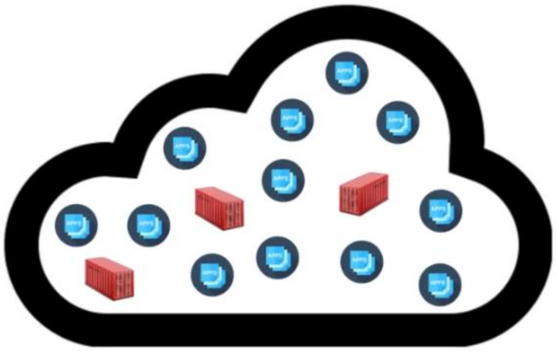
Introduction

Orchestration

The **portability** and **reproducibility** of a containerized process means we have an opportunity to **move** and **scale** our containerized applications across clouds and data centers. Containers effectively *guarantee that those applications run the same way anywhere*, allowing us to quickly and easily take advantage of all these environments. Furthermore, as we scale our applications up, we'll want some tooling to help **automate the maintenance** of those applications, able to replace failed containers automatically and manage the rollout of updates and reconfigurations of those containers during their lifecycle.



> Containers are great, but when you get lots of them running, at some point, you need them **all working together in harmony** to solve business problems.
>
> The problem is, when you have lots of containers running, they need to be managed. There must be **enough capacity** to handle the load, but **no overhead** as it will bog down the machines in the cluster. And well, from time to time, containers are going to crash, and need to be restarted.

In other words › All those containers need to be orchestrated.

Tools to manage, scale, and maintain containerized applications are called **orchestrators**, and the most common examples of these are **Kubernetes** and **Docker Swarm**. Development environment deployments of both of these orchestrators are provided by Docker Desktop, which we'll use throughout this guide to create our first orchestrated, containerized application.

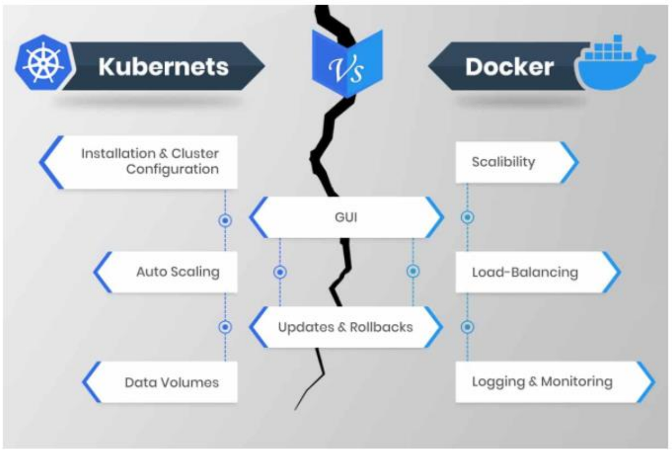
Importance of Orchestration

> Container orchestration is used to automate the following tasks at scale:

- Provisioning and deployments of containers
- Availability of containers
- Load balancing, traffic routing and service discovery of containers
- Health monitoring of containers
- Securing the interactions between containers.
- Configuring and scheduling of containers
- The configuration of applications in terms of the containers that they run in
- Scaling of containers to equally balance application workloads across infrastructure
- Allocation of resources between containers

Container Orchestration Work

Container orchestration works with tools like *Kubernetes* and *Docker Swarm*. Configurations files tell the container orchestration tool for how to **network between containers** and where to **store logs**. The orchestration tool also **schedules** the deployment of containers into clusters and determines the **best host** for the container. After a host is decided, the orchestration tool *manages the lifecycle of the container* based on predetermined specifications. Container orchestration tools work in any environment that runs containers.



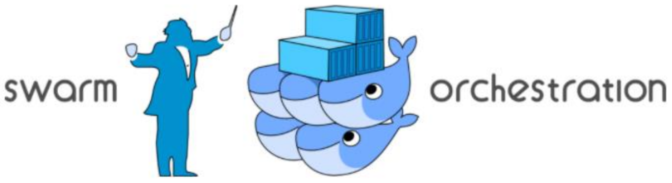
> Orchestration tools for Docker include the following:

- Docker Machine — Provisions hosts and installs Docker Engine.
- Docker Compose — Deploys multi-container applications by creating the required containers.
- Docker Swarm — Clusters multiple Docker hosts under a single host. It can also integrate with any tool that works with a single Docker host.

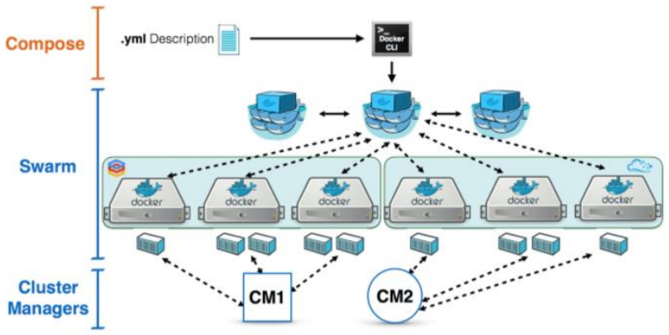
What is Docker Swarm?

Swarm is Docker's native support for orchestrating clusters of Docker engines.

Docker Swarm › A container orchestration tool built and managed by Docker, Inc.



A Docker Swarm is a group of either physical or virtual machines that are running the Docker application and that have been configured to **join together in a cluster**. Once a group of machines have been **clustered together**, you can still run the Docker commands that you're used to, but they will now be carried out by the machines in your cluster. The activities of the cluster are controlled by a **swarm manager**, and machines that have joined the cluster are referred to as *nodes*.



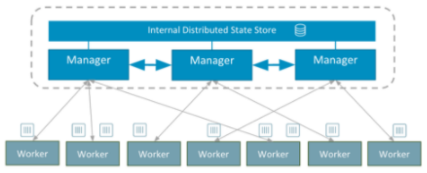
> Q: What is a Docker Swarm?
> A: **Docker Swarm** is native clustering for Docker. It turns a pool of Docker hosts into a single, virtual Docker host. Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts.
>
> — Interview Q&A

Docker Swarm Definitions

A docker swarm is comprised of a group of *physical* or *virtual* machines operating in a cluster. When a machine joins the cluster, it becomes a node in that swarm.

Nodes

The docker swarm function recognizes two types of nodes, each with a different role within the docker swarm ecosystem:



> Manager Node:

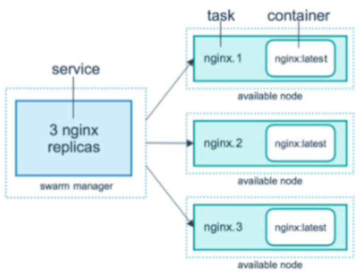
The primary function of manager nodes is *to assign tasks to worker nodes* in the swarm. Manager nodes also help to carry out some of the *managerial tasks* needed to operate the swarm. Docker recommends a maximum of seven manager nodes for a swarm.

> Worker Node:

In a docker swarm with numerous hosts, each worker node functions by receiving and executing the tasks that are allocated to it by manager nodes. By default, all manager nodes are also worker nodes and are capable of executing tasks when they have the resources available to do so.

Services and Tasks

A service is the definition of the tasks to execute on the manager or worker nodes. It is the central structure of the swarm system and the primary root of user interaction with the swarm.

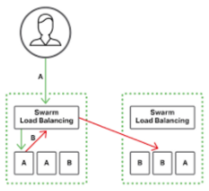


When you create a service, you specify which container image to use and which commands to execute inside running containers.

A task carries a Docker container and the commands to run inside the container. **It is the atomic scheduling unit of swarm**. Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale. Once a task is assigned to a node, it cannot move to another node. It can only run on the assigned node or fail.

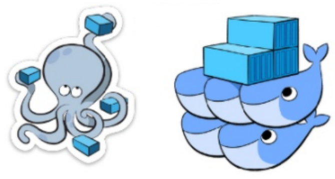
Load balancing

The swarm manager uses ingress load balancing to expose the services you want to make available externally to the swarm. The swarm manager can automatically assign the service a PublishedPort or you can configure a PublishedPort for the service. You can specify any unused port. If you do not specify a port, the swarm manager assigns the service a port in the 30000-32767 range.



Benefits of Docker Swarm

Use the Docker Engine CLI to create a swarm of Docker Engines where you can deploy application services. You don't need additional orchestration software to create or manage a swarm



Decentralized design: Instead of handling differentiation between node roles at deployment time, the Docker Engine handles any specialization at runtime. You can deploy both kinds of nodes, managers and workers, using the Docker Engine. This means you can build an entire swarm from a single disk image.

Declarative service model: Docker Engine uses a declarative approach to let you define the desired state of the various services in your application stack. For example, you might describe an application comprised of a web front end service with message queueing services and a database backend.

Scaling: For each service, you can declare the number of tasks you want to run. When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.

Desired state reconciliation: The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and your expressed desired state. For example, if you set up a service to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.

Multi-host networking: You can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.

Service discovery: Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.

Load balancing: You can expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.

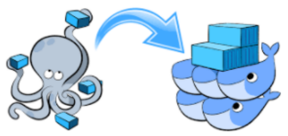
Secure by default: Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between itself and all other nodes. You have the option to use self-signed root certificates or certificates from a custom root CA.

Rolling updates: At rollout time you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service deployment to different sets of nodes. If anything goes wrong, you can roll back to a previous version of the service.

Enable Docker Swarm

Docker Swarm Mode comes integrated with Docker Platform. Starting 1.12, Docker Swarm Mode is rightly integrated which means that you don't need to install anything outside to run Docker Swarm.

> Docker Desktop runs primarily on Docker Engine, which has everything you need to run a Swarm built-in. Follow the setup and validation instructions appropriate for your operating system:



Mac

1. Open a terminal, and initialize Docker Swarm mode:

```
docker swarm init
```

> If all goes well, you should see a message similar to the following:

```
Swarm initialized: current node (tjjggogqpnpj2phbfbz8jd5oq) is now a manager.

To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1
        -3e0hh0jd5t4yjg209f4g5qpowbsczfahv2dea9a1ay218787cf
        -2h4ly330d0j917ocvzw30j5x9 192.168.65.3:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow the
    instructions.
```

2. Run a simple Docker service that uses an alpine-based filesystem, and isolates a ping to 8.8.8.8:

```
docker service create --name demo alpine:3.5 ping 8.8.8.8
```

3. Check that your service created one running container:

```
docker service ps demo
```

> You should see something like:

```
ID                  NAME                IMAGE               NODE
    DESIRED STATE       CURRENT STATE           ERROR               PORTS
axjrrjzkvxnw        demo.1              alpine:3.5          docker-desktop
    Running             Running 19 seconds ago
```

4. Finally, tear down your test service:

```
docker service rm demo
```

Windows

1. Open a powershell, and initialize Docker Swarm mode:

```
docker swarm init
```

2. Run a simple Docker service that uses an alpine-based filesystem, and isolates a ping to 8.8.8.8:

```
docker service create --name demo alpine:3.5 ping 8.8.8.8
```

3. Check that your service created one running container:

```
docker service ps demo
```

> You should see something like:

```
ID                  NAME                IMAGE               NODE
    DESIRED STATE       CURRENT STATE           ERROR              PORTS
463j2s3y4b5o        demo.1              alpine:3.5          docker-desktop
    Running             Running 8 seconds ago
```

4. Finally, tear down your test service:

```
docker service rm demo
```

Windows

1. Open a powershell, and initialize Docker Swarm mode:

```
docker swarm init
```

2. Run a simple Docker service that uses an alpine-based filesystem, and isolates a ping to 8.8.8.8:

```
docker service create --name demo alpine:3.5 ping 8.8.8.8
```

3. Check that your service created one running container:

```
docker service ps demo
```

> You should see something like:

```
ID                  NAME                IMAGE               NODE
    DESIRED STATE       CURRENT STATE           ERROR              PORTS
463j2s3y4b5o        demo.1              alpine:3.5          docker-desktop
    Running             Running 8 seconds ago
```

4. Finally, tear down your test service:

```
docker service rm demo
```

Complementary Interactive Lesson about What's Docker Swarm