Docker Bar Camp

Nicholas Dille, Docker Captain & Microsoft MVP

Nicholas Dille

Husband, father, geek, aikidoka

DevOps Engineer @ Haufe.Group

Microsoft MVP seit 2010

Docker Captain seit 2017

http://dille.name/blog

@NicholasDille

Build your own Agenda

Docker 101	Advanced Concepts	Remoting	Docker-in-Docker	Myths and (Anti)Patterns	Security
Continuous Integration	Continuous Delivery	Monitoring	Running in Production	Ecosystem	

Build your own Agenda

	Duration	Introduction	Advanced	Automation	DIY
Docker 101	90m	X			
Advanced Concepts	60m	X	X		
Remoting	30m	X	X		
Docker-in-Docker	30m		X		
Myths and (Anti)Patterns	60m	X	X		
Security	60m		X		
Continuous Integration	30m			X	
Continuous Delivery	60m			X	
Monitoring	45m				
Running in Production	60m			X	
Ecosystem	45m				

Play with Docker

Cloud-Service zum Erlernen von Docker

Kostenfreie Docker-Hosts für vier Stunden

Jeder Host nutzt Docker-in-Docker

https://labs.play-with-docker.com/

Tooling

Anmeldung per SSH (ssh -p 1022 10–0–1–3–48a594c4@host1.labs.play-with-docker.com)

Treiber für docker-machine (https://github.com/play-with-docker/docker-machine-driver-pwd)

Kudos an die Autoren

Docker Captains Marcos Nils und Jonathan Leibiusky

Prepare your environment

PWD

Go to https://play-with-docker.com

Create account and login

Create instance

Get example code

Clone GitHub repository

```
git clone https://github.com/nicholasdille/docker-ci-cd-examples.git
```

Docker 101

From zero to ~hero

Why is Docker so special?

Once upon a time... there were logical partitions (LPARs)

Released around 1972 by our parents

Next came Linux Containers (LXC)

Released 2008 by our older bothers and sisters

Interface to kernel features

The rise of Docker

Founded (as dotCloud) in 2013 by Solomon Hykes

Revolution of container management through automation

Concepts

Process isolation

Containerized processes cannot see the host OS

Containerized processes cannot see into other containers

Kernel is responsible

Isolation from other containers and the host

Resource management

What are containers?

#TBT: Container are process isolation

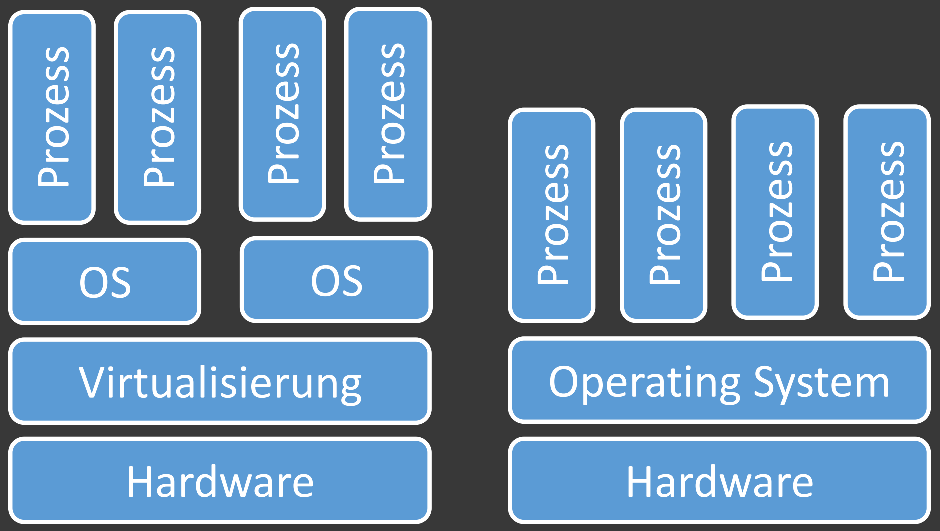
Implemented in the kernel

Resources are shared among all processes

Containers versus virtual machines

Other/higher layer of virtualization

Shared hardware and shared kernel



Containers are just another option

Advantages

Development

Reproducible environment

Packaged runtime environment

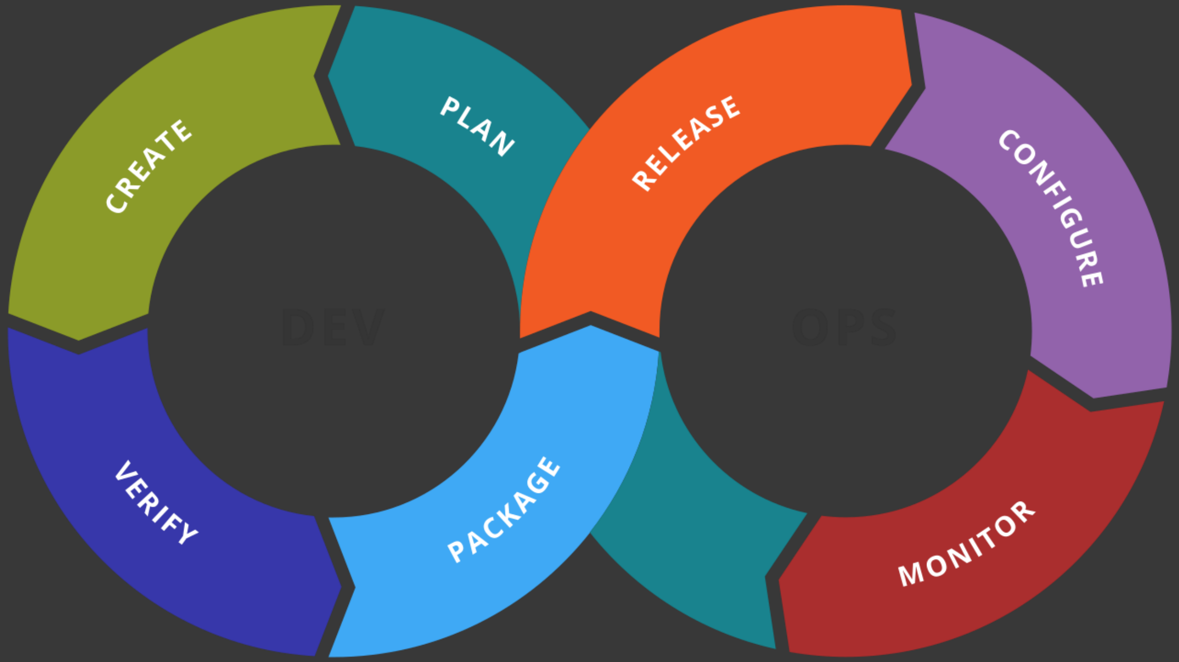
Deployable for testing

Operation

Lightweight virtualization

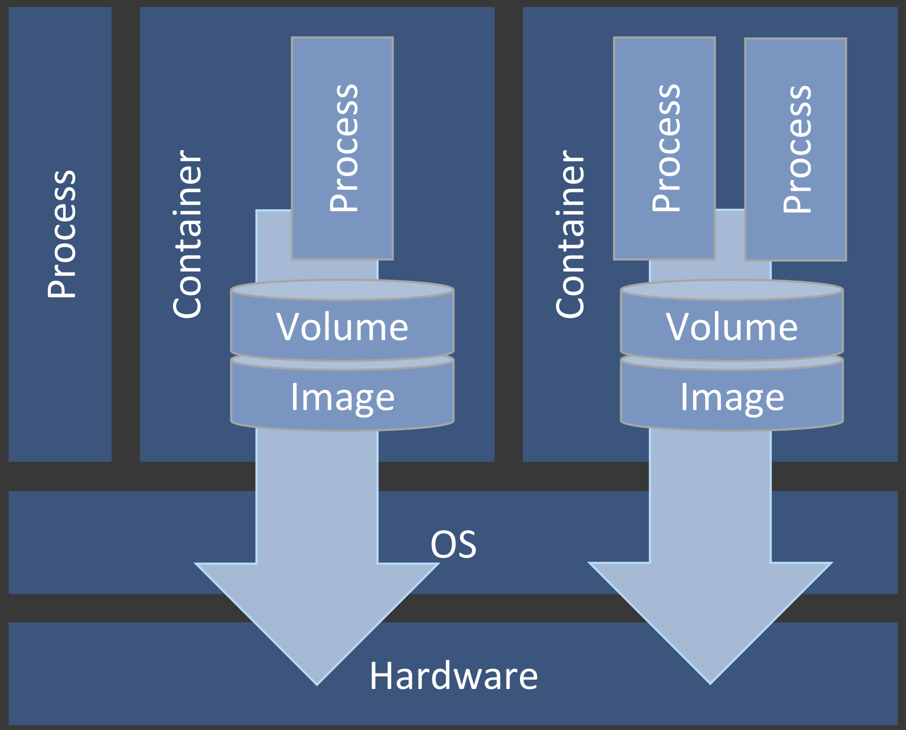
Density

Dependency management



Container Management

- Isolated process(es)
- Shared, read-only image
- Dedicated writable volume
- Network configuration



My First Container

Hands-On

```
$ docker run -it ubuntu
root@12345678# hostname
root@12345678# whoami
root@12345678# ps faux
root@12345678# ls -l /
root@12345678# exit
$ docker run -it ubuntu ping localhost
```

Background containers

First process keeps container alive

Containers are not purged automatically

Hands-On

```
$ docker run -it ubuntu hostname
$ docker run -d ubuntu ping localhost
$ docker ps
$ docker stop <NAME>
$ docker rm <NAME>
```

Exploration

Name containers

```
$ docker run -d --name websrv nginx
$ docker ps
```

Learn about containers

```
$ docker logs websrv
$ docker inspect websrv
```

Execute commands inside containers

```
$ docker exec websrv ps faux
$ ps faux
```

Enter containers interactively

```
$ docker exec -it websrv bash
```

Networking

Internals

Daemon controls 172.16.0.0/12

Containers are assigned a local IP address

Outgoing traffic is translated (source NAT)

Containers are not reachable directly

Incoming traffic requires published port

Published ports are mapped from the host to the container

Only one container can use a published port

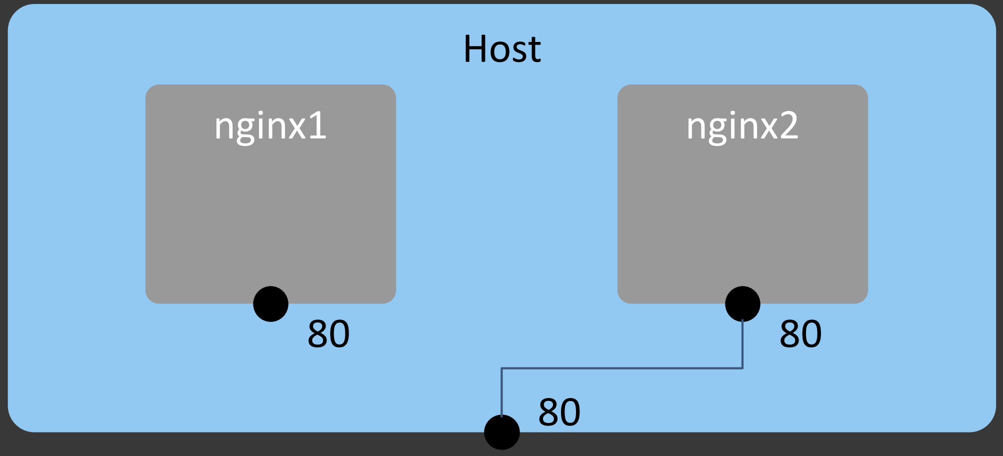
Hands-On

```
$ ifconfig docker0
```

```
$ docker run -d --name nginx1 nginx
```

```
$ docker run -d --name nginx2
-p 80:80 nginx
```

```
$ docker ps
```



Volumes

Non-persistent storage

```
$ docker run -it ubuntu
root@12345678# touch /file.txt
root@12345678# ls -l /
root@12345678# exit
$ docker run -it ubuntu
```

Locally persistent storage

```
$ docker run -it -v /source:/source ubuntu
root@12345678# touch /file
root@12345678# ls -l /
root@12345678# exit
$ docker run -it -v /source:/source ubuntu
root@12345678# ls -l /
```

Image Management

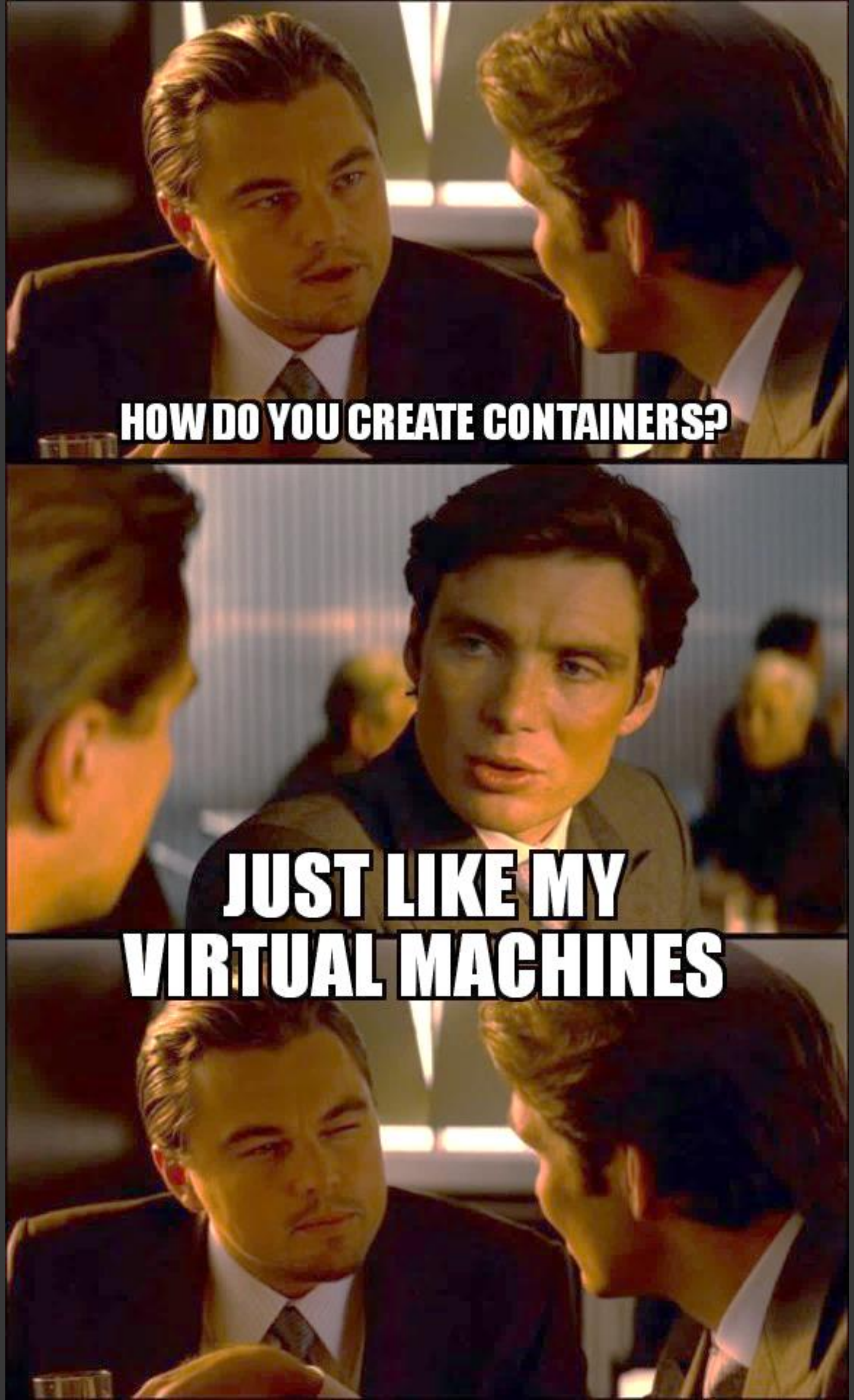
Images are served from Docker Hub

Images are named as **`user/name:tag`**

Hands-On

```
$ docker pull centos

$ docker rmi centos
```


HOW DO YOU CREATE CONTAINERS?
JUST LIKE MY VIRTUAL MACHINES

Custom Images

Custom behaviour

Based on existing image

Adds tools and functionality

Simple but sufficient scripting language

Hands-On

```
$ cat Dockerfile
FROM ubuntu:xenial
RUN apt update && apt -y install nginx

$ docker build --tag myimage .
```

Image Registries

Docker Hub is not the only source for images

Private registries based on Docker Distribution

Hands-On

```
$ docker tag oldname newname
$ docker push newname
```

Private Registry

Security

`localhost:5000` is preconfigured as insecure registry

Other registries must be secure (HTTPS)

Hands-On

```
$ docker run -d --name registry -p 5000:5000 registry
$ docker tag ubuntu localhost:5000/groot/ubuntu
$ docker push localhost:5000/groot/ubuntu
```

Advanced Concepts

From ~hero to hero

Image Management

Downside of docker CLI

Unable to search for image tags

Unable to remove images/tags from registry

Tools filling the gap

Search for image tags: `docker-ls`, `docker-browse`

Remove images/tags: `docker-rm` (part of **`docker-ls`**)

Multi-Stage Builds

New syntax in Dockerfile

Build in multiple steps

Multiple FROM sections

Advantages

Separate build environment and runtime environment

Smaller images

Smaller attack surface

Pipelines are more flexible than multi-stage builds

Kernel internal

Control groups (cgroups)

Resource management for CPU, memory, disk, network

Limitations and priorization

Accounting

Namespaces

Isolation of resources used in cgroups

What images are made of

Images have a naming scheme

`registry/name:tag`

On Docker Hub

- Official images: `[docker.io/_/]name:tag`
- Community images: `[docker.io/]user/name:tag`

Images are described by manifests

Images consist of layers

Instructions in Dockerfile create layers

What images are made of

Images

Name describes purpose

Tag describes version or variant

Images consist of layers

```
$ docker pull ubuntu
Using default tag: latest
latest: Pulling from library/ubuntu
9fb6c798fa41: Pull complete
3b61febd4aef: Pull complete
9d99b9777eb0: Pull complete
d010c8cf75d7: Pull complete
7fac07fb303e: Pull complete
Digest: sha256:31371c117d65387be2640b8254464102c36
Status: Downloaded newer image for ubuntu:latest
```

Layers…

…enable reuse

…increase download speed

Dockerfile and Layers

$ cat Dockerfile	$ docker history hello-world	
FROM ubuntu:xenial		
LABEL maintainer=nicholas.dille@haufe-lexware.com	2b99001c3d7f	0B
ENV JRE_VERSION=8u131	72e9a6db461b	0B
RUN apt-get update \ && apt-get -y install openjdk-8-jdk-headless=${JRE_VERSION}*	ad01f9c12cb6	225MB
ADD HelloWorld.java /tmp	542fdbd5f9b7	112B
RUN javac /tmp/HelloWorld.java	4121bfb4af35	426B
CMD ["java", "-classpath", "/tmp", "HelloWorld"]	c732f0a4d1e7	0B

Image Manifests

Images are described by manifests

Manifests also contain multi-arch information

Information is protected by SHA256 hashes

Hands-On

```
$ curl -sLH "Accept: application/vnd.docker.distribution.manifest.v2+json"
http://localhost:5000/v2/groot/ubuntu/manifests/latest

$ curl -L
http://localhost:5000/v2/groot/ubuntu/blobs/sha256:6b98dfc1607190243b0938e6
2c5ba2b7daedf2c56d7825dfb835208344705641 | wc -c

$ curl -sLH "Accept: application/vnd.docker.container.image.v1+json"
http://localhost:5000/v2/groot/ubuntu/manifests/latest
```

Multi-Arch Images

Images only work on a single platform

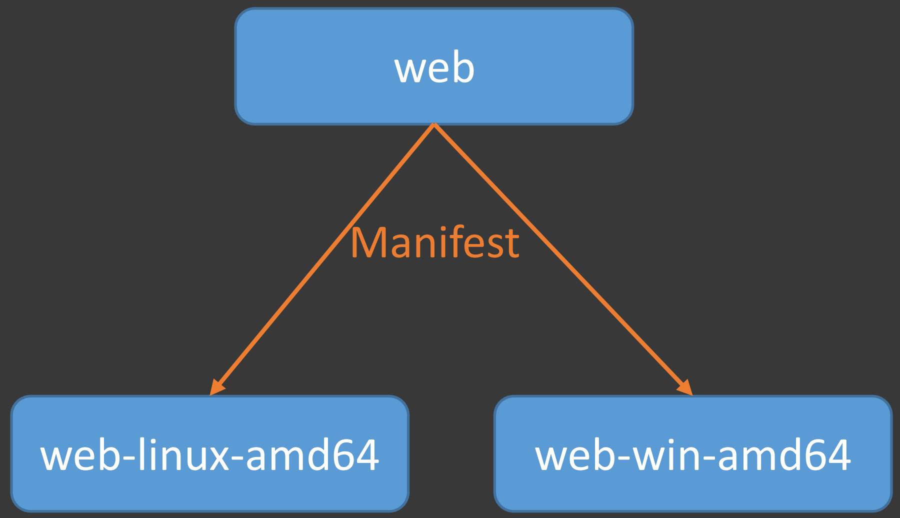
Containers are support on a multitude of architectures and operating systems

Solved by „virtual“ images

Manifest links to platform spacific image

https://github.com/estesp/manifest-tool
(by Docker Captain Phil Estes)

Official images have been migrated



Multi-Arch Images: openjdk

```
$ docker run mplatform/mquery openjdk:8-jdk
Image: openjdk:8-jdk
 * Manifest List: Yes
 * Supported platforms:
   - linux/amd64
   - linux/arm/v5
   - linux/arm/v7
   - linux/arm64/v8
   - linux/386
   - linux/ppc64le
   - linux/s390x

$ docker run mplatform/mquery openjdk:8-jdk-nanoserver
Image: openjdk:8-jdk-nanoserver
 * Manifest List: Yes
 * Supported platforms:
   - windows/amd64:10.0.14393.2363
```

Multi-Arch Images: hello-world

```
$ docker run mplatform/mquery hello-world
Image: hello-world
 * Manifest List: Yes
 * Supported platforms:
   - linux/amd64
   - linux/arm/v5
   - linux/arm/v7
   - linux/arm64/v8
   - linux/386
   - linux/ppc64le
   - linux/s390x
   - windows/amd64:10.0.14393.2363
   - windows/amd64:10.0.16299.547
   - windows/amd64:10.0.17134.165
```

Volume Management

What are volumes

Storage managed by Docker daemon

Locally persisted data

Hands-On

```
$ docker volume create myvolume
$ docker run -v myvolume:/mydata ubuntu  # check df
$ touch /mydata/myfile.txt
```

docker-compose

Infrastructure-as-Code

Deployment of multiple services

`docker-compose.yml` defines networks, volumes and services

Declarative syntax

Convergence

Changes are merged incrementally

Containers are re-created as necessary

docker-compose

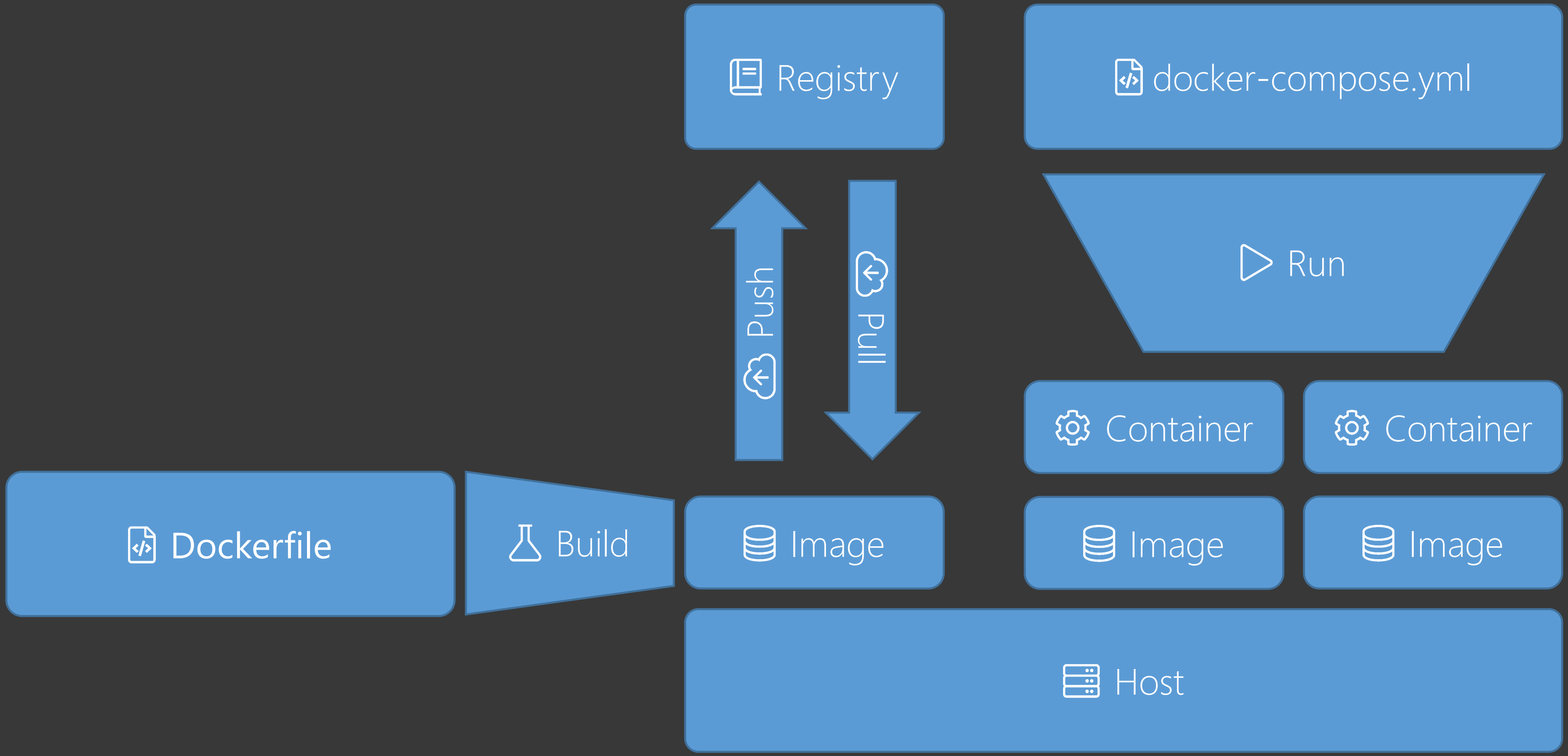
Hands-On

```
$ git clone https://github.com/nicholasdille/docker-ci-cd-demo.git
$ cd docker-ci-cd-demo
$ docker-compose up -d
$ docker ps
$ docker-compose down
$ docker-compose rm
```

Various helpful commands:

```
$ docker-compose ps
$ docker-compose logs
$ docker-compose exec registry bash
```

Tools Overview



Network Context

Default network context

Used by docker run

Containers are “on their own”

User-defined networks

Automatic DNS resolution of service names to containers

Used by docker-compose

docker-app

Package

Service definition in `docker-compose.yml`

Publish

...to Docker Hub

Examples in official repository

Remoting

Docker Engine

Daemon

…controls containers

…provides an API endpoint

Client

…controls the daemon using the API

Communication

Named pipe: `/var/run/docker.sock`

TCP socket: 2375 for HTTP and 2376 for HTTPS

Docker Engine API

Official API documentation

https://docs.docker.com/engine/api/v1.37/

```
$ docker version
$ curl --unix-socket /var/run/docker.sock http:/containers/json?all=true
```

In case of emergency

```
$ docker run -ti -v /var/run/docker.sock:/var/run/docker.sock
nathanleclaire/curl sh
/ # curl --unix-socket /var/run/docker.sock http:/containers/json
```

docker-machine

Management of multiple Docker hosts

Docker remoting using TCP socket

Installs certificates to secure communication

Support for cloud providers

Drivers extend the functionality

Creation of VMs

Examples

```
$ docker-machine create \
  --driver generic \
  --generic-ip-address=10.0.0.100 \
  --generic-ssh-key ~/.ssh/id_rsa \
  vmname
```

Docker-in-Docker

Containerized Docker CLI

Allows using other version

Works by mapping the named pipe

Creates conflicts with other containers

Interferes with the host

Example

```
docker run -it --rm -v /var/run/docker.sock:/var/run/docker.sock
docker:18.06 docker version
```

Inception

Docker-in-Docker

Running a containerized Docker daemon

Hands-On

```
$ docker run -d --rm --privileged --name dind docker:stable-dind
$ export DOCKER_HOST="tcp://$(docker inspect -f '{{range
.NetworkSettings.Networks}}{{.IPAddress}}{{end}}' dind):2375"
$ docker version
```

Myths and (Anti)Patterns

Notes from the Field

Myths and (Anti)Patterns

Designing Images

Tagging Strategies

Latest has not universal meaning

Individual version / build number

Meaningful named tags

`stable` for tested images

`dev` or `vnext` for untested images

More tags = choice

`v1.1.0` should also be tagged as `v1.1` and `v1`

Latest `v1.1.0-alpine` should also be tagged as `stable-alpine` and `alpine`

One Process per Container

The optimist

Separating functionality

Enabling scalability

The realist

Multiple processes in a container may make sense

Depends on server

See also Signal processing

Build versus Runtime

Build parameters

Versions of tools to be installed

Runtime parameters

Configure behaviour (see Tweaking runtime behaviour)

Demo

`advanced/build_versus_runtime`

Using Proxy Servers

Do not hardcore in Dockerfile

During build

```
docker run --build-arg http_proxy --build-arg https_proxy .
```

During runtime

```
docker run --env http_proxy --env https_proxy ubuntu
```

Docker daemon

Honours environment variables `http_proxy`, `https_proxy` and `no_proxy`

Tweaking Runtime Behaviour

ENV

Environment variables

Do not hardcode

Set reasonable defaults

ENTRYPOINT

Changes bahaviour on start

Hands-On

`advanced/runtime_behaviour`

Version Pinning versus Using latest

Downsides of using latest

Breaks reproducibility

Causes conflict with two services based on the same image

Version pinning in Dockerfile

Hard/impossible to determine running image version (see microlabeling)

Upsides of using latest

No need for version pinning

Strong downs but weak ups

Do not use latest

Dockerfile

```
FROM ubuntu
#...
```

Dockerfile

```
FROM nginx
#...
```

Dockerfile

```
FROM ubuntu:xenial-20180123
#...
```

Dockerfile

```
FROM nginx:1.12.1
#...
```

Derive from code

Using community images is like buying a pig in a poke

Community images may not receive updates

Community images may follow undesirable paths

Solution

Fork code repository and build yourself

Myths and (Anti)Patterns

Building Images

Signal Processing (PID 1)

Even containerized services want to exit gracefully

Only containerized PID 1 receives signals

Multiple processes require an init process

Choices include `supervisor`, `dumb-init`, `tini`

How to prevent init processes

Use `exec` when starting from scripts

Isolate in sidekicks

Signal Processing (PID 1)

Dockerfile

```
FROM ubuntu:xenial-20180123

RUN apt update \
 && apt install -y nginx

ADD entrypoint.sh /

ENTRYPOINT /entrypoint.sh
```

entrypoint.sh

```
#!/bin/bash

#...

exec nginx -g daemon=off;
```

Signal Processing (PID 1)

Dockerfile

```
FROM ubuntu:xenial-20180123

RUN apt update \
 && apt install -y \
        nginx \
        supervisor

ADD nginx.conf /etc/supervisor/conf.d/

ENTRYPOINT supervisord
```

nginx.conf

```
[program:nginx]
command=nginx -g daemon=off;
```

Readability beats size

Myth: More layers reduce access time

My own tests prove otherwise

Layers improve performance of pull (parallel downloads)

One layer per installed tool

Separate functionality into chains of images

dind → dind-gocd-agent
→ linux-agent → linux-agent-gocd
→ linux-agent-jenkins

Order of Statements

Build arguments

Used for controlling version pinning

Environment variables

Used for tweaking runtime behaviour

Tools and dependencies

E.g. Install distribution packages

New functionality

Packages and scripts required for the purpose of the image

The build cache helps you build faster!

Validate Downloads

Distribution packages are validated

Downloads from the web

Obtain file hash from the web

Create file hash after manual download

Check file hash during image build

```
$ echo “${HASH} *${FILENAME}“ | sha256sum
```

Run as USER

By default everything as root

Bad idea™

```
FROM ubuntu
# install
USER go
```

Force user context

Add USER statement after setting up image

Some services handle this for you (nginx)

Downstream images

Change to root

Install more tools

Change back to user

```
FROM derived
USER root
# install
USER go
```

Use microlabeling

Mark images with information about origin

Easily find corresponding code

Use image annotations by the OCI

Deprecated: https://label-schema.org

Example...

Use microlabeling

Dockerfile

```
FROM ubuntu:xenial-20180123

LABEL \
org.opencontainers.image.created=“2018-01-31T20:00:00Z+01:00“ \
org.opencontainers.image.authors=“nicholas@dille.name“ \
org.opencontainers.image.source=“https://github.com/nicholasdille/docker“ \
org.opencontainers.image.revision=“566a5e0“ \
org.opencontainers.image.vendor=“Nicholas Dille“

#...
```

Tipps and Tricks

Pull during build

Prevent usage of outdated images

```
docker build --pull ...
```

Timezones

Make sure clocks are synced

```
docker run -v /etc/localtime:/etc/localtime ...
```

Derive dynamically

Specify default tag/version

```
ARG VERSION=xenial-20180123
FROM ubuntu:${VERSION}
```

Myths and (Anti)Patterns

Running Containers

Pitfall of using latest

Old image

New containers are started based on existing images

Images must be updated explicitly (spoiler: `docker run --pull …`)

Difficult to run different versions on the same host

Two services using the same image

One service is updated and pull the new latest

Other service will continue to run on old image (based on image ID)

Re-create of other service will use new latest

Cleaning up automatically

Handling containers required testing

Run containers to test something

Run tools distributed in containers

Many exited containers remain behind

Temporary containers can be removed automatically

docker run --rm ...

Housekeeping

Cleanup before build

Create sane environment to work with

Cleanup after build

Save space

Commands

```
docker images -q | xargs -r docker rmi -f
```

Custom Formats

```
docker ps --format "table {{.Names}}\\t{{.Image}}\\t{{.Status}}"

cat ~/.docker/config.json

{

#...

"psFormat":"table {{.Names}}\\t{{.Image}}\\t{{.Status}}"

#...

}
```

File Permissions on Volumes

Problem statement

Creating files on volumes get owner from container

```
$ docker run -v $PWD:/source ubuntu bash
root@12345678# touch newfile
root@12345678# exit
$ ls -l
total 648
-rw-r--r--. 1 root root 0 Oct 12  2017 newfile
```

Solution

```
$ docker run --user $(id -u):$(id -g) -v ...
$ docker exec --user $(id -u):$(id -g) -v ...
```

Security

Client / Server Authentication

Server authentication

Create certificate authority

Create server certificate

Client can check authenticity based on CA

Client authentication

Create client certificate based on same CA

Server can check authenticity based on CA

Offical documentation

https://docs.docker.com/engine/security/https/

Privileged Containers

Easy way out

```
$ docker run -d --privileged ubuntu
```

Privileged containers cause privilege escalation

Capabilities

Often only capabilities are required

```
$ docker run -d --cap-add SYS_ADMIN ubuntu
```

Dropping Capabilities

```
$ docker run -it --rm --privileged ubuntu:xenial bash -c 'getpcaps $$'
```

```
Capabilities for `1': =
cap_chown,cap_dac_override,cap_dac_read_search,cap_fowner,cap_fsetid,cap_ki
ll,cap_setgid,cap_setuid,cap_setpcap,cap_linux_immutable,cap_net_bind_servi
ce,cap_net_broadcast,cap_net_admin,cap_net_raw,cap_ipc_lock,cap_ipc_owner,c
ap_sys_module,cap_sys_rawio,cap_sys_chroot,cap_sys_ptrace,cap_sys_pacct,cap
_sys_admin,cap_sys_boot,cap_sys_nice,cap_sys_resource,cap_sys_time,cap_sys_
tty_config,cap_mknod,cap_lease,cap_audit_write,cap_audit_control,cap_setfca
p,cap_mac_override,cap_mac_admin,cap_syslog,cap_wake_alarm,cap_block_suspen
d+eip
```

Solution

```
$ docker run -it --rm --privileged ubuntu:xenial bash -c 'capsh --inh="" --
drop="all" -- -c "getpcaps $$"'
```

```
Capabilities for `1': =
```

User Namespace Remapping

How it works

User IDs in containers are mapped to dedicated range on the host

Example: Container UID 0 is mapped to host ID 12345

Advantages

Containers cannot modify files as root

```
$ docker run -it --rm -v /etc:/hostetc ubuntu bash
```

Disadvantages

Files written to local volumes are owned by „strange“ IDs

Official documentation

https://docs.docker.com/engine/security/userns-remap/

Credential Disclosure (Layers)

Layers are treacherous

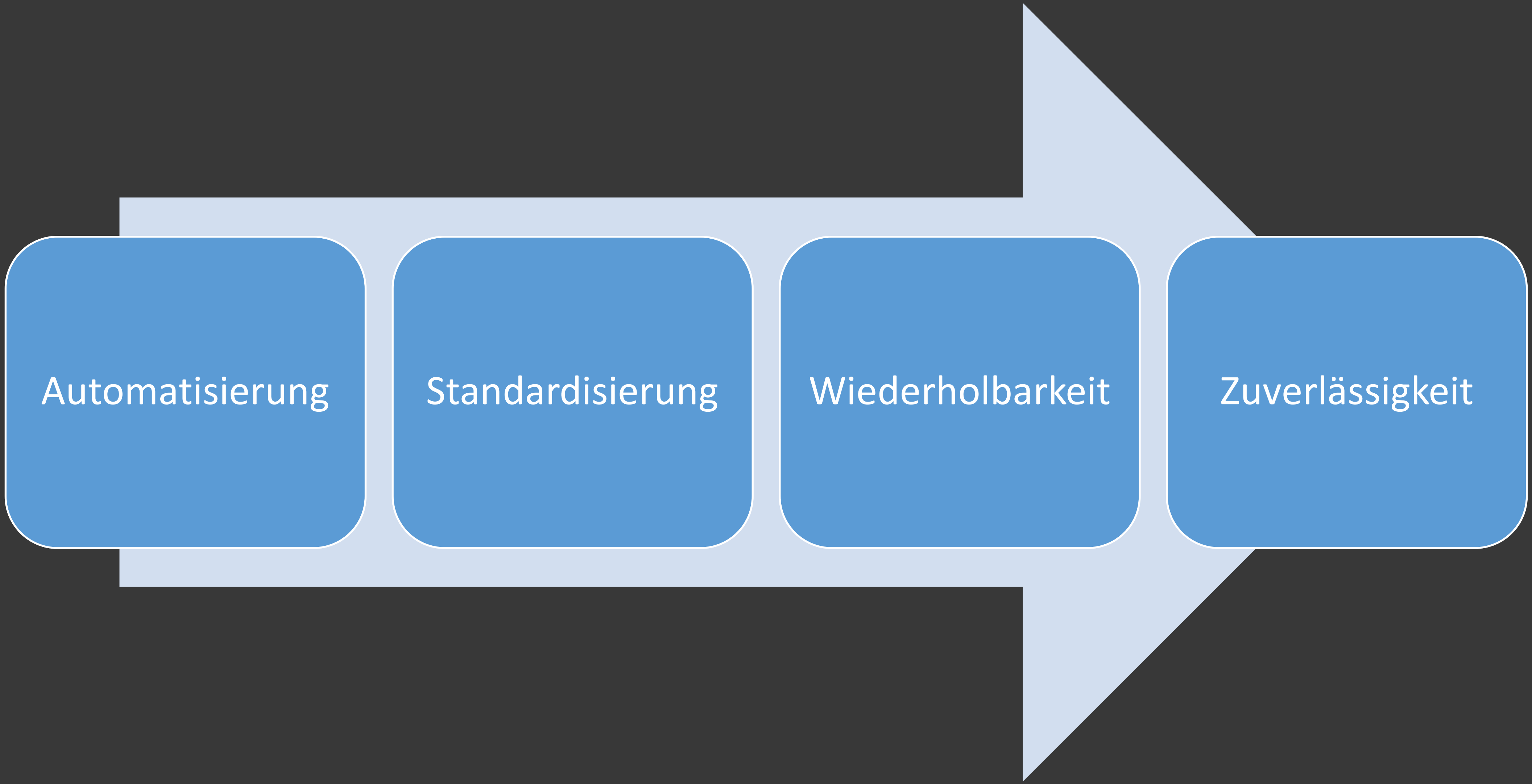
Files removed in higher layer are „invisible“

But they are still present

Example Dockerfile

```
FROM ubuntu
ADD id_rsa /root/.ssh/
RUN scp user@somewhere:/tmp/data .
RUN rm /root/.ssh/id_rsa
```

Continuous Integration


Automatisierung
Standardisierung
Wiederholbarkeit
Zuverlässigkeit

Continuous…

…Integration

Automated build

Automated tests on every commit

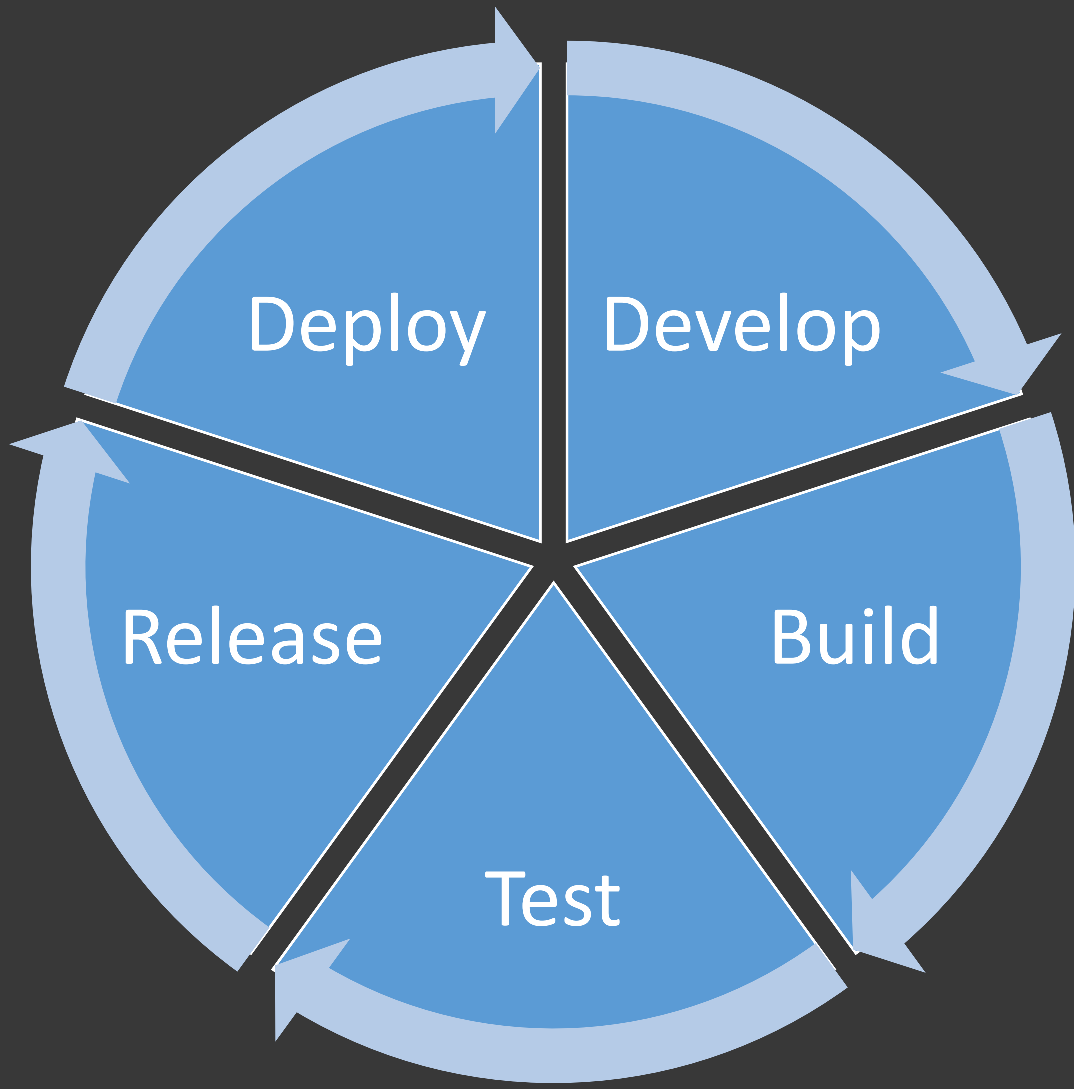
…Deployment

Automated installation but…

…manual approval for production

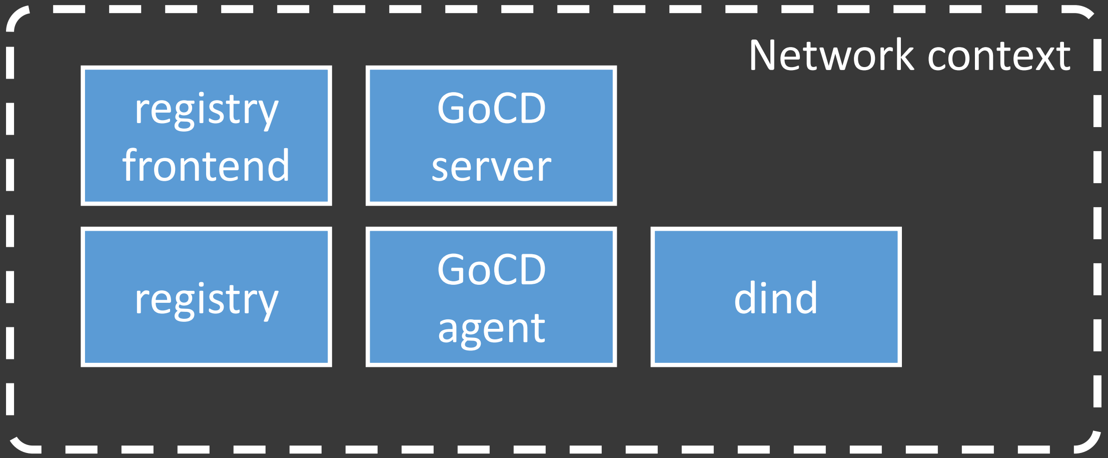
…Delivery

Automated release of the artifact



Prepare Hands-On Environment

What you get



Hands-On

```
$ git clone https://github.com/nicholasdille/docker-ci-cd-demo
$ cd docker-ci-cd-demo
$ docker-compose build
$ docker-compose up -d
```

Automating Image Build

Build on change / commit

Add pipeline-as-code to GoCD server

```
<config-repos>
    <config-repo pluginId="yaml.config.plugin" id="pipelines-yaml">
      <git url="http://gitea:3000/superuser/docker-ci-cd-examples.git" />
    </config-repo>
  </config-repos>
```

Separated Functionality

- One process per container?
- Minimize functionality
- Consider containers as microservices
- Dependencies
- Upstream / downstream images

Build Cache

How it works

Layers have predecessors

Layers are created by commands

Build is like a directed graph starting with upstream image

If commands are unchanged, the outcome already exists

Pre-populating

```
docker pull myimage:5
docker build --tag myimage:6 .
```

Learn from Software Development

Short feedback loops

Run builds on every commit

Build must include tests

Build artifacts should be packaged/deployed

Fail early

Do not ignore failed tests

Assume responsibility

Task is done when build and tests are successful

Continuous Delivery

Pipelines

Build

Create artifacts

- Applications, libraries, images etc.

Deployment

Rollout artifacts to a QA or live environment

Pipelines

…have dependencies

…are triggered by upstream pipelines

Handling Parameters

Pipeline specific parameters

Configure behaviour of pipeline

Environment specific parameters

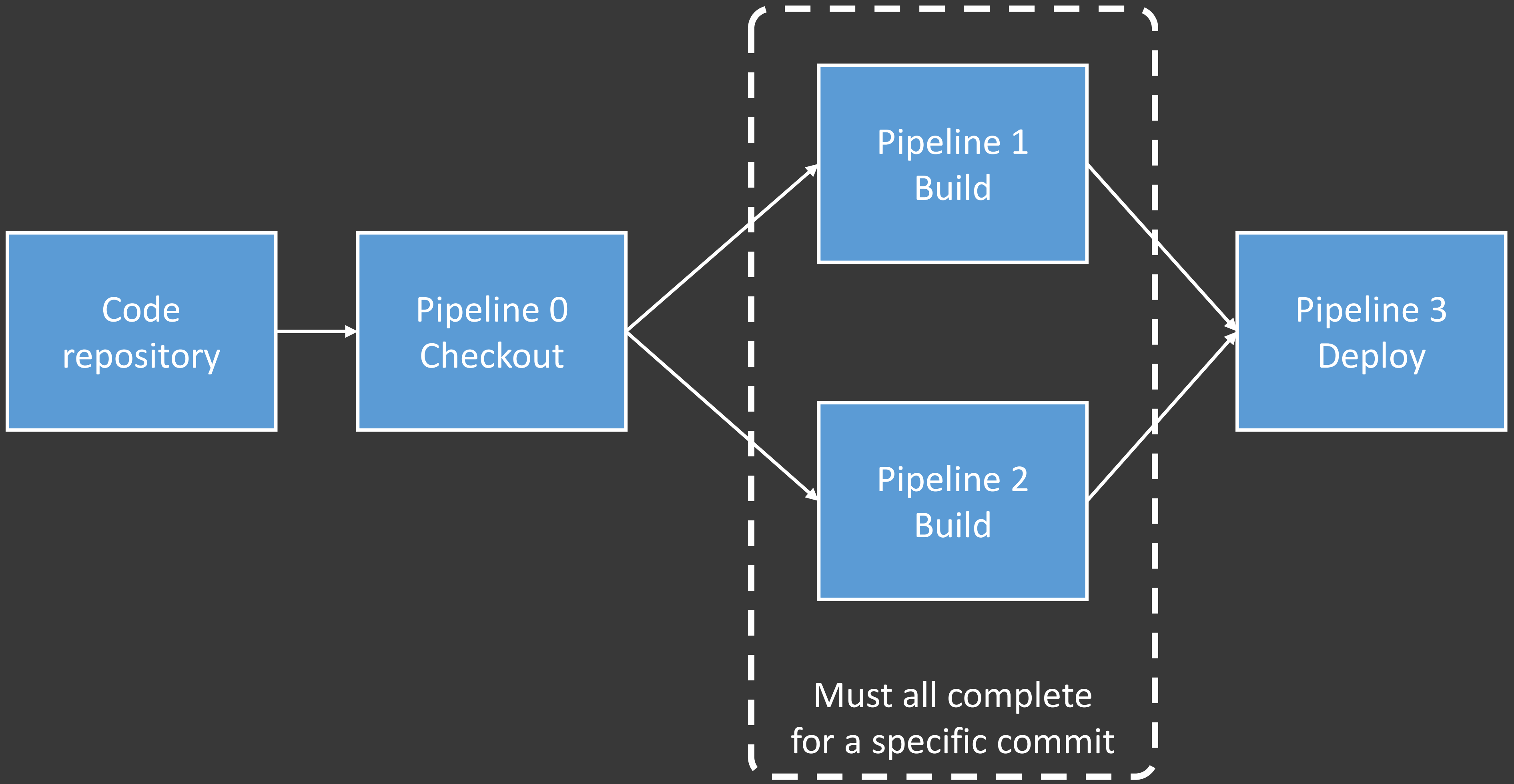
Configure access to target environment

Secrets

Store credentials

Can be external to build tool (e.g. HashiCorp Vault)

Fan Out / Fan In



Deployment Strategies

Immutable infrastructure

Do not update running instances

Blue/green deployment

Create new deployment next to existing one

Test new deployment

Switch to new deployment

Rollback often possible

Downtime

Without downtime is very hard

Usually broken by databases

Running in Production

Orchestrators

Purpose

Manage multiple container hosts

Schedule services across all hosts

Well-known orchestrators

Kubernetes

Docker Swarm

Rancher Cattle (1.6)

Load Balancing

Do not publish ports

Only one container can use one port

Hosts will quickly run out of ports

Services are expected to run on well-known ports (443 for HTTPS)

Deploy containers to handle LB

Forwards traffic to target containers

Use Server Name Indication (SNI) for multiple HTTPS services on the same port (443)

Persistent Storage

Docker volumes are locally persistent

Data only lived on a specific host

Volume drivers integrate with storage system

Volumes are mounted from storage system

Volumes are available on all hosts

Resource Management

Reservations

Containers can have assigned minimum resources

```
$ docker run --memory-reservation 1024m …
```

Limitations

Upper limits apply only if resources are available

```
$ docker run --memory 4096m …
```

Alpine versus Distribution Image

Minimal image size

Ubuntu: 150MB

Alpine: 5MB

Advantages

Faster pull, fast deployments

Disadvantages

Alpine links against libmusl

Well-known tools have less/different parameters

Containerizing System Services

Requirements

Few to no prerequisites for Docker hosts

Automatically created using orchestrator or `docker-machine`

Containerized services

Can be deployed from orchestrator

Health is monitored by orchestrator

Example: ntpd

Third Party Dependencies

Dependency hell

Just like software

Developers rely on existing libraries

Those libraries can and will have security issues

Security scans

Scan for known issues in packaged dependencies

Sometimes hard to remediate

Change Management

ITIL

Company often work by ITIL

Change management is used to document changes to live sytems

Changes usually need to be schedules in advance

Automated deployments

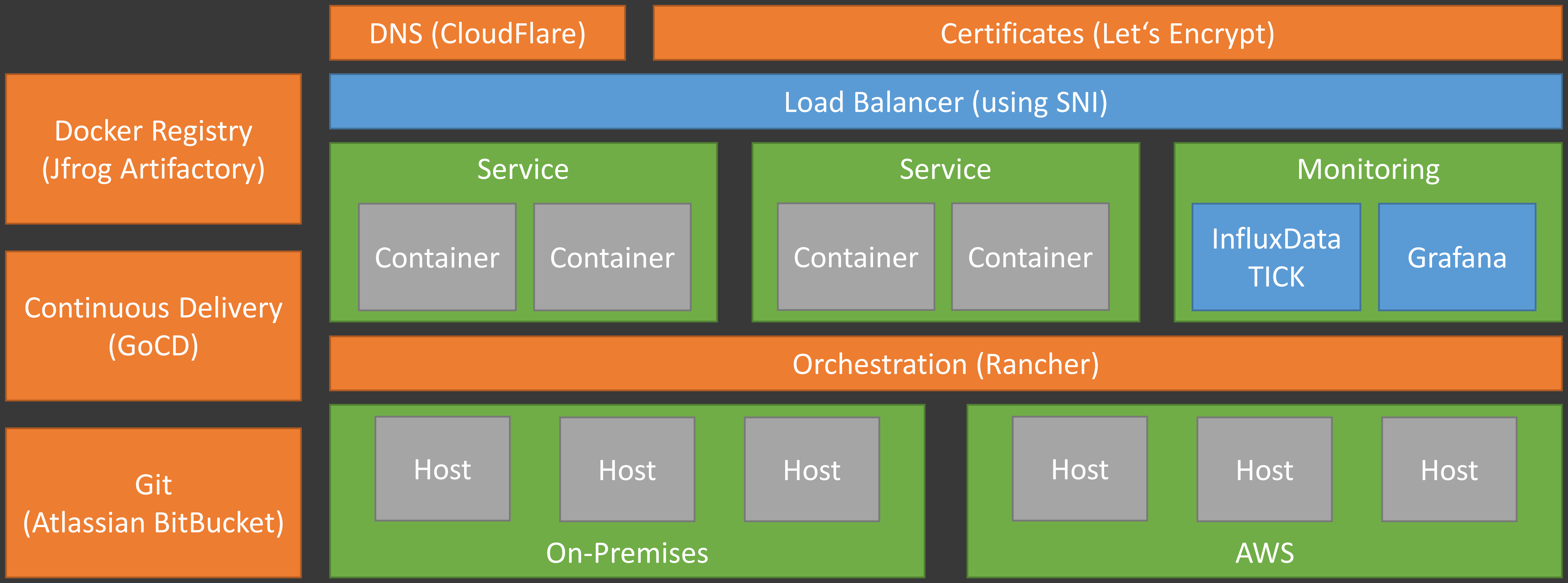
...can cause rollouts without planning a change

...require trust between change manager and DevOps teams

...must be automatically appoved

...must limit change to agreed upon maintenance windows

Example Stack



Monitoring

Hands-On

Basic monitoring out of the box

Containers can be monitored by a single agent on the host

Hands-On

Look at grafana

Show usage of mapped named pipe

Ecosystem

Offerings by Docker

Enterprise Edition

- Universal Control Plane
- Trusted Registry
- Docker Certified

Community Edition

- Docker CLI
- dockerd
- Docker Hub
- Machine

The Moby Projekt

Announced at DockerCon (May 2017)

Released components of Docker CE

Projects

runC / containerd / LinuxKit / SwarmKit

Notary / Registry / Compose

libnetwork / DataKit / BuildKit / VPNKit / InfraKit / HyperKit

Downstream projects

Docker CE / EE

Balena (Container Engine für IoT)

Governed by Technical Steering Committee

Offerings by Docker

Enterprise Edition

- Universal Control Plane
- Trusted Registry
- Docker Certified

Community Edition

- Docker CLI
- dockerd
- Docker Hub
- Machine

Moby

- SwarmKit
- containerd
- runC
- LinuxKit
- Compose
- Registry
- notary
- libnetwork
- DataKit
- BuildKit
- VPNKit
- InfraKit
- HyperKit

LinuxKit

Minimal, immutable Linux

Definition in YAML

Build with Moby

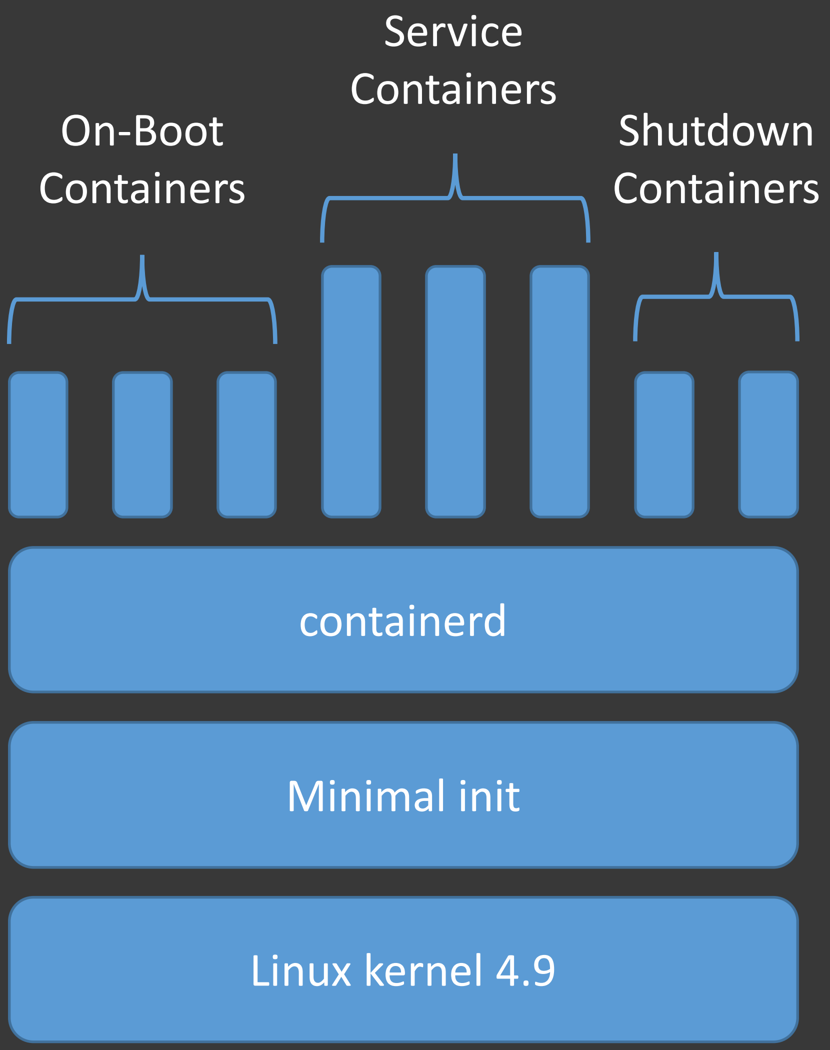
Based on containerd

Everything is a container (including system daemons)

Images are based on Alpine Linux

Images are signed using Notary

Supports multiple platforms



containerd

Manages containers

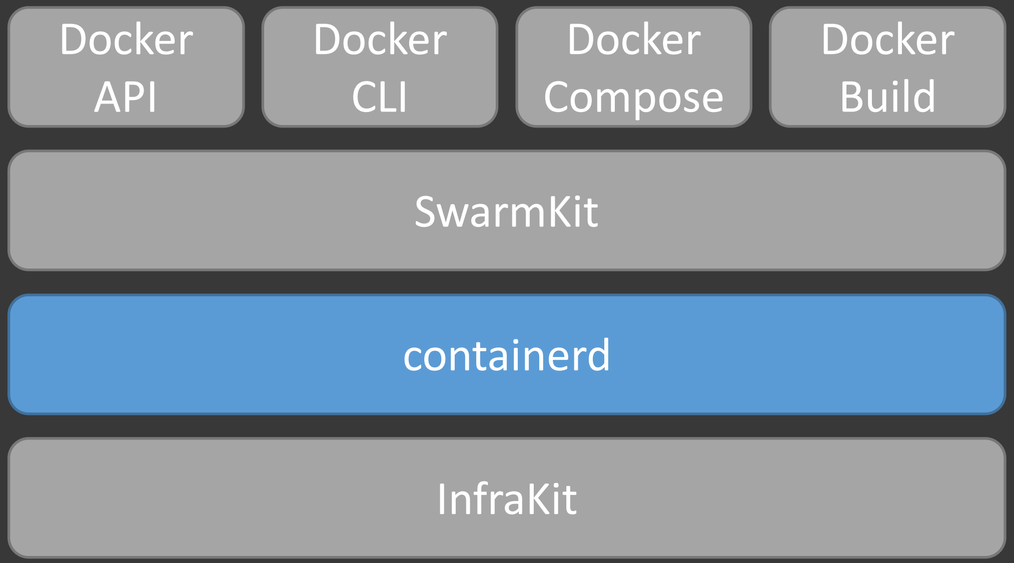
Based on runC (container runtime)

Adds pull and push of images

gRPC API using UNIX socket

CLI `ctr`

Owned by CNCF



Notary

Ensures data integrity

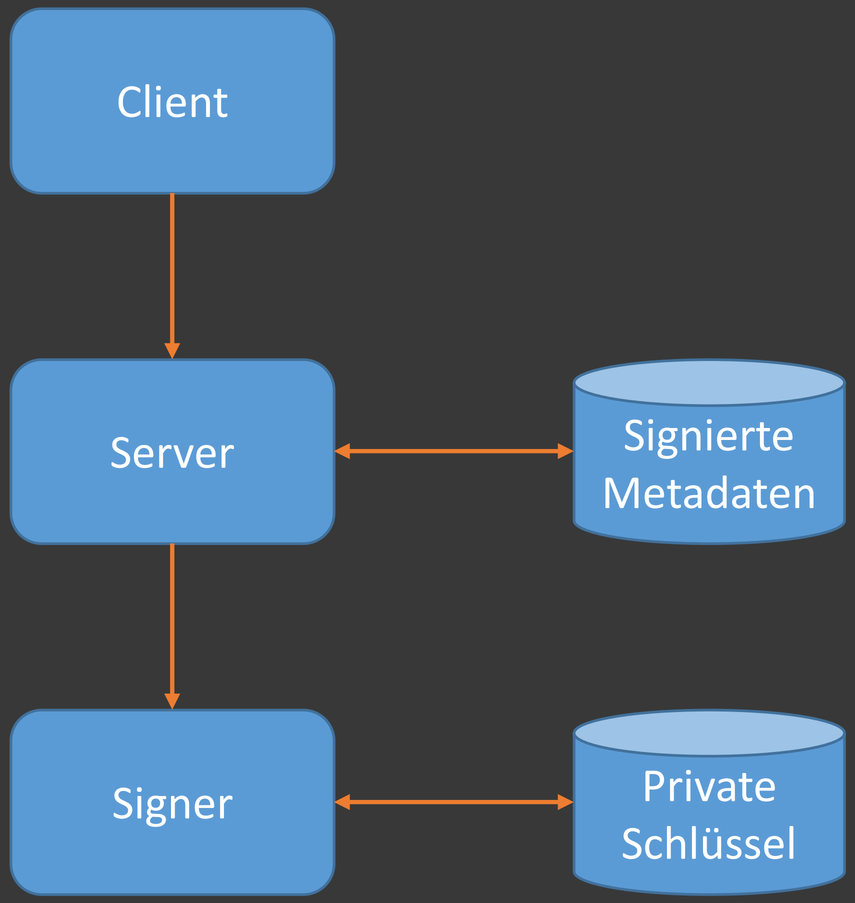
Can sign arbitrary data

Verification of signatures

Client/server architecture

Upstream project for Docker Content Trust

Owned by CNCF



Docker and the ecosystem

…initiated (April 2017)

moby project

12 Projekte

…donated containerd (März 2017) and Notary (Oktober 2017)

THE LINUX FOUNDATION

CLOUD NATIVE COMPUTING FOUNDATION

…donated specification and runC (Juni 2015)

OPEN CONTAINER INITIATIVE

Docker ❤ Kubernetes

Announced at DockerCon EU (October 2017)

Integrated in Docker Universal Control Plane (UCP)

Manages stacks based on SwarmKit and Kubernetes

Translates docker-compose.yml

Bundled Docker CLI and Kubernetes CLI

Installed on new nodes via Docker UCP

Included in Docker für Mac

Included in Docker für Windows

Docker for Mac/Windows installs Kubernetes (single node)

Offerings by Docker

Enterprise Edition

- Universal Control Plane
- Trusted Registry
- Docker Certified

certified
kubernetes
1.8

Community Edition

- Docker CLI
- dockerd
- Docker Hub
- Machine

Moby

- SwarmKit
- containerd
- runC
- LinuxKit
- Compose
- Registry
- notary
- libnetwork
- DataKit
- BuildKit
- VPNKit
- InfraKit
- HyperKit

Kubernetes