Middleware Architectures 2

Lecture 3: Cloud Native and Microservices

doc. Ing. Tomáš Vitvar, Ph.D.

tomas@vitvar.com • @TomasVitvar • https://vitvar.com

Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • https://vitvar.com/lectures

Modified: Mon Mar 07 2022, 14:16:19
Humla v1.0

Overview

- Cloud Native
- Microservices
- Containers
- Kubernetes

Overview

- The Cloud Native Computing Foundation (CNFS)
  – *Motto: Building sustainable ecosystems for cloud native software*
  – *CNFS is part of the nonprofit Linux Foundation*
- Cloud Native = scalable apps running in modern cloud environments
  – *containers, service mashes, microservices*
  – *Apps must be usually re-built from scratch or refactored*
  – *Benefits:*
    → *loosely coupled systems that are resilient, manageable, and observable*
    → *automation allowing for predictable and frequent changes with minimal effort*
  – *Trail Map*
    → *provides an overview for enterprises starting their cloud native journey*
- Lift and Shift
  – *Cloud transition program in organizations*
  – *Move app from on-premise to the cloud*
  – *Benefits*
    → *Infrastructure cost cutting (OPEX vs. CAPEX)*
    → *Improved operations (scaling up/down if possible can be faster)*

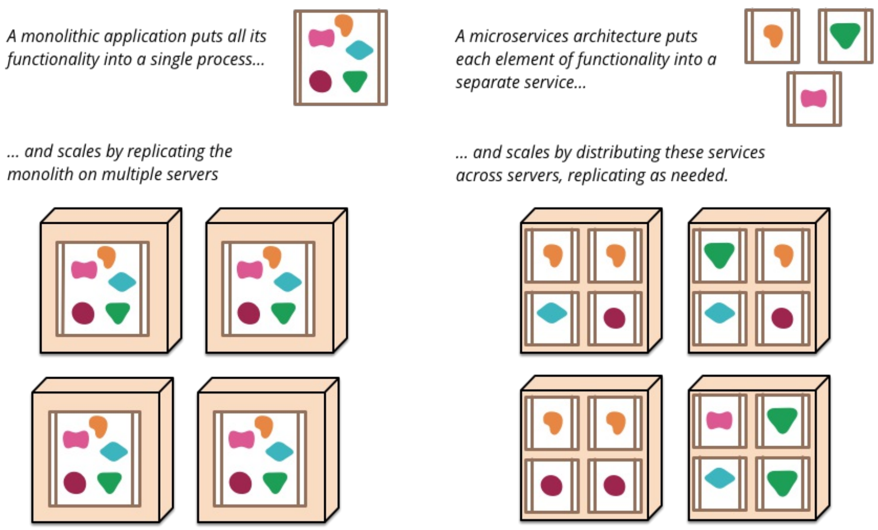
CNFS Trail Map

Overview

- Cloud Native
- Microservices
- Containers
- Kubernetes

Overview

- Emerging software architecture
  - *monolithic vs. decoupled applications*
  - *applications as independenly deployable services*



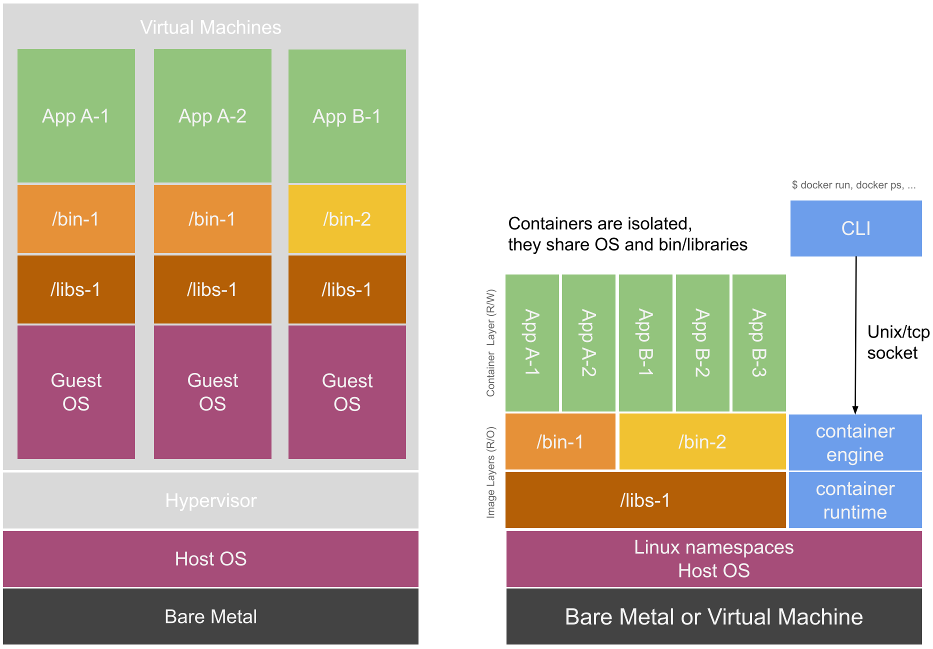
Major Characteristics

- Loosely coupled
  – *Integrated using well-defined interfaces*
- Technology-agnostic protocols
  – *HTTP, they use REST architecture*
- Independently deployable and easy to replace
  – *A change in small part requires to redeploy only that part*
- Organized around capabilities
  – *such as accounting, billing, recommendation, etc.*
- Impplemented using different technologies
  – *polyglot – programming languages, databases*

Overview

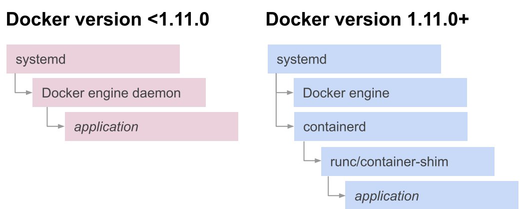
- Cloud Native
- Microservices
- Containers
  – *Overview*
  – *Linux Namespaces*
  – *Images*
  – *Working with Docker*
- Kubernetes

Virtual Machines vs. Containers

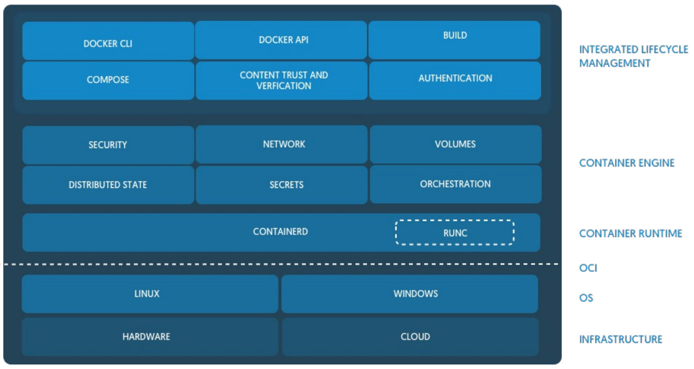


Overview

- Linux Containers
  - *Introduced in 2008*
  - *Allow to run a process tree in a isolated system-level "virtualization"*
  - *Use much less resources and disk space than traditional virtualization*
- Implementations
  - *LXC – default implementation in Linux*
  - *Docker Containers*
    - → *Builds on Linux namespaces and union file system (OverlayFS)*
    - → *A way to build, commit and share images*
    - → *Build images using a description file called Dockerfile*
    - → *Large number of available base and re-usable images*
- Monolithic design originally
  - *Now several layers*
  - *container runtime*
  - *container engine*



Containerd



- Container engine
  – *Accepts user inputs (via CLI or API), pulling images from registry, preparing metadata to be passed to container runtime*
- Container runtime
  – *Abstraction from syscalls or OS specific functionality to run containers on linux, windows, solaris, etc.*
  – *Uses* `runc` *and* `container-shim`
  – *Communicates with kernel to start containerized processes*

Terminology

- Image
  – *An image contains a union of layered filesystems stacked on top of each other*
  – *Immutable, it does not have state and it never changes*
- Container
  – *One or more processes running in one or more isolated namespaces in a filesystem provided by the image*
- Container Engine/Runtime
  – *The core processes providing container capabilities on a host*
- Client
  – *An app (e.g. CLI, custom app), communicates with a container engine by its API*
- Registry
  – *A hosted service containing repository of images*
  – *A registry provides a registry API to search, pull and push images*
  – *Docker Hub is the default Docker registry*
- Swarm
  – *A cluster of one or more docker engines*

Overview

- Cloud Native
- Microservices
- Containers
  - *Overview*
  - *Linux Namespaces*
  - *Images*
  - *Working with Docker*
- Kubernetes

Linux Namespaces

- Isolation of Linux processes, there are **7 namespaces**
  - *Mount, UTS, IPC, PID, Network, User, Cgroup*
  - *By default, every process is a member of a default namespace of each type*
  - *In case no additional namespace configuration is in place, processes and all their direct children will reside in this exact namespace*
  - *Run* `lsns` *to check namespaces the process is in*

```
$ lsns
NS         TYPE  NPROCS   PID USER   COMMAND
4026531836 pid        2 30873 oracle -bash
4026531837 user     108  1636 oracle /bin/bash /u01/oracle/scripts/startWebLogicContainer.sh
4026531838 uts        2 30873 oracle -bash
4026531839 ipc        2 30873 oracle -bash
4026531840 mnt        2 30873 oracle -bash
4026531956 net      108  1636 oracle /bin/bash /u01/oracle/scripts/startWebLogicContainer.sh
4026532185 mnt       13 13542 oracle /bin/bash /u01/oracle/scripts/startNM_ohs.sh
4026532192 pid       13  2798 oracle /bin/bash /u01/oracle/scripts/startNM_ohs.sh
...
```

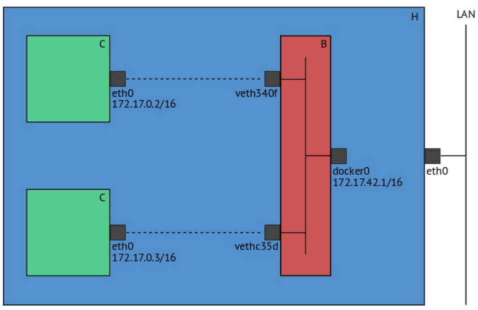
- Flexible configuration, for example:
  - *You can run two apps that only share the network namespace, e.g.* `4026531956`
  - *The apps can talk to each other*
  - *Any other app (not in this namespace) won't be able to talk to the apps*

Types: mnt, uts, ipc and pid

- `mnt` namespace
  – *Isolates filesystem mount points*
  – *Restricts the view of the global file hierarchy*
  – *Each namespace has its own set of mount points*
- `uts` namespace
  – *The value of the hostname is isolated between different UTS namespaces*
- `ipc` namespace
  – *Isolates interprocess communication resources*
  – *message queues, semaphore, and shared memory*
- `pid` namespace
  – *Isolates PID number space*
  – *A process ID number space gets isolated*
    → *Processes can have PIDs starting from the value 1*
    → *Real PIDs outside of the namespace of the same process is a different number*
  – *Containers have their own init processes with a PID value of 1*

Types: net

- `net` namespace
  – *Processes have their own private network stack (interfaces, routing tables, sockets)*
  – *Communication with external network stack is done by a virtual ethernet bridge*



  – *On the host there is a* ***userland proxy*** *or* ***NAT***
    → *NAT is a prefered solution over userland proxy (*`/usr/bin/docker-proxy`*)*
    → *Lack of NAT hairpinning may prevent to use NAT*
  – *Use case*
    → *Multiple services binding to the same port on a single machine, e.g.* `tcp/80`
    → *A port in the host is mapped to the port exposed by a process in the NS*

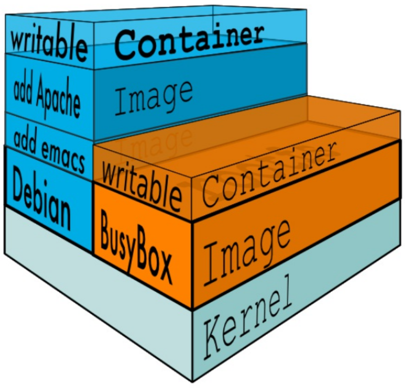
Types: user

- `user` namespace
  – *Isolates UID/GID number spaces*
- `cgroup` namespace
  – *Isolate cgroup root directory*

Overview

- Cloud Native
- Microservices
- Containers
  – *Overview*
  – *Linux Namespaces*
  – *Images*
  – *Working with Docker*
- Kubernetes

Container Images



– *Containers are made up of R/O layers via a storage driver (OverlayFS, AUFS, etc.)*
– *Containers are designed to support a single application*
– *Instances are ephemeral, persistent data is stored in bind mounts or data volume containers.*

Image Layering with OverlayFS

- OverlayFS
  – *A filesystem service implementing a **union mount** for other file systems.*
  – *Docker uses `overlay` and `overlay2` storage drivers to build and manage on-disk structures of images and containers.*
- Image Layering
  – *OverlayFS takes two directories on a single Linux host, layers one on top of the other, and provides a single unified view.*
  – *Only works for two layers, in multi-layered images hard links are used to reference data shared with lower layers.*

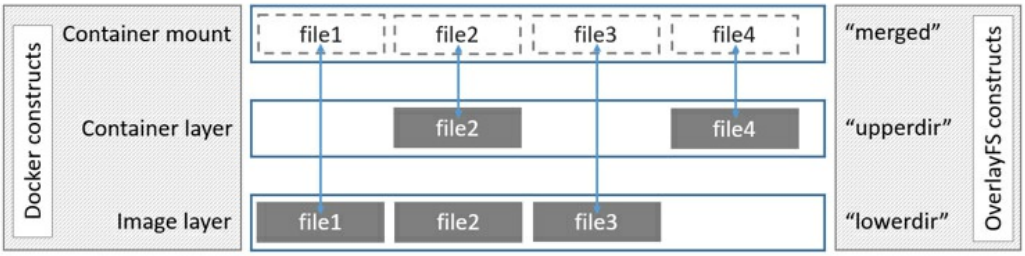


Image Layers Example

- Pulling out the image from the registry

```
$ docker pull ubuntu

Using default tag: latest
latest: Pulling from library/ubuntu

5ba4f30e5bea: Pull complete
9d7d19c9dc56: Pull complete
ac6ad7efd0f9: Pull complete
e7491a747824: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:46fb5d001b88ad904c5c732b086b596b92cfb4a4840a3abd0e35dbb6870585e4
Status: Downloaded newer image for ubuntu:latest
```

  – *Each image layer has its own directory under* `/var/lib/docker/overlay/`.
  – *This is where the contents of each image layer are stored.*

- Directories on the file system

```
$ ls -l /var/lib/docker/overlay/

total 20
drwx------ 3 root root 4096 Jun 20 16:11 38f3ed2eac129654acef11c32670b534670c3a06e483fce313d72e3e
drwx------ 3 root root 4096 Jun 20 16:11 55f1e14c361b90570df46371b20ce6d480c434981cbda5fd68c6ff61
drwx------ 3 root root 4096 Jun 20 16:11 824c8a961a4f5e8fe4f4243dab57c5be798e7fd195f6d88ab06aea92
drwx------ 3 root root 4096 Jun 20 16:11 ad0fe55125ebf599da124da175174a4b8c1878afe6907bf7c7857034
drwx------ 3 root root 4096 Jun 20 16:11 edab9b5e5bf73f2997524eebeac1de4cf9c8b904fa8ad3ec43b35041
```

  – *The organization of files allows for efficient use of disk space.*
  – *There are **files unique to every layer** and **hard links to files** shared with lower layers*

Dockerfile

- Dockerfile is a script that creates a new image

```
# This is a comment
FROM oraclelinux:7
MAINTAINER Tomas Vitvar <tomas@vitvar.com>
RUN yum install -q -y httpd
EXPOSE 80
CMD httpd -X
```

- A line in the Dockerfile will create an intermediary layer

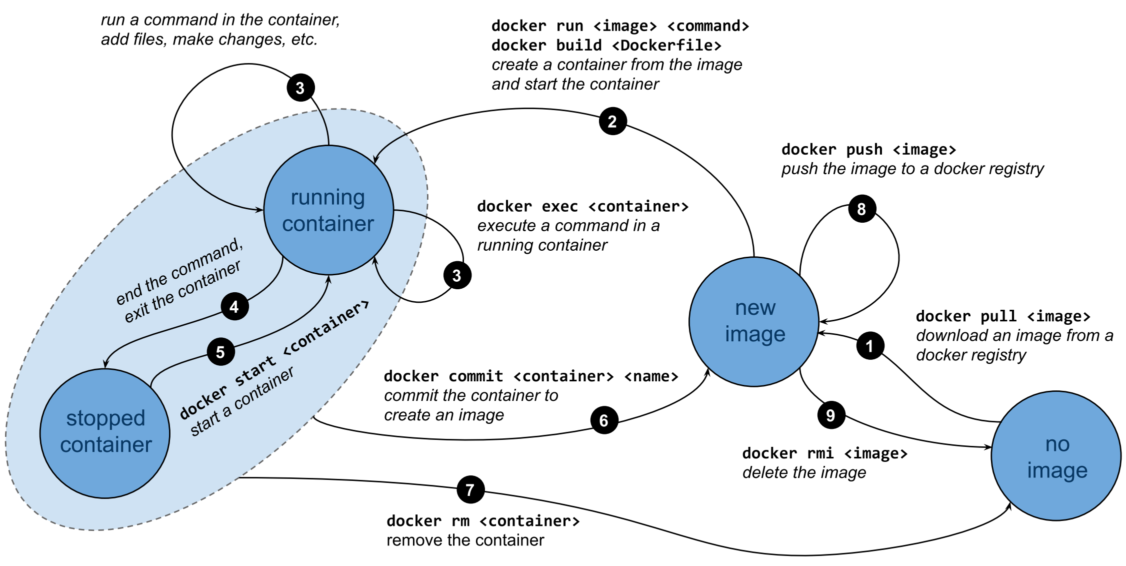
```
$ docker build -t tomvit/httpd:v1 .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM oraclelinux:7
 ---> 4c357c6e421e
Step 2 : MAINTAINER Tomas Vitvar <tomas@vitvar.com>
 ---> Running in 35feebb2ffab
 ---> 95b35d5d793e
Removing intermediate container 35feebb2ffab
Step 3 : RUN yum install -q -y httpd
 ---> Running in 3b9aee3c3ef1
 ---> 888c49141af9
Removing intermediate container 3b9aee3c3ef1
Step 4 : EXPOSE 80
 ---> Running in 03e1ef9bf875
 ---> c28545e3580c
Removing intermediate container 03e1ef9bf875
Step 5 : CMD httpd -X
 ---> Running in 3c1c0273a1ef
```

If processing fails at some step, all preceeding steps will be loaded from the cache on the next run.

Overview

- Cloud Native
- Microservices
- Containers
  - *Overview*
  - *Linux Namespaces*
  - *Images*
  - *Working with Docker*
- Kubernetes

Docker Container State Diagram



1: There is no image in the local store; you pull an image a remote registry.
2: You run a new container on top a specified image.
3: You modify the container by adding a library/content in it; you can also run a command in the container from the host.
4: You stop a running container.
5: You start a stopped container.
6: You commit the container and create a new image from it.
7: You remove the container.
8: You push the image to the remote registry.
9: You can remove the image from the local store.

Commands (1)

`docker version`

list current version of docker engine and client

`docker search <image>`

search for an image in the registry

`docker pull <image[:version]>`

download an image of a specific version from the registry
if the version is not provided, the latest version will be downloaded

`docker images`

list all local images

`docker run -it <image[:version]> <command>`

start the image and run the command inside the image
if the image is not found locally, it will be downloaded from the registry
option `-i` *starts the container in interactive mode*
option `-t` *allocates a pseudo TTY*

`docker ps [-as]`

list all running containers
option `-a` *will list all containers including the stopped ones.*
option `-s` *will list the container's size.*

Commands (2)

`docker rm <container>`

remove the container

`docker rmi <image>`

remove the image

`docker commit <container> <name[:version]>`

create an image from the container with the name and the version

`docker history <image>`

display the image history

Networking and Linking

- There are 3 docker networks by default
  - ***bridge*** – *container can access host's network (default)*
    - → *Docker creates subnet* `172.17.0.0/16` *and gateway to the network*
    - → *When a container is started, it is automatically added to this network*
    - → *All containers in this network can communicate with each other*
  - ***host*** – *all host's network interfaces will be available in the container.*
  - ***none*** – *container will be placed on its own network and no network interfaces will be configured.*
- Custom Network configuration
  - *You can create a new network and add containers to it*
  - *Containers in the new network can communicate with each other but the network will be isolated from the host network*
- Linking containers (legacy)

```
$ docker run -d --name redmine-db postgres
$ docker run -it --link redmine-db:db postgres /bin/bash
root@c4b12143ebe8:/# psql -h db -U postgres
psql (9.6.1)
Type "help" for help.
postgres=# SELECT inet_server_addr();
postgres=# SELECT * FROM pg_stat_activity \x\g\x
```


Networking Commands

`docker network ls`

lists all available networks

`docker network inspect <network-id>`

Returns the details of specific network

`docker network create --driver bridge isolated_nw`

creates a new isolated network

`docker run -it --network=isolated_nw ubuntu bin/bash`

starts the container ubuntu and attaches it to the isolated network

Data Volumes

- Data Volume
  – *A directory that bypass the union file system*
  – *Data volumes can be shared and reused among containers*
  – *Data volume persists even if the container is deleted*
  – *It is possible to mount a shared sotrage volume as a data volue by using a volume plugin to mount e.g. NFS*
- Adding a data volume

```
docker run -d -v /webapp training/webapp python app.py
```

will create a new value with name `webapp`*,*
the location of the volume can be determined by using `docker inspect`*.*

- Mount a host directory as a data volume

```
docker run -d -v /src/webapp:/webapp training/webapp python app.py
```

if the path exists in the container, it will be overlayed (not removed),
if the host directory does not exist, the docker engine creates it.

- Data volume container
  – *Persistent data to be shared among two or more containers*

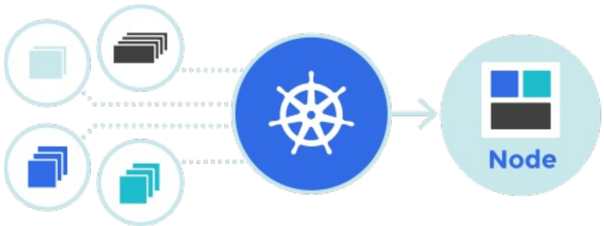
```
docker create -v /dbdata --name dbstore training/postgres /bin/true
docker run -d --volumes-from dbstore --name db1 training/postgres
docker run -d --volumes-from dbstore --name db2 training/postgres
```


Overview

- Cloud Native
- Microservices
- Containers
- Kubernetes

Overview

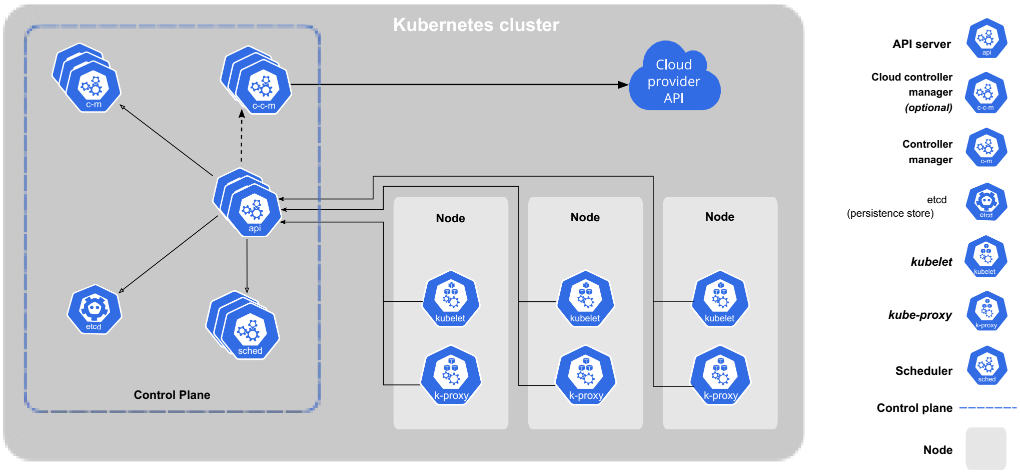
- In your architecture...
  - *Containers are atomic pieces of application architecture*
  - *Containers can be linked (e.g. web server, DB)*
  - *Containers access shared resources (e.g. disk volumes)*
- Kubernetes
  - *Automation of deployments, scaling, management of containerized applications across number of nodes*
  - *Based on Borg, a parent project from Goolge*



Features

- Automatic binpacking
  - *Automatically places containers onto nodes based on their resource requirements and other constraints.*
- Horizontal scaling
  - *Scales your application up and down with a simple command, with a UI, or automatically based on CPU usage.*
- Automated rollouts and rollbacks
  - *Progressive rollout out of changes to application/configuration, monitoring application health and rollback when something goes wrong.*
- Storage orchestration
  - *Automatically mounts the storage system (local or in the cloud)*
- Self-healing
  - *Restarts containers that fail, replaces and reschedules containers when nodes die, kills containers that don't respond to user-defined health checks.*
- Service discovery and load balancing
  - *Gives containers their own IP addresses and a single DNS name for a set of containers, and can load-balance across them.*

Architecture



Control Plane Components

- Global decisions about the cluster
  – *Schedulling*
  – *Detecting and responding to cluster events, starting up new pods*
- kube-apiserver
  – *exposes the Kubernetes API*
  – *The API server is the front end for the Kubernetes control plane.*
- etcd
  – *highly-available key value store used to store all cluster data*
- kube-scheduler
  – *watches for newly created Pods with no assigned node*
  – *selects a node for Pods to run on.*
  – *Decision factors: resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications*

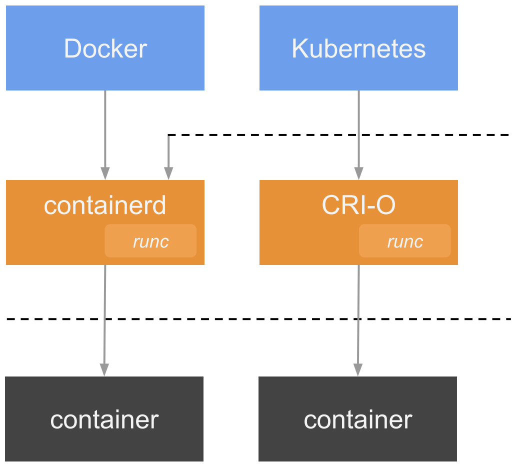
Control Plane Components

- kube-controller-manager
  - *runs controller to ensure the desired state of cluster objects*
  - ***Node controller***
    - *→ noticing and responding when nodes go down*
  - ***Job controller***
    - *→ creates Pods to run one-off tasks to completion.*
  - ***Endpoints controller***
    - *→ Populates the Endpoints object (that is, joins Services & Pods).*
- cloud-controller-manager
  - *Integration with cloud services (when the cluster is running in a cloud)*
  - ***Node controller***
    - *→ checks if a node has been deleted in the cloud after it stops responding*
  - ***Route controller***
    - *→ For setting up routes in the underlying cloud infrastructure*
  - ***Service controller***
    - *→ For creating, updating and deleting cloud provider load balancers*

Node

- Kubernetes runtime environment
  - *Run on every node*
  - *Maintaining running pods*
- kubelet
  - *An agent that runs on each node in the cluster*
  - *It makes sure that containers are running in a Pod.*
- kube-proxy
  - *maintains network rules on nodes*
  - *network rules allow network communication to Pods from inside or outside of the cluster*
  - *uses the operating system packet filtering layer or forwards the traffic itself.*
- Container runtime
  - *Responsible for running containers*
  - *Kubernetes supports several container runtimes (containerd, CRI-O)*
  - *Any implementation of the Kubernetes CRI (Container Runtime Interface)*

Container Stack



Tools to manager containers in both dev and prod environments.

Container Runtime Interface (CRI)
Kubernetes API defines how Kubernetes interact with container runtime.

Container runtimes that implement CRI spec. The containerd is a Docker component, it's CRI-compliant via *cri* plugin. CRI-O is a CRI implementation developed by RedHat/IBM.

Open Container Initiative (OCI)
A specification for container images and running containers.

Running containerized processes.

Pod

- Pod
  – *A group of one or more tightly-coupled containers.*
  – *Containers share storage and network resources.*
  – *A Pod runs a single instance of a given application*
  – *Pod's containers are always co-located and co-scheduled*
  – *Pod's containers run in a shared context, i.e. in a set of Linux namespaces*
- Pods are created using workload resources
  – *You do not create them directly*
- Pods in a Kubernetes cluster are used in two main ways
  – *Run a single container, the most common Kubernetes use case*
  – *Run multiple containers that need to work together*

Workloads

- An application running on Kubernetes
- Workloads run in a set of Pods
- Pre-defined workload resources to manage lifecylce of Pods
  - ***Deployment*** *and ReplicaSet*
    - → *managing a stateless application workload*
    - → *any Pod in the Deployment is interchangeable and can be replaced if needed*
  - ***StatefulSet***
    - → *one or more related Pods that track state*
    - → *For example, if a workload records data persistently, run a StatefulSet that matches each Pod with a persistent volume.*
  - *DaemonSet*
    - → *Ensures that all (or some) Nodes run a copy of a Pod*
    - → *Such as a cluster storage daemon, logs collection, node monitoring running on every node*
  - *Job and CronJob*
    - → *Define tasks that run to completion and then stop.*
    - → *Jobs represent one-off tasks, whereas CronJobs recur according to a schedule.*

Deployment Spec Example

- Deployment spec

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3 # tells deployment to run 3 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

  - *A desired state of an application running in the cluster*
  - *Kubernetes reads the Deployment spec and starts three app instances*
  - *If an instance fails, Kubernetes starts a replacement app instance*

Service

- Networking
  - *Containers within a Pod use networking to communicate via loopback*
  - *Cluster networking provides communication between different Pods.*
- Service resource
  - *An abstract way to expose an application running on a set of Pods*
  - *Example: a set of Pods with a label* `app=nginx`*, each listens on* `tcp/9376`

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

  - *This specification creates a new Service object named* `my-service`
  - *The servive targets* `tcp/9376` *on any Pod with the* `app=nginx` *label.*
  - *Kubernetes assigns this Service a cluster IP address, which is used by the Service proxies.*

Demo

- Environment Setup

  `minikube` – *a local virtual machine (running a master and a single node)*

  `kubectl` – *CLI to access Kubernetes cluster*
- Steps
  1. *create* `hello-node` *app in node.js and test it [see server.js]*
     `node server.js`
  2. *create docker image for the app [see Dockerfile]*
     `docker build -t hello-node:v1 .`
  3. *deploy the app to Kubernetes by using* `kubectl`
     `kubectl run hello-node --image=hello-node:v1 --port=8080`
  4. *Expose the app as a load balancer service.*
     `kubectl expose deployment hello-node --type=LoadBalancer`
  5. *Explore the app in minikube dashboard.*
     `minikube dashboard`
  6. *Fire requests at the service and count them [see test.sh]*
     `./test.sh`.
  7. *Change the number of replicas by using the dashboard or* `kubectl`.