Carnegie Mellon University
Software Engineering Institute

Kubernetes (k8s) in the Air Gap

Matthew Heckathorn & Maxwell Trdina

April 2025

DOI: 10.1184/R1/28792073

Introduction

When attempting to deploy containers across a cluster of machines, a container orchestrator is a required piece of technology. Container orchestrator tools provide a framework for managing and operating containers at scale. Some of the benefits of container orchestration tools include the following:

- providing deployment and configuration standards
- autoscaling
- auto healing
- service discovery
- load balancing

Of the few container orchestration tools available, k8s has emerged as the de facto industry standard. According to Allied Market Research, "Around 60% of respondents who have deployed containers in production are also using Kubernetes."[1] The *DoD DevSecOps Reference Design* uses k8s as its container orchestrator.[2] Additionally, the Cloud Native Computing Foundation's *CNCF Survey 2020*,[3] shows that the adoption of k8s in production has continued to grow.

This growth of k8s adoption has led to the development of a well-supported ecosystem of tools to help tackle numerous problems. However, deploying a k8s cluster in an air gap scenario requires deploying

---

1 See Allied Market Research's report *Container Orchestration Market Size, Share, Competitive Landscape and Trend Analysis Report, by Component, Organization Size and Industry Vertical : Global Opportunity Analysis and Industry Forecast, 2019-2026* (https://www.alliedmarketresearch.com/container-orchestration-market).

2 See the 2021 report *DoD Enterprise DevSecOps Reference Design: CNCF Kubernetes* (https://dodcio.defense.gov/Portals/0/Documents/Library/DoD%20Enterprise%20DevSecOps%20Reference%20Design%20-%20CNCF%20Kubernetes%20w-DD1910_cleared_20211022.pdf).

3 See Cloud Native Computing Foundation's *CNCF Survey 2020* (https://www.cncf.io/wp-content/uploads/2020/12/CNCF_Survey_Report_2020.pdf).

an additional service (i.e., a container registry mirror) and making some configuration changes to alter the behavior of how the k8s cluster sources container images.

The act of mirroring the required container images for a k8s deployment in the air gap was once a tedious and error-prone process. Luckily, thanks to the aforementioned ecosystem, this task has become increasingly simplified.

k8s at a High Level

At a high level, the basic architecture of k8s has two types of nodes: control plane nodes and worker nodes.

Control plane nodes are responsible for managing the worker nodes, and they run multiple processes that enable them to perform that task. These processes include *kube-apiserver, kube-controller-manager, kube-scheduler,* and *etcd.*

- The *kube-apiserver* is the process that exposes the k8s application programming interface (API), and the API allows interaction with the k8s control plane.
- The *kube-controller-manager* process is responsible for running various types of controller processes, including those that monitor k8s node health and create service accounts and API access tokens.
- The *kube-scheduler* process is responsible for selecting which k8s node an unassigned Pod should run on.[4]
- Finally, *etcd* is the key-value store that contains information about all the cluster data, including metadata and objects on the cluster.

Worker nodes are responsible for actually doing the work, and they too run a few processes that enable them to perform that task. These processes include the *kubelet, kube-proxy,* and the container runtime.

- The *kubelet* is an agent that runs on each node in the cluster and is responsible for ensuring that the required containers and Pods are running and healthy.
- The *kube-proxy* is the network proxy that handles k8s networking rules and communications.
- The container runtime is the software that actually runs the containers, and there are a number of supported container runtimes (e.g., Docker, containerd, cri-o).[5]

Sourcing Container Images

When deploying k8s to the air gap, the main issue to consider is how to provide the container images that make up a baseline k8s install. These core container images will vary depending on the choice of

---

[4] A *Pod* in k8s is a group of one or more containers.

[5] Container Runtime Interface (CRI) plus Open Container Initiative (OCI)

k8s engine, but most often these engines include images such as *coredns, kube-apiserver, kube-scheduler,* and *kube-proxy*. In an open, non-air-gapped install, these images will automatically be sourced from an internet connected container registry. Common container registries include Docker Hub (docker.io), registry.k8s.io, and ghcr.io.

Container registries are tools for storing and serving OCI-compliant container images over a network.[6] Registries present these images in a way that containerization tools like Docker understand. Using registries allows the images to be pulled into a remote system in a standardized way, where they can be run. Container registries can contain multiple repositories, which allow multiple different container images and different versions of the same image to be hosted.

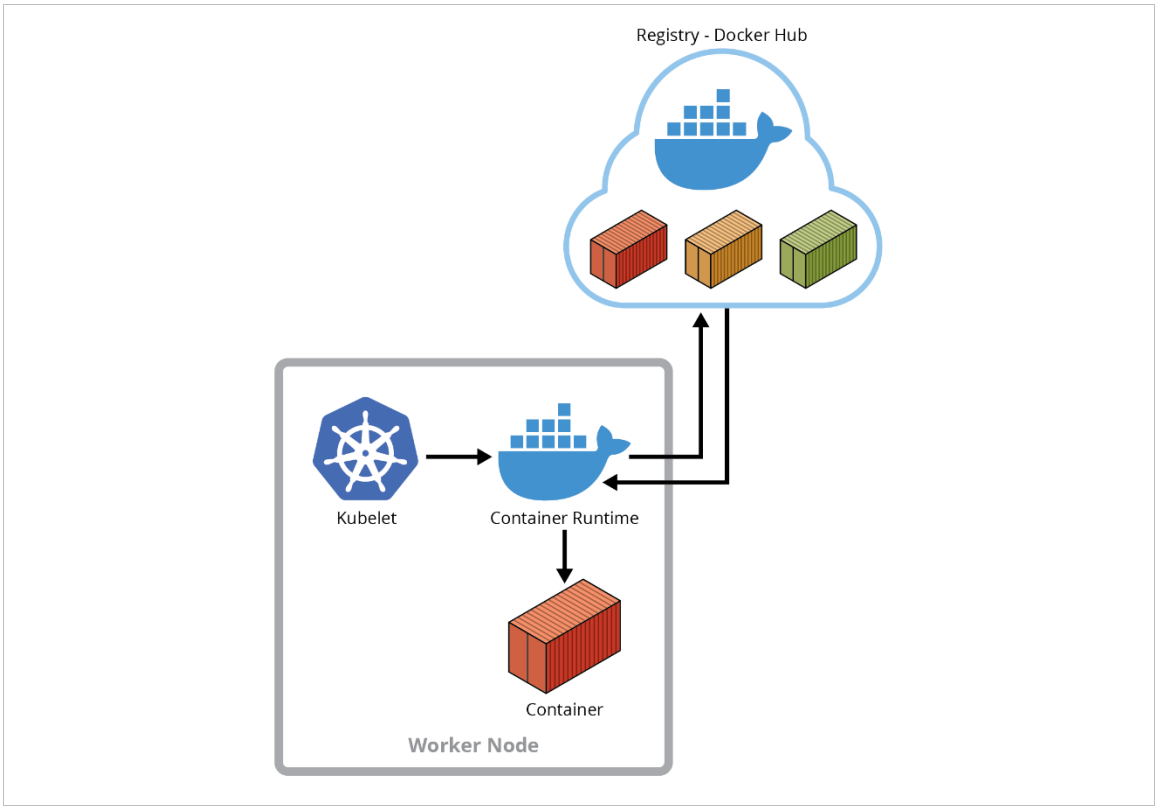

Figure 1: k8s Image Pull Flow

By using an image name, a container runtime is instructed on which image to use, and where to source it from. A full image name has the following format: **[HOST[:PORT_NUMBER]/]PATH**. In this paper, we focus on the **HOST** portion of the name. **HOST** is the optional registry hostname that

[6] OCI stands for the Open Container Initiative (https://opencontainers.org/).

specifies where the container image is located. If you don't specify a hostname, Docker's public registry is used by default (known as Docker Hub). For example:

```
registry.k8s.io/coredns/coredns
```

This image name instructs the container runtime to use the *coredns* container image located in the *coredns* namespace from the registry located at *registry.k8s.io*. Changing the **HOST** portion of the image name controls where the container is pulled from, which could include a self-hosted private container registry deployed in the air gap. For example:

```
192.168.0.1:5000/coredns/coredns
```

This image name instead instructs the container runtime to now pull the *coredns* container image from a private registry listening on port *5000* at the *192.168.0.1* IP. However, manually modifying the **HOST** portion of the image name to point to the air gapped container registry for all required container images is problematic:

- It is an error prone and time-consuming process.
- Doing so for the initial bootstrapping of a k8s cluster's core containers can be difficult and complicated.

Luckily, there is a feature that solves this issue not just for core container images, but for all containers on a k8s deployment.

Deploying k8s Without Internet Access Using Registry Mirroring

Registry mirroring eliminates the need to modify the image tags in your deployment by intercepting and redirecting image pull requests to an alternative container registry. In the case of an air gap deployment, this could be a local container registry configured with all the images needed for a deployment. Registry mirroring is a feature built into many containerization and container orchestration tools.

For example, both the containerd and cri-o container runtimes provide instructions for configuring registry mirroring. Within containerd, creating a *hosts.toml* file within the */etc/containerd/certs.d/_default* directory will create a default mirror for every registry.[7] With cri-o, a *registries.conf* file in */etc/containers/* allows for configuring registry mirrors.[8]

---

[7] See the setup default mirror for all registries for containerd on GitHub (https://github.com/containerd/containerd/blob/main/docs/hosts.md#setup-default-mirror-for-all-registries).

[8] See the Containers registry example on GitHub (https://github.com/containers/image/blob/main/docs/containers-registries.conf.5.md#example).

Talos and k3s also provide options for configuring registry mirroring settings. For example, in k3s, creating a yaml[9] file at */etc/rancher/k3s/registries.yaml* with the following content will transparently pull any Docker Hub hosted container from registry.example.com instead:

```
mirrors:
  docker.io:
    endpoint:
      - "https://registry.example.com:5000"
```

Talos provides the *--registry-mirror* option that can be used at cluster creation time. Configuration can also be done through a yaml file and the Talos configuration patch process.[10]

Accessing Other Open Container Assets

In addition to containers, it is useful to also be able to mirror other OCI artifacts, such as Helm Charts. Helm is a tool used for simplifying the deployment of complex k8s applications. Helm bundles complicated deployments into charts. These charts are pulled from artifact repositories accessible via the internet. Helm is an important tool in the k8s ecosystem, and many different charts are typically used to deploy applications to a k8s cluster. In an air gap scenario, these charts should also be mirrored and can be mirrored through a container registry that supports OCI artifacts.

Deploying an Air Gapped Registry

The tool landscape for deploying an air gapped container registry has matured in the past few years with two main tools: Hauler and Zarf. Both tools help simplify the deployment of a private registry in the air gap. This simplification is achieved through defining the required artifacts (e.g., containers, Helm Charts) as code and allows for reproducible, versioned actions. Once defined, both tools involve a similar process. On an internet-connected system, a fetch operation is run against the defined artifact set, which is then packaged into an archive. This archive, along with the self-contained command line tool for interacting with it, are sneaker-netted into the air gap. Once inside the air gap, the command line tool is used to load the artifact archive. The command line tool then deploys a private registry and serves the artifact content to other air gapped nodes.

Using Hauler, the process looks like the following:

1. On an internet-connected host, define a *hauler-manifest.yaml* file with the container images you want to take into the air gapped environment:

---

[9] The data serialization language named yaml stands for *yet another markup language*.

[10] See machine configuration information on the Talos Linux site (https://www.talos.dev/v1.9/talos-guides/configuration/pull-through-cache/#machine-configuration).

```
apiVersion: content.hauler.cattle.io/v1alpha1
kind: Images
metadata:
  name: hauler-content-images-example
spec:
  images:
    - name: <image tag>
      platform: <optional>
```

2. On the same host, use the Hauler CLI[11] tool to fetch, verify, and export the specified images into a *Haul*, a compressed archive file that Hauler understands:

```
# fetch the specified container images
hauler store sync --files hauler-manifest.yaml
# view and verify the content in the local hauler store
hauler store info
# save and export the content in the local hauler store
hauler store save --filename haul.tar.zst
```

3. Place both the exported Haul—in this case *haul.tar.zst*—and the Hauler CLI executable onto a flash drive or other portable media and connect it to the host in your air gapped environment that will serve as your container registry.

4. On the air gapped host, load the Haul and serve the Hauler registry:

```
# load and import the haul content to the new local hauler store
hauler store load haul.tar.zst
# serve the content as a registry from the hauler store
# defaults to <FQDN or IP>:5000
hauler store serve registry
```

Deploying an Air Gapped k8s Cluster

The tool landscape for deploying an on-premises k8s cluster has also greatly matured in the last few years. What was once a complicated, manual process has been greatly simplified through the proliferation of a variety of k8s *flavors* that target simplified on-premises installations. Some examples include rke2, k3s, and Talos. Regardless of the flavor of k8s, the deployment pattern remains the same: (1) deploy the air gapped container registry and (2) initialize the k8s cluster while passing in the appropriate registry mirror options.

[11] CLI is command line interface.

Using k3s, the process looks like the following:

1. Obtain the *k3s-images.txt* file from GitHub[12] for the release you are working with. Turn it into a Hauler manifest and create a *Haul* as previously outlined.
2. Obtain the *k3s* executable for the same version release as the *k3s-imagest.txt* file.[13]
3. Obtain the *install.sh* script from the get.k3s.io site at (https://get.k3s.io/).
4. Create a *registries.yaml* file and configure it to mirror the internet-connected registries of concern to your private registry. Below is an example:

```
mirrors:
  docker.io:
    endpoint:
      - "http://192.168.0.1:5000"
```

5. Sneaker-net all the content to the air gap. Deploy the Hauler private registry as previously laid out.
6. Copy the *k3s* release to */usr/local/bin* on each air gapped node for the k8s cluster and ensure it is executable.
7. Copy the *install.sh* script to each air gapped node for the k8s cluster and ensure it is executable.
8. Copy the *registries.yaml* file to */etc/rancher/k3s/registries.yaml* for each air gapped node in the k8s cluster.
9. Run the appropriate k3s bootstrap command on each node. For example, the following command will bootstrap a k3s server node, configure the flannel network to use the *eth0* network device, and pass the nodes IP address of 192.168.0.2 to various networking options:

```
INSTALL_K3S_SKIP_DOWNLOAD=true INSTALL_K3S_EXEC="server --
disable=traefik --flannel-backend=vxlan --flannel-iface eth0 --tls-
san=192.168.0.2 --bind-address=192.168.0.2 --advertise-
address=192.168.0.2 --node-external-ip=192.168.0.2 --node-
ip=192.168.0.2 --token=SECRET --cluster-init" ./install.sh
```

[12] See an example on the k3s-io GitHub site (https://github.com/k3s-io/k3s/releases/download/v1.32.1%2Bk3s1/k3s-images.txt).

[13] See an example executable on the k3s-io GitHub site (https://github.com/k3s-io/k3s/releases/download/v1.32.1%2Bk3s1/k3s).

The following example command bootstraps a k3s agent node and configures it to use the previous server node. It also passes the agent node's IP address (i.e., 192.168.0.4) to various networking options and configures the flannel network to use the agent node's *eth0* network device:[14]

```
INSTALL_K3S_SKIP_DOWNLOAD=true INSTALL_K3S_EXEC="agent --bind-
address=192.168.0.4 --flannel-iface eth0 --node-external-ip=192.168.0.4
--node-ip=192.168.0.4 --token=SECRET" K3S_URL=https://192.168.0.2:6443
./install.sh
```

Once completed, a completely air gapped k8s cluster will be deployed that is configured to source all of its container images, including any container images for applications hosted on the k8s cluster, from the private air gapped registry.

Test Procedure

The following subsections present a series of tests with escalating complexity that can be used to verify the successful deployment of an air gapped k8s cluster.

Basic k8s and Registry Mirror Validation

The first test is simple and is meant to validate two core functionalities:

- Pods can be deployed to k8s worker nodes, and logs are returned and accessible by *kubectl*.
- Images pulls are redirected to the local private registry by the registry mirror.

Steps

1. Retrieve the official Docker *hello-world* image from Dockerhub, Haul it to your air gapped environment, and stand up the Hauler registry mirror.
2. Run the following command on the host serving as the k8s master node:

   ```
   kubectl run test --restart=Never --image=hello-world -it
   ```

   This test was successful if the last command returns *Hello from Docker!* to the command line.

[14] More information about various k3s configuration options can be found on the k3s website (https://docs.k3s.io/installation/configuration).

Helm Chart Deployment Validation

The next test confirms that Helm Charts can be deployed from the private registry.

Steps

1. Add a test Helm Chart and any images it requires to the private registry, such as the Helm *hello-world* chart and its required image, *nginx:1.16.0.* You can add a Helm Chart to a Hauler store by declaring it in a Hauler manifest:

```
apiVersion: content.hauler.cattle.io/v1alpha1
kind: Images
metadata:
  name: hauler-content-images-hello-world
spec:
  images:
    - name: nginx:1.16.0
      platform: linux/arm64
---
apiVersion: content.hauler.cattle.io/v1alpha1
kind: Charts
metadata:
  name: hauler-content-charts-example
spec:
  charts:
    - name: hello-world
      repoURL: https://helm.github.io/example
```

2. Deploy the Helm Chart from the k8s master node:

```
helm install ahoy oci://192.168.0.1:5000/hauler/hello-world
```

Note: In our testing, we discovered that the Helm install command required additional flags to indicate which kubeconfig file to use and to use an unsecured connection, since we did not configure certificates for the registry in our proof of concept.

This test is successful if the Helm Chart is deployed properly with all the k8s objects it specifies, which include a Deployment, Pod, Service, and Service Account.[15] The following command will provide information about the *ahoy* Helm release mentioned above:

```
kubectl get all --all-namespaces -l='app.kubernetes.io/managed-
by=Helm,app.kubernetes.io/instance=ahoy'
```

[15] See chart objects for Helm in GitHub (https://github.com/helm/examples/tree/main/charts/hello-world/templates).

Multi-Pod Deployment Validation

This test deploys multiple Pods and validates that Pods can communicate to one another.

Steps

1. Find or create a simple multi-pod deployment example, such as the k8s Wordpress and MySQL with Persistent Volumes example.[16]
2. Identify all the images needed for the deployment and use Hauler to store and bring them to your air gapped deployment, along with any other kustomization files or other files needed.
3. Stand up the Hauler registry.
4. Complete your multi-pod deployment.

This test is successful if the Pods in your deployment can communicate with one another to provide the intended service.[17]

Conclusion

While once an arduous, time-consuming, and error-prone process, the deployment of an air gapped k8s cluster has been greatly simplified. This simplification is due to the proliferation and maturity of tooling in the k8s ecosystem. Tools, such as Hauler, have drastically improved the experience of deploying an air gapped container registry mirror.

The standardization and proper configuration of the registry mirror option in k8s flavors allows easier control over how clusters attempt to source container images. This feature is completely transparent, which means that container images are "magically" pulled from where they need to be. This also means that developers and k8s cluster operators do not need to maintain different versions of k8s manifests or Helm values.yaml files, depending on whether they are in the air gap or not, relieving that maintenance burden and speeding the time to deployment.

---

[16] See Example: Deploying WordPress and MySQL with Persistent Volumes (https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/)

[17] Configuring a PersistentVolume (PV) is outside the scope of this test.

Legal Markings

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM25-0292

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu