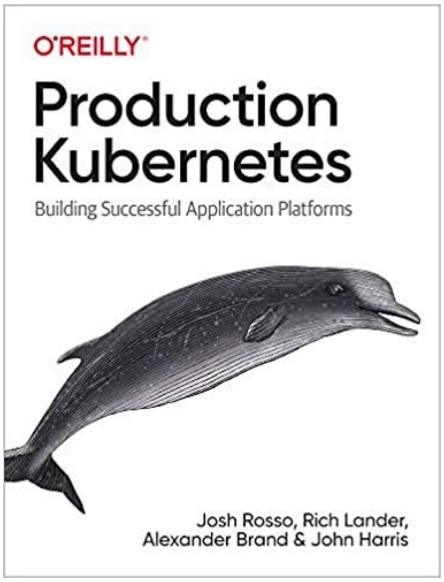
Book review – Production Kubernetes: Building Successful Application Platforms 1st Edition by Josh Rosso (Author), Rich Lander (Author), Alex Brand (Author), John Harris (Author)

Summary: Many useful Practical FIELD k8s Production experience, best practices, tips and advices inside !

1/ free complimentary book by #vmware (need your email): https://tanzu.vmware.com/content/ebooks/production-kubernetes
2/ buy it at Amazon: https://www.amazon.com/Production-Kubernetes-Successful-Application-Platforms/dp/1492092304/ref=sr_1_2 (kindle $48)
3/ authors book talk video: https://www.youtube.com/watch?v=blanHrC0SFQ&feature=youtu.be



A lot of useful tips running/maintaining Production k8s, e.g. dedicated etcd cluster upgrade should be done subtractively because "Adding a fourth member to a three-node etcd cluster, for example, will require an update to all etcd nodes' member list, which will require a restart."(p.58) The other control planes and worker nodes can be replaced additively (p.59)

Useful Production tip: Why unexpected request errors during application rolling updates ? (p.145) because of race conditions between kubelet doing Pod Shutdown while Endpoints controller removing Pod IP addresses at the SAME time. If Endpoints controller/kubeproxy removed the deleted Pod IP too late, then request will fail. So, good suggestions to SIGTERM handlers and sleep pre-stop hooks to solve this !

Another tip: use "DNS cache add-on called NodeLocal DNSCache" to help DNS service performance. How to tune DNS resolver config with ndots and search. There is an example of of the impact of ndots config on Pods to resolve external names. It can result in multiple DNS queries. (p.149)

It shows users a lot of good design factors to consider, e.g. migrating strategy. When to use Cluster replacement, node replacement and in-place upgrades ? Many overview of different choices and examples, e.g. Which container runtimes or CRI ? Which Storage classes or CSI ? Which service meshes ? Which CNI ? Which Ingress controller ? Which Secrets provider ? ...etc.

A lot of valuable discussions and tips in Admission control (OPA), Observability(audit, logging, events, metrics, showbacks, tracings), Identity (e.g.OIDC, kiam, SAT, PSAT), Multitenancy (Namespace Boundary, RBAC, Resource Quotas, Admission Webhooks), Autoscaling (HPA, VPA, Cluster Proportional Autoscaler, Cluster Autoscaling),Application Considerations (State Probes, termination, logs), Software Supply Chain (images, registries, Vulnerability Scanning, Image Signing, CD pipelines, Push-Based Deployments), Platform Abstractions (Command-Line Tooling, Abstraction Through Templating).

I am a CKA and still learn a lot important and useful insights, experience, best practices and tips in this very practical field book!

Strongly recommend!

Machine Installations

When the machines for your cluster are spun up, they will need an operating system, certain packages installed, and configurations written. You will also need some utility or program to determine environmental and other variable values, apply them, and coordinate the process of starting the Kubernetes containerized components.

There are two broad strategies commonly used here:

1. Configuration management tools
2. Machine images

Configuration Management

Configuration management tools such as Ansible, Chef, Puppet, and Salt gained popularity in a world where software was installed on virtual machines and run directly on the host. These tools are quite magnificent for automating the configuration of multitudes of remote machines. They follow varying models but, in general, administrators can declaratively prescribe how a machine must look and apply that prescription in an automated fashion.

These config management tools are excellent in that they allow you to reliably establish machine consistency. Each machine can get an effectively identical set of software and configurations installed. And it is normally done with declarative recipes or playbooks that are checked into version control. These all make them a positive solution.

Where they fall short in a Kubernetes world is the speed and reliability with which you can bring cluster nodes online. If the process you use to join a new worker node to a cluster includes a config management tool performing installations of packages that pull assets over network connections, you are adding significant time to the join process for that cluster node. Furthermore, errors occur during configuration and installation. Everything from temporarily unavailable package repositories to missing or incorrect variables can cause a config management process to fail. This interrupts the cluster node join altogether. And if you're relying on that node to join an autoscaled cluster that is resource constrained, you may well invoke or prolong an availability problem.

Machine Images

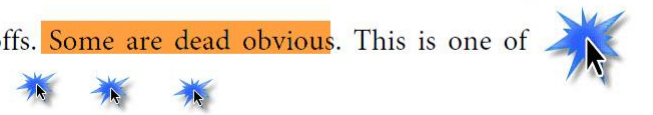
Using machine images is a superior alternative. If you use machine images with all required packages installed, the software is ready to run as soon as the machine boots

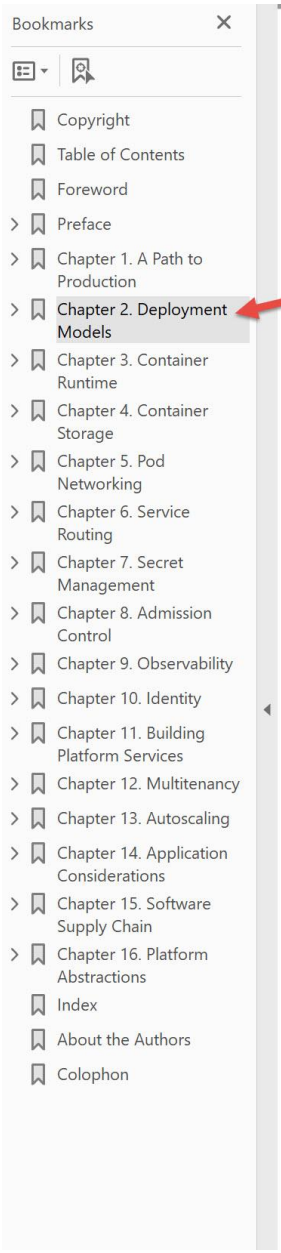
up. There is no package install that depends on the network and an available package repo. Machine images improve the reliability of the node joining the cluster and considerably shorten the lead time for the node to be ready to accept traffic.

The added beauty of this method is that you can often use the config management tools you are familiar with to build the machine images. For example, using Packer from HashiCorp you can employ Ansible to build an Amazon Machine Image and have that prebuilt image ready to apply to your instances whenever they are needed. An error running an Ansible playbook to build a machine image is not a big deal. In contrast, having a playbook error occur that interrupts a worker node joining a cluster could induce a significant production incident.

You can—and should—still keep the assets used for builds in version control, and all aspects of the installations can remain declarative and clear to anyone that inspects the repository. Anytime upgrades or security patches need to occur, the assets can be updated, committed and, ideally, run automatically according to a pipeline once merged.

Some decisions involve difficult trade-offs. Some are dead obvious. This is one of those. Use prebuilt machine images.





Strategies

We're going to look at three strategies for upgrading your Kubernetes-based platforms:

1. Cluster replacement
2. Node replacement
3. In-place upgrades

We're going to address them in order of highest cost with lowest risk to lowest cost with highest risk. As with most things, there is a trade-off that eliminates the opportunity for a one-size-fits-all, universally ideal solution. The costs and benefits need to be considered to find the right solution for your requirements, budget, and risk tolerance. Furthermore, within each strategy, there are degrees of automation and testing that, again, will depend on factors such as engineering budget, risk tolerance, and upgrade frequency.

Keep in mind, these strategies are not mutually exclusive. You can use combinations. For example, you could perform in-place upgrades for a dedicated etcd cluster and then use node replacements for the rest of the Kubernetes cluster. You *can* also use different strategies in different tiers where the risk tolerances are different. However, it is advisable to use the same strategy everywhere so that the methods you employ in production have first been thoroughly tested in development and staging.

Whichever strategy you employ, a few principles remain constant: test thoroughly and automate as much as is practical. If you build automation to perform actions and test that automation thoroughly in testing, development, and staging clusters, your production upgrades will be far less likely to produce issues for end users and far less likely to invoke stress in your platform operations team.

1 Cluster replacement

Cluster replacement is the highest cost, lowest risk solution. It is low risk in that it follows immutable infrastructure principles applied to the entire cluster. An upgrade is performed by deploying an entirely new cluster alongside the old. Workloads are migrated from the old cluster to the new. The new, upgraded cluster is scaled out as needed as workloads are migrated on. The old cluster's worker nodes are scaled in as workloads are moved off. But throughout the upgrade process there is an addition of an entirely distinct new cluster and the costs associated with it. The scaling out of the new and scaling in of the old mitigates this cost, which is to say that if you are upgrading a 300-node production cluster, you do not need to provision a new cluster with 300 nodes at the outset. You would provision a cluster with, say, 20 nodes. And when the first few workloads have been migrated, you can scale in the old cluster that has reduced usage and scale out the new to accommodate other incoming workloads.

Find
velero
Previous

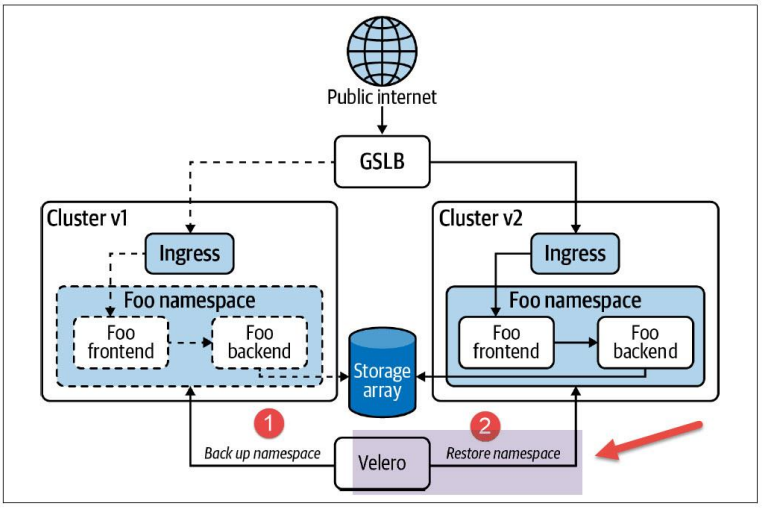


Figure 2-6. Migrating workloads between clusters with a backup and restore using Velero.

2

Node replacement

The node replacement option represents a middle ground for cost and risk. It is a common approach and is supported by Cluster API. It is a palatable option if you're managing larger clusters and compatibility concerns are well understood. Those compatibility concerns represent one of the biggest risks for this method because you are upgrading the control plane in-place as far as your cluster services and workloads are concerned. If you upgrade Kubernetes in-place and an API version that one of your workloads is using is no longer present, your workload could suffer an outage. There are several ways to mitigate this:

- Read the Kubernetes release notes. Before rolling out a new version of your platform that includes a Kubernetes version upgrade, read the CHANGELOG thoroughly. Any API deprecations or removals are well documented there, so you will have plenty of advance notice.
- Test thoroughly before production. Run new versions of your platform extensively in development and staging clusters before rolling out to production. Get the latest version of Kubernetes running in dev shortly after it is released and you will be able to thoroughly test and still have recent releases of Kubernetes running in production.

For the control plane nodes, they can be replaced additively. Using `kubeadm join` with the `--control-plane` flag on new machines that have the upgraded Kubernetes binaries—kubeadm, kubectl, kubelet—installed. As each of the control plane nodes comes online and is confirmed operational, one old-versioned node can be drained and then deleted. If you are running etcd colocated on the control plane nodes, include etcd checks when confirming operationality and etcdctl to manage the members of the cluster as needed.

Then you can proceed to replace the worker nodes. These can be done additively or subtractively—one at a time or several at a time. A primary concern here is cluster utilization. If your cluster is highly utilized, you will want to add new worker nodes before draining and removing existing nodes to ensure you have sufficient compute resources for the displaced Pods. Again, a good pattern is to use machine images that have all the updated software installed that are brought online and use `kubeadm join` to be added to the cluster. And, again, this could be implemented using many of the same mechanisms as used in cluster deployment. Figure 2-7 illustrates this operation of replacing control plane nodes one at a time and worker nodes in batches.

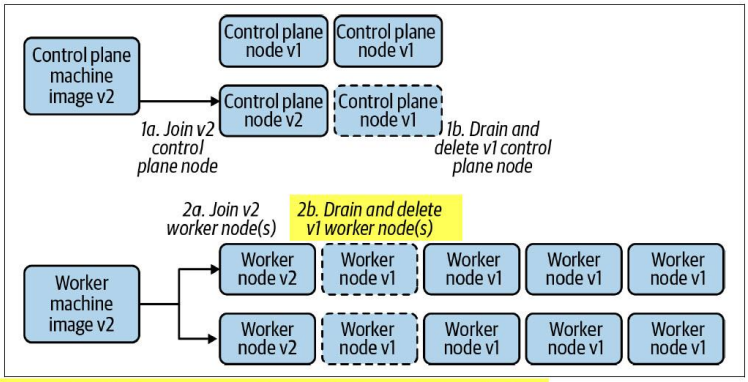


Figure 2-7. Performing upgrades by replacing nodes in a cluster.

3 **In-place upgrades**

In-place upgrades are appropriate in resource-constrained environments where replacing nodes is not practical. The rollback path is more difficult and, hence, the risk is higher. But this can and should be mitigated with comprehensive testing. Keep in mind as well that Kubernetes in production configurations is a highly available system. If in-place upgrades are done one node at a time, the risk is reduced. So, if using a config management tool such as Ansible to execute the steps of this upgrade operation, resist the temptation to hit all nodes at once in production.

❷ Specifies that an attach operation must be completed before volumes are mounted.

❸ Does not need to pass Pod metadata in as context when setting up a mount.

❹ The default model for provisioning persistent volumes. Inline support can be enabled by setting this option to `Ephemeral`. In the ephemeral mode, the storage is expected to last only as long as the Pod.

The settings and objects we have explored so far are artifacts of our bootstrapping process. The CSIDriver object makes for easier discovery of driver details and was included in the driver's deployment bundle. The CSINode objects are managed by the kubelet. A generic registrar sidecar is included in the node plug-in Pod and gets details from the CSI driver and registers the driver with the kubelet. The kubelet then reports up the quantity of CSI drivers available on each host. Figure 4-2 demonstrates this bootstrapping process.

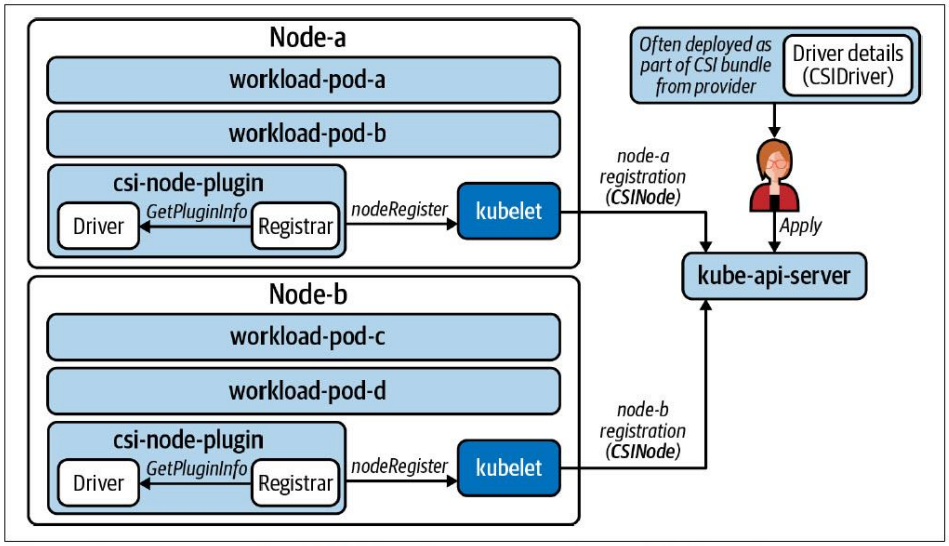


Figure 4-2. CSIDriver object is deployed and part of the bundle while the node plug-in registers with the kubelet. This in turn creates/manages the CSINode objects.

Exposing Storage Options

In order to provide storage options to developers, we need to create StorageClasses. For this scenario we'll assume there are two types of storage we'd like to expose.The first option is to expose cheap disk that can be used for workload persistence needs. Many times, applications don't need an SSD as they are just persisting some files that do not require quick read/write. As such, the cheap disk (HDD) will be the default

for preparing volumes to be consumed by Pods on the node. Often this means setting up the mounts and reporting information about volumes on the node. Both the node and controller service also implement identity services that report plug-in info, capabilities, and whether the plug-in is healthy. With this in mind, Figure 4-1 represents a cluster architecture with these components deployed.

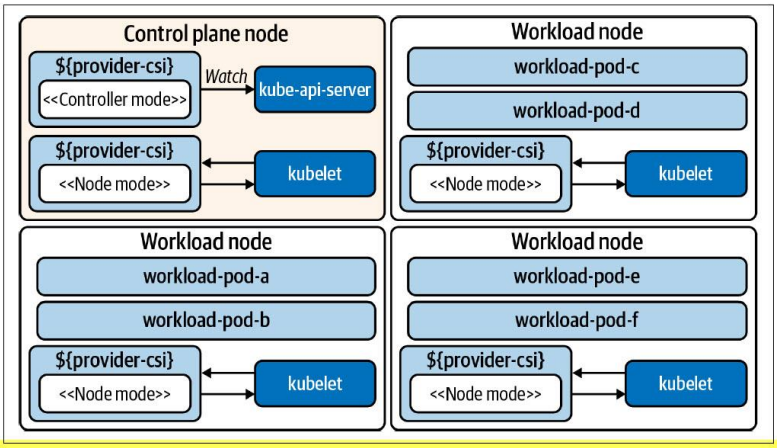


Figure 4-1. Cluster running a CSI plug-in. The driver runs in a node and controller mode. The controller is typically run as a Deployment. The node service is deployed as a DaemonSet, which places a Pod on each host.

Let's take a deeper look at these two components, the controller and the node.

CSI Controller

The CSI Controller service provides APIs for managing volumes in a persistent storage system. The Kubernetes control plane *does not* interact with the CSI Controller service directly. Instead, controllers maintained by the Kubernetes storage community react to Kubernetes events and translate them into CSI instructions, such as CreateVolumeRequest when a new PersistentVolumeClaim is created. Because the CSI Controller service exposes its APIs over UNIX sockets, the controllers are usually deployed as sidecars alongside the CSI Controller service. There are multiple external controllers, each with different behavior:

external-provisioner
: When PersistentVolumeClaims are created, this requests a volume be created from the CSI driver. Once the volume is created in the storage provider, this provisioner creates a PersistentVolume object in Kubernetes.

cluster to detect preallocated storage and expose it, automatically, as PersistentVolumes. It also provides some life cycle management capabilities such as deleting data upon destruction of the PersistentVolumeClaim.

There are multiple ways to achieve local storage that can lead you into a bad practice. For example, it can seem compelling to allow developers to use hostPath rather than needing to preprovision a local storage. `hostPath` enables you to specify a path on the host to bind to rather than having to use a PersistentVolume and PersistentVolumeClaim. This can be a huge security risk as it enables developers to bind to directories on the host, which can have a negative impact on the host and other Pods. If you desire to provide developers ephemeral storage that can withstand a Pod restart but not the Pod being deleted or moved to a different node, you can use EmptyDir. This will allocate storage in the filesystem managed by Kube and be transparent to the Pod.

```
    requests:
      storage: 14Gi
  storageClassName: performance-block ❷
```

❶ This will use the default storage class (`default-block`) and assume other defaults such as RWO and filesystem storage type.

❷ Ensure `performance-block` is requested to the driver rather than `default-block`.

Based on the StorageClass settings, these two will exhibit different provisioning behaviors. The performant storage (from `pvc1`) is created as an unattached volume in AWS. This volume can be attached quickly and is ready to use. The default storage (from `pv0`) will sit in a `Pending` state where the cluster waits until a Pod consumes the PVC to provision storage in AWS. While this will require more work to provision when a Pod finally consumes the claim, you will not be billed for the unused storage! The relationship between the claim in Kubernetes and volume in AWS can be seen in Figure 4-3.

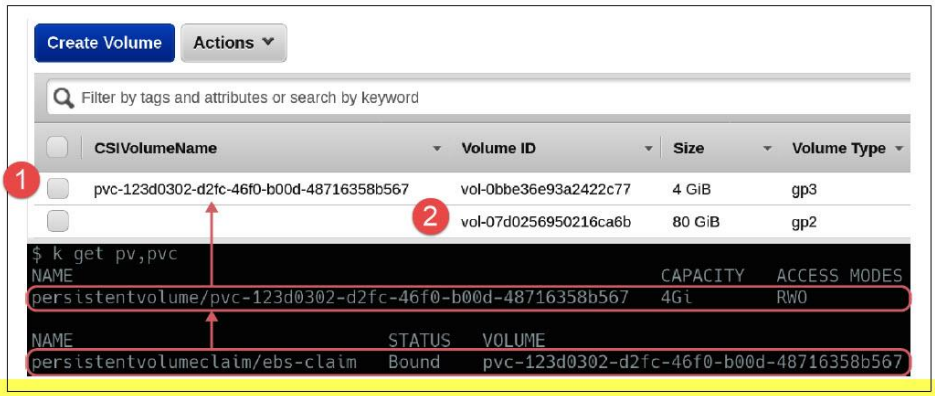


Figure 4-3. `pv1` is provisioned as a volume in AWS, and the CSIVolumeName is propagated for ease of correlation. `pv0` will not have a respective volume created until a Pod references it.

Now let's assume the developer creates two Pods. One Pod references `pv0` while the other references `pv1`. Once each Pod is scheduled on a Node, the volume will be attached to that node for consumption. For `pv0`, before this can occur the volume will also be created in AWS. With the Pods scheduled and volumes attached, a filesystem is established and the storage is mounted into the container. Because these are persistent volumes, we have now introduced a model where even if the Pod is rescheduled to another node, the volume can come with it. The end-to-end flow for how we've satisfied the self-service storage request is shown in Figure 4-4.

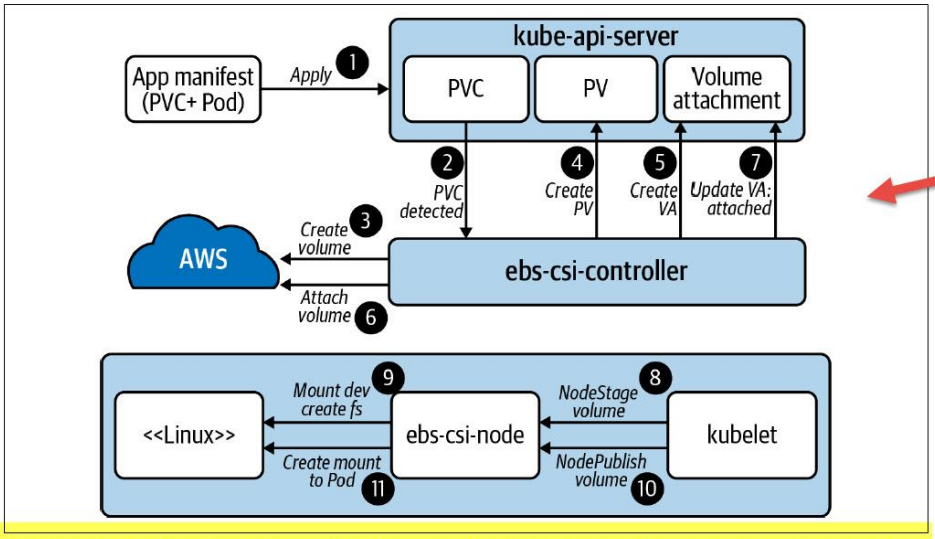


Figure 4-4. End-to-end flow of the driver and Kubernetes working together to satisfy the storage request.

Events are particularly helpful in debugging storage interaction with CSI. Because provisioning, attaching, and mounting are all happening in order to satisfy a PVC, you should view events on these objects as different components report what they have done. `kubectl describe -n $NAMESPACE pvc $PVC_NAME` is an easy way to view these events.

Resizing

Resizing is a supported feature in the `aws-ebs-csi-driver`. In most CSI implementations, the `external-resizer` controller is used to detect changes in PersistentVolumeClaim objects. When a size change is detected, it is forwarded to the driver, which will expand the volume. In this case, the driver running in the controller plug-in will facilitate expansion with the AWS EBS API.

Once the volume is expanded in EBS, the new space is *not* immediately usable to the container. This is because the filesystem still occupies only the original space. In order for the filesystem to expand, we'll need to wait for the node plug-in's driver instance to expand the filesystem. This can all be done *without* terminating the Pod. The filesystem expansion can be seen in the following logs from the node plug-in's CSI driver:

```
mount_linux.go: Attempting to determine if disk "/dev/nvme1n1" is formatted
using blkid with args: ([-p -s TYPE -s PTTYPE -o export /dev/nvme1n1])
```

Border Gateway Protocol (BGP) is one of the most commonly used protocols to distribute workload routes. It is used in projects such as Calico and Kube-Router. Not only does BGP enable communication of workload routes in the cluster but its internal routers can also be peered with external routers. Doing so can make external network fabrics aware of how to route to Pod IPs. In implementations such as Calico, a BGP daemon is run as part of the Calico Pod. This Pod runs on every host. As routes to workloads become known, the Calico Pod modifies the kernel routing table to include routes to each potential workload. This provides native routing via the workload IP, which can work especially well when running in the same L2 segment. Figure 5-2 demonstrates this behavior.

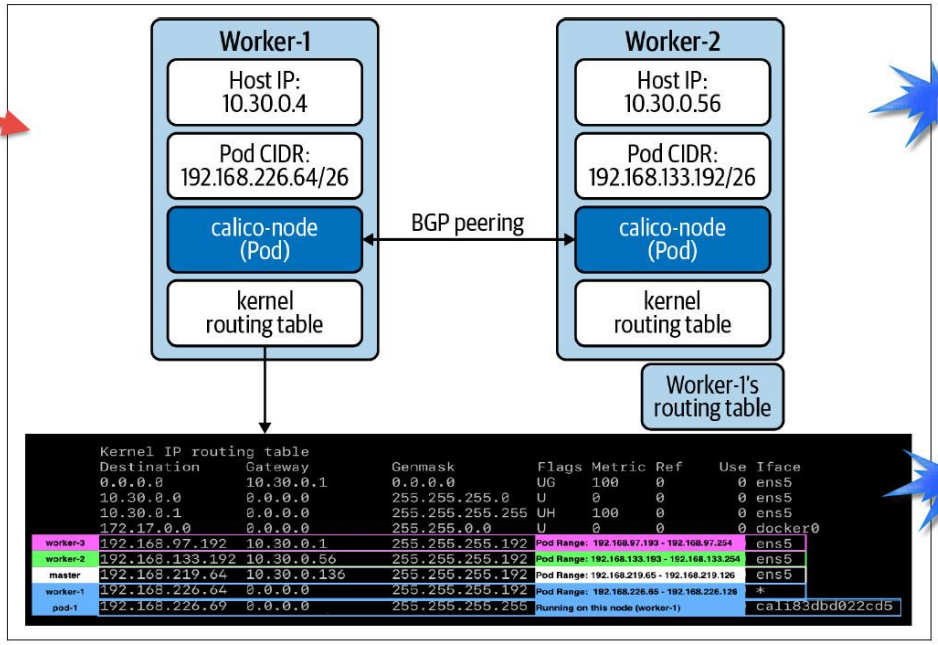


Figure 5-2. The `calico-pod` sharing routes via its BGP peer. The kernel routing table is then programmed accordingly.

Making Pod IPs routable to larger networks may seem appealing at first glance but should be carefully considered. See "Workload Routability" on page 108 for more details.

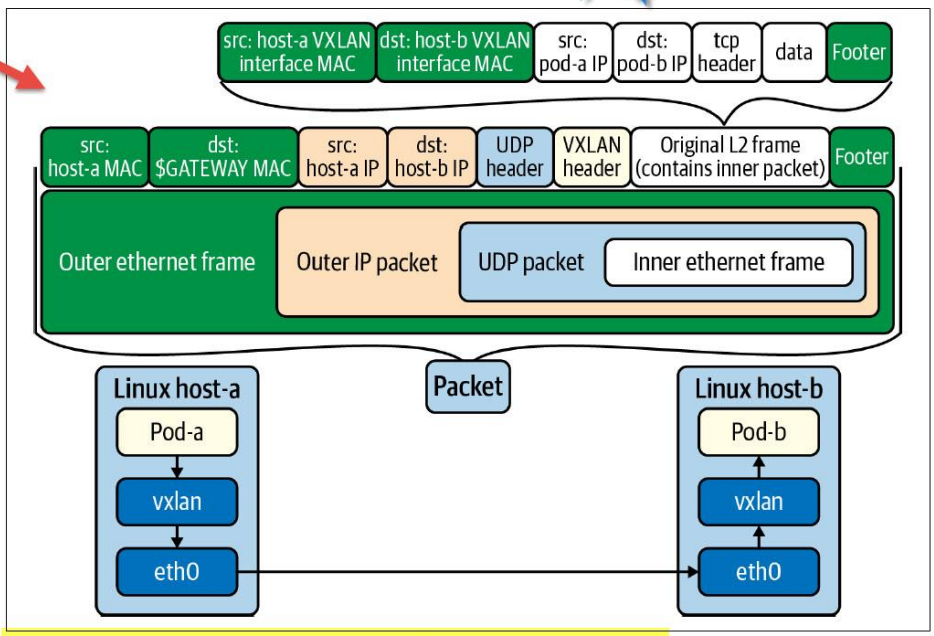


Figure 5-3. VXLAN encapsulation used to move an inner packet, for workloads, across hosts. The network cares only about the outer packet, so it needs to have zero awareness of workload IPs and their routes.

Often, you choose whether to use a tunneling protocol based on the requirements/capabilities of your environment. Encapsulation has the benefit of working in many scenarios since the overlay is abstracted from the underlay network. However, this approach comes with a few key downsides:

Traffic can be harder to understand and troubleshoot
: Packets within packets can create extra complexity when troubleshooting network issues.

Encapsulation/decapsulation will incur processing cost
: When a packet goes to leave a host it must be encapsulated, and when it enters a host it must be decapsulated. While likely small, this will add overhead relative to native routing.

Packets will be larger
: Due to the embedding of packets, they will be larger when transmitted over the wire. This may require adjustments to the maximum transmission unit (MTU) to ensure they fit on the network.

CNI Installation

CNI drivers must be installed on every node taking part in the Pod network. Additionally, the CNI configuration must be established. The installation is typically handled when you deploy a CNI plug-in. For example, when deploying Cilium, a DaemonSet is created, which puts a `cilium` Pod on every node. This Pod features a PostStart command that runs the baked-in script *install-cni.sh*. This script will start by installing two drivers. First it will install the loopback driver to support the `lo` interface. Then it will install the `cilium` driver. The script executes conceptually as follows (the example has been greatly simplified for brevity):

```
# Install CNI drivers to host

# Install the CNI loopback driver; allow failure
cp /cni/loopback /opt/cin/bin/ || true

# install the cilium driver
cp /opt/cni/bin/cilium-cni /opt/cni/bin/
```

After installation, the kubelet still needs to know which driver to use. It will look within */etc/cni/net.d/* (configurable via flag) to find a CNI configuration. The same *install-cni.sh* script adds this as follows:

```
cat > /etc/cni/net.d/05-cilium.conf <<EOF
{
  "cniVersion": "0.3.1",
  "name": "cilium",
  "type": "cilium-cni",
  "enable-debug": ${ENABLE_DEBUG}
}
EOF
```

To demonstrate this order of operations, let's take a look a newly bootstrapped, single-node cluster. This cluster was bootstrapped using `kubeadm`. Examining all Pods reveals that the `core-dns` Pods are not running:

```
NAMESPACE     NAME                              READY   STATUS    RESTARTS   AGE
kube-system   coredns-f9fd979d6-26lfr           0/1     Pending   0          3m14s
kube-system   coredns-f9fd979d6-zqzft           0/1     Pending   0          3m14s
kube-system   etcd-test                         1/1     Running   0          3m26s
kube-system   kube-apiserver-test               1/1     Running   0          3m26s
kube-system   kube-controller-manager-test      1/1     Running   0          3m26s
kube-system   kube-proxy-xhh2p                  1/1     Running   0          3m14s
kube-system   kube-scheduler-test               1/1     Running   0          3m26s
```

After examining the kubelet logs on the host scheduled to run `core-dns`, it becomes clear that the lack of CNI configuration is causing the container runtime to not start the Pod:

Calico

Calico is a well-established CNI plug-in in the cloud native ecosystem. Project Calico is the open source project that supports this CNI plug-in, and Tigera is the commercial company offering enterprise features and support. Calico makes heavy use of BGP to propagate workload routes between nodes and to offer integration with larger datacenter fabrics. Along with installing a CNI binary, Calico runs a `calico-node` agent on each host. This agent features a BIRD daemon for facilitating BGP peering between nodes and a Felix agent, which takes the known routes and programs them into the kernel route tables. This relationship is demonstrated in Figure 5-5.

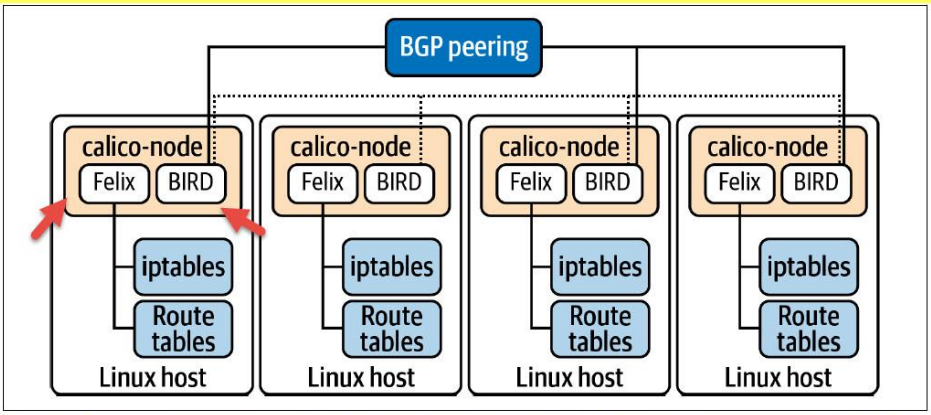


Figure 5-5. Calico component relationship showing the BGP peering to communicate routes and the programming of iptables and kernel routing tables accordingly.

For IPAM, Calico initially respects the `cluster-cidr` setting described in "IP Address Management" on page 102. However, its capabilities are far greater than relying on a CIDR allocation per node. Calico creates CRDs called IPPools. This provides a lot of flexibility in IPAM, specifically enabling features such as:

- Configuring block size per node
- Specifying what node(s) an IPPool applies to
- Allocating IPPools to Namespaces, rather than nodes
- Configuring routing behavior

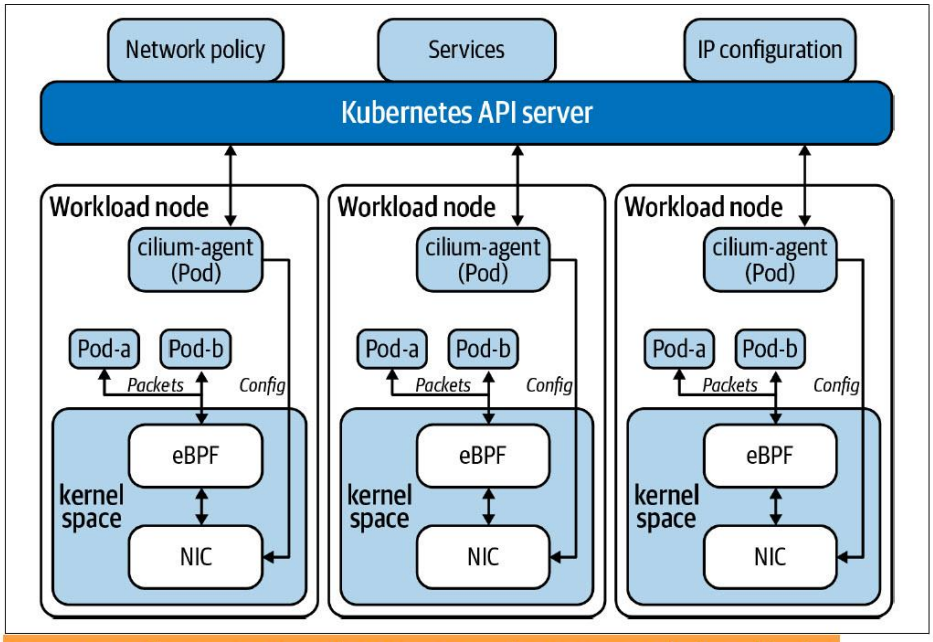


Figure 5-7. Cilium interacts with eBPF maps and programs at the kernel level.

For IPAM, Cilium follows the model of either delegating IPAM to a cloud provider integration or managing it itself. In the most common scenario of Cilium managing IPAM, it will allocate Pod CIDRs to each node. By default, Cilium will manage these CIDRs independent of Kubernetes Node allocations. The node-level addressing will be exposed in the `CiliumNode` CRD. This will provide greater flexibility in management of IPAM and is preferable. If you wish to stick to the default CIDR allocations done in Kubernetes based on its Pod CIDR, Cilium offer a `kubernetes` IPAM mode. This will rely on the Pod CIDR allocated to each node, which is exposed in the Node object. Following is an example of a `CiliumNode` object. You can expect one of these to exist for each node in the cluster:

```
apiVersion: cilium.io/v2
kind: CiliumNode
metadata:
  name: node-a
spec:
  addresses:
  - ip: 192.168.122.126 ❶
    type: InternalIP
  - ip: 10.0.0.245
    type: CiliumInternalIP
  health:
    ipv4: 10.0.0.78
```

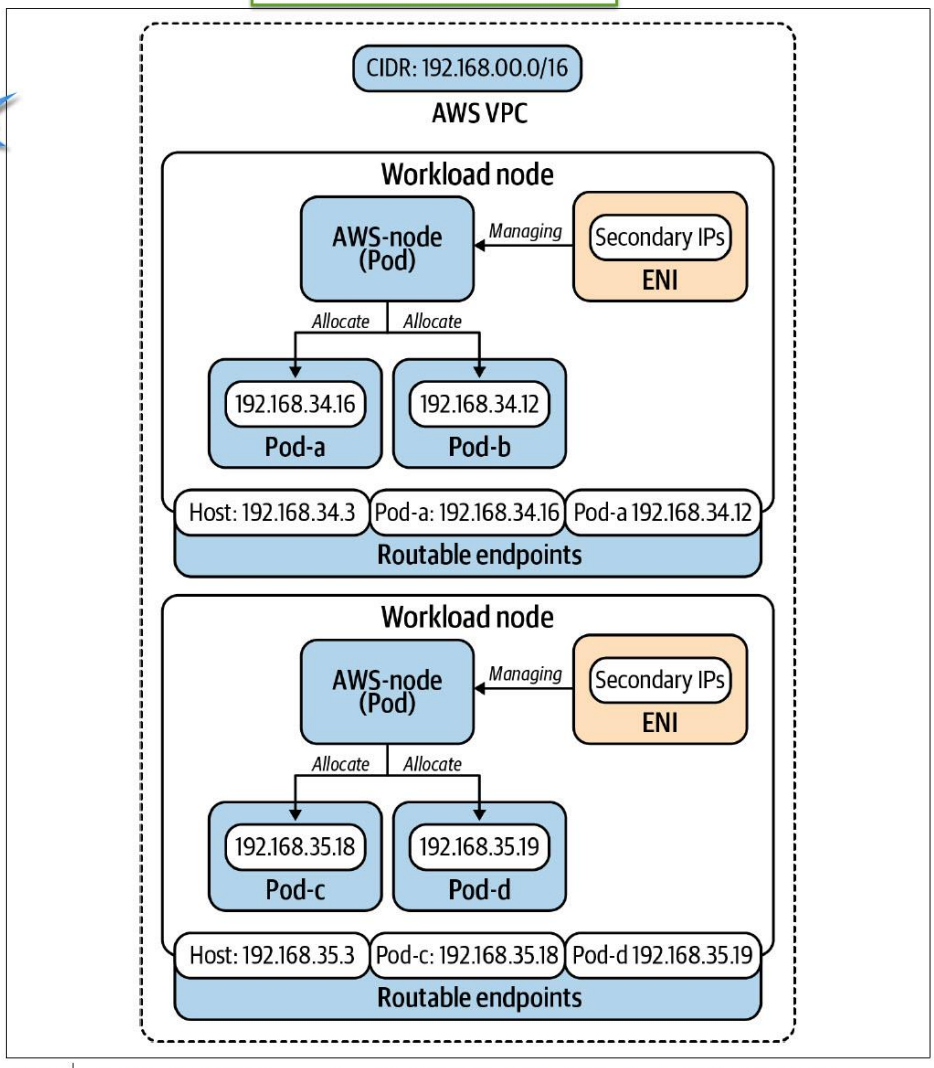


Figure 5-8. The IPAM daemon is responsible for maintaining the ENI and pool of secondary IPs.

The use of ENIs will impact the number of Pods you can run per node. AWS maintains a list on its GitHub page that correlates instance type to max Pods.

VERY IMPORTANT !!!

Linear scaling is an undesirable quality of any system. With that said, based on tests performed by the Kubernetes community, you should not notice any performance degradation unless you are running clusters with tens of thousands of Services. If you are operating at that scale, however, you might benefit from the IPVS mode in kube-proxy, which we'll discuss in the following section.

Rolling Updates and Service Reconciliation

An interesting problem with Services is unexpected request errors during application rolling updates. While this issue is less common in development environments, it can crop up in production clusters that are hosting many workloads.

The crux of the problem is the distributed nature of Kubernetes. As we've discussed in this chapter, multiple components work together to make Services work in Kubernetes, mainly the Endpoints controller and kube-proxy.

When a Pod is deleted, the following happens simultaneously:

- The kubelet initiates the Pod shutdown sequence. It sends a SIGTERM signal to the workload and waits until it terminates. If the process continues to run after the graceful shutdown period, the kubelet sends a SIGKILL to terminate the workload forcefully.
- The Endpoints controller receives a Pod deletion watch event, which triggers the removal of the Pod IP address from the Endpoints resource. Once the Endpoints resource is updated, kube-proxy removes the IP address from the iptables rules (or IPVS virtual service).

This is a classic distributed system race. Ideally, the Endpoints controller and kube-proxy finish their updates before the Pod exits. However, ordering is not guaranteed, given that these workflows are running concurrently. There is a chance that the workload exits (and thus stops accepting requests) before kube-proxy on each node removes the Pod from the list of active endpoints. When this happens, in-flight requests fail because they get routed to a Pod that is no longer running.

To solve this, Kubernetes would have to wait until all kube-proxies finish updating endpoints before stopping workloads, but this is not feasible. For example, how would it handle the case of a node becoming unavailable? With that said, we've used SIGTERM handlers and `sleep` pre-stop hooks to mitigate this issue in practice.

Kube-proxy: IP Virtual Server (IPVS) mode

IPVS is a load balancing technology built into the Linux kernel. Kubernetes added support for IPVS in kube-proxy to address the scalability limitations and performance issues of the iptables mode.

Running without kube-proxy

Historically, kube-proxy has been a staple in all Kubernetes deployments. It is a vital component that makes Kubernetes Services work. As the community evolves, however, we could start seeing Kubernetes deployments that do not have kube-proxy running. How is this possible? What handles Services instead?

With the advent of extended Berkeley Packet Filters (eBPF), CNI plug-ins such as Cilium and Calico can absorb kube-proxy's responsibilities. Instead of handling Services with iptables or IPVS, the CNI plug-ins program Services right into the Pod networking data plane. Using eBPF improves the performance and scalability of Services in Kubernetes, given that the eBPF implementation uses hash tables for endpoint lookups. It also improves Service update processing, as it can handle individual Service updates efficiently.

Removing the need for kube-proxy and optimizing Service routing is a worthy feat, especially for those operating at scale. However, it is still early days when it comes to running these solutions in production. For example, the Cilium implementation requires newer kernel versions to support a kube-proxy-less deployment (at the time of writing, the latest Cilium version is `v1.8`). Similarly, the Calico team discourages the use of eBPF in production because it is still in tech preview. (At the time of writing, the latest calico version is `v3.15.1`.) Over time, we expect to see kube-proxy replacements become more common. Cilium even supports running its proxy replacement capabilities alongside other CNI plug-ins (referred to as CNI chaining).

The downside to this approach is that environment variables cannot be updated without restarting the Pod. Thus, Services must be in place before the Pod starts up.

DNS Service Performance

As mentioned in the previous section, offering DNS-based service discovery to workloads on your platform is crucial. As the size of your cluster and number of applications grows, the DNS service can become a bottleneck. In this section, we will discuss techniques you can use to provide a performant DNS service.

DNS cache on each node

The Kubernetes community maintains a DNS cache add-on called NodeLocal DNSCache. The add-on runs a DNS cache on each node to address multiple problems. First, the cache reduces the latency of DNS lookups, given that workloads get their answers from the local cache (assuming a cache hit) instead of reaching out to the DNS server (potentially on another node). Second, the load on the CoreDNS servers goes down, as workloads are leveraging the cache most of the time. Finally, in the case of a cache miss, the local DNS cache upgrades the DNS query to TCP when reaching out to the central DNS service. Using TCP instead of UDP improves the reliability of the DNS query.

The DNS cache runs as a DaemonSet on the cluster. Each replica of the DNS cache intercepts the DNS queries that originate from their node. There's no need to change application code or configuration to use the cache. The node-level architecture of the NodeLocal DNSCache add-on is depicted in Figure 6-9.

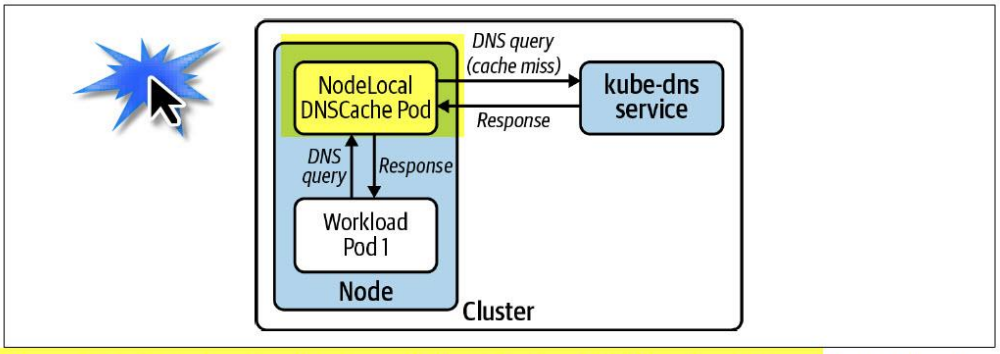


Figure 6-9. Node-level architecture of the NodeLocal DNSCache add-on. The DNS cache intercepts DNS queries and responds immediately if there's a cache hit. In the case of a cache miss, the DNS cache forwards the query to the cluster DNS service.

Auto-scaling the DNS server deployment

In addition to running the node-local DNS cache in your cluster, you can automatically scale the DNS Deployment according to the size of the cluster. Note that this strategy does not leverage the Horizontal Pod Autoscaler. Instead, it uses the cluster Proportional Autoscaler, which scales workloads based on the number of nodes in the cluster.

The Cluster Proportional Autoscaler runs as a Pod in the cluster. It has a configuration flag to set the workload that needs autoscaling. To autoscale DNS, you must set the target flag to the CoreDNS (or kube-dns) Deployment. Once running, the autoscaler polls the API server every 10 seconds (by default) to get the number of nodes and CPU cores in the cluster. Then, it adjusts the number of replicas in the CoreDNS Deployment if necessary. The desired number of replicas is governed by a configurable replicas-to-nodes ratio or replicas-to-cores ratio. The ratios to use depend on your workloads and how DNS-intensive they are.

1 In most cases, using node-local DNS cache is sufficient to offer a reliable DNS service. However, autoscaling DNS is another strategy you can use when autoscaling
2 clusters with a wide-enough range of minimum and maximum nodes.

Ingress

As we've discussed in Chapter 5, workloads running in Kubernetes are typically not accessible from outside the cluster. This is not a problem if your applications do not have external clients. Batch workloads are a great example of such applications. Realistically, however, most Kubernetes deployments host web services that do have end users.

Ingress is an approach to exposing services running in Kubernetes to clients outside of the cluster. Even though Kubernetes does not fulfill the Ingress API out of the box, it is a staple in any Kubernetes-based platform. It is not uncommon for off-the-shelf Kubernetes applications and cluster add-ons to expect that an Ingress controller is running in the cluster. Moreover, your developers will need it to be able to run their applications successfully in Kubernetes.

This section aims to guide you through the considerations you must make when implementing Ingress in your platform. We will review the Ingress API, the most common ingress traffic patterns that you will encounter, and the crucial role of Ingress controllers in a Kubernetes-based platform. We will also discuss different ways to deploy Ingress controllers and their trade-offs. Finally, we will address common challenges you can run into and explore helpful integrations with other tools in the ecosystem.

domain names (`tenantA.example.com` and `tenantB.example.com`, for example). We will further discuss how to implement subdomain-based routing in a later section.

Ingress Configuration Collisions and How to Avoid Them

The Ingress API is prone to configuration collisions in multiteam or multitenant clusters. The primary example is different teams trying to use the same domain name to expose their applications. Consider a scenario where an application team creates an Ingress resource with the host set to `app.bearcanoe.com`. What happens when another team creates an Ingress with the same host? The Ingress API does not specify how to handle this scenario. Instead, it is up to the Ingress controller to decide what to do. Some controllers merge the configuration when possible, while others reject the new Ingress resource and log an error message. In any case, overlapping Ingress resources can result in surprising behavior, and even outages!

Usually, we tackle this problem in one of two ways. The first is using an admission controller that validates the incoming Ingress resource and ensures the hostname is unique across the cluster. We've built many of these admission controllers over time while working in the field. These days, we use the Open Policy Agent (OPA) to handle this concern. The OPA community even maintains a policy for this use case.

The Contour Ingress controller approaches this with a different solution. The HTTPProxy Custom Resource handles this use case with *root* HTTPProxy resources and *inclusion*. In short, a root HTTPProxy specifies the host and then *includes* other HTTPProxy resources that are hosted under that domain. The idea is that the operator manages root HTTPProxy resources and assigns them to specific teams. For example, the operator would create a root HTTPProxy with the host `app1.bearcanoe.com` and *include* all HTTPProxy resources in the `app1` Namespace. See Contour's documentation for more details.

The Ingress API supports features beyond host-based routing. Through the evolution of the Kubernetes project, Ingress controllers extended the Ingress API. Unfortunately, these extensions were made using annotations instead of evolving the Ingress resource. The problem with using annotations is that they don't have a schema. This can result in a poor user experience, as there is no way for the API server to catch misconfigurations. To address this issue, some Ingress controllers provide Custom Resource Definitions (CRDs). These resources have well-defined APIs offering features otherwise not available through Ingress. Contour, for example, provides an `HTTPProxy` custom resource. While leveraging these CRDs gives you access to a broader array of features, you give up the ability to swap Ingress controllers if necessary. In other words, you "lock" yourself into a specific controller.

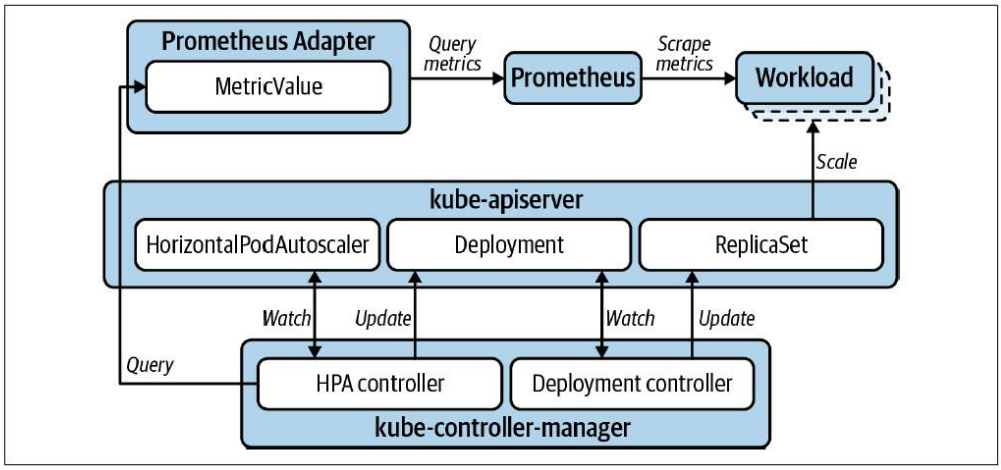


Figure 13-3. Horizontal Pod autoscaling with custom metrics.

Cluster Proportional Autoscaler

The Cluster Proportional Autoscaler (CPA) is a horizontal workload autoscaler that scales replicas based on the number of nodes (or a subset of nodes) in the cluster. So, unlike the HPA, it does not rely on any of the metrics APIs. Therefore, it does not have a dependency on the Metrics Server or Prometheus Adapter. Also, it is not configured with a Kubernetes resource, but rather uses flags to configure target workloads and a ConfigMap for scaling configuration. Figure 13-4 illustrates the CPA's much simpler operational model.

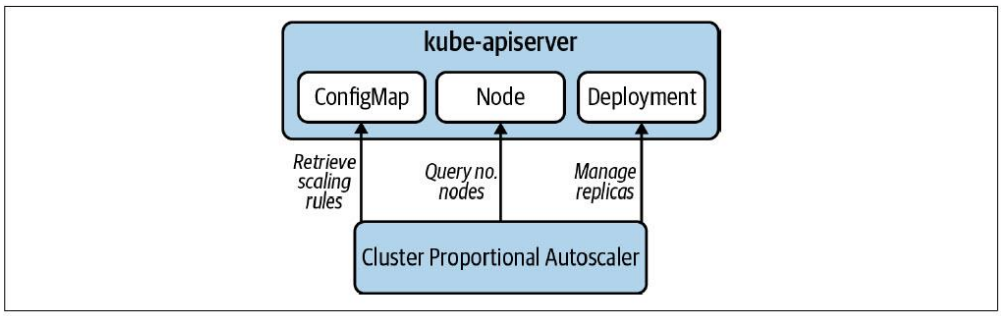


Figure 13-4. Cluster proportional autoscaling.

The CPA has a narrower use case. Workloads that need to scale in proportion to the cluster are generally limited to platform services. When considering the CPA, evaluate whether an HPA would provide a better solution, especially if you are already leveraging the HPA with other workloads. If you are already using HPAs, you will have the Metrics Server or Prometheus Adapter already deployed to implement the necessary metrics APIs. So deploying another autoscaler, and the maintenance overhead that goes with it, may not be the best option. Alternatively, in a cluster where HPAs

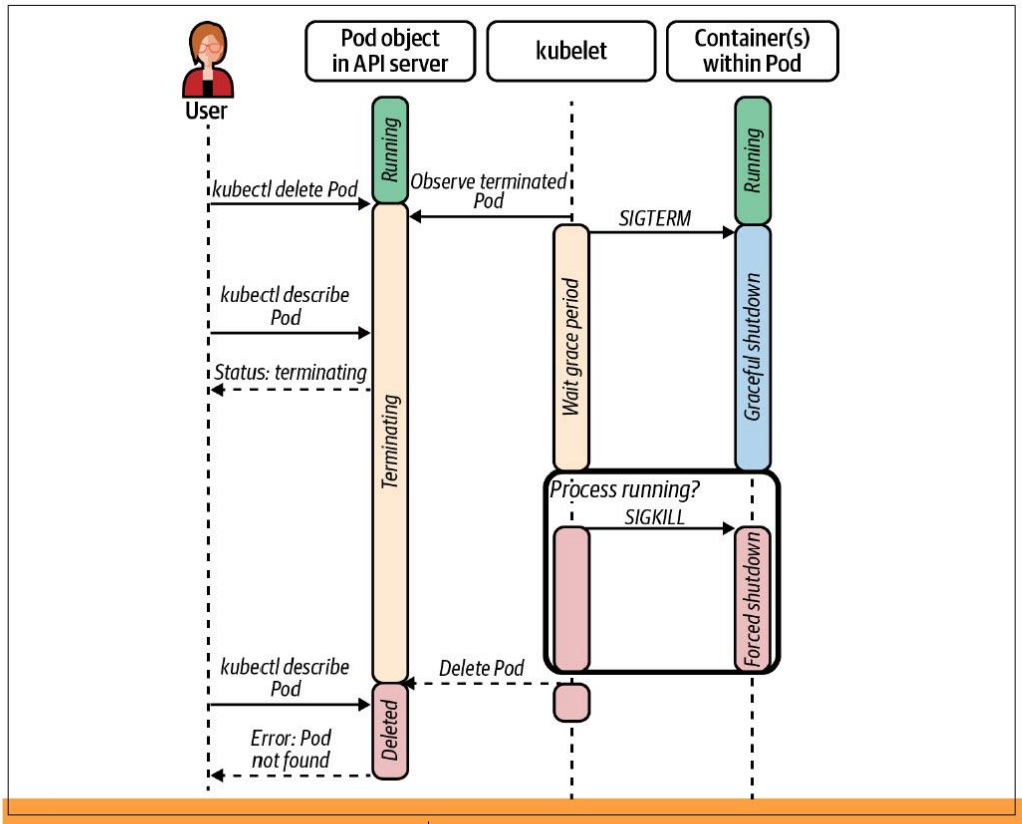


Figure 14-1. Application termination in Kubernetes. The kubelet first sends a SIGTERM signal to the workload and waits up to the configured graceful termination period. If the process is still running after the period expires, the kubelet sends a SIGKILL to terminate the process.

For your application to terminate gracefully, it must handle the SIGTERM signal. Each programming language or framework has its own way of configuring signal handlers. Some application frameworks might even take care of it for you. The following snippet shows a Go application that configures a SIGTERM signal handler, which stops the application's HTTP server upon receipt of the signal:

```
func main() {
        // App initialization code here...
        httpServer := app.NewHTTPServer()

        // Make a channel to listen for an interrupt or terminate signal
        // from the OS.

        // Use a buffered channel because the signal package requires it.
        shutdown := make(chan os.Signal, 1)
        signal.Notify(shutdown, os.Interrupt, syscall.SIGTERM)
```