# Introduction to Docker Certified Infrastructure for Amazon Web Services

Docker Enterprise is the only enterprise-ready container platform that enables organizations to choose how to cost-effectively build and manage their entire application portfolio at their own pace, without fear of architecture and infrastructure lock-in. *Docker Certified Infrastructure* is Docker's prescriptive approach to deploying Docker Enterprise on a range of infrastructure choices. This Docker Certified Infrastructure Reference Architecture documents our best practice guidance for running the Docker container platform on your selected infrastructure.

In conjunction with the Reference Architecture we also publish Ecosystem Solution Briefs (https://success.docker.com/article/certified-infrastructures-aws#dockersolutionbriefs) to help you integrate Docker Enterprise with popular 3rd party tools used in conjunction with our container platform.

Amazon Web Services is a subsidiary of Amazon.com that provides on-demand cloud computing platforms. The associated tooling tied to this reference architecture provides a Docker Enterprise environment which conforms to both Docker and Amazon best practices.

# Reference Architecture

## What You Will Learn

Many enterprise organizations following a hybrid cloud strategy to deploy a scalable container platform on Amazon Web Services (AWS). When doing so, the platform must be deployed in such a way that it makes the most out of the cloud platform. This reference architecture provides best practices and architecture considerations for deploying, scaling, and managing a Docker Enterprise environment on Amazon Web Services. It is designed to provide the reader with the knowledge to deploy new AWS resources or modify existing ones, in order to create the best platform on which to deploy Docker Enterprise.

It describes:

- How to correctly architect and deploy Docker Enterprise on Amazon Web Services
- How to take advantage of features around compute, storage, and networking available on Amazon Web Services to provide the best experience of Docker Enterprise

It provides a checklist of tasks and procedures to ensure that your platform is configured correctly **before** you begin an installation of Docker Enterprise. It details all of the components of both the Amazon Web Services and the Docker Enterprise environment to ensure your platform is both understood and built from supported components.

Additionally, advanced configuration such as the use of storage/networking plugins is explored. A subsequent section details how to scale the platform as application requirements grow, and, finally, there are a number of troubleshooting procedures should there be issues deploying the Docker Enterprise platform.

Installation of Docker Enterprise is not covered in this reference architecture, but detailed installation instructions are provided for each supported operating system at docs.docker.com (https://docs.docker.com/ee/supported-platforms/#on-premises).

# Hardware Configurations

Deploying Docker Enterprise on Amazon Web Services makes use of the compute, networking, and storage resources available in the cloud platform. This guide is designed for a cloud-only deployment with no components of Docker Enterprise deployed outside Amazon Web Services.

To provide an environment that is suitable for High Availability and guarantee the performance of the platform, the following design considerations need to be evaluated:

- Deploy to enough Availability Zones (https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html#concepts-regions-availability-zones) to ensure that there is N+1 resiliency in the event of unplanned hardware failure or planned platform maintenance. Not all regions have 3 Availability Zones. To have a Highly Available cluster you **must** choose a region with 3 Availability Zones. Deploy to the AWS Region (https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html) that offers the lowest latency to the most common users of the platform. There are tools such as AWS Speed (https://cloudharmony.com/speedtest-for-aws) to discover the closest AWS regions to your users. Provide logical separation between management and application compute, network, and storage resources. Determine backing storage for the virtual machines that will host Docker Enterprise. Not only does the management platform need to be considered carefully (large scale storage for Docker Trusted Registry), but also think about the requirements of the applications and services that will be deployed on top of Docker Enterprise.

# Virtual Machine Sizing Recommendations

The hardware requirements for a Docker Enterprise cluster are based upon having sufficient resources for both the Docker Enterprise management platform (including load balancers and additional management tooling) as well as the requirements for the containerized applications that will be hosted on the Docker Enterprise platform. This reference architecture is designed around the best practices for deploying Docker Enterprise; the application and operations teams should be consulted for the sizing around the application hardware requirements.

To provide a highly available platform for Docker Enterprise, additional hardware requirements should be considered in the event of hardware failure or upgrades.

The minimum and recommend system requirements for a UCP cluster can be found on docs.docker.com (https://docs.docker.com/datacenter/ucp/3.0/guides/admin/install/system-requirements/).

These Docker Enterprise hardware requirements coordinate with the following AWS EC2 instance types:

| Requirement Type | Node Type | EC2 Instance Type |
|---|---|---|
| Minimum | Manager (UCP and DTR) | `m4.large, m5.large, c5.xlarge, t2.large` |
| Minimum | Worker | `m3.medium, m4.medium, c3.large, c4.large, c5.large, t2.medium` |
| Recommended | Manager (UCP and DTR) | `m4.xlarge, m5.xlarge, t2.xlarge` |
| Recommended | Worker | Depends on application workloads |

> **Note:** When sizing workers, understanding the resource requirements of your applications will help guide the worker sizes. Are the applications CPU bound? Are they memory bound? Or are they I/O bound? Also worth considering is the desired tenancy of the work nodes.
>
> T2 instance types bill extra for burst performance above a certain baseline (https://aws.amazon.com/about-aws/whats-new/2014/07/01/introducing-t2-the-new-low-cost-general-purpose-instance-type-for-amazon-ec2/). This would make the T2 instances better suited for development, temporary environments, or if you have bursty application workloads.

# Gathering Inventory from Amazon Web Services

There are two options to gather information from Amazon Web Services. This may be done, for example, to validate a deployment has been completed successfully.

First, you can visit the AWS Portal (https://aws.amazon.com/). Use the EC2 menu in the AWS console to view your newly created resources.

The second option is to use the AWS CLI (https://aws.amazon.com/cli/) to view and modify the Docker Enterprise environment.

# AWS Environment and Components

AWS provides a flexible set of cloud resources that can be deployed and configured to ensure that workloads perform effectively. This section covers the key Amazon Web Services resources that should be configured to provide the best experience for running Docker Enterprise on Amazon Web Services.

## AWS Prerequisites

Before Docker Enterprise can be successfully deployed onto Amazon Web Services there are a number of architectural choices that need to be considered. These range from providing access level permissions to network considerations to storage considerations.

## Subscriptions

Unless you are building a large cluster, the default resource limits on most AWS accounts should be adequate for building out most Docker Enterprise clusters.

> **Note:** Depending on the size of the environment that is being built, an increase in account limits may be required. Please see AWS Service Limits (https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html) for more details.

## Amazon Web Services Networking

When selecting a region to deploy Docker Certified Infrastructure templates and scripts on AWS, it's important to choose a region which contains three Availability Zones - a High Availability environment can only be created in regions with three Availability Zones. Most AWS regions have three Availability Zones, however if a High Availability environment is not required, then any region will be OK.

More details on AWS regions and Availability Zones can be found on aws.amazon.com (https://aws.amazon.com/about-aws/global-infrastructure/).

## Ports Used

All traffic within the VPC is open from host to host. From outside of the VPC the following ports are opened:

| Hosts | Direction | Port | Purpose |
|---|---|---|---|
| managers, workers | in | TCP 22 (configurable) | SSH access to the hosts |
| managers, workers | in | TCP 443 (configurable) | UCP web UI and API |
| managers, workers | in | TCP 3389 (configurable) | RDP, close this port and comment out after verifying installation |
| managers, workers | in | TCP 8000 (configurable) | HRM HTTP |
| managers, workers | in | TCP 8443 (configurable) | HRM HTTPS |

# Recommended APIs to Monitor

To programmatically integrate Docker UCP monitoring with existing solutions there are a number of API endpoints that can be accessed in order to determine both the state of the running services along with the Docker Enterprise platform itself.

**Determine the state of the platform**

To get a simple overview of the platform the URI endpoint `/_ping` will return the HTTP status code of the Docker UCP components.

| HTTP Status Code | Reason |
|---|---|
| 200 | **Success**, manager healthy |
| 500 | **Failure**, manager unhealthy |
| Success, manager healthy | default |

**Return a list of events**

To return a list of events that have happened on the Docker Enterprise platform the endpoint `/events` returns a parsable JSON response that can be evaluated to determine what has happened on the Docker Enterprise platform.

**Examine a particular service**

The endpoint `/services/{id}` allows an end-user to programmatically evaluate the health and status of a particular service that is running on the Docker Enterprise platform.

Further information about the API endpoints are available at docs.docker.com (https://docs.docker.com/datacenter/ucp/3.0/reference/api/#/).

# Log Collection and Frequency

Part of deploying Docker Enterprise is configuring logging. Logging provides visibility into the health and performance of both the platform and the services that run on top of it. It is recommended that you send logs to a log aggregator that provides ease of search as well as levels of observability that allow for quick troubleshooting of instability issues.

A reference architecture that details logging design and best practices is available in Docker Logging Design and Best Practices (https://success.docker.com/article/logging-best-practices).

Additionally, Docker provides a number of solutions briefs for integrating Docker logging with other platforms:

- Splunk Enterprise Solution Brief for Docker Enterprise (https://success.docker.com/article/splunk-logging/)
- Logging with Elasticsearch, Logstash, and Kibana Solution Brief for Docker Enterprise (https://success.docker.com/article/elasticsearch-logstash-kibana-logging/)

# Monitoring Docker Enterprise with Prometheus

Currently only the Docker engines can be monitored with Prometheus and not the management platform as a whole. To monitor the Docker Enterprise engines, modify their configurations to expose the metrics. The instructions for enabling Prometheus metrics are available in (https://success.docker.com/article/grafana-prometheus-monitoring/)docs.docker.com (https://docs.docker.com/config/thirdparty/prometheus/) as well as in Grafana/Prometheus Monitoring Solution Brief for Docker Enterprise 17.06.

# Storage

When choosing the EBS storage for the nodes, Amazon provides several different EBS storage types (https://aws.amazon.com/ebs/details/). Due to the transactional nature of the management nodes, `io1` or `gp2` should be used for both UCP and DTR nodes. For the worker nodes, it depends on the nature of your application workloads.

## AWS AMI Templates

One of the **key** requirements for providing a platform that Docker Enterprise can be deployed upon is pre-built virtual machine images. Amazon provides a number of pre-built AMIs, though you may wish to use your own. There are a number of requirements to these images that need to be observed for a successful Docker Enterprise deployment.

Docker Enterprise is validated and supported to work on the OS distributions found on the compatibility matrix (https://success.docker.com/article/compatibility-matrix).

Often, the public AMI provided by an OS vendor on AWS is sufficient to proceed. If you're planning on utilizing your own AMI, be sure that the following requirements are met:

### Required Machine Template Configuration

Ideally a virtual machine image for Docker Enterprise should be based upon a minimal package deployment (dependent on distribution) to ensure that no GUI tooling or programming toolchains are present within the template. Keeping the template as simple as possible both produces an efficient template for multiple deployments as well as provides a virtual machine with the smallest attack vector. It is also recommended that virtual machine templates are tested against the Center for Internet Security (https://www.cisecurity.org) benchmarks; this will provide a report of detected issues that can be corrected before the template is deemed production ready.

Further image requirements:

- At least 40GB for the operating system; there may be additional space required depending on the use case for the virtual machine deployed (DTR image storage, for example).

- The **openSSH** daemon installed and keys created for a user with the following permissions:
  - Modify firewall rules Modify filesystem (mount, create filesystems) Install
  - packages Enable system services (`init.d` / `systemd`)
- Python installed (optional)
- Password-less sudo (optional)

  Generate a key-pair and add the public key in the template's `authorized_keys` file (`/etc/ssh/authorized_keys` or `.ssh/authorized_keys`).

  An SSH agent helps to avoid retyping passwords:

  ```
  $ ssh-add <path to the private key>
  ```

  > See ssh-keygen (https://man.openbsd.org/ssh-keygen), ssh-agent (https://man.openbsd.org/ssh-agent), and ssh-add (https://man.openbsd.org/ssh-add) for complimentary information.

- Agents that provide operating system monitoring or data collection should be configured to ensure they don't perform any degradation to the Docker Engine/UCP/DTR.
- **OPTIONAL**: The template should be created with the correct number of network interfaces, and those interfaces should be configured so that they're placed onto the correct subnets and virtual networks.

# Persistent Storage

Many applications have one or more components that require data to persist for a myriad of reasons, such as data resiliency in the event of application failure. The following guidelines will help meet these data requirements for many use cases.

## Locally Presented Storage

Provision the following partitions as separate EBS volumes. This can vary depending on AMIs used.

| Mount path | Size | Purpose |
|---|---|---|
| `/` | 8 GB | Root partition |
| `/var/lib/docker` | 100 GB | Docker data volume |

## Storage Ecosystem

Docker Enterprise storage capabilities can be extended through the use of plugins. There are a number of Docker storage plugins that support third party storage devices and external management platforms.

For example, Cloudstor is a modern volume plugin built by Docker. Docker Swarm mode tasks and regular Docker containers can use a volume created with Cloudstor to mount a persistent data volume.

### What is Cloudstor?

Cloudstor is a modern volume plugin built by Docker. Docker Swarm mode tasks and regular Docker containers can use a Cloudstor volume as a persistent storage. Cloudstor has two `backing` options:

- `relocatable` data volumes are backed by EBS.
- `shared` data volumes are backed by EFS.

When the Docker CLI is used to create a Swarm service along with the persistent volumes used by the service tasks, you can create three different types of persistent volumes:

- Unique `relocatable` Cloudstor volumes mounted by each task in a Swarm service.
- Global `shared` Cloudstor volumes mounted by all tasks in a Swarm service.
- Unique `shared` Cloudstor volumes mounted by each task in a Swarm service.

Examples of each type of volume are described in the following sections.

## Relocatable Cloudstor Volumes

Workloads running in a Docker service that require access to low latency/high IOPs persistent storage, such as a database engine, can use a `relocatable` Cloudstor volume backed by EBS. When you create the volume, you can specify the type of EBS volume appropriate for the workload (such as `gp2`, `io1`, `st1`, `sc1`). Each `relocatable` Cloudstor volume is backed by a single EBS volume.

If a Swarm task using a `relocatable` Cloudstor volume gets rescheduled to another node within the same availability zone as the original node where the task was running, Cloudstor detaches the backing EBS volume from the original node and attaches it to the new target node automatically.

If the Swarm task gets rescheduled to a node in a different availability zone, Cloudstor transfers the contents of the backing EBS volume to the destination availability zone using a snapshot and cleans up the EBS volume in the original availability zone. To minimize the time necessary to create the snapshot to transfer data across availability zones, Cloudstor periodically takes snapshots of EBS volumes to ensure there is never a large number of writes that need to be transferred as part of the final snapshot when transferring the EBS volume across availability zones.

Typically the snapshot-based transfer process across availability zones takes between 2 and 5 minutes unless the workload is write-heavy. For extremely write-heavy workloads generating several GBs of fresh/new data every few minutes, the transfer may take longer than 5 minutes. The time required to snapshot and transfer increases sharply beyond 10 minutes if more than 20 GB of writes have been generated since the last snapshot interval. A Swarm task is not started until the volume it mounts becomes available.

Sharing/mounting the same Cloudstor volume backed by EBS among multiple Docker Swarm Tasks is not a supported scenario and leads to data loss. If you need a Cloudstor volume to share data between Swarm tasks, choose the appropriate EFS backed `shared` volume option. Using a `relocatable` Cloudstor volume backed by EBS is supported on all AWS regions that support EBS. The default `backing` option is `relocatable` if EFS support is not selected during setup/installation or if EFS is not supported in a region.

## Shared Cloudstor Volumes

When multiple Swarm service tasks need to share data in a persistent storage volume, you can use a `shared` Cloudstor volume backed by EFS. Such a volume and its contents can be mounted by multiple Swarm service tasks without the risk of data loss, since EFS makes the data available to all Swarm nodes over NFS.

When Swarm tasks using a `shared` Cloudstor volume get rescheduled from one node to another within the same or across different availability zones, the persistent data backed by EFS volumes is always available. `shared` Cloudstor volumes only work in those AWS regions where EFS is supported. If EFS Support is selected during setup/installation, the default "backing" option for Cloudstor volumes is set to `shared` so that EFS is used by default.

`shared` Cloudstor volumes backed by EFS (or even EFS MaxIO) may not be ideal for workloads that require very low latency and high IOPSs. For performance details of EFS backed `shared` Cloudstor volumes, see the AWS performance guidelines (http://docs.aws.amazon.com/efs/latest/ug/performance.html).

## Use Cloudstor

After initializing or joining a Swarm, connect to any Swarm manager using SSH. Verify that the Cloudstor plugin is already installed and configured for the stack or resource group:

```
$ docker plugin ls

ID                  NAME                DESCRIPTION                         ENABLED
f416c95c0dcc        cloudstor:aws       cloud storage plugin for Docker     true
```

The following examples show how to create Swarm services that require data persistence using the `--mount` flag and specifying Cloudstor as the volume driver.

## Share the Same Volume among Tasks using EFS

In those regions where EFS is supported and enabled, you can use `shared` Cloudstor volumes to share access to persistent data across all tasks in a Swarm service running in multiple nodes, as in the following example:

```
$ docker service create \
  --replicas 5 \
  --name ping1 \
  --mount type=volume,volume-driver=cloudstor:aws,source=sharedvol1,destination=/shareddata \
  alpine ping docker.com
```

All replicas/tasks of the service `ping1` share the same persistent volume `sharedvol1` mounted at `/shareddata` path within the container. Docker Enterprise interacts with the Cloudstor plugin to ensure that EFS is mounted on all nodes in the Swarm where service tasks are scheduled. Your application needs to be designed to ensure that tasks do not write concurrently on the same file at the same time, to protect against data corruption.

You can verify that the same volume is shared among all the tasks by logging into one of the task containers, writing a file under `/shareddata/`, and logging into another task container to verify that the file is available there as well.

The only option available for EFS is `perfmode`. You can set `perfmode` to `maxio` for high I/O throughput:

```
$ docker service create \
  --replicas 5 \
  --name ping3 \
  --mount type=volume,volume-driver=cloudstor:aws,source={{.Service.Name}}-{{.Task.Slot}}-
vol5,destination=/mydata,volume-opt=perfmode=maxio \
  alpine ping docker.com
```

You can also create `shared` Cloudstor volumes using the `docker volume create` CLI:

```
$ docker volume create -d "cloudstor:aws" --opt backing=shared mysharedvol1
```

## Use a Unique Volume per Task using EBS

If EBS is available and enabled, you can use a templatized notation with the `docker service create` CLI to create and mount a unique `relocatable` Cloudstor volume backed by a specified type of EBS for each task in a Swarm service. New EBS volumes typically take a few minutes to be created. Besides `backing=relocatable`, the following volume options are available:

| Option | Description |
|---|---|
| `size` | Required parameter that indicates the size of the EBS volumes to create in GB. |
| `ebstype` | Optional parameter that indicates the type of the EBS volumes to create (`gp2`, `io1`, `st1`, `sc1`}. The default `ebstype` is Standard/Magnetic. For further details about EBS volume types, see the EBS volume type documentation (http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html). |
| `iops` | Required if `ebstype` specified is `io1`, which enables provisioned IOPs. Needs to be in the appropriate range as required by EBS. |

Example usage:

```
$ docker service create \
  --replicas 5 \
  --name ping3 \
  --mount type=volume,volume-driver=cloudstor:aws,source={{.Service.Name}}-{{.Task.Slot}}-
vol,destination=/mydata,volume-opt=backing=relocatable,volume-opt=size=25,volume-opt=ebstype=gp2 \
  alpine ping docker.com
```

The above example creates and mounts a distinct Cloudstor volume backed by 25 GB EBS volumes of type `gp2` for each task of the `ping3` service. Each task mounts its own volume at `/mydata/`, and all files under that mount point are unique to the task mounting the volume.

It is highly recommended that you use the `.Task.Slot` template to ensure that task `N` always gets access to volume `N`, no matter which node it is executing on/scheduled to. The total number of EBS volumes in the Swarm should be kept below `12 * (minimum number of nodes that are expected to be present at any time)` to ensure that EC2 can properly attach EBS volumes to a node when another node fails. Use EBS volumes only for those workloads where low latency and high IOPs is absolutely necessary.

You can also create EBS backed volumes using the `docker volume create` CLI:

```
$ docker volume create \
  -d "cloudstor:aws" \
  --opt ebstype=io1 \
  --opt size=25 \
  --opt iops=1000 \
  --opt backing=relocatable \
  mylocalvol1
```

Sharing the same `relocatable` Cloudstor volume across multiple tasks of a service or across multiple independent containers is not supported when `backing=relocatable` is specified. Attempting to do so results in I/O errors.

## Use a Unique Volume per Task using EFS

If EFS is available and enabled, you can use templatized notation to create and mount a unique EFS-backed volume into each task of a service. This is useful if you already have too many EBS volumes or want to reduce the amount of time it takes to transfer volume data across availability zones.

```
$ docker service create \
  --replicas 5 \
  --name ping2 \
  --mount type=volume,volume-driver=cloudstor:aws,source={{.Service.Name}}-{{.Task.Slot}}-
vol,destination=/mydata \
  alpine ping docker.com
```

Here, each task has mounted its own volume at `/mydata/` and the files under that mountpoint are unique to that task.

When a task with only `shared` EFS volumes mounted is rescheduled on a different node, Docker interacts with the Cloudstor plugin to create and mount the volume corresponding to the task on the node where the task is rescheduled. Since data on EFS is available to all Swarm nodes and can be quickly mounted and accessed, the rescheduling process for tasks using EFS-backed volumes typically takes a few seconds, as compared to several minutes when using EBS.

It is highly recommended that you use the `.Task.Slot` template to ensure that task `N` always gets access to volume `N` no matter which node it is executing on/scheduled to.

### List or Remove Volumes Created by Cloudstor

Use `docker volume ls` on any node to enumerate all volumes created by Cloudstor across the Swarm.

Use `docker volume rm [volume name]` to remove a Cloudstor volume from any node. If you remove a volume from one node, make sure it is not being used by another active node, since those tasks/containers in another node lose access to their data.

## Networking Ecosystem

The Docker engine provides the capability to extend the networking functionality through the use of plugins. There are a number of Docker networking plugins that support third party networking devices and external management platforms, these plugins can be found in the Docker Store (https://store.docker.com/search?category=network&q=&type=plugin).

> Ensure you are familiar with the best practices for Designing Scalable, Portable Docker Container Networks (https://success.docker.com/article/networking).

## Scaling Considerations

Best practices for running Docker Enterprise at scale is available in in Running Docker Enterprise at Scale (https://success.docker.com/article/running-docker-ee-at-scale).

## Troubleshooting Docker Enterprise

The following articles describe common troubleshooting issues with Docker Enterprise:

- Swarm Troubleshooting Methodology (https://success.docker.com/article/Swarm-troubleshooting-methodology)
- Troubleshooting Container Networking (https://success.docker.com/article/troubleshooting-container-networking)
- Best Docker Support Resources for Troubleshooting (https://success.docker.com/article/best-support-resources)
- Monitor the Swarm status (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/)
- Troubleshoot UCP node states (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/troubleshoot-node-messages/)
- Troubleshoot your Swarm (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/troubleshoot-with-logs/)

- Troubleshoot Swarm configurations (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/troubleshoot-configurations/)
- Troubleshooting External Certificates for UCP/DTR (https://success.docker.com/article/troubleshooting-external-certificates-for-ucp-dtr)

# Troubleshooting Docker Enterprise Environment on AWS

In the event of issues with AWS it is recommended to either consult the AWS knowledge base (https://aws.amazon.com/premiumsupport/knowledge-center/) or work directly with AWS support.
region: regionyourEFSisin
provisioner.name: example.com/aws-efs

---

kind: Deployment
apiVersion: extensions/v1beta1
metadata:
name: efs-provisioner
spec:
replicas: 1
strategy:
type: Recreate
template:
metadata:
labels:
app: efs-provisioner
spec:
serviceAccount: efs-provisioner
containers:
- name: efs-provisioner
image: quay.io/external_storage/efs-provisioner:latest
env:
- name: FILE_SYSTEM_ID
valueFrom:
configMapKeyRef:
name: efs-provisioner
key: file.system.id
- name: AWS_REGION
valueFrom:
configMapKeyRef:
name: efs-provisioner
key: aws.region
- name: PROVISIONER_NAME
valueFrom:
configMapKeyRef:
name: efs-provisioner
key: provisioner.name
volumeMounts:
- name: pv-volume
mountPath: /persistentvolumes
volumes:
- name: pv-volume
nfs:
server: yourEFSsystemID.efs.yourEFSregion.amazonaws.com
path: /

```
Now, let's understand what this `Deployment` describes.

First, the kind of object for this Kubernetes resource is `Deployment`.  The `apiVersion` is set to
`v1` since this will be using the built-in APIs.

A `Deployment` defines a desired state, and for the efs-provisioner the desire is to have one pod, that
if it is destroyed the strategy indicates to `Recreate` it.

The `template` adds the label `efs-provisioner` to all resources, for easily selecting them in kubectl
or other resources.

The Pod defined by the `spec` will be owned by the efs-provisioner service account.  The `container`
that is created in the Pod uses the efs-provisioner image, which is currently a [Kubernetes incubator
external-storage project](https://github.com/kubernetes-incubator/external-
storage/tree/master/aws/efs).  Inside the container the EFS persistent volume `pv-volume` will be
mounted at `/persistentvolumes.`

The Pod will connect to EFS via NFS at the root path of the filesystem.

##### Kubernetes StorageClass for EFS

Below is an example of a `StorageClass` definition for provisioning Volumes in EFS.
```

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: aws-efs
provisioner: example.com/aws-efs
```

```
First, the kind of object for this Kubernetes resource is `StorageClass`.  The `apiVersion` is set to
`storage.k8s.io/v1` since this will be using the built-in APIs.  Additionally, this `StorageClass` will
be referenced by a metadata name: `aws-efs`.

The `StorageClass` will use the `example.com/aws-efs` provisioner, which will provision storages on
behalf of the cluster.


##### Kubernetes PersistentVolumeClaim for EFS

Below is an example of a `PersistentVolumeClaim` for dynamically creating volumes:
```

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
name: efs
annotations:
volume.beta.kubernetes.io/storage-class: "aws-efs"
spec:
accessModes:
- ReadWriteMany
resources:
requests:
storage: 1Mi
```

```
First, the kind of object for this Kubernetes resource is `PersistentVolumeClaim`.  The `apiVersion` is set to `v1` since this will be using the built-in APIs.  Additionally, this `PersistentVolumeClaim` will be referenced by a metadata name: `efs`.

The `spec` for the `PersistentVolumeClaim` references the `storageClassName` to  use for creating volumes.  The accessModes indicate how many nodes may read and write to the volume.  EFS supports `ReadWriteMany` access mode. The volumes created by this `PersistentVolumeClaim` can be read from and written to by many nodes simultaneously. Finally, the `PersistentVolumeClaim` will request 1-megabyte of storage from EFS.

There are many options available, and for more details on them review [the Kubernetes Documentation on Persistent Volume Claims](https://kubernetes.io/docs/concepts/storage/persistent-volumes/#persistentvolumeclaims)

##### Kubernetes Example Pod for EFS
```

kind: Pod
apiVersion: v1
metadata:
name: test-pod
spec:
containers:

- name: test-pod
  image: gcr.io/google_containers/busybox:1.24
  command:
    - "/bin/sh"
      args:
    - "-c"
    - "touch /mnt/SUCCESS && ls -la /mnt && exit 0 || exit 1"
      volumeMounts:
    - name: efs-pvc
      mountPath: "/mnt"
      restartPolicy: "Never"
      volumes:
    - name: efs-pvc
      persistentVolumeClaim:
      claimName: efs

```
## Networking Ecosystem

The Docker engine provides the capability to extend the networking functionality through the use of plugins. There are a number of Docker networking plugins that support third party networking devices and external management platforms, these plugins can be found in the [Docker Store](https://store.docker.com/search?category=network&q=&type=plugin).

> Ensure you are familiar with the best practices for [Designing Scalable, Portable Docker Container Networks](https://success.docker.com/article/networking).

## Scaling Considerations

Best practices for running Docker Enterprise at scale is available in in [Running Docker Enterprise at
```

```
Scale](https://success.docker.com/article/running-docker-ee-at-scale).

### Troubleshooting Docker Enterprise

The following articles describe common troubleshooting issues with Docker Enterprise:

- [Swarm Troubleshooting Methodology](https://success.docker.com/article/Swarm-troubleshooting-
methodology)
- [Troubleshooting Container Networking](https://success.docker.com/article/troubleshooting-container-
networking)
- [Best Docker Support Resources for Troubleshooting](https://success.docker.com/article/best-support-
resources)
- [Monitor the Swarm status](https://docs.docker.com/datacenter/ucp/3.0/guides/admin/monitor-and-
troubleshoot/)
- [Troubleshoot UCP node states](https://docs.docker.com/datacenter/ucp/3.0/guides/admin/monitor-and-
troubleshoot/troubleshoot-node-messages/)
- [Troubleshoot your Swarm](https://docs.docker.com/datacenter/ucp/3.0/guides/admin/monitor-and-
troubleshoot/troubleshoot-with-logs/)
- [Troubleshoot Swarm configurations](https://docs.docker.com/datacenter/ucp/3.0/guides/admin/monitor-
and-troubleshoot/troubleshoot-configurations/)
- [Troubleshooting External Certificates for UCP/DTR]
(https://success.docker.com/article/troubleshooting-external-certificates-for-ucp-dtr)

### Troubleshooting Docker Enterprise Environment on AWS

In the event of issues with AWS it is recommended to either consult the [AWS knowledge base]
(https://aws.amazon.com/premiumsupport/knowledge-center/) or work directly with AWS support.
ave.

![](https://docs.docker.com/datacenter/ucp/2.2/guides/images/license-ucp-2.png)

### Setup of Authentication

Docker supports the use of LDAP (Lightweight Directory Access Protocol) to provide integration with
existing authentication platforms such as Active Directory. Docker UCP integrates with LDAP directory
services, so that you can manage users and groups from your organization’s directory and it will
automatically propagate that information to UCP and DTR.

If you enable LDAP, UCP uses a remote directory server to create users automatically, and all logins
are forwarded to the directory server.

When you switch from built-in authentication to LDAP authentication, all manually created users whose
usernames don’t match any LDAP search results are still available.

You control how UCP integrates with LDAP by creating searches for users. You can specify multiple
search configurations, and you can specify multiple LDAP servers to integrate with. Searches start with
the `Base DN`, which is the distinguished name of the node in the LDAP directory tree where the search
starts looking for users.

Access LDAP settings by navigating to the **Authentication & Authorization** page in the UCP web UI.
There are two sections for controlling LDAP searches and servers.

- **LDAP user search configurations:** This is the section of the Authentication & Authorization page
where you specify search parameters, like `Base DN`, `scope`, `filter`, the `username` attribute, and
the `full name` attribute. These searches are stored in a list, and the ordering may be important,
depending on your search configuration.
- **LDAP server:** This is the section where you specify the URL of an LDAP server, TLS configuration,
```

and credentials for doing the search requests. Also, you provide a domain for all servers but the first one. The first server is considered the default domain server. Any others are associated with the domain that you specify in the page.

### Recommended APIs to Monitor

To programmatically integrate Docker UCP monitoring with existing solutions there are a number of API endpoints that can be accessed in order to determine both the state of the running services along with the Docker EE platform itself.

**Determine the state of the platform**

To get a simple overview of the platform the URI endpoint `/_ping` will return the HTTP status code of the Docker UCP components.

| HTTP Status Code | Reason |
|:---|:---|
| 200 | **Success**, manager healthy |
| 500 | **Failure**, manager unhealthy |
| Success, manager healthy | default |

**Return a list of events**

To return a list of events that have happened on the Docker EE platform the endpoint `/events` returns a parsable JSON response that can be evaluated to determine what has happened on the Docker EE platform.

**Examine a particular service**

The endpoint `/services/{id}` allows an end-user to programmatically evaluate the health and status of a particular service that is running on the Docker EE platform.

Further information about the API endpoints are available at [docs.docker.com](https://docs.docker.com/datacenter/ucp/2.2/reference/api/#/).

### Log Collection and Frequency

Part of deploying Docker EE is configuring logging. Logging provides visibility into the health and performance of both the platform and the services that run on top of it. It is recommended that you send logs to a log aggreagator that provides ease of search as well as levels of observibility that allow for quick troubleshooting of instability issues.

A reference architecture that details logging design and best practices is available in [Docker Logging Design and Best Practices](https://success.docker.com/article/docker-reference-architecture-docker-logging-design-and-best-practices).

You have the option to deploy a Splunk Enterprise stack in your Docker EE cluster to gather, store, search, analyze, and visualize the logs from Docker containers. For documentation on this solution, review the [Splunk Enterprise Solution Brief for Docker EE 17.06](https://success.docker.com/article/splunk-logging).

### Upgrading Individual Components of Docker EE

> Before upgrading, make sure you [create a backup](https://success.docker.com/article/backup-restore-best-practices). This makes it possible to recover if anything goes wrong during the upgrade.

```
Docker Enterprise Edition has the capability to "dial home" to docker.com and determine if a new
version of Docker UCP or DTR is available; this requires Internet access. In the event that an update
is available, a banner will appear in the UI alerting the user to the newly released version of a
particular component of Docker EE.

When a new version of Docker Enterprise Edition is available, an end-user has the option of upgrading
directly through the user interface by clicking the banner alert displaying the new version and then
following the prompts to update. Alternatively a user can download the offline bundle(s) and follow the
CLI steps to perform a command line update to Docker EE components. Use the documentation for the
method you prefer:

- [Perform updates in the UI](https://docs.docker.com/ee/upgrade/)
- [Perform updates from the command line]
(https://docs.docker.com/datacenter/ucp/2.2/reference/cli/upgrade/)

> **Note:** Before starting any upgrade tasks, it is recommended that you study the upgrade matrix to
ensure that the correct upgrade path is followed, the matrix is available on [success.docker.com]
(https://success.docker.com/article/compatibility-matrix).

For more details on upgrading individual components of Docker EE, review the documentation at
<https://docs.docker.com/enterprise/upgrade/>.

### Monitoring Docker EE with Prometheus

Currently only the Docker engines can be monitored with Prometheus and not the management platform as a
whole. To monitor the Docker EE engines, modify their configurations to expose the metrics. The
instructions for enabling Prometheus metrics are available in [Grafana/Prometheus Monitoring Solution
Brief for Docker EE 17.06](https://success.docker.com/article/grafana-prometheus-monitoring/).

## Persistent Storage

The majority of applications have some component that requires data to persist for a myriad of reasons
such as application resiliency in the event of application failure to the size of data to lookup.

### Locally Presented Storage

By default Docker Certified Infrastructures provision the following partitions as separate EBS volumes.
This can vary depending on instances used.

| Mount path        | Size   | Purpose            |
|:------------------|:-------|:-------------------|
| `/`               | 8 GB   | Root partition     |
| `/var/lib/docker` | 100 GB | Docker data volume |

### Storage Ecosystem

Docker EE storage capabilities can be extended through the use of plugins. There are a number of Docker
storage plugins that support third party storage devices and external management platforms.

For example, when provisioning Docker EE using the supplied tooling from Docker, Cloudstor is
configured as part of the installation. Cloudstor is a modern volume plugin built by Docker. Docker
swarm mode tasks and regular Docker containers can use a volume created with Cloudstor to mount a
persistent data volume. In the Docker Certified Infrastructure for AWS, Cloudstor has two backing
options:

#### What is Cloudstor?
```

```
Cloudstor is a modern volume plugin built by Docker. It comes pre-installed and
pre-configured in Docker swarms deployed through the Docker Certified Infrastructure for AWS. Docker
swarm
mode tasks and regular Docker containers can use a volume created with
Cloudstor to mount a persistent data volume. In the Docker Certified Infrastructure for AWS, Cloudstor
has
two `backing` options:

- `relocatable` data volumes are backed by EBS.
- `shared` data volumes are backed by EFS.

When you use the Docker CLI to create a swarm service along with the persistent
volumes used by the service tasks, you can create three different types of
persistent volumes:

- Unique `relocatable` Cloudstor volumes mounted by each task in a swarm service.
- Global `shared` Cloudstor volumes mounted by all tasks in a swarm service.
- Unique `shared` Cloudstor volumes mounted by each task in a swarm service.

 Examples of each type of volume are described in the following sections.

#### Relocatable Cloudstor Volumes

 Workloads running in a Docker service that require access to low latency/high
 IOPs persistent storage, such as a database engine, can use a `relocatable`
 Cloudstor volume backed by EBS. When you create the volume, you can specify the
 type of EBS volume appropriate for the workload (such as `gp2`, `io1`, `st1`,
 `sc1`). Each `relocatable` Cloudstor volume is backed by a single EBS volume.

 If a swarm task using a `relocatable` Cloudstor volume gets rescheduled to
 another node within the same availability zone as the original node where the
 task was running, Cloudstor detaches the backing EBS volume from the original
 node and attaches it to the new target node automatically.

 If the swarm task gets rescheduled to a node in a different availability zone,
 Cloudstor transfers the contents of the backing EBS volume to the destination
 availability zone using a snapshot and cleans up the EBS volume in the
 original availability zone. To minimize the time necessary to create the
 snapshot to transfer data across availability zones, Cloudstor periodically
 takes snapshots of EBS volumes to ensure there is never a large number of
 writes that need to be transferred as part of the final snapshot when
 transferring the EBS volume across availability zones.

 Typically the snapshot-based transfer process across availability zones takes
 between 2 and 5 minutes unless the workload is write-heavy. For extremely
 write-heavy workloads generating several GBs of fresh/new data every few
 minutes, the transfer may take longer than 5 minutes. The time required to
 snapshot and transfer increases sharply beyond 10 minutes if more than 20 GB of
 writes have been generated since the last snapshot interval. A swarm task is
 not started until the volume it mounts becomes available.

 Sharing/mounting the same Cloudstor volume backed by EBS among multiple tasks
 is not a supported scenario and leads to data loss. If you need a Cloudstor
 volume to share data between tasks, choose the appropriate EFS backed `shared`
 volume option. Using a `relocatable` Cloudstor volume backed by EBS is
 supported on all AWS regions that support EBS. The default `backing` option is
 `relocatable` if EFS support is not selected during setup/installation or if
```

```
 EFS is not supported in a region.

#### Shared Cloudstor Volumes

When multiple swarm service tasks need to share data in a persistent storage
volume, you can use a `shared` Cloudstor volume backed by EFS. Such a volume and
its contents can be mounted by multiple swarm service tasks without the risk of
data loss, since EFS makes the data available to all swarm nodes over NFS.

When swarm tasks using a `shared` Cloudstor volume get rescheduled from one node
to another within the same or across different availability zones, the
persistent data backed by EFS volumes is always available. `shared` Cloudstor
volumes only work in those AWS regions where EFS is supported. If EFS
Support is selected during setup/installation, the default "backing" option for
Cloudstor volumes is set to `shared` so that EFS is used by default.

`shared` Cloudstor volumes backed by EFS (or even EFS MaxIO) may not be ideal
for workloads that require very low latency and high IOPSs. For performance
details of EFS backed `shared` Cloudstor volumes, see [the AWS performance
guidelines](http://docs.aws.amazon.com/efs/latest/ug/performance.html).

#### Use Cloudstor

After initializing or joining a swarm on the Docker Certified Infrastructure for AWS, connect to any
swarm
manager using SSH. Verify that the CloudStor plugin is already installed and
configured for the stack or resource group:

```bash
$ docker plugin ls

ID                  NAME                DESCRIPTION                         ENABLED
f416c95c0dcc        cloudstor:aws       cloud storage plugin for Docker     true
```
```

The following examples show how to create swarm services that require data persistence using the `--mount` flag and specifying Cloudstor as the volume driver.

## Share the Same Volume among Tasks using EFS

In those regions where EFS is supported and EFS support is enabled during deployment of the Cloud Formation template, you can use `shared` Cloudstor volumes to share access to persistent data across all tasks in a swarm service running in multiple nodes, as in the following example:

```
$ docker service create \
  --replicas 5 \
  --name ping1 \
  --mount type=volume,volume-driver=cloudstor:aws,source=sharedvol1,destination=/shareddata \
  alpine ping docker.com
```

All replicas/tasks of the service `ping1` share the same persistent volume `sharedvol1` mounted at `/shareddata` path within the container. Docker takes care of interacting with the Cloudstor plugin to ensure that EFS is mounted on all nodes in the swarm where service tasks are scheduled. Your application needs to be designed to ensure that tasks do not write concurrently on the same file at the same time, to protect against data corruption.

You can verify that the same volume is shared among all the tasks by logging into one of the task containers, writing a file under `/shareddata/`, and logging into another task container to verify that the file is available there as well.

The only option available for EFS is `perfmode`. You can set `perfmode` to `maxio` for high IO throughput:

```
$ docker service create \
  --replicas 5 \
  --name ping3 \
  --mount type=volume,volume-driver=cloudstor:aws,source={{.Service.Name}}-{{.Task.Slot}}-
vol5,destination=/mydata,volume-opt=perfmode=maxio \
  alpine ping docker.com
```

You can also create `shared` Cloudstor volumes using the `docker volume create` CLI:

```
$ docker volume create -d "cloudstor:aws" --opt backing=shared mysharedvol1
```

## Use a Unique Volume per Task using EBS

If EBS is available and enabled, you can use a templatized notation with the `docker service create` CLI to create and mount a unique `relocatable` Cloudstor volume backed by a specified type of EBS for each task in a swarm service. New EBS volumes typically take a few minutes to be created. Besides `backing=relocatable`, the following volume options are available:

| Option | Description |
|---|---|
| `size` | Required parameter that indicates the size of the EBS volumes to create in GB. |
| `ebstype` | Optional parameter that indicates the type of the EBS volumes to create (`gp2`, `io1`, `st1`, `sc1`}. The default `ebstype` is Standard/Magnetic. For further details about EBS volume types, see the EBS volume type documentation (http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/EBSVolumeTypes.html). |
| `iops` | Required if `ebstype` specified is `io1`, which enables provisioned IOPs. Needs to be in the appropriate range as required by EBS. |

Example usage:

```
$ docker service create \
  --replicas 5 \
  --name ping3 \
  --mount type=volume,volume-driver=cloudstor:aws,source={{.Service.Name}}-{{.Task.Slot}}-
vol,destination=/mydata,volume-opt=backing=relocatable,volume-opt=size=25,volume-opt=ebstype=gp2 \
  alpine ping docker.com
```

The above example creates and mounts a distinct Cloudstor volume backed by 25 GB EBS volumes of type `gp2` for each task of the `ping3` service. Each task mounts its own volume at `/mydata/` and all files under that mountpoint are unique to the task mounting the volume.

It is highly recommended that you use the `.Task.Slot` template to ensure that task `N` always gets access to volume `N`, no matter which node it is executing on/scheduled to. The total number of EBS volumes in the swarm should be kept below `12 * (minimum number of nodes that are expected to be present at any time)` to ensure that EC2 can properly attach EBS volumes to a node when another node fails. Use EBS volumes only for those workloads where low latency and high IOPs is absolutely necessary.

You can also create EBS backed volumes using the `docker volume create` CLI:

```
$ docker volume create \
  -d "cloudstor:aws" \
  --opt ebstype=io1 \
  --opt size=25 \
  --opt iops=1000 \
  --opt backing=relocatable \
  mylocalvol1
```

Sharing the same `relocatable` Cloudstor volume across multiple tasks of a service or across multiple independent containers is not supported when `backing=relocatable` is specified. Attempting to do so results in IO errors.

## Use a Unique Volume per Task using EFS

If EFS is available and enabled, you can use templatized notation to create and mount a unique EFS-backed volume into each task of a service. This is useful if you already have too many EBS volumes or want to reduce the amount of time it takes to transfer volume data across availability zones.

```
$ docker service create \
  --replicas 5 \
  --name ping2 \
  --mount type=volume,volume-driver=cloudstor:aws,source={{.Service.Name}}-{{.Task.Slot}}-
vol,destination=/mydata \
  alpine ping docker.com
```

Here, each task has mounted its own volume at `/mydata/` and the files under that mountpoint are unique to that task.

When a task with only `shared` EFS volumes mounted is rescheduled on a different node, Docker interacts with the Cloudstor plugin to create and mount the volume corresponding to the task on the node where the task is rescheduled. Since data on EFS is available to all swarm nodes and can be quickly mounted and accessed, the rescheduling process for tasks using EFS-backed volumes typically takes a few seconds, as compared to several minutes when using EBS.

It is highly recommended that you use the `.Task.Slot` template to ensure that task `N` always gets access to volume `N` no matter which node it is executing on/scheduled to.

## List or Remove Volumes Created by Cloudstor

You can use `docker volume ls` on any node to enumerate all volumes created by Cloudstor across the swarm.

You can use `docker volume rm [volume name]` to remove a Cloudstor volume from any node. If you remove a volume from one node, make sure it is not being used by another active node, since those tasks/containers in another node lose access to their data.

# Container Networking

All virtual machines are connected through the use of a Virtual Network.

## Batteries Included, Networking Options

Docker provides an out-of-the box experience — it comes with everything needed to provide the best set of networking features for most workloads. The majority of features should just work with no configuration changes, however there are some networking features that require a security configuration change in order to utilize.

**Bridge**

The bridge driver creates a private network internal to the host so containers on this network can communicate. External access is granted by exposing ports to containers. Docker secures the network by managing rules that block connectivity between different Docker networks.

Behind the scenes, the Docker Engine creates the necessary Linux bridges, internal interfaces, iptables rules, and host routes to make this connectivity possible. In the example highlighted below, a Docker bridge network is created and two containers are attached to it. With no extra configuration the Docker Engine does the necessary wiring, provides service discovery for the containers, and configures security rules to prevent communication to other networks. A built-in IPAM driver provides the container interfaces with private IP addresses from the subnet of the bridge network.

**Overlay**

The built-in Docker overlay network driver radically simplifies many of the complexities in multi-host networking. It is a swarm scope driver, which means that it operates across an entire Swarm or UCP cluster rather than individual hosts. With the overlay driver, multi-host networks are first-class citizens inside Docker without external provisioning or components. IPAM, service discovery, multi-host connectivity, encryption, and load balancing are built right in. For control, the overlay driver uses the encrypted Swarm control plane to manage large scale clusters at low convergence times.

The overlay driver utilizes an industry-standard VXLAN data plane that decouples the container network from the underlying physical network (the underlay). This has the advantage of providing maximum portability across various cloud and on-premises networks. Network policy, visibility, and security is controlled centrally through the Docker Universal Control Plane (UCP).

**MACVLAN**

The macvlan driver is the newest built-in network driver and offers several unique characteristics. It’s a very lightweight driver, because rather than using any Linux bridging or port mapping, it connects container interfaces directly to host interfaces. Containers are addressed with routable IP addresses that are on the subnet of the external network.

As a result of routable IP addresses, containers communicate directly with resources that exist outside a Swarm cluster without the use of NAT and port mapping. This can aid in network visibility and troubleshooting. Additionally, the direct traffic path between containers and the host interface helps reduce latency. macvlan is a local scope network driver which is configured per-host. As a result, there are stricter dependencies between MACVLAN and external networks, which is both a constraint and an advantage that is different from overlay or bridge.

The macvlan driver uses the concept of a parent interface. This interface can be a host interface such as eth0, a sub-interface, or even a bonded host adaptor which bundles Ethernet interfaces into a single logical interface. A gateway address from the external network is required during MACVLAN network configuration, as a MACVLAN network is a L2 segment from the container to the network gateway. Like all Docker networks, MACVLAN networks are segmented from each other – providing access within a network, but not between networks.

## Networking Ecosystem

The Docker engine provides the capability to extend the networking functionality through the use of plugins. There are a number of Docker networking plugins that support third party networking devices and external management platforms, these plugins can be found in the Docker Store (https://store.docker.com/search?category=network&q=&type=plugin).

# Scaling Considerations

The Docker certified infrastructure templates provide a simple and stable method for scaling your Docker Enterprise Edition platform. In order to grow and extend your existing Docker EE platform quite simply do the following:

1. Modify the `tf.vars` and increase the amount of workers required (either windows or linux workers).
2. Use terraform with a `terraform apply` to provision the new worker virtual machines and update the inventory file.
3. Ansible will can now use the updated inventory to provide the final configuration work to scale the Docker EE platform with the `ansible-playbook --private-key=<private key> -i inventory install.yml` command.

Best practices for running Docker EE at scale is available in in Running Docker Enterprise Edition at Scale (https://success.docker.com/article/running-docker-ee-at-scale).

# Troubleshooting

This section of the reference architecture contains a number of sections that can help when there are issues either during or after the deployment of Docker Enterprise Edition.

## Rolling Back an Installation

To uninstall Docker EE but keep the infrastructure in place, run the following:

```
ansible-playbook uninstall.yml
```

To completely destroy the AWS infrastructure, run:

```
terraform destroy
```

This will delete all infrastructure associated with your Docker EE deployment, including the **VPC** it is deployed in and all resources inside it.

Before running these command, ensure you have backed up any docker images you need to a Docker Trusted Registry outside the cluster. In addition, you may want to export an AWS CloudFormation template (https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-using-cloudformer.html) to document and recreate AWS resources, particularly if you modified the default deployment in any way and may wish to do so again at a later point.

> To avoid accidental destruction of your cluster, you may wish to prevent Stack Updates (https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/protect-stack-resources.html) using the updated stack policies.

## Troubleshooting the Docker EE Installation Templates

When using terraform and it encounters an error, it will **pause** and exit at the failed task. If that happens, fix the error and re-run `terraform apply`. If the error is fixed, Terraform will continue on to the next task. Terraform errors are usually fairly descriptive as to why it failed.

## Troubleshooting Docker EE

The following articles describe common troubleshooting issues with Docker EE:

- Swarm Troubleshooting Methodology (https://success.docker.com/article/swarm-troubleshooting-methodology)
- Troubleshooting Container Networking (https://success.docker.com/article/troubleshooting-container-networking)
- Troubleshooting a UCP 2.2.x Cluster (https://success.docker.com/article/troubleshooting-a-ucp-22x-cluster)
- Troubleshooting a DTR 2.3.x Cluster (https://success.docker.com/article/troubleshooting-a-dtr-23x-cluster)
- Best Docker Support Resources for Troubleshooting (https://success.docker.com/article/best-support-resources)
- Monitor the swarm status (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/)
- Troubleshoot UCP node states (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/troubleshoot-node-messages/)
- Troubleshoot your swarm (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/troubleshoot-with-logs/)
- Troubleshoot swarm configurations (https://docs.docker.com/datacenter/ucp/2.2/guides/admin/monitor-and-troubleshoot/troubleshoot-configurations/)
- Troubleshooting External Certificates for UCP/DTR (https://success.docker.com/article/troubleshooting-external-certificates-for-ucp-dtr)

## Troubleshooting Docker EE Environment on AWS

In the event of issues with AWS it is recommended to either consult the AWS knowledge base (https://aws.amazon.com/premiumsupport/knowledge-center/) or work directly with AWS support.

# Docker Solution Briefs

Docker Solution Briefs are short articles that aim to provide basic installation and validation details for specific 3rd-party ecosystem solutions with the Docker Enterprise Edition (EE) platform. The information on each of the following solutions is provided by Docker as a known, working configuration at the time of publishing. Docker does not support these products. Please contact the specific vendor using their approved support methods if you have any questions with the particular solution.

- Amazon CloudWatch Logs Solution Brief for Docker EE 17.06 (https://success.docker.com/article/aws-cloudwatch-logging)

- Amazon CloudWatch Solution Brief for Docker EE 17.06 (https://success.docker.com/article/aws-cloudwatch-metrics)
- Splunk Enterprise Solution Brief for Docker EE 17.06 (https://success.docker.com/article/splunk-logging/)
- Grafana/Prometheus Monitoring Solution Brief for Docker EE 17.06 (https://success.docker.com/article/grafana-prometheus-monitoring/)
- NGINX Solution Brief on Docker EE 17.06 (https://success.docker.com/article/nginx-load-balancer/)
- IBM Security Access Manager v9.0.4 Solution Brief for Docker EE 17.06 (https://success.docker.com/article/ibm-isam-security/)
- Logging with Elasticsearch, Logstash, and Kibana Solution Brief for Docker EE 17.06 (https://success.docker.com/article/elasticsearch-logstash-kibana-logging/)

# Appendix A. Manual Installation of Docker EE

This appendix contains generic instructions for installing Docker EE on an AWS infrastructure.

## Prerequisites

- Permissions to configure DNS, setup firewall rules, and sign-up for Docker EE.
- Existing infrastructure stack with 10 VMs available. All VMs must be able to reach each other over the network, but regular Docker clients will only need access to `ucp.example.com` and `dtr.example.com`. Each of the UCP and DTR managers should reside in their own Availability zone.
  - 3 managers (manager-01, manager-02, manager-03) with 1GB memory or more
  - 3 workers (worker-01, worker-02, worker-03)
  - 3 DTR nodes (dtr-01, dtr-02, dtr-03)
  - 2 ELBs
  - 1 S3 Bucket
- Domain name and TLS certificates for `ucp` and `dtr`.
- Shell access to each VM (e.g. `ssh` or console access)
- (optional) Client with Docker installed (for testing the registry)
- (optional) Set up the following IAM policy on hosts (https://docs.docker.com/registry/storage-drivers/s3/#s3-permission-scopes)

## Recommended: Prepare IPs, Domains, and Certificates

To avoid certificate warnings when connecting to UCP for the first time, an IP address, domain, and SSL certificate should be obtained.

For testing, a self-signed certificate and a temporary domain can be used.

In your DNS system, you'll need at least a host for UCP and DTR. These will typically point to `dtr.example.com` and the UCP load balancer `ucp.example.com`.

```
DOMAIN_NAME=<domain for this Docker EE install>
UCP_HOSTNAME = ucp.${DOMAIN_NAME}
DTR_HOSTNAME = dtr.${DOMAIN_NAME}
```

For the `UCP_HOSTNAME` and the `DTR_HOSTNAME`, you need to create a SSL certificate. You can use your normal process for issuing X.509 server certificates.

# Get Docker EE

Go to https://store.docker.com/my-content (https://store.docker.com/my-content) and download the license key file.
Also note the `DOCKER_EE_URL` (`https://storebits.docker.com/ee/...`).

# Installation

The following steps guide you through setting up Docker EE.

Be sure you have the following variables defined

- `DOCKER_EE_URL`
- `UCP_HOSTNAME`
- `DTR_HOSTNAME`

Docker EE installation is made easy through package managers. The next two sections describe how to setup Docker EE on Ubuntu or Red Hat Enterprise Linux.

This example shows Docker EE on each of the 8 nodes. 3 are managers, 3 are workers, 1 runs DTR, and 1 is the UCP load balancer. All of them run Docker EE.

## Install Docker EE on Ubuntu 16.04 / 17.06

```
curl -fsSL ${DOCKER_EE_URL}/ubuntu/gpg | sudo apt-key add -
sudo add-apt-repository "deb [arch=amd64] ${DOCKER_EE_URL}/ubuntu
$(https://success.docker.com/api/asset/.%2Fpublish%2Fcertified-infrastructures-aws%2Flsb_release -cs)
stable-17.06"
sudo apt-get update
sudo apt-get install docker-ee -y
```

## Install Docker EE on RHEL 7

```
sudo mkdir /etc/docker
sudo cp /tmp/daemon.json /etc/docker/daemon.json
sudo -E sh -c 'echo \"${DOCKER_EE_URL}\" > /etc/yum/vars/dockerurl'
sudo sh -c 'echo 7 > /etc/yum/vars/dockerosversion'
sudo yum install -y yum-utils device-mapper-persistent-data lvm2
sudo yum-config-manager --enable rhel-7-server-extras-rpms
sudo -E yum-config-manager --add-repo \"${DOCKER_EE_URL}/rhel/docker-ee.repo\"
sudo yum -y install docker-ee
sudo systemctl enable docker
```

## Create UCP Manager Nodes

From the set of 7 nodes created, use three of them to serve as managers. On the first manager run the following command:

```
docker container run --rm -it --name ucp \
    -v /var/run/docker.sock:/var/run/docker.sock \
    docker/ucp:3.0.0 install \
    --san ${UCP_HOSTNAME} \
    --host-address <node-ip-address> \
    --interactive
```

The installer will prompt you for a username and password for the administrator. The instructions in this runbook assumes that the chosen username is `admin`, but this is not required.

Once UCP is installed on the first manager, use a web browser to connect to the IP address of `manager-01` using HTTPS (e.g. `https://manager-01`). Then log into the UCP dashboard with the admin user.

A new TLS certificate can be uploaded in **Admin settings** -> **Certificates**.

To add two more managers:

1. Go to the **Nodes** section.
2. Click on **Add Node** in the UI.
3. Select **Manager** to get the join command.
4. Run the command on each manager node.

The UCP dashboard should now show **3 Manager Nodes**.

## Create Worker Nodes

From the set of 7 nodes created, use three of them to serve as workers.

1. Go to `https://manager-01` (replace `manager-01` with IP or domain name of the first manager).
2. Go to the **Nodes** section.
3. Click on **Add Node** in the UI.
4. Select **Worker** to get the join command.
5. Run the command on each worker node.

The UCP dashboard should now show **3 Worker Nodes**.

## Setup Load Balancing

`${UCP_HOSTNAME}` should point to a load balancer that balances traffic to `manager-01`, `manager-02` and `manager-03`. Set up an ELB that points to each of the UCP managers.

`${DTR_HOSTNAME}` should point to a load balancer that balances traffic to `dtr-01`, `dtr-02` and `dtr-03`. Set up an ELB that points to each of the DTR managers.

## Create DTR Node

Create at least one DTR node. The node used as the DTR node has to first be added to the swarm as a worker or manager (see instructions above).

After the node has been added to the swarm, run the following command on `dtr-01`:

```
docker container run -it --rm \
  docker/dtr:2.5.0 install \
  --ucp-node dtr-01 \
  --ucp-url=https://${UCP_HOSTNAME} \
  --ucp-insecure-tls \
    --dtr-external-url ${DTR_HOSTNAME}
```

Enter the UCP username and password when prompted.

If you get `failed to choose ucp node`, just try again.
If it still doesn't work, add `--debug` to the command to find out why.

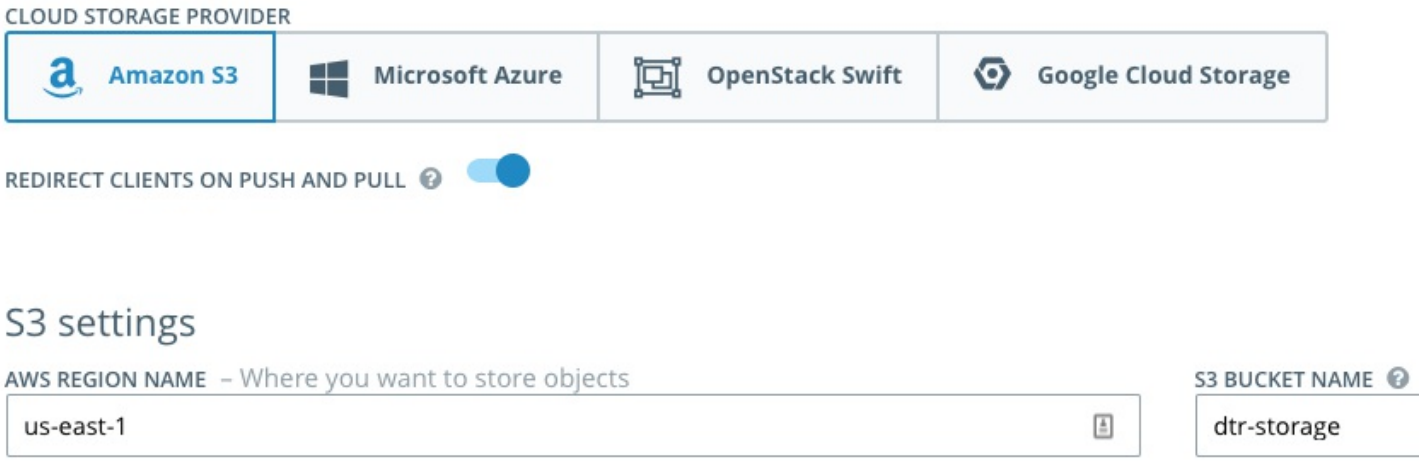
You should now be able to access DTR at `https://${DTR_HOSTNAME}`.
The username and password are the same as for UCP.

Go to `Settings` and set the `Load Balancer/Public Address` to the DNS name (e.g. `DTR_HOSTNAME`).

Log out and log back in, using the `Use Single Sign-On` button.
After confirming that this works, go back to the Settings back and turn on `Automatically redirect users to ucp for login`.

A TLS certificate can be uploaded in `Settings`, `Domain & Proxies`.

Configure DTR to to use the S3 bucket.




## Join DTR Replicas

On the `dtr-01` node, run `docker ps`. Take note of the replica ID that is listed as a string at the end of the DTR containers.

It will look something similar to this:

```
dtr-notary-server-2e337c65a09a
dtr-api-2e337c65a09a
dtr-registry-2e337c65a09a
```

After taking note of the `dtr-01` *existing* replica ID, run the following command on any node in the cluster (it will deploy via Swarm). The existing replica ID from `dtr-01` will be used to run the Docker commands through.

> **Note:** `ucp-node` is the node to deploy the DTR replica to.

```
docker run -it --rm docker/dtr:2.5.0 \
    join \
    --existing-replica-id <existing_replica_id> \
    --ucp-url https://${UCP_HOSTNAME} \
    --ucp-node <node to deploy replica to>
    --ucp-username <admin-username> \
    --ucp-password <admin-password> \
    --ucp-insecure-tls
```

Run one more time with different `ucp-node` to get up to three DTR replicas.

## Test the Registry

> **Note:** This step requires that DTR has been configured to use a signed certificate. For testing with self-signed certificates or insecure connections, see https://docs.docker.com/registry/insecure/ (https://docs.docker.com/registry/insecure/).

Use the DTR web interface to create a new private repository called `admin/test`. If you used a different username than `admin` when you installed DTR/UCP, replace `admin` with the correct name or manually create an `admin` organisation in the web UI first.

Then try pushing an image to it:

```
docker login ${DTR-HOSTNAME}
docker pull busybox
docker tag busybox ${DTR-HOSTNAME}/admin/test
docker push ${DTR-HOSTNAME}/admin/test
```

The image should now appear in the web UI.
You can also go to the **DTR Settings** and turn on image scanning to scan it.

## Download Client Bundle

> **WARNING**: The hostname in the UCP bundle comes from the `Host` field sent by the browser. Therefore, you *must* access the site via the correct DNS name when downloading the bundle. Otherwise, it will contain an IP address and the certificate will be rejected.

Go to `My profile` in the UCP web interface and create a new client bundle.
Check that you can use this bundle to run a container. e.g.

```
mkdir bundle
cd bundle/
unzip ~/Downloads/ucp-bundle-admin.zip
source env.sh
docker run --rm hello-world
Hello from Docker!
```

## Firewall

Depending on the environment is recommended to ensure the firewall enables the minimal set of ports required. Docker clients will need access to the external interface of `ucp` and `dtr` so they can be reached via `${UCP_HOSTNAME}` and `${DTR_HOSTNAME}`. Ports `443` and `6443` will need to be passed through the firewall for UCP. DTR just needs port `443` passed through.

# Appendix B. Quickstart Installation of Docker EE on AWS

This appendix contains an ordered list for installing Docker EE on AWS. All of the previous considerations should be regarded when running through these steps.

1. Install ansible (http://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html)
2. Install terraform (https://www.terraform.io/intro/getting-started/install.html)
3. Install the AWS CLI (https://aws.amazon.com/cli/)
4. Run `aws configure` and enter in your existing AWS IAM credentials
5. Download and extract the `aws-vx.x.x.tar.gz` file that contains the Terraform and Ansible files needed for installation
6. Change into the extracted directory
7. Run `make stacks`
8. Execute `cd stack/aws`
9. Run `terraform init`
10. Create a file named `terrform.tfvars` in the root of the terraform directory. See this section (https://success.docker.com/api/asset/.%2Fpublish%2Fcertified-infrastructures-aws%2F#terraform-usage) and adjust appropriately.
11. Edit `group_vars/all` using this example (https://success.docker.com/api/asset/.%2Fpublish%2Fcertified-infrastructures-aws%2F#example-ansible-config) file and adjust appropriately
12. Run `terraform plan` and verify the changes terraform is about to perform
13. Run `terraform apply`
14. Update DNS entries to use easy to use CNAMEs instead of the ELB A Records (more details in the *Post Installation Steps* section)
15. Install with `ansible-playbook install.yml`
16. Access UCP using the CNAME record from previous step via https
17. Upload UCP license key (if not using ansible for that)
18. Upload UCP TLS certificates (if not using ansible for that)
19. Access DTR using the CNAME record from previous step via https

# Appendix C. Docker EE Overview

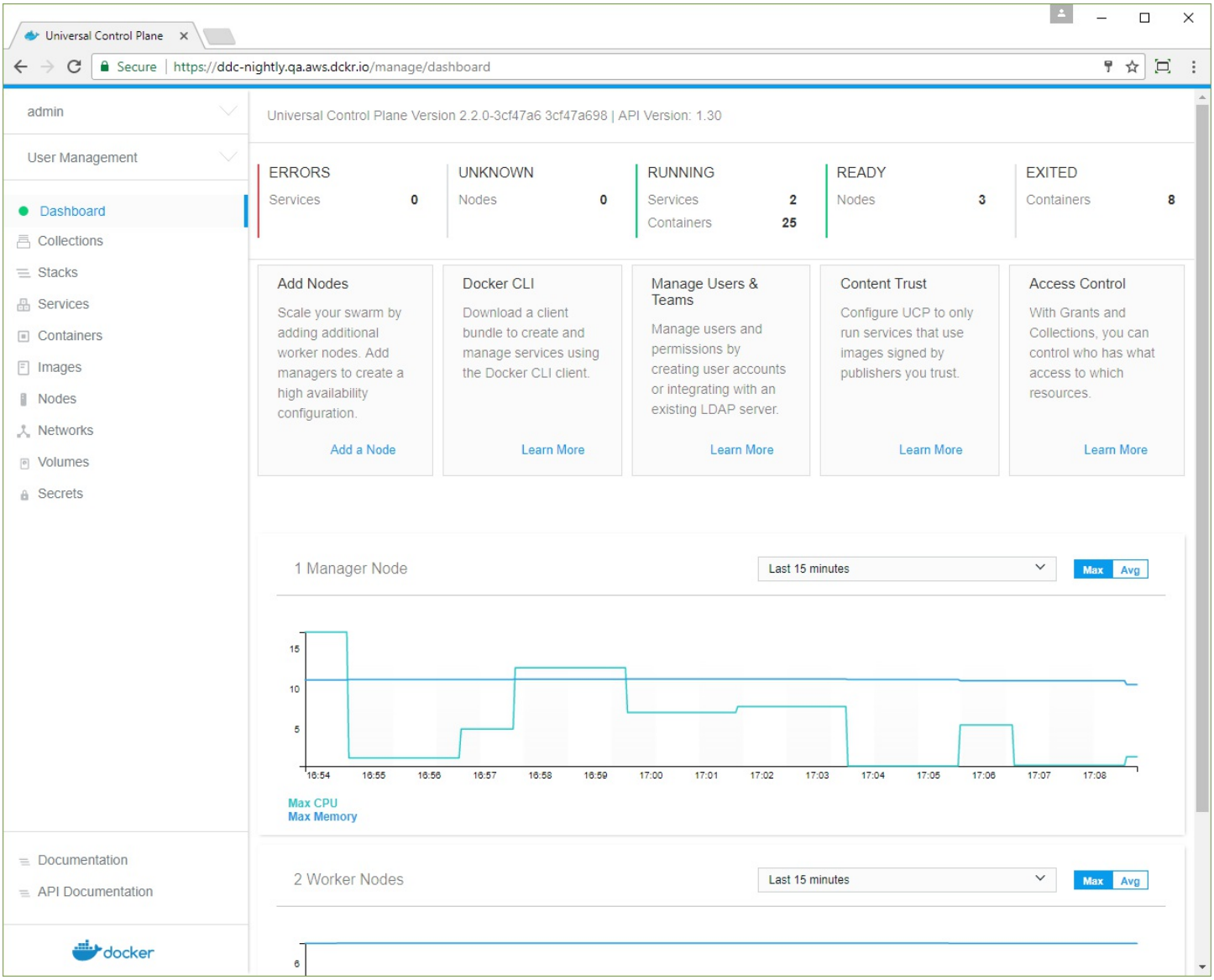
The following appendix describes the three main components of Docker EE – Universal Control Plane, Docker Trusted Registry, and Docker engine.

Refer to Docker Reference Architecture: Docker EE Best Practices and Design Considerations (https://success.docker.com/article/docker-ee-best-practices) for detailed information.

## Universal Control Plane

There are three ways to interact with UCP: the **web UI**, the **API**, or the **CLI**.

You can use the UCP web UI to manage your swarm, grant and revoke user permissions, deploy, configure, manage, and monitor your applications.




Docker UCP secures your swarm by using role-based access control. From the browser, administrators can:

- Manage swarm configurations
- Manage the permissions of users, teams, and organizations

- See all images, networks, volumes, and containers
- Grant permissions to users for scheduling tasks on specific nodes (with the Docker EE Advanced license)

UCP also exposes the standard Docker API, so you can continue using existing tools like the Docker CLI client. Since UCP secures your cluster with role-based access control, you need to configure your Docker CLI client and other client tools to authenticate your requests using client certificates that you can download from your UCP profile page.

Docker UCP secures your swarm by using role-based access control, so that only authorized users can perform changes to the cluster.

For this reason, when running Docker commands on a UCP node, you need to authenticate your request with client certificates. When trying to run Docker commands without a valid certificate, you get an authentication error:

```
$ docker ps

x509: certificate signed by unknown authority
```

There are two different types of client certificates:

- Admin user certificate bundles: allow users to run Docker commands on the Docker Engine of any node
- User certificate bundles: only allow users to run Docker commands through a UCP manager node

To download a client certificate bundle, log into the UCP web UI and navigate to your **My Profile** page. In the left pane, click **Client Bundles**, and click **New Client Bundle** to download the certificate bundle. Further information is available in the Docker docs (https://docs.docker.com/v17.09/datacenter/ucp/2.2/guides/user/access-ucp/cli-based-access/).

There is also the option of interacting with the Universal Control plane through its API, allowing programmatic access to swarm resources that are managed by UCP. The API is secured with role-based access control so that only authorized users can make changes and deploy applications to your Docker swarm.

The UCP API is accessible in the same IP addresses and domain names that you use to access the web UI. It's the same API that the UCP web UI uses, so everything you can do on the UCP web UI from your browser, you can also do programmatically.

The full API documentation is available on docs.docker.com (https://docs.docker.com/datacenter/ucp/2.2/reference/api/).

# Docker Trusted Registry

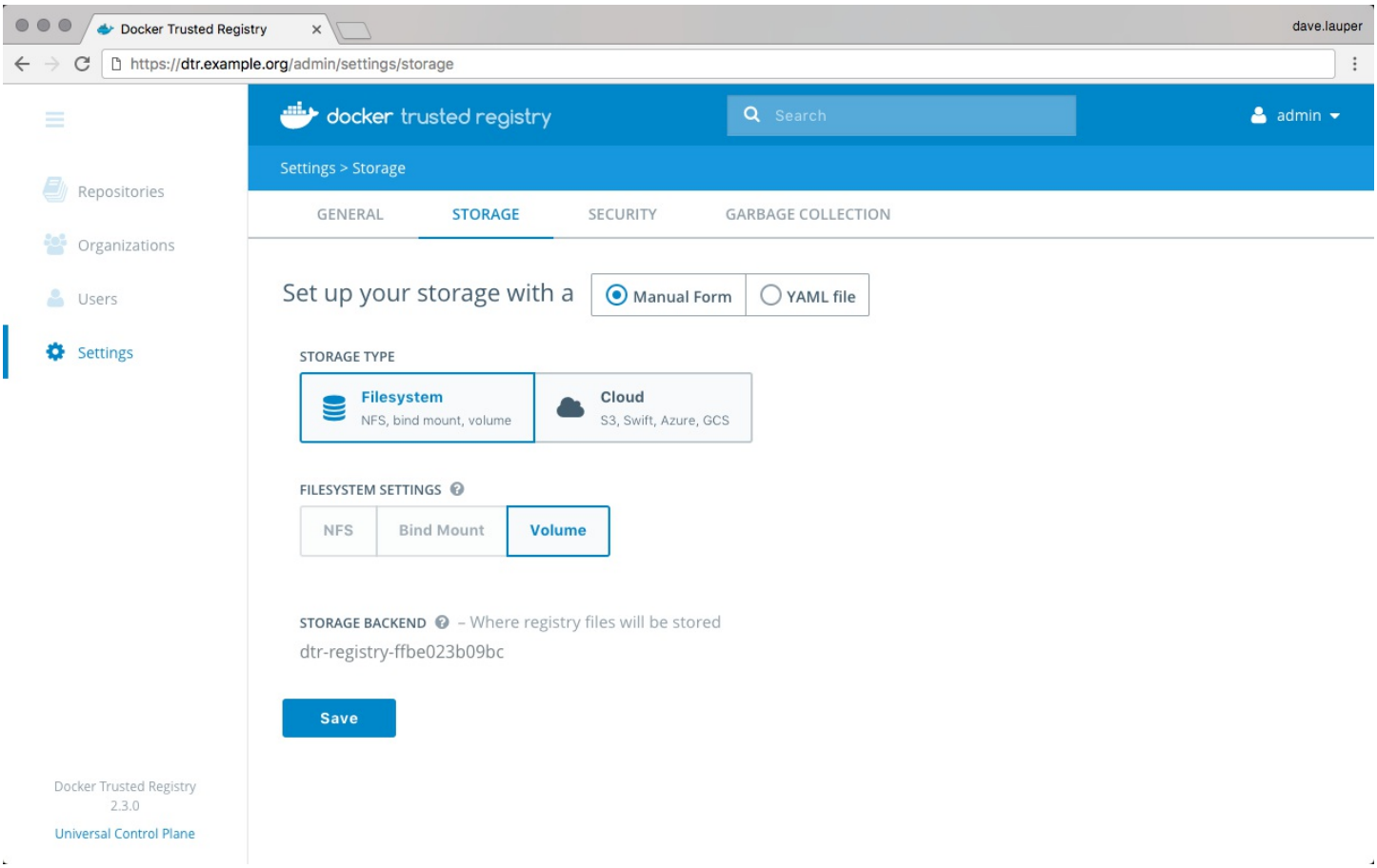
Docker Trusted Registry (DTR) is the enterprise-grade image storage solution from Docker providing secure storage and management of the Docker images you use in your applications.

There are two ways to interact with DTR: the **web UI** or the **CLI**.

You can use DTR as part of your continuous integration, and continuous delivery processes to build, ship, and run your applications.

DTR has a web based user interface that allows authorized users in your organization to browse docker images. It provides information about who pushed what image at what time. It even allows you to see what dockerfile lines were used to produce the image and, if security scanning is enabled, to see a list of all of the software installed in your images.

DTR is **highly available** through the use of multiple replicas of all containers and metadata such that if a machine fails, DTR continues to operate and can be repaired.




DTR uses the same authentication mechanism as Docker Universal Control Plane. Users can be managed manually or synched from LDAP or Active Directory. DTR uses Role Based Access Control (RBAC) to allow you to implement fine-grained access control policies for who has access to your Docker images.

DTR also has a built in security scanner that can be used to discover what versions of software are used in your images. It scans each layer and aggregates the results to give you a complete picture of what you are shipping as a part of your stack. Most importantly, it co-relates this information with a vulnerability database that is kept up to date through periodic updates. This gives you unprecedented insight into your exposure to known security threats.

Docker DTR should be configured to be the default registry for all Docker engines so when an engine needs to pull an image it will automatically pull from the trusted registry. Details on how to configure the Docker engine to communicate
correctly with Docker Trusted Registry can be found in the Docker docs (https://docs.docker.com/datacenter/dtr/2.4/guides/user/access-dtr/).

# Docker Engine

The Docker Enterprise Edition engine is deployed on all hosts that are part of the Docker EE platform. The deployment of applications can be done through the Universal Control Plane or through the CLI of the Docker hosts that are part of the management plane. The Client Bundle is required to authenticate with the Docker EE platform from the CLI. Details around using the client bundle can be found in the Universal Control Plane section.