APPLICATION NOTE

Intel Corporation
Software Defined Datacenter Solutions Group

Enhanced Platform Awareness in Kubernetes

Table of Contents

1.0 Introduction

Enhanced Platform Awareness (EPA) represents a methodology and a related suite of changes across multiple layers of the orchestration stack targeting intelligent platform capability, configuration & capacity consumption. For communications service providers (CommSPs) using virtual network functions (VNFs) to provision services, EPA delivers improved application performance, and input/output throughput and determinism.

EPA underpins a three-fold objective of the discovery, scheduling and isolation of server hardware capabilities. This document is a comprehensive description of EPA capabilities and the related platform features on Intel® Xeon® Scalable processor-based systems exposed by EPA. Intel and partners have worked together to make these technologies available in Kubernetes:

- Node Feature Discovery (NFD) enables Intel Xeon® Processor server hardware capability discovery in Kubernetes
- CPU Manager for Kubernetes (CMK) provides a mechanism for CPU pinning and isolation of containerized workloads
- Huge page support is native in Kubernetes v1.8 and enables the discovery, scheduling and allocation of huge pages as a native first-class resource
- Single Root I/O Virtualization (SR-IOV) for networking

This document details the setup and installation of the above-mentioned technologies. It is written for developers and architects who want to integrate the new technologies into their Kubernetes-based networking solutions in order to achieve the improved network I/O, deterministic compute performance, and server platform sharing benefits offered by Intel Xeon Processor-based platforms.

For detailed performance results, please refer to performance benchmark report titled "Kubernetes and Container Bare Metal Platform for NFV use cases for Intel® Xeon® Gold Processor 6138T." This document can be found in Appendix table A.3-2.

Note: The document does not describe how to setup a Kubernetes cluster; it is assumed that this is available to the reader before undertaking the steps in this document. For more setup and installation guidelines of a complete system, providing a reference architecture for container bare metal setup, please refer to the user guide titled "Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors." Access to this document can be found in Appendix table A.3-2.

This document is part of the Container Experience Kit for EPA. Container Experience Kits are collections of user guides, application notes, feature briefs and other collateral that provide a library of best-practice documents for engineers who are developing container-based applications. Other documents in this Container Experience Kit can be found online at: https://networkbuilders.intel.com/network-technologies/container-experience-kits.

These documents include:

Document Title	Document Type
Enhanced Platform Awareness in Kubernetes	Feature Brief
Enabling New Features with Kubernetes for NFV	White Paper
Kubernetes and Container Bare Metal on Intel Xeon Scalable Plaform for NFV Use Cases	Performance Benchmarking Report

2.0 Enhanced Platform Awareness in Kubernetes

Prior to v1.8, Kubernetes provided very limited support for the discovery and scheduling of cluster server resources. The only managed resources were the CPU, the graphics processor unit (GPU), memory and opaque integer resources (OIR). Kubernetes is evolving to incorporate a wider variety of hardware features and capabilities enabling new use cases with granular resource scheduling policies. This also allows meeting stricter service level agreements (SLAs) and ad-hoc device management. For example, in v1.8, delivered in Sept. 2017, Kubernetes has introduced support for:

- Huge page memory
- Basic CPU pinning and isolation
- OIR was introduced to account for additional resources not managed by Kubernetes. Extended resources is the evolution of OIR.
- Device plugins

The Kubernetes community, Intel included, are continuing to work to develop support for EPA, both natively and via plugin frameworks. This has positively impacted the role of Kubernetes as a convergence platform, broadening the scope of supported use cases and maximizing server resource utilization. Figure 1 shows the solutions introduced by the industry and their current status. The following is a introduction of the resources supported today:

- **CPU Pinning and Isolation**: Under normal circumstances, the kernel task scheduler will treat all CPUs as available for scheduling process threads and regularly preempts executing process threads to give CPU time to other applications. The positive side effect is multitasking and more efficient CPU resource utilization. The negative side effect is non-deterministic behavior which makes it unsuitable for latency sensitive workloads. A solution for these workloads is to 'isolate' a CPU, or a set of CPUs, from the kernel scheduler, such that it will never schedule a process thread there. Then the latency sensitive workload process thread(s) can be pinned to execute on that isolated CPU set only, providing them exclusive access to that CPU set. This results in more deterministic behavior due to reduced or eliminated thread preemption and maximizing CPU cache utilization. While beginning to guarantee the deterministic behavior of priority workloads, isolating CPUs also addresses the need to manage resources permitting multiple VNFs to coexist on the same physical server.
- **Huge Pages**: On Intel CPUs, physical memory is managed in units called pages, organized and managed by the operating system using an in-memory table, called a page table. This table contains one entry per page, the purpose of which is the translation of user space virtual page addresses to physical page addresses. The default page size is 4KB which, on systems with large amounts of memory, results in very large page tables. Walking the page tables to recover mappings is computationally expensive, so a cache called the Translation Lookaside Buffer (TLB) is used by the Memory Management Unit (MMU) to store recent mappings.

  In addition, to support more virtual memory than physically available in the system, the operating system can 'page out' memory pages to a secondary storage disk. While these mechanisms provide security by isolating each process in a virtual memory address space and more efficient utilization of the physical memory resource, deterministic latency for memory access is not achievable as a consequence of the multiple levels of indirection. To address this, Intel CPUs support huge pages, which are pages of size larger than 4 KB. Typically, 2 MB or 1 GB huge page sizes are used and the behavior and impact of these pages are:

  - Fewer larger pages results in reduced size page tables meaning a page table walk is less expensive

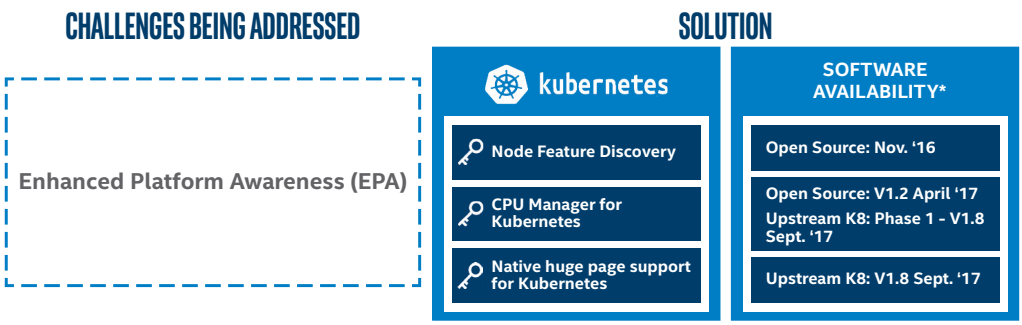


Figure 1. Solutions introduced by the industry and their current status.

- Fewer page table entries increases the chances of a TLB cache hit
- By default, huge pages are locked in memory, meaning that the worst-case scenario of page recovery from disk is eliminated

Collectively, huge pages have a very positive impact on memory access latencies. Single Root I/O Virtualization (SR-IOV) for networking: SR-IOV is a PCI-SIG standardized method for isolating PCI Express (PCIe) native hardware resources for manageability and performance reasons. In effect, this means a single PCIe device, referred to as the physical function (PF), can appear as multiple separate PCIe devices, referred to as virtual functions (VF), with the required resource arbitration occurring in the device itself. In the case of an SR-IOV-enabled network interface card (NIC), each VFs MAC and IP address can be independently configured and packet switching between the VFs occurs in the device hardware. The benefits of using SR-IOV devices in networking Kubernetes pods include:

- Direct communication with the NIC device allows for close to "bare-metal" performance.
- Support for multiple simultaneous fast network packet processing (for example based on Data Plane Development Kit (DPDK)) workloads in user space.
- Leveraging of NIC accelerators and offloads per workload.

The following sections will cover Node Feature Discovery, CPU Manager for Kubernetes and huge page support, showing how to install and run these technologies to identify, expose and utilize platform features.

3.0 Node Feature Discovery

Node Feature Discovery (NFD), a project in the Kubernetes incubator, discovers and advertises the hardware capabilities of a platform that are, in turn, used to facilitate intelligent scheduling of a workload. More information can be found in the NFD GitHub: https://github.com/kubernetes-incubator/node-feature-discovery

The current deployment mechanism of NFD is as a Kubernetes job, which spins up an NFD pod on all nodes of the cluster. This NFD pod discovers hardware capabilities on the node and advertises them through Kubernetes constructs called labels. More information on label structure used in NFD along with examples can be found later in this section.

NFD can currently detect features from a set of feature sources:

1. CPUID for Intel Architecture (IA) CPU details
2. Intel P-State driver
3. Network

One of the key capabilities detected by NFD is SR-IOV through the network feature source previously mentioned.

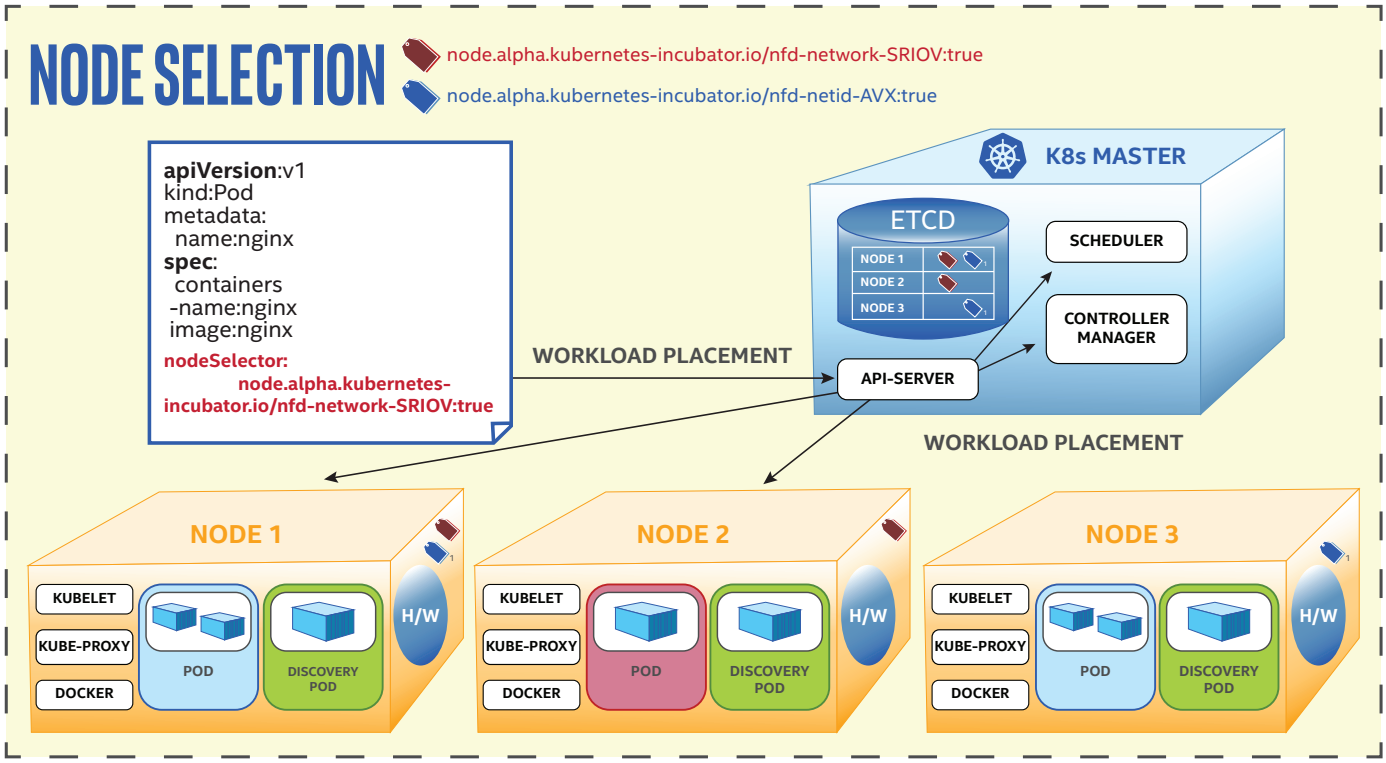


Figure 2. Node feature discovery

As mentioned, NFD uses labels for advertising node-level hardware capabilities. These labels are key/value pairs that are attached to Kubernetes objects, such as pods or nodes for specifying attributes of objects that may be relevant to the end user. They can also be used to organize objects into specific subsets. Labels are a part of the metadata information that is attached to each node's description. All this information is stored in etcd. A node label published by NFD encodes the following information:

- A **namespace**
- The **source** for each label
- The **discovered feature name** for the underlying source
- The **version** of this discovery code that wrote the label

An example of a label created by node feature discovery can be found below:

```
"node.alpha.kubernetes-incubator.io/nfd-network-sriov: true"
```

This indicates that the namespace is **node.alpha.kubernetes-incubator.io**, the source is **network** and feature name is **sriov**.

In addition to the labels corresponding to the node hardware capabilities, a label is created to provide the version of node feature discovery used, which can be found below:

```
"node.alpha.kubernetes-incubator.io/node-feature-discovery.version": "v0.1.0"
```

3.1 Installation of NFD with SR-IOV detection capability

Using NFD, the installation steps below will show how the Kubernetes scheduler can schedule pods that have a requirement for high I/O traffic to nodes where SR-IOV VFs are available.

1. The steps involved in installing NFD with SR-IOV detection start with cloning the following GitHub link:

   https://github.com/kubernetes-incubator/node-feature-discovery

2. Next, the directory should be changed to node-feature-discovery followed by running the make command to build the docker image which is subsequently used in the file node-feature-discovery-job.json.template:

   ```
   # cd <project-root>
   make
   ```

3. Obtain the name of the image built in the previous step using:

   ```
   # docker images
   ```

4. Push NFD image to the Docker Registry. In the commands below **<docker registry>** is the docker registry used in the Kubernetes cluster:

   ```
   #docker tag <image name> <docker registry>/<image name>
   #docker push <docker registry>/<image name>
   ```

5. Edit the node-feature-discovery-job.json.template to change image node (line 40) to: Label Assignment to SR-IOV capable nodes. To use the built image from the step above, run the following command:

   ```
   ...
   "image": "<docker registry>/<image name>"..
   ```

3.2 Running NFD

With the set up done, NFD can now be run. Currently, NFD is deployed as a Kubernetes job and can be deployed using the script **label-nodes.sh**, which can be obtained from the aforementioned NFD GitHub repo. This script identifies the number of nodes in the Kubernetes cluster and creates the "**node-feature discovery-job.json**" file. This file contains the Kubernetes job configuration and replaces placeholder variables from the template file with the number of discovered nodes.

1. Running this script automatically creates a "node-feature-discovery" job.

   ```
   ./label-nodes.sh
   ```

   The "node-feature-discovery" job runs pods on each node, discovers the hardware capabilities of the node it is running on and assigns the proper labels to the node.

2. For verifying the correct execution of NFD, the labels assigned to the nodes can be viewed using the following command:

   ```
   kubectl get nodes -o json | jq .items[].metadata.labels
   ```

Example output of this command for single node looks like this:

```
{
"node.alpha.kubernetes-incubator.io/node-feature-   discovery.version": "a9af7ff-dirty",
"node.alpha.kubernetes-incubator.io/nfd-pstate-turbo": "true",
"node.alpha.kubernetes-incubator.io/nfd-network-SRIOV": "true",
"node.alpha.kubernetes-incubator.io/nfd-cpuid-SSSE3": "true",
"node.alpha.kubernetes-incubator.io/nfd-cpuid-SSE4.2": "true",
"node.alpha.kubernetes-incubator.io/nfd-cpuid-SSE4.1": "true",
"node.alpha.kubernetes-incubator.io/nfd-cpuid-SSE3": "true",
"node.alpha.kubernetes-incubator.io/nfd-cpuid-ERMS": "true",
"node.alpha.kubernetes-incubator.io/nfd-cpuid-F16C": "true",
"node.alpha.kubernetes-incubator.io/nfd-cpuid-HTT": "true",
"node.alpha.kubernetes-incubator.io/nfd-cpuid-MMX": "true",
  }
```

3.2.1 Example usage

An example of how node feature discovery can be deployed can be found in this NFD job discovery template. The template is used by NFD to create the job specification that is used to create an NFD job. See more at the NFD Github: https://github.com/kubernetes-incubator/node-feature-discovery/blob/master/node-feature-discovery-job.json.template

A pod requiring a specific hardware capability can make use of the labels assigned by NFD using the node selector or the nodeAffinity/nodeAntiAffinity field. In the pod specification description below, the pod sriovPod needs to be scheduled on a node with SR-IOV capability and hence provides the label “node.alpha.kubernetes-incubator.io/nfd-network-SRIOV: true” in the nodeSelector field

```
apiVersion: v1
kind: Pod
metadata:
  name: sriovPod
  labels:
    env: test

spec:
  containers:
  - name: nginx
    image: nginx
nodeSelector:
  node.alpha.kubernetes-incubator.io/nfd-network-SRIOV: true
```

In a case where a workload is adversely affected by a specific hardware capability, the nodeAntiAffinity field can be used to ensure that the workload is not scheduled on a node where that hardware capability is available.

With NFD running, the features on the data center hardware platforms are exposed. The next section will explore the core pinning and isolation capabilities of CMK for their contribution to deterministic application performance.

4.0 CPU Core Manager for Kubernetes (CMK)

CPU Manager for Kubernetes (CMK) performs a variety of operations to enable core pinning and isolation on a container or a thread level. These include:

- Discovering the CPU topology of the machine.
- Advertising the resources available via Kubernetes constructs.
- Placing workloads according to their requests.
- Keeping track of the current CPU allocations of the pods, ensuring that an application will receive the requested resources provided they are available.

CMK uses a directory of folders to represent the cores available on the system. The folders in this directory are defined as pools. A pool, in this instance, is a named (e.g. data plane) group of CPU lists. CMK has three distinct pools; the data plane, control plane and infra. A pool may be exclusive, meaning only a single task may be allocated to a CPU at a time, or shared, where multiple processes may be allocated to a CPU.

The data plane pool within CMK is exclusive. To enforce this, CMK advertises the number of cores available as an OIR in Kubernetes. A user can then request a slot via the OIRs. A slot is an exclusive CPU in the data plane pool. The use of OIRs ensures that the Kubernetes scheduler can accurately account for the exclusive slots on a node and schedule pods to appropriate nodes. The control plane and infra pools are shared and are not tracked by OIRs. These pools may be requested via the CMK command line. CMK provides a number of command line arguments that provide different functionality. These will be explained in section 4.3.

When a process makes a request for a CPU, CMK affinitizes the requesting process to a core on the desired pool. Along with this, CMK writes the process ID (PID) to a task file on the allocated CPU. Every CPU list has an associated task file. In the case of an exclusive pool, a PID in the tasks file indicates that a core is already allocated to a workload. CMK uses garbage collection to ensure that the tasks files are free from dead processes.

For ease of use, CMK provides 'node-reports' and 'reconcile-reports' that provide a view of the CMK directory in its current state and the dead PIDs that the garbage collection has removed.

Figure 3. System state directory for CPU Core Manager for Kubernetes

```
#An example directory hierarchy after CPU Core Manager for Kubernetes init
etc
|_ _ _ cmk
    |_ _ _ lock               #system lock to prevent conflict
    |_ _ _ pools
       |_ _ _ controlplane
       |    |_ _ _ 0,1        #logical core IDS
       |    |    |_ _ _ tasks #contains linux Process IDs
       |    |_ _ _ exclusive  #Determine exclusivity (exclusive == 1)
       |_ _ _ dataplane
       |    |_ _ _ 2,3
       |    |    |_ _ _ tasks
       |    |_ _ _ 4,5
       |    |    |_ _ _ tasks
       |    |_ _ _ exclusive
       |_ _ _ infra
           |_ _ _ 6,7
           |    |_ _ _ tasks
           |_ _ _ exclusive
```

4.1 Installation

1. Installing CMK starts with cloning the following Intel GitHub link:

```
# git clone https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes
```

2. From inside the cloned repository the CMK Docker image is built:

```
# make
```

   a. **Note:** The CMK image needs to be available on each node in the Kubernetes cluster where CMK will be deployed.

3. CMK uses RBAC and service accounts for authorization. Deploy the following yaml files:

```
# kubectl create –f cmk-rbac-rules.yaml
# kubectl create –f cmk-serviceaccount.yaml
```

4. Use the isolcpus boot parameter to ensure exclusive cores in CMK are not affected by other system tasks:

```
# isolcpus=0,1,2,3
```

   a. On a hyper threaded system, fully 'isolate' a core by isolating the hyper-thread siblings.

   b. At a minimum, the number of fully isolated cores should be equal to the desired number of cores in the data plane pool.

   c. CMK will work without the isolcpus set but does not guarantee isolation from system processes being scheduled on the exclusive data plane cores.

5. The recommended way to install CMK is through the 'cluster-init' command deployed as part of a Kubernetes pod. 'Cluster-init' creates two additional pods on each node where CMK is to be deployed. The first pod executes the init, install and discover CMK commands and the second deploys a daemonset to execute and keep alive the 'nodereport' and 'reconcile' CMK commands. The function of each of these commands is explained in section 4.3.
   a. 'Cluster-init' accepts a variety of command line configurations. An example 'cluster-init' command:

```
# kubectl create -f resources/pods/cmk-       cluster-init.yaml
# /cmk/cmk.py cluster-init --all-hosts
```

   b. An example 'cluster-init' pod specification can be found at: https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes/blob/master/resources/pods/cmk-cluster-init-pod.yaml
   c. CMK can be deployed through calling the CMK commands individually if 'cluster-init' fails. Information on this can be found at: https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes/blob/master/docs/cli.md

4.2 Example usage

The 'isolate' command is used to pin a workload to a core in the requested pool. CMK uses the binary installed as part of the deployment process to act as a wrapper to the workload and runs before it. This allows CMK to pin the parent process to the allocated core and to clean up on termination.

An example 'isolate' command requesting a core on the data plane pool:

```
# /opt/bin/cmk isolate --conf-dir=/etc/cmk --pool=dataplane sleep inf
```

An example 'isolate' command for a pod requesting an exclusive core on the data plane core can be found at: https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes/blob/master/resources/pods/cmk-isolate-pod.yaml

Intel and its partners are working with the Kubernetes community to upstream the CMK features to be native in Kubernetes. A subset of the CPU pinning and isolation features of CMK have been incorporated into the Kubernetes v1.8 release, with an aim of making native Kubernetes feature compatible with CMK. More information is available on the following website: https://kubernetes.io/docs/tasks/administer-cluster/cpu-management-policies/

4.3 CMK Commands

The full list of commands can be found on the CMK GitHub repository that is listed in table A.2-1 in Appendix A.2. The following subsections provide an overview of each command.

4.3.1 Init

'init' is the first command run when installing CMK on a Kubernetes cluster. The 'init' command creates the directory hierarchy for pools and slots. At a minimum, three pools are created, the data plane, control plane and infra pools. The data plane pool is exclusive whereas the control plane and infra pools are shared.

4.3.2 Install

The 'install' sub-command builds an executable binary for CMK that will be located in the installation directory.

4.3.3 Discover

CMK 'discover' command uses OIR to advertise the number of slots on the relative Kubernetes node. The number of slots advertised is equal to the number of CPU lists available under the data plane pool. After CMK 'discover' command is run, the node will be patched with `pod.alpha.kubernetes.io/opaque-int-resource-CMK'. CMK 'discover' command also taints each node that it has been installed on. This means that no pods will be scheduled on this node unless they have the appropriate toleration to the taint. Any pod that wishes to use CMK must include the correct toleration in the pod specification. CMK 'discover' command will also add a label to the nodes to easily identify them as CMK nodes.

4.3.4 Reconcile

CMK 'reconcile' command creates a long-living daemon that acts as a garbage collector in the event that CMK fails to clean up after itself. The 'reconcile' command process runs periodically at a requested interval (10-second intervals, for example). At each interval, 'reconcile' command verifies the liveness of the process IDs attached to the tasks file. The 'reconcile' command process creates a report of any processes it has killed.

CMK 'reconcile' command creates a Kubernetes ThirdPartyResource (TPR) ReconcileReport and publishes these reports to the API server. The representation will show the tasks that the 'reconcile' command process cleaned up because the CMK did not correctly remove the programs.

4.3.5 Node-Report

CMK 'node-report' prints a JSON representation of the CMK configuration directory and its current state for all the nodes in the cluster that have CMK installed. The representation will show the pools in the directory, the CPU lists of the pools, the exclusivity of the pools, and any process IDs that are currently running on the CPUs. There is an option to publish the 'node-report' to the API server as a TPR. Node reports are generated at a timed interval that is user defined – the default time is every 60 seconds.

4.3.6 Isolate

CMK 'isolate' consumes an available CPU from a specified pool. The CMK 'isolate' sub-command allows a pool to be specified, in the case that the data plane pool is specified, the OIR created in the CMK 'discover' command will be consumed. Exactly one OIR will be consumed per container, this ensures the correct number of containers is allowed to run on a node. In the case of a shared pool, any CPU may be selected regardless of the current process allocations.

CMK 'isolate' writes its own process ID into the tasks file of the chosen core on the specified pool. This is done before executing any other commands in the container. When the process is complete, the CMK program removes the process ID from the tasks file. In the case that this fails, the 'reconcile' command program will clean up any dead process IDs in the task files.

CMK 'isolate' will fail in the case where an exclusive pool is requested and there are no available CPUs left in that pool.

4.3.7 Describe

CMK 'describe' prints a JSON representation of the CMK configuration directory on a specific node and its current state. CMK 'describe' will show the pools in the directory, the CPU lists of the pools, the exclusivity of the pools and any process IDs that are currently running on the CPUs.

4.3.8 Cluster-Init

CMK 'cluster-init' runs a set or subset of CMK sub-commands – 'init', 'install', 'discover', 'reconcile' and 'node-report'. It also prepares specified nodes in a Kubernetes cluster for CMK.

4.3.9 Uninstall

CMK 'uninstall' removes CMK from a node. The 'uninstall' process reverts the CMK 'cluster-init' command:

- deletes cmk-reconcile-nodereport-pod-{node} if present
- removes 'NodeReport' from Kubernetes ThirdPartyResources if present
- removes ReconcileReport from Kubernetes ThirdPartyResources if present
- removes cmk node label if present
- removes cmk node taint if present
- removes cmk node OIR if present
- removes --conf-dir=<dir> if present and no processes are running that use cmk 'isolate'
- removes cmk binary from --install-dir=<dir>, if binary is not present then throws an error

Note: This command requires that there are no processes running that use the CMK 'isolate' command, otherwise it will return an error and fail.

5.0 Setting up Huge Page support in Kubernetes

Up until Kubernetes v1.8, there was no mechanism to discover, request or schedule huge pages as resources, they had to be manually managed. Huge page support has been introduced as an alpha first-class resource, which enables huge page management natively in Kubernetes. The supported huge page sizes are 2 MB and1GB. The following snippet shows a pod specification making a request for huge pages:

```
resources:
  requests:
     hugpages-2Mi: 100Mi
```

There are two main modes of consuming huge pages in a workload. One is to use system calls, such as shmget(), within the workload. At a minimum, the above addition to the pod specification is sufficient to consume huge pages.

Alternatively, EmptyDir, a memory-backed volume can be used. To create an EmptyDir volume that is backed by huge pages, the following addition has to be made to the pod specification along with the resource request for huge pages shown above:

```
volumes:
- name: hugepage
  emptyDir:
    medium: HugePages
```

5.1 Configuration

Kubernetes nodes must pre-allocate huge pages in order for the node to report its huge page capacity. A node may only pre-allocate huge pages for a single size.

A special "alpha feature gate HugePages" has to be set to true across the system:

```
--feature-gates="HugePages=true".
```

5.1.1 Example usage

An example pod spec that uses the first-class resource and the EmptyDir volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: hugepagesPod
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
          hugpages-2Mi: 100Mi
  volumes:
  - name: hugepage
    emptyDir:
      medium: HugePages
```

6.0 Performance test results using CMK

The following tests demonstrate the performance of CMK and Huge Pages utilization in a Kubernetes environment.

6.1 Test setup

To create data flows, the tests used packet generation and testing applications that operate in userspace and in the kernel network stack. To test network throughput leveraging EPA functions, DPDK testpmd was used. Testpmd is an application that tests packet forwarding rates in virtual networks that leverage DPDK. In addition to these throughput tests, bandwidth and latency test using Qperf were also implemented. Qperf is a command line program that measures bandwidth and latency between two nodes.

6.1.1 DPDK Testpmd

The test setup for running DPDK testpmd as workload is shown in Figure 4. The traffic is generated by an Ixia traffic generator running RFC 2544. Up to 16 containers, each running the testpmd application, are instantiated using Kubernetes, with each container assigned one VF instance from each physical port on a 25G Ethernet NIC for a total of two VFs per container supporting bidirectional traffic across them. All results are tuned for zero percent packet loss. A separate container running the stress-ng application is used to simulate a noisy neighbor container.

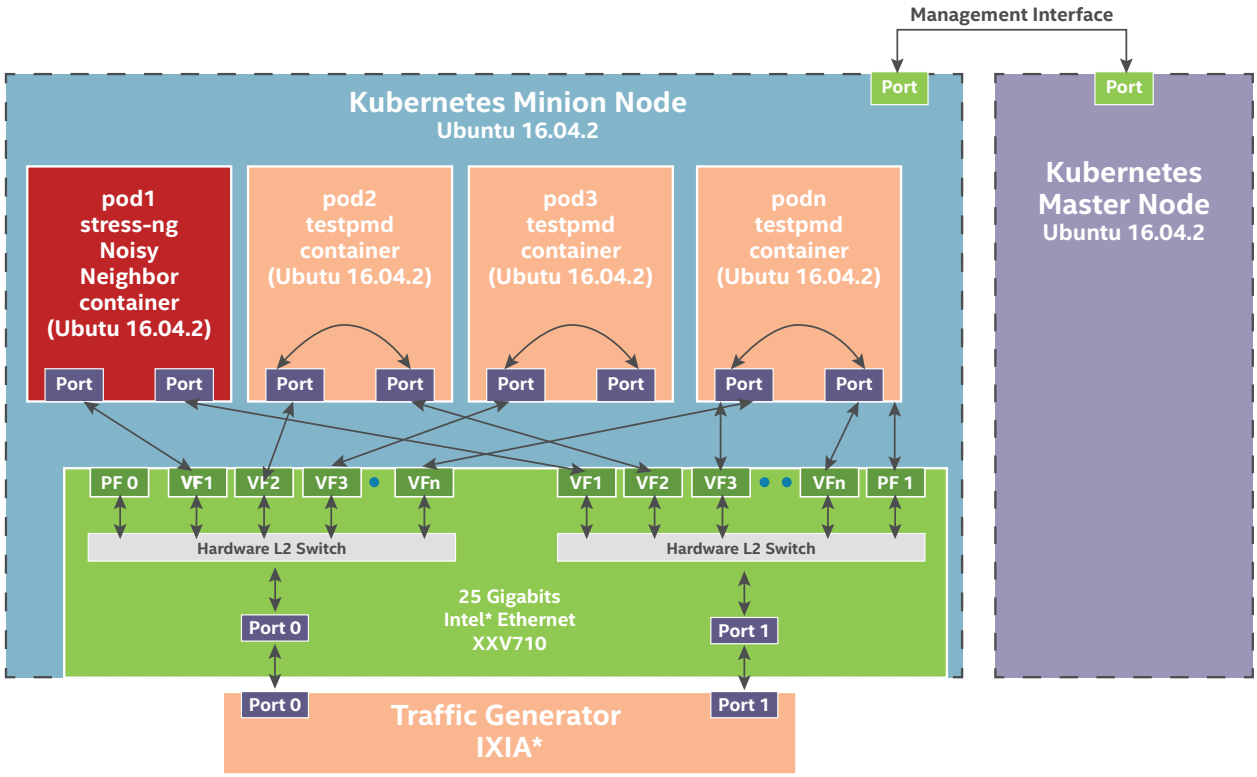


Figure 4. High-Level Overview of DPDK Testpmd Workload Setup

6.1.2 Qperf (L3 workload)

The test setup for running qperf server workload is shown in Figure 5. The qperf clients run on a separate physical server connected to SUT using a single 25GbE NIC. Up to 16 containers, each running a qperf server, are instantiated and each are connected to one qperf client. Each container is assigned one VF instance from the same 25GbE NIC port.

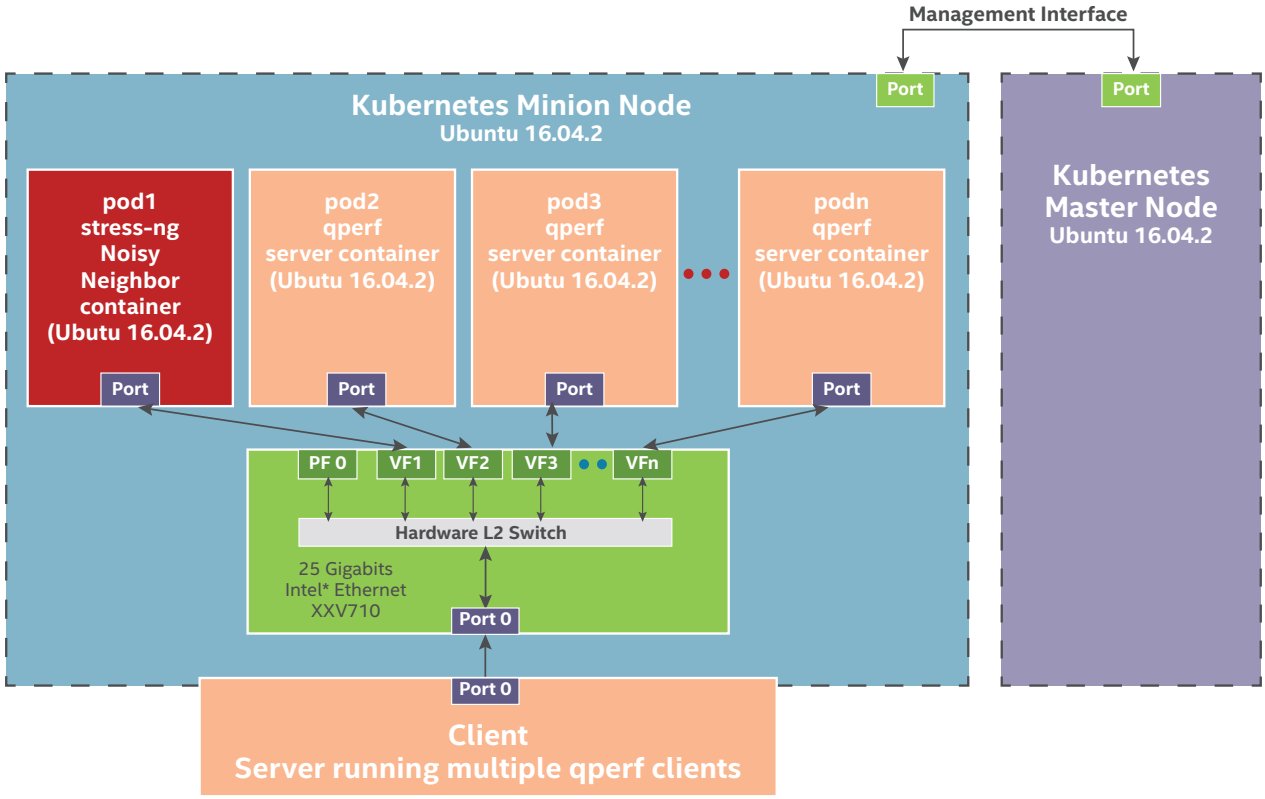


Figure 5. High-Level Overview of qperf Server Workload Setup

6.2 Test results

The test results with both SR-IOV VF kernel network stack and the DPDK VF stack showed consistent performance benefits with CMK, which eliminates the impact of noisy neighbor applications. The test results shown in Figure 6 illustrate system performance, throughput and latency, for 16 containers running the DPDK testpmd forwarding application, with and without the noisy neighbor container running in parallel, and with CMK functions turned on and off.

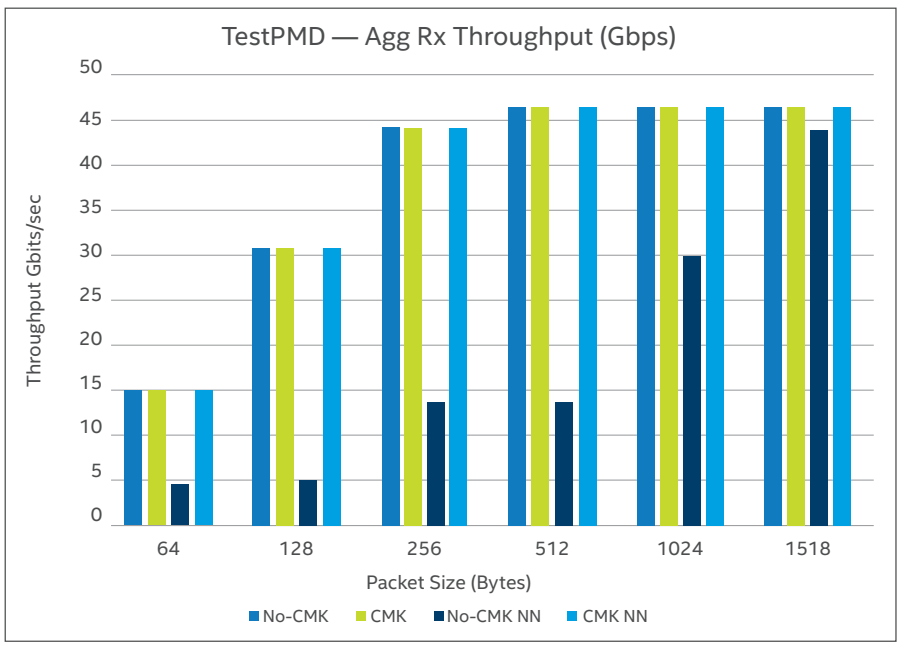


Figure 6. DPDK Testpmd throughput

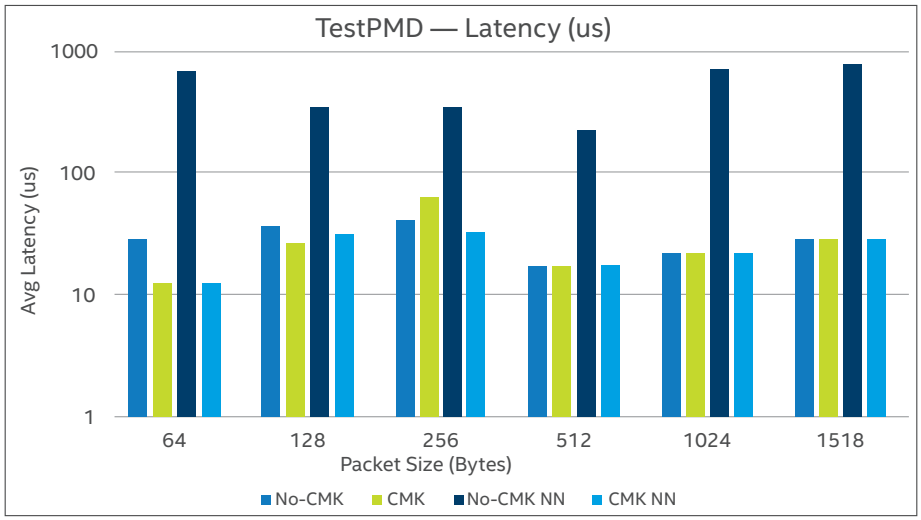


Figure 7. DPDK Testpmd latency

Testpmd containers are deployed using Kubernetes and each container is assigned a pair of dedicated cores when using CMK. CMK assigns two logical cores of same physical core from data plane pool to each container. The DPDK Poll Mode Driver (PMD) threads in each of the testpmd containers utilize the huge pages supported by CMK.

The results shown in Figures 6 and 7 demonstrate the following:

- Without CMK:
  - Performance degrades significantly in the presence of noisy neighbor.
  - Throughput decreases over 70% and latency increases by 10 times.
- With CMK:
  - Performance is not impacted by having a noisy neighbor container running in the system.
  - Deterministic performance is demonstrated due to the fact that the cores are now being isolated and dedicated to the testpmd container and not shared with other containers.

The results show applications that use the DPDK network stack and utilize CMK get improved and consistent throughput and latency performance.

6.2.1 Qperf TCP Performance with and without CMK

Test results in Figures 8 and 9 show the system performance for TCP traffic tests for 16 containers running qperf server with and without noisy neighbor container present. The qperf containers are deployed using Kubernetes and qperf application is run with and without CMK. When qperf is run using CMK, CMK isolates and assigns two hyper-threaded sibling cores to a qperf server instance inside a container from its data plane core pool. When qperf is run without CMK, it is not pinned to any specific cores and thus is free to use any available cores in the system.

Tests are run for both TCP and UDP traffic types. Each test iteration is run for a duration of five minutes. There is one TCP connection per container for a total of 16 TCP connections for 16 containers.

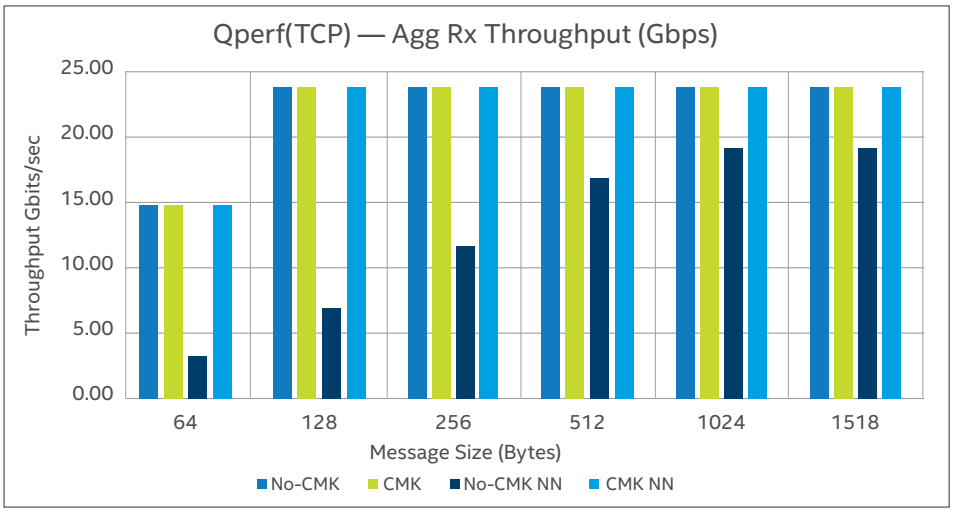


Figure 8. Qperf TCP throughput

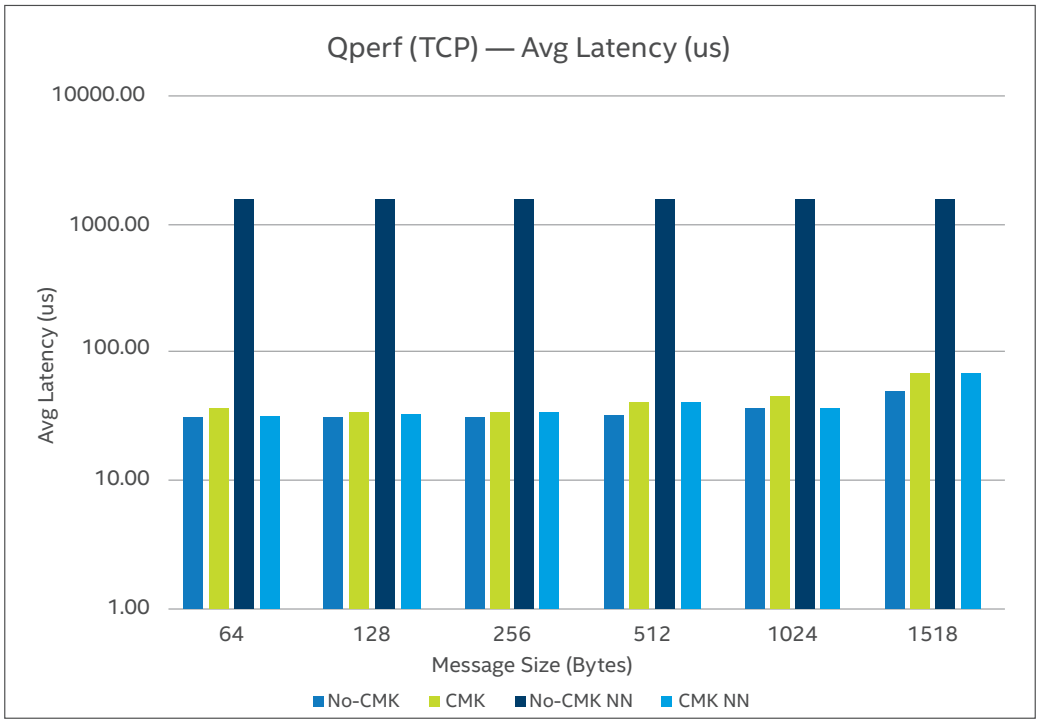


Figure 9. Qperf TCP Latency

6.2.2 Summary and findings from test results

The results of the tests are shown in Figures 8 and 9 and summarized here:

- Without CMK:
  - Performance degrades significantly with noisy neighbor without CMK.
  - Throughput is reduced by more than 70% for small packets and ~25% for packet sizes larger than 512 bytes. Furthermore, latency increases more than 70 times for most packet sizes.
- With CMK:
  - When running the qperf server using CMK, the performance is not impacted by having a noisy neighbor container running in the system.
  - The cores are now isolated and dedicated to the qperf server container and not shared with other containers, leading to a deterministic performance.

Results show the application using kernel network stack and running with CMK gets consistent performance (throughput and latency).

7.0 Summary

In summary, the platform capabilities made usable by NFD and CMK, such as core pinning, core isolation, SR-IOV for network and huge page support can apply to any type of Kubernetes deployment that requires deterministic performance. For in-depth details on performance benchmarks, refer to the performance benchmarking report titled "Kubernetes and Container Bare Metal Platform for NFV use cases for Intel® Xeon® Gold Processor 6138T" (see links in Appendix table A.3-2).

The goal of this document was to demonstrate the essential benefits of EPA in Kubernetes, including:

- Intelligent orchestration of containers through NFD.
- Predictable workload performance and service assurance through CMK.
- Negating adverse effects of a noisy neighbor in production environment through intelligent process scheduling via CMK.
- Native inclusion of essential hardware features such as huge pages in Kubernetes.

These performance benefits were showcased utilizing Intel® Xeon® Scalable servers using a sample DPDK-based user space workload and qperf workload that uses the kernel network stack, demonstrating the importance of EPA in Kubernetes.

For more information on what Intel is doing with containers, go to https://networkbuilders.intel.com/network-technologies/intel-container-experience-kits.

Appendix

The following tables show the hardware platform and software utilized in this demonstration. Below that is a helpful summary of the acronyms used in this document as well as links to reference documents.

A.1 Hardware

Table A.1-1 Intel® Xeon® Gold Processor Scalable Family platform

Item	Description	Notes
Platform	Intel® Server Board S2600WFQ	Intel® Xeon® processor-based dual-processor server board with 2 x 10 GbE integrated LAN ports
Processor	2 Intel® Xeon® Gold Processor 6138T CPUs	2.0 GHz; 125 W; 27.5 MB cache per processor 20 cores, 40 hyper-threaded cores per processor
Memory	192GB Total; Micron MTA36ASF2G72PZ	12x16GB DDR4 2133MHz, 16GB per channel, 6 Channels per socket
NIC	Intel® Ethernet Network Adapter XXV710-DA2 (2x25G)	2 x 1/10/25 GbE ports, Firmware version 5.50
Storage	Intel DC P3700 SSDPE2MD800G4	SSDPE2MD800G4 800 GB SSD 2.5in NVMe/PCIe
BIOS	Intel Corporation SE5C620.86B0X.01.0007.060920171037 Release Date: 06/09/2017	P-state, HT Technology, VT-x & VT-d enabled. Turbo Boost disabled

A.2 Software

The software versions used for the benchmarking tests outlined in this document are listed in Table A.2-1 below. A table of compatible software components for setting up a Kubernetes cluster can be obtained from the user guide called "Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors." The URL for this document can be found in table A.3-2.

Table A.2-1 Software ingredients used in performance tests

Software Component	Description	References
Host Operating System	Ubuntu 16.04.2 x86_64 (Server) Kernel: v4.4.0-301.fc21.x86_64	https://www.ubuntu.com/download/server
NIC Kernel Drivers	i40e v2.0.30 i40evf v2.0.30	https://sourceforge.net/projects/e1000/files/i40e%20stable
DPDK	DPDK 17.05	http://fast.dpdk.org/rel/dpdk-17.05.tar.xz
CMK	V1.1.0	https://github.com/Intel-Corp/CPU-Manager-for-Kubernetes
Ansible*	Ansible 2.3.1	https://github.com/ansible/ansible/releases
Bare Metal Container RA scripts	Includes Ansible scripts to deploy Kubernetes v1.7.0	https://github.com/intel-onp/onp
Docker	v1.13.1	https://docs.docker.com/engine/installation/
SR-IOV-CNI	v0.2-alpha. commit ID: a2b6a7e03d8da456f3848a96c6832e6aefc968a6	https://www.ubuntu.com/download/server

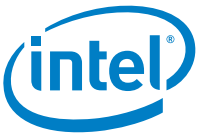
A.3 Terminology and reference documents

Table A.3-1 Terminology

Term	Description
CMK	CPU Manager for Kubernetes
COE	Container Orchestration Engine
CPUID	CPU Identification
DPDK	Data Plane Development Kit
etcd	Distributed key value store that serves as a reliable way to store data across a cluster of machines
HCI	Hyper-Converged Infrastructure
NFV	Network Functions Virtualization
PCH	Platform Controller Hub
PF	Physical Function
PMD	DPDK Poll Mode Driver
p-state	CPU performance state
SDI	Software Defined Infrastructure
SHVS	Standard High-Volume Server
SKU	Stock Keeping Unit
SLA	Service Level Agreement
SSL	Secure Sockets Layer
TLB	Translation Lookaside Buffer
TLS	Transport Layer Security
VF	Virtual Function
VIM	Virtual Infrastructure Manager
VNF	Virtual Network Function

Table A.3-2 EPA for Kubernetes Experience Kit - Reference Documents

Document	Document No./Location
Deploying Kubernetes and Container Bare Metal Platform for NFV Use Cases with Intel® Xeon® Scalable Processors	https://networkbuilders.intel.com/network-technologies/container-experience-kits
NFV Reference Design for A Containerized vEPC application	https://builders.intel.com/docs/networkbuilders/nfv-reference-design-for-a-containerized-vepc-application.pdf
Understanding CNI (Container Networking Interface)	http://www.dasblinkenlichten.com/understanding-cni-container-networking-interface/



 Enhanced Platform Awareness in Kubernetes Application Note 04/18/HM/DJA/PDF002 Please Recycle 336865-002US