Write a script for me using ansible for enabling PXE boot using remote BMC LAN connection. It should be set to use specific interface and boot from - if it is possible - from specific address - 10.123.232.180.

Below is an Ansible script to enable PXE boot using a remote BMC (Baseboard Management Controller) LAN connection, setting a specific network interface for PXE booting and attempting to configure the boot source to a specific PXE server address (10.123.232.180). The script uses the `community.general.redfish_command` module to interact with the BMC via Redfish API, which is commonly supported by modern BMCs (e.g., iDRAC, iLO).

Assumptions and Notes:

- **BMC Compatibility**: The script assumes the target BMC supports Redfish API. If your BMC uses a different protocol (e.g., IPMI), the approach would differ, and you'd need to specify that.
- **Interface Selection**: The script attempts to set the specified network interface for PXE boot. The exact interface name or ID depends on your hardware; you'll need to provide the correct identifier (e.g., `NIC1` or `eth0`).
- **PXE Server Address**: Setting a specific PXE server address (10.123.232.180) is not always supported by BMCs directly. The script configures PXE boot and provides a fallback to ensure the system attempts PXE from the specified interface. If your BMC supports advanced boot configuration, you may need vendor-specific Redfish extensions.
- **Ansible Requirements**:
  - The `community.general` collection must be installed (`ansible-galaxy collection install community.general`).
  - The BMC must be accessible over the network, and you need valid credentials.
- **Inventory**: The script assumes an Ansible inventory file with the target BMC's IP address, username, and password.

Ansible Playbook (`enable_pxe_boot.yml`)

```
---
- name: Enable PXE boot via BMC LAN and configure specific interface
  hosts: bmc_servers
  gather_facts: no
  vars:
    bmc_username: "{{ bmc_user | default('admin') }}"
    bmc_password: "{{ bmc_pass | default('password') }}"
```



```
    pxe_interface: "NIC.Integrated.1-1-1"  # Replace with your interface ID (e.g., NIC1, eth0,
    pxe_server_ip: "10.123.232.180"        # Desired PXE server IP
    boot_mode: "Uefi"                      # Options: Bios or Uefi (adjust as needed)

  tasks:
    - name: Ensure BMC is accessible via Redfish
      community.general.redfish_info:
        baseuri: "{{ inventory_hostname }}"
        username: "{{ bmc_username }}"
        password: "{{ bmc_password }}"
        timeout: 30
      register: redfish_info
      failed_when: redfish_info.failed

    - name: Set boot mode to UEFI (or BIOS) if required
      community.general.redfish_command:
        category: Systems
        command: SetBootMode
        boot_mode: "{{ boot_mode }}"
        baseuri: "{{ inventory_hostname }}"
        username: "{{ bmc_username }}"
        password: "{{ bmc_password }}"
      when: redfish_info.redfish_facts.systems[0].Boot.BootSourceOverrideMode != boot_mode

    - name: Configure PXE boot on specified interface
      community.general.redfish_command:
        category: Systems
        command: SetBootSource
        boot_source: Pxe
        boot_source_target: "{{ pxe_interface }}"
        baseuri: "{{ inventory_hostname }}"
        username: "{{ bmc_username }}"
        password: "{{ bmc_password }}"
      register: pxe_boot_result

    - name: Verify PXE boot configuration
      community.general.redfish_info:
        baseuri: "{{ inventory_hostname }}"
        username: "{{ bmc_username }}"
        password: "{{ bmc_password }}"
        category: Systems
      register: boot_config
      failed_when: "'Pxe' not in boot_config.redfish_facts.systems[0].Boot.BootSourceOverrideTa

    - name: Attempt to set PXE server IP (vendor-specific, may not be supported)
      community.general.redfish_config:
        category: Systems
        command: Patch
        resource: /redfish/v1/Systems/System.Embedded.1
        patch:
          Boot:
            BootSourceOverrideTarget: "Pxe"
```

```yaml
          BootSourceOverrideEnabled: "Once"  # Options: Once, Continuous
          BootSourceOverrideMode: "{{ boot_mode }}"
          # Vendor-specific attempt to set PXE server (may require custom attributes)
          Oem:
            VendorName:  # Replace with your vendor (e.g., Dell, HPE)
              PxeServer: "{{ pxe_server_ip }}"
        baseuri: "{{ inventory_hostname }}"
        username: "{{ bmc_username }}"
        password: "{{ bmc_password }}"
      ignore_errors: yes  # Ignore if PXE server IP is not supported
      when: pxe_boot_result.changed

    - name: Reboot system to apply PXE boot settings
      community.general.redfish_command:
        category: Systems
        command: Reset
        reset_type: ForceRestart
        baseuri: "{{ inventory_hostname }}"
        username: "{{ bmc_username }}"
        password: "{{ bmc_password }}"
      when: pxe_boot_result.changed

    - name: Display configuration result
      ansible.builtin.debug:
        msg: "PXE boot enabled on interface {{ pxe_interface }} for BMC {{ inventory_hostname }
```

Inventory File (`inventory.yml`)

```yaml
---
bmc_servers:
  hosts:
    bmc1:
      ansible_host: 192.168.1.100  # Replace with your BMC IP
      bmc_user: admin              # Replace with your BMC username
      bmc_pass: your_password      # Replace with your BMC password
```

How to Run

- **Install Dependencies**:

```bash
ansible-galaxy collection install community.general
```

- **Update Inventory**:
  - Modify `inventory.yml` with your BMC's IP address, username, and password.
  - Ensure the BMC is reachable over the network.
- **Customize Variables**:
  - In `enable_pxe_boot.yml`, update:
    - `pxe_interface`: Replace with the correct interface ID for your hardware (e.g., `NIC.Integrated.1-1-1` for Dell iDRAC or `NIC1` for HPE iLO). Check your BMC's Redfish API or documentation for valid interface IDs.
    - `boot_mode`: Set to `Uefi` or `Bios` based on your system requirements.
    - `VendorName` in the PXE server IP task: Replace with your BMC vendor (e.g., `Dell`, `HPE`). This is vendor-specific and may not be supported; consult your BMC's Redfish documentation.
- **Run the Playbook**:

```bash
ansible-playbook -i inventory.yml enable_pxe_boot.yml
```

Explanation of Key Tasks

- **Check BMC Accessibility**:
  - Uses `redfish_info` to verify the BMC is reachable and supports Redfish.
- **Set Boot Mode**:
  - Ensures the system is in the correct boot mode (UEFI or BIOS) for PXE.
- **Configure PXE Boot**:
  - Sets the boot source to PXE on the specified interface using `redfish_command`.
- **Verify Configuration**:
  - Confirms that PXE is set as the boot source.
- **Set PXE Server IP (Optional)**:
  - Attempts to configure the PXE server IP (10.123.232.180). This is vendor-specific and may not work on all BMCs. The task is set to `ignore_errors: yes` to avoid failing if unsupported.
- **Reboot System**:
  - Initiates a system reboot to apply the PXE boot settings.

Limitations and Workarounds

- **PXE Server IP**: Most BMCs do not allow specifying a PXE server IP directly via Redfish. Instead, the system will attempt to PXE boot from the first DHCP server it finds on the network. To ensure the system boots from 10.123.232.180:

- Configure your network's DHCP server to direct PXE requests to 10.123.232.180.
- Alternatively, check if your BMC supports vendor-specific Redfish extensions for setting a PXE server (e.g., Dell's iDRAC or HPE's iLO may have custom Oem attributes).

- **Interface ID**: The `pxe_interface` value must match your hardware. You can query available interfaces using:

```bash
ansible -m community.general.redfish_info -a "baseuri=192.168.1.100 username=admin passwor
```

Look for `BootSources` or `NetworkInterfaces` in the output.

- **Non-Redfish BMCs**: If your BMC doesn't support Redfish, you may need to use IPMI (`ipmitool`) or vendor-specific tools (e.g., `racadm` for Dell, `hponcfg` for HPE). Let me know your BMC type for an alternative script.

Troubleshooting

- **Authentication Errors**: Ensure `bmc_user` and `bmc_pass` are correct in the inventory.
- **Interface Not Found**: Verify the `pxe_interface` value using Redfish API or your BMC's web interface.
- **PXE Boot Fails**: Check network configuration (DHCP, TFTP) and ensure the PXE server at 10.123.232.180 is operational.
- **Vendor-Specific Issues**: If the PXE server IP task fails, consult your BMC's Redfish documentation for Oem attributes.

If you provide more details (e.g., BMC vendor, model, or specific interface name), I can refine the script further. Let me know if you need assistance with testing or debugging!

> I have a harvester cluster up and running and accessible under ip address 10.123.235.200/22. I want to deploy a 3 node kubernetes cluster running on virtual machines that uses hardware network interface from host in a pass through or a sriov virtual interface. I need a best known method to do it in an automated way, for example using ansible.

Deploying a 3-node Kubernetes cluster on virtual machines (VMs) within a Harvester cluster, utilizing hardware network interface pass-through or SR-IOV (Single Root I/O Virtualization) for high-performance networking, can be automated using Ansible. Harvester, being a Kubernetes-based hyperconverged infrastructure (HCI) solution, integrates seamlessly with Kubernetes APIs and supports VM management via KubeVirt, making Ansible a natural choice for automation. Below is a best-known method to achieve this, leveraging Ansible for automation, Harvester for VM management, and SR-IOV for network interface virtualization.

Overview and Best-Known Method

The goal is to:

- Create three VMs on the Harvester cluster (accessible at 10.123.235.200/22) to form a Kubernetes cluster (1 control plane, 2 worker nodes).
- Configure the VMs to use SR-IOV virtual functions (VFs) for high-performance networking, as SR-IOV is preferred over direct device pass-through for scalability and flexibility in virtualized environments.
- Automate the deployment using Ansible, including VM provisioning, SR-IOV configuration, and Kubernetes setup with a container runtime (e.g., containerd).
- Ensure the process is repeatable and idempotent for production-grade deployments.

Why SR-IOV?

- SR-IOV allows a single physical network interface card (NIC) to present multiple virtual functions (VFs) that can be directly assigned to VMs, offering near-native network performance.
- Compared to direct device pass-through, SR-IOV supports multiple VMs sharing a physical NIC, is more scalable, and aligns with Harvester's cloud-native architecture.
- Direct pass-through dedicates an entire NIC to a single VM, which is less flexible and may not be practical for a multi-node cluster unless you have multiple dedicated NICs.

Tools and Components:

- **Harvester**: Manages VMs and provides SR-IOV support via Kubernetes and KubeVirt.
- **Ansible**: Automates VM creation, SR-IOV configuration, and Kubernetes deployment.
- **Kubeadm**: Initializes the Kubernetes cluster.
- **Containerd**: Lightweight container runtime for Kubernetes.
- **Multus CNI**: Enables SR-IOV network interfaces in Kubernetes VMs.
- **Calico or Cilium**: CNI plugin for Kubernetes pod networking.

Prerequisites

- **Harvester Cluster**:
  - Accessible at 10.123.235.200/22 (VIP or node IP).
  - Harvester v1.2.0 or later (SR-IOV support improved in recent versions).
  - SR-IOV-capable NICs installed on Harvester nodes (e.g., Intel X710, Mellanox ConnectX-5).

  - SR-IOV enabled in the BIOS and OS of Harvester nodes (check with `lspci` and `dmesg | grep -i sriov`).
  - Harvester nodes configured with a management network and a VLAN-enabled network for SR-IOV VFs.
- **Ansible Control Node**:
  - Ansible 2.9 or later with `kubernetes.core` and `community.general` collections installed:

```bash
ansible-galaxy collection install kubernetes.core community.general
```

  - Access to Harvester's Kubernetes API (kubeconfig file or API token).
  - SSH key pair for passwordless access to VMs.
- **Network Configuration**:
  - SR-IOV VFs configured on Harvester nodes (see Harvester documentation for SR-IOV setup).
  - A VLAN (e.g., VLAN ID 100) for SR-IOV traffic, routable within 10.123.235.200/22.
  - DHCP or static IP configuration for VMs' SR-IOV interfaces.
- **VM Image**:
  - A VM image (e.g., Ubuntu 22.04 Cloud Image) uploaded to Harvester's image storage (via Harvester UI or API).
  - Image URL or name available for Ansible.
- **Hardware Requirements** (per VM):
  - 2 vCPUs, 4GB RAM, 20GB disk (minimum for testing; adjust for production).
  - SR-IOV VF assigned to each VM.

Ansible Playbook Structure

The deployment is split into several playbooks for modularity:

- **Configure SR-IOV on Harvester**: Ensure SR-IOV VFs are available and configured.
- **Create VMs**: Provision three VMs with SR-IOV VFs attached.
- **Setup Kubernetes**: Install containerd, kubeadm, and configure the cluster.
- **Apply CNI**: Deploy a CNI plugin (e.g., Calico) for pod networking.

Inventory File (`inventory.yml`)

```yaml
---
all:
  hosts:
    harvester:
      ansible_host: 10.123.235.200
      ansible_user: kubeadmin  # Replace with your Harvester admin user
      ansible_ssh_private_key_file: ~/.ssh/id_rsa
  vars:
    harvester_api_token: "your_api_token"  # Obtain from Harvester UI
    vm_image_name: "ubuntu-22.04-cloudimg"
    vm_namespace: "default"
    sriov_network_name: "sriov-vlan100"  # Name of SR-IOV network in Harvester
    vlan_id: 100
    kubernetes_version: "1.28.2"
    pod_network_cidr: "10.244.0.0/16"
```

Playbook 1: Configure SR-IOV (`configure_sriov.yml`)

This playbook ensures SR-IOV is enabled and VFs are available on Harvester nodes. It also creates a NetworkAttachmentDefinition for SR-IOV.

```yaml
---
- name: Configure SR-IOV on Harvester
  hosts: harvester
  gather_facts: yes
  tasks:
    - name: Check SR-IOV support on nodes
      ansible.builtin.shell: |
        lspci | grep -i ethernet && dmesg | grep -i sriov
      register: sriov_check
      changed_when: false

    - name: Enable SR-IOV VFs on NIC
      ansible.builtin.shell: |
        echo 4 > /sys/class/net/ens1f0/device/sriov_numvfs
      vars:
        nic: "ens1f0"  # Replace with your NIC name (check with `ip link`)
      when: "'SR-IOV' in sriov_check.stdout"
      register: sriov_vfs
      changed_when: sriov_vfs.rc == 0

    - name: Create SR-IOV NetworkAttachmentDefinition
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: "k8s.cni.cncf.io/v1"
          kind: NetworkAttachmentDefinition
          metadata:
            name: "{{ sriov_network_name }}"
            namespace: "{{ vm_namespace }}"
          spec:
            config: |
```

```
              {
                "cniVersion": "0.3.1",
                "name": "{{ sriov_network_name }}",
                "type": "sriov",
                "vlan": {{ vlan_id }},
                "ipam": {
                  "type": "host-local",
                  "subnet": "10.123.235.0/22",
                  "rangeStart": "10.123.235.100",
                  "rangeEnd": "10.123.235.150"
                }
              }
        kubeconfig: "{{ harvester_kubeconfig | default('/etc/rancher/rke2/rke2.yaml') }}"
      when: sriov_vfs.changed
```

Playbook 2: Create VMs (`create_vms.yml`)
This playbook provisions three VMs with SR-IOV VFs attached.

```
---
- name: Create VMs for Kubernetes cluster
  hosts: harvester
  gather_facts: no
  vars:
    vm_roles:
      - name: k8s-master
        role: control-plane
      - name: k8s-worker1
        role: worker
      - name: k8s-worker2
        role: worker
  tasks:
    - name: Create VMs with SR-IOV interfaces
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: kubevirt.io/v1
          kind: VirtualMachine
          metadata:
            name: "{{ item.name }}"
            namespace: "{{ vm_namespace }}"
          spec:
            running: true
            template:
              metadata:
                labels:
                  role: "{{ item.role }}"
              spec:
                domain:
                  cpu:
```

Grok 3

```
                    cores: 2
                  memory:
                    guest: 4Gi
                  devices:
                    disks:
                      - name: rootdisk
                        disk:
                          bus: virtio
                      - name: cloudinitdisk
                        disk:
                          bus: virtio
                    interfaces:
                      - name: sriov-net
                        sriov: {}
                networks:
                  - name: sriov-net
                    multus:
                      networkName: "{{ sriov_network_name }}"
                volumes:
                  - name: rootdisk
                    dataVolume:
                      name: "{{ item.name }}-root"
                  - name: cloudinitdisk
                    cloudInitNoCloud:
                      userData: |
                        #cloud-config
                        hostname: "{{ item.name }}"
                        users:
                          - name: ubuntu
                            ssh-authorized-keys:
                              - "{{ lookup('file', '~/.ssh/id_rsa.pub') }}"
                            sudo: ALL=(ALL) NOPASSWD:ALL
                        packages:
                          - curl
                          - containerd
                        runcmd:
                          - [ "systemctl", "enable", "--now", "containerd" ]
            dataVolumeTemplates:
              - metadata:
                  name: "{{ item.name }}-root"
                spec:
                  pvc:
                    accessModes:
                      - ReadWriteOnce
                    resources:
                      requests:
                        storage: 20Gi
                  source:
                    http:
                      url: "https://cloud-images.ubuntu.com/releases/22.04/release/ubuntu-22.04
        kubeconfig: "{{ harvester_kubeconfig | default('/etc/rancher/rke2/rke2.yaml') }}"
      loop: "{{ vm_roles }}"
```

```
    register: vm_creation

  - name: Wait for VMs to get IP addresses
    kubernetes.core.k8s_info:
      kind: VirtualMachineInstance
      namespace: "{{ vm_namespace }}"
      name: "{{ item.name }}"
      kubeconfig: "{{ harvester_kubeconfig }}"
    register: vmi_info
    until: "vmi_info.resources[0].status.interfaces[0].ipAddress is defined"
    retries: 30
    delay: 10
    loop: "{{ vm_roles }}"

  - name: Create Ansible inventory for Kubernetes nodes
    ansible.builtin.copy:
      content: |
        [k8s_cluster:children]
        master
        worker

        [master]
        {{ vmi_info.results[0].resources[0].status.interfaces[0].ipAddress }} ansible_user=ub

        [worker]
        {{ vmi_info.results[1].resources[0].status.interfaces[0].ipAddress }} ansible_user=ub
        {{ vmi_info.results[2].resources[0].status.interfaces[0].ipAddress }} ansible_user=ub
      dest: k8s_inventory.yml
```

Playbook 3: Setup Kubernetes (`setup_kubernetes.yml`)
This playbook installs Kubernetes using kubeadm and configures the cluster.

```
---
- name: Setup Kubernetes cluster
  hosts: k8s_cluster
  become: yes
  tasks:
    - name: Install Kubernetes packages
      ansible.builtin.apt:
        name:
          - kubelet={{ kubernetes_version }}-00
          - kubeadm={{ kubernetes_version }}-00
          - kubectl={{ kubernetes_version }}-00
        state: present
        update_cache: yes

    - name: Hold Kubernetes packages
      ansible.builtin.dpkg_selections:
        name: "{{ item }}"
        selection: hold
      loop:
        - kubelet
        - kubeadm
        - kubectl

    - name: Initialize control plane
      ansible.builtin.command:
        cmd: |
          kubeadm init --pod-network-cidr={{ pod_network_cidr }} --apiserver-advertise-address=
      args:
        creates: /etc/kubernetes/admin.conf
      when: "'master' in group_names"
      register: kubeadm_init

    - name: Setup kubeconfig for ubuntu user
      ansible.builtin.command: "{{ item }}"
      with_items:
        - mkdir -p /home/ubuntu/.kube
        - cp -i /etc/kubernetes/admin.conf /home/ubuntu/.kube/config
        - chown ubuntu:ubuntu /home/ubuntu/.kube/config
      when: "'master' in group_names"

    - name: Save join command
      ansible.builtin.copy:
        content: "{{ kubeadm_init.stdout | regex_search('kubeadm join.*--token.*') }}"
        dest: /tmp/join-command.sh
        mode: '0777'
      when: "'master' in group_names"

    - name: Copy join command to workers
      ansible.builtin.copy:
        src: /tmp/join-command.sh
        dest: /tmp/join-command.sh
        mode: '0777'
      when: "'worker' in group_names"

    - name: Join worker nodes to cluster
      ansible.builtin.command:
        cmd: sh /tmp/join-command.sh
      when: "'worker' in group_names"

    - name: Install Calico CNI
      ansible.builtin.command:
        cmd: kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
      when: "'master' in group_names"
      environment:
        KUBECONFIG: /home/ubuntu/.kube/config
```

Main Playbook (`main.yml`)
This orchestrates all steps.

Grok 3

```yaml
---
- name: Deploy 3-node Kubernetes cluster on Harvester with SR-IOV
  hosts: all
  tasks:
    - name: Configure SR-IOV
      import_playbook: configure_sriov.yml

    - name: Create VMs
      import_playbook: create_vms.yml

    - name: Setup Kubernetes
      import_playbook: setup_kubernetes.yml
```

Execution Steps

- **Prepare Ansible Control Node**:
  - Ensure SSH keys are set up and the public key is included in the cloud-init user data.
  - Place the kubeconfig file for Harvester at `/etc/rancher/rke2/rke2.yaml` or update the `harvester_kubeconfig` variable.
- **Update Inventory and Variables**:
  - In `inventory.yml`, set the correct Harvester IP, API token, and SR-IOV network details.
  - Verify the NIC name (`ens1f0`) and VLAN ID (`100`) match your setup.
  - Adjust VM resources (CPU, RAM, disk) as needed.
- **Run the Playbook**:

```bash
ansible-playbook -i inventory.yml main.yml
```

- **Verify the Cluster**:
  - SSH into the control plane VM and check the cluster status:

```bash
kubectl get nodes
```

  - Ensure all three nodes are in the `Ready` state.

Best Practices and Optimizations

- **Idempotency**: The playbooks use `creates` and `when` clauses to ensure idempotent operations where possible.

Grok 3

- **Security**:
  - Use Ansible Vault to encrypt sensitive data (e.g., API tokens, SSH keys).
  - Restrict SR-IOV VLAN traffic to specific subnets for security.
- **High Availability**: For production, consider deploying multiple control plane nodes (requires additional VMs and HAProxy/Keepalived setup).
- **Monitoring**: Integrate Prometheus and Grafana (available in Harvester) to monitor VM and cluster health.
- **Backup**: Use Longhorn (Harvester's default storage) to snapshot VM volumes for backup.
- **Networking**:
  - Use Cilium instead of Calico for advanced networking features (e.g., eBPF-based performance).
  - Ensure your DHCP server assigns stable IPs to SR-IOV interfaces or configure static IPs via cloud-init.

Troubleshooting

- **SR-IOV Not Working**:
  - Verify SR-IOV is enabled in the NIC driver (`cat /sys/class/net/ens1f0/device/sriov_numvfs`).
  - Check Harvester's NetworkAttachmentDefinition (`kubectl get net-attach-def -n default`).
- **VMs Not Getting IPs**:
  - Ensure the SR-IOV network is correctly mapped to the VLAN and DHCP is available.
  - Check VMI status: `kubectl get vmi -n default`.
- **Kubernetes Cluster Issues**:
  - Verify containerd is running (`systemctl status containerd`).
  - Check kubeadm logs on the control plane (`journalctl -u kubelet`).
- **Harvester API Errors**:
  - Validate the API token and kubeconfig file.
  - Ensure the Harvester VIP (10.123.235.200) is reachable.

References and Sources

- Harvester SR-IOV documentation: https://docs.harvesterhci.io/v1.2/networking/sriov

- Ansible `kubernetes.core` module: https://docs.ansible.com/ansible/latest/collections/kubernetes/core/k8s_module.html (https://docs.ansible.com/ansible/latest/collections/kubernetes/core/k8s_module.html)
- Kubeadm deployment guide: https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/[](https://medium.com/%40fenari.kostem/effortless-kubernetes-deployment-setting-up-a-cluster-with-ansible-and-kubeadm-cc40f9e716f4)
- Multus CNI for SR-IOV: https://github.com/k8snetworkplumbingwg/multus-cni

Notes

- **Direct Pass-Through Alternative**: If SR-IOV is not supported, you can use PCI device pass-through by modifying the VM spec to include `devices: [{pciDevice: {pciAddress: "0000:01:00.0"}}]`. However, this requires dedicated NICs per VM and is less flexible.
- **Harvester-Specific Considerations**: Harvester's KubeVirt may have limitations with SR-IOV (e.g., issue #8218 for device passthrough with Pod Security Standards). Check the latest Harvester release notes for fixes.
- **Automation Enhancements**: For larger deployments, integrate with Rancher for centralized management of Harvester and Kubernetes clusters.

15 web pages

c issues or need further customization (e.g., specific NIC models, VLAN configurations, or Kubernetes versions), please provide details, and I can tailor the solution further!

Describe to me what Portainer is (https://www.portainer.io) and how can I use it with my Kubernetes cluster deployed using Harvester knowing that I am focusing on provisioning ephemeral Kubernetes clusters to test Intel ICE drivers on top of witch MTL and MCM from https://www.github.com/OpenVisualCloud are being run.

What is Portainer?

Portainer is an open-source, web-based management platform designed to simplify the deployment, configuration, and management of containerized workloads across Docker, Kubernetes, and other container orchestration platforms. It provides a user-friendly graphical interface that abstracts much of the complexity of managing containerized environments, making it accessible to both developers and IT administrators. Key features include:

- **Multi-Cluster Management**: Manage multiple Kubernetes clusters (or Docker environments) from a single interface.
- **Application Deployment**: Deploy, update, and monitor applications using templates, Helm charts, or Kubernetes manifests.
- **Resource Management**: Control nodes, pods, services, and storage with an intuitive dashboard.
- **Security and Access Control**: Role-based access control (RBAC) and integration with Kubernetes RBAC for secure operations.
- **Monitoring and Logging**: View resource usage, logs, and metrics for troubleshooting and performance analysis.
- **Ease of Use**: Simplifies Kubernetes operations for users unfamiliar with `kubectl` or complex YAML configurations.

Portainer is particularly valuable for managing ephemeral Kubernetes clusters, as it streamlines cluster provisioning, application deployment, and cleanup, which aligns with your use case of testing Intel ICE (In-network Compute Engine) drivers with MTL (Media Transport Library) and MCM (Media Cloud Manager) from OpenVisualCloud.

Using Portainer with a Harvester-Deployed Kubernetes Cluster

To integrate Portainer with your 3-node Kubernetes cluster deployed on Harvester (as described in your previous request), and focus on provisioning ephemeral Kubernetes clusters for testing Intel ICE drivers with OpenVisualCloud's MTL and MCP, follow these steps. This approach leverages Harvester's hyperconverged infrastructure (HCI) and Portainer's management capabilities to create and manage short-lived clusters efficiently.

Prerequisites

- **Harvester Cluster**:
  - Running at IP 10.123.235.200/22 with Harvester v1.2.0 or later.
  - Three VMs configured with SR-IOV virtual functions (VFs) for high-performance networking, as per your previous setup.
  - Kubernetes cluster (1 control plane, 2 workers) deployed using `kubeadm`, running Kubernetes v1.28.2 or compatible.
  - SR-IOV-enabled NICs (e.g., Intel X710) configured with VFs assigned to VMs.
- **Portainer Requirements**:
  - A server or VM to host Portainer (can be outside the Harvester cluster or on a Harvester VM).
  - Docker or Kubernetes environment to deploy Portainer (Kubernetes is preferred for this use case).
  - Access to the Harvester cluster's kubeconfig file for Portainer to manage it.
- **OpenVisualCloud Components**:
  - Familiarity with Intel ICE drivers, MTL, and MCM from OpenVisualCloud.

    - MTL and MCM require SR-IOV or direct NIC access for optimal performance, which your Harvester VMs already support via SR-IOV VFs.
    - Relevant container images for MTL and MCM (available from OpenVisualCloud's Docker Hub or built from source).

- **Network and Access**:
    - Harvester VMs have routable IPs (e.g., 10.123.235.100–150) via SR-IOV interfaces.
    - Portainer server has network access to the Harvester cluster's API (10.123.235.200:6443).
    - SSH access to VMs for initial setup and debugging.
- **Ansible Automation**:
    - Your existing Ansible playbooks (from the previous response) for VM and Kubernetes provisioning.
    - Extend Ansible to automate Portainer deployment and configuration.

Step-by-Step Integration
1. Deploy Portainer on a Kubernetes Cluster
Portainer can be deployed on the Harvester Kubernetes cluster or a separate Kubernetes environment. For simplicity, deploy it on a dedicated VM outside the Harvester cluster to avoid resource contention during ephemeral cluster testing.

a. Prepare a Portainer Host:

- Create a VM on Harvester or use an external server (e.g., Ubuntu 22.04, 2 vCPUs, 4GB RAM, 20GB disk).
- Install Docker:

```bash
sudo apt update && sudo apt install -y docker.io
sudo systemctl enable --now docker
```
bash

b. Deploy Portainer CE (Community Edition):

- Use the Portainer Helm chart for Kubernetes deployment. First, install Helm:

```bash
curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
```
bash

- Add the Portainer Helm repository:

```bash
helm repo add portainer https://portainer-charts.public.portainer.io
helm repo update
```
bash

Grok 3

- Create a namespace and deploy Portainer:

```bash
kubectl create namespace portainer
helm install portainer portainer/portainer \
  --namespace portainer \
  --set service.type=LoadBalancer \
  --set enterpriseEdition.enabled=false
```
bash

- Retrieve the LoadBalancer IP or NodePort:

```bash
kubectl get svc -n portainer
```
bash

- Access Portainer at `http://<loadbalancer-ip>:9000` or `http://<node-ip>:<nodeport>`. Set up an admin user and password.

Ansible Automation:
Extend your Ansible playbook to automate Portainer deployment:

```yaml
- name: Deploy Portainer on Kubernetes
  hosts: portainer_host  # Define in inventory
  become: yes
  tasks:
    - name: Install Docker
      ansible.builtin.apt:
        name: docker.io
        state: present
        update_cache: yes

    - name: Install Helm
      ansible.builtin.shell: |
        curl https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 | bash
      args:
        creates: /usr/local/bin/helm

    - name: Add Portainer Helm repo
      kubernetes.core.helm_repository:
        name: portainer
        repo_url: https://portainer-charts.public.portainer.io

    - name: Deploy Portainer
      kubernetes.core.helm:
        name: portainer
        chart_ref: portainer/portainer
        namespace: portainer
        create_namespace: true
```

Grok 3

```yaml
    values:
      service:
        type: LoadBalancer
      enterpriseEdition:
        enabled: false
    kubeconfig: "{{ harvester_kubeconfig | default('/etc/rancher/rke2/rke2.yaml') }}"
```

2. Connect Portainer to the Harvester Kubernetes Cluster

- **Obtain Kubeconfig**:
  - Copy the kubeconfig file from the Harvester control plane VM (e.g., `/home/ubuntu/.kube/config`) to your Portainer host or Ansible control node.

yaml

Alternatively, generate a service account token for Portainer:

```bash
kubectl create sa portainer -n kube-system
kubectl create clusterrolebinding portainer --clusterrole=cluster-admin --serviceaccou
TOKEN=$(kubectl create token portainer -n kube-system)
```

bash

Create a kubeconfig file with the token and Harvester API endpoint (10.123.235.200:6443).

- **Add Cluster to Portainer**:
  - Log in to the Portainer UI.
  - Navigate to **Environments** > **Add Environment** > **Kubernetes**.
  - Upload the kubeconfig file or enter the API server URL (https://10.123.235.200:6443) and token.
  - Name the environment (e.g., "Harvester-K8s") and save.
- **Verify Connection**:
  - In Portainer, go to **Clusters** and confirm the Harvester cluster appears with its nodes, namespaces, and resources.

Ansible Automation:
Automate kubeconfig setup and environment addition:

```yaml
- name: Configure Portainer to manage Harvester cluster
  hosts: portainer_host
  tasks:
    - name: Copy kubeconfig to Portainer host
      ansible.builtin.copy:
        src: /path/to/harvester/kubeconfig
        dest: /home/ubuntu/harvester-kubeconfig
        mode: '0600'
```

Grok 3

```yaml
  - name: Add Harvester cluster to Portainer
    ansible.builtin.uri:
      url: http://{{ portainer_host_ip }}:9000/api/endpoints
      method: POST
      headers:
        Authorization: "Bearer {{ portainer_api_token }}"  # Obtain from Portainer UI
      body_format: json
      body:
        Name: "Harvester-K8s"
        EndpointType: 4  # Kubernetes
        URL: "https://10.123.235.200:6443"
        Kubeconfig: "{{ lookup('file', '/home/ubuntu/harvester-kubeconfig') | b64encode }}"
      status_code: 200
    register: endpoint_result
```

yaml

3. Provision Ephemeral Kubernetes Clusters

To test Intel ICE drivers with MTL and MCM, create ephemeral Kubernetes clusters within Harvester VMs. Portainer simplifies this by allowing you to manage VM workloads and deploy test applications.

a. Create Ephemeral VMs:

- Use your existing Ansible playbook (`create_vms.yml`) to provision additional VMs for ephemeral clusters.
- Modify the playbook to create smaller, short-lived VMs (e.g., 1 vCPU, 2GB RAM) with SR-IOV VFs for ICE driver testing.
- Add a cloud-init script to install Intel ICE drivers and OpenVisualCloud components:

```yaml
userData: |
  #cloud-config
  packages:
    - curl
    - containerd
  runcmd:
    - curl -fsSL https://download.01.org/intel/ice/<version>/ice-<version>.tar.gz | tar -x
    - cd ice-<version> && ./configure && make && make install
    - modprobe ice
    - curl -fsSL https://github.com/OpenVisualCloud/Media-Transport-Library/archive/refs/t
    - cd Media-Transport-Library-<mtl-version> && ./build.sh
    - curl -fsSL https://github.com/OpenVisualCloud/Media-Cloud-Manager/archive/refs/tags/
    - cd Media-Cloud-Manager-<mcm-version> && ./deploy.sh
```

- Deploy VMs via Portainer:

yaml

- In Portainer, go to **Namespaces** > **default** > **Virtual Machines** (Harvester integrates KubeVirt VMs).
- Import the VM YAML from your Ansible playbook or use Portainer's UI to create VMs with SR-IOV interfaces.

b. Deploy Ephemeral Kubernetes Clusters:

- Use Portainer to deploy a new Kubernetes cluster within the VMs:
  - Create a new namespace (e.g., `ephemeral-test`) in Portainer.
  - Deploy a `kubeadm`-based cluster by applying manifests for control plane and worker nodes, reusing your `setup_kubernetes.yml` playbook.
  - Alternatively, use Portainer's **Application Templates** to deploy a lightweight Kubernetes distribution like k3s for faster testing:

```bash
kubectl apply -f https://github.com/k3s-io/k3s/releases/download/v1.28.2+k3s1/k3s.yaml
```

- Configure the cluster to use SR-IOV CNI (e.g., Multus) for ICE driver compatibility:
  - Apply a NetworkAttachmentDefinition (as in `configure_sriov.yml`).
  - Deploy MTL and MCM containers with SR-IOV interfaces:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mtl-test
  namespace: ephemeral-test
  annotations:
    k8s.v1.cni.cncf.io/networks: sriov-vlan100
spec:
  containers:
  - name: mtl
    image: openvisualcloud/mtl:latest
    resources:
      limits:
        cpu: "2"
        memory: "4Gi"
    securityContext:
      privileged: true  # Required for ICE driver access
```

c. Manage Ephemeral Clusters in Portainer:

- Monitor cluster health, logs, and resource usage in Portainer's **Dashboard**.

Grok 3

- Deploy test workloads (e.g., MTL/MCM media streaming) via **Applications** > **Add Application** > **Custom Template**.
- Destroy clusters after testing:
  - In Portainer, delete the VMs or namespace to clean up resources.
  - Use Ansible to automate cleanup:

```yaml
- name: Delete ephemeral VMs
  kubernetes.core.k8s:
    state: absent
    kind: VirtualMachine
    namespace: "{{ vm_namespace }}"
    name: "{{ item.name }}"
    kubeconfig: "{{ harvester_kubeconfig }}"
  loop: "{{ ephemeral_vms }}"
```

4. Test Intel ICE Drivers with MTL and MCM

- **Validate ICE Drivers**:
  - SSH into the ephemeral VMs and verify ICE driver loading:

```bash
lsmod | grep ice
dmesg | grep -i ice
```

  - Test SR-IOV VF connectivity:

```bash
ip link show
ping 10.123.235.200
```

- **Run MTL/MCM Workloads**:
  - Deploy MTL and MCM containers via Portainer, ensuring they use SR-IOV interfaces.
  - Test media streaming or transcoding workloads as per OpenVisualCloud documentation:

```bash
docker run --network host openvisualcloud/mtl:latest <test-command>
```

  - Monitor performance in Portainer's **Metrics** tab or use `kubectl top` for pod resource usage.
- **Iterate and Destroy**:
  - After testing, use Portainer to inspect logs and metrics for analysis.

- Delete the ephemeral cluster and VMs to free resources for the next test cycle.

Grok 3

Best Practices for Ephemeral Clusters

- **Automation with Ansible**:
  - Extend your playbooks to create, test, and destroy clusters in a single workflow.
  - Use variables to parameterize VM counts, ICE driver versions, and MTL/MCM tags.
- **Resource Optimization**:
  - Use lightweight VMs and k3s for faster provisioning and reduced overhead.
  - Limit SR-IOV VFs to avoid oversubscription (e.g., 4 VFs per NIC, as in `configure_sriov.yml`).
- **Security**:
  - Apply Kubernetes Pod Security Standards (PSS) in Harvester to restrict privileged containers, as noted in the Harvester documentation.
  - Use Portainer's RBAC to limit access to ephemeral clusters.
- **Monitoring**:
  - Integrate Prometheus (available in Harvester) with Portainer for detailed metrics on ICE driver performance and MTL/MCM workloads.
  - Enable Portainer's logging to capture test results.
- **Cleanup**:
  - Schedule automatic VM deletion after a set period (e.g., 1 hour) using Ansible or Harvester's VM lifecycle policies.
  - Monitor Harvester's storage (Longhorn) to ensure ephemeral volumes are reclaimed.

Challenges and Workarounds

- **SR-IOV and ICE Driver Compatibility**:
  - Ensure the Intel ICE driver version matches your NIC (e.g., E810 series for X710). Check OpenVisualCloud's Media-Transport-Library for supported drivers.
  - If VFs fail to initialize, verify SR-IOV settings in the Harvester node's BIOS and kernel (`dmesg | grep -i sriov`).
- **Ephemeral Cluster Stability**:
  - Ephemeral clusters may face issues with node reboots due to Harvester's ephemeral `/etc` (noted in k3os-based systems). Use persistent storage for critical configurations.
  - Enable Kubernetes feature gates for ephemeral volumes if needed (`--feature-gates=GenericEphemeralVolume=true`).
- **Portainer Limitations**:
  - Portainer CE has limited support for advanced Kubernetes features (e.g., custom CRDs). For complex MTL/MCM deployments, combine Portainer with `kubectl` or Helm.

Grok 3

  - If Portainer's UI lacks SR-IOV-specific controls, manage NetworkAttachmentDefinitions via Ansible or `kubectl`.
- **MTL/MCM Deployment**:
  - MTL and MCM require privileged containers and direct NIC access, which may conflict with Harvester's PSS. Disable PSS for the test namespace if needed:

```bash
kubectl label namespace ephemeral-test pod-security.kubernetes.io/enforce=privileged
```

Sample Workflow for Testing

- **Provision Cluster**:
  - Run `ansible-playbook main.yml` to create VMs and a Kubernetes cluster with SR-IOV.
  - Add the cluster to Portainer.
- **Deploy Test Workload**:
  - Use Portainer to deploy an MTL/MCM pod with SR-IOV annotations.
  - Run a media streaming test (e.g., `ffmpeg` transcoding via MTL).
- **Monitor and Analyze**:
  - Check ICE driver performance in Portainer's metrics.
  - Collect logs via Portainer's **Logs** tab or `kubectl logs`.
- **Cleanup**:
  - Delete the namespace or VMs via Portainer or Ansible.
  - Verify resource reclamation in Harvester.

References

- Portainer Documentation: https://www.portainer.io[](https://www.portainer.io/kubernetes-management-platform)
- Harvester SR-IOV Guide: https://docs.harvesterhci.io/v1.2/networking/sriov[](https://docs.harvesterhci.io/v1.5/)
- OpenVisualCloud MTL/MCM: https://github.com/OpenVisualCloud
- Kubernetes Ephemeral Containers: https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/[](https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/)
- Harvester PSS: https://harvesterhci.io/kb/pod-security-standards[] ci.io/kb/)

15 web pages

If you need specific Ansible playbooks for MTL/MCM deployment, ICE driver testing, or Portainer configuration, or if you encounter issues with SR-IOV or Harvester, let me know, and I can provide tailored solutions!

Reiterate last prompt but instead Portainer use Argo.

Grok 3

What is Argo?

Argo is a suite of open-source tools for Kubernetes, primarily focused on automating workflows, continuous deployment (CD), and GitOps practices. Unlike Portainer, which provides a graphical user interface for managing containerized environments, Argo emphasizes declarative, Git-based automation for deploying and managing applications on Kubernetes. The main components relevant to your use case are:

- **Argo Workflows**: A container-native workflow engine for orchestrating complex, parallel jobs on Kubernetes, ideal for automating test pipelines (e.g., Intel ICE driver testing with MTL and MCM).
- **Argo CD**: A GitOps continuous delivery tool that synchronizes Kubernetes resources with Git repository manifests, enabling automated deployment of ephemeral clusters and applications.
- **Argo Events**: Triggers workflows based on events (e.g., Git commits or test completion), useful for automating the lifecycle of ephemeral clusters.

Argo's strengths include its programmatic, Git-driven approach, which aligns well with your goal of provisioning ephemeral Kubernetes clusters on a Harvester cluster to test Intel ICE drivers with OpenVisualCloud's MTL (Media Transport Library) and MCM (Media Cloud Manager). It's particularly suited for automation-heavy environments where repeatability and version control are critical.

Using Argo with a Harvester-Deployed Kubernetes Cluster

To integrate Argo with your 3-node Kubernetes cluster deployed on Harvester (at 10.123.235.200/22, with SR-IOV virtual functions for high-performance networking), and focus on provisioning ephemeral Kubernetes clusters for testing Intel ICE drivers with MTL and MCM, follow these steps. This approach leverages Harvester's KubeVirt for VM management, Argo for workflow and deployment automation, and Ansible for initial setup, ensuring a fully automated, GitOps-driven pipeline.

Prerequisites

- **Harvester Cluster**:
  - Running at IP 10.123.235.200/22 with Harvester v1.2.0 or later.
  - Three VMs (1 control plane, 2 workers) with SR-IOV VFs, running Kubernetes v1.28.2 via `kubeadm`, as per your previous setup.
  - SR-IOV-enabled NICs (e.g., Intel X710) with VFs assigned to VMs.
  - Access to Harvester's Kubernetes API (kubeconfig or API token).
- **Argo Requirements**:
  - Argo CD and Argo Workflows installed on the Harvester Kubernetes cluster or a separate management cluster.
  - A Git repository (e.g., GitHub, GitLab) to store manifests for ephemeral clusters, ICE drivers, and MTL/MCM applications.
  - Access to the Harvester cluster's kubeconfig for Argo to manage resources.
- **OpenVisualCloud Components**:
  - Familiarity with Intel ICE drivers, MTL, and MCM from OpenVisualCloud.
  - MTL and MCM container images (from Docker Hub or built from source) compatible with SR-IOV VFs.
  - Test scripts or workloads for ICE driver validation (e.g., media streaming or transcoding).
- **Network and Access**:
  - VMs have routable IPs (e.g., 10.123.235.100–150) via SR-IOV interfaces.
  - Git repository accessible from the Kubernetes cluster (public or private with credentials).
  - SSH access to VMs for debugging and initial setup.
- **Ansible Automation**:
  - Existing Ansible playbooks (from your previous request) for VM and Kubernetes provisioning.
  - Extend Ansible to deploy Argo and configure GitOps workflows.

Step-by-Step Integration
1. Deploy Argo on the Harvester Kubernetes Cluster
Argo CD and Argo Workflows will be installed on the Harvester cluster to manage ephemeral clusters and automate testing workflows.

a. Install Argo CD:

- Create a namespace and install Argo CD using its manifests:

```bash
kubectl create namespace argocd
kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/v2.12.3/mani
```

- Expose the Argo CD server (e.g., via LoadBalancer or NodePort):

```bash
kubectl patch svc argocd-server -n argocd -p '{"spec": {"type": "LoadBalancer"}}'
```

- Retrieve the LoadBalancer IP:

```bash
kubectl get svc -n argocd
```

- Get the initial admin password:

Grok 3

```bash
kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | 
```

bash

- Access the Argo CD UI at `https://<loadbalancer-ip>` and log in with `admin` and the password.

b. Install Argo Workflows:

- Install Argo Workflows in the same cluster:

```bash
kubectl create namespace argo
kubectl apply -n argo -f https://raw.githubusercontent.com/argoproj/argo-workflows/v3.5.8/
```

bash

- Expose the Argo Workflows server:

```bash
kubectl patch svc argo-server -n argo -p '{"spec": {"type": "LoadBalancer"}}'
```

bash

- Access the Argo Workflows UI at `https://<loadbalancer-ip>`.

Ansible Automation:
Automate Argo CD and Workflows deployment:

```yaml
- name: Deploy Argo CD and Argo Workflows
  hosts: harvester
  become: yes
  tasks:
    - name: Create Argo CD namespace
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: v1
          kind: Namespace
          metadata:
            name: argocd
        kubeconfig: "{{ harvester_kubeconfig | default('/etc/rancher/rke2/rke2.yaml') }}"

    - name: Install Argo CD
      kubernetes.core.k8s:
        state: present
        src: https://raw.githubusercontent.com/argoproj/argo-cd/v2.12.3/manifests/install.yaml
```

Grok 3

```yaml
        namespace: argocd
        kubeconfig: "{{ harvester_kubeconfig }}"

    - name: Expose Argo CD server
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: v1
          kind: Service
          metadata:
            name: argocd-server
            namespace: argocd
          spec:
            type: LoadBalancer
            selector:
              app.kubernetes.io/name: argocd-server
            ports:
              - port: 443
                targetPort: 8080
        kubeconfig: "{{ harvester_kubeconfig }}"

    - name: Create Argo Workflows namespace
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: v1
          kind: Namespace
          metadata:
            name: argo
        kubeconfig: "{{ harvester_kubeconfig }}"

    - name: Install Argo Workflows
      kubernetes.core.k8s:
        state: present
        src: https://raw.githubusercontent.com/argoproj/argo-workflows/v3.5.8/manifests/quick-s
        namespace: argo
        kubeconfig: "{{ harvester_kubeconfig }}"

    - name: Expose Argo Workflows server
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: v1
          kind: Service
          metadata:
            name: argo-server
            namespace: argo
          spec:
            type: LoadBalancer
            selector:
              app: argo-server
            ports:
```

```
    - port: 443
      targetPort: 2746
  kubeconfig: "{{ harvester_kubeconfig }}"
```

2. Configure Argo CD to Manage the Harvester Cluster

- **Prepare Git Repository**:
  - Create a Git repository (e.g., `github.com/your-org/ephemeral-clusters`) with the following structure:

```yaml
ephemeral-clusters/
├── vms/
│   ├── master.yaml
│   ├── worker1.yaml
│   ├── worker2.yaml
├── kubernetes/
│   ├── network-attachment-definition.yaml
│   ├── kubeadm-config.yaml
├── applications/
│   ├── mtl-mcm.yaml
│   ├── ice-test-workflow.yaml
```

  - Add VM manifests (from your `create_vms.yml` playbook), NetworkAttachmentDefinition (from `configure_sriov.yml`), and MTL/MCM pod manifests.
- **Add Cluster to Argo CD**:
  - In the Argo CD UI, go to **Settings** > **Clusters** > **Add Cluster**.
  - Use the Harvester kubeconfig (from `/home/ubuntu/.kube/config` on the control plane VM) or a service account token:

```bash
kubectl create sa argo -n kube-system
kubectl create clusterrolebinding argo --clusterrole=cluster-admin --serviceaccount=ku
TOKEN=$(kubectl create token argo -n kube-system)
```

Create a kubeconfig with the token and API endpoint (https://10.123.235.200:6443).
  - Alternatively, add the cluster via CLI:

```bash
argocd cluster add <context> --name harvester-k8s --kubeconfig /path/to/kubeconfig
```

- **Create Argo CD Application**:

Grok 3

  - Define an Argo CD Application to manage the ephemeral cluster:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: ephemeral-cluster
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/your-org/ephemeral-clusters.git
    targetRevision: HEAD
    path: .
  destination:
    server: https://10.123.235.200:6443
    namespace: ephemeral-test
  syncPolicy:
    automated:
      prune: true
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
```

  - Apply the Application:

```bash
kubectl apply -f application.yaml -n argocd
```

  - Argo CD will synchronize the Git repository's manifests, deploying VMs, Kubernetes configurations, and applications.

Ansible Automation:
Automate Git repository setup and Argo CD configuration:

```
- name: Configure Argo CD for Harvester cluster
  hosts: harvester
  tasks:
    - name: Ensure Git repository is cloned
      ansible.builtin.git:
        repo: https://github.com/your-org/ephemeral-clusters.git
        dest: /tmp/ephemeral-clusters
        update: yes
      delegate_to: localhost

    - name: Copy manifests to Git repository
      ansible.builtin.copy:
```

```yaml
    src: "{{ item.src }}"
    dest: "/tmp/ephemeral-clusters/{{ item.dest }}"
  loop:
    - { src: 'vms/master.yaml', dest: 'vms/master.yaml' }
    - { src: 'vms/worker1.yaml', dest: 'vms/worker1.yaml' }
    - { src: 'vms/worker2.yaml', dest: 'vms/worker2.yaml' }
    - { src: 'network-attachment-definition.yaml', dest: 'kubernetes/network-attachment-def
    - { src: 'mtl-mcm.yaml', dest: 'applications/mtl-mcm.yaml' }

- name: Commit and push changes to Git
  ansible.builtin.shell: |
    cd /tmp/ephemeral-clusters
    git add .
    git commit -m "Update ephemeral cluster manifests"
    git push origin main
  delegate_to: localhost
  environment:
    GIT_AUTHOR_NAME: "Ansible"
    GIT_AUTHOR_EMAIL: "ansible@example.com"

- name: Create Argo CD Application
  kubernetes.core.k8s:
    state: present
    definition:
      apiVersion: argoproj.io/v1alpha1
      kind: Application
      metadata:
        name: ephemeral-cluster
        namespace: argocd
      spec:
        project: default
        source:
          repoURL: https://github.com/your-org/ephemeral-clusters.git
          targetRevision: HEAD
          path: .
        destination:
          server: https://10.123.235.200:6443
          namespace: ephemeral-test
        syncPolicy:
          automated:
            prune: true
            selfHeal: true
          syncOptions:
            - CreateNamespace=true
    kubeconfig: "{{ harvester_kubeconfig }}"
```

3. Provision Ephemeral Kubernetes Clusters with Argo Workflows

Use Argo Workflows to automate the lifecycle of ephemeral clusters, including provisioning, testing ICE drivers, deploying MTL/MCM, and cleanup.

a. Define a Workflow for Ephemeral Cluster:

Grok 3

- Create a workflow manifest (`ice-test-workflow.yaml`) in your Git repository:

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: ice-test-
  namespace: argo
spec:
  entrypoint: test-pipeline
  templates:
  - name: test-pipeline
    steps:
    - - name: deploy-vms
        template: deploy-vms
    - - name: setup-kubernetes
        template: setup-kubernetes
    - - name: run-ice-test
        template: run-ice-test
    - - name: cleanup
        template: cleanup

  - name: deploy-vms
    script:
      image: bitnami/kubectl:latest
      command: [bash]
      source: |
        kubectl apply -f /git/vms -n ephemeral-test
        until kubectl get vmi -n ephemeral-test | grep Running; do sleep 5; done
      volumeMounts:
      - name: git
        mountPath: /git
    volumes:
    - name: git
      git:
        repo: https://github.com/your-org/ephemeral-clusters.git
        revision: main

  - name: setup-kubernetes
    script:
      image: bitnami/kubectl:latest
      command: [bash]
      source: |
        kubectl apply -f /git/kubernetes -n ephemeral-test
        kubectl exec -n ephemeral-test k8s-master -- kubeadm init --pod-network-cidr=10.24
        kubectl exec -n ephemeral-test k8s-worker1 -- sh /tmp/join-command.sh
        kubectl exec -n ephemeral-test k8s-worker2 -- sh /tmp/join-command.sh
      volumeMounts:
      - name: git
        mountPath: /git
```

```
  - name: run-ice-test
    script:
      image: openvisualcloud/mtl:latest
      command: [bash]
      source: |
        # Test ICE driver and MTL/MCM
        lsmod | grep ice
        /usr/local/bin/mtl_test <test-args>
        # Example: Run media streaming test
        ffmpeg -i input.mp4 -f mtl output.mp4
      securityContext:
        privileged: true
      resources:
        limits:
          cpu: "2"
          memory: "4Gi"

  - name: cleanup
    script:
      image: bitnami/kubectl:latest
      command: [bash]
      source: |
        kubectl delete namespace ephemeral-test
```

- This workflow:
  - Deploys VMs using KubeVirt manifests.
  - Initializes a Kubernetes cluster with `kubeadm`.
  - Runs an ICE driver and MTL/MCM test.
  - Deletes the namespace to clean up.

b. Trigger the Workflow:

- Submit the workflow via Argo CLI or UI:

```
argo submit -n argo ice-test-workflow.yaml
```

bash

- Monitor the workflow:

```
argo watch -n argo <workflow-name>
```

bash

Ansible Automation:
Automate workflow submission:


```
- name: Run Argo Workflow for ephemeral cluster
  hosts: harvester
  tasks:
    - name: Submit Argo Workflow
      kubernetes.core.k8s:
        state: present
        definition:
          apiVersion: argoproj.io/v1alpha1
          kind: Workflow
          metadata:
            generateName: ice-test-
            namespace: argo
          spec:
            entrypoint: test-pipeline
            # ... (rest of the workflow spec)
        kubeconfig: "{{ harvester_kubeconfig }}"
```

4. Test Intel ICE Drivers with MTL and MCM

yaml

- **Validate ICE Drivers:**
  - The `run-ice-test` step in the workflow checks ICE driver loading and runs MTL/MCM tests.
  - Verify SR-IOV VF connectivity:

```
ip link show
ping 10.123.235.200
```

bash

- **Run MTL/MCM Workloads:**
  - The workflow deploys MTL/MCM containers with SR-IOV interfaces (via NetworkAttachmentDefinition).
  - Test media streaming or transcoding workloads as per OpenVisualCloud documentation:

```
docker run --network host openvisualcloud/mtl:latest <test-command>
```

bash

  - Capture test output in Argo Workflows logs.
- **Iterate and Destroy:**
  - Argo Workflows automatically cleans up the namespace after testing.
  - Review logs in the Argo UI or CLI (`argo logs -n argo <workflow-name>`).
  - Trigger new workflows for subsequent tests by updating the Git repository.

Best Practices for Ephemeral Clusters

- **GitOps with Argo CD**:
  - Store all manifests in Git for version control and auditability.
  - Use branch-based workflows (e.g., `test/ice-v1.2`) for different ICE driver versions.
- **Workflow Optimization**:
  - Parallelize VM provisioning in Argo Workflows to reduce setup time.
  - Use Argo Events to trigger workflows on Git commits or test schedules.
- **Resource Management**:
  - Limit VM resources (e.g., 1 vCPU, 2GB RAM) for ephemeral clusters to conserve Harvester resources.
  - Restrict SR-IOV VFs to avoid oversubscription (e.g., 4 VFs per NIC).
- **Security**:
  - Use Kubernetes RBAC to restrict Argo's access to the `ephemeral-test` namespace.
  - Disable Pod Security Standards (PSS) for privileged MTL/MCM containers:

```bash
kubectl label namespace ephemeral-test pod-security.kubernetes.io/enforce=privileged
```

- **Monitoring**:
  - Integrate Argo with Prometheus (available in Harvester) to monitor workflow performance and ICE driver metrics.
  - Export workflow logs to a central system (e.g., Loki) for test analysis.
- **Cleanup**:
  - Ensure workflows include a cleanup step to delete namespaces and VMs.
  - Use Harvester's Longhorn storage to reclaim ephemeral volumes.

Challenges and Workarounds

- **SR-IOV and ICE Driver Compatibility**:
  - Verify ICE driver compatibility with your NIC (e.g., E810 for X710) using OpenVisualCloud's Media-Transport-Library.
  - If VFs fail, check SR-IOV settings in Harvester node BIOS and kernel (`dmesg | grep -i sriov`).
- **Ephemeral Cluster Stability**:
  - Harvester's ephemeral `/etc` may cause issues with node reboots. Use persistent storage for critical configurations.
  - Enable Kubernetes ephemeral volumes if needed (`--feature-gates=GenericEphemeralVolume=true`).
- **Argo Limitations**:
  - Argo Workflows require careful error handling for flaky tests. Use `retryStrategy` in workflow templates:

Grok 3

```yaml
retryStrategy:
  limit: 3
```

  - Complex MTL/MCM deployments may require custom scripts in workflow steps.
- **MTL/MCM Deployment**:
  - Privileged containers for MTL/MCM may conflict with Harvester's PSS. Disable PSS for the test namespace as shown above.
  - Ensure MTL/MCM images are pre-pulled to avoid delays in workflows.

Sample Workflow for Testing

- **Provision Cluster**:
  - Update the Git repository with VM and Kubernetes manifests.
  - Argo CD synchronizes the manifests to deploy VMs and a Kubernetes cluster.
- **Run Test Workflow**:
  - Trigger the Argo Workflow to initialize the cluster, test ICE drivers, and deploy MTL/MCM.
  - Monitor progress in the Argo UI.
- **Analyze Results**:
  - Collect ICE driver and MTL/MCM test logs via Argo.
  - Check workflow metrics in Prometheus.
- **Cleanup**:
  - The workflow deletes the namespace, freeing resources.
  - Update the Git repository for the next test cycle.

References

- Argo CD Documentation: https://argoproj.github.io/argo-cd/
- Argo Workflows Documentation: https://argoproj.github.io/argo-workflows/
- Harvester SR-IOV Guide: https://docs.harvesterhci.io/v1.2/networking/sriov
- OpenVisualCloud MTL/MCM: https://github.com/OpenVisualCloud
- Kubernetes Ephemeral Containers: https://kubernetes.io/docs/concepts/workloads/pods/ephemeral-containers/

If you need specific Argo Workflow templates, Ansible playbooks for MTL/MCM testing, or assistance with ICE driver integration, let me know, and I can provide tailored solutions!