TECHNOLOGY GUIDE

Intel Corporation

Multi-Cloud Services on Kubernetes with Cloudify Orchestration and F5 Networks Functions

Authors

Yury Kylulin

Petar Torre

Intel Corporation

Shay Naeh

Cloudify

Philip Klatte

F5 Networks

1 Introduction

Communications Service Providers and other vertical customers with strict compute requirements adopting Cloud Native principles, need orchestration for managing Multi-Cloud and Edge sites that may range from very small to large size. This technology guide describes a solution based on Cloudify* policy-driven orchestration for Kubernetes*-managed containerized network-functions from F5 Networks* using Intel® QuickAssist Technology (Intel® QAT).

The solution was developed as a public showcase demonstrating scalability, with robust automation. It is loosely coupled and fully modular, respecting the boundaries of orchestration, applications, software platform, and hardware platform layers. These attributes ease the application on-boarding and lifecycle management efforts, while allowing performance-optimized deployments. Figure 1 shows a high-level view of the solution, which is described in detail in later sections of this document.

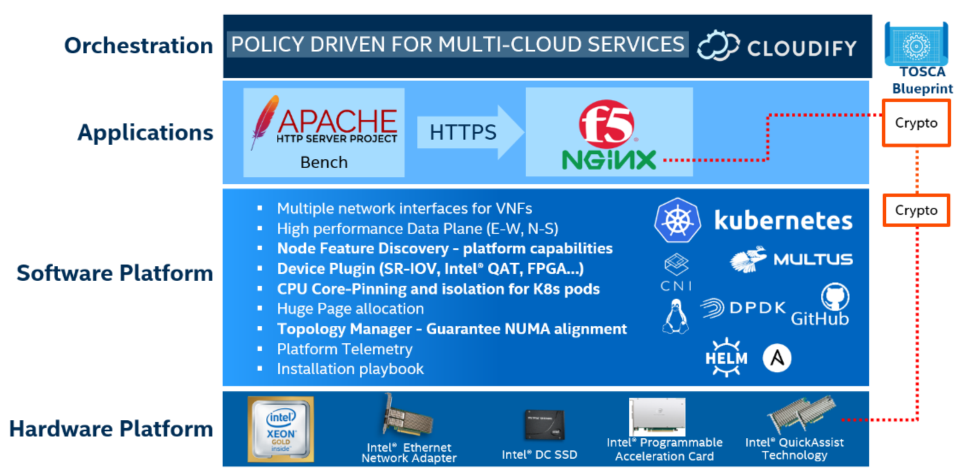

Figure 1. Solution Layered Stack with Orchestration, Applications, Software Platform, and Hardware Platform

This technology guide is intended for architects and engineers in Communication Service Providers and other verticals with strict compute requirements who are interested in best practices for designing fully automated environments based on Kubernetes-managed containers.

This document is part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.

Table of Contents

Figures

Tables

Document Revision History

REVISION	DATE	DESCRIPTION
001	November 2019	Initial release.
002	February 2021	Enabled CPU allocation and pinning with both Kubernetes CPU manager and CMK CPU Manager for Kubernetes (CMK). For QAT resource allocation and life cycle management switched to QAT device plugin in kernel mode.
003	March 2021	Minor clarification on node selector and device resources.

1.1 Motivation

Not all Kubernetes nodes are created equally — Workloads such as Virtualized Router (vRouter), Virtualized Firewall (vFW), Virtualized Deep Packet Inspection (vDPI) in Network Functions Virtualization (NFV), or low-latency trading in Finance require fast processing of network traffic and special consideration for placement on physical servers where they can benefit from appropriate hardware acceleration. Workload placement based on platform capabilities is required on certain Kubernetes nodes equipped with distinctive hardware acceleration capabilities.

Edge environments — Even the biggest cloud environments consist of smaller data centers, some of which can run on small network edge or on-premises locations. The motivation for such design is usually a mix of bandwidth, latency, and privacy requirements. From a workload placement perspective, it is essential to orchestrate which workloads are placed on which edges.

1.2 Terminology

Table 1. Terminology

ABBREVIATION	DESCRIPTION
AWS*	Amazon Web Services*
BMRA	Bare Metal Reference Architecture
CSP	Communication Service Provider
DPDK	Data Plane Development Kit
ENA	Elastic Network Adapter
EPA	Enhanced Platform Awareness
K8s*	Kubernetes
NFD	Node Feature Discovery
NFV	Network Functions Virtualization
NUMA	Non-Uniform Memory Access
QAT	Intel QuickAssist Technology (Intel® QAT)
SR-IOV	Single Root Input/Output Virtualization
TOSCA	Topology and Orchestration Specification for Cloud Applications
vDPI	Virtualized Deep Packet Inspection
vFW	Virtualized Firewall
VNF	Virtualized Network Functions
VPC	Virtual Private Cloud
vRouter	Virtualized Router

1.3 Reference Documentation

Table 2. Reference Documents

REFERENCE	SOURCE
Cloudify website	https://cloudify.co/
F5 Networks website	https://www.f5.com/
NGINX* website	https://nginx.org/
Intel® Network Builders website for Containers Experience Kits	https://networkbuilders.intel.com/network-technologies/container-experience-kits
Container Bare Metal for 2nd Generation Intel® Xeon® Scalable Processor Reference Architecture (installation guide)	https://builders.intel.com/docs/networkbuilders/container-bare-metal-for-2nd-generation-intel-xeon-scalable-processor.pdf
Node Feature Discovery Application Note	https://builders.intel.com/docs/networkbuilders/node-feature-discovery-application-note.pdf
Intel Device Plugins for Kubernetes Application Note	https://builders.intel.com/docs/networkbuilders/intel-device-plugins-for-kubernetes-appnote.pdf
Topology Management – Implementation in Kubernetes Technology Guide	https://builders.intel.com/docs/networkbuilders/topology-management-implementation-in-kubernetes-technology-guide.pdf

REFERENCE	SOURCE
CPU Management – CPU Pinning and Isolation in Kubernetes Technology Guide	https://builders.intel.com/docs/networkbuilders/cpu-pin-and-isolation-in-kubernetes-app-note.pdf
Enhanced Platform Awareness in Kubernetes Application Note	https://builders.intel.com/docs/networkbuilders/enhanced-platform-awareness-in-kubernetes-application-note.pdf

2 How Cloudify Orchestration Works

Cloudify, as a global orchestrator, provisions workloads to run on distributed Kubernetes clusters based on a set of requirements and available resources that match those requirements. Figure 2 shows the Cloudify console view.

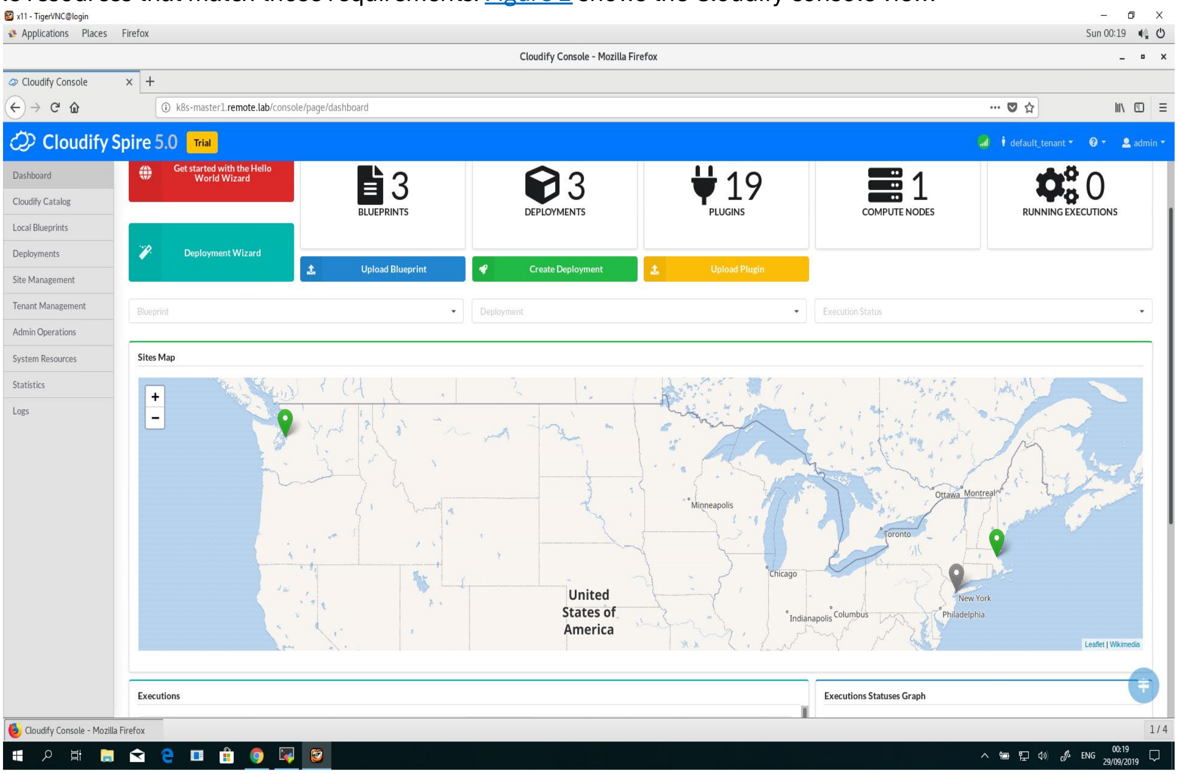

Figure 2. Cloudify Console View

Figure 3 describes a multi-cloud network setting, orchestrated by Cloudify. The diagram shows four Kubernetes-managed locations across multi-clouds. Each Kubernetes cluster supports a set of platform capabilities addressing different performance and operation needs. Based on criteria such as location, resource availability, and special resource requirements, Cloudify provisions a workload to the correct Kubernetes cluster. Yet this is only part of the work- each Kubernetes cluster is composed from multiple nodes, each having different hardware capabilities. Cloudify works with Intel-led Kubernetes enhancements supporting multiple capabilities like Data Plane Development Kit (DPDK), Single Root Input/Output Virtualization (SR-IOV), Intel QuickAssist Technology (Intel QAT) or other hardware accelerators, Non-Uniform Memory Access (NUMA), and CPU Pinning. That way Cloudify can map workloads to the right Kubernetes nodes by utilizing **node labels** per Kubernetes node and **Node Selectors** to match Kubernetes pods to specific nodes, or by requesting acceleration **device resources**, while all intelligence about NUMA topology, CPU pinning, or assignment of specific hardware devices is done within the Kubernetes software platform.

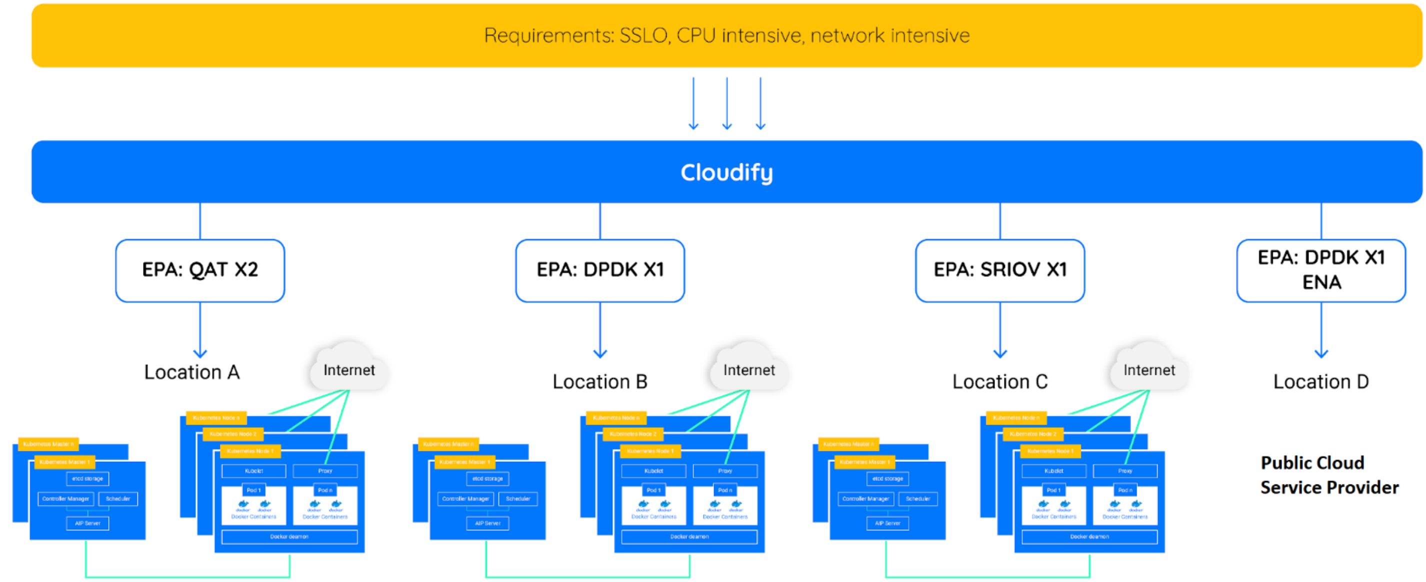

Figure 3. Requirements for Central Cloudify Orchestrated to Distributed Sites

2.1 How to Map a Kubernetes Cluster

```
kind: Pod
apiVersion: v1
metadata:
  generateName: demo-
spec:
  containers:
  - name: demonginx
…
    resources:
      requests:
        memory: "1Gi"
        cpu: "2"
        qat.intel.com/cy1_dc0: '1'
      limits:
        memory: "1Gi"
        cpu: "2"
        qat.intel.com/cy1_dc0: '1'
…
```

Figure 4. NGINX Pod Placement with Requesting QuickAssist Resource

In the case of the demonstration discussed in this paper, we provisioned[1] (1) NodeJS pod on a generic Kubernetes node and (2) NGINX pod on a Kubernetes node identified per NFD with 'load balancing' capability, which supports CPU encryption acceleration. All the Kubernetes nodes supporting the 'load balancing' capability are grouped under a special group named QAT. In Figure 5, the QAT group is marked with a light blue background. This allocation is done on an on-premises Kubernetes cluster.

[1] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

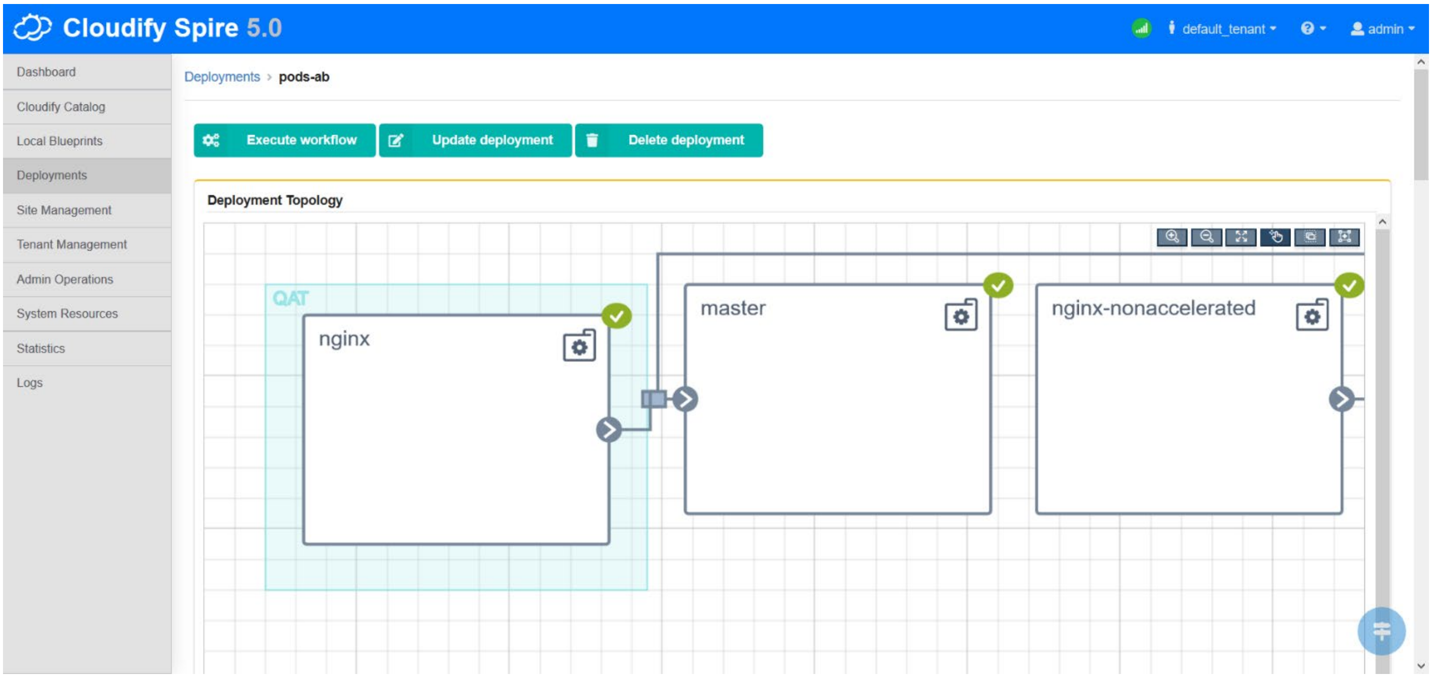

Figure 5. Cloudify Console with Composer View

2.2 Non-Kubernetes and Hybrid Environments

As previously mentioned, Cloudify can provision workloads on both Kubernetes and non-Kubernetes hybrid environments. As shown in Figure 6, workloads can be provisioned to Amazon Web Services* (AWS*). A virtual private cloud (VPC) environment is instantiated on AWS and a VM is created in that VPC. This VM could be a VNF with special requirements for fast/intensive network traffic processing. AWS's ENA (Elastic Network Adapter) supports the Data Plane Development Kit (DPDK), therefore it would be required to install the DPDK driver or choose the right AWS AMI for that. By matching the workload requirements (in this case the VNF requirements), Cloudify places the VNF on the right node in AWS, fulfilling intensive network capabilities.

A mixture of Kubernetes and non-Kubernetes environments can be maintained by the orchestrator. Moreover, these environments can be located on-premises or on any public cloud.

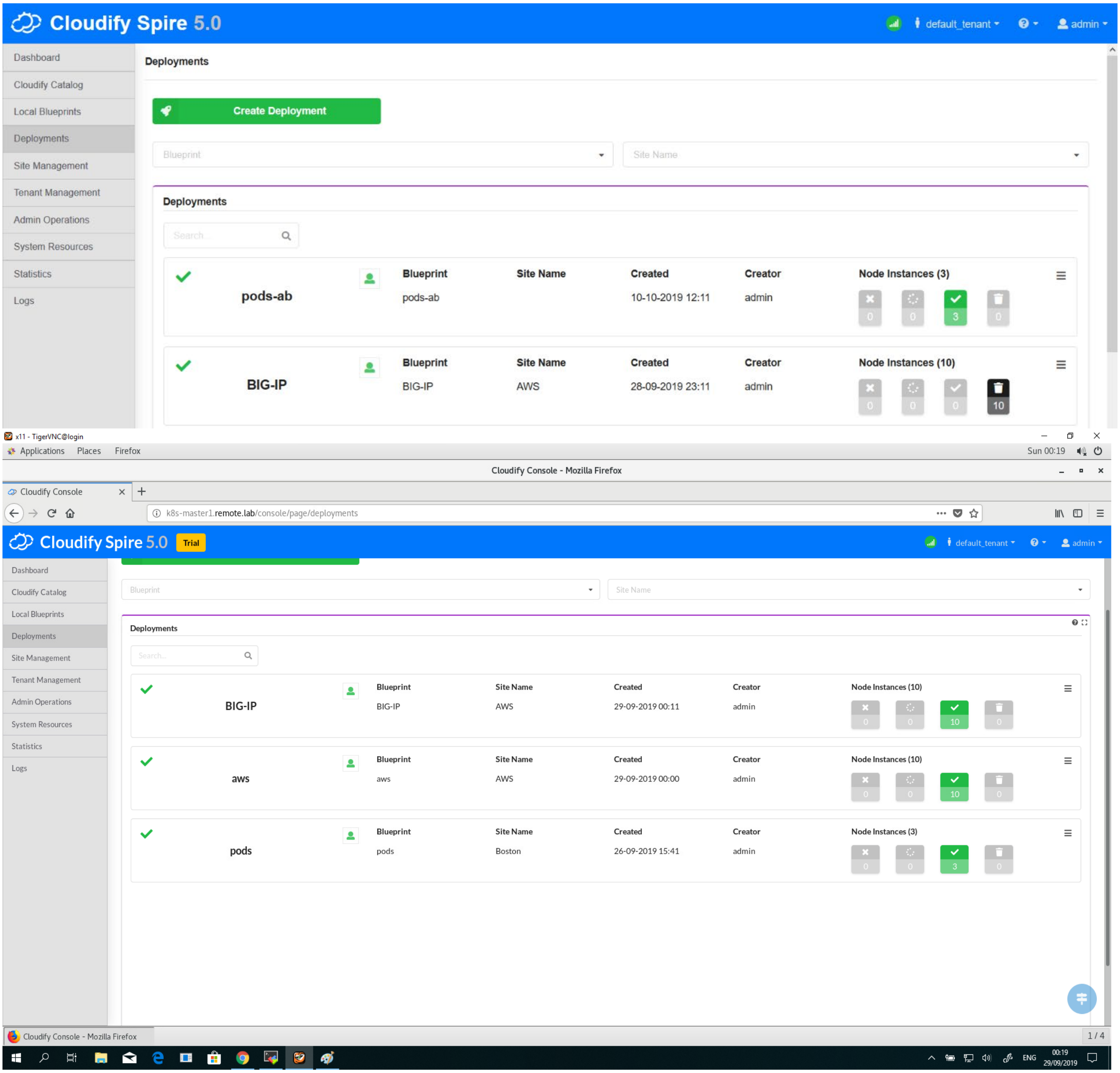

Figure 6. Cloudify Console with Deployments View

2.3 Intent-Based Decoupling of Layers

With the term “intent”, we mean that we specify the ‘what’ and not the ‘how’. The need for ‘CPU intensive’ hardware may differ based on the environment because each environment may hold different definitions and parameters. If we decouple this from the tenant user, it makes the process of application placement to the platform simple and transparent. The user specifies the requirements they need and Cloudify will match those requirements with the right compute nodes per network definitions.

Utilizing Topology and Orchestration Specification for Cloud Applications (TOSCA), we can write an intent-based blueprint that decouples application need from a Kubernetes cluster implementation. In this scenario, the tenant only needs to specify the requirement for nodes with certain capabilities and Cloudify will match the right resources and provision the workloads correctly.

Intent-based definitions decouple the workload requirements from the underlying environment without changing anything at the higher level of the workload definition. Even when changing the environment where the workload runs and moving the workload to a new environment, Cloudify will look for the right resources and definitions on the new environment and will select them based on workload requirements.

TOSCA also helps in the ‘matching’ process. TOSCA defines ‘Requirements’ and ‘Capabilities’ primitives, where a user specifies in the ‘Requirements’ primitive what it needs, e.g., CPU intensive or Network intensive and ‘Capabilities’. TOSCA also holds a list of supported capabilities by a compute node. In Kubernetes 'Requirements' are normally defined by node selectors and 'Capabilities' by node labels, or by acceleration device resources. Hence, TOSCA definitions cover the more generic use case and are not restricted to Kubernetes environments, pods, and nodes.

To summarize, TOSCA requirements and capabilities provide the mechanism to define a generic case for workload requirements and map them to nodes that support the capabilities to fulfill those requirements.

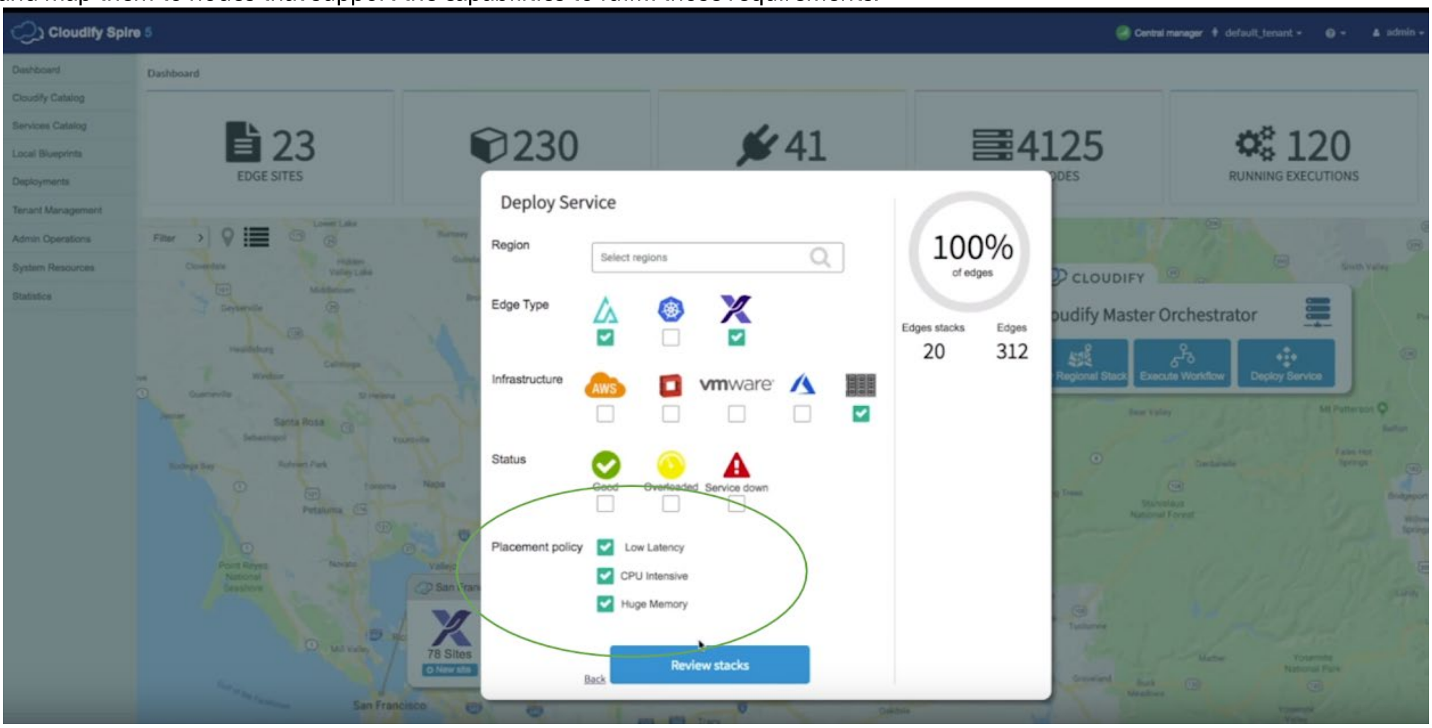

Figure 7. Intent-Based Placement

3 F5 Networks and NGINX

NGINX is a free, open-source, high-performance HTTP server and reverse proxy, as well as an IMAP/POP3 proxy server. NGINX is known for its high performance, stability, rich feature set, simple configuration, and low resource consumption. To handle requests, NGINX uses scalable event-driven (asynchronous) architecture with small and predictable amounts of memory under load, which makes it very scalable. A Netcraft study found NGINX to be the #1 web server with 34% market share (source: https://news.netcraft.com/archives/2020/09/23/september-2020-web-server-survey.html). F5 Networks acquired NGINX in March 2019, which, with other F5 offerings, enable multi-cloud application services across all environments.

4 Software and Hardware Platform

4.1 Physical Topology

The physical topology[2] for the testing uses a Kubernetes cluster based on one control node and two worker nodes. One of the nodes, 'k8s node2', has integrated QAT inside the server chipset. On a separate host, Ansible* runs in the 'Ansible VM,' enabling Kubernetes cluster installation using Bare Metal Reference Architecture (BMRA) v2.0.

[2] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

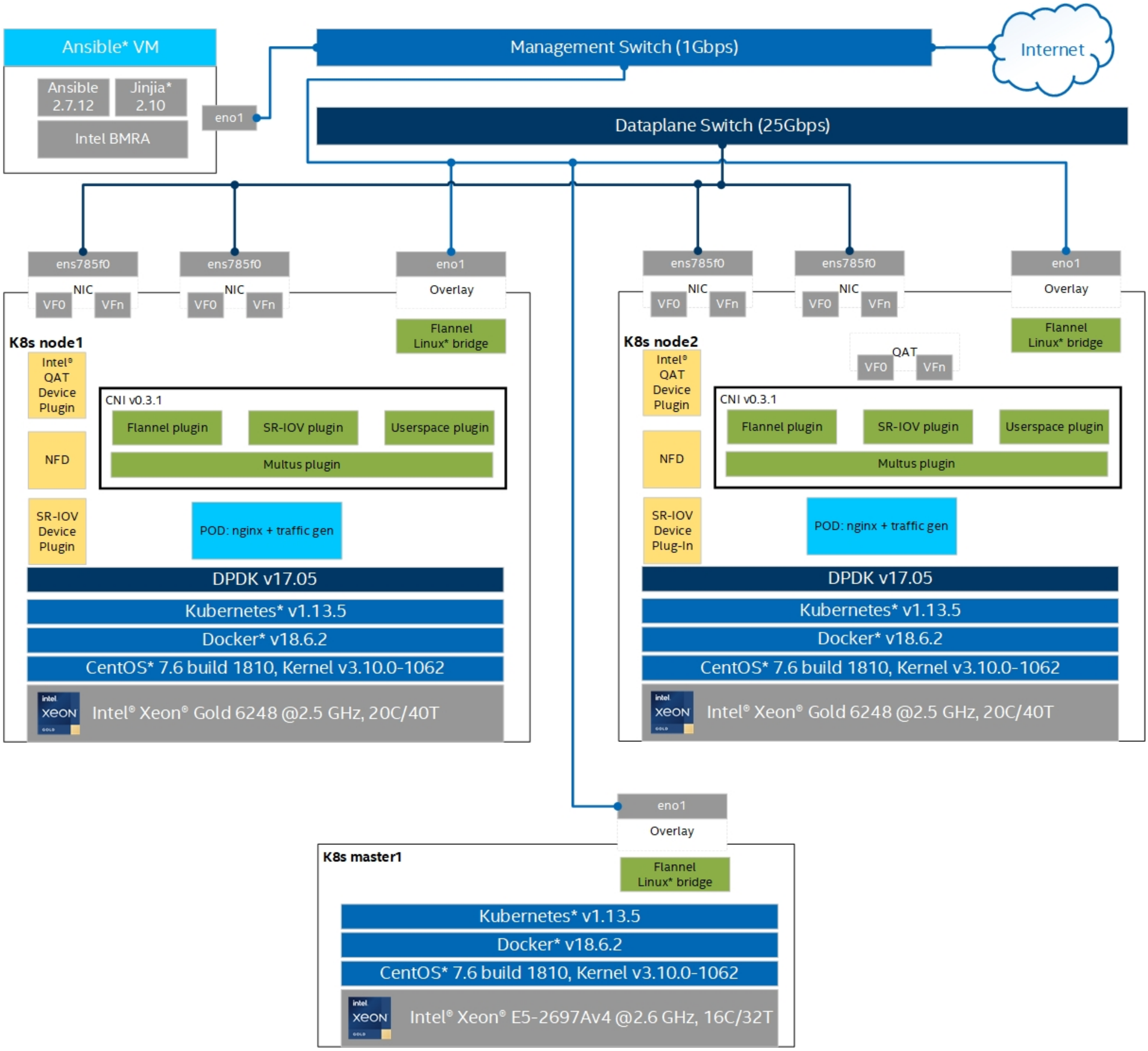

Figure 8. Physical Topology[3]

4.2 Software Topology

For the Kubernetes cluster and plugins setup, we used the Container Bare Metal Reference Architecture Ansible Playbook (available as part of [Container Bare Metal for 2nd Generation Intel® Xeon® Scalable Processor](#)). In addition, the Cloudify Manager was installed to work with the cluster through RESTful APIs.

In this setup, we used the following Kubernetes open source software capabilities to demonstrate the role of intelligent workload placement depending on the requirements: Node Feature Discovery, SR-IOV Network Device Plugin, and Intel Device Plugins for Kubernetes (Intel QuickAssist Device Plugin).

4.3 Hardware Specifications

This section lists the hardware components and systems that were utilized in this test setup. 2nd Generation Intel® Xeon® Scalable processors feature a scalable, open architecture designed for the convergence of key workloads such as applications and services, control plane processing, high-performance packet processing, and signal processing.

[3] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

Table 3. Hardware Specifications[4]

ITEM	DESCRIPTION	NOTES
Platform	Intel® Xeon® Processor Scalable Family	Intel® Xeon® processor-based dual-processor server board 2 x 25 GbE LAN ports
Processors	4x Intel® Xeon® Gold 6248 Processor	20 cores, 40 threads, 2.5 GHz, 150 W, 27.5 MB L3 total cache per processor, 3 UPI Links, DDR4-2933, 6 memory channels
	2x Intel® Xeon® E5-2697v4 Processor	16 cores, 32 threads, 2.6 GHz, 145 W, 40 MB L3 total cache per processor, 2 QPI Links, DDR4-2400, 4 memory channels
Memory	192GB (12 x 16GB 2666MHz DDR RDIMM) or minimum all 6 memory channels populated (1 DPC) to achieve 384 GB	192GB to 384GB
Networking	2 x NICs - Required Each NIC NUMA aligned	2 x Dual Port 25GbE Intel® Ethernet Network Adapter XXV710 SFP28+
		2 x Intel® Ethernet Server Adapter X520-DA2 SFP
		2 x Dual Port 10GbE Intel® Ethernet Converged Network Adapter X722
Local Storage	Intel SSD DC S3500	
Intel® QuickAssist Technology	Intel® C620 Series Chipset Integrated on baseboard Intel® C627/C628 Chipset	Integrated w/NUMA connectivity to each CPU or minimum 16 Peripheral Component Interconnect express* (PCIe*) lane Connectivity to one CPU
BIOS	Intel Corporation SE5C620.86 B.02.01.0008 Release Date: 11/19/2018	Intel® Hyper-Threading Technology (Intel® HT Technology) enabled Intel® Virtualization Technology (Intel® VT-x) enabled Intel® Virtualization Technology for Directed I/O (Intel® VT-d) enabled
Switches	Huawei* S5700-52X-LI-AC Huawei* CE8860-4C-EI with CE88-D24S2CQ module	Management 1 GbE Switch Dataplane 25 GbE Switch

4.4 Software Specifications

Table 4. Software Specifications

SOFTWARE FUNCTION	SOFTWARE COMPONENT	LOCATION
Host OS	CentOS* 7.8 build 2003 Kernel version: 3.10.0-1127.19.1.el7.x86_64	https://www.centos.org/
Ansible	Ansible v2.7.1	https://www.ansible.com/
BMRA 2.0 Ansible Playbook	Master Playbook v1.0	https://github.com/intel/container-experience-kits
Python*	Python 2.7	https://www.python.org/

[4] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

SOFTWARE FUNCTION	SOFTWARE COMPONENT	LOCATION
Kubespray*	Kubespray: v2.8.0-31-g3c44ffc	https://github.com/kubernetes-sigs/kubespray
Docker*	Docker* 18.06.1-ce, build e68fc7a	https://www.docker.com/
Container orchestration engine	Kubernetes v1.13.0	https://github.com/kubernetes/kubernetes
CPU Manager for Kubernetes	CPU Manager for Kubernetes v1.3.0	https://github.com/intel/CPU-Manager-for-Kubernetes
Node Feature Discovery	NFD v0.3.0	https://github.com/kubernetes-sigs/node-feature-discovery
Data Plane Development Kit	DPDK 17.05.0	http://dpdk.org/git/dpdk
Open vSwitch with DPDK	OVS-DPDK 2.11.90	http://docs.openvswitch.org/en/latest/intro/install/dpdk/
Vector Packet Processing	VPP 19.01	https://docs.fd.io/vpp/19.01/index.html
Multus CNI	Multus CNI v4.0	https://github.com/intel/multus-cni
SR-IOV CNI	SR-IOV CNI v1.0	https://github.com/intel/SR-IOV-network-device-plugin
Userspace CNI	Userspace CNI v1.0	https://github.com/intel/userspace-cni-network-plugin
Intel Ethernet Drivers		https://sourceforge.net/projects/e1000/files/ixgbe%20stable/5.2.1 https://sourceforge.net/projects/e1000/files/ixgbevf%20stable/4.2.1 https://sourceforge.net/projects/e1000/files/i40e%20stable/2.0.30 https://sourceforge.net/projects/e1000/files/i40evf%20stable/2.0.30

4.5 Platform BIOS Settings[5]

Table 5. Platform BIOS Settings

MENU (ADVANCED)	PATH TO BIOS SETTING	BIOS SETTING	SETTINGS FOR DETERMINISTIC PERFORMANCE	SETTINGS FOR MAX PERFORMANCE WITH TURBO MODE ENABLED	REQUIRED OR RECOMMENDED
Power Configuration	CPU P State Control	EIST PSD Function	HW_ALL	SW_ALL	*Recommended*
Boot Performance Mode	Max. Performance	Max. Performance	*Required*		
Energy Efficient Turbo	Disable	Disable	*Recommended*		
Turbo Mode	Disable	Enable	*Recommended*		
Intel® SpeedStep® (Pstates) Technology	Disable	Enable	*Recommended*		

[5] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

MENU (ADVANCED)	PATH TO BIOS SETTING	BIOS SETTING	SETTINGS FOR DETERMINISTIC PERFORMANCE	SETTINGS FOR MAX PERFORMANCE WITH TURBO MODE ENABLED	REQUIRED OR RECOMMENDED
Hardware PM State Control	Hardware P-States	Disable	Disable	*Recommended*	
CPU C State Control	Autonomous Core C-State	Disable	Enable	*Recommended*	
CPU C6 Report	Disable	Disable	*Recommended*		
Enhanced Halt State (C1E)	Disable	Enable	*Recommended*		
Energy Perf Bias	Power Performance Tuning	BIOS Controls EPB	BIOS Controls EPB	*Recommended*	
ENERGY_PERF_BIAS_CFG Mode	Perf	Perf	*Recommended*		
Package C State Control	Package C State	C0/C1 State	C6	*Recommended*	
Intel® Ultra Path Interconnect (Intel® UPI) Configuration	Intel® UPI General Configuration	LINK L0P ENABLE	Disable	Disable	*Recommended*
LINK L1 ENABLE	Disable	Disable	*Recommended*		
SNC	Disable	Disable	*Recommended*		
Memory Configuration	Enforce POR	Disable	Disable	*Recommended*	
IMC Interleaving	2-Way Interleave	2-Way Interleave	*Recommended*		
Volatile Memory Mode	2 LM mode	2 LM mode	*Required*		
Force 1-Ch Way in FM	Disabled	Disabled	*Required*		
Platform Configuration	Miscellaneous Configuration	Serial Debug Message Level	Minimum	Minimum	*Recommended*
PCI Express* Configuration	PCIe* ASPM Support	Per Port	Per Port	*Recommended*	
Uncore	Uncore Frequency Scaling	Disable	Disable	*Required*	

Note: To gather performance data[6] required for conformance, use either column with deterministic performance or turbo mode enabled in this table. Some solutions may not provide the BIOS options that are documented in this table. For Intel® Select Solution, the BIOS should be set to the "Max Performance" profile with Virtualization.

[6] See backup for workloads and configurations or visit www.Intel.com/PerformanceIndex. Results may vary.

4.6 Building Software Platform

4.6.1 Ansible Host, Control, and Worker Node Software Prerequisites

1. As root enter the following commands in Ansible Host:

```
# yum install -y epel-release
# wget https://releases.ansible.com/ansible/rpm/release/epel-7-x86_64/ansible-2.7.12-
1.el7.ans.noarch.rpm
# yum install -y ./ansible-2.7.12-1.el7.ans.noarch.rpm
# easy_install pip
# pip2 install jinja2 --upgrade
# yum install -y python36 python2-jmespath
```

2. Enable password-less login between all nodes in the cluster.

   Step 1: Create authentication SSH-keygen keys on Ansible Host:

```
# ssh-keygen
```

   Step 2: Upload generated public keys to all the nodes from Ansible Host:

```
# ssh-copy-id root@node-ip-address
```

4.6.2 Deploy Intel Bare Metal Reference Architecture Using Ansible Playbook

1. Get Ansible playbook:

```
# git clone https://github.com/intel/container-experience-kits.git
# cd container-experience-kits/playbooks
```

2. Copy example inventory file to the playbook home location:

```
# cp examples/inventory.ini .
```

3. Edit the inventory.ini to reflect the requirement. Here is the sample file.

```
[all]
control1 ansible_host=192.168.0.235 ip=192.168.0.235 ansible_user=root
node1 ansible_host=192.168.0.236 ip=192.168.0.236 ansible_user=root
node2 ansible_host=192.168.0.237 ip=192.168.0.237 ansible_user=root

[kube-control]
control1

[etcd]
control1

[kube-node]
node1
node2

[k8s-cluster:children]
kube-control
kube-node

[calico-rr]
```

4. Copy group_vars and host_vars directories to the playbook home location:

```
# cp -r examples/group_vars examples/host_vars .
```

5. Update group_vars to match the desired configuration.

```
# vim group_vars/all.yml

---
## BMRA control playbook variables ##

# Node Feature Discovery
nfd_enabled: true
nfd_build_image_locally: true
nfd_namespace: kube-system
nfd_sleep_interval: 30s

# Intel CPU Manager for Kubernetes
cmk_enabled: true
cmk_namespace: kube-system
cmk_use_all_hosts: false # 'true' will deploy CMK on the control nodes too
#cmk_hosts_list: node1,node2 # allows to control where CMK nodes will run, leave this option
commented out to deploy on all K8s nodes
cmk_shared_num_cores: 12 # number of CPU cores to be assigned to the "shared" pool on each of
the nodes
cmk_exclusive_num_cores: 20 # number of CPU cores to be assigned to the "exclusive" pool on
each of the nodes
```

```
cmk_shared_mode: spread # choose between: packed, spread, default: packed
cmk_exclusive_mode: spread # choose between: packed, spread, default: packed

# Intel SRIOV Network Device Plugin
sriov_net_dp_enabled: true
sriov_net_dp_namespace: kube-system
# whether to build and store image locally or use one from public external registry
sriov_net_dp_build_image_locally: true

# Intel Device Plugins for Kubernetes
qat_dp_enabled: true
qat_dp_namespace: kube-system
gpu_dp_enabled: false
gpu_dp_namespace: kube-system

# Forces installation of the Multus CNI from the official Github repo on top of the Kubespray
built-in one
force_external_multus_installation: true

## Proxy configuration ##
proxy_env:
  http_proxy: ""
  https_proxy: ""
  no_proxy: ""

## Kubespray variables ##

# default network plugins and kube-proxy configuration
kube_network_plugin_multus: true
multus_version: v3.2
```

4.7 Conclusion

The Container Bare Metal for 2nd Generation Intel® Xeon® Scalable Processor Reference Architecture document provides guidelines for setting a Kubernetes performant platform for dataplane and other performance-sensitive workloads, independent of vendor implementations. Based on the platform set, various tests can be done even before such platforms are productized. Companies that plan to develop their own Kubernetes-based platform can refer to this document for additional details.

5 Preparing NGINX and Apache Bench Containerized Images

This section shows how to prepare and use containerized Docker images in the following steps:

- Prepare Linux environment.
- On the host OS, install prerequisites for getting NGINX to use Intel QuickAssist, then build containerized image using a custom Dockerfile.
- Build containerized image for Apache Bench from custom Dockerfile.
- Start procedure with Pushgateway, Prometheus*, and Grafana* using prebuilt images.
- Create Kubernetes pod yaml files.
- Create Cloudify deployments.
- List of related documents.

5.1 Linux Environment

This procedure assumes running CentOS* 7.8 with development packages preinstalled and booted with the following kernel parameters: `isolcpus=4-19,44-59,24-39,64-79 rcu_nocbs=4-19,44-59,24-39,64-79 nohz_full=4-19,44-59,24-39,64-79 intel_iommu=on pci=realloc pci=assign-busses default_hugepagesz=2M hugepagesz=2M hugepages=4096`. The CPU list has to be adjusted according to the CPU model used. `iommu=pt` must not be used.

To run QAT services from within an unprivileged Docker container, system's maximum locked memory size must exceed 64 KB. One way to change this setting is to modify the `/etc/system/system/docker.service` file and add `LimitMEMLOCK=infinity` and restart `docker.service`.

```
systemctl daemon-reload
systemctl restart docker.service
```

If you run Ubuntu* or another distribution, use equivalent commands like with `apt-get` or other package manager.

Do the compilations and integration on node with installed QuickAssist Technology, as root, and have working directory here described as WORK_DIR, like `/root/w`.

In `bash` use the following environment variables, or at the end of `/etc/bashrc` or `~/.bashrc` add:

```
export OPENSSL_INSTALL_DIR=/usr/local/ssl
export OPENSSL_ENGINES=$OPENSSL_INSTALL_DIR/lib/engines-1.1
if [ $LD_LIBRARY_PATH ]; then
   export LD_LIBRARY_PATH="$LD_LIBRARY_PATH":$OPENSSL_INSTALL_DIR/lib:/usr/local/lib64
else
   export LD_LIBRARY_PATH=$OPENSSL_INSTALL_DIR/lib:/usr/local/lib64
fi
export NGINX_INSTALL_DIR=/usr/local/nginx
export WORK_DIR=/root/w
export PUSHGWIP=<YOUR_IPV4_ADDR_WHERE_RUN_GRAFANA_PROMETHEUS_PUSHGATEWAY>
```

Restart bash if using `.bashrc` or similar files.

For start do:

```
mkdir $WORK_DIR
yum update
```

5.2 Intel® QuickAssist Driver and Libraries

Check if your node already has Intel QuickAssist Technology (Intel QAT) enabled and configured with Virtual Functions with:

```
lsmod | grep -e qat -e usdm
lspci | grep QuickAssist
```

If there, note the version of Intel QuickAssist found, for example C62x.

Additional libraries that are required are for qat and usdm, which are used later to build QAT_Engine. If that is already available, then continue with the installation step for OpenSSL*.

Compile and install Intel QuickAssist Driver and libraries with:

```
cd $WORK_DIR
mkdir QATdriver && cd QATdriver
wget https://01.org/sites/default/files/downloads//qat1.7.l.4.11.0-00001.tar.gz
tar xfz qat1.7.l.4.11.0-00001.tar.gz
yum install -y libudev-devel
./configure --enable-icp-sriov=host
make install && make samples-install
```

These commands create files `/etc/c6xx_dev[01].conf` and `/etc/c6xxvf_dev*.conf` configuration files for QuickAssist.

To test the installation, use:

```
lspci | grep QuickAssist
lsmod | grep -e qat -e usdm | sort
```

Last line gives `authenc`, `intel_qat`, `qat_c62x`, `qat_c62xvf`, `uio`, and `usdm_drv`.

5.3 OpenSSL

Compile and install OpenSSL with:

```
cd $WORK_DIR
git clone https://github.com/openssl/openssl.git && cd openssl
git checkout OpenSSL_1_1_1g
./config --prefix=$OPENSSL_INSTALL_DIR -Wl,-rpath,$OPENSSL_INSTALL_DIR/lib
make && make install
```

5.4 QAT_Engine

Compile and install Intel QuickAssist Engine (which will be used later with NGINX) with:

```
cd $WORK_DIR
git clone https://github.com/01org/QAT_Engine.git && cd QAT_Engine
git checkout v0.6.1
./autogen.sh
./configure --with-qat_dir=$WORK_DIR/QATdriver --with-openssl_dir=$WORK_DIR/openssl --with-
openssl_install_dir=$OPENSSL_INSTALL_DIR --enable-upstream-driver --enable-usdm --disable-
qat_lenstra_protection
export PERL5LIB=$PERL5LIB:$WORK_DIR/openssl
make && make install
```

In the `qat/config/c6xx/multi_process_optimized/c6xx_dev0.conf` file, update the value of `NumProcesses` to 2:

```
sed -i "s/NumProcesses = 16/NumProcesses = 2/"
qat/config/c6xx/multi_process_optimized/c6xx_dev0.conf
```

In the same configuration file, modify the [SHIM] section name to reflect information about the QAT device NUMA node. The new name should be in the format [SHIM_NUMAx], where x is the NUMA node id. This section name is exported as an environment variable name in the NGINX testing pod by the QAT device plugin and is used as the hint to the CMK to allocate cores from the same NUMA node. In our setup, the QAT device is attached to the NUMA node 0, so the section name should be modified to [SHIM_NUMA0].

```
sed -i "s/\[SHIM\]/\[SHIM_NUMA0\]/" qat/config/c6xx/multi_process_optimized/c6xx_dev0.conf
```

Then copy that as configuration for QAT Virtual Functions and restart services:

```
for i in {0..31}; do
   cp -f qat/config/c6xx/multi_process_optimized/c6xx_dev0.conf /etc/c6xxvf_dev$i.conf
done
service qat_service shutdown
service qat_service start
service qat_service_vfs start
```

To check the engine was installed:

```
ls $OPENSSL_INSTALL_DIR/lib/engines-1.1
```

The output of which should list `qat.so`.

Then checking:

```
$OPENSSL_INSTALL_DIR/bin/openssl engine -t qat
```

Returns the output `Reference Implementation of QAT crypto engine.` and `[available].`

5.5 QATZip

Compile and install QATZip (which can be used later with NGINX) with:

```
cd $WORK_DIR
git clone https://github.com/intel/QATzip.git && cd QATzip
git checkout v1.0.1
yum install -y zlib-devel
./configure --with-ICP_ROOT=$WORK_DIR/QATdriver
make clean
make all
make install
```

5.6 NGINX with Async Mode Using Intel QuickAssist

Compile and install NGINX with Async Mode using QuickAssist with:

```
cd $WORK_DIR
git clone https://github.com/intel/asynch_mode_nginx.git && cd asynch_mode_nginx
git checkout v0.4.3
yum install -y pcre-devel
export QZ_ROOT="${WORK_DIR}/QATzip"
./configure --prefix=$NGINX_INSTALL_DIR --without-http_rewrite_module --with-http_ssl_module --
add-dynamic-module=modules/nginx_qatzip_module --add-dynamic-module=modules/nginx_qat_module/ -
-with-cc-opt="-DNGX_SECURE_MEM -I$OPENSSL_INSTALL_DIR/include -I$QZ_ROOT/include -Wno-
error=deprecated-declarations" --with-ld-opt="-Wl,-rpath=$OPENSSL_INSTALL_DIR/lib -
L$OPENSSL_INSTALL_DIR/lib -L$QZ_ROOT/src -lqatzip -lz"
make && make install
```

5.7 Configure NGINX

Copy the QAT NGINX configuration file:

```
cp $WORK_DIR/asynch_mode_nginx/conf/nginx.QAT-sample.conf
$NGINX_INSTALL_DIR/conf/nginx_QAT.conf
```

Create key and certificate with the following command (replace country, state, city, company name, org name, FQDN accordingly):

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
   -keyout $NGINX_INSTALL_DIR/conf/cert.key \
   -out $NGINX_INSTALL_DIR/conf/cert.pem \
   -subj "/C=US/ST=California/L=Santa Clara/O=Intel Corporation/CN=www.intel.com"
```

In `$NGINX_INSTALL_DIR/conf/nginx.conf` enable HTTPS server lines near the end of the file:

```
# HTTPS server
#
server {
          listen       443 ssl;
          server_name  localhost;
          ssl_protocols       TLSv1.2;
          ssl_certificate     cert.pem;
```

```
          ssl_certificate_key  cert.key;
          ssl_session_cache   shared:SSL:1m;
          ssl_session_timeout  5m;
          ssl_ciphers  HIGH:!aNULL:!MD5;
          ssl_prefer_server_ciphers  on;
          location / {
          root   html;
          index  index.html index.htm;
          }
}
```

Clean NGINX configuration files from worker_processes option. It is set according to the number of allocated cores for the container:

```
sed -i "/worker_processes/d" $NGINX_INSTALL_DIR/conf/nginx_QAT.conf
sed -i "/worker_processes/d" $NGINX_INSTALL_DIR/conf/nginx.conf
```

Confirm whether the syntax of the NGINX configuration files is OK with:

```
$NGINX_INSTALL_DIR/sbin/nginx -c $NGINX_INSTALL_DIR/conf/nginx_QAT.conf -t
$NGINX_INSTALL_DIR/sbin/nginx -t
```

Create binary file (192 KB) with random values with:

```
dd if=/dev/urandom of=$NGINX_INSTALL_DIR/html/test.bin bs=1k count=192
```

5.8 Build NGINX Container Image

CPU pinning and isolation are desired for many cases and workload types including but not limited to latency sensitive CommSP workloads. To solve this problem, a CPU manager can be used. Looking at native Kubernetes CPU Manager and Intel CPU Manager for Kubernetes (CMK), there are different pros and cons for each CPU manager type.

At the time of writing, one of the limitations of the native Kubernetes CPU Manager is that CPU resources are measured in CPU units. One CPU is equivalent to one hyper-thread on bare metal deployments. For example, there could be a situation that two CPUs requested for the container can be two hyper-threads from the same physical core or from different physical cores but with some other workloads running on the second hyper-thread. Also, native Kubernetes CPU Manager does not consider the isolcpus kernel parameter, which is highly desired for the latency sensitive applications like those based on Data Plane Development Kit (DPDK). This could be solved by disabling hyper-threading in the server BIOS, but this is usually not recommended.

CPU Manager for Kubernetes allocates CPU resources as fully “isolated” cores by isolating all hyper-thread siblings. It works with isolcpus parameters for the best isolation from the system processes but can also work without them.

One more thing to consider for latency-sensitive and high-throughput workloads is device locality and the right placement on typical high-volume dual-socket systems where resources such as different kinds of accelerators can be connected to different sockets. To achieve this, the Topology Manager was introduced in Kubernetes. It works using a Hint Providers interface to send and receive topology information from different components. Currently the only supported components are Device Manager and native Kubernetes CPU Manager.

In CMK, NUMA alignment should be done “manually” and one of the ways to automate the process is shown in the start script for the demonginx container images. At the beginning of the script, the hint from QAT Device Plugin is used to identify the NUMA of the assigned device. The hint comes in the form of the QAT section name exported as an environment variable in a predefined format containing the NUMAx string. See Section 5.4 for more details. Then the NUMA ID is used as an input option for the CMK isolate phase (--socket-id) to request cores from the same NUMA node with allocated QAT device.

The following NGINX and Apache Bench images (demonginx, demoab) are universal from the CPU manager perspective. They can be used with CPU Manager for Kubernetes (CMK) or native Kubernetes CPU Manager. During the container start, the CPU manager type is identified automatically and appropriate settings are used to start NGINX.

Enter the following commands on the node that can access the Docker image repository over `localhost:5000/imagename`:

```
cd $WORK_DIR
mkdir scripts && cd scripts
mkdir demonginx && cd demonginx

cat > build << EOF
#!/bin/bash

tar cfz demo.tar.gz \
   \$NGINX_INSTALL_DIR \
   \$OPENSSL_INSTALL_DIR \
   /usr/lib64/libqatzip.so \
   /usr/local/lib64/libqatzip.so \
   /usr/local/lib64/libqatzip.so.1 \
```

```
    /usr/local/lib64/libqatzip.so.1.0.1 \
    /usr/local/lib/libqat_s.so \
    /usr/local/lib/libusdm_drv_s.so \
    /usr/local/bin/adf_ctl

docker build -t demonginx .
EOF

cat > start << EOF
#!/bin/bash

LOG_FILE="/app/log"
# NGINX
NGINX="\$NGINX_INSTALL_DIR/sbin/nginx"
NGINX_USER="nobody"
NGINX_PARAM_WP="auto"
NGINX_PARAM_WCA="auto"
NGINX_CONF="\$NGINX_INSTALL_DIR/conf/nginx.conf"
NGINX_CONF_QAT="\$NGINX_INSTALL_DIR/conf/nginx_QAT.conf"
# QAT
QAT_DEV_PROCESSES="/dev/qat_dev_processes"
QAT_ADF_CTL="/dev/qat_adf_ctl"
QAT_FOUND="/app/qat_found"
QAT_GROUP_NAME="qat"
# CMK
CMK_BIN="/opt/bin/cmk"
CMK_CONF_DIR="/etc/cmk"
CMK_POOL_DEFAULT="exclusive"
# CPU manager
SYSFS_CPUSET="/sys/fs/cgroup/cpuset/cpuset.cpus"

log() { [ ! -z "\$1" ] && echo "\$1" >> \$LOG_FILE; }
sig_handler() { log "Signal handler..."; touch /app/s; }

log_forced_options()
{
    [ ! -z "\$USE_QAT" ] && log "QAT mode is forced to \$USE_QAT"
    [ ! -z "\$WORKER_PROCESSES" ] && log "Number of NGINX worker processes is forced to
\$WORKER_PROCESSES"
    [ ! -z "\$WORKER_CPU_AFFINITY" ] && log "NGINX worker cpu affinity is forced to
\$WORKER_CPU_AFFINITY"
    [ ! -z "\$USE_CMK" ] && log "CMK usage is forced to \$USE_CMK"
    [ ! -z "\$CMK_POOL" ] && log "CMK pool is forced to \$CMK_POOL"
    [ ! -z "\$CMK_SOCKET_ID" ] && log "CMK socket id is forced to \$CMK_SOCKET_ID"
}

qat_detect()
{
    [ "\$USE_QAT" == "false" ] && return
    log "Looking around for QAT..."
    if [ ! -e "\$QAT_DEV_PROCESSES" ] || [ ! -e "\$QAT_ADF_CTL" ]; then log "No QAT detected.";
return; fi
    log "Found QAT files in /dev..."
    touch \$QAT_FOUND
    QAT_GID=\`stat -c "%g" \$QAT_DEV_PROCESSES\`
    groupadd -g \$QAT_GID \$QAT_GROUP_NAME
    usermod -aG \$QAT_GROUP_NAME \$NGINX_USER
    log "Created \$QAT_GROUP_NAME group with gid \$QAT_GID and added \$NGINX_USER in this group"
    [ ! -z "\$CMK_SOCKET_ID" ] && return
    if [[ \$QAT_SECTION_NAME =~ NUMA[0-9]+$ ]]; then
            CMK_SOCKET_ID=\${QAT_SECTION_NAME##*NUMA}
            log "NUMA pattern is found in the QAT_SECTION_NAME (\$QAT_SECTION_NAME). Request
cores from NUMA\$CMK_SOCKET_ID."
    fi
}

cmk_detect()
{
    [ "\$USE_CMK" == "false" ] && return
```

```
    log "Looking around for CMK..."
    if [ -e "\$CMK_BIN" ]; then
            CMK_VERSION=\`\$CMK_BIN --version\`
            if [ \$? -eq 0 ]; then log "Found CMK (\$CMK_VERSION)..."; CMK_FOUND="true"; else log
"Failed to query the CMK version. Please check the CMK installation/configuration."; fi
    else
            log "No CMK binary found. Seems CMK is not installed."
    fi
}

cmk_nginx_start_stage1()
{
    [ -z "\$CMK_POOL" ] && CMK_POOL=\$CMK_POOL_DEFAULT
    if [ -z "\$CMK_SOCKET_ID" ]; then
            log "Requesting cores from \$CMK_POOL pool and default socket id."
            \$CMK_BIN isolate --conf-dir=\$CMK_CONF_DIR --pool=\$CMK_POOL /app/start --
cmk_nginx_start_stage2
    else
            log "Requesting cores from \$CMK_POOL pool and socket \$CMK_SOCKET_ID."
            \$CMK_BIN isolate --conf-dir=\$CMK_CONF_DIR --pool=\$CMK_POOL --socket-
id=\$CMK_SOCKET_ID /app/start -- cmk_nginx_start_stage2
    fi
}

cmk_nginx_start_stage2()
{
    [ -z \$CMK_CPUS_ASSIGNED ] && log "List of assigned CPUs is not found (CMK_CPUS_ASSIGNED)!"
&& return
    if [ \`lscpu | grep "Thread(s) per core" | awk '{print \$4}'\` -eq 2 ]; then
            log "HT is on"
            HTS=\`cat /sys/devices/system/cpu/cpu*/topology/thread_siblings_list | sort | uniq |
awk -F',' '{ print \$2 }'\`
            CMK_CPUS_ASSIGNED_NOHTS=\`echo \$CMK_CPUS_ASSIGNED | awk -v v1="\$HTS"
'{split(\$0,cpus,","); split(v1,hts," "); for (i in hts) {for (j in cpus) {if (hts[i]==cpus[j])
{delete cpus[j]; break}}}} END {for (i in cpus) {printf "%s%s",s,cpus[i]; s=","}}'\`
    fi
    NGINX_PARAM_WP=\`echo \$CMK_CPUS_ASSIGNED_NOHTS | awk -F',' '{print NF}'\`
    CPU_AFFINITY=\`echo \$CMK_CPUS_ASSIGNED_NOHTS | awk '{split(\$0,a,","); asort(a);
l=a[length(a)]; for (i=0;i<=l;i++) m[i]=0; for (i in a) m[a[i]]=1 } END {for (i=l;i>=0;i--)
printf m[i]}'\`
    log "CPUs assigned by CMK: \$CMK_CPUS_ASSIGNED. Using CPUs: \$CMK_CPUS_ASSIGNED_NOHTS.
Number of workers: \$NGINX_PARAM_WP. Workers CPU affinity string: \$CPU_AFFINITY"
    NGINX_PARAM_WCA="auto \$CPU_AFFINITY"
    nginx_start
    wait_for_stop
}

cpu_manager_nginx_start()
{
    CPU_MANAGER_CPUS_ASSIGNED=\`cat \$SYSFS_CPUSET\`
    NGINX_PARAM_WP=\`echo \$CPU_MANAGER_CPUS_ASSIGNED | awk '{split(\$0,a,","); for (i in a)
{split(a[i],b,"-"); c++; if (b[2]!="") for (j=b[1]+1;j<=b[2];j++) c++}} END {print c}'\`
    log "CPUs assigned by CPU manager: \$CPU_MANAGER_CPUS_ASSIGNED. Number of workers:
\$NGINX_PARAM_WP. No need for workers CPU affinity string."
    NGINX_PARAM_WCA="auto"
    nginx_start
    wait_for_stop
}

nginx_start()
{
    if [ -z "\$WORKER_PROCESSES" ]; then NGINX_PARAMS="worker_processes \$NGINX_PARAM_WP;"; else
NGINX_PARAMS="worker_processes \$WORKER_PROCESSES;"; fi
    if [ -z "\$WORKER_CPU_AFFINITY" ]; then NGINX_PARAMS="\$NGINX_PARAMS worker_cpu_affinity
\$NGINX_PARAM_WCA;"; else NGINX_PARAMS="\$NGINX_PARAMS worker_cpu_affinity
\$WORKER_CPU_AFFINITY;"; fi
    NGINX_PARAMS="\$NGINX_PARAMS user \$NGINX_USER;"
    log "Using the following NGINX parameters: \"\$NGINX_PARAMS\""
    if [ -e "\$QAT_FOUND" ]; then
```

```
          log "Starting NGINX with QAT..."
          \$NGINX -g "\$NGINX_PARAMS" -c \$NGINX_CONF_QAT
          sleep 5
          if [ "\$(is_nginx_started)" == "false" ]; then
                 log "NGINX failed to start... Let's try without QAT..."
                 \$NGINX -s stop && sleep 1
                 \$NGINX -g "\$NGINX_PARAMS" -c \$NGINX_CONF
          fi
   else
          log "Starting NGINX without QAT..."
          \$NGINX -g "\$NGINX_PARAMS" -c \$NGINX_CONF
   fi
   sleep 5
   if [ "\$(is_nginx_started)" == "true" ]; then log "NGINX successfully started!"; else log
"NGINX failed to start!"; fi
}

is_nginx_started() { if [ \`ps -ef | grep nginx | grep -c "worker process"\` -gt 0 ]; then echo
"true"; else echo "false"; fi; }

start()
{
   log_forced_options
   cmk_detect
   qat_detect
   if [ "\$CMK_FOUND" == "true" ]; then cmk_nginx_start_stage1; else cpu_manager_nginx_start;
fi
}

wait_for_stop()
{
   log "Waiting for stop command..."
   trap sig_handler SIGINT SIGKILL SIGTERM
   while [ ! -f /app/s ]; do sleep 1; done
   log "Stopping NGINX..."
   \$NGINX -s stop
}

\$1
EOF

cat > stop << EOF
#!/bin/bash
touch /app/s
EOF

chmod 755 build start stop
cat > Dockerfile << EOF
FROM centos:centos7
COPY start stop /app/
RUN yum install -y pciutils
ADD demo.tar.gz /
ENV OPENSSL_INSTALL_DIR=$OPENSSL_INSTALL_DIR
ENV NGINX_INSTALL_DIR=$NGINX_INSTALL_DIR
#ENV ICP_ROOT /usr/local/qat
ENV LD_LIBRARY_PATH /usr/lib:/usr/lib64:/usr/local/lib:/usr/local/lib64
CMD ["/bin/bash", "/app/start", "start"]
EXPOSE 80 443
EOF
```

Build the Docker image with the following commands:

```
./build
docker tag demonginx:latest localhost:5000/demonginx:latest
docker push localhost:5000/demonginx:latest
```

5.9 Build Apache Benchmark Load Generator Container Image

Enter the following commands:

```
cd $WORK_DIR/scripts
mkdir demoab && cd demoab
```

```
cat > start << EOF
#!/bin/bash

LOG_FILE="/app/log"
# CMK
CMK_BIN="/opt/bin/cmk"
CMK_CONF_DIR="/etc/cmk"
CMK_POOL_DEFAULT="exclusive"

log() { [ ! -z "\$1" ] && echo "\$1" >> \$LOG_FILE; }
sig_handler() { log "Signal handler..."; touch /app/s; }

cmk_detect()
{
   [ "\$USE_CMK" == "false" ] && return
   log "Looking around for CMK..."
   if [ -e "\$CMK_BIN" ]; then
          CMK_VERSION=\`\$CMK_BIN --version\`
          if [ \$? -eq 0 ]; then log "Found CMK (\$CMK_VERSION)..."; CMK_FOUND="true"; else log
"Failed to query the CMK version. Please check the CMK installation/configuration."; fi
   else
          log "No CMK binary found. Seems CMK is not installed."
   fi
}

ab_start()
{
   trap sig_handler SIGINT SIGKILL SIGTERM
   if [ -z \$ABID ]; then ID=\`date +%N\`; else ID=\$ABID; fi
   log "AB instance ID ab\$ID"
   while [ ! -f /app/s ]; do
          tr=\$( ab -n \$ABN -c \$ABC https://\$ABTARGETURL | awk '\$1=="Transfer" &&
\$2=="rate:" {print \$3}')
          log "TransferRate \$tr"
          echo TransferRate \$tr | curl --data-binary @-
http://\$PUSHGWIP:9091/metrics/job/ab/instance/ab\$ID
   done
}

start()
{
   cmk_detect
   if [ "\$CMK_FOUND" == "true" ]; then
          [ -z "\$CMK_POOL" ] && CMK_POOL=\$CMK_POOL_DEFAULT
          \$CMK_BIN isolate --conf-dir=\$CMK_CONF_DIR --pool=\$CMK_POOL /app/start -- ab_start
   else
          ab_start
   fi
}

\$1
EOF

cat > stop << EOF
#!/bin/bash
touch /app/s
EOF

cat > Dockerfile << EOF
FROM centos:centos7
RUN yum install -y httpd-tools curl
COPY start stop /app/
CMD ["/bin/bash", "/app/start", "start"]
EOF

cat > build << EOF
#!/bin/bash
docker build -t demoab .
EOF
```

Build the Docker image with:

```
./build
docker tag demoab:latest localhost:5000/demoab:latest
docker push localhost:5000/demoab:latest
```

5.10 Configure Pushgateway, Prometheus, and Grafana Under Docker

On your (control) node where the IP address is `PUSHGWIP`, perform the following one-time setup procedure:

```
cd $WORK_DIR
mkdir scripts && cd scripts
cat > run_once << EOF
#!/bin/bash
docker run --name grafana -d --network host -e "GF_SECURITY_ADMIN_PASSWORD=password"
grafana/grafana
EOF
cat > start_all << EOF
#!/bin/bash
docker run --name pushgateway -d --network host prom/pushgateway
docker run --name prometheus -d --network host -v
$WORK_DIR/scripts/prometheus.yml:/etc/prometheus/prometheus.yml prom/prometheus
docker start grafana
EOF
cat > stop_all << EOF
#!/bin/bash
docker stop grafana
docker kill prometheus pushgateway
docker rm prometheus pushgateway
EOF
cat > prometheus.yml << EOF
global:
  scrape_interval:     2s
  evaluation_interval: 15s
scrape_configs:
  - job_name: 'pushgateway'
    static_configs:
    - targets: ['$PUSHGWIP:9091']
EOF
chmod 755 run_once start_all stop_all
./run_once
```

After the first-time setup is completed, you only need to use the command:

```
./start_all
```

Verify if Grafana, Prometheus, and Pushgateway are running with the command:

```
docker ps | grep -e grafana -e prometheus -e pushgateway
```

Later, when restarting pods with ab containers, old metrics can be cleaned with the commend:

```
./stop_all && ./start_all
```

5.11 Define Kubernetes Pod

To load each NGINX, we use three instances of ab, as four containers coming together in one pod.

For the built-in CPU manager, use the following template:

```
cat demo.yaml << EOF
kind: Pod
apiVersion: v1
metadata:
  generateName: demo-
spec:
  containers:
  - name: demonginx
    image: localhost:5000/demonginx:latest
    imagePullPolicy: IfNotPresent
    command: [ "/app/start", "start" ]
    env:
    securityContext:
      capabilities:
         add: ["IPC_LOCK"]
    resources:
      requests:
```

```
        memory: "1Gi"
        cpu: "2"
        qat.intel.com/cy1_dc0: '1'
      limits:
        memory: "1Gi"
        cpu: "2"
        qat.intel.com/cy1_dc0: '1'
  - name: demoab1
    image: localhost:5000/demoab:latest
    imagePullPolicy: IfNotPresent
    command: [ "/app/start", "start" ]
    env:
    - name: ABN
      value: "1200"
    - name: ABC
      value: "12"
    - name: ABID
      value: "id1"
    - name: ABTARGETURL
      value: "localhost/test.bin"
    - name: PUSHGWIP
      value: "192.168.0.235"
    resources:
      requests:
        memory: "500Mi"
        cpu: "1"
      limits:
        memory: "500Mi"
        cpu: "1"
  - name: demoab2
    image: localhost:5000/demoab:latest
    imagePullPolicy: IfNotPresent
    command: [ "/app/start", "start" ]
    env:
    - name: ABN
      value: "1200"
    - name: ABC
      value: "12"
    - name: ABID
      value: "id2"
    - name: ABTARGETURL
      value: "localhost/test.bin"
    - name: PUSHGWIP
      value: "192.168.0.235"
    resources:
      requests:
        memory: "500Mi"
        cpu: "1"
      limits:
        memory: "500Mi"
        cpu: "1"
  - name: demoab3
    image: localhost:5000/demoab:latest
    imagePullPolicy: IfNotPresent
    command: [ "/app/start", "start" ]
    env:
    - name: ABN
      value: "1200"
    - name: ABC
      value: "12"
    - name: ABID
      value: "id3"
    - name: ABTARGETURL
      value: "localhost/test.bin"
    - name: PUSHGWIP
      value: "192.168.0.235"
    resources:
      requests:
        memory: "500Mi"
```

```
        cpu: '1'
      limits:
        memory: "500Mi"
        cpu: '1'
  restartPolicy: Never
EOF
```

For CMK, use the following template:

```
cat demo-cmk.yaml << EOF
kind: Pod
apiVersion: v1
metadata:
    generateName: demo-cmk
spec:
  containers:
  - name: demonginx
    image: localhost:5000/demonginx:latest
    imagePullPolicy: IfNotPresent
    command: [ "/app/start", "start" ]
    securityContext:
      capabilities:
         add: ["IPC_LOCK"]
    resources:
      requests:
        memory: "1Gi"
        cmk.intel.com/exclusive-cores: '2'
        qat.intel.com/cy1_dc0: '1'
      limits:
        memory: "1Gi"
        cmk.intel.com/exclusive-cores: '2'
        qat.intel.com/cy1_dc0: '1'
  - name: demoab1
    image: localhost:5000/demoab:latest
    imagePullPolicy: IfNotPresent
    command: [ "/app/start", "start" ]
    env:
    - name: ABN
      value: "1200"
    - name: ABC
      value: "12"
    - name: ABID
      value: "id1"
    - name: ABTARGETURL
      value: "localhost/test.bin"
    - name: PUSHGWIP
      value: "192.168.0.235"
    resources:
      requests:
        memory: "500Mi"
        cmk.intel.com/exclusive-cores: "1"
      limits:
        memory: "500Mi"
        cmk.intel.com/exclusive-cores: "1"
  - name: demoab2
    image: localhost:5000/demoab:latest
    imagePullPolicy: IfNotPresent
    command: [ "/app/start", "start" ]
    env:
    - name: ABN
      value: "1200"
    - name: ABC
      value: "12"
    - name: ABID
      value: "id2"
    - name: ABTARGETURL
      value: "localhost/test.bin"
    - name: PUSHGWIP
      value: "192.168.0.235"
    resources:
```

```
      requests:
        memory: "500Mi"
        cmk.intel.com/exclusive-cores: '1'
      limits:
        memory: "500Mi"
        cmk.intel.com/exclusive-cores: '1'
  - name: demoab3
    image: localhost:5000/demoab:latest
    imagePullPolicy: IfNotPresent
    command: [ "/app/start", "start" ]
    env:
    - name: ABN
      value: "1200"
    - name: ABC
      value: "12"
    - name: ABID
      value: "id3"
    - name: ABTARGETURL
      value: "localhost/test.bin"
    - name: PUSHGWIP
      value: "192.168.0.235"
    resources:
      requests:
        memory: "500Mi"
        cmk.intel.com/exclusive-cores: '1'
      limits:
        memory: "500Mi"
        cmk.intel.com/exclusive-cores: '1'
  restartPolicy: Never
```

The goal of this example is to show how controlled placement and predictable performance can be achieved on a server with Intel® Hyper-Threading Technology (Intel® HT Technology) enabled and by using a QAT device as an accelerator. For that we used CPU Manager for Kubernetes, which uses isolcpus and respects the difference between physical and hyper-threaded cores. In our example, requesting physical core does not schedule anything on hyper-threaded cores. The application in the scheduled pod needs to take care on which cores it runs. The NGINX startup script selects those cores, which is equivalent to how DPDK applications would do that. We did not use Kubernetes CPU Manager because it does not differentiate between physical and hyper-threaded cores. For the management of the accelerator devices we used the QAT Device Plugin.

In the example code, line "`qat.intel.com/cyl_dc0: 1`" controls consumption of QAT, where the QAT Device Plugin maps QAT into the pod by also giving access rights as it is going through the kernel drivers. Line "`cmk.intel.com/exclusive-cores: '2'`" gets cores exclusively allocated to the container.

Using a browser, go to the Grafana page `http://$PUSHGWIP:3000`, login with admin/password, and change the password.

Add data source Prometheus with URL `http://$PUSHGWIP:9090` at desired Scrape interval, save, and test.

Create a dashboard using Add Query Prometheus with Metric Transfer Rate and desired refresh rate.

5.12 Deploy from Cloudify

Cloudify deployment for NGINX with preference to QAT (Figure 6) returns a result similar to Figure 9.

```
[root@master1 ~]# kubectl describe pod nginx
Name:               nginx
Namespace:          default
Priority:           0
PriorityClassName:  <none>
Node:               node1/192.168.0.236
Start Time:         Wed, 09 Oct 2019 10:27:12 +0300
Labels:             env=nginx
Annotations:        k8s.v1.cni.cncf.io/networks-status:
                      [{
                          "name": "cni0",
                          "interface": "eth0",
                          "ips": [
                              "10.244.1.170"
                          ],
                          "mac": "0a:58:0a:f4:01:aa",
                          "default": true,
                          "dns": {}
                      }]
Status:             Running
IP:                 10.244.1.170
Containers:
  nginx:
    Container ID:   docker://1c606f2501f2d0e49f7dfe2b54d0cc846dc928262ff1186c27645918a31c3c5f
    Image:          nginx
    Image ID:       docker-pullable://nginx@sha256:aeded0f2a861747f43a01cf1018cf9efe2bdd02afd57d2b11f
    Port:           <none>
    Host Port:      <none>
    State:          Running
      Started:      Wed, 09 Oct 2019 10:27:22 +0300
    Ready:          True
    Restart Count:  0
    Environment:    <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-mljgc (ro)
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
Volumes:
  default-token-mljgc:
    Type:        Secret (a volume populated by a Secret)
    SecretName:  default-token-mljgc
    Optional:    false
QoS Class:       BestEffort
Node-Selectors:  load-balancer=true
Tolerations:     node.kubernetes.io/not-ready:NoExecute for 300s
                 node.kubernetes.io/unreachable:NoExecute for 300s
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  4m32s  default-scheduler  Successfully assigned default/nginx to node1
  Normal  Pulling    4m26s  kubelet, node1     pulling image "nginx"
```

Figure 9. Describe Pod Correctly Assigned

5.13 Result

After the environment is set correctly, the result can be observed as graphs similar to Figure 10.

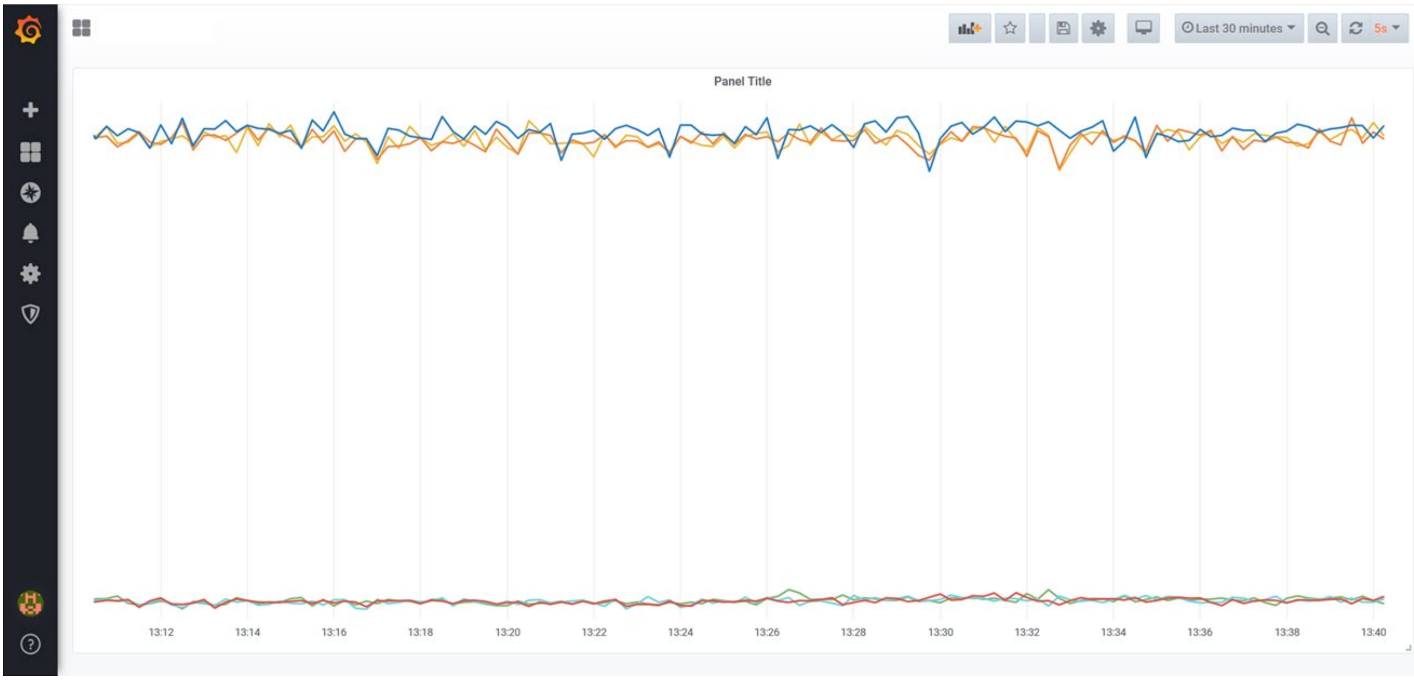

Figure 10. Grafana Graph with Metrics

5.14 Additional Documentation

Links to additional documentation:

SOURCE
https://01.org/packet-processing/intel%C2%AE-quickassist-technology-drivers-and-patches
https://github.com/openssl/openssl
https://github.com/intel/QAT_Engine
https://github.com/intel/QATzip
https://github.com/intel/asynch_mode_nginx
https://nginx.org/
https://www.nginx.com/resources/wiki/start/
https://httpd.apache.org/docs/2.4/programs/ab.html
https://01.org/sites/default/files/downloads/intelr-quickassist-technology/337020-001qatwcontaineranddocker.pdf

6 Summary

This technology guide demonstrates how different types of latency-sensitive workloads can be intelligently placed on the correct node in a mostly fully automated way while consuming available accelerations such as QAT accelerators. All of the components used are open-sourced, ready to be used by the customer. You can use Container Bare Metal Reference Architecture Ansible playbooks as a good starting point to become familiar with different types of available options and Kubernetes plugins.

0321/DN/WIPRO/PDF 616619-003US