This section covers how to track resources created by the helm-controller (https://github.com/k3s-io/helm-controller/).

Note

The below steps assume that `kubectl` has been configured to connect to the cluster where the Upgrade Controller has been deployed to.

1. Locate the `HelmChart` resource for the specific component:

```
kubectl get helmcharts -n kube-system
```

2. Using the name of the `HelmChart` resource, locate the upgrade Pod that was created by the `helm-controller`:

```
kubectl get pods -l helmcharts.helm.cattle.io/chart=<helmchart_name> -n kube-system

# Example for Rancher
kubectl get pods -l helmcharts.helm.cattle.io/chart=rancher -n kube-system
NAME                          READY   STATUS      RESTARTS   AGE
```

```
helm-install-rancher-tv9wn   0/1     Completed   0          16m
```

3. View the logs of the component specific pod:

```
kubectl logs <pod_name> -n kube-system
```

21.8 Known Limitations

- The Upgrade Controller expects any additional SUSE Telco Cloud Helm charts that are deployed through EIB (*Chapter 12, Edge Image Builder*) to have their HelmChart CR (https://docs.rke2.io/helm#using-the-helm-crd) deployed in the `kube-system` namespace. To do this, configure the `installationNamespace` property in your EIB definition file. For more information, see the upstream (https://github.com/suse-edge/edge-image-builder/blob/main/docs/building-images.md#kubernetes) documentation.
- Currently the Upgrade Controller has no way to determine the current running Edge release version on the management cluster. Ensure to provide an Edge release version that is greater than the currently running Edge release version on the cluster.
- Currently the Upgrade Controller supports **non air-gapped** environment upgrades only. **Air-gapped** upgrades are not yet possible.

IV Requirements & Assumptions

22 Hardware 151

23 Network 152

24 Port requirements 154

25 Services (DHCP, DNS, etc.) 161

26 Disabling systemd services 162

22 Hardware

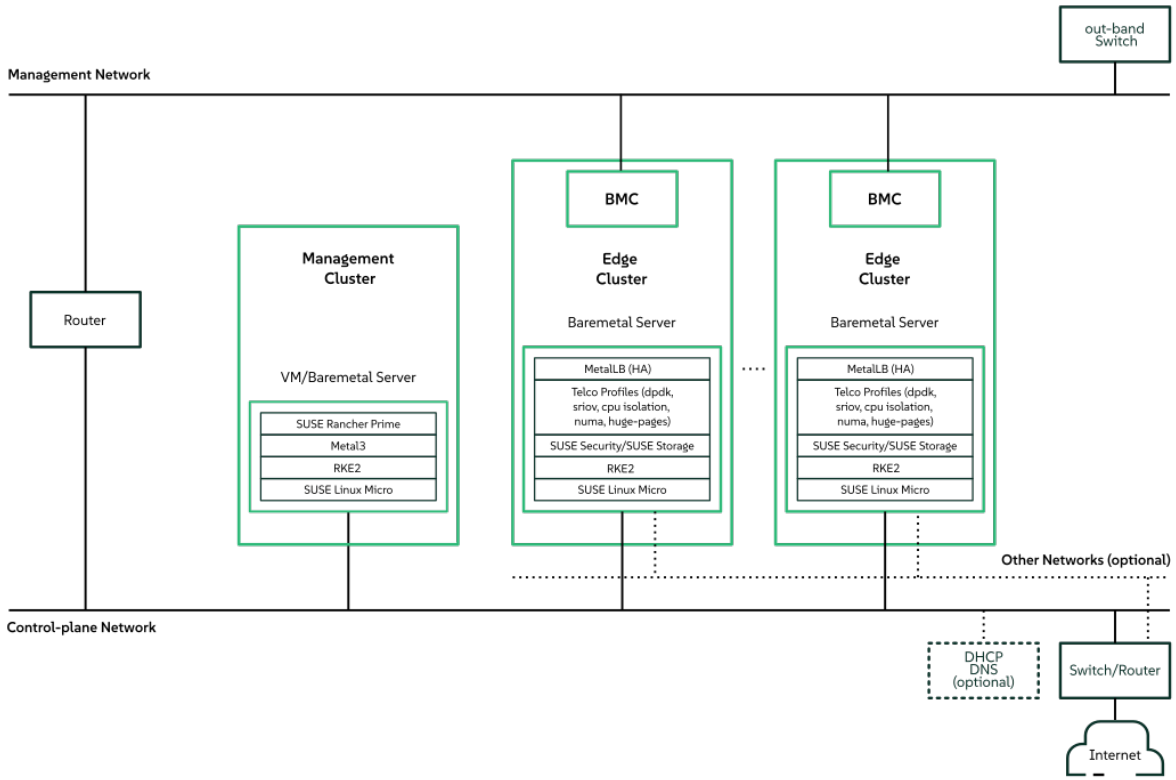
The hardware requirements for SUSE Telco Cloud are as follows:

- **Management cluster**: The management cluster contains components like `SUSE Linux Micro`, `RKE2`, `SUSE Rancher Prime`, `Metal3`, and it is used to manage several downstream clusters. Depending on the number of downstream clusters to be managed, the hardware requirements for the server could vary.
  - Minimum requirements for the server (`VM` or `bare-metal`) are:
    - RAM: 8 GB Minimum (we recommend at least 16 GB)
    - CPU: 2 Minimum (we recommend at least 4 CPU)
- **Downstream clusters**: The downstream clusters are the clusters deployed to run Telco workloads. Specific requirements are needed to enable certain Telco capabilities like `SR-IOV`, `CPU Performance Optimization`, etc.
  - SR-IOV: to attach VFs (Virtual Functions) in pass-through mode to CNFs/VNFs, the NIC must support SR-IOV and VT-d/AMD-Vi be enabled in the BIOS.
  - CPU Processors: To run specific Telco workloads, the CPU Processor model should be adapted to enable most of the features available in this reference table (*Part VI, "Telco features configuration"*).
  - Firmware requirements for installing with virtual media:

Server Hardware	BMC Model	Management
Dell hardware	15th Generation	iDRAC9
Supermicro hardware	01.00.25	Supermicro SMC - redfish
HPE hardware	1.50	iLO6

23 Network

As a reference for the network architecture, the following diagram shows a typical network architecture for a Telco environment:



The network architecture is based on the following components:

- **Management network**: This network is used for the management of downstream cluster nodes. It is used for the out-of-band management. Usually, this network is also connected to a separate management switch, but it can be connected to the same service switch using VLANs to isolate the traffic.
- **Control-plane network**: This network is used for the communication between the downstream cluster nodes and the services that are running on them. This network is also used for the communication between the nodes and the external services, like the `DHCP` or `DNS` servers. In some cases, for connected environments, the switch/router can handle traffic through the Internet.
- **Other networks**: In some cases, nodes could be connected to other networks for specific purposes.

Note

To use the directed network provisioning workflow, the management cluster must have network connectivity to the downstream cluster server Baseboard Management Controller (BMC) so that host preparation and provisioning can be automated.

24 Port requirements

To operate properly, a SUSE Telco Cloud deployment requires a number of ports to be reachable on the management and the downstream Kubernetes cluster nodes.

Note

The exact list depends on the deployed optional components and the selected deployment options (e.g., CNI plug-in).

24.1 Management Nodes

The following table lists the opened ports in nodes running the management cluster:

Note

For CNI plug-in related ports, see CNI specific port requirements (*Section 24.3, "CNI specific port requirements"*).

TABLE 24.1: INBOUND NETWORK RULES FOR MANAGEMENT NODES

Protocol	Port	Source	Description
TCP	22	Any source that requires SSH access	SSH access to management cluster nodes
TCP	80	Load balancer/proxy that does external TLS termination	Rancher UI/API when external TLS termination is used
TCP	443	Any source that requires TLS access to Rancher UI/API	Rancher agent, Rancher UI/API
TCP	2379	RKE2 (management cluster) server nodes	`etcd` client port

Protocol	Port	Source	Description
TCP	2380	RKE2 (management cluster) server nodes	`etcd` peer port
TCP	6180	Any BMC[1] previously instructed by `Metal3/ironic` to pull an IPA[2] ramdisk image from this exposed port (non-TLS)	`Ironic` httpd non-TLS web server serving IPA[2] ISO images for virtual media based boot In case this port is enabled, the functionally equivalent but TLS-enabled one (see below) is not opened
TCP	6185	Any BMC[1] previously instructed by `Metal3/ironic` to pull an IPA[2] ramdisk image from this exposed port (TLS)	`Ironic` httpd TLS-enabled web server serving IPA[2] ISO images for virtual media based boot In case this port is enabled, the functionally equivalent but TLS-disabled one (see above) is not opened
TCP	6385	Any `Metal3/ironic` IPA[1] ramdisk image deployed & running in an "enrolled" `BareMetalHost` instance	Ironic API

Protocol	Port	Source	Description
TCP	6443	Any management cluster node; any external (to the management cluster) Kubernetes client	Kubernetes API
TCP	6545	Any management cluster node	Pull artifacts from OCI-compliant registry (Hauler)
TCP	9345	RKE2 server and agent nodes (management cluster)	RKE2 supervisor API for Node registration (opened port in all RKE2 server nodes)
TCP	10250	Any management cluster node	`kubelet` metrics
TCP/UDP/SCTP	30000-32767	Any external (to the management cluster) source accessing a service exposed on the primary network through a `spec.type: NodePort` or `spec.type: LoadBalancer` Service API object (https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types)	Available `NodePort` port range

(1) BMC: Baseboard Management Controller

(2) IPA: Ironic Python Agent

24.2 Downstream Nodes

In SUSE Telco Cloud, before any (downstream) server becomes part of a running downstream Kubernetes cluster (or runs itself a single-node downstream Kubernetes cluster), it is required to go through some of the BaremetalHost Provisioning states (https://github.com/metal3-io/baremetal-operator/blob/main/docs/baremetalhost-states.md).

- The Baseboard Management Controller (BMC) for a just declared downstream server must be accessible through the out-of-band network. BMC is instructed (from the ironic service running on the management cluster) on the initial steps to take:

  1. Pull and load the indicated IPA ramdisk image in the BMC offered `virtual media`.
  2. Power-on the server.

Following ports are expected to be exposed from the BMC (they could differ depending on the exact hardware):

TABLE 24.2: INBOUND NETWORK RULES FOR BASEBOARD MANAGEMENT CONTROLLERS

Protocol	Port	Source	Description
TCP	80	Ironic conductor (from management cluster)	Redfish API access (HTTP)
TCP	443	Ironic conductor (from management cluster)	Redfish API access (HTTPS)

- Once the IPA ramdisk image loaded on the BMC `virtual media` is used to bootup the downstream server image, the hardware inspection phase begins. The following table lists the ports exposed by a running IPA ramdisk image:

TABLE 24.3: INBOUND NETWORK RULES FOR DOWNSTREAM NODES - `Metal3/Ironic` PROVISIONING PHASE

Protocol	Port	Source	Description
TCP	22	Any source that requires SSH access to IPA ramdisk image	SSH access to a being inspected downstream cluster node

Protocol	Port	Source	Description
TCP	9999	Ironic conductor (from management cluster)	Ironic commands towards the running ramdisk image

- Once the baremetal host is properly provisioned and has joined a downstream Kubernetes cluster, it exposes the following ports:

Note

For CNI plug-in related ports, see CNI specific port requirements (*Section 24.3, "CNI specific port requirements"*).

TABLE 24.4: INBOUND NETWORK RULES FOR DOWNSTREAM NODES

Protocol	Port	Source	Description
TCP	22	Any source that requires SSH access	SSH access to downstream cluster nodes
TCP	80	Load balancer/proxy that does external TLS termination	Rancher UI/API when external TLS termination is used
TCP	443	Any source that requires TLS access to Rancher UI/API	Rancher agent, Rancher UI/API
TCP	2379	RKE2 (downstream cluster) server nodes	`etcd` client port
TCP	2380	RKE2 (downstream cluster) server nodes	`etcd` peer port

Protocol	Port	Source	Description
TCP	6443	Any downstream cluster node; any external (to the downstream cluster) Kubernetes client.	Kubernetes API
TCP	9345	RKE2 server and agent nodes (downstream cluster)	RKE2 supervisor API for Node registration (opened port in all RKE2 server nodes)
TCP	10250	Any downstream cluster node	`kubelet` metrics
TCP	10255	Any downstream cluster node	`kubelet` read-only access
TCP/UDP/SCTP	30000-32767	Any external (to the downstream cluster) source accessing a service exposed on the primary network through a `spec.type: NodePort` or `spec.type: LoadBalancer` Service API object (https://kubernetes.io/docs/concepts/services-networking/service/#publishing-services-service-types)	Available `NodePort` port range

24.3 CNI specific port requirements

Each supported CNI variant comes with its own set of port requirements. For more details, refer CNI Specific Inbound Network Rules (https://docs.rke2.io/install/requirements#cni-specific-inbound-network-rules) in RKE2 documentation.

When `cilium` is set as default/primary CNI plug-in, following TCP port is additionally exposed when the `cilium-operator` workload is configured to expose metrics outside the Kubernetes cluster on which it is deployed. This ensures that an external `Prometheus` server instance running outside that Kubernetes cluster can still collect these metrics.

Note

This is the default option when deploying `cilium` via the rke2-cilium Helm chart.

TABLE 24.5: INBOUND NETWORK RULES FOR MANAGEMENT/DOWNSTREAM NODES - EXTERNAL METRICS EXPOSURE FROM `cilium-operator` ENABLED

Protocol	Port	Source	Description
TCP	9963	External (to the Kubernetes cluster) metrics collector	cilium-operator metrics exposure

25 Services (DHCP, DNS, etc.)

Some external services like `DHCP`, `DNS`, etc. could be required depending on the kind of environment where they are deployed:

- **Connected environment**: In this case, the nodes will be connected to the Internet (via routing L3 protocols) and the external services will be provided by the customer.
- **Disconnected / air-gap environment**: In this case, the nodes will not have Internet IP connectivity and additional services will be required to locally mirror content required by the directed network provisioning workflow.
- **File server**: A file server is used to store the OS images to be provisioned on the downstream cluster nodes during the directed network provisioning workflow. The `Metal3` Helm chart can deploy a media server to store the OS images — check the following section (*Note*), but it is also possible to use an existing local webserver.

26 Disabling systemd services

For Telco workloads, it is important to disable or configure properly some of the services running on the nodes to avoid any impact on the workload performance running on the nodes (latency).

- `rebootmgr` is a service which allows to configure a strategy for reboot when the system has pending updates. For Telco workloads, it is really important to disable or configure properly the `rebootmgr` service to avoid the reboot of the nodes in case of updates scheduled by the system, to avoid any impact on the services running on the nodes.

Note

For more information about `rebootmgr`, see rebootmgr GitHub repository (https://github.com/SUSE/rebootmgr).

Verify the strategy being used by running:

```
cat /etc/rebootmgr.conf
[rebootmgr]
window-start=03:30
window-duration=1h30m
strategy=best-effort
lock-group=default
```

and you could disable it by running:

```
sed -i 's/strategy=best-effort/strategy=off/g' /etc/rebootmgr.conf
```

or using the `rebootmgrctl` command:

```
rebootmgrctl strategy off
```

Note

This configuration to set the `rebootmgr` strategy can be automated using the directed network provisioning workflow. For more information, check the Automated Provisioning documentation (*Part VII, "Fully automated directed network provisioning"*).

- `transactional-update` is a service that allows automatic updates controlled by the system. For Telco workloads, it is important to disable the automatic updates to avoid any impact on the services running on the nodes.

To disable the automatic updates, you can run:

```
systemctl --now disable transactional-update.timer
systemctl --now disable transactional-update-cleanup.timer
```

- `fstrim` is a service that allows to trim the filesystems automatically every week. For Telco workloads, it is important to disable the automatic trim to avoid any impact on the services running on the nodes.

To disable the automatic trim, you can run:

```
systemctl --now disable fstrim.timer
```

V Setting up the management cluster

27 Introduction 165

28 Steps to set up the management cluster 166

29 Image preparation for connected environments 169

30 Image preparation for air-gap environments 192

31 Image creation 201

32 Provision the management cluster 202

33 Dual-stack considerations and configuration 203

27 Introduction

The management cluster is the part of SUSE Telco Cloud that is used to manage the provision and lifecycle of the runtime stacks. From a technical point of view, the management cluster contains the following components:

- `SUSE Linux Micro` as the OS. Depending on the use case, some configurations like networking, storage, users and kernel arguments can be customized.
- `RKE2` as the Kubernetes cluster. Depending on the use case, it can be configured to use specific CNI plugins, such as `Multus`, `Cilium`, `Calico`, etc.
- `Rancher` as the management platform to manage the lifecycle of the clusters.
- `Metal3` as the component to manage the lifecycle of the bare-metal nodes.
- `CAPI` as the component to manage the lifecycle of the Kubernetes clusters (downstream clusters). The `RKE2 CAPI Provider` is used to manage the lifecycle of the RKE2 clusters.

With all components mentioned above, the management cluster can manage the lifecycle of downstream clusters, using a declarative approach to manage the infrastructure and applications.

Note

For more information about `SUSE Linux Micro`, see: SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*)

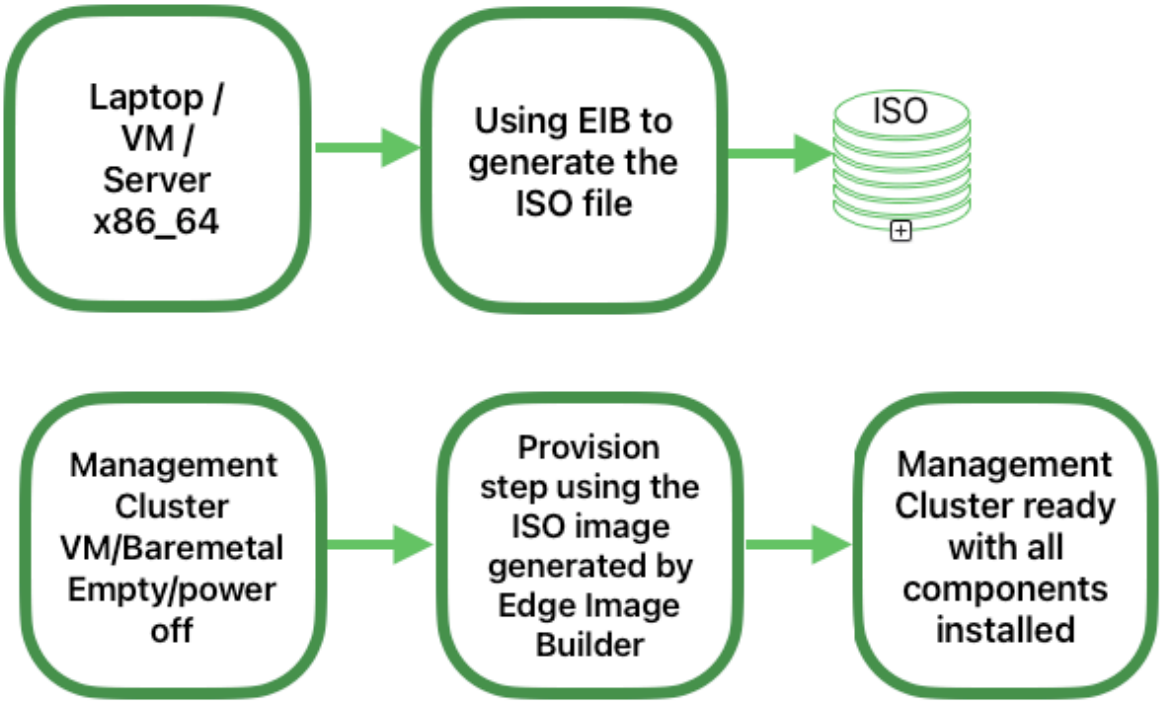
For more information about `RKE2`, see: RKE2 (*Chapter 14, RKE2*)

For more information about `Rancher`, see: Rancher (*Chapter 6, Rancher*)

For more information about `Metal3`, see: Metal3 (*Chapter 11, Metal^3*)

28 Steps to set up the management cluster

The following steps are necessary to set up the management cluster (using a single node):



The following are the main steps to set up the management cluster using a declarative approach:

1. **Image preparation for connected environments (*Chapter 29, Image preparation for connected environments*)**: The first step is to prepare the manifests and files with all the necessary configurations to be used in connected environments.

- Directory structure for connected environments (*Section 29.1, "Directory structure"*): This step creates a directory structure to be used by Edge Image Builder to store the configuration files and the image itself.
- Management cluster definition file (*Section 29.2, "Management cluster definition file"*): The `mgmt-cluster.yaml` file is the main definition file for the management cluster. It contains the following information about the image to be created:
  - Image Information: The information related to the image to be created using the base image.
  - Operating system: The operating system configurations to be used in the image.
  - Kubernetes: Helm charts and repositories, kubernetes version, network configuration, and the nodes to be used in the cluster.
- Custom folder (*Section 29.3, "Custom folder"*): The `custom` folder contains the configuration files and scripts to be used by Edge Image Builder to deploy a fully functional management cluster.
  - Files: Contains the configuration files to be used by the management cluster.
  - Scripts: Contains the scripts to be used by the management cluster.
- Kubernetes folder (*Section 29.4, "Kubernetes folder"*): The `kubernetes` folder contains the configuration files to be used by the management cluster.
  - Manifests: Contains the manifests to be used by the management cluster.
  - Helm: Contains the Helm values files to be used by the management cluster.
  - Config: Contains the configuration files to be used by the management cluster.
- Network folder (*Section 29.5, "Networking folder"*): The `network` folder contains the network configuration files to be used by the management cluster nodes.

2. **Image preparation for air-gap environments (*Chapter 30, Image preparation for air-gap environments*)**: The step is to show the differences to prepare the manifests and files to be used in an air-gap scenario.

- Modifications in the definition file (*Section 30.1, "Modifications in the definition file"*): The `mgmt-cluster.yaml` file must be modified to include the `embeddedArtifactRegistry` section with the `images` field set to all container images to be included into the EIB output image.
- Modifications in the custom folder (*Section 30.2, "Modifications in the custom folder"*): The `custom` folder must be modified to include the resources needed to run the management cluster in an air-gap environment.
  - Register script: The `custom/scripts/99-register.sh` script must be removed when you use an air-gap environment.

3. **Image creation (*Chapter 31, Image creation*)**: This step covers the creation of the image using the Edge Image Builder tool (for both, connected and air-gap scenarios). Check the prerequisites (*Chapter 12, Edge Image Builder*) to run the Edge Image Builder tool on your system.
4. **Management Cluster Provision (*Chapter 32, Provision the management cluster*)**: This step covers the provisioning of the management cluster using the image created in the previous step (for both, connected and air-gap scenarios). This step can be done using a laptop, server, VM or any other AMD64/Intel 64 system with a USB port.

Note

For more information about Edge Image Builder, see Edge Image Builder (*Chapter 12, Edge Image Builder*) and Edge Image Builder Quick Start (*Chapter 5, Standalone clusters with Edge Image Builder*).

29 Image preparation for connected environments

Edge Image Builder is used to create the image for the management cluster, in this document we cover the minimal configuration necessary to set up the management cluster.

Edge Image Builder runs inside a container, so a container runtime is required such as Podman (https://podman.io) or Rancher Desktop (https://rancherdesktop.io). For this guide, we assume podman is available.

Also, as a prerequisite to deploy a highly available management cluster, you need to reserve three IPs in your network:

- `apiVIP` for the API VIP Address (used to access the Kubernetes API server).
- `ingressVIP` for the Ingress VIP Address (consumed, for example, by the Rancher UI).
- `metal3VIP` for the Metal3 VIP Address.

29.1 Directory structure

When running EIB, a directory is mounted from the host, so the first thing to do is to create a directory structure to be used by EIB to store the configuration files and the image itself. This directory has the following structure:

```
eib
├── mgmt-cluster.yaml
├── network
│ └── mgmt-cluster-node1.yaml
├── os-files
│ └── var
│   └── lib
│     └── rancher
│       └── rke2
│         └── server
│           └── manifests
│             └── rke2-ingress-config.yaml
├── kubernetes
│ ├── manifests
│ │ ├── neuvector-namespace.yaml
│ │ ├── ingress-l2-adv.yaml
│ │ └── ingress-ippool.yaml
│ ├── helm
│ │ └── values
│ │     ├── rancher.yaml
```

```
│ │      ├── neuvector.yaml
│ │      ├── longhorn.yaml
│ │      ├── metal3.yaml
│ │      └── certmanager.yaml
│ └── config
│      └── server.yaml
├── custom
│ ├── scripts
│ │ ├── 99-register.sh
│ │ ├── 99-mgmt-setup.sh
│ │ └── 99-alias.sh
│ └── files
│      ├── rancher.sh
│      ├── mgmt-stack-setup.service
│      ├── metal3.sh
│      └── basic-setup.sh
└── base-images
```

Note

The image `SL-Micro.x86_64-6.2-Base-SelfInstall-GM.install.iso` must be downloaded from the SUSE Customer Center (https://scc.suse.com/) or the SUSE Download page (https://www.suse.com/download/sle-micro/), and it must be located under the `base-images` folder.

You should check the SHA256 checksum of the image to ensure it has not been tampered with. The checksum can be found in the same location where the image was downloaded.

An example of the directory structure can be found in the SUSE Telco Cloud GitHub repository under the "telco-examples" folder (https://github.com/suse-edge/telco-cloud-examples).

29.2 Management cluster definition file

The `mgmt-cluster.yaml` file is the main definition file for the management cluster. It contains the following information:

```
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.2-Base-SelfInstall-GM.install.iso
  outputImageName: eib-mgmt-cluster-image.iso
operatingSystem:
```

```
  isoConfiguration:
    installDevice: /dev/sda
  users:
  - username: root
    encryptedPassword: $ROOT_PASSWORD
  packages:
    packageList:
    - jq
    - open-iscsi
    sccRegistrationCode: $SCC_REGISTRATION_CODE
kubernetes:
  version: v1.36.3+rke2r1
  helm:
    charts:
      - name: cert-manager
        repositoryName: jetstack
        version: 1.20.1
        targetNamespace: cert-manager
        valuesFile: certmanager.yaml
        createNamespace: true
        installationNamespace: kube-system
      - name: suse-storage
        releaseName: longhorn
        version: 1.12.1
        repositoryName: rancher-application-collection
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: longhorn.yaml
      - name: metallb
        version: 307.0.3+up0.16.1
        targetNamespace: metallb-system
        createNamespace: true
        repositoryName: suse-edge-charts
        installationNamespace: kube-system
      - name: metal3
        version: 307.0.31+up0.16.0
        repositoryName: suse-edge-charts
        targetNamespace: metal3-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: metal3.yaml
      - name: rancher-turtles-providers
        version: 307.0.8+up0.27.0
        repositoryName: suse-edge-charts
        targetNamespace: cattle-turtles-system
        createNamespace: true
```

```
        installationNamespace: kube-system
      - name: neuvector-crd
        version: 110.0.1+up2.11.1
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector.yaml
      - name: neuvector
        version: 110.0.1+up2.11.1
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector.yaml
      - name: rancher
        version: 2.15.1
        repositoryName: rancher-prime
        targetNamespace: cattle-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: rancher.yaml
    repositories:
      - name: jetstack
        url: https://charts.jetstack.io
      - name: rancher-charts
        url: https://charts.rancher.io/
      - name: suse-edge-charts
        url: oci://registry.suse.com/edge/charts
      - name: rancher-prime
        url: https://charts.rancher.com/server-charts/prime
      - name: rancher-application-collection
        url: oci://dp.apps.rancher.io/charts
        authentication:
          username: $\{APPS.RANCHER.IO_USERNAME\}
          password: $\{APPS.RANCHER.IO_ACCESS_TOKEN\}
  network:
    apiHost: $API_HOST
    apiVIP: $API_VIP
  nodes:
    - hostname: mgmt-cluster-node1
      initializer: true
      type: server
#   - hostname: mgmt-cluster-node2
#     type: server
#   - hostname: mgmt-cluster-node3
#     type: server
```

To explain the fields and values in the `mgmt-cluster.yaml` definition file, we have divided it into the following sections.

- Image section (definition file):

```
image:
  imageType: iso
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.2-Base-SelfInstall-GM.install.iso
  outputImageName: eib-mgmt-cluster-image.iso
```

where the `baseImage` is the original image you downloaded from the SUSE Customer Center or the SUSE Download page. `outputImageName` is the name of the new image that will be used to provision the management cluster.

- Operating system section (definition file):

```
operatingSystem:
  isoConfiguration:
    installDevice: /dev/sda
  users:
  - username: root
    encryptedPassword: $ROOT_PASSWORD
  packages:
    packageList:
    - jq
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

where the `installDevice` is the device to be used to install the operating system, the `username` and `encryptedPassword` are the credentials to be used to access the system, the `packageList` is the list of packages to be installed (`jq` is required internally during the installation process), and the `sccRegistrationCode` is the registration code used to get the packages and dependencies at build time and can be obtained from the SUSE Customer Center. The encrypted password can be generated using the `openssl` command as follows:

```
openssl passwd -6 MyPassword!123
```

This outputs something similar to:

```
$6$UrXB1sAGs46DOiSq$HSwi9GFJLCorm0J53nF2Sq8YEoyINhHcObHzX2R8h13mswUIsMwzx4eUzn/
rRx0QPV4JIb0eWCoNrxGiKH4R31
```

- Kubernetes section (definition file):

```
kubernetes:
```

```
  version: v1.36.3+rke2r1
  helm:
    charts:
      - name: cert-manager
        repositoryName: jetstack
        version: 1.20.1
        targetNamespace: cert-manager
        valuesFile: certmanager.yaml
        createNamespace: true
        installationNamespace: kube-system
      - name: suse-storage
        releaseName: longhorn
        version: 1.12.1
        repositoryName: rancher-application-collection
        targetNamespace: longhorn-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: longhorn.yaml
      - name: metal3
        version: 307.0.31+up0.16.0
        repositoryName: suse-edge-charts
        targetNamespace: metal3-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: metal3.yaml
      - name: metallb
        version: 307.0.3+up0.16.1
        targetNamespace: metallb-system
        createNamespace: true
        repositoryName: suse-edge-charts
        installationNamespace: kube-system
      - name: rancher-turtles-providers
        version: 307.0.8+up0.27.0
        repositoryName: suse-edge-charts
        targetNamespace: cattle-turtles-system
        createNamespace: true
        installationNamespace: kube-system
      - name: neuvector-crd
        version: 110.0.1+up2.11.1
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector.yaml
      - name: neuvector
        version: 110.0.1+up2.11.1
        repositoryName: rancher-charts
```

```
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector.yaml
      - name: rancher
        version: 2.15.1
        repositoryName: rancher-prime
        targetNamespace: cattle-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: rancher.yaml
    repositories:
      - name: jetstack
        url: https://charts.jetstack.io
      - name: rancher-charts
        url: https://charts.rancher.io/
      - name: suse-edge-charts
        url: oci://registry.suse.com/edge/charts
      - name: rancher-prime
        url: https://charts.rancher.com/server-charts/prime
      - name: rancher-application-collection
        url: oci://dp.apps.rancher.io/charts
        authentication:
          username: $\{APPS.RANCHER.IO_USERNAME\}
          password: $\{APPS.RANCHER.IO_ACCESS_TOKEN\}
    network:
      apiHost: $API_HOST
      apiVIP: $API_VIP
    nodes:
    - hostname: mgmt-cluster-node1
      initializer: true
      type: server
#   - hostname: mgmt-cluster-node2
#     type: server
#   - hostname: mgmt-cluster-node3
#     type: server
```

The `helm` section contains the list of Helm charts to be installed, the repositories to be used, and the version configuration for all of them.

The `network` section contains the configuration for the network, like the `apiHost` and `apiVIP` to be used by the `RKE2` component. The `apiVIP` should be an IP address that is not used in the network and should not be part of the DHCP pool (in case we use DHCP). Also, when we use the `apiVIP` in a multi-node cluster, it is used to access the Kubernetes API server. The `apiHost` is the name resolution to `apiVIP` to be used by the `RKE2` component.

The `nodes` section contains the list of nodes to be used in the cluster. In this example, a single-node cluster is being used, but it can be extended to a multi-node cluster by adding more nodes to the list (by uncommenting the lines).

Note

- The names of the nodes must be unique in the cluster.
- Optionally, use the `initializer` field to specify the bootstrap host, otherwise it will be the first node in the list.
- The names of the nodes must be the same as the host names defined in the Network Folder (*Section 29.5, "Networking folder"*) when network configuration is required.

29.3 Custom folder

The `custom` folder contains the following subfolders:

```
...
├── custom
| ├── scripts
| | ├── 99-register.sh
| | ├── 99-mgmt-setup.sh
| | └── 99-alias.sh
| └── files
|     ├── rancher.sh
|     ├── mgmt-stack-setup.service
|     ├── metal3.sh
|     └── basic-setup.sh
...
```

- The `custom/files` folder contains the configuration files to be used by the management cluster.
- The `custom/scripts` folder contains the scripts to be used by the management cluster.

The `custom/files` folder contains the following files:

- `basic-setup.sh`: contains configuration parameters for `Metal3`, `Rancher` and `MetalLB`. Only modify this file if you want to change the namespaces to be used.

  ```
  #!/bin/bash
  ```

```
# Pre-requisites. Cluster already running
export KUBECTL="/var/lib/rancher/rke2/bin/kubectl"
export KUBECONFIG="/etc/rancher/rke2/rke2.yaml"

##################
# METAL3 DETAILS #
##################
export METAL3_CHART_TARGETNAMESPACE="metal3-system"

###########
# METALLB #
###########
export METALLBNAMESPACE="metallb-system"

###########
# RANCHER #
###########
export RANCHER_CHART_TARGETNAMESPACE="cattle-system"
export RANCHER_FINALPASSWORD="adminadminadmin"

die(){
  echo ${1} 1>&2
  exit ${2}
}
```

- `metal3.sh`: contains the configuration for the `Metal3` component to be used (no modifications needed). In future versions, this script will be replaced to use instead `Rancher Turtles` to make it easy.

```
#!/bin/bash
set -euo pipefail

BASEDIR="$(dirname "$0")"
source ${BASEDIR}/basic-setup.sh

METAL3LOCKNAMESPACE="default"
METAL3LOCKCMNAME="metal3-lock"

trap 'catch $? $LINENO' EXIT

catch() {
  if [ "$1" != "0" ]; then
    echo "Error $1 occurred on $2"
    ${KUBECTL} delete configmap ${METAL3LOCKCMNAME} -n ${METAL3LOCKNAMESPACE}
  fi
}
```

```
# Get or create the lock to run all those steps just in a single node
# As the first node is created WAY before the others, this should be enough
# TODO: Investigate if leases is better
if [ $(${KUBECTL} get cm -n ${METAL3LOCKNAMESPACE} ${METAL3LOCKCMNAME} -o name | wc
 -l) -lt 1 ]; then
  ${KUBECTL} create configmap ${METAL3LOCKCMNAME} -n ${METAL3LOCKNAMESPACE} --from-
literal foo=bar
else
  exit 0
fi

# Wait for metal3
while ! ${KUBECTL} wait --for condition=ready -n ${METAL3_CHART_TARGETNAMESPACE}
 $(${KUBECTL} get pods -n ${METAL3_CHART_TARGETNAMESPACE} -l app.kubernetes.io/
name=metal3-ironic -o name) --timeout=10s; do sleep 2 ; done

# Get the ironic IP
IRONICIP=$(${KUBECTL} get cm -n ${METAL3_CHART_TARGETNAMESPACE} ironic -o
 jsonpath='{.data.IRONIC_IP}')

# If LoadBalancer, use metallb, else it is NodePort
if [ $(${KUBECTL} get svc -n ${METAL3_CHART_TARGETNAMESPACE} metal3-metal3-ironic -o
 jsonpath='{.spec.type}') == "LoadBalancer" ]; then
  # Wait for metallb
  while ! ${KUBECTL} wait --for condition=ready -n ${METALLBNAMESPACE} $(${KUBECTL}
 get pods -n ${METALLBNAMESPACE} -l app.kubernetes.io/component=controller -o name)
 --timeout=10s; do sleep 2 ; done

  # Do not create the ippool if already created
  ${KUBECTL} get ipaddresspool -n ${METALLBNAMESPACE} ironic-ip-pool -o name || cat
 <<-EOF | ${KUBECTL} apply -f -
  apiVersion: metallb.io/v1beta1
  kind: IPAddressPool
  metadata:
    name: ironic-ip-pool
    namespace: ${METALLBNAMESPACE}
  spec:
    addresses:
    - ${IRONICIP}/32
    serviceAllocation:
      priority: 100
      serviceSelectors:
      - matchExpressions:
        - {key: app.kubernetes.io/name, operator: In, values: [metal3-ironic]}
 EOF
```

```
  # Same for L2 Advs
  ${KUBECTL} get L2Advertisement -n ${METALLBNAMESPACE} ironic-ip-pool-l2-adv -o
 name || cat <<-EOF | ${KUBECTL} apply -f -
  apiVersion: metallb.io/v1beta1
  kind: L2Advertisement
  metadata:
    name: ironic-ip-pool-l2-adv
    namespace: ${METALLBNAMESPACE}
  spec:
    ipAddressPools:
    - ironic-ip-pool
 EOF
fi

# Clean up the lock cm

${KUBECTL} delete configmap ${METAL3LOCKCMNAME} -n ${METAL3LOCKNAMESPACE}
```

- `rancher.sh`: contains the configuration for the `Rancher` component to be used (no modifications needed).

```
#!/bin/bash
set -euo pipefail

BASEDIR="$(dirname "$0")"
source ${BASEDIR}/basic-setup.sh

RANCHERLOCKNAMESPACE="default"
RANCHERLOCKCMNAME="rancher-lock"

if [ -z "${RANCHER_FINALPASSWORD}" ]; then
  # If there is no final password, then finish the setup right away
  exit 0
fi

trap 'catch $? $LINENO' EXIT

catch() {
  if [ "$1" != "0" ]; then
    echo "Error $1 occurred on $2"
    ${KUBECTL} delete configmap ${RANCHERLOCKCMNAME} -n ${RANCHERLOCKNAMESPACE}
  fi
}

# Get or create the lock to run all those steps just in a single node
# As the first node is created WAY before the others, this should be enough
```

```
# TODO: Investigate if leases is better
if [ $(${KUBECTL} get cm -n ${RANCHERLOCKNAMESPACE} ${RANCHERLOCKCMNAME} -o
 name | wc -l) -lt 1 ]; then
  ${KUBECTL} create configmap ${RANCHERLOCKCMNAME} -n ${RANCHERLOCKNAMESPACE}
 --from-literal foo=bar
else
  exit 0
fi

# Wait for rancher to be deployed
while ! ${KUBECTL} wait --for condition=ready -n
 ${RANCHER_CHART_TARGETNAMESPACE} $(${KUBECTL} get pods -n
 ${RANCHER_CHART_TARGETNAMESPACE} -l app=rancher -o name) --timeout=10s; do
 sleep 2 ; done
until ${KUBECTL} get ingress -n ${RANCHER_CHART_TARGETNAMESPACE} rancher > /
dev/null 2>&1; do sleep 10; done

RANCHERBOOTSTRAPPASSWORD=$(${KUBECTL} get secret -n
 ${RANCHER_CHART_TARGETNAMESPACE} bootstrap-secret -o
 jsonpath='{.data.bootstrapPassword}' | base64 -d)
RANCHERHOSTNAME=$(${KUBECTL} get ingress -n ${RANCHER_CHART_TARGETNAMESPACE}
 rancher -o jsonpath='{.spec.rules[0].host}')

# Skip the whole process if things have been set already
if [ -z $(${KUBECTL} get settings.management.cattle.io first-login -
ojsonpath='{.value}') ]; then
  # Add the protocol
  RANCHERHOSTNAME="https://${RANCHERHOSTNAME}"
  TOKEN=""
  while [ -z "${TOKEN}" ]; do
    # Get token
    sleep 2
    TOKEN=$(curl -sk -X POST ${RANCHERHOSTNAME}/v3-public/localProviders/local?
action=login -H 'content-type: application/json' -d "{\"username\":\"admin\",
\"password\":\"${RANCHERBOOTSTRAPPASSWORD}\"}" | jq -r .token)
  done

  # Set password
  curl -sk ${RANCHERHOSTNAME}/v3/users?action=changepassword -H 'content-type:
 application/json' -H "Authorization: Bearer $TOKEN" -d "{\"currentPassword\":
\"${RANCHERBOOTSTRAPPASSWORD}\",\"newPassword\":\"${RANCHER_FINALPASSWORD}\"}"

  # Create a temporary API token (ttl=60 minutes)
  APITOKEN=$(curl -sk ${RANCHERHOSTNAME}/v3/token -H 'content-
type: application/json' -H "Authorization: Bearer ${TOKEN}" -d
 '{"type":"token","description":"automation","ttl":3600000}' | jq -r .token)
```

```
  curl -sk ${RANCHERHOSTNAME}/v3/settings/server-url -H 'content-type:
 application/json' -H "Authorization: Bearer ${APITOKEN}" -X PUT -d "{\"name\":
\"server-url\",\"value\":\"${RANCHERHOSTNAME}\"}"
  curl -sk ${RANCHERHOSTNAME}/v3/settings/telemetry-opt -X PUT -H 'content-
type: application/json' -H 'accept: application/json' -H "Authorization: Bearer
 ${APITOKEN}" -d '{"value":"out"}'
fi

# Clean up the lock cm
${KUBECTL} delete configmap ${RANCHERLOCKCMNAME} -n ${RANCHERLOCKNAMESPACE}
```

- `mgmt-stack-setup.service`: contains the configuration to create the systemd service to run the scripts during the first boot (no modifications needed).

```
[Unit]
Description=Setup Management stack components
Wants=network-online.target
# It requires rke2 or k3s running, but it will not fail if those services are
 not present
After=network.target network-online.target rke2-server.service k3s.service
# At least, the basic-setup.sh one needs to be present
ConditionPathExists=/opt/mgmt/bin/basic-setup.sh

[Service]
User=root
Type=forking
# Metal3 can take A LOT to download the IPA image
TimeoutStartSec=1800

ExecStartPre=/bin/sh -c "echo 'Setting up Management components...'"
# Scripts are executed in StartPre because Start can only run a single one
ExecStartPre=/opt/mgmt/bin/rancher.sh
ExecStartPre=/opt/mgmt/bin/metal3.sh
ExecStart=/bin/sh -c "echo 'Finished setting up Management components'"
RemainAfterExit=yes
KillMode=process
# Disable & delete everything
ExecStartPost=rm -f /opt/mgmt/bin/rancher.sh
ExecStartPost=rm -f /opt/mgmt/bin/metal3.sh
ExecStartPost=rm -f /opt/mgmt/bin/basic-setup.sh
ExecStartPost=/bin/sh -c "systemctl disable mgmt-stack-setup.service"
ExecStartPost=rm -f /etc/systemd/system/mgmt-stack-setup.service

[Install]
WantedBy=multi-user.target
```

The `custom/scripts` folder contains the following files:

- `99-alias.sh` script: contains the alias to be used by the management cluster to load the kubeconfig file at first boot (no modifications needed).

```
#!/bin/bash
echo "alias k=kubectl" >> /etc/profile.local
echo "alias kubectl=/var/lib/rancher/rke2/bin/kubectl" >> /etc/profile.local
echo "export KUBECONFIG=/etc/rancher/rke2/rke2.yaml" >> /etc/profile.local
```

- `99-mgmt-setup.sh` script: contains the configuration to copy the scripts during the first boot (no modifications needed).

```
#!/bin/bash

# Copy the scripts from combustion to the final location
mkdir -p /opt/mgmt/bin/
for script in basic-setup.sh rancher.sh metal3.sh; do
 cp ${script} /opt/mgmt/bin/
done

# Copy the systemd unit file and enable it at boot
cp mgmt-stack-setup.service /etc/systemd/system/mgmt-stack-setup.service
systemctl enable mgmt-stack-setup.service
```

- `99-register.sh` script: contains the configuration to register the system using the SCC registration code. The `${SCC_ACCOUNT_EMAIL}` and `${SCC_REGISTRATION_CODE}` have to be set properly to register the system with your account.

```
#!/bin/bash
set -euo pipefail

# Registration https://www.suse.com/support/kb/doc/?id=000018564
if ! which SUSEConnect > /dev/null 2>&1; then
 zypper --non-interactive install suseconnect-ng
fi
SUSEConnect --email "${SCC_ACCOUNT_EMAIL}" --url "https://scc.suse.com" --regcode
 "${SCC_REGISTRATION_CODE}"
```

29.4 Kubernetes folder

The `kubernetes` folder contains the following subfolders:

```
...
├── kubernetes
| ├── manifests
| | ├── rke2-ingress-config.yaml
| | ├── neuvector-namespace.yaml
| | ├── ingress-l2-adv.yaml
| | └── ingress-ippool.yaml
| ├── helm
| | └── values
| |     ├── rancher.yaml
| |     ├── neuvector.yaml
| |     ├── metal3.yaml
| |     └── certmanager.yaml
| └── config
|     └── server.yaml
...
```

The `kubernetes/config` folder contains the following files:

- `server.yaml`: The `CNI` plug-in installed by default is `Cilium`, so you do not need to create this folder and file to set that. Just in case you need to customize the `CNI` plug-in, you can use the `server.yaml` file under the `kubernetes/config` folder. It contains the following information:

  ```
  cni:
  - multus
  - cilium
  write-kubeconfig-mode: '0644'
  selinux: true
  system-default-registry: registry.rancher.com
  ```

Note

This is an optional file to define certain Kubernetes customization, like the CNI plug-ins to be used or many options you can check in the official documentation (https://docs.rke2.io/install/configuration).

The `os-files/var/lib/rancher/rke2/server/manifests` folder contains the following file:

- `rke2-ingress-config.yaml`: contains the configuration to create the (Traefik based) `Ingress` service for the management cluster (no modifications needed except for the `.providers` stanza: to be removed in case of not deploying/not planning to deploy `Ingress` API objects containing `ingress-nginx` annotations in its `.metadata` stanza).

```
apiVersion: helm.cattle.io/v1
kind: HelmChartConfig
metadata:
  name: rke2-traefik
  namespace: kube-system
spec:
  valuesContent: |-
    ingressClass:
      isDefaultClass: true
    ports:
      web:
        hostPort: null    # disallow hostPort
        exposedPort: 80
      websecure:
        hostPort: null    # disallow hostPort
        exposedPort: 443
    service:
      enabled: true
      spec:
        type: LoadBalancer
        externalTrafficPolicy: Local
        allocateLoadBalancerNodePorts: false
    providers:
      kubernetesIngressNGINX:  # this provider allows traefik to "understand" most
 of the ingress-nginx annotations
        enabled: true
        ingressClass: "rke2-ingress-nginx-migration"
        controllerClass: "rke2.cattle.io/ingress-nginx-migration"
```

Note

The `HelmChartConfig` must be included via `os-files` to the `/var/lib/rancher/rke2/server/manifests` directory, not via `kubernetes/manifests` as described in previous releases.

The `kubernetes/manifests` folder contains the following files:

- `neuvector-namespace.yaml`: contains the configuration to create the `NeuVector` namespace (no modifications needed).

```
apiVersion: v1
kind: Namespace
metadata:
  labels:
    pod-security.kubernetes.io/enforce: privileged
  name: neuvector
```

- `ingress-l2-adv.yaml`: contains the configuration to create the `L2Advertisement` for the `MetalLB` component (no modifications needed).

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ingress-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
    - ingress-ippool
```

- `ingress-ippool.yaml`: contains the configuration to create the MetalLB `IPAddressPool` object for the `rke2-traefik` component. The `${INGRESS_VIP}` must be set to the external (Virtual) IP address reserved for the `rke2-traefik` LoadBalancer service.

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: ingress-ippool
  namespace: metallb-system
spec:
  addresses:
    - ${INGRESS_VIP}/32
  serviceAllocation:
    priority: 100
    serviceSelectors:
      - matchExpressions:
          - {key: app.kubernetes.io/name, operator: In, values: [rke2-traefik]}
```

The `kubernetes/helm/values` folder contains the following files:

- `rancher.yaml`: contains the configuration to create the `Rancher` component. The `${INGRESS_VIP}` must be set properly to define the IP address to be consumed by the `Rancher` component. The URL to access the `Rancher` component will be `https://rancher-${INGRESS_VIP}.sslip.io`.

```
hostname: rancher-${INGRESS_VIP}.sslip.io
bootstrapPassword: "foobar"
replicas: 1
global:
  cattle:
    systemDefaultRegistry: "registry.rancher.com"
```

- `neuvector.yaml`: contains the configuration to create the `NeuVector` component (no modifications needed).

```
controller:
  replicas: 1
  ranchersso:
    enabled: true
manager:
  enabled: false
cve:
  scanner:
    enabled: false
    replicas: 1
k3s:
  enabled: true
crdwebhook:
  enabled: false
registry: "registry.rancher.com"
global:
  cattle:
    systemDefaultRegistry: "registry.rancher.com"
```

- `longhorn.yaml`: contains the configuration to create the `Longhorn` component. To make sure the necessary container images are downloaded at boot time, please add your Rancher Application Collection credentials.

```
privateRegistry:
  createSecret: true
  registryUrl: dp.apps.rancher.io
  registryUser: $\{APPS.RANCHER.IO_USERNAME\}
  registryPasswd: $\{APPS.RANCHER.IO_ACCESS_TOKEN\}
```

```
  registrySecret: rancher-app-collection
```

- `metal3.yaml`: contains the configuration to create the `Metal3` component. The `${METAL3_VIP}` must be set properly to define the IP address to be consumed by the `Metal3` component.

```
global:
  ironicIP: ${METAL3_VIP}
  enable_vmedia_tls: false
  predictableNicNames: "true"
  # trustedCAs: tls-ca-bundle  # Optional: Uncomment and set to ConfigMap name for
 additional trusted CAs
metal3-ironic:
  ipa:
    useHauler: false
  persistence:
    ironic:
      size: "5Gi"
```

Important

The `metal3-ironic.ipa.useHauler` Helm chart parameter **must** be set to `true` (boolean value) in air-gapped environments; this instructs Metal^3 to retrieve the `ironic-python-agent` (IPA) container image from the embedded artifact registry instead of attempting to download it from the internet. This IPA container image must be additionally included in the `embeddedArtifactRegistry.images` list in the EIB image definition file for air-gap environments (*Section 30.1, "Modifications in the definition file"*).

Note

The Media Server is an optional feature included in Metal^3 (by default is disabled). To use the Metal3 feature, you need to configure it on the previous manifest. To use the Metal^3 media server, specify the following variable:

- add the `enable_metal3_media_server` to `true` to enable the media server feature in the global section.
- include the following configuration about the media server where ${MEDIA_VOLUME_PATH} is the path to the media volume in the media (e.g `/home/metal3/bmh-image-cache`)

```
metal3-media:
```

```
  mediaVolume:
    hostPath: ${MEDIA_VOLUME_PATH}
```

An external media server can be used to store the images, and in the case you want to use it with TLS, you will need to provide the trusted CA bundle for Metal[3].

Prepare the CA bundle:

First, prepare your CA bundle file containing all the certificates needed by Metal[3]. This includes your external media server's CA certificate(s).

+ **Optional - System CA Bundle:** If Metal[3] also needs to trust public CAs (for example, when accessing external HTTPS resources), you can include the system CA bundle. Extract it from a container image and concatenate it with your custom certificates:

+

```
# Extract system CAs from a container image
podman run --rm registry.suse.com/bci/bci-base:latest cat /etc/ssl/certs/ca-
certificates.crt > system-cas.pem

# Concatenate system CAs with your custom CA
cat system-cas.pem your-custom-ca.crt > ca-bundle.pem
```

+ Or, if you only need your custom CA:

+

```
cp your-custom-ca.crt ca-bundle.pem
```

+

If you include the system CA bundle, you become responsible for keeping it up-to-date. System CAs in container images may become outdated as certificates expire or are revoked. Periodically refresh the bundle by re-extracting from an updated container image.

Create the ConfigMap:

You can create the ConfigMap in two ways:

1. **Using a manifest file (recommended for EIB):** Create the file `kubernetes/manifests/metal3-cacert-configmap.yaml` and include the CA bundle content inline. You need to properly indent each line of the certificate with 4 spaces:

```
apiVersion: v1
```

```
kind: Namespace
metadata:
  name: metal3-system
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: tls-ca-bundle
  namespace: metal3-system
data:
  ca-bundle.pem: |
    -----BEGIN CERTIFICATE-----
    MIIDXTCCAkWgAwIBAgIJAKJ... (your certificate content here)
    ...
    -----END CERTIFICATE-----
    -----BEGIN CERTIFICATE-----
    MIIEkjCCA3qgAwIBAgIQCgF... (additional certificates if any)
    ...
    -----END CERTIFICATE-----
```

To fill this file automatically from your `ca-bundle.pem`, you can use:

```
cat > kubernetes/manifests/metal3-cacert-configmap.yaml <<EOF
apiVersion: v1
kind: Namespace
metadata:
  name: metal3-system
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: tls-ca-bundle
  namespace: metal3-system
data:
  ca-bundle.pem: |
$(sed 's/^/    /' ca-bundle.pem)
EOF
```

2. **Using kubectl directly (for manual deployment):** If you're not using EIB and want to create the ConfigMap directly on the cluster:

```
kubectl -n metal3-system create configmap tls-ca-bundle --from-file=ca-bundle.pem=./
ca-bundle.pem
```

- Set the `global.trustedCAs` value in the `metal3.yaml` file to reference the ConfigMap name:

```
global:
  trustedCAs: tls-ca-bundle
```

- `certmanager.yaml`: contains the configuration to create the `Cert-Manager` component (no modifications needed).

```
installCRDs: true
```

29.5 Networking folder

The `network` folder contains as many files as nodes in the management cluster. In our case, we have only one node, so we have only one file called `mgmt-cluster-node1.yaml`. The name of the file must match the host name defined in the `mgmt-cluster.yaml` definition file into the network/node section described above.

If you need to customize the networking configuration, for example, to use a specific static IP address (DHCP-less scenario), you can use the `mgmt-cluster-node1.yaml` file under the `network` folder. It contains the following information:

- `${MGMT_GATEWAY}`: The gateway IP address.
- `${MGMT_DNS}`: The DNS server IP address.
- `${MGMT_MAC}`: The MAC address of the network interface.
- `${MGMT_NODE_IP}`: The IP address of the management cluster.

```
routes:
  config:
  - destination: 0.0.0.0/0
    metric: 100
    next-hop-address: ${MGMT_GATEWAY}
    next-hop-interface: eth0
    table-id: 254
dns-resolver:
  config:
    server:
    - ${MGMT_DNS}
    - 8.8.8.8
```

```
interfaces:
- name: eth0
  type: ethernet
  state: up
  mac-address: ${MGMT_MAC}
  ipv4:
    address:
    - ip: ${MGMT_NODE_IP}
      prefix-length: 24
    dhcp: false
    enabled: true
  ipv6:
    enabled: false
```

If you want to use DHCP to get the IP address, you can use the following configuration (the `MAC` address must be set properly using the `${MGMT_MAC}` variable):

```
## This is an example of a dhcp network configuration for a management cluster
interfaces:
- name: eth0
  type: ethernet
  state: up
  mac-address: ${MGMT_MAC}
  ipv4:
    dhcp: true
    enabled: true
  ipv6:
    enabled: false
```

Note

- Depending on the number of nodes in the management cluster, you can create more files like `mgmt-cluster-node2.yaml`, `mgmt-cluster-node3.yaml`, etc. to configure the rest of the nodes.
- The `routes` section is used to define the routing table for the management cluster.

30 Image preparation for air-gap environments

This section describes how to prepare the image for air-gap environments showing only the differences from the previous sections. The following changes to the previous section (Image preparation for connected environments (*Chapter 29, Image preparation for connected environments*)) are required to prepare the image for air-gap environments:

- The `mgmt-cluster.yaml` file must be modified to include the `embeddedArtifactRegistry` section with the `images` field set to all container images to be included into the EIB output image.
- The `custom/scripts/99-register.sh` script must be removed when use an air-gap environment.

To include the SUSE Private Registry in the management cluster for future downstream deployments, the following changes are also required:

- The `mgmt-cluster.yaml` file must be modified to include the helm chart section as well as the embedded artifacts with the new images to include them in the EIB output image.
- The `kubernetes/values` and `kubernetes/manifests` folders must include additional manifest files to properly configure the SUSE Private Registry in the management cluster.

Note

You will need certain credentials, which can be retrieved by following the official SUSE Private Registry documentation (https://documentation.suse.com/cloudnative/suse-private-registry/html/private-registry/pr-deployment.html#pr-deployment-kube-secrets).

30.1 Modifications in the definition file

The `mgmt-cluster.yaml` file must be modified to include the `embeddedArtifactRegistry` section. In this section the `images` field must contain the list of all container images to be included in the output image.

Note

The following example `mgmt-cluster.yaml` file includes both the `embeddedArtifactRegistry` section and the SUSE Private Registry feature. Make sure to the listed images contain the component versions you need.

```
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.2-Base-SelfInstall-GM.install.iso
  outputImageName: eib-mgmt-cluster-image.iso
operatingSystem:
  isoConfiguration:
    installDevice: /dev/sda
  users:
  - username: root
    encryptedPassword: $ROOT_PASSWORD
  packages:
    packageList:
    - jq
    sccRegistrationCode: $SCC_REGISTRATION_CODE
kubernetes:
  version: v1.36.3+rke2r1
  helm:
    charts:
      - name: private-registry-helm
        createNamespace: true
        installationNamespace: kube-system
        repositoryName: privateregistry
        targetNamespace: suse-private-registry
        valuesFile: privateregistry.yaml
        version: 1.1.1
      - name: cert-manager
        repositoryName: jetstack
        version: 1.20.1
        targetNamespace: cert-manager
        valuesFile: certmanager.yaml
        createNamespace: true
        installationNamespace: kube-system
      - name: suse-storage
        releaseName: longhorn
        version: 1.12.1
        repositoryName: rancher-application-collection
        targetNamespace: longhorn-system
```

```
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: longhorn.yaml
      - name: metallb
        version: 307.0.3+up0.16.1
        targetNamespace: metallb-system
        createNamespace: true
        repositoryName: suse-edge-charts
        installationNamespace: kube-system
      - name: metal3
        version: 307.0.31+up0.16.0
        repositoryName: suse-edge-charts
        targetNamespace: metal3-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: metal3.yaml
      - name: rancher-turtles-providers
        version: 307.0.8+up0.27.0
        repositoryName: suse-edge-charts
        targetNamespace: cattle-turtles-system
        createNamespace: true
        installationNamespace: kube-system
      - name: neuvector-crd
        version: 110.0.1+up2.11.1
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector.yaml
      - name: neuvector
        version: 110.0.1+up2.11.1
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector.yaml
      - name: rancher
        version: 2.15.1
        repositoryName: rancher-prime
        targetNamespace: cattle-system
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: rancher.yaml
    repositories:
      - name: jetstack
        url: https://charts.jetstack.io
      - name: rancher-charts
```

```
        url: https://charts.rancher.io/
      - name: suse-edge-charts
        url: oci://registry.suse.com/edge/charts
      - name: rancher-prime
        url: https://charts.rancher.com/server-charts/prime
      - name: rancher-application-collection
        url: oci://dp.apps.rancher.io/charts
        authentication:
          username: $\{APPS.RANCHER.IO_USERNAME\}
          password: $\{APPS.RANCHER.IO_ACCESS_TOKEN\}
      - name: privateregistry
        authentication:
          username: ${PRIVATE_REGISTRY_USERNAME}
          password: ${PRIVATE_REGISTRY_PASSWORD}
        plainHTTP: false
        skipTLSVerify: false
        url: oci://registry.suse.com/private-registry
    network:
      apiHost: $API_HOST
      apiVIP: $API_VIP
    nodes:
    - hostname: mgmt-cluster-node1
      initializer: true
      type: server
#   - hostname: mgmt-cluster-node2
#     type: server
#   - hostname: mgmt-cluster-node3
#     type: server
embeddedArtifactRegistry:
  registries:
    - uri: dp.apps.rancher.io
      authentication:
        username: $\{APPS.RANCHER.IO_USERNAME\}
        password: $\{APPS.RANCHER.IO_ACCESS_TOKEN\}
    - uri: registry.suse.com
      authentication:
        username: ${PRIVATE_REGISTRY_USERNAME}
        password: ${PRIVATE_REGISTRY_PASSWORD}
  images:
    - name: registry.suse.com/private-registry/harbor-core:1.1.1-1.19
    - name: registry.suse.com/private-registry/harbor-jobservice:1.1.1-1.19
    - name: registry.suse.com/private-registry/harbor-portal:1.1.1-1.20
    - name: registry.suse.com/private-registry/harbor-registry:1.1.1-1.19
    - name: registry.suse.com/private-registry/harbor-registryctl:1.1.1-1.19
    - name: registry.suse.com/private-registry/harbor-trivy-adapter:1.1.1-1.24
    - name: registry.rancher.com/rancher/hardened-cluster-autoscaler:v1.10.3-
build20260717
```

```
    - name: registry.rancher.com/rancher/hardened-cni-plugins:v1.9.1-build20260717
    - name: registry.rancher.com/rancher/hardened-coredns:v1.14.6-build20260722
    - name: registry.rancher.com/rancher/hardened-k8s-metrics-server:v0.9.0-build20260722
    - name: registry.rancher.com/rancher/hardened-multus-cni:v4.3.0-build20260722
    - name: registry.rancher.com/rancher/hardened-traefik:v3.7.8-build20260717
    - name: registry.rancher.com/rancher/klipper-helm:v0.13.3-build20260727
    - name: registry.rancher.com/rancher/mirrored-cilium-cilium:v1.19.6
    - name: registry.rancher.com/rancher/mirrored-cilium-operator-generic:v1.19.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-attacher:4.12.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-provisioner:5.3.0-17.8
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-resizer:2.2.1-10.8
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-snapshotter:8.6.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-livenessprobe:2.19.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-node-driver-
registrar:2.17.0-17.6
    - name: dp.apps.rancher.io/containers/longhorn-backing-image-manager:1.12.1-1.5
    - name: dp.apps.rancher.io/containers/longhorn-engine:1.12.1-1.6
    - name: dp.apps.rancher.io/containers/longhorn-instance-manager:1.12.1-1.7
    - name: dp.apps.rancher.io/containers/longhorn-manager:1.12.1-1.5
    - name: dp.apps.rancher.io/containers/longhorn-share-manager:1.12.1-1.5
    - name: dp.apps.rancher.io/containers/longhorn-ui:1.12.1-1.3
    - name: dp.apps.rancher.io/containers/rancher-support-bundle-kit:0.0.92-13.12
    - name: registry.rancher.com/rancher/mirrored-sig-storage-snapshot-controller:v8.2.0
    - name: registry.rancher.com/rancher/neuvector-compliance-config:1.0.16
    - name: registry.rancher.com/rancher/neuvector-controller:5.6.1
    - name: registry.rancher.com/rancher/neuvector-enforcer:5.6.1
    - name: registry.rancher.com/rancher/neuvector-updater:0.0.14
    - name: registry.rancher.com/rancher/nginx-ingress-controller:v1.14.5-hardened2
    - name: registry.suse.com/rancher/cluster-api-addon-provider-fleet:v0.15.0
    - name: registry.rancher.com/rancher/fleet-agent:v0.16.1
    - name: registry.rancher.com/rancher/fleet:v0.16.1
    - name: registry.rancher.com/rancher/rancher-webhook:v0.11.1
    - name: registry.rancher.com/rancher/turtles:v0.27.1
    - name: registry.rancher.com/rancher/rancher:v2.15.1
    - name: registry.rancher.com/rancher/shell:v0.8.1
    - name: registry.rancher.com/rancher/system-upgrade-controller:v0.20.1
    - name: registry.rancher.com/rancher/cluster-api-controller:v1.13.3
    - name: registry.suse.com/rancher/cluster-api-provider-metal3:v1.13.2
    - name: registry.suse.com/rancher/cluster-api-provider-rke2-bootstrap:v0.25.0
    - name: registry.suse.com/rancher/cluster-api-provider-rke2-controlplane:v0.25.0
    - name: registry.rancher.com/rancher/ip-address-manager:v1.13.2
    - name: registry.rancher.com/rancher/kubectl:v1.35.2
    - name: registry.rancher.com/rancher/scc-operator:v0.5.1
    - name: registry.suse.com/edge/3.7/ironic-python-agent:3.0.10
```

30.2 Modifications in the custom folder

- The `custom/scripts/99-register.sh` script must be removed when using an air-gap environment. As you can see in the directory structure, the `99-register.sh` script is not included in the `custom/scripts` folder.

30.3 Modifications in the kubernetes folder

- Update the `kubernetes/helm/values/metal3.yaml` file to set the `metal3-ironic.ipa.useHauler` Helm chart parameter to `true` (boolean value).

```
global:
  ironicIP: ${METAL3_VIP}
  enable_vmedia_tls: false
  predictableNicNames: "true"
  # trustedCAs: tls-ca-bundle  # Optional: Uncomment and set to ConfigMap name for
 additional trusted CAs
metal3-ironic:
  ipa:
    useHauler: true
  persistence:
    ironic:
      size: "5Gi"
```

- Update the `kubernetes/helm/values/rancher.yaml` file to modify the `hostname` parameter. In air-gapped environments, external DNS resolution services like `sslip.io` are not accessible. You have two options:

  a. **Option 1: Use internal DNS** (Recommended): Configure your internal DNS server to resolve the Rancher hostname and update the file accordingly:

```
hostname: rancher.yourdomain.internal
bootstrapPassword: "foobar"
replicas: 1
global:
  cattle:
```

```
    systemDefaultRegistry: "registry.rancher.com"
```

b. **Option 2: Use /etc/hosts injection**: If you do not have an internal DNS server, you can inject a static `/etc/hosts` entry via EIB. Create a file at `eib/os-files/etc/hosts` in your EIB directory structure with content like:

```
# Static DNS resolution for Rancher in air-gapped environment
${INGRESS_VIP} rancher-${INGRESS_VIP}.sslip.io
```

Then update the `rancher.yaml` file to use this hostname:

```
hostname: rancher-${INGRESS_VIP}.sslip.io
bootstrapPassword: "foobar"
replicas: 1
global:
  cattle:
    systemDefaultRegistry: "registry.rancher.com"
```

> **Important**
>
> When using Option 2, remember that the `/etc/hosts` file will need to be present on any system that needs to access the Rancher UI, not just the management cluster nodes.

- You need to modify the `${MGMT_CLUSTER_REGISTRY_IP}` with a reserved static IP for the SUSE Private Registry in the following files:

  1. `kubernetes/manifests/metallb-registry.yaml`

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: private-registry
  namespace: metallb-system
spec:
  ipAddressPools:
  - private-registry-pool
---
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: private-registry-pool
  namespace: metallb-system
spec:
  addresses:
```

```
  - ${MGMT_CLUSTER_REGISTRY_IP}/32
  serviceAllocation:
    namespaces:
    - suse-private-registry
```

2. `kubernetes/helm/values/privateregistry.yaml`

```
core:
  secretName: suse-registry-tls
expose:
  tls:
    certSource: secret
    enabled: true
    secret:
      secretName: suse-registry-tls
  type: loadBalancer
externalURL: https://${MGMT_CLUSTER_REGISTRY_IP}
persistence:
  persistentVolumeClaim:
    registry:
      size: 20Gi
```

- The `kubernetes/manifests/suse-private-registry-creds.yaml` must be created with the following content:

```
apiVersion: v1
kind: Secret
metadata:
  name: suse-registry
  namespace: suse-private-registry
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: ${DOCKER_CONFIG_JSON_BASE64}
---
apiVersion: v1
kind: Secret
metadata:
    name: suse-registry-tls
    namespace: suse-private-registry
type: kubernetes.io/tls
data:
    tls.crt: ${TLS_CRT_BASE64}
    tls.key: ${TLS_KEY_BASE64}
```

You need to modify the `${DOCKER_CONFIG_JSON_BASE64}`, `${TLS_CRT_BASE64}` and `${TLS_KEY_BASE64}`. To correctly configure the docker config json (base64), you can do the following:

```
# ${DOCKER_CONFIG_JSON_BASE64} CONTENT
echo -n '{"auths":{"<MGMT_CLUSTER_REGISTRY_IP>":
{"username":"<USERNAME>","password":"<PASSWORD>","auth":"<AUTH>"}}}' | base64 -w 0
```

where the IP is the same as the previously configured `${MGMT_CLUSTER_REGISTRY_IP}`, and the `<USERNAME>`, and `<PASSWORD>` values can be retrieved from the SUSE Private Registry official documentation (https://documentation.suse.com/cloudnative/suse-private-registry/html/private-registry/pr-deployment.html#pr-deployment-kube-secrets); the `<AUTH>` value set in the `auth` field is the base64 encoding of the `<USERNAME>:<PASSWORD>` concatenation.

To set the content of the `tls.crt` and `tls.key` fields in the `suse-registry-tls` Secret manifest above, you can generate your own TLS self-signed certificate and related private key by running the following commands:

```
# Generate a self-signed certificate and key
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 365 -nodes

# Convert them to base64 for the suse-private-registry-creds.yaml file
cat cert.pem | base64 -w 0
cat key.pem | base64 -w 0
```

31 Image creation

Once the directory structure is prepared following the previous sections (for both, connected and air-gap scenarios), run the following command to build the image:

```
podman run --rm --privileged -it -v $PWD:/eib \
 registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
 build --definition-file mgmt-cluster.yaml
```

This creates the ISO output image file that, in our case, based on the image definition described above, is `eib-mgmt-cluster-image.iso`.

32 Provision the management cluster

The previous image contains all components explained above, and it can be used to provision the management cluster using a virtual machine or a bare-metal server (using the virtual-media feature).

33 Dual-stack considerations and configuration

The examples shown in the previous sections provide guidance and examples on how to set up a single-stack IPv4 management cluster. Such a management cluster is independent of the operational status of downstream clusters, which can be individually configured to operate in either IPv4/IPv6 single-stack or dual-stack configuration, once deployed. However, they way the management cluster is configured has a direct impact on the communication protocols that can be used during the provisioning phase, where both the in-band and out-of-band communications must happen according to which protocols are supported by the management cluster and downstream host. In case some or all of the BMCs and/or downstream cluster nodes are expected to use IPv6, a dual-stack setup for the management cluster is then required.

Note

Single-stack IPv6 management clusters are not yet supported.

In order to achieve dual-stack functionality, Kubernetes must be provided with both IPv4 and IPv6 CIDRs for PODs and Services. However, other components also require specific tuning before building the management cluster image with EIB. The Metal^3 provisioning services (Ironic) can be configured in different ways, depending on your infrastructure or requirements:

- The Ironic services can be configured to listen on all the interfaces on the system rather than a single IP address, thus, as long as the management cluster host(s) has both IPv4 and IPv6 addresses assigned to the relevant interface, any of them can potentially be used during the provisioning. Note that at this time only one of these addresses can be selected for the URL generation (for consumption by other services, e.g. the Baremetal Operator, BMCs, etc.); as a consequence, to enable IPv6 communications with the BMCs, the Baremetal Operator can be instructed to expose and pass on an IPv6 URL when dealing BMH definitions including an IPv6 address. In other words, when a BMC is identified as IPv6 capable, the provisioning will be performed via IPv6 only, and via IPv4 in all the other cases.

- A single hostname, resolving to both IPv4 and IPv6, can be used by Metal^3 to let Ironic use those addresses for binding and URL creation. This approach allows for an easy configuration and flexible behavior (both IPv4 and IPv6 remain viable at each provisioning step), but requires an infrastructure with preexisting DNS servers, IP allocations and records already in place.

In both cases, Kubernetes will need to know what CIDRs to use for both IPv4 and IPv6, so you can add the following lines to your `kubernetes/config/server.yaml` in the EIB directory, making sure to list IPv4 first:

```
service-cidr: 10.96.0.0/12,fd12:4567:789c::/112
cluster-cidr: 193.168.0.0/18,fd12:4567:789b::/48
```

Some containers leverage the host networking, so modify the network configuration for the host(s), under the `network` directory, to enable IPv6 connectivity:

```
routes:
  config:
  - destination: 0.0.0.0/0
    next-hop-address: ${MGMT_GATEWAY_V4}
    next-hop-interface: eth0
  - destination: ::/0
    next-hop-address: ${MGMT_GATEWAY_V6}
    next-hop-interface: eth0
dns-resolver:
  config:
    server:
    - ${MGMT_DNS}
    - 8.8.8.8
    - 2001:4860:4860::8888
interfaces:
- name: eth0
  type: ethernet
  state: up
  mac-address: ${MGMT_MAC}
  ipv4:
    address:
    - ip: ${MGMT_CLUSTER_IP_V4}
      prefix-length: 24
    dhcp: false
    enabled: true
  ipv6:
    address:
    - ip: ${MGMT_CLUSTER_IP_V6}
      prefix-length: 128
    dhcp: false
    autoconf: false
    enabled: true
```

Replace the placeholders with the gateway IP addresses, additional DNS server (if needed), the MAC address of the network interface and the the IP addressed of the management cluster. If address autoconfiguration is preferred instead, refer to the following excerpt and just set the `${MGMT_MAC}` variable:

```
interfaces:
- name: eth0
  type: ethernet
  state: up
  mac-address: ${MGMT_MAC}
  ipv4:
    enabled: true
    dhcp: true
  ipv6:
    enabled: false
    dhcp: true
    autoconf: true
```

We can now define the remaining files for a single node configuration, starting from the first option, by creating `kubernetes/helm/values/metal3.yaml` as:

```
global:
  ironicIP: ${MGMT_CLUSTER_IP_V4}
  enable_vmedia_tls: false
  # trustedCAs: tls-ca-bundle  # Optional: Uncomment and set to ConfigMap name for
 trusted CA bundle
metal3-ironic:
  global:
    predictableNicNames: true
  listenOnAll: true
  persistence:
    ironic:
      size: "5Gi"
  service:
    type: NodePort
metal3-baremetal-operator:
  baremetaloperator:
    externalHttpIPv6: ${MGMT_CLUSTER_IP_V6}
```

and `kubernetes/helm/values/rancher.yaml` as:

```
hostname: rancher-${MGMT_CLUSTER_IP_V4}.sslip.io
bootstrapPassword: "foobar"
replicas: 1
global:
  cattle:
    systemDefaultRegistry: "registry.rancher.com"
```

where `${MGMT_CLUSTER_IP_V4}` and `${MGMT_CLUSTER_IP_V6}` are the IP addresses previously assigned to the host.

Alternatively, to use the hostname in place of the IP addresses, modify `kubernetes/helm/values/metal3.yaml` to:

```
global:
  provisioningHostname: `${MGMT_CLUSTER_HOSTNAME}`
  enable_vmedia_tls: false
  # trustedCAs: tls-ca-bundle  # Optional: Uncomment and set to ConfigMap name for
 trusted CA bundle
metal3-ironic:
  global:
    predictableNicNames: true
  persistence:
    ironic:
      size: "5Gi"
  service:
    type: NodePort
```

and `kubernetes/helm/values/rancher.yaml` to:

```
hostname: rancher-${MGMT_CLUSTER_HOSTNAME}.sslip.io
bootstrapPassword: "foobar"
replicas: 1
global:
  cattle:
    systemDefaultRegistry: "registry.rancher.com"
```

where `${MGMT_CLUSTER_HOSTNAME}` should be a Fully Qualified Domain Name resolving to your host IP addresses.

For more information visit SUSE Telco Cloud GitHub repository under the "dual-stack" folder (https://github.com/suse-edge/telco-cloud-examples/tree/main/telco-examples/mgmt-cluster/dual-stack), where an example directory structure can be found.

VI Telco features configuration

34 Kernel image for real time 210

35 Kernel arguments for low latency and high performance 211

36 CPU Pinning on Host 215

37 CPU Pinning on Kubernetes 219

38 Network Parameters 222

39 CNI Configuration 224

40 SR-IOV 231

41 DPDK 243

42 vRAN Acceleration (`Intel ACC100/VRB1/VRB2`) 246

43 Huge pages 250

44 NUMA-aware scheduling 253

45 Metal LB 254

46 Private registry configuration 256

47 Precision Time Protocol 258

48 SCTP - Stream Control Transmission Protocol 294

This section documents and explains the configuration of Telco-specific features on clusters deployed via SUSE Telco Cloud.

The directed network provisioning deployment method is used, as described in the Automated Provisioning (*Part VII, "Fully automated directed network provisioning"*) section.

The following topics are covered in this section:

- Kernel image for real time (*Chapter 34, Kernel image for real time*): Kernel image to be used by the real-time kernel.
- Kernel arguments for low latency and high performance (*Chapter 35, Kernel arguments for low latency and high performance*): Kernel arguments to be used for maximum performance and low latency running telco workloads.
- CPU Pinning on Host (*Chapter 36, CPU Pinning on Host*): Isolating the CPUs via kernel arguments and TuneD profile.
- CPU Pinning on Kubernetes (*Chapter 37, CPU Pinning on Kubernetes*): Isolating the CPUs on Kubernetes via Kubelet configuration.
- Network Parameters (*Chapter 38, Network Parameters*): Network tuning parameters for optimal performance and low latency.
- CNI configuration (*Chapter 39, CNI Configuration*): CNI configuration to be used by the Kubernetes cluster.
- SR-IOV configuration (*Chapter 40, SR-IOV*): SR-IOV configuration to be used by the Kubernetes workloads.
- DPDK configuration (*Chapter 41, DPDK*): DPDK configuration to be used by the system.
- vRAN Acceleration (*Chapter 42, vRAN Acceleration (`Intel ACC100/VRB1/VRB2`)*): Offloading FEC algorithm computation to vRAN Acceleration card.
- Huge pages (*Chapter 43, Huge pages*): Huge pages configuration to be used by the Kubernetes workloads.
- NUMA-aware scheduling configuration (*Chapter 44, NUMA-aware scheduling*): NUMA-aware scheduling configuration to be used by the Kubernetes workloads.
- Metal LB configuration (*Chapter 45, Metal LB*): Metal LB configuration to be used by the Kubernetes workloads.
- Private registry configuration (*Chapter 46, Private registry configuration*): Private registry configuration to be used by the Kubernetes workloads.

- Precision Time Protocol configuration (*Chapter 47, Precision Time Protocol*): Configuration files for running PTP telco profiles.
- SCTP configuration (*Chapter 48, SCTP - Stream Control Transmission Protocol*): Enable SCTP layer in Linux kernel IP stack.

34 Kernel image for real time

The real-time kernel image is not necessarily better than a standard kernel. It is a different kernel tuned to a specific use case. The real-time kernel is tuned for lower latency at the cost of throughput. The real-time kernel is not recommended for general purpose use, but in our case, this is the recommended kernel for Telco Workloads where latency is a key factor.

There are four top features:

- Deterministic execution:
  Get greater predictability — ensure critical business processes complete in time, every time and deliver high-quality service, even under heavy system loads. By shielding key system resources for high-priority processes, you can ensure greater predictability for time-sensitive applications.

- Low jitter:
  The low jitter built upon the highly deterministic technology helps to keep applications synchronized with the real world. This helps services that need ongoing and repeated calculation.

- Priority inheritance:
  Priority inheritance refers to the ability of a lower priority process to assume a higher priority when there is a higher priority process that requires the lower priority process to finish before it can accomplish its task. SUSE Linux Enterprise Real Time solves these priority inversion problems for mission-critical processes.

- Thread interrupts:
  Processes running in interrupt mode in a general-purpose operating system are not preemptible. With SUSE Linux Enterprise Real Time, these interrupts have been encapsulated by kernel threads, which are interruptible, and allow the hard and soft interrupts to be preempted by user-defined higher priority processes.
  In our case, if you have installed a real-time image like `SUSE Linux Micro RT`, kernel real time is already installed. From the SUSE Customer Center (https://scc.suse.com/), you can download the real-time kernel image.

Note

For more information about the real-time kernel, visit SUSE Real Time (https://www.suse.com/products/realtime/).

35 Kernel arguments for low latency and high performance

Configuring the appropriate kernel arguments is essential for optimizing performance, achieving low latency, and ensuring successful cluster deployment for telco workloads. While some parameters are designed specifically to enable the real-time kernel to function optimally, this section applies to both RT and default kernel configurations. Additionally, certain arguments are mandatory for Directed Network Provisioning method to successfully deploy downstream cluster nodes.

- Remove `kthread_cpus` when using SUSE real-time kernel. This parameter controls on which CPUs kernel threads are created. It also controls which CPUs are allowed for PID 1 and for loading kernel modules (the kmod user-space helper). This parameter is not recognized and does not have any effect.

- Isolate the CPU cores using `isolcpus`, `nohz_full`, `rcu_nocbs`, and `irqaffinity`. For a comprehensive list of CPU pinning techniques, refer to CPU Pinning on Host (*Chapter 36, CPU Pinning on Host*) chapter.

- Add `domain,nohz,managed_irq` flags to `isolcpus` kernel argument. Without any flags, `isolcpus` is equivalent to specifying only the `domain` flag. This isolates the specified CPUs from scheduling, including kernel tasks. The `nohz` flag stops the scheduler tick on the specified CPUs (if only one task is runnable on a CPU), and the `managed_irq` flag avoids routing managed external (device) interrupts at the specified CPUs. Note that the IRQ lines of NVMe devices are fully managed by the kernel and will be routed to the non-isolated (housekeeping) cores as a consequence. For example, the command line provided at the end of this section will result in only four queues (plus an admin/control queue) allocated on the system:

  ```
  for I in $(grep nvme0 /proc/interrupts | cut -d ':' -f1); do cat /proc/irq/${I}/
  effective_affinity_list; done | column
  39      0       19      20      39
  ```

  This behavior prevents any disruption caused by disk I/O to any time sensitive application running on the isolated cores, but might require attention and careful design for storage focused workloads.

- Tune the ticks (kernel's periodic timer interrupts):
  - `skew_tick=1`: ticks can sometimes happen simultaneously. Instead of all CPUs receiving their timer tick at the exact same moment, `skew_tick=1` makes them occur at slightly offset times. This helps reduce system jitter, resulting in more consistent and lower interrupt response times (an essential requirement for latency-sensitive applications).
  - `nohz=on`: stops the periodic timer tick on idle CPUs.
  - `nohz_full=<cpu-cores>`: Stops the periodic timer tick on specified CPUs that are dedicated for real-time applications.
- Disable Machine Check Exception (MCE) handling by specifying `mce=off`. MCEs are hardware errors detected by the processor and disabling them can avoid noisy logs.
- Add `nowatchdog` to disable the soft-lockup watchdog which is implemented as a timer running in the timer hard-interrupt context. When it expires (i.e. a soft lockup is detected), it will print a warning (in the hard interrupt context), running any latency targets. Even if it never expires, it goes onto the timer list, slightly increasing the overhead of every timer interrupt. This option also disables the NMI watchdog, so NMIs cannot interfere.
- `nmi_watchdog=0` disables the NMI (Non-Maskable Interrupt) watchdog. This can be omitted when `nowatchdog` is used.
- RCU (Read-Copy-Update) is a kernel mechanism that enables concurrent, lock-free access for many readers to shared data. An RCU callback, a function triggered after a 'grace period', ensures all previous readers have finished so old data can be safely reclaimed. We fine-tune RCU, particularly for sensitive workloads, to offload these callbacks from dedicated (pinned) CPUs, preventing kernel operations from interfering with critical, time-sensitive tasks.
  - Specify the pinned CPUs in `rcu_nocbs` so that RCU callbacks do not run on them. This helps reducing jitter and latency for the real-time workloads.
  - `rcu_nocb_poll` makes the no-callback CPUs regularly 'poll' to see if callback handling is required. This can reduce the interrupt overhead.
  - `rcupdate.rcu_cpu_stall_suppress=1` suppresses RCU CPU stall warnings, which can sometimes be false positives in heavily loaded real-time systems
  - `rcupdate.rcu_expedited=1` speeds up the grace period for RCU operations, making read-side critical sections more responsive

- `rcupdate.rcu_normal_after_boot=1` When used with rcu_expedited, it allows RCU to revert to normal (non-expedited) operation after the system boot.
- `rcupdate.rcu_task_stall_timeout=0` disables the RCU task stall detector, preventing potential warnings or system halts from long-running RCU tasks.
- `rcutree.kthread_prio=99` sets the priority of the RCU callback kernel thread to the highest possible value (99), ensuring it is scheduled immediately to handle callbacks promptly when needed.

- Add `ignition.platform.id=openstack` for Metal[3] and Cluster API to successfully provision/deprovision the cluster. This is used by Metal[3] Python agent, which originated from OpenStack Ironic.
- Enable Predictable Network Interface Naming (https://documentation.suse.com/smart/network/html/network-interface-predictable-naming/index.html) via `net.ifnames=1`. From SUSE Linux Micro 6.2 onwards, this is enabled by default and explicit configuration is not required. For versions prior to 6.2, this must be explicitly set as a kernel argument. This aligns with the `predictableNicNames` configuration in the Management Cluster's Metal[3] Helm chart, which is required for Directed Network Provisioning to function correctly. Consistent interface naming is also critical when SR-IOV is utilized.
- Remove `intel_pstate=passive`. This option configures `intel_pstate` to work with generic cpufreq governors, but to make this work, it disables hardware-managed P-states (`HWP`) as a side effect. To reduce the hardware latency, this option is not recommended for real-time workloads.
- Replace `intel_idle.max_cstate=0 processor.max_cstate=1` with `idle=poll`. To avoid C-State transitions, the `idle=poll` option is used to disable the C-State transitions and keep the CPU in the highest C-State. The `intel_idle.max_cstate=0` option disables `intel_idle`, so `acpi_idle` is used, and `acpi_idle.max_cstate=1` then sets max C-state for acpi_idle. On AMD64/Intel 64 architectures, the first ACPI C-State is always `POLL`, but it uses a `poll_idle()` function, which may introduce some tiny latency by reading the clock periodically, and restarting the main loop in `do_idle()` after a timeout (this also involves clearing and setting the `TIF_POLL`

task flag). In contrast, `idle=poll` runs in a tight loop, busy-waiting for a task to be rescheduled. This minimizes the latency of exiting the idle state, but at the cost of keeping the CPU running at full speed in the idle thread.

- Disable C1E in BIOS. This option is important to disable the C1E state in the BIOS to avoid the CPU from entering the C1E state when idle. The C1E state is a low-power state that can introduce latency when the CPU is idle.

The rest of this documentation covers additional parameters, including huge pages and IOMMU.

This provides an example of kernel arguments for a 32-core Intel server, including the aforementioned adjustments:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt root=UUID=77b713de-5cc7-4d4c-8fc6-
f5eca0a43cf9 skew_tick=1 rd.timeout=60 rd.retry=45 console=ttyS1,115200
 console=tty0 default_hugepagesz=1G hugepagesz=1G hugepages=40 hugepagesz=2M
 hugepages=0 ignition.platform.id=openstack net.ifnames=1 intel_iommu=on iommu=pt
 irqaffinity=0,31,32,63 isolcpus=domain,nohz,managed_irq,1-30,33-62 nohz_full=1-30,33-62
 nohz=on mce=off nosoftlockup nowatchdog nmi_watchdog=0 quiet rcu_nocb_poll
 rcu_nocbs=1-30,33-62 rcupdate.rcu_cpu_stall_suppress=1 rcupdate.rcu_expedited=1
 rcupdate.rcu_normal_after_boot=1 rcupdate.rcu_task_stall_timeout=0
 rcutree.kthread_prio=99 security=selinux selinux=1 idle=poll
```

Here is another configuration example for a 64-core AMD server. Among the 128 logical processors (`0-127`), first 8 cores (`0-7`) are designated for housekeeping, while the remaining 120 cores (`8-127`) are pinned for the applications:

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt root=UUID=575291cf-74e8-42cf-8f2c-408a20dc00b8
 skew_tick=1 console=ttyS1,115200 console=tty0 default_hugepagesz=1G hugepagesz=1G
 hugepages=40 hugepagesz=2M hugepages=0 ignition.platform.id=openstack net.ifnames=1
 amd_iommu=on iommu=pt irqaffinity=0-7 isolcpus=domain,nohz,managed_irq,8-127
 nohz_full=8-127 rcu_nocbs=8-127 mce=off nohz=on nowatchdog nmi_watchdog=0 nosoftlockup
 quiet rcu_nocb_poll rcupdate.rcu_cpu_stall_suppress=1 rcupdate.rcu_expedited=1
 rcupdate.rcu_normal_after_boot=1 rcupdate.rcu_task_stall_timeout=0
 rcutree.kthread_prio=99 security=selinux selinux=1 idle=poll
```

36 CPU Pinning on Host

CPU pinning, also known as processor affinity, is the technique of binding a process or thread to a specific CPU core, preventing the operating system's scheduler from moving it. By ensuring a process always runs on the same core, it benefits from faster access to data that remains in that core's cache memory. This practice is common in high-performance computing environments because it dramatically improves performance and reduces overhead.

36.1 Isolating CPUs via TuneD

`tuned` is a system tuning tool that monitors system conditions to optimize performance using various predefined profiles. A key feature is its ability to isolate CPU cores for specific workloads, like real-time applications. This prevents the OS from utilizing these cores and potentially increasing latency.

To enable and configure this feature, the first thing is to create a profile for the CPU cores we want to isolate. In this example, among 64 cores, we dedicate 60 cores (`1-30,33-62`) for the application and remaining 4 cores are used for housekeeping. Note that the design of isolated CPUs heavily depends on the real-time applications.

```
$ echo "export tuned_params" >> /etc/grub.d/00_tuned

$ echo "isolated_cores=1-30,33-62" >> /etc/tuned/cpu-partitioning-variables.conf

$ tuned-adm profile cpu-partitioning
Tuned (re)started, changes applied.
```

36.2 Isolating CPUs via kernel arguments

Then we need to modify the GRUB option to isolate CPU cores and other important parameters for CPU usage. The following options are important to be customized with your current hardware specifications:

parameter	value	description
isolcpus	domain,nohz,managed_irq,1-30,33-62	Isolate the cores 1-30 and 33-62. `domain` indicates these CPUs are part of isolation domain. `nohz` enables tickless operation on these isolated CPUs when

parameter	value	description
		they are idle, to reduce interruptions. `managed_irq` isolates pinned CPUs from being targeted by IRQs. This contemplates `irqaffinity=0-7`, which already directs mosts IRQs to the housekeeping cores.
skew_tick	1	This option allows the kernel to skew the timer interrupts across the isolated CPUs.
nohz	on	When enabled, the kernel’s periodic timer interrupt (the 'tick') stops on any CPU core that becomes idle. This primarily benefits the housekeeping CPUs (`0,31,32,63`), conserving power and reducing unnecessary wake-ups on those general-purpose cores.
nohz_full	1-30,33-62	For the isolated cores, this stops the tick and it does so even when the CPU is running a single active task. It means it makes the CPU run in full tickless mode (or 'dyntick'). The kernel will only deliver timer interrupts when they are actually needed.
rcu_nocbs	1-30,33-62	This option offloads the RCU callback processing from specified CPU cores.

parameter	value	description
rcu_nocb_poll		When this option is set, no-RCU-callback CPUs will regularly 'poll' to see if callback handling is required, rather than being explicitly woken up by other CPUs. This can reduce the interrupt overhead.
irqaffinity	0,31,32,63	This option allows the kernel to run the interrupts to the housekeeping cores.
idle	poll	This minimizes the latency of exiting the idle state, but at the cost of keeping the CPU running at full speed in the idle thread.
nmi_watchdog	0	This option disables only the NMI watchdog. This can be omitted when `nowatchdog` is set.
nowatchdog		This option disables the soft-lockup watchdog which is implemented as a timer running in the timer hard-interrupt context.

The following commands modify the GRUB configuration and apply the changes mentioned above to be present on the next boot:

Edit the `/etc/default/grub` file with above parameters and the file will look like this:

```
GRUB_CMDLINE_LINUX="BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt
 root=UUID=77b713de-5cc7-4d4c-8fc6-f5eca0a43cf9 skew_tick=1 rd.timeout=60
 rd.retry=45 console=ttyS1,115200 console=tty0 default_hugepagesz=1G hugepagesz=1G
 hugepages=40 hugepagesz=2M hugepages=0 ignition.platform.id=openstack
 net.ifnames=1 intel_iommu=on iommu=pt irqaffinity=0,31,32,63
 isolcpus=domain,nohz,managed_irq,1-30,33-62 nohz_full=1-30,33-62 nohz=on
```

```
mce=off nosoftlockup nowatchdog nmi_watchdog=0 quiet rcu_nocb_poll
rcu_nocbs=1-30,33-62 rcupdate.rcu_cpu_stall_suppress=1 rcupdate.rcu_expedited=1
rcupdate.rcu_normal_after_boot=1 rcupdate.rcu_task_stall_timeout=0
rcutree.kthread_prio=99 security=selinux selinux=1 idle=poll"
```

Update the GRUB configuration:

```
$ transactional-update grub.cfg
$ reboot
```

To validate that the parameters are applied after the reboot, the following command can be used to check the kernel command line:

```
$ cat /proc/cmdline
```

There is another script that can be used to tune the CPU configuration and network parameters, which basically is doing the following steps:

- Set the CPU governor to `performance`.
- Unset the timer migration to the isolated CPUs.
- Migrate the kdaemon threads to the housekeeping CPUs.
- Set the isolated CPUs latency to the lowest possible value.
- Delay the vmstat updates to 300 seconds.
- Configure network tuning parameters for optimal performance.

The script is available at SUSE Telco Cloud Examples repository (https://raw.githubusercontent.com/suse-edge/telco-cloud-examples/refs/heads/release-3.7/telco-examples/downstream-clusters/dhcp-less/eib/custom/files/performance-settings.sh).

37 CPU Pinning on Kubernetes

37.1 RKE2 Versions < v1.32

Enable CPU pinning in your RKE2 cluster by editing RKE2 config file. Add below kubelet arguments in `/etc/rancher/rke2/config.yaml` file. Make sure specifying the housekeeping CPU cores in `kubelet-reserved` and `system-reserved` arguments:

```
kubelet-arg:
- "cpu-manager-policy=static"
- "cpu-manager-policy-options=full-pcpus-only=true"
- "cpu-manager-reconcile-period=0s"
- "kubelet-reserved=cpu=0,31,32,63"
- "system-reserved=cpu=0,31,32,63"
```

37.2 RKE2 Versions >= v1.32

If your RKE2 version is v1.32 or higher, command-line arguments cannot be used to configure kubelet, following upstream Kubernetes practice. To set up CPU pinning, a kubelet config file needs to be created. Refer to RKE2 documentation (https://documentation.suse.com/cloudnative/rke2/latest/en/install/configuration.html#_kubelet_configuration).

Create a new kubelet config file such as `01-cpu-pinning.conf` and place it in the `/var/lib/rancher/rke2/agent/etc/kubelet.conf.d/` directory:

```
apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
cpuManagerPolicy: static
reservedSystemCPUs: 0,31,32,63
topologyManagerPolicy: single-numa-node
```

For configuration changes to take effect, a restart of the appropriate RKE2 service (server or agent) is required. This action will briefly interrupt RKE2 service on the host. Run only one of the following commands, depending on the node type:

```
# If the node is RKE2 agent
systemctl restart rke2-agent
# Else if the node is RKE2 server
systemctl restart rke2-server
```

37.3 Deploy Workloads Leveraging Pinned CPUs

There are three ways to use the feature using the `Static Policy` defined in kubelet depending on the requests and limits you define on your workload:

1. `BestEffort` QoS Class: If you do not define any request or limit for `CPU`, the pod is scheduled on the first `CPU` available on the system.
   An example of using the `BestEffort` QoS Class could be:

   ```
   spec:
     containers:
     - name: nginx
       image: nginx
   ```

2. `Burstable` QoS Class: If you define a request for CPU, which is not equal to the limits, or there is no CPU request.
   Examples of using the `Burstable` QoS Class could be:

   ```
   spec:
     containers:
     - name: nginx
       image: nginx
       resources:
         limits:
           memory: "200Mi"
         requests:
           memory: "100Mi"
   ```

   or

   ```
   spec:
     containers:
     - name: nginx
       image: nginx
       resources:
         limits:
           memory: "200Mi"
           cpu: "2"
         requests:
           memory: "100Mi"
           cpu: "1"
   ```

3. `Guaranteed` QoS Class: If you define a request for CPU, which is equal to the limits.

An example of using the `Guaranteed` QoS Class could be:

```
spec:
  containers:
    - name: nginx
      image: nginx
      resources:
        limits:
          memory: "200Mi"
          cpu: "2"
        requests:
          memory: "200Mi"
          cpu: "2"
```

38 Network Parameters

Network tuning is essential for achieving optimal performance in telco workloads. The performance-settings.sh script includes a `configure_network_tuning` function that sets various kernel network parameters to optimize buffer sizes, packet processing, and UDP performance.

Note

The network tuning values provided are examples that have been tested and validated. However, they should be adapted to your specific use case, workload requirements, and hardware configuration.

The following network parameters are configured by the script:

38.1 Core Network Parameters

- `net.core.rmem_max` (1342177280 bytes): Maximum receive socket buffer size. This parameter sets the maximum amount of memory that can be allocated for receiving data on any socket.
- `net.core.wmem_max` (516777216 bytes): Maximum send socket buffer size. This parameter sets the maximum amount of memory that can be allocated for sending data on any socket.
- `net.core.rmem_default` (10000000 bytes): Default receive socket buffer size. This is the initial buffer size for receiving data on new sockets.
- `net.core.wmem_default` (10000000 bytes): Default send socket buffer size. This is the initial buffer size for sending data on new sockets.
- `net.core.netdev_max_backlog` (416384 packets): Maximum number of packets queued on the input side when the interface receives packets faster than the kernel can process them. Increasing this value helps prevent packet drops during traffic bursts.
- `net.core.optmem_max` (25165824 bytes): Maximum ancillary buffer size allowed per socket. This affects the size of option memory buffers.
- `net.core.netdev_budget` (1024 packets): Maximum number of packets that can be processed in one softirq run. This helps control CPU utilization during packet processing.

38.2 UDP-Specific Parameters

- `net.ipv4.udp_mem` (11416320 15221760 22832640 pages): Controls the amount of memory (in pages) used by UDP. The three values represent:
  - Minimum threshold: Below this, UDP is not pressured
  - Pressure threshold: UDP starts applying memory pressure
  - Maximum threshold: Maximum pages of memory for UDP
- `net.ipv4.udp_rmem_min` (16384 bytes): Minimum receive buffer size for UDP sockets. Ensures a baseline buffer size for UDP reception.
- `net.ipv4.udp_wmem_min` (16384 bytes): Minimum send buffer size for UDP sockets. Ensures a baseline buffer size for UDP transmission.

38.3 Real-Time Scheduling

- `kernel.sched_rt_runtime_us` (-1): Disables the real-time throttling mechanism. Setting this to -1 allows real-time tasks unlimited CPU time, which is essential for telco workloads requiring deterministic latency.

Important

These parameters are automatically configured when using the performance-settings.sh script from the SUSE Telco Cloud Examples repository. The script should be integrated into your image build process as described in the Automated Provisioning (*Part VII, "Fully automated directed network provisioning"*) section.

39 CNI Configuration

39.1 Cilium

`Cilium` is the default CNI plug-in for SUSE Telco Cloud. To enable Cilium on RKE2 cluster as the default plug-in, the following configuration is required in the `/etc/rancher/rke2/config.yaml` file:

```
cni:
- cilium
```

This can also be specified with command-line arguments, that is, `--cni=cilium` into the server line in `/etc/systemd/system/rke2-server` file.

To use the `SR-IOV` network described in *Section 40.2, "Option 2 (Recommended): SR-IOV Network Operator"* along with Cilium, deploy `multus` meta plugin. Make sure `multus` is listed before other CNIs.

```
cni:
- multus
- cilium
```

39.2 Calico

`Calico` is another CNI plug-in for SUSE Telco Cloud for Telco. To enable Calico on RKE2 cluster as the default plug-in, the following configuration is required in the `/etc/rancher/rke2/config.yaml` file:

```
cni:
- calico
```

This can also be specified with command-line arguments, that is, `--cni=calico` into the server line in `/etc/systemd/system/rke2-server` file.

To use the `SR-IOV` network described in *Section 40.2, "Option 2 (Recommended): SR-IOV Network Operator"* along with Calico, deploy `multus` meta plugin. Make sure `multus` is listed before other CNIs.

```
cni:
- multus
- calico
```

> **Note**
>
> For more information about CNI plug-ins, see Network Options (https://docs.rke2.io/install/network_options).

39.3 Bond CNI

In general terms, bonding provides a method for aggregating multiple network interfaces into a single logical "bonded" interface. This is typically used to increase service availability by introducing redundant networking paths, but can also be used to increase bandwidth with certain bond modes. The following CNI plug-ins are supported for the Bond CNI plugin in combination with multus:

- MACVLAN
- Host Device
- SR-IOV

39.3.1 Bond CNI with MACVLAN

To use the Bond CNI plugin with MACVLAN two free interfaces are needed in the container. The following example uses 'enp8s0' and 'enp9s0'. Start by creating network attachment definitions for them:

NetworkAttachmentDefinition enp8s0

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: enp8s0-conf
spec:
  config: '{
      "cniVersion": "0.3.1",
      "plugins": [
        {
          "type": "macvlan",
          "capabilities": { "ips": true },
          "master": "enp8s0",
          "mode": "bridge",
          "ipam": {}
        }, {
          "capabilities": { "mac": true },
```

```
          "type": "tuning"
        }
      ]
    }'
```

NetworkAttachmentDefinition enp9s0

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: enp9s0-conf
spec:
  config: '{
      "cniVersion": "0.3.1",
      "plugins": [
        {
          "type": "macvlan",
          "capabilities": { "ips": true },
          "master": "enp9s0",
          "mode": "bridge",
          "ipam": {}
        }, {
          "capabilities": { "mac": true },
          "type": "tuning"
        }
      ]
    }'
```

After this, add a network attachment definition for the bond itself.

NetworkAttachmentDefinition bond

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: bond-net1
spec:
  config: '{
  "type": "bond",
  "cniVersion": "0.3.1",
  "name": "bond-net1",
  "mode": "active-backup",
  "failOverMac": 1,
  "linksInContainer": true,
  "miimon": "100",
  "mtu": 1500,
  "links": [
     {"name": "net1"},
```

```
     {"name": "net2"}
  ],
  "ipam": {
    "type": "static",
    "addresses": [
      {
        "address": "192.168.200.100/24",
        "gateway": "192.168.200.1"
      }
    ],
    "subnet": "192.168.200.0/24",
    "routes": [{
      "dst": "0.0.0.0/0"
    }]
  }
}'
```

The IP address assignment here is static and defines the address of the bond as '192.168.200.100' on a /24 network, with a gateway residing on the network's first available address. In the bond's network attachment we also define the type of bond we want. In this case it is active-backup.

To use this bond, the pod needs to know about all interfaces. An example pod definition might look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
  annotations:
        k8s.v1.cni.cncf.io/networks: '[
{"name": "enp8s0-conf",
"interface": "net1"
},
{"name": "enp9s0-conf",
"interface": "net2"
},
{"name": "bond-net1",
"interface": "bond0"
}]'
spec:
  restartPolicy: Never
  containers:
  - name: bond-test
    image: alpine:latest
    command:
      - /bin/sh
      - "-c"
```

```
      - "sleep 60m"
    imagePullPolicy: IfNotPresent
```

Note how the annotation refers to all networks and how it defines the mapping between the interfaces 'enp8s0 → net1', and 'enp9s0→net2'.

39.3.2 Bond CNI with Host Device

To use the Bond CNI plugin with host device, two free interfaces are needed on the host. These interfaces are then mapped through to the container. The following example uses 'enp8s0' and 'enp9s0'. Start by creating network attachment definitions for them:

NetworkAttachmentDefinition enp8s0

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: enp8s0-hostdev
spec:
  config: '{
      "cniVersion": "0.3.1",
      "plugins": [
        {
          "type": "host-device",
          "name": "host0",
          "device": "enp8s0",
          "ipam": {}
        }]
    }'
```

NetworkAttachmentDefinition enp9s0

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: enp9s0-hostdev
spec:
  config: '{
      "cniVersion": "0.3.1",
      "plugins": [
        {
          "type": "host-device",
          "name": "host0",
          "device": "enp9s0",
          "ipam": {}
```

```
        }]
    }'
```

After this, add network attachment definition for the bond itself. This is similar to the MACVLAN use case.

NetworkAttachmentDefinition bond

```
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: bond-net1
spec:
  config: '{
  "type": "bond",
  "cniVersion": "0.3.1",
  "name": "bond-net1",
  "mode": "active-backup",
  "failOverMac": 1,
  "linksInContainer": true,
  "miimon": "100",
  "mtu": 1500,
  "links": [
     {"name": "net1"},
     {"name": "net2"}
  ],
  "ipam": {
    "type": "static",
    "addresses": [
      {
        "address": "192.168.200.100/24",
        "gateway": "192.168.200.1"
      }
    ],
    "subnet": "192.168.200.0/24",
    "routes": [{
      "dst": "0.0.0.0/0"
    }]
  }
}'
```

The IP address assignment here is static and defines the address of the bond as '192.168.200.100' on a /24 network, with a gateway residing on the network’s first available address. In the bond’s network attachment, define the type of bond. In this case it is active-backup.

To use this bond, the pod needs to know about all interfaces. An example pod definition for bond with host devices might look like this:

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: test-pod
  annotations:
        k8s.v1.cni.cncf.io/networks: '[
{"name": "enp8s0-hostdev",
"interface": "net1"
},
{"name": "enp9s0-hostdev",
"interface": "net2"
},
{"name": "bond-net1",
"interface": "bond0"
}]'
spec:
  restartPolicy: Never
  containers:
  - name: bond-test
    image: alpine:latest
    command:
      - /bin/sh
      - "-c"
      - "sleep 60m"
    imagePullPolicy: IfNotPresent
```

39.3.3 Bond CNI with SR-IOV

Using the Bond CNI with SR-IOV is fairly straightforward. For more details on how to set up SR-IOV, see *Chapter 40, SR-IOV*. As described there, you have to create `SriovNetworkNodePolicies` that defines `resourceNames`, as well as number of virtual functions and such. The `resourceNames` are being used by the `SriovNetwork` which is used as interfaces in the pod definition. The bond definition is exactly the same as for the MACVLAN and host device cases.

Note

Bond CNI with SR-IOV is only applicable to SRIOV Virtual Functions (VF) using the kernel driver. Userspace driver VFs - such as those used in DPDK workloads - cannot be bonded with the Bond CNI.

40 SR-IOV

SR-IOV (Single Root I/O Virtualization) allows a single physical device, such as a network adapter, to separate its resources across multiple `PCIe` hardware functions. This enables direct resource access for various applications.

We provide two distinct methods for deploying `SR-IOV` in your cluster:

- *Section 40.1, "Option 1: SR-IOV Network Device Plugin Daemonset and configMap"*: This method supports both network devices and vRAN accelerator.
- *Section 40.2, "Option 2 (Recommended): SR-IOV Network Operator"*: This automated method provides simpler deployment. This method is only for network devices.

In rare cases where you need both solutions - using the Network Operator for network devices and the Device Plugin for vRAN Accelerators - you must deploy them into separate Kubernetes namespaces. This separation is essential to prevent conflicts between two deployments.

40.1 Option 1: SR-IOV Network Device Plugin Daemonset and configMap

SR-IOV Network Device Plugin (https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin) discovers and advertises network resources, such as PCI physical functions (PFs), and their virtual functions (VFs), on a Kubernetes host.

- Prepare the config map for the device plugin

We need to create a config map that defines SR-IOV resource pools. Run `lspci` command to retrieve the information:

```
$ lspci | grep -i acc
07:00.0 Processing accelerators: Intel Corporation Device 57c2
07:00.1 Processing accelerators: Intel Corporation Device 57c3
07:00.2 Processing accelerators: Intel Corporation Device 57c3
07:00.3 Processing accelerators: Intel Corporation Device 57c3
07:00.4 Processing accelerators: Intel Corporation Device 57c3
07:00.5 Processing accelerators: Intel Corporation Device 57c3
07:00.6 Processing accelerators: Intel Corporation Device 57c3
07:00.7 Processing accelerators: Intel Corporation Device 57c3
```

```
07:01.0 Processing accelerators: Intel Corporation Device 57c3
07:01.1 Processing accelerators: Intel Corporation Device 57c3
07:01.2 Processing accelerators: Intel Corporation Device 57c3
07:01.3 Processing accelerators: Intel Corporation Device 57c3
07:01.4 Processing accelerators: Intel Corporation Device 57c3
07:01.5 Processing accelerators: Intel Corporation Device 57c3
07:01.6 Processing accelerators: Intel Corporation Device 57c3
07:01.7 Processing accelerators: Intel Corporation Device 57c3
07:02.0 Processing accelerators: Intel Corporation Device 57c3
0a:00.0 Processing accelerators: Intel Corporation Device 57c2

$ lspci | grep -i net
19:00.0 Ethernet controller: Broadcom Inc. and subsidiaries BCM57504 NetXtreme-E
 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet (rev 11)
19:00.1 Ethernet controller: Broadcom Inc. and subsidiaries BCM57504 NetXtreme-E
 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet (rev 11)
19:00.2 Ethernet controller: Broadcom Inc. and subsidiaries BCM57504 NetXtreme-E
 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet (rev 11)
19:00.3 Ethernet controller: Broadcom Inc. and subsidiaries BCM57504 NetXtreme-E
 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet (rev 11)
51:00.0 Ethernet controller: Intel Corporation Ethernet Controller E810-C for QSFP (rev
 02)
51:00.1 Ethernet controller: Intel Corporation Ethernet Controller E810-C for QSFP (rev
 02)
51:01.0 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual Function (rev
 02)
51:01.1 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual Function (rev
 02)
51:01.2 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual Function (rev
 02)
51:01.3 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual Function (rev
 02)
51:11.0 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual Function (rev
 02)
51:11.1 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual Function (rev
 02)
51:11.2 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual Function (rev
 02)
51:11.3 Ethernet controller: Intel Corporation Ethernet Adaptive Virtual Function (rev
 02)
```

The SR-IOV Device Plugin uses a configMap containing a JSON file to define which hardware resources Kubernetes should expose. This configuration is based on two core concepts: `selectors` (for hardware discovery) and `resources` (for Kubernetes exposure).

A `resource` is the named entity that pods consume (e.g. `rancher.io/intel_fec_5g`). Resources can be defined as one of two types:

- `accelerator`: Used for vRAN accelerator cards (like ACC100/vRAN Boost).
- `netdevice`: Used for standard network interfaces (NICs).

You define the target devices using `selectors` to filter the hardware on the node:

- `vendors`: `8086` (Intel)
- `devices`: `57c3` (FEC VF), `1889` (NIC VF)
- `drivers`: `vfio-pci`
- `pfNames`: `eno16595np0` (physical interface name)

For network cards, you can also select a subset of Virtual Functions (VFs) from a Physical Function:

- `pfNames`: `["eth1#1,2,3,4,5,6"]` or `[eth1#1-6]`

To allow pods to request the devices, each resource must have a name, which is composed of a prefix and a name:

- `resourceName`: `pci_sriov_net_bh_dpdk`
- `resourcePrefix`: `rancher.io`

Pods would then request the combined resource name: `rancher.io/pci_sriov_net_bh_dpdk` .

Note

This document does not list all possible selectors. Different resource types use different sets of selectors. For comprehensive details, refer to the SR-IOV Network Device Plugin repository (https://github.com/k8snetworkplumbingwg/sriov-network-device-plugin).

The ConfigMap below is an example that creates three resources: one for the vRAN Accelerator card (FEC) and two for two different NIC ports.

For FEC card, you must first retrieve the device ID and VFIO token. Follow the instructions in *Chapter 42, vRAN Acceleration (`Intel ACC100/VRB1/VRB2`)* chapter for prerequisites.

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: sriovdp-config
  namespace: kube-system
data:
  config.json: |
    {
        "resourceList": [
            {
             "resourcePrefix": "rancher.io",
                "resourceName": "intel_fec_5g",
                "deviceType": "accelerator",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["57c3"]
                },
          "additionalInfo": {
             "*": {
               "VFIO_TOKEN": "00112233-4455-6677-8899-aabbccddeeff"
             }
            }
            },
            {
             "resourcePrefix": "rancher.io",
                "resourceName": "intel_sriov_odu",
                "deviceType": "netdevice",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["1889"],
                    "drivers": ["vfio-pci"],
                    "pfNames": ["eno16595np0"]
                }
            },
            {
             "resourcePrefix": "rancher.io",
                "resourceName": "intel_sriov_oru",
                "deviceType": "netdevice",
                "selectors": {
                    "vendors": ["8086"],
                    "devices": ["1889"],
                    "drivers": ["vfio-pci"],
                    "pfNames": ["eno16605np1"]
                }
            }
        ]
    }
```

- Prepare the `daemonset` file to deploy the device plugin.

The device plugin supports several architectures (`arm`, `amd`, `ppc64le`), so the same file can be used for different architectures by deploying several `daemonset` for each architecture.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sriov-device-plugin
  namespace: kube-system
---
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: kube-sriov-device-plugin-amd64
  namespace: kube-system
  labels:
    tier: node
    app: sriovdp
spec:
  selector:
    matchLabels:
      name: sriov-device-plugin
  template:
    metadata:
      labels:
        name: sriov-device-plugin
        tier: node
        app: sriovdp
    spec:
      hostNetwork: true
      nodeSelector:
        kubernetes.io/arch: amd64
      tolerations:
      - key: node-role.kubernetes.io/master
        operator: Exists
        effect: NoSchedule
      serviceAccountName: sriov-device-plugin
      containers:
      - name: kube-sriovdp
        image: registry.suse.com/edge/3.7/sriov-network-device-plugin:v3.10.0
        imagePullPolicy: IfNotPresent
        args:
        - --log-dir=sriovdp
        - --log-level=10
        securityContext:
          privileged: true
```

```
      resources:
        requests:
          cpu: "250m"
          memory: "40Mi"
        limits:
          cpu: 1
          memory: "200Mi"
      volumeMounts:
      - name: devicesock
        mountPath: /var/lib/kubelet/
        readOnly: false
      - name: log
        mountPath: /var/log
      - name: config-volume
        mountPath: /etc/pcidp
      - name: device-info
        mountPath: /var/run/k8s.cni.cncf.io/devinfo/dp
    volumes:
      - name: devicesock
        hostPath:
          path: /var/lib/kubelet/
      - name: log
        hostPath:
          path: /var/log
      - name: device-info
        hostPath:
          path: /var/run/k8s.cni.cncf.io/devinfo/dp
          type: DirectoryOrCreate
      - name: config-volume
        configMap:
          name: sriovdp-config
          items:
          - key: config.json
            path: config.json
```

- After applying the configMap and the `daemonset`, the device plugin will be deployed and the interfaces will be discovered and available for the pods.

```
$ kubectl get pods -n kube-system | grep sriov
kube-system  kube-sriov-device-plugin-amd64-twjfl  1/1  Running  0  2m
```

- Verify all nodes if interfaces were discovered and became available for the pods:

```
$ kubectl get nodes -o json | jq '.items[] | {name: .metadata.name,
 allocatable: .status.allocatable}'
{
  "name": "node1.suse.edge.com",
```

```
  "allocatable": {
   "cpu": "64",
   "ephemeral-storage": "256196109726",
   "hugepages-1Gi": "40Gi",
   "hugepages-2Mi": "0",
   "rancher.io/intel_fec_5g": "16",
   "rancher.io/intel_sriov_odu": "4",
   "rancher.io/intel_sriov_oru": "4",
   "memory": "221396384Ki",
   "pods": "110"
  }
}
```

- The resourceName for `FEC` accelerator is `rancher.io/intel_fec_5g` and 16 VFs are available for use.
- The resourceName for NIC cards are `rancher.io/intel_sriov_odu` and `rancher.io/intel_sriov_oru`. Each resource provides 4 VFs.

Important

If no interfaces are detected as allocatable resources in the kubernetes nodes, it is essential to resolve this issue. One common cause is ill-formed configMap spec, so better review the configMap and its selectors.

40.2 Option 2 (Recommended): SR-IOV Network Operator

- Get Helm if not present:

```
$ curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```

- Install SR-IOV Network Operator on `sriov-network-operator` namespace:

```
helm install sriov-crd oci://registry.suse.com/edge/charts/sriov-crd -n sriov-network-
operator
helm install sriov-network-operator oci://registry.suse.com/edge/charts/sriov-network-
operator -n sriov-network-operator
```

- Check the deployed CRDs and pods:

```
$ kubectl get crd
$ kubectl -n sriov-network-operator get pods
```

- Check if SR-IOV label is applied to the nodes.

With all resources running, the label appears automatically in your node:

```
$ kubectl get nodes -oyaml | grep feature.node.kubernetes.io/network-sriov.capable

feature.node.kubernetes.io/network-sriov.capable: "true"
```

- Review the `daemonset` to see the new `sriov-network-config-daemon` and `sriov-rancher-nfd-worker` as active and ready:

```
$ kubectl get daemonset -n sriov-network-operator
NAMESPACE             NAME                            DESIRED   CURRENT   READY   UP-TO-
DATE   AVAILABLE   NODE SELECTOR                                                AGE
sriov-network-operator   sriov-network-config-daemon     1         1         1       1
         1           feature.node.kubernetes.io/network-sriov.capable=true   45m
sriov-network-operator   sriov-rancher-nfd-worker        1         1         1       1
         1           <none>                                                  45m
```

In a few minutes, the nodes will be detected and fully configured with `SR-IOV` capabilities. The update can sometimes take up to 10 minutes:

```
$ kubectl get sriovnetworknodestates -A
NAMESPACE             NAME     AGE
sriov-network-operator   xr11-2   83s
```

- Check if the interfaces were detected.

The interfaces discovered should be the PCI address of the network device. Check this information with the `lspci` command in the host.

```
$ kubectl get sriovnetworknodestates -n sriov-network-operator -oyaml
apiVersion: v1
items:
- apiVersion: sriovnetwork.openshift.io/v1
  kind: SriovNetworkNodeState
  metadata:
    creationTimestamp: "2023-06-07T09:52:37Z"
```

```
    generation: 1
    name: xr11-2
    namespace: sriov-network-operator
    ownerReferences:
    - apiVersion: sriovnetwork.openshift.io/v1
      blockOwnerDeletion: true
      controller: true
      kind: SriovNetworkNodePolicy
      name: default
      uid: 80b72499-e26b-4072-a75c-f9a6218ec357
    resourceVersion: "356603"
    uid: e1f1654b-92b3-44d9-9f87-2571792cc1ad
  spec:
    dpConfigVersion: "356507"
  status:
    interfaces:
    - deviceID: "1592"
      driver: ice
      eSwitchMode: legacy
      linkType: ETH
      mac: 40:a6:b7:9b:35:f0
      mtu: 1500
      name: eno16595np0
      pciAddress: "0000:51:00.0"
      totalvfs: 128
      vendor: "8086"
    - deviceID: "1592"
      driver: ice
      eSwitchMode: legacy
      linkType: ETH
      mac: 40:a6:b7:9b:35:f1
      mtu: 1500
      name: eno16605np1
      pciAddress: "0000:51:00.1"
      totalvfs: 128
      vendor: "8086"
    syncStatus: Succeeded
kind: List
metadata:
  resourceVersion: ""
```

Note

If your interface is not detected here, ensure that it is present in the next config map:

```
$ kubectl get cm supported-nic-ids -oyaml -n sriov-network-operator
```

If your device is not listed, edit the config map by adding the right values to be discovered. Then restart the `sriov-network-config-daemon` pods on each node for update to take effect.

- Create the `SriovNetworkNodePolicy` to configure the `VFs`

This policy creates the resource `intelnicsDpdk` for pod consumption. It also binds `vfio-pci` driver to the provided PCI device and creates 8 VFs with an MTU size of 1500:

Note

The `resourceName` field must not contain any special characters and must be unique across the cluster. The example uses the `deviceType: vfio-pci` because DPDK is used in combination with SR-IOV. If you don't use DPDK, configure `deviceType: netdevice` (default value).

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetworkNodePolicy
metadata:
  name: policy-dpdk
  namespace: sriov-network-operator
spec:
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  resourceName: intelnicsDpdk
  deviceType: vfio-pci
  numVfs: 8
  mtu: 1500
  nicSelector:
    deviceID: "1592"
    vendor: "8086"
    rootDevices:
    - 0000:51:00.0
```

- Validate configurations on all nodes:

With the predefined resourcePrefix `rancher.io`, a resource `rancher.io/intelnicsDpdk` with 8 VFs should be discovered.

```
$ kubectl get nodes -o jsonpath='{"items": [ { "name": @.metadata.name, "allocatable":
 @.status.allocatable } ]}' | jq
{
  "name": "node1.suse.edge.com",
```

```
  "allocatable": {
   "cpu": "64",
   "ephemeral-storage": "256196109726",
   "hugepages-1Gi": "60Gi",
   "hugepages-2Mi": "0",
   "rancher.io/intel_fec_5g": "16",
   "memory": "200424836Ki",
   "pods": "110",
   "rancher.io/intelnicsDpdk": "8"
  }
}
```

- (Optional) Create the `sriovnetwork`

This step is optional and only required for custom network definitions. Specify the `resourceName` to bind to the previously created node policy.

If the `networkNamespace` is set, the network is exposed to pods in that namespace. Otherwise, the network becomes available in the Network Operator's installation namespace.

```
apiVersion: sriovnetwork.openshift.io/v1
kind: SriovNetwork
metadata:
  name: network-dpdk
  namespace: sriov-network-operator  # where SRIOV Operator is installed
spec:
  ipam: |
    {
      "type": "host-local",
      "subnet": "192.168.0.0/24",
      "rangeStart": "192.168.0.20",
      "rangeEnd": "192.168.0.60",
      "routes": [{
        "dst": "0.0.0.0/0"
      }],
      "gateway": "192.168.0.1"
    }
  vlan: 500
  resourceName: intelnicsDpdk
  networkNamespace: default   # where workloads are deployed
```

- If the update is successful, a NetworkAttachmentDefinition (NAD) is created in target cluster.

```
$ kubectl get net-attach-def -A -oyaml

apiVersion: v1
```

```
items:
- apiVersion: k8s.cni.cncf.io/v1
  kind: NetworkAttachmentDefinition
  metadata:
    annotations:
      k8s.v1.cni.cncf.io/resourceName: rancher.io/intelnicsDpdk
    creationTimestamp: "2023-06-08T11:22:27Z"
    generation: 1
    name: network-dpdk
    namespace: default
    resourceVersion: "13124"
    uid: df7c89f5-177c-4f30-ae72-7aef3294fb15
  spec:
    config: '{ "cniVersion":"0.4.0", "name":"network-
dpdk","type":"sriov","vlan":500,"vlanQoS":0,"ipam":{"type":"host-
local","subnet":"192.168.0.0/24","rangeStart":"192.168.0.10","rangeEnd":"192.168.0.60","routes":
[{"dst":"0.0.0.0/0"}],"gateway":"192.168.0.1"}
      }'
kind: List
metadata:
  resourceVersion: ""
```

The workload pods could use the resourceName `rancher.io/intelnicsDpdk` to use the VFs of the network interface.

41 DPDK

DPDK (Data Plane Development Kit) is a set of libraries and drivers for fast packet processing. It is used to accelerate packet processing workloads running on a wide variety of CPU architectures. The DPDK includes data plane libraries and optimized network interface controller (NIC) drivers for the following:

1. A queue manager implements lockless queues.
2. A buffer manager pre-allocates fixed size buffers.
3. A memory manager allocates pools of objects in memory and uses a ring to store free objects; ensures that objects are spread equally on all DRAM channels.
4. Poll mode drivers (PMD) are designed to work without asynchronous notifications, reducing overhead.
5. A packet framework as a set of libraries that are helpers to develop packet processing.

The following steps will show how to enable DPDK and how to create VFs from the NICs to be used by the DPDK interfaces:

- Install the DPDK package:

```
$ transactional-update pkg install dpdk dpdk-tools libdpdk-25
$ reboot
```

- Kernel parameters:

To use DPDK, employ some drivers to enable certain parameters in the kernel:

parameter	value	description
iommu	pt	This option enables the use of the vfio driver for the DPDK interfaces.
intel_iommu or amd_iommu	on	This option enables the use of vfio for VFs.

To enable the parameters, add them to the /etc/default/grub file:

```
GRUB_CMDLINE_LINUX="BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt
 root=UUID=77b713de-5cc7-4d4c-8fc6-f5eca0a43cf9 skew_tick=1 rd.timeout=60
```

```
rd.retry=45 console=ttyS1,115200 console=tty0 default_hugepagesz=1G hugepagesz=1G
hugepages=40 hugepagesz=2M hugepages=0 ignition.platform.id=openstack
net.ifnames=1 intel_iommu=on iommu=pt irqaffinity=0,31,32,63
isolcpus=domain,nohz,managed_irq,1-30,33-62 nohz_full=1-30,33-62 nohz=on
mce=off nosoftlockup nowatchdog nmi_watchdog=0 quiet rcu_nocb_poll
rcu_nocbs=1-30,33-62 rcupdate.rcu_cpu_stall_suppress=1 rcupdate.rcu_expedited=1
rcupdate.rcu_normal_after_boot=1 rcupdate.rcu_task_stall_timeout=0
rcutree.kthread_prio=99 security=selinux selinux=1 idle=poll"
```

Update the GRUB configuration and reboot the system to apply the changes:

```
$ transactional-update grub.cfg
$ reboot
```

- Load `vfio-pci` kernel module and enable `SR-IOV` on the NICs. First argument indicates `vfio-pci` driver to support SR-IOV, and second argument prevents the PCI device from entering a low-power state when it's idle:

```
$ modprobe vfio-pci enable_sriov=1 disable_idle_d3=1
```

- Create some virtual functions (`VFs`) from the `NICs`.

To create for `VFs`, for example, for two different `NICs`, the following commands are required:

```
$ echo 4 > /sys/bus/pci/devices/0000:51:00.0/sriov_numvfs
$ echo 4 > /sys/bus/pci/devices/0000:51:00.1/sriov_numvfs
```

- Bind the new VFs with the `vfio-pci` driver:

```
$ dpdk-devbind.py -b vfio-pci 0000:51:01.0 0000:51:01.1 0000:51:01.2 0000:51:01.3 \
                              0000:51:11.0 0000:51:11.1 0000:51:11.2 0000:51:11.3
```

- Review the configuration is correctly applied:

```
$ dpdk-devbind.py -s

Network devices using DPDK-compatible driver
============================================
0000:51:01.0 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci unused=iavf,igb_uio
0000:51:01.1 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci unused=iavf,igb_uio
0000:51:01.2 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci unused=iavf,igb_uio
0000:51:01.3 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci unused=iavf,igb_uio
0000:51:01.0 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci unused=iavf,igb_uio
0000:51:11.1 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci unused=iavf,igb_uio
0000:51:21.2 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci unused=iavf,igb_uio
```

```
0000:51:31.3 'Ethernet Adaptive Virtual Function 1889' drv=vfio-pci unused=iavf,igb_uio

Network devices using kernel driver
====================================
0000:19:00.0 'BCM57504 NetXtreme-E 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet 1751'
 if=eno8303np0 drv=bnxt_en unused=igb_uio,vfio-pci *Active*
0000:19:00.1 'BCM57504 NetXtreme-E 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet 1751'
 if=eno8403np1 drv=bnxt_en unused=igb_uio,vfio-pci
0000:19:00.2 'BCM57504 NetXtreme-E 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet 1751'
 if=eno8503np2 drv=bnxt_en unused=igb_uio,vfio-pci
0000:19:00.3 'BCM57504 NetXtreme-E 10Gb/25Gb/40Gb/50Gb/100Gb/200Gb Ethernet 1751'
 if=eno8603np3 drv=bnxt_en unused=igb_uio,vfio-pci
0000:51:00.0 'Ethernet Controller E810-C for QSFP 1592' if=eth13 drv=ice
 unused=igb_uio,vfio-pci
0000:51:00.1 'Ethernet Controller E810-C for QSFP 1592' if=rename8 drv=ice
 unused=igb_uio,vfio-pci
```

42 vRAN Acceleration (Intel ACC100/VRB1/VRB2)

As communications service providers move from 4G to 5G networks, many are adopting virtualized radio access network (vRAN) architectures for higher channel capacity and easier deployment of edge-based services and applications. vRAN solutions are ideally located to deliver low-latency services with the flexibility to increase or decrease capacity based on the volume of real-time traffic and demand on the network.

One of the most compute-intensive 4G and 5G workloads is RAN layer 1 (`L1`) `FEC`, which resolves data transmission errors over unreliable or noisy communication channels. `FEC` technology detects and corrects a limited number of errors in 4G or 5G data, eliminating the need for retransmission. Since the `FEC` acceleration transaction does not contain cell state information, it can be easily virtualized, enabling pooling benefits and easy cell migration.

Historically, Intel provided the ACC100 vRAN Accelerator card to rapidly execute Layer 1 FEC algorithms, freeing up host processing power for the main CPU. Intel has since integrated this technology directly into newer CPUs, starting with Sapphire Rapids, under the name `Intel vRAN Boost` (VRB). Intel vRAN Boost acts as an offload accelerator on the CPU itself, eliminating the need for a separate hardware card. This section details configuration of SUSE Telco Cloud for workloads to leverage ACC100 or Intel vRAN Boost.

42.1 Kernel parameters

To enable the `vRAN` acceleration, we need to enable the following kernel parameters (if not present yet):

parameter	value	description
iommu	pt	This option enables the use of vfio for the DPDK interfaces.
intel_iommu or amd_iommu	on	This option enables the use of vfio for VFs.

Modify the GRUB file `/etc/default/grub` to add them to the kernel command line:

```
GRUB_CMDLINE_LINUX="BOOT_IMAGE=/boot/vmlinuz-6.4.0-9-rt
 root=UUID=77b713de-5cc7-4d4c-8fc6-f5eca0a43cf9 skew_tick=1 rd.timeout=60
 rd.retry=45 console=ttyS1,115200 console=tty0 default_hugepagesz=1G hugepagesz=1G
 hugepages=40 hugepagesz=2M hugepages=0 ignition.platform.id=openstack
```

```
net.ifnames=1 intel_iommu=on iommu=pt irqaffinity=0,31,32,63
isolcpus=domain,nohz,managed_irq,1-30,33-62 nohz_full=1-30,33-62 nohz=on
mce=off nosoftlockup nowatchdog nmi_watchdog=0 quiet rcu_nocb_poll
rcu_nocbs=1-30,33-62 rcupdate.rcu_cpu_stall_suppress=1 rcupdate.rcu_expedited=1
rcupdate.rcu_normal_after_boot=1 rcupdate.rcu_task_stall_timeout=0
rcutree.kthread_prio=99 security=selinux selinux=1 idle=poll"
```

Update the GRUB configuration and reboot the system to apply the changes:

```
$ transactional-update grub.cfg
$ reboot
```

To verify that the parameters are applied after the reboot, check the command line:

```
$ cat /proc/cmdline
```

42.2 Configure SR-IOV on FEC Accelerators

- Load `vfio-pci` kernel module to enable vRAN acceleration. First argument indicates `vfio-pci` module to support SR-IOV, and second argument prevents the PCI device from entering a low-power state when it's idle.

```
$ modprobe vfio-pci enable_sriov=1 disable_idle_d3=1
```

- Retrieve the PCI device address of FEC accelerator:

```
$ lspci -DPPnn | grep -i acc
0000:07:00.0 Processing accelerators [1200]: Intel Corporation Device [8086:57c2]
0000:0a:00.0 Processing accelerators [1200]: Intel Corporation Device [8086:57c2]
```

- Bind the physical interface (`PF`) with `vfio-pci` driver:

```
$ dpdk-devbind.py -b vfio-pci 0000:07:00.0
```

- Create the virtual functions (`VFs`) from the physical interface (`PF`).

Check the maximum `VF` capacity of the accelerator card. Next, configure the card to expose the desired number of VFs, not exceeding the maximum number. In this example, we configure the card for its full capacity of 16 VFs:

```
$ cat /sys/bus/pci/devices/0000:07:00.0/sriov_totalvfs
```

```
64
$ echo 16 > /sys/bus/pci/devices/0000:07:00.0/sriov_numvfs
```

- Configure the accelerator card and its virtual functions with a 4G or 5G profile, selecting the right `vRAN Boost` device type depending on the exact Intel processor generation: `VRB1` (formerly known as ACC200) if Sapphire Rapids Edge Enhanced (SPR-EE), `VRB2` for Granite Rapids-D (GNR-D). A unique VF token (UUID) must be provided (the workload will consume this VF token to utilize the card's FEC acceleration capabilities).

Warning

Deprecation from `pf-bb-config`: new rpm release supporting VRB2 device type is now installing the configuration example files in `/usr/share/pf-bb-config/examples` location; the old `/opt/pf-bb-config/` path is still supported but to be removed in future releases. Please, update your scripts and documentation to reflect the new path (as done in the example below).

```
$ pf_bb_config VRB2 -c /usr/share/pf-bb-config/examples/vrb2/vrb2_config_vf_5g.cfg
 -f /usr/share/pf-bb-config/examples/vrb2/srs_fft_windows_coefficient.bin -v
 00112233-4455-6677-8899-aabbccddeeff -p 0000:07:00.0
== pf_bb_config Version 25.11 ==
VRB2 PF [0000:07:00.0] configuration complete!
Log file = /var/log/pf_bb_cfg_0000:07:00.0.log
```

- Verify the new VFs created from the FEC PF. Note that the VFs got `0d5d` device ID. This information is required in next step to expose these VFs as Kubernetes resource:

```
$ # dpdk-devbind.py -s | grep -A18 Baseband
Baseband devices using DPDK-compatible driver
============================================
0000:07:00.0 'Device 57c2' numa_node=0 drv=vfio-pci unused=
0000:07:00.1 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:00.2 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:00.3 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:00.4 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:00.5 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:00.6 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:00.7 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:01.0 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:01.1 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:01.2 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:01.3 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:01.4 'Device 57c3' numa_node=0 drv=vfio-pci unused=
```

```
0000:07:01.5 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:01.6 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:01.7 'Device 57c3' numa_node=0 drv=vfio-pci unused=
0000:07:02.0 'Device 57c3' numa_node=0 drv=vfio-pci unused=
--
Other Baseband devices
======================
0000:0a:00.0 'Device 57c2' numa_node=0 unused=vfio-pci

Other Crypto devices
====================
0000:01:00.0 '420xx Series QAT 4946' numa_node=0 unused=vfio-pci
0000:03:00.0 'Device 2714' numa_node=0 unused=vfio-pci

DMA devices using kernel driver
===============================
0000:00:01.0 'Device 11fb' numa_node=0 drv=idxd unused=vfio-pci

Other Eventdev devices
======================
0000:03:00.0 'Device 2714' numa_node=0 unused=vfio-pci

No 'Mempool' devices detected
=============================
```

42.3 Configure Kubernetes for FEC Acceleration

The final step is exposing the VFs to Kubernetes with help of SR-IOV device plugin. Create a ConfigMap using the VFs' deviceID gathered from previous step, and install SR-IOV device plugin. Once Kubernetes nodes display the FEC VFs as `Allocatable` resources, the cluster is ready for the workloads to enjoy the FEC acceleration.

Follow the steps of Option 1 in *Chapter 40, SR-IOV* chapter. SRIOV Network Operator isn't applicable for FEC Accelerator, so Option 2 is not applicable.

43 Huge pages

When a process uses `RAM`, the `CPU` marks the exact portion utilized by that process. To achieve efficiency, the `CPU` allocates `RAM` in blocks known as `pages`; the default `page` size on many platforms is `4 KB`. Pages can be swapped to disk (among other operations).

Since the process address space is virtual, the `CPU` and the operating system must keep track of which `pages` belong to each process and where each one is stored. The greater the number of `pages`, the longer it takes to look up memory mappings. For example, a process using `1 GB` of memory involves 262,144 entries to check (1 GB / 4 KB). If each `page table entry` occupies 8 bytes, this results in a total of `2 MB` (262,144 * 8) of data to be consulted.

Most current `CPU` architectures allow for the definition of blocks of `pages` each with a size larger than the default `4 KB` value (with `1 GB` and `2 MB` being typical supported values) known as `huge pages`, which reduces the number of entries the `CPU/Operating System` must consult.

- Kernel parameters

To enable `huge pages`, we must add the following kernel parameters. In this example, we configure 40 pages of `1 GB` size each and 0 pages of `2 MB` size each, though the huge page sizes to support and the exact amount of pages for each should be adjusted according to your application's memory requirements:

parameter	value	description
hugepagesz	1G	This option allows to set the size of "huge pages" to 1 GB
hugepages	40	This is the number of "huge pages" requested to allocate at boot for the size defined just before (1 GB)
hugepagesz	2M	This option allows to set the size of "huge pages" to 2 MB
hugepages	0	This is the number of "huge pages" requested to allocate at boot for the size defined just before (2 MB)

parameter	value	description
default_hugepagesz	1G	This is the default value for the "huge pages" size

Modify the GRUB file `/etc/default/grub` to add these parameters in `GRUB_CMDLINE_LINUX`:

```
default_hugepagesz=1G hugepagesz=1G hugepages=40 hugepagesz=2M hugepages=0
```

Update the GRUB configuration and reboot the system to apply the changes:

```
$ transactional-update grub.cfg
$ reboot
```

To validate that the parameters are applied after the reboot, you can check the command line:

```
$ cat /proc/cmdline
```

- Using huge pages

To use the huge pages, we need to mount them:

```
$ mkdir -p /hugepages
$ mount -t hugetlbfs nodev /hugepages
```

Deploy a Kubernetes workload, creating the resources and the volumes:

```
...
 resources:
   requests:
     memory: "24Gi"
     hugepages-1Gi: 16Gi
     intel.com/intel_sriov_oru: '4'
   limits:
     memory: "24Gi"
     hugepages-1Gi: 16Gi
     intel.com/intel_sriov_oru: '4'
...
```

```
...
volumeMounts:
  - name: hugepage
    mountPath: /hugepages
...
volumes:
```

```
  - name: hugepage
    emptyDir:
      medium: HugePages
...
```

44 NUMA-aware scheduling

Non-Uniform Memory Access or Non-Uniform Memory Architecture (NUMA) is a physical memory design used in SMP (multiprocessors) architecture, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors.

44.1 Identifying NUMA nodes

To identify the NUMA nodes, on your system use the following command:

```
$ lscpu | grep NUMA
NUMA node(s):                        1
NUMA node0 CPU(s):                   0-63
```

Note

For this example, we have only one NUMA node showing 64 CPUs.

NUMA needs to be enabled in the BIOS. If dmesg does not have records of NUMA initialization during the bootup, then NUMA-related messages in the kernel ring buffer might have been overwritten.

45 Metal LB

`MetalLB` is a load-balancer implementation for bare-metal Kubernetes clusters, using standard routing protocols like `L2` and `BGP` as advertisement protocols. It is a network load balancer that can be used to expose services in a Kubernetes cluster to the outside world due to the need to use Kubernetes Services type `LoadBalancer` with bare-metal.

To enable `MetalLB` in the `RKE2` cluster, the following steps are required:

- Install `MetalLB` using the following command:

```
$ kubectl apply <<EOF -f
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: metallb
  namespace: kube-system
spec:
  chart: oci://registry.suse.com/edge/charts/metallb
  targetNamespace: metallb-system
  version: 307.0.3+up0.16.1
  createNamespace: true
---
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: endpoint-copier-operator
  namespace: kube-system
spec:
  chart: oci://registry.suse.com/edge/charts/endpoint-copier-operator
  targetNamespace: endpoint-copier-operator
  version: 307.0.1+up0.3.0
  createNamespace: true
EOF
```

- Create the `IpAddressPool` and the `L2advertisement` configuration:

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: kubernetes-vip-ip-pool
  namespace: metallb-system
spec:
  addresses:
    - 10.168.200.98/32
```

```
  serviceAllocation:
    priority: 100
    namespaces:
      - default
---
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ip-pool-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
    - kubernetes-vip-ip-pool
```

- Create the endpoint service to expose the `VIP`:

```
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-vip
  namespace: default
spec:
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: rke2-api
    port: 9345
    protocol: TCP
    targetPort: 9345
  - name: k8s-api
    port: 6443
    protocol: TCP
    targetPort: 6443
  sessionAffinity: None
  type: LoadBalancer
```

- Check the `VIP` is created and the `MetalLB` pods are running:

```
$ kubectl get svc -n default
$ kubectl get pods -n default
```

46 Private registry configuration

`Containerd` can be configured to connect to private registries and use them to pull private images on each node.

Upon startup, `RKE2` checks if a `registries.yaml` file exists at `/etc/rancher/rke2/` and instructs `containerd` to use any registries defined in the file. If you wish to use a private registry, create this file as root on each node that will use the registry.

To add the private registry, create the file `/etc/rancher/rke2/registries.yaml` with the following content:

```
mirrors:
  docker.io:
    endpoint:
      - "https://registry.example.com:5000"
configs:
  "registry.example.com:5000":
    auth:
      username: xxxxxx # this is the registry username
      password: xxxxxx # this is the registry password
    tls:
      cert_file:            # path to the cert file used to authenticate to the registry
      key_file:             # path to the key file for the certificate used to
 authenticate to the registry
      ca_file:              # path to the ca file used to verify the registry's
 certificate
      insecure_skip_verify: # may be set to true to skip verifying the registry's
 certificate
```

or without authentication:

```
mirrors:
  docker.io:
    endpoint:
      - "https://registry.example.com:5000"
configs:
  "registry.example.com:5000":
    tls:
      cert_file:            # path to the cert file used to authenticate to the registry
      key_file:             # path to the key file for the certificate used to
 authenticate to the registry
      ca_file:              # path to the ca file used to verify the registry's
 certificate
      insecure_skip_verify: # may be set to true to skip verifying the registry's
 certificate
```

For the registry changes to take effect, you need to either configure this file before starting RKE2 on the node, or restart RKE2 on each configured node.

Note

For more information about this, please check containerd registry configuration rke2 (https://documentation.suse.com/cloudnative/rke2/latest/en/install/containerd_registry_configuration.html#_registries_configuration_file).

47 Precision Time Protocol

47.1 Introduction

Precision Time Protocol (PTP), defined by the IEEE 1588 standard, distributes highly accurate time across a packet network. When the network interface cards (NICs) timestamp PTP packets in hardware, it reaches sub-microsecond accuracy, well beyond what NTP can offer. In telco environments PTP is a core building block of 5G radio access networks (RAN), where the radio units require tight phase and time alignment to transmit correctly. SUSE Telco Cloud implements PTP through the `linuxptp` package and targets the telco profiles ITU-T G.8275.1 and ITU-T G.8275.2 described below.

A PTP domain is built from a small set of clock roles:

- **Grandmaster (GM)**: the primary source of time for the domain. It is normally locked to a Primary Reference Time Clock (PRTC), usually driven by a GNSS receiver, and transmits time towards the downstream nodes. The accuracy of the GM sets the quality ceiling for the whole domain.
- **Ordinary Clock (OC)**: an end node with a single PTP port. In a telco deployment it acts as a Time Receiver, disciplining its local clock from an upstream Grand Master or Boundary Clock. A compute node running a radio workload is a typical OC.
- **Boundary Clock (BC)**: a multi-port node that receives time on one port (Time Receiver) and regenerates it towards other nodes on its remaining ports (Time Transmitter). Inserting Boundary Clocks along the path shortens the chain between each OC and the reference, limiting the error the network would otherwise accumulate hop after hop.

The distinction between an Ordinary Clock and a Boundary Clock is central when designing the topology:

- An **Ordinary Clock** has a **single** PTP port and sits at the edge of the domain. It only consumes time to discipline its own clock and never forwards time to other nodes.
- A **Boundary Clock** has **multiple** PTP ports. It consumes time on its Time Receiver port and **regenerates** it on its Time Transmitter ports, terminating the upstream timing chain and starting a new one downstream. This regeneration is what prevents error from accumulating across a long chain of nodes.

The following hardware and signalling elements support these roles:

- **PTP Hardware Clock (PHC)**: the clock built into the NIC that PTP actually synchronizes. Aligning the operating system clock to the PHC is a separate step handled by a dedicated daemon.
- **Synchronous Ethernet (SyncE)**: physical-layer frequency synchronization. PTP carries phase and time inside packets, whereas SyncE recovers a stable frequency directly from the Ethernet signal. SyncE is an **optional enhancement**: PTP on its own already delivers phase and time, and SyncE is added only when a node needs stronger frequency stability and longer holdover should the PTP time source be temporarily lost.
- **Digital Phase-Locked Loop (DPLL)**: a hardware timing block that locks onto a reference signal and filters it to output a stable, clean clock. It is what disciplines the PHC from the available inputs and, on platforms with more than one NIC, keeps the different NIC clocks aligned.

On each node these functions are carried out by the `linuxptp` daemons: `ptp4l` runs the PTP protocol and disciplines the PHC, and `phc2sys` aligns the system clock to the PHC. On a Grandmaster, `ts2phc` additionally steers the PHC from the GNSS reference. Their roles and installation are covered in *Section 47.2, "Prerequisites"*.

The hardware clock itself is handled by the Linux kernel DPLL subsystem together with the network driver. The degree of timing control, and how timing is distributed within a multi-NIC node, depends on where the manageable DPLL sits:

- **NIC with an integrated DPLL**: the DPLL is embedded in the NIC and is configured through the Linux DPLL subsystem. It selects and locks to the chosen reference and disciplines the PHC, giving full control over frequency and holdover.
- **NIC without an integrated DPLL**: the NIC exposes no configurable DPLL. `ptp4l` disciplines the PHC directly, and the subsystem can at most report the lock status (read-only); advanced frequency handling then relies on an external timing source.
- **On-board DPLL shared across several NICs**: the DPLL is not embedded in a single NIC but sits on the platform, external to the NICs. It disciplines and keeps the PHCs of several NICs aligned from a common reference, so every NIC in the node shares the same time. This arrangement is used on multi-NIC platforms that distribute timing internally instead of relying on external cabling.

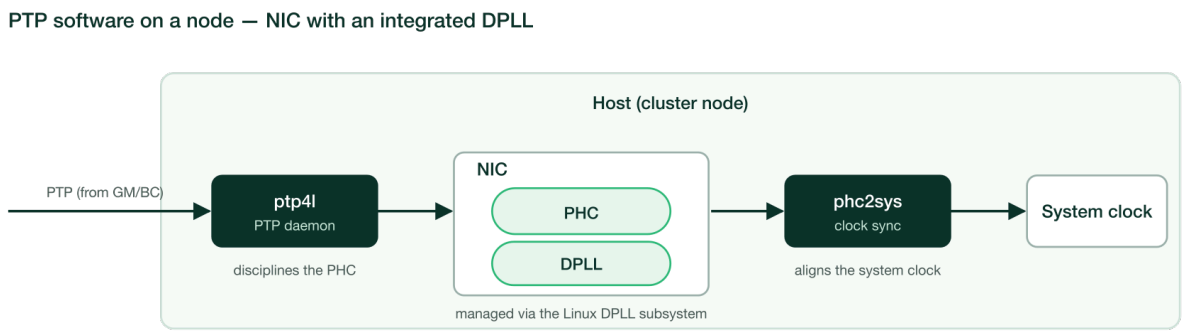


FIGURE 47.1: PTP SOFTWARE ON A NODE WHOSE NIC INTEGRATES A MANAGEABLE DPLL

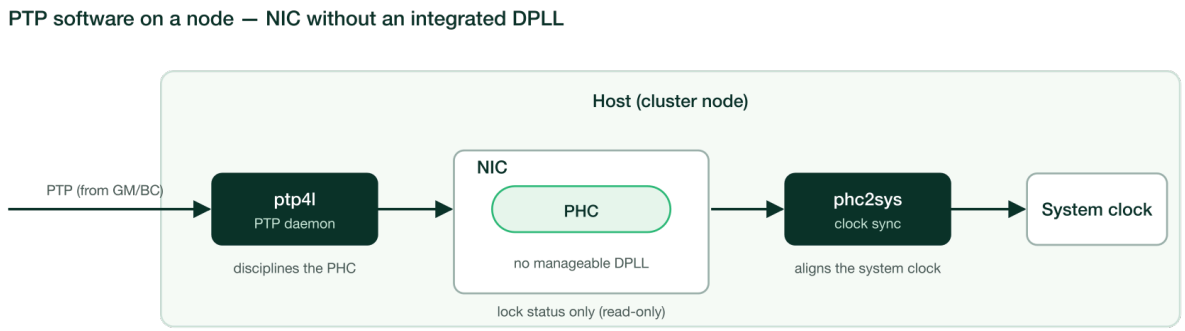


FIGURE 47.2: PTP SOFTWARE ON A NODE WHOSE NIC HAS NO MANAGEABLE DPLL

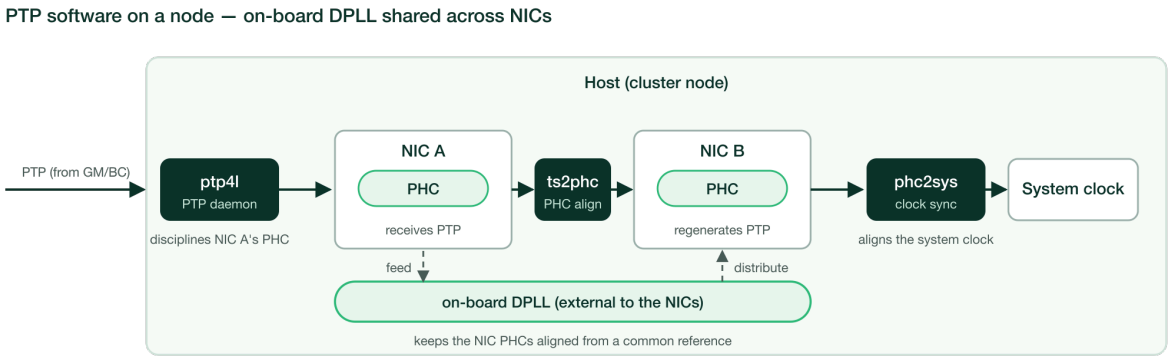


FIGURE 47.3: PTP SOFTWARE ON A NODE WITH AN ON-BOARD DPLL SHARED ACROSS NICS

Telco networks standardize on two ITU-T profiles, which differ mainly in the transport used and in how much of the underlying network must be PTP-aware:

- **ITU-T G.8275.1** (full timing support): runs directly over Ethernet (Layer 2) using multicast. Every node along the path must be PTP-aware and act as a Boundary Clock, which yields the highest accuracy. Unless stated otherwise, the scenarios in this guide use this profile.
- **ITU-T G.8275.2** (partial timing support): runs over IP (Layer 3) using unicast. Intermediate nodes are not required to support PTP, so timing can cross network segments that are not timing-aware, at the cost of lower accuracy.

The following figures show time distribution with PTP only and with SyncE added as an optional frequency enhancement. In both cases the Grandmaster derives its reference from a PRTC/GNSS source and the Boundary Clock regenerates timing towards the endpoints.

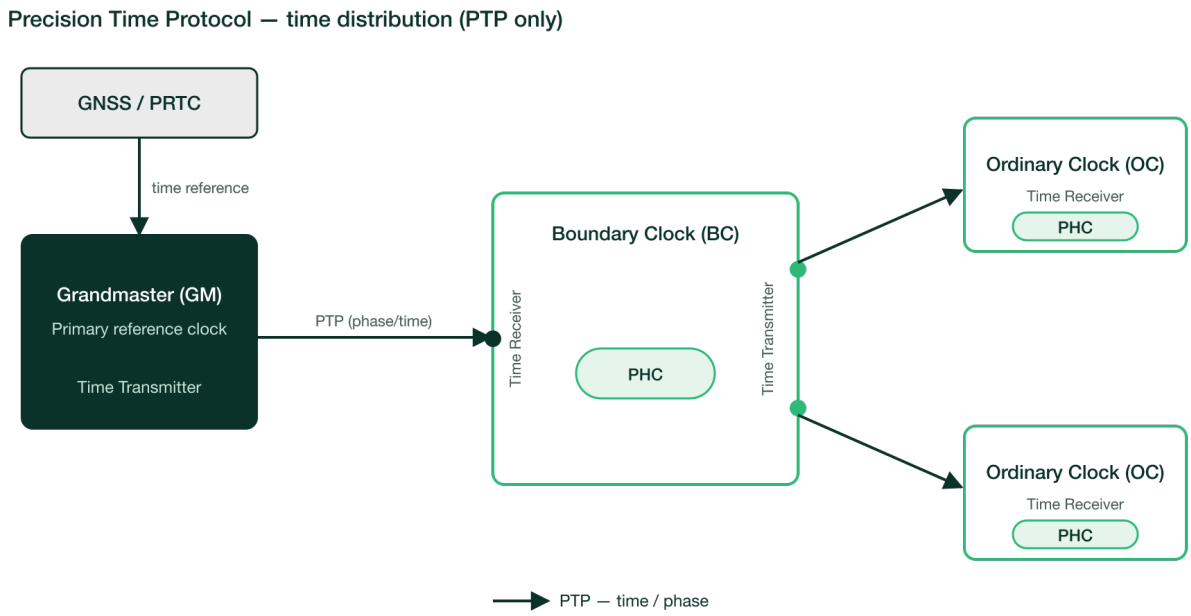


FIGURE 47.4: TIME DISTRIBUTION WITH PTP ONLY

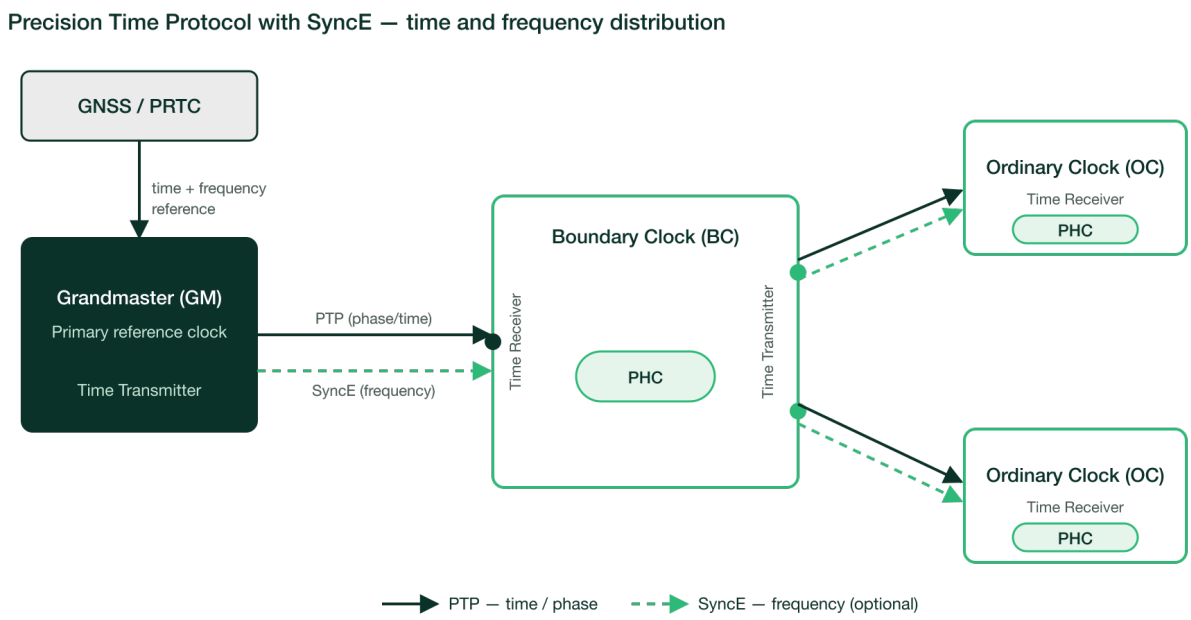


FIGURE 47.5: TIME AND FREQUENCY DISTRIBUTION WITH PTP AND SYNCE (OPTIONAL ENHANCEMENT)

The following scenarios are covered in this guide. Each scenario includes a link to its Cluster API (CAPI) automation.

- Scenario 1a - Ordinary Clock (PTP-only) (*Section 47.3, "Scenario 1a - Ordinary Clock (PTP-only)"*)
- Scenario 1b - Ordinary Clock (Hybrid mode: PTP & SyncE) (*Section 47.4, "Scenario 1b - Ordinary Clock (Hybrid mode: PTP & SyncE)"*)
- Scenario 2a - Boundary Clock (PTP-only) with Multiport Single NIC including propagation to Ordinary Clock server (*Section 47.5, "Scenario 2a - Boundary Clock (PTP-only) with Multiport Single NIC including propagation to Ordinary Clock server"*)
- Scenario 2b - Boundary Clock (Hybrid mode: PTP & SyncE) with Multiport Single NIC including propagation to Ordinary Clock server (*Section 47.6, "Scenario 2b - Boundary Clock (Hybrid mode: PTP & SyncE) with Multiport Single NIC including propagation to Ordinary Clock server"*)
- Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based) (*Section 47.7, "Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based)"*)
- Scenario 3b - Boundary Clock (Hybrid mode: PTP & SyncE) Multi-NIC with internal timing module (DPLL GNR-D based) (*Section 47.8, "Scenario 3b - Boundary Clock (Hybrid mode: PTP & SyncE) Multi-NIC with internal timing module (DPLL GNR-D based)"*)

Note

Only Scenario 1a (*Section 47.3, "Scenario 1a - Ordinary Clock (PTP-only)"*) shows the configuration for both telco profiles, ITU-T G.8275.1 and ITU-T G.8275.2. The remaining scenarios focus only on the ITU-T G.8275.1 case for brevity: the deployment is identical and only the `ptp4l` configuration file changes between the two profiles.

47.2 Prerequisites

This guide covers only telco-specific profiles, so hardware time-stamping and a PTP hardware clock (PHC) in the NIC are assumed. Telco-grade network adapters provide PTP support in hardware; you can confirm the capabilities of a given interface with `ethtool -T`:

```
# ethtool -T eno16495np0
Time stamping parameters for eno16495np0:
Capabilities:
        hardware-transmit
        software-transmit
        hardware-receive
        software-receive
        software-system-clock
        hardware-raw-clock
PTP Hardware Clock: 0
Hardware Transmit Timestamp Modes:
        off
        on
Hardware Receive Filter Modes:
        none
        all
```

Check for the `hardware-transmit`, `hardware-receive` and `hardware-raw-clock` capabilities and a `PTP Hardware Clock` index of 0 or higher (a value of `-1` means the interface has no PHC). Replace `eno16495np0` with the interface you intend to use for PTP.

PTP is provided by the `linuxptp` package. It ships three user-space programs — `ptp4l`, `phc2sys` and `ts2phc` — that run as long-running daemons managed by `systemd` (a dedicated service each: `pt-p4l.service`, `phc2sys.service`, and `ts2phc.service`). Collectively, they execute the protocol and keep the clocks synchronized:

- `ptp4l`: the PTP daemon. It runs the IEEE 1588 protocol on the configured interfaces, applies the Best Master Clock Algorithm (BMCA) to determine the role of each port, and disciplines the NIC's PHC. A single `ptp4l` instance can manage several ports, which is what allows a node to act as a Boundary Clock.
- `phc2sys`: synchronizes two clocks. It is typically used to align the system (operating system) clock to the PHC that `ptp4l` keeps synchronized, so that the system time also follows PTP.
- `ts2phc`: "time stamp to PHC". It disciplines the PHC from an external time signal, such as the PPS delivered by a GNSS receiver. It is used on a Grandmaster and to keep multiple PHCs aligned within the same node.

The recommended way to install PTP on a downstream cluster is to add the `linuxptp` package to the `packageList` of the Edge Image Builder (EIB) definition file, so that it is deployed automatically during provisioning. See the EIB documentation (*Section 5.3.6, "Configuring RPM packages"*) for more information on installing packages.

```
apiVersion: 1.3
image:
  imageType: RAW
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.2-Base-RT-GM.raw
  outputImageName: eibimage-slmicrort-telco.raw
operatingSystem:
  systemd:
    enable:
      - ptp4l
      - phc2sys
  packages:
    packageList:
      - linuxptp
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

Note

The `linuxptp` package does not enable `ptp4l`, `phc2sys`, or `ts2phc` by default. After deploying the configuration files, use the `systemd` section of the manifest to enable the required node daemons (see the scenarios below).

Finally, the PTP profile in use determines whether the network switches between nodes must be configured as PTP-aware devices:

- **ITU-T G.8275.1 (full timing support)**: runs directly over Ethernet (Layer 2) using multicast. Every node along the path must be PTP-aware and act as a Boundary Clock, which yields the highest accuracy; the intervening switches must therefore be configured for PTP. Unless stated otherwise, the scenarios in this guide use this profile.
- **ITU-T G.8275.2 (partial timing support)**: runs over IP (Layer 3) using unicast. Intermediate nodes are not required to support PTP, so timing can cross network segments that are not timing-aware and the switches do not need PTP configuration, at the cost of lower accuracy.

47.3 Scenario 1a - Ordinary Clock (PTP-only)

Explanation

This is the simplest PTP deployment: a single cluster node consumes time from an upstream source and disciplines its local clocks, without ever forwarding time to another node. The node runs as an Ordinary Clock (OC) with a single PTP port in the Time Receiver role.

On the OC node the timing is handled entirely by the `linuxptp` daemons, regardless of the profile in use. No DPLL configuration or `ts2phc` is required, because the node uses a single NIC and `ptp4l` disciplines its PHC directly:

- `ptp4l` runs the PTP protocol on the receiving interface. The Best Master Clock Algorithm (BMCA) selects the upstream node as the master, the local port transitions to the `SLAVE` state and `ptp4l` disciplines the NIC's PTP Hardware Clock (PHC). Setting `clientOnly 1` restricts the node to the Time Receiver role, so it can never be selected as a master.
- `phc2sys` aligns the system clock (`CLOCK_REALTIME`) to the PHC that `ptp4l` keeps synchronized, so the operating-system time also follows PTP.

The scenario can be implemented with either telco profile. They differ in transport and in what the intervening network must provide, and are documented below as two independent models:

- **Model 1 — ITU-T G.8275.1 (*Section 47.3, "Scenario 1a - Ordinary Clock (PTP-only)" (page 266)*)** (Layer 2 multicast): use it when every hop between the master and the OC is PTP-aware. It provides the highest accuracy and is the default for telco deployments.
- **Model 2 — ITU-T G.8275.2 (*Section 47.3, "Scenario 1a - Ordinary Clock (PTP-only)" (page 269)*)** (Layer 3 unicast): use it when the OC must reach the master across a network that is not PTP-aware, at the cost of lower accuracy.

> **Important**
>
> An Ordinary Clock runs a single PTP profile at a time. The two models are mutually exclusive: configure only one of them on a given node. Create only the configuration file of the chosen profile and reference only that file in `/etc/sysconfig/ptp4l`. Do not create both `/etc/ptp4l-G.8275.1.conf` and `/etc/ptp4l-G.8275.2.conf`, and never pass both files to the `OPTIONS=` line.

Model 1 — ITU-T G.8275.1 (Layer 2 multicast)

G.8275.1 runs directly over Ethernet and addresses its messages to the non-forwardable multicast MAC `01:80:C2:00:00:0E`, which standard bridges do not forward, so every device on the path must terminate and regenerate PTP. The components are:

- **Grandmaster (GM)**: the upstream reference for the PTP domain (domain `24` for G.8275.1), typically locked to a GNSS/PRTC source.
- **PTP-aware switch**: acts as a Boundary Clock. Its port towards the GM is a Time Receiver and its port towards the OC is a Time Transmitter. Both ports must have PTP enabled.
- **Ordinary Clock (OC)**: the cluster node, receiving time on its single PTP port.

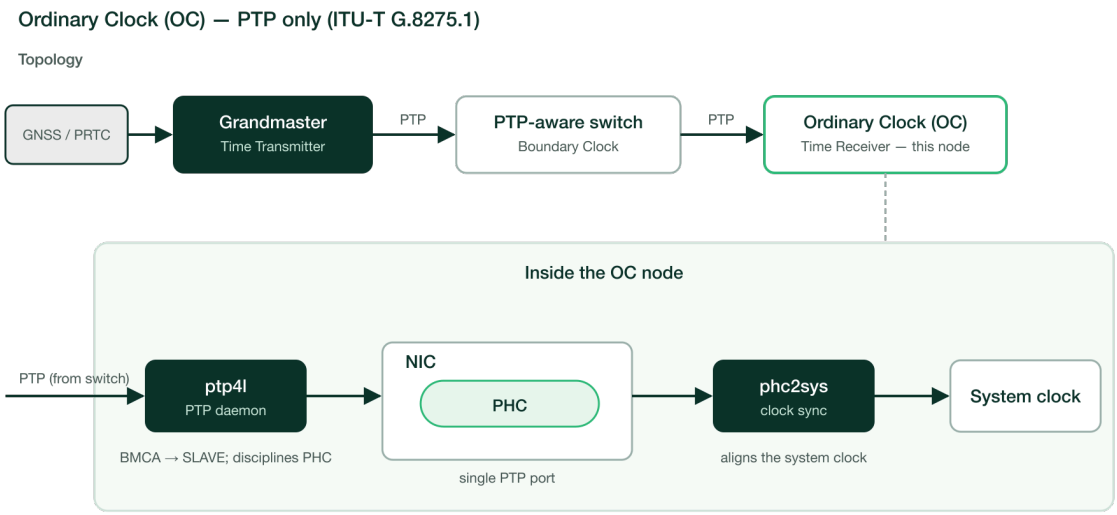


FIGURE 47.6: ORDINARY CLOCK (OC) — G.8275.1 (LAYER 2 MULTICAST)

Configure the node as follows:

1. Identify the PHC index `<N>` of the interface used for PTP (here `eno8403np1`); the value of the printed field `PTP Hardware Clock` (which may also appear as `Hardware timestamp provider index`) gives the `/dev/ptp<N>` device that `ptp4l` disciplines and that `phc2sys` reads as its source:

```
# ethtool -T eno8403np1
Time stamping parameters for eno8403np1:
Capabilities:
        hardware-transmit
        hardware-receive
        hardware-raw-clock
PTP Hardware Clock: 1
...
```

   In this example the PHC of `eno8403np1` is `/dev/ptp1`.

2. Create the `ptp4l` configuration file `/etc/ptp4l-G.8275.1.conf`. The interface is declared as a section (`[eno8403np1]`), so the daemon does not need the `-i` flag:

EXAMPLE 47.1: ORDINARY CLOCK CONFIGURATION — ITU-T G.8275.1

```
# Telecom G.8275.1 — Ordinary Clock (Time Receiver)
[global]
domainNumber                    24
priority2                       255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     128
```

```
G.8275.defaultDS.localPriority  128
maxStepsRemoved                 255
logAnnounceInterval             -3
logSyncInterval                 -4
logMinDelayReqInterval          -4
announceReceiptTimeout          3
clientOnly                      1
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport               L2
summary_interval                3
message_tag                     "ptp4l"
uds_address                     /var/run/ptp4l
uds_ro_address                  /var/run/ptp4lro

[eno8403np1]
```

The `ptp_dst_mac` value is fixed for G.8275.1: it is the non-forwardable multicast address `01:80:C2:00:00:0E` that confines PTP to a single link and is the reason why every device on the path must be PTP-aware. The `uds_address` exposes a management socket that `phc2sys` uses to follow the port state.

3. Point `ptp4l` at this file in `/etc/sysconfig/ptp4l`:

```
OPTIONS="-f /etc/ptp4l-G.8275.1.conf"
```

4. Configure `phc2sys` in `/etc/sysconfig/phc2sys` to discipline the system clock from the PHC of the PTP interface (`/dev/ptp1`). The `--uds_address` and `--domainNumber` values must match the `ptp4l` configuration:

```
OPTIONS="-s /dev/ptp1 -w -m --uds_address /var/run/ptp4l --domainNumber 24"
```

`-s /dev/ptp1` selects the PHC as the source clock, `-w` waits for `ptp4l` to synchronize before stepping the system clock, and `-m` prints the servo status to the log.

5. Enable and start both services:

```
# systemctl enable --now ptp4l phc2sys
```

6. Verify that `ptp4l` has locked to the upstream master. The port moves to `SLAVE` and the reported offset (`rms`) converges to a low, stable value:

```
# journalctl -u ptp4l -f
ptp4l[...]: "ptp4l" port 1 (eno8403np1): UNCALIBRATED to SLAVE on
 MASTER_CLOCK_SELECTED
```

```
ptp4l[...]: "ptp4l" rms 1502498567 max 8499416372 freq  -3977 +/- 43609 delay  11
 +/- 607
ptp4l[...]: "ptp4l" rms   40 max   99 freq +10129 +/-   30 delay 211 +/-   3
ptp4l[...]: "ptp4l" rms    7 max   15 freq +10125 +/-   13 delay 211 +/-   2
ptp4l[...]: "ptp4l" rms    6 max   16 freq +10118 +/-   12 delay 213 +/-   2
```

7. Verify that `phc2sys` is disciplining the system clock. The servo state progresses from `s0` (unlocked) through `s1` (initial step) to `s2` (locked), and the offset settles around zero:

```
# journalctl -u phc2sys -f
phc2sys[...]: CLOCK_REALTIME phc offset 45583145631 s0 freq   -1673 delay   0
phc2sys[...]: CLOCK_REALTIME phc offset 45583157513 s1 freq  +10207 delay   0
phc2sys[...]: CLOCK_REALTIME phc offset          4 s2 freq  +10211 delay   0
phc2sys[...]: CLOCK_REALTIME phc offset         13 s2 freq  +10221 delay   0
```

A steady `s2` state with a small offset confirms that the node is locked and the system clock is following PTP.

Model 2 — ITU-T G.8275.2 (Layer 3 unicast)

G.8275.2 runs over IP (Layer 3) using unicast: the OC exchanges messages directly with the master's IP address, so the intervening network does not need to be PTP-aware and no PTP-enabled switch is required. The components are:

- **Master (GM or Boundary Clock)**: the upstream master, reachable by IP on the G.8275.2 domain (domain `44`).
- **IP network**: a standard routed or switched network. It forwards the unicast PTP packets without any PTP configuration.
- **Ordinary Clock (OC)**: the cluster node, sending unicast requests to the master from its single PTP port.

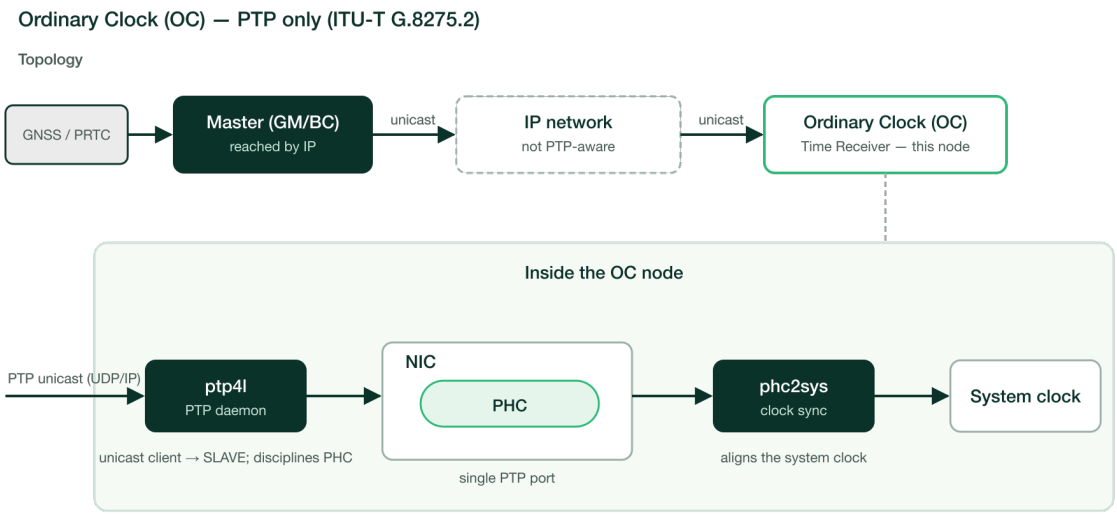


FIGURE 47.7: ORDINARY CLOCK (OC) — G.8275.2 (LAYER 3 UNICAST)

Configure the node as follows:

1. Identify the PHC of the PTP interface as in Model 1 (`ethtool -T eno8403np1`); in this example the PHC of `eno8403np1` is `/dev/ptp1`.

2. Create the `ptp4l` configuration file `/etc/ptp4l-G.8275.2.conf`. There is no `ptp_dst_mac`, because the master is reached by IP through the `unicast_master_table`. Replace `$MASTER_IP_ADDRESS` with the IP address of the upstream master:

EXAMPLE 47.2: ORDINARY CLOCK CONFIGURATION — ITU-T G.8275.2

```
# Telecom G.8275.2 — Ordinary Clock (Time Receiver)
[global]
domainNumber                    44
priority2                       255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     128
G.8275.defaultDS.localPriority  128
maxStepsRemoved                 255
logAnnounceInterval             0
logSyncInterval                 -5
logMinDelayReqInterval          -4
announceReceiptTimeout          2
clientOnly                      1
network_transport               UDPv4
hybrid_e2e                      1
inhibit_multicast_service       1
```

```
unicast_listen                  1
unicast_req_duration            60
message_tag                     "ptp4l"
uds_address                     /var/run/ptp4l
uds_ro_address                  /var/run/ptp4lro

[unicast_master_table]
table_id                        1
logQueryInterval                2
UDPv4                           $MASTER_IP_ADDRESS

[eno8403np1]
unicast_master_table            1
```

3. Point `ptp4l` at this file in `/etc/sysconfig/ptp4l`:

```
OPTIONS="-f /etc/ptp4l-G.8275.2.conf"
```

4. Configure `phc2sys` in `/etc/sysconfig/phc2sys` with the G.8275.2 domain number (`44`); the source PHC and `--uds_address` are unchanged:

```
OPTIONS="-s /dev/ptp1 -w -m --uds_address /var/run/ptp4l --domainNumber 44"
```

5. Enable and start both services:

```
# systemctl enable --now ptp4l phc2sys
```

6. Verify synchronization as in Model 1: `ptp4l` moves the port to `SLAVE` (`journalctl -u ptp4l`) and `phc2sys` reaches the `s2` (locked) state with an offset near zero (`journalctl -u phc2sys`).

CAPI automation

To deploy this scenario using the automated provisioning workflow, use the generic Cluster API template described in Fully automated directed network provisioning (*Section 55.1, "Precision Time Protocol (PTP)"*), overriding its placeholders with the values for this scenario shown below.

TABLE 47.1: MODEL 1 — ITU-T G.8275.1

Placeholder	Value
`${PTP4L_CONF_PATH}`	`/etc/ptp4l-G.8275.1.conf`
`${PTP4L_CONF_CONTENTS}`	Copy the content from the G.8275.1 configuration above (*Example 47.1, "Ordinary Clock configuration — ITU-T G.8275.1"*).

Placeholder	Value
`${PTP4L_OPTIONS}`	`-f /etc/ptp4l-G.8275.1.conf`
`${PHC2SYS_OPTIONS}`	`-s /dev/ptp1 -w -m --uds_address /var/run/ptp4l --domainNumber 24`

TABLE 47.2: MODEL 2 — ITU-T G.8275.2

Placeholder	Value
`${PTP4L_CONF_PATH}`	`/etc/ptp4l-G.8275.2.conf`
`${PTP4L_CONF_CONTENTS}`	Copy the content from the G.8275.2 configuration above (*Example 47.2, "Ordinary Clock configuration — ITU-T G.8275.2"*).
`${PTP4L_OPTIONS}`	`-f /etc/ptp4l-G.8275.2.conf`
`${PHC2SYS_OPTIONS}`	`-s /dev/ptp1 -w -m --uds_address /var/run/ptp4l --domainNumber 44`

47.4 Scenario 1b - Ordinary Clock (Hybrid mode: PTP & SyncE)

This scenario is not yet supported; it will be included in a future SUSE Telco Cloud release.

47.5 Scenario 2a - Boundary Clock (PTP-only) with Multiport Single NIC including propagation to Ordinary Clock server

Explanation

This scenario extends the timing chain with a Boundary Clock (BC) that regenerates PTP towards a downstream Ordinary Clock (OC), entirely over ITU-T G.8275.1 (PTP-only, Layer 2). The BC uses a single multi-port NIC: one `ptp4l` instance manages several ports on the same NIC, so neither `ts2phc` nor DPLL routing is required (unlike the multi-NIC scenarios).

The scenario involves the following components:

- **Grandmaster (GM)**: the reference for the PTP domain (domain `24`), normally locked to a GNSS/PRTC source (the test bench used here has no GNSS, hence the degraded `gm.ClockClass` shown later). It sends time on the upstream segment.
- **PTP-aware switch**: acts as a Boundary Clock (T-BC) on the timing path; every port carrying PTP has `ptp enable`.
- **Boundary Clock (BC server)**: a multi-port node on a single NIC that bridges two PTP segments. Its `eno8403np1` port faces the **upstream** segment as a Time Receiver (SLAVE) and disciplines the NIC's PHC; `eno8503np2` (and `eno8603np3`) face the **downstream** segment as Time Transmitters that regenerate PTP. The BC terminates the upstream timing chain and starts a new one downstream.
- **Ordinary Clock (OC server)**: a single-port node (`ens6f0np0`) on the **downstream** segment, where its only PTP master is the BC. It disciplines its local clocks from the time the BC regenerates.

The timing is handled by the `linuxptp` daemons on each node:

- On the **BC server**, a single `ptp4l` instance manages all its ports. The BMCA selects the GM as master on `eno8403np1`, which moves to `SLAVE` and disciplines the PHC. `eno8503np2` and `eno8603np3` are forced to `serverOnly 1`, so they act only as Time Transmitters. Because all ports share the same NIC PHC, the received time is directly available to the transmit ports; a Boundary Clock is implied automatically when more than one port is configured, so no `clock_type` or `ts2phc` setting is needed.
- On the **OC server**, `ptp4l` runs with `clientOnly 1` on `ens6f0np0`, so the port only ever acts as a Time Receiver. `phc2sys` aligns the system clock to the PHC.
- On the OC server, `clientOnly 1` is what guarantees it never becomes a master. The `localPriority` attribute — used by the G.8275.x BMCA only as a tie-breaker, when clockClass, clockAccuracy, variance and priority2 are equal, where a lower value is preferred — reinforces this ordering: the BC server keeps the default `128`, while the OC server uses the lower-preference `250`.

The BC bridges two separate PTP segments: an **upstream** segment between the GM and its `eno8403np1` port, and a **downstream** segment between its `eno8503np2` port and the OC. These are kept apart, for example on different VLANs of the switch or on separate networks. Because the BC is the only master on the downstream segment, the OC synchronizes from the BC and never directly from the GM.

As in Scenario 1a (*Section 47.3, "Scenario 1a - Ordinary Clock (PTP-only)"*), this scenario uses the non-forwardable multicast MAC `01:80:C2:00:00:0E`; standard bridges do not forward it, so every PTP-aware device terminates and regenerates PTP. The same address must be set on every node of the timing path.

Diagram

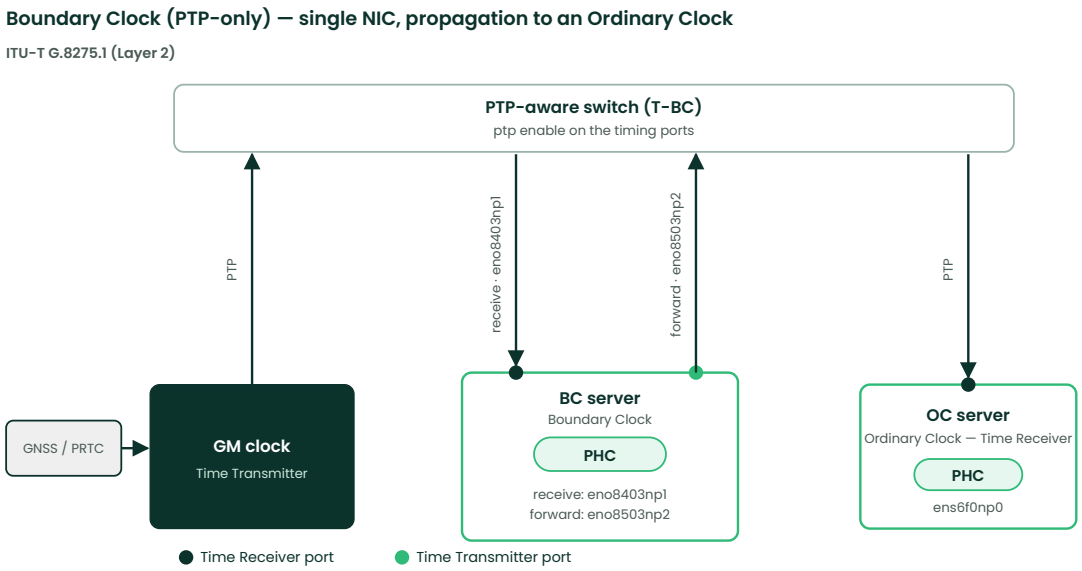


FIGURE 47.8: BOUNDARY CLOCK (PTP-ONLY), SINGLE NIC, WITH PROPAGATION TO AN ORDINARY CLOCK

Configuration

Every switch port on the timing path (towards the GM, the BC and the OC) must have PTP enabled. The two cluster nodes are then configured as follows.

Boundary Clock (BC server)

1. Identify the PHC of the PTP NIC (`ethtool -T eno8403np1`); on a single NIC all ports share the same PHC. In this example it is `/dev/ptp1`.

2. Create `/etc/ptp4l-G.8275.1.conf`. The receiving port `[eno8403np1]` has no role flag, so the BMCA selects it as Time Receiver; `[eno8503np2]` and `[eno8603np3]` are set to `serverOnly 1` to act as Time Transmitters. Declaring more than one port implies a Boundary Clock:

   EXAMPLE 47.3: BOUNDARY CLOCK CONFIGURATION

   ```
   # Telecom G.8275.1 — Boundary Clock (multi-port, single NIC)
   [global]
   domainNumber                    24
   ```

```
priority2                       255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     128
G.8275.defaultDS.localPriority  128
maxStepsRemoved                 255
logAnnounceInterval             -3
logSyncInterval                 -4
logMinDelayReqInterval          -4
announceReceiptTimeout          3
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport               L2
summary_interval                3
message_tag                     "ptp4l"
uds_address                     /var/run/ptp4l
uds_ro_address                  /var/run/ptp4lro

[eno8403np1]

[eno8503np2]
serverOnly                      1

[eno8603np3]
serverOnly                      1
```

3. Point `ptp4l` at this file in `/etc/sysconfig/ptp4l`:

```
OPTIONS="-f /etc/ptp4l-G.8275.1.conf"
```

4. Optionally align the system clock with `phc2sys` in `/etc/sysconfig/phc2sys`, using the shared NIC PHC as source:

```
OPTIONS="-s /dev/ptp1 -w -m --uds_address /var/run/ptp4l --domainNumber 24"
```

5. Enable and start the services and verify that `eno8403np1` is locked to the GM. The receiver port moves to `SLAVE` and the offset (`rms`) converges to a low value:

```
# systemctl enable --now ptp4l phc2sys
# journalctl -u ptp4l -f
ptp4l[...]: "ptp4l" selected best master clock d0460c.ffff.05c130
ptp4l[...]: "ptp4l" port 1 (eno8403np1): UNCALIBRATED to SLAVE on
 MASTER_CLOCK_SELECTED
ptp4l[...]: "ptp4l" rms    8 max   24 freq +10092 +/-  18 delay   211 +/-   2
ptp4l[...]: "ptp4l" rms    4 max   11 freq +10069 +/-   7 delay   210 +/-   2
ptp4l[...]: "ptp4l" rms    3 max    9 freq +10070 +/-   6 delay   213 +/-   3
```

Ordinary Clock (OC server)

1. Identify the PHC of the PTP interface (`ethtool -T ens6f0np0`); in this example it is `/dev/ptp1`.

2. Create `/etc/ptp4l-G.8275.1.conf` with a single port and `clientOnly 1`, so the node only ever acts as a Time Receiver. The higher `localPriority` (`250`) keeps it below the BC in the BMCA:

EXAMPLE 47.4: ORDINARY CLOCK CONFIGURATION

```
# Telecom G.8275.1 — Ordinary Clock (Time Receiver)
[global]
domainNumber                    24
priority2                       255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     250
G.8275.defaultDS.localPriority  250
maxStepsRemoved                 255
logAnnounceInterval             -3
logSyncInterval                 -4
logMinDelayReqInterval          -4
announceReceiptTimeout          3
clientOnly                      1
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport               L2
summary_interval                3
message_tag                     "ptp4l"
uds_address                     /var/run/ptp4l
uds_ro_address                  /var/run/ptp4lro

[ens6f0np0]
```

3. Configure `/etc/sysconfig/ptp4l` and `/etc/sysconfig/phc2sys` as for the BC (the `phc2sys` source is the PHC of `ens6f0np0`), then enable and start the services:

```
# /etc/sysconfig/ptp4l
OPTIONS="-f /etc/ptp4l-G.8275.1.conf"

# /etc/sysconfig/phc2sys
OPTIONS="-s /dev/ptp1 -w -m --uds_address /var/run/ptp4l --domainNumber 24"
```

4. Verify that `ens6f0np0` is locked to the regenerated time from the BC. The port moves to `SLAVE` and the offset (`rms`) stabilizes:

```
# journalctl -u ptp4l -f
ptp4l[...]: "ptp4l" selected best master clock d0460c.ffff.05c130
ptp4l[...]: "ptp4l" port 1 (ens6f0np0): UNCALIBRATED to SLAVE on
 MASTER_CLOCK_SELECTED
ptp4l[...]: "ptp4l" rms   15 max   34 freq +11699 +/-  25 delay   468 +/-   5
```

```
ptp4l[...]: "ptp4l" rms   13 max   30 freq +11701 +/-  23 delay   468 +/-   4
ptp4l[...]: "ptp4l" rms   14 max   30 freq +11697 +/-  24 delay   467 +/-   4
```

The OC takes its time from the upstream Boundary Clock, which is its immediate master. The `selected best master clock` line, however, reports the **Grandmaster** identity (`d0460c.ffff.05c130`, the GM) and not the Boundary Clock's: a Boundary Clock forwards the Grandmaster identity unchanged while advertising its own port as the parent, so the chain stays traceable to the Grandmaster. To see both identities explicitly, query the parent data set on the OC (example output):

```
# pmc -f /etc/ptp4l-G.8275.1.conf -u -b 0 'GET PARENT_DATA_SET'
sending: GET PARENT_DATA_SET
 507c6f.fffe.4ae178-0 seq 0 RESPONSE MANAGEMENT PARENT_DATA_SET
  parentPortIdentity                    000000.fffe.001101-2
  parentStats                           0
  observedParentOffsetScaledLogVariance 0xffff
  observedParentClockPhaseChangeRate    0x7fffffff
  grandmasterPriority1                  128
  gm.ClockClass                         165
  gm.ClockAccuracy                      0xfe
  gm.OffsetScaledLogVariance            0xffff
  grandmasterPriority2                  120
  grandmasterIdentity                   d0460c.ffff.05c130
```

The two identities differ, which is exactly what a Boundary Clock deployment should show:

- `parentPortIdentity` (`000000.fffe.001101-2`) — the **immediate master**: the Boundary Clock's transmit port, the clock the OC actually receives time from.
- `grandmasterIdentity` (`d0460c.ffff.05c130`) — the **root of the timing tree**: the Grandmaster, forwarded unchanged through the Boundary Clock.
  The `gm.ClockClass`, `gm.ClockAccuracy` and `grandmasterPriority` values carry the Grandmaster's advertised quality used by the BMCA. If the two identities were equal, the OC would be synchronizing directly from a Grandmaster, with no Boundary Clock in the path.

CAPI automation

To deploy this scenario using the automated provisioning workflow, use the generic Cluster API template described in Fully automated directed network provisioning (*Section 55.1, "Precision Time Protocol (PTP)"*) on each node, overriding its placeholders with the values below.

TABLE 47.3: BOUNDARY CLOCK (BC SERVER)

Placeholder	Value
`${PTP4L_CONF_PATH}`	`/etc/ptp4l-G.8275.1.conf`
`${PTP4L_CONF_CONTENTS}`	Copy the content from the Boundary Clock configuration above (*Example 47.3, "Boundary Clock configuration"*).
`${PTP4L_OPTIONS}`	`-f /etc/ptp4l-G.8275.1.conf`
`${PHC2SYS_OPTIONS}`	`-s /dev/ptp1 -w -m --uds_address /var/run/ptp4l --domainNumber 24`

TABLE 47.4: ORDINARY CLOCK (OC SERVER)

Placeholder	Value
`${PTP4L_CONF_PATH}`	`/etc/ptp4l-G.8275.1.conf`
`${PTP4L_CONF_CONTENTS}`	Copy the content from the Ordinary Clock configuration above (*Example 47.4, "Ordinary Clock configuration"*).
`${PTP4L_OPTIONS}`	`-f /etc/ptp4l-G.8275.1.conf`
`${PHC2SYS_OPTIONS}`	`-s /dev/ptp1 -w -m --uds_address /var/run/ptp4l --domainNumber 24`

47.6 Scenario 2b - Boundary Clock (Hybrid mode: PTP & SyncE) with Multiport Single NIC including propagation to Ordinary Clock server

This scenario is not yet supported; it will be included in a future SUSE Telco Cloud release.

47.7 Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based)

Explanation

SUSE Telco Cloud supports precise timing on a variety of platforms, including the latest Intel Granite Rapids-D based designs (GNR-D). These servers typically include an embedded high-speed Ethernet controller and one or more add-in cards (AICs) with an Ethernet controller for additional ports. Contrary to previous platform generations, they are also equipped with a dedicated DPLL that routes the timing signals across the system and the different NICs internally, no longer requiring external cabling to keep all NIC PHCs in sync. The overall deployment is thus simplified, but specific steps and software components are required, such as boot-time DPLL and internal signal configuration.

This scenario extends Scenario 2a (*Section 47.5, "Scenario 2a - Boundary Clock (PTP-only) with Multiport Single NIC including propagation to Ordinary Clock server"*) to a Boundary Clock (BC) built on several NICs, entirely over ITU-T G.8275.1 (PTP-only, Layer 2). One port of the integrated controller receives PTP from the Grandmaster and the add-in cards regenerate PTP downstream. Unlike the single-NIC case, each NIC has its own PHC; the on-board DPLL and `ts2phc` keep them aligned, and one `ptp4l` instance runs per PHC.

Important

In this SUSE Telco Cloud release this scenario is only supported for the "default" (i.e., non "PREEMPT_RT") kernel flavour. Support for the "RT" kernel flavour will be added in a following release.

PREREQUISITES

- Download and install the latest kernel driver for the *Intel 800 Series Network Devices* from the Intel website (https://www.intel.com/content/www/us/en/download/19630/intel-network-adapter-driver-for-800-series-devices-under-linux.html).
- Select the zip file containing the "ice_RPM_Files", unpack the archive and install the `ice-kmp-default-*sles16sp0.x86_64.rpm`. This RPM contains, for example, `ice-kmp-default-2.4.5_k6.12.0_160000.5-1.sles16sp0.x86_64.rpm`:

  ```
  # transactional-update pkg install ice-kmp-
  default-2.4.5_k6.12.0_160000.5-1.sles16sp0.x86_64.rpm
  ```

  Reboot the server to make the change persistent and effective.

Note

- A Dell PowerEdge XR8720t system is used as the reference system throughout the examples; the kernel interfaces and port configuration might be different on a different GNR-D server. Adjust the device names accordingly.
- A telco deployment is assumed and the telco profile G.8275.1 is used throughout the following sections.

The scenario involves the following components:

- **Grandmaster (GM)**: the reference for the PTP domain (domain `24`), on the upstream segment.
- **PTP-aware switch**: acts as a Boundary Clock (T-BC) on the timing path; every port carrying PTP has `ptp enable`. The upstream and downstream segments are kept apart on different VLANs of the switch (a dedicated VLAN for each segment).
- **Boundary Clock (BC server)**: the GNR-D node. Its integrated controller receives PTP from the GM on `eno8403np1` and disciplines the integrated PHC (`/dev/ptp1`). The on-board DPLL distributes that timing to the add-in cards and `ts2phc` keeps their PHCs (`/dev/ptp2`, `/dev/ptp3`) aligned with `/dev/ptp1`. Each add-in card regenerates PTP downstream (for example on `eno16495np0`), acting as a Time Transmitter.
- **Ordinary Clock (OC server)**: a single-port node (`ens6f0np0`) on the downstream segment, whose only PTP master is the BC.

The timing on the **BC server** is handled by several `linuxptp` daemons working together, because the PHCs are physically separate:

- A `ptp4l` instance on the integrated controller (`/dev/ptp1`) runs the PTP protocol on `eno8403np1`. The BMCA selects the GM as master, the port moves to `SLAVE` and `ptp4l` disciplines `/dev/ptp1`.
- The on-board DPLL is wired so that the integrated PHC drives it through the SDP pins, and the DPLL in turn drives the add-in cards' clocks. This is the internal replacement for the external 1PPS cabling used on previous platforms.
- `ts2phc` keeps the add-in cards' PHCs aligned to `/dev/ptp1` from the signal the DPLL distributes.

- A separate `ptp4l` instance per add-in card (`/dev/ptp2`, `/dev/ptp3`) regenerates PTP on its ports, all forced to `serverOnly 1` (Time Transmitters).
- A Boundary Clock must forward the Grandmaster's ancillary attributes (clock class, accuracy, and so on) in addition to time. This happens automatically only within a single `ptp4l` instance, so `pmc` is used to forward those values from the integrated instance to the add-in instances (see step 5). `pmc` (PTP management client) is the `linuxptp` tool that queries and updates the runtime state of a running `ptp4l` instance over its management socket.

Note

On the integrated controller only **one** of its ports is the administrative port that manages the PHC. Inspect `/sys/class/ptp/ptp1/device/net/` to find it (for example `eno8703np0`); it is the port used to configure the DPLL signals, and it is not necessarily the port that carries PTP (`eno8403np1`).

Note

The setup described here is completely static. If the GM becomes unavailable, downstream synchronization relies exclusively on the host holdover capabilities; no automatic switching to a different time reference is performed.

Diagram

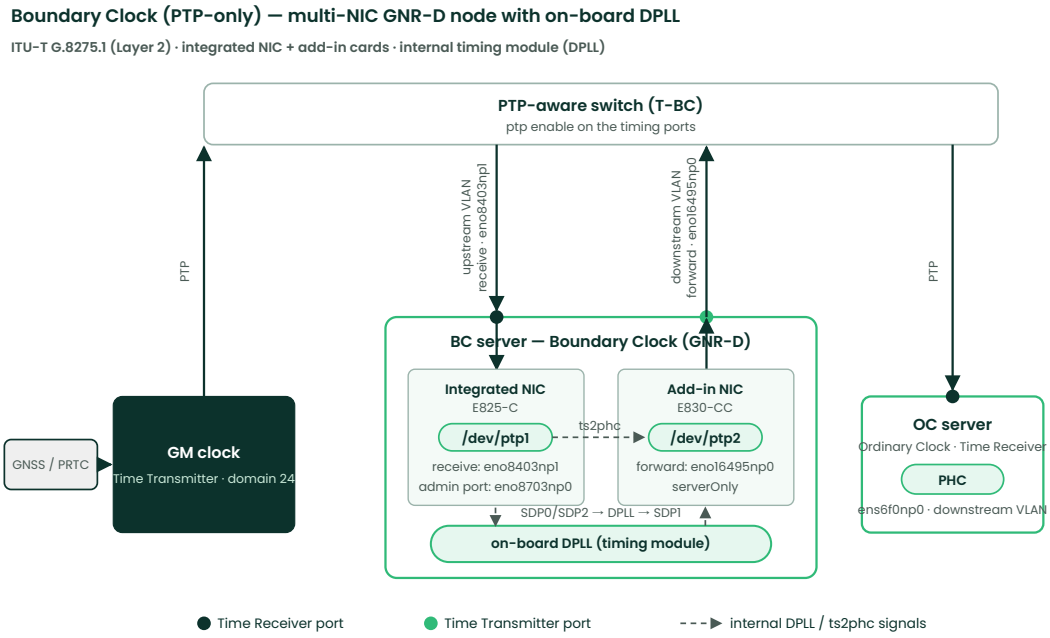


FIGURE 47.9: BOUNDARY CLOCK (PTP-ONLY), MULTI-NIC GNR-D NODE WITH ON-BOARD DPLL, WITH PROPAGATION TO AN ORDINARY CLOCK

Configuration

Every switch port on the timing path must have PTP enabled. The BC server is then brought up in the order below. Each runtime step is also mapped to a `systemd` unit so that the configuration survives reboots and can be automated with Cluster API (see the GNR-D CAPI automation (*Section 55.1.2, "GNR-D multi-NIC scenarios (DPLL)"*)). The OC server is configured exactly as in Scenario 2a (*Section 47.5, "Scenario 2a - Boundary Clock (PTP-only) with Multiport Single NIC including propagation to Ordinary Clock server"*).

1. Identify the PTP devices and the integrated controller.

   a. List the PTP clocks and the PCI device each is bound to:

```
# ls -l /sys/class/ptp/
lrwxrwxrwx 1 root root 0 ... ptp1 -> ../../devices/
pci0000:12/0000:12:04.0/0000:13:00.0/ptp/ptp1
lrwxrwxrwx 1 root root 0 ... ptp2 -> ../../devices/
pci0000:6b/0000:6b:02.0/0000:6c:00.0/ptp/ptp2
```

```
lrwxrwxrwx 1 root root 0 ... ptp3 -> ../../devices/
pci0000:6b/0000:6b:06.0/0000:6e:00.0/ptp/ptp3
```

b. Run `lspci` to identify each device type, allowing you to distinguish between the integrated controller and the add-in cards:

```
# lspci -s 0000:13:00.0
13:00.0 Ethernet controller: Intel Corporation Ethernet Connection E825-C for
 SFP (rev 04)
# lspci -s 0000:6c:00.0
6c:00.0 Ethernet controller: Intel Corporation Ethernet Controller E830-CC for
 SFP
# lspci -s 0000:6e:00.0
6e:00.0 Ethernet controller: Intel Corporation Ethernet Controller E830-CC for
 SFP
```

This reference server has an *Intel E825-C* integrated controller (`/dev/ptp1`) and two *E830-CC* add-in cards (`/dev/ptp2`, `/dev/ptp3`).

c. Identify the administrative port of the integrated controller, which manages its PHC:

```
# ls /sys/class/ptp/ptp1/device/net/
eno8703np0
# ethtool -i eno8703np0
driver: ice
version: 2.4.5
[...]
```

2. Configure the internal timing signals (DPLL).
Wire the integrated PHC to the DPLL and let the DPLL feed the add-in cards, using the administrative port found above (`eno8703np0`) and the correct `ptpN` indices:

EXAMPLE 47.5: `01-gnrd-startup-setup.sh`

```
#!/bin/bash

# The admin port that manages the PHC is under /sys/class/ptp/ptp1/device/net/ (e.g.
 eno8703np0).
# Only one port of the integrated E825-C NIC exposes it.

# Let the DPLL drive the Ethernet clock on the integrated controller
echo 4 1 > /sys/class/net/eno8703np0/device/tspll_cfg

# Feed the DPLL from the integrated PHC through the SDP pins:
# SDP0 -> 1 pulse per second (1PPS, 1 Hz): enable the pin, then set a 1 s period
echo 2 2 > /sys/class/ptp/ptp1/pins/SDP0
```

```
echo 2 0 0 1 0 > /sys/class/ptp/ptp1/period

# SDP2 -> 1000 pulses per second (1kPPS, 1 kHz): enable the pin, then set a 1 ms
 period
echo 2 1 > /sys/class/ptp/ptp1/pins/SDP2
echo 1 0 0 0 1000000 > /sys/class/ptp/ptp1/period

# Propagate the pulses from the DPLL to the add-in cards (SDP1 on each add-in PHC)
echo 1 1 > /sys/class/ptp/ptp2/pins/SDP1
echo 1 1 > /sys/class/ptp/ptp3/pins/SDP1
```

These settings are lost on reboot, so run the script at boot from a oneshot `systemd` unit, ordered before `ts2phc` and `ptp4l`:

```
# /etc/systemd/system/gnrd-ptp-dpll-setup.service
[Unit]
Description=GNR-D PTP DPLL and internal signal setup
Before=ts2phc.service ptp4l@nac.service ptp4l@aic1.service ptp4l@aic2.service

[Service]
Type=oneshot
RemainAfterExit=yes
ExecStart=/usr/local/bin/01-gnrd-startup-setup.sh

[Install]
WantedBy=multi-user.target
```

3. Align the add-in cards' PHCs with `ts2phc`.
   `ts2phc` ("time stamp to PHC") aligns the add-in cards' PHCs to a reference clock. Here, the reference is the PHC from the integrated NIC (`/dev/ptp1`, already disciplined by the GM through its `ptp4l` instance), passed on the command line with `-s /dev/ptp1`. The on-board DPLL delivers the 1PPS derived from that reference to each add-in card on its `SDP1` pin; `ts2phc` timestamps those input pulses (EXTTS) and steps each add-in PHC accordingly, so every PHC in the node ends up sharing the same time.

Note

Aligning the add-in PHCs takes two complementary settings, at different levels:

- **Hardware**: the `SDP1` input pin of each add-in PHC must be enabled for the on-board DPLL to deliver the 1PPS pulse to it. This is done in the DPLL setup script of the previous step (`echo 1 1 > /sys/class/ptp/ptpN/pins/SDP1`).
- **Software**: `ts2phc` must be told to timestamp that input and step the PHC. This is what the configuration below does (`ts2phc.channel`, `ts2phc.pin_index`, `ts2phc.extts_polarity`).

Without enabling the pin, the pulse never reaches the PHC; without the `ts2phc` configuration the pulse is never applied to it. Both are required.

Create its configuration file:

EXAMPLE 47.6: `/etc/ts2phc-cf-all.cfg` — `ts2phc` CONFIGURATION FILE

```
[global]
use_syslog              0
verbose                 1
logging_level           7
ts2phc.pulsewidth       100000000

[eno16495np0]
ts2phc.channel          1
ts2phc.extts_polarity   rising
ts2phc.pin_index        1

[eno16595np0]
ts2phc.channel          1
ts2phc.extts_polarity   rising
ts2phc.pin_index        1
```

Each add-in card is configured through a section named after its first port (`[eno16495np0]`, `[eno16595np0]`), which identifies the PHC to discipline. The parameters map the DPLL pulse to that card:

- `ts2phc.pin_index` and `ts2phc.channel`: the pin and channel where the DPLL delivers the 1PPS (`SDP1` in this setup, hence `pin_index 1`).
- `ts2phc.extts_polarity`: the pulse edge to timestamp (`rising`).
- `ts2phc.pulsewidth`: the expected input pulse width, in nanoseconds (100 ms here).
  Run it manually as `ts2phc -f /etc/ts2phc-cf-all.cfg -s /dev/ptp1 -m`, or let `systemd` manage it through `/etc/sysconfig/ts2phc`:

  ```
  # /etc/sysconfig/ts2phc
  OPTIONS="-f /etc/ts2phc-cf-all.cfg -s /dev/ptp1 -m"
  ```

4. Run one `ptp4l` instance per PHC.
   The integrated controller runs a BMCA-driven instance (the receiver), and each add-in card runs an instance with all ports forced to `serverOnly 1`. Each instance must use a different control and read-only socket. Create the three configuration files:

EXAMPLE 47.7: `/etc/ptp4l-nac.conf` — INTEGRATED CONTROLLER (`/dev/ptp1`)

```
# Telecom G.8275.1 — integrated controller (receiver)
[global]
domainNumber                    24
priority2                       255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     128
G.8275.defaultDS.localPriority  128
maxStepsRemoved                 255
logAnnounceInterval             -3
logSyncInterval                 -4
logMinDelayReqInterval          -4
announceReceiptTimeout          3
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport               L2
summary_interval                3
message_tag                     "ptp4l-nac"
uds_address                     /var/run/ptp4l-nac
uds_ro_address                  /var/run/ptp4lro-nac

[eno8403np1]
```

The integrated instance declares only `eno8403np1`, the port that receives PTP from the GM, and sets no `serverOnly` or `clientOnly` on it: it relies on the BMCA to assign the role. With a GM reachable on `eno8403np1`, the BMCA selects it as the Time Receiver (`SLAVE`) and disciplines `/dev/ptp1`. The other ports of the integrated controller are not declared, because on this node the downstream regeneration is performed by the add-in cards rather than by the integrated controller. The add-in cards, in contrast, force `serverOnly 1` because they must always be Time Transmitters — they take their time from the DPLL and `ts2phc`, not by receiving PTP, so they must never be elected as receivers. If tighter control is needed, add the `serverOnly` or `clientOnly` flag explicitly under the interface.

The `G.8275.portDS.localPriority` and `G.8275.defaultDS.localPriority` values (`128` on the integrated instance, `250` on the add-in cards) act as a G.8275.x BMCA tie-breaker: they are compared only after `clockClass`, `clockAccuracy`, variance and `priority2`, and a lower value is preferred. Keeping the default `128` on the integrated instance makes it outrank the add-in instances (`250`), which keeps the Time Transmitter/Time Receiver ordering within the node deterministic and drives which port the BMCA elects, as reflected in the port states reported by `ptp4l`.

EXAMPLE 47.8: `/etc/ptp4l-aic1.conf` — ADD-IN CARD 1 (`/dev/ptp2`)

```
# Telecom G.8275.1 – add-in card (Time Transmitter)
[global]
domainNumber                    24
priority2                       255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     250
G.8275.defaultDS.localPriority  250
maxStepsRemoved                 255
logAnnounceInterval             -3
logSyncInterval                 -4
logMinDelayReqInterval          -4
announceReceiptTimeout          3
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport               L2
summary_interval                3
message_tag                     "ptp4l-aic1"
uds_address                     /var/run/ptp4l-aic1
uds_ro_address                  /var/run/ptp4lro-aic1

[eno16495np0]
serverOnly                      1
```

EXAMPLE 47.9: `/etc/ptp4l-aic2.conf` — ADD-IN CARD 2 (`/dev/ptp3`)

```
# Telecom G.8275.1 – add-in card (Time Transmitter)
[global]
domainNumber                    24
priority2                       255
dataset_comparison              G.8275.x
G.8275.portDS.localPriority     250
G.8275.defaultDS.localPriority  250
maxStepsRemoved                 255
logAnnounceInterval             -3
logSyncInterval                 -4
logMinDelayReqInterval          -4
announceReceiptTimeout          3
ptp_dst_mac                     01:80:C2:00:00:0E
network_transport               L2
summary_interval                3
message_tag                     "ptp4l-aic2"
uds_address                     /var/run/ptp4l-aic2
uds_ro_address                  /var/run/ptp4lro-aic2

[eno16595np0]
serverOnly                      1
```

Note

As in Scenario 2a (*Section 47.5, "Scenario 2a - Boundary Clock (PTP-only) with Multiport Single NIC including propagation to Ordinary Clock server"*), every node on the timing path uses the same non-forwardable G.8275.1 multicast MAC `01:80:C2:00:00:0E`, including the downstream OC.

Manage the instances with a templated `systemd` unit that reads `/etc/ptp4l-<instance>.conf`:

```
# /etc/systemd/system/ptp4l@.service
[Unit]
Description=PTP Boundary Clock instance %i
After=gnrd-ptp-dpll-setup.service
Requires=gnrd-ptp-dpll-setup.service

[Service]
ExecStart=/usr/sbin/ptp4l -f /etc/ptp4l-%i.conf

[Install]
```

```
WantedBy=multi-user.target
```

Enable and start the three instances:

```
# systemctl enable --now ptp4l@nac ptp4l@aic1 ptp4l@aic2
```

5. Forward the Grandmaster parameters across the `ptp4l` instances.
   A Boundary Clock must forward not only time but also the Grandmaster's ancillary parameters (clock class, accuracy, UTC offset, and so on). This happens automatically between ports of the **same** `ptp4l` instance, but not across the separate instances that manage different PHCs. The values are applied at runtime with `pmc` over each instance's UDS socket (`SET GRANDMASTER_SETTINGS_NP`).

> **Important**
>
> The values applied with `pmc` are held only in the running `ptp4l` process; they are **not** written to the `.conf` file and do **not** survive a restart. If a `ptp4l` instance is restarted, it reverts to the file defaults and the values must be forwarded again. Automate the forwarding (see below) rather than running it once by hand.

Once the integrated controller is locked to the GM, query the Grandmaster values. The helper below takes the configuration file of the instance to query as its argument:

EXAMPLE 47.10: `get_pmc_values.sh` — QUERY THE GRANDMASTER VALUES FROM A `ptp4l` INSTANCE

```
#!/bin/bash
# Retrieve Grandmaster information via pmc for the given ptp4l config file
set -euo pipefail

echo "PARENT_DATA_SET (Grandmaster clock values):"
pmc -f "$1" -u -b 0 'GET PARENT_DATA_SET' | grep gm

echo "TIME_PROPERTIES_DATA_SET:"
pmc -f "$1" -u -b 0 'GET TIME_PROPERTIES_DATA_SET'
```

Then forward those values to the add-in instances. The following script reads the values from the integrated instance and applies them to each add-in instance. Edit the "Input variables" section to match your configuration file names, and run it once the system is synchronized to a GM:

EXAMPLE 47.11: `04-ptp-forward-GM.sh` — FORWARD THE GRANDMASTER VALUES TO THE ADD-IN INSTANCES

```
#!/bin/bash
```

```
set -euo pipefail

################## Input variables ##################
VERBOSE=1

CONFIG_FILE_NAC=/etc/ptp4l-nac.conf
CONFIG_FILES_AICS=(/etc/ptp4l-aic1.conf /etc/ptp4l-aic2.conf)

#####################################################


# Check the system is synchronized to a GM
GM_PRESENT=$(pmc -f ${CONFIG_FILE_NAC} -u -b 0 'GET TIME_STATUS_NP' | awk '/
gmPresent/ {print $2}')

if [ -z "$GM_PRESENT" ]; then
    echo "Failed to get GM value from ptp4l instance for NAC, exiting..."
    exit 1
elif [ "${GM_PRESENT}" != "true" ]; then
    echo "No GM present, exiting..."
    exit 1
fi

# Get the "upstream" Grandmaster values
PDS_GM_VALUES=$(pmc -f ${CONFIG_FILE_NAC} -u -b 0 'GET PARENT_DATA_SET' | awk 'BEGIN
 {ORS=" "} /gm./ {print $2}')
TPDS_GM_VALUES=$(pmc -f ${CONFIG_FILE_NAC} -u -b 0 'GET TIME_PROPERTIES_DATA_SET' |
 awk 'BEGIN {ORS=" "} NR>2 {print $2}')

if [ -z "$PDS_GM_VALUES" ]; then
    echo "No PARENT_DATA_SET values retrieved, exiting ..."
    exit 2
elif [ -z "$TPDS_GM_VALUES" ]; then
    echo "No TIME_PROPERTIES_DATA_SET values retrieved, exiting ..."
    exit 2
fi

# Unpack from PARENT_DATA_SET
read CLOCK_CLASS CLOCK_ACCURACY VARIANCE <<< $PDS_GM_VALUES

# Unpack from TIME_PROPERTIES_DATA_SET
read UTC_OFFSET LEAP_61 LEAP_59 UTC_OFFSET_VALID PTP_TIMESCALE T_TRACEABLE
 F_TRACEABLE T_SOURCE <<< $TPDS_GM_VALUES

if [ "$VERBOSE" -eq 1 ]; then
```

```
    echo "Read from PARENT_DATA_SET: clockClass=${CLOCK_CLASS} clockAccuracy=
${CLOCK_ACCURACY} offsetScaledLogVariance=${VARIANCE}"
    echo "Read from TIME_PROPERTIES_DATA_SET: currentUtcOffset=${UTC_OFFSET} leap61=
${LEAP_61} leap59=${LEAP_59}" \
         "currentUtcOffsetValid=${UTC_OFFSET_VALID} ptpTimescale=${PTP_TIMESCALE}
 timeTraceable=${T_TRACEABLE}" \
         "frequencyTraceable=${F_TRACEABLE} timeSource=${T_SOURCE}"
fi


# Forward these values to the "downstream" Time Receivers, updating the ptp4l
 instances
for FILE in "${CONFIG_FILES_AICS[@]}"; do
    if [ "$VERBOSE" -eq 1 ]; then
        echo "Updating GRANDMASTER_SETTINGS_NP (${FILE})"
    fi

    pmc -f ${FILE} -u -b 0 'SET GRANDMASTER_SETTINGS_NP
       clockClass              '${CLOCK_CLASS}'
       clockAccuracy           '${CLOCK_ACCURACY}'
       offsetScaledLogVariance '${VARIANCE}'
       currentUtcOffset        '${UTC_OFFSET}'
       leap61                  '${LEAP_61}'
       leap59                  '${LEAP_59}'
       currentUtcOffsetValid   '${UTC_OFFSET_VALID}'
       ptpTimescale            '${PTP_TIMESCALE}'
       timeTraceable           '${T_TRACEABLE}'
       frequencyTraceable      '${F_TRACEABLE}'
       timeSource              '${T_SOURCE}'' &> /dev/null

    if [ $? -ne 0 ]; then
        echo "Failed to update ptp4l instance using config file \"${FILE}\",
 exiting ..."
        exit 1
    fi
done
```

Because the values are not persistent, run this script from a `systemd` timer so that they are refreshed periodically and after any instance restart:

```
# /etc/systemd/system/ptp-forward-gm.service
[Unit]
Description=Forward Grandmaster parameters to the add-in ptp4l instances
Wants=ptp4l@nac.service ptp4l@aic1.service ptp4l@aic2.service
After=ptp4l@nac.service ptp4l@aic1.service ptp4l@aic2.service
```

```
[Service]
Type=oneshot
SELinuxContext=system_u:system_r:unconfined_t:s0
ExecStart=/usr/local/bin/04-ptp-forward-GM.sh
```

```
# /etc/systemd/system/ptp-forward-gm.timer
[Unit]
Description=Refresh forwarded Grandmaster parameters

[Timer]
OnBootSec=2min
OnUnitActiveSec=5min

[Install]
WantedBy=timers.target
```

The effect is visible by querying an add-in `ptp4l` instance on the BC: before forwarding, the node advertises its file defaults; after forwarding begins, it advertises the Grand Master's actual values (which then propagate to the downstream OC). For example, `gm.ClockClass` changes:

```
# ./get_pmc_values.sh /etc/ptp4l-aic1.conf     # before forwarding
        gm.ClockClass                          165
        gm.ClockAccuracy                       0xfe
        gm.OffsetScaledLogVariance             0xffff
# ./get_pmc_values.sh /etc/ptp4l-aic1.conf     # after forwarding
        gm.ClockClass                          8
        gm.ClockAccuracy                       0xfe
        gm.OffsetScaledLogVariance             0xffff
```

6. Optionally, derive the system clock from PTP with `phc2sys`, using the integrated PHC as source (`/etc/sysconfig/phc2sys`):

```
OPTIONS="-f /etc/ptp4l-nac.conf -s /dev/ptp1 -c CLOCK_REALTIME -w"
```

CAPI automation

To deploy this scenario with the automated provisioning workflow, use the GNR-D-specific Cluster API template described in Fully automated directed network provisioning — GNR-D scenarios (*Section 55.1.2, "GNR-D multi-NIC scenarios (DPLL)"*). Unlike the standard scenarios, it deploys several `ptp4l` instances, the `ts2phc` service, the DPLL boot-time setup and the Grandmaster forwarding timer, so it requires the additional configuration files and `systemd` units listed above. Override its placeholders with the values below.

TABLE 47.5: GNR-D BOUNDARY CLOCK

Placeholder	Value
`${GNRD_DPLL_SETUP_CONTENTS}`	Copy the content from 01-gnrd-startup-setup.sh (*Example 47.5, "`01-gnrd-startup-setup.sh`"*).
`${GNRD_FORWARD_GM_CONTENTS}`	Copy the content from 04-ptp-forward-GM.sh (*Example 47.11, "`04-ptp-forward-GM.sh` — forward the Grandmaster values to the add-in instances"*).
`${TS2PHC_CONF_CONTENTS}`	Copy the content from /etc/ts2phc-cf-all.cfg (*Example 47.6, "`/etc/ts2phc-cf-all.cfg` — `ts2phc` configuration file"*).
`${TS2PHC_OPTIONS}`	`-f /etc/ts2phc-cf-all.cfg -s /dev/ptp1 -m`
`${PTP4L_NAC_CONTENTS}`	Copy the content from /etc/ptp4l-nac.conf (*Example 47.7, "`/etc/ptp4l-nac.conf` — integrated controller (`/dev/ptp1`)"*).
`${PTP4L_AIC1_CONTENTS}`	Copy the content from /etc/ptp4l-aic1.conf (*Example 47.8, "`/etc/ptp4l-aic1.conf` — add-in card 1 (`/dev/ptp2`)"*).
`${PTP4L_AIC2_CONTENTS}`	Copy the content from /etc/ptp4l-aic2.conf (*Example 47.9, "`/etc/ptp4l-aic2.conf` — add-in card 2 (`/dev/ptp3`)"*).

47.8 Scenario 3b - Boundary Clock (Hybrid mode: PTP & SyncE) Multi-NIC with internal timing module (DPLL GNR-D based)

This scenario is not yet supported; it will be included in a future SUSE Telco Cloud release.

48 SCTP - Stream Control Transmission Protocol

From Stream Control Transmission Protocol - Wikipedia (https://en.wikipedia.org/wiki/Stream_Control_Transmission_Protocol)

> The Stream Control Transmission Protocol (SCTP) is a computer networking communications protocol in the transport layer of the Internet protocol suite. Originally intended for Signaling System 7 (SS7) message transport in telecommunication, the protocol provides the message-oriented feature of the User Datagram Protocol (UDP) while ensuring reliable, in-sequence transport of messages with congestion control like the Transmission Control Protocol (TCP). Unlike UDP and TCP, the protocol supports multihoming and redundant paths to increase resilience and reliability.
>
> SCTP is standardized by the Internet Engineering Task Force (IETF) in RFC 9260. The SCTP reference implementation was released as part of FreeBSD version 7 and has since been widely ported to other platforms.

In 3GPP 4G (LTE) specifications, SCTP (Stream Control Transmission Protocol) acts as the foundational Layer 4 transport for control-plane signaling messages across the Evolved Packet Core (EPC), including the signaling connection with the Radio Access Network (RAN) domain. 3GPP 5G specifications have instead removed the need for SCTP inside the Core Network through the introduction of the Service Based Architecture (SBA) which replaces all the Diameter-over-SCTP based signalling interfaces by HTTP/2. However it is still required inside the 5G RAN domain:

- in the signaling interfaces across the network elements making up a "disaggregated" gNodeB instance (F1-C interface between DU and CU-CP and E1 interface between CU-CP and CU-UP).
- in the Xn-C interface connecting gNodeB instances for signaling purposes.
- in the N2 (aka NG-C) reference point used to connect gNodeB instances (RAN domain) with AMF instances (Core Nework domain) for signaling purposes.

You must then enable the `sctp` linuk kernel module (`lksctp`) on downstream cluster nodes hosting the SCTP-capable Telco workloads that leverage it:

- To manually load the `sctp` kernel module on a linux node:

```
$ sudo modprobe sctp
```

- To automate loading the `sctp` kernel module at every node boot, create a file named `sctp.conf` in `/etc/modules-load.d/` directory that simply contains the string `sctp` (that is, the name of the kernel module that systemd-modules-load.service (https://www.freedesktop.org/software/systemd/man/latest/modules-load.d.html) unit should load at boot).

```
$ echo "sctp" > /etc/modules-load.d/sctp.conf
$ reboot
```

The easiest and best way to ensure the `sctp` kernel module loads at every node boot is to add that `/etc/modules-load.d/sctp.conf` file to the Edge Image Builder (EIB) built raw image used to provision downstream nodes; see the EIB documentation (*Section 5.3.5, "Adding Operating System Files"*) for more information on how to do it.

VII Fully automated directed network provisioning

49 Introduction 297

50 Prepare downstream cluster image for connected scenarios 299

51 Prepare downstream cluster image for air-gap scenarios 308

52 Downstream cluster provisioning with Directed network provisioning (single-node) 315

53 Downstream cluster provisioning with Directed network provisioning (multi-node) 324

54 Advanced Network Configuration 339

55 Telco features (DPDK, SR-IOV, CPU isolation, huge pages, NUMA, etc.) 344

56 Private registry 361

57 Downstream cluster provisioning in air-gapped scenarios 364

49 Introduction

This chapter describes how to provision downstream clusters using Directed Network Provisioning, the supported provisioning method for SUSE Telco Cloud. Unlike Image-based Provisioning, this workflow keeps the OS image generic and drives all cluster-specific configuration from the management cluster, enabling consistent and fully automated deployments at scale across distributed sites.

Directed Network Provisioning is a fully automated, zero-touch workflow driven by the management cluster. Each bare-metal host is first pre-enrolled in Metal3 from the management cluster by registering its BMC credentials and hardware details. Once the host is racked, connected to the required networks, and powered on, the management cluster takes over: it provisions the OS using an EIB-generated base image via the out-of-band management interface, and deploys the full Kubernetes stack with all telco profiles applied, without any further manual intervention.

The management cluster automates the deployment of the following components on each downstream cluster node:

- SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*) (or `SUSE Linux Micro RT` for Real-Time kernel) as the operating system. Networking, storage, users, and kernel arguments can be customized depending on the use case.
- RKE2 (*Chapter 14, RKE2*) as the Kubernetes distribution. The default CNI plug-in is `Cilium`. Other CNI combinations such as `Cilium+Multus` can be used depending on the use case.
- (Optional) SUSE Storage (*Chapter 15, SUSE Storage*)
- (Optional) SUSE Security (*Chapter 16, SUSE Security*)
- (Optional) `MetalLB` as the load balancer for highly available multi-node clusters.

The following sections describe the different directed network provisioning workflows and some additional features that can be added to the provisioning process:

- *Chapter 50, Prepare downstream cluster image for connected scenarios*
- *Chapter 51, Prepare downstream cluster image for air-gap scenarios*
- *Chapter 52, Downstream cluster provisioning with Directed network provisioning (single-node)*
- *Chapter 53, Downstream cluster provisioning with Directed network provisioning (multi-node)*
- *Chapter 54, Advanced Network Configuration*

- *Chapter 55, Telco features (DPDK, SR-IOV, CPU isolation, huge pages, NUMA, etc.)*
- *Chapter 56, Private registry*
- *Chapter 57, Downstream cluster provisioning in air-gapped scenarios*

Note

The following sections show how to prepare the different scenarios for the directed network provisioning workflow using SUSE Telco Cloud. For examples of the different configurations options for deployment (incl. air-gapped environments, DHCP and DHCP-less networks, private container registries, etc.), see the SUSE Telco Cloud repository (https://github.com/suse-edge/telco-cloud-examples/tree/release-3.7/telco-examples/downstream-clusters).

50 Prepare downstream cluster image for connected scenarios

Edge Image Builder (*Chapter 12, Edge Image Builder*) is used to prepare a modified SLEMicro base image which is provisioned on downstream cluster hosts.

Much of the configuration via Edge Image Builder is possible, but in this guide, we cover the minimal configurations necessary to set up the downstream cluster.

50.1 Prerequisites for connected scenarios

- A container runtime such as Podman (https://podman.io) or Rancher Desktop (https://rancherdesktop.io) is required to run Edge Image Builder.
- The base image will be built using the following guide *Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi* with the profile `Base` (or `Base-RT` for the Real-Time kernel). The process is the same for both architectures (x86-64 and aarch64).

Note

It is required to use a build host with the same architecture of the images being built. In other words, to build an `aarch64` image, it is required to use an `aarch64` build host, and vice-versa for `x86-64` (cross-builds are not supported at this time).

50.2 Image configuration for connected scenarios

When running Edge Image Builder, a directory is mounted from the host, so it is necessary to create a directory structure to store the configuration files used to define the target image.

- `downstream-cluster-config.yaml` is the image definition file, see *Chapter 5, Standalone clusters with Edge Image Builder* for more details.
- The base image folder will contain the output raw image generated following the guide *Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi* with the profile `Base` (or `Base-RT` for the Real-Time kernel) must be copied/moved under the `base-images` folder.

- The `custom/files` directory contains the `performance-settings.sh` and `sriov-auto-filler.sh` files to be copied to the image during the image creation process.
- The `custom/scripts` directory contains scripts to be run on first-boot:
  1. `01-fix-growfs.sh` script is required to resize the OS root partition on deployment
  2. `02-performance.sh` script is optional and can be used to configure the system for performance tuning.
  3. `03-sriov.sh` script is optional and can be used to configure the system for SR-IOV.
- The `network` folder is optional, see *Section 50.2.6, "Additional script for Advanced Network Configuration"* for more details.
- The `os-files` folder contains the necessary configuration files to load, at each node boot, the `sctp` and `vfio-pci` kernel modules with the required options.

```
├── downstream-cluster-config.yaml
├── base-images
│   └── SL-Micro.x86_64-6.2-Base-GM.raw
├── custom
│   ├── files
│   │   ├── performance-settings.sh
│   │   └── sriov-auto-filler.sh
│   └── scripts
│       ├── 01-fix-growfs.sh
│       ├── 02-performance.sh
│       └── 03-sriov.sh
├── network
│   └── configure-network.sh
└── os-files
    └── etc
        ├── modprobe.d
        │   └── vfio-pci-options.conf
        └── modules-load.d
            ├── sctp.conf
            └── vfio-pci.conf
```

50.2.1 Downstream cluster image definition file

The `downstream-cluster-config.yaml` file is the main configuration file for the downstream cluster image. The following is a minimal example for deployment via Metal[3]:

```
apiVersion: 1.3
image:
  imageType: raw
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.2-Base-GM.raw
  outputImageName: eibimage-output-telco.raw
operatingSystem:
  kernelArgs:
    - ignition.platform.id=openstack
  systemd:
    disable:
      - rebootmgr
      - transactional-update.timer
      - transactional-update-cleanup.timer
      - fstrim
      - time-sync.target
  users:
    - username: root
      encryptedPassword: $ROOT_PASSWORD
      sshKeys:
      - $USERKEY1
      createHomeDir: true
  packages:
    packageList:
      - jq
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

Where `$SCC_REGISTRATION_CODE` is the registration code copied from SUSE Customer Center (https://scc.suse.com/), and the package list contains `jq` which is required.

`$ROOT_PASSWORD` is the encrypted password for the root user, which can be useful for test/debugging. It can be generated with the `openssl passwd -6 PASSWORD` command

For the production environments, it is recommended to use the SSH keys that can be added to the users block replacing the `$USERKEY1` with the real SSH keys.

Note

`arch: x86_64` is the architecture of the image. For arm64 architecture, use `arch: aarch64`.

Note `ignition.platform.id=openstack` is mandatory, without this argument SLEMicro configuration via ignition will fail in the Metal[3] automated flow.

For SUSE Linux Micro versions earlier than 6.2, `net.ifnames=1` must be explicitly set as a kernel argument to enable Predictable Network Interface Naming (https://documentation.suse.com/smart/network/html/network-interface-predictable-naming/index.html). This is enforced by default from 6.2 onwards and explicit configuration is not required. This aligns with the `predictableNicNames` configuration in the Management Cluster's Metal[3] Helm chart, which is required for Directed Network Provisioning to function correctly. Consistent interface naming is also critical when SR-IOV is utilized.

50.2.2 Growfs script

Currently, a custom script (`custom/scripts/01-fix-growfs.sh`) is required to grow the file system to match the disk size on first-boot after provisioning. The `01-fix-growfs.sh` script contains the following information:

```
#!/bin/bash
growfs() {
  mnt="$1"
  dev="$(findmnt --fstab --target ${mnt} --evaluate --real --output SOURCE --noheadings)"
  # /dev/sda3 -> /dev/sda, /dev/nvme0n1p3 -> /dev/nvme0n1
  parent_dev="/dev/$(lsblk --nodeps -rno PKNAME "${dev}")"
  # Last number in the device name: /dev/nvme0n1p42 -> 42
  partnum="$(echo "${dev}" | sed 's/^.*[^0-9]\([0-9]\+\)$/\1/')"
  ret=0
  growpart "$parent_dev" "$partnum" || ret=$?
  [ $ret -eq 0 ] || [ $ret -eq 1 ] || exit 1
  /usr/lib/systemd/systemd-growfs "$mnt"
}
growfs /
```

50.2.3 Performance script

The following optional script (`custom/scripts/02-performance.sh`) can be used to configure the system for performance tuning:

```
#!/bin/bash

# create the folder to extract the artifacts there
mkdir -p /opt/performance-settings

# copy the artifacts
```

```
cp performance-settings.sh /opt/performance-settings/
```

The content of `custom/files/performance-settings.sh` is a script that can be used to configure the system for performance tuning and can be downloaded from the following link (https://github.com/suse-edge/telco-cloud-examples/blob/release-3.7/telco-examples/downstream-clusters/dhcp/eib/custom/files/performance-settings.sh).

50.2.4 SR-IOV script

The following optional script (`custom/scripts/03-sriov.sh`) can be used to configure the system for SR-IOV:

```
#!/bin/bash

# create the folder to extract the artifacts there
mkdir -p /opt/sriov
# copy the artifacts
cp sriov-auto-filler.sh /opt/sriov/sriov-auto-filler.sh
```

The content of `custom/files/sriov-auto-filler.sh` is a script that can be used to configure the system for SR-IOV and can be downloaded from the following link (https://github.com/suse-edge/telco-cloud-examples/blob/release-3.7/telco-examples/downstream-clusters/dhcp/eib/custom/files/sriov-auto-filler.sh).

Note

Add your own custom scripts to be executed during the provisioning process using the same approach. For more information, see *Chapter 5, Standalone clusters with Edge Image Builder*.

50.2.5 Additional configuration for Telco workloads

To enable Telco features like `dpdk`, `sr-iov` or `FEC`, additional packages may be required as shown in the following example.

```
apiVersion: 1.3
image:
  imageType: raw
  arch: x86_64
  baseImage: SL-Micro.x86_64-6.2-Base-GM.raw
  outputImageName: eibimage-output-telco.raw
operatingSystem:
```

```
  kernelArgs:
    - ignition.platform.id=openstack
  systemd:
    disable:
      - rebootmgr
      - transactional-update.timer
      - transactional-update-cleanup.timer
      - fstrim
      - time-sync.target
  users:
    - username: root
      encryptedPassword: $ROOT_PASSWORD
      sshKeys:
      - $user1Key1
      createHomeDir: true
  packages:
    packageList:
      - jq            # Non Telco specific - category: testing/utils
      - dpdk          # TelCo Core-Network UserPlane (5G UPF) and RAN UserPlane (gNB: DU
and CU-UP) - category: runtime
      - dpdk-tools    # TelCo Core-Network UserPlane (5G UPF) and RAN UserPlane (gNB: DU
and CU-UP) - category: testing/utils
      - libdpdk-25    # TelCo Core-Network UserPlane (5G UPF) and RAN UserPlane (gNB: DU
and CU-UP) - category: runtime
      - pf-bb-config  # TelCo RAN (gNB: DU) - category: config.
      - open-iscsi    # Non Telco specific (required by SUSE Storage/Longhorn) -
category: runtime
      - tuned         # Non Telco specific (linux tuning) - category: config.
      - cpupower      # Non Telco specific (linux tuning - CPU power related settings) -
category: config.
      - rt-tests      # Non Telco specific (RT scheduler-latency testing) - category:
testing/utils
      - linuxptp      # TelCo RAN; time sync (gNB: DU) - category: runtime
      - synce4l       # TelCo RAN; time sync (gNN: DU) - category: runtime
      - pciutils      # Non Telco specific (provides "lspci" command) - category:
testing/utils
      - numactl       # Non Telco specific (provides "numactl" command) - category:
testing/utils
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

Where `$SCC_REGISTRATION_CODE` is the registration code copied from SUSE Customer Center (https://scc.suse.com/), and the package list contains the minimum packages to be used for the Telco profiles.

Note

`arch: x86_64` is the architecture of the image. For arm64 architecture, use `arch: aarch64`.

50.2.6 Additional script for Advanced Network Configuration

If you need to configure static IPs or more advanced networking scenarios as described in *Chapter 54, Advanced Network Configuration*, the following additional configuration is required.

In the `network` folder, create the following `configure-network.sh` file - this consumes configuration drive data on first-boot, and configures the host networking using the NM Configurator tool (https://github.com/suse-edge/nm-configurator).

```
#!/bin/bash

set -eux

# Attempt to statically configure a NIC in the case where we find a network_data.json
# In a configuration drive

CONFIG_DRIVE=$(blkid --label config-2 || true)
if [ -z "${CONFIG_DRIVE}" ]; then
  echo "No config-2 device found, skipping network configuration"
  exit 0
fi

mount -o ro $CONFIG_DRIVE /mnt

META_DATA_FILE="/mnt/openstack/latest/meta_data.json"
if [ ! -f "${META_DATA_FILE}" ]; then
  umount /mnt
  echo "No meta_data.json found, skipping hostname configuration"
  exit 0
fi

DESIRED_HOSTNAME=$(cat /mnt/openstack/latest/meta_data.json | tr ',{}' '\n' | grep
 '"metal3-name"' | sed 's/.*\"metal3-name\": \"\(.*\)\"/\1/')
echo "${DESIRED_HOSTNAME}" > /etc/hostname

NETWORK_DATA_FILE="/mnt/openstack/latest/network_data.json"

if [ ! -f "${NETWORK_DATA_FILE}" ]; then
  umount /mnt
  echo "No network_data.json found, skipping network configuration"
  exit 0
fi

mkdir -p /tmp/nmc/{desired,generated}
cp ${NETWORK_DATA_FILE} /tmp/nmc/desired/_all.yaml
umount /mnt
```

```
./nmc generate --config-dir /tmp/nmc/desired --output-dir /tmp/nmc/generated
./nmc apply --config-dir /tmp/nmc/generated
```

50.2.7 Telco required kernel modules

To ensure that the kernel modules required for telecom workloads are loaded on each node boot, the following configuration files must be added to the `os-files` EIB folder:

50.2.7.1 sctp.conf

To be placed in the `os-files/etc/modules-load.d/` path, instructing the systemd-modules-load.service (https://www.freedesktop.org/software/systemd/man/latest/modules-load.d.html) to load the `sctp` kernel module at node boot.

```
$ cat os-files/etc/modules-load.d/sctp.conf
# Request "systemd-modules-load.service" to load sctp module at boot time
sctp
```

50.2.7.2 vfio-pci.conf

To be placed in the `os-files/etc/modules-load.d/` path, instructing the systemd-modules-load.service (https://www.freedesktop.org/software/systemd/man/latest/modules-load.d.html) to load the `vfio-pci` kernel module at node boot.

```
$ cat os-files/etc/modules-load.d/vfio-pci.conf
# Request "systemd-modules-load.service" to load vfio-pci module at boot time
# NOTE: The (SRIOV & DPDK related) required load-module options settings are defined in
 related "/etc/modprobe.d/vfio-pci-options.conf" file
vfio_pci
```

50.2.7.3 vfio-pci-options.conf

To be placed in the `os-files/etc/modprobe.d/` path, setting the required `vfio-pci` kernel module parameters to be applied when the kernel module is loaded.

```
$ cat os-files/etc/modprobe.d/vfio-pci-options.conf
# Enable SR-IOV and disable idle D3 state for vfio-pci driver
options vfio_pci enable_sriov=1 disable_idle_d3=1
```

50.3 Image creation

Once the directory structure is prepared following the previous sections, run the following command to build the image:

```
podman run --rm --privileged -it -v $PWD:/eib \
 registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
 build --definition-file downstream-cluster-config.yaml
```

This creates the output RAW image file named `eibimage-output-telco.raw`, based on the definition described above.

The output image must then be made available via a webserver, either the media-server container enabled via the Management Cluster Documentation (*Note*) or some other locally accessible server. In the examples below, we refer to this server as `imagecache.local:8080`

51 Prepare downstream cluster image for air-gap scenarios

Edge Image Builder (*Chapter 12, Edge Image Builder*) is used to prepare a modified SLEMicro base image which is provisioned on downstream cluster hosts.

Much of the configuration is possible with Edge Image Builder, but in this guide, we cover the minimal configurations necessary to set up the downstream cluster for air-gap scenarios.

51.1 Prerequisites for air-gap scenarios

- A container runtime such as Podman (https://podman.io) or Rancher Desktop (https://rancherdesktop.io) is required to run Edge Image Builder.
- The base image will be built using the following guide *Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi* with the profile `Base` (or `Base-RT` for the Real-Time kernel). The process is the same for both architectures (x86-64 and aarch64).
- If you want to use SR-IOV or any other workload which require a container image, a local private registry must be deployed and already configured (with/without TLS and/or authentication). This registry will be used to store the images and the helm chart OCI images.

Note

It is required to use a build host with the same architecture of the images being built. In other words, to build an `aarch64` image, it is required to use an `aarch64` build host, and vice-versa for `x86-64` (cross-builds are not supported at this time).

51.2 Image configuration for air-gap scenarios

When running Edge Image Builder, a directory is mounted from the host, so it is necessary to create a directory structure to store the configuration files used to define the target image.

- `downstream-cluster-airgap-config.yaml` is the image definition file, see *Chapter 5, Standalone clusters with Edge Image Builder* for more details.
- The base image folder will contain the output raw image generated following the guide *Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi* with the profile `Base` (or `Base-RT` for the Real-Time kernel) must be copied/moved under the `base-images` folder.
- The `network` folder is optional, see *Section 50.2.6, "Additional script for Advanced Network Configuration"* for more details.
- The `custom/scripts` directory contains scripts to be run on first-boot:
  1. `01-fix-growfs.sh` script is required to resize the OS root partition on deployment.
  2. `02-airgap.sh` script is required to copy the images to the right place during the image creation process for air-gapped environments.
  3. `03-performance.sh` script is optional and can be used to configure the system for performance tuning.
  4. `04-sriov.sh` script is optional and can be used to configure the system for SR-IOV.
- The `custom/files` directory contains the `rke2` and the `cni` images to be copied to the image during the image creation process. Also, the optional `performance-settings.sh` and `sriov-auto-filler.sh` files can be included.
- The `os-files` folder contains the necessary configuration files to load, at each node boot, the `sctp` and `vfio-pci` kernel modules with the required options.

```
├── downstream-cluster-airgap-config.yaml
├── base-images
│   └── SL-Micro.x86_64-6.2-Base-GM.raw
├── custom
│   ├── files
│   │   ├── install.sh
│   │   ├── rke2-images-cilium.linux-amd64.tar.zst
│   │   ├── rke2-images-core.linux-amd64.tar.zst
│   │   ├── rke2-images-multus.linux-amd64.tar.zst
│   │   ├── rke2-images.linux-amd64.tar.zst
│   │   ├── rke2.linux-amd64.tar.zst
│   │   ├── sha256sum-amd64.txt
│   │   ├── performance-settings.sh
│   │   └── sriov-auto-filler.sh
│   └── scripts
```

```
│       ├── 01-fix-growfs.sh
│       ├── 02-airgap.sh
│       ├── 03-performance.sh
│       └── 04-sriov.sh
├── network
│   └── configure-network.sh
└── os-files
    └── etc
        ├── modprobe.d
        │   └── vfio-pci-options.conf
        └── modules-load.d
            ├── sctp.conf
            └── vfio-pci.conf
```

51.2.1 Downstream cluster image definition file

The `downstream-cluster-airgap-config.yaml` file is the main configuration file for the downstream cluster image and the content has been described in the previous section (*Section 50.2.5, "Additional configuration for Telco workloads"*).

51.2.2 Growfs script

Currently, a custom script (`custom/scripts/01-fix-growfs.sh`) is required to grow the file system to match the disk size on first-boot after provisioning. The `01-fix-growfs.sh` script contains the following information:

```
#!/bin/bash
growfs() {
  mnt="$1"
  dev="$(findmnt --fstab --target ${mnt} --evaluate --real --output SOURCE --noheadings)"
  # /dev/sda3 -> /dev/sda, /dev/nvme0n1p3 -> /dev/nvme0n1
  parent_dev="/dev/$(lsblk --nodeps -rno PKNAME "${dev}")"
  # Last number in the device name: /dev/nvme0n1p42 -> 42
  partnum="$(echo "${dev}" | sed 's/^.*[^0-9]\([0-9]\+\)$/\1/')"
  ret=0
  growpart "$parent_dev" "$partnum" || ret=$?
  [ $ret -eq 0 ] || [ $ret -eq 1 ] || exit 1
  /usr/lib/systemd/systemd-growfs "$mnt"
}
growfs /
```

51.2.3 Air-gap script

The following script (`custom/scripts/02-airgap.sh`) is required to copy the images to the right place during the image creation process:

```
#!/bin/bash

# create the folder to extract the artifacts there
mkdir -p /opt/rke2-artifacts
mkdir -p /var/lib/rancher/rke2/agent/images

# copy the artifacts
cp install.sh /opt/
cp rke2-images*.tar.zst rke2.linux-amd64.tar.gz sha256sum-amd64.txt /opt/rke2-artifacts/
```

51.2.4 Performance script

The following optional script (`custom/scripts/03-performance.sh`) can be used to configure the system for performance tuning:

```
#!/bin/bash

# create the folder to extract the artifacts there
mkdir -p /opt/performance-settings

# copy the artifacts
cp performance-settings.sh /opt/performance-settings/
```

The content of `custom/files/performance-settings.sh` is a script that can be used to configure the system for performance tuning and can be downloaded from the following link (https://github.com/suse-edge/telco-cloud-examples/blob/release-3.7/telco-examples/downstream-clusters/dhcp/eib/custom/files/performance-settings.sh).

51.2.5 SR-IOV script

The following optional script (`custom/scripts/04-sriov.sh`) can be used to configure the system for SR-IOV:

```
#!/bin/bash
```

```
# create the folder to extract the artifacts there
mkdir -p /opt/sriov
# copy the artifacts
cp sriov-auto-filler.sh /opt/sriov/sriov-auto-filler.sh
```

The content of `custom/files/sriov-auto-filler.sh` is a script that can be used to configure the system for SR-IOV and can be downloaded from the following link (https://github.com/suse-edge/telco-cloud-examples/blob/release-3.7/telco-examples/downstream-clusters/dhcp/eib/custom/files/sriov-auto-filler.sh).

51.2.6 Telco required kernel modules

As described for the non-airgap scenario in section (*Section 50.2.7, "Telco required kernel modules"*).

51.2.7 Preparing the air-gap artifacts

The following steps are required to prepare the air-gap artifacts using the release container image in order to populate a registry with the specific version artifacts. It handles RKE2 tarball generation, Helm chart OCI mirroring, and container image mirroring in a single command — no separate scripts are needed.

1. If your private registry requires authentication, create a registry auth file with base64-encoded credentials:

```
$ echo -n "$(echo -n 'myusername' | base64 -w 0):$(echo -n 'mypassword' | base64 -w
 0)" > registry-auth.txt
```

2. If you use a Rancher Apps chart repository (required for Longhorn and Rancher-sourced charts), create a Rancher Apps auth file:

```
$ echo -n "$(echo -n 'myusername@apps.rancher.io' | base64 -w 0):$(echo -n
 'mypassword' | base64 -w 0)" > rancher-apps-auth.txt
```

3. (Optional) If you want to mirror SUSE Private Registry artifacts (Harbor charts/images), create a SUSE Private Registry auth file using your `SCC mirroring credentials` retrieved following the SUSE Private Registry documentation (https://documentation.suse.com/cloudnative/suse-private-registry/html/private-registry/pr-introduction.html):

```
$ echo -n "$(echo -n 'SUSE_REGISTRY_USERNAME' | base64 -w 0):$(echo -n
 'SUSE_REGISTRY_PASSWORD' | base64 -w 0)" > suse-private-registry-auth.txt
```

4. Run the `mirror` command using the release container image to populate your private registry with all required artifacts for a specific release version (RKE2 images, Helm chart OCI images, and container images). Place any auth files and certificates in the current directory so they are accessible inside the container via the `-v ./:/opt:z` bind mount.
   Without SUSE Private Registry (Harbor charts/images will be skipped):

   ```
   $ podman run --rm \
     -v ./:/opt:z \
     registry.suse.com/edge/3.7/release-manifest:3.7.0 \
     mirror \
     -o /opt/output \
     -a /opt/registry-auth.txt \
     -c /opt/cert.pem \
     -r ${REGISTRY_IP}:5000 \
     --rancher-apps-authfile /opt/rancher-apps-auth.txt \
     --debug
   ```

   where `${REGISTRY_IP}` is the IP address of the registry instance to populate.
   With SUSE Private Registry:

   ```
   $ podman run --rm \
     -v ./:/opt:z \
     registry.suse.com/edge/3.7/release-manifest:3.7.0 \
     mirror \
     -o /opt/output \
     -a /opt/registry-auth.txt \
     -c /opt/cert.pem \
     -r ${MGMT_CLUSTER_REGISTRY_IP}:5000 \
     --rancher-apps-authfile /opt/rancher-apps-auth.txt \
     --suse-private-registry-authfile /opt/suse-private-registry-auth.txt \
     --debug
   ```

   where `${MGMT_CLUSTER_REGISTRY_IP}` is the reserved static IP address given to the SUSE Private Registry instance deployed in the Management cluster, as described in the previous section (*Section 30.3, "Modifications in the kubernetes folder"*).

5. Copy the generated RKE2 artifacts from the output directory (`/opt/output` in the example) to the `custom/files` folder to be consumed by EIB for Downstream clusters during the build process:

   ```
   $ cp output/rke2-images*.tar.zst custom/files/
   $ cp output/rke2.linux-amd64.tar.gz custom/files/
   $ cp output/sha256sum-amd64.txt custom/files/
   ```

Note

The release container image already bundles `/release_manifest.yaml` and `/release_images.yaml` internally, so no additional manifest files need to be provided. For full flag reference and advanced usage, see the seactl documentation (https://github.com/suse-edge/seactl/blob/main/README.md).

51.3 Image creation for air-gap scenarios

Once the directory structure is prepared following the previous sections, run the following command to build the image:

```
podman run --rm --privileged -it -v $PWD:/eib \
 registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
 build --definition-file downstream-cluster-airgap-config.yaml
```

This creates the output RAW image file named `eibimage-output-telco.raw`, based on the definition described above.

The output image must then be made available via a webserver, either the media-server container enabled via the Management Cluster Documentation (*Note*) or some other locally accessible server. In the examples below, we refer to this server as `imagecache.local:8080`.

52 Downstream cluster provisioning with Directed network provisioning (single-node)

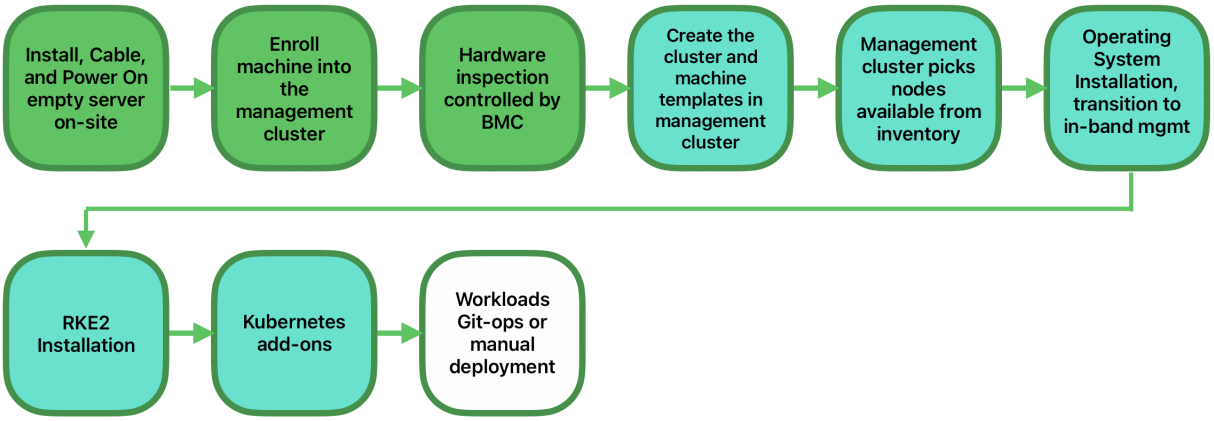
This section describes the workflow used to automate the provisioning of a single-node downstream cluster using directed network provisioning. This is the simplest way to automate the provisioning of a downstream cluster.

Requirements

- The image generated using `EIB`, as described in the previous section (*Chapter 50, Prepare downstream cluster image for connected scenarios*), with the minimal configuration to set up the downstream cluster has to be located in the management cluster exactly on the path you configured on this section (*Note*).
- The management server created and available to be used on the following sections. For more information, refer to the Management Cluster section *Part V, "Setting up the management cluster"*.

Workflow

The following diagram shows the workflow used to automate the provisioning of a single-node downstream cluster using directed network provisioning:



There are two different steps to automate the provisioning of a single-node downstream cluster using directed network provisioning:

1. Enroll the bare-metal host to make it available for the provisioning process.
2. Provision the bare-metal host to install and configure the operating system and the Kubernetes cluster.

Enroll the bare-metal host

The first step is to enroll the new bare-metal host in the management cluster to make it available to be provisioned. To do that, the following file (`bmh-example.yaml`) has to be created in the management cluster, to specify the `BMC` credentials to be used and the `BaremetalHost` object to be enrolled:

```
apiVersion: v1
kind: Secret
metadata:
  name: example-demo-credentials
type: Opaque
data:
  username: ${BMC_USERNAME}
  password: ${BMC_PASSWORD}
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: example-demo
  labels:
    cluster-role: control-plane
spec:
  architecture: x86_64
  online: true
  bootMACAddress: ${BMC_MAC}
  rootDeviceHints:
    deviceName: /dev/nvme0n1
  bmc:
    address: ${BMC_ADDRESS}
    disableCertificateVerification: true
    credentialsName: example-demo-credentials
```

where:

- `${BMC_USERNAME}` — The user name for the `BMC` of the new bare-metal host.
- `${BMC_PASSWORD}` — The password for the `BMC` of the new bare-metal host.
- `${BMC_MAC}` — The `MAC` address of the new bare-metal host to be used.
- `${BMC_ADDRESS}` — The `URL` for the bare-metal host `BMC` (for example, `redfish-virtualmedia://192.168.200.75/redfish/v1/Systems/1/`). To learn more about the different options available depending on your hardware provider, check the following link (https://github.com/metal3-io/baremetal-operator/blob/main/docs/api.md).

Note

- Architecture must be either `x86_64` or `aarch64`, depending on the architecture of the bare-metal host to be enrolled.
- If no network configuration for the host has been specified, either at image build time or through the `BareMetalHost` definition, an autoconfiguration mechanism (DHCP, DHCPv6, SLAAC) will be used. For more details or complex configurations, check the *Chapter 54, Advanced Network Configuration*.

Once the file is created, the following command has to be executed in the management cluster to start enrolling the new bare-metal host in the management cluster:

```
$ kubectl apply -f bmh-example.yaml
```

The new bare-metal host object will be enrolled, changing its state from registering to inspecting and available. The changes can be checked using the following command:

```
$ kubectl get bmh
```

Note

The `BaremetalHost` object is in the `registering` state until the `BMC` credentials are validated. Once the credentials are validated, the `BaremetalHost` object changes its state to `inspecting`, and this step could take some time depending on the hardware (up to 20 minutes). During the inspecting phase, the hardware information is retrieved and the Kubernetes object is updated. Check the information using the following command: `kubectl get bmh -o yaml`.

Provision step

Once the bare-metal host is enrolled and available, the next step is to provision the bare-metal host to install and configure the operating system and the Kubernetes cluster. To do that, the following file (`capi-provisioning-example.yaml`) has to be created in the management-cluster with the following information (the `capi-provisioning-example.yaml` can be generated by joining the following blocks).

Note

Only values between `$\{…\}` must be replaced with the real values.

The following block is the cluster definition, where the networking can be configured using the `pods` and the `services` blocks. Also, it contains the references to the control plane and the infrastructure (using the `Metal3` provider) objects to be used.

```
apiVersion: cluster.x-k8s.io/v1beta2
kind: Cluster
metadata:
  name: single-node-cluster
  namespace: default
  labels:
    cluster-api.cattle.io/rancher-auto-import: "true"
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
        - fd00:bad:cafe::/48
    services:
      cidrBlocks:
        - 10.96.0.0/12
        - fd00:bad:bad:cafe::/112
  controlPlaneRef:
    apiGroup: controlplane.cluster.x-k8s.io
    kind: RKE2ControlPlane
    name: single-node-cluster
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3Cluster
    name: single-node-cluster
```

Note

- Both single-stack and dual-stack deployments are possible, remove the IPv6 CIDRs from the above definition for an IPv4 only cluster.
- Single-stack IPv6 deployments are in tech preview status and not yet officially supported.
- Adding the label `cluster-api.cattle.io/rancher-auto-import: "true"` to the `cluster.x-k8s.io` objects will import the cluster into Rancher (by creating a corresponding `clusters.management.cattle.io` object). See the Cluster API documentation (https://documentation.suse.com/cloudnative/cluster-api/latest/en/tutorials/first-cluster.html#_mark_namespace_for_auto_import) for more information.

The `Metal3Cluster` object specifies the control-plane endpoint (replacing the `${DOWNSTREAM_CONTROL_PLANE_IPV4}`) to be configured and the `cloudProviderEnabled` flag set to `false` because a bare-metal node is used.

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3Cluster
metadata:
  name: single-node-cluster
  namespace: default
spec:
  controlPlaneEndpoint:
    host: ${DOWNSTREAM_CONTROL_PLANE_IPV4}
    port: 6443
  cloudProviderEnabled: false
```

The `RKE2ControlPlane` object specifies the control-plane configuration to be used and the `Metal3MachineTemplate` object specifies the control-plane image to be used. Also, it contains the information about the number of replicas required (in this case, one) and the `CNI` plug-in to use (in this case, `Cilium`). The agentConfig block contains the `Ignition` format to be used and the `additionalUserData` to be used to configure the `RKE2` node with information like a systemd unit service named `rke2-preinstall.service` to replace automatically the `PROVIDERID` and `node-name` during the provisioning process using the Ironic information. The configuration required for the `RKE2` (Traefik based) ingress controller is also included, together the `MetalLB` custom resources (`IPAddressPool` and `L2Advertisement`) instance manifests required to externally expose the Ingress service through the `${INGRESS_IPV4}` and `${INGRESS_IPV6}` Virtual IPs. The last block of information contains the Kubernetes version to be used. `${RKE2_VERSION}` is the version of `RKE2` to be used replacing this value (for example, `v1.36.3+rke2r1`).

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
  annotations: {
    rke2.controlplane.cluster.x-k8s.io/load-balancer-exclusion: "true"
  }
spec:
  machineTemplate:
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
```

```
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  serverConfig:
    cni: cilium
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
          - name: rke2-preinstall.service
            enabled: true
            contents: |
              [Unit]
              Description=rke2-preinstall
              Wants=network-online.target
              Before=rke2-install.service
              ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
              [Service]
              Type=oneshot
              User=root
              ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
              # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
              ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
              ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStartPost=/bin/sh -c "umount /mnt"
              [Install]
              WantedBy=multi-user.target
        storage:
          directories:
          - path: /var/lib/rancher/rke2/server/manifests
            overwrite: true
          files:
          - path: /var/lib/rancher/rke2/server/manifests/rke2-traefik-config.yaml
            overwrite: true
```

```
          contents:
            inline: |
              apiVersion: helm.cattle.io/v1
              kind: HelmChartConfig
              metadata:
                name: rke2-traefik
                namespace: kube-system
              spec:
                valuesContent: |-
                  ingressClass:
                    isDefaultClass: true
                  ports:
                    web:
                      hostPort: null    # disallow hostPort
                      exposedPort: 80
                    websecure:
                      hostPort: null    # disallow hostPort
                      exposedPort: 443
                  service:
                    enabled: true
                    spec:
                      type: LoadBalancer
                      externalTrafficPolicy: Local
                      allocateLoadBalancerNodePorts: false
          mode: 0644
          user:
            name: root
          group:
            name: root
        - path: /var/lib/rancher/rke2/server/manifests/metallb-traefik-cr.yaml
          overwrite: true
          contents:
            inline: |
              apiVersion: metallb.io/v1beta1
              kind: IPAddressPool
              metadata:
                name: ingress-ippool
                namespace: metallb-system
              spec:
                addresses:
                  - ${INGRESS_IPV4}/32
                  - ${INGRESS_IPV6}/128
                serviceAllocation:
                  priority: 100
                  serviceSelectors:
                  - matchExpressions:
```

```
                      - {key: app.kubernetes.io/name, operator: In, values: [rke2-
traefik]} # assign to rke2-traefik LB svc
                ---
                apiVersion: metallb.io/v1beta1
                kind: L2Advertisement
                metadata:
                  name: ingress-l2-adv
                  namespace: metallb-system
                spec:
                  ipAddressPools:
                    - ingress-ippool
            mode: 0644
            user:
              name: root
            group:
              name: root
    kubelet:
      extraArgs:
      - provider-id=metal3://PROVIDERID
    nodeName: "localhost.localdomain"
```

The `Metal3MachineTemplate` object specifies the following information:

- The `dataTemplate` to be used as a reference to the template.
- The `hostSelector` to be used matching with the label created during the enrollment process.
- The `image` to be used as a reference to the image generated using `EIB` on the previous section (*Chapter 50, Prepare downstream cluster image for connected scenarios*), and the `checksum` and `checksumType` to be used to validate the image.

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: single-node-cluster-controlplane
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: single-node-cluster-controlplane-template
      hostSelector:
        matchLabels:
          cluster-role: control-plane
      image:
        checksum: http://imagecache.local:8080/eibimage-output-telco.raw.sha256
        checksumType: sha256
```

```
        diskFormat: raw
        url: http://imagecache.local:8080/eibimage-output-telco.raw
```

The `Metal3DataTemplate` object specifies the `metaData` for the downstream cluster.

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3DataTemplate
metadata:
  name: single-node-cluster-controlplane-template
  namespace: default
spec:
  clusterName: single-node-cluster
  metaData:
    objectNames:
    - key: name
      object: machine
    - key: local-hostname
      object: machine
    - key: local_hostname
      object: machine
```

Once the file is created by joining the previous blocks, the following command must be executed in the management cluster to start provisioning the new bare-metal host:

```
$ kubectl apply -f capi-provisioning-example.yaml
```

53 Downstream cluster provisioning with Directed network provisioning (multi-node)

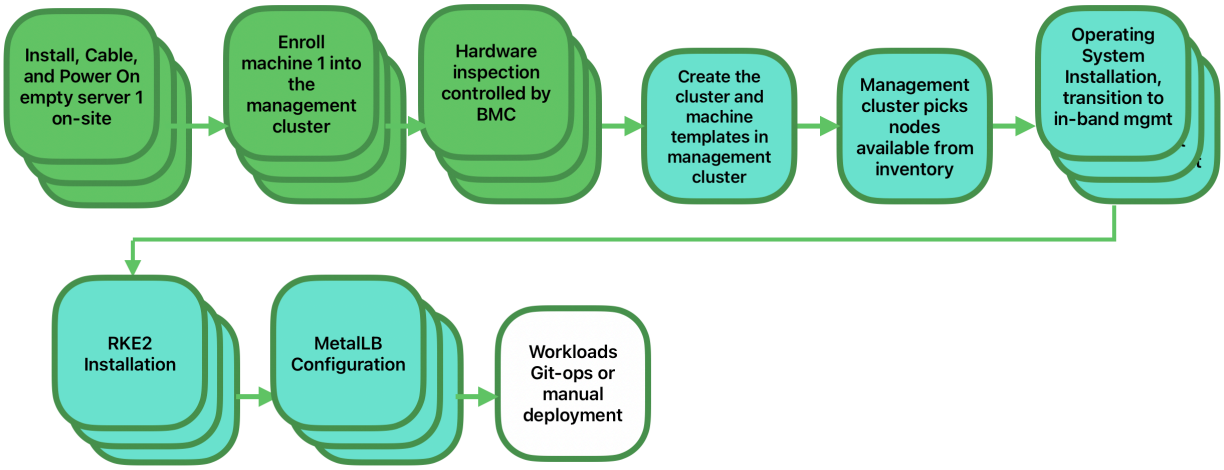
This section describes the workflow used to automate the provisioning of a multi-node downstream cluster using directed network provisioning and `MetalLB` as a load-balancer strategy. This is the simplest way to automate the provisioning of a downstream cluster. The following diagram shows the workflow used to automate the provisioning of a multi-node downstream cluster using directed network provisioning and `MetalLB`.

Requirements

- The image generated using `EIB`, as described in the previous section (*Chapter 50, Prepare downstream cluster image for connected scenarios*), with the minimal configuration to set up the downstream cluster has to be located in the management cluster exactly on the path you configured on this section (*Note*).
- The management server created and available to be used on the following sections. For more information, refer to the Management Cluster section: *Part V, "Setting up the management cluster"*.

Workflow

The following diagram shows the workflow used to automate the provisioning of a multi-node downstream cluster using directed network provisioning:



1. Enroll the three bare-metal hosts to make them available for the provisioning process.
2. Provision the three bare-metal hosts to install and configure the operating system and the Kubernetes cluster using `MetalLB`.

Enroll the bare-metal hosts

The first step is to enroll the three bare-metal hosts in the management cluster to make them available to be provisioned. To do that, the following files (`bmh-example-node1.yaml`, `bmh-example-node2.yaml` and `bmh-example-node3.yaml`) must be created in the management cluster, to specify the `BMC` credentials to be used and the `BaremetalHost` object to be enrolled in the management cluster.

Note

- Only the values between `$\{…\}` have to be replaced with the real values.
- We will walk you through the process for only one host. The same steps apply to the other two nodes.

```
apiVersion: v1
kind: Secret
metadata:
  name: node1-example-credentials
type: Opaque
data:
  username: ${BMC_NODE1_USERNAME}
  password: ${BMC_NODE1_PASSWORD}
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: node1-example
  labels:
    cluster-role: control-plane
spec:
  architecture: x86_64
  online: true
  bootMACAddress: ${BMC_NODE1_MAC}
  bmc:
    address: ${BMC_NODE1_ADDRESS}
    disableCertificateVerification: true
    credentialsName: node1-example-credentials
```

Where:

- `${BMC_NODE1_USERNAME}` — The username for the BMC of the first bare-metal host.
- `${BMC_NODE1_PASSWORD}` — The password for the BMC of the first bare-metal host.

- `${BMC_NODE1_MAC}` — The MAC address of the first bare-metal host to be used.
- `${BMC_NODE1_ADDRESS}` — The URL for the first bare-metal host BMC (for example, `redfish-virtualmedia://192.168.200.75/redfish/v1/Systems/1/`). The host part of the URL can be an IP address (v4 or v6) or a domain name, where the existing infrastructure allows. To learn more about the different options available depending on your hardware provider, check the following link (https://github.com/metal3-io/baremetal-operator/blob/main/docs/api.md).

Note

- If no network configuration for the host has been specified, either at image build time or through the `BareMetalHost` definition, an autoconfiguration mechanism (DHCP, DHCPv6, SLAAC) will be used. For more details or complex configurations, check the *Chapter 54, Advanced Network Configuration*.
- Single-stack IPv6 clusters require a dual-stack management cluster.
- Architecture must be either `x86_64` or `aarch64`, depending on the architecture of the bare-metal host to be enrolled.
- All modern servers come with a dual-stack capable BMC, however IPv6 support (and possibly the option of using hostnames for the VirtualMedia capability) should be verified before use in production in a dual-stack environment.

Once the file is created, the following command must be executed in the management cluster to start enrolling the bare-metal hosts in the management cluster:

```
$ kubectl apply -f bmh-example-node1.yaml
$ kubectl apply -f bmh-example-node2.yaml
$ kubectl apply -f bmh-example-node3.yaml
```

The new bare-metal host objects are enrolled, changing their state from registering to inspecting and available. The changes can be checked using the following command:

```
$ kubectl get bmh -o wide
```

Note

The `BaremetalHost` object is in the `registering` state until the `BMC` credentials are validated. Once the credentials are validated, the `BaremetalHost` object changes its state to `inspecting`, and this step could take some time depending on the hardware (up to 20 minutes). During the inspecting phase, the hardware information is retrieved and the Kubernetes object is updated. Check the information using the following command: `kubectl get bmh -o yaml`.

Provision step

Once the three bare-metal hosts are enrolled and available, the next step is to provision the bare-metal hosts to install and configure the operating system and the Kubernetes cluster, creating a load balancer to manage them. To do that, the following file (`capi-provisioning-example.yaml`) must be created in the management cluster with the following information (the `capi-provisioning-example.yaml can be generated by joining the following blocks).

Note

- Only values between `$\{…\}` must be replaced with the real values.
- The `VIP` address is a reserved IP address that is not assigned to any node and is used to configure the load balancer. In a dual-stack cluster, both an IPv4 and IPv6 can be specified, but in the following examples priority will be given to the IPv4 address.

Below is the cluster definition, where the cluster network can be configured using the `pods` and the `services` blocks. Also, it contains the references to the control plane and the infrastructure (using the `Metal3` provider) objects to be used.

```
apiVersion: cluster.x-k8s.io/v1beta2
kind: Cluster
metadata:
  name: multinode-cluster
  namespace: default
  labels:
    cluster-api.cattle.io/rancher-auto-import: "true"
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
```

```
        - fd00:1234:4321::/48
    services:
      cidrBlocks:
        - 10.96.0.0/12
        - fd00:5678:8765:4321::/112
  controlPlaneRef:
    apiGroup: controlplane.cluster.x-k8s.io
    kind: RKE2ControlPlane
    name: multinode-cluster
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3Cluster
    name: multinode-cluster
```

Note

- Both single-stack and dual-stack deployments are possible, remove the IPv6 CIDRs and IPv6 VIP addresses (in the subsequent sections) for an IPv4 only cluster.
- Adding the label `cluster-api.cattle.io/rancher-auto-import: "true"` to the `cluster.x-k8s.io` objects will import the cluster into Rancher (by creating a corresponding `clusters.management.cattle.io` object). See the Cluster API documentation (https://documentation.suse.com/cloudnative/cluster-api/latest/en/tutorials/first-cluster.html#_mark_namespace_for_auto_import) for more information.

The `Metal3Cluster` object specifies the control-plane endpoint that uses the `VIP` address already reserved (replacing the `${EDGE_VIP_ADDRESS_IPV4}`) to be configured and the `cloudProviderEnabled` flag set to `false` because the three bare-metal nodes are used.

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3Cluster
metadata:
  name: multinode-cluster
  namespace: default
spec:
  controlPlaneEndpoint:
    host: ${EDGE_VIP_ADDRESS_IPV4}
    port: 6443
  cloudProviderEnabled: false
```

The `RKE2ControlPlane` object specifies the control-plane configuration to be used, and the `Metal3MachineTemplate` object specifies the control-plane image to be used.

- A load balancer exclusion annotation that informs external load balancers like MetalLB that a node is going to be drained during lifecycle operations like upgrades of downstream clusters. For details see: *Section 60.1, "Load Balancer Exclusion"*
- The number of replicas to be used (in this case, three).
- The advertisement mode to be used by the Load Balancer (`address` uses the L2 implementation), as well as the address to be used (replacing the `${EDGE_VIP_ADDRESS}` with the `VIP` address).
- The `serverConfig` with the `CNI` plug-in to be used (in this case, `Cilium`), and the additional `VIP` address(es) and name(s) to be listed under `tlsSan`.
- The agentConfig block contains the `Ignition` format to be used and the `additionalUserData` to be used to configure the `RKE2` node with information like:
  - The systemd service named `rke2-preinstall.service` to replace automatically the `PROVIDERID` and `node-name` during the provisioning process using the Ironic information plus adding the `metal3.io/uuid` label to Node objects with the `BareMetalHost` UUID.
  - The `storage` block which contains the required manifests to install the `MetalLB` and the `endpoint-copier-operator` Helm chats.
  - The `metalLB` custom resource file including the `IPaddressPool` and the `L2Advertisement` used to externally expose the API server for this download cluster through the configured `VIP` address (replacing `${EDGE_VIP_ADDRESS_IPV4}` with that selected `VIP` address).
  - The `endpoint-svc.yaml` file to be used to configure the `kubernetes-vip` service to be exposed by `MetalLB` through the configured `VIP` address.
  - The `rke2-traefik-config.yaml` file providing the deployment configuration for the `RKE2` built-in `rke2-traefik` Helm chart .
  - The `metallb-traefik-cr.yaml` file containing the `MetalLB` custom resources (`IPAddressPool` and `L2Advertisement`) instance manifests required to externally expose the Ingress service through the `${INGRESS_IPV4}` and `${INGRESS_IPV6}` Virtual IPs
- The last block of information contains the Kubernetes version to be used. The `${RKE2_VERSION}` is the version of `RKE2` to be used replacing this value (for example, `v1.36.3+rke2r1`).

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: multinode-cluster
```

```
  namespace: default
  annotations: {
    rke2.controlplane.cluster.x-k8s.io/load-balancer-exclusion: "true"
  }
spec:
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: multinode-cluster-controlplane
  replicas: 3
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  registrationMethod: "control-plane-endpoint"
  registrationAddress: ${EDGE_VIP_ADDRESS}
  serverConfig:
    cni: cilium
    tlsSan:
      - ${EDGE_VIP_ADDRESS_IPV4}
      - ${EDGE_VIP_ADDRESS_IPV6}
      - https://${EDGE_VIP_ADDRESS_IPV4}.sslip.io
      - https://${EDGE_VIP_ADDRESS_IPV6}.sslip.io
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
          - name: rke2-preinstall.service
            enabled: true
            contents: |
              [Unit]
              Description=rke2-preinstall
              Wants=network-online.target
              Before=rke2-install.service
              ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
              [Service]
              Type=oneshot
              User=root
              ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
              # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
```

```
              ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
              ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStartPost=/bin/sh -c "umount /mnt"
              [Install]
              WantedBy=multi-user.target
        storage:
          directories:
          - path: /var/lib/rancher/rke2/server/manifests
            overwrite: true
          files:
            # https://docs.rke2.io/networking/multus_sriov#using-multus-with-cilium
            - path: /var/lib/rancher/rke2/server/manifests/rke2-cilium-config.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChartConfig
                  metadata:
                    name: rke2-cilium
                    namespace: kube-system
                  spec:
                    valuesContent: |-
                      cni:
                        exclusive: false
              mode: 0644
              user:
                name: root
              group:
                name: root
            - path: /var/lib/rancher/rke2/server/manifests/endpoint-copier-operator.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
                  metadata:
                    name: endpoint-copier-operator
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/endpoint-copier-operator
                    targetNamespace: endpoint-copier-operator
```

```
                version: 307.0.1+up0.3.0
                createNamespace: true
        - path: /var/lib/rancher/rke2/server/manifests/metallb.yaml
          overwrite: true
          contents:
            inline: |
              apiVersion: helm.cattle.io/v1
              kind: HelmChart
              metadata:
                name: metallb
                namespace: kube-system
              spec:
                chart: oci://registry.suse.com/edge/charts/metallb
                targetNamespace: metallb-system
                version: 307.0.3+up0.16.1
                createNamespace: true

        - path: /var/lib/rancher/rke2/server/manifests/metallb-cr.yaml
          overwrite: true
          contents:
            inline: |
              apiVersion: metallb.io/v1beta1
              kind: IPAddressPool
              metadata:
                name: kubernetes-vip-ip-pool
                namespace: metallb-system
              spec:
                addresses:
                  - ${EDGE_VIP_ADDRESS_IPV4}/32
                  - ${EDGE_VIP_ADDRESS_IPV6}/128
                serviceAllocation:
                  priority: 100
                  namespaces:
                    - default
                  serviceSelectors:
                    - matchExpressions:
                      - {key: "serviceType", operator: In, values: [kubernetes-vip]}
              ---
              apiVersion: metallb.io/v1beta1
              kind: L2Advertisement
              metadata:
                name: ip-pool-l2-adv
                namespace: metallb-system
              spec:
                ipAddressPools:
                  - kubernetes-vip-ip-pool
        - path: /var/lib/rancher/rke2/server/manifests/endpoint-svc.yaml
```

```
            overwrite: true
            contents:
              inline: |
                apiVersion: v1
                kind: Service
                metadata:
                  name: kubernetes-vip
                  namespace: default
                  labels:
                    serviceType: kubernetes-vip
                spec:
                  ipFamilyPolicy: PreferDualStack
                  ports:
                  - name: rke2-api
                    port: 9345
                    protocol: TCP
                    targetPort: 9345
                  - name: k8s-api
                    port: 6443
                    protocol: TCP
                    targetPort: 6443
                  type: LoadBalancer
        - path: /var/lib/rancher/rke2/server/manifests/rke2-traefik-config.yaml
          overwrite: true
          contents:
            inline: |
              apiVersion: helm.cattle.io/v1
              kind: HelmChartConfig
              metadata:
                name: rke2-traefik
                namespace: kube-system
              spec:
                valuesContent: |-
                  ingressClass:
                    isDefaultClass: true
                  ports:
                    web:
                      hostPort: null    # disallow hostPort
                      exposedPort: 80
                    websecure:
                      hostPort: null    # disallow hostPort
                      exposedPort: 443
                  service:
                    enabled: true
                    spec:
                      type: LoadBalancer
                      externalTrafficPolicy: Local
```

```
                        allocateLoadBalancerNodePorts: false
            mode: 0644
            user:
              name: root
            group:
              name: root
            mode: 0644
            user:
              name: root
            group:
              name: root
          - path: /var/lib/rancher/rke2/server/manifests/metallb-traefik-cr.yaml
            overwrite: true
            contents:
              inline: |
                apiVersion: metallb.io/v1beta1
                kind: IPAddressPool
                metadata:
                  name: ingress-ippool
                  namespace: metallb-system
                spec:
                  addresses:
                    - ${INGRESS_IPV4}/32
                    - ${INGRESS_IPV6}/128
                  serviceAllocation:
                    priority: 100
                    serviceSelectors:
                    - matchExpressions:
                      - {key: app.kubernetes.io/name, operator: In, values: [rke2-
traefik]} # assign to rke2-traefik LB svc
                ---
                apiVersion: metallb.io/v1beta1
                kind: L2Advertisement
                metadata:
                  name: ingress-l2-adv
                  namespace: metallb-system
                spec:
                  ipAddressPools:
                    - ingress-ippool
            mode: 0644
            user:
              name: root
            group:
              name: root
    kubelet:
      extraArgs:
      - provider-id=metal3://PROVIDERID
```

```
    nodeName: "Node-multinode-cluster"
```

The `Metal3MachineTemplate` object specifies the following information:

- The `dataTemplate` to be used as a reference to the template.
- The `hostSelector` to be used matching with the label created during the enrollment process.
- The `image` to be used as a reference to the image generated using `EIB` on the previous section (*Chapter 50, Prepare downstream cluster image for connected scenarios*), and `checksum` and `checksumType` to be used to validate the image.

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: multinode-cluster-controlplane
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: multinode-cluster-controlplane-template
      hostSelector:
        matchLabels:
          cluster-role: control-plane
      image:
        checksum: http://imagecache.local:8080/eibimage-output-telco.raw.sha256
        checksumType: sha256
        diskFormat: raw
        url: http://imagecache.local:8080/eibimage-output-telco.raw
```

The `Metal3DataTemplate` object specifies the `metaData` for the downstream cluster.

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3DataTemplate
metadata:
  name: multinode-cluster-controlplane-template
  namespace: default
spec:
  clusterName: multinode-cluster
  metaData:
    objectNames:
    - key: name
      object: machine
    - key: local-hostname
      object: machine
    - key: local_hostname
```

```
      object: machine
```

The following yaml files are an example configuration for the worker nodes.

A `MachineDeployment`:

```
apiVersion: cluster.x-k8s.io/v1beta2
kind: MachineDeployment
metadata:
  labels:
    cluster.x-k8s.io/cluster-name: multinode-cluster
    nodepool: nodepool-0
  name: multinode-cluster-workers
  namespace: default
spec:
  clusterName: multinode-cluster
  replicas: 3
  selector:
    matchLabels:
      cluster.x-k8s.io/cluster-name: multinode-cluster
      nodepool: nodepool-0
  template:
    metadata:
      labels:
        cluster.x-k8s.io/cluster-name: multinode-cluster
        nodepool: nodepool-0
    spec:
      bootstrap:
        configRef:
          apiGroup: bootstrap.cluster.x-k8s.io
          kind: RKE2ConfigTemplate
          name: multinode-cluster-workers
      clusterName: multinode-cluster
      infrastructureRef:
        apiGroup: infrastructure.cluster.x-k8s.io
        kind: Metal3MachineTemplate
        name: multinode-cluster-workers
      deletion:
        nodeDrainTimeoutSeconds: 0
      version: ${RKE2_VERSION}
```

The RKE2ConfigTemplate` object specifies the configuration template to be used for multinode cluster worker nodes.

```
apiVersion: bootstrap.cluster.x-k8s.io/v1beta2
kind: RKE2ConfigTemplate
metadata:
  name: multinode-cluster-workers
```

```
  namespace: default
spec:
  template:
    spec:
      agentConfig:
        format: ignition
        kubelet:
          extraArgs:
          - provider-id=metal3://PROVIDERID
        nodeName: "Node-multinode-cluster-worker"
        additionalUserData:
          config: |
            variant: fcos
            version: 1.4.0
            systemd:
              units:
              - name: rke2-preinstall.service
                enabled: true
                contents: |
                  [Unit]
                  Description=rke2-preinstall
                  Wants=network-online.target
                  Before=rke2-install.service
                  ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                  [Service]
                  Type=oneshot
                  User=root
                  ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                  # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
                  ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
                  ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                  ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
                  ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                  ExecStartPost=/bin/sh -c "umount /mnt"
                  [Install]
                  WantedBy=multi-user.target
```

The `Metal3MachineTemplate` object contain references to `dataTemplate`, `hostSelector`, and `image` for the worker nodes:

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
```

```
metadata:
  name: multinode-cluster-workers
  namespace: default
spec:
  template:
    spec:
      dataTemplate:
        name: multinode-cluster-workers-template
      hostSelector:
        matchLabels:
          cluster-role: worker
      image:
        checksum: http://imagecache.local:8080/eibimage-slmicro-rt-telco.raw.sha256
        checksumType: sha256
        diskFormat: raw
        url: http://imagecache.local:8080/eibimage-slmicro-rt-telco.raw
```

The `Metal3DataTemplate` object specifies the `metaData` for the downstream cluster for the worker nodes:

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3DataTemplate
metadata:
  name: multinode-cluster-workers-template
  namespace: default
spec:
  clusterName: multinode-cluster
  metaData:
    objectNames:
    - key: name
      object: machine
    - key: local-hostname
      object: machine
    - key: local_hostname
      object: machine
```

Once the file is created by joining the previous blocks, run the following command in the management cluster to start provisioning the new three bare-metal hosts:

```
$ kubectl apply -f capi-provisioning-example.yaml
```

54 Advanced Network Configuration

The directed network provisioning workflow allows for specific network configurations in downstream clusters, such as static IPs, bonding, VLANs, IPv6, etc.

The following sections describe the additional steps required to enable provisioning downstream clusters using advanced network configuration.

Requirements

- The image generated using `EIB` has to include the network folder and the script following this section (*Section 50.2.6, "Additional script for Advanced Network Configuration"*).

Configuration

Before proceeding refer to one of the following sections for guidance on the steps required to enroll and provision the host(s):

- Downstream cluster provisioning with Directed network provisioning (single-node) (*Chapter 52, Downstream cluster provisioning with Directed network provisioning (single-node)*)
- Downstream cluster provisioning with Directed network provisioning (multi-node) (*Chapter 53, Downstream cluster provisioning with Directed network provisioning (multi-node)*)

Any advanced network configuration must be applied at enrollment time through the `BareMetalHost` host definition and an associated Secret containing an `nmstate` formatted `networkData` block. The following example file defines a secret containing the required `networkData` that requests a static `IP` and `VLAN` for the downstream cluster host:

```
apiVersion: v1
kind: Secret
metadata:
  name: controlplane-0-networkdata
type: Opaque
stringData:
  networkData: |
    interfaces:
    - name: ${CONTROLPLANE_INTERFACE}
      type: ethernet
      state: up
      mtu: 1500
      identifier: mac-address
      mac-address: "${CONTROLPLANE_MAC}"
      ipv4:
```

```
      address:
      - ip:  "${CONTROLPLANE_IP}"
        prefix-length: "${CONTROLPLANE_PREFIX}"
      enabled: true
      dhcp: false
  - name: floating
    type: vlan
    state: up
    vlan:
      base-iface: ${CONTROLPLANE_INTERFACE}
      id: ${VLAN_ID}
  dns-resolver:
    config:
      server:
      - "${DNS_SERVER}"
  routes:
    config:
    - destination: 0.0.0.0/0
      next-hop-address: "${CONTROLPLANE_GATEWAY}"
      next-hop-interface: ${CONTROLPLANE_INTERFACE}
```

As you can see, the example shows the configuration to enable the interface with static IPs, as well as the configuration to enable the VLAN using the base interface, once the following variables are replaced with the actual values, according to your infrastructure:

- `${CONTROLPLANE_INTERFACE}` — The control-plane interface to be used for the downstream cluster (for example, `eth0`). Including `identifier: mac-address` the naming is inspected automatically by the MAC address so any interface name can be used.
- `${CONTROLPLANE_IP}` — The IP address to be used as an endpoint for the downstream cluster (must match with the kubeapi-server endpoint).
- `${CONTROLPLANE_PREFIX}` — The CIDR to be used for the downstream cluster (for example, `24` if you want `/24` or `255.255.255.0`).
- `${CONTROLPLANE_GATEWAY}` — The gateway to be used for the downstream cluster (for example, `192.168.100.1`).
- `${CONTROLPLANE_MAC}` — The MAC address to be used for the control-plane interface (for example, `00:0c:29:3e:3e:3e`).
- `${DNS_SERVER}` — The DNS to be used for the downstream cluster (for example, `192.168.100.2`).
- `${VLAN_ID}` — The VLAN ID to be used for the downstream cluster (for example, `100`).

Any other `nmstate`-compliant definition can be used to configure the network for the downstream cluster to adapt to the specific requirements. For example, it is possible to specify a static dual-stack configuration:

```
apiVersion: v1
kind: Secret
metadata:
  name: controlplane-0-networkdata
type: Opaque
stringData:
  networkData: |
    interfaces:
    - name: ${CONTROLPLANE_INTERFACE}
      type: ethernet
      state: up
      mac-address: ${CONTROLPLANE_MAC}
      ipv4:
        enabled: true
        dhcp: false
        address:
        - ip: ${CONTROLPLANE_IP_V4}
          prefix-length: ${CONTROLPLANE_PREFIX_V4}
      ipv6:
        enabled: true
        dhcp: false
        autoconf: false
        address:
        - ip: ${CONTROLPLANE_IP_V6}
          prefix-length: ${CONTROLPLANE_PREFIX_V6}
    routes:
      config:
      - destination: 0.0.0.0/0
        next-hop-address: ${CONTROLPLANE_GATEWAY_V4}
        next-hop-interface: ${CONTROLPLANE_INTERFACE}
      - destination: ::/0
        next-hop-address: ${CONTROLPLANE_GATEWAY_V6}
        next-hop-interface: ${CONTROLPLANE_INTERFACE}
    dns-resolver:
      config:
        server:
        - ${DNS_SERVER_V4}
        - ${DNS_SERVER_V6}
```

As for the previous example, replace the following variables with actual values, according to your infrastructure:

- `${CONTROLPLANE_IP_V4}` - the IPv4 address to assign to the host
- `${CONTROLPLANE_PREFIX_V4}` - the IPv4 prefix of the network to which the host IP belongs
- `${CONTROLPLANE_IP_V6}` - the IPv6 address to assign to the host
- `${CONTROLPLANE_PREFIX_V6}` - the IPv6 prefix of the network to which the host IP belongs
- `${CONTROLPLANE_GATEWAY_V4}` - the IPv4 address of the gateway for the traffic matching the default route
- `${CONTROLPLANE_GATEWAY_V6}` - the IPv6 address of the gateway for the traffic matching the default route
- `${CONTROLPLANE_INTERFACE}` - the name of the interface to assign the addresses to and to use for egress traffic matching the default route, for both IPv4 and IPv6
- `${DNS_SERVER_V4}` and/or `${DNS_SERVER_V6}` - the IP address(es) of the DNS server(s) to use, which can be specified as single or multiple entries. Both IPv4 and/or IPv6 addresses are supported

Note

- You can refer to SUSE Telco Cloud examples repo (https://github.com/suse-edge/telco-cloud-examples/tree/main/telco-examples/downstream-clusters) for more complex examples, including IPv6 only and dual-stack configurations.
- Single-stack IPv6 deployments are in tech preview status and not yet officially supported.

Lastly, regardless of the network configuration details, ensure that the secret is referenced by appending `preprovisioningNetworkDataName` to the `BaremetalHost` object to successfully enroll the host in the management cluster.

```
apiVersion: v1
kind: Secret
metadata:
  name: example-demo-credentials
type: Opaque
data:
```

```
  username: ${BMC_USERNAME}
  password: ${BMC_PASSWORD}
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: example-demo
  labels:
    cluster-role: control-plane
spec:
  architecture: x86_64
  online: true
  bootMACAddress: ${BMC_MAC}
  rootDeviceHints:
    deviceName: /dev/nvme0n1
  bmc:
    address: ${BMC_ADDRESS}
    disableCertificateVerification: true
    credentialsName: example-demo-credentials
  preprovisioningNetworkDataName: controlplane-0-networkdata
```

Note

- If you need to deploy a multi-node cluster, the same process must be done for each node.
- The `Metal3DataTemplate`, `networkData` and `Metal3 IPAM` are currently not supported; only the configuration via static secrets is fully supported.
- Architecture must be either `x86_64` or `aarch64`, depending on the architecture of the bare-metal host to be enrolled.

55 Telco features (DPDK, SR-IOV, CPU isolation, huge pages, NUMA, etc.)

The directed network provisioning workflow allows to automate the Telco features to be used in the downstream clusters to run Telco workloads on top of those servers.

Requirements

- The image generated using `EIB`, as described in the previous section (*Chapter 50, Prepare downstream cluster image for connected scenarios*), has to be located in the management cluster exactly on the path you configured on this section (*Note*).
- The image generated using `EIB` has to include the specific Telco packages following this section (*Section 50.2.5, "Additional configuration for Telco workloads"*).
- The management server created and available to be used on the following sections. For more information, refer to the Management Cluster section: *Part V, "Setting up the management cluster"*.

Configuration

Use the following two sections as the base to enroll and provision the hosts:

- Downstream cluster provisioning with Directed network provisioning (single-node) (*Chapter 52, Downstream cluster provisioning with Directed network provisioning (single-node)*)
- Downstream cluster provisioning with Directed network provisioning (multi-node) (*Chapter 53, Downstream cluster provisioning with Directed network provisioning (multi-node)*)

The Telco features covered in this section are the following:

- DPDK and VFs creation
- SR-IOV and VFs allocation to be used by the workloads
- CPU isolation and performance tuning
- Huge pages configuration
- Kernel parameters tuning

Note

For more information about the Telco features, see *Part VI, "Telco features configuration"*.

The changes required to enable the Telco features shown above are all inside the `RKE2ControlPlane` block in the provision file `capi-provisioning-example.yaml`. The rest of the information inside the file `capi-provisioning-example.yaml` is the same as the information provided in the provisioning section (*Chapter 52, Downstream cluster provisioning with Directed network provisioning (single-node)* (page 317)).

To make the process clear, the changes required on that block (`RKE2ControlPlane`) to enable the Telco features are the following:

- The ignition file `/var/lib/rancher/rke2/server/manifests/configmap-sriov-custom-auto.yaml` to be used to define the interfaces, drivers and the number of `VFs` to be created and exposed to the workloads.
    - The values inside the config map `sriov-custom-auto-config` are the only values to be replaced with real values.
        - `${RESOURCE_NAME1}` — The resource name to be used for the first `PF` interface (for example, `sriov-resource-du1`). It is added to the prefix `rancher.io` to be used as a label to be used by the workloads (for example, `rancher.io/sriov-resource-du1`).
        - `${SRIOV-NIC-NAME1}` — The name of the first `PF` interface to be used (for example, `eth0`).
        - `${PF_NAME1}` — The name of the first physical function `PF` to be used. Generate more complex filters using this (for example, `eth0#2-5`).
        - `${DRIVER_NAME1}` — The driver name to be used for the first `VF` interface (for example, `vfio-pci`).
        - `${NUM_VFS1}` — The number of `VFs` to be created for the first `PF` interface (for example, `8`).
- The `/var/sriov-auto-filler.sh` to be used as a translator between the high-level config map `sriov-custom-auto-config` and the `sriovnetworknodepolicy` which contains the low-level hardware information. This script has been created to abstract the user from the complexity to know in advance the hardware information. No changes are required in this file, but it should be present if we need to enable `sr-iov` and create `VFs`.
- The kernel arguments to be used to enable the following features:

Parameter	Value	Description

isolcpus	domain,nohz,managed_irq,1-30,33-62	Isolate the cores 1-30 and 33-62.
skew_tick	1	Allows the kernel to skew the timer interrupts across the isolated CPUs.
nohz	on	Allows the kernel to run the timer tick on a single CPU when the system is idle.
nohz_full	1-30,33-62	kernel boot parameter is the current main interface to configure full dynticks along with CPU Isolation.
rcu_nocbs	1-30,33-62	Allows the kernel to run the RCU callbacks on a single CPU when the system is idle.
irqaffinity	0,31,32,63	Allows the kernel to run the interrupts on a single CPU when the system is idle.
idle	poll	Minimizes the latency of exiting the idle state.
iommu	pt	Allows to use vfio for the dpdk interfaces.
intel_iommu	on	Enables the use of vfio for VFs.
hugepagesz	1G	Allows to set the size of "huge pages" to 1 GB.
hugepages	40	Number of "huge pages" requested to allocate for the size defined just before (1 GB).

hugepagesz	2M	Allows to set the size of "huge pages" to 2 MB.
hugepages	0	Number of "huge pages" requested to allocate for the size defined just before (2 MB).
default_hugepagesz	1G	Default value for the "huge pages" size.
nowatchdog		Disables the watchdog.
nmi_watchdog	0	Disables the NMI watchdog.

- The following systemd services are used to enable the following:
  - `rke2-preinstall.service` to replace automatically the `PROVIDERID` and `node-name` during the provisioning process using the Ironic information.
  - `cpu-partitioning.service` to enable the isolation cores of the `CPU` (for example, `1-30,33-62`).
  - `performance-settings.service` to enable the CPU performance tuning.
  - `sriov-custom-auto-vfs.service` to install the `sriov` Helm chart, wait until custom resources are created and run the `/var/sriov-auto-filler.sh` to replace the values in the config map `sriov-custom-auto-config` and create the `sriovnetworknodepolicy` to be used by the workloads.
- The `${RKE2_VERSION}` is the version of `RKE2` to be used replacing this value (for example, `v1.36.3+rke2r1`).

With all these changes mentioned, the `RKE2ControlPlane` block in the `capi-provisioning-example.yaml` will look like the following:

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
```

```
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  serverConfig:
    cni: calico
    cniMultusEnable: true
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        storage:
          files:
          - path: /var/lib/rancher/rke2/server/manifests/configmap-sriov-custom-auto.yaml
            overwrite: true
            contents:
              inline: |
                apiVersion: v1
                kind: ConfigMap
                metadata:
                  name: sriov-custom-auto-config
                  namespace: kube-system
                data:
                  config.json: |
                    [
                        {
                          "resourceName": "${RESOURCE_NAME1}",
                          "interface": "${SRIOV-NIC-NAME1}",
                          "pfname": "${PF_NAME1}",
                          "driver": "${DRIVER_NAME1}",
                          "numVFsToCreate": ${NUM_VFS1}
                        },
                        {
                          "resourceName": "${RESOURCE_NAME2}",
                          "interface": "${SRIOV-NIC-NAME2}",
                          "pfname": "${PF_NAME2}",
                          "driver": "${DRIVER_NAME2}",
                          "numVFsToCreate": ${NUM_VFS2}
                        }
                    ]
```

```
            mode: 0644
            user:
              name: root
            group:
              name: root
          - path: /var/lib/rancher/rke2/server/manifests/sriov-crd.yaml
            overwrite: true
            contents:
              inline: |
                apiVersion: helm.cattle.io/v1
                kind: HelmChart
                metadata:
                  name: sriov-crd
                  namespace: kube-system
                spec:
                  chart: oci://registry.suse.com/edge/charts/sriov-crd
                  targetNamespace: sriov-network-operator
                  version: 307.1.0+up1.6.0
                  createNamespace: true
          - path: /var/lib/rancher/rke2/server/manifests/sriov-network-operator.yaml
            overwrite: true
            contents:
              inline: |
                apiVersion: helm.cattle.io/v1
                kind: HelmChart
                metadata:
                  name: sriov-network-operator
                  namespace: kube-system
                spec:
                  chart: oci://registry.suse.com/edge/charts/sriov-network-operator
                  targetNamespace: sriov-network-operator
                  version: 307.1.0+up1.6.0
                  createNamespace: true
        kernel_arguments:
          should_exist:
            - intel_iommu=on
            - iommu=pt
            - idle=poll
            - mce=off
            - hugepagesz=1G hugepages=40
            - hugepagesz=2M hugepages=0
            - default_hugepagesz=1G
            - irqaffinity=${NON-ISOLATED_CPU_CORES}
            - isolcpus=domain,nohz,managed_irq,${ISOLATED_CPU_CORES}
            - nohz_full=${ISOLATED_CPU_CORES}
            - rcu_nocbs=${ISOLATED_CPU_CORES}
            - rcu_nocb_poll
```

```
            - nosoftlockup
            - nowatchdog
            - nohz=on
            - nmi_watchdog=0
            - skew_tick=1
            - quiet
        systemd:
          units:
          - name: rke2-preinstall.service
            enabled: true
            contents: |
              [Unit]
              Description=rke2-preinstall
              Wants=network-online.target
              Before=rke2-install.service
              ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
              [Service]
              Type=oneshot
              User=root
              ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
              # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
              ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
              ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStartPost=/bin/sh -c "umount /mnt"
              [Install]
              WantedBy=multi-user.target
          - name: cpu-partitioning.service
            enabled: true
            contents: |
              [Unit]
              Description=cpu-partitioning
              Wants=network-online.target
              After=network.target network-online.target
              [Service]
              Type=oneshot
              User=root
              ExecStart=/bin/sh -c "echo isolated_cores=${ISOLATED_CPU_CORES} > /etc/
tuned/cpu-partitioning-variables.conf"
              ExecStartPost=/bin/sh -c "tuned-adm profile cpu-partitioning"
              ExecStartPost=/bin/sh -c "systemctl enable tuned.service"
```

```
              [Install]
              WantedBy=multi-user.target
          - name: performance-settings.service
            enabled: true
            contents: |
              [Unit]
              Description=performance-settings
              Wants=network-online.target
              After=network.target network-online.target cpu-partitioning.service
              [Service]
              Type=oneshot
              User=root
              ExecStart=/bin/sh -c "/opt/performance-settings/performance-settings.sh"
              [Install]
              WantedBy=multi-user.target
          - name: sriov-custom-auto-vfs.service
            enabled: true
            contents: |
              [Unit]
              Description=SRIOV Custom Auto VF Creation
              Wants=network-online.target  rke2-server.target
              After=network.target network-online.target rke2-server.target
              [Service]
              User=root
              Type=forking
              TimeoutStartSec=900
              ExecStart=/bin/sh -c "while ! /var/lib/rancher/rke2/bin/kubectl --
kubeconfig=/etc/rancher/rke2/rke2.yaml wait --for condition=ready nodes --all ; do sleep
 2 ; done"
              ExecStartPost=/bin/sh -c "while [ $(/var/lib/rancher/
rke2/bin/kubectl --kubeconfig=/etc/rancher/rke2/rke2.yaml get
 sriovnetworknodestates.sriovnetwork.openshift.io --ignore-not-found --no-headers -A | wc
 -l) -eq 0 ]; do sleep 1; done"
              ExecStartPost=/bin/sh -c "/opt/sriov/sriov-auto-filler.sh"
              RemainAfterExit=yes
              KillMode=process
              [Install]
              WantedBy=multi-user.target
    kubelet:
      extraArgs:
      - provider-id=metal3://PROVIDERID
    nodeName: "localhost.localdomain"
```

Once the file is created by joining the previous blocks, the following command must be executed in the management cluster to start provisioning the new downstream cluster using the Telco features:

```
$ kubectl apply -f capi-provisioning-example.yaml
```

55.1 Precision Time Protocol (PTP)

PTP synchronization is enabled at provisioning time in two steps: enabling the daemons in the EIB image and deploying the configuration files through Cluster API. Two provisioning models are documented, depending on the target platform:

- Standard scenarios (*Section 55.1.1, "Standard scenarios (single ptp4l + phc2sys)"*): a single `ptp4l` instance and, optionally, `phc2sys`. Covers Scenario 1a (*Section 47.3, "Scenario 1a - Ordinary Clock (PTP-only)"*) through Scenario 2a (*Section 47.5, "Scenario 2a - Boundary Clock (PTP-only) with Multiport Single NIC including propagation to Ordinary Clock server"*).
- GNR-D multi-NIC scenarios (*Section 55.1.2, "GNR-D multi-NIC scenarios (DPLL)"*): several `ptp4l` instances, `ts2phc` and on-board DPLL setup, which require additional files and services. Covers Scenario 3a (*Section 47.7, "Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based)"*).

55.1.1 Standard scenarios (single ptp4l + phc2sys)

Since only the configuration files change between these PTP scenarios, this section provides a single generic template followed by the specific values for each scenario.

Enable the daemons in the EIB image

The `linuxptp` package is already part of the Telco package list (see Additional configuration for Telco workloads (*Section 50.2.5, "Additional configuration for Telco workloads"*)). However, the `ptp4l` and `phc2sys` daemons are not enabled by default, so they must be added to the `systemd.enable` list of the same EIB image definition file:

```
operatingSystem:
  systemd:
    enable:
      - ptp4l
      - phc2sys
  packages:
    packageList:
      - linuxptp      # already present in the Telco package list
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

With the daemons enabled at image-build time, both services start automatically once their configuration files are present on the host.

Deploy the configuration files through Cluster API

The PTP configuration is deployed through a modified `RKE2ControlPlane` object that writes the `ptp4l` profile file and the two `/etc/sysconfig` files that control the `ptp4l` and `phc2sys` daemons. The rest of the object is the same as the one described in the provisioning section (*Chapter 52, Downstream cluster provisioning with Directed network provisioning (single-node)* (page 317)); complete it with the other Cluster API objects and replace the placeholders with the values of the chosen scenario:

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  registrationMethod: "control-plane-endpoint"
  serverConfig:
    cni: canal
  agentConfig:
    format: ignition
    cisProfile: cis
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
```

```
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
                ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
                ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
        storage:
          files:
            - path: ${PTP4L_CONF_PATH}
              overwrite: true
              contents:
                inline: |
                  ## Replace ${PTP4L_CONF_CONTENTS} with the ptp4l configuration content
 of the PTP scenario to deploy
                  ${PTP4L_CONF_CONTENTS}
              mode: 0644
              user:
                name: root
              group:
                name: root
            - path: /etc/sysconfig/ptp4l
              overwrite: true
              contents:
                inline: |
                  ## Path:           Network/LinuxPTP
                  ## Description:    Precision Time Protocol (PTP): ptp4l settings
                  ## Replace ${PTP4L_OPTIONS} with the ptp4l OPTIONS value of the PTP
 scenario to deploy
                  OPTIONS="${PTP4L_OPTIONS}"
              mode: 0644
              user:
                name: root
              group:
                name: root
            - path: /etc/sysconfig/phc2sys
              overwrite: true
              contents:
                inline: |
                  ## Path:           Network/LinuxPTP
```

```
                  ## Description:    Precision Time Protocol (PTP): phc2sys settings
                  ## Replace ${PHC2SYS_OPTIONS} with the phc2sys OPTIONS value of the PTP
scenario to deploy
                  OPTIONS="${PHC2SYS_OPTIONS}"
              mode: 0644
              user:
                name: root
              group:
                name: root
    kubelet:
      extraArgs:
        - provider-id=metal3://PROVIDERID
    nodeName: "localhost.localdomain"
```

The placeholders are provided by each scenario:

Placeholder	Description
`${PTP4L_CONF_PATH}`	Path of the `ptp4l` profile file, for example `/etc/pt-p4l-G.8275.1.conf`.
`${PTP4L_CONF_CONTENTS}`	Contents of that `ptp4l` file, taken from the chosen scenario.
`${PTP4L_OPTIONS}`	Value of the `OPTIONS=` line in `/etc/sysconfig/ptp4l`.
`${PHC2SYS_OPTIONS}`	Value of the `OPTIONS=` line in `/etc/sysconfig/phc2sys`.

For the specific values to substitute in each case, see the **CAPI automation** of the corresponding scenario in the PTP scenarios (*Section 47.1, "Introduction"* (page 262)) of the PTP documentation.

Note

This approach is convenient only if the hardware in the cluster is uniform and the same PTP configuration is needed on all hosts, interface name included. The Boundary Clock scenarios that rely on `ts2phc` or on-board DPLL routing require additional files and services in addition to the base example, which are described in the next section.

55.1.2 GNR-D multi-NIC scenarios (DPLL)

The GNR-D Boundary Clock (Scenario 3a (*Section 47.7, "Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based)"*)) cannot be described with a single `ptp4l` profile: it runs one `ptp4l` instance per PHC, aligns the add-in cards with `ts2phc`, wires the on-board DPLL

at boot and forwards the Grandmaster parameters periodically. The provisioning therefore extends the base example with several configuration files, scripts and `systemd` units, all deployed through the same `RKE2ControlPlane` Ignition block. The contents of every file and unit are the ones listed in Scenario 3a (*Section 47.7, "Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based)"*).

Important

In this SUSE Telco Cloud release this scenario is only supported for the "default" (i.e., non "PREEMPT_RT") kernel flavour. Support for the "RT" kernel flavour will be added in a following release.

Enable the daemons in the EIB image

The GNR-D units require a custom setup: a DPLL service, one templated `ptp4l` instance per PHC, and a Grandmaster forwarding timer. Consequently, you must deploy and enable them via the Ignition block below instead of using the EIB `systemd.enable` list. The EIB image only needs the `linuxptp` package, which provides the `ptp4l`, `ts2phc` and `pmc` binaries:

```
operatingSystem:
  packages:
    packageList:
      - linuxptp      # already present in the Telco package list
    sccRegistrationCode: $SCC_REGISTRATION_CODE
```

Deploy the files and units through Cluster API

The `RKE2ControlPlane` object below deploys the GNR-D files and units. It follows the same approach as the standard example: the file contents are provided as placeholders and taken from Scenario 4 (*Section 47.7, "Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based)"*), while the `systemd` units (constant for this scenario) are shown inline. Complete it with the other Cluster API objects as in the standard example, and replace the placeholders with the values of the scenario:

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
```

```
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  registrationMethod: "control-plane-endpoint"
  serverConfig:
    cni: canal
  agentConfig:
    format: ignition
    cisProfile: cis
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
                ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
                ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
            - name: gnrd-ptp-dpll-setup.service
              enabled: true
              contents: |
                [Unit]
```

```
                Description=GNR-D PTP DPLL and internal signal setup
                Before=ts2phc.service ptp4l@nac.service ptp4l@aic1.service
ptp4l@aic2.service
                [Service]
                Type=oneshot
                RemainAfterExit=yes
                ExecStart=/usr/local/bin/01-gnrd-startup-setup.sh
                [Install]
                WantedBy=multi-user.target
            - name: ptp4l@.service
              contents: |
                [Unit]
                Description=PTP Boundary Clock instance %i
                After=gnrd-ptp-dpll-setup.service
                Requires=gnrd-ptp-dpll-setup.service
                [Service]
                ExecStart=/usr/sbin/ptp4l -f /etc/ptp4l-%i.conf
                [Install]
                WantedBy=multi-user.target
            - name: ptp4l@nac.service
              enabled: true
            - name: ptp4l@aic1.service
              enabled: true
            - name: ptp4l@aic2.service
              enabled: true
            - name: ts2phc.service
              enabled: true
            - name: ptp-forward-gm.service
              contents: |
                [Unit]
                Description=Forward Grandmaster parameters to the add-in ptp4l instances
                Wants=ptp4l@nac.service ptp4l@aic1.service ptp4l@aic2.service
                After=ptp4l@nac.service ptp4l@aic1.service ptp4l@aic2.service
                [Service]
                Type=oneshot
                SELinuxContext=system_u:system_r:unconfined_t:s0
                ExecStart=/usr/local/bin/04-ptp-forward-GM.sh
            - name: ptp-forward-gm.timer
              enabled: true
              contents: |
                [Unit]
                Description=Refresh forwarded Grandmaster parameters
                [Timer]
                OnBootSec=2min
                OnUnitActiveSec=5min
                [Install]
                WantedBy=timers.target
```

```
      storage:
        files:
          - path: /usr/local/bin/01-gnrd-startup-setup.sh
            mode: 0755
            overwrite: true
            contents:
              inline: |
                ${GNRD_DPLL_SETUP_CONTENTS}
          - path: /usr/local/bin/04-ptp-forward-GM.sh
            mode: 0755
            overwrite: true
            contents:
              inline: |
                ${GNRD_FORWARD_GM_CONTENTS}
          - path: /etc/ts2phc-cf-all.cfg
            mode: 0644
            overwrite: true
            contents:
              inline: |
                ${TS2PHC_CONF_CONTENTS}
          - path: /etc/sysconfig/ts2phc
            mode: 0644
            overwrite: true
            contents:
              inline: |
                OPTIONS="${TS2PHC_OPTIONS}"
          - path: /etc/ptp4l-nac.conf
            mode: 0644
            overwrite: true
            contents:
              inline: |
                ${PTP4L_NAC_CONTENTS}
          - path: /etc/ptp4l-aic1.conf
            mode: 0644
            overwrite: true
            contents:
              inline: |
                ${PTP4L_AIC1_CONTENTS}
          - path: /etc/ptp4l-aic2.conf
            mode: 0644
            overwrite: true
            contents:
              inline: |
                ${PTP4L_AIC2_CONTENTS}
    kubelet:
      extraArgs:
        - provider-id=metal3://PROVIDERID
```

```
    nodeName: "localhost.localdomain"
```

The placeholders are provided by the GNR-D scenario:

Placeholder	Description
`${GNRD_DPLL_SETUP_CONTENTS}`	Contents of the DPLL boot-time setup script (`01-gnrd-start-up-setup.sh`).
`${GNRD_FORWARD_GM_CONTENTS}`	Contents of the Grandmaster forwarding script (`04-ptp-forward-GM.sh`).
`${TS2PHC_CONF_CONTENTS}`	Contents of the `ts2phc` configuration file (`/etc/ts2phc-cf-all.cfg`).
`${TS2PHC_OPTIONS}`	Value of the `OPTIONS=` line in `/etc/sysconfig/ts2phc`.
`${PTP4L_NAC_CONTENTS}`	Contents of the integrated-controller `ptp4l` file (`/etc/ptp4l-nac.conf`).
`${PTP4L_AIC1_CONTENTS}`	Contents of the add-in card 1 `ptp4l` file (`/etc/ptp4l-aic1.conf`).
`${PTP4L_AIC2_CONTENTS}`	Contents of the add-in card 2 `ptp4l` file (`/etc/ptp4l-aic2.conf`).

For the specific values to substitute, see the **CAPI automation** of Scenario 4 (*Section 47.7, "Scenario 3a - Boundary Clock (PTP-only) Multi-NIC with internal timing module (DPLL GNR-D based)"*).

Note

The `ptp4l@.service` template is deployed without `enabled`; the instances are created by enabling `ptp4l@nac`, `ptp4l@aic1` and `ptp4l@aic2`. `ptp-forward-gm.service` is triggered by its timer, so only the timer is enabled. `ts2phc.service` is shipped by the `linuxptp` package and reads `/etc/sysconfig/ts2phc`; the boot ordering is enforced by the `Before=` directive of `gnrd-ptp-dpll-setup.service`. Adjust the device names (`eno8703np0`, `ptp<N>`, `eno16495np0`, `eno16595np0`) and the number of add-in cards to match the target hardware.

56 Private registry

It is possible to configure a private registry as a mirror for images used by workloads.

To do this we create the secret containing the information about the private registry to be used by the downstream cluster.

```
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-cert
  namespace: default
data:
  tls.crt: ${TLS_CERTIFICATE}
  tls.key: ${TLS_KEY}
  ca.crt: ${CA_CERTIFICATE}
type: kubernetes.io/tls
---
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-auth
  namespace: default
data:
  username: ${REGISTRY_USERNAME}
  password: ${REGISTRY_PASSWORD}
```

The `tls.crt`, `tls.key` and `ca.crt` are the certificates to be used to authenticate the private registry. The `username` and `password` are the credentials to be used to authenticate the private registry.

Note

The `tls.crt`, `tls.key`, `ca.crt` , `username` and `password` have to be encoded in base64 format before to be used in the secret.

With all these changes mentioned, the `RKE2ControlPlane` block in the `capi-provisioning-example.yaml` will look like the following:

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
```

```
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  privateRegistriesConfig:
    mirrors:
      "registry.example.com":
        endpoint:
          - "https://registry.example.com:5000"
    configs:
      "registry.example.com":
        authSecret:
          apiVersion: v1
          kind: Secret
          namespace: default
          name: private-registry-auth
        tls:
          tlsConfigSecret:
            apiVersion: v1
            kind: Secret
            namespace: default
            name: private-registry-cert
  serverConfig:
    cni: calico
    cniMultusEnable: true
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
          - name: rke2-preinstall.service
            enabled: true
            contents: |
              [Unit]
              Description=rke2-preinstall
              Wants=network-online.target
              Before=rke2-install.service
              ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
              [Service]
```

```
              Type=oneshot
              User=root
              ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
              # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
              ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
              ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStartPost=/bin/sh -c "umount /mnt"
              [Install]
              WantedBy=multi-user.target
    kubelet:
      extraArgs:
      - provider-id=metal3://PROVIDERID
    nodeName: "localhost.localdomain"
```

Where the `registry.example.com` is the example name of the private registry to be used by the downstream cluster, and it should be replaced with the real values.

57 Downstream cluster provisioning in air-gapped scenarios

The directed network provisioning workflow allows to automate the provisioning of downstream clusters in air-gapped scenarios.

57.1 Requirements for air-gapped scenarios

1. The `raw` image generated using `EIB` must include the specific container images (helm-chart OCI and container images) required to run the downstream cluster in an air-gapped scenario. For more information, refer to this section (*Chapter 51, Prepare downstream cluster image for air-gap scenarios*).

2. In case of using SR-IOV or any other custom workload, the images required to run the workloads must be preloaded in your private registry following the preload private registry section (*Section 51.2.7, "Preparing the air-gap artifacts"* (page 312)).

57.2 Enroll the bare-metal hosts in air-gap scenarios

The process to enroll the bare-metal hosts in the management cluster is the same as described in the previous section (*Chapter 52, Downstream cluster provisioning with Directed network provisioning (single-node)* (page 316)).

57.3 Provision the downstream cluster in air-gap scenarios

There are some important changes required to provision the downstream cluster in air-gapped scenarios:

1. The `RKE2ControlPlane` block in the `capi-provisioning-example.yaml` file must include the `spec.agentConfig.airGapped: true` directive.
2. The private registry configuration must be included in the `RKE2ControlPlane` block in the `capi-provisioning-airgap-example.yaml` file following the private registry section (*Chapter 56, Private registry*).
3. If you are using SR-IOV or any other `AdditionalUserData` configuration (combustion script) which requires the helm-chart installation, you must modify the content to reference the private registry instead of using the public registry.

The following example shows the SR-IOV configuration in the `AdditionalUserData` block in the `capi-provisioning-airgap-example.yaml` file with the modifications required to reference the private registry

- Private Registry secrets references
- Helm-Chart definition using the private registry instead of the public OCI images.

```
# secret to include the private registry certificates
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-cert
  namespace: default
data:
  tls.crt: ${TLS_BASE64_CERT}
  tls.key: ${TLS_BASE64_KEY}
  ca.crt: ${CA_BASE64_CERT}
type: kubernetes.io/tls
---
# secret to include the private registry auth credentials
apiVersion: v1
kind: Secret
metadata:
  name: private-registry-auth
  namespace: default
data:
  username: ${REGISTRY_USERNAME}
```

```
  password: ${REGISTRY_PASSWORD}
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: single-node-cluster
  namespace: default
spec:
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: single-node-cluster-controlplane
  replicas: 1
  version: ${RKE2_VERSION}
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 0
  privateRegistriesConfig:       # Private registry configuration to add your own mirror
 and credentials
    mirrors:
      docker.io:
        endpoint:
          - "${PRIVATE_REGISTRY_URL}"
        rewrite:
          "^(.*)$": "mirror/$1"
      registry.suse.com:
        endpoint:
          - "${PRIVATE_REGISTRY_URL}"
        rewrite:
          "^(.*)$": "mirror/$1"
      registry.suse.de:
        endpoint:
          - "${PRIVATE_REGISTRY_URL}"
        rewrite:
          "^(.*)$": "mirror/$1"
      registry.opensuse.org:
        endpoint:
          - "${PRIVATE_REGISTRY_URL}"
        rewrite:
          "^(.*)$": "mirror/$1"
      registry.rancher.com:
        endpoint:
          - "${PRIVATE_REGISTRY_URL}"
        rewrite:
          "^(.*)$": "mirror/$1"
    configs:
```

```
    "192.168.100.22:5000":
      authSecret:
        apiVersion: v1
        kind: Secret
        namespace: default
        name: private-registry-auth
      tls:
        tlsConfigSecret:
          apiVersion: v1
          kind: Secret
          namespace: default
          name: private-registry-cert
        insecureSkipVerify: false
serverConfig:
  cni: calico
  cniMultusEnable: true
agentConfig:
  airGapped: true       # Airgap true to enable airgap mode
  format: ignition
  additionalUserData:
    config: |
      variant: fcos
      version: 1.4.0
      storage:
        files:
        - path: /var/lib/rancher/rke2/server/manifests/configmap-sriov-custom-auto.yaml
          overwrite: true
          contents:
            inline: |
              apiVersion: v1
              kind: ConfigMap
              metadata:
                name: sriov-custom-auto-config
                namespace: sriov-network-operator
              data:
                config.json: |
                  [
                      {
                        "resourceName": "${RESOURCE_NAME1}",
                        "interface": "${SRIOV-NIC-NAME1}",
                        "pfname": "${PF_NAME1}",
                        "driver": "${DRIVER_NAME1}",
                        "numVFsToCreate": ${NUM_VFS1}
                      },
                      {
                        "resourceName": "${RESOURCE_NAME2}",
                        "interface": "${SRIOV-NIC-NAME2}",
```

```
                        "pfname": "${PF_NAME2}",
                        "driver": "${DRIVER_NAME2}",
                        "numVFsToCreate": ${NUM_VFS2}
                      }
                ]
          mode: 0644
          user:
            name: root
          group:
            name: root
        - path: /var/lib/rancher/rke2/server/manifests/sriov.yaml
          overwrite: true
          contents:
            inline: |
              apiVersion: v1
              data:
                .dockerconfigjson: ${REGISTRY_AUTH_DOCKERCONFIGJSON}
              kind: Secret
              metadata:
                name: privregauth
                namespace: kube-system
              type: kubernetes.io/dockerconfigjson
              ---
              apiVersion: v1
              kind: ConfigMap
              metadata:
                namespace: kube-system
                name: example-repo-ca
              data:
                ca.crt: |-
                  -----BEGIN CERTIFICATE-----
                  ${CA_BASE64_CERT}
                  -----END CERTIFICATE-----
              ---
              apiVersion: helm.cattle.io/v1
              kind: HelmChart
              metadata:
                name: sriov-crd
                namespace: kube-system
              spec:
                chart: oci://${PRIVATE_REGISTRY_URL}/mirror/sriov-crd
                dockerRegistrySecret:
                  name: privregauth
                repoCAConfigMap:
                  name: example-repo-ca
                createNamespace: true
                set:
```

```
                global.clusterCIDR: 192.168.0.0/18
                global.clusterCIDRv4: 192.168.0.0/18
                global.clusterDNS: 10.96.0.10
                global.clusterDomain: cluster.local
                global.rke2DataDir: /var/lib/rancher/rke2
                global.serviceCIDR: 10.96.0.0/12
              targetNamespace: sriov-network-operator
              version: 307.1.0+up1.6.0
            ---
            apiVersion: helm.cattle.io/v1
            kind: HelmChart
            metadata:
              name: sriov-network-operator
              namespace: kube-system
            spec:
              chart: oci://${PRIVATE_REGISTRY_URL}/mirror/sriov-network-operator
              dockerRegistrySecret:
                name: privregauth
              repoCAConfigMap:
                name: example-repo-ca
              createNamespace: true
              set:
                global.clusterCIDR: 192.168.0.0/18
                global.clusterCIDRv4: 192.168.0.0/18
                global.clusterDNS: 10.96.0.10
                global.clusterDomain: cluster.local
                global.rke2DataDir: /var/lib/rancher/rke2
                global.serviceCIDR: 10.96.0.0/12
              targetNamespace: sriov-network-operator
              version: 307.1.0+up1.6.0
        mode: 0644
        user:
          name: root
        group:
          name: root
      kernel_arguments:
        should_exist:
        - intel_iommu=on
        - iommu=pt
        - idle=poll
        - mce=off
        - hugepagesz=1G hugepages=40
        - hugepagesz=2M hugepages=0
        - default_hugepagesz=1G
        - irqaffinity=${NON-ISOLATED_CPU_CORES}
        - isolcpus=domain,nohz,managed_irq,${ISOLATED_CPU_CORES}
        - nohz_full=${ISOLATED_CPU_CORES}
```

```
            - rcu_nocbs=${ISOLATED_CPU_CORES}
            - rcu_nocb_poll
            - nosoftlockup
            - nowatchdog
            - nohz=on
            - nmi_watchdog=0
            - skew_tick=1
            - quiet
        systemd:
          units:
          - name: rke2-preinstall.service
            enabled: true
            contents: |
              [Unit]
              Description=rke2-preinstall
              Wants=network-online.target
              Before=rke2-install.service
              ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
              [Service]
              Type=oneshot
              User=root
              ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
              # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
              ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
              ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStartPost=/bin/sh -c "umount /mnt"
              [Install]
              WantedBy=multi-user.target
          - name: cpu-partitioning.service
            enabled: true
            contents: |
              [Unit]
              Description=cpu-partitioning
              Wants=network-online.target
              After=network.target network-online.target
              [Service]
              Type=oneshot
              User=root
              ExecStart=/bin/sh -c "echo isolated_cores=${ISOLATED_CPU_CORES} > /etc/
tuned/cpu-partitioning-variables.conf"
```

```
              ExecStartPost=/bin/sh -c "tuned-adm profile cpu-partitioning"
              ExecStartPost=/bin/sh -c "systemctl enable tuned.service"
              [Install]
              WantedBy=multi-user.target
          - name: performance-settings.service
            enabled: true
            contents: |
              [Unit]
              Description=performance-settings
              Wants=network-online.target
              After=network.target network-online.target cpu-partitioning.service
              [Service]
              Type=oneshot
              User=root
              ExecStart=/bin/sh -c "/opt/performance-settings/performance-settings.sh"
              [Install]
              WantedBy=multi-user.target
          - name: sriov-custom-auto-vfs.service
            enabled: true
            contents: |
              [Unit]
              Description=SRIOV Custom Auto VF Creation
              Wants=network-online.target  rke2-server.target
              After=network.target network-online.target rke2-server.target
              [Service]
              User=root
              Type=forking
              TimeoutStartSec=1800
              ExecStart=/bin/sh -c "while ! /var/lib/rancher/rke2/bin/kubectl --
kubeconfig=/etc/rancher/rke2/rke2.yaml wait --for condition=ready nodes --timeout=30m --
all ; do sleep 10 ; done"
              ExecStartPost=/bin/sh -c "/opt/sriov/sriov-auto-filler.sh"
              RemainAfterExit=yes
              KillMode=process
              [Install]
              WantedBy=multi-user.target
    kubelet:
      extraArgs:
      - provider-id=metal3://PROVIDERID
    nodeName: "localhost.localdomain"
```

VIII Day 2 Operations

58 Edge 3.7 migration 373

59 Management Cluster 377

60 Lifecycle actions 432

This section explains how administrators can handle different "Day Two" operation tasks both on the management and on the downstream clusters.

58 Edge 3.7 migration

This section explains how to migrate your `management` and `downstream` clusters from `SUSE Telco Cloud 3.6` to `SUSE Telco Cloud 3.7.0`.

Important

Always perform cluster migrations from the `latest Z-stream` release of `SUSE Telco Cloud 3.6`.

Always migrate to the `SUSE Telco Cloud 3.7.0` release. For subsequent post-migration upgrades, refer to the management (*Chapter 59, Management Cluster*) sections.

The following table lists the different types of clusters and the methods to upgrade clusters:

TABLE 58.1: CLUSTERS AND METHODS TO UPGRADE DOWNSTREAM CLUSTERS

Cluster type	Method
EIB provisioned clusters	See *Section 58.1.3, "Fleet"* for details.
Metal[3] provisioned clusters	See Downstream cluster upgrades (*Section 60.3, "Downstream cluster upgrades"*) for details.

58.1 Management Cluster

This section covers the following topics:

Section 58.1.1, "Prerequisites" - prerequisite steps to complete before starting the migration.

Section 58.1.2, "Upgrade Controller" - how to do a `management` cluster migration using the *Chapter 21, Upgrade Controller*.

Section 58.1.3, "Fleet" - how to do a `management` cluster migration using *Chapter 9, Fleet*.

58.1.1 Prerequisites

58.1.1.1 Upgrade the Bare Metal Operator CRDs

Note

Applies only to CAPI/Metal3 management clusters that require a *Chapter 11, Metal*[3] chart upgrade.

The `Metal3` Helm chart includes the Bare Metal Operator (BMO) (https://book.metal3.io/bmo/introduction.html) CRDs by leveraging Helm's CRD (https://helm.sh/docs/chart_best_practices/custom_resource_definitions/#method-1-let-helm-do-it-for-you) directory.

However, this approach has certain limitations, particularly the inability to upgrade CRDs in this directory using Helm. For more information, refer to the Helm documentation (https://helm.sh/docs/chart_best_practices/custom_resource_definitions/#some-caveats-and-explanations).

As a result, before upgrading Metal[3] to an `SUSE Telco Cloud 3.7.0` compatible version, users must manually upgrade the underlying BMO CRDs.

On a machine with `Helm` installed and `kubectl` configured to point to your `management` cluster:

1. Manually apply the BMO CRDs:

```
helm show crds oci://registry.suse.com/edge/charts/metal3 --version
 307.0.31+up0.16.0 | kubectl apply -f -
```

58.1.2 Upgrade Controller

Important

The `Upgrade Controller` currently supports `SUSE Telco Cloud` release migrations only for **non air-gapped management** clusters.

The following topics are covered as part of this section:

Section 58.1.2.1, "Prerequisites" - prerequisites specific to the `Upgrade Controller`.

Section 58.1.2.2, "Migration steps" - steps for migrating a `management` cluster to a new `SUSE Telco Cloud` version using the `Upgrade Controller`.

58.1.2.1 Prerequisites

58.1.2.1.1 SUSE Telco Cloud 3.7 Upgrade Controller

Before using the `Upgrade Controller`, you must first ensure that it is running a version that is capable of migrating to the desired `SUSE Telco Cloud` release.

To do this:

1. If you already have `Upgrade Controller` deployed from a previous `SUSE Telco Cloud` release, upgrade its chart:

   ```
   helm upgrade upgrade-controller -n upgrade-controller-system oci://
   registry.suse.com/edge/charts/upgrade-controller --version 307.0.4+up0.1.3
   ```

2. If you do **not** have `Upgrade Controller` deployed, follow *Section 21.3, "Installing the Upgrade Controller"*.

58.1.2.2 Migration steps

Performing a `management` cluster migration with the `Upgrade Controller` is fundamentally similar to executing an upgrade.

The only difference is that your `UpgradePlan` **must** specify the `3.7.0` release version:

```
apiVersion: lifecycle.suse.com/v1alpha1
kind: UpgradePlan
metadata:
  name: upgrade-plan-mgmt
  # Change to the namespace of your Upgrade Controller
  namespace: CHANGE_ME
spec:
  releaseVersion: 3.7.0
```

For information on how to use the above `UpgradePlan` to do a migration, refer to Upgrade Controller upgrade process (*Section 59.1, "Upgrade Controller"*).

58.1.3 Fleet

Note

Whenever possible, use the *Section 58.1.2, "Upgrade Controller"* for migration.

Refer to this section only for use cases not covered by the `Upgrade Controller`.

Performing a `management` cluster migration with `Fleet` is fundamentally similar to executing an upgrade. The **key** differences being that:

1. The fleets **must be used** from the release-3.7.0 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.7.0) release of the `suse-edge/fleet-examples` repository.
2. Charts scheduled for an upgrade **must** be upgraded to versions compatible with the `SUSE Telco Cloud 3.7.0` release. For a list of the `SUSE Telco Cloud 3.7.0` components, refer to *Section 78.3, "Release 3.7.0"*.

Important

To ensure a successful `SUSE Telco Cloud 3.7.0` migration, it is important that users comply with the points outlined above.

58.2 Downstream Clusters

Section 58.2.1, "Fleet" - how to do a `downstream` cluster migration using *Chapter 9, Fleet*.

58.2.1 Fleet

Performing a `downstream` cluster migration with `Fleet` is fundamentally similar to executing an upgrade. The **key** differences being that:

1. The fleets **must be used** from the release-3.7.0 (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.7.0) release of the `suse-edge/fleet-examples` repository.
2. Charts scheduled for an upgrade **must** be upgraded to versions compatible with the `SUSE Telco Cloud 3.7.0` release. For a list of the `SUSE Telco Cloud 3.7.0` components, refer to *Section 78.3, "Release 3.7.0"*.

Important

To ensure a successful `SUSE Telco Cloud 3.7.0` migration, it is important that users comply with the points outlined above.

59 Management Cluster

Currently, there are two ways to perform "Day 2" operations on your `management` cluster:

1. Through *Chapter 21, Upgrade Controller* - *Section 59.1, "Upgrade Controller"*
2. Through *Chapter 9, Fleet* - *Section 59.2, "Fleet"*

59.1 Upgrade Controller

Important

The `Upgrade Controller` currently only supports `Day 2` operations for **non air-gapped management** clusters.

This section covers how to perform the various `Day 2` operations related to upgrading your `management` cluster from one `SUSE Telco Cloud` platform version to another.

The `Day 2` operations are automated by the Upgrade Controller (*Chapter 21, Upgrade Controller*) and include:

- SUSE Linux Micro (*Chapter 10, SUSE Linux Micro*) OS upgrade
- *Chapter 14, RKE2* Kubernetes upgrade
- SUSE additional components (SUSE Rancher Prime, SUSE Security, etc.) upgrade

59.1.1 Prerequisites

Before upgrading your `management` cluster, the following prerequisites must be met:

1. `SCC registered nodes` - ensure your cluster nodes' OS are registered with a subscription key that supports the OS version specified in the `SUSE Telco Cloud` release (*Chapter 78, Release Notes*) you intend to upgrade to.
2. `Upgrade Controller` - make sure that the `Upgrade Controller` has been deployed on your `management` cluster. For installation steps, refer to *Section 21.3, "Installing the Upgrade Controller"*.

59.1.2 Upgrade

1. Determine the `SUSE Telco Cloud` release (*Chapter 78, Release Notes*) version that you wish to upgrade your `management` cluster to.

2. In the `management` cluster, deploy an `UpgradePlan` that specifies the desired `release version`. The `UpgradePlan` must be deployed in the namespace of the `Upgrade Controller`.

```
kubectl apply -n <upgrade_controller_namespace> -f - <<EOF
apiVersion: lifecycle.suse.com/v1alpha1
kind: UpgradePlan
metadata:
  name: upgrade-plan-mgmt
  namespace: upgrade-controller-system  # replaces it if the upgrade controller has
 been deployed in a different namespace
spec:
  # Version retrieved from release notes
  releaseVersion: 3.X.Y
EOF
```

Note

There may be use-cases where you would want to make additional configurations over the `UpgradePlan`. For all possible configurations, refer to *Section 21.6.1, "UpgradePlan"*.

3. Deploying the `UpgradePlan` to the `Upgrade Controller's` namespace will begin the `upgrade process`.

Note

For more information on the actual `upgrade process`, refer to *Section 21.5, "How does the Upgrade Controller work?"*.

For information on how to track the `upgrade process`, refer to *Section 21.7, "Tracking the upgrade process"*.

59.1.3 Post-Upgrade Steps

`SUSE Telco Cloud` upgrades from latest `3.6` z-stream to `3.7.0` require a final manual steps to be performed after the `Upgrade Controller` has completed the upgrade process, related to some Helm parameters changes in the `rke2-traefik` Helm chart included in `SUSE Telco Cloud` `3.7.0` release.

- apply the following changes in the `helmchartconfigs.helm.cattle.io/rke2-traefik` API object in `kube-system` namespace (or, in case that API object does not exist, apply them in the `helmcharts.helm.cattle.io/rke2-traefik` API object):

Use the `HelmChartConfig` (or `HelmChart`) manifests to apply these changes as shown below, to assure the `Upgrade Controller` will keep them in future Management cluster upgrades.

```
# You can use `kubectl edit', `kubectl patch` or `kubectl apply` commands to apply the
 changes indicated below
apiVersion: helm.cattle.io/v1
kind: HelmChartConfig
metadata:
  name: rke2-traefik
  namespace: kube-system
spec:
  failurePolicy: reinstall
  valuesContent: |-
    ingressClass:
      isDefaultClass: true
    ports:
      web:
        hostPort: null
        exposedPort: 80
      websecure:
        hostPort: null
        exposedPort: 443
    service:
      enabled: true
      # type: LoadBalancer              # CHANGE_1: This Helm parameter has been REMOVED
      # ipFamilyPolicy: PreferDualStack # CHANGE_2: This Helm parameter has been REMOVED
      spec:
        type: LoadBalancer              # CHANGE_3: NEW Helm parameter replacing the old/
removed one (above)
        ipFamilyPolicy: PreferDualStack # CHANGE_4: NEW Helm parameter replacing the old/
removed one (above)
        externalTrafficPolicy: Local
        allocateLoadBalancerNodePorts: false
    providers:
      kubernetesIngressNGINX:           # CHANGE_5: NEW Helm parameter replacing the old/
removed "KubernetesIngressNginx"
```

```
        enabled: true
        ingressClass: "rke2-ingress-nginx-migration"
        controllerClass: "rke2.cattle.io/ingress-nginx-migration"
EOF
```

Or, in case the `helmchartconfigs.helm.cattle.io/rke2-traefik` API object in the `kube-system` namespace does not exist:

```
# You can use `kubectl edit', `kubectl patch` or `kubectl apply` commands to apply the
 changes indicated below
apiVersion: helm.cattle.io/v1
kind: HelmChart
metadata:
  name: rke2-traefik
  namespace: kube-system
spec:
  chart: rke2-traefik
  version: {rke2-traefik Helm chart version}
  repo: https://rke2-charts.rancher.io
  bootstrap: false
  failurePolicy: reinstall
  backOffLimit: 20
  targetNamespace: kube-system
  set:
    global.cattle.systemDefaultRegistry: registry.rancher.com
    global.rke2DataDir: /var/lib/rancher/rke2
    global.systemDefaultRegistry: registry.rancher.com
  valuesContent: |-
    ingressClass:
      isDefaultClass: true
    ports:
      web:
        hostPort: null
        exposedPort: 80
      websecure:
        hostPort: null
        exposedPort: 443
    service:
      enabled: true
      # type: LoadBalancer              # CHANGE_1: This Helm parameter has been REMOVED
      # ipFamilyPolicy: PreferDualStack # CHANGE_2: This Helm parameter has been REMOVED
      spec:
        type: LoadBalancer              # CHANGE_3: NEW Helm parameter replacing the old/
removed one (above)
        ipFamilyPolicy: PreferDualStack # CHANGE_4: NEW Helm parameter replacing the old/
removed one (above)
        externalTrafficPolicy: Local
```

```
        allocateLoadBalancerNodePorts: false
    providers:
      kubernetesIngressNGINX:           # CHANGE_5: NEW Helm parameter replacing the old/
removed "KubernetesIngressNginx"
        enabled: true
        ingressClass: "rke2-ingress-nginx-migration"
        controllerClass: "rke2.cattle.io/ingress-nginx-migration"
EOF
```

The `{rke2-traefik Helm chart version}` shown above is the one provided by the RKE2/k3s release used in the `SUSE Telco Cloud 3.7.0` release to upgrade to (you can get it from the corresponding RKE2 release page (https://github.com/rancher/rke2/releases/) / k3s release page (https://github.com/k3s-io/k3s/releases/)).

59.2 Fleet

This section offers information on how to perform "Day 2" operations using the Fleet (*Chapter 9, Fleet*) component.

The following topics are covered as part of this section:

1. *Section 59.2.1, "Components"* - default components used for all "Day 2" operations.
2. *Section 59.2.2, "Determine your use-case"* - provides an overview of the Fleet custom resources that will be used and their suitability for different "Day 2" operations use-cases.
3. *Section 59.2.3, "Day 2 workflow"* - provides a workflow guide for executing "Day 2" operations with Fleet.
4. *Section 59.2.4, "OS upgrade"* - describes how to do OS upgrades using Fleet.
5. *Section 59.2.5, "Kubernetes version upgrade"* - describes how to do Kubernetes version upgrades using Fleet.
6. *Section 59.2.6, "Helm chart upgrade"* - describes how to do Helm chart upgrades using Fleet.

59.2.1 Components

Below you can find a description of the default components that should be set up on your `management` cluster so that you can successfully perform "Day 2" operations using Fleet.

59.2.1.1 Rancher

Optional; Responsible for managing `downstream clusters` and deploying the `System Upgrade Controller` on your `management cluster`.

For more information, see *Chapter 6, Rancher*.

59.2.1.2 System Upgrade Controller (SUC)

System Upgrade Controller is responsible for executing tasks on specified nodes based on configuration data provided through a custom resource, called a `Plan`.

SUC is actively utilized to upgrade the operating system and Kubernetes distribution.

For more information about the **SUC** component and how it fits in the Edge stack, see *Chapter 20, System Upgrade Controller*.

59.2.2 Determine your use-case

Fleet uses two types of custom resources (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) to enable the management of Kubernetes and Helm resources.

Below you can find information about the purpose of these resources and the use-cases they are best suited for in the context of "Day 2" operations.

59.2.2.1 GitRepo

A `GitRepo` is a Fleet (*Chapter 9, Fleet*) resource that represents a Git repository from which `Fleet` can create `Bundles`. Each `Bundle` is created based on configuration paths defined inside of the `GitRepo` resource. For more information, see the GitRepo (https://fleet.rancher.io/gitrepo-add) documentation.

In the context of "Day 2" operations, `GitRepo` resources are normally used to deploy `SUC` or `SUC Plans` in **non air-gapped** environments that utilize a *Fleet GitOps* approach.

Alternatively, `GitRepo` resources can also be used to deploy `SUC` or `SUC Plans` on **air-gapped** environments, **provided you mirror your repository setup through a local git server**.

59.2.2.2 Bundle

`Bundles` hold **raw** Kubernetes resources that will be deployed on the targeted cluster. Usually they are created from a `GitRepo` resource, but there are use-cases where they can be deployed manually. For more information refer to the Bundle (https://fleet.rancher.io/bundle-add) documentation.

In the context of "Day 2" operations, `Bundle` resources are normally used to deploy `SUC` or `SUC Plans` in **air-gapped** environments that do not use some form of *local GitOps* procedure (e.g. a **local git server**).

Alternatively, if your use-case does not allow for a *GitOps* workflow (e.g. using a Git repository), `Bundle` resources could also be used to deploy `SUC` or `SUC Plans` in **non air-gapped** environments.

59.2.3 Day 2 workflow

The following is a "Day 2" workflow that should be followed when upgrading a management cluster to a specific Edge release.

1. OS upgrade (*Section 59.2.4, "OS upgrade"*)
2. Kubernetes version upgrade (*Section 59.2.5, "Kubernetes version upgrade"*)
3. Helm chart upgrade (*Section 59.2.6, "Helm chart upgrade"*)

59.2.4 OS upgrade

This section describes how to perform an operating system upgrade using *Chapter 9, Fleet* and the *Chapter 20, System Upgrade Controller*.

The following topics are covered as part of this section:

1. *Section 59.2.4.1, "Components"* - additional components used by the upgrade process.
2. *Section 59.2.4.2, "Overview"* - overview of the upgrade process.
3. *Section 59.2.4.3, "Requirements"* - requirements of the upgrade process.
4. *Section 59.2.4.4, "OS upgrade - SUC plan deployment"* - information on how to deploy `SUC plans`, responsible for triggering the upgrade process.

59.2.4.1 Components

This section covers the custom components that the `OS upgrade` process uses over the default "Day 2" components (*Section 59.2.1, "Components"*).

59.2.4.1.1 systemd.service

The OS upgrade on a specific node is handled by a systemd.service (https://www.freedesktop.org/software/systemd/man/latest/systemd.service.html).

A different service is created depending on what type of upgrade the OS requires from one Edge version to another:

- For Edge versions that require the same OS version (e.g. `6.1`), the `os-pkg-update.service` will be created. It uses transactional-update (https://kubic.opensuse.org/documentation/man-pages/transactional-update.8.html) to perform a normal package upgrade (https://en.opensuse.org/SDB:Zypper_usage#Updating_packages).
- For Edge versions that require an OS version migration (e.g `6.1` → `6.2`), the `os-migration.service` will be created. It uses transactional-update (https://kubic.opensuse.org/documentation/man-pages/transactional-update.8.html) to perform:
  a. A normal package upgrade (https://en.opensuse.org/SDB:Zypper_usage#Updating_packages) which ensures that all packages are at up-to-date in order to mitigate any failures in the migration related to old package versions.
  b. An OS migration by utilizing the `zypper migration` command.

The services mentioned above are shipped on each node through a `SUC plan` which must be located on the management cluster that is in need of an OS upgrade.

59.2.4.2 Overview

The upgrade of the operating system for management cluster nodes is done by utilizing `Fleet` and the `System Upgrade Controller (SUC)`.

Fleet is used to deploy and manage `SUC plans` onto the desired cluster.

Note

`SUC plans` are custom resources (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) that describe the steps that `SUC` needs to follow in order for a specific task to be executed on a set of nodes. For an example of how an `SUC plan` looks like, refer to the upstream repository (https://github.com/rancher/system-upgrade-controller?tab=readme-ov-file#example-plans).

The `OS SUC plans` are shipped to each cluster by deploying a GitRepo (https://fleet.rancher.io/gitrepo-add) or Bundle (https://fleet.rancher.io/bundle-add) resource to a specific Fleet workspace (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups). Fleet retrieves the deployed `GitRepo/Bundle` and deploys its contents (the `OS SUC plans`) to the desired cluster(s).

Note

`GitRepo/Bundle` resources are always deployed on the `management cluster`. Whether to use a `GitRepo` or `Bundle` resource depends on your use-case, check *Section 59.2.2, "Determine your use-case"* for more information.

`OS SUC plans` describe the following workflow:

1. Always cordon (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_cordon/) the nodes before OS upgrades.
2. Always upgrade `control-plane` nodes before `worker` nodes.
3. Always upgrade the cluster on a **one** node at a time basis.

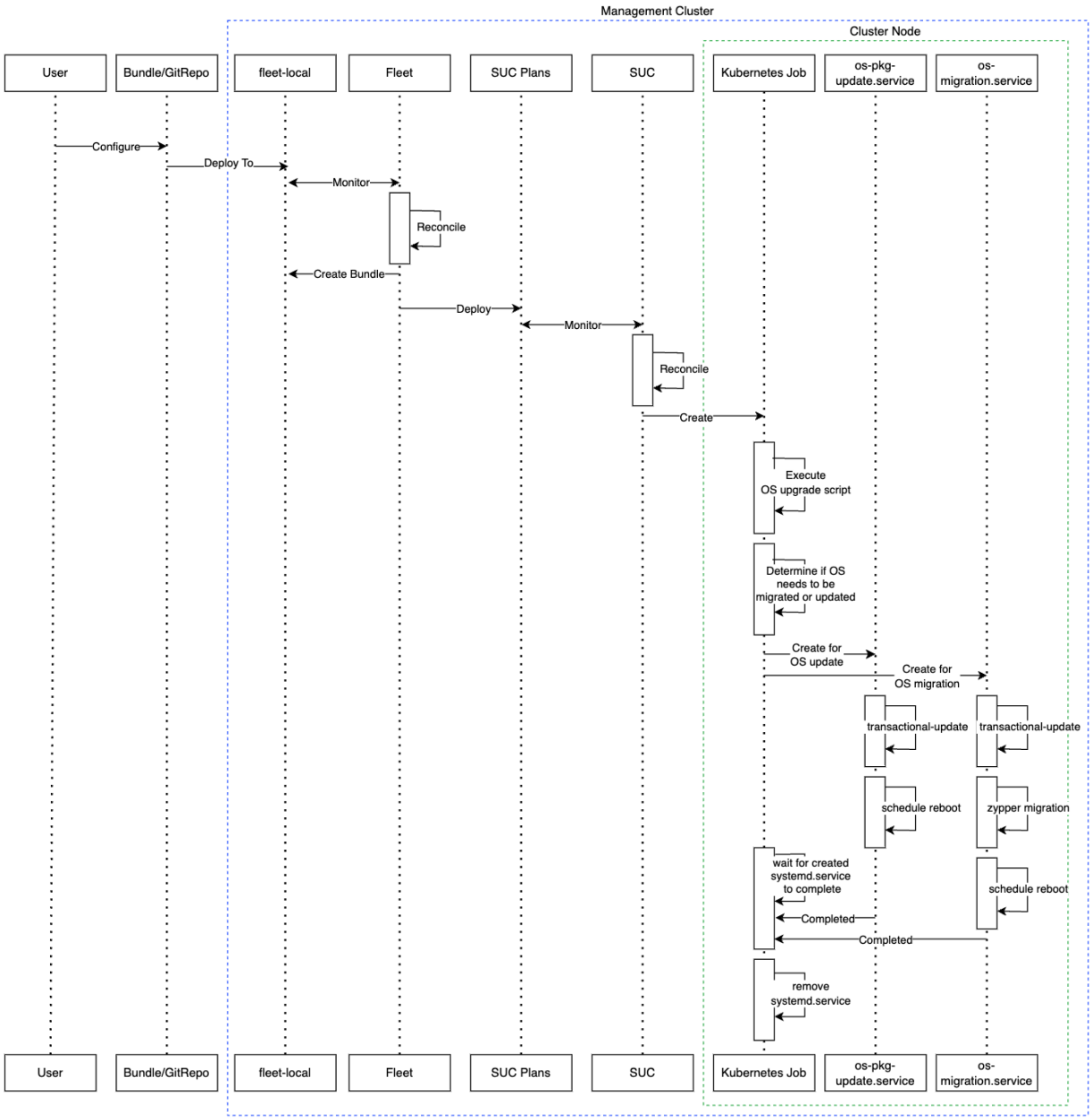
Once the `OS SUC plans` are deployed, the workflow looks like this:

1. SUC reconciles the deployed `OS SUC plans` and creates a `Kubernetes Job` on **each node**.
2. The `Kubernetes Job` creates a systemd.service (*Section 59.2.4.1.1, "systemd.service"*) for either package upgrade, or OS migration.
3. The created `systemd.service` triggers the OS upgrade process on the specific node.

Important

Once the OS upgrade process finishes, the corresponding node will be `rebooted` to apply the updates on the system.

Below you can find a diagram of the above description:



59.2.4.3 Requirements

General:

1. **SCC registered machine** - All management cluster nodes should be registered to `https://scc.suse.com/` which is needed so that the respective `systemd.service` can successfully connect to the desired RPM repository.

   Important

   For Edge releases that require an OS version migration (e.g. `6.1` → `6.2`), make sure that your SCC key supports the migration to the new version.

2. **Make sure that SUC Plan tolerations match node tolerations** - If your Kubernetes cluster nodes have custom **taints**, make sure to add tolerations (https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/) for those taints in the **SUC Plans**. By default, **SUC Plans** have tolerations only for **control-plane** nodes. Default tolerations include:

   - *CriticalAddonsOnly=true:NoExecute*
   - *node-role.kubernetes.io/control-plane:NoSchedule*
   - *node-role.kubernetes.io/etcd:NoExecute*

   Note

   Any additional tolerations must be added under the `.spec.tolerations` section of each Plan. **SUC Plans** related to the OS upgrade can be found in the suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) repository under `fleets/day2/system-upgrade-controller-plans/os-upgrade`. **Make sure you use the Plans from a valid repository release (https://github.com/suse-edge/fleet-examples/releases) tag.**

   An example of defining custom tolerations for the **control-plane** SUC Plan would look like this:

```
apiVersion: upgrade.cattle.io/v1
kind: Plan
metadata:
  name: os-upgrade-control-plane
spec:
  ...
```

```
  tolerations:
  # default tolerations
  - key: "CriticalAddonsOnly"
    operator: "Equal"
    value: "true"
    effect: "NoExecute"
  - key: "node-role.kubernetes.io/control-plane"
    operator: "Equal"
    effect: "NoSchedule"
  - key: "node-role.kubernetes.io/etcd"
    operator: "Equal"
    effect: "NoExecute"
  # custom toleration
  - key: "foo"
    operator: "Equal"
    value: "bar"
    effect: "NoSchedule"
...
```

Air-gapped:

1. **Mirror SUSE RPM repositories** - OS RPM repositories should be locally mirrored so that the `systemd.service` can have access to them. This can be achieved by using either RMT (https://documentation.suse.com/sles/15-SP6/html/SLES-all/book-rmt.html) or SUSE Multi-Linux Manager Documentation (https://documentation.suse.com/multi-linux-manager/5.1/en/docs/index.html)..

59.2.4.4 OS upgrade - SUC plan deployment

Important

For environments previously upgraded using this procedure, users should ensure that **one** of the following steps is completed:

- `Remove any previously deployed SUC Plans related to older Edge release versions from the management cluster` - can be done by removing the desired cluster from the existing `GitRepo/Bundle` target configuration (https://fleet.rancher.io/gitrepo-targets#target-matching), or removing the `GitRepo/Bundle` resource altogether.
- `Reuse the existing GitRepo/Bundle resource` - can be done by pointing the resource's revision to a new tag that holds the correct fleets for the desired `suse-edge/fleet-examples` release (https://github.com/suse-edge/fleet-examples/releases).

This is done in order to avoid clashes between `SUC Plans` for older Edge release versions.

If users attempt to upgrade, while there are existing `SUC Plans` on the management cluster, they will see the following fleet error:

```
Not installed: Unable to continue with install: Plan <plan_name> in namespace
 <plan_namespace> exists and cannot be imported into the current release: invalid
 ownership metadata; annotation validation error..
```

As mentioned in *Section 59.2.4.2, "Overview"*, OS upgrades are done by shipping `SUC plans` to the desired cluster through one of the following ways:

- Fleet `GitRepo` resource - *Section 59.2.4.4.1, "SUC plan deployment - GitRepo resource"*.
- Fleet `Bundle` resource - *Section 59.2.4.4.2, "SUC plan deployment - Bundle resource"*.

To determine which resource you should use, refer to *Section 59.2.2, "Determine your use-case"*.

For use-cases where you wish to deploy the `OS SUC plans` from a third-party GitOps tool, refer to *Section 59.2.4.4.3, "SUC Plan deployment - third-party GitOps workflow"*

59.2.4.4.1 SUC plan deployment - GitRepo resource

A **GitRepo** resource, that ships the needed `OS SUC plans`, can be deployed in one of the following ways:

1. Through the `Rancher UI` - *Section 59.2.4.4.1.1, "GitRepo creation - Rancher UI"* (when `Rancher` is available).
2. By manually deploying (*Section 59.2.4.4.1.2, "GitRepo creation - manual"*) the resource to your `management cluster`.

Once deployed, to monitor the OS upgrade process of the nodes of your targeted cluster, refer to *Section 20.3, "Monitoring System Upgrade Controller Plans"*.

59.2.4.4.1.1 GitRepo creation - Rancher UI

To create a `GitRepo` resource through the Rancher UI, follow their official documentation (https://ranchermanager.docs.rancher.com/v2.15/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui).

The Edge team maintains a ready to use fleet (https://github.com/suse-edge/fleet-examples/tree/release-3.7.0/fleets/day2/system-upgrade-controller-plans/os-upgrade). Depending on your environment this fleet could be used directly or as a template.

> **Important**
>
> Always use this fleet from a valid Edge release (https://github.com/suse-edge/fleet-examples/releases) tag.

For use-cases where no custom changes need to be included to the `SUC plans` that the fleet ships, users can directly refer the `os-upgrade` fleet from the `suse-edge/fleet-examples` repository.

In cases where custom changes are needed (e.g. to add custom tolerations), users should refer the `os-upgrade` fleet from a separate repository, allowing them to add the changes to the SUC plans as required.

An example of how a `GitRepo` can be configured to use the fleet from the `suse-edge/fleet-examples` repository, can be viewed here (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/gitrepos/day2/os-upgrade-gitrepo.yaml).

59.2.4.4.1.2 GitRepo creation - manual

1. Pull the **GitRepo** resource:

```
curl -o os-upgrade-gitrepo.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.7.0/gitrepos/day2/os-upgrade-gitrepo.yaml
```

2. Edit the **GitRepo** configuration:

   - Remove the `spec.targets` section - only needed for downstream clusters.

```
# Example using sed
sed -i.bak '/^  targets:/,$d' os-upgrade-gitrepo.yaml && rm -f os-upgrade-gitrepo.yaml.bak

# Example using yq (v4+)
yq eval 'del(.spec.targets)' -i os-upgrade-gitrepo.yaml
```

   - Point the namespace of the `GitRepo` to the `fleet-local` namespace - done in order to deploy the resource on the management cluster.

```
# Example using sed
sed -i.bak 's/namespace: fleet-default/namespace: fleet-local/' os-upgrade-gitrepo.yaml && rm -f os-upgrade-gitrepo.yaml.bak

# Example using yq (v4+)
yq eval '.metadata.namespace = "fleet-local"' -i os-upgrade-gitrepo.yaml
```

3. Apply the **GitRepo** resource your `management cluster`:

```
kubectl apply -f os-upgrade-gitrepo.yaml
```

4. View the created **GitRepo** resource under the `fleet-local` namespace:

```
kubectl get gitrepo os-upgrade -n fleet-local

# Example output
NAME            REPO                                                    COMMIT
 BUNDLEDEPLOYMENTS-READY   STATUS
```

```
os-upgrade      https://github.com/suse-edge/fleet-examples.git   release-3.7.0  0/0
```

59.2.4.4.2 SUC plan deployment - Bundle resource

A **Bundle** resource, that ships the needed `OS SUC Plans`, can be deployed in one of the following ways:

1. Through the `Rancher UI` - *Section 59.2.4.4.2.1, "Bundle creation - Rancher UI"* (when `Rancher` is available).
2. By manually deploying (*Section 59.2.4.4.2.2, "Bundle creation - manual"*) the resource to your `management cluster`.

Once deployed, to monitor the OS upgrade process of the nodes of your targeted cluster, refer to *Section 20.3, "Monitoring System Upgrade Controller Plans"*.

59.2.4.4.2.1 Bundle creation - Rancher UI

The Edge team maintains a ready to use bundle (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml) that can be used in the below steps.

Important

Always use this bundle from a valid Edge release (https://github.com/suse-edge/fleet-examples/releases) tag.

To create a bundle through Rancher's UI:

1. In the upper left corner, click # → **Continuous Delivery**
2. Go to **Advanced** > **Bundles**
3. Select **Create from YAML**

4. From here you can create the Bundle in one of the following ways:

> **Note**
>
> There might be use-cases where you would need to include custom changes to the `SUC plans` that the bundle ships (e.g. to add custom tolerations). Make sure to include those changes in the bundle that will be generated by the below steps.

   a. By manually copying the bundle content (https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.7.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml) from `suse-edge/fleet-examples` to the **Create from YAML** page.

   b. By cloning the suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) repository from the desired release (https://github.com/suse-edge/fleet-examples/releases) tag and selecting the **Read from File** option in the **Create from YAML** page. From there, navigate to the bundle location (`bundles/day2/system-upgrade-controller-plans/os-upgrade`) and select the bundle file. This will auto-populate the **Create from YAML** page with the bundle content.

5. Edit the Bundle in the Rancher UI:

   - Change the **namespace** of the `Bundle` to point to the `fleet-local` namespace.

```
# Example
kind: Bundle
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: os-upgrade
  namespace: fleet-local
...
```

   - Change the **target** clusters for the `Bundle` to point to your `local`(management) cluster:

```
spec:
  targets:
  - clusterName: local
```

> **Note**
>
> There are some use-cases where your `local` cluster could have a different name.

To retrieve your `local` cluster name, execute the command below:

```
kubectl get clusters.fleet.cattle.io -n fleet-local
```

6. Select **Create**

59.2.4.4.2.2 Bundle creation - manual

1. Pull the **Bundle** resource:

```
curl -o os-upgrade-bundle.yaml https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.7.0/bundles/day2/system-upgrade-controller-plans/os-upgrade/os-upgrade-bundle.yaml
```

2. Edit the `Bundle` configuration:

   - Change the **target** clusters for the `Bundle` to point to your `local`(management) cluster:

```
spec:
  targets:
  - clusterName: local
```

> **Note**
>
> There are some use-cases where your `local` cluster could have a different name.
>
> To retrieve your `local` cluster name, execute the command below:
>
> ```
> kubectl get clusters.fleet.cattle.io -n fleet-local
> ```

   - Change the **namespace** of the `Bundle` to point to the `fleet-local` namespace.

```
# Example
kind: Bundle
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: os-upgrade
  namespace: fleet-local
...
```

3. Apply the **Bundle** resource to your `management cluster`:

```
kubectl apply -f os-upgrade-bundle.yaml
```

4. View the created **Bundle** resource under the `fleet-local` namespace:

```
kubectl get bundles -n fleet-local
```

59.2.4.4.3 SUC Plan deployment - third-party GitOps workflow

There might be use-cases where users would like to incorporate the `OS SUC plans` to their own third-party GitOps workflow (e.g. `Flux`).

To get the OS upgrade resources that you need, first determine the Edge release (https://github.com/suse-edge/fleet-examples/releases) tag of the suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) repository that you would like to use.

After that, resources can be found at `fleets/day2/system-upgrade-controller-plans/os-upgrade`, where:

- `plan-control-plane.yaml` is a SUC plan resource for **control-plane** nodes.
- `plan-worker.yaml` is a SUC plan resource for **worker** nodes.
- `secret.yaml` is a Secret that contains the `upgrade.sh` script, which is responsible for creating the systemd.service (*Section 59.2.4.1.1, "systemd.service"*).
- `config-map.yaml` is a ConfigMap that holds configurations that are consumed by the `upgrade.sh` script.

Important

These `Plan` resources are interpreted by the `System Upgrade Controller` and should be deployed on each downstream cluster that you wish to upgrade. For SUC deployment information, see *Section 20.2, "Installing the System Upgrade Controller"*.

To better understand how your GitOps workflow can be used to deploy the **SUC Plans** for OS upgrade, it can be beneficial to take a look at overview (*Section 59.2.4.2, "Overview"*).

59.2.5 Kubernetes version upgrade

This section describes how to perform a Kubernetes upgrade using *Chapter 9, Fleet* and the *Chapter 20, System Upgrade Controller*.

The following topics are covered as part of this section:

1. *Section 59.2.5.1, "Components"* - additional components used by the upgrade process.
2. *Section 59.2.5.2, "Overview"* - overview of the upgrade process.
3. *Section 59.2.5.3, "Requirements"* - requirements of the upgrade process.
4. *Section 59.2.5.4, "K8s upgrade - SUC plan deployment"* - information on how to deploy `SUC plans`, responsible for triggering the upgrade process.

59.2.5.1 Components

This section covers the custom components that the `K8s upgrade` process uses over the default "Day 2" components (*Section 59.2.1, "Components"*).

59.2.5.1.1 rke2-upgrade

Container image responsible for upgrading the RKE2 version of a specific node.

Shipped through a Pod created by **SUC** based on a **SUC Plan**. The Plan should be located on each **cluster** that is in need of a RKE2 upgrade.

For more information regarding how the `rke2-upgrade` image performs the upgrade, see the upstream (https://github.com/rancher/rke2-upgrade/tree/master) documentation.

59.2.5.1.2 k3s-upgrade

Container image responsible for upgrading the K3s version of a specific node.

Shipped through a Pod created by **SUC** based on a **SUC Plan**. The Plan should be located on each **cluster** that is in need of a K3s upgrade.

For more information regarding how the `k3s-upgrade` image performs the upgrade, see the upstream (https://github.com/k3s-io/k3s-upgrade) documentation.

59.2.5.2 Overview

The Kubernetes distribution upgrade for management cluster nodes is done by utilizing `Fleet` and the `System Upgrade Controller (SUC)`.

`Fleet` is used to deploy and manage `SUC plans` onto the desired cluster.

> **Note**
>
> `SUC plans` are custom resources (https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/) that describe the steps that **SUC** needs to follow in order for a specific task to be executed on a set of nodes. For an example of how an `SUC plan` looks like, refer to the upstream repository (https://github.com/rancher/system-upgrade-controller?tab=readme-ov-file#example-plans).

The `K8s SUC plans` are shipped on each cluster by deploying a GitRepo (https://fleet.rancher.io/gitrepo-add) or Bundle (https://fleet.rancher.io/bundle-add) resource to a specific Fleet workspace (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups). Fleet retrieves the deployed `GitRepo/Bundle` and deploys its contents (the `K8s SUC plans`) to the desired cluster(s).

> **Note**
>
> `GitRepo/Bundle` resources are always deployed on the `management cluster`. Whether to use a `GitRepo` or `Bundle` resource depends on your use-case, check *Section 59.2.2, "Determine your use-case"* for more information.

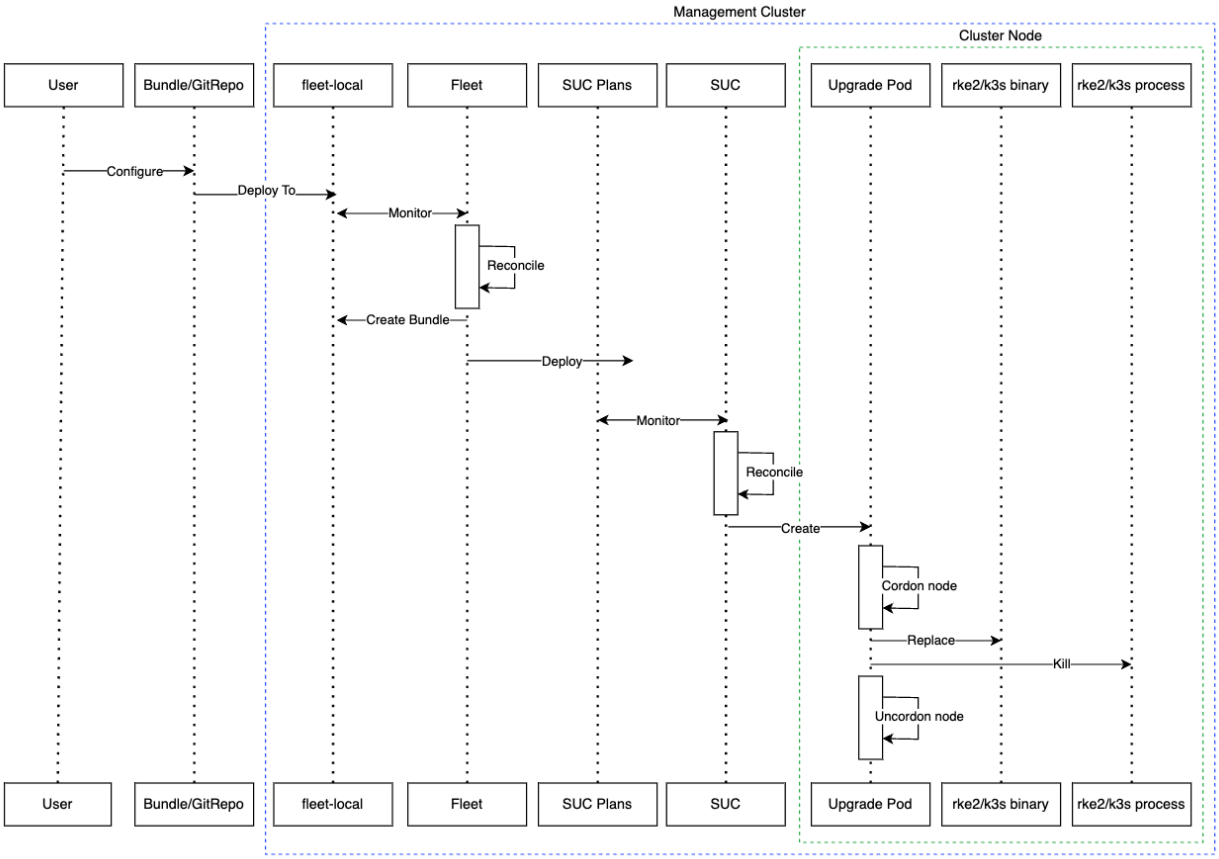
`K8s SUC plans` describe the following workflow:

1. Always cordon (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_cordon/) the nodes before K8s upgrades.
2. Always upgrade `control-plane` nodes before `worker` nodes.
3. Always upgrade the `control-plane` nodes **one** node at a time and the `worker` nodes **two** nodes at a time.

Once the `K8s SUC plans` are deployed, the workflow looks like this:

1. SUC reconciles the deployed `K8s SUC plans` and creates a `Kubernetes Job` on **each node**.
2. Depending on the Kubernetes distribution, the Job will create a Pod that runs either the rke2-upgrade (*Section 59.2.5.1.1, "rke2-upgrade"*) or the k3s-upgrade (*Section 59.2.5.1.2, "k3s-upgrade"*) container image.
3. The created Pod will go through the following workflow:
   a. Replace the existing `rke2/k3s` binary on the node with the one from the `rke2-upgrade/k3s-upgrade` image.
   b. Kill the running `rke2/k3s` process.
4. Killing the `rke2/k3s` process triggers a restart, launching a new process that runs the updated binary, resulting in an upgraded Kubernetes distribution version.

Below you can find a diagram of the above description:



59.2.5.3 Requirements

1. **Backup your Kubernetes distribution:**

   a. For **RKE2 clusters**, see the RKE2 Backup and Restore (https://docs.rke2.io/datastore/backup_restore) documentation.

   b. For **K3s clusters**, see the K3s Backup and Restore (https://docs.k3s.io/datastore/backup-restore) documentation.

2. **Make sure that SUC Plan tolerations match node tolerations** - If your Kubernetes cluster nodes have custom **taints**, make sure to add tolerations (https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/) for those taints in the **SUC Plans**. By default **SUC Plans** have tolerations only for **control-plane** nodes. Default tolerations include:

   - *CriticalAddonsOnly=true:NoExecute*
   - *node-role.kubernetes.io/control-plane:NoSchedule*
   - *node-role.kubernetes.io/etcd:NoExecute*

   Note

   Any additional tolerations must be added under the `.spec.tolerations` section of each Plan. **SUC Plans** related to the Kubernetes version upgrade can be found in the suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) repository under:

   - For **RKE2** - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade`
   - For **K3s** - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade`

   Make sure you use the Plans from a valid repository release (https://github.com/suse-edge/fleet-examples/releases) tag.

   An example of defining custom tolerations for the RKE2 **control-plane** SUC Plan, would look like this:

```
apiVersion: upgrade.cattle.io/v1
kind: Plan
metadata:
```

```
  name: rke2-upgrade-control-plane
spec:
  ...
  tolerations:
  # default tolerations
  - key: "CriticalAddonsOnly"
    operator: "Equal"
    value: "true"
    effect: "NoExecute"
  - key: "node-role.kubernetes.io/control-plane"
    operator: "Equal"
    effect: "NoSchedule"
  - key: "node-role.kubernetes.io/etcd"
    operator: "Equal"
    effect: "NoExecute"
  # custom toleration
  - key: "foo"
    operator: "Equal"
    value: "bar"
    effect: "NoSchedule"
...
```

59.2.5.4 K8s upgrade - SUC plan deployment

Important

For environments previously upgraded using this procedure, users should ensure that **one** of the following steps is completed:

- `Remove any previously deployed SUC Plans related to older Edge release versions from the management cluster` - can be done by removing the desired cluster from the existing `GitRepo/Bundle` target configuration (https://fleet.rancher.io/gitrepo-targets#target-matching), or removing the `GitRepo/Bundle` resource altogether.
- `Reuse the existing GitRepo/Bundle resource` - can be done by pointing the resource's revision to a new tag that holds the correct fleets for the desired `suse-edge/fleet-examples` release (https://github.com/suse-edge/fleet-examples/releases).

This is done in order to avoid clashes between `SUC Plans` for older Edge release versions.

If users attempt to upgrade, while there are existing `SUC Plans` on the management cluster, they will see the following fleet error:

```
Not installed: Unable to continue with install: Plan <plan_name> in namespace
 <plan_namespace> exists and cannot be imported into the current release: invalid
 ownership metadata; annotation validation error..
```

As mentioned in *Section 59.2.5.2, "Overview"*, Kubernetes upgrades are done by shipping `SUC plans` to the desired cluster through one of the following ways:

- Fleet GitRepo resource (*Section 59.2.5.4.1, "SUC plan deployment - GitRepo resource"*)
- Fleet Bundle resource (*Section 59.2.5.4.2, "SUC plan deployment - Bundle resource"*)

To determine which resource you should use, refer to *Section 59.2.2, "Determine your use-case"*.

For use-cases where you wish to deploy the `K8s SUC plans` from a third-party GitOps tool, refer to *Section 59.2.5.4.3, "SUC Plan deployment - third-party GitOps workflow"*

59.2.5.4.1 SUC plan deployment - GitRepo resource

A **GitRepo** resource, that ships the needed `K8s SUC plans`, can be deployed in one of the following ways:

1. Through the `Rancher UI` - *Section 59.2.5.4.1.1, "GitRepo creation - Rancher UI"* (when `Rancher` is available).
2. By manually deploying (*Section 59.2.5.4.1.2, "GitRepo creation - manual"*) the resource to your `management cluster`.

Once deployed, to monitor the Kubernetes upgrade process of the nodes of your targeted cluster, refer to *Section 20.3, "Monitoring System Upgrade Controller Plans"*.

59.2.5.4.1.1 GitRepo creation - Rancher UI

To create a `GitRepo` resource through the Rancher UI, follow their official documentation (https://ranchermanager.docs.rancher.com/v2.15/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui).

The Edge team maintains ready to use fleets for both rke2 (https://github.com/suse-edge/fleet-examples/tree/release-3.7.0/fleets/day2/system-upgrade-controller-plans/rke2-upgrade) and k3s (https://github.com/suse-edge/fleet-examples/tree/release-3.7.0/fleets/day2/system-upgrade-controller-plans/k3s-upgrade) Kubernetes distributions. Depending on your environment, this fleet could be used directly or as a template.

Important

Always use these fleets from a valid Edge release (https://github.com/suse-edge/fleet-examples/releases) tag.

For use-cases where no custom changes need to be included to the `SUC plans` that these fleets ship, users can directly refer the fleets from the `suse-edge/fleet-examples` repository.

In cases where custom changes are needed (e.g. to add custom tolerations), users should refer the fleets from a separate repository, allowing them to add the changes to the SUC plans as required.

Configuration examples for a `GitRepo` resource using the fleets from `suse-edge/fleet-examples` repository:

- RKE2 (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/gitrepos/day2/rke2-upgrade-gitrepo.yaml)
- K3s (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/gitrepos/day2/k3s-upgrade-gitrepo.yaml)

59.2.5.4.1.2 GitRepo creation - manual

1. Pull the **GitRepo** resource:

   - For **RKE2** clusters:

     ```
     curl -o rke2-upgrade-gitrepo.yaml https://raw.githubusercontent.com/suse-edge/
     fleet-examples/refs/tags/release-3.7.0/gitrepos/day2/rke2-upgrade-gitrepo.yaml
     ```

   - For **K3s** clusters:

     ```
     curl -o k3s-upgrade-gitrepo.yaml https://raw.githubusercontent.com/suse-edge/
     fleet-examples/refs/tags/release-3.7.0/gitrepos/day2/k3s-upgrade-gitrepo.yaml
     ```

2. Edit the **GitRepo** configuration:

   - Remove the `spec.targets` section - only needed for downstream clusters.

     - For RKE2:

       ```
       # Example using sed
       sed -i.bak '/^  targets:/,$d' rke2-upgrade-gitrepo.yaml && rm -f rke2-
       upgrade-gitrepo.yaml.bak

       # Example using yq (v4+)
       yq eval 'del(.spec.targets)' -i rke2-upgrade-gitrepo.yaml
       ```

     - For K3s:

       ```
       # Example using sed
       sed -i.bak '/^  targets:/,$d' k3s-upgrade-gitrepo.yaml && rm -f k3s-
       upgrade-gitrepo.yaml.bak

       # Example using yq (v4+)
       yq eval 'del(.spec.targets)' -i k3s-upgrade-gitrepo.yaml
       ```

   - Point the namespace of the `GitRepo` to the `fleet-local` namespace - done in order to deploy the resource on the management cluster.

     - For RKE2:

       ```
       # Example using sed
       sed -i.bak 's/namespace: fleet-default/namespace: fleet-local/' rke2-
       upgrade-gitrepo.yaml && rm -f rke2-upgrade-gitrepo.yaml.bak
       ```

```
# Example using yq (v4+)
yq eval '.metadata.namespace = "fleet-local"' -i rke2-upgrade-
gitrepo.yaml
```

- For K3s:

```
# Example using sed
sed -i.bak 's/namespace: fleet-default/namespace: fleet-local/' k3s-
upgrade-gitrepo.yaml && rm -f k3s-upgrade-gitrepo.yaml.bak

# Example using yq (v4+)
yq eval '.metadata.namespace = "fleet-local"' -i k3s-upgrade-gitrepo.yaml
```

3. Apply the **GitRepo** resources to your `management cluster`:

```
# RKE2
kubectl apply -f rke2-upgrade-gitrepo.yaml

# K3s
kubectl apply -f k3s-upgrade-gitrepo.yaml
```

4. View the created **GitRepo** resource under the `fleet-local` namespace:

```
# RKE2
kubectl get gitrepo rke2-upgrade -n fleet-local

# K3s
kubectl get gitrepo k3s-upgrade -n fleet-local

# Example output
NAME           REPO                                                COMMIT
 BUNDLEDEPLOYMENTS-READY   STATUS
k3s-upgrade    https://github.com/suse-edge/fleet-examples.git   fleet-local   0/0
rke2-upgrade   https://github.com/suse-edge/fleet-examples.git   fleet-local   0/0
```

59.2.5.4.2 SUC plan deployment - Bundle resource

A **Bundle** resource, that ships the needed `Kubernetes upgrade SUC Plans`, can be deployed in one of the following ways:

1. Through the `Rancher UI` - *Section 59.2.5.4.2.1, "Bundle creation - Rancher UI"* (when `Rancher` is available).
2. By manually deploying (*Section 59.2.5.4.2.2, "Bundle creation - manual"*) the resource to your `management cluster`.

Once deployed, to monitor the Kubernetes upgrade process of the nodes of your targeted cluster, refer to *Section 20.3, "Monitoring System Upgrade Controller Plans"*.

59.2.5.4.2.1 Bundle creation - Rancher UI

The Edge team maintains ready to use bundles for both rke2 (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml) and k3s (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml) Kubernetes distributions. Depending on your environment these bundles could be used directly or as a template.

> **Important**
>
> Always use this bundle from a valid Edge release (https://github.com/suse-edge/fleet-examples/releases) tag.

To create a bundle through Rancher's UI:

1. In the upper left corner, click # → **Continuous Delivery**
2. Go to **Advanced** > **Bundles**
3. Select **Create from YAML**
4. From here you can create the Bundle in one of the following ways:

> **Note**
>
> There might be use-cases where you would need to include custom changes to the `SUC plans` that the bundle ships (e.g. to add custom tolerations). Make sure to include those changes in the bundle that will be generated by the below steps.

   a. By manually copying the bundle content for RKE2 (https://raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.7.0/bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml) or K3s (https://

raw.githubusercontent.com/suse-edge/fleet-examples/refs/tags/release-3.7.0/bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml) from `suse-edge/fleet-examples` to the **Create from YAML** page.

b. By cloning the suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples.git) repository from the desired release (https://github.com/suse-edge/fleet-examples/releases) tag and selecting the **Read from File** option in the **Create from YAML** page. From there, navigate to the bundle that you need (`bundles/day2/system-upgrade-controller-plans/rke2-upgrade/plan-bundle.yaml` for RKE2 and `bundles/day2/system-upgrade-controller-plans/k3s-upgrade/plan-bundle.yaml` for K3s). This will auto-populate the **Create from YAML** page with the bundle content.

5. Edit the Bundle in the Rancher UI:

- Change the **namespace** of the `Bundle` to point to the `fleet-local` namespace.

```
# Example
kind: Bundle
apiVersion: fleet.cattle.io/v1alpha1
metadata:
  name: rke2-upgrade
  namespace: fleet-local
...
```

- Change the **target** clusters for the `Bundle` to point to your `local`(management) cluster:

```
spec:
  targets:
  - clusterName: local
```

Note

There are some use-cases where your `local` cluster could have a different name.

To retrieve your `local` cluster name, execute the command below:

```
kubectl get clusters.fleet.cattle.io -n fleet-local
```

6. Select **Create**

59.2.5.4.2.2 Bundle creation - manual

1. Pull the **Bundle** resources:

   - For **RKE2** clusters:

     ```
     curl -o rke2-plan-bundle.yaml https://raw.githubusercontent.com/suse-edge/
     fleet-examples/refs/tags/release-3.7.0/bundles/day2/system-upgrade-controller-
     plans/rke2-upgrade/plan-bundle.yaml
     ```

   - For **K3s** clusters:

     ```
     curl -o k3s-plan-bundle.yaml https://raw.githubusercontent.com/suse-edge/fleet-
     examples/refs/tags/release-3.7.0/bundles/day2/system-upgrade-controller-plans/
     k3s-upgrade/plan-bundle.yaml
     ```

2. Edit the `Bundle` configuration:

   - Change the **target** clusters for the `Bundle` to point to your `local`(management) cluster:

     ```
     spec:
       targets:
       - clusterName: local
     ```

     Note

     There are some use-cases where your `local` cluster could have a different name.

     To retrieve your `local` cluster name, execute the command below:

     ```
     kubectl get clusters.fleet.cattle.io -n fleet-local
     ```

   - Change the **namespace** of the `Bundle` to point to the `fleet-local` namespace.

     ```
     # Example
     kind: Bundle
     apiVersion: fleet.cattle.io/v1alpha1
     metadata:
       name: rke2-upgrade
       namespace: fleet-local
     ...
     ```

3. Apply the **Bundle** resources to your `management cluster`:

```
# For RKE2
kubectl apply -f rke2-plan-bundle.yaml

# For K3s
kubectl apply -f k3s-plan-bundle.yaml
```

4. View the created **Bundle** resource under the `fleet-local` namespace:

```
# For RKE2
kubectl get bundles rke2-upgrade -n fleet-local

# For K3s
kubectl get bundles k3s-upgrade -n fleet-local

# Example output
NAME           BUNDLEDEPLOYMENTS-READY   STATUS
k3s-upgrade    0/0
rke2-upgrade   0/0
```

59.2.5.4.3 SUC Plan deployment - third-party GitOps workflow

There might be use-cases where users would like to incorporate the `Kubernetes upgrade SUC plans` to their own third-party GitOps workflow (e.g. `Flux`).

To get the K8s upgrade resources that you need, first determine the Edge release (https://github.com/suse-edge/fleet-examples/releases) tag of the suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples.git) repository that you would like to use.

After that, the resources can be found at:

- For a RKE2 cluster upgrade:
  - For `control-plane` nodes - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade/plan-control-plane.yaml`
  - For `worker` nodes - `fleets/day2/system-upgrade-controller-plans/rke2-upgrade/plan-worker.yaml`
- For a K3s cluster upgrade:
  - For `control-plane` nodes - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade/plan-control-plane.yaml`
  - For `worker` nodes - `fleets/day2/system-upgrade-controller-plans/k3s-upgrade/plan-worker.yaml`

Important

These `Plan` resources are interpreted by the `System Upgrade Controller` and should be deployed on each downstream cluster that you wish to upgrade. For SUC deployment information, see *Section 20.2, "Installing the System Upgrade Controller"*.

To better understand how your GitOps workflow can be used to deploy the **SUC Plans** for Kubernetes version upgrade, it can be beneficial to take a look at the overview (*Section 59.2.5.2, "Overview"*) of the update procedure using `Fleet`.

59.2.6 Helm chart upgrade

This section covers the following parts:

1. *Section 59.2.6.1, "Preparation for air-gapped environments"* - holds information on how to ship Edge related OCI charts and images to your private registry.
2. *Section 59.2.6.2, "Upgrade procedure"* - holds information on different Helm chart upgrade use-cases and their upgrade procedure.

59.2.6.1 Preparation for air-gapped environments

59.2.6.1.1 Ensure you have access to your Helm chart Fleet

Depending on what your environment supports, you can take one of the following options:

1. Host your chart's Fleet resources on a local Git server that is accessible by your `management cluster`.
2. Use Fleet's CLI to convert a Helm chart into a Bundle (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle) that you can directly use and will not need to be hosted somewhere. Fleet's CLI can be retrieved from their release (https://github.com/rancher/fleet/releases/tag/v0.16.1) page, for Mac users there is a fleet-cli (https://formulae.brew.sh/formula/fleet-cli) Homebrew Formulae.

59.2.6.1.2 Find the required assets for your Edge release version

1. Go to the "Day 2" release (https://github.com/suse-edge/fleet-examples/releases) page and find the Edge release that you want to upgrade your chart to and click **Assets**.
2. From the **"Assets"** section, download the following files:

Release File	Description
edge-save-images.sh	Pulls the images specified in the `edge-release-images.txt` file and packages them inside of a '.tar.gz' archive.
edge-save-oci-artefacts.sh	Pulls the OCI chart images related to the specific Edge release and packages them inside of a '.tar.gz' archive.
edge-load-images.sh	Loads images from a '.tar.gz' archive, retags and pushes them to a private registry.
edge-load-oci-artefacts.sh	Takes a directory containing Edge OCI '.tgz' chart packages and loads them to a private registry.

edge-release-helm-oci-artefacts.txt	Contains a list of OCI chart images related to a specific Edge release.
edge-release-images.txt	Contains a list of images related to a specific Edge release.

59.2.6.1.3 Create the Edge release images archive

On a machine with internet access:

1. Make `edge-save-images.sh` executable:

```
chmod +x edge-save-images.sh
```

2. Generate the image archive:

```
./edge-save-images.sh --source-registry registry.suse.com
```

3. This will create a ready to load archive named `edge-images.tar.gz`.

> **Note**
> If the `-i|--images` option is specified, the name of the archive may differ.

4. Copy this archive to your **air-gapped** machine:

```
scp edge-images.tar.gz <user>@<machine_ip>:/path
```

59.2.6.1.4 Create the Edge OCI chart images archive

On a machine with internet access:

1. Make `edge-save-oci-artefacts.sh` executable:

```
chmod +x edge-save-oci-artefacts.sh
```

2. Generate the OCI chart image archive:

```
./edge-save-oci-artefacts.sh --source-registry registry.suse.com
```

3. This will create an archive named `oci-artefacts.tar.gz`.

> **Note**
>
> If the `-a|--archive` option is specified, the name of the archive may differ.

4. Copy this archive to your **air-gapped** machine:

```
scp oci-artefacts.tar.gz <user>@<machine_ip>:/path
```

59.2.6.1.5 Load Edge release images to your air-gapped machine

On your air-gapped machine:

1. Log into your private registry (if required):

```
podman login <REGISTRY.YOURDOMAIN.COM:PORT>
```

2. Make `edge-load-images.sh` executable:

```
chmod +x edge-load-images.sh
```

3. Execute the script, passing the previously **copied** `edge-images.tar.gz` archive:

```
./edge-load-images.sh --source-registry registry.suse.com --registry
 <REGISTRY.YOURDOMAIN.COM:PORT> --images edge-images.tar.gz
```

> **Note**
>
> This will load all images from the `edge-images.tar.gz`, retag and push them to the registry specified under the `--registry` option.

59.2.6.1.6 Load the Edge OCI chart images to your air-gapped machine

On your air-gapped machine:

1. Log into your private registry (if required):

```
podman login <REGISTRY.YOURDOMAIN.COM:PORT>
```

2. Make `edge-load-oci-artefacts.sh` executable:

```
chmod +x edge-load-oci-artefacts.sh
```

3. Untar the copied `oci-artefacts.tar.gz` archive:

```
tar -xvf oci-artefacts.tar.gz
```

4. This will produce a directory with the naming template `edge-release-oci-tgz-<date>`

5. Pass this directory to the `edge-load-oci-artefacts.sh` script to load the Edge OCI chart images to your private registry:

> **Note**
>
> This script assumes the `helm` CLI has been pre-installed on your environment. For Helm installation instructions, see Installing Helm (https://helm.sh/docs/intro/install/).

```
./edge-load-oci-artefacts.sh --archive-directory edge-release-oci-tgz-<date> --
registry <REGISTRY.YOURDOMAIN.COM:PORT> --source-registry registry.suse.com
```

59.2.6.1.7 Configure your private registry in your Kubernetes distribution

For RKE2, see Private Registry Configuration (https://docs.rke2.io/install/private_registry)

For K3s, see Private Registry Configuration (https://docs.k3s.io/installation/private-registry)

59.2.6.2 Upgrade procedure

This section focuses on the following Helm upgrade procedure use-cases:

1. *Section 59.2.6.2.1, "I have a new cluster and would like to deploy and manage an Edge Helm chart"*
2. *Section 59.2.6.2.2, "I would like to upgrade a Fleet managed Helm chart"*
3. *Section 59.2.6.2.3, "I would like to upgrade a Helm chart deployed via EIB"*

Important

Manually deployed Helm charts cannot be reliably upgraded. We suggest to redeploy the Helm chart using the *Section 59.2.6.2.1, "I have a new cluster and would like to deploy and manage an Edge Helm chart"* method.

59.2.6.2.1 I have a new cluster and would like to deploy and manage an Edge Helm chart

This section covers how to:

1. *Section 59.2.6.2.1.1, "Prepare the fleet resources for your chart"*.
2. *Section 59.2.6.2.1.2, "Deploy the fleet for your chart"*.
3. *Section 59.2.6.2.1.3, "Manage the deployed Helm chart"*.

59.2.6.2.1.1 Prepare the fleet resources for your chart

1. Acquire the chart's Fleet resources from the Edge release (https://github.com/suse-edge/fleet-examples/releases) tag that you wish to use.
2. Navigate to the Helm chart fleet (`fleets/day2/chart-templates/<chart>`)
3. **If you intend to use a GitOps workflow**, copy the chart Fleet directory to the Git repository from where you will do GitOps.

4. **Optionally**, if the Helm chart requires configurations to its **values**, edit the `.helm.values` configuration inside the `fleet.yaml` file of the copied directory.

5. **Optionally**, there may be use-cases where you need to add additional resources to your chart's fleet so that it can better fit your environment. For information on how to enhance your Fleet directory, see Git Repository Contents (https://fleet.rancher.io/gitrepo-content).

Note

In some cases, the default timeout Fleet uses for Helm operations may be insufficient, resulting in the following error:

```
failed pre-install: context deadline exceeded
```

In such cases, add the timeoutSeconds (https://fleet.rancher.io/ref-crds#helmoptions) property under the `helm` configuration of your `fleet.yaml` file.

An **example** for the `longhorn` helm chart would look like:

- User Git repository structure:

```
<user_repository_root>
├── longhorn
│   └── fleet.yaml
└── longhorn-crd
    └── fleet.yaml
```

- `fleet.yaml` content populated with user `Longhorn` data:

```
defaultNamespace: longhorn-system

helm:
  # timeoutSeconds: 10
  releaseName: "longhorn"
  chart: "longhorn"
  repo: "https://charts.rancher.io/"
  version: "1.12.1"
  takeOwnership: true
  # custom chart value overrides
  values:
    # Example for user provided custom values content
    defaultSettings:
      deletingConfirmationFlag: true
```

```
# https://fleet.rancher.io/bundle-diffs
diff:
  comparePatches:
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: engineimages.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: nodes.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
  - apiVersion: apiextensions.k8s.io/v1
    kind: CustomResourceDefinition
    name: volumes.longhorn.io
    operations:
    - {"op":"remove", "path":"/status/conditions"}
    - {"op":"remove", "path":"/status/storedVersions"}
    - {"op":"remove", "path":"/status/acceptedNames"}
```

Note

These are just example values that are used to illustrate custom configurations over the `longhorn` chart. They should **NOT** be treated as deployment guidelines for the `longhorn` chart.

59.2.6.2.1.2 Deploy the fleet for your chart

You can deploy the fleet for your chart by either using a GitRepo (*Section 59.2.6.2.1.2.1, "GitRepo"*) or Bundle (*Section 59.2.6.2.1.2.2, "Bundle"*).

Note

While deploying your Fleet, if you get a `Modified` message, make sure to add a corresponding `comparePatches` entry to the Fleet's `diff` section. For more information, see Generating Diffs to Ignore Modified GitRepos (https://fleet.rancher.io/bundle-diffs).

59.2.6.2.1.2.1 GitRepo

Fleet's GitRepo (https://fleet.rancher.io/ref-gitrepo) resource holds information on how to access your chart's Fleet resources and to which clusters it needs to apply those resources.

The `GitRepo` resource can be deployed through the Rancher UI (https://ranchermanager.docs.rancher.com/v2.15/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui), or manually, by deploying (https://fleet.rancher.io/tut-deployment) the resource to the `management cluster`.

Example **Longhorn** `GitRepo` resource for **manual** deployment:

```
apiVersion: fleet.cattle.io/v1alpha1
kind: GitRepo
metadata:
  name: longhorn-git-repo
  namespace: fleet-local
spec:
  # If using a tag
  # revision: user_repository_tag
  #
  # If using a branch
  # branch: user_repository_branch
  paths:
  # As seen in the 'Prepare your Fleet resources' example
  - longhorn
  - longhorn-crd
  repo: user_repository_url
```

59.2.6.2.1.2.2 Bundle

Bundle (https://fleet.rancher.io/bundle-add) resources hold the raw Kubernetes resources that need to be deployed by Fleet. Normally it is encouraged to use the `GitRepo` approach, but for use-cases where the environment is air-gapped and cannot support a local Git server, `Bundles` can help you in propagating your Helm chart Fleet to your target clusters.

A `Bundle` can be deployed either through the Rancher UI (`Continuous Delivery → Advanced → Bundles → Create from YAML`) or by manually deploying the `Bundle` resource in the correct Fleet namespace. For information about Fleet namespaces, see the upstream documentation (https://fleet.rancher.io/namespaces#gitrepos-bundles-clusters-clustergroups).

`Bundles` for Edge Helm charts can be created by utilizing Fleet's Convert a Helm Chart into a Bundle (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle) approach.

Below you can find an example on how to create a `Bundle` resource from the longhorn (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/fleets/day2/chart-templates/longhorn/longhorn/fleet.yaml) and longhorn-crd (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/fleets/day2/chart-templates/longhorn/longhorn-crd/fleet.yaml) Helm chart fleet templates and manually deploy this bundle to your `management cluster`.

Note

To illustrate the workflow, the below example uses the suse-edge/fleet-examples (https://github.com/suse-edge/fleet-examples) directory structure.

1. Navigate to the longhorn (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/fleets/day2/chart-templates/longhorn/longhorn/fleet.yaml) Chart fleet template:

```
cd fleets/day2/chart-templates/longhorn/longhorn
```

2. Create a `targets.yaml` file that will instruct Fleet to which clusters it should deploy the Helm chart:

```
cat > targets.yaml <<EOF
targets:
# Match your local (management) cluster
- clusterName: local
EOF
```

Note

There are some use-cases where your local cluster could have a different name.

To retrieve your local cluster name, execute the command below:

```
kubectl get clusters.fleet.cattle.io -n fleet-local
```

3. Convert the `Longhorn` Helm chart Fleet to a Bundle resource using the fleet-cli (https://fleet.rancher.io/cli/fleet-cli/fleet).

Note

Fleet's CLI can be retrieved from their release (https://github.com/rancher/fleet/releases/tag/v0.16.1) **Assets** page (`fleet-linux-amd64`).

For Mac users there is a fleet-cli (https://formulae.brew.sh/formula/fleet-cli) Homebrew Formulae.

```
fleet apply --compress --targets-file=targets.yaml -n fleet-local -o - longhorn-
bundle > longhorn-bundle.yaml
```

4. Navigate to the longhorn-crd (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/fleets/day2/chart-templates/longhorn/longhorn-crd/fleet.yaml) Chart fleet template:

```
cd fleets/day2/chart-templates/longhorn/longhorn-crd
```

5. Create a `targets.yaml` file that will instruct Fleet to which clusters it should deploy the Helm chart:

```
cat > targets.yaml <<EOF
targets:
# Match your local (management) cluster
- clusterName: local
EOF
```

6. Convert the `Longhorn CRD` Helm chart Fleet to a Bundle resource using the fleet-cli (https://fleet.rancher.io/cli/fleet-cli/fleet).

```
fleet apply --compress --targets-file=targets.yaml -n fleet-local -o - longhorn-crd-
bundle > longhorn-crd-bundle.yaml
```

7. Deploy the `longhorn-bundle.yaml` and `longhorn-crd-bundle.yaml` files to your `management cluster`:

```
kubectl apply -f longhorn-crd-bundle.yaml
kubectl apply -f longhorn-bundle.yaml
```

Following these steps will ensure that `SUSE Storage` is deployed on all of the specified management cluster.

59.2.6.2.1.3 Manage the deployed Helm chart

Once deployed with Fleet, for Helm chart upgrades, see *Section 59.2.6.2.2, "I would like to upgrade a Fleet managed Helm chart"*.

59.2.6.2.2 I would like to upgrade a Fleet managed Helm chart

1. Determine the version to which you need to upgrade your chart so that it is compatible with the desired Edge release. Helm chart version per Edge release can be viewed from the release notes (*Chapter 78, Release Notes*).
2. In your Fleet monitored Git repository, edit the Helm chart's `fleet.yaml` file with the correct chart **version** and **repository** from the release notes (*Chapter 78, Release Notes*).
3. After committing and pushing the changes to your repository, this will trigger an upgrade of the desired Helm chart

59.2.6.2.3 I would like to upgrade a Helm chart deployed via EIB

Chapter 12, Edge Image Builder deploys Helm charts by creating a `HelmChart` resource and utilizing the `helm-controller` introduced by the RKE2 (https://docs.rke2.io/helm)/K3s (https://docs.k3s.io/helm) Helm integration feature.

To ensure that a Helm chart deployed via `EIB` is successfully upgraded, users need to do an upgrade over the respective `HelmChart` resources.

Below you can find information on:

- The general overview (*Section 59.2.6.2.3.1, "Overview"*) of the upgrade process.
- The necessary upgrade steps (*Section 59.2.6.2.3.2, "Upgrade Steps"*).
- An example (*Section 59.2.6.2.3.3, "Example"*) showcasing a Longhorn (https://longhorn.io) chart upgrade using the explained method.
- How to use the upgrade process with a different GitOps tool (*Section 59.2.6.2.3.4, "Helm chart upgrade using a third-party GitOps tool"*).

59.2.6.2.3.1 Overview

Helm charts that are deployed via `EIB` are upgraded through a `fleet` called eib-charts-upgrader (https://github.com/suse-edge/fleet-examples/tree/release-3.7.0/fleets/day2/eib-charts-upgrader).

This `fleet` processes **user-provided** data to **update** a specific set of HelmChart resources.

Updating these resources triggers the helm-controller (https://github.com/k3s-io/helm-controller), which **upgrades** the Helm charts associated with the modified `HelmChart` resources.

The user is only expected to:

1. Locally pull (https://helm.sh/docs/helm/helm_pull/) the archives for each Helm chart that needs to be upgraded.
2. Pass these archives to the generate-chart-upgrade-data.sh (https://github.com/suse-edge/fleet-examples/blob/release-3.7.0/scripts/day2/generate-chart-upgrade-data.sh) `generate-chart-upgrade-data.sh` script, which will include the data from these archives to the `eib-charts-upgrader` fleet.
3. Deploy the `eib-charts-upgrader` fleet to their `management cluster`. This is done through either a `GitRepo` or `Bundle` resource.

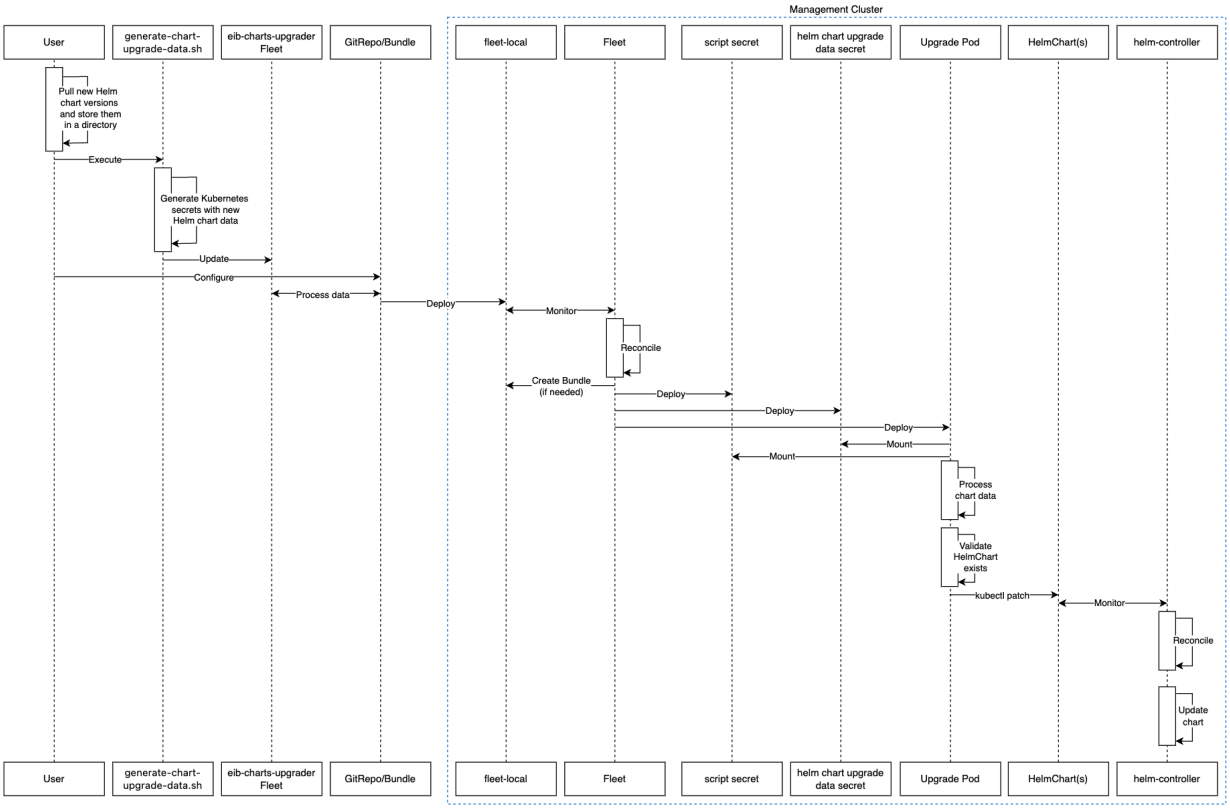
Once deployed, the `eib-charts-upgrader`, with the help of Fleet, will ship its resources to the desired management cluster.

These resources include:

1. A set of `Secrets` holding the **user-provided** Helm chart data.
2. A `Kubernetes Job` which will deploy a `Pod` that will mount the previously mentioned `Secrets` and based on them patch (https://kubernetes.io/docs/reference/kubectl/generated/kubectl_patch/) the corresponding HelmChart resources.

As mentioned previously this will trigger the `helm-controller` which will perform the actual Helm chart upgrade.

Below you can find a diagram of the above description:



59.2.6.2.3.2 Upgrade Steps

1. Clone the `suse-edge/fleet-examples` repository from the correct release tag (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.7.0).

2. Create a directory in which you will store the pulled Helm chart archive(s).

```
mkdir archives
```

3. Inside of the newly created archive directory, pull (https://helm.sh/docs/helm/helm_pull/) the archive(s) for the Helm chart(s) you wish to upgrade:

```
cd archives
helm pull [chart URL | repo/chartname]

# Alternatively if you want to pull a specific version:
# helm pull [chart URL | repo/chartname] --version 0.0.0
```

4. From **Assets** of the desired release tag (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.7.0), download the `generate-chart-upgrade-data.sh` script.

5. Execute the `generate-chart-upgrade-data.sh` script:

```
chmod +x ./generate-chart-upgrade-data.sh

./generate-chart-upgrade-data.sh --archive-dir /foo/bar/archives/ --fleet-path /foo/
bar/fleet-examples/fleets/day2/eib-charts-upgrader
```

   For each chart archive in the `--archive-dir` directory, the script generates a `Kubernetes Secret YAML` file containing the chart upgrade data and stores it in the `base/secrets` directory of the fleet specified by `--fleet-path`.
   The `generate-chart-upgrade-data.sh` script also applies additional modifications to the fleet to ensure the generated `Kubernetes Secret YAML` files are correctly utilized by the workload deployed by the fleet.

> **Important**
> Users should not make any changes over what the `generate-chart-upgrade-data.sh` script generates.

The steps below depend on the environment that you are running:

1. For an environment that supports GitOps (e.g. is non air-gapped, or is air-gapped, but allows for local Git server support):

   a. Copy the `fleets/day2/eib-charts-upgrader` Fleet to the repository that you will use for GitOps.

   > **Note**
   > Make sure that the Fleet includes the changes that have been made by the `generate-chart-upgrade-data.sh` script.

   b. Configure a `GitRepo` resource that will be used to ship all the resources of the `eib-charts-upgrader` Fleet.

i. For `GitRepo` configuration and deployment through the Rancher UI, see Accessing Fleet in the Rancher UI (https://ranchermanager.docs.rancher.com/v2.15/integrations-in-rancher/fleet/overview#accessing-fleet-in-the-rancher-ui).

ii. For `GitRepo` manual configuration and deployment, see Creating a Deployment (https://fleet.rancher.io/tut-deployment).

2. For an environment that does not support GitOps (e.g. is air-gapped and does not allow local Git server usage):

   a. Download the `fleet-cli` binary from the `rancher/fleet` release (https://github.com/rancher/fleet/releases/tag/v0.16.1) page (`fleet-linux-amd64` for Linux). For Mac users, there is a Homebrew Formulae that can be used - fleet-cli (https://formulae.brew.sh/formula/fleet-cli).

   b. Navigate to the `eib-charts-upgrader` Fleet:

```
cd /foo/bar/fleet-examples/fleets/day2/eib-charts-upgrader
```

   c. Create a `targets.yaml` file that will instruct Fleet where to deploy your resources:

```
cat > targets.yaml <<EOF
targets:
# To map the local(management) cluster
- clusterName: local
EOF
```

> **Note**
>
> There are some use-cases where your `local` cluster could have a different name.
>
> To retrieve your `local` cluster name, execute the command below:
>
> ```
> kubectl get clusters.fleet.cattle.io -n fleet-local
> ```

   d. Use the `fleet-cli` to convert the Fleet to a `Bundle` resource:

```
fleet apply --compress --targets-file=targets.yaml -n fleet-local -o - eib-charts-upgrade > bundle.yaml
```

This will create a Bundle (`bundle.yaml`) that will hold all the templated resource from the `eib-charts-upgrader` Fleet.

For more information regarding the `fleet apply` command, see fleet apply (https://fleet.rancher.io/cli/fleet-cli/fleet_apply).

For more information regarding converting Fleets to Bundles, see Convert a Helm Chart into a Bundle (https://fleet.rancher.io/bundle-add#convert-a-helm-chart-into-a-bundle).

e. Deploy the `Bundle`. This can be done in one of two ways:

   i. Through Rancher's UI - Navigate to **Continuous Delivery → Advanced → Bundles → Create from YAML** and either paste the `bundle.yaml` contents, or click the `Read from File` option and pass the file itself.

   ii. Manually - Deploy the `bundle.yaml` file manually inside of your `management cluster`.

Executing these steps will result in a successfully deployed `GitRepo/Bundle` resource. The resource will be picked up by Fleet and its contents will be deployed onto the target clusters that the user has specified in the previous steps. For an overview of the process, refer to *Section 59.2.6.2.3.1, "Overview"*.

For information on how to track the upgrade process, you can refer to *Section 59.2.6.2.3.3, "Example"*.

Important

Once the chart upgrade has been successfully verified, remove the `Bundle/GitRepo` resource.

This will remove the no longer necessary upgrade resources from your `management` cluster, ensuring that no future version clashes might occur.

59.2.6.2.3.3 Example

Note

The example below demonstrates how to upgrade a Helm chart deployed via `EIB` from one version to another on a `management` cluster. Note that the versions used in this example are **not** recommendations. For version recommendations specific to an Edge release, refer to the release notes (*Chapter 78, Release Notes*).

Use-case:

- A `management` cluster is running an older version of Longhorn (https://longhorn.io).
- The cluster has been deployed through EIB, using the following image definition *snippet*:

```
kubernetes:
  helm:
    charts:
    - name: longhorn-crd
      repositoryName: rancher-charts
      targetNamespace: longhorn-system
      createNamespace: true
      version: 104.2.0+up1.7.1
      installationNamespace: kube-system
    - name: longhorn
      repositoryName: rancher-charts
      targetNamespace: longhorn-system
      createNamespace: true
      version: 104.2.0+up1.7.1
      installationNamespace: kube-system
    repositories:
    - name: rancher-charts
      url: https://charts.rancher.io/
...
```

- `SUSE Storage` needs to be upgraded to a version that is compatible with the Edge 3.7 release. Meaning it needs to be upgraded to `1.12.1`.
- It is assumed that the `management cluster` is **air-gapped**, without support for a local Git server and has a working Rancher setup.

Follow the Upgrade Steps (*Section 59.2.6.2.3.2, "Upgrade Steps"*):

1. Clone the `suse-edge/fleet-example` repository from the `release-3.7.0` tag.

```
git clone -b release-3.7.0 https://github.com/suse-edge/fleet-examples.git
```

2. Create a directory where the `Longhorn` upgrade archive will be stored.

```
mkdir archives
```

3. Pull the desired `Longhorn` chart archive version:

```
# First add the Rancher Helm chart repository
helm repo add rancher-charts https://charts.rancher.io/
```

```
# Pull the Longhorn 1.12.1 chart archive
helm pull oci://dp.apps.rancher.io/charts/suse-storage --version 1.12.1
```

4. Outside of the `archives` directory, download the `generate-chart-upgrade-data.sh` script from the `suse-edge/fleet-examples` release tag (https://github.com/suse-edge/fleet-examples/releases/tag/release-3.7.0).

5. Directory setup should look similar to:

```
.
├── archives
│   └── longhorn-1.12.1.tgz
├── fleet-examples
...
│   ├── fleets
│   │   ├── day2
│   │   │   ├── ...
│   │   │   ├── eib-charts-upgrader
│   │   │   │   ├── base
│   │   │   │   │   ├── job.yaml
│   │   │   │   │   ├── kustomization.yaml
│   │   │   │   │   ├── patches
│   │   │   │   │   │   └── job-patch.yaml
│   │   │   │   │   ├── rbac
│   │   │   │   │   │   ├── cluster-role-binding.yaml
│   │   │   │   │   │   ├── cluster-role.yaml
│   │   │   │   │   │   ├── kustomization.yaml
│   │   │   │   │   │   └── sa.yaml
│   │   │   │   │   └── secrets
│   │   │   │   │       ├── eib-charts-upgrader-script.yaml
│   │   │   │   │       └── kustomization.yaml
│   │   │   │   ├── fleet.yaml
│   │   │   │   └── kustomization.yaml
│   │   │   └── ...
│   └── ...
└── generate-chart-upgrade-data.sh
```

6. Execute the `generate-chart-upgrade-data.sh` script:

```
# First make the script executable
chmod +x ./generate-chart-upgrade-data.sh

# Then execute the script
./generate-chart-upgrade-data.sh --archive-dir ./archives --fleet-path ./fleet-
examples/fleets/day2/eib-charts-upgrader
```

The directory structure after the script execution should look similar to:

```
.
├── archives
│   └── longhorn-1.12.1.tgz
├── fleet-examples
...
│   ├── fleets
│   │   ├── day2
│   │   │   ├── ...
│   │   │   ├── eib-charts-upgrader
│   │   │   │   ├── base
│   │   │   │   │   ├── job.yaml
│   │   │   │   │   ├── kustomization.yaml
│   │   │   │   │   ├── patches
│   │   │   │   │   │   └── job-patch.yaml
│   │   │   │   │   ├── rbac
│   │   │   │   │   │   ├── cluster-role-binding.yaml
│   │   │   │   │   │   ├── cluster-role.yaml
│   │   │   │   │   │   ├── kustomization.yaml
│   │   │   │   │   │   └── sa.yaml
│   │   │   │   │   └── secrets
│   │   │   │   │       ├── eib-charts-upgrader-script.yaml
│   │   │   │   │       ├── kustomization.yaml
│   │   │   │   │       ├── longhorn-VERSION.yaml - secret created by the generate-
chart-upgrade-data.sh script
│   │   │   │   │       └── longhorn-crd-VERSION.yaml - secret created by the
 generate-chart-upgrade-data.sh script
│   │   │   │   ├── fleet.yaml
│   │   │   │   └── kustomization.yaml
│   │   │   └── ...
│   └── ...
└── generate-chart-upgrade-data.sh
```

The files changed in git should look like this:

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
 modified:   fleets/day2/eib-charts-upgrader/base/patches/job-patch.yaml
 modified:   fleets/day2/eib-charts-upgrader/base/secrets/kustomization.yaml

Untracked files:
  (use "git add <file>..." to include in what will be committed)
 fleets/day2/eib-charts-upgrader/base/secrets/longhorn-VERSION.yaml
 fleets/day2/eib-charts-upgrader/base/secrets/longhorn-crd-VERSION.yaml
```

7. Create a `Bundle` for the `eib-charts-upgrader` Fleet:

   a. First, navigate to the Fleet itself:

   ```
   cd ./fleet-examples/fleets/day2/eib-charts-upgrader
   ```

   b. Then create a `targets.yaml` file:

   ```
   cat > targets.yaml <<EOF
   targets:
   - clusterName: local
   EOF
   ```

   c. Then use the `fleet-cli` binary to convert the Fleet to a Bundle:

   ```
   fleet apply --compress --targets-file=targets.yaml -n fleet-local -o - eib-charts-upgrade > bundle.yaml
   ```

8. Deploy the Bundle through the Rancher UI:

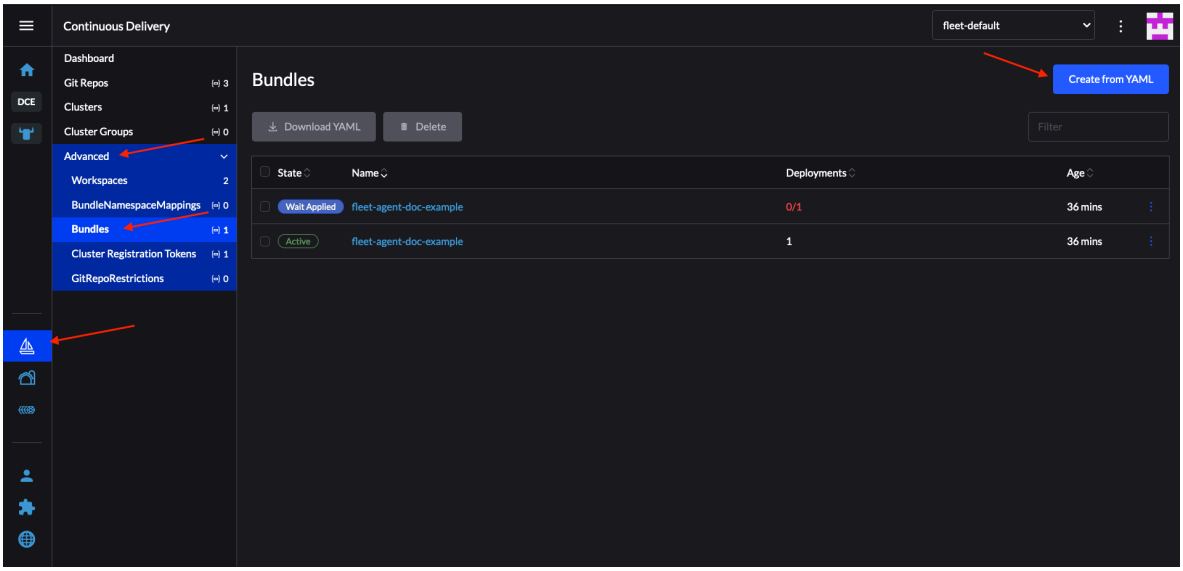


FIGURE 59.1: DEPLOY BUNDLE THROUGH RANCHER UI

From here, select **Read from File** and find the `bundle.yaml` file on your system. This will auto-populate the `Bundle` inside of Rancher's UI.

Select **Create**.

9. After a successful deployment, your Bundle would look similar to:

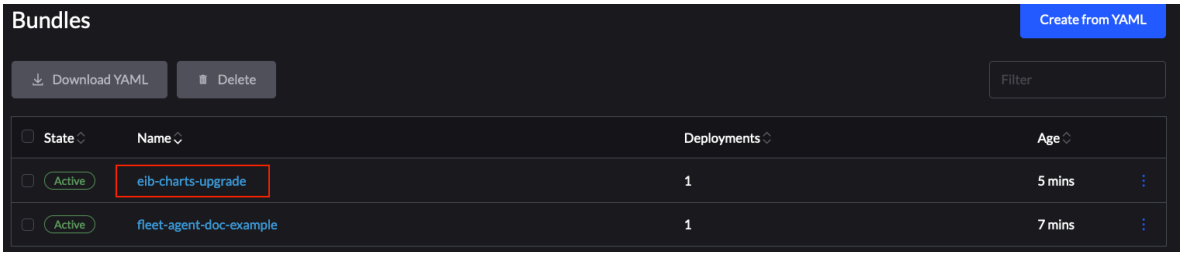
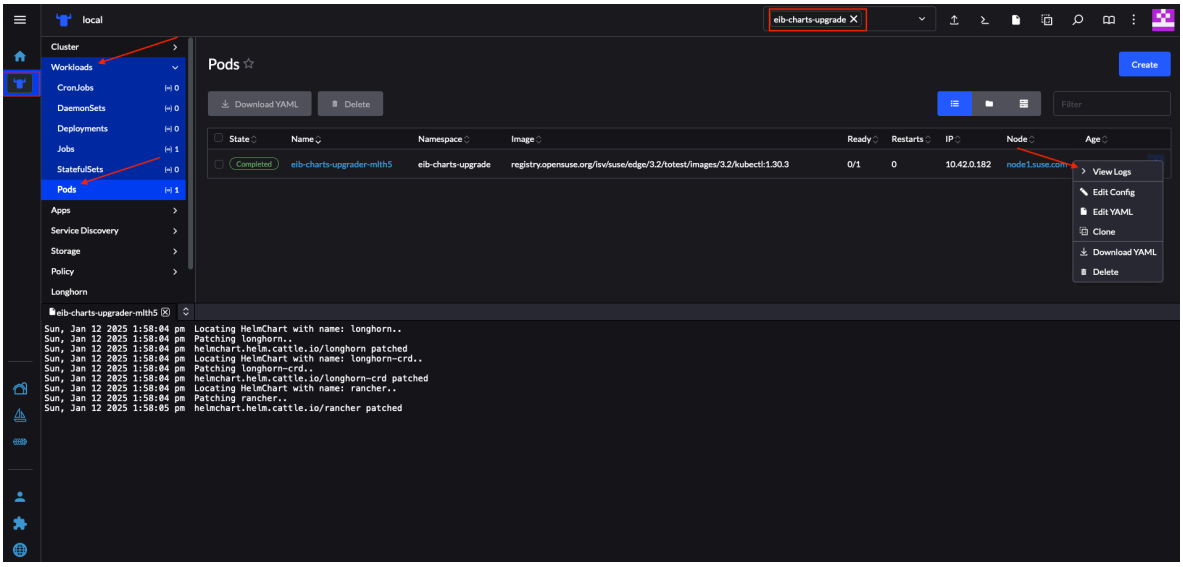


FIGURE 59.2: SUCCESSFULLY DEPLOYED BUNDLE

After the successful deployment of the `Bundle`, to monitor the upgrade process:

1. Verify the logs of the `Upgrade Pod`:



2. Now verify the logs of the Pod created for the upgrade by the helm-controller:

   a. The Pod name will be with the following template - `helm-install-longhorn-<random-suffix>`

   b. The Pod will be in the namespace where the `HelmChart` resource was deployed. In our case this is `kube-system`.

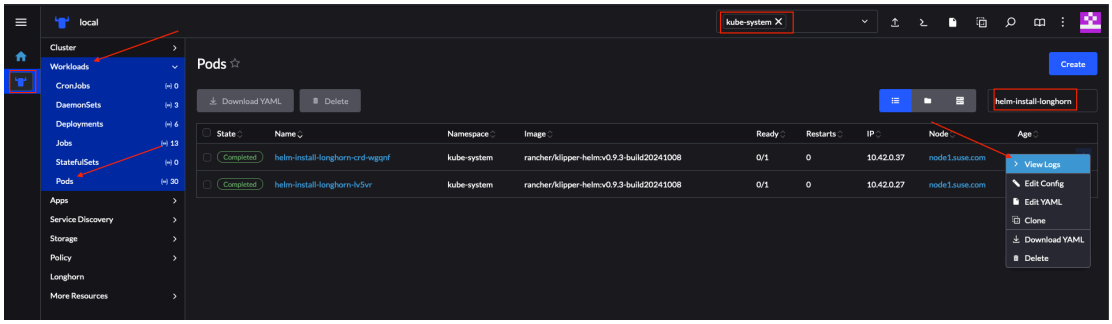


FIGURE 59.3: LOGS FOR SUCCESSFULLY UPGRADED LONGHORN CHART

3. Verify that the `HelmChart` version has been updated by navigating to Rancher's `HelmCharts` section (`More Resources → HelmCharts`). Select the namespace where the chart was deployed, for this example it would be `kube-system`.

4. Finally check that the Longhorn Pods are running.

After making the above validations, it is safe to assume that the Longhorn Helm chart has been upgraded to the `1.12.1` version.

59.2.6.2.3.4 Helm chart upgrade using a third-party GitOps tool

There might be use-cases where users would like to use this upgrade procedure with a GitOps workflow other than Fleet (e.g. `Flux`).

To produce the resources needed for the upgrade procedure, you can use the `generate-chart-upgrade-data.sh` script to populate the `eib-charts-upgrader` Fleet with the user provided data. For more information on how to do this, see *Section 59.2.6.2.3.2, "Upgrade Steps"*.

After you have the full setup, you can use kustomize (https://kustomize.io) to generate a full working solution that you can deploy in your cluster:

```
cd /foo/bar/fleets/day2/eib-charts-upgrader

kustomize build .
```

If you want to include the solution to your GitOps workflow, you can remove the `fleet.yaml` file and use what is left as a valid `Kustomize` setup. Just do not forget to first run the `generate-chart-upgrade-data.sh` script, so that it can populate the `Kustomize` setup with the data for the Helm charts that you wish to upgrade to.

To understand how this workflow is intended to be used, it can be beneficial to look at *Section 59.2.6.2.3.1, "Overview"* and *Section 59.2.6.2.3.2, "Upgrade Steps"*.

60 Lifecycle actions

This section covers the lifecycle management actions for clusters deployed via SUSE Telco Cloud.

60.1 Load Balancer Exclusion

There are many lifecycle actions that require nodes to be drained. During the draining process, all pods will be moved to other nodes in the cluster. After the draining process is finished, the node does not host any services and therefore should not have any traffic routed to it. Load balancers, such as MetalLB, can be made aware of this by applying a label to the node:

```
node.kubernetes.io/exclude-from-external-load-balancers: "true"
```

For more details see: Kubernetes Documentation (https://kubernetes.io/docs/reference/labels-annotations-taints/#node-kubernetes-io-exclude-from-external-load-balancers).

To see the labels on all your nodes in a cluster, you can run:

```
kubectl get nodes -o json | jq -r '.items[].metadata | .name, .labels'
```

In the case of upgrades of downstream clusters, this can be automated by annotating the RKE2ControlPlane on the management cluster:

```
rke2.controlplane.cluster.x-k8s.io/load-balancer-exclusion="true"
```

This immediately creates an annotation on all machine objects on the management cluster for that RKE2ControlPlane.

```
pre-drain.delete.hook.machine.cluster.x-k8s.io/rke2-lb-exclusion: ""
```

With this annotation on the machine objects, any node on the downstream cluster that is scheduled for draining will get the above node label attached prior to the start of the draining process. The label will be removed from the node once it is available and ready again.

60.2 Management cluster upgrades

The upgrade of the management cluster is described in the `Day 2` management cluster (*Chapter 59, Management Cluster*) documentation.

60.3 Downstream cluster upgrades

Upgrading downstream clusters involves updating several components. The following sections cover the upgrade process for each of the components.

Upgrading the operating system

For this process, check the following reference (*Chapter 50, Prepare downstream cluster image for connected scenarios*) to build the new image with a new operating system version. With this new image generated by `EIB`, the next provision phase uses the new operating version provided. In the following step, the new image is used to upgrade the nodes.

Upgrading the RKE2 cluster

The changes required to upgrade the `RKE2` cluster using the automated workflow are the following:

- Change the block `RKE2ControlPlane` in the `capi-provisioning-example.yaml` shown in the following section (*Chapter 52, Downstream cluster provisioning with Directed network provisioning (single-node)* (page 317)):
  - Specify the desired `rolloutStrategy`
    - Note that in a multi-node `RKE2` cluster with at least 3 control plane nodes, setting the `RKE2ControlPlane.spec.rolloutStrategy.type` field to `"RollingUpdate"` and the `rolloutStrategy.rollingUpdate.maxSurge` to `0` ensures no extra nodes being created/needed during the upgrade when the related `Metal3MachineTemplate.spec.nodeReuse` is also set to `True` (as explained below).
  - Change the version of the `RKE2` cluster to the new version replacing `${RKE2_NEW_VERSION}`.

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: multinode-cluster
  namespace: default
spec:
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3MachineTemplate
    name: multinode-cluster-controlplane
  version: ${RKE2_NEW_VERSION}
  replicas: 3
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
```

```
      maxSurge: 0
  serverConfig:
    cni: cilium
    tlsSan:
      - ${EDGE_VIP_ADDRESS_IPV4}
      - ${EDGE_VIP_ADDRESS_IPV6}
      - https://${EDGE_VIP_ADDRESS_IPV4}.sslip.io
      - https://${EDGE_VIP_ADDRESS_IPV6}.sslip.io
  rolloutStrategy:
    rollingUpdate:
      maxSurge: 0
  registrationMethod: "control-plane-endpoint"
  registrationAddress: ${EDGE_VIP_ADDRESS}
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        systemd:
          units:
          - name: rke2-preinstall.service
            enabled: true
            contents: |
              [Unit]
              Description=rke2-preinstall
              Wants=network-online.target
              Before=rke2-install.service
              ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
              [Service]
              Type=oneshot
              User=root
              ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
              # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
              ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /mnt/
openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/openstack/
latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
              ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
              ExecStartPost=/bin/sh -c "umount /mnt"
              [Install]
              WantedBy=multi-user.target
        storage:
```

```
        directories:
        - path: /var/lib/rancher/rke2/server/manifests
          overwrite: true
        files:
          - path: /var/lib/rancher/rke2/server/manifests/endpoint-copier-operator.yaml
            overwrite: true
            contents:
              inline: |
                apiVersion: helm.cattle.io/v1
                kind: HelmChart
                metadata:
                  name: endpoint-copier-operator
                  namespace: kube-system
                spec:
                  chart: oci://registry.suse.com/edge/charts/endpoint-copier-operator
                  targetNamespace: endpoint-copier-operator
                  version: {version-endpoint-copier-operator-chart}
                  createNamespace: true
          - path: /var/lib/rancher/rke2/server/manifests/metallb.yaml
            overwrite: true
            contents:
              inline: |
                apiVersion: helm.cattle.io/v1
                kind: HelmChart
                metadata:
                  name: metallb
                  namespace: kube-system
                spec:
                  chart: oci://registry.suse.com/edge/charts/metallb
                  targetNamespace: metallb-system
                  version: {version-metallb-chart}
                  createNamespace: true
          - path: /var/lib/rancher/rke2/server/manifests/metallb-cr.yaml
            overwrite: true
            contents:
              inline: |
                apiVersion: metallb.io/v1beta1
                kind: IPAddressPool
                metadata:
                  name: kubernetes-vip-ip-pool
                  namespace: metallb-system
                spec:
                  addresses:
                    - ${EDGE_VIP_ADDRESS_IPV4}/32
                    - ${EDGE_VIP_ADDRESS_IPV6}/128
                  serviceAllocation:
                    priority: 100
```

```
                    namespaces:
                      - default
                    serviceSelectors:
                      - matchExpressions:
                        - {key: "serviceType", operator: In, values: [kubernetes-vip]}
                ---
                apiVersion: metallb.io/v1beta1
                kind: L2Advertisement
                metadata:
                  name: ip-pool-l2-adv
                  namespace: metallb-system
                spec:
                  ipAddressPools:
                    - kubernetes-vip-ip-pool
          - path: /var/lib/rancher/rke2/server/manifests/endpoint-svc.yaml
            overwrite: true
            contents:
              inline: |
                apiVersion: v1
                kind: Service
                metadata:
                  name: kubernetes-vip
                  namespace: default
                  labels:
                    serviceType: kubernetes-vip
                spec:
                  ipFamilyPolicy: PreferDualStack
                  ports:
                  - name: rke2-api
                    port: 9345
                    protocol: TCP
                    targetPort: 9345
                  - name: k8s-api
                    port: 6443
                    protocol: TCP
                    targetPort: 6443
                  type: LoadBalancer

################################################################################
          # In case of deploying the RKE2 ingress controller, the following 2 files
must
          # be added.
          # NOTE: The .providers stanza can be removed from the Helm parameters
settings
          #       here below in case of not deploying/not planning to deploy Ingress
          #       objects including ingress-nginx annotations in its `.metadata`
stanza.
```

```
##############################################################################
           - path: /var/lib/rancher/rke2/server/manifests/rke2-traefik-config.yaml
             overwrite: true
             contents:
               inline: |
                 apiVersion: helm.cattle.io/v1
                 kind: HelmChartConfig
                 metadata:
                   name: rke2-traefik
                   namespace: kube-system
                 spec:
                   valuesContent: |-
                     ingressClass:
                       isDefaultClass: false
                     ports:
                       web:
                         hostPort: null    # disallow hostPort
                         exposedPort: 80
                       websecure:
                         hostPort: null    # disallow hostPort
                         exposedPort: 443
                     service:
                       enabled: true
                       spec:
                         type: LoadBalancer
                         ipFamilyPolicy: PreferDualStack
                         externalTrafficPolicy: Local
                         allocateLoadBalancerNodePorts: false
                     providers:
                       kubernetesIngressNGINX:  # this provider allows Traefik to
"understand" most of the Ingress-NGINX annotations
                         enabled: true
                         ingressClass: "rke2-ingress-nginx-migration"
                         controllerClass: "rke2.cattle.io/ingress-nginx-migration"
             mode: 0644
             user:
               name: root
             group:
               name: root
           - path: /var/lib/rancher/rke2/server/manifests/metallb-traefik-cr.yaml
             overwrite: true
             contents:
               inline: |
                 apiVersion: metallb.io/v1beta1
                 kind: IPAddressPool
                 metadata:
```

```
                    name: ingress-ippool
                    namespace: metallb-system
                  spec:
                    addresses:
                      - ${INGRESS_IPV4}/32
                      - ${INGRESS_IPV6}/128
                    serviceAllocation:
                      priority: 100
                      serviceSelectors:
                      - matchExpressions:
                        - {key: app.kubernetes.io/name, operator: In, values: [rke2-
traefik]} # assign to rke2-traefik LB svc
                  ---
                  apiVersion: metallb.io/v1beta1
                  kind: L2Advertisement
                  metadata:
                    name: ingress-l2-adv
                    namespace: metallb-system
                  spec:
                    ipAddressPools:
                      - ingress-ippool
              mode: 0644
              user:
                name: root
              group:
                name: root
    kubelet:
      extraArgs:
      - provider-id=metal3://PROVIDERID
    nodeName: "localhost.localdomain"
```

- Change the block `Metal3MachineTemplate` in the `capi-provisioning-example.yaml` shown in the following section (*Chapter 52, Downstream cluster provisioning with Directed network provisioning (single-node)* (page 317)):
  - Change the image name and checksum to the new version generated in the previous step.
  - Add the directive `nodeReuse` to `true` to avoid creating a new node.
  - Add the directive `automatedCleaningMode` to `metadata` to enable the automated cleaning for the node.

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: multinode-cluster-controlplane
```

```
  namespace: default
spec:
  nodeReuse: True
  template:
    spec:
      automatedCleaningMode: metadata
      dataTemplate:
        name: multinode-cluster-controlplane-template
      hostSelector:
        matchLabels:
          cluster-role: control-plane
      image:
        checksum: http://imagecache.local:8080/${NEW_IMAGE_GENERATED}.sha256
        checksumType: sha256
        diskFormat: raw
        url: http://imagecache.local:8080/${NEW_IMAGE_GENERATED}.raw
```

Before applying the `capi-provisioning-example.yaml` file, it is always a good practice to inform external load balancers (e.g. MetalLB) about nodes being drained so that they do not route traffic to nodes in this state. As mentioned in the *Section 60.1, "Load Balancer Exclusion"* section, you can automate this by annotating the RKE2ControlPlane on the management cluster. In this example, an RKE2ControlPlane object called multinode-cluster is annotated:

```
kubectl annotate  RKE2ControlPlane/multinode-cluster  rke2.controlplane.cluster.x-k8s.io/
load-balancer-exclusion="true"
```

Verify that the machine objects have been annotated:

```
pre-drain.delete.hook.machine.cluster.x-k8s.io/rke2-lb-exclusion: ""
```

Fetch the annotations for all your machine objects:

```
kubectl get machines -o json | jq -r '.items[].metadata | .name, .annotations'
```

Note

Without these annotations users might experience longer response times for services as the load-balancers are unaware of drained nodes.

After making these changes, the `capi-provisioning-example.yaml` file can be applied to the cluster using the following command:

```
kubectl apply -f capi-provisioning-example.yaml
```

60.4 Longhorn Engine Image Cleanup

This section describes how to implement automated cleanup of obsolete Longhorn engine images that accumulate after cluster upgrades.

60.4.1 Problem statement

Longhorn does not automatically remove old engine images after upgrades. This is an intentional design decision to prevent accidental data loss — Longhorn retains previous engine image versions as a safety measure in case rollback is required.

However, in production environments with frequent upgrades, these unused engine images accumulate over time, resulting in:

- **Storage consumption**: Each engine image consumes disk space on cluster nodes
- **Resource overhead**: Kubernetes manages additional image resources unnecessarily
- **Operational complexity**: Manual identification and cleanup of obsolete images

60.4.2 When to implement cleanup

Engine image cleanup is recommended in the following scenarios:

- **Regular cluster upgrades**: Environments that perform frequent Longhorn version upgrades
- **Storage-constrained deployments**: Clusters with limited node storage capacity
- **Operational hygiene**: As part of routine Day 2 maintenance procedures

Note

This cleanup procedure is classified as **low risk** because it only removes engine images with zero active references (`refCount == 0`). Images currently in use by volumes are protected and will not be deleted.

60.4.3 Solution architecture

The cleanup solution uses a Kubernetes CronJob that executes periodic maintenance tasks:

1. **Discovery**: Identifies the latest Longhorn engine image version deployed in the cluster
2. **Candidate selection**: Locates all older engine image versions with no active volume references
3. **Cleanup execution**: Deletes obsolete engine images that meet safety criteria
4. **Audit logging**: Records cleanup operations for operational visibility

The implementation requires:

- **ServiceAccount**: For authentication to the Kubernetes API
- **RBAC permissions**: Read and delete access to `engineimages.longhorn.io` resources
- **CronJob**: Scheduled execution container with cleanup logic

60.4.4 Implementation

The complete cleanup solution consists of four Kubernetes manifests deployed to the `longhorn-system` namespace.

1. **Create ServiceAccount**
   Create a dedicated ServiceAccount for the cleanup CronJob:

   ```
   apiVersion: v1
   kind: ServiceAccount
   metadata:
     name: longhorn-engineimage-cleanup
     namespace: longhorn-system
     labels:
       app: longhorn-engineimage-cleanup
   ```

2. **Configure RBAC permissions**

   a. Define a Role with minimal required permissions:

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: Role
   metadata:
     name: longhorn-engineimage-cleanup
     namespace: longhorn-system
     labels:
   ```

```
    app: longhorn-engineimage-cleanup
rules:
- apiGroups: [longhorn.io]
  resources: [engineimages]
  verbs: [get, list, delete]
```

b. Bind the Role to the ServiceAccount:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: longhorn-engineimage-cleanup
  namespace: longhorn-system
  labels:
    app: longhorn-engineimage-cleanup
subjects:
- kind: ServiceAccount
  name: longhorn-engineimage-cleanup
  namespace: longhorn-system
roleRef:
  kind: Role
  name: longhorn-engineimage-cleanup
  apiGroup: rbac.authorization.k8s.io
```

3. **Deploy cleanup CronJob**

   Deploy the automated cleanup CronJob:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: longhorn-engineimage-cleanup
  namespace: longhorn-system
  labels:
    app: longhorn-engineimage-cleanup
  annotations:
    description: "Cleanup old Longhorn engine images left over after upgrades"
    source: "SUSE Edge Telco Cloud - Day 2 Operations"
spec:
  # Execute weekly on Sunday at 2:00 AM UTC
  schedule: "0 2 * * 0"
  successfulJobsHistoryLimit: 3
  failedJobsHistoryLimit: 3
  concurrencyPolicy: Forbid
  jobTemplate:
    spec:
      backoffLimit: 2
      activeDeadlineSeconds: 600
```

```
      template:
        metadata:
          labels:
            app: longhorn-engineimage-cleanup
        spec:
          serviceAccountName: longhorn-engineimage-cleanup
          restartPolicy: Never
          automountServiceAccountToken: true
          containers:
          - name: cleanup
            image: registry.suse.com/bci/bci-base:15.6
            command:
            - sh
            - -c
            - |
              set -e

              # Install required tools
              zypper --non-interactive install curl jq

              # Install kubectl from official Kubernetes
              curl -LO "https://dl.k8s.io/release/v1.36.3/bin/linux/amd64/kubectl"
              chmod +x kubectl
              mv kubectl /usr/local/bin/

              echo "=== Longhorn Engine Image Cleanup ==="
              echo "Automated maintenance - Day 2 Operations"
              echo ""

              echo "Step 1: Discovering latest engine image version..."

              ENGINE_IMAGES=$(kubectl get engineimages.longhorn.io -n longhorn-
system -o json)

              LATEST_VERSION=$(echo "$ENGINE_IMAGES" | jq -r '
                .items[]
                | select(.status.version != null)
                | .status.version' |
                sed 's/^v//' |
                sort -V |
                tail -1)

              if [ -z "$LATEST_VERSION" ]; then
                echo "No engine images found. Nothing to cleanup."
                exit 0
              fi
```

```
              echo "Latest version detected: v${LATEST_VERSION}"
              echo ""

              echo "Step 2: Identifying obsolete engine images..."

              TO_DELETE=$(echo "$ENGINE_IMAGES" | jq -r --arg latest "v
${LATEST_VERSION}" '
                .items[]
                | select(.status.version != $latest)
                | select(.status.refCount == 0)
                | .metadata.name')

              if [ -z "$TO_DELETE" ]; then
                echo "No obsolete engine images found"
                echo "All engine images are either:"
                echo "  - Latest version (v${LATEST_VERSION})"
                echo "  - Still in use (refCount > 0)"
                exit 0
              fi

              DELETE_COUNT=$(echo "$TO_DELETE" | wc -l | tr -d ' ')
              echo "Found $DELETE_COUNT obsolete engine image(s) for deletion:"
              echo "$TO_DELETE"
              echo ""

              COUNT=0
              FAILED=0
              for IMAGE_NAME in $TO_DELETE; do
                VERSION=$(echo "$ENGINE_IMAGES" | jq -r --arg name "$IMAGE_NAME" \
                  '.items[] | select(.metadata.name == $name) | .status.version')
                REFCOUNT=$(echo "$ENGINE_IMAGES" | jq -r --arg name "$IMAGE_NAME" \
                  '.items[] | select(.metadata.name == $name) | .status.refCount')

                echo "Deleting: $IMAGE_NAME (version: $VERSION, refCount:
 $REFCOUNT)"

                if kubectl delete engineimages.longhorn.io "$IMAGE_NAME" -n
 longhorn-system; then
                  COUNT=$((COUNT + 1))
                  echo "  ✓ Deleted successfully"
                else
                  FAILED=$((FAILED + 1))
                  echo "  ✗ Failed to delete"
                fi
              done

              echo ""
```

```
              echo "=== Cleanup Summary ==="
              echo "Latest version: v${LATEST_VERSION}"
              echo "Successfully deleted: $COUNT obsolete engine images"
              echo "Failed deletions: $FAILED engine images"

              if [ $FAILED -gt 0 ]; then
                exit 1
              fi

              echo ""
              echo "Cleanup completed successfully"
```

4. **Apply the configuration**

   a. Save all four manifests to a single file (e.g. `longhorn-engine-cleanup.yaml`), then deploy them to the cluster:

   ```
   kubectl apply -f longhorn-engine-cleanup.yaml
   ```

   b. Verify the CronJob was created:

   ```
   kubectl get cronjob -n longhorn-system longhorn-engineimage-cleanup
   ```

   Expected output:

   ```
   NAME                           SCHEDULE     SUSPEND   ACTIVE   LAST SCHEDULE
    AGE
   longhorn-engineimage-cleanup    0 2 * * 0    False     0        <none>
     5s
   ```

60.4.5 Operational procedures

This section describes the day-to-day operational tasks for managing the Longhorn engine image cleanup CronJob, including manual execution, schedule customization, monitoring, and verification.

60.4.5.1 Manual execution

To execute cleanup immediately without waiting for the scheduled time, run the following command:

```
kubectl create job --from=cronjob/longhorn-engineimage-cleanup \
  longhorn-engineimage-cleanup-manual-$(date +%Y%m%d-%H%M%S) \
  -n longhorn-system
```

Monitor manual execution:

1. Watch job status:

```
kubectl get jobs -n longhorn-system -w
```

2. View job logs:

```
kubectl logs -n longhorn-system -l app=longhorn-engineimage-cleanup --tail=100
```

60.4.5.2 Schedule configuration

The default schedule executes cleanup weekly on Sunday at 2:00 AM UTC:

```
schedule: "0 2 * * 0"
```

Common schedule alternatives:

- **Daily at 3:00 AM**: `"0 3 * * *"`
- **Monthly on the 1st at 4:00 AM**: `"0 4 1 * *"`
- **Every 6 hours**: `"0 */6 * * *"`

Modify the schedule:

```
kubectl patch cronjob longhorn-engineimage-cleanup \
  -n longhorn-system \
  --type='merge' \
  -p '{"spec":{"schedule":"0 3 * * *"}}'
```

60.4.5.3 Monitoring and logs

View cleanup history:

- List completed jobs:

```
kubectl get jobs -n longhorn-system -l app=longhorn-engineimage-cleanup
```

- View logs from last successful run:

```
kubectl logs -n longhorn-system \
```

```
  -l app=longhorn-engineimage-cleanup \
  --tail=200 \
  | grep -A 50 "Cleanup Summary"
```

Inspect specific job execution:

```
# Get job name
JOB_NAME=$(kubectl get jobs -n longhorn-system \
  -l app=longhorn-engineimage-cleanup \
  --sort-by=.metadata.creationTimestamp \
  -o jsonpath='{.items[-1].metadata.name}')

# View job details
kubectl describe job $JOB_NAME -n longhorn-system

# View job logs
kubectl logs job/$JOB_NAME -n longhorn-system
```

60.4.5.4 Verification

After cleanup execution, verify that only current engine images remain:

1. List all engine images:

   ```
   kubectl get engineimages.longhorn.io -n longhorn-system
   ```

2. Check reference counts:

   ```
   kubectl get engineimages.longhorn.io -n longhorn-system \
     -o custom-
   columns=NAME:.metadata.name,VERSION:.status.version,REFCOUNT:.status.refCount
   ```

Expected output showing only the latest version with active references:

```
NAME                                VERSION     REFCOUNT
ei-acba2c08                         v1.12.1     3
```

IX How-To Guides

61 MetalLB on K3s (using Layer 2 Mode) 449

62 MetalLB on K3s (using Layer 3 Mode) 458

63 MetalLB in front of the Kubernetes API server 464

64 Air-gapped deployments with Edge Image Builder 471

65 Building Updated SUSE Linux Micro Images with Kiwi 498

66 SUSE Storage V2 Data Engine 503

67 Using clusterclass to deploy downstream clusters 509

68 Longhorn Volume Encryption 528

How-to guides and best practices

61 MetalLB on K3s (using Layer 2 Mode)

MetalLB is a load-balancer implementation for bare-metal Kubernetes clusters, using standard routing protocols.

In this guide, we demonstrate how to deploy MetalLB in layer 2 (L2) mode.

61.1 Why use MetalLB

MetalLB is a compelling choice for load balancing in bare-metal Kubernetes clusters for several reasons:

1. Native Integration with Kubernetes: MetalLB seamlessly integrates with Kubernetes, making it easy to deploy and manage using familiar Kubernetes tools and practices.
2. Bare-Metal Compatibility: Unlike cloud-based load balancers, MetalLB is designed specifically for on-premises deployments where traditional load balancers might not be available or feasible.
3. Supports Multiple Protocols: MetalLB supports both Layer 2 and BGP (Border Gateway Protocol) modes, providing flexibility for different network architectures and requirements.
4. High Availability: By distributing load-balancing responsibilities across multiple nodes, MetalLB ensures high availability and reliability for your services.
5. Scalability: MetalLB can handle large-scale deployments, scaling alongside your Kubernetes cluster to meet increasing demand.

In layer 2 mode, one node assumes the responsibility of advertising a service to the local network. From the network's perspective, it simply looks like that machine has multiple IP addresses assigned to its network interface.

The major advantage of the layer 2 mode is its universality: it works on any Ethernet network, with no special hardware required, not even fancy routers.

61.2 MetalLB on K3s (using L2)

In this quick start, L2 mode will be used. This means we do not need any special network equipment but three free IPs within the network range.

61.3 Prerequisites

- A K3s cluster where MetalLB is going to be deployed.

Warning

K3S comes with its own service load balancer named Klipper. You need to disable it to run MetalLB (https://metallb.universe.tf/configuration/k3s/). To disable Klipper, K3s needs to be installed using the `--disable=servicelb` flag.

- Helm
- Three free IP adressess within the network range. In this example `192.168.122.10-192.168.122.12`

Important

You must make sure these IP addresses are unassigned. In a DHCP environment these addresses must not be part of the DHCP pool to avoid dual assignments.

61.4 Deployment

We will be using the MetalLB Helm chart published as part of the SUSE Telco Cloud solution:

```
helm install \
  metallb oci://registry.suse.com/edge/charts/metallb \
  --namespace metallb-system \
  --create-namespace

while ! kubectl wait --for condition=ready -n metallb-system $(kubectl get\
 pods -n metallb-system -l app.kubernetes.io/component=controller -o name)\
 --timeout=10s; do
 sleep 2
done
```

61.5 Configuration

At this point, the installation is completed. Now it is time to configure (https://metallb.universe.tf/configuration/) using our example values:

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: ip-pool
  namespace: metallb-system
spec:
  addresses:
  - 192.168.122.10/32
  - 192.168.122.11/32
  - 192.168.122.12/32
EOF
```

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: ip-pool-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
  - ip-pool
EOF
```

Now, it is ready to be used. You can customize many things for L2 mode, such as:

- IPv6 And Dual Stack Services (https://metallb.universe.tf/usage/#ipv6-and-dual-stack-services)
- Control automatic address allocation (https://metallb.universe.tf/configuration/_advanced_ipaddresspool_configuration/#controlling-automatic-address-allocation)
- Reduce the scope of address allocation to specific namespaces and services (https://metallb.universe.tf/configuration/_advanced_ipaddresspool_configuration/#reduce-scope-of-address-allocation-to-specific-namespace-and-service)

- Limiting the set of nodes where the service can be announced from (https://metallb.universe.tf/configuration/_advanced_l2_configuration/#limiting-the-set-of-nodes-where-the-service-can-be-announced-from)
- Specify network interfaces that LB IP can be announced from (https://metallb.universe.tf/configuration/_advanced_l2_configuration/#specify-network-interfaces-that-lb-ip-can-be-announced-from)

And a lot more for BGP (https://metallb.universe.tf/configuration/_advanced_bgp_configuration/).

61.5.1 Traefik and MetalLB

Traefik is deployed by default with K3s (it can be disabled (https://docs.k3s.io/networking#traefik-ingress-controller) with `--disable=traefik`) and it is by default exposed as `LoadBalancer` (to be used with Klipper). However, as Klipper needs to be disabled, Traefik service for ingress is still a `LoadBalancer` type. So at the moment of deploying MetalLB, the first IP will be assigned automatically to Traefik Ingress.

```
# Before deploying MetalLB
kubectl get svc -n kube-system traefik
NAME      TYPE           CLUSTER-IP     EXTERNAL-IP   PORT(S)                      AGE
traefik   LoadBalancer   10.43.44.113   <pending>     80:31093/TCP,443:32095/TCP   28s
# After deploying MetalLB
kubectl get svc -n kube-system traefik
NAME      TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)                      AGE
traefik   LoadBalancer   10.43.44.113   192.168.122.10   80:31093/TCP,443:32095/TCP
 3m10s
```

This will be applied later (*Section 61.6.1, "Ingress with MetalLB"*) in the process.

61.6 Usage

Let us create an example deployment:

```
cat <<- EOF | kubectl apply -f -
---
apiVersion: v1
kind: Namespace
metadata:
  name: hello-kubernetes
---
apiVersion: v1
```

```
kind: ServiceAccount
metadata:
  name: hello-kubernetes
  namespace: hello-kubernetes
  labels:
    app.kubernetes.io/name: hello-kubernetes
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-kubernetes
  namespace: hello-kubernetes
  labels:
    app.kubernetes.io/name: hello-kubernetes
spec:
  replicas: 2
  selector:
    matchLabels:
      app.kubernetes.io/name: hello-kubernetes
  template:
    metadata:
      labels:
        app.kubernetes.io/name: hello-kubernetes
    spec:
      serviceAccountName: hello-kubernetes
      containers:
        - name: hello-kubernetes
          image: "paulbouwer/hello-kubernetes:1.10"
          imagePullPolicy: IfNotPresent
          ports:
            - name: http
              containerPort: 8080
              protocol: TCP
          livenessProbe:
            httpGet:
              path: /
              port: http
          readinessProbe:
            httpGet:
              path: /
              port: http
          env:
          - name: HANDLER_PATH_PREFIX
            value: ""
          - name: RENDER_PATH_PREFIX
            value: ""
          - name: KUBERNETES_NAMESPACE
```

```
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace
          - name: KUBERNETES_POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: KUBERNETES_NODE_NAME
            valueFrom:
              fieldRef:
                fieldPath: spec.nodeName
          - name: CONTAINER_IMAGE
            value: "paulbouwer/hello-kubernetes:1.10"
EOF
```

And finally, the service:

```
cat <<- EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: hello-kubernetes
  namespace: hello-kubernetes
  labels:
    app.kubernetes.io/name: hello-kubernetes
spec:
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: hello-kubernetes
EOF
```

Let us see it in action:

```
kubectl get svc -n hello-kubernetes
NAME               TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)        AGE
hello-kubernetes   LoadBalancer   10.43.127.75   192.168.122.11   80:31461/TCP   8s

curl http://192.168.122.11
<!DOCTYPE html>
<html>
<head>
    <title>Hello Kubernetes!</title>
    <link rel="stylesheet" type="text/css" href="/css/main.css">
```

```
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Ubuntu:300" >
</head>
<body>

  <div class="main">
    <img src="/images/kubernetes.png"/>
    <div class="content">
      <div id="message">
  Hello world!
</div>
<div id="info">
  <table>
    <tr>
      <th>namespace:</th>
      <td>hello-kubernetes</td>
    </tr>
    <tr>
      <th>pod:</th>
      <td>hello-kubernetes-7c8575c848-2c6ps</td>
    </tr>
    <tr>
      <th>node:</th>
      <td>allinone (Linux 5.14.21-150400.24.46-default)</td>
    </tr>
  </table>
</div>
<div id="footer">
  paulbouwer/hello-kubernetes:1.10 (linux/amd64)
</div>
    </div>
  </div>

</body>
</html>
```

61.6.1 Ingress with MetalLB

As Traefik is already serving as an ingress controller, we can expose any HTTP/HTTPS traffic via an `Ingress` object such as:

```
IP=$(kubectl get svc -n kube-system traefik -o
 jsonpath="{.status.loadBalancer.ingress[0].ip}")
cat <<- EOF | kubectl apply -f -
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
  name: hello-kubernetes-ingress
  namespace: hello-kubernetes
spec:
  rules:
  - host: hellok3s.${IP}.sslip.io
    http:
      paths:
        - path: "/"
          pathType: Prefix
          backend:
            service:
              name: hello-kubernetes
              port:
                name: http
EOF
```

And then:

```
curl http://hellok3s.${IP}.sslip.io
<!DOCTYPE html>
<html>
<head>
    <title>Hello Kubernetes!</title>
    <link rel="stylesheet" type="text/css" href="/css/main.css">
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Ubuntu:300" >
</head>
<body>

  <div class="main">
    <img src="/images/kubernetes.png"/>
    <div class="content">
      <div id="message">
  Hello world!
</div>
<div id="info">
  <table>
    <tr>
      <th>namespace:</th>
      <td>hello-kubernetes</td>
    </tr>
    <tr>
      <th>pod:</th>
      <td>hello-kubernetes-7c8575c848-fvqm2</td>
    </tr>
    <tr>
      <th>node:</th>
      <td>allinone (Linux 5.14.21-150400.24.46-default)</td>
```

```
    </tr>
  </table>
</div>
<div id="footer">
  paulbouwer/hello-kubernetes:1.10 (linux/amd64)
</div>
    </div>
  </div>

</body>
</html>
```

Verify that MetalLB works correctly:

```
% arping hellok3s.${IP}.sslip.io

ARPING 192.168.64.210
60 bytes from 92:12:36:00:d3:58 (192.168.64.210): index=0 time=1.169 msec
60 bytes from 92:12:36:00:d3:58 (192.168.64.210): index=1 time=2.992 msec
60 bytes from 92:12:36:00:d3:58 (192.168.64.210): index=2 time=2.884 msec
```

In the example above, the traffic flows as follows:

1. `hellok3s.${IP}.sslip.io` is resolved to the actual IP.
2. Then the traffic is handled by the `metallb-speaker` pod.
3. `metallb-speaker` redirects the traffic to the `traefik` controller.
4. Finally, Traefik forwards the request to the `hello-kubernetes` service.

62 MetalLB on K3s (using Layer 3 Mode)

MetalLB is a load-balancer implementation for bare-metal Kubernetes clusters, using standard routing protocols.

In this guide, we demonstrate how to deploy MetalLB in layer 3 (L3) BGP mode.

62.1 Why use MetalLB

MetalLB is a compelling choice for load balancing in bare-metal Kubernetes clusters for several reasons:

1. Native Integration with Kubernetes: MetalLB seamlessly integrates with Kubernetes, making it easy to deploy and manage using familiar Kubernetes tools and practices.
2. Bare-Metal Compatibility: Unlike cloud-based load balancers, MetalLB is designed specifically for on-premises deployments where traditional load balancers might not be available or feasible.
3. Supports Multiple Protocols: MetalLB supports both Layer 2 and Layer 3 BGP (Border Gateway Protocol) modes, providing flexibility for different network architectures and requirements.
4. High Availability: By distributing load-balancing responsibilities across multiple nodes, MetalLB ensures high availability and reliability for your services.
5. Scalability: MetalLB can handle large-scale deployments, scaling alongside your Kubernetes cluster to meet increasing demand.

In layer 2 mode, one node assumes the responsibility of advertising a service to the local network. From the network's perspective, it simply looks like that machine has multiple IP addresses assigned to its network interface.

The major advantage of the layer 2 mode is its universality: it works on any Ethernet network, with no special hardware required, not even fancy routers.

62.2 MetalLB on K3s (using L3)

In this quick start, L3 mode is used. This means that we need to have neighboring router(s) with BGP capabilities within the network range.

Note

From version 0.16.0, `frr-k8s` is the default backend for BGP. The FRR mode will be deprecated with the ultimate goal of removing it in the future.

62.3 Prerequisites

- A K3s cluster where MetalLB is deployed.
- Router(s) on the network that support the BGP protocol.
- A free IP address within the network range for the service. In this example `192.168.10.100`

Important

You must make sure this IP address is unassigned. In a DHCP environment this address must not be part of the DHCP pool to avoid dual assignments.

62.4 Configuration to Advertise Service IP Addresses

Out of the box BGP advertises a Service IP address to all the peers that are configured. These peers, which are usually routers, will receive a route for each Service IP address with a 32 bit network mask. In this examlple we will use an FRR based router and is on the same network as our cluster. We will then use MetalLB's BGP capability to advertise a service to that FRR based router.

62.5 Deployment

MetalLB is deployed by default for L2 VIP functionality, so there is no need to install it separately.

Verify that you have 4 pods in the `metallb-system` namespace and that they are all running without a problem:

```
k get pods -n metallb-system
NAME                                                    READY   STATUS    RESTARTS
 AGE
metallb-controller-7fbfd8977d-m2q9t                     1/1     Running   0
 46s
```

```
metallb-metallb-frr-k8s-9w7wl                                6/6     Running   0
 46s
metallb-metallb-frr-k8s-webhook-server-5d9d67ffd6-8jqnc   1/1     Running   1 (6s ago)
 46s
metallb-speaker-qx8bl                                        1/1     Running   0
 46s
```

62.6 Configuration

1. Create an `IPAddressPool`:

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: bgp-pool
  namespace: metallb-system
  labels:
    app: httpd
spec:
  addresses:
  - 192.168.10.100/32
  autoAssign: true
  avoidBuggyIPs: false
  serviceAllocation:
    namespaces:
    - metallb-system
    priority: 100
    serviceSelectors:
    - matchExpressions:
      - key: serviceType
        operator: In
        values:
        - httpd
EOF
```

2. Configure a `BGPPeer`.

Note

The FRR router has ASN 1000 while our `BGPPeer` will have 1001. We can also see that the FRR Router has an IP address that is 192.168.3.140.

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta2
kind: BGPPeer
metadata:
  namespace: metallb-system
  name: mypeertest
spec:
  peerAddress: 192.168.3.140
  peerASN: 1000
  myASN: 1001
  routerID: 4.4.4.4
EOF
```

3. Create the BGPAdvertisement (L3):

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: BGPAdvertisement
metadata:
  name: bgpadvertisement-test
  namespace: metallb-system
spec:
  ipAddressPools:
  - bgp-pool
EOF
```

62.7 Usage

1. Create an example application with a service. In this case, IP address from the `IPAddressPool` is `192.168.10.100` for that service.

```
cat <<- EOF | kubectl apply -f -
apiVersion: apps/v1
kind: Deployment
metadata:
  name: httpd-deployment
  namespace: metallb-system
  labels:
    app: httpd
spec:
  replicas: 3
  selector:
    matchLabels:
```

```
      pod-label: httpd
  template:
    metadata:
      labels:
        pod-label: httpd
    spec:
      containers:
      - name: httpdcontainer
        image: image: docker.io/library/httpd:2.4
        ports:
          - containerPort: 80
            protocol: TCP
      restartPolicy: Always

---
apiVersion: v1
kind: Service
metadata:
  name: http-service
  namespace: metallb-system
  labels:
    serviceType: httpd
spec:
  selector:
    pod-label: httpd
  type: LoadBalancer
  ports:
  - protocol: TCP
    port: 8080
    name: 8080-tcp
    targetPort: 80
EOF
```

2. To verify, log onto the FRR Router to can see the routes created from the BGP advertisement.

```
42178089cba5# show ip bgp all

For address family: IPv4 Unicast
BGP table version is 3, local router ID is 2.2.2.2, vrf id 0
Default local pref 100, local AS 1000
Status codes:  s suppressed, d damped, h history, * valid, > best, = multipath,
               i internal, r RIB-failure, S Stale, R Removed
Nexthop codes: @NNN nexthop's vrf id, < announce-nh-self
Origin codes:  i - IGP, e - EGP, ? - incomplete
RPKI validation codes: V valid, I invalid, N Not found

   Network          Next Hop            Metric LocPrf Weight Path
```

```
* i172.16.0.0/24    1.1.1.1                  0    100      0 i
*>                  0.0.0.0                  0         32768 i
* i172.17.0.0/24    3.3.3.3                  0    100      0 i
*>                  0.0.0.0                  0         32768 i
*= 192.168.10.100/32
                    192.168.3.162                          0 1001 i
*=                  192.168.3.163                          0 1001 i
*>                  192.168.3.161                          0 1001 i

Displayed  3 routes and 7 total paths
kubectl get svc -n hello-kubernetes
NAME               TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)        AGE
hello-kubernetes   LoadBalancer   10.43.127.75   192.168.122.11   80:31461/TCP   8s
```

3. If this router is the default gateway for your network, you can run the `curl` command from a box on that network to verify that they can reach the httpd sample app

```
# curl http://192.168.10.100:8080
<html><body><h1>It works!</h1></body></html>
#
```

63 MetalLB in front of the Kubernetes API server

This guide demonstrates using a MetalLB service to expose the RKE2/K3s API externally on an HA cluster with three control-plane nodes. To achieve this, a Kubernetes Service of type `LoadBalancer` will be manually created. Then an `EndpointSlices` object will be automatically created which keeps the IPs of all control plane nodes available in the cluster. For the EndpointSlices to be continuously synchronized with the events occurring in the cluster (adding/removing a node or a node goes offline), the Endpoint Copier Operator (*Chapter 18, Endpoint Copier Operator*) will be deployed. The operator monitors the events happening in the default `kubernetes` EndpointSlices and updates the managed one automatically to keep them in sync. Since the managed Service is of type `LoadBalancer`, MetalLB assigns it a static `ExternalIP`. This `ExternalIP` will be used to communicate with the API Server.

63.1 Prerequisites

- Three hosts to deploy RKE2/K3s on top.
  - Ensure the hosts have different host names.
  - For testing, these could be virtual machines
- At least 2 available IPs in the network (one for the Traefik ingress-controller exposed service and one for the managed service).
- Helm

63.2 Installing RKE2/K3s

Note

If you do not want to use a fresh cluster but want to use an existing one, skip this step and proceed to the next one.

First, a free IP in the network must be reserved that will be used later for `ExternalIP` of the managed Service.

SSH to the first host and install the wanted distribution in cluster mode.

For RKE2:

```
# As a root user, create the /etc/rancher/rke2/config.yaml config file with the following
 content:

mkdir -p /etc/rancher/rke2/
cat <<EOF > /etc/rancher/rke2/config.yaml
# An example of the config.yaml file for a server node:
write-kubeconfig-mode: "0644"
tls-san:
  - "${VIP_SERVICE_IP}"
  - "https://${VIP_SERVICE_IP}.sslip.io"
EOF

# Install RKE2
curl -sfL https://get.rke2.io | INSTALL_RKE2_EXEC="server" sh -

# Enable and start the RKE2 service with the configuration specified in the config.yaml
 file
systemctl enable rke2-server.service
systemctl start rke2-server.service

# Fetch the cluster token to be used later:
RKE2_TOKEN=$(tr -d '\n' < /var/lib/rancher/rke2/server/node-token)
```

For K3s:

```
# Export the free IP mentioned above
export VIP_SERVICE_IP=<ip>
export INSTALL_K3S_SKIP_START=false

curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="server --cluster-init \
 --disable=servicelb --write-kubeconfig-mode=644 --tls-san=${VIP_SERVICE_IP} \
 --tls-san=https://${VIP_SERVICE_IP}.sslip.io" K3S_TOKEN=foobar sh -
```

Note

Make sure that `--disable=servicelb` flag is provided in the `k3s server` command.

Important

From now on, the commands should be run on the local machine.

To access the API server from outside, the IP of the RKE2/K3s VM will be used.

```
# Replace <node-ip> with the actual IP of the machine
export NODE_IP=<node-ip>
export KUBE_DISTRIBUTION=<k3s/rke2>

scp ${NODE_IP}:/etc/rancher/${KUBE_DISTRIBUTION}/${KUBE_DISTRIBUTION}.yaml ~/.kube/config
 && sed \
 -i '' "s/127.0.0.1/${NODE_IP}/g" ~/.kube/config && chmod 600 ~/.kube/config
```

63.3 Configuring an existing cluster

Note

This step is valid only if you intend to use an existing RKE2/K3s cluster.

To use an existing cluster the `tls-san` flags should be modified. Additionally, the `servicelb` LB should be disabled for K3s.

To change the flags for RKE2 or K3s servers, you need to modify either the `/etc/systemd/system/rke2.service` or `/etc/systemd/system/k3s.service` file on all the VMs in the cluster, depending on the distribution.

The flags should be inserted in the `ExecStart`. For example:

For RKE2:

```
# Replace the <vip-service-ip> with the actual ip
ExecStart=/usr/local/bin/rke2 \
    server \
        '--write-kubeconfig-mode=644' \
        '--tls-san=<vip-service-ip>' \
        '--tls-san=https://<vip-service-ip>.sslip.io' \
```

For K3s:

```
# Replace the <vip-service-ip> with the actual ip
ExecStart=/usr/local/bin/k3s \
    server \
        '--cluster-init' \
        '--write-kubeconfig-mode=644' \
        '--disable=servicelb' \
        '--tls-san=<vip-service-ip>' \
        '--tls-san=https://<vip-service-ip>.sslip.io' \
```

Then the following commands should be executed to load the new configurations:

```
systemctl daemon-reload
systemctl restart ${KUBE_DISTRIBUTION}
```

63.4 Installing MetalLB

To deploy `MetalLB`, the MetalLB on K3s (*Chapter 61, MetalLB on K3s (using Layer 2 Mode)*) guide can be used.

NOTE: Ensure that the `VIP_SERVICE_IP` IP address does not overlap with the existing `IPAddressPools` in the cluster.

Create a separate `IpAddressPool` and `L2Advertisement` that will be used only for the managed Service.

NOTE: The IPAddressPool below will be assigned to a Service of type `LoadBalancer` in the `default` Namespace. If multiple `LoadBalancer` services exist there, additional ServiceSelectors (https://metallb.universe.tf/configuration/_advanced_ipaddresspool_configuration/#reduce-scope-of-address-allocation-to-specific-namespace-and-service) may be configured to match this VIP service explicitly.

```
# Export the VIP_SERVICE_IP on the local machine
# Replace with the actual IP
export VIP_SERVICE_IP=<ip>

cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: kubernetes-vip-ip-pool
  namespace: metallb-system
spec:
  addresses:
  - ${VIP_SERVICE_IP}/32
  serviceAllocation:
    priority: 100
    namespaces:
      - default
EOF
```

```
cat <<-EOF | kubectl apply -f -
apiVersion: metallb.io/v1beta1
```

```
kind: L2Advertisement
metadata:
  name: kubernetes-vip-l2-adv
  namespace: metallb-system
spec:
  ipAddressPools:
  - kubernetes-vip-ip-pool
EOF
```

63.5 Installing the Endpoint Copier Operator

```
helm install \
endpoint-copier-operator oci://registry.suse.com/edge/charts/endpoint-copier-operator \
--namespace endpoint-copier-operator \
--create-namespace
```

The command above will deploy the `endpoint-copier-operator` operator Deployment with two replicas. One will be the leader and the other will take over the leader role if needed.

Now, the `kubernetes-vip` Service should be deployed, which will be reconciled by the operator and an EndpointSlices with the configured ports and IP will be created.

For RKE2:

```
cat <<-EOF | kubectl apply -f -
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-vip
  namespace: default
spec:
  ports:
  - name: rke2-api
    port: 9345
    protocol: TCP
    targetPort: 9345
  - name: k8s-api
    port: 6443
    protocol: TCP
    targetPort: 6443
  type: LoadBalancer
EOF
```

For K3s:

```
cat <<-EOF | kubectl apply -f -
```

```
apiVersion: v1
kind: Service
metadata:
  name: kubernetes-vip
  namespace: default
spec:
  internalTrafficPolicy: Cluster
  ipFamilies:
  - IPv4
  ipFamilyPolicy: SingleStack
  ports:
  - name: https
    port: 6443
    protocol: TCP
    targetPort: 6443
  sessionAffinity: None
  type: LoadBalancer
EOF
```

Verify that the `kubernetes-vip` Service has the correct IP address:

```
kubectl get service kubernetes-vip -n default \
 -o=jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

Ensure that the `kubernetes-vip-*` and `kubernetes` EndpointSlices resources in the `default` namespace point to the same IPs.

```
kubectl get endpointslices | grep kubernetes
```

If everything is correct, the last thing left is to use the `VIP_SERVICE_IP` in our `Kubeconfig`.

```
sed -i '' "s/${NODE_IP}/${VIP_SERVICE_IP}/g" ~/.kube/config
```

From now on, all the `kubectl` will go through the `kubernetes-vip` service.

63.6 Adding control-plane nodes

To monitor the entire process, two more terminal tabs can be opened.

First terminal:

```
watch kubectl get nodes
```

Second terminal:

```
watch kubectl get endpointslices
```

Now execute the commands below on the second and third nodes.

For RKE2:

```
# As a root user, create the /etc/rancher/rke2/config.yaml config file with the following
 content:

mkdir -p /etc/rancher/rke2/
cat <<EOF > /etc/rancher/rke2/config.yaml
# An example of the config.yaml file for an additional server node:
server: https://${VIP_SERVICE_IP}:9345
write-kubeconfig-mode: "0644"
tls-san:
  - "${VIP_SERVICE_IP}"
  - "https://${VIP_SERVICE_IP}.sslip.io"
# The one from above
token: ${RKE2_TOKEN}
EOF

# Install RKE2
curl -sfL https://get.rke2.io | INSTALL_RKE2_TYPE="server" sh -

# Enable the RKE2 service with the configuration specified in the config.yaml file

systemctl enable --now rke2-server.service

# Fetch the cluster token to be used later:
RKE2_TOKEN=$(tr -d '\n' < /var/lib/rancher/rke2/server/node-token)
```

For K3s:

```
# Export the VIP_SERVICE_IP in the VM
# Replace with the actual IP
export VIP_SERVICE_IP=<ip>
export INSTALL_K3S_SKIP_START=false

curl -sfL https://get.k3s.io | INSTALL_K3S_EXEC="server \
 --server https://${VIP_SERVICE_IP}:6443 --disable=servicelb \
 --write-kubeconfig-mode=644" K3S_TOKEN=foobar sh -
```

64 Air-gapped deployments with Edge Image Builder

64.1 Intro

This guide will show how to deploy several of the SUSE Telco Cloud components completely air-gapped on SUSE Linux Micro 6.2 utilizing Edge Image Builder(EIB) (*Chapter 12, Edge Image Builder*). With this, you'll be able to boot into a customized, ready to boot (CRB) image created by EIB and have the specified components deployed on either a RKE2 or K3s cluster without an Internet connection or any manual steps. This configuration is highly desirable for customers that want to pre-bake all artifacts required for deployment into their OS image, so they are immediately available on boot.

We will cover an air-gapped installation of:

- *Chapter 6, Rancher*
- *Chapter 16, SUSE Security*
- *Chapter 15, SUSE Storage*
- *Chapter 19, Edge Virtualization*
- SUSE Private Registry

Warning

EIB will parse and pre-download all images referenced in the provided Helm charts and Kubernetes manifests. However, some of those may be attempting to pull container images and create Kubernetes resources based on those at runtime. In these cases we have to manually specify the necessary images in the definition file if we want to set up a completely air-gapped environment.

64.2 Prerequisites

If you're following this guide, it's assumed that you are already familiar with EIB (*Chapter 12, Edge Image Builder*). If not, please follow the quick start guide (*Chapter 5, Standalone clusters with Edge Image Builder*) to better understand the concepts shown in practice below.

64.3 Libvirt Network Configuration

Note

To demo the air-gapped deployment, this guide will be done using a simulated air-gapped `libvirt` network and the following configuration will be tailored to that. For your own deployments, you may have to modify the `host1.local.yaml` configuration that will be introduced in the next step.

If you would like to use the same `libvirt` network configuration, follow along. If not, skip to *Section 64.4, "Base Directory Configuration"*.

Let's create an isolated network configuration with an IP address range `192.168.100.2/24` for DHCP:

```
cat << EOF > isolatednetwork.xml
<network>
  <name>isolatednetwork</name>
  <bridge name='virbr1' stp='on' delay='0'/>
  <ip address='192.168.100.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.100.2' end='192.168.100.254'/>
    </dhcp>
  </ip>
</network>
EOF
```

Now, the only thing left is to create the network and start it:

```
virsh net-define isolatednetwork.xml
virsh net-start isolatednetwork
```

64.4 Base Directory Configuration

The base directory configuration is the same across all different components, so we will set it up here.

We will first create the necessary subdirectories:

```
export CONFIG_DIR=$HOME/config
mkdir -p $CONFIG_DIR/base-images
mkdir -p $CONFIG_DIR/network
mkdir -p $CONFIG_DIR/kubernetes/helm/values
```

Make sure to add whichever base image you plan to use into the `base-images` directory. This guide will focus on the Self Install ISO found here (https://www.suse.com/download/sle-micro/).

Let’s copy the downloaded image:

```
cp SL-Micro.x86_64-6.2-Base-SelfInstall-GM.install.iso $CONFIG_DIR/base-images/
slemicro.iso
```

Note

EIB is never going to modify the base image input.

Let’s create a file containing the desired network configuration:

```
cat << EOF > $CONFIG_DIR/network/host1.local.yaml
routes:
  config:
  - destination: 0.0.0.0/0
    metric: 100
    next-hop-address: 192.168.100.1
    next-hop-interface: eth0
    table-id: 254
  - destination: 192.168.100.0/24
    metric: 100
    next-hop-address: 192.168.122.1
    next-hop-interface: eth0
    table-id: 254
dns-resolver:
  config:
    server:
    - 192.168.100.1
    - 8.8.8.8
interfaces:
- name: eth0
  type: ethernet
  state: up
  mac-address: 34:8A:B1:4B:16:E7
  ipv4:
    address:
    - ip: 192.168.100.50
      prefix-length: 24
    dhcp: false
    enabled: true
  ipv6:
    enabled: false
EOF
```

This configuration ensures the following are present on the provisioned systems (using the specified MAC address):

- an Ethernet interface with a static IP address
- routing
- DNS
- hostname (`host1.local`)

The resulting file structure should now look like:

```
├── kubernetes/
│   └── helm/
│       └── values/
├── base-images/
│   └── slemicro.iso
└── network/
    └── host1.local.yaml
```

64.5 Base Definition File

Edge Image Builder is using *definition files* to modify the SUSE Linux Micro images. These files contain the majority of configurable options. Many of these options will be repeated across the different component sections, so we will list and explain those here.

Tip

Full list of customization options in the definition file can be found in the upstream documentation (https://github.com/suse-edge/edge-image-builder/blob/release-1.1/docs/building-images.md#image-definition-file)

We will take a look at the following fields which will be present in all definition files:

```
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
```

```
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.36.3+rke2r1
embeddedArtifactRegistry:
  images:
    - ...
```

The `image` section is required, and it specifies the input image, its architecture and type, as well as what the output image will be called.

The `operatingSystem` section is optional, and contains configuration to enable login on the provisioned systems with the `root/eib` username/password.

The `kubernetes` section is optional, and it defines the Kubernetes type and version. We are going to use the RKE2 distribution. Use `kubernetes.version: v1.36.3+k3s1` if K3s is desired instead. Unless explicitly configured via the `kubernetes.nodes` field, all clusters we bootstrap in this guide will be single-node ones.

The `embeddedArtifactRegistry` section will include all images which are only referenced and pulled at runtime for the specific component.

64.6 Rancher Installation

Note

The Rancher (*Chapter 6, Rancher*) deployment that will be demonstrated will be highly slimmed down for demonstration purposes. For your actual deployments, additional artifacts may be necessary depending on your configuration.

The Rancher 2.15.1 (https://github.com/rancher/rancher/releases/tag/v2.15.1) release assets contain a `rancher-images.txt` file which lists all the images required for an air-gapped installation.

There are over 600 container images in total which means that the resulting CRB image would be roughly 30GB. For our Rancher installation, we will strip down that list to the smallest working configuration. From there, you can add back any images you may need for your deployments.

We will create the definition file and include the stripped down image list:

```
apiVersion: 1.3
image:
```

```
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.36.3+rke2r1
  manifests:
    urls:
    - https://github.com/cert-manager/cert-manager/releases/download/v1.20.1/cert-
manager.crds.yaml
  helm:
    charts:
      - name: rancher
        version: 2.15.1
        repositoryName: rancher-prime
        valuesFile: rancher-values.yaml
        targetNamespace: cattle-system
        createNamespace: true
        installationNamespace: kube-system
      - name: cert-manager
        installationNamespace: kube-system
        createNamespace: true
        repositoryName: jetstack
        targetNamespace: cert-manager
        version: 1.20.1
    repositories:
      - name: jetstack
        url: https://charts.jetstack.io
      - name: rancher-prime
        url: https://charts.rancher.com/server-charts/prime
embeddedArtifactRegistry:
  images:
    - name: registry.rancher.com/rancher/backup-restore-operator:v10.0.2
    - name: registry.rancher.com/rancher/compliance-operator:v1.4.1
    - name: registry.rancher.com/rancher/fleet-agent:v0.16.1
    - name: registry.rancher.com/rancher/fleet:v0.16.1
    - name: registry.rancher.com/rancher/hardened-addon-resizer:1.8.23-build20260717
    - name: registry.rancher.com/rancher/hardened-calico:v3.32.1-build20260722
    - name: registry.rancher.com/rancher/hardened-cluster-autoscaler:v1.10.3-
build20260717
    - name: registry.rancher.com/rancher/hardened-cni-plugins:v1.9.1-build20260717
    - name: registry.rancher.com/rancher/hardened-coredns:v1.14.6-build20260722
```

```
    - name: registry.rancher.com/rancher/hardened-dns-node-cache:1.26.8-build20260722
    - name: registry.rancher.com/rancher/hardened-etcd:v3.6.14-k3s1-build20260723
    - name: registry.rancher.com/rancher/hardened-flannel:v0.28.8-build20260722
    - name: registry.rancher.com/rancher/hardened-k8s-metrics-server:v0.9.0-build20260722
    - name: registry.rancher.com/rancher/hardened-kubernetes:v1.36.3-rke2r1-build20260723
    - name: registry.rancher.com/rancher/hardened-multus-cni:v4.3.0-build20260722
    - name: registry.rancher.com/rancher/hardened-multus-dynamic-networks-
controller:v0.3.8-build20260722
    - name: registry.rancher.com/rancher/hardened-multus-thick:v4.3.0-build20260722
    - name: registry.rancher.com/rancher/hardened-traefik:v3.7.8-build20260717
    - name: registry.rancher.com/rancher/hardened-whereabouts:v0.9.4-build20260721
    - name: registry.rancher.com/rancher/k3s-upgrade:v1.36.3-k3s1
    - name: registry.rancher.com/rancher/klipper-helm:v0.13.3-build20260727
    - name: registry.rancher.com/rancher/klipper-lb:v0.4.17
    - name: registry.rancher.com/rancher/kubectl:v1.35.2
    - name: registry.rancher.com/rancher/kuberlr-kubectl:v7.1.1
    - name: registry.rancher.com/rancher/local-path-provisioner:v0.0.36
    - name: registry.rancher.com/rancher/machine:v0.15.0-rancher142
    - name: registry.rancher.com/rancher/nginx-ingress-controller:v1.14.5-hardened2
    - name: registry.rancher.com/rancher/prom-prometheus:v3.8.1
    - name: registry.rancher.com/rancher/prometheus-federator:v6.0.0
    - name: registry.rancher.com/rancher/pushprox:v0.1.10
    - name: registry.rancher.com/rancher/rancher-agent:v2.15.1
    - name: registry.rancher.com/rancher/rancher-csp-adapter:v10.0.0
    - name: registry.rancher.com/rancher/rancher-webhook:v0.11.1
    - name: registry.rancher.com/rancher/rancher:v2.15.1
    - name: registry.rancher.com/rancher/remotedialer-proxy:v0.7.2
    - name: registry.rancher.com/rancher/rke2-cloud-
provider:v1.36.2-0.20260610225606-10b320a3ba51-build20260709
    - name: registry.rancher.com/rancher/rke2-runtime:v1.36.3-rke2r1
    - name: registry.rancher.com/rancher/rke2-upgrade:v1.36.3-rke2r1
    - name: registry.rancher.com/rancher/scc-operator:v0.5.1
    - name: registry.rancher.com/rancher/security-scan:v0.9.1
    - name: registry.rancher.com/rancher/shell:v0.8.1
    - name: registry.rancher.com/rancher/supportability-review-app-frontend:v0.19.0
    - name: registry.rancher.com/rancher/supportability-review-internal:latest
    - name: registry.rancher.com/rancher/supportability-review-operator:v0.19.0
    - name: registry.rancher.com/rancher/supportability-review:latest
    - name: registry.rancher.com/rancher/system-agent-installer-k3s:v1.36.3-k3s1
    - name: registry.rancher.com/rancher/system-agent-installer-rke2:v1.36.3-rke2r1
    - name: registry.rancher.com/rancher/system-agent:v0.3.16-suc
    - name: registry.rancher.com/rancher/system-upgrade-controller:v0.20.1
    - name: registry.rancher.com/rancher/turtles:v0.27.1
    - name: registry.rancher.com/rancher/ui-plugin-catalog:4.15.0
    - name: registry.rancher.com/rancher/mirrored-ingress-nginx-kube-webhook-
certgen:v1.6.7
```

As compared to the full list of 600+ container images, this slimmed down version only contains ~60 which makes the new CRB image only about 7GB.

We also need to create a Helm values file for Rancher:

```
cat << EOF > $CONFIG_DIR/kubernetes/helm/values/rancher-values.yaml
hostname: 192.168.100.50.sslip.io
replicas: 1
bootstrapPassword: "adminadminadmin"
systemDefaultRegistry: registry.rancher.com
useBundledSystemChart: true
EOF
```

Warning

Setting the `systemDefaultRegistry` to `registry.rancher.com` allows Rancher to automatically look for images in the embedded artifact registry started within the CRB image at boot. Omitting this field may result in failure to find the container images on the node.

Let's build the image:

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
build --definition-file eib-iso-definition.yaml
```

The output should be similar to the following:

```
Downloading file: dl-manifest-1.yaml 100% |
└└└
```

```
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Populating Embedded Artifact Registry... 100% |
███████████████████████████████████████████████████████████████████████████| (56/56, 8
 it/min)
Embedded Artifact Registry ... [SUCCESS]
Keymap ....................... [SUCCESS]
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Downloading file: rke2-images-core.linux-amd64.tar.zst 100% |
███████████████████████████████████████████████████████████| (644/644 MB, 29 MB/s)
Downloading file: rke2-images-cilium.linux-amd64.tar.zst 100% |
█████████████████████████████████████████████████████████| (400/400 MB, 29 MB/s)
Downloading file: rke2.linux-amd64.tar.gz 100% |
███████████████████████████████████████████████████████████████████████████| (36/36 MB,
 30 MB/s)
Downloading file: sha256sum-amd64.txt 100% |
█████████████████████████████████████████████████████████████████████████████| (4.3/4.3
 kB, 29 MB/s)
Kubernetes ................... [SUCCESS]
Certificates ................. [SKIPPED]
Cleanup ...................... [SKIPPED]
Building ISO image...
Kernel Params ................ [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

Once a node using the built image is provisioned, we can verify the Rancher installation:

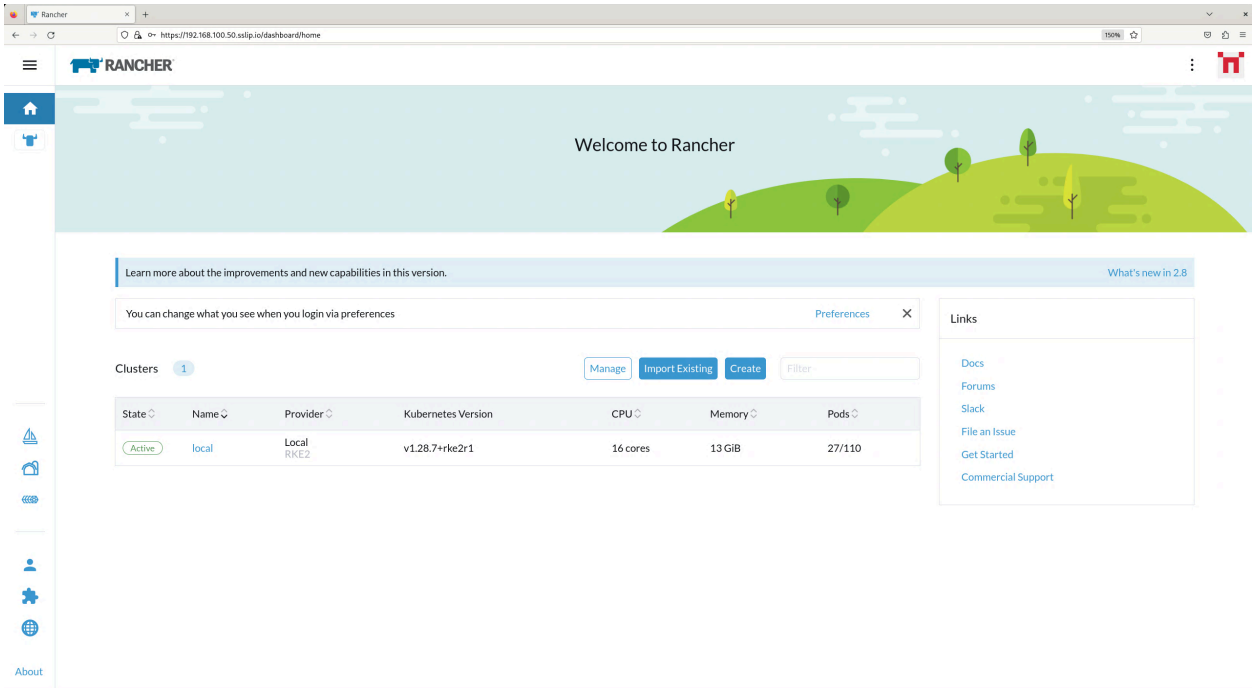
```
/var/lib/rancher/rke2/bin/kubectl get all -n cattle-system --kubeconfig /etc/rancher/
rke2/rke2.yaml
```

The output should be similar to the following, showing that everything has been successfully deployed:

```
NAME                                                READY   STATUS      RESTARTS   AGE
pod/helm-operation-6l6ld                            0/2     Completed   0          107s
pod/helm-operation-8tk2v                            0/2     Completed   0          2m2s
pod/helm-operation-blnrr                            0/2     Completed   0          2m49s
pod/helm-operation-hdcmt                            0/2     Completed   0          3m19s
pod/helm-operation-m74c7                            0/2     Completed   0          97s
pod/helm-operation-qzzr4                            0/2     Completed   0          2m30s
pod/helm-operation-s9jh5                            0/2     Completed   0          3m
pod/helm-operation-tq7ts                            0/2     Completed   0          2m41s
pod/rancher-99d599967-ftjkk                         1/1     Running     0          4m15s
pod/rancher-webhook-79798674c5-6w28t                1/1     Running     0          2m27s
pod/system-upgrade-controller-56696956b-trq5c       1/1     Running     0          104s

NAME                          TYPE         CLUSTER-IP     EXTERNAL-IP   PORT(S)         AGE
```

```
service/rancher           ClusterIP   10.43.255.80   <none>        80/TCP,443/TCP   4m15s
service/rancher-webhook   ClusterIP   10.43.7.238    <none>        443/TCP          2m27s

NAME                                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/rancher                   1/1     1            1           4m15s
deployment.apps/rancher-webhook           1/1     1            1           2m27s
deployment.apps/system-upgrade-controller 1/1     1            1           104s

NAME                                                 DESIRED   CURRENT   READY   AGE
replicaset.apps/rancher-99d599967                    1         1         1       4m15s
replicaset.apps/rancher-webhook-79798674c5           1         1         1       2m27s
replicaset.apps/system-upgrade-controller-56696956b  1         1         1       104s
```

And when we go to `https://192.168.100.50.sslip.io` and log in with the `adminadminadmin` password that we set earlier, we are greeted with the Rancher dashboard:



64.7 SUSE Security Installation

Unlike the Rancher installation, the SUSE Security installation does not require any special handling in EIB. EIB will automatically air-gap every image required by its underlying component NeuVector.

We will create the definition file:

```
apiVersion: 1.3
image:
```

```
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.36.3+rke2r1
  helm:
    charts:
      - name: neuvector-crd
        version: 110.0.1+up2.11.1
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector-values.yaml
      - name: neuvector
        version: 110.0.1+up2.11.1
        repositoryName: rancher-charts
        targetNamespace: neuvector
        createNamespace: true
        installationNamespace: kube-system
        valuesFile: neuvector-values.yaml
    repositories:
      - name: rancher-charts
        url: https://charts.rancher.io/
```

We will also create a Helm values file for NeuVector:

```
cat << EOF > $CONFIG_DIR/kubernetes/helm/values/neuvector-values.yaml
controller:
  replicas: 1
manager:
  enabled: false
cve:
  scanner:
    enabled: false
    replicas: 1
k3s:
  enabled: true
crdwebhook:
  enabled: false
EOF
```

Let’s build the image:

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
build --definition-file eib-iso-definition.yaml
```

The output should be similar to the following:

```
Pulling selected Helm charts... 100% |
└────────────
```

```
pod/neuvector-cert-upgrader-job-bxbnz           0/1     Completed   0          3m39s
pod/neuvector-controller-pod-7d854bfdc7-nhxjf   1/1     Running     0          3m44s
pod/neuvector-enforcer-pod-ct8jm                1/1     Running     0          3m44s

NAME                                        TYPE        CLUSTER-IP      EXTERNAL-IP
 PORT(S)                          AGE
service/neuvector-svc-admission-webhook   ClusterIP   10.43.234.241   <none>        443/
TCP                           3m44s
service/neuvector-svc-controller          ClusterIP   None            <none>
 18300/TCP,18301/TCP,18301/UDP   3m44s
service/neuvector-svc-crd-webhook         ClusterIP   10.43.50.190    <none>        443/
TCP                           3m44s

NAME                                     DESIRED   CURRENT   READY   UP-TO-DATE
 AVAILABLE   NODE SELECTOR   AGE
daemonset.apps/neuvector-enforcer-pod   1         1         1       1            1
    <none>          3m44s

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/neuvector-controller-pod   1/1     1            1           3m44s

NAME                                                   DESIRED   CURRENT   READY   AGE
replicaset.apps/neuvector-controller-pod-7d854bfdc7   1         1         1       3m44s

NAME                                          SCHEDULE    TIMEZONE   SUSPEND   ACTIVE
 LAST SCHEDULE   AGE
cronjob.batch/neuvector-cert-upgrader-pod   0 0 1 1 *   <none>     True      0
 <none>          3m44s
cronjob.batch/neuvector-updater-pod         0 0 * * *   <none>     False     0
 <none>          3m44s

NAME                                     STATUS     COMPLETIONS   DURATION   AGE
job.batch/neuvector-cert-upgrader-job   Complete   1/1           7s         3m39s
```

64.8 SUSE Storage Installation

The official documentation (https://longhorn.io/docs/1.12.1/deploy/install/airgap/) for Longhorn contains a `longhorn-images.txt` file which lists all the images required for an air-gapped installation. We will be including their mirrored counterparts from the Rancher container registry in our definition file. Let's create it:

```
apiVersion: 1.3
image:
  imageType: iso
```

```
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/zb4r8EmnrrNCF.P/
  packages:
    sccRegistrationCode: [reg-code]
    packageList:
      - open-iscsi
kubernetes:
  version: v1.36.3+rke2r1
  helm:
    charts:
      - name: suse-storage
        releaseName: longhorn
        repositoryName: rancher-application-collection
        targetNamespace: longhorn-system
        createNamespace: true
        version: 1.12.1
    repositories:
      - name: rancher-application-collection
        url: oci://dp.apps.rancher.io/charts
        authentication:
          username: $APPS.RANCHER.IO_USERNAME
          password: $APPS.RANCHER.IO_ACCESS_TOKEN
embeddedArtifactRegistry:
  registries:
    - uri: dp.apps.rancher.io
      authentication:
        username: $APPS.RANCHER.IO_USERNAME
        password: $APPS.RANCHER.IO_ACCESS_TOKEN
  images:
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-attacher:4.12.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-provisioner:5.3.0-17.8
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-resizer:2.2.1-10.8
    - name: dp.apps.rancher.io/containers/kubernetes-csi-external-snapshotter:8.6.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-livenessprobe:2.19.0-17.6
    - name: dp.apps.rancher.io/containers/kubernetes-csi-node-driver-
registrar:2.17.0-17.6
    - name: dp.apps.rancher.io/containers/longhorn-backing-image-manager:1.12.1-1.5
    - name: dp.apps.rancher.io/containers/longhorn-engine:1.12.1-1.6
    - name: dp.apps.rancher.io/containers/longhorn-instance-manager:1.12.1-1.7
    - name: dp.apps.rancher.io/containers/longhorn-manager:1.12.1-1.5
    - name: dp.apps.rancher.io/containers/longhorn-share-manager:1.12.1-1.5
```

```
    - name: dp.apps.rancher.io/containers/longhorn-ui:1.12.1-1.3
    - name: dp.apps.rancher.io/containers/rancher-support-bundle-kit:0.0.92-13.12
```

Note

You will notice that the definition file lists the `open-iscsi` package. This is necessary since Longhorn relies on a `iscsiadm` daemon running on the different nodes to provide persistent volumes to Kubernetes.

Let's build the image:

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
build --definition-file eib-iso-definition.yaml
```

The output should be similar to the following:

```
Setting up Podman API listener...
Pulling selected Helm charts... 100% |
 (2/2, 3 it/s)
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Resolving package dependencies...
Rpm .......................... [SUCCESS]
Os Files ..................... [SKIPPED]
Systemd ...................... [SKIPPED]
Fips ......................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Populating Embedded Artifact Registry... 100% |
 (15/15, 20956 it/s)
Embedded Artifact Registry ... [SUCCESS]
Keymap ....................... [SUCCESS]
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Downloading file: rke2-images-core.linux-amd64.tar.zst 100% (782/782 MB, 108 MB/s)
```

```
Downloading file: rke2-images-cilium.linux-amd64.tar.zst 100% (367/367 MB, 104 MB/s)
Downloading file: rke2.linux-amd64.tar.gz 100% (34/34 MB, 108 MB/s)
Downloading file: sha256sum-amd64.txt 100% (3.9/3.9 kB, 7.5 MB/s)
Kubernetes ................... [SUCCESS]
Certificates ................. [SKIPPED]
Cleanup ...................... [SKIPPED]
Building ISO image...
Kernel Params ............... [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

Once a node using the built image is provisioned, we can verify the Longhorn installation:

```
/var/lib/rancher/rke2/bin/kubectl get all -n longhorn-system --kubeconfig /etc/rancher/
rke2/rke2.yaml
```

The output should be similar to the following, showing that everything has been successfully deployed:

```
NAME                                                     READY   STATUS    RESTARTS   AGE
pod/csi-attacher-787fd9c6c8-sf42d                        1/1     Running   0
 2m28s
pod/csi-attacher-787fd9c6c8-tb82p                        1/1     Running   0
 2m28s
pod/csi-attacher-787fd9c6c8-zhc6s                        1/1     Running   0
 2m28s
pod/csi-provisioner-74486b95c6-b2v9s                     1/1     Running   0
 2m28s
pod/csi-provisioner-74486b95c6-hwllt                     1/1     Running   0
 2m28s
pod/csi-provisioner-74486b95c6-mlrpk                     1/1     Running   0
 2m28s
pod/csi-resizer-859d4557fd-t54zk                         1/1     Running   0
 2m28s
pod/csi-resizer-859d4557fd-vdt5d                         1/1     Running   0
 2m28s
pod/csi-resizer-859d4557fd-x9kh4                         1/1     Running   0
 2m28s
pod/csi-snapshotter-6f69c6c8cc-r62gr                     1/1     Running   0
 2m28s
pod/csi-snapshotter-6f69c6c8cc-vrwjn                     1/1     Running   0
 2m28s
pod/csi-snapshotter-6f69c6c8cc-z65nb                     1/1     Running   0
 2m28s
pod/engine-image-ei-4623b511-9vhkb                       1/1     Running   0
 3m13s
pod/instance-manager-6f95fd57d4a4cd0459e469d75a300552    1/1     Running   0
 2m43s
pod/longhorn-csi-plugin-gx98x                            3/3     Running   0
 2m28s
```

```
pod/longhorn-driver-deployer-55f9c88499-fbm6q           1/1     Running   0
 3m28s
pod/longhorn-manager-dpdp7                              2/2     Running   0
 3m28s
pod/longhorn-ui-59c85fcf94-gg5hq                        1/1     Running   0
 3m28s
pod/longhorn-ui-59c85fcf94-s49jc                        1/1     Running   0
 3m28s

NAME                                  TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)
 AGE
service/longhorn-admission-webhook    ClusterIP   10.43.77.89    <none>        9502/TCP
 3m28s
service/longhorn-backend              ClusterIP   10.43.56.17    <none>        9500/TCP
 3m28s
service/longhorn-conversion-webhook   ClusterIP   10.43.54.73    <none>        9501/TCP
 3m28s
service/longhorn-frontend             ClusterIP   10.43.22.82    <none>        80/TCP
 3m28s
service/longhorn-recovery-backend     ClusterIP   10.43.45.143   <none>        9503/TCP
 3m28s

NAME                                      DESIRED   CURRENT   READY   UP-TO-DATE
 AVAILABLE   NODE SELECTOR   AGE
daemonset.apps/engine-image-ei-4623b511   1         1         1       1            1
      <none>          3m13s
daemonset.apps/longhorn-csi-plugin        1         1         1       1            1
      <none>          2m28s
daemonset.apps/longhorn-manager           1         1         1       1            1
      <none>          3m28s

NAME                                        READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/csi-attacher                3/3     3            3           2m28s
deployment.apps/csi-provisioner             3/3     3            3           2m28s
deployment.apps/csi-resizer                 3/3     3            3           2m28s
deployment.apps/csi-snapshotter             3/3     3            3           2m28s
deployment.apps/longhorn-driver-deployer    1/1     1            1           3m28s
deployment.apps/longhorn-ui                 2/2     2            2           3m28s

NAME                                                DESIRED   CURRENT   READY   AGE
replicaset.apps/csi-attacher-787fd9c6c8             3         3         3       2m28s
replicaset.apps/csi-provisioner-74486b95c6          3         3         3       2m28s
replicaset.apps/csi-resizer-859d4557fd              3         3         3       2m28s
replicaset.apps/csi-snapshotter-6f69c6c8cc          3         3         3       2m28s
replicaset.apps/longhorn-driver-deployer-55f9c88499 1         1         1       3m28s
replicaset.apps/longhorn-ui-59c85fcf94              2         2         2       3m28s
```

64.9 KubeVirt and CDI Installation

The Helm charts for both KubeVirt and CDI are only installing their respective operators. It is up to the operators to deploy the rest of the systems which means we will have to include all necessary container images in our definition file. Let's create it:

```
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.36.3+rke2r1
  helm:
    charts:
      - name: kubevirt
        repositoryName: suse-edge
        version: 307.0.3+up0.8.0
        targetNamespace: kubevirt-system
        createNamespace: true
        installationNamespace: kube-system
      - name: cdi
        repositoryName: suse-edge
        version: 307.0.3+up0.8.0
        targetNamespace: cdi-system
        createNamespace: true
        installationNamespace: kube-system
    repositories:
      - name: suse-edge
        url: oci://registry.suse.com/edge/charts
embeddedArtifactRegistry:
  images:
    - name: registry.suse.com/suse/sles/16.0/cdi-apiserver:1.65.0
    - name: registry.suse.com/suse/sles/16.0/cdi-cloner:1.65.0
    - name: registry.suse.com/suse/sles/16.0/cdi-controller:1.65.0
    - name: registry.suse.com/suse/sles/16.0/cdi-importer:1.65.0
    - name: registry.suse.com/suse/sles/16.0/cdi-operator:1.65.0
    - name: registry.suse.com/suse/sles/16.0/cdi-uploadproxy:1.65.0
    - name: registry.suse.com/suse/sles/16.0/cdi-uploadserver:1.65.0
    - name: registry.suse.com/suse/sles/16.0/virt-api:1.8.3
```

```
    - name: registry.suse.com/suse/sles/16.0/virt-controller:1.8.3
    - name: registry.suse.com/suse/sles/16.0/virt-exportproxy:1.8.3
    - name: registry.suse.com/suse/sles/16.0/virt-exportserver:1.8.3
    - name: registry.suse.com/suse/sles/16.0/virt-handler:1.8.3
    - name: registry.suse.com/suse/sles/16.0/virt-launcher:1.8.3
    - name: registry.suse.com/suse/sles/16.0/virt-operator:1.8.3
```

Let's build the image:

```
podman run --rm -it --privileged -v $CONFIG_DIR:/eib \
registry.suse.com/edge/3.7/edge-image-builder:1.3.4 \
build --definition-file eib-iso-definition.yaml
```

The output should be similar to the following:

```
Pulling selected Helm charts... 100% |
 (2/2, 48 it/min)
Generating image customization components...
Identifier ................... [SUCCESS]
Custom Files ................. [SKIPPED]
Time ......................... [SKIPPED]
Network ...................... [SUCCESS]
Groups ....................... [SKIPPED]
Users ........................ [SUCCESS]
Proxy ........................ [SKIPPED]
Rpm .......................... [SKIPPED]
Os Files ..................... [SKIPPED]
Systemd ...................... [SKIPPED]
Fips ......................... [SKIPPED]
Elemental .................... [SKIPPED]
Suma ......................... [SKIPPED]
Populating Embedded Artifact Registry... 100% |
 (15/15, 4 it/min)
Embedded Artifact Registry ... [SUCCESS]
Keymap ....................... [SUCCESS]
Configuring Kubernetes component...
The Kubernetes CNI is not explicitly set, defaulting to 'cilium'.
Downloading file: rke2_installer.sh
Kubernetes ................... [SUCCESS]
Certificates ................. [SKIPPED]
Cleanup ...................... [SKIPPED]
Building ISO image...
Kernel Params ................ [SKIPPED]
Build complete, the image can be found at: eib-image.iso
```

Once a node using the built image is provisioned, we can verify the installation of both KubeVirt and CDI.

Verify KubeVirt:

```
/var/lib/rancher/rke2/bin/kubectl get all -n kubevirt-system --kubeconfig /etc/rancher/
rke2/rke2.yaml
```

The output should be similar to the following, showing that everything has been successfully deployed:

```
NAME                                  READY   STATUS    RESTARTS   AGE
pod/virt-api-59cb997648-mmt67         1/1     Running   0          2m34s
pod/virt-controller-69786b785-7cc96   1/1     Running   0          2m8s
pod/virt-controller-69786b785-wq2dz   1/1     Running   0          2m8s
pod/virt-handler-2l4dm                1/1     Running   0          2m8s
pod/virt-operator-7c444cff46-nps4l    1/1     Running   0          3m1s
pod/virt-operator-7c444cff46-r25xq    1/1     Running   0          3m1s

NAME                                  TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
 AGE
service/kubevirt-operator-webhook     ClusterIP   10.43.167.109   <none>        443/TCP
 2m36s
service/kubevirt-prometheus-metrics   ClusterIP   None            <none>        443/TCP
 2m36s
service/virt-api                      ClusterIP   10.43.18.202    <none>        443/TCP
 2m36s
service/virt-exportproxy              ClusterIP   10.43.142.188   <none>        443/TCP
 2m36s

NAME                          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE
 SELECTOR            AGE
daemonset.apps/virt-handler   1         1         1       1            1
 kubernetes.io/os=linux   2m8s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/virt-api           1/1     1            1           2m34s
deployment.apps/virt-controller    2/2     2            2           2m8s
deployment.apps/virt-operator      2/2     2            2           3m1s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/virt-api-59cb997648           1         1         1       2m34s
replicaset.apps/virt-controller-69786b785     2         2         2       2m8s
replicaset.apps/virt-operator-7c444cff46      2         2         2       3m1s

NAME                              AGE    PHASE
kubevirt.kubevirt.io/kubevirt     3m1s   Deployed
```

Verify CDI:

```
/var/lib/rancher/rke2/bin/kubectl get all -n cdi-system --kubeconfig /etc/rancher/rke2/
rke2.yaml
```

The output should be similar to the following, showing that everything has been successfully deployed:

```
NAME                                    READY   STATUS    RESTARTS   AGE
pod/cdi-apiserver-5598c9bf47-pqfxw      1/1     Running   0          3m44s
pod/cdi-deployment-7cbc5db7f8-g46z7     1/1     Running   0          3m44s
pod/cdi-operator-777c865745-2qcnj       1/1     Running   0          3m48s
pod/cdi-uploadproxy-646f4cd7f7-fzkv7    1/1     Running   0          3m44s

NAME                              TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
service/cdi-api                   ClusterIP   10.43.2.224    <none>        443/TCP
 3m44s
service/cdi-prometheus-metrics    ClusterIP   10.43.237.13   <none>        8080/TCP
 3m44s
service/cdi-uploadproxy           ClusterIP   10.43.114.91   <none>        443/TCP
 3m44s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/cdi-apiserver      1/1     1            1           3m44s
deployment.apps/cdi-deployment     1/1     1            1           3m44s
deployment.apps/cdi-operator       1/1     1            1           3m48s
deployment.apps/cdi-uploadproxy    1/1     1            1           3m44s

NAME                                          DESIRED   CURRENT   READY   AGE
replicaset.apps/cdi-apiserver-5598c9bf47      1         1         1       3m44s
replicaset.apps/cdi-deployment-7cbc5db7f8     1         1         1       3m44s
replicaset.apps/cdi-operator-777c865745       1         1         1       3m48s
replicaset.apps/cdi-uploadproxy-646f4cd7f7    1         1         1       3m44s
```

64.10 SUSE Private Registry Installation

To include the SUSE Private Registry in an air-gapped deployment, we must update the definition file to include the required helm chart as well as the embedded artifacts for the new images.

Let’s update the definition file:

```
apiVersion: 1.3
image:
  imageType: iso
  arch: x86_64
  baseImage: slemicro.iso
  outputImageName: eib-image.iso
operatingSystem:
  users:
    - username: root
```

```
      encryptedPassword: $6$jHugJNNd3HElGsUZ
$eodjVe4te5ps44SVcWshdfWizrP.xAyd71CVEXazBJ/.v799/WRCBXxfYmunlBO2yp1hm/zb4r8EmnrrNCF.P/
kubernetes:
  version: v1.36.3+rke2r1
  helm:
    charts:
      - name: metallb
        version: 307.0.3+up0.16.1
        targetNamespace: metallb-system
        createNamespace: true
        repositoryName: suse-edge-charts
        installationNamespace: kube-system
      - name: suse-storage
        releaseName: longhorn
        repositoryName: rancher-application-collection
        targetNamespace: longhorn-system
        createNamespace: true
        version: 1.12.1
      - name: private-registry-helm
        createNamespace: true
        installationNamespace: kube-system
        repositoryName: privateregistry
        targetNamespace: suse-private-registry
        valuesFile: privateregistry.yaml
        version: 1.1.1
    repositories:
      - name: privateregistry
        authentication:
          username: ${PRIVATE_REGISTRY_USERNAME}
          password: ${PRIVATE_REGISTRY_PASSWORD}
        plainHTTP: false
        skipTLSVerify: false
        url: oci://registry.suse.com/private-registry
      - name: rancher-application-collection
        url: oci://dp.apps.rancher.io/charts
        authentication:
          username: $APPS.RANCHER.IO_USERNAME
          password: $APPS.RANCHER.IO_ACCESS_TOKEN
embeddedArtifactRegistry:
  registries:
    - uri: registry.suse.com
      authentication:
        username: ${PRIVATE_REGISTRY_USERNAME}
        password: ${PRIVATE_REGISTRY_PASSWORD}
    - uri: dp.apps.rancher.io
        authentication:
          username: $APPS.RANCHER.IO_USERNAME
```

```
        password: $APPS.RANCHER.IO_ACCESS_TOKEN
images:
  - name: registry.suse.com/private-registry/harbor-core:1.1.1-1.19
  - name: registry.suse.com/private-registry/harbor-jobservice:1.1.1-1.19
  - name: registry.suse.com/private-registry/harbor-portal:1.1.1-1.20
  - name: registry.suse.com/private-registry/harbor-registry:1.1.1-1.19
  - name: registry.suse.com/private-registry/harbor-registryctl:1.1.1-1.19
  - name: registry.suse.com/private-registry/harbor-trivy-adapter:1.1.1-1.24
```

Note

You will need certain credentials, which can be retrieved by following the official SUSE Private Registry documentation (https://documentation.suse.com/cloudnative/suse-private-registry/html/private-registry/pr-deployment.html#pr-deployment-kube-secrets). You must also modify the `${PRIVATE_REGISTRY_USERNAME}` and `${PRIVATE_REGISTRY_PASSWORD}` variables. Make sure to list the images containing the component versions you need.

Now we need to add the required Kubernetes manifests to properly configure the SUSE Private Registry. You need to modify the `${MGMT_CLUSTER_REGISTRY_IP}` with a reserved static IP for the SUSE Private Registry in the following files:

1. `kubernetes/manifests/metallb-registry.yaml`

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: private-registry
  namespace: metallb-system
spec:
  ipAddressPools:
  - private-registry-pool
---
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: private-registry-pool
  namespace: metallb-system
spec:
  addresses:
  - ${MGMT_CLUSTER_REGISTRY_IP}/32
  serviceAllocation:
    namespaces:
```

```
    - suse-private-registry
```

2. `kubernetes/helm/values/privateregistry.yaml`

```
core:
  secretName: suse-registry-tls
expose:
  tls:
    certSource: secret
    enabled: true
    secret:
      secretName: suse-registry-tls
  type: loadBalancer
externalURL: https://${MGMT_CLUSTER_REGISTRY_IP}
persistence:
  persistentVolumeClaim:
    registry:
      size: 20Gi
```

Finally, the `kubernetes/manifests/suse-private-registry-creds.yaml` must be created with the following content:

```
apiVersion: v1
kind: Secret
metadata:
  name: suse-registry
  namespace: suse-private-registry
type: kubernetes.io/dockerconfigjson
data:
  .dockerconfigjson: ${DOCKER_CONFIG_JSON_BASE64}
---
apiVersion: v1
kind: Secret
metadata:
    name: suse-registry-tls
    namespace: suse-private-registry
type: kubernetes.io/tls
data:
    tls.crt: ${TLS_CRT_BASE64}
    tls.key: ${TLS_KEY_BASE64}
```

To correctly configure the docker config json (base64) for `${DOCKER_CONFIG_JSON_BASE64}`, run:

```
# ${DOCKER_CONFIG_JSON_BASE64} CONTENT
echo -n '{"auths": {"<MGMT_CLUSTER_REGISTRY_IP>": {"username": "<USERNAME>", "password":
 "<PASSWORD>", "auth": "<AUTH>"}}}' | base64
```

Where the IP is the same as the previously configured `${MGMT_CLUSTER_REGISTRY_IP}`, and the `username`, `password`, and `auth` can be retrieved from the SUSE Private Registry official documentation (https://documentation.suse.com/cloudnative/suse-private-registry/html/private-registry/pr-deployment.html#pr-deployment-kube-secrets).

To generate the base64-encoded TLS certificate and key (`tls.crt` and `tls.key`) for `${TLS_CRT_BASE64}` and `${TLS_KEY_BASE64}`, you can create your own by running:

```
# Generate a self-signed certificate and key
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -sha256 -days 365 -nodes

# Convert them to base64 for the suse-private-registry-creds.yaml file
cat cert.pem | base64 -w 0
cat key.pem | base64 -w 0
```

Verify SUSE Private Registry:

```
/var/lib/rancher/rke2/bin/kubectl get pods -n suse-private-registry --kubeconfig /etc/
rancher/rke2/rke2.yaml
```

The output should be similar to the following, showing that everything has been successfully deployed:

```
NAME                                                      READY   STATUS    RESTARTS
 AGE
pod/private-registry-harbor-core-588fd4876f-8tqnv         1/1     Running   0
 4m30s
pod/private-registry-harbor-database-0                    1/1     Running   0
 4m30s
pod/private-registry-harbor-jobservice-7658f97fbc-4vq6n   1/1     Running   0
 4m30s
pod/private-registry-harbor-portal-5455ccc4bc-jpmt5       1/1     Running   0
 4m30s
pod/private-registry-harbor-redis-0                       1/1     Running   0
 4m30s
pod/private-registry-harbor-registry-5648b9d89-wdswz      2/2     Running   0
 4m30s
pod/private-registry-harbor-trivy-0                       1/1     Running   0
 4m30s
```

64.11 Metal^3 Installation

When deploying Metal^3 in an air-gapped environment using EIB and Hauler, special configuration is required for the Ironic Python Agent (IPA) image. The IPA image is used by Metal^3 for bare-metal host inspection and provisioning.

Note

For more information about Metal^3 installation and configuration, including MetalLB setup, see *Chapter 4, BMC automated deployments with Metal^3*.

To include Metal^3 in an air-gapped deployment, you need to:

1. Add the Metal^3 Helm chart to your definition file
2. Add the IPA image to the embedded artifact registry
3. Create a Helm values file with `useHauler: true`

Add the following to the `kubernetes.helm.charts` section of your definition file:

```
  helm:
    charts:
      - name: metal3
        version: 307.0.31+up0.16.0
        targetNamespace: metal3-system
        createNamespace: true
        repositoryName: suse-edge-charts
        installationNamespace: kube-system
        valuesFile: metal3-values.yaml
```

Add the IPA image to the `embeddedArtifactRegistry.images` section:

```
embeddedArtifactRegistry:
  images:
    - name: registry.suse.com/edge/3.7/ironic-python-agent:3.0.10
```

Create a Helm values file for Metal^3:

```
cat << EOF > $CONFIG_DIR/kubernetes/helm/values/metal3-values.yaml
global:
  ironicIP: <STATIC_IRONIC_IP>
metal3-ironic:
  ipa:
    useHauler: true
EOF
```

> **Important**
>
> The `metal3-ironic.ipa.useHauler` value **must** be set to `true` (boolean value) when deploying in air-gapped environments using EIB/Hauler. This instructs Metal[3] to retrieve the IPA image from the embedded artifact registry instead of attempting to download it from the internet.

64.12 Troubleshooting

If you run into any issues while building the images or are looking to further test and debug the process, please refer to the upstream documentation (https://github.com/suse-edge/edge-image-builder/tree/release-1.1/docs).

65 Building Updated SUSE Linux Micro Images with Kiwi

This section explains how to generate updated SUSE Linux Micro images to be used with Edge Image Builder, with Cluster API (CAPI) + Metal[3], or to write the disk image directly to a block device. This process is useful in situations where the latest patches are required to be included in the initial system boot images (to minimise patch transfer post-installation), or for scenarios where CAPI is used, where it's preferred to reinstall the operating system with a new image rather than upgrading the hosts in place.

This process makes use of Kiwi (https://osinside.github.io/kiwi/) to run the image build. SUSE Telco Cloud ships with a containerized version that simplifies the overall process with a helper utility baked in, allowing to specify the target **profile** required. The profile defines the type of output image that is required, with the common ones listed below:

- "**Base**" - A SUSE Linux Micro disk image with a reduced package set (it includes podman).
- "**Base-SelfInstall**" - A SelfInstall image based on the "Base" above.
- "**Base-RT**" - Same as "Base" above but using a real-time (rt) kernel instead.
- "**Base-RT-SelfInstall**" - A SelfInstall image based on the "Base-RT" above
- "**Default**" - A SUSE Linux Micro disk image based on the "Base" above but with a few more tools, including the virtualization stack, Cockpit and salt-minion.
- "**Default-SelfInstall**" - A SelfInstall image based on the "Default" above

See SUSE Linux Micro 6.2 (https://documentation.suse.com/sle-micro/6.2/html/Micro-deployment-images/index.html#alp-images-installer-type) documentation for more details.

Note

This process works for both AMD64/Intel 64 and AArch64 architectures but it is necessary to use a build host with the same architecture of the images being built. In other words, to build an AArch64 image, it is required to use an AArch64 build host, and vice-versa for AMD64/Intel 64 - cross-builds are not supported at this time.

65.1 Prerequisites

Kiwi image builder requires the following:

- A SUSE Linux Micro 6.2 host ("build system") with the same architecture of the image being built.
- The build system needs to be already registered via `SUSEConnect` (the registration is used to pull the latest packages from the SUSE repositories)
- An internet connection that can be used to pull the required packages. If connected via proxy, the build host needs to be pre-configured.
- SELinux needs to be disabled on the build host (as SELinux labelling takes place in the container and it can conflict with the host policy)
- At least 10GB free disk space to accommodate the container image, the build root, and the resulting output image(s)

65.2 Getting Started

Due to certain limitations, it is currently required to disable SELinux. Connect to the SUSE Linux Micro 6.2 image build host and ensure SELinux is disabled:

```
# setenforce 0
```

Create an output directory to be shared with the Kiwi build container to save the resulting images:

```
# mkdir ~/output
```

Pull the latest Kiwi builder image from the SUSE Registry:

```
# podman pull registry.suse.com/edge/3.7/kiwi-builder:10.2.29.1
(...)
```

65.3 Building the Default Image

This is the default behavior of the Kiwi image container if no arguments are provided during the container image run. The following command runs `podman` with two directories mapped to the container:

- The `/etc/zypp/repos.d` SUSE Linux Micro package repository directory from the underlying host.
- The output `~/output` directory created above.

The Kiwi image container requires to run the `build-image` helper script as:

```
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/tmp/output
 \
    -it registry.suse.com/edge/3.7/kiwi-builder:10.2.29.1 build-image
(...)
```

Note

It's expected that if you're running this script for the first time that it will **fail** shortly after starting with "**ERROR: Early loop device test failed, please retry the container run.**", this is a symptom of loop devices being created on the underlying host system that are not immediately visible inside of the container image. Simply re-run the command again and it should proceed without issue.

After a few minutes the images can be found in the local output directory:

```
(...)
INFO: Image build successful, generated images are available in the 'output' directory.

# ls -1 output/
SLE-Micro.x86_64-6.2.changes
SLE-Micro.x86_64-6.2.packages
SLE-Micro.x86_64-6.2.raw
SLE-Micro.x86_64-6.2.verified
build
kiwi.result
kiwi.result.json
```

65.4 Building images with other profiles

In order to build different image profiles, the "**-p**" command option in the Kiwi container image helper script is used. For example, to build the "**Default-SelfInstall**" ISO image:

```
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/tmp/output
 \
    -it registry.suse.com/edge/3.7/kiwi-builder:10.2.29.1 build-image -p Default-
SelfInstall
(...)
```

Note

To avoid data loss, Kiwi will refuse to run if there are images in the `output` directory. It is required to remove the contents of the output directory before proceeding with `rm -f output/*`.

Alternatively, to build a SelfInstall ISO image with the RealTime kernel ("**kernel-rt**"):

```
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/tmp/output
 \
    -it registry.suse.com/edge/3.7/kiwi-builder:10.2.29.1 build-image -p Base-RT-
SelfInstall
(...)
```

65.5 Building images with large sector sizes

Some hardware requires an image with a large sector size, i.e. **4096 bytes** rather than the standard 512 bytes. The containerized Kiwi builder supports the ability to generate images with large block size by specifying the "**-b**" parameter. For example, to build a "**Default-SelfInstall**" image with a large sector size:

```
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/tmp/output
 \
    -it registry.suse.com/edge/3.7/kiwi-builder:10.2.29.1 build-image -p Default-
SelfInstall -b
(...)
```

65.6 Using a custom Kiwi image definition file

For advanced use-cases a custom Kiwi image definition file (`SL-Micro.kiwi`) can be used along with any necessary post-build scripts. This requires overriding the default definitions pre-packaged by the SUSE Telco Cloud team.

Create a new directory and map it into the container image where the helper script is looking (`/micro-sdk/defs`):

```
# mkdir ~/mydefs/
# cp /path/to/SL-Micro.kiwi ~/mydefs/
# cp /path/to/config.sh ~/mydefs/
# podman run --privileged -v /etc/zypp/repos.d:/micro-sdk/repos/ -v ~/output:/tmp/output
 -v ~/mydefs/:/micro-sdk/defs/ \
    -it registry.suse.com/edge/3.7/kiwi-builder:10.2.29.1 build-image
(...)
```

Warning

This is only required for advanced use-cases and may cause supportability issues. Please contact your SUSE representative for further advice and guidance.

To get the default Kiwi image definition files included in the container, the following commands can be used:

```
$ podman create --name kiwi-builder registry.suse.com/edge/3.7/kiwi-builder:10.2.29.1
$ podman cp kiwi-builder:/micro-sdk/defs/SL-Micro.kiwi .
$ podman cp kiwi-builder:/micro-sdk/defs/SL-Micro.kiwi.4096 .
$ podman rm kiwi-builder
$ ls ./SL-Micro.*
(...)
```

66 SUSE Storage V2 Data Engine

SUSE Storage 1.12.1 includes the Longhorn V2 Data Engine as a Generally Available feature. Starting with SUSE Telco Cloud 3.7, the V2 Data Engine is recommended for newly provisioned volumes when the cluster meets the V2 prerequisites.

Important

The SUSE Storage Helm chart does not convert existing V1 volumes to V2. The data engine is an immutable volume property. Existing V1 volumes continue to use V1 until their data is migrated to newly provisioned V2 volumes.

66.1 Prerequisites

Before enabling the V2 Data Engine, review the upstream V2 requirements (https://longhorn.io/docs/1.12.1/deploy/install/#v2-data-engine-requirements). In particular, every node that can host a V2 replica requires:

- A dedicated, unformatted block device. NVMe devices are recommended for production workloads.
- The `nvme-tcp`, `vfio_pci`, and `uio_pci_generic` kernel modules.
- Sufficient CPU and memory for the V2 instance manager. The default configuration uses 2 GiB of huge pages per instance manager.
- A supported CPU and kernel. AMD64 CPUs require SSE4.2, and kernel 6.7 or later is recommended.

Longhorn recommends using only one data engine after migration is complete. Running both engines creates separate instance managers and therefore consumes additional CPU and memory.

66.2 Configure V2 for new volumes

Prepare the required block devices before provisioning application volumes. Register each device as a `block` disk on its Longhorn node by using the SUSE Storage UI, or by updating the corresponding `nodes.longhorn.io` resource.

For a new cluster that has no V1 volumes, use the following Helm values:

```
defaultSettings:
```

```
  v1DataEngine: false
  v2DataEngine: true
persistence:
  dataEngine: v2
```

For an existing cluster with V1 volumes, keep both engines enabled during the migration:

```
defaultSettings:
  v1DataEngine: true
  v2DataEngine: true
persistence:
  dataEngine: v2
```

Apply the values with your normal Helm deployment or upgrade workflow. For example:

```
helm upgrade --install longhorn oci://dp.apps.rancher.io/charts/suse-storage \
  --version 1.12.1 \
  --namespace longhorn-system \
  --create-namespace \
  --reuse-values \
  --values suse-storage-v2-values.yaml
```

On an existing installation, confirm that the running Longhorn setting was enabled. If it remains disabled after the Helm upgrade, enable it explicitly:

```
kubectl -n longhorn-system patch settings.longhorn.io v2-data-engine \
  --type merge --patch '{"value":"true"}'
```

Caution

Do not disable the V1 Data Engine while any V1 volumes remain in the cluster. When upgrading an existing release, also confirm that `--reuse-values` does not preserve an older `persistence.dataEngine` value of `v1`.

Create a dedicated V2 StorageClass rather than changing an existing StorageClass in place:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: longhorn-v2
provisioner: driver.longhorn.io
allowVolumeExpansion: true
reclaimPolicy: Delete
volumeBindingMode: Immediate
```

```
parameters:
  dataEngine: "v2"
  numberOfReplicas: "3"
  staleReplicaTimeout: "2880"
  fsType: "ext4"
```

To make V2 the default for PVCs that do not specify `storageClassName`, move the default annotation to the new StorageClass:

```
kubectl annotate storageclass longhorn \
  storageclass.kubernetes.io/is-default-class="false" --overwrite

kubectl annotate storageclass longhorn-v2 \
  storageclass.kubernetes.io/is-default-class="true" --overwrite
```

This change affects only newly provisioned volumes. It does not change the data engine of existing volumes.

Verify the settings, StorageClass, disks, and volumes:

```
kubectl -n longhorn-system get settings.longhorn.io \
  v1-data-engine v2-data-engine

kubectl get storageclass longhorn-v2 \
  -o jsonpath='dataEngine={.parameters.dataEngine}{"\n"}'

kubectl -n longhorn-system get nodes.longhorn.io

kubectl -n longhorn-system get volumes.longhorn.io \
  -o custom-
columns='NAME:.metadata.name,ENGINE:.spec.dataEngine,STATE:.status.state,ROBUSTNESS:.status.robustness'
```

66.3 Migrate a V1 volume to V2

SUSE Storage 1.12.1 does not support converting an existing volume from V1 to V2 in place. Plan a maintenance window and migrate each workload to a new V2 volume.

1. Back up the application and its V1 volume.
2. Stop or scale down every workload that writes to the source PVC.
3. Create a destination PVC in the same namespace, using the `longhorn-v2` StorageClass and a capacity equal to or greater than the source PVC.

4. Copy the data using the method supported by the application.
5. Validate the copied data before changing the workload to use the destination PVC.
6. Update or recreate the workload so that it references the V2 PVC.
7. Keep the V1 volume until the application has been validated on V2, then remove it according to your retention policy.

For a filesystem volume that is not managed by a database or another application requiring a consistency-aware migration, a single maintenance pod can mount the V1 source read-only and the V2 destination read/write:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: copy-v1-to-v2
  namespace: <application-namespace>
spec:
  backoffLimit: 3
  template:
    spec:
      restartPolicy: Never
      containers:
        - name: copy
          image: registry.suse.com/bci/bci-micro:latest
          command:
            - /bin/bash
            - -c
            - |
              set -euo pipefail
              cp -a /source/. /destination/
              sync
          volumeMounts:
            - name: source
              mountPath: /source
              readOnly: true
            - name: destination
              mountPath: /destination
      volumes:
        - name: source
          persistentVolumeClaim:
            claimName: <v1-pvc-name>
        - name: destination
          persistentVolumeClaim:
            claimName: <v2-pvc-name>
```

Warning

The copy Job is only a generic filesystem example. Use application-native backup and restore or replication for databases, raw block volumes, and applications with their own consistency requirements. Verify ownership, permissions, extended attributes, and application data before deleting the source PVC.

After every workload has been migrated, confirm that no V1 volumes remain:

```
kubectl -n longhorn-system get volumes.longhorn.io \
  -o custom-columns='NAME:.metadata.name,ENGINE:.spec.dataEngine,STATE:.status.state'
```

You can then disable the V1 Data Engine in the Helm values and upgrade the release:

```
defaultSettings:
  v1DataEngine: false
  v2DataEngine: true
persistence:
  dataEngine: v2
```

If the running setting remains enabled after the Helm upgrade, disable it explicitly:

```
kubectl -n longhorn-system patch settings.longhorn.io v1-data-engine \
  --type merge --patch '{"value":"false"}'
```

Run the volume inventory command again and confirm that no V1 volumes were overlooked before removing any V1-specific node storage.

66.4 Usage limitations

Before moving a workload to V2, review the V1 and V2 feature comparison (https://longhorn.io/docs/1.12.1/v1-v2-volume-behavior-and-feature-parity/). The following limitations are especially relevant to SUSE Storage 1.12.1:

- V1 volumes cannot be converted to V2 in place.
- V2 backing images are not supported. Use CDI for KubeVirt image import workflows.
- Strict-local volumes, offline fast replica rebuilding, revision counters, and orphaned instance management are not supported by the V2 Data Engine.

- V2 volumes must be detached before upgrading between SUSE Storage 1.12 patch releases because V2 engine live upgrade is not supported.
- The sharded V2 Data Engine is experimental and does not have feature parity with replicated V2 volumes. Do not use it for production workloads that require unsupported capabilities such as backup and restore, disaster recovery, cloning, or live migration.
- On ARM64, V2 volumes backed by NVMe block devices have an upstream limitation. Review the Longhorn important notes (https://longhorn.io/docs/1.12.1/important-notes/) before deployment.

67 Using clusterclass to deploy downstream clusters

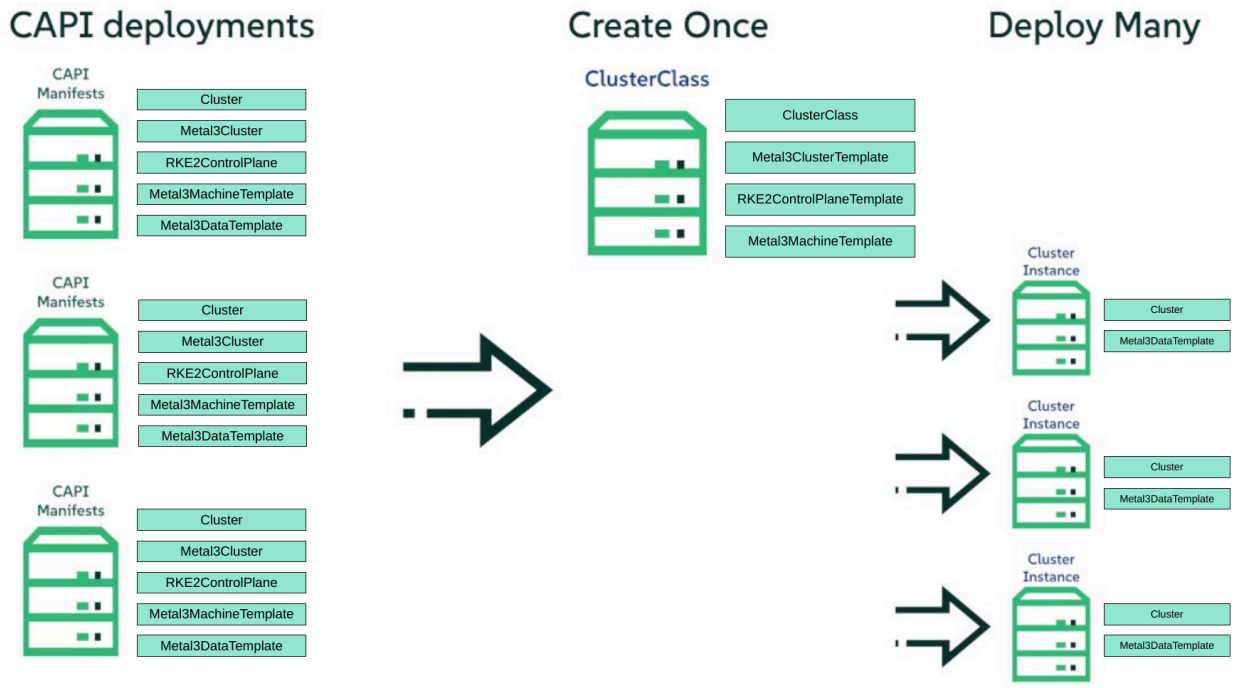
67.1 Introduction

Provisioning Kubernetes clusters is a complex task that demands deep expertise in configuring cluster components. As configurations grow more intricate, or as the demands of different providers introduce numerous provider-specific resource definitions, cluster creation can feel daunting. Thankfully, Kubernetes Cluster API (CAPI) offers a more elegant, declarative approach that is further enhanced by ClusterClass. This feature introduces a template-driven model, allowing you to define a reusable cluster class that encapsulates complexity and promotes consistency.

67.2 What is ClusterClass?

The CAPI project introduced the ClusterClass feature as a paradigm shift in Kubernetes cluster lifecycle management through the adoption of a template-based methodology for cluster instantiation. Instead of defining resources independently for every cluster, users define a ClusterClass, which serves as a comprehensive and reusable blueprint. This abstract representation encapsulates the desired state and configuration of a Kubernetes cluster, enabling the rapid and consistent creation of multiple clusters that adhere to the defined specifications. This abstraction reduces the configuration burden, resulting in more manageable deployment manifests. This means that the core components of a workload cluster are defined at the class level allowing users to use these templates as Kubernetes cluster flavors that can be reused one/many times for cluster provisioning. The implementation of ClusterClass yields several key advantages that address the inherent challenges of traditional CAPI management at scale:

- Substantial Reduction in Complexity and YAML Verbosity
- Optimized Maintenance and Update Processes
- Enhanced Consistency and Standardization Across Deployments
- Improved Scalability and Automation Capabilities
- Declarative Management and Robust Version Control



67.3 Example of current CAPI provisioning file

Deploying a Kubernetes cluster leveraging Cluster API (CAPI) with the CAPRKE2 (control plane & bootstrap) and CAPM3 (infrastructure) providers requires you to define several custom resources. These resources define the desired state of the cluster and its underlying infrastructure, enabling CAPI to orchestrate the provisioning and management lifecycle. The code snippet below illustrates the resource types that must be configured to instantiate a Kubernetes cluster with only control plane nodes.:

- **Cluster**: This resource encapsulates high-level configurations, including the network topology that will govern inter-node communication and service discovery. Furthermore, it establishes essential linkages to the control plane specification and the designated infrastructure provider resource, thereby informing CAPI about the desired cluster architecture and the underlying infrastructure upon which it will be provisioned.
- **Metal3Cluster**: This resource defines infrastructure-level attributes unique to Metal3, for example the external endpoint through which the Kubernetes API server will be accessible.

- **RKE2ControlPlane**: The RKE2ControlPlane resource defines the characteristics and behavior of the cluster's control plane nodes. Within this specification, parameters such as the desired number of control plane replicas (crucial for ensuring high availability and fault tolerance), the specific Kubernetes distribution version (aligned with the chosen RKE2 release), and the strategy for rolling out updates to the control plane components are configured. Additionally, this resource dictates the Container Network Interface (CNI) to be employed within the cluster and facilitates the injection of agent-specific configurations, often leveraging Ignition for seamless and automated provisioning of the RKE2 agents on the control plane nodes.

- **Metal3MachineTemplate**: This resource serves as a blueprint for creating the individual compute instances that form the cluster's control plane nodes.

- **Metal3DataTemplate**: Complementing the Metal3MachineTemplate, the Metal3DataTemplate resource enables additional metadata to be specified for the newly provisioned machine instances.

```
apiVersion: cluster.x-k8s.io/v1beta2
kind: Cluster
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
  labels:
    cluster-api.cattle.io/rancher-auto-import: "true"
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/18
    services:
      cidrBlocks:
        - 10.96.0.0/12
  controlPlaneRef:
    apiGroup: controlplane.cluster.x-k8s.io
    kind: RKE2ControlPlane
    name: emea-spa-cluster-3
  infrastructureRef:
    apiGroup: infrastructure.cluster.x-k8s.io
    kind: Metal3Cluster
    name: emea-spa-cluster-3
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3Cluster
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
```

```
spec:
  controlPlaneEndpoint:
    host: 192.168.122.203
    port: 6443
  cloudProviderEnabled: false
---
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlane
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  machineTemplate:
    spec:
      infrastructureRef:
        apiGroup: infrastructure.cluster.x-k8s.io
        kind: Metal3MachineTemplate
        name: emea-spa-cluster-3
  replicas: 1
  version: v1.36.3+rke2r1
  rolloutStrategy:
    type: "RollingUpdate"
    rollingUpdate:
      maxSurge: 1
  registrationMethod: "control-plane-endpoint"
  registrationAddress: 192.168.122.203
  serverConfig:
    cni: cilium
    cniMultusEnable: true
    tlsSan:
      - 192.168.122.203
      - https://192.168.122.203.sslip.io
  agentConfig:
    format: ignition
    additionalUserData:
      config: |
        variant: fcos
        version: 1.4.0
        storage:
          files:
            - path: /var/lib/rancher/rke2/server/manifests/endpoint-copier-
operator.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
```

```
                  metadata:
                    name: endpoint-copier-operator
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/endpoint-copier-
operator
                    targetNamespace: endpoint-copier-operator
                    version: 307.0.1+up0.3.0
                    createNamespace: true
            - path: /var/lib/rancher/rke2/server/manifests/metallb.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: helm.cattle.io/v1
                  kind: HelmChart
                  metadata:
                    name: metallb
                    namespace: kube-system
                  spec:
                    chart: oci://registry.suse.com/edge/charts/metallb
                    targetNamespace: metallb-system
                    version: 307.0.3+up0.16.1
                    createNamespace: true
            - path: /var/lib/rancher/rke2/server/manifests/metallb-cr.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: metallb.io/v1beta1
                  kind: IPAddressPool
                  metadata:
                    name: kubernetes-vip-ip-pool
                    namespace: metallb-system
                  spec:
                    addresses:
                      - 192.168.122.203/32
                    serviceAllocation:
                      priority: 100
                      namespaces:
                        - default
                      serviceSelectors:
                        - matchExpressions:
                          - {key: "serviceType", operator: In, values: [kubernetes-
vip]}
                  ---
                  apiVersion: metallb.io/v1beta1
                  kind: L2Advertisement
                  metadata:
```

```
                    name: ip-pool-l2-adv
                    namespace: metallb-system
                  spec:
                    ipAddressPools:
                      - kubernetes-vip-ip-pool
            - path: /var/lib/rancher/rke2/server/manifests/endpoint-svc.yaml
              overwrite: true
              contents:
                inline: |
                  apiVersion: v1
                  kind: Service
                  metadata:
                    name: kubernetes-vip
                    namespace: default
                    labels:
                      serviceType: kubernetes-vip
                  spec:
                    ports:
                    - name: rke2-api
                      port: 9345
                      protocol: TCP
                      targetPort: 9345
                    - name: k8s-api
                      port: 6443
                      protocol: TCP
                      targetPort: 6443
                    type: LoadBalancer
        systemd:
          units:
            - name: rke2-preinstall.service
              enabled: true
              contents: |
                [Unit]
                Description=rke2-preinstall
                Wants=network-online.target
                Before=rke2-install.service
                ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                [Service]
                Type=oneshot
                User=root
                ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                # Note we use _ as a delimiter for sed here because the providerid
 includes / characters
                ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -r .providerid /
mnt/openstack/latest/meta_data.json)_\" /etc/rancher/rke2/config.yaml"
                ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
```

```
                ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
                ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                ExecStartPost=/bin/sh -c "umount /mnt"
                [Install]
                WantedBy=multi-user.target
    kubelet:
      extraArgs:
      - provider-id=metal3://PROVIDERID
    nodeName: "localhost.localdomain"
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  nodeReuse: True
  template:
    spec:
      automatedCleaningMode: metadata
      dataTemplate:
        name: emea-spa-cluster-3
      hostSelector:
        matchLabels:
          cluster-role: control-plane
          deploy-region: emea-spa
          cluster-type: group-3
      image:
        checksum: http://fileserver.local:8080/eibimage-downstream-
cluster.raw.sha256
        checksumType: sha256
        diskFormat: raw
        url: http://fileserver.local:8080/eibimage-downstream-cluster.raw
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3DataTemplate
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
  clusterName: emea-spa-cluster-3
  metaData:
    objectNames:
    - key: name
      object: machine
```

```
    - key: local-hostname
      object: machine
    - key: local_hostname
      object: machine
```

Note

Adding the label `cluster-api.cattle.io/rancher-auto-import: "true"` to a `Cluster` API object in the `cluster.x-k8s.io` API group will import the cluster into Rancher (by creating a corresponding Rancher `Cluster` API object in the `management.cattle.io` API group). See the Cluster API documentation (https://documentation.suse.com/cloudnative/cluster-api/latest/en/user/rancher-cluster-registration.html#_how_to_register_a_capi_cluster_in_rancher_using_turtles_auto_import) for more information.

67.4 Transforming the CAPI provisioning file to ClusterClass

67.4.1 ClusterClass definition

The following code defines a ClusterClass resource, a declarative template for consistently deploying specific types of Kubernetes clusters. This specification includes common infrastructure and control plane configurations, enabling efficient provisioning and uniform lifecycle management across a cluster fleet. There are some variables defined in the following clusterclass example, that will be replaced during the cluster instantiation process using the real values requested for each cluster instance. The following variables are used in the example:

- `metallbHelmChartVersion`: This optional variable allows to specify the version of the `MetalLB` Helm chart to be deployed in the cluster; when not provided, the `MetalLB` Helm chart will not be deployed.
- `endpointCopierOperatorHelmChartVersion`: This optional variable allows to specify the version of the `Endpoint Copier Operator` Helm chart to be deployed in the cluster (when a version for the `MetalLB` Helm chart has been also provided); when not provided, the `Endpoint Copier Operator` Helm chart will not be deployed.

- `controlPlaneMachineTemplateImageURL`: This required variable allows to specify the URL of the image to be used to provision the control plane machines.
- `controlPlaneMachineTemplateHostSelectorDeployRegion`: This required variable allows to specify the `deploy-region` host selector value for the control plane machines.
- `controlPlaneMachineTemplateHostSelectorClusterType`: This required variable allows to specify the `cluster-type` host selector value for the control plane machines.
- `controlPlaneMachineTemplateDataTemplateName`: This required variable allows to specify the name of the `Metal3DataTemplate` API object to use for the control plane machines.
- `controlPlaneEndpointHost`: This required variable allows to specify the host name or IP address of the control plane endpoint.
- `tlsSan`: This required variable allows to specify the list of TLS Subject Alternative Names for the control plane endpoint.

The clusterclass definition file consists of the following four resources:

- **ClusterClass**: This resource encapsulates the entire cluster class definition, including the control plane and infrastructure templates. Moreover, it include the list of variables that will be replaced during the instantiation process (and defines the patches using them).
- **RKE2ControlPlaneTemplate**: This resource defines the control plane template, specifying the desired configuration for the control plane nodes. Also, some parameters will be replaced with the right values during the instantiation process.
- **Metal3ClusterTemplate**: This resource defines the infrastructure template, specifying the desired configuration for the underlying infrastructure. It includes parameters such as the control plane endpoint and the cloudProviderEnabled (true|false) flag. Also, some parameters will be replaced with the right values during the instantiation process.
- **Metal3MachineTemplate**: This resource defines the machine template blueprint for the control plane nodes, providing the placeholders for those parameters to be replaced with the right values during the instantiation process.

```
apiVersion: controlplane.cluster.x-k8s.io/v1beta2
kind: RKE2ControlPlaneTemplate
metadata:
  name: example-controlplane
  namespace: emea-spa
spec:
  template:
```

```
    spec:
      rolloutStrategy:
        type: "RollingUpdate"
        rollingUpdate:
          maxSurge: 1
      registrationMethod: "control-plane-endpoint"
      registrationAddress: TO-BE-PATCHED-AUTOMATICALLY  # This will be replaced by
 the involved patch when applying the cluster instance
      serverConfig:
        cni: cilium
        cniMultusEnable: true
        tlsSan:
        - TO-BE-PATCHED-AUTOMATICALLY    # This will be replaced by the involved
 patch when applying the cluster instance
      agentConfig:
        format: ignition
        additionalUserData:
          config: |          # This will be replaced by the involved patch when
 applying the cluster instance
            TO-BE-PATCHED-AUTOMATICALLY
        kubelet:
          extraArgs:
          - provider-id=metal3://PROVIDERID
        nodeName: "localhost.localdomain"
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3ClusterTemplate
metadata:
  name: example-cluster-template
  namespace: emea-spa
spec:
  template:
    spec:
      controlPlaneEndpoint:
        host: TO-BE-PATCHED-AUTOMATICALLY  # This will be replaced by the involved
 patch when applying the cluster instance
        port: 6443
      cloudProviderEnabled: false
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: example-controlplane-machineinfrastructure
  namespace: emea-spa
spec:
  nodeReuse: true
  template:
```

```
    spec:
      automatedCleaningMode: metadata
      dataTemplate:
        name: to-be-patched-automatically  # This will be replaced by the involved
 patch when applying the cluster instance
      hostSelector:
        matchLabels:  # The labels used to identify the BMHs to select.
          cluster-role: control-plane
          deploy-region: TO-BE-PATCHED-AUTOMATICALLY # This will be replaced by the
 involved patch when applying the cluster instance
          cluster-type: TO-BE-PATCHED-AUTOMATICALLY  # This will be replaced by the
 involved patch when applying the cluster instance
      image:
        checksum: http://TO-BE-PATCHED-AUTOMATICALLY # This will be replaced by the
 involved patch when applying the cluster instance
        checksumType: sha256
        diskFormat: raw
        url: http://TO-BE-PATCHED-AUTOMATICALLY      # This will be replaced by the
 involved patch when applying the cluster instance
---
apiVersion: cluster.x-k8s.io/v1beta2
kind: ClusterClass
metadata:
  name: example-clusterclass
  namespace: emea-spa
spec:
  infrastructure:
    templateRef:
      kind: Metal3ClusterTemplate
      apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
      name: example-cluster-template
  controlPlane:
    templateRef:
      kind: RKE2ControlPlaneTemplate
      apiVersion: controlplane.cluster.x-k8s.io/v1beta2
      name: example-controlplane
    machineInfrastructure:
      templateRef:
        kind: Metal3MachineTemplate
        apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
        name: example-controlplane-machineinfrastructure
  variables:
  - name: metallbHelmChartVersion
    required: false
    schema:
      openAPIV3Schema:
        type: string
```

```
  - name: endpointCopierOperatorHelmChartVersion
    required: false
    schema:
      openAPIV3Schema:
        type: string
  - name: controlPlaneMachineTemplateImageURL
    required: true
    schema:
      openAPIV3Schema:
        type: string
  - name: controlPlaneMachineTemplateHostSelectorDeployRegion
    required: true
    schema:
      openAPIV3Schema:
        type: string
  - name: controlPlaneMachineTemplateHostSelectorClusterType
    required: true
    schema:
      openAPIV3Schema:
        type: string
  - name: controlPlaneMachineTemplateDataTemplateName
    required: true
    schema:
      openAPIV3Schema:
        type: string
  - name: controlPlaneEndpointHost
    required: true
    schema:
      openAPIV3Schema:
        type: string
  - name: tlsSan
    required: true
    schema:
      openAPIV3Schema:
        type: array
        items:
          type: string
  patches:
  - name: setControlPlaneMachineTemplateImageURL
    definitions:
    - selector:
        apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
        kind: Metal3MachineTemplate
        matchResources:
          controlPlane: true        # Added to select ControlPlane
      jsonPatches:
      - op: replace
```

```
        path: "/spec/template/spec/image/url"
        valueFrom:
          variable: controlPlaneMachineTemplateImageURL
      - op: replace
        path: "/spec/template/spec/image/checksum"
        valueFrom:
          template: "{{ .controlPlaneMachineTemplateImageURL }}.sha256"
  - name: setControlPlaneMachineTemplateHostSelectorDeployRegion
    definitions:
    - selector:
        apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
        kind: Metal3MachineTemplate
        matchResources:
          controlPlane: true
      jsonPatches:
      - op: replace
        path: "/spec/template/spec/hostSelector/matchLabels/deploy-region"
        valueFrom:
          variable: controlPlaneMachineTemplateHostSelectorDeployRegion
  - name: setControlPlaneMachineTemplateHostSelectorClusterType
    definitions:
    - selector:
        apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
        kind: Metal3MachineTemplate
        matchResources:
          controlPlane: true        # Added to select ControlPlane
      jsonPatches:
      - op: replace
        path: "/spec/template/spec/hostSelector/matchLabels/cluster-type"
        valueFrom:
          variable: controlPlaneMachineTemplateHostSelectorClusterType
  - name: setControlPlaneMachineTemplateDataTemplateName
    definitions:
    - selector:
        apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
        kind: Metal3MachineTemplate
        matchResources:
          controlPlane: true        # Added to select ControlPlane
      jsonPatches:
      - op: replace
        path: "/spec/template/spec/dataTemplate/name"
        valueFrom:
          variable: controlPlaneMachineTemplateDataTemplateName
  - name: setControlPlaneEndpoint
    definitions:
    - selector:
        apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
```

```
        kind: Metal3ClusterTemplate
        matchResources:
          infrastructureCluster: true  # Added to select InfraCluster
      jsonPatches:
      - op: replace
        path: "/spec/template/spec/controlPlaneEndpoint/host"
        valueFrom:
          variable: controlPlaneEndpointHost
  - name: setRegistrationAddress
    definitions:
    - selector:
        apiVersion: controlplane.cluster.x-k8s.io/v1beta2
        kind: RKE2ControlPlaneTemplate
        matchResources:
          controlPlane: true        # Added to select ControlPlane
      jsonPatches:
      - op: replace
        path: "/spec/template/spec/registrationAddress"
        valueFrom:
          variable: controlPlaneEndpointHost
  - name: setTlsSan
    definitions:
    - selector:
        apiVersion: controlplane.cluster.x-k8s.io/v1beta2
        kind: RKE2ControlPlaneTemplate
        matchResources:
          controlPlane: true        # Added to select ControlPlane
      jsonPatches:
      - op: replace
        path: "/spec/template/spec/serverConfig/tlsSan"
        valueFrom:
          variable: tlsSan
  - name: updateAdditionalUserData
    definitions:
    - selector:
        apiVersion: controlplane.cluster.x-k8s.io/v1beta2
        kind: RKE2ControlPlaneTemplate
        matchResources:
          controlPlane: true
      jsonPatches:
      - op: replace
        path: "/spec/template/spec/agentConfig/additionalUserData"
        valueFrom:
          template: |
            config: |
              variant: fcos
              version: 1.4.0
```

```
              {{ if .metallbHelmChartVersion }}
              storage:
                files:
                - path: /var/lib/rancher/rke2/server/manifests/metallb.yaml
                  overwrite: true
                  contents:
                    inline: |
                      apiVersion: helm.cattle.io/v1
                      kind: HelmChart
                      metadata:
                        name: metallb
                        namespace: kube-system
                      spec:
                        chart: oci://registry.suse.com/edge/charts/metallb
                        targetNamespace: metallb-system
                        version: {{ .metallbHelmChartVersion }}
                        createNamespace: true
                {{ if .endpointCopierOperatorHelmChartVersion }}
                - path: /var/lib/rancher/rke2/server/manifests/endpoint-copier-
operator.yaml
                  overwrite: true
                  contents:
                    inline: |
                      apiVersion: helm.cattle.io/v1
                      kind: HelmChart
                      metadata:
                        name: endpoint-copier-operator
                        namespace: kube-system
                      spec:
                        chart: oci://registry.suse.com/edge/charts/endpoint-copier-
operator
                        targetNamespace: endpoint-copier-operator
                        version: {{ .endpointCopierOperatorHelmChartVersion }}
                        createNamespace: true
                - path: /var/lib/rancher/rke2/server/manifests/metallb-cr.yaml
                  overwrite: true
                  contents:
                    inline: |
                      apiVersion: metallb.io/v1beta1
                      kind: IPAddressPool
                      metadata:
                        name: kubernetes-vip-ip-pool
                        namespace: metallb-system
                      spec:
                        addresses:
                          - {{ .controlPlaneEndpointHost }}/32
                        serviceAllocation:
```

```
                        priority: 100
                        namespaces:
                          - default
                        serviceSelectors:
                          - matchExpressions:
                            - {key: "serviceType", operator: In, values:
[kubernetes-vip]}
                    ---
                    apiVersion: metallb.io/v1beta1
                    kind: L2Advertisement
                    metadata:
                      name: ip-pool-l2-adv
                      namespace: metallb-system
                    spec:
                      ipAddressPools:
                        - kubernetes-vip-ip-pool
              - path: /var/lib/rancher/rke2/server/manifests/endpoint-svc.yaml
                overwrite: true
                contents:
                  inline: |
                    apiVersion: v1
                    kind: Service
                    metadata:
                      name: kubernetes-vip
                      namespace: default
                      labels:
                        serviceType: kubernetes-vip
                    spec:
                      ports:
                      - name: rke2-api
                        port: 9345
                        protocol: TCP
                        targetPort: 9345
                      - name: k8s-api
                        port: 6443
                        protocol: TCP
                        targetPort: 6443
                      type: LoadBalancer
              {{ end }}
            {{ end }}
            systemd:
              units:
              - name: rke2-preinstall.service
                enabled: true
                contents: |
                  [Unit]
                  Description=rke2-preinstall
```

```
                    Wants=network-online.target
                    Before=rke2-install.service
                    ConditionPathExists=!/run/cluster-api/bootstrap-success.complete
                    [Service]
                    Type=oneshot
                    User=root
                    ExecStartPre=/bin/sh -c "mount -L config-2 /mnt"
                    # Note we use _ as a delimiter for sed here because the
 providerid includes / characters
                    ExecStart=/bin/sh -c "sed -i \"s_PROVIDERID_$(jq -
r .providerid /mnt/openstack/latest/meta_data.json)_\" /etc/rancher/rke2/
config.yaml"
                    ExecStart=/bin/sh -c "echo \"node-name: $(jq -r .name /mnt/
openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                    ExecStart=/bin/sh -c "echo \"node-label:\" >> /etc/rancher/rke2/
config.yaml"
                    ExecStart=/bin/sh -c "echo \"  - metal3.io/uuid=$(jq -r .uuid /
mnt/openstack/latest/meta_data.json)\" >> /etc/rancher/rke2/config.yaml"
                    ExecStartPost=/bin/sh -c "umount /mnt"
                    [Install]
                    WantedBy=multi-user.target
```

67.4.2 Cluster instance definition

Within the context of ClusterClass, a cluster instance refers to a specific, running instantiation of a cluster that has been created based on a defined ClusterClass. It represents a concrete deployment with its unique configurations, resources, and operational state, directly derived from the blueprint specified in the ClusterClass. This includes the specific set of machines, networking configurations, and associated Kubernetes components that are actively running. Understanding the cluster instance is crucial for managing the lifecycle, performing upgrades, executing scaling operations, and conducting monitoring of a particular deployed cluster that was provisioned using the ClusterClass framework.

To define a cluster instance we need to define the following two resources:

- **Cluster**: This resource represents the actual cluster instance, including a reference to its `topology` (i.e., `ClusterClass` object), the required Kubernetes version, the number of control plane nodes to instantiate and the values provided to each of the variables defined in the referenced `ClusterClass`, which will be used during the instantiation process.

- **Metal3DataTemplate**: This resource provides additional metadata for the machines in the cluster (control plane machines only in this example), complementing the `Metal3MachineTemplate` object automatically created per each instantiated cluster. Note that it is not possible to define a single

`Metal3DataTemplate` resource valid for all clusters instantiated from a common `ClusterClass`, since the `clusterName` field in the `Metal3DataTemplate` specification must reference the specific cluster instance to which it "belongs".

```
apiVersion: cluster.x-k8s.io/v1beta2
kind: Cluster
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
  labels:
    cluster-api.cattle.io/rancher-auto-import: "true"
spec:
  topology:
    classRef:
      name: example-clusterclass
      namespace: emea-spa
    version: v1.36.3+rke2r1
    controlPlane:
      replicas: 1
    variables:                       # Values to be replaced in the clusterclass
 template variables to create this specific cluster
    - name: endpointCopierOperatorHelmChartVersion
      value: 307.0.1+up0.3.0
    - name: metallbHelmChartVersion
      value: 307.0.3+up0.16.1
    - name: controlPlaneMachineTemplateImageURL
      value: http://fileserver.local:8080/eibimage-downstream-cluster.raw.sha256
    - name: controlPlaneMachineTemplateHostSelectorDeployRegion
      value: emea-spa
    - name: controlPlaneMachineTemplateHostSelectorClusterType
      value: group-3
    - name: controlPlaneMachineTemplateDataTemplateName
      value: emea-spa-cluster-3
    - name: controlPlaneEndpointHost
      value: 192.168.122.203
    - name: tlsSan
      value:
      - 192.168.122.203
      - https://192.168.122.203.sslip.io
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3DataTemplate
metadata:
  name: emea-spa-cluster-3
  namespace: emea-spa
spec:
```

```
  clusterName: emea-spa-cluster-3
  metaData:
    objectNames:
    - key: name
      object: machine
    - key: local-hostname
      object: machine
    - key: local_hostname
      object: machine
```

Note

Adding the label `cluster-api.cattle.io/rancher-auto-import: "true"` to a `Cluster` API object in the `cluster.x-k8s.io` API group will import the cluster into Rancher (by creating a corresponding Rancher `Cluster` API object in the `management.cattle.io` API group). See the Cluster API documentation (https://documentation.suse.com/cloudnative/cluster-api/latest/en/user/rancher-cluster-registration.html#_how_to_register_a_capi_cluster_in_rancher_using_turtles_auto_import) for more information.

This approach streamlines the process, allowing you to deploy a cluster using only two resources once the clusterclass blueprints are defined.

68 Longhorn Volume Encryption

Longhorn is a distributed block storage system for Kubernetes that provides persistent storage with replication and high availability capabilities. This guide demonstrates how to configure Longhorn with LUKS/AES-256 encryption to ensure data protection at rest.

68.1 Introduction

Data security is a critical requirement for modern cloud-native applications, particularly in environments handling sensitive information. Volume encryption provides a fundamental security layer that protects data stored on physical media from unauthorized access, even in scenarios where physical security has been compromised.

Longhorn implements volume-level encryption using industry-standard cryptographic technologies:

- **LUKS (Linux Unified Key Setup)**: A specification for block device encryption that provides a platform-independent standard on-disk format
- **AES-256**: Advanced Encryption Standard with 256-bit keys, providing robust cryptographic protection

68.2 Why implement volume encryption

Volume encryption addresses several security and compliance requirements:

- **Data Protection at Rest**: Encrypted data on physical storage media remains protected against unauthorized access, even if disks are physically removed or stolen.
- **Regulatory Compliance**: Industry regulations such as GDPR, HIPAA, and PCI-DSS mandate encryption for sensitive data, making volume encryption essential for compliance.
- **Defense in Depth**: Encryption provides an additional security layer that complements Kubernetes RBAC and network policies.
- **Secure Data Disposal**: Encrypted volumes can be securely decommissioned by simply destroying the encryption keys, ensuring data is rendered permanently inaccessible.

> **Note**
>
> Encryption is applied transparently at the volume level. Applications interact with encrypted volumes through standard Kubernetes persistent volume claims without requiring code modifications.

68.3 Architecture overview

Longhorn encryption integrates with the Kubernetes Container Storage Interface (CSI) to provide transparent volume encryption. The encryption process occurs during volume provisioning and is managed through the following components:

- **Kubernetes Secrets**: Encryption keys are stored securely as Kubernetes Secret resources
- **StorageClass Parameters**: Encryption configuration is specified through CSI-specific parameters
- **LUKS Device Mapper**: Each encrypted volume is backed by a LUKS-encrypted device mapper target
- **CSI Driver**: The Longhorn CSI driver manages encryption key retrieval and volume staging

The encryption workflow operates as follows:

1. When a PersistentVolumeClaim (PVC) requests storage from an encrypted StorageClass, the Longhorn CSI driver retrieves the encryption key from the designated Secret
2. A LUKS-encrypted device mapper target is created using the encryption key
3. All data written to the volume is encrypted before being committed to the underlying storage
4. When the volume is mounted, the CSI driver stages the volume by unlocking the LUKS device using the stored encryption key

68.4 Prerequisites

Before implementing Longhorn volume encryption, ensure the following prerequisites are met:

- A functional Kubernetes cluster with SUSE Storage 1.12.1 installed
- The V2 Data Engine enabled and prepared as described in *Chapter 66, SUSE Storage V2 Data Engine*

- Cluster administrative access with appropriate RBAC permissions
- `kubectl` command-line tool configured for cluster access
- `openssl` utility for cryptographic key generation

68.5 Implementation

1. **Generate the encryption key**

   Encryption key generation must be performed using cryptographically secure random number generation. Run the following command to generate a 256-bit encryption key encoded in `base64` format:

   ```
   openssl rand -base64 32
   ```

   Example output:

   ```
   Kq8Xh2vN9pL3mZ1wR4tY6uI0oP5aS7dF8gH9jK1lM2n=
   ```

   > **Warning**
   >
   > **Key Management Critical Requirements**
   >
   > - **Never commit encryption keys to version control systems**
   > - Loss of the encryption key results in **permanent and irreversible data loss**

2. **Create the encryption key Secret**

   Once generated, the encryption key must be stored as a Kubernetes Secret in the `longhorn-system` namespace:

   ```
   CRYPTO_KEY_VALUE="Kq8Xh2vN9pL3mZ1wR4tY6uI0oP5aS7dF8gH9jK1lM2n="

   kubectl create secret generic longhorn-crypto \
     --from-literal=CRYPTO_KEY_VALUE="${CRYPTO_KEY_VALUE}" \
     -n longhorn-system
   ```

   Verify successful Secret creation:

   ```
   kubectl get secret longhorn-crypto -n longhorn-system
   ```

> **Tip**
>
> In production environments, consider using external secret management solutions that integrate with Kubernetes, such as the External Secrets Operator or sealed-secrets, to avoid storing sensitive data directly in the cluster's etcd.

3. **Configure the encrypted StorageClass**

   a. Define a `StorageClass` that specifies encryption parameters for the Longhorn CSI driver:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: longhorn-encrypted
provisioner: driver.longhorn.io
allowVolumeExpansion: true
parameters:
  dataEngine: "v2"
  numberOfReplicas: "3"
  staleReplicaTimeout: "2880"
  fromBackup: ""
  fsType: "ext4"
  dataLocality: "disabled"
  # Encryption configuration
  encrypted: "true"
  csi.storage.k8s.io/provisioner-secret-name: "longhorn-crypto"
  csi.storage.k8s.io/provisioner-secret-namespace: "longhorn-system"
  csi.storage.k8s.io/node-publish-secret-name: "longhorn-crypto"
  csi.storage.k8s.io/node-publish-secret-namespace: "longhorn-system"
  csi.storage.k8s.io/node-stage-secret-name: "longhorn-crypto"
  csi.storage.k8s.io/node-stage-secret-namespace: "longhorn-system"
```

   The encryption-specific parameters configure CSI secret references:

   - `encrypted: "true"` - Enables LUKS encryption for all volumes provisioned from this StorageClass
   - `csi.storage.k8s.io/provisioner-secret-name` - Specifies the Secret containing the encryption key used during volume provisioning

- `csi.storage.k8s.io/node-publish-secret-name` - Identifies the Secret used when mounting volumes to pods
- `csi.storage.k8s.io/node-stage-secret-name` - References the Secret used during volume staging operations

b. Apply the StorageClass definition:

```
kubectl apply -f longhorn-encrypted-storageclass.yaml
```

c. Verify StorageClass creation and configuration:

```
kubectl get storageclass longhorn-encrypted
kubectl get storageclass longhorn-encrypted -o yaml
```

4. **Deploy a test workload**

To validate the encryption implementation, deploy a test pod that writes data to an encrypted volume.

a. Create the following manifest (`test-encryption.yaml`):

```
# Test Longhorn Encryption
# Deploys a simple pod that writes data to an encrypted volume

---
# PVC using the encrypted StorageClass
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: test-encrypted-pvc
  namespace: default
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: longhorn-encrypted
  resources:
    requests:
      storage: 1Gi

---
# Pod that writes sensitive data to the volume
apiVersion: v1
kind: Pod
metadata:
  name: encryption-test-pod
  namespace: default
```

```
  labels:
    app: encryption-test
spec:
  containers:
  - name: writer
    image: busybox
    command:
      - sh
      - -c
      - |
        echo "=== Writing sensitive data to encrypted volume ==="
        echo "SECRET DATA: Credit Card 4111-1111-1111-1111" > /data/secret.txt
        echo "SECRET DATA: Password MySuperSecret123!" >> /data/secret.txt
        echo "SECRET DATA: API Key sk-1234567890abcdef" >> /data/secret.txt
        date >> /data/secret.txt
        echo "=== Data written successfully ==="
        echo "Content of /data/secret.txt:"
        cat /data/secret.txt
        echo "=== Sleeping forever (use 'kubectl exec' to verify) ==="
        sleep infinity
    volumeMounts:
    - name: encrypted-storage
      mountPath: /data
  volumes:
  - name: encrypted-storage
    persistentVolumeClaim:
      claimName: test-encrypted-pvc
```

This manifest defines:

- A `PersistentVolumeClaim` requesting 1Gi of storage from the `longhorn-encrypted` StorageClass
- A Pod that writes test data to the encrypted volume and remains running for verification

b. Deploy the test workload:

```
kubectl apply -f test-encryption.yaml
```

c. Monitor pod startup:

```
kubectl get pod encryption-test-pod -w
```

5. **Verify volume functionality**

Once the pod reaches `Running` state, verify that data operations complete successfully:

a. View pod logs:

```
kubectl logs encryption-test-pod
```

b. Run a shell inside the pod:

```
kubectl exec -it encryption-test-pod -- sh
```

c. Inside the pod, verify the content:

```
cat /data/secret.txt
ls -lh /data/
```

Expected output:

```
SECRET DATA: Credit Card 4111-1111-1111-1111
SECRET DATA: Password MySuperSecret123!
SECRET DATA: API Key sk-1234567890abcdef
Thu Jul 03 10:30:45 UTC 2026
```

> **Note**
>
> Successful data operations confirm volume provisioning and mounting functionality. However, this verification alone does not validate encryption. The following steps provide cryptographic verification.

6. **Cryptographic verification**

   To conclusively verify that encryption is active, direct inspection of the underlying storage device is required.

   a. **Identify the volume location**

   Determine which node hosts the volume replica:

```
# Get the PVC details
kubectl get pvc test-encrypted-pvc

# Get the associated PersistentVolume
kubectl get pv | grep test-encrypted

# View Longhorn volume details
```

```
kubectl get volumes -n longhorn-system | grep test-encrypted
```

b. **Access the node and locate volume files**

SSH to the node hosting the volume (identified from the previous output):

```
# Connect to the node
ssh root@<node-ip>

# Locate the volume image file
find /var/lib/longhorn -name "*.img*" -type f | grep -i test
```

Typical volume path:

```
/var/lib/longhorn/replicas/pvc-abc12345-r-6789/volume-head-001.img
```

c. **Attempt plaintext data extraction**

Run the following command to search for unencrypted data:

```
# Attempt to find plaintext strings in the volume file
strings /var/lib/longhorn/replicas/pvc-*/volume-head-*.img | grep -i "credit
 card"
```

Verification results:

- **Encryption active**: No plaintext strings found (data is encrypted)
- **Encryption inactive**: Plaintext strings detected (data is unencrypted)

d. **Hexadecimal inspection**

Examine the raw byte content of the volume:

```
# Navigate to the replica directory
cd /var/lib/longhorn/replicas/pvc-XXXXX-r-YYYY/

# Display raw volume content
hexdump -C volume-head-*.img | head -100
```

Encrypted volume output (random bytes):

```
00000000  a3 7f 2c d9 8b 4e f1 2a  9c 5d 8e 6b 4f 1d 3c 7a  |..,..N.*.[.kO.<z|
00000010  d2 9f 4a 8c 3b e7 6d 1e  5f 0c b8 a4 7d 2f 9e 6c  |..J.;.m._...}/.l|
```

Unencrypted volume output (readable ASCII):

```
00000000  53 45 43 52 45 54 20 44  41 54 41 3a 20 43 72 65  |SECRET DATA: Cre|
```

```
00000010  64 69 74 20 43 61 72 64  20 34 31 31 31 2d 31 31  |dit Card 4111-11|
```

7. **LUKS device verification**

   Longhorn implements encryption using the Linux LUKS subsystem. Verify LUKS device mapper targets:

   ```
   # List block devices with encryption type
   lsblk -o NAME,FSTYPE,SIZE,MOUNTPOINT | grep -i crypt

   # List device mapper crypt targets
   dmsetup ls --target crypt
   ```

   The presence of `crypto_LUKS` filesystem types or device mapper crypt targets confirms active LUKS encryption.

8. **Data persistence verification**

   Verify that encrypted volumes maintain data integrity across pod lifecycle events:

   ```
   # Delete the pod (volume persists)
   kubectl delete pod encryption-test-pod

   # Recreate the pod
   kubectl apply -f test-encryption.yaml

   # Wait for Running state
   kubectl get pod encryption-test-pod -w

   # Verify data persistence
   kubectl exec -it encryption-test-pod -- cat /data/secret.txt
   ```

   Data should remain intact, demonstrating proper volume persistence.

68.6 Encryption verification summary

The following table summarizes encryption verification indicators:

Verification Method	Encrypted	Not Encrypted
`strings` output	No plaintext found	Plaintext strings visible
`hexdump` output	Random binary data	Readable ASCII text
`lsblk` output	`crypto_LUKS` type present	No `crypto_LUKS` type

Verification Method	Encrypted	Not Encrypted
`dmsetup ls --target crypt`	Lists crypt devices	No crypt devices listed

68.7 Test environment cleanup

Remove test resources:

```
# Delete the test pod
kubectl delete -f test-encryption.yaml

# Delete the PVC and volume
kubectl delete pvc test-encrypted-pvc
```

68.8 CAPI integration

When deploying clusters using Cluster API (CAPI) with Metal[3], encryption configuration can be embedded in cluster provisioning manifests, enabling encrypted storage from initial cluster bootstrap.

A complete reference implementation is available in the SUSE Edge telco-cloud-examples repository (https://github.com/suse-edge/telco-cloud-examples/tree/main/telco-examples/downstream-clusters/longhorn/longhorn-with-encryption).

This reference implementation includes:

- **BareMetalHost** definitions for multi-node control plane deployment
- **CAPI cluster manifests** with integrated Longhorn encryption configuration
- **Automatic Secret provisioning** during cluster initialization
- **Pre-configured encrypted StorageClass** set as default
- **MetalLB** configuration for VIP management
- **Endpoint Copier Operator** for control plane endpoint synchronization

In CAPI-based deployments, the encryption key is typically provided through cluster provisioning variables:

```
# Example variable reference from CAPI manifest
```

```
# Full manifests available in the repository
CRYPTO_KEY_VALUE: "YOUR_GENERATED_KEY_HERE"
```

This variable is consumed during cluster provisioning to automatically create the `longhorn-crypto` Secret in the downstream cluster's `longhorn-system` namespace.

Caution

CAPI encryption deployment considerations:

- Generate encryption keys using `openssl rand -base64 32` before cluster provisioning
- Replace placeholder values in manifests with actual encryption keys
- **Never commit manifests containing real encryption keys to version control**
- Implement GitOps workflows with secret management solutions
- Maintain encryption keys in enterprise secret management systems for disaster recovery
- Use unique encryption keys per cluster to limit blast radius in case of key compromise

X Tips and Tricks

69 Edge Image Builder 540

70 **Metal³** 542

Tips and tricks for Edge components

69 Edge Image Builder

69.1 Common

- If you are in a non-Linux environment and following these instructions to build an image, then you are likely running `Podman` via a virtual machine. By default, this virtual machine will be configured to have a small amount of system resources allocated to it and can cause instability for `Edge Image Builder` during resource intensive operations, such as the RPM resolution process. You will need to adjust the resources of the podman machine, either by using Podman Desktop (settings cogwheel → podman machine edit icon) or directly via the `podman-machine-set` command (https://docs.podman.io/en/stable/markdown/podman-machine-set.1.html)

- At this point in time, the `Edge Image Builder` is not able to build images in a cross architecture setup, i.e. you have to run it on:

  - AArch64 systems (such as Apple Silicon) to build SL Micro `aarch64` images

  - AMD64/Intel 64 systems to build SL Micro `x86_64` images.

69.2 SUSE Linux Micro

- Loading kernel modules at boot can be done using the corresponding `/etc/modprobe.d/module.conf` file. Create the corresponding `os-files` folder using Edge Image Builder:

```
.
├── definition.yaml
└── os-files
    └── etc
        └── modprobe.d
            └── module.conf
```

For more information, please refer to the "Managing kernel modules" section of the SUSE Linux Enterprise Server Documentation (https://documentation.suse.com/sles/15-SP7/html/SLES-all/cha-mod.html#sec-mod-modprobe-d)

69.3 Kubernetes

- Creating multi node Kubernetes clusters requires adjusting the `kubernetes` section in the definition file to:
  - list all server and agent nodes under `kubernetes.nodes`
  - set a virtual IP address that would be used for all non-initializer nodes to join the cluster under `kubernetes.network.apiVIP`
  - optionally, set an API host to specify a domain address for accessing the cluster under `kubernetes.network.apiHost` To learn more about this configuration, please refer to the Kubernetes section docs (https://github.com/suse-edge/edge-image-builder/blob/main/docs/building-images.md#kubernetes).
- `Edge Image Builder` relies on the hostnames of the different nodes to determine their Kubernetes type (`server` or `agent`). While this configuration is managed in the definition file, for the general networking setup of the machines we can utilize either DHCP configuration as described in *Chapter 13, Edge Networking*.

70 Metal³

70.1 BareMetalHost selection and Cluster association

Once a Metalˆ3ˆ cluster object and its corresponding associated objects are created, a process to choose which `BareMetalHost` will be part of the cluster is performed. This process connects a `BareMetalHost` with a specific `Metal3MachineTemplate` using standard Kubernetes labels (https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/) and selectors.

As an example, each `BareMetalHost` is labeled to identify its properties and intended cluster (e.g., its cluster-role, the cluster name, location, etc.):

```
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: mynode1
  labels:
    cluster-role: control-plane
    cluster: foobar
    location: madrid
    datacenter: xyz
<snip>
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: mynode2
  labels:
    cluster-role: worker
    cluster: foobar
    location: madrid
    datacenter: xyz
<snip>
---
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
metadata:
  name: mynode3
  labels:
    cluster-role: worker
    cluster: foobar2
    location: madrid
```

```
    datacenter: xyz
<snip>
...
```

Then, the `Metal3MachineTemplate` object uses the `spec.hostSelector` (https://doc.crds.dev/github.com/metal3-io/cluster-api-provider-metal3/infrastructure.cluster.x-k8s.io/Metal3MachineTemplate/v1beta1@v1.13.2#spec-template-spec-hostSelector) field to match the desired `BareMetalHost`.

Both `matchLabels` (https://doc.crds.dev/github.com/metal3-io/cluster-api-provider-metal3/infrastructure.cluster.x-k8s.io/Metal3MachineTemplate/v1beta1@v1.13.2#spec-template-spec-hostSelector-matchLabels) (for exact key-value matching) and `matchExpressions` (https://doc.crds.dev/github.com/metal3-io/cluster-api-provider-metal3/infrastructure.cluster.x-k8s.io/Metal3MachineTemplate/v1beta1@v1.13.2#spec-template-spec-hostSelector-matchExpressions) (for more complex rules) can be used:

```
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: foobar-cluster-controlplane
  namespace: mynamespace
spec:
  template:
    spec:
      hostSelector:
        matchLabels:
          cluster-role: control-plane
          cluster: foobar
<snip>
---
apiVersion: infrastructure.cluster.x-k8s.io/v1beta2
kind: Metal3MachineTemplate
metadata:
  name: foobar-cluster-worker
  namespace: mynamespace
spec:
  template:
    spec:
      hostSelector:
        matchExpressions:
          - { key: cluster-role, operator: In, values: [worker] }
          - { key: cluster, operator: In, values: [foobar] }
<snip>
```

> **Note**
> Kubernetes namespaces can be also used to better organize the different objects.

70.2 Clean up old EFI boot entries

Sometimes, the UEFI boot manager (https://en.wikipedia.org/wiki/UEFI#UEFI_booting) contains multiple entries for older operating systems that are probably not needed anymore (especially for host being re-provisioned multiple times). You can clean up those old entries by following any of the following procedures:

- Delete them on the BIOS/EFI setup interface directly (the exact procedure will depend on the hardware).
- Run the UEFI `bcfg` (https://uefi.org/sites/default/files/resources/UEFI_Shell_2_2.pdf) shell as:

```
# List the entries
bcfg boot dump -b
# Delete entry number X
bcfg boot rm X
# X is the number associated the entry to remove. For example, if the entry is
 "Boot0002 foobar", then X is 2.
```

- Use `efibootmgr` on a Linux system as:

```
# List the entries
efibootmgr -v
# Delete entry number X
efibootmgr -b X -B
```

The process may leave orphaned files on the EFI System Partition (ESP), usually found under subdirectories named by the vendor (e.g., `EFI/opensuse` or `EFI/Microsoft`). While these files are generally harmless, they should be deleted if they consume excessive space as it can prevent the installation of a new OS or a boot manager update. Removal may require explicitly mounting the ESP, typically mounted as `/boot/efi/EFI` on Linux systems.

70.3 Custom network configuration using the two-secrets approach

When Metal[3] provisions a bare metal node, it goes through two distinct phases that may each require different network configuration:

- The **IPA phase**, where the Ironic Python Agent (IPA) ramdisk runs during hardware inspection and provisioning
- The **target OS phase**, where the deployed SLE Micro system runs after first boot

The two-secrets approach addresses this by allowing a separate network configuration secret for each phase, using the `preprovisioningNetworkDataName` field for the IPA phase and the `networkData` field for the target OS phase. This is particularly useful when interface names differ between phases, which can happen because the IPA kernel and the SLE Micro kernel may discover the same hardware under different names.

70.3.1 Example of interface renaming for VLANs

A common scenario is when hardware gets a long PCI-based interface name such as `enp1s0np123`. Adding a VLAN on top of it may exceed the Linux kernel hard limit of **15 characters** for interface names:

```
enp1s0np123.100   = 15 chars  (barely fits, risky)
enp1s0np123.3669  = 17 chars  (exceeds limit, fails)
eth0.3669         =  9 chars  (works)
```

The IPA phase must reference `enp1s0np123` (the kernel-discovered name), while the target OS should use a short name like `eth0` so that `eth0.3669` stays under the limit. `nmc` (nm-configurator) bridges the two phases by matching interfaces via MAC address rather than name — you declare `name: eth0` alongside the hardware MAC address, and `nmc` creates the NetworkManager profile with the desired name regardless of what the kernel assigned.

70.3.2 Prerequisites:

70.3.2.1 EIB image setup

As per the static network configuration guide (https://documentation.suse.com/suse-edge/3.7/html/edge/quickstart-metal3.html#id-configuring-static-ips) the EIB image must include a first-boot script that reads the network configuration from the `config-2` partition Metal[3] writes during provisioning. Create the following script at `/opt/EIB/network/configure-network.sh`:

```
#!/bin/bash
set -eux

# Source: https://documentation.suse.com/suse-edge/3.7/html/edge/quickstart-
metal3.html#metal3-add-network-eib

CONFIG_DRIVE=$(blkid --label config-2 || true)
if [ -z "${CONFIG_DRIVE}" ]; then
  echo "No config-2 device found, skipping network configuration"
  exit 0
fi

mount -o ro $CONFIG_DRIVE /mnt

NETWORK_DATA_FILE="/mnt/openstack/latest/network_data.json"

if [ ! -f "${NETWORK_DATA_FILE}" ]; then
  umount /mnt
  echo "No network_data.json found, skipping network configuration"
  exit 0
fi

DESIRED_HOSTNAME=$(cat /mnt/openstack/latest/meta_data.json | tr ',\{\}' '\n' | grep
 '\"metal3-name\"' | sed 's/.*\"metal3-name\": \"\(.*\)\"/\1/')
echo "${DESIRED_HOSTNAME}" > /etc/hostname

mkdir -p /tmp/nmc/\{desired,generated\}
cp ${NETWORK_DATA_FILE} /tmp/nmc/desired/_all.yaml
umount /mnt

./nmc generate --config-dir /tmp/nmc/desired --output-dir /tmp/nmc/generated
./nmc apply --config-dir /tmp/nmc/generated
```

Then make it executable and build the EIB image as normal:

```
mkdir -p /opt/EIB/network
```

```
chmod +x /opt/EIB/network/configure-network.sh
```

Note

=== EIB automatically picks up scripts from the `network/` directory. Combustion runs them on first boot in initramfs, before the full OS starts. ===

Note

=== The script also sets the node hostname from Metal[3]'s `metal3-name` metadata field. ===

70.3.3 Configuring the two secrets

The following examples use dummy values throughout: data NIC MAC `aa:bb:cc:11:22:33`, boot NIC MAC `aa:bb:cc:44:55:66`, node IP `10.0.0.10/24`, gateway `10.0.0.1`, DNS `10.0.0.53`, VLAN ID `100`, and BMC address `10.1.0.10`.

Secret 1 — IPA phase (`static-networkdata-ipa.yaml`): references the kernel-assigned interface name. DHCP is used here to keep it simple during hardware discovery:

```
apiVersion: v1
kind: Secret
metadata:
  name: static-networkdata-ipa
  namespace: default
type: Opaque
stringData:
  networkData: |
    interfaces:
    - name: enp1s0np123
      type: ethernet
      state: up
      mac-address: "aa:bb:cc:11:22:33"
      ipv4:
        enabled: true
        dhcp: true
    dns-resolver:
      config:
        server:
        - 10.0.0.53
```

Secret 2 — target OS phase (`static-networkdata-os.yaml`): references the desired short name and declares the VLAN. The same MAC address is used so `nmc` can match the interface:

```
apiVersion: v1
kind: Secret
metadata:
  name: static-networkdata-os
  namespace: default
type: Opaque
stringData:
  networkData: |
    interfaces:
    - name: eth0
      type: ethernet
      state: up
      mac-address: "aa:bb:cc:11:22:33"
      mtu: 1500
      ipv4:
        enabled: false
        dhcp: false
    - name: eth0.100
      type: vlan
      state: up
      mtu: 1500
      vlan:
        base-iface: eth0
        id: 100
      ipv4:
        address:
        - ip: 10.0.0.10
          prefix-length: 24
        enabled: true
        dhcp: false
    dns-resolver:
      config:
        server:
        - 10.0.0.53
    routes:
      config:
      - destination: 0.0.0.0/0
        next-hop-address: 10.0.0.1
        next-hop-interface: eth0.100
```

The `BareMetalHost` object references both secrets:

```
apiVersion: metal3.io/v1alpha1
kind: BareMetalHost
```

```
metadata:
  name: my-node
  namespace: default
spec:
  online: true
  bootMACAddress: "aa:bb:cc:44:55:66"
  rootDeviceHints:
    deviceName: /dev/nvme0n1
  bmc:
    address: redfish-virtualmedia://10.1.0.10/redfish/v1/Systems/1/
    disableCertificateVerification: true
    credentialsName: my-node-credentials
  preprovisioningNetworkDataName: static-networkdata-ipa
  networkData:
    name: static-networkdata-os
```

Warning

`preprovisioningNetworkDataName` is a plain string field, while `networkData` is a SecretReference object requiring a `name:` sub-key. The syntax differs between the two and is a common source of errors.

Apply all objects:

```
kubectl apply -f bmc-credentials.yaml
kubectl apply -f static-networkdata-ipa.yaml
kubectl apply -f static-networkdata-os.yaml
kubectl apply -f baremetalhost.yaml
```

After provisioning, SSH to the node and verify:

```
# Interface names
ip link show
# Expected: eth0 and eth0.100@eth0

# IP on VLAN interface
ip addr show eth0.100

# NetworkManager profiles
nmcli connection show

# VLAN details
nmcli connection show eth0.100 | grep -E '(vlan.parent|vlan.id)'
```

XI Troubleshooting

71 General Troubleshooting Principles 551

72 Troubleshooting Kiwi 552

73 Troubleshooting Edge Image Builder (EIB) 554

74 Troubleshooting Edge Networking (NMC) 556

75 Troubleshooting Directed-network provisioning 558

76 Troubleshooting Other components 563

77 Collecting Diagnostics for Support 564

This section provides guidance to diagnose and resolve common issues with SUSE Telco Cloud deployments and operations. It covers various topics, offering component-specific troubleshooting steps, key tools, and relevant log locations.

71 General Troubleshooting Principles

Before diving into component-specific issues, consider these general principles:

- **Check logs**: Logs are the primary source of information. Most of the times the errors are self explanatory and contain hints on what failed.
- **Check clocks**: Having clock differences between systems can lead to all kinds of different errors. Ensure clocks are in sync. EIB can be instructed to force clock sync at boot time, see Configuring OS Time (*Chapter 5, Standalone clusters with Edge Image Builder*).
- **Boot Issues**: If the system is stuck during boot, note down the last messages displayed. Access the console (physical or via BMC) to observe boot messages.
- **Network Issues**: Verify network interface configuration (`ip a`), routing table (`ip route`), test connectivity from/to other nodes and external services (`ping`, `nc`). Ensure firewall rules are not blocking necessary ports.
- **Verify component status**: Use `kubectl get` and `kubectl describe` for Kubernetes resources. Use `kubectl get events --sort-by='.lastTimestamp' -n <namespace>` to see the events on a particular Kubernetes namespace.
- **Verify services status**: Use `systemctl status <service>` for systemd services.
- **Check syntax**: Software expects certain structure and syntax on configuration files. For yaml files, for example, use `yamllint` or similar tools to verify the proper syntax.
- **Isolate the problem**: Try to narrow down the issue to a specific component or layer (for example, network, storage, OS, Kubernetes, Metal[3], Ironic,…).
- **Documentation**: Always refer to the official SUSE Telco Cloud documentation (https://documentation.suse.com/suse-edge/) and also upstream documentation for detailed information.
- **Versions**: SUSE Telco Cloud is an opinionated and thoroughly tested version of different SUSE components. The versions of each component per SUSE Telco Cloud release can be observed in the SUSE Telco Cloud support matrix (https://documentation.suse.com/suse-edge/support-matrix/html/support-matrix/index.html).
- **Known issues**: For each SUSE Telco Cloud release there is a “Known issues” section on the release notes that contains information of issues that will be fixed on future releases but can affect the current one.

72 Troubleshooting Kiwi

Kiwi is used to generate updated SUSE Linux Micro images to be used with Edge Image Builder.

COMMON ISSUES

- **SL Micro Version Mismatch**: The build host operating system version must match the operating system version being built (SL Micro 6.0 host → SL Micro 6.0 image).
- **SELinux in Enforcing State**: Due to certain limitations, it is currently required to disable SELinux temporarily to be able to build images with Kiwi. Check the SElinux status with `getenforce` and disable it before running the build process with `setenforce 0`.
- **Build host not registered**: The build process uses the build host subscriptions to be able to pull packages from SUSE SCC. If the host is not registered it fails.
- **Loop Device Test Failure**: The first time that the Kiwi build process is executed, it will fail shortly after starting with "ERROR: Early loop device test failed, please retry the container run.", this is a symptom of loop devices being created on the underlying host system that are not immediately visible inside of the container image. Re-run the Kiwi build process again and it should proceed without issue.
- **Missing Permissions**: The build process expects to be run as root user (or via sudo).
- **Wrong Privileges**: The build process expects the `--privileged` flag when running the container. Double-check that it is present.

LOGS

- **Build container logs**: Check the logs of the build container. The logs are generated in the directory that was used to store the artifacts. Check docker logs or podman logs for the necessary information as well.
- **Temporary build directories**: Kiwi creates temporary directories during the build process. Check these for intermediate logs or artifacts if the main output is insufficient.

TROUBLESHOOTING STEPS

1. **Review `build-image` output**: The error message in the console output is usually very indicative.
2. **Check build environment**: Ensure all prerequisites for Kiwi itself (for example, docker/podman, SElinux, sufficient disk space) are met on the machine running Kiwi.

3. **Inspect build container logs**: Review the logs of the failed container for more detailed errors (see above).

4. **Verify definition file**: If you are using a custom Kiwi image definition file, double-check the file for any typos or syntax.

Note

Check the Kiwi Troubleshooting Guide (https://documentation.suse.com/appliance/kiwi-9/html/kiwi/troubleshooting.html).

73 Troubleshooting Edge Image Builder (EIB)

EIB is used to create custom SUSE Telco Cloud images.

COMMON ISSUES

- **Wrong SCC code**: Ensure the SCC code used in the EIB definition file matches the SL Micro version and architecture.
- **Missing dependencies**: Ensure there are no missing packages or tools within the build environment.
- **Incorrect image size**: For raw images, the `diskSize` parameter is required and it depends heavily on the images, RPMs, and other artifacts being included in the image.
- **Permissions**: If storing a script on the custom/files directory, ensure it has executable permissions as those files are just available at combustion time but no changes are performed by EIB.
- **Operating system group dependencies**: When creating an image with custom users and groups, the groups being set as "`primaryGroup`" should be explicitly created.
- **Operating system user's sshkeys requires a home folder**: When creating an image with users with sshkeys, the home folder needs to be created as well with `createHomeDir=true`.
- **Combustion issues**: EIB relies on combustion for the customization of the OS and deployment of all the other SUSE Telco Cloud components. This also includes custom scripts being placed in the custom/scripts folder. Note that the combustion process is being executed at `initrd` time, so the system is not completely booted when the scripts are executed.
- **Podman machine size**: As explained in the EIB Tips and Tricks section (*Part X, "Tips and Tricks"*), verify the podman machine has enough CPU/memory to run the EIB container on non-Linux operating systems.
- **Incorrect image**: Ensure the base image being used is properly downloaded by verifying the `checksum` (https://www.suse.com/support/security/download-verification/). If you are building the image with kiwi-builder (*Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi*), check the sum file generated by the process as well.

LOGS

- **EIB output**: The console output of the `eib build` command is crucial.
- **Build container logs**: Check the logs of the build container. The logs are generated in the directory that was used to store the artifacts. Check `docker logs` or `podman logs` for the necessary information as well.

> **Note**
>
> For more information, see Debugging (https://github.com/suse-edge/edge-image-builder/blob/main/docs/debugging.md).

- **Temporary build directories**: EIB creates temporary directories during the build process. Check these for intermediate logs or artifacts if the main output is insufficient.
- **Combustion logs**: If the image being built with EIB does not boot for any reason, a root shell is available. Connect to the host console (either physically, via BMC, etc.) and check combustion logs with `journalctl -u combustion` and in general all the operating system logs with `journalctl` to find the root cause of the failure.

TROUBLESHOOTING STEPS

1. **Review `eib-build` output**: The error message in the console output is usually very indicative.
2. **Check build environment**: Ensure all prerequisites for EIB itself (for example, docker/podman, sufficient disk space) are met on the machine running EIB.
3. **Inspect build container logs**: Review the logs of the failed container for more detailed errors (see above).
4. **Verify `eib` configuration"**: Double-check the `eib` configuration file for any typos or incorrect paths to source files or build scripts.
    - **Test components individually**: If your EIB build involves custom scripts or stages, run them independently to isolate failures.

> **Note**
>
> Check Edge Image Builder Debugging (https://github.com/suse-edge/edge-image-builder/blob/main/docs/debugging.md).

74 Troubleshooting Edge Networking (NMC)

NMC is injected on SL Micro EIB images to configure the network of the Edge hosts at boot time via combustion. It is also being executed on the Metal3 workflow as part of the inspection process. Issues can happen when the host is being booted for the first time or on the Metal3 inspection process.

COMMON ISSUES

- **Host not being able to boot properly the first time**: Malformed network definition files can lead to the combustion phase to fail and then the host drops a root shell.
- **Files are not properly generated**: Ensure the network files matches NMState (https://nmstate.io/examples.html) format.
- **Network interfaces are not correctly configured**: Ensure the MAC addresses match the interfaces being used on the host.
- **Mismatch between interface names**: SL Micro enables Predictable Naming Scheme for Network Interfaces (https://documentation.suse.com/smart/network/html/network-interface-predictable-naming/index.html) by default so there is no `eth0` anymore but other naming schema such as `enp2s0`.

LOGS

- **Combustion logs**: As nmc is being used at combustion time, check combustion logs with `journalctl -u combustion` on the host being provisioned.
- **NetworkManager logs**: On the Metal^3 deployment workflow, nmc is part of the IPA execution and it is being executed as a dependency of the NetworkManager service using systemd's ExecStartPre functionality. Check NetworkManager logs on the IPA host as `journalctl -u NetworkManager` (see the Troubleshooting Directed-network provisioning (*Chapter 75, Troubleshooting Directed-network provisioning*) section to understand how to access the host when booted with IPA).

TROUBLESHOOTING STEPS

1. **Verify the yaml syntax**: nmc configuration files are yaml files, check the proper syntax with `yamllint` or similar tools.
2. **Run nmc manually**: As nmc is part of the EIB container, to debug any issues, a local podman command can be used.
    a. Create a temporary folder to store the nmc files.

```
mkdir -p ${HOME}/tmp/foo
```

b. Save the nmc files on that location.

```
❯ tree --noreport ${HOME}/tmp/foo
/Users/johndoe/tmp/foo
├── host1.example.com.yaml
└── host2.example.com.yaml
```

c. Run the EIB container with nmc as the entrypoint and the generate command to perform the same tasks nmc would do at combustion time:

```
podman run -it --rm -v ${HOME}/tmp/foo:/tmp/foo:Z --entrypoint=/usr/bin/nmc
 registry.suse.com/edge/3.7/edge-image-builder:1.3.4 generate --config-dir /
tmp/foo --output-dir /tmp/foo/

[2025-06-04T11:58:37Z INFO  nmc::generate_conf] Generating config from "/tmp/
foo/host2.example.com.yaml"...
[2025-06-04T11:58:37Z INFO  nmc::generate_conf] Generating config from "/tmp/
foo/host1.example.com.yaml"...
[2025-06-04T11:58:37Z INFO  nmc] Successfully generated and stored network
 config
```

d. Observe the logs and files being generated on the temporary folder.

75 Troubleshooting Directed-network provisioning

Directed-network provisioning scenarios involve using Metal^3 and CAPI elements to provision the Downstream cluster. It also includes EIB to create an OS image. Issues can happen when the host is being booted for the first time or during the inspection or provisioning processes.

COMMON ISSUES

- **Old firmware**: Verify all the different firmware on the physical hosts being used are up to date. This includes the BMC firmware as some times Metal^3 requires specific/updated ones (https://book.metal3.io/bmo/supported_hardware#redfish-and-its-variants).
- **Provisioning failed with SSL errors**: If the webserver serving the images uses https, Metal^3 needs to be configured to inject and trust the certificate on the IPA image. See Kubernetes folder (*Section 29.4, "Kubernetes folder"*) on how to include a `ca-additional.crt` file to the Metal^3 chart.
- **Certificates issues when booting the hosts with IPA**: Some server vendors verify the SSL connection when attaching virtual-media ISO images to the BMC, which can cause a problem because the generated certificates for the Metal3 deployment are self-signed. It can happen that the host is being booted but it drops to an UEFI shell. See Disabling TLS for virtualmedia ISO attachment (*Section 4.7.2, "Disabling TLS for virtualmedia ISO attachment"*) on how to fix it.
- **Wrong name or label reference**: If the cluster references a node by the wrong name or label, the cluster results as deployed but the BMH remains as "Available". Double-check the references on the involved objects for the BMHs.
- **BMC communication issues**: Ensure the Metal^3 pods running on the management cluster can reach the BMC of the hosts being provisioned (usually the BMC network is very restricted).
- **Incorrect bare metal host state**: The BMH object goes to different states (inspecting, preparing, provisioned, etc.) during its lifetime Lifetime of State machine (https://book.metal3.io/bmo/state_machine). If detected an incorrect state, check the `status` field of the BMH object as it contains more information as `kubectl get bmh <name> -o jsonpath='{.status}'| jq`.
- **Host not being deprovisioned**: In the event of a host being intended to be deprovisioned fails, the removal can be attempted after adding the "detached" annotation to the BMH object as: `kubectl annotate bmh/<BMH> baremetalhost.metal3.io/detached=""`.
- **Image errors**: Verify the image being built with EIB for the downstream cluster is available, has a proper checksum and it is not too large to decompress or too large for disk.

- **Disk size mismatch**: By default, the disk would not expand to fill the whole disk. As explained in the Growfs script (*Section 4.4.4.1.2, "Growfs script"*) section, a growfs script needs to be included in the image being built with EIB for the downstream cluster hosts.
- **Cleaning process stuck**: The cleaning process is retried several times. If due to a problem with the host cleaning is no longer possible, disable cleaning first by setting the `automatedCleanMode` field to `disabled` on the BMH object.

> **Warning**
>
> It is not recommended to manually remove the finalizer when the cleaning process is taking longer than desired or is failing. Doing so, removes the host record from Kubernetes but leave it in Ironic. The currently running action continues in the background, and an attempt to add the host again may fail because of the conflict.

- **Metal3/Rancher Turtles/CAPI pods issues**: The deployment flow for all the required components is:
  - The Rancher Turtles controller deploys the CAPI operator controller.
  - The CAPI operator controller then deploys the provider controllers (CAPI core, CAPM3 and RKE2 controlplane/bootstrap).

Verify all the pods are running correctly and check the logs otherwise.

LOGS

- **Metal^3 logs**:Check logs for the different pods.

```
kubectl logs -n metal3-system -l app.kubernetes.io/component=baremetal-operator
kubectl logs -n metal3-system -l app.kubernetes.io/component=ironic
```

> **Note**
>
> The metal3-ironic pod contains at least 4 different containers (`ironic-httpd`,` ironic-log-watch`, `ironic` & `ironic-ipa-downloader` (init)) on the same pod. Use the `-c` flag when using `kubectl logs` to verify the logs of each of the containers.

> **Note**
>
> The `ironic-log-watch` container exposes console logs from the hosts after inspection/provisioning, provided network connectivity enables sending these logs back to the management cluster. This can be useful in cases where there are provisioning errors but you do not have direct access to the BMC console logs.

- **Rancher Turtles logs**: Check logs for the different pods.

```
kubectl logs -n cattle-turtles-system -l control-plane=controller-manager
kubectl logs -n cattle-turtles-system -l app.kubernetes.io/name=cluster-api-operator
kubectl logs -n rke2-bootstrap-system -l cluster.x-k8s.io/provider=bootstrap-rke2
kubectl logs -n rke2-control-plane-system -l cluster.x-k8s.io/provider=control-
plane-rke2
kubectl logs -n cattle-capi-system -l cluster.x-k8s.io/provider=cluster-api
kubectl logs -n capm3-system -l cluster.x-k8s.io/provider=infrastructure-metal3
```

- **BMC logs**: Usually BMCs have a UI where most of the interaction can be done. There is usually a "logs" section that can be observed for potential issues (not being able to reach the image, hardware failures, etc.).

- **Console logs**: Connect to the BMC console (via the BMC webui, serial, etc.) and check for errors on the logs being written.

TROUBLESHOOTING STEPS

1. **Check `BareMetalHost` status**:

   - `kubectl get bmh -A` shows the current state. Look for `provisioning`, `ready`, `error`, `registering`.

   - `kubectl describe bmh -n <namespace> <bmh_name>` provides detailed events and conditions explaining why a BMH might be stuck.

2. **Test RedFish connectivity**:

   - Use `curl` from the Metal[3] control plane to test connectivity to the BMCs via redfish.

   - Ensure correct BMC credentials are provided in the `BareMetalHost-Secret` definition.

3. **Verify turtles/CAPI/metal3 pod status**: Ensure the containers on the management cluster are up and running: `kubectl get pods -n metal3-system` and `kubectl get pods -n cattle-turtles-system` (also see `cattle-capi-system`, `capm3-system`, `rke2-bootstrap-system` and `rke2-control-plane-system`).

4. **Verify the ironic endpoint is reachable from the host being provisioned**: The host being provisioned needs to be able to reach out the Ironic endpoint to report back to Metal^3. Check the IP with `kubectl get svc -n metal3-system metal3-metal3-ironic` and try to reach it via `curl/nc`.

5. **Verify the IPA image is reachable from the BMC**: IPA is being served by the Ironic endpoint and it needs to be reachable from the BMC as it is being used as a virtual CD.

6. **Verify the OS image is reachable from the host being provisioned**: The image being used to provision the host needs to be reachable from the host itself (when running IPA) as it will be downloaded temporarily and written to the disk.

7. **Examine Metal^3 component logs**: See above.

8. **Retrigger BMH Inspection**: If an inspection failed or the hardware of an available host changed, a new inspection process can be triggered by annotating the BMH object with `inspect.metal3.io: ""`. See the Metal^3 Controlling inspection (https://book.metal3.io/bmo/inspect_annotation) guide for more information.

9. **Bare metal IPA console**: To troubleshoot IPA issues a couple of alternatives exist:

   - Enable "autologin". This enables the root user to be logged automatically when connecting to the IPA console.

   > **Warning**
   >
   > This is only for debug purposes as it gives full access to the host.

   To enable autologin, the Metal3 helm `global.ironicKernelParams` value should look like: `console=ttyS0 suse.autologin=ttyS0` (depending on the console, `ttyS0` can be changed). Then a redeployment of the Metal^3 chart should be performed. (Note `ttyS0` is an example, this should match the actual terminal e.g may be `tty1` in many cases on bare metal, this can be verified by looking at the console output from the IPA ramdisk on boot where `/etc/issue` prints the console name).

Another way to do it is by changing the `IRONIC_KERNEL_PARAMS` parameter on the `ironic` configmap on the `metal3-system` namespace. This can be easier as it can be done via `kubectl` edit but it will be overwritten when updating the chart. Then the Metal^3 pod needs to be restarted with `kubectl delete pod -n metal3-system -l app.kubernetes.io/component=ironic`.

- Inject an ssh key for the root user on the IPA.

Warning

This is only for debug purposes as it gives full access to the host.

To inject the ssh key for the root user, the Metal^3 helm `debug.ironicRamdiskSshKey` value should be used. Then a redeployment of the Metal^3 chart should be performed.
Another way to do it is by changing the `IRONIC_RAMDISK_SSH_KEY` parameter on the `ironic configmap` on the `metal3-system` namespace. This can be easier as it can be done via `kubectl` edit but it will be overwritten when updating the chart. Then the Metal^3 pod needs to be restarted with `kubectl delete pod -n metal3-system -l app.kubernetes.io/component=ironic`

Note

Check the CAPI troubleshooting (https://cluster-api.sigs.k8s.io/user/troubleshooting) and Metal^3 troubleshooting (https://book.metal3.io/troubleshooting) guides.

76 Troubleshooting Other components

Other SUSE Telco Cloud components troubleshooting guides can be consulted on their official documentation:

- SUSE Linux Micro Troubleshooting (https://documentation.suse.com/smart/micro-clouds/html/SLE-Micro-5.5-admin/index.html#id-1.10)
- RKE2 Known Issues (https://docs.rke2.io/known_issues)
- K3s Known Issues (https://docs.k3s.io/known-issues)
- Rancher General Troubleshooting (https://ranchermanager.docs.rancher.com/troubleshooting/general-troubleshooting)
- SUSE Multi-Linux Manager Troubleshooting (https://documentation.suse.com/multi-linux-manager/5.1/en/docs/administration/troubleshooting/tshoot-intro.html)
- Elemental Support (https://elemental.docs.rancher.com/troubleshooting-support/)
- Rancher Turtles Troubleshooting (https://turtles.docs.rancher.com/turtles/stable/en/troubleshooting/troubleshooting.html)
- Longhorn Troubleshooting (https://longhorn.io/docs/1.12.1/troubleshoot/troubleshooting/)
- Neuvector Troubleshooting (https://open-docs.neuvector.com/next/troubleshooting/troubleshooting/)
- Fleet Troubleshooting (https://fleet.rancher.io/troubleshooting)

You can also see SUSE Knowledgebase (https://www.suse.com/support/kb/).

77 Collecting Diagnostics for Support

When contacting SUSE Support, providing comprehensive diagnostic information is crucial.

ESSENTIAL INFORMATION TO COLLECT

- **Detailed problem description**: What happened, when did it happen, what were you doing, what is the expected behavior, and what is the actual behavior?
- **Steps to reproduce**: Can you reliably reproduce the issue? If so, list the exact steps.
- **Component versions**: SUSE Telco Cloud version, components versions (RKE2/K3, EIB, Metal^3, Elemental,..).
- **Relevant logs**:
  - `journalctl` output (filtered by service if possible, or full boot logs).
  - Kubernetes pod logs (kubectl logs).
  - Metal^3/Elemental component logs.
  - EIB build logs and other logs
- **System information**:
  - `uname -a`
  - `df -h`
  - `ip a`
  - `/etc/os-release`
- **Configuration files**: Relevant configuration files for Elemental, Metal^3, EIB such as helm chart values, configmaps, etc.
- **Kubernetes information**: Nodes, Services, Deployments, etc.
- **Kubernetes objects affected**: BMH, MachineRegistration, etc.

HOW TO COLLECT

- **For logs**: Redirect command output to files (for example, `journalctl -u k3s > k3s_logs.txt`).

- **For Kubernetes resources**: Use `kubectl get <resource> -o yaml > <resource_name>.yaml` to get detailed YAML definitions.
- **For system information**: Collect output of the commands listed above.
- **For SL Micro**: Check the SUSE Linux Micro Troubleshooting Guide (https://documentation.suse.com/sle-micro/5.5/html/SLE-Micro-all/cha-adm-support-slemicro.html) documentation on how to gather system information for support with `supportconfig`.
- **For RKE2/Rancher**: Check the The Rancher v2.x Linux log collector script (https://www.suse.com/support/kb/doc/?id=000020191) article to run The Rancher v2.x Linux log collector script.
- **For Edge (Nessie)**: Nessie 1.1.0 is a powerful diagnostic tool designed to collect logs and configuration data from SUSE Telco Cloud environments. It gathers comprehensive information from both the host system and Kubernetes clusters, making it invaluable for troubleshooting and support.
  - Nessie has two "modes" a kubernetes mode and a system mode.
    - To collect logs from a SUSE Telco Cloud cluster, run (provided that you have access to the kubeconfig file locally):

```
podman run --rm --privileged \
  -v /etc/rancher/k3s/k3s.yaml:/etc/rancher/k3s/k3s.yaml:ro \
  -v /var/log/journal:/var/log/journal:ro \
  -v /run/systemd:/run/systemd:ro \
  -v /etc/machine-id:/etc/machine-id:ro \
  -v /tmp:/tmp \
  -e NESSIE_LOG_DIR="/tmp" \
  -e NESSIE_ZIP_DIR="/tmp" \
  registry.suse.com/edge/3.7/nessie:1.1.0
```

> **Note**
>
> Adjust the paths of the `k3s.yaml/rke2.yaml` file if needed. See Nessie (https://github.com/suse-edge/support-tools/blob/main/nessie/README.md) for more information. You should be able to run this container in non-privileged mode if you have proper permissions (typically `k3s.yaml` / `rke2-server.yaml` files are owned by root).

- To collect logs in the system mode from the actual operating system, run:

```
podman run --rm --privileged \
  -v /var/log/journal:/var/log/journal:ro \
  -v /run/systemd:/run/systemd:ro \
  -v /etc/machine-id:/etc/machine-id:ro \
  -v /tmp:/tmp \
  -e NESSIE_LOG_DIR="/tmp" \
  -e NESSIE_ZIP_DIR="/tmp" \
  -e NESSIE_VERBOSE="1" \
  -e NESSIE_SKIP_POD_LOGS="true" \
  -e NESSIE_SKIP_K8S_CONFIGS="true" \
  -e NESSIE_SKIP_METRICS="true" \
  registry.suse.com/edge/3.7/nessie:1.1.0
```

> **Note**
>
> Please make sure to check Nessie (https://github.com/suse-edge/support-tools/blob/main/nessie/README.md) for more details and information on how to run Nessie in your environment. Likewise, you should be able to run this container in non-privileged mode provided you have proper permissions.

Contact Support. Please check the article available at How-to effectively work with SUSE Technical Support (https://www.suse.com/support/kb/doc/?id=000019452) and the support handbook located at SUSE Technical Support Handbook (https://www.suse.com/support/handbook/) for more details on how to contact SUSE support.

XII Appendix

78 Release Notes 568

78 Release Notes

78.1 Abstract

SUSE Telco Cloud 3.7 is a tightly integrated and comprehensively validated end-to-end solution for addressing the unique challenges of the deployment of infrastructure and cloud-native applications at the edge. Its driving focus is to provide an opinionated, yet highly flexible, highly scalable, and secure platform that spans initial deployment image building, node provisioning and onboarding, application deployment, observability, and lifecycle management.

The solution is designed with the notion that there is no "one-size-fits-all" edge platform due to our customers' widely varying requirements and expectations. Edge deployments push us to solve, and continually evolve, some of the most challenging problems, including massive scalability, restricted network availability, physical space constraints, new security threats and attack vectors, variations in hardware architecture and system resources, the requirement to deploy and interface with legacy infrastructure and applications, and customer solutions that have extended lifespans.

SUSE Telco Cloud is built on best-of-breed open source software from the ground up, consistent with both our 30-year history in delivering secure, stable, and certified SUSE Linux platforms and our experience in providing highly scalable and feature-rich Kubernetes management with our Rancher portfolio. SUSE Telco Cloud builds on-top of these capabilities to deliver functionality that can address a wide number of market segments, including retail, medical, transportation, logistics, telecommunications, smart manufacturing, and Industrial IoT.

For more information on product support lifecycle updates for SUSE Telco Cloud, see Product Support Lifecycle (https://www.suse.com/lifecycle/#suse-edge-37).

Note

SUSE Telco Cloud is a derivative of SUSE Edge, with additional optimizations and components that enable the platform to address the requirements found in telecommunications use-cases.

78.2 About

These Release Notes are, unless explicitly specified and explained, identical across all architectures, and the most recent version, along with the release notes of all other SUSE products are always available online at https://www.suse.com/releasenotes.

Entries are only listed once, but they can be referenced in several places if they are important and belong to more than one section. Release notes usually only list changes that happened between two subsequent releases. Certain important entries from the release notes of previous product versions may be repeated. To make these entries easier to identify, they contain a note to that effect.

However, repeated entries are provided as a courtesy only. Therefore, if you are skipping one or more releases, check the release notes of the skipped releases also. If you are only reading the release notes of the current release, you could miss important changes that may affect system behavior. SUSE Telco Cloud versions are defined as x.y.z, where 'x' denotes the major version, 'y' denotes the minor, and 'z' denotes the patch version, also known as the "z-stream". SUSE Telco Cloud product lifecycles are defined based around a given minor release, e.g. "3.7", but ship with subsequent patch updates through its lifecycle, e.g. "3.7.1".

Note

SUSE Telco Cloud z-stream releases are tightly integrated and thoroughly tested as a versioned stack. Upgrade of any individual components to a different versions to those listed above is likely to result in system downtime. While it’s possible to run Edge clusters in untested configurations, it is not recommended, and it may take longer to provide resolution through the support channels.

78.3 Release 3.7.0

Availability Date: 23rd September 2026

Full Support End Date: 22nd March 2027

Maintenance Support End Date: 22nd March 2028

EOL: 23rd March 2028

Summary: SUSE Telco Cloud 3.7.0 is the first release in the SUSE Telco Cloud 3.7 release stream.

78.3.1 New Features

- Updated to Kubernetes 1.36.3 and Rancher Prime 2.15.1
- Updated to SUSE Security (NeuVector) 5.6.1 NeuVector Release Notes (https://open-docs.neu-vector.com/releasenotes/5x)

- Updated to SUSE Storage (Longhorn) 1.12.1 Upstream Longhorn Release Notes (https://longhorn.io/docs/1.12.1/)
- SUSE Storage V2 Data Engine is the recommended default for newly provisioned volumes on clusters that meet the V2 prerequisites. See *Chapter 66, SUSE Storage V2 Data Engine* for setup and migration guidance.
- Updated to Rancher Turtles (CAPI) 0.27.0 Rancher Turtles Documentation (https://turtles.docs.rancher.com/)
- Updated to MetalLB 0.16.1 Upstream Release Notes (https://metallb.universe.tf/release-notes/#version-0-16-1)
- Updated to KubeVirt 1.8.3 and CDI (Containerized Data Importer) 1.65.0, using SLES 16.0 container images
- Updated the KubeVirt Dashboard Extension to 1.4.2
- Updated to Elemental 1.9.2 Elemental Release Notes (https://elemental.docs.rancher.com/release-notes/)
- Updated to Cert-Manager 1.20.1 Upstream Release Notes (https://cert-manager.io/docs/release-notes/)
- Updated Metal3 to 0.16.0 with Bare Metal Operator 0.13.3 and Ironic 38.0.0

78.3.2 Bug & Security Fixes

- Kubernetes 1.36.3 contains bug fixes and security updates Kubernetes Changelog (https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/CHANGELOG-1.36.md)
- Rancher Prime 2.15.1 contains bug fixes Upstream Rancher Release Notes (https://github.com/rancher/rancher/releases/tag/v2.15.1)
- SUSE Storage (Longhorn) 1.12.1 contains bug fixes Upstream Longhorn Release Notes (https://github.com/longhorn/longhorn/releases/tag/v1.12.1)

- SUSE Security (NeuVector) 5.6.1 contains new features and bug fixes NeuVector Release Notes (https://open-docs.neuvector.com/releasenotes/5x)
- Metal3 chart 0.16.0 includes the following bug fixes:
  - Fix race condition in IPA image with NTP sync
  - Fix an issue with iDRAC 10 compatibility
  - Fix autologin in IPA image

78.3.3 Known Issues

Warning

If deploying new clusters, please follow *Chapter 65, Building Updated SUSE Linux Micro Images with Kiwi* to build fresh images first. This is suggested for management and downstream clusters to ensure the images contain the latest security and bug fixes.

Release-specific known issues will be added as release validation completes.

78.3.3.1 SUSE Storage V2 Data Engine limitations

SUSE Storage 1.12.1 cannot convert an existing V1 volume to V2 in place. Existing volumes remain on V1 after an upgrade and must be migrated to newly provisioned V2 volumes. Keep both data engines enabled until all V1 workloads have been migrated and validated.

V2 volumes must be detached before upgrading between SUSE Storage 1.12 patch releases because V2 engine live upgrade is not supported. V2 backing images are also not supported; KubeVirt deployments should use CDI for image import workflows. Additional feature differences and the customer migration procedure are documented in *Chapter 66, SUSE Storage V2 Data Engine*.

78.3.4 Component Versions

The following table describes the individual components that make up the 3.7.0 release, including the version, the Helm chart version (if applicable), and from where the released artifact can be pulled in the binary format. Please follow the associated documentation for usage and deployment examples.

Name	Version	Helm Chart Version	Artifact Location (URL/Image)
SUSE Linux Micro	6.2 (latest)	N/A	SUSE Linux Micro Download Page (https://www.suse.com/download/sle-micro/)
SUSE Linux Micro	6.2 (latest)	N/A	Checksums and signatures are available for download at SUSE Linux Micro Download Page (https://www.suse.com/download/sle-micro/) SL-Micro.x86_64-6.2-Base-SelfInstall-GM.install.iso SL-Micro.x86_64-6.2-Base-RT-SelfInstall-GM.install.iso SL-Micro.x86_64-6.2-Base-GM.raw.xz SL-Micro.x86_64-6.2-Base-RT-GM.raw.xz
SUSE Multi-Linux Manager	5.2	N/A	SUSE Multi-Linux Manager Download Page (https://)

			www.suse.com/download/suse-manager/)
K3s	1.36.3	N/A	Upstream K3s Release (https://github.com/k3s-io/k3s/releases/tag/v1.36.3%2Bk3s1)
RKE2	1.36.3	N/A	Upstream RKE2 Release (https://github.com/rancher/rke2/releases/tag/v1.36.3%2Brke2r1)
SUSE Rancher Prime	2.15.1	2.15.1	Rancher Prime Helm Repository (https://charts.rancher.com/server-charts/prime/index.yaml) Rancher 2.15.1 Container Images (https://prime.ribs.rancher.io/rancher/v2.15.1/rancher-images.txt)
SUSE Storage (Longhorn)	1.12.1	1.12.1	SUSE Storage Helm Repository (https://apps.rancher.io/applications/suse-storage) SUSE Storage Container Images (https://apps.rancher.io/applications/suse-storage/components)

SUSE Security (NeuVector)	5.6.1	110.0.1+up2.11.1	Rancher Charts Helm Repository (https://charts.rancher.io/index.yaml) registry.rancher.com/rancher/neuvector-controller:5.6.1 registry.rancher.com/rancher/neuvector-enforcer:5.6.1 registry.rancher.com/rancher/neuvector-compliance-config:1.0.16 registry.rancher.com/rancher/neuvector-updater:0.0.14
Rancher Turtles Providers (CAPI)	0.27.0	307.0.8+up0.27.0	registry.suse.com/edge/3.7/rancher-turtles-providers-chart:307.0.8+up0.27.0 registry.rancher.com/rancher/cluster-api-controller:v1.13.3 registry.rancher.com/rancher/turtles:v0.27.1 registry.suse.com/rancher/cluster-api-provider-rke2-bootstrap:v0.25.0

			registry.suse.com/rancher/cluster-api-provider-rke2-control-plane:v0.25.0
Metal[3]	0.16.0	307.0.31+up0.16.0	registry.suse.com/edge/3.7/metal3-chart:307.0.31+up0.16.0 registry.suse.com/edge/3.7/baremetal-operator:0.13.3.0 registry.suse.com/edge/3.7/ironic:38.0.0.0 registry.suse.com/edge/3.7/ironic-ipa-downloader:3.1.2 registry.suse.com/edge/3.7/ironic-python-agent:3.0.10
MetalLB	0.16.1	307.0.3+up0.16.1	registry.suse.com/edge/3.7/metallb-chart:307.0.3+up0.16.1 registry.suse.com/edge/3.7/metallb-controller:v0.16.1

			registry.suse.com/edge/3.7/metallb-speaker:v0.16.1 registry.suse.com/edge/3.7/frr:10.2.1 registry.suse.com/edge/3.7/frr-k8s:v0.0.25
Elemental	1.9.2	1.9.2	registry.suse.com/rancher/elemental-operator-chart:1.9.2 registry.suse.com/rancher/elemental-operator-crds-chart:1.9.2 registry.suse.com/rancher/elemental-operator:1.9.2
Elemental Dashboard Extension	3.0.1	3.0.1	Elemental Extension Helm Chart (https://github.com/rancher/ui-plugin-charts/tree/main/charts/elemental/3.0.1)
Edge Image Builder	1.3.4	N/A	registry.suse.com/edge/3.7/edge-image-builder:1.3.4

KubeVirt	1.8.3	307.0.3+up0.8.0	registry.suse.com/edge/3.7/kubevirt-chart:307.0.3+up0.8.0 registry.suse.com/suse/sles/16.0/virt-operator:1.8.3 registry.suse.com/suse/sles/16.0/virt-api:1.8.3 registry.suse.com/suse/sles/16.0/virt-controller:1.8.3 registry.suse.com/suse/sles/16.0/virt-handler:1.8.3 registry.suse.com/suse/sles/16.0/virt-launcher:1.8.3 registry.suse.com/suse/sles/16.0/virt-exportproxy:1.8.3 registry.suse.com/suse/sles/16.0/virt-exportserver:1.8.3
KubeVirt Dashboard Extension	1.4.2	307.0.5+up1.4.2	registry.suse.com/edge/3.7/kubevirt-dashboard-extension-chart:307.0.5+up1.4.2
Containerized Data Importer (CDI)	1.65.0	307.0.3+up0.8.0	registry.suse.com/edge/3.7/cdi-chart:307.0.3+up0.8.0

			registry.suse.com/suse/sles/16.0/cdi-operator:1.65.0 registry.suse.com/suse/sles/16.0/cdi-controller:1.65.0 registry.suse.com/suse/sles/16.0/cdi-apiserver:1.65.0 registry.suse.com/suse/sles/16.0/cdi-uploadproxy:1.65.0 registry.suse.com/suse/sles/16.0/cdi-cloner:1.65.0 registry.suse.com/suse/sles/16.0/cdi-importer:1.65.0 registry.suse.com/suse/sles/16.0/cdi-uploadserver:1.65.0
Endpoint Copier Operator	0.3.0	307.0.1+up0.3.0	registry.suse.com/edge/3.7/endpoint-copier-operator-chart:307.0.1+up0.3.0 registry.suse.com/edge/3.7/endpoint-copier-operator:0.3.0
SR-IOV Network Operator	1.6.0	307.1.0+up1.6.0	registry.suse.com/edge/3.7/sriov-network-operator-chart:307.1.0+up1.6.0

			registry.suse.com/edge/3.7/sriov-crd-chart:307.1.0+up1.6.0 registry.suse.com/edge/3.7/sriov-network-metrics-exporter:v1.2.0 registry.suse.com/edge/3.7/kube-rbac-proxy:0.22.1
System Upgrade Controller	0.20.1	110.0.0	Rancher Charts Helm Repository (https://charts.rancher.io/index.yaml) registry.rancher.com/rancher/system-upgrade-controller:v0.20.1
Upgrade Controller	0.1.3	307.0.4+up0.1.3	registry.suse.com/edge/3.7/upgrade-controller-chart:307.0.4+up0.1.3 registry.suse.com/edge/3.7/upgrade-controller:0.1.3 registry.rancher.com/rancher/kubectl:v1.35.2 registry.suse.com/edge/3.7/release-manifest:3.7.0

SUSE Private Registry	1.1.1	1.1.1	oci://registry.suse.com/private-registry/private-registry-helm[SUSE Private Registry Helm Repository] registry.suse.com/private-registry/harbor-core:1.1.1-1.19 registry.suse.com/private-registry/harbor-jobservice:1.1.1-1.19 registry.suse.com/private-registry/harbor-portal:1.1.1-1.20 registry.suse.com/private-registry/harbor-registry:1.1.1-1.19 registry.suse.com/private-registry/harbor-registryctl:1.1.1-1.19 registry.suse.com/private-registry/harbor-trivy-adapter:1.1.1-1.24
Kiwi Builder	10.2.29.1	N/A	registry.suse.com/edge/3.7/kiwi-builder:10.2.29.1
Cert-Manager	1.20.1	1.20.1	Jetstack Helm Repository (https://charts.jetstack.io)

			quay.io/jetstack/cert-manager-controller:v1.20.1 quay.io/jetstack/cert-manager-webhook:v1.20.1 quay.io/jetstack/cert-manager-cainjector:v1.20.1

78.4 Removed features

Unless otherwise stated, these apply to the 3.7.0 release and all subsequent z-stream versions.

- Akri was a Technology Preview offering in previous Edge releases and deprecated from 3.4.0 onwards. It is now completely removed from the offering.

78.5 Technology Previews

Unless otherwise stated, these apply to the 3.7.0 release and all subsequent z-stream versions.

- Single-stack IPv6 management cluster deployments are a Technology Preview offering and are not subject to the standard scope of support.

78.6 Component Verification

The components mentioned above may be verified using the Software Bill Of Materials (SBOM) data - for example, using `cosign` as outlined below:

Download the SUSE Telco Cloud Container public key from the SUSE Signing Keys source (https://www.suse.com/support/security/keys/):

```
> cat key.pem
-----BEGIN PUBLIC KEY-----
MIICIjANBgkqhkiG9w0BAQEFAAOCAg8AMIICCgKCAgEA7N0S2d8LFKW4WU43bq7Z
IZT537xlKe17OQEpYjNrdtqnSwA0/jLtK83m7bTzfYRK4wty/so0g3BGo+x6yDFt
SVXTPBqnYvabU/j7UKaybJtX3jc4SjaezeBqdi96h6yEslvg4VTZDpy6TFP5ZHxZ
```

```
A0fX6m5kU2/RYhGXItoeUmL5hZ+APYgYG4/455NBaZT2yOywJ6+1zRgpR0cRAekI
OZXl51k0ebsGV6ui/NGECO6MB5e3arAhszf8eHDE02FeNJw5cimXkgDh/1Lg3KpO
dvUNm0EPWvnkNYeMCKR+687QG0bXqSVyCbY6+HG/HLkeBWkv6Hn41oeTSLrjYVGa
T3zxPVQM726sami6pgZ5vULyOleQuKBZrlFhFLbFyXqv1/DokUqEppm2Y3xZQv77
fMNogapp0qYz+nE3wSK4UHPd9z+2bq5WEkQSalYxadyuqOzxqZgSoCNoX5iIuWte
Zf1RmHjiEndg/2UgxKUysVnyCpiWoGbalM4dnWE24102050Gj6M4B5fe73hbaRlf
NBqP+97uznnRlSl8FizhXzdzJiVPcRav1tDdRUyDE2XkNRXmGfD3aCmILhB27SOA
Lppkouw849PWBt9kDMvzelUYLpINYpHRi2+/eyhHNlufeyJ7e7d6N9VcvjR/6qWG
64iSkcF2DTW61CN5TrCe0k0CAwEAAQ==
-----END PUBLIC KEY-----
```


Verify the container image hash, for example using `crane`:

```
> crane digest registry.suse.com/edge/3.7/baremetal-operator:0.13.3.0 --platform linux/
amd64
sha256:247e687143b2d21d9cebcef0bbac8ca3d14b4d1b6e0ba04812dd61d44b32b776
```

Note

For multi-arch images it is also necessary to specify a platform when obtaining the digest, e.g `--platform linux/amd64` or `--platform linux/arm64`. Failure to do this will result in an error in the following step (`Error: no matching attestations`).

Verify with `cosign`:

```
> cosign verify-attestation --type spdxjson --key key.pem registry.suse.com/edge/3.7/
baremetal-
operator@sha256:247e687143b2d21d9cebcef0bbac8ca3d14b4d1b6e0ba04812dd61d44b32b776 > /dev/
null
#
Verification for registry.suse.com/edge/3.7/baremetal-operator@sha256:example-digest-
placeholder --
The following checks were performed on each of these signatures:
  - The cosign claims were validated
  - Existence of the claims in the transparency log was verified offline
  - The signatures were verified against the specified public key
```

Extract SBOM data as described at the SUSE SBOM documentation (https://www.suse.com/support/security/sbom/):

```
> cosign verify-attestation --type spdxjson --key key.pem registry.suse.com/edge/3.7/
baremetal-
operator@sha256:247e687143b2d21d9cebcef0bbac8ca3d14b4d1b6e0ba04812dd61d44b32b776 | jq
 '.payload | @base64d | fromjson | .predicate'
```

78.7 Upgrade Steps

Refer to the *Part VIII, "Day 2 Operations"* for details around how to upgrade to a new release.

78.8 Product Support Lifecycle

SUSE Telco Cloud is backed by award-winning support from SUSE, an established technology leader with a proven history of delivering enterprise-quality support services. For more information, see https://www.suse.com/lifecycle and the Support Policy page at https://www.suse.com/support/policy.html. If you have any questions about raising a support case, how SUSE classifies severity levels, or the scope of support, please see the Technical Support Handbook at https://www.suse.com/support/handbook/.

SUSE Telco Cloud "3.7" is supported for 18-months of production support, with an initial 6-months of "full support", followed by 12-months of "maintenance support". After these support phases the product reaches "end of life" (EOL) and is no longer supported. More info about the lifecycle phases can be found in the table below:

Full Support (6 months)	Urgent and selected high-priority bug fixes will be released during the full support window, and all other patches (non-urgent, enhancements, new capabilities) will be released via the regular release schedule.
Maintenance Support (12 months)	During this period, only critical fixes will be released via patches. Other bug fixes may be released at SUSE's discretion but should not be expected.
End of Life (EOL)	Once a product release reaches its End of Life date, the customer may continue to use the product within the terms of product licensing agreement. Support Plans from SUSE do not apply to product releases past their EOL date.

Unless explicitly stated, all components listed are considered Generally Available (GA), and are covered by SUSE's standard scope of support. Some components may be listed as "Technology Preview", where SUSE is providing customers with access to early pre-GA features and functionality for evaluation, but

are not subject to the standard support policies and are not recommended for production use-cases. SUSE very much welcomes feedback and suggestions on the improvements that can be made to Technology Preview components, but SUSE reserves the right to deprecate a Technology Preview feature before it becomes Generally Available if it doesn't meet the needs of our customers or doesn't reach a state of maturity that we require.

Please note that SUSE must occasionally deprecate features or change API specifications. Reasons for feature deprecation or API change could include a feature being updated or replaced by a new implementation, a new feature set, upstream technology is no longer available, or the upstream community has introduced incompatible changes. It is not intended that this will ever happen within a given minor release (x.z), and so all z-stream releases will maintain API compatibility and feature functionality. SUSE will endeavor to provide deprecation warnings with plenty of notice within the release notes, along with workarounds, suggestions, and mitigations to minimize service disruption.

The SUSE Telco Cloud team also welcomes community feedback, where issues can be raised within the respective code repository within https://www.github.com/suse-edge.

78.9 Obtaining source code

This SUSE product includes materials licensed to SUSE under the GNU General Public License (GPL) and various other open source licenses. The GPL requires SUSE to provide the source code that corresponds to the GPL-licensed material, and SUSE conforms to all other open-source license requirements. As such, SUSE makes all source code available, and can generally be found in the SUSE Telco Cloud GitHub repository (https://www.github.com/suse-edge), the SUSE Rancher GitHub repository (https://www.github.com/rancher) for dependent components, and specifically for SUSE Linux Micro, the source code is available for download at https://www.suse.com/download/sle-micro (https://www.suse.com/download/sle-micro/) on "Medium 2".

78.10 Legal notices

SUSE makes no representations or warranties with regard to the contents or use of this documentation, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, SUSE reserves the right to revise this publication and to make changes to its content, at any time, without the obligation to notify any person or entity of such revisions or changes.

Further, SUSE makes no representations or warranties with regard to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, SUSE reserves the right to make changes to any and all parts of SUSE software, at any time, without any obligation to notify any person or entity of such changes.

Any products or technical information provided under this Agreement may be subject to U.S. export controls and the trade laws of other countries. You agree to comply with all export control regulations and to obtain any required licenses or classifications to export, re-export, or import deliverables. You agree not to export or re-export to entities on the current U.S. export exclusion lists or to any embargoed or terrorist countries as specified in U.S. export laws. You agree to not use deliverables for prohibited nuclear, missile, or chemical/biological weaponry end uses. Refer to https://www.suse.com/company/legal/ for more information on exporting SUSE software. SUSE assumes no responsibility for your failure to obtain any necessary export approvals.

Copyright © 2024 SUSE LLC.

This release notes document is licensed under a Creative Commons Attribution-NoDerivatives 4.0 International License (CC-BY-ND-4.0). You should have received a copy of the license along with this document. If not, see https://creativecommons.org/licenses/by-nd/4.0/.

SUSE has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at https://www.suse.com/company/legal/ and one or more additional patents or pending patent applications in the U.S. and other countries.

For SUSE trademarks, see the SUSE Trademark and Service Mark list (https://www.suse.com/company/legal/). All third-party trademarks are the property of their respective owners. For SUSE brand information and usage requirements, please see the guidelines published at https://brand.suse.com/.