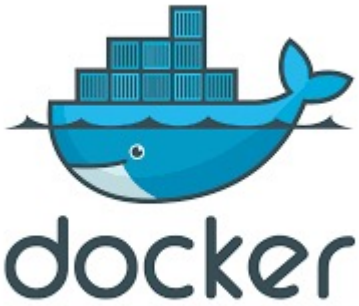
Docker

Containers

Shipping Containers

- Shipping containers are ubiquitous, standardized, and available anywhere in the world.
- The contents of each container are kept isolated from that of the others.

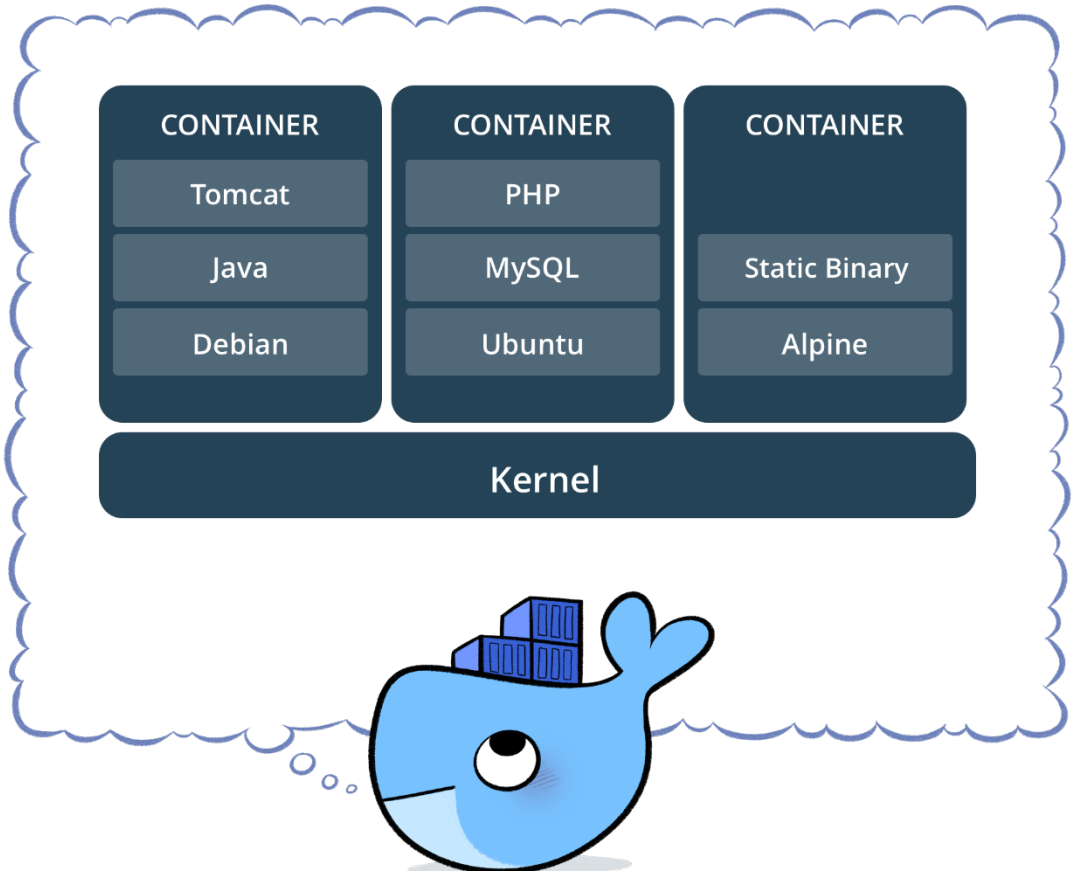
Software Containers



- Packing the container involves defining what needs to be there for your application to work – operating system, libraries, configuration files, application binaries, and other parts of your technology stack.
- Once the container has been defined, that "image" is used to create containers that run in any environment, from the developer's laptop to your test/QA rig, to the production data center, on-premises or in the cloud, without any changes.
- This consistency can be very useful: for example, a support engineer can spin up a container to replicate an issue and be confident that it exactly matches what's running.

Software Containers

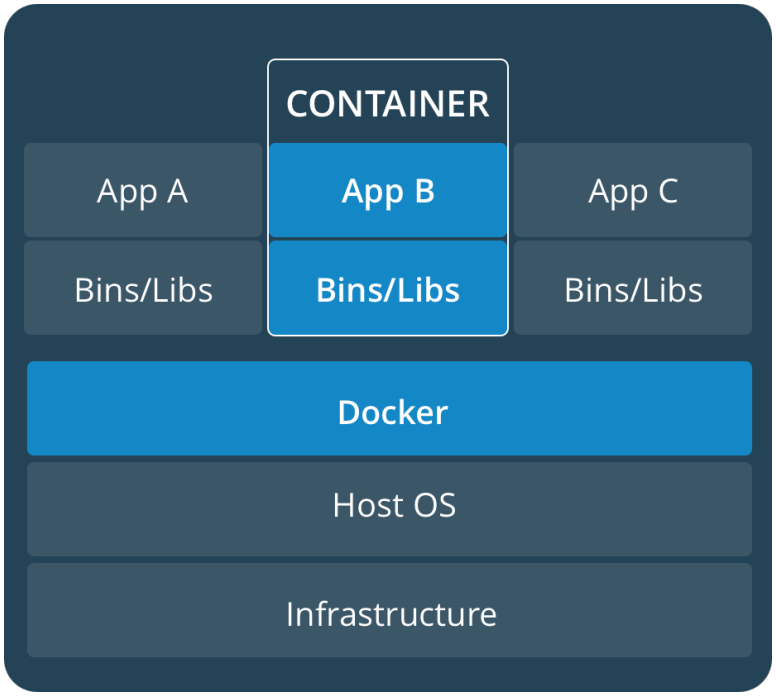
- A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings.

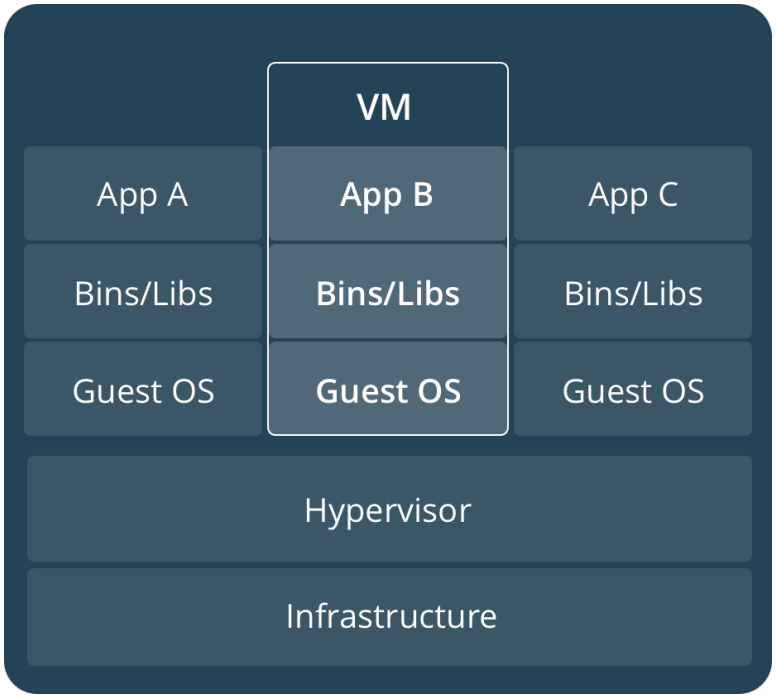


Containers VS Virtual Machines

- Multiple containers share the OS kernel. Containers take up less space than VMs, and start almost instantly.

- Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. Each VM includes a full copy of an operating system, one or more apps, necessary binaries and libraries.





How Containers Benefit Your Business?

- With modern web services:
  - users expect applications to be available 24/7
  - developers expect to deploy new versions of those applications several times a day.

How Containers Benefit Your Business?

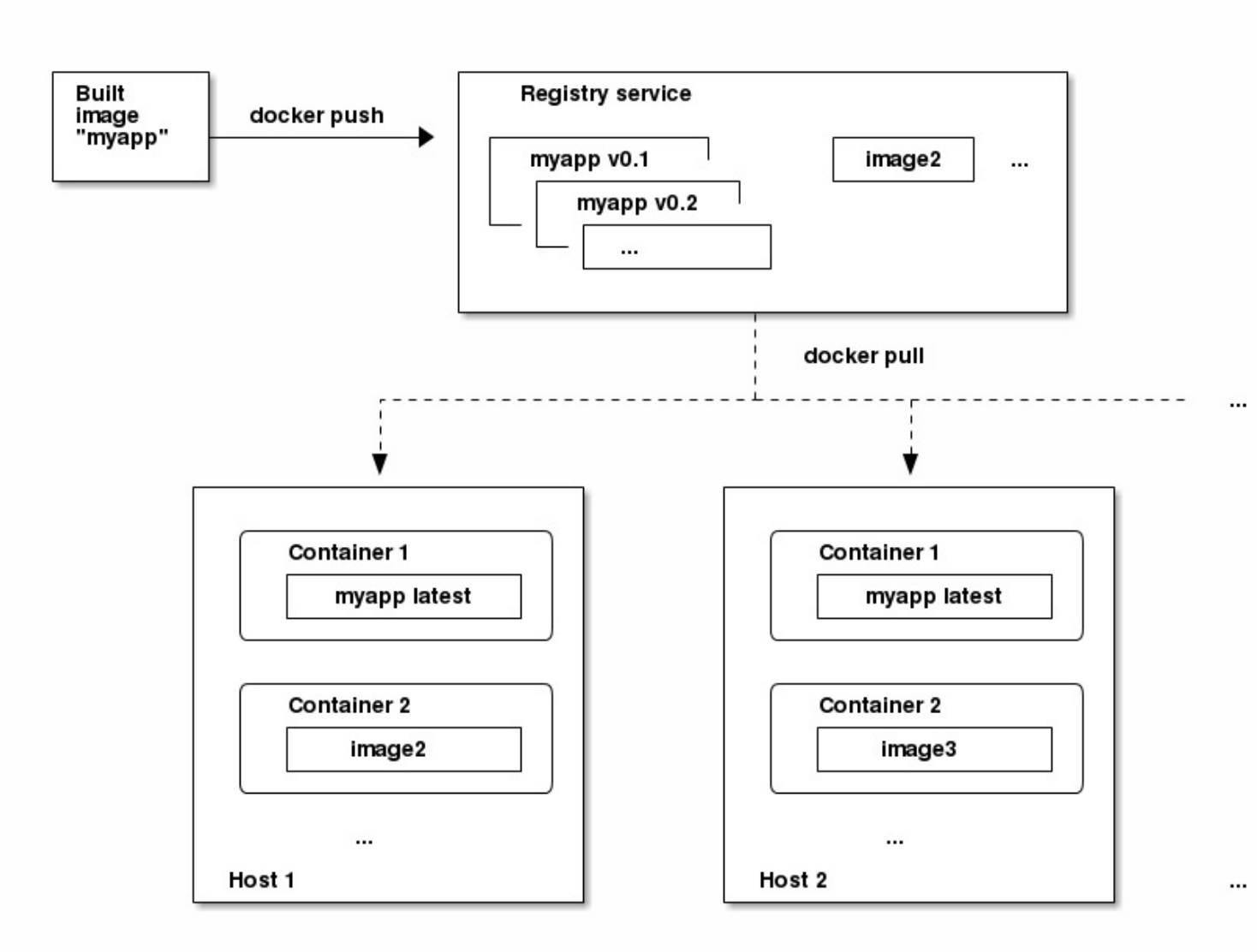
- **Isolation**. Every container running on the same host is independent and isolated from the others as well as from the host itself.
- **Performance**. Unlike VMs containers are lightweight and have minimal impact on performance.
- **High Availability**. With the addition of some automation, failed containers can be automatically recreated (rescheduled) either on the same or a different host, restoring full capacity and redundancy.
- **Scalability**. By architecting an application to be built from multiple container instances, adding more containers scales out capacity and throughput.
- **Replicating Environments**. When using containers, it's a trivial matter to instantiate identical copies of your full application stack and configuration for partners, support teams…
- **Accurate Testing**. You can have confidence that your test environment exactly matches what will be deployed – down to the exact version of every library.

Docker

Images, Registry and Containers

- A Docker image is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing its needs and capabilities.
- A container is a runtime instance of a docker image. A Docker container consists of:
    - A Docker image
    - An execution environment
    - A standard set of instructions
- A Docker registry is a service for storing and retrieving Docker images. A registry contains a collection of one or more Docker image repositories. Each image repository contains one or more tagged images.

Docker images, registry and containers



Dockerfile

Dockerfile

- Compose is a tool for Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession.

Dockerfile

- FROM

```
FROM <image> [AS <name>]
Or
FROM <image>[:<tag>] [AS <name>]
Or
FROM <image>[@<digest>] [AS <name>]
```

- The FROM instruction initializes a new build stage and sets the Base Image for subsequent instructions. As such, a valid Dockerfile must start with a FROM instruction. The image can be any valid image – it is especially easy to start by pulling an image from the Public Repositories.

```
ARG  CODE_VERSION=latest
FROM base:${CODE_VERSION}
CMD  /code/run-app

FROM extras:${CODE_VERSION}
CMD  /code/run-extras
```

Dockerfile

- A Dockerfile must start with a `FROM` instruction. The FROM instruction specifies the Base Image from which you are building. FROM may only be preceded by one or more ARG instructions, which declare arguments that are used in FROM lines in the Dockerfile.
- Environment variables (declared with the ENV statement) can also be used in certain instructions as variables to be interpreted by the Dockerfile. Escapes are also handled for including variable-like syntax into a statement literally.

  Environment variables are notated in the Dockerfile either with $variable_name or ${variable_name}

```
FROM busybox
ENV foo /bar
WORKDIR ${foo}   # WORKDIR /bar
ADD . $foo       # ADD . /bar
COPY \$foo /quux # COPY $foo /quux
```

Dockerfile

- RUN : execute commands inside of a Docker image. These commands get executed once at build time and get written into your Docker image. Example: RUN mkdir /path/to/folders

- RUN has 2 forms:

  RUN <command> (shell form, the command is run in a shell, which by default is /bin/sh -c on Linux or cmd /S /C on Windows)
  RUN ["executable", "param1", "param2"] (exec form)

- The RUN instruction will execute any commands in a new layer on top of the current image and commit the results.

Dockerfile

CMD : define a default command to run when the container starts. Need to re-build the Docker image. Example: Start the web application's app server

- The CMD instruction has three forms:

  CMD ["executable","param1","param2"] (exec form, this is the preferred form)
  CMD ["param1","param2"] (as default parameters to ENTRYPOINT)
  CMD command param1 param2 (shell form)

- There can only be one CMD instruction in a Dockerfile. If you list more than one CMD then only the last CMD will take effect.
- The main purpose of a CMD is to provide defaults for an executing container.

Dockerfile

- LABEL
- LABEL <key>=<value> <key>=<value> <key>=<value> ...
- The LABEL instruction adds metadata to an image. A LABEL is a key-value pair. To include spaces within a LABEL value, use quotes and backslashes as you would in command-line parsing. A few usage examples:

  ```
  LABEL "com.example.vendor"="ACME Incorporated"
  LABEL com.example.label-with-value="foo"
  LABEL version="1.0"
  LABEL description="This text illustrates \
  that label-values can span multiple lin
  ```

Dockerfile

- EXPOSE
- EXPOSE <port> [<port>/<protocol>...]

- The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

```
EXPOSE 80/udp
```

Dockerfile

- ENV

  ENV <key> <value>
  ENV <key>=<value> ...
  The ENV instruction sets the environment variable <key> to the value <value>

- ADD

- ADD has two forms:

  ADD [--chown=<user>:<group>] <src>... <dest>
  ADD [--chown=<user>:<group>] ["<src>",... "<dest>"] (this form is required for paths containing whitespace)

The ADD instruction copies new files, directories or remote file URLs from <src> and adds them to the filesystem of the image at the path <dest>.

Dockerfile

- COPY

- COPY has two forms:

  COPY [--chown=<user>:<group>] <src>... <dest>
  COPY [--chown=<user>:<group>] ["<src>",... "<dest>"] (this form is required for paths containing whitespace)

The COPY instruction copies new files or directories from <src> and adds them to the filesystem of the container at the path <dest>.

- ENTRYPOINT

- ENTRYPOINT has two forms:

  ENTRYPOINT ["executable", "param1", "param2"] (exec form, preferred)
  ENTRYPOINT command param1 param2 (shell form)
  An ENTRYPOINT allows you to configure a container that will run as an executable.

Dockerfile

- VOLUME
- VOLUME ["/data"]
- The VOLUME instruction creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.
- USER
- USER <user>[:<group>] or
  USER <UID>[:<GID>]
- The USER instruction sets the user name (or UID) and optionally the user group (or GID) to use when running the image and for any RUN, CMD and ENTRYPOINT instructions that follow it in the Dockerfile.

Dockerfile

- WORKDIR
- WORKDIR /path/to/workdir
- The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.
- ARG
- ARG <name>[=<default value>]
- The ARG instruction defines a variable that users can pass at build-time to the builder with the docker build command using the --build-arg <varname>=<value> flag.

Dockerfile

- ONBUILD
- ONBUILD [INSTRUCTION]
- The ONBUILD instruction adds to the image a trigger instruction to be executed at a later time, when the image is used as the base for another build.
- STOPSIGNAL
- STOPSIGNAL signal
- The STOPSIGNAL instruction sets the system call signal that will be sent to the container to exit.

Dockerfile

- HEALTHCHECK
- The HEALTHCHECK instruction has two forms:

  HEALTHCHECK [OPTIONS] CMD command (check container health by running a command inside the container)

  HEALTHCHECK NONE (disable any healthcheck inherited from the base image)
  The HEALTHCHECK instruction tells Docker how to test a container to check that it is still working.

- EXPOSE <port> [<port>/<protocol>...]

- The EXPOSE instruction informs Docker that the container listens on the specified network ports at runtime. You can specify whether the port listens on TCP or UDP, and the default is TCP if the protocol is not specified.

  EXPOSE 80/udp

Dockerfile

- ENTRYPOINT: configure a container that will run as an executable. Example: ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]
- ENTRYPOINT has two forms:
    - The exec form, which is the preferred form: ENTRYPOINT ["executable", "param1", "param2"]
    - The shell form: ENTRYPOINT command param1 param2

Docker Compose

Docker Compose

- Compose is a tool for defining and running multi-container Docker applications.
- A YAML file configures the application's services. Then, with a single command, you create and start all the services from your configuration.
- Common use cases:
    - Development environments: configure all of the application's service dependencies (databases, queues, caches, web service APIs, etc) in multiple containers.
    - Automated testing environments: Compose provides a convenient way to create and destroy isolated testing environments for your test suite.
    - Single host deployments: Compose to deploy to a remote Docker Engine. The Docker Engine may be a single instance provisioned with Docker machine or an entire Docker Swarm cluster

The docker-compose.yml file

```
version: "3"
services:
  web:
    # replace username/repo:tag with your
name and image details
    image: username/repo:tag
    deploy:
      replicas: 5
      resources:
        limits:
          cpus: "0.1"
          memory: 50M
        restart_policy:
          condition: on-failure
    ports:
      - "80:80"
    networks:
      - webnet
networks: webnet:
```

Run 5 instances of the image as a service called web, limiting each one to use, at most, 10% of the CPU (across all cores), and 50MB of RAM

Immediately restart containers if one fails

Map port 80 on the host to web's port 80

containers share port 80 via a load-balanced network called webnet

Define the webnet network with the default settings (which is a load-balanced overlay network)

Docker Compose features

- Multiple isolated environments on a single host: compose uses a project name to isolate environments from each other.
- Preserve volume data when containers are created: compose preserves all volumes used by your services. When docker-compose up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container. This process ensures that any data you've created in volumes isn't lost.
- Only recreate containers that have changed: compose caches the configuration used to create a container. When you restart a service that has not changed, Compose re-uses the existing containers
- Variables and moving a composition between environments: compose supports variables in the Compose file. You can use these variables to customize your composition for different environments, or different users.

Orchestration

Orchestration

- The process of deploying multiple containers becomes more and more valuable as the number of containers and hosts grow. This type of automation is referred to as orchestration.
- Orchestration can include a number of features, including:
    - Provisioning hosts
    - Instantiating a set of containers
    - Rescheduling failed containers
    - Linking containers together through agreed interfaces
    - Exposing services to machines outside of the cluster
    - Scaling out or down the cluster by adding or removing containers

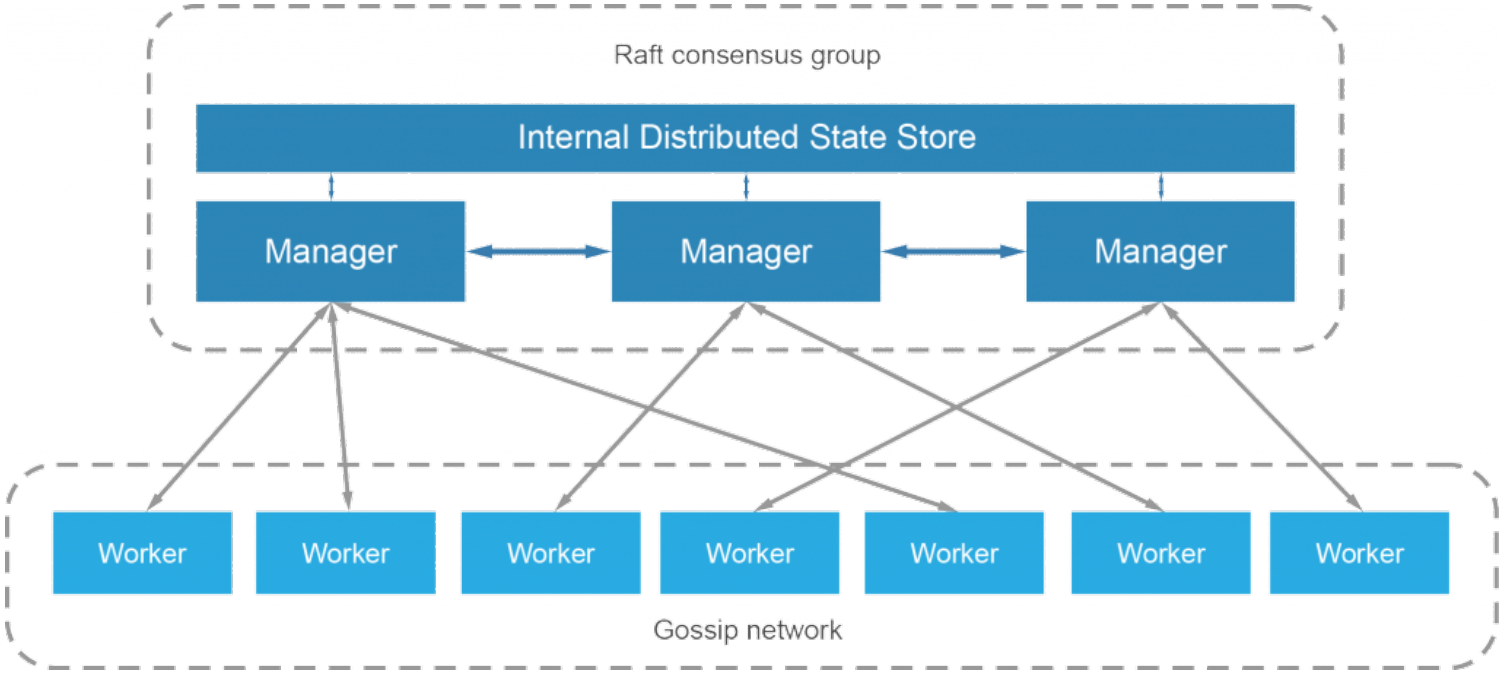
Orchestration tools

- **Docker Machine**: Provisions hosts and installs Docker Engine (the lightweight runtime and tooling used to run Docker containers) software on them.
- **Docker Swarm**: Produces a single, virtual Docker host by clustering multiple Docker hosts together. It presents the same Docker API; allowing it to integrate with any tool that works with a single Docker host.
- **Docker Compose**: Takes a file defining a multi-container application (including dependencies) and deploys the described application by creating the required containers. It is mostly aimed at development, testing, and staging environments.
- **Kubernetes**: was created by Google and is one of the most feature-rich and widely used orchestration frameworks.

Cluster of machines

using Swarm

Cluster


Raft consensus group
Internal Distributed State Store
Manager
Manager
Manager
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Gossip network

Cluster

- A swarm is a group of machines that are running Docker and joined into a cluster.
- The machines in a swarm can be physical or virtual.
- After joining a swarm, they are referred to as **nodes**.
- Swarm managers can use several strategies to run containers (described into the Compose file):
    - emptiest node: which fills the least utilized machines with containers.
    - Global: which ensures that each machine gets exactly one instance of the specified container.
    - ...