Pavilion NVMe-oF Storage Platform Kubernetes Reference Guide

Version 2.3.5

Last Updated: April 2022
Document Version: Version 2.3.5

Legal Notice:

Technical Support:

Technical Support maintains support centre's globally. If you have questions regarding an existing support agreement, please email/phone the support agreement administration team as follows:

Phone:

USA & Canada: 1-888-342-0461

United Kingdom: 0800–69-8055

Netherlands: 0800-022-2832

Australia: 1800-719-861

India: 000-800-919-1301

International: 1-408-684-4958

Email: support@pavilion.io

Documentation:
Make sure that you have the current version of the documentation. Each document displays the date of the last update on page 2.

Documentation Feedback:
Your feedback is important for us.
For documentation feedback send your comments to your Pavilion field support contact.

Contents

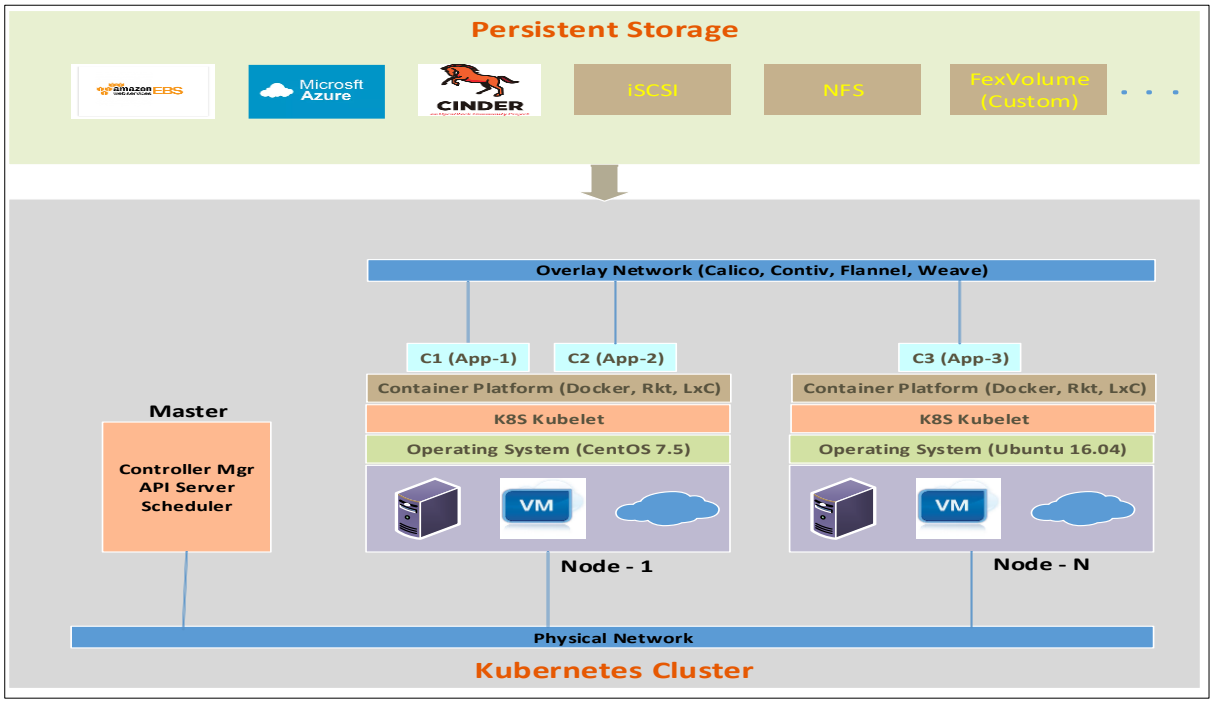
1. About Kubernetes

Kubernetes (commonly abbreviated as **K8s**) is an open-source **container-orchestration system** for automating deployment, scaling, and management of containerized applications. It was originally designed by Google and is now maintained by the Cloud Native Computing Foundation.

The **Image: K8s Components and Deployment Overview**, provides an overview of the various components of Kubernetes and gives a crisp picture of its deployment.

The **Table: Kubernetes Components** that follows the diagram gives an overview of the K8s components.

Image: K8s Components and Deployment Overview



The **Table: Kubernetes Components** gives an overview of K8s components.

Table: Kubernetes Components

Kubernetes Components	Overview
Kubernetes Cluster	▪ **Kubernetes Cluster** is a collection of either physical, or virtual machines (VMs) on which containerised applications are to be run. ▪ A cluster contains **at-least one master node**, and **multiple worker nodes** (also referenced as worker machines or minions). ▪ All cluster members run the K8s orchestration system.
Master	▪ **Master** is a collection of services that make up the K8s control plane of the cluster. This is the brain of the cluster where **all control and scheduling decisions** are made. ▪ The **services** that the Master offers include: ✓ an *API Server* which exposes a RESTful interface ✓ the *Cluster Store* which provides a consistent key-value store ✓ the *Controller Manager* which provides the reconciliatory monitoring loop, and ✓ the *Scheduler* which watches for new workloads and assigns them to nodes as required
Nodes	▪ Nodes are the machine units (physical or VM based) on which the **actual containerised applications** are run. ▪ The K8s elements that make up a node include the **Kubelet,** which is the **main K8s agent** on the node which communicates with the master, and the

	container runtime framework which provides for all the container management tasks: ✓ like pulling requisite images ✓ starting/ stopping containers etc, and ✓ the **Kube-Proxy** which provides the networking related services like IP assignment, load-balancing etc on the node.
Pods	▪ **Pods** are the **minimum units of scaling** in K8s. A Pod provides a shared execution environment for one or more containers. ▪ All the containers in a Pod share the same external IP address, hostname, sockets, memory, and volumes. ▪ One or more than one container can be run from within a Pod.
Overlay Network	▪ **Overlay Network** is a network built on top of another low-level network. ▪ A Pod network in K8s is an overlay network that takes individual private networks within each node and transforms them into a new software defined network (**SDN**) with a shared namespace, allowing pods to communicate across nodes.
Persistent Storage	▪ **Persistent Storage** is important for running stateful containerised applications. ▪ K8s provides access to persistent storage via **in-tree plugins** i.e. plugins which were built, compiled, linked, and shipped with the core K8s binaries. ▪ Also, via **out-of-tree plugins** including **Container Storage Interface** (**CSI**) and **FlexVolume plugins** K8s can be setup to access storage from a custom device.

Applications running on containers have historically been ephemeral i.e. when a Pod hosting a Container stops, fails, or restarts, all the data associated with the application is lost.

However, with the rapid development and adoption of containerised applications, use cases for data storage to be made available across pod failures demand a case for persistent storage.

The sub-section below depicts the two major workflows related to allocation, and provisioning storage using K8s.

1.1Storage Allocation and Provisioning

The term **Allocation** here, refers to the methodology by which **storage is managed** (created/deleted) on the underlying storage entity (custom chassis in the case of **Pavilion**).

The term **Provisioning** here, refers to the methodology by which **storage** allocated on the underlying storage entity is **discovered, mounted** and **used by the containerised application**.

Two **different methods** for Storage Allocation and Provisioning from within a K8s cluster are as follows:

- **Static Storage Allocation and Provisioning**
- **Dynamic Storage Allocation and Provisioning**

The primary difference between the two modes of allocation is the demarcation of roles expected from the **Storage Admin** (the personnel responsible for managing the storage) and the **Application Developer** (the personnel responsible for developing and deploying containerised applications which require persistent storage).

The workflow related to both methods is explained in detail in the below sections.

1.1.1 Static Storage Allocation and Provisioning

In the **Static Storage Allocation** model, there is a clear demarcation of roles between the Storage Administrator, and the Application Developer.

The **Storage Admin** is responsible for maintaining the backend storage entities (creating, deleting) and providing references of the same to the K8s cluster.

Whereas, the **Application Developer** is responsible for using the storage which has been pre-provisioned**.**

Note: The Application Developer **cannot create/destroy storage entities** from within the application.

There are further **two discrete ways** by which the Application Developer can make use of the provisioned storage, they are:

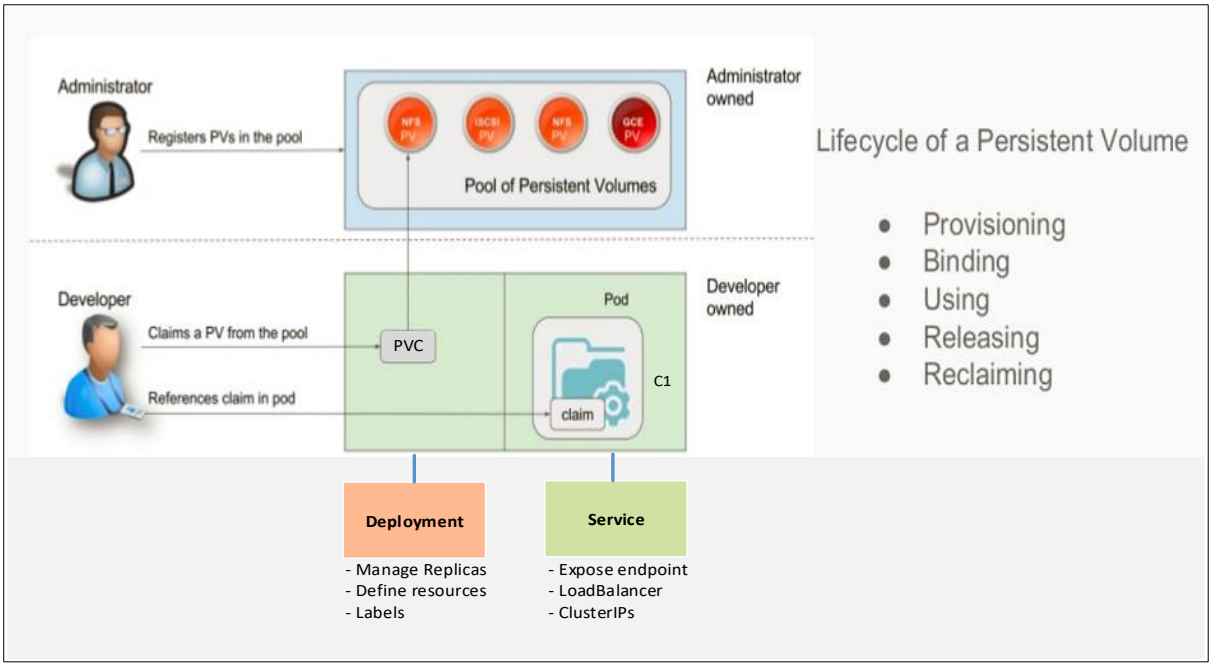
- Volume Provisioning with **Persistent Volume (PV)**
- Volume Provisioning from App

The below sections elaborate the same.

1.1.1.1 Volume Provisioning with Persistent Volume (PV)

The **Image: Dynamic Volume Provisioning Workflow** gives an overview of dynamic volume provisioning.

Image: Dynamic Volume Provisioning Workflow



Listed below is the Dynamic Volume Provisioning Workflow:

Step 1: The Storage Admin creates a volume with the desired flavour independent of K8s.

Step 2: Storage Admin creates a K8s **Persistent Volume (PV)** specification file which has a section containing all the required fields, allowing for mapping of storage entity on chassis

Step 3: Each PV has a unique name using which the PV can be mapped uniquely.

Step 4: K8s developer creates a **Persistent Volume Claim (PVC)** specification file which has the name and storage class mapping with a PV specification, created

previously by the Storage Admin.

Step 5: Now, when a containerised application is desirous of using the storage pre-provisioned above, in the **`yaml`** specification file for the application, the mapping to PVC needs to be specified, along with the desired mount path.

Step 6: When the application is provisioned by K8s user, all the necessary steps required for mounting the volume are run, so that the containerised application can access the provisioned storage at the specified mount path.

1.1.1.2 Volume Provisioning from App

For volume provisioning from the APP follows the below workflow:

Note: In this method, the Application Developer requires all the information of the backend storage when developing the application specification file.

Step 1: Storage Admin creates a volume with desired flavour independent of K8s.

Step 2: K8s developer creates an application specification file which consists of a sub-section containing all required fields, which allow mapping of volume to a chassis (like Chassis IP, name of volume to be accessed, parent media group name, size of volume etc)

Step 3: When the App is provisioned by K8s user, all the necessary steps required for mounting the volume shall be run, such that the containerised application can access the provisioned storage at the specified mount path.

1.1.2 Dynamic Storage Allocation and Provisioning

While there is a clear demarcation of role of the Storage Admin and the Application Developer in the **Static Storage Allocation model**; there is also a limitation where an application developer cannot create/ delete of storage entities, which may result in sub-optimal usage of the provisioned backend storage.

In the **Dynamic Storage Allocation Model**, there is a **`StorageClass`** associated with each backend storage entity, and using the **`StorageClass`**, and the provided tool(s), an Application Developer can have full control of the underlying provisioned storage entity i.e. can create/ delete storage entities and use the allocated storage entities in the containerised applications.

.

1.2 Out of Tree Storage Provisioning

The K8s specification has **in-built support** for **In-Tree plugins** for providing persistent storage for generic storage classes/storage providers such as AWS EBS, GCE and OpenStack.

The **Out-of-tree volume plugins** enable storage vendors to create custom storage plugins.

Note: The Out-of-tree plugins include the **Container Storage Interface (CSI**) and **FlexVolume plugins**.

Both CSI, and FlexVolume plugins allow for development which is independent of the K8s code base and can be deployed on K8s clusters as extensions.

1.2.1 K8s Flex Volume Plugin

The **FlexVolume plugin** allows for an **execution-based model** to interact with the underlying storage driver. The FlexVolume binary needs to be installed on all the nodes at a pre-defined volume-plugin path.

The K8s master in the cluster guides the **Kubelet** running on the selected node/s to invoke the plugin with intended call-out, passing the required options in string as well as JSON format, depending on the call-out parameter.

K8s ensures that the calling-plugin options are sent in an orderly sequence to interact with the underlying volume.

The volume plugin driver returns the operation status and other information in given **`JSON`** format by writing a **`JSON`** string on **`stdout,`** which is captured further by the **Kubelet** process running on the node.

K8s allows usage of the FlexVolume plugin to pass standard parameters, as well as vendor defined parameters, as defined in the **`PersistentVolume`** object definition.

The default path for installation of the FlexVolume plugin is as follows:

```
/usr/libexec/kubernetes/kubelet-plugins/volume/exec/vendor~driver/<driver>
```

Note: It can be changed in **Kubelet** via the **`--volume-plugin-dir`** flag, and in **controller manager** via the **`--flex-volume-plugin-dir`** flag.

As part of the mount process, the **Kubelet** process running on the node mounts the external volume on the node under the **`/var/lib/Kubelet`** directory, and then from there, it can map the volume into the container at the mount point documented in the application specification file.

1.2.2 Container Storage Interface (CSI)

Container Storage Interface (**CSI**) defines a standard interface which can be used by different container orchestration systems (like Kubernetes, Apache Mesos, DC/OS etc) to expose arbitrary/custom storage systems to containerised workloads.

While the FlexVolume plugin is based on the **executable call-out model**; the CSI is based on **Remote Procedure Calls (RPC),** and the CSI interface declares the RPCs that a plugin must expose.

Once a CSI compatible volume driver is deployed on a cluster, users may use the CSI volume type to attach, mount (etc) the volumes exposed by the CSI driver.

2. Pavilion Data Storage Allocation and Provisioning for Kubernetes

Pavilion provides a high density, low latency NVMe-oF storage solution for rack-scale modern applications. **Pavilion** also provides K8s plugins allowing for containerised applications orchestrated by K8s to dynamically allocate storage, and access shared storage on the chassis.

Pavilion provides a tool **`pvlctl`** which when deployed on the K8s master node, can be used to dynamically allocate/ delete storage entities (volumes, snapshots, clones) on the underlying **Pavilion** chassis(s). The tool is required to be run only on the K8s master prior to deploying the containerised App.

Pavilion also provides a standard compliant FlexVolume driver which can be used to access shared storage on a **Pavilion** chassis, over standard NVMe-oF transport paths (RoCE or TCP). **Pavilion** also offers a FlexVolume configuration option, which allows access to storage on a **Pavilion** chassis over NFS, such that individual containers can be provided with a dedicated quota for storage.

The **Table: Pavilion Data support for Out-Of-Tree K8s Volume Plugins** lists the two out-of-tree plugins and their status and integration status with **Pavilion**.

Table: Pavilion Data support for Out-Of-Tree K8s Volume Plugins

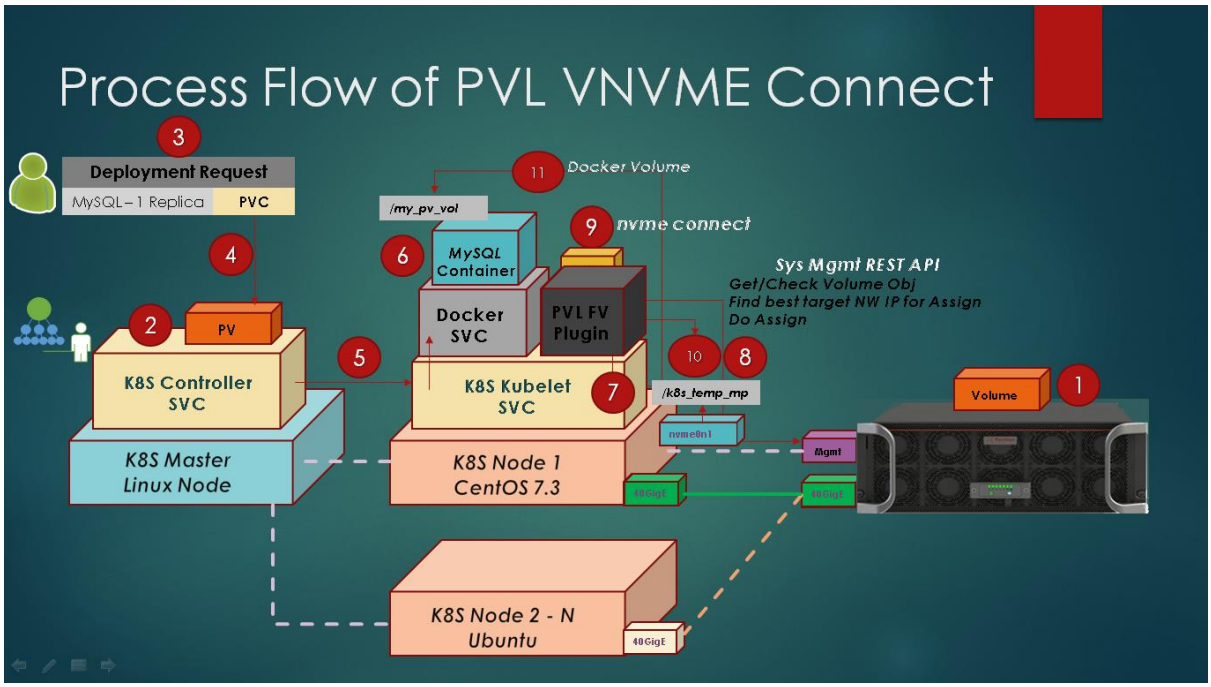
Sr. No.	Out-of-tree Plugin	Current K8s Status	Pavilion Integration
1.	**FlexVolume**	Introduced in K8s release v1.2; GA since K8s release v1.8	Supported
2.	**CSI**	Introduced in K8s release v1.9, GA in K8s release v1.13	**Pavilion** would support CSI for the upcoming releases.

2.1 Kubernetes Pavilion Workflow

The **Image: Kubernetes Pavilion Workflow** displays the detailed block diagram of K8s components involved in the **Pavilion** K8s integration (for statically allocated storage), and the numbered steps displayed in the image enumerate the sequential workflow.

The end goal of the application user as shown in the diagram below, is to run a MySQL application in a container, and access storage which has been provisioned on a **Pavilion** chassis.

Image: Kubernetes Pavilion Workflow



The numbered steps displayed in the above image are explained as follows:

Step 1: Storage Admin creates a volume of desired size, media group flavour, stripe size etc on **Pavilion** chassis.

Step 2: K8s Storage Admin creates a Persistent Volume (PV) specification file which has all the mapping data required to map storage to the volume created in **Step 1.**

Step 3: The K8s user creates a **Persistent Volume Claim (PVC)** specification file which maps to the PV specification file as per **Step 2**.

Step 4: Once the PVC as in **Step 3** is bound to the PV in **Step 2**, the K8s can create a containerised App which can access the storage using the said PV.

Step 5: The K8s user then instantiates a containerised App which has the required mapping to the PVC created as in **Step 3**. At this point, the K8s master instructs the Kubelet on a node (Node 1 in above example) to spin up a new container with specified App and storage mapping.

Step 6: The required container image/s are then pulled from the container repository.

Step 7: At this point of time, the **Pavilion** FlexVolume **`pvlnvmefv`** plugin is invoked, and a connection handle with the **Pavilion** chassis management port is established.

Step 8: The required checks for presence of volume of specified name, media group mapping, state, size etc are carried out by the **`PVLFV`** plugin. At this point, the mapping to the required chassis data ports is also completed.

Step 9: The suitable **`nvme`** commands are run to discover the said volume from the said chassis dataport.

Step 10: On successful completion of the discovery of the underlying target volume, the block device is formatted to the said file system in the PV spec, and appropriately mounted in the Pod.

Step 11: Once the mount operation is completed, the volume can be accessed by the containerised application at the mount location specified in the specification file as per **Step 5**.

3. Pavilion Flex Volume Deployment and Configuration

3.1 Pre-Requisites on Chassis

Pavilion FlexVolume deployment and configuration pre-requisites on chassis are as follows:

- Install and load appropriate host-sider driver, on all your Kubernetes nodes, based on the protocol, you would use, to access **Pavilion** volumes.
  For example, load/install iSCSI drivers if you are to access iSCSI volumes.
- Media group of desired flavor/stripe setting is to be provisioned.
- Volumes of desired size/s to be provisioned in the said media group/s.
- Controller data port transport protocol configuration is to be setup (either NVMe over RoCE or NVMe over TCP or NFS).
- Controller dataport IP/subnet is to be configured.
- Dataport is to be configured as part of management port setup, if the dataport is to be doubled up as the management port.

3.2 Pre-Requisites on K8s Nodes and Master

For **Pavilion** FlexVolume deployment and configuration pre-requisites on K8s Nodes and master is as follows:

- NVMe utils/tools is to be installed on all the nodes in the cluster.
- Host driver (RDMA/ TCP) setup is to be completed on all the nodes in the cluster. See *Pavilion NVMe-oF Storage Platform Host Installation and Configuration Guide* to determine the steps required to setup the host drivers, and the OS Versions supported.

3.3 Configuration of Pavilion Flex Volume on K8s Nodes

Pavilion installation script is to be run on all the nodes and master (if master also doubles up as a node). The installation script performs the following operations:

- Get user input for preferred transport to use (NVMe over RoCE or NVMe over TCP).
- Check for presence of system commands (`command`, `ip`, `lsblk`, `umount`, `grep`, `awk`, `mkfs`) / utils (`nvme`) required for further operations.
- Discover all the IPv4 enabled network interfaces on host, as the communication with the **Pavilion** chassis is supported over IPv4.
- Get inputs from user for preferred path/s over which to establish communication with **Pavilion** chassis (It is essential that the host NW interface/s are on the same subnet as the dataport IP)
- Get inputs from the user for HA configuration requirements to allow for application HA solutions.
- Allow configuration for load balancing, such as to get optimum performance from the **Pavilion** chassis.
- At the end of the install script, if all of the requisite host drivers/ tools / system utilities are determined to be present the K8s Node, a network configuration file **`pvl-fv-config.json`** is created under **`/etc/pavilion`**, and the **Pavilion** flex volume driver `pvlnvmefv`is installed at the following default path:

  `/usr/libexec/kubernetes/kubelet-plugins/volume/exec/pavilion~pvlnvmefv/pvlnvmefv`

Note: As mentioned above, the drive can be over-ridden using the options provided. From usability perspective, once the initial deployment script is run, all interactions with the PVL FlexVolume are managed by the K8s environment itself, and **no manual intervention** is required.

4. Pavilion pvlctl Deployment and Configuration

4.1 Pre-Requisites on Chassis

Pavilion `pvlctl` deployment and configuration pre-requisites on chassis is as follows:

- Media group of desired flavour/stripe setting is to be provisioned.

4.2 Pre-Requisites in K8s cluster

Pavilion `pvlctl` deployment and configuration pre-requisites on K8s master is as follows:

- NVMeutils/ tools is to be installed on all nodes in the cluster.
- Host driver (RDMA/ TCP) setup is to be completed on all the nodes in the cluster.
  See *Pavilion NVMe-oF Storage Platform Host Installation and Configuration Guide* to determine the steps required to setup the host drivers, and the OS Versions supported.
- **Pavilion `pvlctl`** installation script is run on the **master** node. The installation script performs the following operations:
  - At the end of the Install script, using the **command-line (CLI)** options, and a configuration file in the prescribed format, storage entities (volumes, snapshots, clones) can be dynamically created/ deleted on the **Pavilion** chassis from within the K8s cluster.

The command-line (**CLI**) options supported by the **pvlctl** utility are mentioned in the **Table: CLI options supported by the pvlctl utility.**

Table: CLI options supported by the `pvlctl` utility

CLI Option	Description
create	• Allows to create a new storage entity (volume, snapshot, or clone) on the **Pavilion** Chassis; along with creation of its associated PV and PVC specification file. • **Example: `pvlctl create <Pavilion_Storage.yaml>`**
delete	• Allows to delete a storage entity (volume, snapshot or clone) on the **Pavilion** Chassis; along with deletion of its linked PV and PVC. • **Example: `pvlctl delete <PVC_Spec_Name>`**
view	• Allows to view the storage parameters linked with a **Pavilion** PVC. • **Example: `pvlctl view <PVC_Spec_Name>`**

A **sample Pavilion Storage file (`pvlctl` utility configuration file)** has the following format:

```
apiVersion: v1
kind: pavilion
metadata:
  storageEntity: Volume
spec:
  storageProps:
    fsType: "ext4"
    options:
      chassis: "172.25.50.41"
      chassis_version: "2.0"
      uid: "operator"
```

```
      pwd:
"b30cfbf4765c76fba16840a6d692c03f6d0040da9705e7452db252b9a818031f"
      pvlvolsizeingb: "111"
      pvlvolname: "autokube"
      pvlmgname: "kubemg"
      pvlvolsecret: ""
```

Note:

For creating snapshots and clones, the **`storageEntity`** attribute in above **`pvlctl` utility configuration file** is to be changed to "**Snapsho**t" and "**Clone**" and the required parent volume details also need to be added.

Install **Pavilion Flex Volume /PvlCtl**, to obtain these sample templates available under **`/etc/pavilion`**

5. Pavilion Application Proxy for Secure Communication

As explained in **Pavilion-Kubernetes** integration, Kubernetes nodes directly communicates with **Pavilion** chassis system management to invoke its REST APIs. Hence, it expects network path between the nodes and **Pavilion** chassis, which might not exist due to private network setup, or because of physical network isolation.

Other way to avoid a separate network connection between the nodes and **Pavilion** chassis system management is to enable **Dataport** through management option on **Pavilion** chassis, which enables the nodes to invoke **Pavilion** Management REST APIs through the network configured for accessing **Pavilion** persistent volumes.

Though in both the scenarios, Kubernetes-cluster nodes are directly communicating with the **Pavilion** system management web-service with valid admin credentials, still there are security concerns if the persistent volumes are to store vital data.

To overcome these security concerns, one can leverage **Pavilion Application Proxy** to secure communication between Kubernetes cluster nodes to **Pavilion** chassis.

Pavilion Application Proxy supports the following use-cases:

1. **Pavilion Application Proxy** is a lightweight service that acts as a **gateway** to allow managed and controlled communication between Application/Framework deployments with **Pavilion** chassis system management web-service.

2. **Pavilion** chassis management IP-address can be abstracted from Application cluster like K8s cluster nodes, both from the **master** and **slave** nodes. Network path may not exist between nodes and chassis management.

3. The proxy serves only those **Pavilion** system management APIs which are required for managing application deployment. For instance, in case of K8s, it does not allow execution of APIs like delete media group from K8s nodes.

4. The proxy can serve as a gateway for one or more **Pavilion** chassis at the same time.

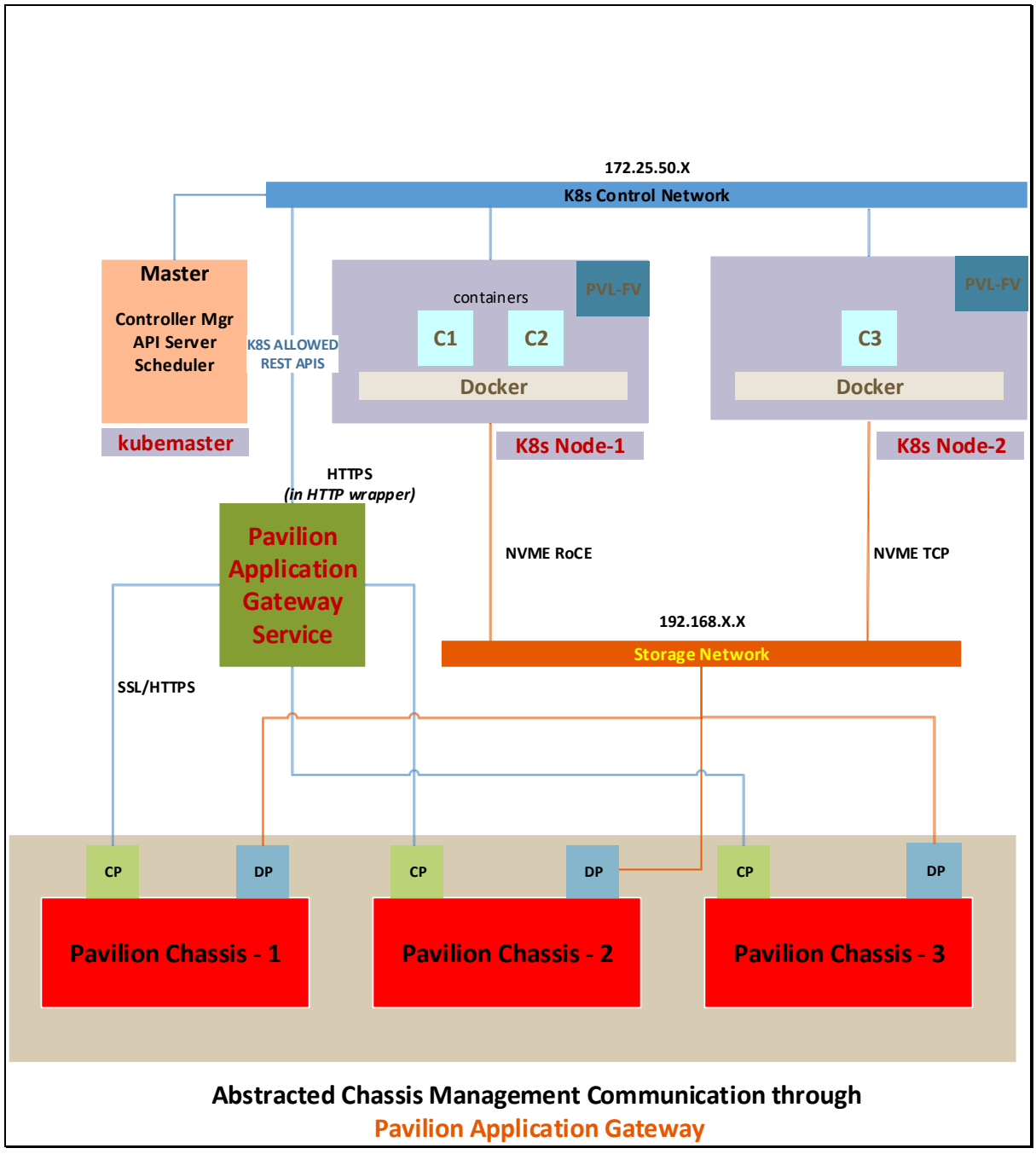
5. The proxy service can run on any system or within a container as well. For K8s, a docker container can be prepared and executed on a K8s node, only

when there is a physical network connection to system management web-service. Alternatively, the proxy executable can be configured as a Linux service.

6. The proxy service allows application nodes communication to fire REST APIs of system management web-service, only via **HTTPs**.

The **Image: Pavilion Application Proxy Configuration in K8s environment** depicts how **Pavilion Application Proxy** can be configured in Kubernetes environment. It also depicts the use cases mentioned in the above section.

Image: Pavilion Application Proxy Configuration in K8s environment



Abstracted Chassis Management Communication through Pavilion Application Gateway

The above image depicts the following:

1. The **kubemaster** (master), **K8s Node-1**, and **K8s Node-2** are the **hosts** that communicate with the **Pavilion Application Gateway Service** *(depicted in green).* The proxy service can run on any system or within a container as well.

2. For **Pavilion Chassis 1, 2, and 3** *(depicted in red)*, IP-address can be abstracted from application cluster like K8s cluster nodes, both from the master and slave nodes. Network path may not exist between nodes and chassis management.

3. The proxy serves only those **Pavilion** system management APIs which are required for managing application deployment. For instance, in case of K8s, it *does not allow* execution of APIs like delete media group from K8s nodes.

See *Pavilion NVMe-oF Storage Platform Application Proxy Service Guide* to learn more about installation and configuration.

The user is to append the following **three additional parameters** in the above explained "**`pvlctl`**" utility configuration file:

```
pvlproxy: "127.0.0.1:3129"
proxy_user: "pvlproxyadmin"
proxy_pwd: "proxypassword"
```

Further using **`pvlctl`** utility, defines and creates **Pavilion Flex Volume** based kuberneted persistent volume (**pv**) and (**pvc**) objects **yaml** configuration files.

This would enable Kubernetes nodes **Pavilion** Proxy to communicate with **Pavilion** system management web-service.

6. Kubernetes Pavilion Supported Use Cases

Using the **Pavilion Flex Volume plugin** and the **pvlctl utility**, allows for ease in deployment, thus allowing for further automation.

Listed below are the K8s **Pavilion** supported use cases.

- **Pavilion** volume access using either NVMe over RoCE or TCP based protocol, as K8s node set preference during the plugin deployment.
- Supported file-systems are: `ext-2`, `ext-3`, `ext-4`, `xfs`, and `btrfs`.
- **Pavilion** FlexVolume plugin also allows for accessing storage provisioned on the **Pavilion** chassis over NFS. For this, the `fsType` param needs to be set as `nfs`. In this case, the underlying volume on the **Pavilion** chassis is formatted as `ext4`.
- Both static and dynamic volume allocation and provisioning methods are supported.
- Container stop/start and re-attach volume automatically persisting any old data. This allows for persistence of data over container failovers.
- One container can attach/mount multiple volumes from same or different **Pavilion** chassis.
- In container HA case, where K8s master moves it over different node, **Pavilion** data volume gets auto-attached to the container as it was mounted earlier.
- Container 1 attached to Volume 1 over Node 1 over RDMA; can theoretically on failover be restarted on another Node 2 and attach to the same Volume V1 on another transport (TCP) protocol.
- Volume HA is supported.
- K8s node can leverage volume HA feature if it has multiple network ports connected to **Pavilion** chassis different controller ports.

7. Kubernetes Pavilion Rules and Limitations

Rules and Limitations while deploying the K8s solution from **Pavilion** are as follows:

- K8s node cannot use both NVMe over RoCE and TCP protocols simultaneously to access a **Pavilion** volume.

- RW-many access works only when C1 and C2 containers run over different nodes, N1 and N2 K8s nodes. Though in order to ensure file system level integrity it requires some cluster file-system configured over underline volume/s.

  Note: If C1 is using V1 over N1 and C2 too lands over N1 accessing V1, then C1 loses V1. This is a K8s FlexVolume issue/ limitation.

- All K8s nodes are recommended to have same network subnets configuration for paths going to chassis.

- IPv4 based volume access as restricted protocol by **Pavilion** chassis.

- K8s supports Windows containers too based on Windows 2016 release; however,the support for the **Pavilion** NVMe driver for Windows is not supported for version 2.0, it would be supported for the upcoming releases.

8. Kubernetes Pavilion Troubleshooting

K8s **Pavilion** troubleshooting measures are listed below:

- The logs related to the PVL FV Plugin can be monitored by standard **`journalctl utils`** on the node.

- The debug log level for the PVL FV Plugin can be changed by manually editing the **`pvl-fv-config.json`** file (which is created after running the PVL installation process) incase more detailed logs need to be captured.

- User can run the sample **`showinfo.sh`** script provided in the **Pavilion** K8s bundle, which allows for a quick mapping of the nodes/pods/storage objects, and their status in the current cluster. The script displays the following output:

```
# "Node details"
kubectl get nodes -o wide

# "PV details"
kubectl get pv -o wide

# "PVC details"
kubectl get pvc -o wide

# "POD details"
kubectl get pods -o wide

# "Service details"
kubectl get service -o wide

# "ReplicationController details"
kubectl get rc -o wide

# "Deployment details"
kubectl get deployments -o wide
```