Propose a solution for me as I want to be able to schedule hithub runners on kubernetes with possibility of a 3 node reservation per one task for multinode testing

15 web pages

To schedule GitHub runners on Kubernetes with the capability to reserve three nodes per task for multinode testing, you can leverage the Actions Runner Controller (ARC) for Kubernetes. Here's a step-by-step solution based on the available information:

1. Setup Kubernetes Cluster

First, ensure you have a Kubernetes cluster. If you're just testing or don't have an existing cluster, you might use:

- **Minikube** for local testing:

```bash
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
sudo install minikube-linux-amd64 /usr/local/bin/minikube
minikube start --driver=docker
```

- **EKS, GKE, or your preferred cloud provider** for production environments.

2. Install Actions Runner Controller (ARC)

- Use Helm to install ARC:

```bash
helm repo add actions-runner-controller https://actions-runner-controller.github.io/act
helm repo update
helm install actions-runner-controller actions-runner-controller/actions-runner-control
  --namespace actions-runner-system \
  --create-namespace
```

3. Authentication Setup

- **Create a GitHub Personal Access Token (PAT)** with the necessary permissions or use a GitHub App for more security:
  - For PAT, ensure it has `repo` and `admin:org` scopes for organization-level runners.

4. Configure ARC for Multinode Testing

- **Create a custom `RunnerDeployment` for your needs**:

```yaml
apiVersion: actions.summerwind.dev/v1alpha1
kind: RunnerDeployment
metadata:
  name: multinode-runner
spec:
  replicas: 1  # Start with one, scale as needed
  template:
    spec:
      repository: your-org/your-repo
      labels:
        - multinode-runner
      nodeSelector:
        multinode-capable: "true"  # Ensure nodes are tagged appropriately
      resources:
        requests:
          cpu: "1500m"  # Set to ensure 1.5 CPU per runner
          memory: "3Gi"  # Adjust as per node requirements
```

- **Implement Node Reservation with Pod Anti-Affinity**: To ensure each task gets three nodes, you can use pod anti-affinity rules:

```yaml
apiVersion: actions.summerwind.dev/v1alpha1
kind: RunnerDeployment
metadata:
  name: multinode-runner
```

```
spec:
  template:
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: "app"
                operator: In
                values:
                - multinode-runner
            topologyKey: "kubernetes.io/hostname"
      nodeSelector:
        multinode-capable: "true"
```

This ensures that each pod (runner) is scheduled on different nodes.

- **HorizontalRunnerAutoscaler (HRA) for scaling**:

```yaml
apiVersion: actions.summerwind.dev/v1alpha1
kind: HorizontalRunnerAutoscaler
metadata:
  name: multinode-runner-hra
spec:
  scaleTargetRef:
    kind: RunnerDeployment
    name: multinode-runner
  minReplicas: 0
  maxReplicas: 15  # Adjust based on your capacity needs
  scaleUpTriggers:
  - githubEvent:
      workflowJob: {}
    duration: "30m"
```

This will manage scaling runners based on workflow job events.

5. Modify Your GitHub Actions Workflow

- In your GitHub Actions workflow, specify to use these runners:

```yaml
name: Multinode Testing
on: [push]
jobs:
  multinode-test:
    runs-on: multinode-runner  # Matches your runner label
    steps:
    - uses: actions/checkout@v3
    - name: Run multinode test
      run: echo "Running test across 3 nodes"
```

6. Testing and Monitoring

- Monitor your runners with `kubectl get pods -n actions-runner-system` to ensure they're scheduling correctly across nodes.
- You might need to adjust `replicas`, `resources`, or `affinity` rules based on your testing outcomes and node availability.

This setup ensures that for each task, you're reserving three nodes by design through anti-affinity, although actual node usage depends on the specifics of your testing tasks. Remember, this is a foundational setup; you might need to tweak based on your exact requirements and the scale of operations.

How to attach physical interface - NIC - to a pod. For example ensure that each pod have one unique e810 intel cars attached

15 web pages

Attaching a physical Network Interface Card (NIC) like the Intel E810 directly to a Kubernetes pod involves using SR-IOV (Single Root I/O Virtualization) technology. Here's how you can ensure each pod has one unique E810 NIC attached:

Prerequisites:

- **Enable SR-IOV**:
  - Ensure SR-IOV is enabled in the BIOS/UEFI of your nodes.
  - Check if the Intel E810 NIC supports SR-IOV (it does, according to available documentation).
- **Install Necessary Components:**
  - **SR-IOV Network Operator** for managing SR-IOV resources in Kubernetes.
  - **Intel Ethernet Operator** for specific configurations with Intel E810 NICs.

Step-by-Step Guide:

1. Install SR-IOV Network Operator

- Use Helm or Operator Lifecycle Manager (OLM) to install:

```bash
helm repo add sriov-network-operator https://sriov-network.github.io/sriov-network-opera
helm repo update
helm install sriov-network-operator sriov-network-operator/sriov-network-operator \
  --namespace sriov-network-operator \
  --create-namespace
```

2. Configure SR-IOV for E810 NICs

- **Enable SR-IOV for the E810 NICs** on your nodes:
  - Check if SR-IOV is enabled:

    ```bash
    lspci -vv -s <PCI_address> | grep SR-IOV
    ```

  - Enable VFs:

    ```bash
    echo '2' > /sys/class/net/ens1f0/device/sriov_numvfs
    ```

- **Create a NetworkAttachmentDefinition** for each unique NIC:

```yaml
apiVersion: "sriovnetwork.openshift.io/v1"
kind: SriovNetworkNodePolicy
metadata:
  name: e810-policy
  namespace: sriov-network-operator
spec:
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
  resourceName: intel.com/e810
  numVfs: 2  # Adjust this based on how many VFs per PF you need
  nicSelector:
    deviceID: "1592"  # Device ID for E810
    vendorID: "8086"  # Vendor ID for Intel
```

Then, define the actual network:

```yaml
apiVersion: "k8s.cni.cncf.io/v1"
kind: NetworkAttachmentDefinition
metadata:
  name: e810-network
  namespace: default
spec:
  config: '{
    "cniVersion": "0.3.1",
    "type": "sriov",
    "vlan": 0,
    "resourceName": "intel.com/e810",
    "capabilities": {"ips": true}
  }'
```

3. Use in Pod Definitions

- Ensure your pods request the SR-IOV resources:

```yaml
```

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: sriov-pod
  annotations:
    k8s.v1.cni.cncf.io/networks: e810-network
spec:
  containers:
  - name: test-container
    image: ubuntu
    command: ["sleep", "3600"]
    resources:
      limits:
        intel.com/e810: 1
  nodeSelector:
    feature.node.kubernetes.io/network-sriov.capable: "true"
```

- This setup ensures each pod gets one unique E810 NIC VF. The `nodeSelector` ensures the pod is scheduled on nodes with SR-IOV capabilities for E810.

4. Additional Considerations:

- **Node Feature Discovery (NFD)**: To better manage node capabilities, install NFD to label nodes with hardware features like SR-IOV capabilities for E810.
- **Resource Management**: Ensure you have enough VFs created on your physical NICs to support the number of pods you intend to run with this setup.
- **Validation**: After deploying, check if the pod has indeed attached the E810 VF:

```bash
kubectl exec -it sriov-pod -- ip a
```

This setup leverages Kubernetes' ability to manage hardware resources through SR-IOV, ensuring each pod gets a dedicated network interface from the Intel E810 NICs. Remember, the exact configuration might need adjustments based on your specific environment and the number of physical NICs available.

how to attach a nic using passthrought

Attaching a Network Interface Card (NIC) to a virtual machine or container using passthrough (often referred to as PCI passthrough) in Kubernetes requires a few steps, primarily involving SR-IOV (Single Root I/O Virtualization) or VFIO (Virtual Function I/O) for direct hardware access. Here's how you can achieve this specifically for passthrough:

Prerequisites:

- **Hardware Support**: Ensure your hardware supports IOMMU (Intel VT-d or AMD-Vi) for PCI passthrough.
- **Kernel Modules**: Have `vfio-pci` loaded for enabling VFIO.
- **Kubernetes Version**: Preferably a recent version with good support for device plugins.

Step-by-Step Guide:

1. Enable IOMMU in BIOS/UEFI

- Make sure IOMMU is enabled in your server's BIOS or UEFI settings.

2. Configure Kernel for VFIO

- Add or ensure these kernel parameters in your bootloader configuration (`/etc/default/grub` for GRUB):

```bash
GRUB_CMDLINE_LINUX_DEFAULT="intel_iommu=on iommu=pt"
```

- Update GRUB and reboot:

```bash
```

```
sudo update-grub
sudo reboot
```

- After reboot, load necessary modules:

```bash
sudo modprobe vfio-pci
```

3. Identify and Bind NIC to VFIO

- Identify the physical NIC:

```bash
lspci | grep -i network  # Find your Intel E810 NIC
```

- For example, if the PCI address of your E810 is `0000:04:00.0`, you would bind it to VFIO:

```bash
echo 8086 1592 > /sys/bus/pci/drivers/vfio-pci/new_id  # Vendor and Device ID for E810
```

- Unbind the NIC from its current driver:

```bash
echo 0000:04:00.0 > /sys/bus/pci/devices/0000:04:00.0/driver/unbind
```

- Bind to VFIO:

```bash
echo 0000:04:00.0 > /sys/bus/pci/drivers/vfio-pci/bind
```

4. Kubernetes Setup with Device Plugin

- **Device Plugin**: Kubernetes uses device plugins to expose hardware devices to the scheduler. You'll need a VFIO device plugin or a more specific Intel E810 plugin if available.
    - **Install/Build a VFIO Device Plugin**:
        - There isn't a standardized VFIO plugin for Kubernetes, so you might need to use or adapt an existing one like NVIDIA's GPU plugin or write your own. Here's a basic outline:

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: vfio-device-plugin
  namespace: kube-system
spec:
  selector:
    matchLabels:
      name: vfio-device-plugin-ds
  template:
    metadata:
      labels:
        name: vfio-device-plugin-ds
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - image: your-vfio-plugin-image:latest
        name: vfio-device-plugin-ctr
        securityContext:
          privileged: true
        volumeMounts:
        - name: devfs
          mountPath: /dev
      volumes:
      - name: devfs
        hostPath:
          path: /dev
```

- **Register the Device**: Your plugin should register the device with Kubernetes so that it can be allocated. The plugin would look for the VFIO devices and register them as resources.

5. Use in Pod Definitions

- Once your device is registered, you can request it in your pod definitions:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: passthrough-pod
spec:
  containers:
  - name: test-container
    image: ubuntu
    command: ["sleep", "infinity"]
    resources:
      limits:
        vfio-device: "1"  # Assuming your plugin registers the device as 'vfio-device'
  nodeSelector:
    # Ensure this node has the device plugin running and the hardware
    vfio-capable: "true"
```

Notes:

- **Security**: Passthrough can bypass some VM/container security features, consider the implications.
- **Compatibility**: Not all workloads or applications will work directly with passthrough without modification.
- **Maintenance**: Direct hardware passthrough can make upgrades and maintenance complex; plan accordingly.

This approach allows for direct hardware access, which can be beneficial for performance-critical applications but requires careful management of resources and security considerations.

SR-IOV vs VFIO

SR-IOV (Single Root I/O Virtualization) and **VFIO (Virtual Function I/O)** are technologies used to provide direct hardware access to virtual machines (VMs) or containers, but they work differently and serve slightly different purposes. Here's a comparison:

SR-IOV:

Concept:

- SR-IOV allows a single physical PCIe device to appear as multiple separate physical devices. This is achieved by creating Virtual Functions (VFs) from a Physical Function (PF) of a network card or other supported hardware.

Key Points:

- **Performance**: Offers near-native performance since the guest OS communicates directly with the hardware through its own VF.
- **Scalability**: Can support many VFs on a single physical device, making it scalable for environments where many VMs or containers need network resources.
- **Management**: Requires specific driver support on both the host and guest sides. The management can involve configuring network policies, VLANs, etc., directly on the VF.
- **Security**: Since each VF can be isolated, security between VMs or containers can be enhanced, although it does introduce complexities in management.
- **Use Cases**: Ideal for high-performance networking scenarios in data centers, cloud environments, where low latency and high throughput are critical.

Drawbacks:

- **Hardware Dependency**: Not all NICs support SR-IOV, and those that do need to be configured properly.
- **Resource Allocation**: The number of VFs is limited by the hardware, and each VF consumes physical resources.

VFIO:

Concept:

- VFIO is part of Linux's IOMMU (Input/Output Memory Management Unit) support, allowing for direct access to PCI devices from within a VM or container by passing through the entire device (not just part of it like with SR-IOV).

Key Points:

- **Performance**: Provides direct access to the device, thus offering native performance similar to SR-IOV but without the need for VF creation.
- **Simplicity**: Once configured, it's simpler in terms of setup for the guest since it sees the device as if it's directly installed in its system.
- **Security**: Uses IOMMU to isolate devices from the host, enhancing security by preventing DMA attacks from the guest to the host memory.
- **Flexibility**: Can be used for any type of PCI device beyond just networking, like GPUs, storage controllers, etc.
- **Use Cases**: Useful in scenarios where you want to dedicate an entire device to a VM or container, particularly for graphics processing or specialized hardware acceleration.

Drawbacks:

- **Resource Consumption**: You're dedicating an entire physical device to one VM or container, which might not be efficient if you need to share resources.
- **Complex Setup**: Requires more setup on the host side, including kernel parameters, driver unbinding/binding, and IOMMU setup.
- **Limited Scalability**: Since you're passing through whole devices, the scalability is limited by the number of physical devices you have.

Comparison:

- **Resource Sharing**: SR-IOV is better for sharing a single device across multiple VMs or containers. VFIO is used when you want to give exclusive access to a device to a single VM or container.
- **Performance**: Both offer near-native performance, but SR-IOV might have slightly better latency due to no need for additional virtualization layers.
- **Management Overhead**: SR-IOV involves managing VFs, which can be both an advantage for network configurations and a challenge for administration. VFIO, once set up, requires less ongoing management but more initial configuration.
- **Use Case Fit**: Choose SR-IOV for high-density, high-performance networking environments. Opt for VFIO when dealing with specialized hardware that needs to be fully dedicated, like GPUs for intense compute tasks.

In summary, your choice between SR-IOV and VFIO would depend on your specific requirements regarding performance, scalability, security, and hardware availability.

And how can I request a physical function inside a pod?

Requesting a Physical Function (PF) of a device inside a Kubernetes pod directly is not a standard operation because PFs are typically used to manage and configure Virtual Functions (VFs) in SR-IOV scenarios, not for direct use by pods. However, if you're looking to pass through a PF to a pod for testing, development, or specific use cases where full device control is necessary, here are some conceptual steps you can take:

Challenges and Considerations:

- **Direct PF Passthrough**: Unlike VFs, which are designed for virtualization, passing through a PF directly to a pod is less common because PFs manage the hardware and control VFs. This can lead to complications in management and potentially security issues.
- **Security and Isolation**: Passing a PF directly to a pod could bypass some of Kubernetes' security mechanisms like network policies, leading to potential security concerns.
- **Resource Management**: Kubernetes is not inherently designed to manage PFs in the same way it does with VFs or other resources.

Steps for PF Passthrough (Conceptual):

1. Use VFIO for Passthrough

- **Enable IOMMU**: Ensure IOMMU is enabled in your system's BIOS/UEFI and kernel parameters as described in previous steps for VFIO.
- **Bind PF to VFIO**: Similar to binding VFs but with the PF:

```bash
# Assuming your PF has PCI address 0000:04:00.0
echo 8086 1592 > /sys/bus/pci/drivers/vfio-pci/new_id  # Vendor and Device ID for E810
echo 0000:04:00.0 > /sys/bus/pci/devices/0000:04:00.0/driver/unbind
echo 0000:04:00.0 > /sys/bus/pci/drivers/vfio-pci/bind
```

2. Kubernetes Device Plugin

- You would need a custom device plugin that can recognize and expose the PF as a resource. There isn't a standard plugin for this because it's not a common practice, but you can:
  - **Develop or Adapt a Device Plugin**: You'd need to write or modify a device plugin to detect and register the PF with Kubernetes. Here's a very basic example of what this plugin would do:

```go
// This is a conceptual snippet; actual implementation would require more complexit
func main() {
    // Detect PF
    // Register PF with Kubernetes as a custom resource
    // Listen for pod requests for this resource
    // Allocate the PF when requested by a pod
}
```

  - **Deploy the Plugin**: Deploy this plugin in Kubernetes, typically as a DaemonSet on nodes where the PFs are available.

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: pf-device-plugin
  namespace: kube-system
spec:
  selector:
    matchLabels:
      name: pf-device-plugin-ds
  template:
    metadata:
      labels:
        name: pf-device-plugin-ds
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - image: your-pf-plugin-image:latest
        name: pf-device-plugin-ctr
        securityContext:
          privileged: true
        volumeMounts:
        - name: devfs
          mountPath: /dev
      volumes:
      - name: devfs
        hostPath:
          path: /dev
```

3. Request PF in Pod Spec

- Once your device plugin registers the PF as a resource, you can request it in your pod spec:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pf-pod
spec:
  containers:
  - name: test-container
    image: ubuntu
    command: ["sleep", "infinity"]
    resources:
      limits:
        pf-device: "1"  # Assuming your plugin registers the PF as 'pf-device'
  nodeSelector:
    # Ensure this node has the PF plugin running and the hardware
    pf-capable: "true"
```

Caveats:

- **Not Production-Ready**: This is more of a research or experimental setup rather than something you'd deploy in production due to security, management, and scalability concerns.
- **Maintenance**: Managing PFs in this manner would complicate updates, security patches, and hardware maintenance.
- **Legal and Support**: Check with hardware vendors for support; some might not officially support or recommend this use case.

Remember, this setup is highly unconventional, and you might run into numerous operational, security, and performance issues. If your goal is to leverage hardware for high-performance computing or networking, consider using VFs or looking into alternatives like DPDK (Data Plane Development Kit) for networking within Kubernetes.

I am interested in DPDK

15 web pages

DPDK (Data Plane Development Kit) is an open-source framework for high-speed packet processing in user space. Here's an overview and how you can leverage DPDK in various contexts:

What is DPDK?

- **Purpose**: DPDK provides libraries and drivers to accelerate packet processing on standard hardware. It bypasses the traditional kernel network stack, which can be a bottleneck, by allowing applications direct access to the NIC (Network Interface Card) for packet handling.
- **Core Concepts**:
  - **Fast Path**: DPDK creates a fast path from NIC to the application, reducing context switches and kernel overhead.
  - **Poll Mode Drivers (PMD)**: Instead of interrupts, DPDK uses polling to check for new packets, which can be more efficient for high packet rates.
  - **Memory Management**: Utilizes huge pages to reduce TLB (Translation Lookaside Buffer) misses, enhancing memory access performance.
  - **NUMA Awareness**: Optimizes data locality by considering NUMA (Non-Uniform Memory Access) topology.

Using DPDK in Kubernetes:

1. Containerizing DPDK Applications:

- **Image Preparation**: Your Docker images should include DPDK libraries, PMDs, and any other necessary dependencies. You'll need to compile DPDK for your target architecture.
- **Huge Pages**: DPDK benefits from huge pages for memory management. You'll need to configure your Kubernetes nodes to support this:

  ```yaml
  apiVersion: v1
  kind: Node
  ```

```yaml
metadata:
  name: <node-name>
  annotations:
    hugepages-2Mi: "true"
```

Then, in your pod:

```yaml
spec:
  containers:
  - name: dpdk-container
    resources:
      limits:
        hugepages-2Mi: 100Mi
      requests:
        hugepages-2Mi: 100Mi
```

2. **NIC Passthrough for DPDK:**

- **SR-IOV**: Use SR-IOV to allocate Virtual Functions (VFs) to pods for direct hardware access. Here's a basic setup:
  - **Create SR-IOV Network Policy**:

    ```yaml
    apiVersion: sriovnetwork.openshift.io/v1
    kind: SriovNetworkNodePolicy
    metadata:
      name: dpdk-policy
      namespace: openshift-sriov-network-operator
    spec:
      resourceName: intel.com/dpdk
      nodeSelector:
        feature.node.kubernetes.io/network-sriov.capable: "true"
      numVfs: 4  # Adjust based on your needs
      nicSelector:
        vendor: "8086"
        deviceID: "1592"  # For E810
    ```

  - **Network Attachment Definition**:

    ```yaml
    apiVersion: "k8s.cni.cncf.io/v1"
    kind: NetworkAttachmentDefinition
    metadata:
      name: dpdk-network
    spec:
      config: '{
          "cniVersion": "0.3.1",
          "type": "sriov",
          "vlan": 0,
          "resourceName": "intel.com/dpdk"
      }'
    ```

  - **Use in Pod**:

    ```yaml
    apiVersion: v1
    kind: Pod
    metadata:
      name: dpdk-pod
      annotations:
        k8s.v1.cni.cncf.io/networks: dpdk-network
    spec:
      containers:
      - name: dpdk-container
        image: your-dpdk-image
        resources:
          limits:
            intel.com/dpdk: '1'
      nodeSelector:
        feature.node.kubernetes.io/network-sriov.capable: "true"
    ```

3. **DPDK with Kubernetes Enhancements:**

- **Device Plugins**: Use or develop a device plugin to manage DPDK resources. For example, the AF_XDP Device Plugin for Kubernetes can work with DPDK applications.
- **Performance Tuning**: Use tools like the Node Tuning Operator on OpenShift or equivalent in other Kubernetes flavors to optimize CPU isolation, memory allocation, and NUMA settings for DPDK workloads.

4. Practical Considerations:

- **DPDK Version Compatibility**: Ensure your DPDK version is compatible with your Kubernetes setup, especially when considering device plugins or specific networking hardware.
- **Security**: Since DPDK applications run in userspace, consider security implications, especially with direct hardware access.
- **Resource Management**: DPDK can be resource-intensive; manage CPU and memory carefully to avoid overcommitting and ensure performance.
- **Testing and Validation**: Test your setup thoroughly with the actual workloads you plan to run, as performance can vary greatly with different applications and network conditions.

Using DPDK in Kubernetes can significantly boost network performance for applications that require high throughput and low latency, but it requires careful configuration and management.

How about using DPDK and physical functions for attaching to a pod

Using DPDK with Physical Functions (PFs) for attaching directly to a pod is less conventional than using Virtual Functions (VFs) because PFs are typically used to manage VFs in SR-IOV setups. However, if you're looking to explore this for specific applications or scenarios, here's how you might approach it:

Challenges:

- **Resource Management**: PFs are not meant to be shared in the same way VFs are, so assigning a PF to a pod means dedicating an entire physical device to one pod, which might not be resource-efficient.
- **Security**: Direct access to PFs can bypass Kubernetes network policies and other security mechanisms because you're giving raw hardware access.
- **Complexity**: Managing PFs directly in containers requires custom configuration, monitoring, and potentially custom device plugins.

Steps for DPDK with PF Passthrough:

1. Enable IOMMU and VFIO

- **Enable IOMMU**: In your server's BIOS/UEFI and add kernel parameters if necessary:

  ```bash
  GRUB_CMDLINE_LINUX_DEFAULT="intel_iommu=on iommu=pt"
  ```

  After updating your bootloader configuration, reboot.

- **Load VFIO Module**:

  ```bash
  sudo modprobe vfio-pci
  ```

- **Bind PF to VFIO**:
  - Identify your PF:

    ```bash
    lspci | grep -i network
    ```

  - Bind to VFIO:

```bash
# Assuming your PF has PCI address 0000:04:00.0
echo 8086 1592 > /sys/bus/pci/drivers/vfio-pci/new_id  # Vendor and Device ID for E
echo 0000:04:00.0 > /sys/bus/pci/devices/0000:04:00.0/driver/unbind
echo 0000:04:00.0 > /sys/bus/pci/drivers/vfio-pci/bind
```

2. DPDK Application Preparation

- **Compile DPDK**: Compile DPDK for your system, ensuring it includes support for your NIC's driver in VFIO mode.
- **Application Development**: Write or adapt your application to use DPDK with VFIO for the PF.

3. Kubernetes Setup with Custom Device Plugin

- **Custom Device Plugin**: You'll need a device plugin that can identify and manage PFs:
  - **Development**: Develop a plugin that can detect PFs bound to VFIO and register them as Kubernetes resources. Here's a conceptual example:

```go
import (
    "k8s.io/kubelet/pkg/apis/deviceplugin/v1beta1"
)

func main() {
    // Detect PFs bound to VFIO
    // Register PF as a device resource in Kubernetes
    // Handle allocation and deallocation of PFs to pods
}
```

  - **Deploy the Plugin**:

```yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: pf-dpdk-plugin
  namespace: kube-system
spec:
  selector:
    matchLabels:
      name: pf-dpdk-plugin-ds
  template:
    metadata:
      labels:
        name: pf-dpdk-plugin-ds
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - image: your-pf-dpdk-plugin-image:latest
        name: pf-dpdk-plugin-ctr
        securityContext:
          privileged: true
        volumeMounts:
        - name: devfs
          mountPath: /dev
      volumes:
      - name: devfs
        hostPath:
          path: /dev
```

4. Pod with DPDK and PF Access

- **Pod Definition**:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: dpdk-pf-pod
spec:
  containers:
  - name: dpdk-container
    image: your-dpdk-pf-image
    command: ["/usr/local/bin/your-dpdk-app"]  # Your DPDK application
    resources:
      limits:
        pf-dpdk-device: "1"  # Assuming your plugin registers the PF as 'pf-dpdk-device
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "IPC_LOCK"]
    volumeMounts:
```

```
  - name: hugepage
    mountPath: /dev/hugepages
volumes:
- name: hugepage
  emptyDir:
    medium: HugePages
nodeSelector:
  pf-dpdk-capable: "true"
```

Considerations:

- **Security and Isolation**: You might need to add more security measures since you're essentially giving a pod direct hardware access.
- **Performance**: While performance can be maximized, you're dedicating resources to a single pod, which might not be optimal for multi-tenant environments.
- **Scalability**: This setup isn't scalable in a traditional sense; you're limited by the number of physical NICs you have.
- **Maintenance**: Updates or changes to the NIC or system configuration can be more complex with hardware directly bound to pods.

This setup is more exploratory or suited for specific high-performance scenarios where you need to leverage the full capabilities of a PF with DPDK in a containerized environment. Remember, this is not a standard Kubernetes practice, and you should weigh the benefits against the operational complexity.