

NVIDIA Base Command Manager 11

Containerization Manual

Revision: 0584cb1c5

Date: Mon Aug 10 2026

©2026 NVIDIA Corporation & affiliates. All Rights Reserved. This manual or parts thereof may not be reproduced in any form unless permitted by contract or by written permission of NVIDIA Corporation.

Trademarks

Linux is a registered trademark of Linus Torvalds. PathScale is a registered trademark of Cray, Inc. Red Hat and all Red Hat-based trademarks are trademarks or registered trademarks of Red Hat, Inc. SUSE is a registered trademark of SUSE LLC. NVIDIA, CUDA, GPUDirect, HPC SDK, NVIDIA DGX, NVIDIA Nsight, and NVLink are registered trademarks of NVIDIA Corporation. FLEXlm is a registered trademark of Flexera Software, Inc. PBS Professional, and Green Provisioning are trademarks of Altair Engineering, Inc. All other trademarks are the property of their respective owners.

Rights and Restrictions

All statements, specifications, recommendations, and technical information contained herein are current or planned as of the date of publication of this document. They are reliable as of the time of this writing and are presented without warranty of any kind, expressed or implied. NVIDIA Corporation shall not be liable for technical or editorial errors or omissions which may occur in this document. NVIDIA Corporation shall not be liable for any damages resulting from the use of this document.

Limitation of Liability and Damages Pertaining to NVIDIA Corporation

The NVIDIA Base Command Manager product principally consists of free software that is licensed by the Linux authors free of charge. NVIDIA Corporation shall have no liability nor will NVIDIA Corporation provide any warranty for the NVIDIA Base Command Manager to the extent that is permitted by law. Unless confirmed in writing, the Linux authors and/or third parties provide the program as is without any warranty, either expressed or implied, including, but not limited to, marketability or suitability for a specific purpose. The user of the NVIDIA Base Command Manager product shall accept the full risk for the quality or performance of the product. Should the product malfunction, the costs for repair, service, or correction will be borne by the user of the NVIDIA Base Command Manager product. No copyright owner or third party who has modified or distributed the program as permitted in this license shall be held liable for damages, including general or specific damages, damages caused by side effects or consequential damages, resulting from the use of the program or the un-usability of the program (including, but not limited to, loss of data, incorrect processing of data, losses that must be borne by you or others, or the inability of the program to work together with any other program), even if a copyright owner or third party had been advised about the possibility of such damages unless such copyright owner or third party has signed a writing to the contrary.

Table of Contents

Table of Contents	3
0.1 About This Manual	9
0.2 About The Manuals In General	9
0.3 Getting Administrator-Level Support	9
0.4 Getting Professional Services	10
1 Introduction To Containerization On NVIDIA Base Command Manager	11
2 Docker Engine	13
2.1 Docker Setup	13
2.2 Integration With Workload Managers	15
2.3 DockerHost Role	15
2.4 Iptables	18
2.5 Storage Backends	18
2.6 Docker Monitoring	19
2.7 Docker Setup For NVIDIA	20
3 Container Registries	23
3.1 Container Registries: Introduction	23
3.1.1 Docker Hub, A Remote Registry	23
3.1.2 Local Image Registry Options: Classic Docker Registry Vs Harbor	23
3.2 Container Registries: Setup And Configuration	24
3.2.1 Docker Registry As A Pull Through Cache For Docker Hub	24
3.2.2 Harbor UI	25
3.2.3 Docker Registry And Harbor Configuration	25
4 Kubernetes	27
4.1 Reference Architecture	28
4.1.1 Hardware Requirements	28
4.2 Kubernetes Setup	28
4.2.1 Kubernetes Networking	29
4.2.2 Kubernetes Core Add-ons	30
4.2.3 Kubernetes Optional Components	31
4.2.4 Kubernetes Deprecated And Retired Components	32
4.2.5 Helm Kubernetes Package Manager	32
4.2.6 Note On Run:ai Operator Dependencies	33
4.2.7 Kubernetes Setup From The Command Line	34
4.2.8 Kubernetes Setup From A TUI Session	38
4.2.9 Testing Kubernetes	40
4.3 Using GPUs With Kubernetes: NVIDIA GPUs	41
4.3.1 Prerequisites	41
4.3.2 New Kubernetes Installation	42

4.3.3	Existing Kubernetes Installation	43
4.4	Using GPUs With Kubernetes: AMD GPUs	43
4.4.1	Prerequisites	43
4.4.2	Managing The YAML File Through CMDaemon	43
4.4.3	Including Head Nodes as part of the DaemonSet:	44
4.4.4	Running The DaemonSet Only On Specific Nodes	45
4.4.5	Running An Example Workload	46
4.5	Kubernetes Configuration Overlays	47
4.6	Removing A Kubernetes Cluster	48
4.7	Kubernetes Cluster Configuration Options	49
4.8	EtcdCluster	51
4.9	Kubernetes Roles	52
4.9.1	EtcdHost Role	53
4.9.2	The KubernetesAPIServerProxy Role	54
4.9.3	The KubernetesIngressServerProxy Role	55
4.9.4	The Kubelet Role	55
4.9.5	Containerd Role	56
4.10	Security Model	57
4.10.1	Kyverno	57
4.10.2	PodSecurityPolicy	58
4.11	Addition Of New Kubernetes Users	58
4.11.1	Adding Users Non-Interactively With cm-kubernetes-setup	59
4.12	Getting Information And Modifying Existing Kubernetes Users	61
4.13	List Of Resources Defined For Users	61
4.14	Kyverno	63
4.14.1	Kyverno Installation	63
4.14.2	Kyverno Policies	64
4.15	Kubernetes Permission Manager	65
4.16	Providing Access To External Users	68
4.16.1	External Access To The Kubernetes Control Plane (kubect1, helm)	70
4.16.2	External Access To Kubernetes-hosted Applications And Services	70
4.17	Kubernetes Gateway	70
4.17.1	Gateway API	71
4.17.2	Kgateway	71
4.17.3	The bcm-gateway Gateway	71
4.17.4	MetalLB	72
4.17.5	Creating Additional Gateways	74
4.17.6	Gateway Proxy	77
4.17.7	Configuring The Gateway TLS Certificate	80
4.17.8	Migrating An Existing Kubernetes Cluster From Ingress To Gateway	81
4.18	Networking Model	83
4.19	Kubernetes Monitoring	83
4.20	Local Path Storage Class	83
4.21	Integration With Harbor	84
4.22	Kubernetes Upgrades	84
4.22.1	Upgrade Prerequisites	85

4.22.2	Example RHEL9 Cluster	86
4.22.3	Before Starting The Upgrade	86
4.22.4	Updating The First Control Plane Node	86
4.22.5	Updating Subsequent Control Plane Nodes	89
4.22.6	Updating The Worker Nodes	90
4.22.7	Updating The Status In BCM	91
4.22.8	Notes For Upgrading To Kubernetes v1.31.x	92
4.22.9	Notes For Ubuntu	92
4.22.10	Notes For SLES	92
4.22.11	Other Approaches	93
5	Kubernetes Apps	95
6	Kubernetes Operators	97
6.1	Versions Of Operators Available	97
6.2	Helm Charts For The BCM Operators	99
6.3	The Jupyter Kernel Operator	100
6.3.1	Installing The Jupyter Kernel Operator	100
6.3.2	Architecture Overview	100
6.3.3	Running Jupyter Kernel Using The Operator	102
6.3.4	Jupyter Kernel Operator Tunables	103
6.3.5	Sidecar Arguments And Environment Variables	104
6.3.6	Running Spark-based Kernels In Jupyter Kernel Operator	105
6.3.7	Example: Creating An R Kernel From The Kernel Template	105
6.3.8	Example: Letting Kubernetes Access Private Registries From The Kernel Template	110
6.3.9	Example: Adding The PVC Parameter To The Kernel Template	112
6.4	The NVIDIA GPU Operator	114
6.4.1	Installing The NVIDIA GPU Operator Using The Wizard	115
6.4.2	Installing The NVIDIA GPU Operator On An Existing Kubernetes Cluster	117
6.4.3	Removing The NVIDIA GPU Operator	117
6.4.4	Validating The NVIDIA GPU Operator	118
6.4.5	Validating The NVIDIA GPU Operator In Detail	118
6.4.6	Running A GPU Workload	122
6.5	The NVIDIA Network Operator	123
6.5.1	Installing The NVIDIA Network Operator	123
6.6	The NVIDIA NetQ Operator	125
6.6.1	NVIDIA NetQ Operator Installation	125
6.6.2	Accessing The NVIDIA NetQ Operator UI	128
6.7	The Prometheus Operator Stack	128
6.7.1	Exporting And Reusing Grafana Dashboards	128
6.8	The Run:ai Operator	129
6.8.1	Run:ai Operator Installation Prerequisites	129
6.8.2	Installing Run:ai Operator	131
6.8.3	Completing The Run:ai Installation	132
6.8.4	Post-installation	132
6.9	Kubernetes Spark Operator	133
6.9.1	Installing The Kubernetes Spark Operator	133

6.9.2	Example Spark Operator Run: Calculating Pi	135
6.10	The Zalando Postgres Operator	136
6.10.1	Installing The Zalando Postgres Operator	136
6.11	The NVIDIA NIM Operator	137
6.11.1	Installing The NVIDIA NIM Operator	137
7	Kubernetes On Edge	139
7.1	Flags For Edge Installation	139
7.1.1	Speeding Up Kubernetes Installation To Edge Nodes With The --skip-* Flags: Use Cases	140
8	Kubernetes Cluster API	141
8.1	Kubernetes Cluster API Components	141
8.1.1	Kubernetes Management Cluster	141
8.1.2	Kubernetes CAPI Cluster	142
8.1.3	BCM CAPI Infrastructure Provider	142
8.2	The Kubernetes CAPI Wizard	143
8.2.1	The Install CAPI Option	143
8.2.2	The Assign CAPI Role Option	145
8.3	Deploying A Kubernetes Cluster Through CAPI	147
8.3.1	Machine Provisioning	148
8.3.2	Accessing The Cluster	150
8.3.3	Scaling Control Planes Or Workers	152
8.3.4	Upgrading Control Planes Or Workers	152
8.4	BCM Host Agent Registration	154
8.5	Install Process BCM CAPI	156
8.5.1	Registration Process Of The Node With BCM	158
8.5.2	Creating A Kubernetes Cluster Via CAPI	160
8.6	Configuring CAPI Versions In Software Images	161
8.7	Removing Kubernetes CAPI clusters	161
8.8	Kubernetes CAPI Templates	163
A	Create Self-Signed Server Certificate Pair For Testing Purposes	165
B	Kubernetes Installation On An Air-gapped BCM Cluster	169
B.1	Initial Host Preparations For BCM Kubernetes Air-gapped Installation	169
B.2	Head Node Preparation Steps	173
B.2.1	Preparing Docker Registry And Docker Installation	173
B.2.2	Unpacking The Files	174
B.2.3	Configuring The Container Registry	175
B.2.4	Configuring Docker For An Air-gapped Cluster	175
B.2.5	Setting Up The Container Images And Helm Charts For An Air-gapped Cluster During Installation	176
B.2.6	Extracting The CRD Files For An Air-gapped Cluster During Installation	176
B.2.7	Deploying The Air-gapped Kubernetes Cluster	176
B.2.8	Decoupled Run:ai Deployment	176
B.2.9	Post-installation Automatic Provisioning	178

C	Kubernetes Upgrade On An Air-gapped BCM Cluster	179
C.1	Air-gapped Kubernetes Upgrade Prerequisites	179
C.2	Initial Host Preparations For BCM Kubernetes Air-gapped Upgrade	180
C.3	Head Node Preparation Steps	184
C.3.1	Unpacking The Files	185
C.3.2	Setting Up The Container Images And Helm Charts For An Air-gapped Cluster During Upgrade	185
C.3.3	Extracting The CRD Files For An Air-gapped Cluster During Upgrade	185
C.4	Upgrading The Kubernetes Cluster	185
C.4.1	Updating The First Control Plane Node	185

Preface

Welcome to the *Containerization Manual* for NVIDIA Base Command Manager 11.

0.1 About This Manual

This manual is aimed at helping cluster administrators install, understand, configure, and manage the containerization integration capabilities of NVIDIA Base Command Manager. The administrator is expected to be reasonably familiar with the *Administrator Manual*.

0.2 About The Manuals In General

Regularly updated versions of the NVIDIA Base Command Manager 11 manuals are available on updated clusters by default at `/cm/shared/docs/cm`. The latest updates are always online at <https://docs.nvidia.com/base-command-manager>.

- The *Installation Manual* describes installation procedures for the basic cluster.
- The *Administrator Manual* describes the general management of the cluster.
- The *User Manual* describes the user environment and how to submit jobs for the end user.
- The *Cloudbursting Manual* describes how to deploy the cloud capabilities of the cluster.
- The *Developer Manual* has useful information for developers who would like to program with BCM.
- The *Edge Manual* explains how BCM can be used with edge sites.
- The *NVIDIA Mission Control Manual* describes NVIDIA Mission Control capabilities and integration with BCM.

If the manuals are downloaded and kept in one local directory, then in most pdf viewers, clicking on a cross-reference in one manual that refers to a section in another manual opens and displays that section in the second manual. Navigating back and forth between documents is usually possible with keystrokes or mouse clicks.

For example: <Alt>-<Backarrow> in Acrobat Reader, or clicking on the bottom leftmost navigation button of xpdf, both navigate back to the previous document.

The manuals constantly evolve to keep up with the development of the BCM environment and the addition of new hardware and/or applications. The manuals also regularly incorporate feedback from administrators and users, who can submit comments, suggestions or corrections via the website

<https://enterprise-support.nvidia.com/s/create-case>

Section 14.2 of the *Administrator Manual* has more details on submitting an issue.

0.3 Getting Administrator-Level Support

If the reseller from whom BCM was bought offers direct support, then the reseller should be contacted.

Otherwise the primary means of support is via the website <https://enterprise-support.nvidia.com/s/create-case>. This allows the administrator to submit a support request via a web form, and opens up a trouble ticket. It is a good idea to try to use a clear subject header, since that is used as part of a reference tag as the ticket progresses. Also helpful is a good description of the issue. The followup communication for this ticket goes via standard e-mail. Section 14.2 of the *Administrator Manual* has more details on working with support.

0.4 Getting Professional Services

The BCM support team normally differentiates between

- regular support (customer has a question or problem that requires an answer or resolution), and
- professional services (customer asks for the team to do something or asks the team to provide some service).

Professional services can be provided via the NVIDIA Enterprise Services page at:

<https://www.nvidia.com/en-us/support/enterprise/services/>

1

Introduction To Containerization On NVIDIA Base Command Manager

Containerization is a technology that allows processes to be isolated by combining *cgroups*, *Linux namespaces*, and *(container) images*.

- Cgroups are introduced in section 7.10 on workload management of the *Administrator Manual*
- Linux namespaces represent independent spaces for different operating system facilities: process IDs, network interfaces, mount points, inter-process communication resources and others. Cgroups and namespaces allow processes to be isolated from each other by separating the available resources as much as possible.
- A container image is a component of a container, and is a file that contains one or several layers. The layers cannot be altered as far the container is concerned, and a snapshot of the image can be used for other containers. A union file system is used to combine these layers into a single image. Union file systems allow files and directories of separate file systems to be transparently overlaid, forming a single coherent file system.

Cgroups, namespaces and image are the basis of a container. When the container is created, then a new process can be started within the container. Containerized processes running on a single machine all share the same operating system kernel, so they start immediately, without the delay of requiring a kernel to first boot up. No process is allowed to change the layers of the image. All changes are applied on a temporary layer created on top of the image, and these changes are destroyed when the container is removed.

There are several ways to manage the containers, but the most powerful approaches use Docker, also known as Docker Engine, and Kubernetes.

Docker manages containers on individual hosts, while Kubernetes manages containers across a cluster. BCM integrates both of these solutions, so that setup, configuration and monitoring of containers becomes an easily-managed part of BCM.

Chapter 2 describes how Docker integration with BCM works.

Chapter 3 covers how container registries are integrated.

Chapter 4 covers Kubernetes integration.

Chapter 5 covers Kubernetes application configuration and groups of Kubernetes applications.

Chapter 6 covers Kubernetes operators, which are a way to manage Kubernetes cluster applications.

Chapter 7 covers Kubernetes deployment on edge sites.

Chapter 8 describes the installation and usage of the NVIDIA Base Command Manager CAPI extension called BCM Kubernetes CAPI Infrastructure Provider. Kubernetes Cluster API (CAPI), is an API for managing Kubernetes clusters.

2

Docker Engine

Docker integration with NVIDIA Base Command Manager 11 for Docker version 28.0.4 is available at the time of writing of this paragraph (October 2024) on the x86_64 architecture for all the BCM-supported Linux distributions. For a more up-to-date status, the feature matrix at <https://service.bcm.nvidia.com/feature-matrix/> can be checked.

Docker Engine (or just Docker) is a tool for container management. Docker allows containers and their images to be created, controlled, and monitored on a host using Docker command line tools or the Docker API.

Swarm mode, which allows containers to spawn on several hosts, is not formally supported by NVIDIA Base Command Manager 11. This is because NVIDIA Base Command Manager 11 provides Kubernetes for this purpose instead.

Docker provides a utility called `docker`, and two daemons: one called `containerd` (the default provided by BCM), and the other called `dockerd`. Additional functionality includes pulling the container image from a specific image registry (Chapter 3), configuring the container network, setting `systemd` limits, and attaching volumes.

2.1 Docker Setup

BCM provides the `cm-docker` package. The package includes the following components:

- Docker itself, that provides an API and delegates the container management to Containerd;
- Containerd runtime, that manages container registry images and container registry containers (using `runC`);
- `runC`, a CLI tool for spawning and running containers according to the container registry specification runtime;
- `docker-py`, a Python library for the Docker API.

Typically, however, the administrator is expected to simply run the `cm-docker-setup` utility, which is provided by BCM's `cm-setup` package. Running `cm-docker-setup` takes care of the installation of the `cm-docker` package and also takes care of Docker setup. If run without options, then the utility starts up a TUI dialog (figure 2.1).

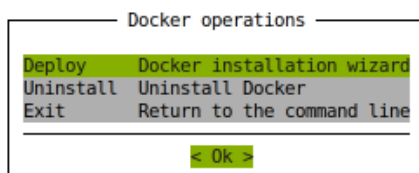


Figure 2.1: `cm-docker-setup` TUI startup

The `cm-docker-setup` utility asks several questions, such as which Docker registries are to be used, what nodes Docker is to be installed on, whether the NVIDIA container runtime should be installed, and so on. If `cm-docker-setup` is used with the `-c` option, and given a YAML configuration file `<YAMLfile>`, then a runtime configuration is loaded from that file. The YAML file is typically generated and saved from an earlier setup run.

The setup run displays a summary as its final TUI screen. In the final summary screen, as with other scripts that come with the `cm-setup` package, the configuration setting can be one of the following kinds:

- **minimal config:** For this minimal kind of configuration setting, the configuration consists only of the settings that differ from the default application configuration. The remaining default application configuration settings are not included in the configuration.
- **config:** For this non-minimal kind of configuration setting, the configuration consists of the minimal configuration settings (the ones changed by BCM), as well as the remaining default configuration settings (the ones provided with the application). That is to say: the remaining default configuration application settings are included in the configuration. The default application configuration settings may change as the upstream packages change.

When the questions in the TUI dialog have been answered and the deployment is carried out, the utility:

- installs the `cm-docker` package, if it has not been installed yet
- then assigns the `DockerHost` role to the node categories or head nodes that were specified
- adds health checks to the BCM monitoring configuration
- performs the initial configuration of Docker.

The regular nodes on which Docker is to run, are restarted by the utility, if needed. The restart operation provisions the updated images from the image directory onto the nodes.

The `cm-docker` package also includes a modules environment file, which must be loaded in order to use the `docker` command. The modules environment and modules are introduced in section 2.2 of the *Administrator Manual*.

By default only the administrator can run the `docker` commands after setup (some output ellipsized):

Example

```
[root@basecm11 ~]# ssh node001
[root@node001 ~]# module load docker
[root@node001 ~]# docker info
...
Containers: 0
Images: 0
...
Docker Root Dir: /var/lib/docker
Debug Mode: false
Experimental: false
Insecure Registries:
  127.0.0.0/8
Registry Mirrors:
  https://harbor-proxy.brightcomputing.com/
Live Restore Enabled: false
[root@node001 ~]#
```

and the `hello-world` image can be run as usual with:

Example

```
[root@node001 ~]# docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:cc15c5b292d8525effc0f89cb299f1804f3a725c8d05e158653a563f15e4f685
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

Or, for example, importing and running Apache containers with Docker may result in the following output:

Example

```
[root@node001 ~]# module load docker
[root@node001 ~]# docker run httpd & docker run httpd &
... runs a couple of Apache containers...
[root@node001 ~]# docker container ls
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
acdbe2f3667b	httpd	"httpd-foreground"	13 seconds ago	Up 11 seconds	80/tcp	quizzical_bhabha
64787a8524dd	httpd	"httpd-foreground"	13 seconds ago	Up 11 seconds	80/tcp	funny_hypatia

```
...
[root@node001 ~]#
```

Using Docker directly means being root on the host. It is rarely sensible to carry out regular user actions as the root user at all times.

So, to make Docker available to regular users, Kubernetes provides a user management layer and restrictions.

After Docker has been installed, Kubernetes can be set up to allow regular user access to the Docker containers as covered in Chapter 4. It is a best practice for regular users to use Kubernetes instead of Docker commands directly.

2.2 Integration With Workload Managers

BCM does not provide integration of Docker with workload managers. The administrator can however tune the workload managers in some cases to enable Docker support.

- LSF – An open beta version of LSF with Docker support is available from the IBM web site. This LSF version allows jobs to run in Docker containers, and monitors the container resources per job.
- PBS Professional – Altair provides a hook script that allows jobs to start in Docker containers. Altair should be contacted to obtain the script and instructions.

2.3 DockerHost Role

When `cm-docker-setup` is executed, the DockerHost role is assigned to nodes or categories. The DockerHost role is responsible for Docker service management and configuration.

From `cmsh`, the configuration parameters can be managed from the `Docker::Host` role:

Example

```

[root@basecm11 ~]# cmsh
[basecm11]% category use default
[basecm11->category[default]]% roles
[basecm11->category[default]->roles]% assign docker::host
[basecm11->category*[default*]->roles*[Docker::Host*]]% show
Parameter                                Value
-----
Name                                     Docker::Host
Revision
Type                                     DockerHostRole
Add services                             yes
Spool                                    /var/lib/docker
Tmp dir                                  $spool/tmp
Enable SELinux                           yes
Default Ulimits
Debug                                    no
Log Level                                info
Bridge IP
Bridge
MTU                                       0
API Sockets                             unix:///var/run/docker.sock
Iptables                                yes
User Namespace Remap
Insecure Registries
Enable TLS                               no
Verify TLS                              no
TLS CA
TLS Certificate
TLS Key
Certificates Path                        /etc/docker
Storage Backends                         <0 in submode>
Containerd Socket
Runtime                                  runc
Options
[basecm11->category*[default*]->roles*[Docker::Host*]]%

```

The Docker host parameters that CMDaemon can configure in the DockerHost role, along with a description, are shown in table 2.1:

Parameter	Description
Add services*	Add services to nodes belonging to this node. Care must be taken if setting this to no. (default: yes)
Spool	Root of the Docker runtime (default: /var/lib/docker)
Tmp dir	Location for temporary files. Default: \$<spool>/tmp, where \$<spool> is replaced by the path to the Docker runtime root directory

...continues

...continued

Parameter	Description
Enable SELinux*	Enable selinux support in Docker daemon (default: yes)
Default Ulimits	Set the default ulimit options for all containers
Debug*	Enable debug mode (default: no)
Log Level	Set the daemon logging level. In order of increasing verbosity: fatal, error, warn, info, debug. (default: info)
Bridge IP	Network bridge IP (not defined by default)
Bridge	Attach containers to a network bridge (not defined by default)
MTU	Set the containers network MTU, in bytes (default: 0, which does not set the MTU at all)
API Sockets	Daemon socket(s) to connect to (default: unix:///var/run/docker.sock)
Iptables*	Enable iptables rules (default: yes)
User Namespace Remap	User/Group setting for user namespaces (not defined by default). It can be set to any of <UID>, <UID:GID>, <username>, <username:groupname>. If it is used, then user_namespace.enable=1 must be set in the kernel options for the relevant nodes, and those nodes must be rebooted to pick up the new option.
Insecure Registries	If registry access uses HTTPS but does not have proper certificates distributed, then the administrator can make Docker accept this situation by adding the registry to this list (empty by default)
Enable TLS*	Use TLS (default: no)
Verify TLS*	Use TLS and verify the remote (default: no)
TLS CA	Trust only certificates that are signed by this CA (not defined by default)

...continues

...continued

Parameter	Description
TLS Certificate	Path to TLS certificate file (not defined by default)
TLS Key	Path to TLS key file (not defined by default)
Certificates Path	Path to Docker certificates (default: <code>/etc/docker</code>)
Storage Backends	Docker storage back ends. Storage types can be created and managed, in a submode under this mode. For BCM the only available type is <code>overlay2</code> . Each of these storage types has options that can be set from within the submode.
Containerd Socket	Path to the containerd socket (default: not used)
Runtime	Docker runtime
Options	Additional parameters for docker daemon

* Boolean (takes yes or no as a value)

Table 2.1: `Docker::Host` role options

2.4 Iptables

By default iptables rules have been added to nodes that function as a Docker host, to let network traffic go from the containers to outside the pods network. If this conflicts with other software that uses iptables, then this option can be disabled. For example, if the `docker::host` role has already been assigned to the nodes using the default category, then the iptables rules that are set can be disabled by setting the iptables parameter in the `Docker::Host` role to no:

Example

```
[root@basecm11 ~]# cmsh
[basecm11]% category use default
[basecm11->category[default]]% roles
[basecm11->category[default]->roles]% use docker::host
[basecm11->category[default]->roles[Docker::Host]]% set iptables no
[basecm11->category*[default*]->roles*[Docker::Host*]]% commit
```

2.5 Storage Backends

A core part of the Docker model is the efficient use of containers based on layered images. To implement this, Docker provides different storage back ends, also called storage drivers. These storage back ends rely heavily on various filesystem features in the kernel or volume manager. Some storage back ends perform better than others, depending on the circumstances.

For BCM 11.0 the storage backend configured by `cm-docker-setup` is `overlay2`.

The `docker info` command, amongst many other items, shows the storage driver and related settings that are being used in Docker:

Example

```
[root@basecm11 ~]# module load docker
[root@basecm11 ~]# docker info

Client:
...
Context:      default
Debug Mode: false
...
Server:
Containers: 18
  Running: 8
  Paused: 6
  Stopped: 4
Images: 1
Server Version: 26.1.5
Storage Driver: overlay2
  Backing Filesystem: xfs
  Supports d_type: true
  Using metacopy: false
  Native Overlay Diff: true
  userxattr: false
Logging Driver: json-file
Cgroup Driver: systemd
Cgroup Version: 2
Plugins:
  Volume: local
  Network: bridge host ipvlan macvlan null overlay
  Log: awslogs fluentd gcplogs gelf journald json-file local splunk syslog
...
```

Docker data volumes are not controlled by the storage driver. Reads and writes to data volumes bypass the storage driver. It is possible to mount any number of data volumes into a container. Multiple containers can also share one or more data volumes.

More information about Docker storage back ends is available at <https://docs.docker.com/engine/storage/drivers/>.

For overlay2 back end driver storage, settings can be added as options if needed. In cmsh this can be done by setting the options parameter in the storagebackend submode under the docker::host role.

2.6 Docker Monitoring

When cm-docker-setup runs, it configures and runs the following Docker health checks:

1. makes a test API call to the endpoint of the Docker daemon
2. checks containers to see that none is in a dead state

The Docker daemon can be monitored outside of BCM with the usual commands directly.

BCM ways to manage or check on Docker include the following:

In CMDaemon, the docker service can be checked:

Example

```
[basecm11->device[node001]->services]% list
Service (key)      Monitored  Autostart
-----
docker             yes        yes
```

```

nslcd          yes          yes
[basecm11->device[node001]->services]% show docker
Parameter      Value
-----
Revision
Service         docker
Run if          ALWAYS
Monitored       yes
Autostart       yes
Timeout         -1
Belongs to role yes
Sickness check script
Sickness check script timeout 10
Sickness check interval 60

```

The docker0 interface statistics can be checked within the nodeoverview output:

Example

```

[basecm11->device[node001]]% nodeoverview
...

Interface      Received      Transmitted
-----
docker0        16.0 KiB     3.16 KiB
ens3           492 MiB     72.5 MiB
ens4           0 B         0 B
...

```

The health check measurable Docker checks if the docker service is healthy.

Example

```

[basecm11->device[node001]]% dumppmonitoringdata -1h now Docker
...
Timestamp      Value      Info
-----
2021/11/29 11:52:44.146 PASS
2021/11/30 18:28:44.146 PASS

```

2.7 Docker Setup For NVIDIA

NVIDIA GPU Cloud (NGC) is a cloud platform that runs on NVIDIA GPUs. NGC containers are lightweight containers that run applications on NVIDIA GPUs. The applications are typically HPC, machine learning, or deep learning applications.

An NGC can be set up to run NGC containers from the registry <http://ngc.nvidia.com>.

Docker can be configured as an NGC running NGC containers by using the NVIDIA Container Toolkit.

The BCM package provided for this is: `cm-nvidia-container-toolkit`.

One way to install and deploy this package is as part of the Docker installation, when running `cm-docker-setup` (section 2.1), where the cluster administrator selects yes as the answer to the request: ``Do you want to install the NVIDIA Runtime for Docker''.

Alternatively, if Docker has already been installed via `cm-docker-setup`, and if the package has not been installed, then it can be installed via the package manager, `yum` or `apt`. The toolkit has to be running on the compute nodes that have GPUs, which means that the installation must go to the appropriate node image (section 9.4 of the *Administrator Manual*). For example, if the appropriate image is `gpu-image`, then the package manager command for RHEL-based distributions would be:

Example

```
# yum install --installroot=/cm/images/gpu-image cm-nvidia-container-toolkit
```

The nodes that use that GPU image can then be rebooted to pick up the new package.

The GPU status can then be printed with the NVIDIA system management interface command. For example, if the image has been picked up by node001:

Example

```
[root@basecm11 ~]# ssh node001
Last login: Thu Dec  2 09:24:03 2021 from basecm11.cm.cluster
[root@node001 ~]# module load docker
[root@node001 ~]# docker run --runtime=nvidia --rm nvidia/cuda:11.4-base nvidia-smi
Unable to find image 'nvidia/cuda:11.4.0-base' locally
11.4.0-base: Pulling from nvidia/cuda
...
Digest: sha256:f0a5937399da5e4efb37fd7b75beb8c484b84dc381243c4b81fc5f9fcad42b66
Status: Downloaded newer image for nvidia/cuda:11.4.0-base
Mon Mar  7 17:30:48 2022
+-----+
| NVIDIA-SMI 440.33.01      Driver Version: 440.33.01      CUDA Version: 11.4      |
+-----+-----+-----+-----+-----+-----+
| GPU   Name                Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan   Temp  Perf  Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
+-----+-----+-----+-----+-----+-----+
|    0   Tesla K40c           On       | 00000000:00:06:0 Off |              Off    |
| 23%    32C    P8      22W / 235W |      0MiB / 12206MiB |      0%      Default |
+-----+-----+-----+-----+-----+-----+

+-----+
| Processes:                                                       GPU Memory |
|  GPU       PID    Type    Process name                               Usage      |
+-----+-----+-----+-----+-----+-----+
| No running processes found                                     |
+-----+

[root@node001 ~]# logout
Connection to node001 closed.
```

The available CUDA Docker images can be found at <https://hub.docker.com/r/nvidia/cuda>.

Container Registries

When a user creates a new container, an image specified by the user should be used. The images are kept either locally on a host, or in a registry. The image registry can be provided by a cloud provider or locally.

3.1 Container Registries: Introduction

3.1.1 Docker Hub, A Remote Registry

By default, Docker searches for images in Docker Hub, which is a cloud-hosted public and private image registry. Docker Hub serves a huge collection of existing images that users can make use of. Every user is allowed to create a new account, and to upload and share images with other users. Using the Docker client, a user can search for already-published images, and then pull them down to a host in order to build containers from them.

When an image is found in the registry, the Docker client verifies if the latest version of the image has already been downloaded. If not, then it downloads the images, and stores them locally. It also tries to synchronize them when a new container is created.

Docker Hub maintains limits for the amount of requests. To help get around the issues caused by the limits:

- A personal account can be created on Docker Hub, and the credentials used for pulls. Authenticated requests for personal accounts have a greater pull rate allowance than unauthenticated requests.
- A Pro, Team, or Business account (<https://docs.docker.com/docker-hub/usage/pulls/>) can be created on Docker Hub, and the credentials used for pulls. Authenticated requests for these accounts have an unlimited pull rate allowance.
- For a connected cluster, configuring a Docker Registry as a caching proxy mirror for Docker Hub (section 3.2.1) reduces the number of Docker Hub GET requests.
- Using Skopeo to manage container images and repositories helps minimize pulls for air-gapped clusters (Appendix C).

3.1.2 Local Image Registry Options: Classic Docker Registry Vs Harbor

Besides using Docker Hub for the image registry, the administrator can also install a local image registry on the cluster. BCM provides two ways to integrate such a local registry with the cluster, based on existing open source projects:

- The first one is the classic Docker registry originally provided by Docker Inc, and can be useful if the registry is used by trusted users, or when a Docker Hub proxy cache is required. Maintenance has been handed over to the CNCF, and future releases will be named CNCF Distribution.

- The second registry, Harbor, provides additional features such as security and identity management, and is aimed at the enterprise.

Both options can be installed with the `cm-container-registry-setup` utility, which comes with BCM's `cm-setup` package.

3.2 Container Registries: Setup And Configuration

Docker Registry and Harbor can be installed using the `cm-container-registry-setup` TUI utility, which comes with the `cm-setup` package.

They can also be installed using Base View as follows:

- The Container Registry Deployment Wizard is launched via the Base View navigation path: Containers > Container Registry Wizard
- Either Docker Registry, or Harbor, should be chosen as a registry.
- A single node is ticked for the deployment. The address, port, and the root directory for storing the container images are configured. If the head node is selected for Harbor, then the setup later on asks to open the related port on the head node. This is to make Harbor and the Harbor UI externally accessible.
- In the summary page, if the Ready for deployment box is ticked, then the administrator can go ahead with deploying the registry.
- When the deployment is complete, the Docker Registry becomes ready for use. Harbor can take a few additional minutes to be ready for use.

Similar to the case of `etcd` nodes (section 4.2), nodes that run Harbor or Docker Registry have the `datanode` option (page 265 of the *Administrator Manual*) when installed by BCM utilities. The option helps prevent the registry being wiped out by accident, and is added when the `cm-container-registry-setup` utility is used to install Harbor or Docker Registry. This extra protection is put into place because if a registry is wiped out, then the state of images in the container becomes incoherent and cannot be restored.

3.2.1 Docker Registry As A Pull Through Cache For Docker Hub

Docker Hub has a pull rate limit since November 2020 to limit the amount of images that can be pulled per hour (<https://docs.docker.com/docker-hub/usage/>). To limit the number of direct requests to Docker Hub, Docker Registry can be configured as a caching proxy mirror for Docker Hub.

When configured as a proxy mirror, Docker Registry serves cached images from Docker Hub, and downloads content from Docker Hub only when required.

Docker Registry is configured for proxy mode by selection during a `cm-container-registry-setup` run (figure 3.1):

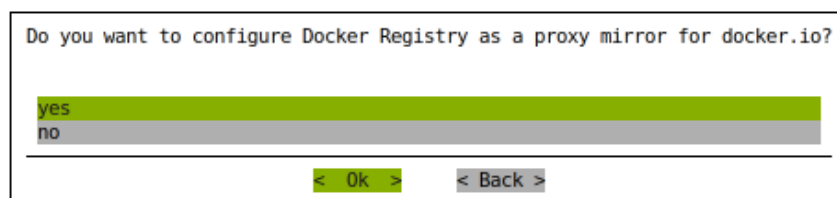


Figure 3.1: `cm-container-registry-setup`: configure Docker Registry as proxy cache

To access the proxy mirror, the port must be specified with the URL. For example, if Docker Registry is installed on `node001` using the default port, (5000) with `cm-kubernetes-setup`, then it is specified for Kubernetes as `https://node001:5000` (figure 3.2):

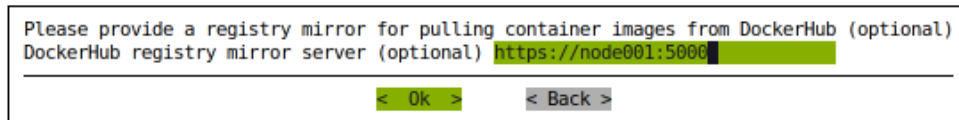


Figure 3.2: cm-kubernetes-setup: configuring a local proxy mirror

For example, if a Docker Registry proxy cache is configured on node001, port 5000, then the hello-world image is pulled explicitly through the proxy with:

Example

```
root@node001:~# docker run node001:5000/library/hello-world
Unable to find image 'node001:5000/library/hello-world:latest' locally
latest: Pulling from library/hello-world
17eec7bbc9d7: Pull complete
Digest: sha256:f7931603f70e13dbd844253370742c4fc4202d290c80442b2e68706d8f33ce26
Status: Downloaded newer image for node001:5000/library/hello-world:latest
```

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

3.2.2 Harbor UI

If the head node is where Harbor is to be installed, and is to be made externally accessible, then the Harbor UI can be accessed at `https://<head node hostname>:9443`.

If a different node is used for Harbor to be installed, then the related port must be forwarded locally.

Harbor can be logged into by default with the admin user and the default Harbor12345 password.

It is recommended to change that password after the first login.

3.2.3 Docker Registry And Harbor Configuration

Configuration is carried out using generic roles, which are roles with the prefix `generic::`, and that follow a generic template used by many roles.

- Generic Docker Registry configuration is managed with the `generic::docker-registry` role. The `docker-registry` daemon is documented upstream at <https://github.com/docker/distribution>.
- Generic Harbor configuration is managed with the `generic::harbor` role. The Harbor registry is documented upstream at <https://goharbor.io>.

Inspecting the options in the roles for Docker and Harbor shows what can be managed.

4

Kubernetes

Kubernetes is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts. With Kubernetes, it is possible to:

- scale applications on the fly
- seamlessly update running services
- optimize hardware usage by using only the resources that are needed

BCM provides the administrator with the required packages, allows Kubernetes to be set up on a cluster, and manages and monitors Kubernetes. More information about the design of Kubernetes, its command line interfaces, and other Kubernetes-specific details, can be found at the official online documentation at <https://kubernetes.io/docs/>.

To deploy most of Kubernetes, NVIDIA Base Command Manager from version 10.0 onwards uses kubeadm (<https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/>).

BCM runs CoreDNS, the Kubelet component (<https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/>), and the entire control plane inside Kubernetes.

The distributed key-value store used for Kubernetes, etcd, is typically run inside of Kubernetes on control planes. However, in BCM etcd is run outside of Kubernetes.

Kubernetes integration with BCM is available for Kubernetes v1.32, v1.33, v1.34, and v1.35 at the time of writing of this paragraph (May 2026). Kubernetes runs on the x86_64 and aarch64 architecture for all the BCM-supported Linux distributions. For a more up-to-date status, the features matrix at <https://service.bcm.nvidia.com/feature-matrix/> can be checked.

The version that is running on a cluster can be found with:

Example

```
root@basecm11:~# kubeadm version -o short
v1.34.5
```

The versions that are available with BCM integration can be found with the `cm-kubernetes-setup` utility (section 4.2):

Example

```
root@basecm11:~# cm-kubernetes-setup --list-versions
1.35
1.34 (NVIDIA AI Enterprise certified)
1.33 (NVIDIA AI Enterprise certified)
1.32 (NVIDIA AI Enterprise certified)
```

4.1 Reference Architecture

A reference architecture for Kubernetes in BCM comprises:

- *etcd nodes*: An etcd cluster—the Kubernetes distributed key-value storage—runs on an odd number (1, 3, 5 ...) of nodes.
- *control plane nodes*: typically run on head nodes or on dedicated nodes. 2 or 3 are recommended.
- *worker nodes*: typically run on regular nodes that are designated to run user workloads.

To avoid single point of failure and to achieve high availability, a minimum of three nodes is recommended for etcd, and a minimum of two control plane nodes is recommended.

Kubernetes API server proxy: An NGINX server is configured on the head node(s) and on all other nodes involved in the Kubernetes cluster (control plane and worker nodes). This NGINX server takes care of exposing the Kubernetes API server on a specific port, and load balances requests to the control plane nodes. Should one of those nodes go down it can detect this and stop sending requests until the node comes back up.

Since BCM version 8.2, multiple clusters of Kubernetes can be deployed. In such a configuration, the same nodes cannot be shared across different Kubernetes clusters. Because of the NGINX server, a port is reserved on the head node(s) for every Kubernetes cluster. This is required for Kubernetes HA, and it also allows kubectl and other tools such as Helm to be used from the head node, to access each Kubernetes cluster.

Gateway Proxy: The same NGINX server is also used as the entry point on the head node(s) to forward traffic to the Kubernetes Gateways. This functionality is referred to as the “Gateway proxy” and is described in detail in section 4.17.

4.1.1 Hardware Requirements

To run properly, Kubernetes deployment requires a per-node minimum of

- 16 GB RAM
- two CPU cores

Without these minimum requirements, deployment may succeed, but further configuration, such as starting up Jupyter kernels, may fail with problems that are hard to diagnose. For example, error reporting may not show a lack of resources because some components are automatically deleted.

4.2 Kubernetes Setup

The usual, and recommended way, to install Kubernetes with BCM is to install it interactively from a TUI session (section 4.2.8).

An alternative way, typically used for automated non-interactive setups, is to use a plain command line specification (section 4.2.7), which usually results in a long command line.

BCM deploys Kubernetes with the `cm-kubernetes-setup` utility, which is part of the `cm-setup` package. Several recent versions of Kubernetes are offered (figure 4.1):

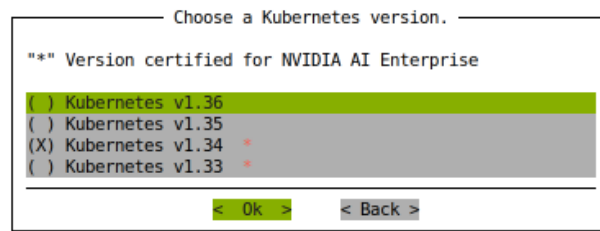


Figure 4.1: Kubernetes setup TUI session (section 4.2.8): version selection screen

An air-gapped installation is possible (Appendix B).

BCM provides or uses the following Kubernetes-related packages:

- `contrack` (`contrack-tools` on RHEL-based distributions) and `nginx`: These distribution packages are always installed on the head node(s) and on the control-plane and worker node(s).
- `cm-etcd`: This BCM package is installed on the nodes selected for etcd. In a similar way to the case of Harbor or Docker Registry (section 3.2), the nodes that run etcd are protected by BCM with the `datanode` option (page 265 of the *Administrator Manual*). For etcd nodes, the option is added during the `cm-kubernetes-setup` installation. As in the case for the registries, the `datanode` option is set in order to help prevent the administrator from wiping out the existing state of etcd nodes. Wiping out the state of etcd nodes means that the Kubernetes cluster becomes incoherent and that it cannot be restored to where it was just before the etcd nodes were wiped.
- `cm-containerd`: This BCM package has the containerd runtime.
- `cm-nvidia-container-toolkit`: This BCM package has the NVIDIA container toolkit (includes NVIDIA container runtime).
- `cm-kube-diagnose`: This BCM package has Helper tools to diagnose malfunctioning Kubernetes clusters.

Kubernetes .rpm and .deb packages themselves are installed from the Kubernetes community-owned software repositories (<https://kubernetes.io/blog/2023/08/15/pkg-k8s-io-introduction/>). These repositories host Kubernetes versions starting from v1.24.0 at the time of writing of this paragraph (March 2024).

4.2.1 Kubernetes Networking

Early on during the wizard session (figure 4.2), a name for the cluster is requested. The wizard pre-fills it with `default`, but this should not be confused with the Kubernetes `default` namespace. Here, the name is used instead, inside BCM, to identify the cluster, configuration files, and other resources such as module files.

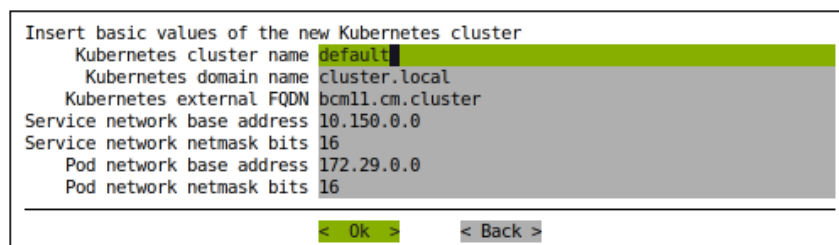


Figure 4.2: Kubernetes setup TUI session: networking selection screen

This screen also allows the following important choices:

- `Kubernetes external FQDN`: This is the FQDN that is placed as one of the entries in the public-facing certificates generated for this Kubernetes cluster.

Configuring the public-facing certificate of the `bcm-gateway` Gateway (served by `Kgateway`) is discussed further in section 4.17.7.

- `Service network base address` and `Service network netmask bits`: These define the CIDR for the service network. The wizard pre-fills the fields. It also tries to avoid pre-filling them with overlapping network ranges, by taking any existing network known to BCM into account.
- `Pod network base address` and `Pod network netmask bits`: These define the CIDR for the pod network. The wizard pre-fills these. It also tries to avoid pre-filling them with overlapping network ranges, by taking any existing network known to BCM into account. By default, entire /24 network ranges are assigned to individual Kubernetes nodes from the pod CIDR.

The packages are installed automatically from the repository when the administrator runs `cm-kubernetes-setup` from the command line.

The log file produced by the setup can be found in `/var/log/cm-kubernetes-setup.log`.

4.2.2 Kubernetes Core Add-ons

During setup, some critical add-on components such as the Networking Component for Kubernetes are automatically deployed. Most components are in the `kube-system` namespace, but others have their own namespaces. In BCM some add-ons are treated as Kubernetes applications (Chapter 5), and belong to the default app group, `system`. In the future Helm is expected to manage most, if not all, components instead of BCM Kubernetes applications.

The user is prompted for a Networking Component, Kubernetes operators (managed using Helm), Kubernetes add-ons (managed using Kubernetes applications), and more. They all result in either Helm charts being deployed, or in Kubernetes applications being managed from BCM.

A `cmsh` treeview illustrating the hierarchy to access these applications is:

```
[cmsh]
|-- ...
|
|-- kubernetes[default]
|       |-- appgroups[system]
|               |-- applications
|-- ...
```

Helm charts can be found using the `helm` command. For example using `helm list -A -a` on a cluster (after loading the correct module file).

In the past the DNS component was also provided as an add-on. However since BCM version 10.0, `kubeadm` is used to bundle CoreDNS as part of the default control plane.

Kubernetes apps are discussed further in Chapter 5. Kubernetes operators are discussed further in Chapter 6.

Kubeadm Components

Components deployed as part of the default Kubernetes control plane are:

- Kubernetes API server
- Kubernetes Scheduler
- Kubernetes Controller Manager
- Kubernetes Proxy

- Core DNS

These are deployed through the “static pod” mechanism (<https://kubernetes.io/docs/tasks/configure-pod-container/static-pod/>). Kubernetes control plane components are described at: <https://kubernetes.io/docs/concepts/overview/components/>.

CoreDNS: CoreDNS is the DNS server add-on for internal service discovery. It reads the IP addresses of services and pods from etcd, and resolves domain names for them. If a domain name is not found because the domain is external to the Kubernetes cluster, then CoreDNS forwards the request to the main DNS server. BCM uses CoreDNS version 1.12.0 with Kubernetes version 1.34.

Networking Component

An important component managed through BCM’s Kubernetes Apps is the Networking component. This adheres to the Container Networking Interface, or CNI interface. At the time of writing the available CNIs in the setup wizard are either Calico or Flannel. However BCM does support other CNIs, for example Cilium, as described in the knowledge base article at <https://kb.brightcomputing.com/knowledge-base/installing-cilium-networking-for-kubernetes/>.

Flannel: Flannel is a simple and easy way to configure a layer 3 network fabric designed for Kubernetes.

Calico: Calico is a popular open source networking and network security solution for Kubernetes. It provides network connectivity between workloads and security policies features. It can be configured to use eBPF.

Further details on Calico can be found at <https://docs.projectcalico.org/>.

If the Kubernetes cluster is composed of more than 50 nodes, then the Calico component Typha is also automatically deployed for better scalability. The number of Typha replicas is calculated by allocating one Typha replica per 150 nodes, with a minimum of 3 (above 50 nodes) and a maximum of 20.

If an initial deployment of the Kubernetes cluster has fewer than 50 nodes, but nodes are then added to the Kubernetes cluster so that the 50 node threshold is exceeded, then Typha is not automatically enabled. In this case, Typha can be enabled manually using `cmsh` as follows:

Example

```
[basecm11->kubernetes[default]->appgroups[system]->applications[calico]]% environment
[basecm11->kubernetes[default]->appgroups[system]->applications[calico]->environment]% list
Name (key)          Value          Nodes environment
-----
calico_typha_replicas 0              no
calico_typha_service  none          no
cidr                 10.141.0.0/16 no
[basecm11->kubernetes...[calico]->environment]% set calico_typha_service value calico-typha
[basecm11->kubernetes*...[calico*]->environment*]% set calico_typha_replicas value 3
[basecm11->kubernetes*[default*]->appgroups*[system*]->applications*[calico*]->environment*]% commit
[basecm11->kubernetes[default]->appgroups[system]->applications[calico]->environment]%
```

Cilium: Cilium (<https://cilium.io/>) is a networking, observability, and security solution with an eBPF-based dataplane.

4.2.3 Kubernetes Optional Components

The following components are installed by default unless otherwise noted. However, the user can choose to skip some or all of them during the setup.

Kgateway

Kgateway is the implementation that BCM uses to fulfill the Kubernetes Gateway API specification. It is discussed further in section 4.17.

MetalLB

MetalLB is a bare-metal load balancer that allocates external IP addresses to any LoadBalancer-type Service on the cluster. BCM deploys it alongside Kgateway to back the Kubernetes Gateways (section 4.17.4), but it can also be used outside the Gateway scope.

Kubernetes Metrics Server

The Kubernetes Metrics Server is an add-on that is a replacement for Heapster. It aggregates metrics, and provides container monitoring and performance analysis. It exposes metrics using an API.

Kubernetes State Metrics

kube-state-metrics is an add-on agent to generate and expose cluster-level Kubernetes metrics (section G.1.14 of the *Administrator Manual*) for objects. The project is not focused on the health of the individual Kubernetes components, but rather on the health of the various objects inside, such as deployments, nodes and pods.

4.2.4 Kubernetes Deprecated And Retired Components

The following components were part of BCM's Kubernetes installation in earlier releases. They are no longer installed by default, and are documented here for reference and to point users to their replacements.

Kubernetes Dashboard (retired)

The Kubernetes Dashboard provided a web UI for cluster administration and metrics visualization in BCM versions before 11.32.0. It was retired upstream and has been removed in BCM version 11.32.0.

Ingress NGINX Controller (deprecated)

The Ingress NGINX Controller (`ingress-nginx`) provided ingress routing and web load balancing in BCM versions before 11.33.0 using standard Kubernetes Ingress resources, which were consumed together with `ingress-nginx`-specific annotations, such as `nginx.ingress.kubernetes.io/*`.

The Ingress NGINX Controller was deployed as a `ingress-nginx` Helm release. In BCM, it was exposed externally using the legacy Ingress proxy on the non-standard ports 30080 (HTTP) and 30443 (HTTPS), and the TLS certificate for externally-exposed services was served on 30443. It was deprecated upstream and in BCM starting with version 11.32.0, and is disabled by default from BCM 11.33.0, where it is replaced by Kgateway (section 4.17). For BCM clusters that were originally deployed with the Ingress NGINX Controller, the procedure to migrate to the Gateway-based stack is described in section 4.17.8.

4.2.5 Helm Kubernetes Package Manager

Helm is an add-on that manages *charts*, which are packages of pre-configured Kubernetes resources. The Helm component is installed and properly configured with BCM's Kubernetes installation by default. It is initialized and ready for use by every Kubernetes user when the Kubernetes module is loaded. BCM uses Helm version 3.18.

When the Helm binary is installed during the Kubernetes setup, an offer is made to deploy charts during the setup process. Most operators are documented in Chapter 6.

Other parts of the wizard also determine which Helm charts are deployed as part of the setup for the following:

- Kyverno and Kyverno Policies (section 4.10.1)

- BCM Permissions Manager (section 4.15)
- BCM Local Path Provisioner (section 4.20)

Example

```
root@basecm11:~# helm list -A -a
```

NAME	NAMESPACE	CHART	APP VERSION
cluster-installer	runai	... cluster-installer-2.15.9	2.15.9
cm-jupyter-kernel-operator	cm	... cm-jupyter-kernel-operator-0.1.10	0.1.10
cm-kubernetes-mpi-operator	cm	... mpi-operator-0.4.0	0.4.0
gpu-operator	gpu-operator	... gpu-operator-v23.9.1	v23.9.1
kyverno	kyverno	... kyverno-3.0.4	v1.10.2
kyverno-policies	kyverno	... kyverno-policies-3.0.3	v1.10.2
local-path-provisioner	cm	... cm-kubernetes-local-path-provisioner-0.0.26	0.0.26
metallb	metallb-system	... metallb-0.14.3	v0.14.3
network-operator	network-operator	... network-operator-23.10.0	v23.10.0
permissions-manager	cm	... cm-kubernetes-permissions-manager-0.4.8	0.4.8
postgres-operator	postgres-operator	... postgres-operator-1.10.1	1.10.1
prometheus-adapter	prometheus	... prometheus-adapter-3.3.1	v0.9.1
prometheus-operator	prometheus	... kube-prometheus-stack-35.5.1	0.56.3
spark-operator	spark-operator	... spark-operator-1.1.27	v1beta2-1.3.8-3.1.1

4.2.6 Note On Run:ai Operator Dependencies

Selecting The Run:ai Operator Dependencies

When the cm-kubernetes-setup wizard is run, with the intent of running Run:ai as well, the operator selection screen must include the dependencies that Run:ai requires. For Run:ai 2.25, these are:

- NVIDIA GPU Operator
- Grafana Operator
- Kgateway
- Knative Operator
- Kubeflow Training Operator
- Kubernetes Metrics Server
- Kubernetes MPI Operator
- Kubernetes State Metrics
- LeaderWorkerSet Operator
- MetalLB
- Network Operator (technically optional, but recommended)
- NIM Operator (optional)
- Prometheus Adapter
- Prometheus Operator Stack

The Run:ai operator and its prerequisites are described in more detail in section 6.8.

4.2.7 Kubernetes Setup From The Command Line

The cm-kubernetes-setup command line utility has the following usage synopsis:

```
[root@basecm11 ~]# cm-kubernetes-setup -h
usage: Kubernetes Setup cm-kubernetes-setup [-c <config_file>]
[--list-versions]
[--list-operators-versions]
[--cluster CLUSTER_NAME]
[--skip-docker]
[--skip-reboot]
[--skip-image-update]
[--skip-kube-version-check]
[--skip-dns-configuration-check]
[--skip-package-manager-update-check]
[--skip-install-repos]
[--adv-configuration-bcm-nvidia-packages]
[--add-user USERNAME_ADD]
[--list-users]
[--get-user GET_USER]
[--modify-user USERNAME_MODIFY]
[--remove-user USERNAME_REMOVE]
[--namespace NAMESPACE]
[--add-to-namespace]
[--remove-from-namespace]
[--role edit,admin,view,cluster-admin]
[--runas-uid RUNAS_UID]
[--runas-gids RUNAS_GIDS]
[--user-paths USER_PATHS]
[--allow-all-uids]
[--operators OPERATORS]
[--backup-permissions FILE]
[--restore-permissions FILE]
[--list-operators]
[--update-addons]
[--remove]
[--yes-i-really-mean-it] [--pull]
[--images IMAGES]
[--nodes NODES]
[--node-selector NODE_SELECTOR]
[--pull-registry-server PULL_REGISTRY_SERVER]
[--pull-registry-username PULL_REGISTRY_USERNAME]
[--pull-registry-email PULL_REGISTRY_EMAIL]
[--pull-registry-password PULL_REGISTRY_PASSWORD]
[--docker-io-username DOCKER_IO_USERNAME]
[--docker-io-password DOCKER_IO_PASSWORD]
[--kubeadm-image-repository KUBEADM_IMAGE_REPOSITORY]
[--containerd-sandbox-image CONTAINERD_SANDBOX_IMAGE]
[--enable-ovn] [--latest-versions]
[--skip-ovn-prerequisites-checks]
[--skip-netq-prerequisites-checks]
[--provision-node PROVISION_NODE]
[--airgap-registry AIRGAP_REGISTRY]
[--airgap-registry-as-helm | --airgap-helm-repo AIRGAP_HELM_REPO]
[-v] [--store-name-aliases]
[--no-distro-checks] [--json]
[--output-remote-execution-runner]
```

```
[--on-error-action debug,remotedebug,undo,abort]
[--skip-packages]
[--min-reboot-timeout <reboot_timeout_seconds>]
[--allow-running-from-secondary]
[--dev] [-h]
```

optional arguments:

```
--cluster CLUSTER_NAME      Name of the referred Kubernetes cluster
-h, --help                  Print this screen
```

common:

```
Common arguments

-c <config_file>           Load runtime configuration for plugins from a YAML config file
```

Supported versions information:

```
--list-versions            Show supported Kubernetes versions
--list-operators-versions  List available for selection Kubernetes operators versions
```

installing Kubernetes clusters:

Flags for installing or managing Kubernetes clusters

```
--skip-docker              Skip the Docker installation steps.
--skip-reboot              Skip the reboot steps.
--skip-image-update        Skip the image update steps.
--skip-kube-version-check  Skip Kubernetes version check before install (setup will not continue if it finds
                           unsupported version).
--skip-dns-configuration-check
                           Skip DNS configuration check before install (setup will not continue if it finds DNS
                           malfunctioning).
--skip-package-manager-update-check
                           Skip Package Manager update/refresh check before install (setup will not continue if
                           'apt update' fails on Ubuntu for example).
--skip-install-repos       Skip installation of kubernetes and helm package repos.(assumes they have been manually
                           configured, e.g., in the case of air-gapped).
--adv-configuration-bcm-nvidia-packages
                           Allow the option to override the selection of BCM NVIDIA packages to be installed when
                           setting up the NVIDIA GPU operator
```

user management:

Flags for adding a new user in a Kubernetes cluster

```
--add-user USERNAME_ADD    Create a new user in a Kubernetes cluster
--list-users                Get information about configured Kubernetes users
--get-user GET_USER         Get information about configured Kubernetes users
--modify-user USERNAME_MODIFY
                           Modify user in a Kubernetes cluster
--remove-user USERNAME_REMOVE
                           Remove existing user from a Kubernetes cluster
--namespace NAMESPACE     Specify namespace for user (--get-user, --modify-user) role binding
```

```

--add-to-namespace      Indicate if permissions to manage namespace needs to be granted for a given user
                        (--modify-user)
--remove-from-namespace
                        Indicate if permissions to manage namespace needs to be revoked for a given user
                        (--modify-user)
--role edit,admin,view,cluster-admin
                        Specify role for the new (--add-user) and existing (--modify-user) role binding
                        (Default: edit)
                        For 'cluster-admin' namespace flag is ignored
--runas-uid RUNAS_UID
                        UID is allowed to be used in unprivileged pods (--add-user, --modify-user)
--runas-gids RUNAS_GIDS
                        Comma-separated list of GIDs allowed to be used in unprivileged pods (--add-user,
                        --modify-user)
--user-paths USER_PATHS
                        Comma-separated list of paths user is able to mount in pods (--add-user, --modify-user)
--allow-all-uids       Allow user to run processes in pods as any user (--add-user, --modify-user)
                        hostPath volumes will be disabled for such pods
--operators OPERATORS
                        Comma-separated list of operators user has access to (--add-user, --modify-user)

```

backup or restore Permission Manager user configurations:

Flag for managing permission manager user configuration

```

--backup-permissions FILE
                        Save permissions to file
--restore-permissions FILE
                        Restore permissions from file. Workload which is already run by users in their
                        namespaces will be affected

```

list available operators:

Flag to list available Kubernetes operators

```

--list-operators      List available Kubernetes operators

```

update kubernetes addons:

Flags for updating Kubernetes addons

```

--update-addons       Update Addons
--patch-kube-prometheus-stack
                        Patch to improve kube-prometheus-stack helm chart (scrape config)

```

removing Kubernetes clusters:

Flags for removing a Kubernetes cluster

```

--remove              Remove a Kubernetes cluster
--yes-i-really-mean-it
                        Required for additional safety

```

pulling images to the nodes:

Flags for pulling images to the nodes

```

--pull                Pull images to the nodes
--images IMAGES       Comma-separated list of images to pull (--pull)
--nodes NODES         Comma-separated list of nodes to pull images to (--pull)

```

```
--node-selector NODE_SELECTOR
    Selector (label query) to filter on, supports '=', '==', and '!='.(e.g.
    key1=value1,key2=value2) (--pull)
--pull-registry-server PULL_REGISTRY_SERVER
    Registry server to authenticate
--pull-registry-username PULL_REGISTRY_USERNAME
    Registry username
--pull-registry-email PULL_REGISTRY_EMAIL
    Email to authenticate
--pull-registry-password PULL_REGISTRY_PASSWORD
    Registry password
```

docker.io:

Flags for authenticating to docker hub

```
--docker-io-username DOCKER_IO_USERNAME
    Username
--docker-io-password DOCKER_IO_PASSWORD
    Password
```

kubeadm:

Flags for configuring kubeadm

```
--kubeadm-image-repository KUBEADM_IMAGE_REPOSITORY
    Custom imageRepository
```

containerd:

Flags for configuring containerd

```
--containerd-sandbox-image CONTAINERD_SANDBOX_IMAGE
    Custom sandbox image (e.g. my-registry:5000/pause:3.8)
```

ovn:

Flags for configuring OVN CNI

```
--enable-ovn          Allow selection of OVN CNI during Kubernetes setup (currently not shown by default).
--latest-versions      Dynamically determine and use latest versions of OVN Kubernetes CNI and Dashboard
                        charts.
--skip-ovn-prerequisites-checks
                        Provide this flag to skip precondition checks such as checking for Network Layouts,
                        Linux distro requirements for OVN CNI.
```

netq:

Flags for configuring NetQ

```
--skip-netq-prerequisites-checks
    Provide this flag to skip precondition checks such as checking for disk size, Linux
    distro, etc.
```

provisioning:

Flags for provisioning Kube nodes

```
--provision-node PROVISION_NODE
    Provision a node with Kube packages and prepare
```

Air-gapped deployment:

```
--airgap-registry AIRGAP_REGISTRY
                        AirGap container registry address
--airgap-registry-as-helm
                        Use AirGap container registry as helm repository
--airgap-helm-repo AIRGAP_HELM_REPO
                        Helm repository address
```

advanced:

Various **advanced** configuration options flags.

```
-v, --verbose           Verbose output
--store-name-aliases   Store hostname aliases for head nodes (active and passive) and default category
--no-distro-checks     Disable distribution checks based on ds.json
--json                 Use json formatting for log lines printed to stdout
--output-remote-execution-runner
                        Format output for CMDaemon
--on-error-action debug,remotedebug,undo,abort
                        Upon encountering a critical error, instead of asking the user for choice, setup will
                        do selected action.
--skip-packages        Skip any stages which install packages. Requires packages to be already installed.
--min-reboot-timeout <reboot_timeout_seconds>
                        How long to wait for nodes to finish reboot (default and minimum allowed: 300 seconds).
--allow-running-from-secondary
                        Allow to run the wizard from the secondary when it is the active head node.
--dev                  Enables additional command line arguments
```

The `cm-kubernetes-setup` utility should be executed on the console.

Dealing With A Pre-existing Docker Installation

Docker (Chapter 2) is no longer a requirement for Kubernetes configured by BCM. This is because Kubernetes can directly interface with containerd through its Container Runtime Interface (CRI). Docker can co-exist with Kubernetes, and can be set up as discussed in section 2.1.

4.2.8 Kubernetes Setup From A TUI Session

When the Kubernetes installation is carried out using `cm-kubernetes-setup` without any options, a TUI wizard starts up. The administrator can answer several questions in the wizard. Questions that are asked include questions about which of the node categories or which of the individual nodes should be configured to run the Kubernetes services. There are also questions about the service and pod networks parameters, the port numbers that will be configured for the daemons, whether to install specific add-ons, and so on. After the wizard has been completed, a configuration file can be saved that can be used to set up Kubernetes.

The configuration file can be deployed immediately from the wizard, or it can be deployed later by specifying it as an option to `cm-kubernetes-setup`, in the form `-c <file>`.

If no deployment has been carried out earlier, then the main operations screen of the wizard shows just two options, Deploy and Exit.

If deployment has already been carried out, then the further options that are available are also displayed (figure 4.3):

Example

```
[root@basecm11 ~]# cm-kubernetes-setup
```

TUI session starts up:

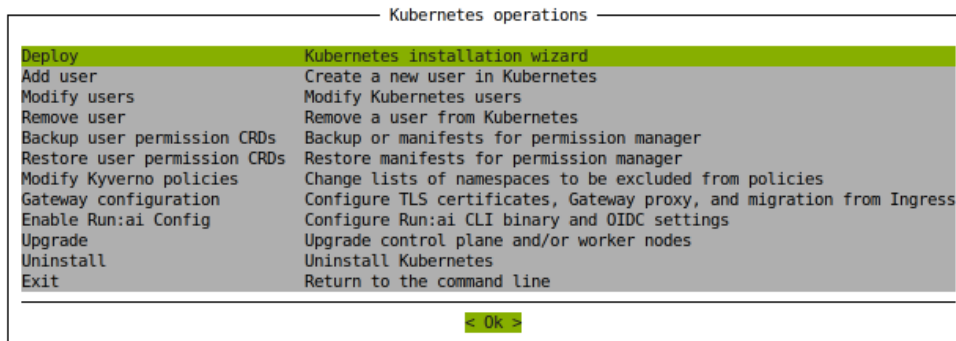


Figure 4.3: Kubernetes setup TUI session: main operations screen after a deployment

The deployment using CLI or using TUI assigns the appropriate roles, and adds the new Kubernetes cluster. The deployment adds health checks to the monitoring configuration, and it generates certificates for the Kubernetes daemons.

Calico is set as the Kubernetes network plugin by default. Flannel is an option.

The control-plane, worker, and etcd nodes can be assigned to specific nodes or categories.

The network configuration settings for the Kubernetes cluster can be specified. Ports have default assignments, but can be re-assigned as needed. The etcd spool file path can be set.

The following options are also possible:

- a registry mirror from DockerHub can be specified
- the Kubernetes API server can be exposed to the external network
- the internal network used by Kubernetes nodes can be selected

Add-ons that are available are:

- Kubernetes Metrics Server
- Kubernetes State Metrics
- Kubernetes Policy Engine

Operator packages are application managers, and are described further in Chapter 6. Operators that can be installed are:

- NVIDIA GPU Operator
- cm-jupyter-kernel-operator
- cm-kubernetes-mpi-operator
- Kgateway
- KubeFlow Training Operator
- MetalLB
- Network Operator
- NIM Operator
- postgresql-operator
- Prometheus Operator Stack

- spark-operator
- Tiger operator (Calico)

The permission manager—an application for role-based access control—can also be configured.

4.2.9 Testing Kubernetes

To test that Kubernetes works, the `cluster-info` command can be run. For example, on the head node, `basecm11`:

Example

```
[root@basecm11 ~]# module load kubernetes      #not actually needed -- autoloaded these days
[root@basecm11 ~]# kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:10443
CoreDNS is running at https://127.0.0.1:10443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

To further debug and diagnose cluster problems, use `'kubectl cluster-info dump'`.

After `cm-kubernetes-setup` finishes, and the regular nodes have been rebooted, the state of the nodes can be found by running the `get nodes` command:

Example

```
[root@basecm11 ~]# kubectl get nodes
NAME      STATUS    ROLES    AGE   VERSION
node001   Ready     worker   18h   v1.34
node002   Ready     worker   18h   v1.34
basecm11  Ready     control-plane,master  18h   v1.34
```

A six node cluster should show the following Kubernetes installed add-ons, when using `kubectl` with the `get all -n kube-system` option (some lines truncated):

Example

```
[root@basecm11 ~]# kubectl get all -n kube-system
```

NAME	READY	STATUS	RESTARTS	AGE
pod/calico-kube-controllers-58497c65d5-skhgw	1/1	Running	0	26m
pod/calico-node-27xj7	1/1	Running	0	26m
pod/calico-node-6hmm5	0/1	Running	1	26m
pod/calico-node-987qv	1/1	Running	0	26m
pod/calico-node-gcbcm	1/1	Running	0	26m
pod/calico-node-hlsrj	1/1	Running	0	26m
pod/calico-node-q7k4v	1/1	Running	0	26m
pod/calico-node-qdbq5	0/1	Running	0	26m
pod/calico-node-v2dxj	1/1	Running	0	26m
pod/coredns-6768db756-8l9fs	1/1	Running	0	26m
pod/coredns-6768db756-cs58q	1/1	Running	0	26m
pod/kube-state-metrics-758ccc75d6-75dsn	1/1	Running	0	26m
pod/metrics-server-7b477dd7b9-2drkg	1/1	Running	0	26m
pod/metrics-server-7b477dd7b9-z6nch	1/1	Running	0	26m

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/calico-typha	ClusterIP	10.150.121.25	<none>	5473/TCP	26m
service/kube-dns	ClusterIP	10.150.255.254	<none>	53/UDP,53/TCP,9153/TCP	26m
service/kube-state-metrics	ClusterIP	None	<none>	8080/TCP,8081/TCP	26m
service/metrics-server	ClusterIP	10.150.99.149	<none>	443/TCP	26m

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
daemonset.apps/calico-node	8	8	6	8	6	kubernetes.io/os=linux	26m

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/calico-kube-controllers	1/1	1	1	26m
deployment.apps/calico-typha	0/0	0	0	26m
deployment.apps/coredns	2/2	2	2	26m
deployment.apps/kube-state-metrics	1/1	1	1	26m
deployment.apps/metrics-server	2/2	2	2	26m

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/calico-kube-controllers-58497c65d5	1	1	1	26m
replicaset.apps/calico-typha-68857595fc	0	0	0	26m
replicaset.apps/coredns-6768db756	2	2	2	26m
replicaset.apps/kube-state-metrics-758ccc75d6	1	1	1	26m
replicaset.apps/metrics-server-7b477dd7b9	2	2	2	26m

The administrator can now configure the cluster to suit the particular site requirements.

4.3 Using GPUs With Kubernetes: NVIDIA GPUs

4.3.1 Prerequisites

The GPUs have to be recognized by the nodes, and have the appropriate drivers (such as `nvidia-driver`) installed. Details on how to do this are given in Chapter 9 of the *Installation Manual*.

To verify the GPUs are recognized and have drivers in place, the `nvidia-smi` command can be run. The response displayed for a GPU should look similar to the following:

Example

```
root@node001:~# nvidia-smi
Mon May  5 13:03:52 2025

+-----+
| NVIDIA-SMI 570.82                Driver Version: 570.82                CUDA Version: 12.8                |
+-----+-----+-----+
| GPU   Name                               Persistence-M | Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf              Pwr:Usage/Cap |      Memory-Usage | GPU-Util  Compute M. |
|                                           MIG M. |
+=====+=====+=====+
|   0   NVIDIA Graphics Device              On | 00000008:01:00.0 Off |                    0 |
| N/A   38C    P0              184W / 1200W | 1MiB / 189471MiB |      0%      Default |
|                                           Disabled |
+-----+-----+-----+
|   1   NVIDIA Graphics Device              On | 00000009:01:00.0 Off |                    0 |
| N/A   36C    P0              160W / 1200W | 1MiB / 189471MiB |      0%      Default |
|                                           Disabled |
+-----+-----+-----+

+-----+
| Processes: |
| GPU   GI    CI          PID    Type    Process name                        GPU Memory |
|      ID    ID              |                 |           Usage |
+=====+
| No running processes found |
+-----+
```

If a non-BCM Containerd has already been deployed before Kubernetes is deployed, then `cm-kubernetes-setup` may replace an existing Containerd configuration file in order to enable NVIDIA GPU integration using a Kubernetes CNI plugin. This is because Containerd is configured by `cm-kubernetes-setup`, overwriting any previous configuration.

4.3.2 New Kubernetes Installation

As part of the setup, `cm-kubernetes-setup` assigns a new role to the Kubernetes worker nodes: `generic::containerd`.

The role has a Configurations submode, in which the `containerd-cri` object can be configured. The entry for `Filename` specifies the path to the `cri.toml` file, which contains content used by the container runtime interface on the Kubernetes worker nodes that have been assigned the role.

Example

```
[basecm11->configurationoverlay[kube-default-worker]->roles]% use generic::containerd
[...]->roles[generic::containerd]]% show
Parameter                                Value
-----
Name                                     generic::containerd
Type                                     GenericRole
Add services                             yes
Services                                 containerd
Configurations                           <2 in submode>
Environments                             <1 in submode>
Exclude list snippets                    <1 in submode>
Data node                                no
[...]->roles[generic::containerd]]% configurations
[...]->roles[generic::containerd]->configurations]% use containerd-cri
[...]->roles[generic::containerd]->configurations[containerd-cri]]% show
Parameter                                Value
-----
Name                                     containerd-cri
Type                                     static
Create directory                         yes
Filename                                 /cm/local/apps/containerd/var/etc/conf.d/cri.toml
Filemask directory                       0644
User name
Group name
Disabled                                 no
Service action on write                  RESTART
Service stop on failure                  yes
Content                                  <645B>
Filemask                                 0644
```

The file with the CRI (Container Runtime Interface) configuration is created in directory:

```
/cm/local/apps/containerd/var/etc/conf.d
```

and included into the main Containerd configuration file:

```
/cm/local/apps/containerd/var/etc/config.toml
```

with the imports statement:

```
imports = ["/cm/local/apps/containerd/var/etc/conf.d/*.toml"]
```

Whatever the container runtime that is selected, if NVIDIA GPU integration is required then the NVIDIA container toolkit is taken care of by the installer. The configuration for Containerd is added as a separate configuration file in the configurations submode, as `nvidia-cri`.

NVIDIA GPUs are integrated into Kubernetes using the NVIDIA GPU operator. This is discussed further in section 6.4.

4.3.3 Existing Kubernetes Installation

The NVIDIA GPU operator can always be deployed through Helm. The official documentation for the NVIDIA GPU operator is at

<https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/overview.html>

BCM also has a KB article with more information at:

<https://kb.brightcomputing.com/knowledge-base/the-nvidia-gpu-operator-with-kubernetes-on-a-bright-cluster/>

4.4 Using GPUs With Kubernetes: AMD GPUs

4.4.1 Prerequisites

The GPUs have to be recognized by the node. One way to check this from within BCM is to run `sysinfo` for the node:

Example

```
[basecm11->device[basecm11]]% sysinfo | grep GPU
Number of GPUs          1
GPU Driver Version      4.18.0-193.el8.x86_64
GPU0 Name               Radeon Instinct MI25
```

In order to make Kubernetes aware of nodes that have AMD GPUs, the AMD GPU device plugin has to be deployed as a DaemonSet inside Kubernetes. The official GitHub repository that hosts this plugin can be found at:

<https://github.com/RadeonOpenCompute/k8s-device-plugin>

The device plugin requires Kubernetes v1.16+, which has been around since BCM version 9.0. With some extra instructions, the plugin can also be made a part of BCM version 8.2.

The DaemonSet YAML file can be deployed with:

Example

```
kubectl create -f https://raw.githubusercontent.com/RadeonOpenCompute/k8s-device-plugin/v1.16/k8s-ds-amdgpu-dp.yaml
```

4.4.2 Managing The YAML File Through CMDaemon

The plugin can be added by the user using the Kubernetes appgroups as an application. In the session that follows, it is given the arbitrary name `device-plugin`:

Example

```
[root@basecm11 ~]# wget https://raw.githubusercontent.com/RadeonOpenCompute/k8s-device-plugin/v1.16/k8s-ds-amdgpu-dp.yaml -O /tmp/k8s-ds-amdgpu-dp.yaml
[root@basecm11 ~]# cmsg
[basecm11]% kubernetes
[basecm11->kubernetes[default]]% appgroups
[basecm11->kubernetes[default]->appgroups]% add amd
[basecm11->kubernetes*[default*]->appgroups*[amd*]]% applications
[basecm11->kubernetes*[default*]->appgroups*[amd*]->applications]% add device-plugin
```

The configuration of the plugin can be set to the YAML file, by setting the `config` parameter to take the value of the YAML file path.

Example

```
[basecm11->...[amd*]->applications*[device-plugin]]% set config /tmp/k8s-ds-amdgpu-dp.yaml
[basecm11->kubernetes*[default*]->appgroups*[amd*]->applications*[device-plugin]]% show
Parameter                                Value
-----
Name                                    device-plugin
Revision
Format                                  Yaml
Enabled                                  yes
Config                                  <914B>
Environment                             <0 in submode>
Exclude list snippets                    <0 in submode>
[basecm11->kubernetes*[default*]->appgroups*[amd*]->applications*[device-plugin]]% commit
```

The YAML file can also be edited within `cmsh` after it has been set, by running `set config` without a value.

There are older releases available, starting from Kubernetes v1.10, if needed. Saving this device-plugin YAML should result in pods being scheduled on all the non-tainted nodes, as seen by listing the pods (some columns elided):

```
[root@basecm11 ~]# kubectl get pod -n kube-system -l name=amdgpu-dp-ds -o wide
NAME                                READY   STATUS    ... IP                                NODE   ...
amdgpu-device-plugin-daemonset-66jl7 1/1     Running   ... 172.29.112.135  gpu001 ...
amdgpu-device-plugin-daemonset-8mh9w 1/1     Running   ... 172.29.152.130  gpu002 ...
```

4.4.3 Including Head Nodes as part of the DaemonSet:

BCM taints head nodes, so that they do not run non-critical pods. The taint can be removed with the “-” operator to allow non-critical pods to run:

Example

```
kubectl taint nodes basecm11 node-role.kubernetes.io/control-plane-
```

However, a more specific exception can be configured in the DaemonSet itself.

Within the YAML file, the following existing tolerations definition has to be modified, from:

```
tolerations:
- key: CriticalAddonsOnly
  operator: Exists

to:

tolerations:
- key: node-role.kubernetes.io/control-plane
  effect: NoSchedule
  operator: Exists
```

The modified toleration tolerates this taint, and therefore has the device plugin run on such tainted nodes.

Verifying That AMD GPUs Are Recognized By Kubernetes

If Kubernetes is aware of the AMD GPUs for a node then several mentions of `amd.com/gpu` are displayed when running the `kubectl describe node` command for the node. The following session shows output for a node `gpu01`, ellipsized for clarity:

Example

```
[root@basecm11 ~]# kubectl describe node gpu01
Name:                gpu01
...
Capacity:
  amd.com/gpu:        3
  cpu:                64
  ephemeral-storage:  1813510Mi
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:              527954676Ki
  pods:               50
...
```

4.4.4 Running The DaemonSet Only On Specific Nodes

The AMD GPU device plugin, unlike the NVIDIA GPU device plugin Daemonset, is scheduled to run on each Kubernetes host. This means that it runs even if the host has no GPU.

This can be prevented with the following steps:

A LabelSet can be created using `cmsh`, and the nodes or categories that have GPUs are assigned within the `labelsets` mode:

Example

```
[root@basecm11 ~]# cmsh
[basecm11]% kubernetes
[basecm11->kubernetes[default]]% labelsets
[basecm11->kubernetes[default]->labelsets]% use nvidia
[basecm11->kubernetes[default]->labelsets[nvidia]]% ..    #but, we're using AMD GPUs, so let's go back up:
[basecm11->kubernetes[default]->labelsets]% add amd
[basecm11->kubernetes*[default*]->labelsets*[amd*]]% set labels nvidia.com/amd-gpu-accelerator=
[basecm11->kubernetes*[default*]->labelsets*[amd*]]% append categories gpu-nodes
[basecm11->kubernetes*[default*]->labelsets*[amd*]]% commit
```

This assigns the labels to the nodes with GPUs. This can be verified with:

Example

```
kubectl get nodes -l nvidia.com/amd-gpu-accelerator=
NAME      STATUS   ROLES    AGE   VERSION
gpu001    Ready    master   66m   v1.18.8
gpu002    Ready    master   66m   v1.18.8
...
```

The DaemonSet YAML can now be adjusted to only run the device plugin on nodes with this new label. This can be done by adding an affinity block after the tolerations block:

Example

```

tolerations:
- key: CriticalAddonsOnly # toleration may be different, if changes were made to it
  operator: Exists
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
            - key: 'nvidia.com/amd-gpu-accelerator'
              operator: Exists

```

This results in the device plugin pods being removed immediately from all nodes that do not have the label.

4.4.5 Running An Example Workload

An example workload can be run as described in the official AMD GPU Kubernetes device plugin documentation at:

<https://github.com/RadeonOpenCompute/k8s-device-plugin/tree/v1.16#example-workload>

Thus it should now be possible to run:

```
[root@basecm11 ~]# kubectl create -f https://raw.githubusercontent.com/RadeonOpenCompute/k8s-device-plugin/v1.16/example/pod/alexnet-gpu.yaml
```

The YAML requests only one GPU at the bottom of the YAML file:

```

apiVersion: v1
kind: Pod
metadata:
  name: alexnet-tf-gpu-pod
  labels:
    purpose: demo-tf-amdgpu
spec:
  containers:
    - name: alexnet-tf-gpu-container
      image: rocm/tensorflow:latest
      workingDir: /root
      env:
        - name: HIP_VISIBLE_DEVICES
          value: "0" # # 0,1,2,...,n for running on GPU and select the GPUs, -1 for running on CPU
      command: ["/bin/bash", "-c", "--"]
  args: ["python3 benchmarks/scripts/tf_cnn_benchmarks/tf_cnn_benchmarks.py --model=alexnet;\ntrap : TERM INT; sleep infinity & wait"]
  resources:
    limits:
      amd.com/gpu: 1 # requesting a GPU

```

Container creation might take a while due to the image size. Once scheduled, it prints out that it found exactly one GPU, and proceeds to run a TensorFlow workload.

Example

```
[root@basecm11 ~]# kubectl logs -f alexnet-tf-gpu-pod
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/compat/v2_compat.py:96:
```

```

disable_resource_variables (from tensorflow.python.ops.variable_scope) is deprecated and will be removed
in a future version.
Instructions for updating:
non-resource variables are not supported in the long term
2021-01-08 21:03:29.222293: I tensorflow/core/platform/profile_utils/cpu_utils.cc:104]
CPU Frequency: 2495445000 Hz
2021-01-08 21:03:29.222398: I tensorflow/compiler/xla/service/service.cc:168]
XLA service 0x39f62f0 initialized for platform Host (this does not guarantee that XLA will be used). Devices:
2021-01-08 21:03:29.222420: I tensorflow/compiler/xla/service/service.cc:176]
    StreamExecutor device (0): Host, Default Version
2021-01-08 21:03:29.223754: I tensorflow/stream_executor/platform/default/dso_loader.cc:48]
Successfully opened dynamic library libamdhip64.so
2021-01-08 21:03:31.635339: I tensorflow/compiler/xla/service/service.cc:168]
XLA service 0x3a40bb0 initialized for platform ROCm (this does not guarantee that XLA will be used). Devices:
2021-01-08 21:03:31.635363: I tensorflow/compiler/xla/service/service.cc:176]
    StreamExecutor device (0): Device 738c, AMDGPU ISA version: gfx908
2021-01-08 21:03:31.931125: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1734]
Found device 0 with properties:
    pciBusID: 0000:27:00.0 name: Device 738c      ROCm AMD GPU ISA: gfx908
    coreClock: 1.502GHz coreCount: 120 deviceMemorySize: 31.98GiB deviceMemoryBandwidth: 1.12TiB/s
...
TensorFlow: 2.3
Model:      alexnet
Dataset:    imagenet (synthetic)
Mode:       training
SingleSess: False
Batch size: 512 global
            512 per device
Num batches: 100
Num epochs: 0.04
Devices:    ['/gpu:0']
...

```

Had more GPUs been requested, more would have been made available to the container.

For comparison, a CPU version of the container is also available. The official instructions can be referred to for these, too.

4.5 Kubernetes Configuration Overlays

A list of configuration overlays can be seen from within configurationoverlay mode:

Example

```

[basecm11->configurationoverlay]% list

```

Name (key)	Priority	Nodes	Categories	Roles
kube-default-etcd	500	node001..node003		Etcd::Host
kube-default-master	510	node001..node003		generic::containerd, kube...
kube-default-worker	500	node004..node006	default	generic::containerd, kube...
kube-default-netq	501	node001		generic::netq

The NetQ configuration overlay is only normally displayed if NetQ is deployed.

Configuration overlays can be used to manage the Kubernetes services used with a particular configuration. For example, when managing the Kubernetes services used for a Kubernetes engine within an Auto Scale tracker (section 8.4.9 of the *Administrator Manual*).

4.6 Removing A Kubernetes Cluster

A Kubernetes cluster can be removed using `cm-kubernetes-setup` with the `--remove` and `--yes-i-really-mean-it` options. Also, if there more than one cluster present, then the cluster name must be specified using the `--cluster` parameter.

A removal run looks as follows (some output ellipsized):

Example

```
[root@basecm11 ~]# cm-kubernetes-setup --remove --cluster default --yes-i-really-mean-it
```

Connecting to CMDaemon

Executing 20 stages

Starting execution for 'Kubernetes Setup'

- kubernetes

- docker

Progress: 0

stage: kubernetes: Get Kube Cluster

Progress: 5

stage: kubernetes: Check Kube Cluster Exists

Progress: 10

stage: kubernetes: Find Installed Components

Progress: 15

stage: kubernetes: Find Files On Headnodes

Progress: 20

stage: kubernetes: Firewall Zone Close

Progress: 25

stage: kubernetes: Firewall Interface Close

Progress: 30

stage: kubernetes: Firewall Policy Close

Progress: 35

stage: kubernetes: Nginx Reverse Proxy Close

Progress: 40

stage: kubernetes: IP Ports Close

Progress: 60

stage: kubernetes: Remove Installed Components

Progress: 65

stage: kubernetes: Remove Files On Headnodes

Progress: 70

stage: kubernetes: Remove Etcd Spool

Progress: 80

stage: kubernetes: Set Reboot Required

You need to reboot 2 nodes to cleanup the network configuration

Progress: 85

stage: kubernetes: Collection Update Provisioners

Progress: 100

Took: 00:08 min.

Progress: 100/100

Finished execution for 'Kubernetes Setup', status: completed

Kubernetes Setup finished!

Using the `--remove` option removes the Kubernetes cluster configuration from BCM, unassigns Kubernetes-related roles—including the `EtcdHost` role—and removes Kubernetes health checks. The command does not remove packages that were installed with a `cm-kubernetes-setup` command before that.

After the disabling procedure has finished, the cluster has no Kubernetes configured and running.

4.7 Kubernetes Cluster Configuration Options

Kubernetes allows many Kubernetes clusters to be configured. These are separated sets of hosts with different certificates, users and other global settings.

When carrying out the Kubernetes setup run, a Kubernetes cluster name is asked, and a new object with the cluster settings is then added into the CMDaemon configuration. The administrator can change the settings of the cluster from within the kubernetes mode of cmsh, or within the Kubernetes Clusters options window of Base View accessible using the navigation path Containers > Kubernetes Clusters.

The cmsh equivalent looks like:

Example

```
[root@basecm11 ~]# cmsh
[basecm11]% kubernetes list
Name (key)
-----
default
[basecm11]% kubernetes use default
[basecm11->kubernetes[default]]% show
Parameter                                     Value
-----
Name                                           default
Revision
Etcd Cluster                                 kube-default
Pod Network                                  kube-default-pod
Pod Network Node Mask
Internal Network                             internalnet
KubeDNS IP                                   10.150.255.254
Kubernetes API server
Kubernetes API server proxy port             10443
App Groups                                   <1 in submode>
Label Sets                                   <3 in submode>
Notes
Version                                       1.27.10-150500.1.1
Trusted domains                             basecm11.openstacklocal,master,localhost,10.141.255.254
Module file template                         <1.01KiB>
Kubeadm init file                           <1.19KiB>
Service Network                             kube-default-service
Kubeadm CERT Key                            *****
Kube CA Cert                                *****
Kube CA Key                                 *****
Kubernetes users                             <0 in submode>
External                                     no
External Kubernetes Ingress server
External port                                0
Capi template                               no
Capi namespace                              default
Kubernetes management cluster
[basecm11->kubernetes[default]]%
```

The preceding kubernetes mode parameters are described in table 4.1:

Parameter	Description
App Groups	Groups of Kubernetes add-ons managed by CMDaemon.
CAPI namespace	Namespace where CAPI is deployed (optional).
CAPI template	Is this Kubernetes cluster configuration a template for CAPI workload clusters?
Etcd Cluster	The etcd cluster instance.
External	Is this Kubernetes cluster configuration for an external Kubernetes instance (e.g., cloud)?
External Kubernetes Ingress server	The Ingress endpoint for the external Kubernetes.
External port	Additional port used by external Kubernetes cluster (unused).
Internal Network	Network to back the internal communications.
Kube CA Cert	Path to PEM-encoded RSA or ECDSA certificate used for the CA
Kube CA Key	Path to PEM-encoded RSA or ECDSA private key used for the CA
KubeDNS IP	CoreDNS IP Address.
Kubeadm Cert Key	Key used to encrypt the control plane certificates in the kubeadm-certs Secret.
Kubeadm Init file	Contents of the init configuration file provided to Kubeadm to initialize or join nodes.
Kubernetes API Server	Kubernetes API server address (format: <code>https://<host>:<port number></code>).
Kubernetes API Server Proxy Port	Kubernetes API server proxy (NGINX LoadBalancer) port (default: 10443).

...continues

...continued

Parameter	Description
Kubernetes Management cluster	Relevant to CAPI only: the Kubernetes cluster managing this CAPI workload cluster (optional).
Kubernetes Users	Submode to manage users for this Kubernetes cluster (e.g.: whether or not to manage their Kube config files)
Label Sets	Submode to manage assignment of labels to Kubernetes nodes
Module file template	Template used for writing the Kubernetes TCL module file.
Name	The name, identifier, or label for the Kubernetes cluster.
Pod Network	Network where pod IP addresses are assigned from
Pod Network Node Mask	Corresponding subnet mask used to define the size of the address space allocated to each node for its pods
Service Network	Network from which the service cluster IP addresses are assigned, in IPv4 CIDR format. Must not overlap with any IP address ranges assigned to nodes for pods. Default: 172.29.0.0/16
Trusted Domains	Trusted domains to be included in Kubernetes-related certificates as Alternative Subject Names.
Version	Version of the Kubernetes cluster.

Table 4.1: kubernetes mode parameters

4.8 EtcdCluster

The EtcdCluster mode sets the global etcd cluster settings. It can be accessed using the top level etcd mode of cmsh.

Parameter	Description	Option to etcd
Name	etcd cluster name.	--initial-cluster-token
Election Timeout	Election timeout, in milliseconds.	--election-timeout

...continues

...continued

Parameter	Description	Option to etcd
Heart Beat Interval	Heart beat interval, in milliseconds.	--heartbeat-interval
CA	The certificate authority (CA) Certificate path for etcd, used to generate certificates for etcd.	--peer-trusted-ca-file
CA Key	The CA Key path for etcd, used to generate certificates for etcd.	
Member Certificate	The certificate path to use for etcd cluster members, signed with the etcd CA. The EtcdHost role can specify a member CA as well, and in that case it overwrites any value set here.	--peer-cert-file
Member Certificate Key	The key path to use for etcd cluster members, signed with the etcd CA. The EtcdHost role can specify a member CA as well, and in that case it overwrites any value set here.	--peer-key-file
Client CA	The CA used for client certificates. When set it is assumed client certificate and key are generated and signed with this CA by another party. etcd still expects the path to be correct for the client certificate and key.	--trusted-ca-file
Client Certificate	The client certificate, used by etcdctl, for example.	--cert-file
Client Certificate Key	The client certificate key, used by etcdctl for example.	--key-file

* Boolean (takes yes or no as a value)

Table 4.2: EtcdCluster role parameters and etcd options

4.9 Kubernetes Roles

Kubernetes roles include the following roles:

- EtcdHost (page 53)
- KubernetesApiServerProxy (page 54)
- Kubelet (page 55)
- generic::containerd (page 56)
- generic::netq (page 47)

When nodes are configured using Kubernetes roles, then settings in these roles may sometimes use the same values (pointer variables).

Example

```
[basecm11->configurationoverlay[kube-default-etcd]->roles[Etcd::Host]]% get etcdcluster
kube-default
```

and

```
[basecm11->kubernetes[default]]% get etcdcluster
kube-default
```

Pointer variables such as these have definitions that are shared across the roles, as indicated by the parameter description tables for the roles, and which are described in the following pages.

In `cmsh`, the roles can be assigned:

- for individual nodes using the `roles` submode of `device` mode
- for a category using the `roles` submode of a category
- for a configuration overlay using the `roles` submode of `configurationoverlay` mode

4.9.1 EtcdHost Role

The `EtcdHost` role is used to configure and manage the `etcd` service for a node.

The `etcd` service manages the `etcd` database, which is a hierarchical distributed key-value database. The database is used by Kubernetes to store its configurations. The `EtcdHost` role parameters are described in table 4.3:

Parameter	Description	Option to <code>etcd</code>
Member Name	The human-readable name for this <code>etcd</code> member (<code>\$hostname</code> is replaced by the node hostname)	<code>--name</code>
Spool	Path to the data directory (default: <code>/var/lib/etcd</code>)	<code>--data-dir</code>
Advertise Client URLs	List of client URLs for this member to advertise publicly (default: <code>http://\$hostname:5001</code>)	<code>--advertise-client-urls</code>
Advertise Peers URLs	List of peer URLs for this member to advertise to the rest of the cluster (default: <code>http://\$hostname:5002</code>)	<code>--initial-advertise-peer-urls</code>
Listen Client URLs	List of URLs to listen on for client traffic (default: <code>http://\$hostname:5001</code> , <code>http://127.0.0.1:2379</code>)	<code>--listen-client-urls</code>
Listen Peer URLs	List of URLs to listen on for peer traffic (default: <code>http://\$hostname:5002</code>)	<code>--listen-peer-urls</code>
Snapshot Count	Number of committed transactions that trigger a snapshot to disk (default: 5000)	<code>--snapshot-count</code>
Debug*	Drop the default log level to <code>DEBUG</code> for all subpackages? (default: no)	<code>--debug</code>
Member Certificate	<code>etcd</code> member certificate, signed with CA specified in the <code>etcd</code> cluster. Setting it overrules the value set in the <code>Etcd-Cluster</code> object (<code>etcd</code> mode, section 4.8). Default empty.	<code>--peer-cert-file</code>

...continues

...continued

Parameter	Description	Option to etcd
Member Certificate Key	etcd member certificate key, signed with CA specified in the etcd cluster. Setting it overrides the value set in the EtcdCluster object (etcd mode, section 4.8). Default empty.	--peer-key-file
Options	Additional parameters for the etcd daemon (empty by default)	

* Boolean (takes yes or no as a value)

Table 4.3: EtcdHost role parameters and etcd options

The etcd settings are updated by BCM in /cm/local/apps/etcd/current/etc/cm-etcd.conf.

4.9.2 The KubernetesAPIServerProxy Role

The KubernetesAPIServerProxy role sets up a proxy that provides the entry point for one or more instances of the Kubernetes API server. The proxy runs on every node of a Kubernetes cluster instance, including the head node.

If multiple Kubernetes control-plane nodes are present, then it enables HA for the Kubernetes control plane components (section 4.1).

This means that local port 10443 on all these nodes load balances over all control-plane node API servers.

Port 10443 can also be opened up for external access. The cm-kubernetes-setup wizard (section 4.2) in one of its screens (figure 4.4) prompts for this.

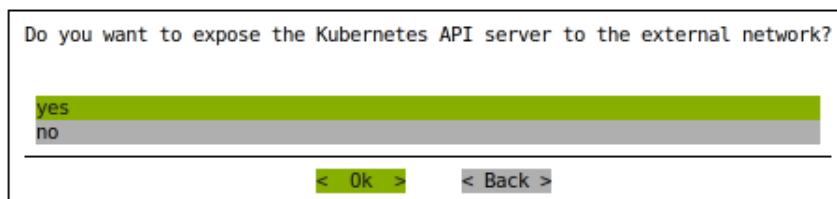


Figure 4.4: Wizard question

If the prompt is answered with no, then the firewall role on the head node can still be customized with cmsh later on:

Example

```

root@basecm11:~# cmsh
[basecm11]% device use master
[basecm11->device[basecm11]]% roles
[basecm11->device[basecm11]->roles]% use firewall
[basecm11->device[basecm11]->roles[firewall]]% openports
[basecm11->device[basecm11]->roles[firewall]->openports]% list
Index  Action    Network    Port    Destination  Protocol  Address    Description
-----
[basecm11->device[basecm11]->roles[firewall]->openports]% add ACCEPT net 10443 tcp fw
ok.
[basecm11->device*[basecm11*]->roles*[firewall*]->openports[0]]% commit
[basecm11->device[basecm11]->roles[firewall]->openports[0]]% list
Index  Action    Network    Port    Destination  Protocol  Address    Description

```

```

0      ACCEPT      net      10443      fw      TCP      0.0.0.0/0
[basecm11->device[basecm11]->roles[firewall]->openports[0]]%
Thu Oct 31 15:58:56 2024 [notice] basecm11: Service shorewall was restarted

```

4.9.3 The KubernetesIngressServerProxy Role

This role used to provide the “Ingress Proxy” (also known as “Ingress Server Proxy”) functionality, configuring the head-node NGINX to expose the Kubernetes ingress externally. In BCM 11.33.0 and later the role is almost a placeholder, because the head-node NGINX configuration is now offloaded to the Kubernetes setup itself, which generates and applies it using the Configure Gateway proxy to route traffic from head node(s) to Gateways item of cm-kubernetes-setup (section 4.17.6).

The role and the related Kubernetes cluster entity parameters (for example Ingress Proxy Enable, Ingress Proxy Listen Port, Ingress Proxy Backend Port) keep their original names for backward compatibility.

4.9.4 The Kubelet Role

The Kubelet role is used to configure and manage the kubelet service. BCM takes care of joining these nodes with `kubeadm join` when needed.

Control plane: The Kubelet role has a parameter `Control plane` that is set to the value `yes` for control plane nodes. In that case it also runs Kubernetes control plane services, such as:

- Kubernetes API server (`kube-apiserver`),
- Kubernetes scheduler (`kube-scheduler`),
- Kubernetes controller manager (`kube-controller-manager`),
- Kubernetes network proxy (`kube-proxy`),
- CoreDNS (`coredns`).

Workers: The Kubelet role has a parameter `worker` that is set to `yes` for worker nodes. In that case, the control plane pods will not be running on the node.

The Kubelet role parameters are described in table 4.4:

Parameter	Description
Kubernetes Cluster	The Kubernetes cluster instance (a pointer)
Control plane	Is Kubelet running services on this node, making it a control plane node?
Worker	Kubelet is a worker flag
Max Pods	Configuration to change max number of pods on Kubelet
Options	Submode that allows configuration of additional flags to containers specified in manifest <code>/etc/kubernetes/manifests</code>

* Boolean (takes `yes` or `no` as a value)

Table 4.4: Kubelet role parameters and kubelet options

Further details on the Kubelet service can be found at <https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/>.

The options Submode

```
[basecm11->configurationoverlay[kube-default-master]->roles[kubelet]]% help options
```

Name:

options - Manage kubelet container command options

Usage:

```
options [kubelet]
options [kubelet] <filename>
options [kubelet] <filename> set [name] key=value
options [kubelet] <filename> set [name] key
options [kubelet] <filename> clear [name] key
```

Examples:

```
options kube-apiserver.yaml set advertise-address=192.168.200.148
```

```
[basecm11->configurationoverlay[kube-default-master]->roles[kubelet]]% options
```

Filename	Container	Key	Value
kube-apiserver.yaml	kube-apiserver	cors-allowed-origins	https://name.run.ai
kube-apiserver.yaml	kube-apiserver	oidc-client-id	runai
kube-apiserver.yaml	kube-apiserver	oidc-issuer-url	https://app.run.ai/auth/realms/name
kube-apiserver.yaml	kube-apiserver	oidc-username-prefix	-

The full path to the files should be specified for the options command. For example, kube-apiserver.yaml may be specified as /etc/kubernetes/manifests/kube-apiserver.yaml.

4.9.5 Containerd Role

The Containerd role is used to configure and manage the containerd daemon. This is done through a generic role in BCM. Generic roles under Kubernetes are found under the cmsh path indicated by: cmsh > configurationoverlay[kube-object] > roles[generic::role]

Example

```
[basecm11->configurationoverlay[kube-default-master]->roles[generic::containerd]]% show
```

Parameter	Value
Name	generic::containerd
Revision	
Type	GenericRole
Add services	yes
Services	containerd
Configurations	<2 in submode>
Environments	<1 in submode>
Exclude list snippets	<2 in submode>
Data node	no

The role has several submodes:

- The configurations submode: This contains various configuration drop-ins.

Configuration drop-ins that BCM may manage in the generic::containerd role are:

- NVIDIA Container Toolkit configuration.
- Docker registry certs directory configuration.
- Docker Hub credentials configuration.
- Harbor Registry Mirror configuration.

- CRI cgroup configuration.
- CNI bin dir configuration.
- CRI registry configuration.
- CRI custom sandbox image configuration.

The drop-ins that are actually available depend on choices made by the cluster administrator during deployment.

- The `environment` submode: This has optional environment variables that are made available to the configuration files and that are to be used as templates.
- The `excludelistsnippets` submode: This has files related to containerd that need to be excluded, such as the most obvious directory `/var/lib/containerd`.

4.10 Security Model

The Kubernetes security model allows authentication using a certificate authority (CA), with the user and daemon certificates signed by a Kubernetes CA. The Kubernetes CA should not be confused with the BCM CA.

BCM lets `kubeadm` create a CA specifically for issuing all Kubernetes-related certificates. The certificates are put into `/etc/kubernetes/pki/<kubeclusterlabel>/` by default.

In Kubernetes terminology a user is a unique identity accessing the Kubernetes API server. The user may be a human or an automated process. For example an admin or a developer are human users, but kubelet represents an infrastructure user. Both types of users are authorized and authenticated in the same way against the API server.

Kubernetes uses client certificates, tokens, or HTTP basic authentication methods to authenticate users for API calls. BCM configures client certificate usage by default. The authentication is performed by the API server which validates the user certificate using the common name part of the certificate subject.

In Kubernetes, authorization happens as a separate step from authentication. Authorization applies to all HTTP accesses on the main (secure) API server port. BCM by default enables RBAC (Role-Based Access Control) combined with Node Authorization. The authorization check for any request thus takes the common name and/or organization part of the certificate subject to determine which roles the user or service has associated. Roles carry a certain set of privileges for resources within Kubernetes.

4.10.1 Kyverno

BCM has support for the Kyverno policy engine (<https://kyverno.io/>). If Kyverno is installed, then Kubernetes Permissions Manager (section 4.15) creates policy manifests packed as a Helm chart for every user added to Kubernetes using `cm-kubernetes-setup`. In addition, a `kyverno-policy` chart is installed in enforce mode to implement Pod Security Standards (<https://kyverno.io/policies/>). During installation, some exclusions are added to the policies automatically to make chosen features of Kubernetes cluster work.

For every created user the following defaults are applied:

- The user has an associated service account with the same name
- A `<username>-restricted` namespace is created. So, for a user `john` the namespace is `john-restricted`.
- An `edit` cluster role is bound to the service account in `<username>-restricted` namespace. The user is allowed to create pods, services, configmaps, etc. in the namespace
- The user is allowed to list nodes in the cluster

- Kyverno policies are applied to the resources in `<username>-restricted` namespace or to pod created or updated by the associated user
 - If `hostPath` is not the home directory of the user (of the format `/home/<username>`) then the creation of the resource is denied
 - The UID and GID of the running process are set to the same value as the UID and GID of the PAM user

Modifications from the defaults are:

- If the `Allow any UID process in pods` checkbox is ticked, or if the `--allow-all-uids` argument is specified, then the UID and GID of the running process becomes the user's UID and GID only if the `hostPath` volume is specified. Otherwise it can be set to any UID and GID.
- Cluster roles can be set not only to
 - `edit`

but also to

- `view`
- `admin`
- `cluster-admin`

More details on these roles can be found at:

<https://kubernetes.io/docs/reference/access-authn-authz/rbac/#user-facing-roles>.

- In addition, the user can be given access to custom CRDs, such as Postgres Operator, Jupyter Operator or Google Spark Operator

4.10.2 PodSecurityPolicy

PodSecurityPolicy (PSP) was available in older versions of BCM. However, PSP support was completely removed in Kubernetes v1.25. BCM therefore now uses Kyverno (section 4.10.1) for an equivalent functionality.

4.11 Addition Of New Kubernetes Users

BCM users can use Kubernetes by making them Kubernetes users. This means having Kubernetes configuration and access set up for them. This can be carried out using the `cm-kubernetes-setup` TUI utility, and choosing the `Add user` option (figure 4.3). The utility then prompts for

- a Kubernetes cluster
- a user name
- a namespace that the privileges are to be assigned to
- a role for the user, with choices provided from:
 - `cluster-admin`: cluster-wide administrator
 - `admin`: administrator
 - `edit`: regular user
 - `view`: read-only user
- a switch if the user is allowed to run as any user, including root, inside pods

- a comma-separated list of paths that the user is able to mount to pods
- the UIDs and GIDs for user processes in pods
- a list of the Kubernetes operators that a user can use

Based on the input, a YAML for the Kubernetes Permission Manager is generated. This in turn, creates a Helm chart with all the required roles, role bindings, and Kyverno rules.

Creation of the user also triggers CMDaemon to create certificate and configuration files in the `~/.kube` directory

4.11.1 Adding Users Non-Interactively With `cm-kubernetes-setup`

The `cm-kubernetes-setup` CLI wizard provides the following options:

```
cm-kubernetes-setup -h
usage: Kubernetes Setup cm-kubernetes-setup
       [-c <config_file>]
       [--list-versions]
       [--list-operators-versions]
       [--cluster CLUSTER_NAME]
       [--skip-docker] [--skip-reboot]
       [--skip-image-update]
       [--skip-kube-version-check]
       [--skip-dns-configuration-check]
       [--skip-package-manager-update-check]
       [--skip-install-repos]
       [--adv-configuration-bcm-nvidia-packages]
       [--add-user USERNAME_ADD]
       [--list-users]
       [--get-user GET_USER]
       [--modify-user USERNAME_MODIFY]
       [--remove-user USERNAME_REMOVE]
       [--namespace NAMESPACE]
       [--add-to-namespace]
       [--remove-from-namespace]
       [--role edit,admin,view,cluster-admin]
       [--runas-uid RUNAS_UID]
       [--runas-gids RUNAS_GIDS]
       [--user-paths USER_PATHS]
       [--allow-all-uids]
       [--operators OPERATORS]
       [--backup-permissions FILE]
       [--restore-permissions FILE]
       [--list-operators]
       [--update-addons]
       [--patch-kube-prometheus-stack]
       [--remove]
       [--yes-i-really-mean-it] [--pull]
       [--images IMAGES] [--nodes NODES]
       [--node-selector NODE_SELECTOR]
       [--pull-registry-server PULL_REGISTRY_SERVER]
       [--pull-registry-username PULL_REGISTRY_USERNAME]
       [--pull-registry-email PULL_REGISTRY_EMAIL]
       [--pull-registry-password PULL_REGISTRY_PASSWORD]
       [--docker-io-username DOCKER_IO_USERNAME]
       [--docker-io-password DOCKER_IO_PASSWORD]
```

```

[--kubeadm-image-repository KUBEADM_IMAGE_REPOSITORY]
[--containerd-sandbox-image CONTAINERD_SANDBOX_IMAGE]
[--enable-ovn] [--latest-versions]
[--skip-ovn-prerequisites-checks]
[--skip-netq-prerequisites-checks]
[--provision-node PROVISION_NODE]
[--airgap-registry AIRGAP_REGISTRY]
[--airgap-registry-as-helm | --airgap-helm-repo AIRGAP_HELM_REPO]
[-v] [--store-name-aliases]
[--no-distro-checks] [--json]
[--output-remote-execution-runner]
[--on-error-action debug,remotedebug,undo,abort]
[--skip-packages]
[--min-reboot-timeout <reboot_timeout_seconds>]
[--allow-running-from-secondary]
[--dev] [-h]

```

...

The user has to be a user that exists on the cluster already and available using PAM.

If `--add-to-namespace` is specified, then the namespace has to exist on the Kubernetes cluster already.

Example

```
cm-kubernetes-setup --add-user john
```

The preceding example creates a user `john` for the default `john-restricted` namespace. It also assigns the `edit` role, and gives permission to run processes in the pod with the current UID/GIDs of the user. The ability to mount `~/john` as a `hostPath` is also provided.

A way to assign any of the default Kubernetes user-facing roles is also provided by using `--role` key, as documented at <https://kubernetes.io/docs/reference/access-authn-authz/rbac/#user-facing-roles>

The possible roles are: `view`, `edit`, `admin`, and `cluster-admin`.

Example

```
cm-kubernetes-setup --add-user john --role view
```

The preceding example creates a user `john` with `view` privileges only, for the default `john-restricted` namespace.

Example

```
cm-kubernetes-setup --add-user john --user-paths /home/john,/scratch --allow-all-uids \
--operators cm-jupyter-kernel-operator
```

The preceding example creates a user `john` with the following privileges:

- `edit` privileges
- able to mount `/home/john` and `/scratch` as `hostPath` volume, when the process runs with UID/GIDs taken from the PAM subsystem on the moment of creation
- able to run as any user, including `root` (attempts to mount any `hostPath` volume will be rejected)
- access to the Jupyter Kernel Operator, i.e. with access to the resource kind: `CmKubernetesOperatorPermissionsJupyterKernel`

4.12 Getting Information And Modifying Existing Kubernetes Users

It is possible to edit user properties and permissions. `cm-kubernetes-setup` provides 2 ways of doing it: interactively or using CLI options.

Modifying users can be done interactively by choosing `Modify User` in the `cm-kubernetes-setup` main menu. Guidance is then given on choosing the cluster, users, and on modifying permissions.

Modifying users can also be done using CLI options, by specifying the `--modify-user` argument:

Example

```
cm-kubernetes-setup --add-user john --user-paths /home/john \  
  --allow-all-uids --operators cm-jupyter-kernel-operator
```

The default `john-restricted` namespace is created as user `john` is added, along with the settings specified by the other options. The following example then adds permission to mount the `/scratch` hostPath into pods, and gives access to the Postgres Operator:

```
cm-kubernetes-setup --modify-user john --user-paths /home/john,/scratch --namespace john-restricted\  
  --allow-all-uids --operators cm-jupyter-kernel-operator,cm-kubernetes-postgresql-operator
```

Information about existing users can be found with:

Example

```
cm-kubernetes-setup --list-users
```

Permission for user `john` to operate in the `dev` namespace can be added with:

Example

```
kubectl create namespace dev  
cm-kubernetes-setup --modify-user john --namespace dev --add-to-namespace
```

Permission for the user `john` to operate in the `dev` namespace can be revoked with:

Example

```
kubectl create namespace dev  
cm-kubernetes-setup --modify-user john --namespace dev --remove-from-namespace
```

4.13 List Of Resources Defined For Users

These resources are rendered by the Permission Manager Operator, and can therefore be found inside Kubernetes.

The Role Bindings Deployed For Every User By Default

By default, the role bindings deployed for the user `john` created in the preceding section are:

- `ClusterRole/john-nodes` (in namespace `john-restricted`)
- `ClusterRoleBinding/john-nodes` (in namespace `john-restricted`)

User `john` is given read-only rights for the `Nodes` resource (for `kubectl get nodes`).

The Secure Namespace Related Resources

The secure namespace for user john is:

- Namespace/john-restricted

The service account used by john:

- ServiceAccount/john (in namespace john-restricted)

This is found referenced, for example, in john's \$HOME/.kube/config.

The PodSecurityPolicy that defines the user can run non-privileged pods, and use only ports above 1024, and so on:

- PodSecurityPolicy/john-restricted (in namespace john-restricted)

More details on this can be found in section 4.10, page 58. This policy will only do something as soon as the PodSecurityPolicy Admission Controller is enabled in the API server.

A PodSecurityPolicy that defines the user can run as root as well, but *without* hostPath volumes:

- PodSecurityPolicy/john-restricted-root (in namespace john-restricted)

To give the aforementioned privileges to john's secure namespace, so that john can run workloads, execute `kubectl get all`, and more:

- Role/john-restricted (in namespace john-restricted)
- RoleBinding/john-restricted (in namespace john-restricted)

The RoleBinding assigns it to the user john and ServiceAccount account for john. The upstream documentation at <https://kubernetes.io/docs/reference/access-authn-authz/service-accounts-admin> has more details on this.

The user john can be given the ability to use the PodSecurityPolicy defined earlier in his secure namespace, but also in other namespaces:

- ClusterRole/john-psp (in namespace john-restricted)
- ClusterRoleBinding/john-psp (in namespace john-restricted)

The same ability can be given for the second root but no hostPath PodSecurityPolicy:

- ClusterRole/john-psp-root (in namespace john-restricted)
- ClusterRoleBinding/john-psp-root (in namespace john-restricted)

If the Kyverno engine is installed then several policies are added:

- `clusterpolicies.kyverno.io/john-*-drop-privs-w-hostpath*`: Policy to modify pod manifests to run process with specified UID/GID
- `clusterpolicies.kyverno.io/john-*-limit-hostpath-vols`: Policy to deny pods if hostPath volumes does not match specified paths

The full content of all the documents created for the user can be viewed by checking the generated Helm manifest:

Example

```
helm get manifest -n cm-permissions john-XXXXXX
```

4.14 Kyverno

Kyverno (<https://kyverno.io/>) is a policy engine designed for Kubernetes. With Kyverno, policies are managed as Kubernetes resources, and no new language is required to write policies. This allows the use of familiar tools such as `kubectl`, `git`, and `kustomize` to manage policies. Kyverno policies can validate, mutate, and generate Kubernetes resources, as well as ensure container registry image supply chain security.

4.14.1 Kyverno Installation

Kyverno engine and Kyverno policy Helm charts can be installed as a part of `cm-kubernetes-setup`:

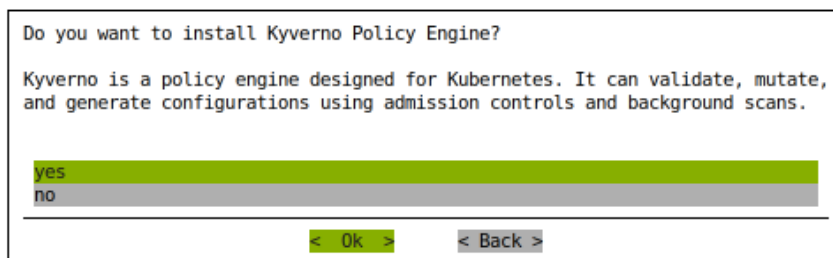


Figure 4.5: Choosing Kyverno installation

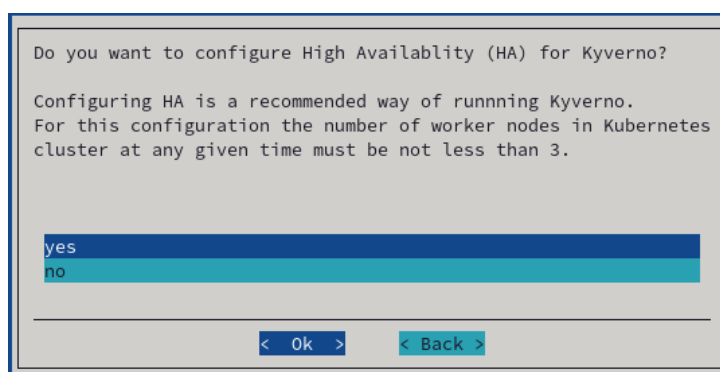


Figure 4.6: Kyverno high availability setup

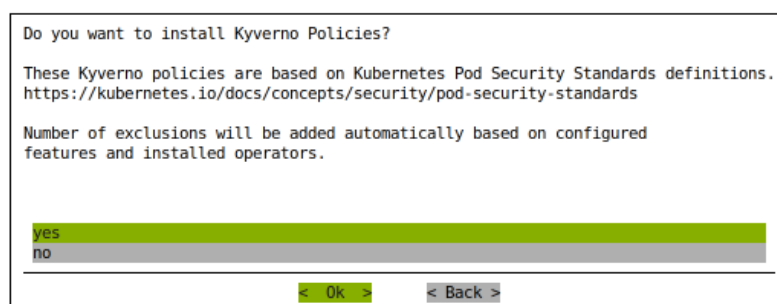


Figure 4.7: Kyverno policy setup

The installation adds 2 Helm charts in the namespace 'kyverno':

```
[root@basecm11 ~]# module load kubernetes/
[root@basecm11 ~]# helm list -n kyverno
```

NAME	NAMESPACE	STATUS	...	CHART	APP VERSION
kyverno	kyverno	deployed	...	kyverno-v2.5.2	v1.7.2

```
kyverno-policies kyverno    deployed ... kyverno-policies-v2.5.2 v1.7.2
```

If the HA option is chosen, then the replica count value is set to 3.

```
[root@basecm11 ~]# helm get values -n kyverno kyverno
USER-SUPPLIED VALUES:
replicaCount: 3
```

This means that at any given time Kubernetes scheduler tries to run 3 pods at the same time:

```
[root@basecm11 ~]# kubectl get pods -n kyverno
NAME                                READY   STATUS    RESTARTS   AGE
kyverno-5bfb99b9c9-ddmmw           1/1     Running   0           1h
kyverno-5bfb99b9c9-hgfsc           1/1     Running   0           1h
kyverno-5bfb99b9c9-n67rv           1/1     Running   0           1h
```

4.14.2 Kyverno Policies

It is also recommended to install Kyverno policies in order to enforce Pod Security Standards <https://kyverno.io/policies/>. BCM configures Kyverno policies in 'enforce' mode, adding service namespaces during cm-kubernetes-setup as exclusions (figure 4.8).

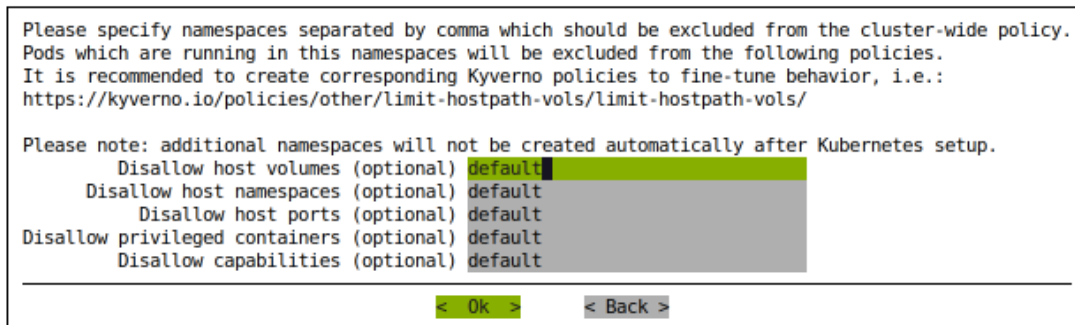


Figure 4.8: Kyverno policy: Defining the excluded namespaces

The list of namespaces to be excluded from particular policies depend on features selected during installation:

```
[root@basecm11 ~]# helm get values -n kyverno kyverno-policies
USER-SUPPLIED VALUES:
validationFailureAction: enforce
policyExclude:
  disallow-host-namespaces:
    any:
      - resources:
          kinds:
            - Pod
          namespaces:
            - default
            - prometheus
  disallow-host-path:
    any:
      - resources:
          kinds:
            - Pod
          namespaces:
            - default
```

```

- local-path-storage
- '*-restricted'
- prometheus
- kube-system
- gpu-operator
disallow-host-ports:
  any:
  - resources:
      kinds:
      - Pod
      namespaces:
      - default
      - prometheus

```

In the preceding output, all namespaces that match the wildcard `*-restricted` are excluded from the policy named `'disallow-host-path'` (<https://kyverno.io/policies/pod-security/baseline/disallow-host-path/disallow-host-path/>). This means that, without additional restrictions, all pods in the user namespaces can mount any host path from an underlying node.

To prevent that Kubernetes Permission Manager creates a Kyverno Cluster Policy for every newly-created user, and restricts the `hostPath` to only the home directory of the user:

```

[root@basecm11 ~]# kubectl get clusterpolicies.kyverno.io | grep john
john-n730sr0-drop-privs-w-hostpath                false      enforce    true
john-n730sr0-drop-privs-w-hostpath-containers      false      enforce    true
john-n730sr0-drop-privs-w-hostpath-initcontainers  false      enforce    true
john-n730sr0-limit-hostpath-vols                   false      enforce    true

```

4.15 Kubernetes Permission Manager

The Kubernetes permission manager is a custom operator based on Helm. It helps to manage user and system account permissions, roles, role bindings and pod security policies. The operator itself is packed and distributed as a Helm chart, so it can be installed during Kubernetes cluster creation using the `cm-kubernetes-setup` TUI. The Helm chart for the operator is located in `/cm/shared/apps/kubernetes-permissions-manager/current/helm`. The output to the following command shows if it is installed:

```

[root@basecm11 ~]# module load kubernetes/
[root@basecm11 ~]# helm list -n cm
NAME                NAMESPACE STATUS ... CHART                                     APP VERSION
local-path-provisioner cm          deployed ... cm-kubernetes-local-path-provisioner-0.0.20 0.0.20
permissions-manager cm          deployed ... cm-kubernetes-permissions-manager-0.0.1     0.0.1

```

The Helm chart of the operator includes custom resource definitions (CRD), and makes it possible for the administrator to manage resources using the `kubectl` tool:

Example

```

[root@basecm11 ~]# cat > permissions.yaml<<EOF
apiVersion: charts.brightcomputing.com/v1alpha1
kind: CmKubernetesPermissionUser
metadata:
  labels:
    namespace: cmsupport-restricted
    username: cmsupport
    name: cmsupport-c7tk7ft
    namespace: cm-permissions

```

```
spec:
  allow_all_uids: false
  allowPrivilegeEscalation: false
  allowPrivileged: false
  create_namespace: true
  create_service_account: true
  gids:
  - 1000
  namespace: cmsupport-restricted
  psp_spec_override:
  role: edit
  uid: 1000
  user_paths:
  - /home/cmsupport
  username: cmsupport
EOF
[root@basecm11 ~]# kubectl apply -f permissions.yaml
cmkubernetespermissionuser.charts.brightcomputing.com/cmsupport-c7tk7ft created
[root@basecm11 ~]# kubectl get cmkubernetespermissionusers -A
NAMESPACE      NAME                      AGE
cm-permissions  cmsupport-c7tk7ft        22s
```

At the time of writing of this section (December 2021), the permission manager handles these 6 CRDs:

1. cmkubernetespermissionusers to manage user access to generic resources of the cluster, such as pods, services, secrets, configmaps, etc.
2. cmkubernetesoperatorpermissionsjupyterkernels to manage access to the Jupyter Kernels.
3. cmkubernetesoperatorpermissionspostgresqls to manage access to the Postgres operator (<https://github.com/zalando/postgres-operator>).
4. cmkubernetesoperatorpermissionsmpis to manage access to the MPI operator (<https://github.com/kubeflow/mpi-operator>).
5. cmkubernetesoperatorpermissionsnims to manage access to the NIM operator (<https://docs.nvidia.com/nim-operator/>).
6. cmkubernetesoperatorpermissionssparks to manage access to the Google Spark operator (<https://github.com/GoogleCloudPlatform/spark-on-k8s-operator>).

Providing access to third party operators is necessary if pod security policy is enabled. This is because, by default, not only does the user have no access to CRDs, but also the service accounts of the third party operators have no access to the user namespace.

The following example is a YAML document that provides access to the Jupyter Kernel Operator:

```
apiVersion: charts.brightcomputing.com/v1alpha1
kind: CmKubernetesOperatorPermissionsJupyterKernel
metadata:
  labels:
    namespace: cmsupport-restricted
    username: cmsupport
  name: cmsupport-unz4wlf
  namespace: cm-permissions
spec:
  namespace: cmsupport-restricted
  username: cmsupport
```

Every installed CRD document triggers the Kubernetes permission operator to create a corresponding Helm chart:

```
# helm get values -n cm-permissions cmsupport-unz4wlf
USER-SUPPLIED VALUES:
namespace: cmsupport-restricted
username: cmsupport

# helm get manifest -n cm-permissions cmsupport-unz4wlf
---
# Source: cm-kubernetes-operator-permissions-jupyter-kernel/templates/user-permissions.yaml
# Bind policy to service user
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cmsupport-unz4wlf-cmsupport-psp
  labels:
    helm.sh/chart: cmsupport-unz4wlf
    app.kubernetes.io/name: cm-kubernetes-operator-permissions-jupyter-kernel
    app.kubernetes.io/instance: cmsupport-unz4wlf
    app.kubernetes.io/version: "0.0.1"
    app.kubernetes.io/managed-by: Helm
subjects:
- kind: ServiceAccount
  name: default
  namespace: cmsupport-restricted
roleRef:
  kind: ClusterRole
  name: cmsupport-psp
  apiGroup: rbac.authorization.k8s.io
...
```

It is also possible to customize the resulting Helm chart by specifying additional values to specify a section of the CRD. The 6 Helm chart CRD directories can be seen with:

Example

```
root@basecm11:~# kubectl exec -it -n cmkpm-system $(kubectl get pods -n cmkpm-system -l \
control-plane=controller-manager -o name) -c manager -- ls -la helm-charts
.
..
cm-kubernetes-operator-permissions-jupyter-kernel
cm-kubernetes-operator-permissions-mpi
cm-kubernetes-operator-permissions-nim
cm-kubernetes-operator-permissions-postgresql
cm-kubernetes-operator-permissions-spark
cm-kubernetes-permission-user
```

For example, available values for the Jupyter kernel can be checked using the following command:

Example

```
kubectl exec -it -n cmkpm-system \
$(kubectl get pods -n cmkpm-system -l control-plane=controller-manager -o name) \
-c manager -- \
cat /opt/helm/helm-charts/cm-kubernetes-operator-permissions-jupyter-kernel/values.yaml
```

Similarly, tunables for the generic user permissions of the user are available using:

Example

```
kubect1 exec -it -n cmkpm-system \
  $(kubect1 get pods -n cmkpm-system -l control-plane=controller-manager -o name) \
  -c manager -- cat /opt/helm/helm-charts/cm-kubernetes-permission-user/values.yaml
```

The output should be similar to:

```
username: "" # name of the user
create_service_account: true # whether to create kubernetes serviceaccount for the user
role: edit # user role
user_paths: [] # hostPath user able to mount to pods
uid: -1 # UID to run process inside pods
gids: [-1] # list of the GIDs for process inside pods
namespace: "" # namespace to give user permissions to
create_namespace: true # create or not the namespace
allowPrivilegeEscalation: false
allowPrivileged: false
allow_all_uids: true # allow or not to run process as any user including root
                    # if 'true' and process is run not with user's UID, then
                    # all hostPath volumes are denied
psp_spec_override: # custom PSP definition for the user
```

4.16 Providing Access To External Users

External access to a BCM Kubernetes cluster falls into two distinct paths, each using its own port on the head node(s) and described in their own subsections:

- access to the Kubernetes control plane itself, for administrative tooling such as `kubect1` and `helm`, using the Kubernetes API server proxy port (10443 by default), and covered in section 4.16.1.
- access to applications and services running on the cluster, using the Gateway proxy port (443 by default), and covered in section 4.16.2.

A schematic overview of the data and control paths is shown in figure 4.9:

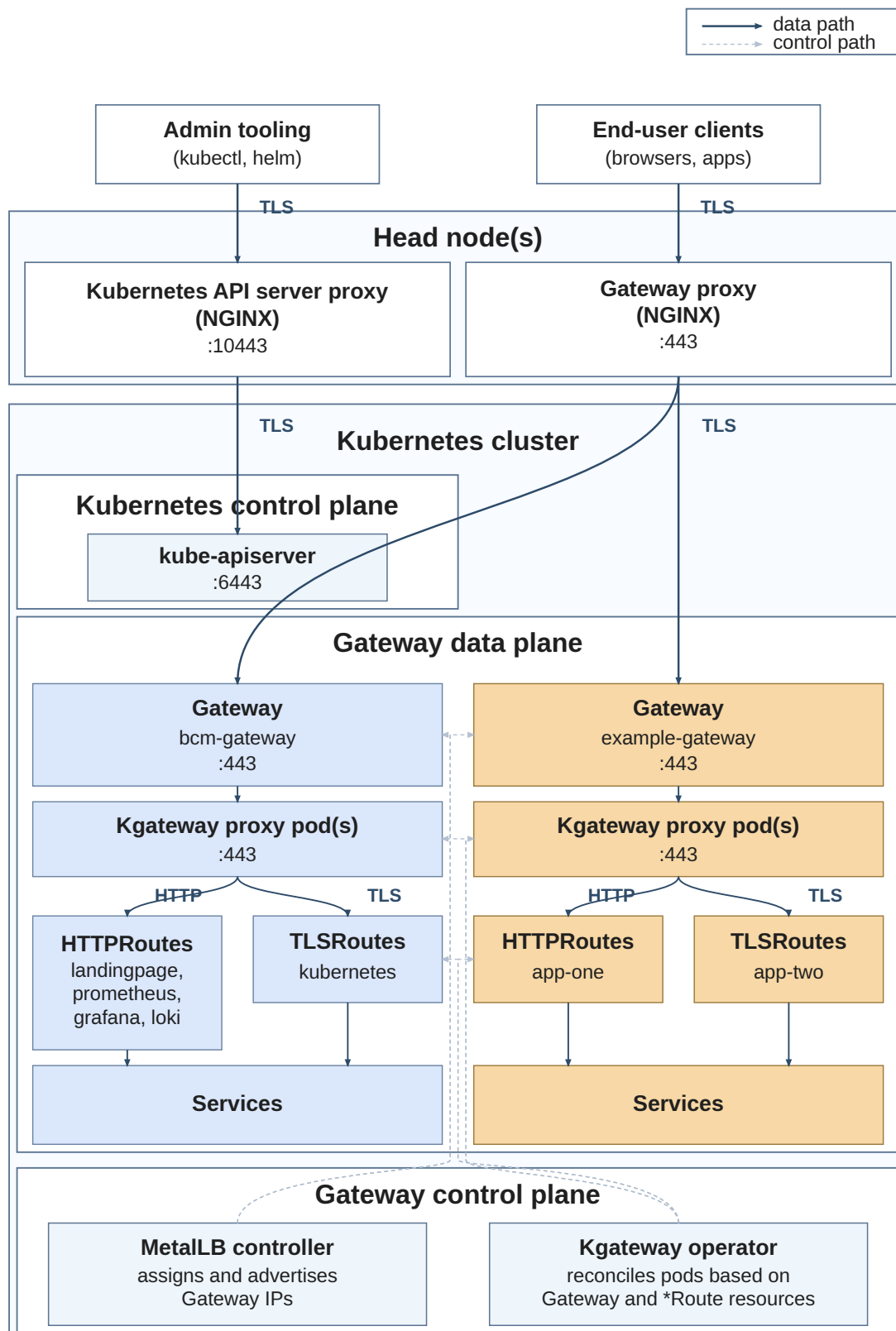


Figure 4.9: Data and control paths between the Kubernetes gateways and proxies

If exposing the Kubernetes API server to the external network is selected during setup with `cm-kubernetes-setup`, then both the Kubernetes API server proxy port and the Gateway proxy port

are opened on the Shorewall service that runs on the head node. Exposure to the external network is enabled by default.

4.16.1 External Access To The Kubernetes Control Plane (`kubectl`, `helm`)

For administrative tooling such as `kubectl` and `helm` to reach the Kubernetes API server from outside the cluster, an entry in the company/internal DNS server should resolve the external FQDN of the cluster to the head node (or to one of the nodes where Kubernetes is running). The external FQDN is set during the Kubernetes cluster setup, and is the first item in the list of trusted domains configured on the Kubernetes cluster entity. It can be retrieved with `cmsh` as follows:

Example

```
[basecm11->kubernetes[default]]% get trusteddomains
basecm11.example.com
kubernetes
kubernetes.default
kubernetes.default.svc
master
localhost
```

In the preceding example, the FQDN of the cluster is `basecm11.example.com`. The cluster administrator managing their own cluster will have another FQDN, and not this FQDN.

The port on which the Kubernetes API server is exposed externally is the Kubernetes API server proxy port. It can be retrieved from the Kubernetes cluster entity as follows:

```
[basecm11->kubernetes[default]]% get kubernetesapiserverproxyport
10443
```

This port is opened automatically by Shorewall on the head node when the exposing the Kubernetes API server to the external network option is selected during `cm-kubernetes-setup`, as mentioned earlier.

4.16.2 External Access To Kubernetes-hosted Applications And Services

Applications and services running in Kubernetes are exposed externally through the Gateway components, which are fully covered in section 4.17. For any service that needs its own subdomain (for example, `myapp.<cluster-fqdn>`, used as a hostname in a Gateway API route resource—such as an `HTTPRoute` or `TLSRoute`—attached to a Kubernetes Gateway), a corresponding DNS entry under the external FQDN must be added.

4.17 Kubernetes Gateway

Starting with BCM 11.33.0, support for the Kubernetes Gateway API (currently version 1.5.x) is introduced, provided by Kgateway and MetalLB. External traffic is exposed using the Gateway proxy on its listening port (443 by default) on the head node(s), which forwards that traffic to the configured Gateways served by Kgateway (the `bcm-gateway` Gateway by default, and any additional Gateways that have been added).

The 3 BCM Kubernetes Gateway components are:

1. *Kgateway* (<https://kgateway.dev/>): the controller that fulfills the Kubernetes Gateway API (<https://gateway-api.sigs.k8s.io/>) specification on the cluster, reconciling Gateway, `HTTPRoute`, `TLSRoute`, and related resources, and spinning up Kgateway proxy pods that actually handle traffic for each Gateway.
2. *MetalLB* (<https://metallb.universe.tf/>): provides a LoadBalancer Service IP address for each Gateway in the same way that a cloud provider's load balancer would do for Kubernetes running in a managed cloud, and makes that IP address routable from outside the cluster.

3. The *Gateway proxy*: an NGINX server running on the head node(s) and edge directors, provided by the `KubernetesIngressServerProxy` role (section 4.9.3). It serves as the single entry point on the head node(s) for external clients, and forwards incoming traffic on its listening port to the appropriate Gateway based on the TLS Server Name Indication (SNI) hostname.

4.17.1 Gateway API

The Kubernetes Gateway API is a Kubernetes-defined API specification, not a controller. The Gateway, HTTPRoute, TLSRoute, and related resources follow this specification, and are intended to be implementation-agnostic. BCM provides Gateway API 1.5.x. Kgateway is the implementation that BCM uses with the specification, and the resources that the applications and operators define according to the specification are, in most cases, portable across any conformant Gateway API implementation. The official documentation for the Kubernetes Gateway API explains the model in detail.

4.17.2 Kgateway

With Kgateway the control plane and the data plane are split:

- The Kgateway operator itself is the *control plane*. It watches the Gateway API resources (Gateway, HTTPRoute, etc.) and reconciles them.
- The *data plane* is created on demand, per Gateway, in the form of one or more Kgateway proxy pods that actually handle incoming traffic.

A consequence of this split is that only installing the Kgateway operator does not, by itself, expose any traffic: a Gateway (with a corresponding GatewayClass) must also be created so that the Kgateway operator can spin up the proxy pod(s) for it.

4.17.3 The bcm-gateway Gateway

All BCM-shipped applications and operators expose their Kubernetes-hosted services through the configured Gateways. The configured gateways are the `bcm-gateway` Gateway by default, and any additional Gateways that have been added (section 4.17.5) using Gateway API HTTPRoute or TLSRoute resources attached to those Gateways. The configured routes and the Gateways themselves can be inspected with `kubectl`:

```
root@basecm11:~# kubectl get gateway -A
NAMESPACE      NAME          CLASS          ADDRESS          PROGRAMMED    AGE
bcm-gateway    bcm-gateway   kgateway-ha2   10.141.255.253   True         9s

root@basecm11:~# kubectl get httproute -A
NAMESPACE      NAME          HOSTNAMES      AGE
bcm-gateway    landingpage                  17s
loki           loki                         16s
prometheus     grafana                      16s
prometheus     prometheus                   16s

root@basecm11:~# kubectl get tlsroute -A
NAMESPACE      NAME          AGE
default        kubernetes    21s
```

The full YAML of an individual route can be inspected with `kubectl`. For example, the BCM-shipped HTTPRoute `landingpage` attached to the default `bcm-gateway` Gateway can be viewed with:

```
root@basecm11:~# kubectl get httproute landingpage -n bcm-gateway -o yaml
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
```

```

creationTimestamp: "2026-04-21T10:17:37Z"
generation: 1
name: landingpage
namespace: bcm-gateway
spec:
  parentRefs:
  - group: gateway.networking.k8s.io
    kind: Gateway
    name: bcm-gateway
    namespace: bcm-gateway
  rules:
  - backendRefs:
    - group: gateway.kgateway.dev
      kind: Backend
      name: landingpage
      weight: 1
    matches:
    - path:
        type: PathPrefix
        value: /
status:
...
```

The `spec.parentRefs` entry binds this HTTPRoute to the default `bcm-gateway` Gateway (named `bcm-gateway` in the `bcm-gateway` namespace), and the `spec.rules` entry forwards traffic matching the / path prefix to a Kgateway Backend named `landingpage`, which is configured as a static Backend pointing to `master:80`, where the BCM landing page service runs on the head node(s) and is served by `apache2` or `httpd`. Multiple backends and path-based routing can be configured through additional HTTPRoute resources attached to the same `bcm-gateway` Gateway.

Attaching Application Routes To The `bcm-gateway` Gateway

Applications can attach their own HTTPRoute or TLSRoute resources to the `bcm-gateway` Gateway alongside the BCM-shipped routes shown in the preceding text. This is the simplest way to expose an application externally, as it avoids having to allocate and configure an extra IP address. For an application to be allowed to attach routes, its namespace must first be added under the relevant listener's `allowedRoutes.namespaces.selector.matchExpressions` on the `bcm-gateway` Gateway.

4.17.4 MetalLB

Gateway IP Address Assignment And Routability

Each Gateway is exposed using a LoadBalancer Service that requires its own IP address, which is unique across all Gateways, and also unique across different Kubernetes clusters. In a bare-metal BCM deployment there is no cloud load balancer to allocate such IP addresses, so this responsibility is delegated to MetalLB (<https://metallb.universe.tf/>), which is installed alongside Kgateway. MetalLB allocates an IP address from a configured pool to each Gateway's LoadBalancer Service and makes that IP address routable from outside the cluster.

MetalLB supports two modes for advertising the assigned IP addresses:

- L2 mode (Layer 2 / ARP): No special network equipment configuration is required, which makes L2 mode the simplest to deploy. It provides basic failover-style HA, but not true load-balancing. The main limitation is that traffic can only reach the Gateway through nodes that are in the same L2 network as the client. When the Gateway proxy is enabled, this means that at least one Kubernetes node must be in the same L2 network as the head node(s), so incoming traffic can be forwarded through the head node(s) to reach Kubernetes services. Further details can be found at <https://metallb.universe.tf/concepts/layer2/>.

- In L2 mode, `cm-kubernetes-setup` suggests sensible default IP addresses on a best-effort basis by filtering out many known used or assigned IP addresses and IP ranges. These suggestions may still conflict with the environment into which the setup is carried out. The customer must ensure that each Gateway uses a unique, routable, IP address.
- BGP mode: The MetalLB speakers establish BGP sessions with the network’s BGP-capable routers and advertise the Service IP addresses as routes. This provides a higher level of HA together with true load balancing, since traffic is distributed across nodes. This requires a BGP peer to be available upfront—typically a network router, switch, or similar device.
Further details can be found at <https://metallb.universe.tf/concepts/bgp/>.

During the initial `cm-kubernetes-setup` session, the wizard prompts for the MetalLB mode to use for Kubernetes Gateways (figure 4.10).

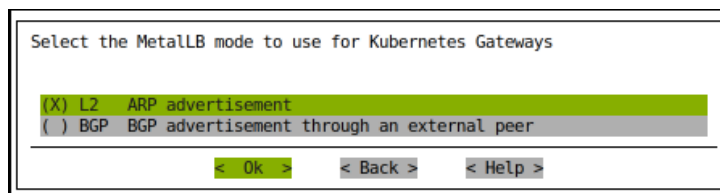


Figure 4.10: Wizard prompt: select the MetalLB mode for Kubernetes Gateways

If BGP mode is selected, the wizard then prompts for the specific BGP settings (figure 4.11).

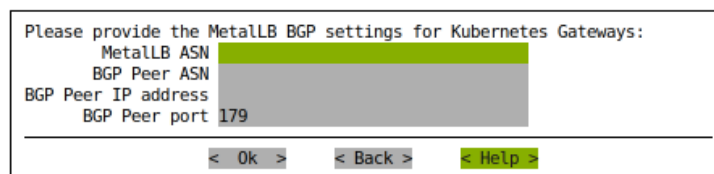


Figure 4.11: Wizard prompt: MetalLB BGP settings for Kubernetes Gateways

The MetalLB mode is configured *per Kubernetes cluster* and therefore applies to all Gateways in that cluster. The number of Kgateway proxy pods, by contrast, is determined *per Gateway* using the chosen GatewayClass (section 4.17.5), so different applications within the same cluster can scale their data plane independently. The same Gateways work with either MetalLB mode.

L2 Mode: Restricting Which Nodes Participate In The Gateway Data Plane

The L2-mode limitation described in the preceding text means that some care is needed in deciding which Kubernetes nodes participate in the Gateway data plane. BCM manages this automatically when the Gateway proxy is enabled, but the cluster administrator should still validate whether other BCM compute nodes need to reach the Gateway from a different L2 network.

BCM uses a dedicated labelset in `cmsh`, `exclude-lb`, backed by the standard upstream Kubernetes node label `node.kubernetes.io/exclude-from-external-load-balancers` (<https://kubernetes.io/docs/reference/labels-annotations-taints/>). When this label is present on a node:

- MetalLB does not use that node to advertise any LoadBalancer Service IP address.
- Kgateway does not schedule any of its proxy pods on that node.

The label thus keeps the Gateway IP address advertisement and the corresponding proxy pods co-located on the same set of nodes, which avoids the failure mode where an IP address is announced from a node that does not actually host the proxy pod.

`cm-kubernetes-setup` manages the `exclude-lb` labelset as follows:

- In L2 mode with the Gateway proxy enabled, the labelset is applied to Kubernetes nodes that are not on a network shared with head nodes. This ensures that MetalLB advertisement and Kgateway proxy pod scheduling occur only on the other nodes that do share networks with the head node(s).
- In L2 mode with the Gateway proxy disabled, the labelset is not applied to any node. It is then up to the cluster administrator to optionally add nodes to the labelset based on node needs.
- In BGP mode no nodes are added to the labelset either. This means that all Kubernetes nodes can participate in the Gateway data plane, and no further action is needed from the cluster administrator.

Nodes can be added to or removed from the `exclude-lb` labelset in `cmsh`, for example:

Example

```
[root@basecm11 ~]# cmsh
[basecm11]% kubernetes use default
[basecm11->kubernetes[default]]% labelsets
[basecm11->kubernetes[default]->labelsets]% use exclude-lb
[basecm11->kubernetes[default]->labelsets[exclude-lb]]% append nodes node001
[basecm11->kubernetes*[default*]->labelsets*[exclude-lb*]]% commit
[basecm11->kubernetes[default]->labelsets[exclude-lb]]% removefrom nodes node001
[basecm11->kubernetes*[default*]->labelsets*[exclude-lb*]]% commit
```

4.17.5 Creating Additional Gateways

Applications that need to be exposed externally can either attach their routes to the default `bcm-gateway` Gateway (section 4.17.3), or create an additional Gateway as described in the following subsections.

Provided Gateway Classes

A Gateway references a `GatewayClass` that determines how the Gateway is implemented (which controller fulfills it, and with how many proxy pods). BCM provisions three Kgateway-backed `GatewayClass` resources, differing only in the number of Kgateway proxy pods that back the corresponding Gateway:

- `kgateway`: 1 Kgateway proxy pod (suitable for non-HA deployments)
- `kgateway-ha2`: 2 Kgateway proxy pods
- `kgateway-ha3`: 3 Kgateway proxy pods

The default `bcm-gateway` Gateway is provisioned against the `kgateway-ha2` class, giving it two Kgateway proxy pods. Additional Gateways can reference any of the provided classes.

The available classes can be listed with:

```
bash$ kubectl get gatewayclass
NAME            CONTROLLER          ACCEPTED  AGE
kgateway        kgateway.dev/kgateway  True      104m
kgateway-ha2    kgateway.dev/kgateway  True      104m
kgateway-ha3    kgateway.dev/kgateway  True      104m
```

MetalLB Resources

When an additional Gateway is created, the MetalLB resources needed to be created as well, in order to assign and advertise an IP address for that Gateway. This subsection describes the manifests that are applied, mirroring what `cm-kubernetes-setup` creates for the default `bcm-gateway` Gateway (section 4.17.4).

For each additional Gateway, two MetalLB resources are required, both in the `metal1b-system` namespace:

- An `IPAddressPool` that holds the IP address (or range of IP addresses) that MetalLB can assign to the Gateway, and—optionally—restricts the pool to that one Gateway using `serviceAllocation`.
- An *advertisement* resource that exposes the assigned IP address to the network. The value of `kind` sets the cluster's MetalLB mode (section 4.17.4):
 - `L2Advertisement` for L2 mode
 - `BGPAdvertisement` for BGP mode.

In BGP mode the `BGPPeer` that BCM provisions in the `metallb-system` namespace is cluster-wide, and does not need to be duplicated for additional Gateways. The new `BGPAdvertisement` reuses it automatically.

The manifests that follow are a valid starting point, but additional Gateways can deviate from this shape. For example:

- a wider CIDR or a range in `spec.addresses` can be used, which means that the pool can serve multiple `LoadBalancer` Services
- omitting `serviceAllocation` makes the pool generally available,
- services can be selected on different labels or namespaces.

The upstream MetalLB documentation has the full schema of each resource.

Detecting the MetalLB mode: Whether BGP mode is available for an additional Gateway can be checked by looking for the BCM-managed `BGPPeer` named `gateway` in the `metallb-system` namespace:

Example

```
root@basecm11:~# kubectl -n metallb-system get bgppeer gateway
NAME      AGE
gateway   2d
```

If the `BGPPeer` exists, then the cluster was set up with BGP mode and an additional Gateway can reuse that peer by applying a `BGPAdvertisement`. BGP mode is generally the better choice when it is available—the trade-offs between the two modes are described in section 4.17.4. L2 mode is always usable as well, regardless of whether the `BGPPeer` exists, by applying an `L2Advertisement` instead.

Naming convention: The `gateway-` name prefix in the `metallb-system` namespace is reserved for MetalLB resources that `cm-kubernetes-setup` creates and manages itself, so the names of MetalLB resources for an additional Gateway must not start with that prefix. A common convention is to reuse the Gateway's own name (or `<gateway-namespace>-<gateway-name>`) as the MetalLB resource name.

IP address selection: The IP address, or range chosen for the additional Gateway must:

- be unique across all Gateways (and across other Kubernetes clusters that may share the same network), as described in section 4.17.4
- not overlap with any other `IPAddressPool` in the cluster, including the ones created by `cm-kubernetes-setup`
- be routable from the intended clients, taking into account the L2-mode restriction on which nodes participate in the Gateway data plane (section 4.17.4)

IPAddressPool: Apply an IPAddressPool containing the chosen IP address as a /32 entry, and use serviceAllocation to bind the pool to the specific additional Gateway using the gateway.networking.k8s.io/gateway-name label that Kgateway sets on the LoadBalancer Service:

Example

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: <gateway-name>
  namespace: metallb-system
spec:
  addresses:
    - <gateway-ip>/32
  autoAssign: true
  serviceAllocation:
    namespaces:
      - <gateway-namespace>
  serviceSelectors:
    - matchExpressions:
      - key: gateway.networking.k8s.io/gateway-name
        operator: In
        values:
          - <gateway-name>
```

The serviceAllocation stanza is what guarantees that MetalLB only uses this pool for the additional Gateway, even when other pools also have free addresses with autoAssign: true.

L2Advertisement (L2 mode): In L2 mode, apply an L2Advertisement that references the IPAddressPool:

Example

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: <gateway-name>
  namespace: metallb-system
spec:
  ipAddressPools:
    - <gateway-name>
```

BGPAdvertisement (BGP mode): In BGP mode, apply a BGPAdvertisement instead, again referencing the IPAddressPool. No additional BGPPeer is needed—the BGPPeer that cm-kubernetes-setup provisioned for the default bcm-gateway Gateway handles the BGP session with the upstream router:

Example

```
apiVersion: metallb.io/v1beta1
kind: BGPAdvertisement
metadata:
  name: <gateway-name>
  namespace: metallb-system
spec:
  ipAddressPools:
    - <gateway-name>
```

4.17.6 Gateway Proxy

The Gateway proxy is implemented by NGINX on the head node(s) and edge directors. Its configuration is generated and applied by `cm-kubernetes-setup`, which also opens the Gateway proxy port (443 by default) in the firewall. The related parameters (such as `Ingress Proxy Enable` and `Ingress Proxy Listen Port`) live on the Kubernetes cluster entity in `cmsh`, and the corresponding head-node role is the `KubernetesIngressServerProxy` role (section 4.9.3).

Architecture

The Gateway proxy listens on the head node(s) on its listening port and uses TLS Server Name Indication (SNI) to decide where to forward each incoming connection:

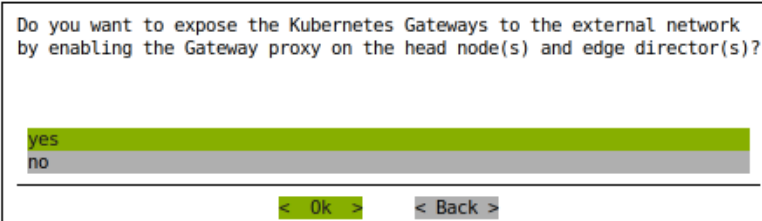
- Connections that present a SNI hostname matching a hostname declared by one of the configured Gateways' listeners are forwarded to that specific Gateway's LoadBalancer Service IP address.
- Connections without a SNI hostname (for example, when a client connects to the head node by IP address), and connections with a SNI hostname that does not match any of the configured Gateway listeners, are forwarded to the default `bcm-gateway` Gateway.

On the Gateway side, the matching listener must be on port 443. The backend port that the Gateway proxy connects to on each Gateway's LoadBalancer Service is exposed by the `Ingress Proxy Backend Port` parameter on the Kubernetes cluster entity in `cmsh`, which defaults to 443. This value is used internally by the Gateway proxy and is not intended to be customized.

Configuration During Setup

During the initial `cm-kubernetes-setup` session, two wizard screens drive the Gateway proxy configuration:

- A yes/no prompt (figure 4.12) asking whether to expose the Kubernetes Gateways to the external network by enabling the Gateway proxy on the head node(s) and edge director(s). Selecting yes (recommended) enables it. Selecting no skips it; the Gateway proxy can still be configured later using the menu item described in section 4.17.6, or by following the manual steps in section 4.17.6.



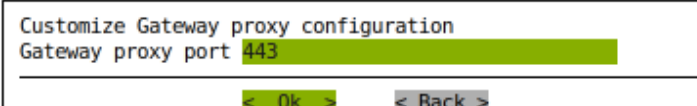
Do you want to expose the Kubernetes Gateways to the external network by enabling the Gateway proxy on the head node(s) and edge director(s)?

yes
no

< Ok > < Back >

Figure 4.12: Wizard prompt: expose the Kubernetes Gateways by enabling the Gateway proxy?

- A follow-up `Customize Gateway proxy configuration` screen (figure 4.13) asking on which TCP port the Gateway proxy should listen. The default is 443, which matches the standard HTTPS port and is what the rest of this documentation assumes.



Customize Gateway proxy configuration

Gateway proxy port 443

< Ok > < Back >

Figure 4.13: Wizard prompt: Customize Gateway proxy configuration (Gateway proxy port)

The Configure Gateway proxy Menu Item

After the initial setup, the same head-node NGINX configuration can be regenerated and re-applied at any time using the Configure Gateway proxy to route traffic from head node(s) to Gateways item, found under the Gateway configuration menu of cm-kubernetes-setup. Selecting that item inspects the Gateways currently present on the cluster, collects their LoadBalancer Service IP addresses assigned by MetallB and the hostnames declared on their listeners, and then refreshes the head-node NGINX Gateway proxy configuration accordingly so that the SNI-based routing described in section 4.17.6 reflects the current set of Gateways. This item is therefore re-run whenever a new Gateway is added or an existing one is changed.

Manual Configuration

If the wizard prompt about exposing the Kubernetes Gateways by enabling the Gateway proxy was answered with no during the initial cm-kubernetes-setup session, the Gateway proxy can still be enabled afterwards by carrying out the following steps.

Step 1: ensure the Gateway proxy port is not already in use: If the configured Gateway proxy port is already in use (for example, by the Gateway proxy of another Kubernetes cluster, or by the BCM landing page when served on that port using apache2 or httpd), then that port must be changed.

If the Gateway proxy port is the default 443, then for Ubuntu-based systems this means commenting out Listen 443 in /etc/apache2/ports.conf. For RHEL-based systems that line is located at /etc/httpd/conf.d/ssl.conf. If a different Gateway proxy port was configured, then the corresponding Listen directive for that port must be adjusted instead.

The web service should then be restarted (apache2 for Ubuntu, httpd for RHEL and Rocky).

Step 2: enable the Gateway proxy in the Kubernetes cluster entity: On the Kubernetes cluster entity in cmsh, Ingress Proxy Enable should be set to yes, and Ingress Proxy Listen Port should be set to the desired Gateway proxy port.

The parameter gateway_proxy_enabled must also be set. It is an additional entry for BCM 11 versions later than point release version 11.33.0, required as the Gateway proxy replaces the legacy Ingress.

The example below uses the default port 443.

Example

```
root@basecm11:~# cmsh
[basecm11]% kubernetes use default
[basecm11->kubernetes[default]]% set -e "gateway_proxy_enabled" true
[basecm11->kubernetes*[default*]]% set ingressproxyenable yes
[basecm11->kubernetes*[default*]]% set ingressproxylistenport 443
[basecm11->kubernetes*[default*]]% set ingressproxybackendport 443
[basecm11->kubernetes*[default*]]% commit
[basecm11->kubernetes[default]]%
```

Step 3: assign the role to the head nodes: This can be done as follows in cmsh, and should be repeated for the passive head node in case of an HA BCM cluster.

Example

```
root@basecm11:~# cmsh
[basecm11]% device use master
[basecm11->device[basecm11]]% roles
[basecm11->device[basecm11]->roles]% list
Name (key)
-----
```

```

Kubernetes::ApiServerProxy
[overlay:kube-default-worker] generic::containerd
[overlay:kube-default-worker] kubelet
backup
boot
firewall
headnode
monitoring
nginx
provisioning
storage
[basecm11->device[basecm11]->roles]% assign Kubernetes::IngressServerProxy
Creating new role 'Kubernetes::IngressServerProxy' ... succeeded.
[basecm11->device*[basecm11*]->roles*[Kubernetes::IngressServerProxy*]]% show
Parameter                               Value
-----
Name                                    Kubernetes::IngressServerProxy
Revision
Type                                    KubernetesIngressServerProxyRole
Add services                            yes
Worker connections                      1024
Send file                               yes
TCP no push                             yes
TCP no delay                            yes
Keep alive timeout                      1m 5s
Types hash max size                     2048
Listen Port                             443
Kubernetes Ingress Server Port          0
[basecm11->device*[basecm11*]->roles*[Kubernetes::IngressServerProxy*]]% commit
[basecm11->device[basecm11]->roles[Kubernetes::IngressServerProxy]]%
Thu Oct 31 16:29:51 2024 [notice] basecm11: Service nginx was reloaded

```

As the preceding session shows, assigning the role immediately affects the `/etc/nginx/nginx.conf` file and the `nginx` service, but the actual Gateway proxy routing configuration is generated and applied in Step 4.

Step 4: run the Configure Gateway proxy menu item: This is described in section 4.17.6.

Step 5: open the Gateway proxy port in the firewall: The port in the default firewall, Shorewall (section 7.2 of the *Installation Manual*), is managed using the `firewall` role in `cmsh`. Only the Gateway proxy port needs to be opened, in addition to the Kubernetes API server proxy port. The example below uses the default Gateway proxy port 443; if a different port was configured in Step 2, use that value instead.

```

root@basecm11:~# cmsh
[basecm11]% device use master
[basecm11->device[basecm11]]% roles
[basecm11->device[basecm11]->roles]% use firewall
[basecm11->device[basecm11]->roles[firewall]]% openports
[basecm11->device[basecm11]->roles[firewall]->openports]% list -v
Index  Action   Network  Port  Count  Destination  Protocol  Address  Description
-----
0      ACCEPT   net      10443  1      fw           TCP       0.0.0.0/0  Kubernetes 'default'
                                                API Server proxy
[basecm11->device[basecm11]->roles[firewall]->openports]% add ACCEPT net 443 tcp fw
ok.

```

```
[basecm11->device*[basecm11*]->roles*[firewall*]->openports*[1]]% commit
[basecm11->device[basecm11]->roles[firewall]->openports[1]]% list -v
Index  Action    Network    Port    Count    Destination    Protocol    Address    Description
-----
0      ACCEPT    net        10443   1        fw             TCP         0.0.0.0/0   Kubernetes 'default'
                                           API Server proxy
1      ACCEPT    net        443     1        fw             TCP         0.0.0.0/0
[basecm11->device[basecm11]->roles[firewall]->openports[1]]%
Thu Oct 31 16:35:19 2024 [notice] basecm11: Service shorewall was restarted
```

The pre-existing 10443 entry shown above with the description Kubernetes 'default' API Server proxy is automatically opened by the KubernetesApiServerProxy role when exposing the Kubernetes API server to the external network was selected during cm-kubernetes-setup. In the example above, the newly-added port 443 entry has no description because it was added manually; it can be commented or otherwise tagged later if needed. As shown above, the change is reflected immediately in the /etc/shorewall/rules file and shorewall service.

4.17.7 Configuring The Gateway TLS Certificate

Kubernetes applications that are exposed externally on BCM use HTTPS using the Gateway proxy port, served by the configured Gateways (the bcm-gateway Gateway by default, and any additional Gateways that have been added). The TLS certificate that the bcm-gateway Gateway presents is the certificate that needs to be configured for external access.

For the bcm-gateway Gateway, it is assumed that

- the cluster administrator has provided the cluster with a domain name, for example my-cluster.nvidia.local
- there is a DNS entry present that makes this domain name resolve to the cluster IP address
- a matching bcm-gateway.key and bcm-gateway.crt TLS certificate key pair file has been provided

Gateway Self-signed Certificate

For testing purposes it is possible to create a self-signed certificate with a private certificate authority (CA) owned by the cluster administrator, and configure the bcm-gateway Gateway to present it. That is, using that instead of a trusted (third-party) CA. However the effort needed to configure this means that the procedure is not generally recommended. Reasons to avoid using a self-signed certificate are:

- each user needs the self-signed CA certificate pairs working on their system between the appropriate applications
- each user also needs to override a warning about the untrusted certificate by accepting the self-signed certificate on browsers such as Chrome.
- it is very easy to run into subtle issues later on that are hard to uncover

That said, appendix A has instructions on setting up a self-signed certificate for testing purposes. The steps that are needed to also make the certificates trusted on any given subsystem or application such as the web browser, are outside the scope of the manuals and BCM support. It is assumed that this has already been set up by the organization.

Gateway Trusted CA Certificate

In case no self-signed certificate is to be used—for example if there is already a trusted CA certificate to use for this purpose—then another Kubernetes wizard session can be run from the active head node to configure the bcm-gateway Gateway TLS certificate. The cluster administrator carries out the following procedure:

- uses SSH to get into the active head node
- copies certificate pair files, `bcm-gateway.crt` and `bcm-gateway.key`, over to the head node (for example: to the `/root` directory)
- invokes `cm-kubernetes-setup`, opens the Gateway configuration menu, and then chooses (figure 4.14): `Configure TLS Certificate for Gateway`
- answers `yes` when the wizard prompts:
Do you want to configure an existing, properly signed certificate pair?
- sets paths to the certificate pair files; the wizard then stores the certificate pair as the Kubernetes TLS Secret `gateway-bcm-gateway-cert` in the `bcm-gateway` namespace, and references that Secret from the `bcm-gateway` Gateway listener
- confirms that the HTTPS certificate is working correctly, using the SAN check (described shortly)

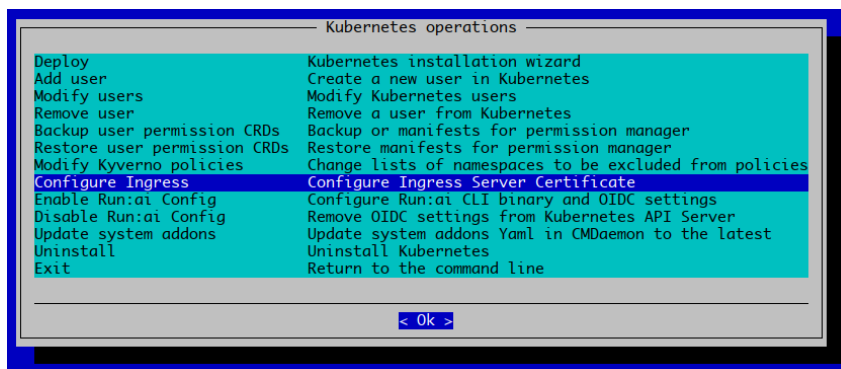


Figure 4.14: The `Configure TLS Certificate for Gateway` item under the Gateway configuration menu

The SAN check: The following command can be used to inspect the SAN (Subject Alternative Name) part of the certificate currently presented by the `bcm-gateway` Gateway using the Gateway proxy port:

Example

```
[root@basecm11 ~]# openssl s_client -connect localhost:443 < /dev/null 2>/dev/null | \
openssl x509 -noout -text | grep -A1 'Subject Alternative Name'
X509v3 Subject Alternative Name:
DNS:my-cluster.nvidia.local, DNS:*.apps.my-cluster.nvidia.local, DNS:master.cm.cluster
```

4.17.8 Migrating An Existing Kubernetes Cluster From Ingress To Gateway

For BCM clusters that were originally deployed with the legacy Ingress NGINX Controller (BCM versions before 11.33.0), the following procedure migrates the cluster to Kubernetes Gateway without recreating the cluster.

Prerequisites: Before starting the migration:

- All applications and operators that currently rely on Kubernetes Ingress resources should either already support, or be ready to be migrated to, the Kubernetes Gateway API.
- If the legacy Ingress proxy is currently used for external access, a maintenance window should be planned, since that external traffic will be interrupted during the migration.

Step 1: deploy the Gateway components: The `cm-kubernetes-setup` utility is started on the active head node. From its Gateway configuration menu, the `Install Gateway components and replacements for legacy BCM Ingresses` item is selected. The exact actions performed by this step are summarized by the wizard confirmation dialog.

This step is intentionally non-destructive in the sense that it does *not* uninstall the Ingress NGINX Controller, and it does *not* delete any existing Ingress resources. However, it does reconfigure the legacy Ingress proxy on the head node(s) to behave as the Gateway proxy, so external traffic through the legacy Ingress path no longer works after this step.

Step 2: adjust external load balancer configuration: If any external load balancer currently sends traffic to the legacy Ingress proxy ports (30080 and/or 30443, with 30443 the default HTTPS port), it should be updated to use the Gateway proxy port instead. This ensures that external traffic reaches the head-node Gateway proxy rather than the legacy Ingress proxy path while the Gateway-based path is being validated.

Step 3: validate traffic for the BCM-shipped services: With the Gateway components running, the BCM-shipped services that were previously fronted by Ingress—the landing page, the Kubernetes API at the configured external FQDN, Prometheus, Grafana, and Loki, where applicable—should be verified through the new `bcm-gateway` Gateway using the Gateway proxy port.

The Gateway and its HTTPRoute/TLSRoute resources can be inspected with:

- `kubect1 get gateway -A`
- `kubect1 get httproute -A`
- `kubect1 get tlsroute -A`

Step 4: migrate non-BCM Ingress resources to Gateway API: Any non-BCM-provided application that still relies on Kubernetes Ingress resources is now migrated. For each such Ingress:

- an equivalent HTTPRoute (or TLSRoute for TLS pass-through) is authored, targeting the same back-end Service
- the new route is attached to a Gateway—either to the default `bcm-gateway` Gateway (section 4.17.3), or to a dedicated Gateway created in the application’s namespace (section 4.17.5)
- the original Ingress resource is left in place for the time being, so traffic keeps flowing through the legacy path while the new path is being validated

Step 5: refresh the Gateway proxy for non-BCM Gateways: After non-BCM Gateways have been created on the cluster

- `cm-kubernetes-setup` is started again
- the Gateway configuration menu is opened
- the `Configure Gateway proxy to route traffic from head node(s) to Gateways` item is selected (section 4.17.6).

This refreshes the head-node NGINX Gateway proxy so that traffic for the new Gateways is forwarded from the head node(s) to their respective LoadBalancer Service IP addresses assigned by MetalLB, using the SNI-based routing described in section 4.17.6. Connections that do not match any non-BCM Gateway listener (including connections without a SNI hostname) continue to be forwarded to the default `bcm-gateway` Gateway.

Step 6: validate traffic for the non-BCM services: Each migrated application is verified through its HTTPRoute or TLSRoute on the bcm-gateway Gateway (or on its dedicated Gateway). At this point, both the BCM-shipped services and the non-BCM services are reachable through the Gateway-based path, while the legacy Ingress NGINX Controller NodePorts are still in place as a fallback.

Step 7: remove the legacy Ingress NGINX controller: Once all traffic has been confirmed on the Gateway-based path, the legacy controller is removed. To carry this out, cm-kubernetes-setup is started again, the Gateway configuration menu is opened and the Remove Ingress NGINX Controller (ingress-nginx) and legacy BCM Ingresses item is selected.

The exact actions performed by the removal are summarized by the wizard confirmation dialog.

This step does *not* delete any Ingress resources that were not provided by BCM. Any non-BCM Ingress resources that have not been removed yet remain present on the cluster.

Step 8: complete the cleanup: Once the previous steps have all completed successfully, any non-BCM Ingress resources that are still present on the cluster can be safely deleted by the cluster administrator. For example:

```
root@basecm11:~# kubectl get ingress -A
root@basecm11:~# kubectl delete ingress <name> -n <namespace>
```

After this, the cluster is fully migrated to the Kubernetes Gateway API, and the legacy Ingress NGINX Controller is no longer in use on the cluster.

4.18 Networking Model

Kubernetes expects all pods to have unique IP addresses, which are reachable from within the cluster. This can be implemented in several ways, including adding pod network interfaces to a network bridge created on each host, or by using 3rd party tools to manage pod virtual networks.

With BCM the default pod network provider is Calico (<https://www.projectcalico.org/>). Calico uses the Border Gateway Protocol (BGP) to distribute routes for every Kubernetes pod. This allows the Kubernetes cluster to be integrated without the need for overlays (IP-in-IP). Calico is particularly suitable for large Kubernetes deployments on bare metal, or in private clouds. This is because for larger deployments the performance and complexity costs of overlay networks can become significant.

4.19 Kubernetes Monitoring

When cm-kubernetes-setup is run, it configures the following Kubernetes-related health checks:

1. KubernetesChildNode: checks if all the expected agents and services are up and running for active nodes
2. KubernetesComponentsStatus: checks if all the daemons running on a node are healthy
3. KubernetesNodesStatus: checks if Kubernetes nodes have a status of Ready
4. KubernetesPodsStatus: checks if all the pods are in one of these states: Running, Succeeded, or Pending

4.20 Local Path Storage Class

For storage, instead of creating Kubernetes PersistentVolumes every time, a modern and practical way is to use the StorageClass feature.

Further documentation on StorageClass is available at:

- <http://blog.kubernetes.io/2016/10/dynamic-provisioning-and-storage-in-kubernetes.html>

- <https://kubernetes.io/docs/concepts/storage/persistent-volumes/#storageclasses>

As a part of initial installation it is possible to choose a Local Path Storage class to utilize the shared storage mounted on every node of the Kubernetes cluster. Possible options include any POSIX shared filesystems, such as NFS, BeeGFS, LustreFS, etc.

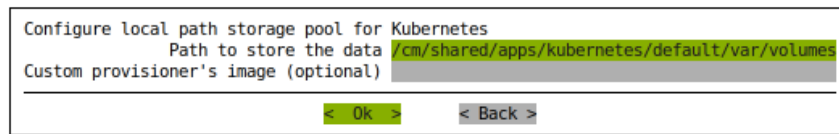


Figure 4.15: Kubernetes setup TUI session: local storage configuration

During setup, the installation wizard asks for a path for where Kubernetes physical volumes (PV) will be physically located. This path should be located on a shared filesystem accessible from all nodes.

After installation, the storage class can be seen to be available with:

```
# kubectl get storageclasses.storage.k8s.io
```

NAME	PROVISIONER	RECLAIMPOLICY	VOLUMEBINDINGMODE	ALLOWVOLUMEEXPANSION	AGE
local-path (default)	rancher.io/local-path	Delete	Immediate	false	1h

Users of the cluster can then freely create persistent volume claims (PVC) resources and use them in running pods.

4.21 Integration With Harbor

In order to spawn pods that use images from the Harbor registry, a secret must first be created with the credentials:

```
[root@basecm11 ~]# kubectl create secret docker-registry myregistrykey \
--docker-server=node001:9443 --docker-username=admin --docker-password=Harbor12345
```

The secret must then be referenced from the pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: foo
spec:
  containers:
  - name: foo
    image: node001:9443/library/nginx
    imagePullSecrets:
    - name: myregistrykey
```

Further information on this is available at <https://kubernetes.io/docs/concepts/containers/images/#specifying-imagepullsecrets-on-a-pod>.

4.22 Kubernetes Upgrades

This section assumes that an upgrade to a BCM Kubernetes is being considered. This means that it is assumed to have been set up with `cm-kubernetes-setup`.

Upgrades to Kubernetes can be done by following the principles behind the official upstream documentation at <https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/>.

The following points must be kept in mind when upgrading a BCM Kubernetes cluster:

- Nodes that are part of the cluster may be provisioned through their software image.

- The Kubernetes version in BCM must manually be updated to match the version of the upgrade.
- Air-gapped upgrades follow the principles of this section, but have some extra considerations. Air-gapped upgrades are discussed in Appendix C, and the instructions there can be followed after reading this section (section 4.22) first.
- Only some Kubernetes versions are supported. A list of supported available versions can be retrieved with:

Example

```
[root@basecm11 ~]# cm-kubernetes-setup --list-versions
1.34
1.33 (NVIDIA AI Enterprise certified)
1.32 (NVIDIA AI Enterprise certified)
```

In the example that follows, a BCM cluster is upgraded from Kubernetes 1.31 to Kubernetes 1.32.

At the time of writing of this section (March 2026), the URL for the upstream documentation redirects the readers to the upgrade instructions for, among others, 1.33 to 1.34.

4.22.1 Upgrade Prerequisites

Kubernetes is not a single entity for upgrade purposes. There is a single driver, the `kubeadm` program, which is used to plan and execute the upgrade, but it does it only for itself and a handful of components central to Kubernetes' functionality. Kubernetes does not have ways to manage its third-party components with `kubeadm`'s upgrade planning and execution. This means that multiple Kubernetes components that are essential to Kubernetes are not involved in the upgrade process. If these components are neglected during an upgrade, then they can either prevent successful completion of the upgrade, or can themselves fail as a result of the upgrade.

Some noteworthy components that are excluded by upgrades are the `etcd` database, CNIs, CSIs, container runtimes, as well as individual pods not managed by higher-level entities such as `Deployment`, `ReplicaSet` and so on. The Kubernetes documentation at

<https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#pod-disruption-budgets>

describes the workings of the eviction process in greater detail.

Kubernetes upgrades assume that various entities, in particular, pods, can migrate away from the node being upgraded. This assumption may not be true because of various constraints on pod scheduling. The constraints need to be satisfied or worked around in order for migration to work. Particular care has to be taken when dealing with persistent volumes or physical resources such as GPUs that cannot migrate, but are needed for a pod to be deployed.

Hot And Cold Upgrades

- A hot upgrade is one that allows the system to stay online at all times.
- A cold upgrade is one that requires at least a brief service interruption.

Since Kubernetes is aimed at providing 100% system uptime, the upgrade process is designed to be hot. The cluster administrator carrying out the hot upgrade does however need to make an effort for the instances where Kubernetes cannot ensure service uptime.

Upgrading components that cannot be stopped in a regular manner: One common problem with hot upgrades is the need to migrate or to stop DaemonSet pods. These are intended to run on all Kubernetes nodes, even those being drained. Kubernetes can be told to ignore DaemonSet pods by using the `--ignore-daemonsets` option. The problem with doing that is that it avoids upgrading those pods, and also typically avoids upgrading the third-party component that created them. To ensure upgrade completion, the administrator has to carry out a separate upgrade action, appropriate for each component ignored in this way.

Upgrading gradually: Kubernetes upgrades come without any tools to perform upgrades at scale. The available tools can only upgrade a single node at a time. This means that, especially for larger clusters, the administrator needs to carry out the upgrade gradually, and has to plan and allocate extra capacity to accommodate pods migrated from nodes being upgraded. Special precautions need to be taken when dealing with deployments which require larger replica counts. Administrators may need to perform special procedures. Third-party components in particular may require the simultaneous presence of multiple entities across multiple nodes, in order to be able to upgrade those components without a service interruption.

Finally, it is possible that some components are simply not designed for hot upgrades. In that case, administrators have to remove those components from the system, and install the newer version, with the downtime depending on each individual case.

4.22.2 Example RHEL9 Cluster

A cluster with 3 nodes that run the control plane, and based on RHEL9, is considered as a reference example. One control plane node is the single head node, and the other two control plane nodes are two regular (compute) nodes. There are four additional worker-only nodes, and all the non-head nodes share the same category and software image.

```
[root@basecm11 ~]# module load kubernetes
[root@basecm11 ~]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
node001	Ready	control-plane,master,worker	8m25s	v1.27.13
node002	Ready	control-plane,master,worker	8m25s	v1.27.13
node003	Ready	worker	8m43s	v1.27.13
node004	Ready	worker	8m41s	v1.27.13
node005	Ready	worker	8m42s	v1.27.13
node006	Ready	worker	8m41s	v1.27.13
basecm11	Ready	control-plane,master	9m22s	v1.27.13

4.22.3 Before Starting The Upgrade

The upstream instructions at <https://v1-32.docs.kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/#before-you-begin>, on preparing for the upgrade should be read. A node that is to be upgraded must be drained.

4.22.4 Updating The First Control Plane Node

The first control plane node is upgraded in this section.

Instead of going into much detail on upgrades—the official documentation at <https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/> does a good job—an example covering an upgrade to a Rocky cluster running Kubernetes version 1.27 is shown that uses the official documentation for guidance. Instructions for Ubuntu differ slightly but can be carried out in a similar way.

Typically the control plane is on the head node, so that the cluster administrator starts with updating the control plane being run by the head node. However, for completeness, the two subsections that follow describe updating the control plane if starting with:

- either the head node
- or a compute node

Upgrading A Control Plane Node Starting With The Head Node

To upgrade the control plane when the head node is the first node to be upgraded, the repository entry for picking up packages from the Kubernetes repository should have the new version set.

Example

```
[root@basecm11 ~]# grep -ir kubernetes /etc/yum.repos.d/*          #is anything already there?
/etc/yum.repos.d/kubernetes.repo: [kubernetes]
/etc/yum.repos.d/kubernetes.repo:name=Kubernetes
/etc/yum.repos.d/kubernetes.repo:exclude=kubelet kubeadm kubectl cri-tools kubernetes-cni
[root@basecm11 ~]# grep v /etc/yum.repos.d/kubernetes.repo        #what version is fetched?
baseurl=https://pkgs.k8s.io/core:/stable:/v1.31/rpm/
gpgkey=https://pkgs.k8s.io/core:/stable:/v1.31/rpm/repodata/repomd.xml.key
```

The grep outputs confirm there is a repository file, and that it is set to pick up Kubernetes version 1.31. The .31 indicates what is called the minor version in Kubernetes. To instead pick up version 1.32, the upstream documentation instructs changing the minor version in the repository file to the next minor version upgrade number, so here from v1.31 to v1.32. Skipping minor versions for upgrades is unsupported.

An additional check to confirm the running version is:

```
[root@basecm11 ~]# kubeadm version
kubeadm version: &version.Info{Major:"1", Minor:"31", GitVersion:"v1.31.10", GitCommit: ...}
```

The minor version in `kubernetes.repo` is changed from v1.31 to v1.32 using a text editor. The installation for a new kubeadm can then be carried out with:

```
[root@basecm11 ~]# yum install -y kubeadm --disableexcludes=kubernetes
...
[root@basecm11 ~]# kubeadm version      #should now show we are at version 1.32
kubeadm version: &version.Info{Major:"1", Minor:"32", GitVersion:"v1.32.6", ...}
```

The upgrade plan can be inspected:

```
[root@basecm11 ~]# kubeadm upgrade plan
...
You can now apply the upgrade by executing the following command:
```

```
kubeadm upgrade apply v1.32.6
...
```

In this case it gives a recommended upgrade command to run on the head node:

The upgrade is now made to version 1.32.6, as suggested. The curious cluster administrator can view the list of available versions can be viewed with the `-showduplicates` option to yum:

Example

```
[root@basecm11 ~]# yum list --showduplicates kubeadm --disableexcludes=kubernetes
...
Installed Packages
kubeadm.x86_64                1.32.6-150500.1.1      @kubernetes
Available Packages
kubeadm.aarch64              1.32.6-150500.1.1      kubernetes
kubeadm.ppc64le              1.32.6-150500.1.1      kubernetes
```

```
kubeadm.s390x          1.32.6-150500.1.1    kubernetes
kubeadm.src            1.32.6-150500.1.1    kubernetes
kubeadm.x86_64        1.32.6-150500.1.1    kubernetes
...
```

Most cluster administrators stick to the recommendation.

```
[root@basecm11 ~]# kubeadm upgrade apply v1.32.6
...
[upgrade/version] You have chosen to change the cluster version to "v1.32.6"
[upgrade/versions] Cluster version: v1.31.10
[upgrade/versions] kubeadm version: v1.32.6
[upgrade] Are you sure you want to proceed? [y/N]: y
...
[upgrade/successful] SUCCESS! Your cluster was upgraded to "v1.32.6". Enjoy!

[upgrade/kubelet] Now that your control plane is upgraded, please proceed with upgrading your kubelets
if you haven't already done so.
```

The host should now be drained, if it has not already been drained:

```
[root@basecm11 ~]# kubectl drain $(hostname) --ignore-daemonsets
```

As suggested, the kubelets are now upgraded. Updating the kubectl version at the same time is also a good idea:

```
[root@basecm11 ~]# yum install -y kubelet kubectl --disableexcludes=kubernetes
```

The kubelets can be restarted so that the new versions run:

```
[root@basecm11 ~]# kubelet --version
Kubernetes v1.32.6
[root@basecm11 ~]# systemctl status kubelet
...
Active: active (running) since ...; 21h ago    #need to reload kubelets to new versions
...
[root@basecm11 ~]# systemctl daemon-reload
[root@basecm11 ~]# systemctl restart kubelet && systemctl status kubelet | grep Active
    Active: active (running) since ... 14ms ago
```

This ends the procedure of upgrading the control plane by first starting on a head node. Upgrading the control plane by first starting with a regular node instead is described next.

Upgrading The Control Plane Node Starting With A Compute Node

This section describes the list of commands for upgrading the control plane when it is the first control plane node to be upgraded. For example node001, rather than a head node.

The appropriate software image and related information for the node must first be found. For node001 this can be done with:

```
[root@basecm11 ~]# cmsg
[basecm11]% device use node001
[basecm11->device[node001]]% get softwareimage
default-image (category:default)
[basecm11->device[node001]]% softwareimage
[basecm11->softwareimage]% use default-image
[basecm11->softwareimage[default-image]]% get path
/cm/images/default-image
```

The software image is then entered and modified using a chroot jail.

The repository file of the image, with an absolute path outside the chroot jail of `/cm/images/default-image/etc/yum.repos.d/kubernetes.repo`, has its minor version changed appropriately:

```
[root@basecm11 ~]# cm-chroot-sw-img /cm/images/default-image
[root@default-image /]# grep v /etc/yum.repos.d/kubernetes.repo #what version would be fetched?
/etc/yum.repos.d/kubernetes.repo:baseurl=https://pkgs.k8s.io/core:/stable:/v1.31/rpm/
/etc/yum.repos.d/kubernetes.repo:gpgkey=https://pkgs.k8s.io/core:/stable:/v1.31/rpm/repodata/repomd.xml.key
[root@default-image /]# vi /etc/yum.repos.d/kuberenetes.repo #modify the point version in the image
... modifications done...
[root@default-image /]# grep v /etc/yum.repos.d/kubernetes.repo #will fetch this version now
/etc/yum.repos.d/kubernetes.repo:baseurl=https://pkgs.k8s.io/core:/stable:/v1.32/rpm/
/etc/yum.repos.d/kubernetes.repo:gpgkey=https://pkgs.k8s.io/core:/stable:/v1.32/rpm/repodata/repomd.xml.key
```

The package versions available for installation in the image can be checked with:

```
[root@default-image /]# yum info kubeadm kubelet kubect1 --disableexcludes=kubernetes
```

In this case versions 1.32.6 are seen to be available.

The kubeadm, kubelet and kubect1 packages are upgraded in advance, and the chroot environment is then exited:

```
[root@default-image /]# yum install -y kubeadm kubelet kubect1 --disableexcludes=kubernetes
...
[root@default-image /]# exit #leave the chroot
```

The following set of commands then carries out the update on node001:

```
[root@basecm11 ~]# export host=node001
[root@basecm11 ~]# cmsh -c "device use $host; imageupdate -w --wait" #update running image on node
[root@basecm11 ~]# ssh $host kubeadm version #what version to fetch now
[root@basecm11 ~]# ssh $host kubeadm upgrade plan #get suggested upgrade plan
[root@basecm11 ~]# ssh $host kubeadm upgrade apply v1.32.6 #apply the upgrade
[root@basecm11 ~]# kubect1 drain $host --ignore-daemonsets #drain the host
[root@basecm11 ~]# ssh $host sudo systemctl daemon-reload #reload systemd manager config
[root@basecm11 ~]# ssh $host sudo systemctl restart kubelet #restart kubelet
[root@basecm11 ~]# kubect1 uncordon $host #new pods may now run on node
```

4.22.5 Updating Subsequent Control Plane Nodes

If there is only one control plane node, then this section can be skipped.

If there is more than one control plane node, and if the first control plane node is a head node that has been updated, then the remaining control plane nodes should be updated. The kubect1 command can be used to orient the cluster administrator, by displaying the state of the upgrade, roles, and Kubernetes versions per node:

Example

```
[root@basecm11 ~]# kubect1 get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
node001	Ready	control-plane,master,worker	26m	v1.31.10
node002	Ready	control-plane,master,worker	26m	v1.31.10
node003	Ready	worker	26m	v1.31.10
node004	Ready	worker	26m	v1.31.10
node005	Ready	worker	26m	v1.31.10
node006	Ready	worker	26m	v1.31.10
basecm11	Ready	control-plane,master	27m	v1.32.6

The remaining control plane nodes can now be updated. In the reference example case, these control plane nodes are node001 and node002 as seen in the preceding example. The commands in the section on **Upgrading The Control Plane Node Starting With A Compute Node** (page 88) should be followed, but with one small difference. Instead of running:

```
kubeadm upgrade apply v1.32.6
```

the command

```
kubeadm upgrade node
```

is run.

The software image should also be prepared with the new packages before exiting the chroot jail. After exiting the chroot jail the remaining control plane nodes can then be updated with the following set of commands:

Example

```
[root@basecm11 ~]# export host=node001
[root@basecm11 ~]# cmsh -c "device use $host; imageupdate -w --wait"
[root@basecm11 ~]# ssh $host kubeadm version
[root@basecm11 ~]# ssh $host kubeadm upgrade plan
[root@basecm11 ~]# ssh $host kubeadm upgrade node           # I'm special! (apply not used here)
[root@basecm11 ~]# kubectl drain $host --ignore-daemonsets
[root@basecm11 ~]# ssh $host sudo systemctl daemon-reload
[root@basecm11 ~]# ssh $host sudo systemctl restart kubelet
[root@basecm11 ~]# kubectl uncordon $host
```

Output after this should now look similar to:

```
[root@basecm11 ~]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
node001	Ready	control-plane,master,worker	30m	v1.32.6
node002	Ready	control-plane,master,worker	30m	v1.31.10
node003	Ready	worker	30m	v1.31.10
node004	Ready	worker	30m	v1.31.10
node005	Ready	worker	30m	v1.31.10
node006	Ready	worker	30m	v1.31.10
basecm11	Ready	control-plane,master	31m	v1.32.6

The procedure can be repeated for node002 in the reference example system. After that, the control plane nodes are all in an updated state. The worker nodes still need updating.

4.22.6 Updating The Worker Nodes

The commands for the worker nodes follow a similar pattern. They are updated one by one, and the software images must also be updated, if it has not already been done. The software image update follows the procedure described in the section on **Upgrading The Control Plane Node Starting With A Compute Node** (page 88), where a chroot jail is entered, the updates are carried out, and the chroot jail is left.

Since nodes are drained as part of their update procedure, it is best to do them one by one, or at most in limited batches to avoid doing too many at once.

The set of commands to update a single worker is:

```
[root@basecm11 ~]# export host=node003
[root@basecm11 ~]# cmsh -c "device use $host; imageupdate -w --wait"
[root@basecm11 ~]# ssh $host kubeadm upgrade node
[root@basecm11 ~]# kubectl drain $host --ignore-daemonsets
[root@basecm11 ~]# ssh $host sudo systemctl daemon-reload
[root@basecm11 ~]# ssh $host sudo systemctl restart kubelet
[root@basecm11 ~]# kubectl uncordon $host
```

The `kubectl drain` command might complain for other reasons. The administrator can decide on proceeding further by adding additional flags.

For example:

```
kubectl drain $host --ignore-daemonsets --delete-emptydir-data
```

forces the drain even if some pods have stateful data in `emptyDir` volumes.

Multiple nodes can be updated as follows:

Example

```
[root@basecm11 ~]# export hosts='node00[4-6]'
[root@basecm11 ~]# cmsh -c "device; imageupdate -n $hosts -w --wait"
[root@basecm11 ~]# pdsh -w $hosts kubeadm upgrade node
[root@basecm11 ~]# pdsh -N -w $hosts hostname | xargs -i -n 1 kubectl drain {} --ignore-daemonsets \
--delete-emptydir-data
[root@basecm11 ~]# pdsh -w $hosts sudo systemctl daemon-reload
[root@basecm11 ~]# pdsh -w $hosts sudo systemctl restart kubelet
[root@basecm11 ~]# pdsh -N -w $hosts hostname | xargs -i -n 1 kubectl uncordon {}
[root@basecm11 ~]#
```

Output after all the worker nodes are updated too should look similar to:

```
[root@basecm11 ~]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
node001	Ready	control-plane,master,worker	35m	v1.32.6
node002	Ready	control-plane,master,worker	35m	v1.32.6
node003	Ready	worker	35m	v1.32.6
node004	Ready	worker	35m	v1.32.6
node005	Ready	worker	35m	v1.32.6
node006	Ready	worker	35m	v1.32.6
basecm11	Ready	control-plane,master	36m	v1.32.6

4.22.7 Updating The Status In BCM

For releases of BCM before BCM 11.32, setting the attribute `version` for the Kubernetes object in CM-Daemon is not carried out automatically by BCM. Setting that attribute is however necessary in order to get the correct version of Kubernetes showing up in the available module files. A session to update the old version is as follows:

```
[root@basecm11 ~]# module avail -l | grep kubernetes      #what Kubernetes version is defined?
kubernetes/default/1.31.10-150500.1.1                    2025/06/30 18:01:08
[root@basecm11 ~]# cmsh
[basecm11]% kubernetes
[basecm11->kubernetes[default]]% get version
1.31.10-150500.1.1
[basecm11->kubernetes[default]]% !yum info kubeadm | grep -E '(^Source|^Version|^Release)'
Version      : 1.32.6
Release      : 150500.1.1
Source       : kubeadm-1.32.6-150500.1.1.src.rpm
[basecm11->kubernetes[default]]% set version 1.32.6-150500.1.1
[basecm11->kubernetes*[default*]]% commit
[basecm11->kubernetes[default]]% quit
[root@basecm11 ~]# module avail -l | grep kubernetes      #what Kubernetes version is defined now?
kubernetes/default/1.32.6-150500.1.1                    2025/06/30 21:40:57
```

4.22.8 Notes For Upgrading To Kubernetes v1.31.x

The `kubeadm-init-*` files generated by BCM for Kubernetes clusters older than v1.31, contain an incompatibility with v1.31 that cannot be solved automatically by `kubeadm config migrate` when carrying out a migration as described in <https://kubernetes.io/docs/reference/setup-tools/kubeadm/kubeadm-init/#config-file>.

To prevent errors while executing the `kubeadm config migrate` command, any `JoinConfiguration` block needs to be removed from the `kubeadm-init-*` files.

The `kubeadm-init-*` files are files that, for a Kubernetes cluster name, follow the pattern:

```
/root/.kube/kubeadm-init-<Kubernetes cluster name>.yaml
```

Thus, for the default BCM Kubernetes cluster installation that creates a Kubernetes cluster called `default`, this would be the single YAML file:

```
/root/.kube/kubeadm-init-default.yaml
```

The part to remove in such files looks like this:

```
---
apiVersion: kubeadm.k8s.io/v1beta3
kind: JoinConfiguration
# do not uncomment the following, kubelets will not be able to start
# caCertPath: "/etc/kubernetes/pki/kc.name/ca.crt"
```

4.22.9 Notes For Ubuntu

The upstream documentation covers Ubuntu upgrades when it diverges from RHEL procedures. As an aid, the following commands, discussed in the procedure for RHEL within section 4.22 are followed by the corresponding commands for Ubuntu:

```
# RHEL
grep v /etc/yum.repos.d/kubernetes.repo
vi /etc/yum.repos.d/kubernetes.repo      #to change minor number

yum list --showduplicates kubeadm --disableexcludes=kubernetes      #see the versions

sudo yum install -y kubeadm kubelet kubectl --disableexcludes=kubernetes      #installs from repo

# Ubuntu
grep v /etc/apt/sources.list.d/kubernetes.list
vi /etc/apt/sources.list.d/kubernetes.list      #to change minor number

apt-get update; apt-cache madison kubeadm      # update for a fresh cache to see the versions
                                              # madison option lists versions and origin repos

apt-mark unhold kubeadm kubelet kubectl && \
apt-get update && apt-get install -y kubeadm kubelet kubectl && \
apt-mark hold kubeadm kubelet kubectl          #steps from upstream docs, installs from repo
```

4.22.10 Notes For SLES

For v1.32, for SLES, BCM can use the “tarball” approach. The installation of such non-package-manager packages is documented at <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/>, in the “Without a package manager” section. The commands in that section there can be used as a guide to find alternatives for the `yum` and `apt` commands used so far.

4.22.11 Other Approaches

When carrying out the steps described in this chapter, some things may in practice end up being done differently from what is suggested.

For example: control plane nodes often have different categories and software-images compared to the workers. For example, perhaps the control-plane nodes do not have GPUs, and need different packages. In that case, multiple software images have to be prepared with new packages.

Earlier on, a process to update nodes one at a time was described. However, when a software image is updated, and multiple nodes are tied to that software image, then those nodes can all be provisioned at the same time. The binaries in the updated software image can therefore also be provisioned to all the nodes using that image.

The problem with this is that it could result in nodes getting the new binaries prematurely if they happen to reboot during the update. If this is an unwanted risk, then it can be avoided in several ways, described next.

Additional Software Images

A separate software image can be introduced. Nodes can be moved to the new software image one at a time. A software image can also be overruled at the level of a node.

Example

```
[root@basecm11 ~]# cmsht
[basecm11]% device use node001
[basecm11->device[node001]]% get softwareimage
default-image (category:default)
[basecm11->device[node001]]% set softwareimage my-new-image
[basecm11->device*[node001*]]% commit
[basecm11->device[node001]]% get softwareimage
my-new-image
[basecm11->device[node001]]% quit
```

The cluster administrator should remember to undo the preceding settings, or should move the new software image to the appropriate category, and then clear the node-level override again. This is so that, after the upgrade, the system is organized as before.

Using NOSYNC

The other option is to configure the nodes with NOSYNC for their Next install mode. This prevents them from syncing with their software image when rebooting (in the case that they still reboot):

Example

```
[root@basecm11 ~]# cmsht
[basecm11]% device use node001
[basecm11->device[node001]]% set nextinstallmode nosync
[basecm11->device*[node001*]]% commit
```

Both these approaches make updating slightly more tedious, but also more straightforward.

During testing by BCM developers, nodes getting their binaries updated prematurely due to an unexpected reboot was not seen to be a significant issue. This is presumably because as long as the first control plane node is updated successfully, and the reboot of the extra node is by accident, there is an interruption anyway. The kubelet simply comes back up with the new version. However this is not the official recommended approach.

5

Kubernetes Apps

Kubernetes add-ons were introduced in an older version NVIDIA Base Command Manager (at that time still called Bright Cluster Manager) version 8.1 Add-ons could be managed in that version as part of the addons submode of the kubernetes mode in cmsh. The feature later (in version 8.2) was expanded into the *Kubernetes Applications & Groups* feature. Kubernetes Applications & Groups, less formally called app groups, can be accessed via the appgroups submode of cmsh:

Example

```
root@basecm11 ~# cmsh
[basecm11]% kubernetes
[basecm11->kubernetes[default]]% appgroups
[basecm11->kubernetes[default]->appgroups]% list
Name (key)    Applications
-----
system        <13 in submode>
[basecm11->kubernetes[default]->appgroups]%
```

Versions from 9.2 onward started deploying operators as Helm charts, such as for the NVIDIA GPU operator, Run:ai, and NetQ. The remaining apps have since been migrated to equivalent Helm charts except for Flannel.

The legacy version 8.1 addons mode parameters are now accessed from version 8.2 onward via a default system app group instance. The system instance is accessed in the appsgroup submode.

Example

```
[basecm11->kubernetes[default]->appgroups]% use system
[basecm11->kubernetes[default]->appgroups[system]]% show
Parameter          Value
-----
Name                system
Revision
Enabled             yes
applications        <13 in submode>
[basecm11->kubernetes[default]->appgroups[system]]% applications
[basecm11->kubernetes[default]->appgroups[system]->applications]% list
Name (key)          Format Enabled
-----
bootstrap           Yaml    yes
calico              Yaml    yes
flannel             Yaml    no
kubestatemetrics    Yaml    yes
metrics_server      Yaml    yes
root                Yaml    yes
```

A Kubernetes application can span multiple namespaces. A name in appgroups therefore only exists to group logically-related applications. Each application contains a YAML configuration file, which BCM synchronizes to the Kubernetes API.

The default system app group is pre-defined. Other app groups can be created as needed. For example, an app group called `monitoring` could be created to group applications for running Prometheus, node exporters, and anything else related to exposing or viewing Prometheus metrics.

Toggling the `Enable` parameter of an app group enables or disables all of its application components in Kubernetes. Finer-grained control is possible within the applications mode level, by toggling the `enabled` parameter per application component instance. For example, within the `calico` application component instance:

Example

```
[basecm11->kubernetes[default]->appgroups[system]->applications]% use calico
[basecm11->kubernetes[default]->appgroups[system]->applications[calico]]% show
```

Parameter	Value

Name	calico
Revision	
Format	Yaml
Enabled	yes
Config	<244KiB>
Environment	<3 in submode>
Exclude list snippets	<2 in submode>

A large YAML configuration file for each application component instance can be configured via the `Config` parameter property, using the `set` option of `cmsh`. This opens up a text editor and allows the environment variables in the YAML configuration file to be managed.

Exclude list snippets are short exclude lists that can be set up for Kubernetes apps computing within the `excludelistsnippets` submode. They are used to prevent BCM software image updates from overwriting the provisioned files or directories of the container image that are important to the associated Kubernetes application.

Using `exclude list snippets` within an `excludelistsnippets` submode is discussed in detail in section 4.3.1 of the *Cloudbursting Manual*. Similar to the case of Kubernetes apps images, in cloud computing `exclude list snippets` are used to prevent overwriting of the provisioned files and directories of cloud images.

Environment entries can be set via the `Environment` submode. Environment entries are similar to environment variables, and are used to replace variables inside the YAML configuration file. Environment entries can be added to the environment as well, if the `Nodes environment` value inside the `Environment` submode is set to `yes`.

Example

```
[basecm11->kubernetes[default]->appgroups[system]->applications[calico]]% environment
[basecm11->kubernetes[default]->appgroups[system]->applications[calico]->environment]% list
```

Name (key)	Value	Nodes environment

calico_typha_replicas	0	no
calico_typha_service	none	no
head_node_internal_ip	10.141.255.254	no

6

Kubernetes Operators

Kubernetes operators are the modern way to manage Kubernetes cluster applications (<https://kubernetes.io/docs/concepts/extend-kubernetes/operator/>). It is usually recommended that Kubernetes operators are used instead of the legacy applications.

At the time of writing of this section (July 2025), NVIDIA Base Command Manager provides and packages several operators which are validated to perform basic functionalities on a Kubernetes BCM setup.

- the NVIDIA GPU Operator (section 6.4)
- the NVIDIA Network Operator (section 6.5)
- the NVIDIA NetQ Operator (section 6.6)
- the Prometheus Operator Stack (section 6.7)
- the Prometheus Adapter Operator
- the Run:ai Operator (section 6.8)
- the Jupyter Kernel Operator (section 6.3)
- the Spark Operator (section 6.9)
- the PostgreSQL Operator (section 6.10)
- the Kubernetes MPI Operator
- the MetalLB Operator
- the Kubeflow Training Operator
- the NVIDIA NIM Operator (section 6.11)

6.1 Versions Of Operators Available

The versions of the operators that are available can be listed with the `cm-kubernetes-setup --list-operators-versions` command.

The versions available at the time of writing of this section (November 2025) for an installation in Ubuntu 24.04 are shown in the following table:

Kubernetes Operator	Default	Possible choices
NVIDIA GPU Operator	25.3.4 ^E	latest, 25.3.3 ^E , 25.3.2 ^E
MetalLB	0.15.2	latest
NetQ Operator	latest	latest
Network Operator	25.7.0 ^E	latest, 25.4.0 ^E
OVN CNI	1.1.3	1.1.3
Postgres Operator	1.14.0	1.12.2, 1.13.0
Prometheus Adapter	5.1.0	5.1.0
Prometheus Operator Stack	76.6.2	76.6.2
Run:ai	2.22.x	2.17.x, 2.18.x, 2.19.x, 2.20.x, 2.21.x, 2.22.x, custom
SDN	1.0.3	1.0.3
CM Jupyter Kernel Operator	latest	latest
CM Kubernetes MPI Operator	latest	latest
Spark Operator	2.3.0	2.3.0
NVIDIA NIM Operator	3.0.0	3.0.0
Grafana Alloy	1.2.1	latest
Grafana Loki	6.40.0	latest
Grafana Promtail	6.17.0	latest
Kgateway	v2.3.0	v2.3.0
Kata	latest	latest
Tigera	3.30.3	3.30.3

^E NVAIE-certified

During the initial setup, the installation wizard displays a menu to select which operators are to be installed (figure 6.1).

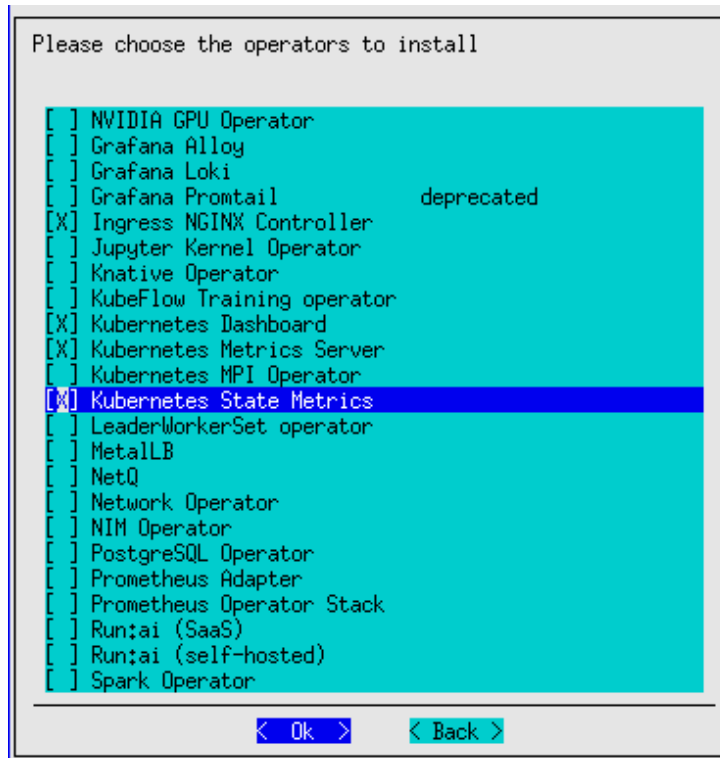


Figure 6.1: Kubernetes setup TUI session: selection of operator packages to be installed

After selecting the operators to be used, the actual operator versions available depend on package management resolution.

In the case of BCM operators, the installation wizard uses a Helm chart from the OS package manager for some of the .deb or .rpm packages being deployed. In the case of cloud operators, the installation wizard uses a Helm chart from the Helm repository for deployment.

6.2 Helm Charts For The BCM Operators

Based on the operators that are to be installed, the wizard asks for configuration options to the Helm charts that are to be installed (figure 6.2):

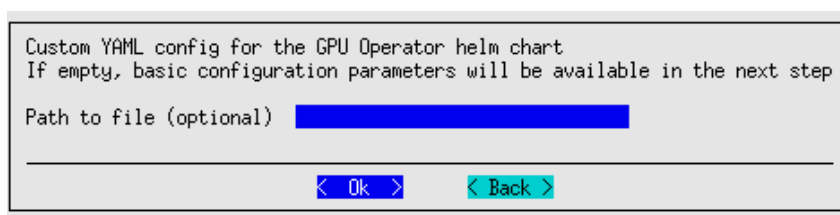


Figure 6.2: Kubernetes setup TUI session: prompting for Helm chart configuration options

The Helm charts that are selected are installed with sensible defaults. If additional tuning is needed, then the charts can be installed manually after `cm-kubernetes-setup` finishes:

```
[root@basecm11 ~]# yum install cm-jupyter-kernel-operator -y
[root@basecm11 ~]# helm install cm-jupyter-kernel-operator -n cm --wait \
  /cm/shared/apps/jupyter-kernel-operator/current/helm/*.tgz
```

If additional tuning is required, then tunable values can be set with a command line similar to the following:

```
[root@basecm11 ~]# helm install cm-jupyter-kernel-operator -n cm --wait \
  --values tunables.yaml \
  /cm/shared/apps/jupyter-kernel-operator/current/helm/*.tgz
```

Possible values can be displayed as follows:

```
[root@basecm11 ~]# helm show values /cm/shared/apps/jupyter-kernel-operator/current/helm/*.tgz
```

Installed operators can be listed by using the CLI option `--list-operators`:

```
[root@basecm11 ~]# cm-kubernetes-setup --list-operators
...
#### stage: kubernetes: Display Available Operators
OPERATOR-----: api_available-----
cm-jupyter-kernel-operator      : 1
postgresql-operator           : 1
spark-operator                 : 1
cm-kubernetes-mpi-operator     : 1
...
```

6.3 The Jupyter Kernel Operator

6.3.1 Installing The Jupyter Kernel Operator

The Kubernetes Jupyter Kernel Operator can be installed as a part of the `cm-kubernetes-setup` procedure (section 4.2.8), which eventually leads to the selection screen displayed in figure 6.1.

The Kubernetes Jupyter Kernel Operator can alternatively be installed later on using the OS package manager and Helm:

```
[root@basecm11 ~]# yum install cm-jupyter-kernel-operator -y
[root@basecm11 ~]# helm install cm-jupyter-kernel-operator \
  /cm/shared/apps/jupyter-kernel-operator/current/helm/cm-jupyter-kernel-operator-*.tgz
```

The Jupyter Kernel Operator can be removed with:

Example

```
[root@basecm11 ~]# helm uninstall cm-jupyter-kernel-operator
```

It is recommended to enable a pod security policy using Kyverno (section 4.10.1) for the cluster before allowing a user, for example `alice`, to create resources in the Kubernetes cluster.

The Kubernetes Jupyter Kernel Operator Helm chart creates a CRD that can be used in the Kubernetes API. To check the availability of the CRD, the following command can be run:

Example

```
[root@basecm11 ~]# module load kubernetes
[root@basecm11 ~]# kubectl get crd | grep jupyterkernels
cmjupyterkernels.apps.brightcomputing.com          2022-11-07T09:49:48Z
cmkubernetesoperatorpermissionsjupyterkernels.charts.brightcomputing.com 2022-11-07T09:18:32Z
```

6.3.2 Architecture Overview

The Kubernetes Jupyter Kernel Operator has two main components:

- the operator itself
- the sidecar. This is attached to every user-defined kernel pod, and communicates with Jupyter Enterprise Gateway, acting as a proxy for the kernel process.

The following is an overview of the kernel setup and pod lifecycle when the user runs the Kubernetes Jupyter Kernel Operator:

1. User initiates creating kernel in JupyterLab.
2. JupyterLab delegates this task to Jupyter Enterprise Gateway (JEG).
3. JEG opens a service TCP/IP socket and creates a CRD in Kubernetes specifying this port.
4. KubeApi notifies Jupyter Kernel Operator about the newly created CRD.
5. Jupyter Kernel Operator creates services, configmaps, secrets.
6. Jupyter Kernel Operator creates pod to run Jupyter kernel based on the specification. The sidecar is added to the kernel pod during this step.
7. The sidecar waits for the connection file created by the kernel. Alternatively, it relies on the connection file created by the operator (if requested), as not all kernels create a connection file.
8. The sidecar runs a proxy to forward kernel communications to JEG (stdin, shell, iopub, etc).
9. The sidecar notifies JEG about connection parameters and handles kernel communications.
10. If JEG disappears, or if communication drops, then the sidecar stops. This causes the kernel operator to get a notification via the KubeApi service.
11. The Kubernetes Jupyter Kernel Operator removes the unneeded pod, service, configmap and secrets. It also tries to gather stdout and std error of the kernel pod for debug purposes.

The pod created in step 6 is heavily customized by the kernel operator. For security reasons, running a process inside the pod must be carried out as an unprivileged user.

For the convenience of the Jupyter user, the UID/GID of the process inside the pod should match the UID/GID of the Jupyter user. If that is not the case, then the files created in the container are inaccessible for the Jupyter user.

To achieve matching UID/GIDs, the operator dynamically creates `/etc/passwd` and `/etc/group` files inside the pod and populates them with the data from corresponding templates. At the same time the operator can create a kernel communication file, if requested—some kernels rely on that.

6.3.3 Running Jupyter Kernel Using The Operator

An example of a basic YAML definition for the CMJupyterKernel is:

```
[alice@basecm11 ~]$ cat cmjk.yaml
---
apiVersion: apps.brightcomputing.com/v1
kind: CMJupyterKernel
metadata:
  name: cmjk-test
  namespace: alice-restricted
spec:
  username: alice
  uid: 1001
  gid: 1001
  kernel_id: testtesttest
  homedir: /home/alice
  pod:
    volumes:
    - name: homedir
      hostPath:
        path: /home/alice
        type: DirectoryOrCreate
    containers:
    - name: kernel
      image: jupyter/datascience-notebook
      command:
      - "python"
      args:
      - "-m"
      - "ipykernel_launcher"
      - "-f"
      - "/var/tmp/kernel-parm.json"
      workingDir: /home/alice
      securityContext:
        allowPrivilegeEscalation: false
        privileged: false
        runAsNonRoot: true
        runAsUser: 1001
        runAsGroup: 1001
      volumeMounts:
      - name: homedir
        mountPath: /home/alice
```

This can be submitted, but the operator removes it in approximately 1 minute:

```
[alice@basecm11 ~]$ module load kubernetes
[alice@basecm11 ~]$ kubectl apply -f cmjk.yaml
```

The logs of the operator can be checked for debug purposes:

```
[root@basecm11 ~]# module load kubernetes
[root@basecm11 ~]# kubectl logs \
  -n cm-jupyter-kernel-operator-system \
  -l control-plane=controller-manager \
  --tail -1 \
  -c manager
```

```
...
2022-02-14T18:26:01.005Z INFO controllers.CMJupyterKernel Container is
stopped. Logs are below "cmjupyterkernel": "alice-restricted/cmjk-test"
...
2022-02-14T18:26:01.190Z INFO controllers.CMJupyterKernel cmjksidecar:
2022/02/14 18:26:00 Timeout receiving pings from server for 60 sec.
Shutting down. "cmjupyterkernel": "alice-restricted/cmjk-test"
...
```

This indicates that the sidecar was stopped because there was no connection from Jupyter Enterprise Gateway to the kernel. This is expected, since the kernel has been run manually, and not using Jupyter. After the sidecar shutdown, the kube-api server notifies the operator, which, in turn, removes objects such as CMJupyterKernel, pods, and services.

6.3.4 Jupyter Kernel Operator Tunables

Table 6.1: Tunable Options for Jupyter Kernel Operators

Option	Description
<code>kernel_id^R</code>	Kernel identifier (random UUID) given by Jupyter server
<code>username^O</code>	Name of the user
<code>uid^R, gid^R, homedir^O, usershell^O</code>	UID, GID, home directory and default shell of the user
<code>image_os_flavor^O</code>	Defines template of <code>/etc/passwd</code> and <code>/etc/group</code> files, where <code>uid</code> , <code>gid</code> , <code>homedir</code> , and <code>usershell</code> will be added. Could be one of <code>ubuntu1604</code> , <code>ubuntu1804</code> , <code>ubuntu2004</code> , <code>ubuntu2404</code> , <code>rhel7</code> , <code>centos7</code> , <code>rhel8</code> , <code>centos8</code> , <code>sles12</code> , <code>sles15</code> .
<code>etc_passwd^O, etc_group^O</code>	Custom content of the <code>/etc/passwd</code> or <code>/etc/group</code> , if necessary.
<code>sidecar_command^O, sidecar_args^O</code>	Commands and arguments to run the sidecar. By default empty. Most of the arguments for the sidecar are passed via environment variables (section 6.3.5).

...continues

...continued

Option	Description
kernel_connection_file_path ^O	Where to expect to find kernel connection file. Default: /var/tmp/kernel-parm.json
create_connection_file ^O	Does the operator need to create and populate kernel connection file before the pod starts? Default: false
spark_pod_template_path ^O , spark_pod_template ^O pod ^R	Options to store or override the Spark executor template Kubernetes Pod definition
service ^O	Kubernetes Service definition

Legend:

O: Optional

R: Required

6.3.5 Sidecar Arguments And Environment Variables

Sidecar Arguments

A timeout can be set as an argument for the sidecar.

- --timeout: Defines how long, in seconds, that the sidecar waits for the Jupyter Enterprise Gateway proxy to connect before shutdown. Default: 60

Environment Variables

The following environment variables can be used by the sidecar:

Table 6.2: Environment Variables For The Sidecar

Environment Variable	Description
CMJK_CONNECTION_FILE	Path to find a connection file. The sidecar uses the file to establish a connection to the kernel and to pass data between Jupyter Enterprise Gateway and the kernel. Default: /var/tmp/kernel-parm.json.
CMJK_KERNEL_ID	Unique identifier of the kernel. Usually the UUID in table 6.1.

...continues

...continued

Environment Variable	Description
CMJK_SHELL_PORT	Proxy port to open to forward shell communication. Default: 5001.
CMJK_IOPUB_PORT	Proxy port to open to forward iopub communication. Default: 5002.
CMJK_STDIN_PORT	Proxy port to open to forward stdin communication. Default: 5003.
CMJK_CONTROL_PORT	Proxy port to open to forward control communication. Default: 5004.
CMJK_HB_PORT	Proxy port to open to forward heartbeat communication. Default: 5005.
CMJK_COMM_PORT	Proxy port to open to forward comm communication. Default: 5006.

6.3.6 Running Spark-based Kernels In Jupyter Kernel Operator

Jupyter integration for BCM provides a kernel template (jupyter-eg-kernel-k8s-cmjkop-py-spark) and a sample container image (brightcomputing/jupyter-kernel-sample:k8s-spark-py39-2.0.0) to run Jupyter kernels in a Spark environment. The image can be altered or created from scratch based on the scripts provided in /cm/shared/examples/jupyter/kubernetes-kernel-image-spark-py39/.

The Jupyter kernel is not run directly. Instead, the kernel process is run and controlled by the spark-submit executable inside the container.

Jupyter Kernel Operator alters the provided image based on the CRD definition.

Spark-specific tunables are spark_pod_template_path and spark_pod_template. The operator creates a file inside of the Spark driver pod and puts the content of spark_pod_template in it. After that, spark-submit uses this file, via the --spark.kubernetes.executor.podTemplateFile configuration option, to create executor pods.

6.3.7 Example: Creating An R Kernel From The Kernel Template

The Jupyter Kernel Operator can be used out-of-the-box to support more kernels.

For example, an R kernel can be added.

The official jupyter/r-notebook R image can be taken from the Jupyter project, from <https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html#jupyter-r-notebook>.

The default entry point cannot be used as that would start Jupyter notebook, while the aim for this section is to use the kernel only.

Some exploratory investigation should reveal the command to start the kernel:

The pod can be run interactively in a Jupyter notebook terminal by a user:

```
kubect1 run -i --tty testnotebook --image=jupyter/r-notebook --restart=Never -- bash
```

The kernel specifications can then be investigated:

```
jupyter-kernelspec list
Available kernels:
  ir          /opt/conda/share/jupyter/kernels/ir
  python3     /opt/conda/share/jupyter/kernels/python3
```

The ir kernel is what is of interest here. The command line that is used to start the kernel can be found:

```
cat /opt/conda/share/jupyter/kernels/ir/kernel.json
"argv": ["R", "--slave", "-e", "IRkernel::main()", "--args", "connection_file"],
"display_name": "R",
"language": "R"
```

Based on this information the Jupyter Kernel Operator CRD can be created for the user:

```
cat cmjk-ir.yaml
---
apiVersion: apps.brightcomputing.com/v1
kind: CMJupyterKernel
metadata:
  name: cmjk-test
  namespace: alice-restricted
spec:
  username: alice
  uid: 1001
  gid: 1001
  kernel_id: testtesttest
  homedir: /home/alice
  create_connection_file: true # R kernel expects connection file be created
  pod:
    volumes:
      - name: homedir
        hostPath:
          path: /home/alice
          type: DirectoryOrCreate
    containers:
      - name: kernel
        image: jupyter/r-notebook # image
        command:
          - "R"
        args:
          - "--slave"
          - "-e"
          - "IRkernel::main()"
          - "--args"
          - "/var/tmp/kernel-parm.json" # we have static connection file
        workingDir: /home/alice
        securityContext:
          allowPrivilegeEscalation: false
          privileged: false
          runAsNonRoot: true
          runAsUser: 1001
          runAsGroup: 1001
        volumeMounts:
          - name: homedir
            mountPath: /home/alice
```

There are several changes from the previous (section 6.3.3) YAML, and from the IR command line:

- `create_connection_file: true`

If this is not specified then the kernel complains with the following message during startup:

```
kernel: cannot open file '/var/tmp/kernel-parm.json': No such file or directory
```

This means that the kernel expected this file to be created before the start.

- image: jupyter/r-notebook

Another image needs to be used.

- args: ... "/var/tmp/kernel-parm.json"

The spec file has a fixed path and name, instead of "connection_file" as in kernel.json earlier

The resulting cmjk-ir.yaml file can be submitted to Kubernetes, but it will be removed by the operator after one minute, as it is not being started from the Jupyter Enterprise Gateway.

The next step is to create a kernel template. The Python kernel can be used as a reference:

```
cd /cm/shared/apps/jupyter/current
cd lib/python*/site-packages/cm_jupyter_kernel_creator/kerneltemplates
cp -pr jupyter-eg-kernel-k8s-cmjkop-py jupyter-eg-kernel-k8s-cmjkop-r
```

The files meta.yaml, kernel.json, and templates/cmjk.yaml.j2 need to be changed in order to be able to provide the correct image and command:

```
vim jupyter-eg-kernel-k8s-cmjkop-r/meta.yaml
vim jupyter-eg-kernel-k8s-cmjkop-r/kernel.json.j2
vim templates/cmjk.yaml.j2
```

The changes that are applied should look similar to the following:

```
diff -u jupyter-eg-kernel-k8s-cmjkop-py/kernel.json.j2 jupyter-eg-kernel-k8s-cmjkop-r/kernel.json.j2
--- jupyter-eg-kernel-k8s-cmjkop-py/kernel.json.j2 2022-01-25 21:13:52.000000000 +0100
+++ jupyter-eg-kernel-k8s-cmjkop-r/kernel.json.j2 2022-02-16 12:22:23.610382929 +0100
@@ -15,8 +15,8 @@
     }
   },
   "argv": [
-    "python",
-    "-m", "ipykernel_launcher",
-    "-f", "/var/tmp/kernel-parm.json"
+    "R",
+    "--slave", "-e", "IRkernel::main()",
+    "--args", "/var/tmp/kernel-parm.json"
   ]
 }
```

```
diff -u jupyter-eg-kernel-k8s-cmjkop-py/meta.yaml jupyter-eg-kernel-k8s-cmjkop-r/meta.yaml
--- jupyter-eg-kernel-k8s-cmjkop-py/meta.yaml 2022-01-25 21:13:52.000000000 +0100
+++ jupyter-eg-kernel-k8s-cmjkop-r/meta.yaml 2022-02-16 12:20:57.500974886 +0100
@@ -1,5 +1,5 @@
---
-display_name: "Python on Kubernetes Operator"
+display_name: "R on Kubernetes Operator"
+features: "k8s-jupyter-operator-enabled"
+parameters:
+  display_name:
```

```

@@ -7,7 +7,7 @@
    definition:
      getter: shell
      exec:
-      - echo "Python on Kubernetes Operator $(date +%y%m%d%H%M%S)"
+      - echo "R on Kubernetes Operator $(date +%y%m%d%H%M%S)"
      display_name: "Display name of the kernel"
    k8s_env_module:
      type: str
@@ -20,9 +20,9 @@
    definition:
      getter: static
      default:
-      - "jupyter/datascience-notebook"
+      - "jupyter/r-notebook"
      values:
-      - "jupyter/datascience-notebook"
+      - "jupyter/r-notebook"
      display_name: "Image to run"
    limits:
      max_len: 1

diff -u jupyter-eg-kernel-k8s-cmjkop-py/templates/cmjk.yaml.j2\
jupyter-eg-kernel-k8s-cmjkop-r/templates/cmjk.yaml.j2
--- jupyter-eg-kernel-k8s-cmjkop-py/templates/cmjk.yaml.j2 2022-01-25 21:13:52.000000000 +0100
+++ jupyter-eg-kernel-k8s-cmjkop-r/templates/cmjk.yaml.j2 2022-02-16 12:24:25.375373991 +0100
@@ -9,6 +9,7 @@
    gid: gid
    kernel_id: kernel_id
    homedir: homedir
+   create_connection_file: true
    pod:
      volumes:
-      - name: homedir

```

After instantiating a kernel spec from the template, the R kernel is ready for use:

New kernel ×

Kernel name:

Display name of the kernel:

Environment module to load:

Image to run:

jupyter/r-notebook × ⌵

Image pull policy:

IfNotPresent × ⌵

Namespace for kernel:

Number of GPUs container can use:

Cancel

Create

Figure 6.3: Jupyter Kernel Creator: creating the IR kernel spec

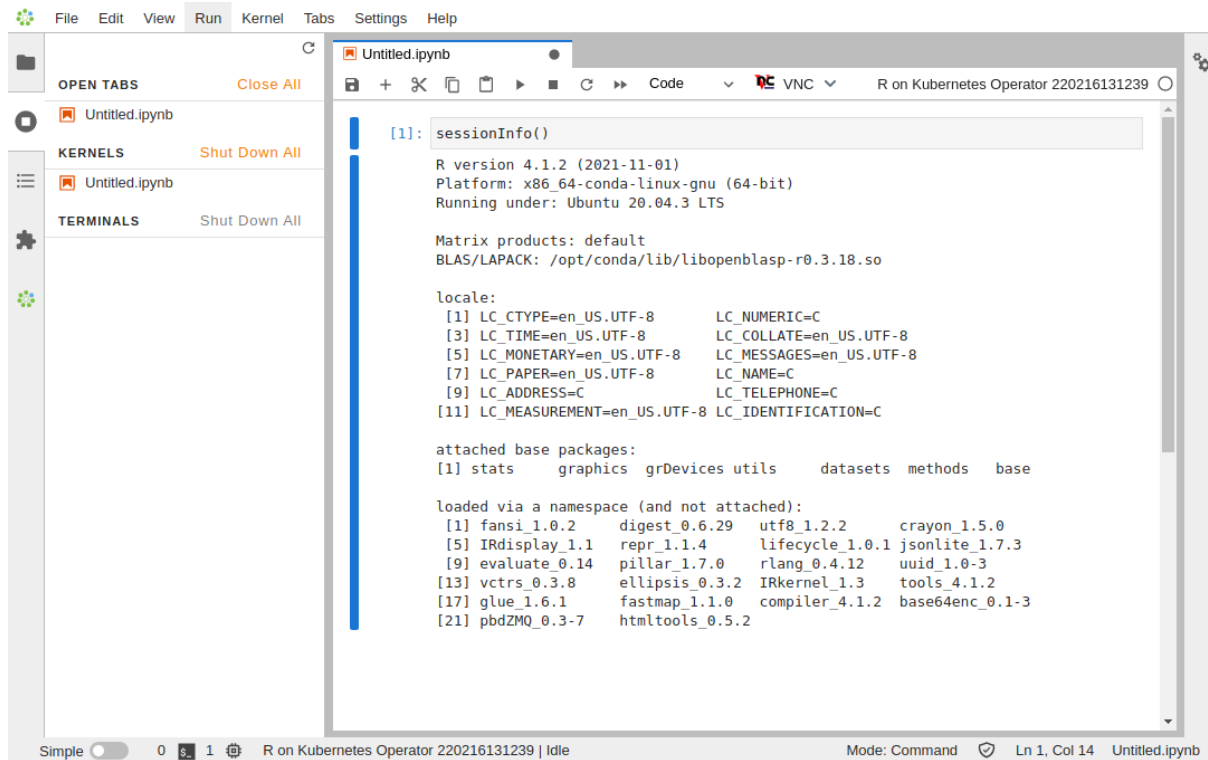


Figure 6.4: JupyterLab: running the IR kernel

6.3.8 Example: Letting Kubernetes Access Private Registries From The Kernel Template

To be able to pull images from private registries, Kubernetes needs to be instructed about the credentials to use.

Creating The Secret

This can be achieved by specifying the secret to `spec.imagePullSecrets` of the pod definition.

For Jupyter Kernel Operator this is `spec.pod.imagePullSecrets`:

Example

```
pod:
  containers:
  - name: kernel
    image: image
    ...
  imagePullSecrets:
  - name: regcred
    ...
```

Creating secrets can be carried out with `kubectl`

Example

```
kubectl create \
  --namespace alice-restricted \
  secret docker-registry regcred \
  --docker-server=<your-registry-server> \
  --docker-username=<your-name> \
  --docker-password=<your-pword> \
  --docker-email=<your-email>
```

More details about managing certificates can be found in the Kubernetes documentation at: <https://kubernetes.io/docs/tasks/configure-pod-container/pull-image-private-registry/>

Parameterizing The Secret

The name of the secret can be parameterized, so that users are allowed to select from secrets in their namespaces.

Figure 6.5: Jupyter Kernel Creator: secret selection

For parameterization, the `meta.yaml` and `kernel.json.j2` files must also then be modified:

Example

```
# cat meta.yaml
...
parameters:
...
  image_pull_secret_name:
    type: str
    definition:
      getter: static
      default: ""
      display_name: "Name of the secret to pull images"

# cat kernel.json.j2
```

```

...
  "metadata": {
    "process_proxy": {
      "class_name": "cm_jupyter_kernel_creator.eg_processproxies.k8scmjkop.KubernetesCMJupyterKernelOperator",
      "config": {
...
        "image_pull_policy": "{{ image_pull_policy[0] }}",
        "namespace": "{{ kubernetes_namespace }}",
        "image_pull_secret_name": "{{ image_pull_secret_name }}",
        "gpu_limit": {{ gpu_limit }}
      }
    }
  }
...

# cat templates/cmjk.yaml.j2
...
  pod:
...
    {%- if image_pull_secret_name %}
    imagePullSecrets:
    - name: {{ image_pull_secret_name }}
    {%- endif %}
    containers:
    - name: kernel
      image: {{ image }}
...

```

6.3.9 Example: Adding The PVC Parameter To The Kernel Template

The PVC that is to be mounted can be set in the template:

New kernel [X]

Kernel name:
jupyter-eg-kernel-k8s-cmjkop-py-pvc-1h93hfe4h

Display name of the kernel:
Python on Kubernetes Operator 230830173731

Environment module to load:
kubernetes

Image to run:
jupyter/datascience-notebook [X] [v]

Image pull policy:
IfNotPresent [X] [v]

Namespace for kernel:
cmsupport-restricted

Number of GPUs container can use:
0

PVC to mount:
pvc01 [X] [v]

Mountpoint to PVC:
/data

[Cancel] [Create]

Figure 6.6: Jupyter Kernel Creator, PVC selection.

The settings in `meta.yaml`, `kernel.json.j2` and `templates/cmjk.yaml.j2` for this are:

Example

```
# cat meta.yaml
...
parameters:
...
  pvc_name:
    type: list
    definition:
      getter: shell
      exec:
        - source /etc/profile.d/modules.sh
        - module load kubernetes
        - kubectl get pvc -o jsonpath="{range .items[*]}{.metadata.name}{'\n'}{end}"
      display_name: "PVC to mount"
    limits:
      max_len: 1
      min_len: 1
```

```

pvc_mountpoint:
  type: str
  definition:
    getter: static
    default: "/data"
    display_name: "Mountpoint to PVC"

# cat kernel.json.j2
...
"metadata": {
  "process_proxy": {
    "class_name": "cm_jupyter_kernel_creator.eg_processproxies.k8scmjkop.KubernetesCMJupyterKernelOperator",
    "config": {
...
      "image_pull_policy": "{{ image_pull_policy[0] }}",
      "namespace": "{{ kubernetes_namespace }}",
      "pvc_name": "{{ pvc_name[0] }}",
      "pvc_mountpoint": "{{ pvc_mountpoint }}",
      "gpu_limit": {{ gpu_limit }}
    }
  }
...

# cat templates/cmjk.yaml.j2
...
pod:
  volumes:
...
  {%- if pvc_name and pvc_mountpoint %}
  - name: pvc
    persistentVolumeClaim:
      claimName: {{ pvc_name }}
  {%- endif %}
  containers:
  - name: kernel
    image: {{ image }}
    imagePullPolicy: {{ image_pull_policy }}
    command: {{ command }}
...
  volumeMounts:
  - name: homedir
    mountPath: {{ homedir }}
  {%- if pvc_name and pvc_mountpoint %}
  - name: pvc
    mountPath: {{ pvc_mountpoint }}
  {%- endif %}
...

```

6.4 The NVIDIA GPU Operator

The NVIDIA GPU Operator (<https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/>) deploys and manages the software stack that exposes NVIDIA GPUs as first-class (standard) resources to a Kubernetes cluster. The components it manages include the NVIDIA Container Toolkit, the Kubernetes device plugin for NVIDIA GPUs, Node Feature Discovery, GPU Feature Discovery, the DCGM exporter, and the operator validator daemonsets that tie those pieces together.

The NVIDIA driver itself

- can either be installed by the operator as a container

- or be provisioned on the host by BCM. On DGX OS images the driver is always provisioned by the host.

In BCM 11.0 the NVIDIA GPU Operator is usually installed and configured using the `cm-kubernetes-setup` wizard (section 6.4.1).

However, it can also be installed directly with Helm on an already-running BCM Kubernetes cluster, as described in section 6.4.2. A more precise Helm installation might be preferred in order to avoid possible Kubernetes disruptions. The `cm-kubernetes-setup` wizard can, for example, cause such disruptions due to image updates and node reboots during its run.

For air-gapped clusters, the operator's container images and Helm chart are staged into the local Kubernetes container registry as part of the Kubernetes air-gapped installation. An air-gapped installation procedure for Kubernetes is described in Appendix B. The `cm-kubernetes-setup` wizard can then be used as described in section 6.4.1 to install the operator.

6.4.1 Installing The NVIDIA GPU Operator Using The Wizard

The `cm-kubernetes-setup` wizard (section 4.2.8) installs and configures the NVIDIA GPU Operator as part of the larger Kubernetes installation.

Enabling the GPU Operator is done in the Operators installation screen (figure 6.1).

Version selection, custom Helm values, and some basic configuration checklist options are carried out next:

Version Selection

After the operator-selection screen, the wizard presents a version-selection screen (figure 6.7) that lists the GPU Operator chart versions known to this release of BCM.

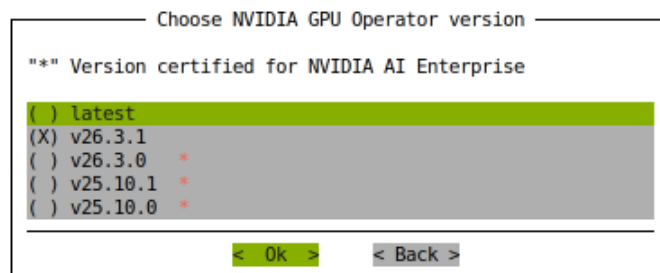


Figure 6.7: GPU operator version selection screen

Versions that are marked as certified for this BCM release are flagged with an asterisk. A `latest` entry is always available at the top of the list and tracks the newest version the chart repository publishes at install time.

Custom Helm Values (Optional)

The wizard then asks for an optional path to a YAML file that contains Helm values for the GPU Operator chart (<https://github.com/NVIDIA/gpu-operator/blob/main/deployments/gpu-operator/values.yaml> for the upstream reference).

If a valid path is supplied, then the contents of that file are merged into the values that the wizard deploys with, and the next screen (the basic configuration checklist) is skipped. This is the escape hatch for any configuration that the basic checklist does not expose, such as:

- MIG (Multi-Instance GPU) partitioning strategy and configuration
- time-slicing configuration for the device plugin
- custom resource-name labels for heterogeneous GPU fleets
- mirror or registry overrides beyond what the air-gapped deployment flow already handles

- any other chart value not surfaced as a checkbox in the wizard

If the path is left empty, the wizard proceeds to the basic configuration checklist, described next.

Basic Configuration Checklist Options

When no custom values YAML has been provided, the wizard presents a 3-item checklist for the most common configuration choices:

1. **Deploy and Manage the NVIDIA Container Toolkit using the operator** (`toolkit.enabled`, default: on)
 - on: The operator's toolkit daemonset installs and manages the NVIDIA Container Toolkit on GPU nodes. In turn, it uses CDI (Container Device Interface) Plugins, to hook the toolkit into the container runtime.
 - With older operator versions, or if CDI is explicitly disabled in the `cm-kubernetes-setup` configuration file, the containerd runtime class drop-ins are used instead of CDI for device injection.
 - off: The operator does not manage the toolkit and BCM provides the toolkit as a system package, and provides workload GPU access via containerd runtime-class drop-in configuration files.
2. **Deploy NVIDIA drivers as a container on the system** (`driver.enabled`, default: off)
 - off: CUDA drivers are expected to be present on the GPU node itself in order to present kernel devices for this hardware, and enable userspace/containerized-interactions with this hardware.
 - on: The operator deploys a driver container on each GPU node, installing the indicated kernel drivers onto the host, and maintaining dynamic provisioning of the hardware.
3. **Deploy Node Feature Discovery (NFD) plugin as a daemonset** (`nfd.enabled`, default: on)
 - on: The operator deploys NFD, which labels nodes based on detected hardware so that the operator's daemonsets schedule only to the nodes that they apply to.
 - off: This should likely be left on, unless this is undesirable in the Kubernetes deployment, or if a separately-managed NFD is already deployed in the cluster.

These 3 options are independent: any combination may be selected.

Post-install Configuration State

Once the wizard completes, the GPU Operator can be seen to be deployed with `helm`:

Example

```
helm list -n gpu-operator
```

NAME	NAMESPACE	REVISION	UPDATED ...	STATUS	CHART	APP VERSION
gpu-operator	gpu-operator	1	2026-04-...	deployed	gpu-operator-v26.3.0	v26.3.0

The wizard does not provide a dedicated way to reconfigure the GPU Operator. To change a configuration value after installation, either

- `cm-kubernetes-setup` is run again, but with a saved configuration file that reflects the new selections
- or the application's `config` attribute is edited directly and committed
- or `helm upgrade` is used against the running release.

Section 6.4.3 gives details on removing the NVIDIA GPU Operator.

6.4.2 Installing The NVIDIA GPU Operator On An Existing Kubernetes Cluster

Using `cm-kubernetes-setup` (section 6.4.1) is the recommended way to install the NVIDIA GPU Operator on both new and existing BCM Kubernetes clusters. Re-running `cm-kubernetes-setup` against an existing cluster with a configuration file that adds the operator is supported.

When re-running the wizard is not desired, for example because the cluster is already running production workloads and a direct Helm deployment is preferred, then the operator can be installed by the root user with Helm:

Example

```
helm install --wait -n gpu-operator --create-namespace \
  --version v26.3.0 \
  --set driver.enabled=false \
  --set toolkit.enabled=true \
  --set toolkit.env[0].name=CONTAINERD_CONFIG \
  --set toolkit.env[0].value=/cm/local/apps/containerd/var/etc/conf.d/nvidia-cri.toml \
  --set toolkit.env[1].name=RUNTIME_EXECUTABLE_PATH \
  --set toolkit.env[1].value=/cm/local/apps/containerd/current/bin/containerd \
  gpu-operator nvidia/gpu-operator
```

The values used in the preceding Helm session correspond to the defaults that the wizard applies.

Helm chart options for the NVIDIA GPU Operator are documented upstream at:

<https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/latest/getting-started.html#common-chart-customization-options>.

Deploying the Prometheus Operator Stack and Prometheus Adapter for scraping GPU metrics is optional and is covered separately in section 6.7.

The `--version` used in the preceding Helm session should be adjusted to match a version that is certified for the target BCM release. The set of certified versions is the same as that offered in the wizard's version-selection screen (figure 6.7).

6.4.3 Removing The NVIDIA GPU Operator

Whether the GPU operator has been installed using the BCM `cm-kubernetes-setup` wizard or whether it has been set with Helm directly, it must always be removed with Helm directly.

The GPU operator occupies the `gpu-operator` namespace:

Example

```
root@basecm11:~# helm list -n gpu-operator
NAME            NAMESPACE    REVISION ... STATUS    CHART                APP VERSION
gpu-operator    gpu-operator  1          ... deployed  gpu-operator-v26.3.0 v26.3.0
```

Removal is complete once the `gpu-operator` namespace no longer contains operator-managed pods:

Example

```
root@basecm11:~# helm uninstall -n gpu-operator gpu-operator
...
root@basecm11:~# kubectl get pods -n gpu-operator
No resources found in gpu-operator namespace.
root@basecm11:~# helm list -n gpu-operator
NAME NAMESPACE REVISION UPDATED STATUS CHART APP VERSION
root@basecm11:~#
```

6.4.4 Validating The NVIDIA GPU Operator

A pragmatic way to validate the NVIDIA GPU Operator is to check if the validator pods can be run. A Running status for the pods that are to have a GPU on them can be seen with:

Example

```
root@basecm11 ~# kubectl get pod -n gpu-operator -l app=nvidia-operator-validator -o wide
```

NAME	READY	STATUS	... IP	NODE	NOMINATED NODE	READINESS GATES
nvidia-operator-validator-2qvz6	1/1	Running	... 172.29.152.172	node001	<none>	<none>
nvidia-operator-validator-xkwwv	1/1	Running	... 172.29.112.154	node002	<none>	<none>

The preceding shows successfully-running pods. The log output should show all validations are successful:

Example

```
root@basecm11 ~# kubectl logs -n gpu-operator -l app=nvidia-operator-validator -c nvidia-operator-validator
all validations are successful
all validations are successful
```

6.4.5 Validating The NVIDIA GPU Operator In Detail

The set of pods associated with the NVIDIA GPU Operator can be examined in more detail. The following shows outputs from a GPU operator deployment that is working correctly:

Example

```
root@basecm11 ~# helm list -n gpu-operator
```

NAME	NAMESPACE	... STATUS	CHART	APP VERSION
gpu-operator	gpu-operator	... deployed	gpu-operator-v25.10.1	v25.10.1

```
root@basecm11 ~# kubectl get all -n gpu-operator -o wide
```

NAME	READY	STATUS	RESTARTS	...	NODE
pod/gpu-feature-discovery-gk892	1/1	Running	0	...	node001
pod/gpu-feature-discovery-rmkvj	1/1	Running	0	...	node002
pod/gpu-operator-798c6ddc97-lmclm	1/1	Running	0	...	basecm11
pod/gpu-operator-node-feature-discovery-master-6c65c99969-cjlpq	1/1	Running	0	...	basecm11
pod/gpu-operator-node-feature-discovery-worker-cgxzl	1/1	Running	0	...	node002
pod/gpu-operator-node-feature-discovery-worker-ds5mb	1/1	Running	0	...	basecm11
pod/gpu-operator-node-feature-discovery-worker-jf65c	1/1	Running	0	...	node001
pod/nvidia-container-toolkit-daemonset-ffbk7	1/1	Running	1 (46m ago)	...	node002
pod/nvidia-container-toolkit-daemonset-lqfkq	1/1	Running	0	...	node001
pod/nvidia-cuda-validator-pxs9b	0/1	Completed	0	...	node001
pod/nvidia-cuda-validator-v7gfz	0/1	Completed	0	...	node002
pod/nvidia-dcgm-exporter-bxjrv	1/1	Running	0	...	node001
pod/nvidia-dcgm-exporter-ql19z	1/1	Running	0	...	node002
pod/nvidia-device-plugin-daemonset-698hd	1/1	Running	0	...	node001
pod/nvidia-device-plugin-daemonset-xd4kj	1/1	Running	0	...	node002
pod/nvidia-device-plugin-validator-5crlc	0/1	Completed	0	...	node001
pod/nvidia-device-plugin-validator-wh27x	0/1	Completed	0	...	node002
pod/nvidia-operator-validator-2qvz6	1/1	Running	0	...	node001
pod/nvidia-operator-validator-xkwwv	1/1	Running	0	...	node002
...					

On this particular example cluster, there are two compute nodes with GPUs, and there is one control plane node without a GPU:

```
root@basecm11 ~# kubectl get nodes
NAME          STATUS    ROLES          AGE    VERSION
node001       Ready    worker         3h2m   v1.34.7
node002       Ready    worker         3h2m   v1.34.7
basecm11      Ready    control-plane, 3h2m   v1.34.7
master
```

Feature discovery pods: *Node Feature Discovery* (NFD, <https://intel.github.io/kubernetes-docs/nfd/index.html>) is an add-on that is initiated after the operator is installed. A master pod collects discovery information from the worker pods, and schedules more pods in case GPUs have been detected.

In the preceding GPU operator output,

- the master pod is running on node001 with the name:
gpu-operator-node-feature-discovery-master-6c65c99969-wtzcx
- the worker pods run on each node. For example, the worker pod for node002 is:
gpu-operator-node-feature-discovery-worker-z4skv

The output for the pods is not very verbose by default, but if more pods under the `nvidia-` namespace are scheduled on a node, besides the `gpu-operator-node-feature-discovery-*` pods, then that means that NFD has detected one or more GPUs.

For example, a GPU discovered on node001 results in a scheduling of the following pods on that node:

- container toolkit
- device plugin
- validator

Container toolkit pods: For nodes that have GPUs, the NVIDIA container toolkit installation pods are started. Pod logs show exactly what is being installed.

One of the requirements for the NVIDIA container toolkit installation pods is that the driver has to be in working order, or the init container `driver-validation` will fail. The following is the log from a successful installation:

Example

```
root@basecm11 ~# kubectl logs -f -n gpu-operator nvidia-container-toolkit-daemonset-ffbk7
Defaulted container "nvidia-container-toolkit-ctr" out of: nvidia-container-toolkit-ctr, driver-validation (init)
...
time="2022-12-06T14:31:36Z" level=info msg="Installing toolkit"
time="2022-12-06T14:31:36Z" level=info msg="Parsing arguments: [/usr/local/nvidia/toolkit]"
time="2022-12-06T14:31:36Z" level=info msg="Successfully parsed arguments"
time="2022-12-06T14:31:36Z" level=info msg="Installing NVIDIA container toolkit to '/usr/local/nvidia/toolkit'"
...
time="2022-12-06T14:31:36Z" level=info msg="Installing NVIDIA container toolkit config
      '/usr/local/nvidia/toolkit/.config/nvidia-container-runtime/config.toml'"
time="2022-12-06T14:31:36Z" level=info msg="Setting up runtime"
time="2022-12-06T14:31:36Z" level=info msg="Starting 'setup' for containerd"
time="2022-12-06T14:31:36Z" level=info msg="Parsing arguments: [/usr/local/nvidia/toolkit]"
time="2022-12-06T14:31:36Z" level=info msg="Successfully parsed arguments"
time="2022-12-06T14:31:36Z" level=info msg="Loading config: /runtime/config-dir/nvidia-cri.toml"
...
```

Device plugin pods: The device plugin pods are started up next. These have the toolkit as a requirement. If the toolkit is not in working order, then the init container toolkit-validation fails. The following is the log from a successful startup:

Example

```
root@basecm11 ~# kubectl logs -f -n gpu-operator          nvidia-device-plugin-daemonset-698hd
Defaulted container "nvidia-device-plugin-ctr" out of: nvidia-device-plugin-ctr, toolkit-validation (init)
2022/12/06 14:32:20 Loading NVML
2022/12/06 14:32:20 Starting FS watcher.
2022/12/06 14:32:20 Starting OS watcher.
2022/12/06 14:32:20 Retrieving plugins.
2022/12/06 14:32:20 No MIG devices found. Falling back to mig.strategy=&
2022/12/06 14:32:20 Starting GRPC server for 'nvidia.com/gpu'
2022/12/06 14:32:20 Starting to serve 'nvidia.com/gpu' on /var/lib/kubelet/device-plugins/nvidia-gpu.sock
2022/12/06 14:32:20 Registered device plugin for 'nvidia.com/gpu' with Kubelet
```

The pod log output suggests that the GPU is now registered with the Kubelet as a resource. This can be checked by querying the Node resource:

Example

```
root@basecm11 ~# kubectl describe node node001 | grep nvidia
nvidia.com/cuda.driver.major=520
nvidia.com/cuda.driver.minor=61
nvidia.com/cuda.driver.rev=05
nvidia.com/cuda.runtime.major=11
nvidia.com/cuda.runtime.minor=8
nvidia.com/gfd.timestamp=1670337142
nvidia.com/gpu.compute.major=7
nvidia.com/gpu.compute.minor=0
nvidia.com/gpu.count=1
nvidia.com/gpu.deploy.container-toolkit=true
nvidia.com/gpu.deploy.dcgmm=true
nvidia.com/gpu.deploy.dcgmm-exporter=true
nvidia.com/gpu.deploy.device-plugin=true
nvidia.com/gpu.deploy.driver=true
nvidia.com/gpu.deploy.gpu-feature-discovery=true
nvidia.com/gpu.deploy.node-status-exporter=true
nvidia.com/gpu.deploy.operator-validator=true
nvidia.com/gpu.family=volta
nvidia.com/gpu.machine=OpenStack-Nova
nvidia.com/gpu.memory=32768
nvidia.com/gpu.present=true
nvidia.com/gpu.product=Tesla-V100-SXM3-32GB
nvidia.com/mig.strategy=single
nvidia.com/run.ai-swap.enabled=false
nvidia.com/gpu:      1
nvidia.com/gpu:      1
...
```

Validator pods: If anything goes wrong with either the driver, toolkit, CUDA, or the plugin, then validator pods are a good place to start looking.

If all goes well, the main container outputs all validations are successful:

Example

```
root@basecm11 ~# kubectl logs -f -n gpu-operator nvidia-operator-validator-2qvz6
Defaulted container "nvidia-operator-validator" out of: nvidia-operator-validator, driver-validation (init),
toolkit-validation (init), cuda-validation (init), plugin-validation (init)
all validations are successful
```

It is possible for an init container to fail. The output for the container should then be checked.
The following shows output from successful init containers:

```
root@basecm11 ~# kubectl logs -f -n gpu-operator nvidia-operator-validator-2qvz6 -c driver-validation
running command chroot with args [/run/nvidia/driver nvidia-smi]
Tue Dec 6 15:32:14 2022
```

```
+-----+
| NVIDIA-SMI 520.61.05      Driver Version: 520.61.05      CUDA Version: 11.8      |
+-----+-----+-----+-----+-----+-----+
| GPU  Name          Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|                               |                      | MIG M. |
+-----+-----+-----+-----+-----+-----+
|   0   Tesla V100-SXM3...  On   | 00000000:00:06:0 Off |              0      |
| N/A   32C    P0    46W / 350W |      2MiB / 32768MiB |      0%      Default |
|                               |                      | N/A |
+-----+-----+-----+-----+-----+-----+

+-----+
| Processes: |
| GPU  GI    CI          PID    Type    Process name                      GPU Memory |
|      ID    ID                                   Usage     |
+-----+-----+-----+-----+-----+-----+
| No running processes found |
+-----+
```

```
root@basecm11 ~# kubectl logs -f -n gpu-operator nvidia-operator-validator-2qvz6 -c toolkit-validation
Tue Dec 6 14:32:16 2022
```

```
+-----+
| NVIDIA-SMI 520.61.05      Driver Version: 520.61.05      CUDA Version: 11.8      |
+-----+-----+-----+-----+-----+-----+
| GPU  Name          Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|                               |                      | MIG M. |
+-----+-----+-----+-----+-----+-----+
|   0   Tesla V100-SXM3...  On   | 00000000:00:06:0 Off |              0      |
| N/A   32C    P0    46W / 350W |      2MiB / 32768MiB |      0%      Default |
|                               |                      | N/A |
+-----+-----+-----+-----+-----+-----+

+-----+
| Processes: |
| GPU  GI    CI          PID    Type    Process name                      GPU Memory |
|      ID    ID                                   Usage     |
+-----+-----+-----+-----+-----+-----+
| No running processes found |
+-----+
```

```
root@basecm11 ~# kubectl logs -f -n gpu-operator nvidia-operator-validator-2qvz6 -c cuda-validation
time="2022-12-06T14:32:17Z" level=info msg="pod nvidia-cuda-validator-pxs9b is curently in Pending phase"
time="2022-12-06T14:32:22Z" level=info msg="pod nvidia-cuda-validator-pxs9b is curently in Pending phase"
```

```
time="2022-12-06T14:32:27Z" level=info msg="pod nvidia-cuda-validator-pxs9b is curently in Pending phase"
time="2022-12-06T14:32:32Z" level=info msg="pod nvidia-cuda-validator-pxs9b have run successfully"
```

```
root@basecm11 ~# kubectl logs -f -n gpu-operator nvidia-operator-validator-2qyz6 -c plugin-validation
time="2022-12-06T14:32:33Z" level=info msg="pod nvidia-device-plugin-validator-5crlc is curently in Pending phase"
time="2022-12-06T14:32:38Z" level=info msg="pod nvidia-device-plugin-validator-5crlc is curently in Pending phase"
time="2022-12-06T14:32:43Z" level=info msg="pod nvidia-device-plugin-validator-5crlc have run successfully"
```

This also explains where the pods earlier on came from, the ones marked with status Completed. They are used as part of certain validation steps.

Which init container prints out error messages should indicate where the problem lies—either with the CUDA drivers, or the toolkit, and so on. If the driver or toolkit is not validating correctly, then it may result in a lot of pods stuck in a Pending or an Init stage. Looking at what init container is associated with the stuck pod helps in diagnosing the problem.

DCGM exporter pods: These pods expose metrics endpoints for scraping, and can be considered less critical. They are involved in GPU metrics collection, and can be utilized with, for example, Prometheus Stack Operator, or the Prometheus Adapter, for *horizontal pod autoscaling* based on GPU metrics.

More information on the Prometheus Stack Operator and the Prometheus Adapter Operator can be found at:

<https://kb.brightcomputing.com/knowledge-base/the-nvidia-gpu-operator-with-kubernetes-on-a-bright-cluster>

6.4.6 Running A GPU Workload

A GPU workload can be run with the following configuration:

Example

```
root@basecm11 ~# cat << EOF > gpu.yaml
apiVersion: v1
kind: Pod
metadata:
  name: gpu-pod
spec:
  restartPolicy: Never
  containers:
  - name: cuda-container
    image: nvidia/cuda:11.0.3-base-ubuntu20.04
    command: ["nvidia-smi"]
    resources:
      limits:
        nvidia.com/gpu: 1
EOF
root@basecm11 ~# kubectl create -f gpu.yaml
pod/gpu-pod created
```

On a cluster with GPUs available, this pod should get scheduled, and should not stay stuck in the Pending phase.

The preceding example just invokes `nvidia-smi` in the container. The output can be viewed to confirm that it worked:

Example

```
root@basecm11 ~# kubectl logs -f gpu-pod
Tue Dec 6 15:08:03 2022
```

```

+-----+
| NVIDIA-SMI 520.61.05      Driver Version: 520.61.05      CUDA Version: 11.8      |
+-----+
| GPU  Name          Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|      Memory-Usage | GPU-Util  Compute M. |
|                                          MIG M. |
+-----+
|    0   Tesla V100-SXM3...  On   | 00000000:00:06:0 Off |              0      |
| N/A    34C    P0      47W / 350W |      2MiB / 32768MiB |      0%    Default   |
|                                          N/A |
+-----+

+-----+
| Processes: |
| GPU  GI    CI          PID    Type    Process name          GPU Memory |
|      ID    ID                                   Usage      |
+-----+
| No running processes found |
+-----+

```

6.5 The NVIDIA Network Operator

6.5.1 Installing The NVIDIA Network Operator

The upstream documentation for the NVIDIA Network Operator is quite extensive, and available at <https://docs.nvidia.com/networking/software/cloud-orchestration/index.html#network-operator>.

Going Through The Kubernetes Setup Wizard

The Network operator installation is also part of the initial Kubernetes cluster setup. One of the steps in the wizard asks the cluster administrator to select operators from a list. The Network operator is one of them. If the Network operator is selected (figure 6.8), then the setup wizard continues with some follow-up checks and questions.

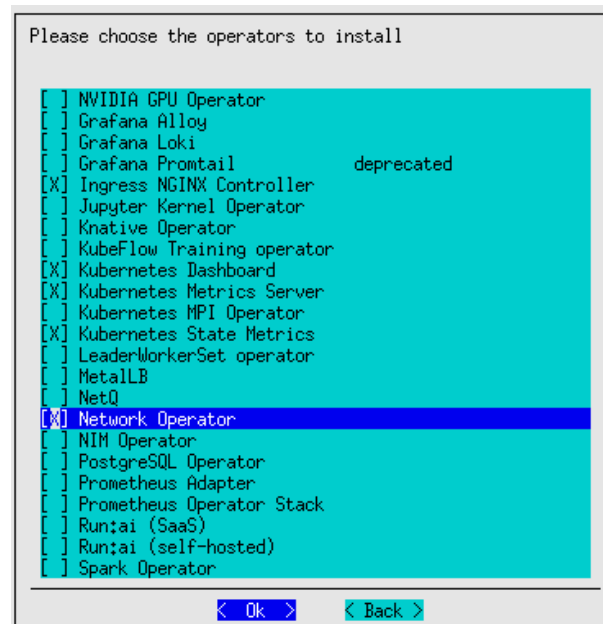


Figure 6.8: Kubernetes operators selection, with Network operator selected

The wizard then asks which version of the operator should be installed (figure 6.9):

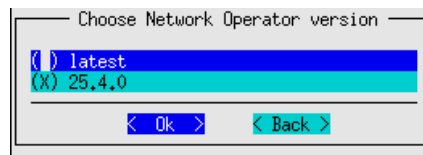


Figure 6.9: Network operator version selection screen

The wizard then prompts for a custom YAML configuration file to use for the Helm chart deployment (figure 6.10):

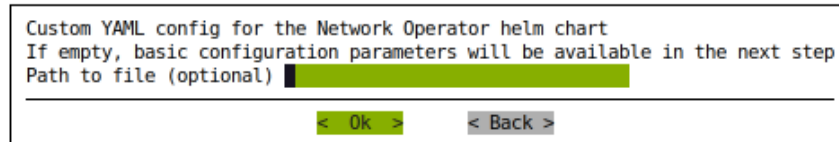


Figure 6.10: Network operator prompt for custom YAML configuration for Helm

If a custom YAML configuration file is not chosen, then options for a Helm chart can be set (figure 6.11):

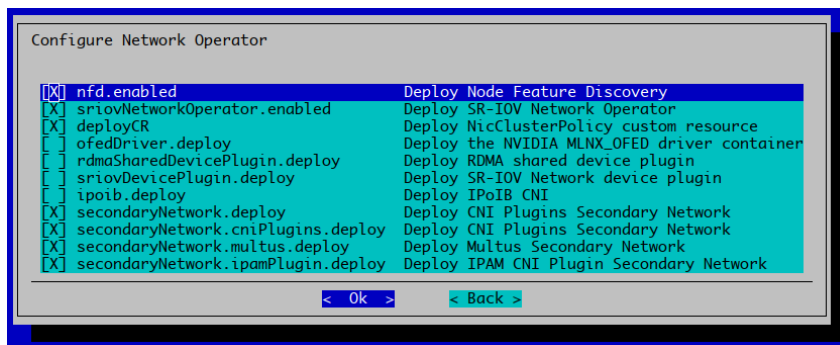


Figure 6.11: Network operator options for Helm chart prompt.

Result Of The Network Operator Helm Chart

If all goes well, then the Helm chart should be deployed successfully. Helm output for Kubernetes 1.34 should show something similar to:

Example

```
root@basecm11 ~# helm list -A -a
```

NAME	NAMESPACE	...STATUS	CHART	APP VERSION
calico	tigera-operator	...deployed	tigera-operator-v3.31.0	v3.31.0
grafana	grafana	...deployed	grafana-operator-v5.20.0	v5.20.0
kgateway	kgateway-system	...deployed	kgateway-v2.3.0	v2.3.0
kgateway-crds	kgateway-system	...deployed	kgateway-crds-v2.3.0	v2.3.0
kube-prometheus-stack	prometheus	...deployed	kube-prometheus-stack-79.5.0	v0.86.2
kube-state-metrics	kube-system	...deployed	kube-state-metrics-6.4.1	2.17.0
kyverno	kyverno	...deployed	kyverno-3.6.0	v1.16.0
kyverno-policies	kyverno	...deployed	kyverno-policies-3.6.0	v1.16.0
local-path-provisioner	cm	...deployed	local-path-provisioner-0.0.32	v0.0.32
metallb	metallb-system	...deployed	metallb-0.15.3	v0.15.3
metrics-server	kube-system	...deployed	metrics-server-3.13.0	0.8.0
permissions-manager	cm	...deployed	cm-kubernetes-permissions-manager-0.6.6	0.6.6

```
prometheus-adapter    prometheus    ...deployed    prometheus-adapter-5.2.0    v0.12.0
```

6.6 The NVIDIA NetQ Operator

An external NetQ server can be configured outside of Kubernetes (section 3.10 of the *Administrator Manual*) or it can be configured with Kubernetes (section 6.6.1 in the current manual).

6.6.1 NVIDIA NetQ Operator Installation

Prerequisites

- Only NetQ 4.15 and later NetQ 4 versions are supported, and only for Ubuntu 24.04 and later.
 - For BCM 11, installation using the `cm-kubernetes-setup` wizard for NetQ 5 versions is not supported at the time of writing (November 2025). If a NetQ 5 installation is required, then it can be carried out with the help of BCM support. Getting support is described in section 14.2 of the *Administrator Manual*.
- The number of NetQ nodes can be 1 or 3 nodes only. These must be regular compute nodes.
- The NetQ nodes require at least 250 GiB of disk space to be available.
- The NetQ nodes require at least 64 GiB of memory.
- The NVIDIA Base Command Manager default ports cannot be used.
- Kyverno is not supported in combination with NetQ at the time of writing (March 2024) of this section.

The default ports can be changed on the active and passive head nodes as follows.

```
cm-cmd-ports --http 8083 --https 8084 && systemctl restart cmd
```

Going Through The Kubernetes Setup Wizard

The NetQ installation is part of the initial Kubernetes cluster setup. One of the steps in the wizard asks the cluster administrator to choose operators from a list of operators. NetQ is one of the possible operators. If the NetQ operator is selected (figure 6.12), then the setup wizard continues with some follow-up checks and questions.

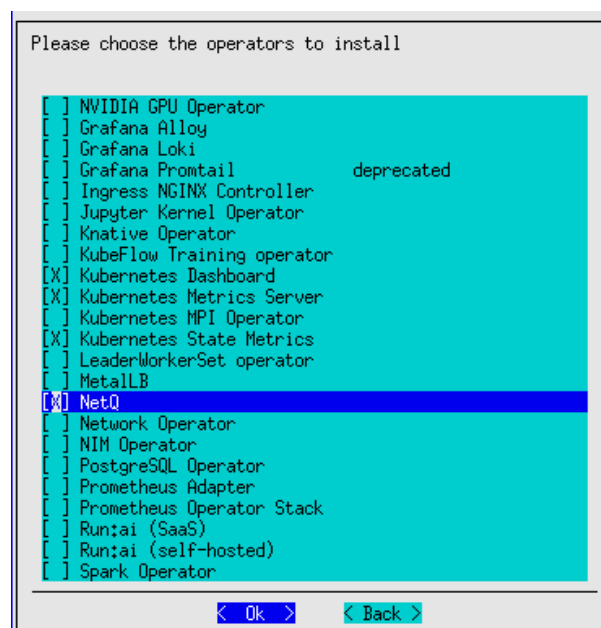


Figure 6.12: Kubernetes Operator Selection

Node selection asks which nodes are to be assigned to NetQ (figure 6.13). The options are:

- for a single-deployment (this is NetQ terminology), one node should be selected
- For a cluster-deployment (also NetQ terminology), three nodes should be selected

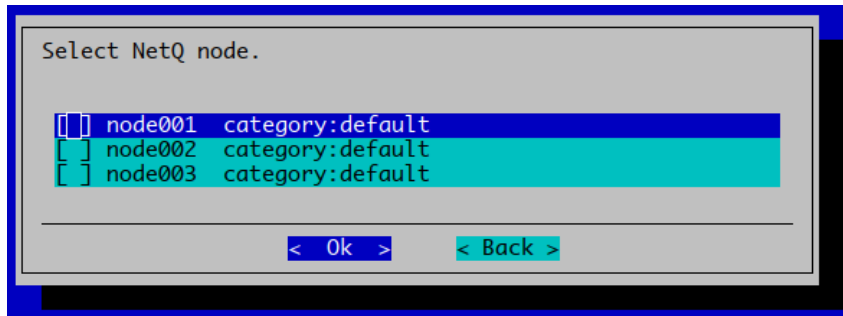


Figure 6.13: NetQ nodes selection

After node selection, the nodes are inspected to see if they meet the prerequisites for running NetQ. One potential issue it detects is if NVIDIA Base Command Manager is already running on the default ports. It suggests a `cm-cmd-ports` command to execute to fix that, (figure 6.14). NetQ prerequisites and how to carry out changing the default ports is described further in section 6.6.1. After carrying out the port changes, the wizard must be restarted from the beginning.

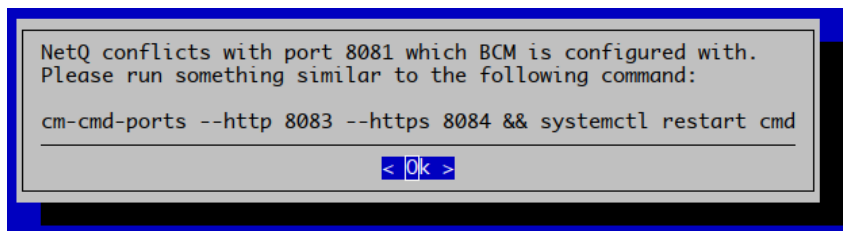


Figure 6.14: Warning presented if NVIDIA Base Command Manager still uses default port 8081.

If all prerequisites are met, then the next dialog asks for a few files. These have to be provided by NetQ and stored somewhere on the active head node (figure 6.15).

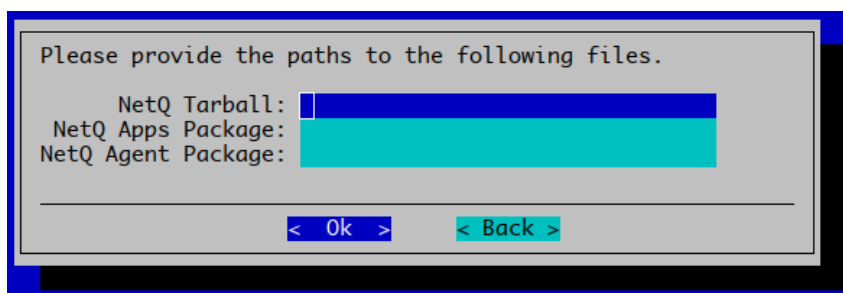


Figure 6.15: Dialog asking for NetQ tarball and packages for agent and apps.

When the required files are downloaded, and saved to, for example, `/root`:

Example

```
root@basecm11 ~# ls -alh | grep -i netq
-rw-r--r-- 1 root root 35M Jan 9 09:05 netq-agent_4.9.0-ub20.04u45~1703950858.128b0741e_amd64.deb
-rw-r--r-- 1 root root 32M Jan 9 09:05 netq-apps_4.9.0-ub20.04u45~1703950858.128b0741e_amd64.deb
```

```
-rw-r--r-- 1 root root 13G Jan 9 09:09 NetQ-bcm-4.9.0-SNAPSHOT.tgz
```

the dialog can then be filled in (figure 6.16):

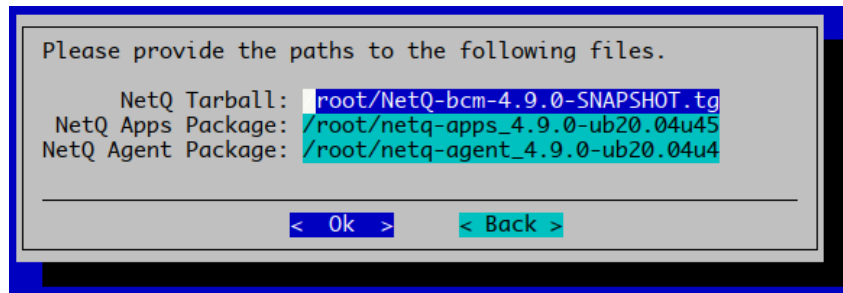


Figure 6.16: Filled-in dialog asking for NetQ files.

Depending on whether a single-node or cluster (three-node) setup is being done, the next dialog will only be shown in the case of a cluster install, and asks for a virtual IP to be used for the NetQ LoadBalancer (figure 6.17).

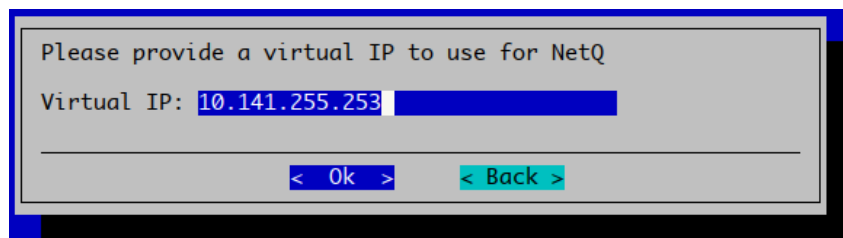


Figure 6.17: NetQ prompt for virtual IP to use for load balancing

The dialog by default suggests an IP address that is not in use, according to NVIDIA Base Command Manager's internal database. The address is part of the chosen internal network for the Kubernetes pod network.

After the wizard completes its input stage, the actual setup is executed. The setup can take significantly more time than the wizard input stage. The amount of time taken is largely dependent on file I/O speed—bigger files require some time to be synchronized to the appropriate nodes. In addition, NetQ installation itself can take around an hour to finish deploying. The deployment of NetQ is almost the very last step of the setup process. For a successful installation, the last stage of the output displayed looks similar to figure 6.18:

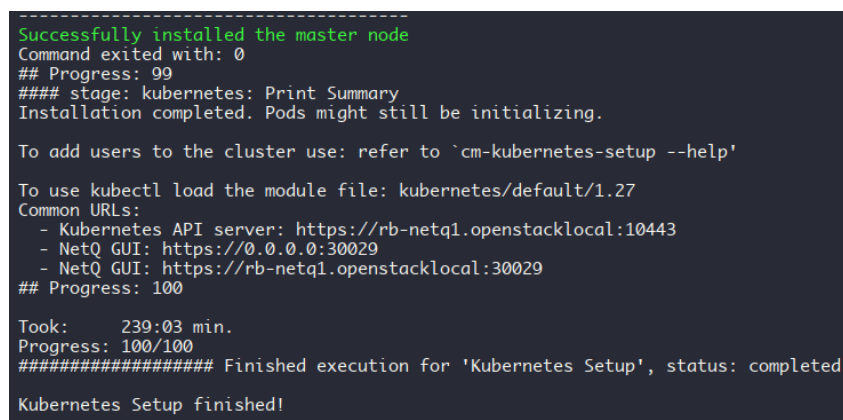


Figure 6.18: Kubernetes with NetQ installation finished, and relevant URLs on display

6.6.2 Accessing The NVIDIA NetQ Operator UI

In section 6.6.1 a NetQ deployment is described.

If the setup is for a single-node deployment, then a NodePort service is used to expose the NetQ UI. The URL is printed at the end of the setup (figure 6.18), but it can also be found by one of the following methods:

- via kubectl

```
root@basecm11 ~# module load kubernetes/default/1.27.11-1.1
root@basecm11 ~# kubectl get svc -l app=netq-gui -n netq
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
netq-gui      NodePort    10.150.11.218  <none>         80:30029/TCP     17m
```

- via searching back in the Kubernetes setup file:

```
root@basecm11 ~# tac /var/log/cm-kubernetes-setup.log | grep -m 1 "NetQ GUI:"
I 24-02-29 13:40:46 | cmsetup.plugins.kubernetes.stages.stages | - NetQ GUI: \
https://basecm11.openstacklocal:30029
```

If the setup is for a cluster deployment, then the virtual IP address also exposes the NetQ GUI on the default HTTPS port. For the cluster, that is: <https://10.141.255.253>. This virtual IP address is the one that is set during installation (figure 6.17). Figure 6.19 shows that IP address in use:

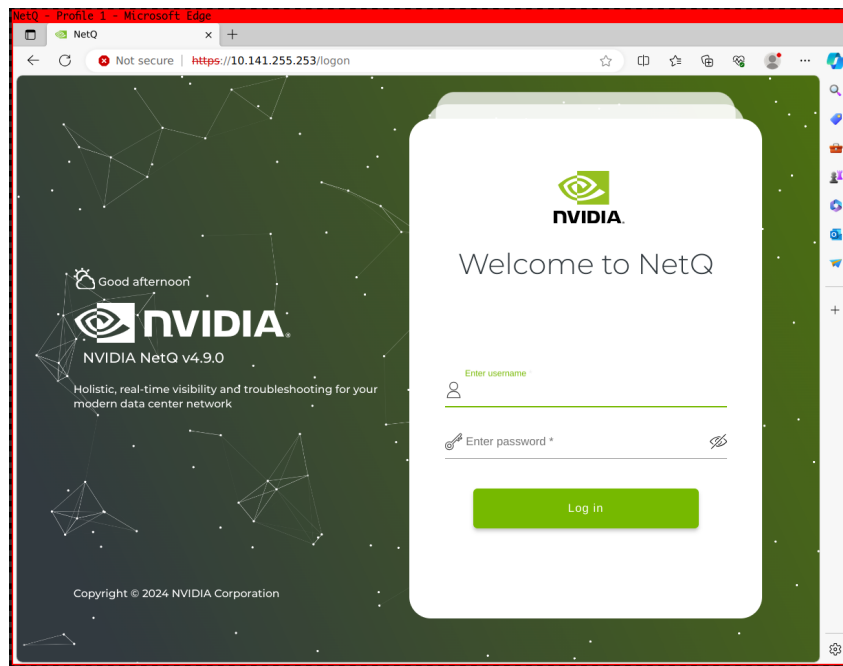


Figure 6.19: NetQ GUI screenshot

6.7 The Prometheus Operator Stack

6.7.1 Exporting And Reusing Grafana Dashboards

After a dashboard has been created, the user may wish to export it for backup purposes, or to reuse it in another Grafana instance.

Exporting A Dashboard

Exporting the dashboard as a JSON file can be carried out by following the Grafana documentation instructions at:

<https://grafana.com/docs/grafana/latest/dashboards/share-dashboards-panels/#export-a-dashboard-as-json>

Storing A Dashboard

The exported JSON file can be added to a Git repository, or stored in a ConfigMap. A ConfigMap can be created and labeled for Grafana with:

Example

```
kubectl create configmap my-grafana-dashboard --from-file=my-dashboard-1739813945293.json -n default
kubectl label configmap my-grafana-dashboard grafana_dashboard='1' -n default
```

By default, Grafana automatically loads dashboards when using ConfigMaps containing the

```
grafana_dashboard='1'
label in any namespace.
```

6.8 The Run:ai Operator

Run:ai (<https://run.ai>) is a way to manage resources and workloads on GPUs that are in a cluster.

Run:ai can be installed in two ways:

- SaaS (or Classic): where the Run:ai control plane is in a cloud
- self-hosted: where the Run:ai control plane is on-premises.

These installation types are described at <https://docs.run.ai/latest/admin/runai-setup/installation-types/>.

The installation type can be selected and configured by the cluster administrator during Kubernetes setup, when `cm-kubernetes-setup` is run, and a screen comes up for the selection of operator packages (figure 6.1).

The Run:ai deployment is carried out on top of BCM Kubernetes. Run:ai consists of two components: the datascience GPU cluster (BCM cluster in BCM terms) and the control plane.

- For a SaaS deployment, the control plane runs in the Run:ai cloud.
- For a self-hosted deployment, the control plane is installed on the BCM cluster. The self-hosted deployment comes in two variants:
 1. Connected: with this, download is allowed from the internet, but upload is not allowed.
 2. Air-gapped: with this, the cluster is completely disconnected from the internet.

All installation options are possible on top of a BCM Kubernetes cluster.

Installation can be carried out with the `cm-kubernetes-setup` TUI, where Run:ai is installed as a Kubernetes operator. or with the Kubernetes wizard GUI in Base View using the navigation path:

```
Containers > Kubernetes > Kubernetes Management > Launch Wizard
```

6.8.1 Run:ai Operator Installation Prerequisites

Run:ai can be installed as a SaaS installation or as a self-hosted installation. Either installation can be carried out as part of the `cm-kubernetes-setup` procedure (section 4.2.8). Credentials appropriate for the installation are needed.

- Page 130 covers prerequisites and credentials for the SaaS installation.
- Page 130 covers prerequisites and credentials for the self-hosted installation.

Prerequisites For SaaS Installation

The following prerequisites must be met for a Kubernetes and Run:ai SaaS installation:

- The cluster must be running an updated BCM. For a cluster running on Ubuntu this typically means first running: `apt update`; `apt upgrade`
- The cluster must allow outbound traffic to the Run:ai cloud.
- The user must have the following Run:ai credentials:
 - tenant name
 - application secret
 - username and password for the admin user
- A hostname must be available that can be used instead of the cluster IP external address
- A DNS entry for the hostname must exist that resolves to the cluster IP address.
- There must be a trusted certificate and key file for the hostname

The hostname is used by clients, such as the primary Run:ai web interface, to connect to Run:ai APIs on the cluster through a Gateway (served by Kgateway) over HTTPS.

Prerequisites For Self-hosted Installation

The following prerequisites must be met for a Kubernetes and Run:ai self-hosted installation:

- The cluster must be running an updated BCM. For a cluster running on Ubuntu this typically means first running: `apt update`; `apt upgrade`
- Run:ai JFrog credentials must be available. These can be as a base64 format string or as a file
- The local CA certificate path (a .crt file) is optional, but required for self-signed domain certificates. The CA certifies the control plane domain is trusted.
- A hostname must be available that can be used instead of the cluster IP external address.
- A DNS entry for the hostname must exist that resolves to the cluster IP address.
- The domain certificate (a .crt or .pem file) must be available.
- The domain key (a .key file) must be available.

By default, the Run:ai dashboard and the Run:ai APIs can be accessed with the browser at the domain. .

Validation Checks During Run:ai Setup

The `cm-kubernetes-setup` wizard carries out several validation checks to ensure that Run:ai deploys successfully. The checks take place during both the questions phase and during the actual deployment (setup phase).

Required field validation: The wizard checks that the following fields are not empty:

- Run:ai JFrog credentials: These must be provided as a file or as a base64 encoded string.
- Run:ai Control plane domain name: This is in the form of an FQDN.
- Run:ai Domain certificate path pair: These are .crt/.pem and .key files.

Note: The wizard allows the local CA certificate field to be empty, but this should be used with caution. A local CA is mandatory in almost all cases when dealing with self-signed certificates. The system will work without a local CA certificate only for publicly-signed domain certificates (that is, of the kind used for public websites).

Question phase validation: During the questions phase, the wizard performs these additional checks:

- **Domain name resolution:** The provided domain name is resolved to an IP address. A warning is displayed if resolution fails.
- **Certificate verification:** If a local CA is provided, then the wizard executes:
`openssl verify -CAfile <local CA .crt file> <local .crt file>`
 A warning is displayed if verification fails.
- **Domain FQDN validation:** The wizard checks if the provided domain FQDN is present in the certificate's Subject Alternative Name (SAN) list, including any wildcards present in the SAN list.
- **IP routability check:** The wizard verifies if the IP address that the FQDN resolved to is routable to one of the nodes within the BCM cluster.

Note: The IP routability check is present since BCM 11.25.08. It is however disabled if MetalLB is used with Run:ai, because it is inappropriate. This is because the IP address is not yet in use during the questions phase in that case.

Consistency validation: Starting with BCM 11.25.08, a consistency check ensures that:

- If a local CA certificate is provided, but a custom CA is not enabled in the Helm values, then the setup aborts.
- If a local CA certificate is NOT provided but a custom CA is enabled in the Helm values, then the setup aborts.

6.8.2 Installing Run:ai Operator

The Run:ai operator deploys the cluster-installer Helm chart package. The Helm status can be checked with:

Example

```
root@basecm11 ~# helm list -n runai
```

NAME	NAMESPACE	...	STATUS	CHART	APP VERSION
cluster-installer	runai	...	deployed	cluster-installer-2.8.8	0.0.1

```
root@basecm11 ~# helm history -n runai runai-cluster
```

REVISION	...	STATUS	CHART	APP VERSION	DESCRIPTION
1	...	deployed	runai-cluster-2.13.7		Install complete

```
root@basecm11 ~# kubectl get all -n runai
```

NAME	READY	STATUS	RESTARTS	AGE
pod/cluster-installer-deployment-5f4c4cbf4c-82gmx	1/1	Running	0	5m9s

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/cluster-installer-service	ClusterIP	10.150.117.247	<none>	8080/TCP	5m9s

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
deployment.apps/cluster-installer-deployment	1/1	1	1	5m9s

NAME	DESIRED	CURRENT	READY	AGE
replicaset.apps/cluster-installer-deployment-5f4c4cbf4c	1	1	1	5m9s

Removing The Run:ai Operator

The Run:ai operator can be removed via Helm:

Example

```
[root@basecm11 ~]# helm uninstall cluster-installer -n runai
```

Uninstalling the Kubernetes cluster automatically cleans up everything associated with it.

6.8.3 Completing The Run:ai Installation

When `cm-kubernetes-setup` is run, a summary is shown at the end of the Run:ai operator installation.

Example

```
[root@basecm11 ~]# cm-kubernetes-setup
...
Installing NVIDIA GPU Operator...
Installing Prometheus Operator Stack...
Installing Prometheus Adapter...
Installing Run:ai (SaaS)...
Installing Kubernetes State Metrics...
...
## Progress: 99
#### stage: kubernetes: Print Summary
Installation completed. Pods might still be initializing.
```

To add users to the cluster use: refer to ``cm-kubernetes-setup --help``

```
To use kubectl load the module file: kubernetes/default/1.32
Common URLs:
- Kubernetes API server: https://basecm11.cm.cluster:10443
## Progress: 100
Disconnect from cluster.
```

```
Took:      19:15 min.
Progress: 100/100
```

Whether SaaS or self-hosted has been chosen as an operator is indicated in the preceding session by the line:

```
Installing Run:ai (SaaS)...
OR Installing Run:ai (self-hosted)...
```

6.8.4 Post-installation

The SaaS OIDC configuration in NVIDIA Base Command Manager version 10.23.11 and later uses an interactive wizard. The wizard is accessed and run as follows:

- An SSH connection is made to the active head node
- `cm-kubernetes-setup` is run, and “Enable Run:ai Config (Configure Run:ai CLI binary and OIDC settings)” is chosen (figure 6.20).
- The steps of the wizard are followed. The wizard takes care of configuring the Kubernetes API server.
- The configuration change may take a minute to restart the Kubernetes API server, and may result in a minute of downtime.

- The HTTPS certificate can now be checked to see if it is working correctly

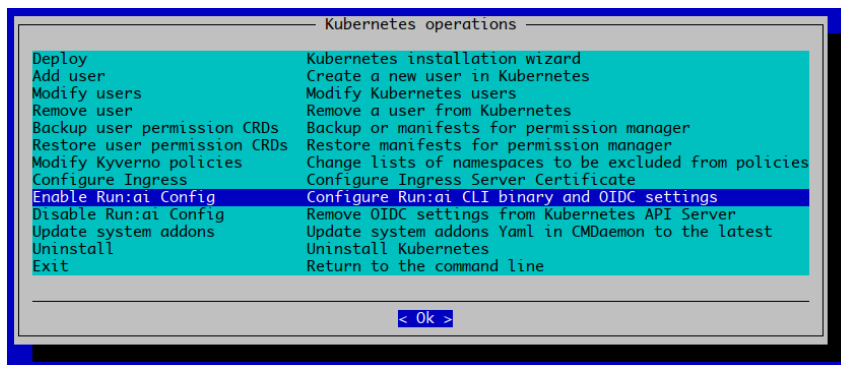


Figure 6.20: Option to automatically add Run:ai configuration to Kubernetes API server and installing the correct runai binary on the system

The most important aspect is configuring Researcher Access Control in BCM: <https://docs.run.ai/latest/admin/runai-setup/authentication/researcher-authentication/>.

This includes going to cmsh and configuring the Kubernetes API server with additional OIDC parameters.

The runai binary can be downloaded in various ways, the Run:ai environment has an option where the binary can be downloaded from the cluster itself. The binary can be copied to /usr/bin and made executable by the system administrator.

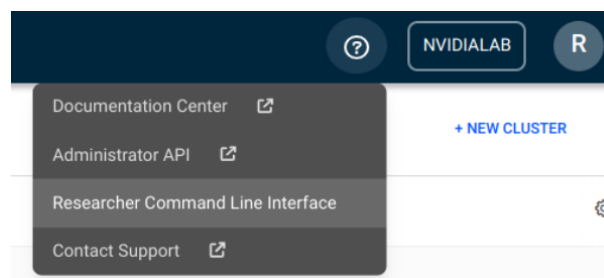


Figure 6.21: runai binary download

6.9 Kubernetes Spark Operator

Using the Kubernetes Spark Operator is a simpler alternative to using the spark-submit tool for job submission.

6.9.1 Installing The Kubernetes Spark Operator

The Kubernetes Spark Operator can be installed as a part of the cm-kubernetes-setup procedure (section 4.2.8), which eventually leads to a display listing the operator packages that may be installed (figure 6.1).

The Kubernetes Spark Operator can alternatively be installed later on using the OS package manager and Helm:

```
[root@basecm11 ~]# yum install cm-kubernetes-spark-operator -y
[root@basecm11 ~]# helm install cm-kubernetes-spark-operator \
  /cm/shared/apps/kubernetes-spark-operator/current/helm/spark-operator-*.tgz
```

The Kubernetes Spark Operator can be removed with:

Example

```
[root@basecm11 ~]# helm uninstall cm-kubernetes-spark-operator
```

The operator installation state can be verified with `--list-operators`:

Example

```
[root@basecm11 ~]# cm-kubernetes-setup --list-operators
...
OPERATOR_: api_available_
cm-jupyter-kernel-operator      : 0
cm-kubernetes-postgresql-operator : 0
cm-kubernetes-spark-operator    : 1
...
```

The Helm status can be checked with, for example:

Example

```
[root@basecm11 ~]# helm list
NAME                                NAMESPACE ... STATUS    CHART                                APP VERSION
cm-kubernetes-spark-operator        default    ...  deployed  spark-operator-1.0.8  v1beta2-1.2.0-3.0.0
[root@basecm11 ~]# helm status cm-kubernetes-spark-operator
NAME: cm-kubernetes-spark-operator
LAST DEPLOYED: Mon Sep 19 11:17:21 2022
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
[root@basecm11 ~]#
```

The Permission Manager (section 4.15) and PodSecurityPolicy (PSP, section 4.10.2) must both be enabled for the cluster, before allowing a user to create resources in the Kubernetes cluster in their namespace:

Example

```
[root@basecm11 ~]# cm-kubernetes-setup --psp
```

The user `alice` can be allowed to use the Spark operator, and allowed to run a process as any UID in the pod:

Example

```
[root@basecm11 ~]# cm-kubernetes-setup --add-user alice --operators cm-kubernetes-spark-operator \
--allow-all-uids
```

The Kubernetes Spark operator Helm chart creates a CRD that can be used in the Kubernetes API.

For Alice, the CRD is available and can be used with a Spark operator YAML, to build a Spark application carry out a pi run in the restricted namespace.

6.9.2 Example Spark Operator Run: Calculating Pi

Continuing on with the user alice of the preceding section, a YAML file based on the specification at <https://github.com/GoogleCloudPlatform/spark-on-k8s-operator/blob/master/examples/spark-py-pi.yaml> can be used:

Example

```
[root@basecm11 ~]# su - alice
[alice@basecm11 ~]$ module load kubernetes
[alice@basecm11 ~]$ cat <<EOF > pi-spark.yaml
apiVersion: "sparkoperator.k8s.io/v1beta2"
kind: SparkApplication
metadata:
  name: pyspark-pi
spec:
  type: Python
  pythonVersion: "3"
  mode: cluster
  image: "gcr.io/spark-operator/spark-py:v3.1.1"
  imagePullPolicy: Always
  mainApplicationFile: local:///opt/spark/examples/src/main/python/pi.py
  sparkVersion: "3.1.1"
  restartPolicy:
    type: OnFailure
    onFailureRetries: 3
    onFailureRetryInterval: 10
    onSubmissionFailureRetries: 5
    onSubmissionFailureRetryInterval: 20
  driver:
    cores: 1
    coreLimit: "1200m"
    memory: "512m"
    labels:
      version: 3.1.1
      serviceAccount: spark
  executor:
    cores: 1
    instances: 1
    memory: "512m"
    labels:
      version: 3.1.1
EOF
[alice@basecm11 ~]$ kubectl apply -f pi-spark.yaml
sparkapplication.sparkoperator.k8s.io/pyspark-pi created
[alice@basecm11 ~]$ kubectl get pods
NAME                READY   STATUS             RESTARTS   AGE
pyspark-pi-driver   0/1     ContainerCreating   0           1s
[alice@basecm11 ~]$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
pyspark-pi-driver   1/1     Running   0           3s
[alice@basecm11 ~]$ kubectl get sparkapplications
NAME                AGE
pyspark-pi          7s
[alice@basecm11 ~]$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
pyspark-pi-driver   1/1     Running   0           14s
```

```
pythonpi-e768128383a881b3-exec-1 0/1 ContainerCreating 0 0s
[alice@basecm11 ~]$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
pyspark-pi-driver                  0/1     Completed 0           34s
pythonpi-e768128383a881b3-exec-1 0/1     Terminating 0           20s
[alice@basecm11 ~]$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
pyspark-pi-driver                  0/1     Completed 0           36s
```

Instead of tracking the pod with:

```
kubectl get pods
```

as in the preceding session, or with the more convenient:

```
watch kubectl get pods
```

the pod could be tracked with the `-f` | `--follow` option to stream the driver logs:

Example

```
[alice@basecm11 ~]$ kubectl logs pyspark-pi-driver -f
```

To get intended output of the pi run—the calculated value of pi—it is sufficient to grep the log as follows:

Example

```
[alice@basecm11 ~]$ kubectl logs pyspark-pi-driver | grep ^Pi
Pi is roughly 3.148800
```

After the pi run has completed, the resources can be removed from the namespace:

```
[alice@basecm11 ~]$ kubectl delete -f pi-spark.yaml
sparkapplication.sparkoperator.k8s.io "pyspark-pi" deleted
```

```
[alice@basecm11 ~]$ kubectl get pods
No resources found in alice-restricted namespace.
[alice@basecm11 ~]$ kubectl get sparkapplications
No resources found in alice-restricted namespace.
```

6.10 The Zalando Postgres Operator

6.10.1 Installing The Zalando Postgres Operator

The upstream documentation for the Postgres Operator is at <https://operatorhub.io/operator/postgres-operator>.

Going Through The Kubernetes Setup Wizard

The Postgres operator installation is part of the initial Kubernetes cluster setup. Its installation is one of the options made available to the cluster administrator during the selection of operator packages (figure 6.1).

The wizard later asks which version of the Postgres operator should be installed (figure 6.22):

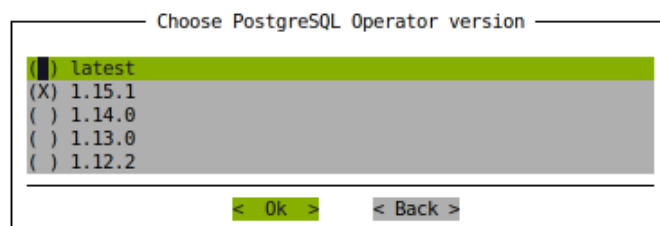


Figure 6.22: Postgres operator version selection screen

6.11 The NVIDIA NIM Operator

6.11.1 Installing The NVIDIA NIM Operator

The upstream documentation for the NIM Operator is at <https://docs.nvidia.com/nim-operator/2.0.0/>

Going Through The Kubernetes Setup Wizard

The NIM operator installation is part of the initial Kubernetes cluster setup. Its installation is one of the options made available to the cluster administrator during the selection of operator packages (figure 6.1).

7

Kubernetes On Edge

How edge sites can be configured is described in Chapter 2 of the *Edge Manual*.

If there are BCM Edge sites configured in the cluster, then the Kubernetes setup prompts the user with edge sites that Kubernetes can be deployed on.

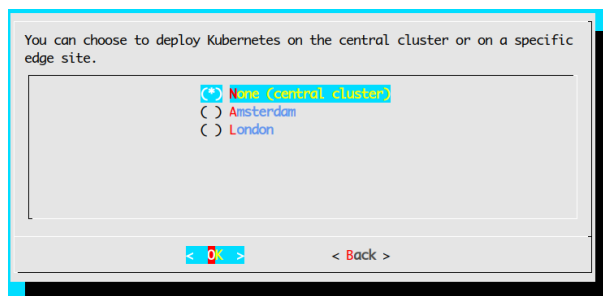


Figure 7.1: cm-kubernetes-setup prompting for edge sites.

If an edge site is selected, then the rest of the wizard prompts only for nodes available within that edge site; prompts only for the associated network interfaces; and so on.

7.1 Flags For Edge Installation

Edge directors often lack high-bandwidth connectivity to the central head node, or they often may benefit from coming up as quickly as possible. It can therefore sometimes be useful to skip stages of the setup.

Running `cm-kubernetes-setup --help` displays some additional flags that allow some setup stages, that bring up a cloud director, to be skipped explicitly:

```
cm-kubernetes-setup --help
...
installing Kubernetes clusters:
  Flags for installing or managing Kubernetes clusters

--skip-package-install
    Skip the package installation steps. Ignores skip_packages
    flags in the config.
--skip-reboot
    Skip the reboot steps.
--skip-image-update
    Skip the image update steps.
--skip-disksetup-changes
    Never change the disk-setup. Use this flag if you manually
    configure a partition or device for docker thin pool devices
    for example.
```

...

7.1.1 Speeding Up Kubernetes Installation To Edge Nodes With The `--skip-*` Flags: Use Cases

Explanations and use cases for these flags are given in the following table:

Flag	Use case
<code>--skip-package-install</code>	all edge directors share the same software image, and the image is already up to date. So the installer does not need to install packages from that image to the edge director.
<code>--skip-image-update</code> and <code>--skip-reboot</code>	all edge directors are already provisioned with the up-to-date software image. So the installer does not need to carry out an update from the ISO or head node, and then reboot the edge director.
<code>--skip-disksetup-changes</code>	all edge directors already have the correct disk layout. This flag can be set if the disk layout was already configured upfront, in order to avoid full provisioning.

These flags can also be configured in the YAML configuration file of the `cm-kubernetes-setup` wizard.

The flags can be used for scripted installations for quick Kubernetes setups. For a scripted installation of an edge director, preparations can be done beforehand so that all the requirements in the software images that the edge directors use are already installed, the right disk layouts are already configured, and packages are already updated.

All the stages in the flag options can then be skipped for installing onto edge sites. This can make the setup take just a few seconds per Kubernetes deployment.

Kubernetes Cluster API

The Kubernetes Cluster API (CAPI), as explained in the introduction to the online Cluster API Book at <https://cluster-api.sigs.k8s.io/>, "is a Kubernetes sub-project focused on providing declarative APIs and tooling to simplify provisioning, upgrading, and operating multiple Kubernetes clusters". The Cluster API Book is the official Kubernetes project documentation for CAPI.

This chapter describes the installation and usage of the NVIDIA Base Command Manager CAPI extension called BCM Kubernetes CAPI Infrastructure Provider.

8.1 Kubernetes Cluster API Components

An overview of the CAPI components is shown in figure 8.1. Further details about the components are given in the sections of this chapter that follow.

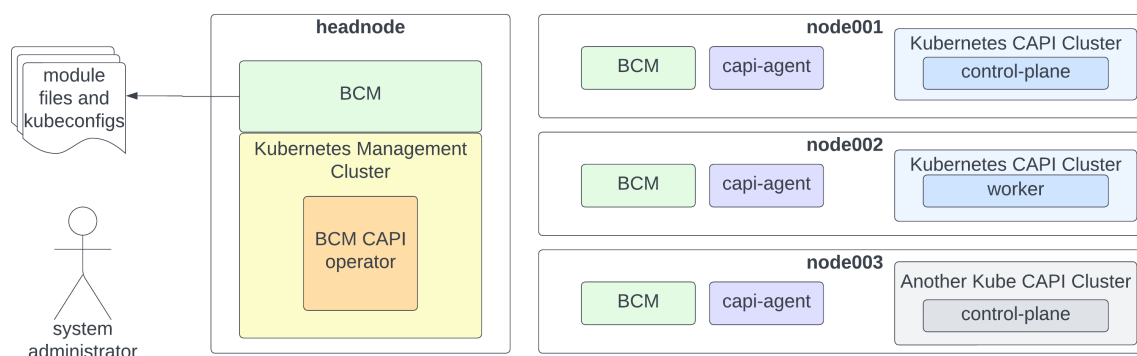


Figure 8.1: CAPI components

Figure 8.1 shows a standard Kubernetes cluster deployed on an active head node. The cluster has been modified to become a Kubernetes management cluster using the BCM CAPI Infrastructure Provider, and has successfully deployed two additional Kubernetes clusters through CAPI. These additional clusters may be running different versions of Kubernetes.

The system administrator receives module files and kubeconfig files for all three Kubernetes clusters.

8.1.1 Kubernetes Management Cluster

In BCM documentation, the term Kubernetes management cluster is used specifically and precisely to refer to a Kubernetes cluster that operates as the management cluster for CAPI.

The ability to modify an *external* Kubernetes cluster to operate as a Kubernetes management cluster is under preparation at the time of writing (June 2023) of this section. This includes scenarios such as using an existing Kubernetes cluster, hosted on a public cloud, to serve as the Kubernetes management

cluster for a Kubernetes cluster deployed through BCM. At present, any Kubernetes cluster deployed through BCM (Chapter 4) can be modified into a CAPI management cluster.

8.1.2 Kubernetes CAPI Cluster

BCM documentation uses the term Kubernetes CAPI cluster to refer specifically to a Kubernetes cluster deployed via the Cluster API. Upstream Kubernetes documentation also sometimes refers to a Kubernetes cluster deployed via CAPI as a workload cluster.

8.1.3 BCM CAPI Infrastructure Provider

The BCM CAPI Infrastructure Provider is a derivative of the Bring Your Own Host (BYOH) CAPI provider. The BYOH CAPI provider can also be referred to as the BYOH CAPI Infrastructure Provider, and is available as a GitHub project at:

<https://github.com/vmware-tanzu/cluster-api-provider-bringyourownhost>

Similar to other CAPI providers, the BCM CAPI Infrastructure Provider is an extension of CAPI itself, and both the CAPI provider and CAPI are required on the Kubernetes management cluster.

The BCM CAPI Operator

The BCM CAPI operator is an operator deployed in the `byoh-system` namespace on the Kubernetes management cluster. The operator serves as the central point of connection for BCM CAPI host agents (section 8.1.3). It also monitors the state of clusters and machines for the Kubernetes CAPI clusters.

```
root@headnode:~# kubectl get pod -n byoh-system
```

NAME	READY	STATUS	RESTARTS	AGE
byoh-controller-manager-6c98cbf44c-7gdbx	2/2	Running	2 (5h55m ago)	23h

BCM CAPI Host Agents

BCM CAPI host agents are scheduled to run on selected hosts. These hosts are defined by the cluster administrator when the `cm-kubernetes-capi-setup` wizard (section 8.2.2) is run. The wizard ensures that a selected host gets the right packages in its software image, and that the host is assigned the right BCM roles.

For CAPI calls to function, the `CapiRole` is assigned. Assignment can be done either before or after creating a Kubernetes CAPI cluster via the Kubernetes management cluster. The role sets up a `capi-agent` service on the host for the associated Kubernetes management cluster.

When the role is:

- assigned: BCM initiates and bootstraps the service
- unassigned: BCM halts the service and makes it unavailable for use by CAPI on the associated Kubernetes management cluster

This differs from cloud-based CAPI providers, which do not require managing a limited number of pre-existing nodes, but instead start up new nodes on demand as needed.

If a Kubernetes CAPI cluster receives a cluster request and there are no hosts available, then machines remain in a `Pending` state until hosts become available. Once hosts become available, the status transitions to `Provisioning`, and then eventually to `Running` (section 8.3.1).

BCM CAPI Vs BYOH

Once the actual Kubernetes CAPI clusters are deployed on designated CAPI hosts, BCM uses its Python API to carry out the following operations:

- The requested Kubernetes version for each node is stored in the BCM database.
- The requested Kubernetes version is made available in the corresponding software image for the node.

- The node is provisioned with the software image.
- The installer is notified that the Kubernetes installation can proceed.

In contrast to the BYOH provider, BCM CAPI has the following extra features:

- Linux distributions can be other than just Ubuntu 20.04.
- Kubernetes versions can be other than only those for which parcels have been created, as the BCM Kubernetes integration does not rely on parcels.

8.2 The Kubernetes CAPI Wizard

The `cm-kubernetes-capi-setup` wizard installs the BCM CAPI Infrastructure Provider, and assigns the CAPI role to nodes.

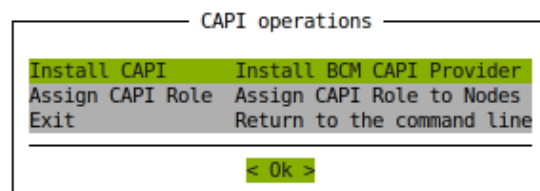


Figure 8.2: CAPI setup wizard

8.2.1 The Install CAPI Option

The `Install CAPI` option of figure 8.2 leads to a screen that prompts the user to select the Kubernetes cluster instance on which to install the BCM CAPI Infrastructure Provider.

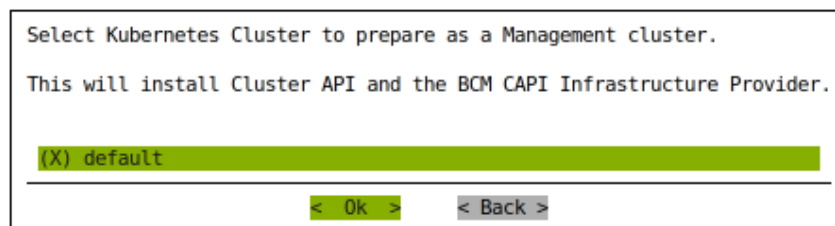


Figure 8.3: Selection of Kubernetes cluster for installation of the BCM CAPI Infrastructure Provider

If needed:

- `cert-manager` and the Cluster API itself are installed
- the CAPI `clusterctl` tool is run in the back end to carry out initialization

The Install CAPI Option Actions

Execution of the `Install CAPI` operation:

- prepares a Kubernetes cluster template for CAPI (section 8.8).
- installs the BCM CAPI Infrastructure Provider on the Kubernetes Cluster with the command: `clusterctl init infrastructure byoh`.
- waits for the BCM CAPI operator to be ready for operation.
- patches the BCM CAPI operator to use the appropriate image. Patching is required until the BCM changes get into the upstream repository.
- makes the `capi` module available for loading

The Install CAPI Option Changes To Kubernetes Management Cluster

After the Install CAPI option has run, the operators installed and running on the Kubernetes management cluster can be seen (some output truncated):

Example

```
[root@basecm11 ~]# kubectl get pod -A | grep -E "byoh-system|capi|cert-manager"
byoh-system                byoh-controller-manager-ff7f68bb4-vprz      2/2 ...
capi-kubeadm-bootstrap-system  capi-kubeadm-bootstrap-controller-manager-7945579f8c-4ntb2  1/1 ...
capi-kubeadm-control-plane-system  capi-kubeadm-control-plane-controller-manager-66cdfb477b-h5779  1/1 ...
capi-system                capi-controller-manager-64cb86f545-8k9zt    1/1 ...
cert-manager               cert-manager-5d4c5bc8bc-j5hbp              1/1 ...
cert-manager               cert-manager-cainjector-79bc559d9d-qfvph    1/1 ...
cert-manager               cert-manager-webhook-5cf45f5b6-9gwlx       1/1 ...
```

The Install CAPI Changes To BCM

A new Kubernetes cluster template is generated in BCM on running Install CAPI. The template naming convention used follows the form:

`capi-<mgmt_cluster_name>-template`

In the present example, where the name of the Kubernetes management cluster is default, the resulting template would be called `capi-default-template`. The name default is set by default when creating a Kubernetes instance (figure 4.2).

```
[root@basecm11 ~]# cmsh
[basecm11]% kubernetes
[basecm11->kubernetes]% list
Name (key)
-----
capi-default-template
default
[basecm11->kubernetes]% show capi-default-template
Parameter                               Value
-----
Name                                    capi-default-template
Revision
Etcd Cluster
Pod Network
Pod Network Node Mask
Internal Network
KubeDNS IP                             0.0.0.0
Kubernetes API server
Kubernetes API server proxy port        6444
App Groups                              <0 in submode>
Label Sets                              <0 in submode>
Notes
Version
Trusted domains                         kubernetes,kubernetes.default,kubernetes.default.svc,localhost
Module file template                    <690B>
Kubeadm init file                       <0B>
Service Network
Kubeadm CERT Key                        < not set >
Kube CA Cert                            < not set >
Kube CA Key                             < not set >
Kubernetes users                        <0 in submode>
```

```

External                               no
External Kubernetes Ingress server
External port                           0
Capi template                           yes
Capi namespace                          default
Kubernetes management cluster            default

```

This becomes the default template used for generating Kubernetes CAPI clusters associated with this specific Kubernetes management cluster. Users can modify this template, and can even create additional templates as needed. The YAML configuration for a CAPI cluster also lets users specify a template, via annotations within the YAML file (section 8.8).

8.2.2 The Assign CAPI Role Option

After the Install CAPI operation has completed its run, the Assign CAPI Role operation can be carried out from the setup screen of figure 8.2. The Assign CAPI Role operation carries out a CAPI role assignment to a group of nodes by creating a configuration overlay (section 4.5). The role can then be assigned either to a category, or to individual nodes.

The software images on the chosen nodes get the packages needed to run the CAPI agents.

When the Assign CAPI Role operation is carried out, the screens in figures 8.4-8.7 may be displayed:

```

Select a Kubernetes Management Cluster for Capi Role assignment to nodes.

(X) default

< Ok >    < Back >

```

Figure 8.4: Select a Kubernetes management cluster

```

The Capi Role will be assigned using a Configuration Overlay.
Optionally customize the overlay name and priority here.
Name capi-default
Priority 500

< Ok >    < Back >

```

Figure 8.5: Customize the configuration overlay

```

Select node categories to run CAPI Agent

[ ] default

< Ok >    < Back >

```

Figure 8.6: Choose nodes via categories

```

Select individual nodes to run CAPI Agent

[X] node001 category:default
[X] node002 category:default
[ ] node003 category:default
[ ] node004 category:default
[ ] node005 category:default
[ ] node006 category:default

< Ok >    < Back >

```

Figure 8.7: Choose nodes directly

Some of the screens may be skipped if they are not needed. For example, the screen for selecting

individual nodes (figure 8.7) is not displayed if all the nodes have already been assigned during selection via categories.

The Assign CAPI Role Actions

During execution of the Assign CAPI Role operation, the wizard:

- prepares the Kubeadm and Helm repositories in the software images for the nodes.
- writes the IP Forwarding configuration to the software images for the nodes.
- installs necessary packages in the software images for the nodes (for example: `cm-capi`).
- provisions the nodes from their software images.
- creates a configuration overlay in BCM.
- assigns the selected categories and nodes to the overlay.
- assigns the `capi` role and `containerd` roles to the overlay.

The Assign CAPI Role Changes To BCM

The configuration overlay created during the execution of the Assign CAPI Role operation is now visible in the `configurationoverlay` mode (some output elided):

Example

```
[root@basecm11 ~]# cmsh
[basecm11]% configurationoverlay
[basecm11->configurationoverlay]% list
```

Name (key)	Priority	All head nodes	Nodes	Categories	Roles
capi-default	500	no	node001,node002		capi
kube-default-etcd	500	no	node003		Etcd::Host
...					

```
[basecm11->configurationoverlay]% show capi-default
```

Parameter	Value
Name	capi-default
Revision	
All head nodes	no
Priority	500
Nodes	node001,node002
Categories	
Roles	capi
Customizations	<0 in submode>

The preceding overlay shows that the `capi` role has been assigned to nodes `node001` and `node002`.

The `capi` role sets the `capi-agent` service for a node. If it is set, then BCM bootstraps the agent (section 8.4) and initiates the `capi-agent systemd` service that runs on the node.

The `containerd` role takes care of the `containerd` service, so that `containerd` is also started on the nodes.

The `containerd` service is needed after the agent initiates the provisioning of its node to integrate it into a Kubernetes CAPI cluster. This is because a `kubelet` service is eventually started, which relies on `containerd`.

CAPI uses `kubeadm` for node provisioning and management, which means that `containerd` must be running in advance. Therefore, as part of its pre-launch checks, `kubeadm` verifies that the `containerd` service is operational.

The Assign CAPI Role Changes To the Kubernetes management cluster

The BCM CAPI host agents should register with the Kubernetes management cluster, and can then be seen on running `kubectl` with the `get byohost` option:

Example

```
root@basecm11:~# kubectl get byohost -A
NAMESPACE   NAME      OSNAME  OSIMAGE          ARCH
default     node001   linux   Ubuntu 22.04.4 LTS amd64
default     node002   linux   Ubuntu 22.04.4 LTS amd64
```

Registration does not necessarily make these hosts a part of any Kubernetes CAPI cluster. However, if there are clusters that have been created beforehand, and there are machines that are awaiting additional resources, then it might be that some or all of these hosts are immediately provisioned. A simple method to check this is by querying the machine resource.

```
[root@basecm11 ~]# kubectl get machine -A
No resources found
```

If however there were machines that were pending, the output might look like:

```
[root@basecm11 ~]# kubectl get machines
NAME                                CLUSTER      NODENAME  PROVIDERID  PHASE      AGE  VERSION
byoh-cluster-control-plane-h2d64    byoh-cluster                               Provisioning 4s    v1.26.1
byoh-cluster-md-0-56985bf9d6xkhj68-dxvkc byoh-cluster                               Pending     6s    v1.26.1
```

When the machines are provisioned, the Kubernetes CAPI cluster output state might look like:

```
[root@basecm11 ~]# kubectl get ma      #ma|machine|machines are synonymous. Also byom|byomachine|byomachines
NAME                                CLUSTER      NODENAME  PROVIDERID  PHASE      AGE  ...
byoh-cluster-control-plane-h2d64    byoh-cluster  node001    byoh://node001/e7fq1b  Running    3h3m...
byoh-cluster-md-0-56985bf9d6xkhj68-dxvkc byoh-cluster  node002    byoh://node002/ngkju0  Running    3h3m...
```

This is discussed further in section 8.3.

8.3 Deploying A Kubernetes Cluster Through CAPI

The BCM CAPI Infrastructure Provider configures the Kubernetes management cluster. A Kubernetes cluster can then be created through CAPI, and deployed on the available CAPI nodes.

If resources are not available, then the cluster cannot assign nodes to the newly-created cluster definition for provisioning. This results in a persistent Pending state for various resources.

The `cap` module makes the command line tool `clusterctl` available, which can be used to generate a cluster manifest. In the following example, the tool is used to define a cluster with Kubernetes version 1.32.10, comprising one control plane node and one worker node.

Example

```
[root@headnode ~]# module load capi/1.10.2
[root@headnode ~]# CONTROL_PLANE_ENDPOINT_IP=10.141.168.1 clusterctl generate cluster byoh-cluster \
--from /cm/local/apps/capi/var/cluster-template.yaml --kubernetes-version v1.34.2 \
--control-plane-machine-count 1 --worker-machine-count 1 > cluster.yaml
```

The environment variable `CONTROL_PLANE_ENDPOINT_IP` must be set by the cluster administrator to a valid unused IP address. The IP address must be within the internal network of the nodes that have been assigned the CAPI role. Here the IP address of 10.141.168.1 is set.

The YAML file that is generated is typically several hundred lines long:

Example

```
[root@headnode ~]# wc -l cluster.yaml
285 cluster.yaml
```

The start of it looks like:

Example

```
[root@headnode ~]# head cluster.yaml
apiVersion: bootstrap.cluster.x-k8s.io/v1beta1
kind: KubeadmConfigTemplate
metadata:
  name: byoh-cluster-md-0
  namespace: default
spec:
  template:
    spec: {}
---
```

The cluster can now be created with `kubect1`:

Example

```
[root@headnode ~]# kubect1 create -f cluster.yaml
kubeadmconfigtemplate.bootstrap.cluster.x-k8s.io/byoh-cluster-md-0 created
cluster.cluster.x-k8s.io/byoh-cluster created
machinedeployment.cluster.x-k8s.io/byoh-cluster-md-0 created
kubeadmcontrolplane.controlplane.cluster.x-k8s.io/byoh-cluster-control-plane created
byoccluster.infrastructure.cluster.x-k8s.io/byoh-cluster created
byomachinetemplate.infrastructure.cluster.x-k8s.io/byoh-cluster-control-plane created
byomachinetemplate.infrastructure.cluster.x-k8s.io/byoh-cluster-md-0 created
k8sinstallerconfigtemplate.infrastructure.cluster.x-k8s.io/byoh-cluster-control-plane created
k8sinstallerconfigtemplate.infrastructure.cluster.x-k8s.io/byoh-cluster-md-0 created
```

The very first control plane node is assigned this IP address initially, and any other control planes nodes do not try to take it at that time. Only one of the control plane nodes use this IP address at a time.

Load balancing between all three nodes is however possible—it is just not currently configured out-of-the-box at the time of writing (July 2023).

When all prerequisites are met, the BCM CAPI Infrastructure Provider initiates node provisioning, and creates the cluster. BCM generates a module file and kubeconfig for the cluster automatically.

8.3.1 Machine Provisioning

Applying the YAML for the cluster creates the machine resources. These are initially in a Pending state:

Example

```
root@basecm11:~# kubect1 get machines
```

NAME	CLUSTER	NODENAME	PROVIDERID	PHASE	AGE	VERSION
byoh-cluster-control-plane-gdm7n	byoh-cluster			Pending	3s	v1.34.2
byoh-cluster-md-0-khrk6-mszxr	byoh-cluster			Pending	5s	v1.34.2

The operator in the `byoh-system` namespace then allocates resources. It first selects the machine for the control plane:

```
[root@basecm11 ~]# kubectl logs -n byoh-system -l cluster.x-k8s.io/provider=infrastructure-byoh
I0601 06:16:18.202787 .. "msg"="Attempting host reservation" "cluster"="byoh-cluster" ...
I0601 06:16:18.318478 .. "msg"="Successfully attached Byohost" "byohost"="node001" "cluster"="byoh-cluster" ...
```

The capi-agent service on the node finds an installation script and invokes it, as indicated by the log entry:

Example

```
[root@node001 ~]# journalctl -u capi-agent.service -g executing
Jun 01 06:16:20 .. controller/byohost "msg"="executing install script" "name"="node001" ...
```

At this point the ByoHost resource should be linked to a ByoMachine resource, which is linked to a Machine resource. Until the installation script has completed, it is not easy to go the other way around from a Machine resource to find the related ByoHost resource. Once the installation script has completed, the Machine resource updates the ProviderID column with a value, but it can take some time to show up. It shows up later in this session as the value byoh://node001/f3j7mt in this example session.

The output for the installation script is only printed on completion, and the installation script is automatically removed. Figure 8.9 has more details on the installation process.

The machine transitions to the Provisioning state, and also the ByoHost resource is now tied to the given Machine:

Example

```
[root@basecm11 ~]# kubectl get machines
```

NAME	CLUSTER	NODENAME	PROVIDERID	PHASE	AGE	VERSION
byoh-cluster-control-plane-gdm7n	byoh-cluster			Provisioning	8s	v1.34.2
byoh-cluster-md-0-khrk6-mszxr	byoh-cluster			Pending	10s	v1.34.2

The journal for BCM displays output similar to the following (some output ellipsized):

Example

```
[root@basecm11 ~]# journalctl -u cmd.service | grep -i register
Nov 25 cmd[... [CapiNodeVersionManager::register_node], create new kube cluster: byoh-cluster, ...
Nov 25 cmd[... [CapiNodeVersionManager::register_node], add node: 1f6520c9-1475-4493-825c-ac83f...
Nov 25 cmd[... [CapiRegisterNode::async_work::async]: Registration of nodes completed: node002
Nov 25 cmd[... [CapiNodeVersionManager::register_node], add node: 42c06248-fae1-423a-82d0-9e2fd...
Nov 25 cmd[... [CapiRegisterNode::async_work::async]: Registration of nodes completed: node001
```

The registration of a node by the wizard is done automatically in the background, by using BCM's Python API (PythonCM, Chapter 1 of the *Developer Manual*). The node is registered with the active head node, triggering a cascade of events using:

```
cm-kubernetes-capi-setup --register-node node001
```

Script logs can be found in the log file `/var/log/cm-kubernetes-capi-setup.log`. More on what the script does can be found in section 8.5.

If multiple nodes are being provisioned at the same time, then BCM invokes the script with more nodes as arguments so that the work is parallelized. On completion, the machine transitions to the Running state:

Example

```
[root@basecm11 ~]# kubectl get machine
```

NAME	CLUSTER	NODENAME	PROVIDERID	PHASE	AGE	VERSION
byoh-clus..	byoh-cluster	node001	byoh://node001/e7fq1b	Running	3m4s	v1.26.0
byoh-clus..	byoh-cluster			Provisioning	3m6s	v1.26.0

Section 8.5.1 covers the process from a different perspective, which may clarify matters further.

8.3.2 Accessing The Cluster

Assuming a Kubernetes CAPI cluster named byoh-cluster has been deployed, with three control planes, one worker, with the following machines all running:

Example

```
[root@basecm11 ~]# kubectl get machines
```

NAME	CLUSTER	NODENAME	PROVIDERID	PHASE	AGE	VERSION
byoh-clus...	byoh-cluster	node006	byoh://node006/ytpvbd	Running	26m	v1.26.1
byoh-clus...	byoh-cluster	node004	byoh://node004/7rczjg	Running	22m	v1.26.1
byoh-clus...	byoh-cluster	node002	byoh://node002/i0tj4j	Running	44m	v1.26.1
byoh-clus...	byoh-cluster	node003	byoh://node003/f18zem	Running	44m	v1.26.1

The KubeCluster entity can then be seen in cmsh:

Example

```
[root@basecm11 ~]# cmsh
[basecm11]% kubernetes
[basecm11->kubernetes]% show byoh-cluster
```

Parameter	Value
Name	byoh-cluster
Revision	
Etc Cluster	
Pod Network	
Pod Network Node Mask	
Internal Network	
KubeDNS IP	0.0.0.0
Kubernetes API server	
Kubernetes API server proxy port	6444
App Groups	<0 in submode>
Label Sets	<0 in submode>
Notes	
Version	1.26.1
Trusted domains	kubernetes,kubernetes.default,kubernetes.default.svc,master,localhost
Module file template	<690B>
Kubeadm init file	<0B>
Service Network	
Kubeadm CERT Key	< not set >
Kube CA Cert	< not set >
Kube CA Key	< not set >
Kubernetes users	<0 in submode>
External	no
External Kubernetes Ingress server	
External port	0
Capi template	no
Capi namespace	default
Kubernetes management cluster	default
Ingress Proxy Enable	no
Ingress Proxy Listen Port	443
Ingress Proxy Backend Port	0

There should also be a byoh-cluster module file available on the head node:

Example

```
[root@basecm11 ~]# module load kubernetes/byoh-cluster/1.26.1
[root@basecm11 ~]# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
node002	NotReady	control-plane	29m	v1.26.1
node003	NotReady	<none>	25m	v1.26.1
node004	NotReady	control-plane	23m	v1.26.1
node006	NotReady	control-plane	25m	v1.26.1

By default, Kubernetes CAPI clusters do not come with a networking implementation configured. This can be created by the cluster administrator, to see if this improves the state of the cluster (section 4.2.2):

Example

```
root@rb-capi2:~# kubectl create -f https://raw.githubusercontent.com/projectcalico/calico/v3.24.5 ...
... /manifests/calico-typha.yaml
poddisruptionbudget.policy/calico-kube-controllers created
poddisruptionbudget.policy/calico-typha created
serviceaccount/calico-kube-controllers created
serviceaccount/calico-node created
configmap/calico-config created
customresourcedefinition.apiextensions.k8s.io/bgpconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/blockaffinities.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/caliconodestatuses.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamblocks.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamconfigs.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamhandles.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippreservations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/kubecontrollersconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networksets.crd.projectcalico.org created
clusterrole.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrole.rbac.authorization.k8s.io/calico-node created
clusterrolebinding.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrolebinding.rbac.authorization.k8s.io/calico-node created
service/calico-typha created
daemonset.apps/calico-node created
deployment.apps/calico-kube-controllers created
deployment.apps/calico-typha created
```

The preceding session ends up with the nodes then ends up configured with Calico networking:

Example

```
root@rb-capi2:~# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
node002	Ready	control-plane	31m	v1.26.1
node003	Ready	<none>	27m	v1.26.1
node004	Ready	control-plane	25m	v1.26.1
node006	Ready	control-plane	28m	v1.26.1

8.3.3 Scaling Control Planes Or Workers

Control planes can be scaled at the level of the `KubeadmControlPlane` resource (some output ellipsized):

Example

```
[root@basecm11 ~]# module load kubernetes/default
[root@basecm11 ~]# kubectl get kubeadmcontrolplane
NAME                                CLUSTER      INITIALIZED  ... REPLICAS  ... UPDATED  UNAVAILABLE  ...
byoh-cluster-control-plane        byoh-cluster  true         ... 1         ... 1         1           ...
[root@basecm11 ~]# kubectl patch kubeadmcontrolplane byoh-cluster-control-plane \
--patch='{ "spec": { "replicas": 3 } }' \
--type=merge
kubeadmcontrolplane.controlplane.cluster.x-k8s.io/byoh-cluster-control-plane patched
```

Workers can be scaled at the level of the `MachineDeployment` resource:

Example

```
[root@basecm11 ~]# kubectl get machinedeployment
NAME                CLUSTER      REPLICAS  ... UPDATED  UNAVAILABLE  PHASE      AGE    VERSION
byoh-cluster-md-0   byoh-cluster  1         ... 1         1           ScalingUp   14h    v1.26.1
[root@basecm11 ~]# kubectl scale --replicas=2 machinedeployment/byoh-cluster-md-0
machinedeployment.cluster.x-k8s.io/byoh-cluster-md-0 scaled
```

8.3.4 Upgrading Control Planes Or Workers

For this section, some background understanding of how Kubernetes upgrades work is recommended. A good introduction can be found at:

<https://cluster-api.sigs.k8s.io/tasks/upgrading-clusters.html>

Further hints and suggestions on upgrading can be found at:

- <https://github.com/kubernetes/sig-release/blob/master/release-engineering/versioning.md#kubernetes-release-versioning>: discusses release versioning, and is a recommended first read
- <https://kubernetes.io/releases/version-skew-policy/>: discusses version skew policy, but is also a document that provides more detailed information

One suggestion from these resources that should be followed, is first to upgrade to the latest patch version of the current minor version, and then to upgrade to the next minor version.

Typically all the control planes are upgraded first, and then the workers.

Rolling Upgrades

The default upgrade method is with rolling upgrades. More information on rolling upgrades can be found at:

<https://cluster-api.sigs.k8s.io/tasks/upgrading-clusters.html#upgrading-machines-managed-by-a-machinedeployment>

The rolling upgrades method requires that at least one spare `ByoHost` is available, since machines are replaced one by one, and both the old and new machine need to run at the same time during a part of the procedure. If assigning an extra CAPI role is a problem, then the upgrade strategy based on `OnDelete` can be followed instead, which is also described at that URL.

Another option, depending on the cluster, could be to temporarily scale down the workers by one during the rolling upgrade, via the `MachineDeployment` resource.

Deprecated APIs

- The deprecation guide at:

<https://kubernetes.io/docs/reference/using-api/deprecation-guide/>

should be read before carrying out upgrades. The cluster administrator should check the changelogs and upstream documentation for obsolete APIs in the target version.

- The pluto utility (<https://github.com/FairwindsOps/pluto>) can check for deprecated API usage.
- The kubent utility (<https://github.com/doitintl/kube-no-trouble>) can also be used, but seems a few Kubernetes versions behind at the time of writing (June 2023).

Upgrading The Control Plane

The following example session upgrades the control plane from version v1.26.1 to v1.26.2:

Example

```
[root@basecm11 ~]# kubectl get kubeadmcontrolplane
NAME                                CLUSTER      INITIALIZED    .. REPLICAS    .. UPDATED    UNAVAILABLE    AGE    VERSION
byoh-cluster-control-plane         byoh-cluster  true           .. 1           .. 1          1             16h    v1.26.1
[root@basecm11 ~]# kubectl patch kubeadmcontrolplane byoh-cluster-control-plane \
--type=merge \
-p '{"spec": {"version": "v1.26.2"}}'
kubeadmcontrolplane.controlplane.cluster.x-k8s.io/byoh-cluster-control-plane patched
```

The resource immediately shows the new version. However, the upgrade is not performed immediately:

```
[root@basecm11 ~]# kubectl get kubeadmcontrolplane
NAME                                CLUSTER      INITIALIZED    .. REPLICAS    .. UPDATED    UNAVAILABLE    AGE    VERSION
byoh-cluster-control-plane         byoh-cluster  true           .. 1           .. 0          1             16h    v1.26.2
```

First a new control plane is provisioned with the new version. Only when it is fully up, is the old control plane deleted:

```
[root@basecm11 ~]# kubectl get machine
NAME          CLUSTER      NODENAME    PROVIDERID          PHASE          AGE    VERSION
byoh-clus.. byoh-cluster  node001     byoh://node005/gije0o Running        125m    v1.26.1
byoh-clus.. byoh-cluster  node002     byoh://node004/98s8gu Running        98m     v1.26.1
```

The deletion is not carried out immediately. There is a grace period in which both control planes run. The old control plane is deleted shortly afterwards.

```
[root@basecm11 ~]# kubectl get machine
NAME          CLUSTER      NODENAME    PROVIDERID          PHASE          AGE    VERSION
byoh-clus.. byoh-cluster  node001     byoh://node005/gije0o Running        127m    v1.26.1
byoh-clus.. byoh-cluster  node003     byoh://node003/mjvvvg Running        2m30s    v1.26.2
byoh-clus.. byoh-cluster  node004     byoh://node004/98s8gu Running        101m    v1.26.1
```

If there is than one control plane, then a rolling upgrade takes place, one control plane at a time.

Upgrading The Workers

The following session shows the workers being upgraded from version v1.26.1 to v1.26.2.

Example

```
[root@basecm11 ~]# kubectl get machinedeployment
```

NAME	CLUSTER	REPLICAS	.. UPDATED	UNAVAILABLE	PHASE	AGE	VERSION
byoh-cluster-md-0	byoh-cluster	3	.. 3	3	ScalingUp	17h	v1.26.1

In this case there are three workers. The version is patched with:

Example

```
[root@basecm11 ~]# kubectl patch machinedeployment byoh-cluster-md-0 \
    --type=merge \
    -p '{"spec": {"template": {"spec": {"version": "v1.26.2"}}}}'
machinedeployment.cluster.x-k8s.io/byoh-cluster-md-0 patched
```

The resource immediately shows the new desired version. However, the upgrade is not performed immediately:

Example

```
[root@basecm11 ~]# kubectl get machinedeployment
```

NAME	CLUSTER	REPLICAS	.. UPDATED	UNAVAILABLE	PHASE	AGE	VERSION
byoh-cluster-md-0	byoh-cluster	4	.. 1	4	ScalingUp	17h	v1.26.2

First a new worker is provisioned with the new version. Only when it is fully up, is the old worker deleted:

```
[root@basecm11 ~]# kubectl get machine
```

NAME	CLUSTER	NODENAME	PROVIDERID	PHASE	AGE	VERSION
byoh-clus..	byoh-cluster	node003	byoh://node003/mjvvvg	Running	5m45s	v1.26.2
byoh-clus..	byoh-cluster	node004	byoh://node004/98s8gu	Running	104m	v1.26.1
byoh-clus..	byoh-cluster	node002	byoh://node002/f3j7mt	Running	107m	v1.26.1
byoh-clus..	byoh-cluster	node006	byoh://node006/paityf	Running	139m	v1.26.1
byoh-clus..	byoh-cluster			Provisioning	7s	v1.26.2

The process repeats itself until all the workers are upgraded.

8.4 BCM Host Agent Registration

The registration process in figure 8.8 is sourced from the documentation at the upstream BYOH project:

<https://github.com/vmware-tanzu/cluster-api-provider-bringyourownhost/blob/main/docs/>

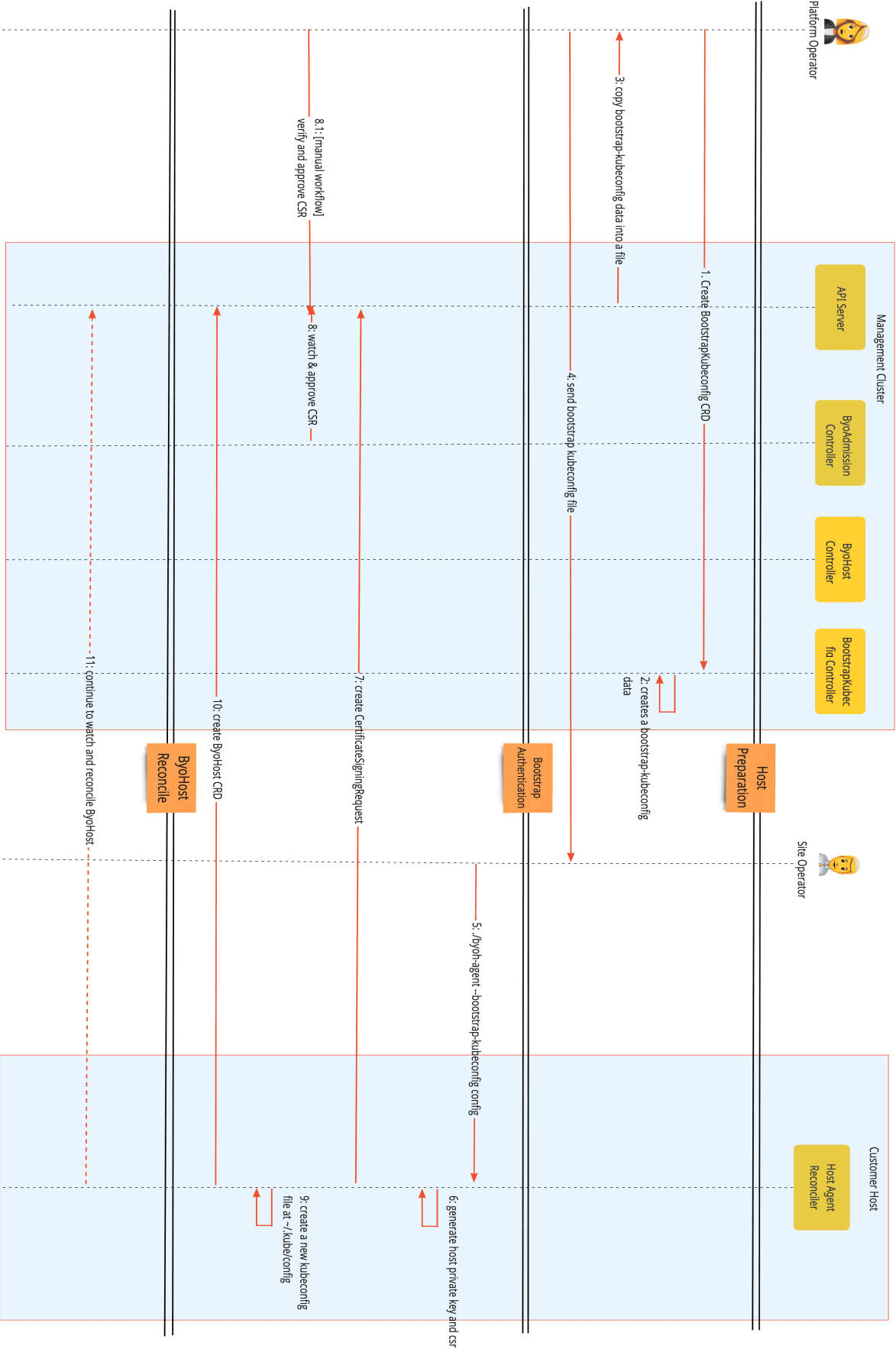


Figure 8.8: BYOH bootstrap flow

The roles of the platform operator and site operator in the illustration are automated by BCM.

The process initiated by BCM CAPI Infrastructure Provider starts with BCM.

When initiating, the capi-agent service verifies the existence of a configuration before launching. If a configuration does not exist, then the BCM API is used to request one.

BCM then automates all the steps necessary to generate this configuration, and prepares it for the capi-agent so it can start its process.

To clarify figure 8.8 further, before step 9 (create a new kubeconfig file at `~/.kube/config`), only the *bootstrap* configuration exists on the node. However, after this step, *two* configuration files exist: the *bootstrap* and the *definitive* configuration files. The presence of these files enables BCM to determine the current phase of the bootstrap process that the capi-agent is in.

Manual generation of bootstrap configurations is also possible. Details on this are given in the documentation at https://github.com/vmware-tanzu/cluster-api-provider-bringyourownhost/blob/main/docs/getting_started.md#register-byoh-host-to-management-cluster

The host agent registration process described in the preceding is part of what is carried out for nodes when running the Assign Capi Role option in section 8.2.2.

```
[root@basecm11 ~]# kubectl get byohost -A
```

NAMESPACE	NAME	OSNAME	OSIMAGE	ARCH
default	node001	linux	Rocky Linux 8.7 (Green Obsidian)	amd64
default	node002	linux	Rocky Linux 8.7 (Green Obsidian)	amd64
default	node003	linux	Rocky Linux 8.7 (Green Obsidian)	amd64
default	node004	linux	Rocky Linux 8.7 (Green Obsidian)	amd64
default	node005	linux	Rocky Linux 8.7 (Green Obsidian)	amd64
default	node006	linux	Rocky Linux 8.7 (Green Obsidian)	amd64

8.5 Install Process BCM CAPI

The preceding section (section 8.4) focused on BCM CAPI host agent registration. The current section (section 8.5) discusses the deployment of Kubernetes clusters on these CAPI Hosts.

The deployment uses installation scripts transmitted to the designated nodes through “installation secrets”.

Node registration with BCM is also carried out as part of the deployment of Kubernetes on the CAPI hosts, and is distinct from the host agent registration of section 8.4. The deployment process flow is illustrated in figure 8.9:

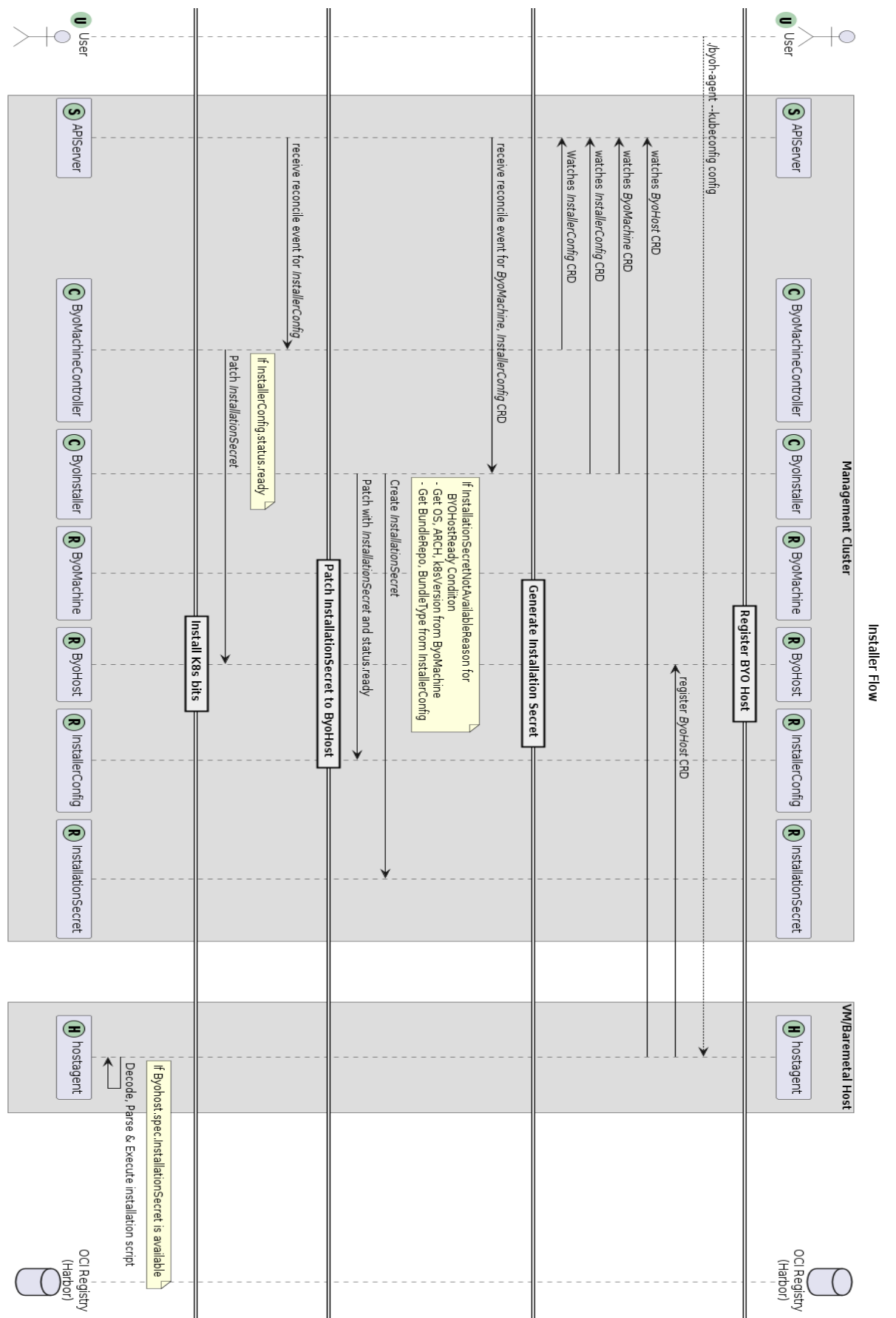


Figure 8.9: BYOH installer flow

8.5.1 Registration Process Of The Node With BCM

With BCM CAPI Infrastructure Provider, the installation script is a `pythoncm` script. It establishes communication with BCM on the active Head Node. This is referred to as “registering” the node with BCM.

The registration process involves essential bookkeeping tasks, such as tracking which nodes belong to specific Kubernetes CAPI clusters, and preparing software images with the appropriate Kubernetes versions.

The example that follows illustrates various components involved in creating a Kubernetes cluster using CAPI. In the example, a single head node operates as the Kubernetes management cluster, and multiple nodes are assigned the CAPI role. A cluster definition is generated for a control plane node and a worker node using the procedure described in section 8.3. An overview of the components involved when the cluster definition is applied is seen in figure 8.10:

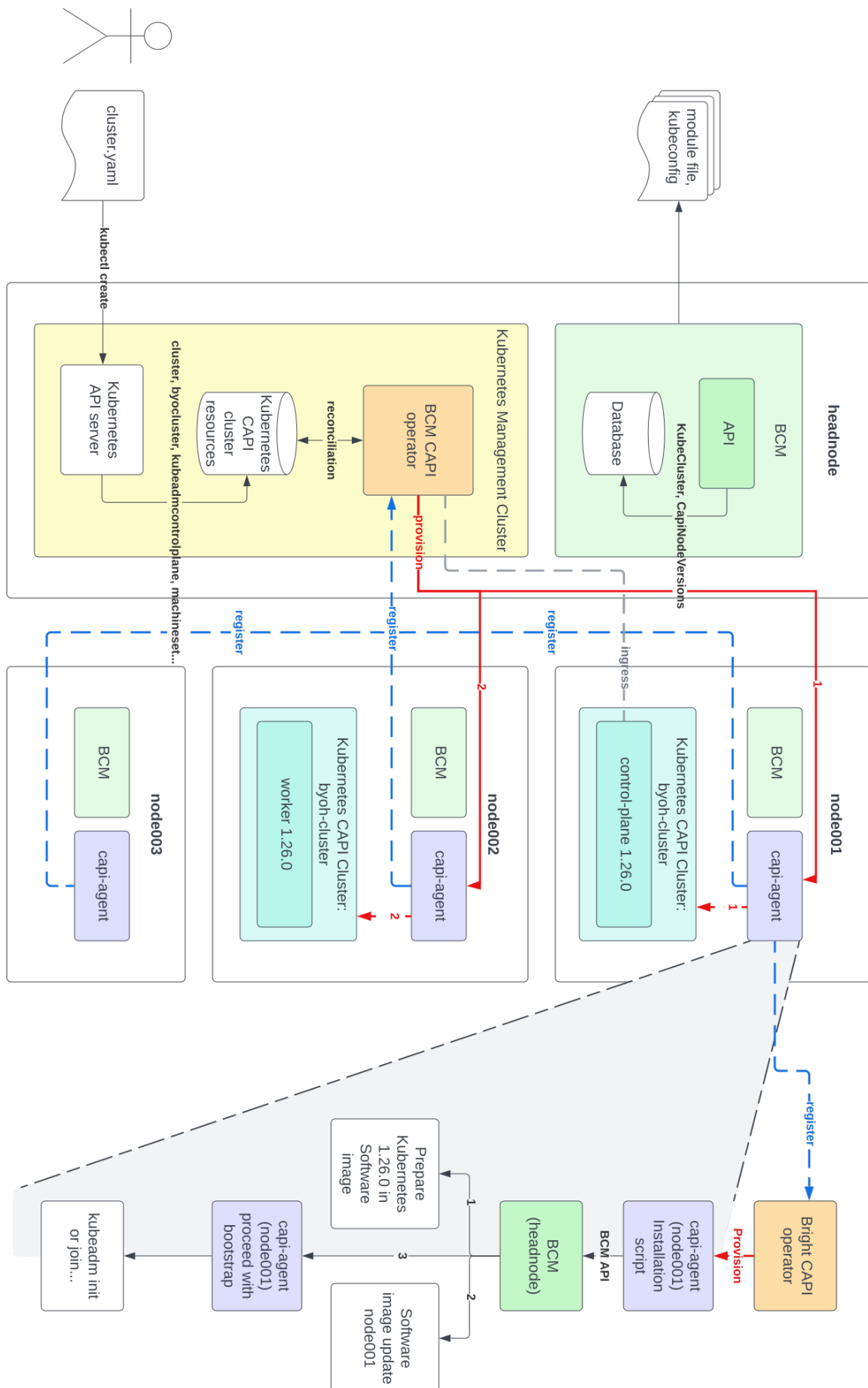


Figure 8.10: CAPI summary

In figure 8.10, the Kubernetes CAPI cluster has the control plane deployed on node001, followed by the deployment of the worker node software image on node002. The diagram of the is expanded from the node001 CAPI agent block illustrates the steps executed by the node001 capi-agent.

Upon registering with the BCM API, the more important actions that BCM carries out are:

- **Step 1:** Preparation of Kubernetes version 1.26.0 within the software image for the registering node.
- **Step 2:** Provisioning of the node through an image update, for example using the `imageupdate` command of `cmsh` (section 5.6.2 of the *Administrator Manual*), to ensure that it stays synchronized with its software image.
- **Step 3:** Handing control back to the capi-agent, which proceeds with the subsequent step in the CAPI cluster creation process.

The steps also include processes beside the ones illustrated in figure 8.10. The more complete sequence is:

- **Step 1:** The installation script is invoked with contextual information, such as the desired Kubernetes version (for example: 1.26.0) and the cluster name.
- **Step 2:** The installation script, implemented as a PythonCM script, establishes communication with BCM. It registers itself with the active head node and remains in a waiting state until notified to terminate.
- **Step 3:** BCM creates a mapping that specifies which node should be provisioned with the corresponding Kubernetes version.
- **Step 4:** BCM ensures that the software image for the node contains the requested Kubernetes version.
- **Step 5:** BCM ensures that the node is provisioned with its designated software image.
- **Step 6:** BCM signals to the node that the installation script has completed.

From this point onward, the default logic for the BCM CAPI operator takes over, and `kubeadm` initializes the node accordingly.

8.5.2 Creating A Kubernetes Cluster Via CAPI

The steps in section 8.5.1, are about the node registration process and node provisioning during Kubernetes cluster creation with CAPI. The following steps zoom out further, and describe the complete process of creating a Kubernetes cluster through CAPI.

- **Step 1:** The system administrator defines a cluster to be deployed by the BCM CAPI operator, and feeds it to the Kubernetes API server using `kubectl`.
- **Step 2:** The definition results in a number of Kubernetes resources being created, such as a `Cluster`, `ByoCluster`, `MachineDeployment`.
- **Step 3:** The BCM CAPI operator responds to these newly-created resources.
- **Step 4:** CAPI starts assigning CAPI agents to reconcile specific `Machines`, beginning with the first control plane. This is indicated by the red provisioning line, numbered with a 1.
- **Step 5:** BCM prepares a module file and `kubeconfig` for the new Kubernetes CAPI cluster, and writes these to disk for the system administrator.
- **Step 6:** BCM CAPI operator updates its records using the newly-created control plane and proceeds with provisioning the additional control planes or workers, such as the additional worker. This is indicated by the red line numbered with a 2.

8.6 Configuring CAPI Versions In Software Images

BCM takes care of configuring CAPI versions in software images automatically. This is part of its BCM CAPI node registration process, as mentioned earlier in section 8.5.

An alternative is to manually pre-install a specific version of Kubernetes in a software image. This can be done from within the softwareimage mode of cmsh:

```
[root@basecm11 ~]# cmsh
[basecm11]% softwareimage
[basecm11->softwareimage]% use default-image
[basecm11->softwareimage[default-image]]% help capi
Name:
capi - Manage Kubernetes CAPI versions on the image

Usage:
capi [OPTIONS] list
capi [OPTIONS] add <version> [<version> ...]
capi [OPTIONS] remove <version> [<version> ...]
capi [OPTIONS] clear

Options:
-v, --verbose
Be more verbose

-d, --delimiter <string>
Use <string> as delimiter between columns. Use {} for JSON, and {<digit>} for JSON with a specific indentation.

--image, -i <list of images>
Perform action on comma separated list of images

--repo-refresh, -r
Refresh the repository cache before adding new versions

--debug
Run script with debug on

Examples:
capi list                List CAPI versions on all / current image
capi clear               Remove all CAPI versions on all / current image
capi add 1.26.0 1.27.*   Add specified versions on all / current image
```

Example

```
[basecm11->softwareimage[default-image]]% capi list
```

Node	image	versions	Result	Error
basecm11	/cm/images/default-image	1.24.9, 1.23.0, 1.26.1, 1.26.2	good	

```
[basecm11->softwareimage[default-image]]% capi add 1.27.*
```

Node	image	versions	Result	Error
basecm11	/cm/images/default-image	1.24.9, 1.23.0, 1.26.1, 1.26.2, 1.27.0, 1.27.1, 1.27.2	good	

8.7 Removing Kubernetes CAPI clusters

The following steps remove the CAPI clusters:

- **Step 1:** The removal of the CAPI clusters is started with the `kubect1 delete` command:

```
[root@basecm11 ~]# kubectl delete -f cluster.yaml
...
```

- **Step 2:** If all CAPI hosts are not part of any significant clusters, then the configuration overlay configuration overlays are removed:

```
[root@basecm11 ~]# cmsh
[basecm11]% configurationoverlay
[basecm11->configurationoverlay]% remove capi-mgmt
[basecm11->configurationoverlay*]% commit
Successfully removed 1 ConfigurationOverlays
Successfully committed 0 ConfigurationOverlays
```

The removal of the configuration overlays causes containerd and the CAPI agents to stop on the hosts.

- **Step 3:** For additional cleanliness, the bootstrap configurations for each of the hosts should be removed:

```
[root@basecm11 ~]# kubectl delete bootstrapkubeconfig --all
bootstrapkubeconfig.infrastructure.cluster.x-k8s.io "bootstrap-kubeconfig-node001" deleted
bootstrapkubeconfig.infrastructure.cluster.x-k8s.io "bootstrap-kubeconfig-node002" deleted
bootstrapkubeconfig.infrastructure.cluster.x-k8s.io "bootstrap-kubeconfig-node003" deleted
bootstrapkubeconfig.infrastructure.cluster.x-k8s.io "bootstrap-kubeconfig-node004" deleted
bootstrapkubeconfig.infrastructure.cluster.x-k8s.io "bootstrap-kubeconfig-node005" deleted
bootstrapkubeconfig.infrastructure.cluster.x-k8s.io "bootstrap-kubeconfig-node006" deleted
```

- **Step 4:** The infrastructure provider is then removed:

```
[root@basecm11 ~]# module load capi/1.3.0
[root@basecm11 ~]# clusterctl delete --infrastructure byoh
Deleting Provider="infrastructure-byoh" Version="" Namespace="byoh-system"
```

- **Step 5:** The removal of all the ByoHost resources can be checked:

```
[root@basecm11 ~]# kubectl get byohost
No resources found
```

For clean-up the command

```
kubectl delete byohost ...
```

can be used.

- **Step 6:** Finally, the Kubernetes CAPI clusters is eliminated from BCM itself:

```
[root@basecm11 ~]# cmsh
[basecm11]% kubernetes
[basecm11->kubernetes]% remove byoh-cluster
[basecm11->kubernetes*]% commit
[basecm11->kubernetes]%
```

8.8 Kubernetes CAPI Templates

In BCM, each Kubernetes cluster is represented as a KubeCluster entity stored in the BCM database. These entities can be viewed in the kubernetes submode of cmsh:

Example

```
[root@basecm11 ~]# cmsh
[basecm11]% kubernetes
[basecm11->kubernetes]% list
Name (key)
-----
default
```

- For every Kubernetes management cluster, a default KubeCluster CAPI template is generated once the CAPI role node is assigned using the wizard, as discussed in section 8.2.2. This template serves as a base for all subsequent Kubernetes clusters created via CAPI.
- For example, if an administrator creates a cluster `my-capi-cluster` via CAPI, for the Kubernetes management cluster `mgmt`, then BCM clones the `capi-mgmt-template` KubeCluster entity to create a new KubeCluster `my-capi-cluster`.
- At the time of CAPI cluster creation, the KubeCluster template can also be customized using an annotation in the Cluster resource definition:

Example

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  labels:
    cni: byoh-cluster-crs-0
    crs: "true"
    infrav1.nvidia.x-k8s.io/capi-kube-template: "my-capi-cluster-template"
  name: byoh-cluster
  namespace: default
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.0.0/16
      serviceDomain: cluster.local
    services:
      cidrBlocks:
        - 10.128.0.0/12
  controlPlaneRef:
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    kind: KubeadmControlPlane
    name: byoh-cluster-control-plane
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1beta1
    kind: ByoCluster
    name: byoh-cluster
```

The provided YAML illustrates a cluster `byoh-cluster` definition, with the specified `capi-template` set to `my-capi-cluster-template`. If present, then this is used instead of the default.

- The fallback sequence for the CAPI template is: The specified label (for example: `my-capi-cluster-template`) is used if it exists; otherwise the default `capi-mgmt-template` is used. If neither is available, then `capi-template` is used.

A

Create Self-Signed Server Certificate Pair For Testing Purposes

The cluster administrator can issue self-signed certificates. For testing purposes, a self-signed certificate pair can be installed on any device where authentication is needed via the web interface, for example on a client connecting to the bcm-gateway Gateway (section 4.17.7). The configuration is carried out as follows:

The cm-kubernetes-setup wizard is started up on the active head node by the cluster administrator.

The Gateway configuration menu is opened, and the Configure TLS Certificate for Gateway item is selected from it (figure A.1).

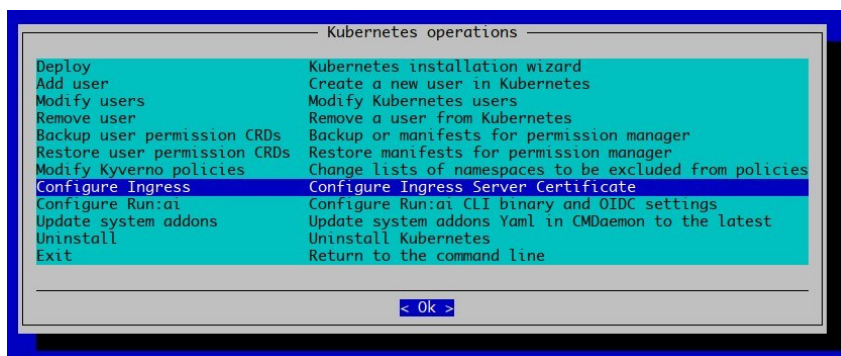


Figure A.1: The Configure TLS Certificate for Gateway item under the Gateway configuration menu

The Kubernetes cluster is chosen in the next screen. The Kubernetes cluster is the one whose bcm-gateway Gateway TLS certificate is to be configured (figure A.2)

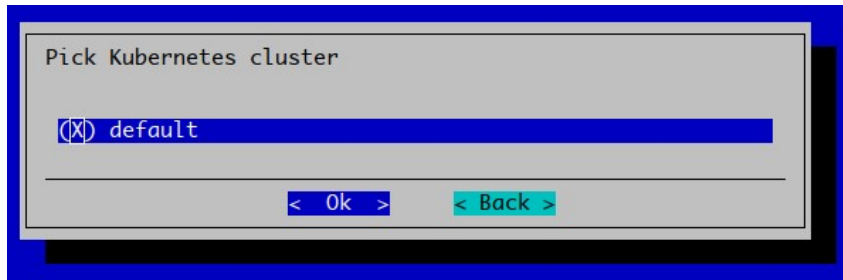


Figure A.2: Prompt for the Kubernetes cluster whose bcm-gateway Gateway TLS certificate is to be configured

A prompt appears asking if an existing certificate pair should be used. Since a self-signed pair is to be used for testing purposes, no is selected (figure A.3)

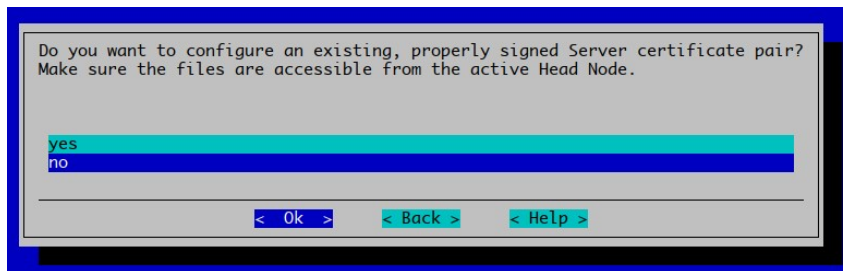


Figure A.3: Prompt to select existing certificate (choose “no” here for self-signed.)

A list of domains to customize is displayed (figure A.4)

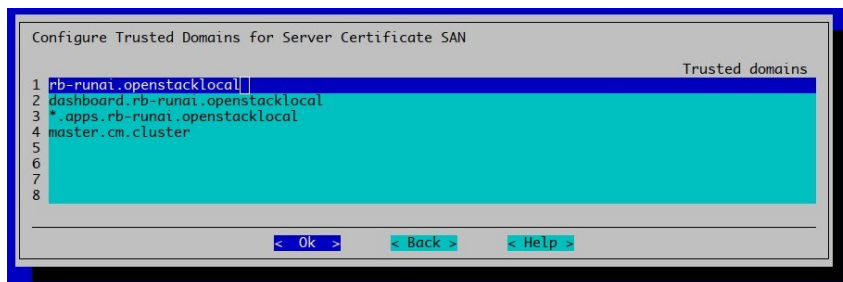


Figure A.4: Prompt for customizing trusted domains that need to be part of the self-signed certificate.

For this example, the domain is customized to superpod.nvidia.local (figure A.5)

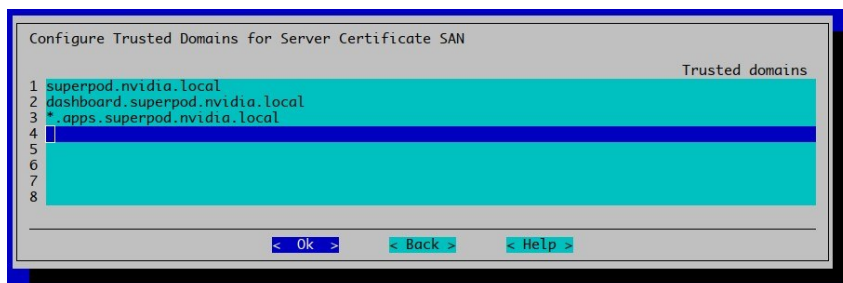


Figure A.5: Prompt answered with an entry superpod.nvidia.local

All the names entered end up in the SAN part of the certificate as valid DNS names. Continuing with Ok executes steps to take care of the configuration (figure A.6)

```

Executing 5 stages
##### Starting execution for 'Kubernetes Setup'
- kubernetes
- docker
## Progress: 0
### stage: kubernetes: Generate Self Signed CA
## Progress: 20
### stage: kubernetes: Generate Signed Server Cert Pair
## Progress: 40
### stage: kubernetes: Create Kubernetes Server Cert Pair Secret
## Progress: 60
### stage: kubernetes: Find Ingress Enabled
## Progress: 80
### stage: kubernetes: Patch Ingress Deployment
## Progress: 100

Took:    00:03 min.
Progress: 100/100
##### Finished execution for 'Kubernetes Setup', status: completed

Kubernetes Setup finished!

root@rb-runai:~# 

```

Figure A.6: The output when the self-signed configuration has completed.

So far in the procedure, the following has been carried out:

- a private CA key pair has been created (`gateway-ca.key` and `gateway-ca.crt` in figure A.7)
- a certificate (`bcm-gateway.crt`) has been made and signed by the private CA
- the `bcm-gateway` Gateway has been updated to present the certificate pair via the `gateway-bcm-gateway-cert` TLS Secret in the `bcm-gateway` namespace

```

root@rb-runai:~# ls -al /etc/kubernetes/pki/default/ | grep ingress-
-rw-r--r-- 1 root root 1017 Dec 14 11:19 ingress-ca.crt
-rw-r--r-- 1 root root 1675 Dec 14 11:19 ingress-ca.key
-rw-r--r-- 1 root root 1147 Dec 14 11:19 ingress-server.crt
-rw-r--r-- 1 root root 907 Dec 14 11:19 ingress-server.csr
-rw-r--r-- 1 root root 1675 Dec 14 11:19 ingress-server.key
root@rb-runai:~# 

```

Figure A.7: The certificate files on the active head node.

The default in this path is the label of the Kubernetes cluster for which the certificates have been created.

Specific Note on Run:ai The use of self-signed certificates is not recommended. It is only useful for testing purposes. The use of self-signed certificates may cause users can run into obscure issues, where it is hard to uncover that the problem is because of self-signed certificates.

In case the self-signed certificates are being used for a Run:ai SaaS deployment, the Run:ai cluster installer can be run again and all the fields can be configured (figure A.8).

run:ai Install Run.ai and get started

Prerequisites Setup Installation

Fill in the details below to set the parameters required for installing Run.ai

Tenant name
nvidialab

Application secret key
.....

Cluster URL
https:// superpod.nvidia.local:30443

Private key
ingress-server.key
Upload private key 1 (1.6KB)

Certificate
ingress-server.crt
Upload certificate 1 (1.1KB)

Need help? Contact [Run.ai support](#) or visit the [Run.ai documentation center](#)

BACK CONTINUE

Figure A.8: The Run.ai wizard.

In figure A.8 the domain `superpod.nvidia.local` is used, and some HTTPS port (served by a Gateway) is used for the cluster URL.

The private key for the certificate has been uploaded, from its location on the active head node at:
`/etc/kubernetes/pki/default/bcm-gateway.key`

The certificate has also been uploaded from its location on the active head node at:
`/etc/kubernetes/pki/default/bcm-gateway.crt`

The self-signed CA, taken from: `/etc/kubernetes/pki/default/gateway-ca.crt`, can alternatively be installed on the local machine (the cluster administrator laptop, for example) in order for the browser to recognize the certificate as trusted. Details are typically OS dependent, but for Chrome on a Linux system it follows a method of accepting the certificate by ignoring a warning about the site certificate being untrusted.

The warning is due to the CA server not being a recognized Certificate Authority (CA) like the CAs that are recognized by a browser.

If there is no internet access to the cluster URL, then the warning about the CA not being a recognized Certificate Authority is not an issue, and the user can simply accept the “untrusted” certificate.

If there is internet access to the cluster URL, then some cluster administrators may regard it as more secure to trust the self-signed certificate rather than external certificate authorities anyway.

B

Kubernetes Installation On An Air-gapped BCM Cluster

An air-gapped cluster environment is one where the cluster is not directly connected to the internet. Typically this is done to protect the cluster from remote intrusions.

Installation of Kubernetes in a non-air-gapped BCM cluster is covered in Chapter 4.

This appendix (Appendix B) describes how the following reference BCM air-gapped Kubernetes cluster can be created:

- The cluster is air-gapped
- The cluster uses Kubernetes version 1.34.7
- The cluster is running BCM version 11.33.0
- The BCM version is running on top of one of the following Linuxes:
 - Ubuntu 22.04
 - Ubuntu 24.04
 - Rocky Linux 9u3
 - RHEL 9u3

Appendix C describes how an existing BCM air-gapped Kubernetes cluster can be upgraded.

B.1 Initial Host Preparations For BCM Kubernetes Air-gapped Installation

The initial steps must be performed on an initial host computer that has internet access. The initial host requires:

- Internet access
 - An OS that matches that of the air-gapped cluster: Ubuntu 22.04, Ubuntu 24.04, Rocky Linux 9u3, or RHEL 9u3.
 - An architecture that matches that of the air-gapped cluster
 - Currently, the tarball supports one architecture at a time. So if the cluster has a mixed architecture, for example with the head node being x86-64 (using x86_64 RPM packages or amd64 APT packages), but the Kubernetes control plane nodes being ARMv8 (using arm64 packages), then that requires a more customized approach.
- For architecture configurations, the `--help` output of the download Bash scripts (format: `download_<descriptive label>.sh`) indicates how the `--arch` option can be used. These scripts are located in the `airgap-scripts` directory, described shortly.

- The BCM package repositories must be configured on the initial host. The relevant BCM repository list files can be copied over from a working cluster with the same OS.
 - For Ubuntu, these files are of the form `cm-*.list` and `ubuntu.sources`, and are located within the `/etc/apt/sources.list.d/` directory.

There are some known architecture-related bugs in the file specifications for the Ubuntu repositories. These should be fixed for the BCM 11.34 point release. The issues and associated workarounds for the 11.33 release are in the following files:

- * `ubuntu.sources`: The default Ubuntu sources file needs an architecture specification to be added. The default file can be replaced with the following deb822-format (man `sources.list.5`) text:

Example

```
root@basecm11:/etc/apt/sources.list.d# cat ubuntu.sources
Types: deb
URIs: http://archive.ubuntu.com/ubuntu/
Suites: noble noble-updates noble-backports
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
Architectures: amd64
```

```
Types: deb
URIs: http://security.ubuntu.com/ubuntu/
Suites: noble-security
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
Architectures: amd64
```

```
Types: deb
URIs: http://ports.ubuntu.com/ubuntu-ports/
Suites: noble noble-updates noble-backports
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
Architectures: arm64
```

```
Types: deb
URIs: http://ports.ubuntu.com/ubuntu-ports/
Suites: noble-security
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
Architectures: arm64
```

- * `cm-cuda-ubuntu2404-*.list`: The file for the default CUDA repositories list is modified as needed. Here, the `"*"` is either `x86_64` or `arm64`. The change is to allow CUDA packages to be picked up from whichever architecture is needed:

Example

```
root@basecm11:/etc/apt/sources.list.d# cat cm-cuda-ubuntu2404-x86_64.list
deb [signed-by=/usr/share/keyrings/cm-cuda-archive-keyring.gpg arch=amd64] https://\
developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/x86_64/ /
deb [signed-by=/usr/share/keyrings/cm-cuda-archive-keyring.gpg arch=arm64] https://\
developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/arm64/ /
```

- * `cm.list`. The BCM repository list file is modified so that the extra architectures that are needed are specified:

Example

```
root@basecm11:/etc/apt/sources.list.d# cat cm.list
deb [arch=amd64] http://<user>:<password>@updates-us-east.brightcomputing.com/\
deb/cm/11.0/amd64/ubuntu/ noble main
deb [arch=arm64] http://<user>:<password>@updates-us-east.brightcomputing.com/\
deb/cm/11.0/arm64/ubuntu/ noble main
deb [arch=amd64] http://<user>:<password>@updates.brightcomputing.com/\
deb/cm/11.0/amd64/ubuntu/ noble main
deb [arch=arm64] http://<user>:<password>@updates.brightcomputing.com/\
deb/cm/11.0/arm64/ubuntu/ noble main
```

In the preceding session:

- The architecture key/value entries are specified for the repositories in the square brackets, using the traditional one-line format, but using a deb822 format is also possible.
 - The URLs specified in the preceding session take the `<user>:<password>` format indicated. The values that are used for `<user>:<password>` should already be configured in the original `cm.list` repository file, and can simply be taken over.
 - The server and path of the URL on the actual cluster may differ from the preceding example.
- For RHEL variants these files should be `cm.repo` and `epel.repo`, located within `/etc/yum.repos.d/`
- The BCM `airgap-scripts` directory from the matching BCM version must be on the initial host. How the installation of that directory is carried out depends on if `cm-setup` is on the initial host, as explained next.

Initial Host Has `cm-setup` Installed

If the `cm-setup` package is already installed on the initial host, then for BCM version 11, the air-gap scripts are by default in the path defined by the environment variable `K8S_AG_SCRIPTS`.

Example

```
# echo $K8S_AG_SCRIPTS
/cm/local/apps/cm-setup/lib/python3.12/site-packages/cmsetup/plugins/kubernetes/airgap-scripts/
```

For BCM version 10, the Python directory path is `python3.9` instead.

This environment variable path should be added to the existing `PATH`:

```
# export PATH=$PATH:$K8S_AG_SCRIPTS
```

Initial Host Does Not Have `cm-setup` Installed

If the `cm-setup` package is not already installed on the initial host, then the administrator should define the air-gap scripts path, `$K8S_AG_SCRIPTS`, and then add the air-gap scripts to the `PATH` environment.

```
# export PATH=$PATH:$K8S_AG_SCRIPTS
```

Using The Air-gap Scripts

- `helm 3.x` (not `4.x` at the time of writing, April 2026) must be added to `/usr/local/bin/helm` as per the instructions at <https://helm.sh/docs/intro/install/>.

Alternatively, the following helper script, that BCM ships with `cm-setup`, can be used:

Example

```
root@internet-host:~# download_helm_binary.sh
Downloading helm v3.20.2 for linux-amd64...
Helm binary installed at /usr/local/bin/helm
root@internet-host:~#
```

Optionally, a Helm check can be carried out with:

Example

```
root@internet-host:~# helm version
version.BuildInfo{Version:"v3.20.2", GitCommit:"8fb76d6ab555577e98e23b7500009537a471feee",
  GitTreeState:"clean", GoVersion:"go1.25.9"}
```

- Skopeo, an image manager, is installed:

Example

```
root@internet-host:~# apt install skopeo          #or dnf install skopeo for RHEL variants
```

- To work around a potential Docker Hub pull rate limit issue (<https://docs.docker.com/docker-hub/usage/pulls/>) the administrator can now sign up for an account at <https://app.docker.com/signup>, and authenticate with `skopeo login docker.io`. This is to help with running the image installation scripts from the airgapped directory in the next bullet point. Any credentials are fine—the credentials will not end up in the air-gapped tarball or environment. They are only needed for authentication to conveniently pull images from Docker Hub. This is because unauthenticated pulls are limited to a rate of 100 per 6 hours. Authenticated pulls, on the other hand, are allowed at a rate of 200 per 6 hours, which should be sufficient for preparing the airgapped tarball.

- A working directory, for example `/root/airgapped`, can then be created, and installation scripts within the airgapped directory can then be run:

Example

```
root@internet-host:~# pwd
/root
root@internet-host:~# mkdir airgapped; cd airgapped
root@internet-host:~/airgapped# download_ubuntu_packages.sh --kube-version=1.35 # for Ubuntu Linux
...
[root@internet-host airgapped]# download_r9_packages.sh --kube-version=1.35      # for RHEL variants
...
root@internet-host:~/airgapped# download_container_images.sh --kube-version=1.35
...
root@internet-host:~/airgapped# download_helm_charts.sh
...
root@internet-host:~/airgapped# download_crds.sh
...
```

The preceding scripted downloads are for Kubernetes version 1.35. To use Kubernetes v1.33 or v1.34, the `--kube-version` flag should be adjusted accordingly. If `--kube-version` is not set, then the latest available version is picked up, which may not be compatible with what is to be deployed.

To ensure the commands all succeed, it is best to execute them in sequence and let them complete. The download scripts all write their output to the current working directory.

- Run:ai air-gapped setup

If Run:ai is required in the air-gapped installation, then NVIDIA Run:ai Support should be contacted to request the appropriate tarball. A typical name for such a tarball is:

```
runai-airgapped-package-2.23.60.tar.gz
```

A minimum version of 2.23.60 is required, since that is when multi-arch support was added.

The tarball file should be downloaded to the same directory. The directory contents should then appear similar to:

```
root@internet-host:~/airgapped# ls -alrth
total 14G
drwx----- 10 root root 287 Jun  8 04:21 .
drwxr-xr-x  4 root root  30 Jun  8 04:21 packages
drwxr-xr-x  4 root root  30 Jun  8 04:21 packages-head
drwxr-xr-x  3 root root  19 Jun  8 04:21 packages-non-dgx
drwxr-xr-x  3 root root  19 Jun  8 04:21 packages-cuda
drwxr-xr-x  2 root root  88 Jun  8 04:27 k8s-images
drwxr-xr-x  2 root root 4.0K Jun  8 04:29 helm-charts
drwxr-xr-x  2 root root  40 Jun  8 04:29 crds
drwxr-xr-x  9 root root 183 Jun  8 04:30 .
-rw-r--r--  1 root root 14G Jun  8 04:31 runai-airgapped-package-2.23.60.tar.gz
```

- The airgapped directory can now be tarballed.

In the tarball, the packages that are to be used by the software images must include all the architectures that are to be used by the Kubernetes cluster.

Example

```
root@internet-host:~/airgapped# ls
crds helm-charts k8s-images packages packages-cuda packages-head packages-non-dgx
runai-airgapped-package-2.23.60.tar.gz
root@internet-host:~/airgapped# cd ..
root@internet-host:~# tar -cvzf airgapped.tgz airgapped
```

The `airgapped.tgz` file should be saved to some transfer media (for example: a USB stick), and should be copied over to the active head node of the air-gapped cluster.

B.2 Head Node Preparation Steps

B.2.1 Preparing Docker Registry And Docker Installation

A local container registry (section 3.2) is needed on the air-gapped head node, so that Kubernetes operators can pull images. The registry itself is deployed in a later step, using the `cm-container-registry-setup` wizard (section ??). First, however, a `cm-kubernetes-setup.conf` configuration file that is aware of this registry must be generated.

The container registry can be configured with the `cm-container-registry-setup` wizard TUI running with special `airgap-` flags:

Example

```
cm-kubernetes-setup --airgap-registry <container_registry_address> --airgap-registry-as-helm
```

The `<container_registry_address>` part is replaced with the value: `master.cm.cluster:5000` for the default Docker registry installation. When the wizard reaches the summary screen, the setup is not deployed—it merely has its configuration saved, so that there is a working `cm-kubernetes-setup.conf` file available for when the actual deployment is carried out later on.

Selecting The Run:ai Operator Dependencies

When the `cm-kubernetes-setup` wizard is run, with the intent of running Run:ai as well, the operator selection screen must include the dependencies that Run:ai requires. For Run:ai 2.25, these are:

- NVIDIA GPU Operator
- Grafana Operator
- Kgateway
- Knative Operator
- Kubeflow Training Operator
- Kubernetes Metrics Server
- Kubernetes MPI Operator
- Kubernetes State Metrics
- LeaderWorkerSet Operator
- MetalLB
- Network Operator (technically optional, but recommended)
- NIM Operator (optional)
- Prometheus Adapter
- Prometheus Operator Stack

The Run:ai operator and its prerequisites are described in more detail in section 6.8.

B.2.2 Unpacking The Files

The `airgapped.tgz` file that was copied over to the head node can then be extracted. For Ubuntu, extracting to `/tmp` is recommended, because `apt` runs as the `apt` user, which does not have sufficient permissions in a normal directory to process the extracted files as needed.

Example

```
root@basecm11:~# mv airgapped.tgz /tmp/
root@basecm11:~# cd /tmp
root@basecm11:/tmp# tar xvf airgapped.tgz
```

The system packages can now be installed onto the head node and the required software images, using the untarred airgap system files together with the configuration file saved from earlier on:

Example

```
# cm-kubernetes-setup --airgap-install-system-packages -c /root/cm-kubernetes-setup.conf
```

Run:ai is deployed separately, after the Kubernetes cluster has been deployed, with the `cm-runai-setup` wizard (section B.2.8).

If the air-gapped cluster has a secondary head node (section 15.1 of the *Administrator Manual*), then the steps until here should be repeated for the secondary head node. However, instead of running exactly the same command as in the preceding example, the flag `--skip-headnode-check` must also be used for the secondary head node. A normal installation is not allowed from a secondary head node.

- On BCM versions 11.33 and earlier, the error message:

This node is not the active head node (PASSIVE)

is displayed on the passive head node during installation of the system packages. The user should select Skip to get past this message.

Package installation for multiarch software images in the Kubernetes cluster is automatic. This is assuming that the software images have the packages for the required architectures in the tgz file.

B.2.3 Configuring The Container Registry

The container registry can now be configured with:

Example

```
root@basecm11:~# cm-container-registry-setup --skip-packages
```

The `--skip-packages` option means that the wizard assumes that the packages needed are already installed.

When configuring using the registry wizard, the active head node should be selected by the cluster administrator as the node to be used for the deployment.

The TUI asks for hosts to be set for the SSL certificate (figure B.1):

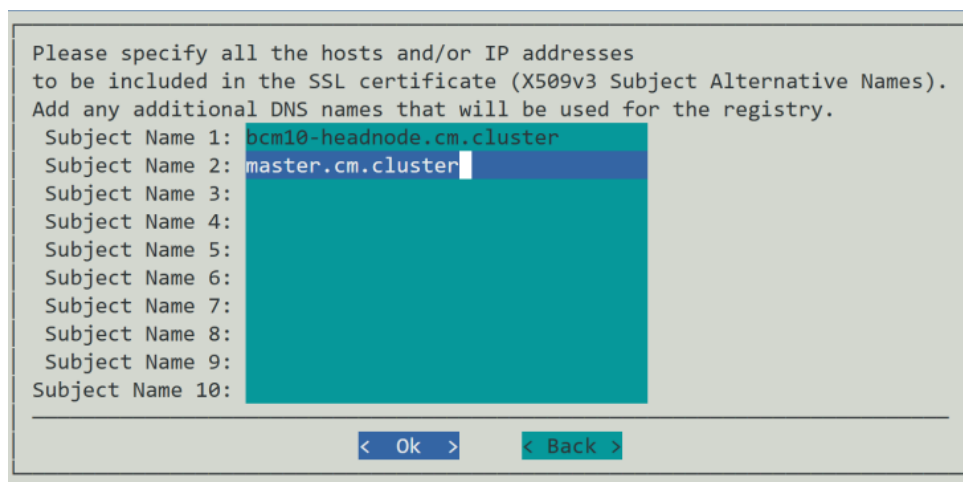


Figure B.1: SAN specification for SSL certificate

A custom domain name or additional subject name is set (`master.cm.cluster` in figure B.1), and the configuration is saved and deployed with the `save` and `deploy` option in the summary screen of the wizard.

Helm on the head node must be able to talk to this registry without TLS issues. The custom registry CA is therefore copied and trusted system-wide:

```
root@basecm11:~# cp /cm/local/apps/docker-registry/var/etc/$(hostname):5000-ca.crt \
    /usr/local/share/ca-certificates/bcm-docker-registry.crt
root@basecm11:~# update-ca-certificates
```

B.2.4 Configuring Docker For An Air-gapped Cluster

Docker can now be configured. Docker is currently only needed for the BCM Run:ai installation, and is configured with:

```
root@basecm11:~# cm-docker-setup --skip-packages
```

The active head node is chosen for the node for Docker, and the configuration is saved and deployed.

The successful deployment of Docker can be confirmed by running `docker ps` on the nodes on which it has been installed. The following output indicates readiness:

```
root@basecm11:~# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

Docker service initialization may take a few seconds.

Authenticating with `docker login` is not needed in this case, since images from Docker Hub are not going to be pulled.

B.2.5 Setting Up The Container Images And Helm Charts For An Air-gapped Cluster During Installation

From the airgapped directory, all Docker images and Helm charts can be pushed to the registry with:

```
root@basecm11:~# cm-kubernetes-setup --airgap-push-container-images -c /root/cm-kubernetes-setup.conf
root@basecm11:~# cm-kubernetes-setup --airgap-push-helm-charts -c /root/cm-kubernetes-setup.conf
```

B.2.6 Extracting The CRD Files For An Air-gapped Cluster During Installation

The downloaded CRD files can then be extracted using:

```
root@basecm11:~# $K8S_AG_SCRIPTS/extract_crds.sh
```

B.2.7 Deploying The Air-gapped Kubernetes Cluster

When the configuration is generated with the `--airgap-registry` and `--airgap-registry-as-helm` options (section ??), the `air-gap` section of `/root/cm-kubernetes-setup.conf` is populated automatically. No manual editing of the file is normally needed. For reference, the relevant section then appears similar to:

```
modules:
  kubernetes:
    airgap:
      helm:
        ca: '/cm/local/apps/containerd/var/etc/certs.d/master.cm.cluster:5000/ca.crt'
        repo: 'oci://master.cm.cluster:5000/helm-charts'
      registry: 'master.cm.cluster:5000'
      registry_username: ''
      registry_password: ''
```

If `Run:ai` is to be deployed, then the operators that `Run:ai` depends on must have been selected when this configuration was generated (section B.2.1).

Kubernetes can then be deployed with:

```
# cm-kubernetes-setup --airgap-install-system-packages -c /root/cm-kubernetes-setup.conf
```

`Run:ai` itself is not deployed by `cm-kubernetes-setup`. It is deployed afterwards with the separate `cm-runai-setup` wizard (section B.2.8).

B.2.8 Decoupled Run:ai Deployment

`Run:ai`'s separate wizard, `cm-runai-setup`, is provided by the `cm-setup-runai-bcm11` package. This package is normally already installed on the BCM head nodes. The `cm-kubernetes-setup` wizard is used to deploy Kubernetes itself and the operators that `Run:ai` depends on. The `Run:ai` ver-

sion deployed with BCM point release 11.33 is 2.25.x, with an associated air-gapped tarball package `runai-airgapped-package-2.25.12.tar.gz`.

Obtaining A Newer `cm-runai-setup` Build

The `cm-setup-runai-bcm11` package that provides the `cm-runai-setup` wizard is normally already installed on the head node. Updating it is only needed if a version of Run:ai is to be tested that has not yet been released by the NVIDIA Mission Control team.

To obtain a newer build for an air-gapped cluster, the package is first downloaded on the internet-connected host, while the air-gapped tarball is being prepared. The development repository package is installed first. Even though the URL in the example that follows says `x86_64`, this is a noarch package that also works on `arm64`:

Example

```
root@internet-host:~# wget http://dev.bcm.nvidia.com/bright/packages/x86_64/11.0/\
ubuntu/2404/all/cm-config-apt-nmc-dev.all.deb
root@internet-host:~# apt install -y ./cm-config-apt-nmc-dev.all.deb
root@internet-host:~# apt update
root@internet-host:~# apt download cm-setup-runai-bcm11
```

This downloads a package `cm-setup-runai-bcm11_2.3.0~pr.194.1968b4d8_amd64.deb` (the exact name can vary). The `.deb` file should be placed inside the `airgapped` directory, so that it is included in the tarball.

On the air-gapped head node, after the system packages have been installed (section B.2), the newer wizard package can then be installed.

Example

```
root@basecm11:~# apt install ./cm-setup-runai-bcm11_2.3.0~pr.194.1968b4d8_amd64.deb
```

Running The `cm-runai-setup` Wizard

After Kubernetes has been deployed, together with the Run:ai operator dependencies, Run:ai itself is deployed with the `cm-runai-setup` wizard.

For an air-gapped deployment, the wizard must be run with the `--skip-packages` flag. Without this flag, the automatic update feature of the wizard attempts to reach the internet, and fails.

Example

```
root@basecm11:~# cm-runai-setup --skip-packages
```

The wizard prompts for:

- the deployment type (self-hosted)
- the Run:ai version (2.25.x is chosen here; the x is replaced by the latest available patch version for the chosen minor version)
- the node category for the Run:ai control plane (the same category as the Kubernetes control-plane nodes is recommended)
- the LoadBalancer IP address

If the wizard detects that it is deploying onto an air-gapped Kubernetes cluster, then it also prompts for the location of the Run:ai air-gapped tarball.

At the summary screen, `Save & Exit` must be selected. Deploying in a first run is not possible at the time of writing (June 2026) of this section due to a bug.

The configuration is saved to `/root/cm-runai-setup.conf`. The deployment is then carried out with:

Example

```
root@basecm11:~# cm-runai-setup --skip-packages -c /root/cm-runai-setup.conf
```

The `--skip-packages` flag is only needed for air-gapped deployments.

When the setup completes, Run:ai should be up and running. This can be checked with, for example:

Example

```
root@basecm11:~# curl -s https://runai.superpod-nvidia.local | grep title
<title>Run.ai</title>
```

Uninstalling Kubernetes And Run:ai

Uninstalling Kubernetes also removes Run:ai. The simplest method is to run the `cm-kubernetes-setup` wizard and select the uninstall option. Alternatively, the removal can be carried out from the command line:

Example

```
root@basecm11:~# cm-kubernetes-setup --cluster <kube cluster> --remove --yes-i-really-mean-it
```

To remove only Run:ai, while leaving the Kubernetes cluster in place, the `cm-runai-setup` wizard provides an uninstall option. Equivalently, the removal can be carried out from the command line:

Example

```
root@basecm11:~# cm-runai-setup --cluster <kube cluster> --remove --yes-i-really-mean-it
```

B.2.9 Post-installation Automatic Provisioning

Section B.2 describes installing packages from the air-gapped tarball into software images. When deploying new software images for worker nodes, BCM cannot automatically provision them in air-gapped environments; this automatic provisioning functionality is available only in non-air-gapped clusters.

For air-gapped clusters, administrators must manually execute the relevant installation commands, adjusting the configuration file to specify the appropriate target nodes and node categories.

C

Kubernetes Upgrade On An Air-gapped BCM Cluster

Section 4.22 covers Kubernetes upgrades in the case of a BCM Kubernetes cluster connected directly to the internet. An air-gapped cluster environment is one where the cluster is not directly connected to the internet.

A BCM cluster that air-gapped and already runs Kubernetes can be upgraded with the procedure outlined in this current section (section C). It is based on the upgrade procedure in section 4.22. The main differences are that the BCM version requirements differ, and that a local registry must be built.

Appendix B describes how an air-gapped cluster can be created if it does not already exist.

C.1 Air-gapped Kubernetes Upgrade Prerequisites

Before beginning the upgrade, the following general prerequisites and version checks must be carried out:

- The cluster version must be at a sufficiently high point release number (section 9.1.2 of the *Administrator Manual*):
 - for BCM version 10 the point release version number must be 10.31.0 or later
 - for BCM version 11 the point release version number must be 11.32.0 or later
- The upgrade path should take into account the Kubernetes version skew policy, as defined at <https://kubernetes.io/releases/version-skew-policy/>. The principles to take into account are:
 - Only two adjacent minor releases may exist in the cluster at any time. So, v1.34 and v1.35 are acceptable; but v1.33, v1.34, and v1.35 are not acceptable.
 - The target version must not violate the skew values. So, if the cluster nodes are currently on two different adjacent minor releases, then the only valid upgrade path is to the newest of the two minor releases. For example if some nodes are running v1.34 and some are running v1.35, then the only valid upgrade path is to upgrade the v1.34 nodes to v1.35. This ensures that the cluster remains functional and compliant with the Kubernetes version skew policies during the rolling upgrade process.
- The upgrade path should also take into account the NVIDIA BCM Release Notes and Compatibility Matrix corresponding to the BCM version that is running. This must be consulted, to see if there is support for the desired target Kubernetes version (for example v1.35.x).
- An upgrade path preference, for minor or patch, must be decided on for the case where all the nodes are on the same minor version:

- Minor release upgrade: the latest release of the next minor release (for example: v1.34 to v1.35)
- Patch release upgrade: the latest patch release of the current minor release (for example: v1.35.3 to v1.35.4)
- The current Kubernetes cluster must already be operational, with healthy and ready nodes.
- A local registry must already be deployed, and its FQDN must be retrievable (for example: master.cm.cluster:5000)
- The upstream instructions on preparing for the upgrade (<https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade/#before-you-begin>) should be read.
- A node that is to be upgraded must be drained.

C.2 Initial Host Preparations For BCM Kubernetes Air-gapped Upgrade

The initial steps must be performed on an initial host computer that has internet access. The initial host requires:

- Internet access
 - An OS that matches that of the air-gapped cluster: Ubuntu 22.04, Ubuntu 24.04, Rocky Linux 9u3, or RHEL 9u3.
 - An architecture that matches that of the air-gapped cluster
 - Currently, the tarball supports one architecture at a time. So if the cluster has a mixed architecture, for example with the head node being x86-64 (using x86_64 RPM packages or amd64 APT packages), but the Kubernetes control plane nodes being ARMv8 (using arm64 packages), then that requires a more customized approach.
- For architecture configurations, the `--help` output of the download Bash scripts (format: `download_<descriptive label>.sh`) indicates how the `--arch` option can be used. These scripts are located in the `airgap-scripts` directory, described shortly.
- The BCM package repositories must be configured on the initial host. The relevant BCM repository list files can be copied over from a working cluster with the same OS.
 - For Ubuntu, these files are of the form `cm-*.list` and `ubuntu.sources`, and are located within the `/etc/apt/sources.list.d/` directory.

There are some known architecture-related bugs in the file specifications for the Ubuntu repositories. These should be fixed for the BCM 11.34 point release. The issues and associated workarounds for the 11.33 release are in the following files:

- * `ubuntu.sources`: The default Ubuntu sources file needs an architecture specification to be added. The default file can be replaced with the following deb822-format (man `sources.list.5`) text:

Example

```
root@basecm11:/etc/apt/sources.list.d# cat ubuntu.sources
Types: deb
URIs: http://archive.ubuntu.com/ubuntu/
Suites: noble noble-updates noble-backports
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
Architectures: amd64
```

```
Types: deb
URIs: http://security.ubuntu.com/ubuntu/
Suites: noble-security
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
Architectures: amd64
```

```
Types: deb
URIs: http://ports.ubuntu.com/ubuntu-ports/
Suites: noble noble-updates noble-backports
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
Architectures: arm64
```

```
Types: deb
URIs: http://ports.ubuntu.com/ubuntu-ports/
Suites: noble-security
Components: main restricted universe multiverse
Signed-By: /usr/share/keyrings/ubuntu-archive-keyring.gpg
Architectures: arm64
```

- * `cm-cuda-ubuntu2404-*.list`: The file for the default CUDA repositories list is modified as needed. Here, the `"*"` is either `x86_64` or `arm64`. The change is to allow CUDA packages to be picked up from whichever architecture is needed:

Example

```
root@basecm11:/etc/apt/sources.list.d# cat cm-cuda-ubuntu2404-x86_64.list
deb [signed-by=/usr/share/keyrings/cm-cuda-archive-keyring.gpg arch=amd64] https://\
developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/x86_64/ /
deb [signed-by=/usr/share/keyrings/cm-cuda-archive-keyring.gpg arch=arm64] https://\
developer.download.nvidia.com/compute/cuda/repos/ubuntu2404/arm64/ /
```

- * `cm.list`. The BCM repository list file is modified so that the extra architectures that are needed are specified:

Example

```
root@basecm11:/etc/apt/sources.list.d# cat cm.list
deb [arch=amd64] http://<user>:<password>@updates-us-east.brightcomputing.com/\
deb/cm/11.0/amd64/ubuntu/ noble main
deb [arch=arm64] http://<user>:<password>@updates-us-east.brightcomputing.com/\
deb/cm/11.0/arm64/ubuntu/ noble main
deb [arch=amd64] http://<user>:<password>@updates.brightcomputing.com/\
deb/cm/11.0/amd64/ubuntu/ noble main
deb [arch=arm64] http://<user>:<password>@updates.brightcomputing.com/\
deb/cm/11.0/arm64/ubuntu/ noble main
```

In the preceding session:

- The architecture key/value entries are specified for the repositories in the square brackets, using the traditional one-line format, but using a `deb822` format is also possible.
- The URLs specified in the preceding session take the `<user>:<password>` format indicated. The values that are used for `<user>:<password>` should already be configured in the original `cm.list` repository file, and can simply be taken over.
- The server and path of the URL on the actual cluster may differ from the preceding example.

- For RHEL variants these files should be `cm.repo` and `epel.repo`, located within `/etc/yum.repos.d/`
- The BCM `airgap-scripts` directory from the matching BCM version must be on the initial host. How the installation of that directory is carried out depends on if `cm-setup` is on the initial host, as explained next.

Initial Host Has `cm-setup` Installed

If the `cm-setup` package is already installed on the initial host, then for BCM version 11, the air-gap scripts are by default in the path defined by the environment variable `K8S_AG_SCRIPTS`.

Example

```
# echo $K8S_AG_SCRIPTS
/cm/local/apps/cm-setup/lib/python3.12/site-packages/cmsetup/plugins/kubernetes/airgap-scripts/
```

For BCM version 10, the Python directory path is `python3.9` instead.
This environment variable path should be added to the existing `PATH`:

```
# export PATH=$PATH:$K8S_AG_SCRIPTS
```

Initial Host Does Not Have `cm-setup` Installed

If the `cm-setup` package is not already installed on the initial host, then the administrator should define the air-gap scripts path, `$K8S_AG_SCRIPTS`, and then add the air-gap scripts to the `PATH` environment.

```
# export PATH=$PATH:$K8S_AG_SCRIPTS
```

Using The Air-gap Scripts

- `helm 3.x` (not `4.x` at the time of writing, April 2026) must be added to `/usr/local/bin/helm` as per the instructions at <https://helm.sh/docs/intro/install/>.

Alternatively, the following helper script, that BCM ships with `cm-setup`, can be used:

Example

```
root@internet-host:~# download_helm_binary.sh
Downloading helm v3.20.2 for linux-amd64...
Helm binary installed at /usr/local/bin/helm
root@internet-host:~#
```

Optionally, a Helm check can be carried out with:

Example

```
root@internet-host:~# helm version
version.BuildInfo{Version:"v3.20.2", GitCommit:"8fb76d6ab555577e98e23b7500009537a471feee",
  GitTreeState:"clean", GoVersion:"go1.25.9"}
```

- Skopeo, an image manager, is installed:

Example

```
root@internet-host:~# apt install skopeo          #or dnf install skopeo for RHEL variants
```

- To work around a potential Docker Hub pull rate limit issue (<https://docs.docker.com/docker-hub/usage/pulls/>) the administrator can now sign up for an account at <https://app.docker.com/signup>, and authenticate with `skopeo login docker.io`. This is to help with running the image installation scripts from the airgapped directory in the next bullet point. Any credentials are fine—the credentials will not end up in the air-gapped tarball or environment. They are only needed for authentication to conveniently pull images from Docker Hub. This is because unauthenticated pulls are limited to a rate of 100 per 6 hours. Authenticated pulls, on the other hand, are allowed at a rate of 200 per 6 hours, which should be sufficient for preparing the airgapped tarball.
- A working directory, for example `/root/airgapped`, can then be created, and installation scripts within the airgapped directory can then be run:

Example

```
root@internet-host:~# pwd
/root
root@internet-host:~# mkdir airgapped; cd airgapped
root@internet-host:~/airgapped# download_ubuntu_packages.sh --kube-version=1.35 # for Ubuntu Linux
...
[root@internet-host airgapped]# download_r9_packages.sh --kube-version=1.35 # for RHEL variants
...
root@internet-host:~/airgapped# download_container_images.sh --kube-version=1.35
...
root@internet-host:~/airgapped# download_helm_charts.sh
...
root@internet-host:~/airgapped# download_crds.sh
...
```

The preceding scripted downloads are for Kubernetes version 1.35. To use Kubernetes v1.33 or v1.34, the `--kube-version` flag should be adjusted accordingly. If `--kube-version` is not set, then the latest available version is picked up, which may not be compatible with what is to be deployed. To ensure the commands all succeed, it is best to execute them in sequence and let them complete. The download scripts all write their output to the current working directory.

- Run:ai air-gapped setup

If Run:ai is required in the air-gapped installation, then NVIDIA Run:ai Support should be contacted to request the appropriate tarball. A typical name for such a tarball is:

```
runai-airgapped-package-2.23.60.tar.gz
```

A minimum version of 2.23.60 is required, since that is when multi-arch support was added.

The tarball file should be downloaded to the same directory. The directory contents should then appear similar to:

```
root@internet-host:~/airgapped# ls -alrth
total 14G
drwx----- 10 root root 287 Jun  8 04:21 ..
drwxr-xr-x  4 root root  30 Jun  8 04:21 packages
drwxr-xr-x  4 root root  30 Jun  8 04:21 packages-head
drwxr-xr-x  3 root root  19 Jun  8 04:21 packages-non-dgx
drwxr-xr-x  3 root root  19 Jun  8 04:21 packages-cuda
drwxr-xr-x  2 root root  88 Jun  8 04:27 k8s-images
drwxr-xr-x  2 root root 4.0K Jun  8 04:29 helm-charts
```

```
drwxr-xr-x  2 root root   40 Jun  8 04:29 crds
drwxr-xr-x  9 root root  183 Jun  8 04:30 .
-rw-r--r--  1 root root 14G Jun  8 04:31 runai-airgapped-package-2.23.60.tar.gz
```

- The airgapped directory can now be tarballed.

In the tarball, the packages that are to be used by the software images must include all the architectures that are to be used by the Kubernetes cluster.

Example

```
root@internet-host:~/airgapped# ls
crds helm-charts k8s-images packages packages-cuda packages-head packages-non-dgx
runai-airgapped-package-2.23.60.tar.gz
root@internet-host:~/airgapped# cd ..
root@internet-host:~# tar -cvzf airgapped.tgz airgapped
```

The `airgapped.tgz` file should be saved to some transfer media (for example: a USB stick), and should be copied over to the active head node of the air-gapped cluster.

C.3 Head Node Preparation Steps

Selecting The Run:ai Operator Dependencies

When the `cm-kubernetes-setup` wizard is run, with the intent of running Run:ai as well, the operator selection screen must include the dependencies that Run:ai requires. For Run:ai 2.25, these are:

- NVIDIA GPU Operator
- Grafana Operator
- Kgateway
- Knative Operator
- Kubeflow Training Operator
- Kubernetes Metrics Server
- Kubernetes MPI Operator
- Kubernetes State Metrics
- LeaderWorkerSet Operator
- MetalLB
- Network Operator (technically optional, but recommended)
- NIM Operator (optional)
- Prometheus Adapter
- Prometheus Operator Stack

The Run:ai operator and its prerequisites are described in more detail in section 6.8.

C.3.1 Unpacking The Files

The `airgapped.tgz` file that was copied over to the head node can then be extracted. For Ubuntu, extracting to `/tmp` is recommended, because `apt` runs as the `apt` user, which does not have sufficient permissions in a normal directory to process the extracted files as needed.

Example

```
root@basecm11:~# mv airgapped.tgz /tmp/
root@basecm11:~# cd /tmp
root@basecm11:/tmp# tar xvf airgapped.tgz
```

The system packages can now be installed onto the head node and the required software images, using the untarred airgap system files together with the configuration file saved from earlier on:

Example

```
# cm-kubernetes-setup --airgap-install-system-packages -c /root/cm-kubernetes-setup.conf
```

If the air-gapped cluster has a secondary head node (section 15.1 of the *Administrator Manual*), then the steps until here should be repeated for the secondary head node. However, instead of running exactly the same command as in the preceding example, the flag `--skip-headnode-check` must also be used for the secondary head node. A normal installation is not allowed from a secondary head node.

- On BCM versions 11.33 and earlier, the error message:

```
This node is not the active head node (PASSIVE)
```

is displayed on the passive head node during installation of the system packages. The user should select `Skip` to get past this message.

Package installation for multiarch software images in the Kubernetes cluster is automatic. This is assuming that the software images have the packages for the required architectures in the `tgz` file.

C.3.2 Setting Up The Container Images And Helm Charts For An Air-gapped Cluster During Upgrade

From the airgapped directory, all Docker images and Helm charts can be pushed to the registry with:

```
root@basecm11:~# cm-kubernetes-setup --airgap-push-container-images -c /root/cm-kubernetes-setup.conf
root@basecm11:~# cm-kubernetes-setup --airgap-push-helm-charts -c /root/cm-kubernetes-setup.conf
```

C.3.3 Extracting The CRD Files For An Air-gapped Cluster During Upgrade

The downloaded CRD files can then be extracted using:

```
root@basecm11:~# $K8S_AG_SCRIPTS/extract_crds.sh
```

C.4 Upgrading The Kubernetes Cluster

The upgrade steps start with the control plane nodes.

C.4.1 Updating The First Control Plane Node

The node that is to be upgraded should first be updated to the latest binaries. This can be done with an `imageupdate` run from `cmsh` on the head node (section 5.6.2 of the *Administrator Manual*).

For example for a node with the arbitrary name of `controlplane01`:

Example

```

root@headnode:~# cmsh -c 'device; imageupdate -w controlplane01 --wait'
...
...[notice] headnode: Provisioning started: sending headnode:/cm/images/default-image
to controlplane01:/, mode UPDATE, dry run = no
...
...[notice] headnode: Provisioning completed: sent headnode:/cm/images/default-image
to controlplane01:/, mode UPDATE, dry run = no

```

If the `-w|--write` flag is not used, then a dry-run is carried out.

After the image update is done, the `kubeadm upgrade plan` command is run from the node itself:

Example

```

root@headnode:~# ssh controlplane01
...
root@controlplane01:~# kubeadm upgrade plan --kubeconfig=/root/.kube/config-default

```

Explicitly providing the kubeconfig for the correct Kubernetes cluster is a good practice. This is important if the BCM cluster has multiple Kubernetes clusters.

The following session shows output from an upgrade plan from 1.32.10 to 1.33.11. The plan shows that there is indeed a newer 1.32.11 available already:

Example

```

...
[upgrade/versions] Target version: v1.32.10
[upgrade/versions] Latest version in the v1.32 series: v1.32.11

```

Components that must be upgraded manually after you have upgraded the control plane with `'kubeadm upgrade apply'`:

COMPONENT	NODE	CURRENT	TARGET
kubelet	node001	v1.32.10	v1.32.11
kubelet	node002	v1.32.10	v1.32.11
kubelet	node003	v1.32.10	v1.32.11
kubelet	node004	v1.32.10	v1.32.11
kubelet	node005	v1.32.10	v1.32.11
kubelet	node006	v1.32.10	v1.32.11
kubelet	node007	v1.32.10	v1.32.11
kubelet	node008	v1.32.10	v1.32.11

Upgrade to the latest version in the v1.32 series:

COMPONENT	NODE	CURRENT	TARGET
kube-apiserver	node001	v1.32.10	v1.32.11
kube-apiserver	node002	v1.32.10	v1.32.11
kube-apiserver	node003	v1.32.10	v1.32.11
kube-controller-manager	node001	v1.32.10	v1.32.11
kube-controller-manager	node002	v1.32.10	v1.32.11
kube-controller-manager	node003	v1.32.10	v1.32.11
kube-scheduler	node001	v1.32.10	v1.32.11
kube-scheduler	node002	v1.32.10	v1.32.11

kube-scheduler	node003	v1.32.10	v1.32.11
kube-proxy		1.32.10	v1.32.11
CoreDNS		v1.11.3	v1.12.0

You can now apply the upgrade by executing the following command:

```
kubeadm upgrade apply v1.32.11
```

The upgrade plan suggests that an upgrade to v1.32.11 must be done first.

However, the skew versions policies (page 179) allow jumping to v1.33.6.

A check of the changelogs through the appropriate versions should be done. The changelogs are located under: <https://github.com/kubernetes/kubernetes/blob/master/CHANGELOG/> and are well-organized. After studying these, if the cluster administrator decides it makes sense to just jump ahead to v1.33.6, then that can be carried out as follows:

Instead using v1.32.11 as suggested in the preceding session plan, the value v1.33.6 is used. The `--kubeconfig` flag is also set as a matter of best practice:

Example

```
root@controlplane01:~# kubeadm upgrade apply v1.33.6 --kubeconfig=/root/.kube/config-default
```

The following should then show in the output:

Example

```
[upgrade] SUCCESS! A control plane node of your cluster was upgraded to "v1.33.6".
```

Air-gapped Control Plane Validation

- checking that the first control plane node is in the Ready state, and running the target Kubernetes version

Example

```
root@controlplane01:~# kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
basecm11	Ready	control-plane,master	6h7m	v1.33.6
controlplane01	Ready	control-plane,master,worker	6h5m	v1.33.6
...				

- checking that all critical pods are running and in the Ready state, especially those in kube-system namespace:

Example

```
root@controlplane01:~# kubectl get pod -A
```

NAMESPACE	NAME	READY	STATUS	RESTARTS	AGE
calico-system	calico-apiserver-5699d94dbc-hmj76	1/1	Running	0	6h11m
calico-system	calico-apiserver-5699d94dbc-hmtmw	1/1	Running	0	6h11m
calico-system	calico-kube-controllers-67c964bc98-5957z	1/1	Running	0	6h11m
calico-system	calico-node-2jrxv	1/1	Running	0	6h11m
...					
kube-system	coredns-66bc5c9577-lsvdt	1/1	Running	0	6h13m
kube-system	coredns-66bc5c9577-w7jhm	1/1	Running	0	6h13m
kube-system	kube-apiserver-basecm11	1/1	Running	0	6h12m
kube-system	kube-apiserver-controlplane01	1/1	Running	0	6h8m
kube-system	kube-controller-manager-basecm11	1/1	Running	1	6
...					

An extra check is to see if the appropriate kube-apiserver version image, v1.33.6, is being used by the upgraded controlplane01 node, by executing:

Example

```
root@controlplane01:~# kubectl describe pod -n kube-system kube-apiserver-controlplane01 | grep Image:
Image:          master.cm.cluster:5000/kube-apiserver:v1.33.6
```

Updating Subsequent Control Plane Nodes

The following commands can be repeated on the other control plane nodes.

From the head node:

Example

```
root@basecm11:~# cmsh -c 'device; imageupdate -w <node> --wait'
```

From the node itself:

Example

```
root@node001:~# kubectl upgrade node --kubeconfig /root/.kube/config-default
```

In the preceding, an upgrade apply with version does not need to be provided.

The validation steps from before (page 187) should be repeated.

If all control-plane nodes are running without issues, then the worker nodes can also be updated.

Updating The Worker Nodes In The Air-gapped Update

Updating worker nodes is done in a similar way to updating control plane nodes, but usually with some additional care, since worker nodes can be running user workloads.

Section 4.22.6, about updating the worker nodes in a non-air-gapped cluster, has more on being careful during worker node updates. In particular, it explains how to drain nodes beforehand, so that end users do not get upset by lost jobs.

The upgrade step itself is carried out with:

Example

```
root@workernode001:~# kubectl upgrade node --kubeconfig /root/.kube/config-default
```

The --kubeconfig flag should be provided in the preceding command.

To summarize, a node that needs to be drained requires the following commands:

Draining On The Head Node

Example

```
root@basecm11:~# kubectl drain $host --ignore-daemonsets # check if drain succeeds
```

```
# update its software image if needed
```

```
root@basecm11:~# cmsh -c "device use $host; imageupdate -w --wait"
```

Draining On The Non-head Node

Example

```
root@<node>:~# kubectl upgrade node --kubeconfig=/root/.kube/config-<Kubernetes instance>
```

```
root@<node>:~# systemctl daemon-reload
```

```
root@<node>:~# systemctl restart kubelet
```

```
root@<node>:~# kubectl uncordon $host # check if uncordon succeeds
```

Validation can be done, as before (page 187), by:

- checking that all critical pods are running and in the Ready state, especially those in kube-system namespace, by running: `kubectl get nodes`
- checking that all critical pods are running and in the Ready state (especially those in kube-system namespace, but not only those), by running: `kubectl get pod -A`