EVE Kubernetes Control Plane Integration - Draft

What does it mean to integrate EVE with Kubernetes? This does not refer to running Kubernetes *on top of* EVE (k3s solutions already exist for this), but rather how do we integrate EVE the platform with Kubernetes at the control plane level?

Comparison

To explore this further, we need to outline in a brief and limited fashion what each product - EVE and Kubernetes - does.

Kubernetes

Kubernetes handles:

- Serving an API for updating workloads, with authentication/authorization.
- Accepting registration of candidate nodes for workloads.
- Scheduling workloads on nodes according to deployment rules.
- Ensuring the correct number of workload replicas exists.

The above occurs primarily via the API and controllers, which implement control loops over changed data sets.

Kubernetes does not handle operating system deployment and software installation. A node begins its “Kubernetes life” when it already has an operating system and relevant Kubernetes software (container runtime, kubelet, kube-proxy) installed, and it performs communication and registration with the API server. The primary node-side agent between the Kubernetes control plane and the local node is the kubelet.

- Kubernetes concerns itself with everything from the kubelet up.
- Kubernetes does not concern itself with the upgrade and management of the operating system, container runtime, or the kubelet itself, i.e. anything from the kubelet down.

blocked URL

EVE

EVE varies from Kubernetes in several ways, beyond the actual API implementation itself.

1. Device management: The most fundamental difference between EVE and Kubernetes is that EVE explicitly takes responsibility for the entire device, from the hardware up. It manages the operating system itself along with the software that connects with the control plane, securing the OS stack and device, managing upgrades.
2. Narrower scenarios: EVE’s array of possible deployment scenarios is narrower than Kubernetes. It cannot create custom arbitrary resources, set deployment rules to have a fixed or varying number of a given workload and automatically schedule it, automatically deploy a workload on all nodes, have complex workload-to-node deployment rules.
3. Virtualization: EVE has a native first-class ability to deploy both containerized and virtualized workloads.
4. Device control: EVE has a native first-class ability to provision devices to workloads, both pass-through and, where relevant, virtualized.
5. Network flexibility: EVE has a native first-class ability to deploy multiple networks of multiple types on each node and workloads.

Summarizing the differences between EVE and Kubernetes:

- EVE has more “around the workload” capabilities, such as devices, virtualization and networks;
- Kubernetes has more “schedule the workload” capabilities;
- EVE manages the entire device, whereas Kubernetes focuses only on workloads.

Integration Scenarios

There are several ways to consider EVE Kubernetes integration. Each of these needs to be explored for usefulness and feasibility. Each also needs review to ensure that it actually covers all of the EVE use cases.

At a high-level, there are only two communications channels: EVE device to EVE controller; user to EVE controller. Thus the possible points of integration are:

- EVE device communicates with kube-apiserver as kubelet
- User communicates with EVE controller via Kubernetes-compatible API

There are several variants of each.

1. EVE as kubelet
   a. Managed entirely from Kubernetes. EVE controller replaced entirely by native Kubernetes.
   b. Managed partially from Kubernetes. Workload management performed by Kubernetes control plane, OS management performed by EVE controller.
2. EVE controller exposes Kubernetes-compatible API
   a. EVE controller as Kubernetes API
   b. EVE controller deployed onto Kubernetes

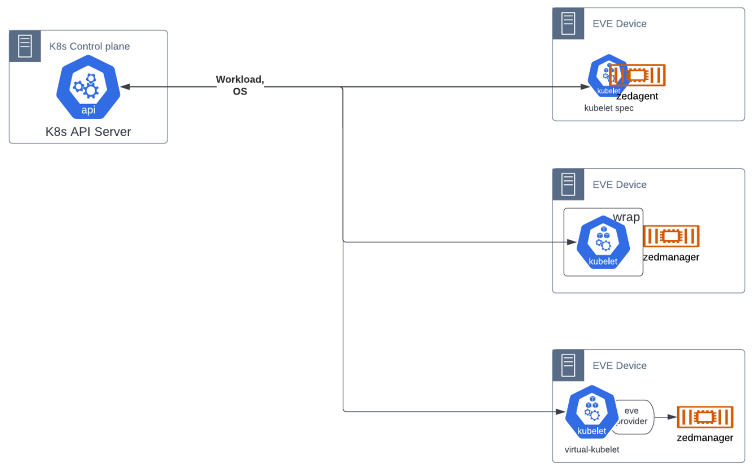
Finally, it may be possible to combine them, where the controller exposes a Kubernetes-compatible API, and uses native Kubernetes to control EVE itself.

EVE as Kubelet

An EVE device is expected to receive workload assignment from a standard native Kubernetes cluster.

This can be implemented in several ways:

- Have pillar, specifically zedagent, implement the kubelet API. Note that the API is undocumented. The kubelet has its own communications API to apiserver, and needs to implement the PodSpec API, as well as handle pod management functions.
- Deploy the actual kubelet (current size v1.25.0 ~114MB), but control its interactions. This may be very difficult.
- Deploy virtual-kubelet (current size ~47MB) along with an eve-os provider. Note that virtual-kubelet has all providers in-tree, making it grow over time.

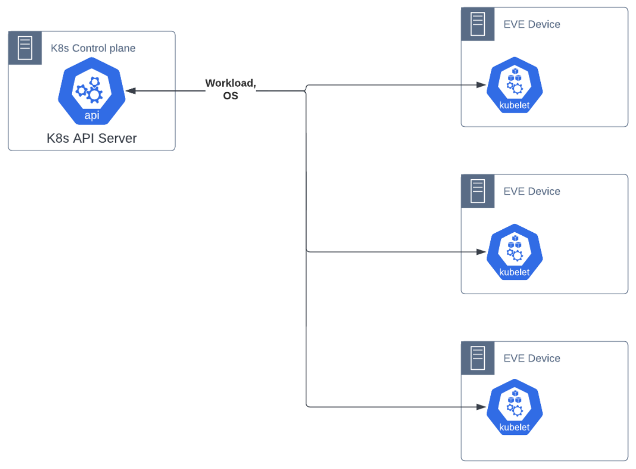


The functionality that is not available within Kubernetes, as listed above, is provided in one of two ways, either interfacing solely with Kubernetes, or with both Kubernetes and EVE controller.

Complete vs Partial Kubernetes

Complete Kubernetes

All functionality is received from Kubernetes. The necessary additional OS management parts of the EVE API are provided by utilizing existing capabilities in the kubelet-apiserver API, and other native Kubernetes artifacts. **This assumes that such is possible; TBD.**

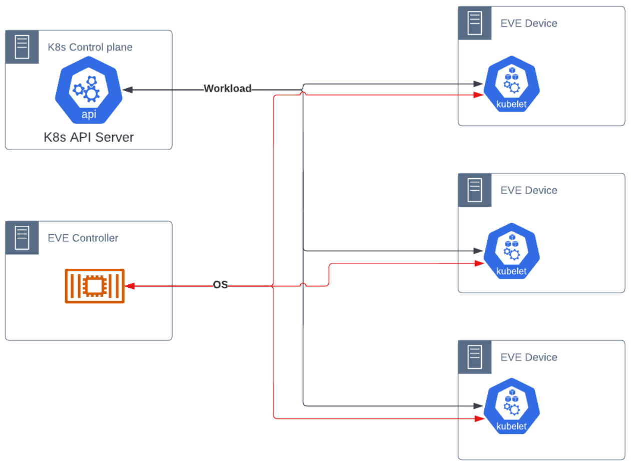


Typical flow is:

1. Device is flashed with eve-os.
2. Device registers to kube-apiserver as a new node with annotations/labels that identify it as an eve-os node. Registration process is similar to native Kubernetes, but may require additional validations. **TBD**
3. Device receives workloads to deploy using normal Kubernetes kubelet API
4. Device receives updates to operating system via override of existing kubelet API or pod workload functions. **TBD**

Partial Kubernetes

Functionality that fits within Kubernetes' native space, specifically workload scheduling, is provided by Kubernetes. Functionality that is beyond the scope of Kubernetes, specifically management and security of the operating system, is provided by a distinct EVE controller, as today.



Typical flow is:

1. Device is flashed with eve-os
2. Device registers to EVE controller as today
3. EVE controller sends configuration including sufficient information to register with Kubernetes, including certificates, secrets and address of kube-apiserver
4. EVE device uses information to launch kubelet.
5. Device kubelet registers with Kubernetes control plane
6. Device receives:
    a. Workload instructions from Kubernetes via kubelet
    b. OS instructions from EVE controller via zedagent

This method requires each EVE device to maintain active communications to two distinct controllers simultaneously. This may be simplified in one of the EVE Controller cases.

Multiple Networks

While Kubernetes itself provides only two network interfaces per pod - one loopback and one primary - it does not prevent a pod from having more network interfaces.

When Kubernetes launches a pod, the kubelet calls a local binary on the host, a CNI plugin, to: create the network interface (normally eth0), attach it to a network, and assign an IP. This interface is expected to provide communication to all other pods in a flat manner, services and the off-cluster network.

The CNI plugin, however, can do whatever it wants with the pod, provided it meets the minimum defined requirements of Kubernetes and the CNI specification. Thus, a CNI plugin could assign the normal default network that Kubernetes expects, and then 3 more interfaces connected to different networks. Further, a plugin could act as an intelligent multiplexer, calling the default plugin and then one or more different ones, each of which assigns different interfaces.

To provide support for these capabilities, the Kubernetes Network Plumbing Working Group has been defining standards for pods to declare their need for additional networks without breaking the existing Kubernetes API. They do so by:

- Declaring additional network types globally across the cluster via the CRD NetworkAttachmentDefinition
- Annotating pods that wish to use additional network interfaces with k8s.v1.cni.cncf.io/networks

blocked URL

The specification for multiple networks is available here. A reference implementation of a multiple network CNI plugin, multus, is available here.

EVE Controller Exposes Kubernetes API

We keep the existing EVE device structure and EVE-controller API. Instead, the controller API is replaced with a Kubernetes-compatible API. This API is not native Kubernetes, but instead uses Custom Resource Definitions (CRDs) to create appropriate resources for managing EVE devices, both workloads and operating systems.

The primary goal would be standardization on a Kubernetes-compatible API that could be used across controllers.This would greatly improve adam, simplify eden, and provide better integrations with controllers.

EVE might or might not be a kubelet in this scenario; these are agnostic as to how the controller communicates with EVE.

The API is to be defined and published as part of lfedge, as an optional basis for EVE controllers, but would be extensible so that commercial controllers can provide additional capabilities.

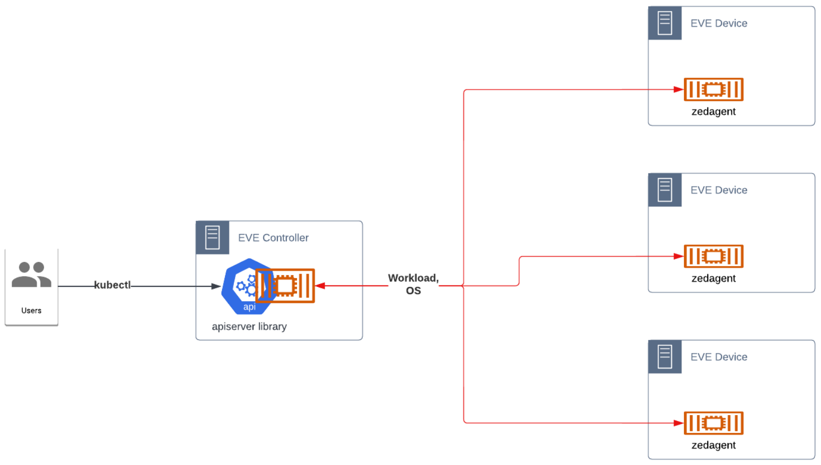
There are two implementation variants:

- Single process that implements the Kubernetes API
- Containers and manifests that deploy into a true Kubernetes cluster

Note that it is not strictly necessary to implement anything at all. We could define APIs that would work as a standard. However, lack of a proven reference implementation would weaken adoption.

EVE Controller as Kubernetes API

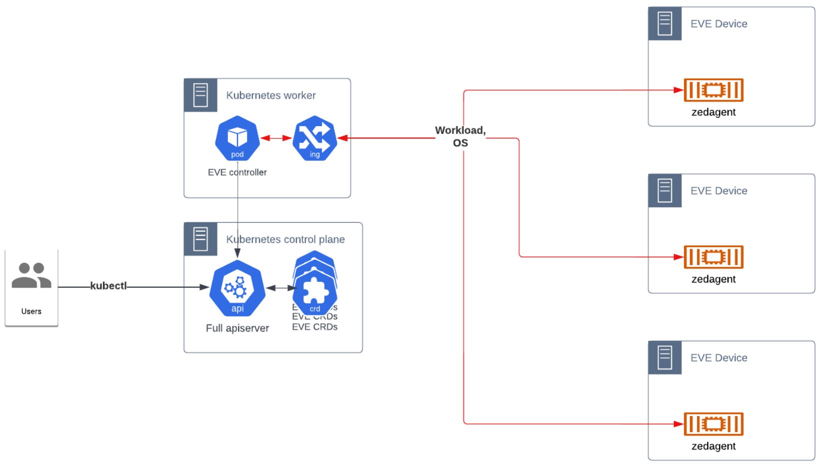
Adam, the current OSS controller, is a single process which exposes a minimal REST API. The design would extend Adam to expose a Kubernetes-compatible API to the end-user.



Implementing Kubernetes API directly is complex. A starting point may be https://github.com/kubernetes/sample-apiserver or kcp. Part of kcp is a bundling of multiple apiserver libraries into a single reusable library. However, much of the project has been focused on multiple cluster management.

EVE Controller deployed into Kubernetes

Adam, or a next-generation version, is recreated as a deployment intended to run in Kubernetes. The manifests create the CRDs necessary, while the container implements the control loops for managing those resources, as well as exposing the endpoints to handle the EVE device API.



This is much more straightforward than implementing the actual API as in the previous, but would require running a dedicated Kubernetes cluster. It is possible to scale it down to a single node cluster, if desired.

Combination

A possible combination of the above is the EVE Controller deployed into Kubernetes, exposing a Kubernetes-compatible API, with the kubelet API leveraged to provide communications with EVE devices. While more difficult to design, it would simplify the deployment of and integration with controllers in the future.