Task - Find at least TWO state-of-the-art implementation for measurement

KubeVirt

KubeVirt is a framework, built on top of Kubernetes, to manage virtual machines alongside containers on k8s cluster. KubeVirt uses custom resources along with controllers to provide VM management on cluster.

Components

KubeVirt has the following components;

Virtual Machine Instance (VMI) - VMI represents a virtualized workload. A new type is added to K8s API server through Custom Resource Definitions (CRD) to register VMI.

KubeVirt VM Controller - VMI is scheduled and handled by this component (cluster wide logic to manage VMs). It creates a pod for VM process to launch in. It makes sure that desired state described in CRD is achieved for the creation of VMI. VM controller ensure state persistence of VMs in case of node failures and multiple restarts of its underlying VMI. VM controller can shut down a VM, and then choose to start that exact same VM up again at later time.

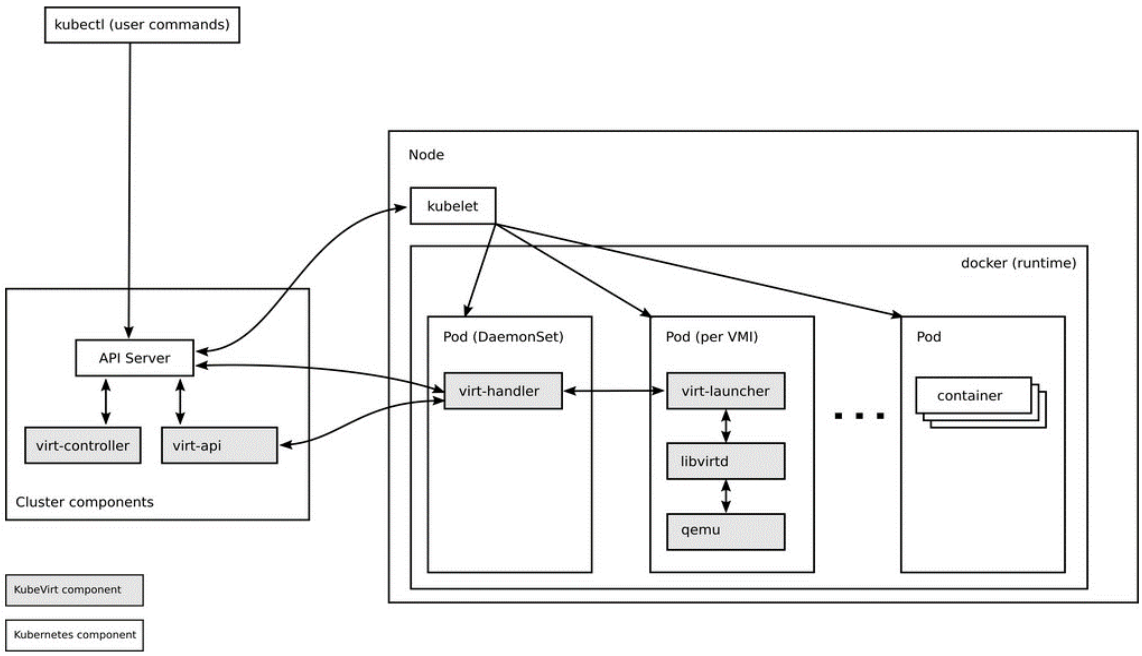
virt-handler - Present alongside Kubelet to launch and configure VMI on a node until VMI desired state reaches. API server sends request to virt-handler to launch or delete the VM. virt-handler then sends the request to virt-launcher to take required action. It continuously monitors the VMs and in case of unhealthy VM, it sends signal to VM launcher to stop or restarting the VM (Low-level orchestration).

virt-launcher – Launches and delete VM and acquire the resource based on signals from virt-handler. It interacts with underlying hypervisor or virtualization technology to handle the resources to launch VM.

virt-handler and virt-launcher collaborate closely to manage VMs within KubeVirt. virt-handler handles lifecycle management, and event handling for VMs, while virt-launcher is responsible for creating and managing the runtime environment and virtual devices required by the VMs. Together, they enable the seamless integration and operation of VMs within a Kubernetes cluster.

API Server - An entry point for KubeVirt. It works with Dynamic Admission Controller (DAC) to validate VM create/delete requests and forward them to kubevirt HTTP endpoints.

VirtualMachineInstanceReplicaSet (VMIRS) workload controller which manages scaling out the identical VMI objects. This controller behaves identical to the Kubernetes ReplicSet controller.



https://github.com/kubevirt/kubevirt/blob/main/docs/architecture.md

KubeVirt Stack

```
+----------------------+
| KubeVirt             |
+----------------------+
| Orchestration (K8s)  |
+----------------------+
| Scheduling (K8s)     |
+----------------------+
| Container Runtime    |
+----------------------+
| Operating System     |
+----------------------+
| (Virtual)            |
+----------------------+
| Physical             |
+----------------------+
```

KNative

Knative is framework to deploy and manage serverless workloads on K8s through CRDs and custom controllers. It abstracts and takes care of underlying infrastructure required to deploy and scale the serverless application.

Knative Serving

Serving is the component of Knative which deploys the serverless application. It defines set of objects as CRDs. And, CRDs are then added to K8s API server

Components of Knative Serving

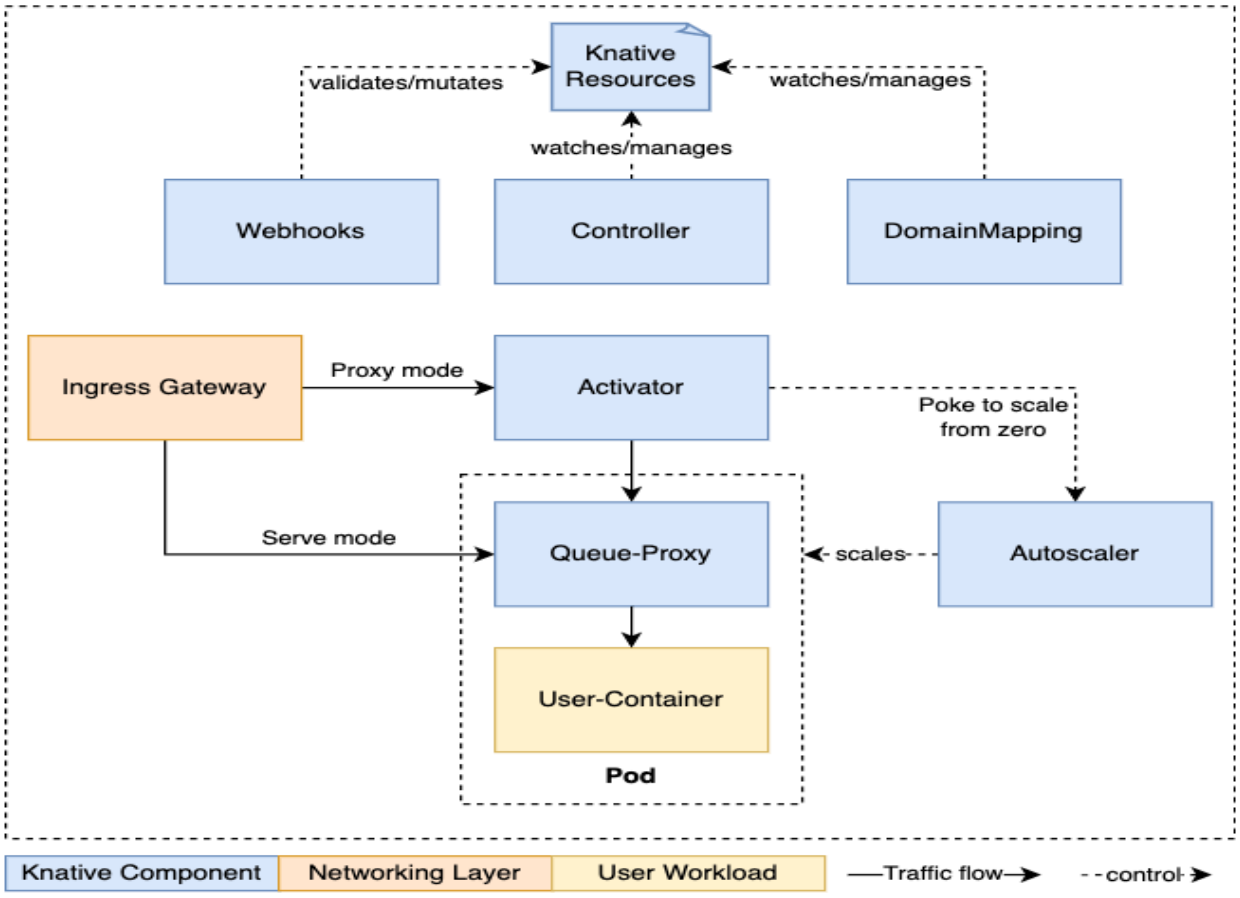
The primary resources in Knative Serving (defined as CRDs) are;

Route – provides a named endpoint and mechanism to route traffic to revisions

Revisions – a snapshot of code + config which is created by configurations

Configurations – represents the stream of environments for revisions. It also represents desired state of latest revision.

Service – a high-level container for managing a Routes and Configuration which implement a network service. Manages the whole lifecycle of serverless workload.



https://knative.dev/docs/serving/architecture/

Custom Controller - handles all these custom resources to deploy workloads.

Docker runtime environment is used to compile the programs written in different languages in Knative.

API Server with Dynamic Admission Controller (DAC) – An entry point to Knative serving. DAC validates the user requests.

Activator – Queues the incoming requests and communicates with the autoscaler to bring scaled-to-zero services back up.

Autoscaler - Scales the Knatve Services based on configuration and incoming traffic.

Performance Evaluation Metrics

Workflow

The two above mentioned state-of-the-art implementations have similar workflows.

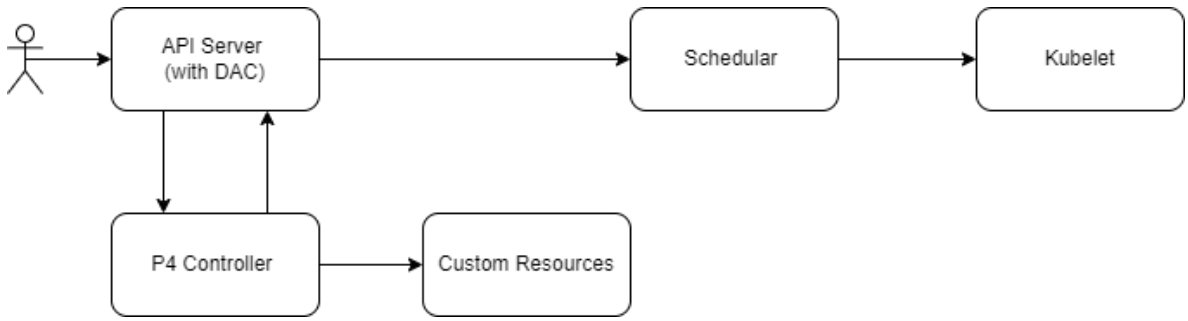
1- User send the request (VM, Serverless workload) to API server, API server works closely with DAC (Dynamic Admission Control) to validate the requests.

2- After the successful creation of CRs, controller observes the creation of new custom resource and creates a Pod where object will be deployed. It communicates with K8s scheduler to deploy the created pod.

3- K8s Scheduler schedules the Pod on the available hosts.

4- There is controller inside of each Pod which monitors the state of deployment of Pod on hosts.

A similar approach can be used for implementation of KubeP4. The below diagram describes the tentative workflow of KubeP4;



1- User sends the request (to deploy P4) to API server, DAC (Dynamic Admission Control) validates the request (checks if request is malformed). After validation of request, it invokes the P4 controller.

2- P4 Controller receives the P4 program and creates the custom resource (i.e, type p4) to represent a p4 program configuration. After creation of custom resource, it observes the creation of new custom resource and creates a Pod where the resource will be deployed. Finally, it communicates back to API server to schedule the created pod.

3- The API server will talk to K8s Scheduler to schedule the Pod on the available nodes.

4- P4Runtime will be deployed on worker nodes. Once, node gets a scheduled Pod, Kubelet uses P4Runtime to compile the P4 program which will manage devices accordingly.

5- P4 controller continuously monitors the Pod state and make decisions about the pod deployments.

Compilers (with and without DPDK)

Other Metric to evaluate our approach is to measure the latency with using different frameworks to compile P4 programs. The suggested compilers are P4c, T4P4s and P4LLVM [1]. The P4c and P4LLVM does not have support for DPDK but T4P4s works with DPDK. As DPDK skips the kernel space to process the packet, we can compare the latency of deploying p4 program on a server with P4c and T4P4s.

[1] https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2020-11-1/NET-2020-11-1_20.pdf

Scheduling

The last metric is to compare the scheduling algorithms used in our approach vs state-of-the-art approaches. A K8s scheduler is used to schedule the pods in KubeVirt and Knative. We can also integrate the K8s scheduler to schedule the P4 programs. In future, we will see which best suits our use-case, a default K8s scheduler or a custom scheduler.

References

https://github.com/kubevirt/kubevirt/blob/main/docs/components.md

https://github.com/kubevirt/kubevirt/blob/main/docs/architecture.md

https://github.com/kubevirt/kubevirt

KubeVirt Google Group : https://groups.google.com/g/kubevirt-dev

Adding support for DPDK in KubeVirt : https://groups.google.com/g/kubevirt-dev/c/87f_0RTMQj0

https://knative.dev/docs/concepts/

P4Runtime: https://github.com/p4lang/p4runtime