Containers 101

Tal Neeman
Developer Advocate, IBM

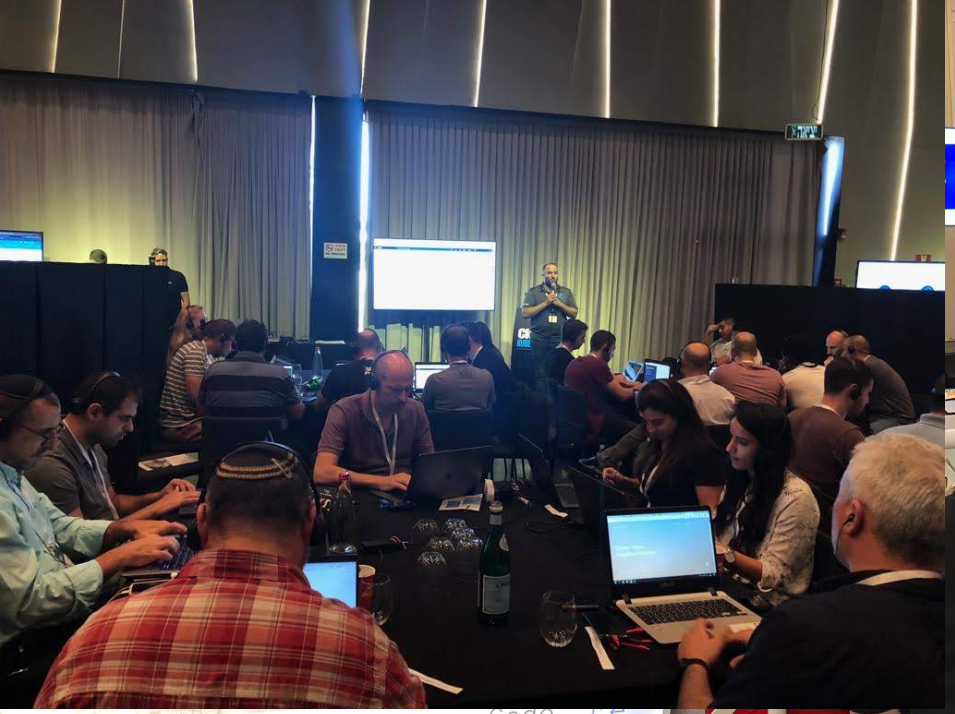
IBM **Developer**

IBM

Hi, I’m Tal

I’m a Developer Advocate at IBM.

I lecture at meetups, participate as mentor at hackathons, having fun in awesome webinars and I also write articles / tutorials about open source or new technologies on IBM Cloud.

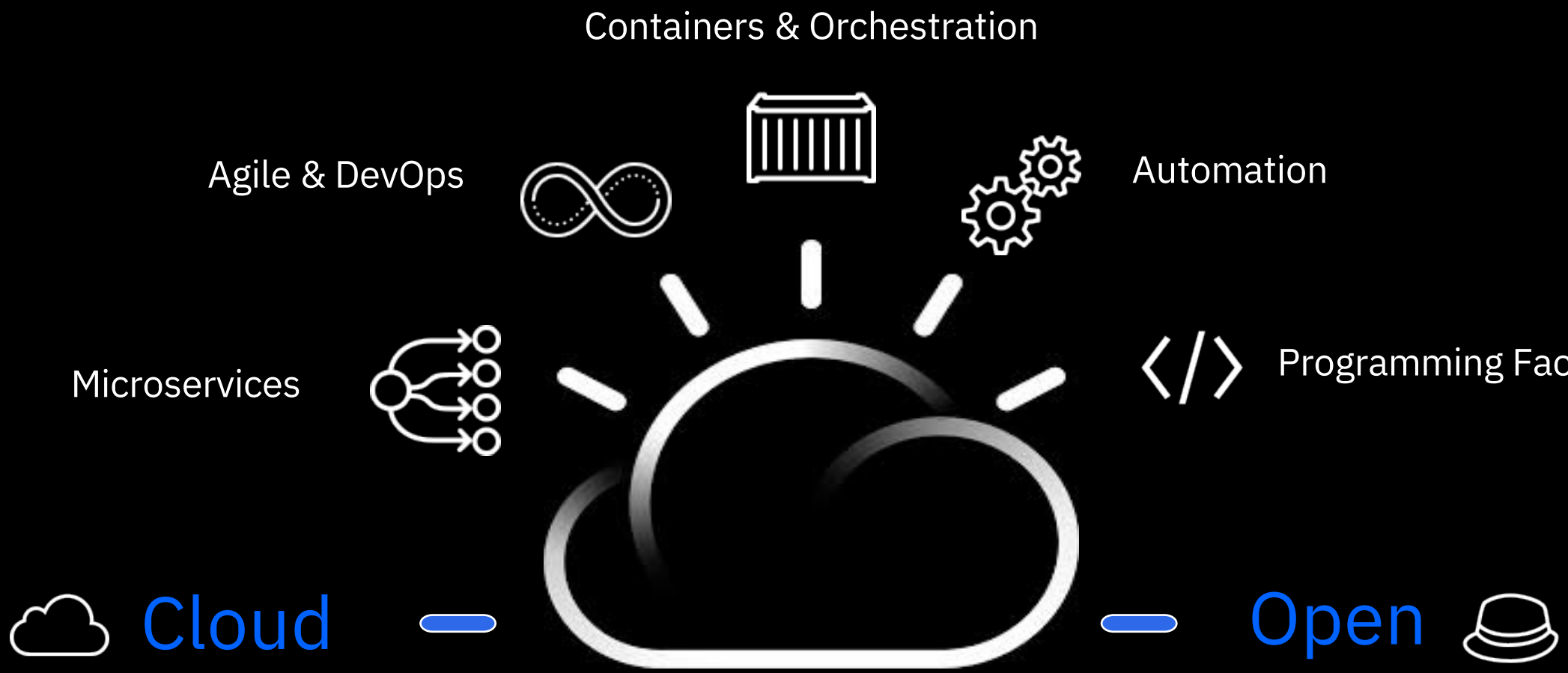


developer.ibm.com

Cloud-native development is an approach to building and running applications that exploits the advantages of the cloud computing delivery model. The Cloud Native Computing Foundation (CNFC) talks about using "… an open source software stack to deploy applications as microservices, packaging each part into its own container, and dynamically orchestrating those containers to optimize resource utilization. Cloud native technologies enable software developers to build great products faster." (CNCF, 2019)

CLOUD NATIVE
COMPUTING FOUNDATION

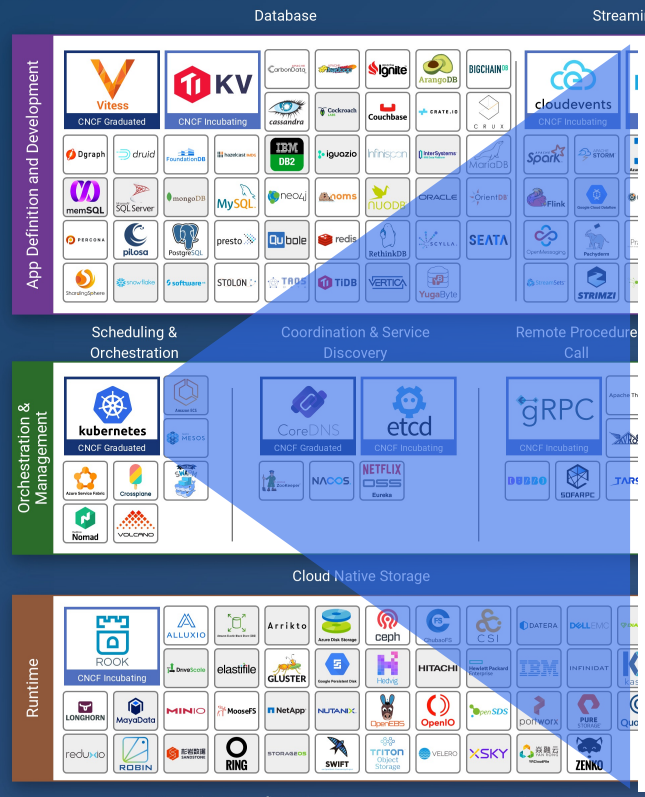
Cloud Native Development



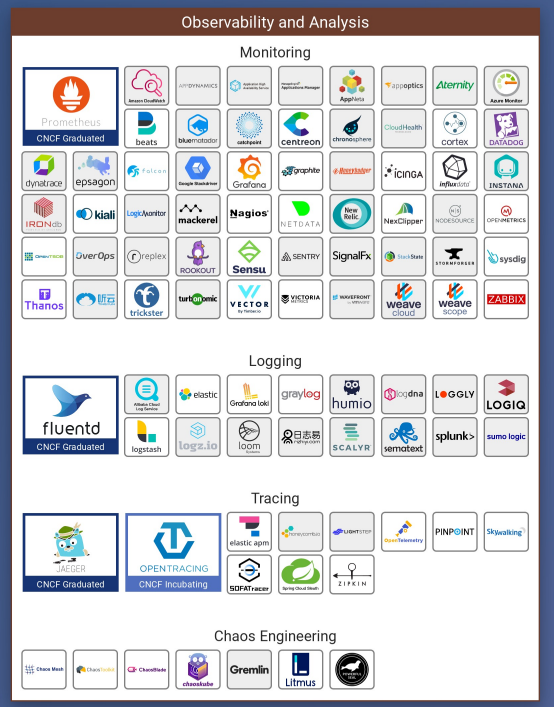
CNCF Cloud Native Landscape
2020-05-05T22:49:53Z 8a2f76a

Overwhelmed? Please see the CNCF Trail Map. That and the interactive landscape are at l.cncf.io

Greyed logos are not open source



kubernetes



CLOUD NATIVE Landscape
CLOUD NATIVE COMPUTING FOUNDATION
Redpoint
Amplify

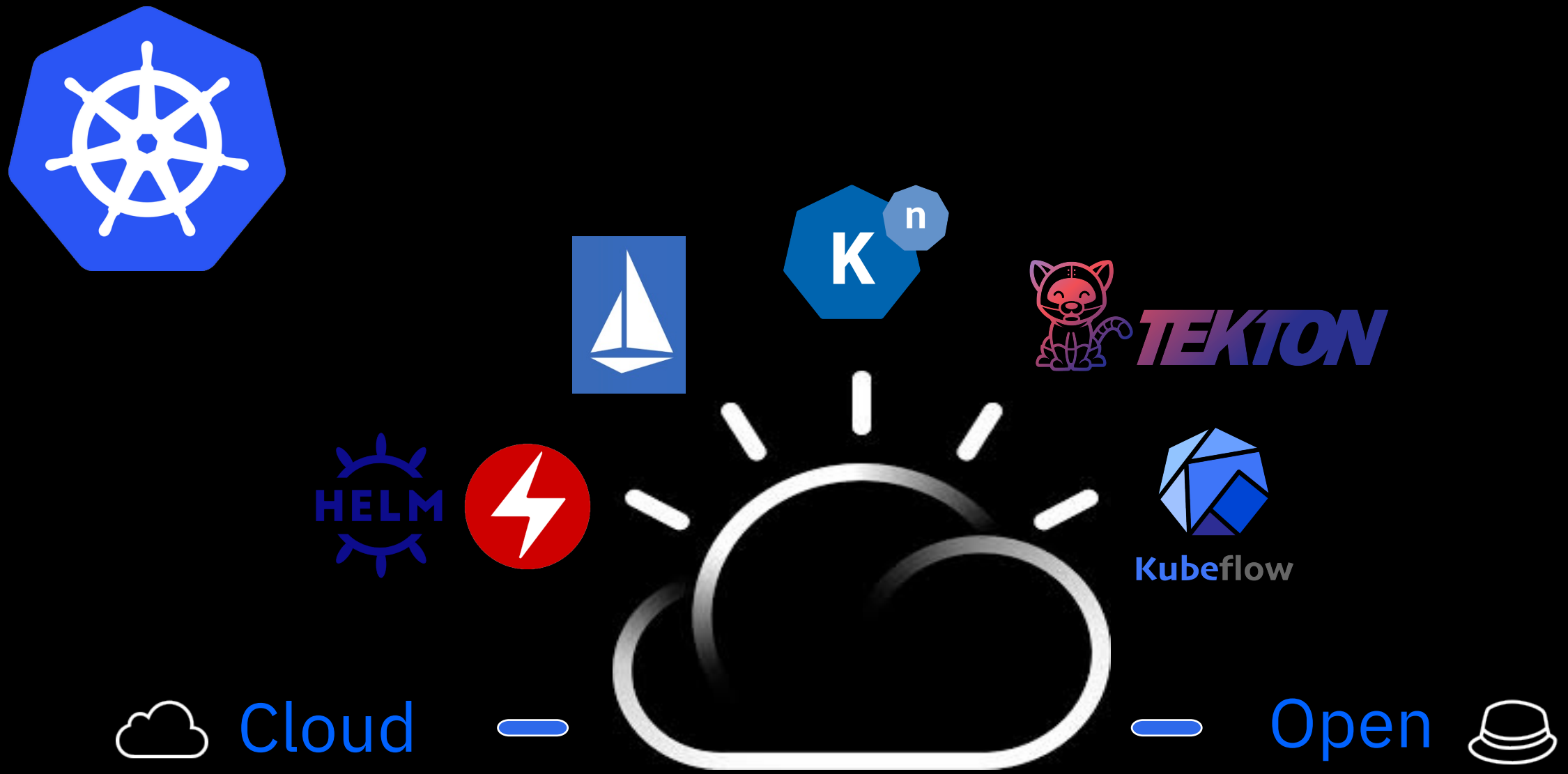
l.cncf.io

This landscape is intended as a map through the previously uncharted terrain of cloud native technologies. There are many routes to deploying a cloud native application, with CNCF Projects representing a particularly well-traveled path.

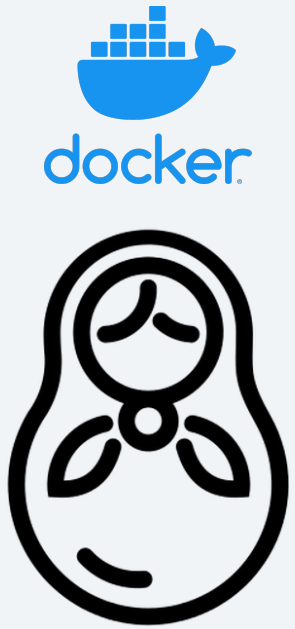
Kubernetes Certified Service Provider

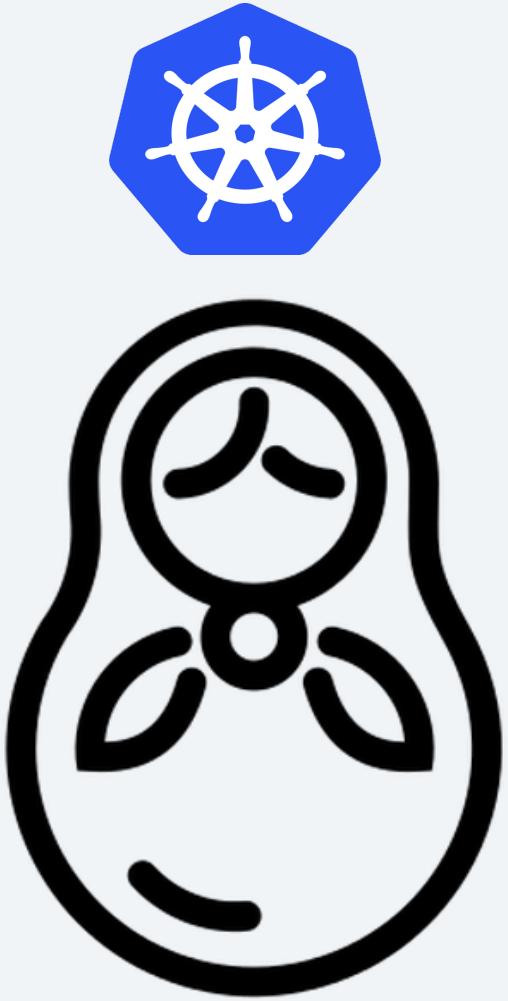
Special

Kubernetes Training Partner


HELM
TEKTON
Kubeflow
Cloud
Open

^^ Functionality





>> Enterprise ready

Containers

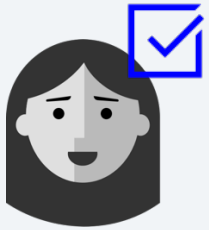
What are Containers?

OS level virtualization

- Isolate user space instances on a single host machine
- Gives isolated view of processes, user space and file system for user owner
- Share host Linux kernel
- A self-contained sealed unit of software
- Contains everything required to run the code

A Container Includes :
Code, configs, Processes, Networking, Dependencies, OS (just enough)

Why do containers arise?



Deb
The Developer

App.war

Guide

Deploy

App.war

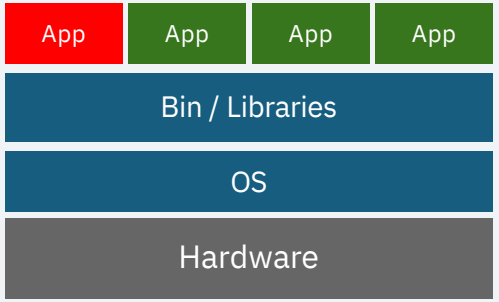
Guide

Mike
The Ops Guy

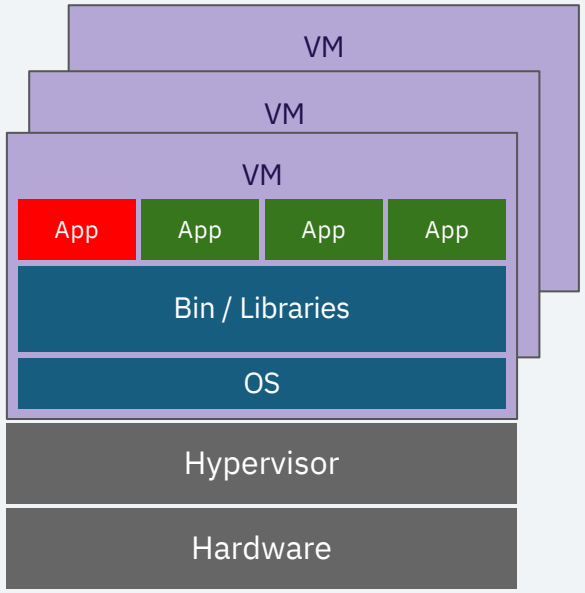
Compatibility, dependency, versions...

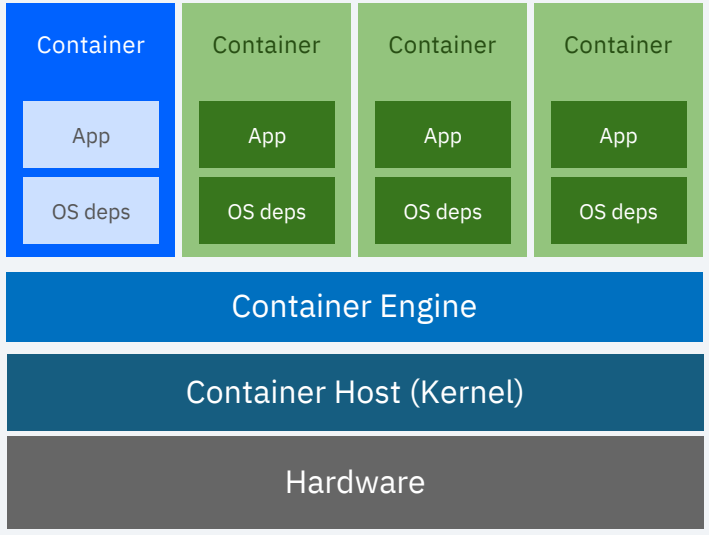
Long time setup

Different dev/test/prod environments



VMs vs Containers



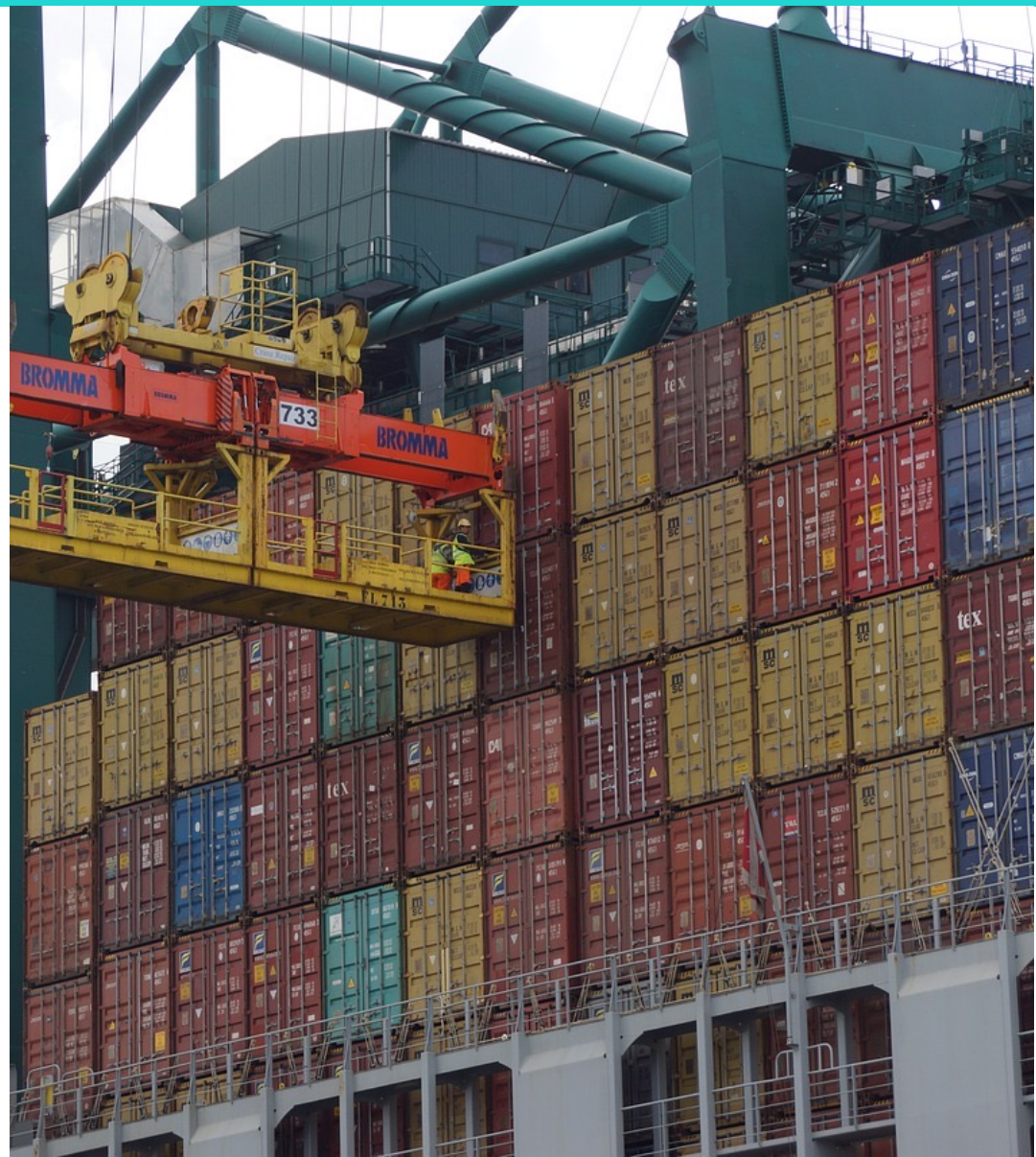


- VMs are a better choice for running apps that require all of the operating system's resources and functionality when you need to run multiple applications on servers or have a wide variety of operating systems to manage.
- Containers are a better choice when your biggest priority is maximizing the number of applications running on a minimal number of servers.

VMs	Containers
Heavyweight	Lightweight
Limited performance	Native performance
Each VM runs in its own OS	All containers share the host OS
Hardware-level virtualization	OS virtualization
Startup time in minutes	Startup time in milliseconds
Allocates required memory	Requires less memory space
Fully isolated and hence more secure	Process-level isolation, possibly less secure

Why Containers?

- Agile
- Continuous Deployment
- Dev and Ops
- Observability
- Consistency
- Management
- Microservices
- Resource Isolation
- Resource Utilization
- Portable
- Easy to manage
- Containers provide “just enough” isolation
- Immutable



What is a container?

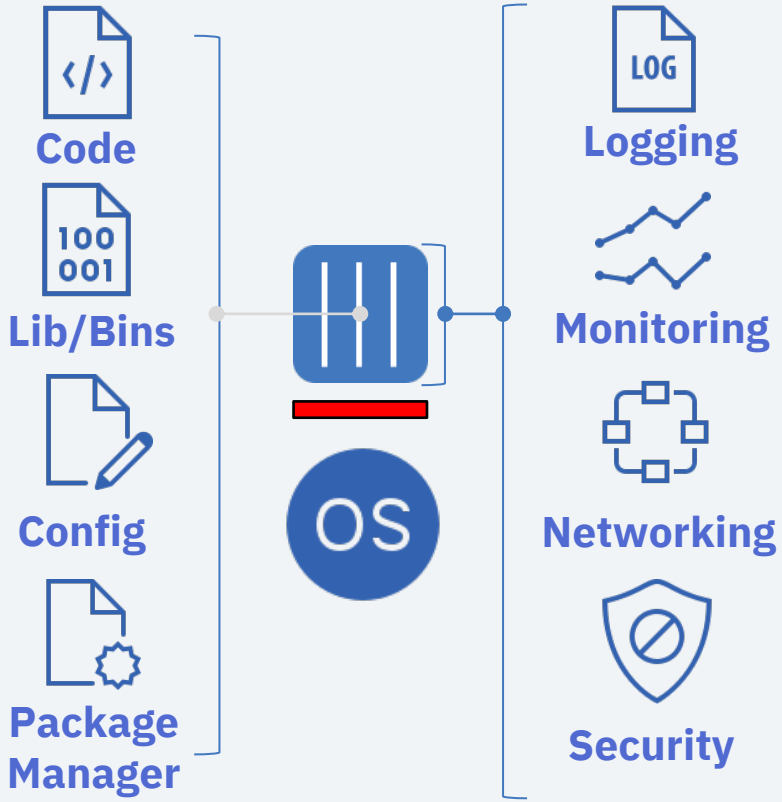
A standard way to **package** an application and all its dependencies so that it can be moved between environments and **run** without changes.

Containers work by **isolating** the differences between applications **inside** the container so that everything **outside** the container can be standardized.

Deb
The Developer

- Package apps with all dependencies
- Deploy to any environment in seconds
- Easily accessed and shared

Worries about what's « **inside** » the container



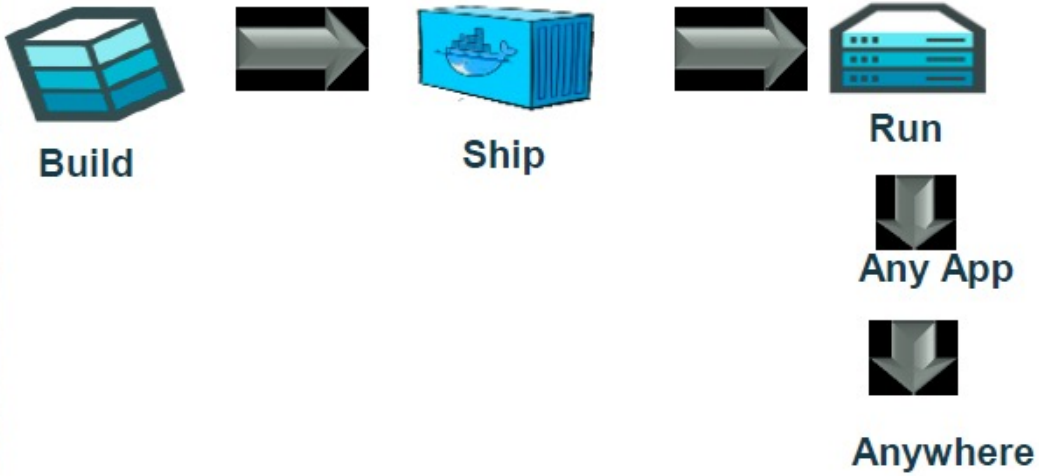
Mike
The Ops Guy

- Application processes on a shared kernel
- Simpler, lighter, and denser than VMs
- Portable across different environments

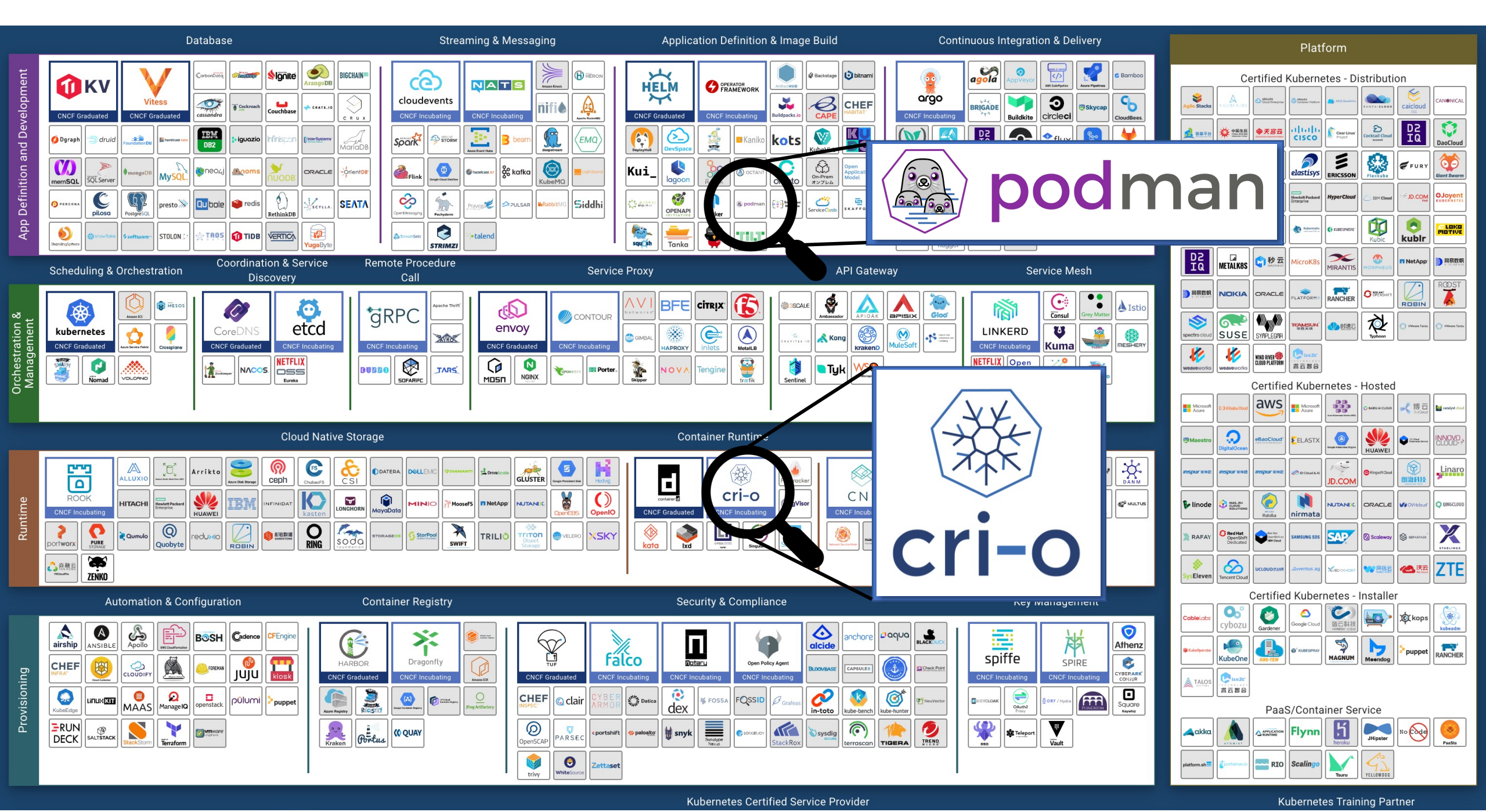
Worries about what's « **outside** » the container

What is Docker?

Docker is the world's leading containerization standard platform



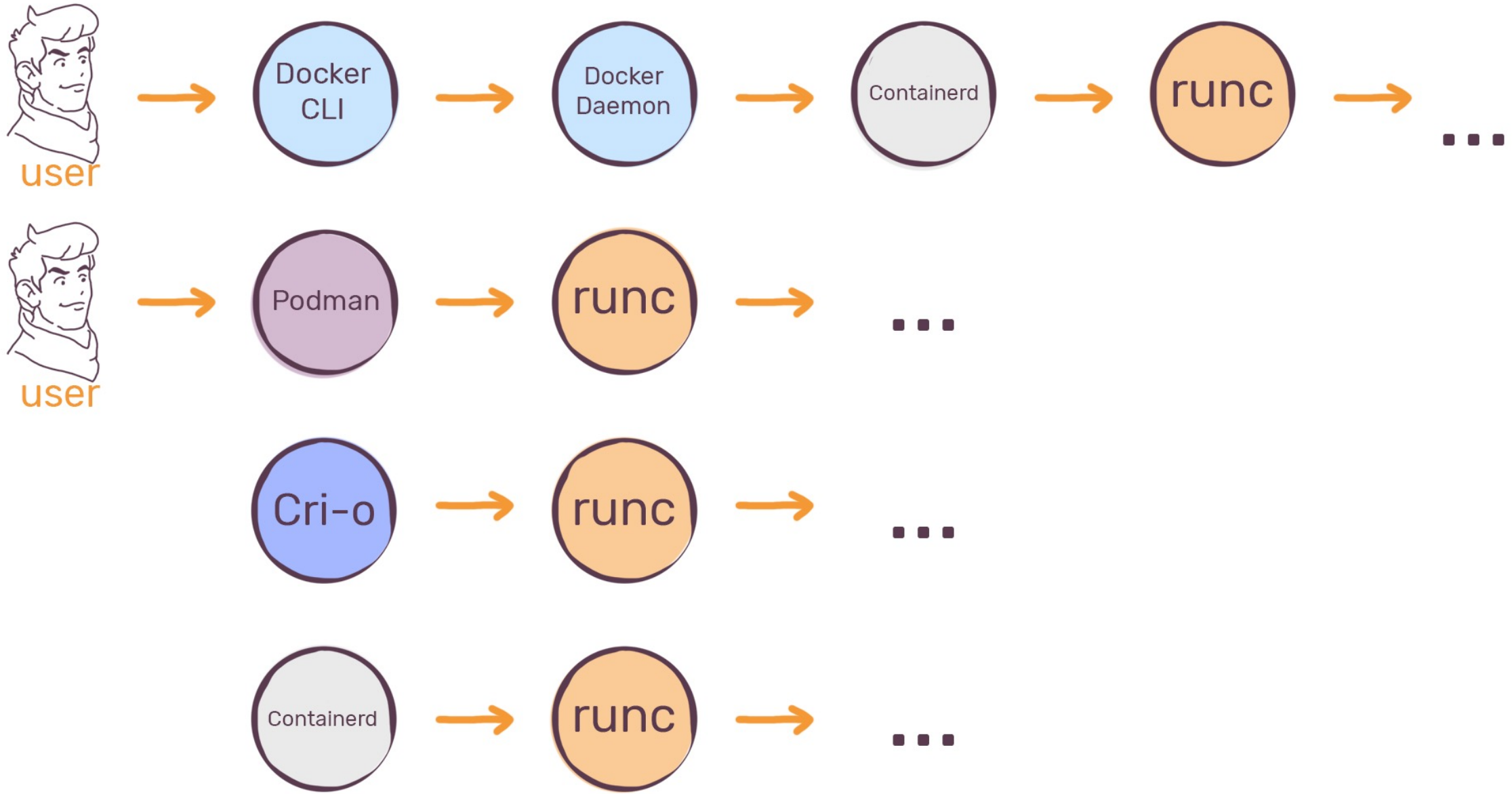
Docker is a platform for developing, shipping and running applications using **container** virtualization technology.


Database
Streaming & Messaging
Application Definition & Image Build
Continuous Integration & Delivery
App Definition and Development
Scheduling & Orchestration
Coordination & Service Discovery
Remote Procedure Call
Service Proxy
API Gateway
Service Mesh
Orchestration & Management
Cloud Native Storage
Container Runtime
Runtime
Automation & Configuration
Container Registry
Security & Compliance
Key Management
Provisioning
podman
cri-o
Platform
Certified Kubernetes - Distribution
Certified Kubernetes - Hosted
Certified Kubernetes - Installer
PaaS/Container Service
Kubernetes Certified Service Provider
Kubernetes Training Partner

Open Container Initiative

The **Open Container Initiative** is an open governance structure for the express purpose of creating open industry standards around container formats and runtimes.

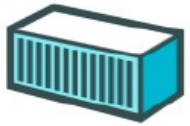
Resource: https://mkdev.me/en/posts/the-tool-that-really-runs-your-containers-deep-dive-into-runc-and-oci-specifications



Resource: https://mkdev.me/en/posts/the-tool-that-really-runs-your-containers-deep-dive-into-runc-and-oci-specifications

What is Docker?

Image
Template to use for creating containers. Snapshot of Read-Only stored in Docker Hub。



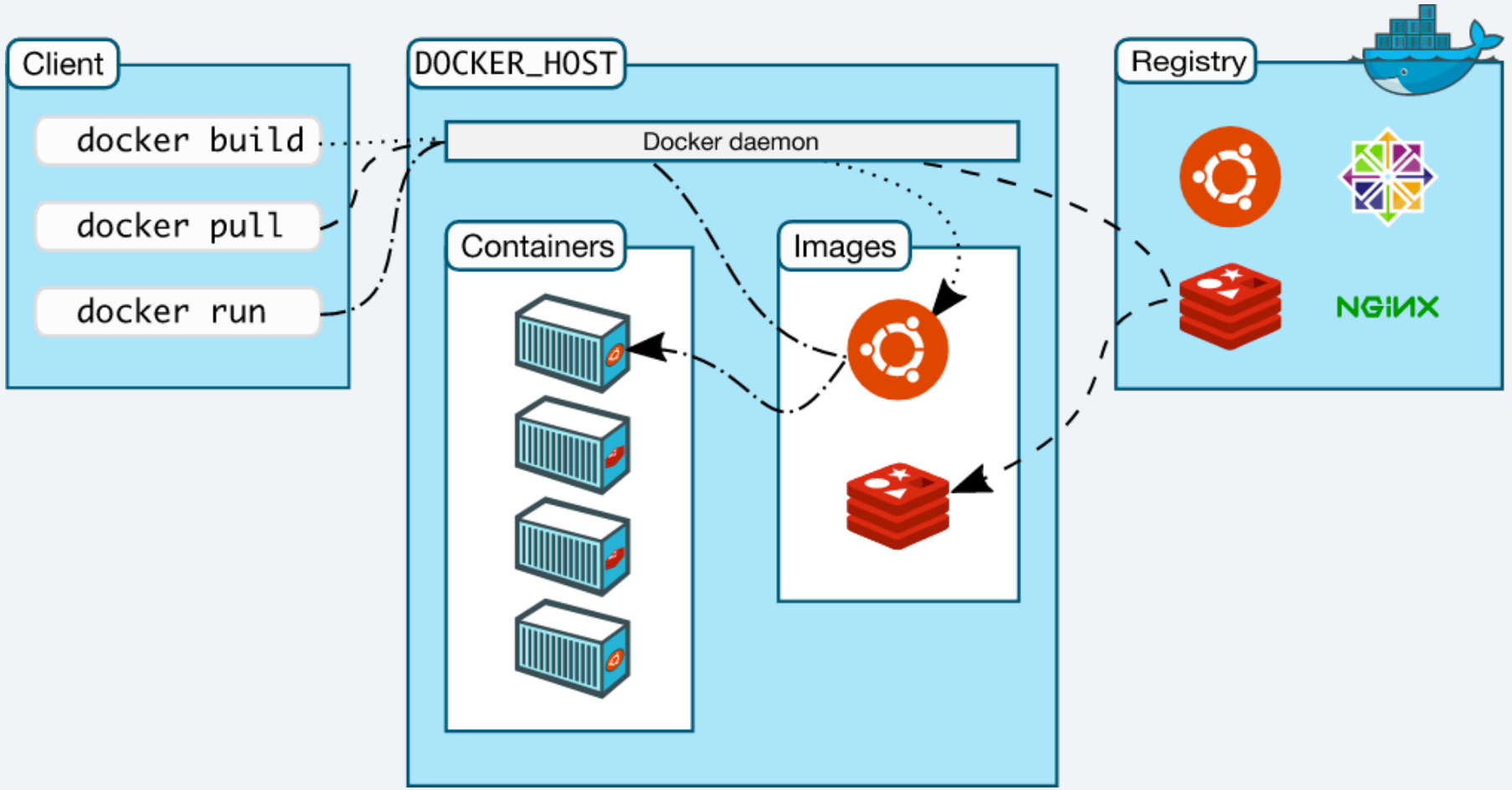
Container
Basic unit on which application service runs。

Docker Hub
Store, distribute and share images of containers.
Establish and use in SaaS or enterprise.

Docker Repository
Pulling an image from Docker Trusted Registry is the same as pulling an image from Docker Hub or any other registry. Since DTR is secure by default, you always need to authenticate before pulling images.。

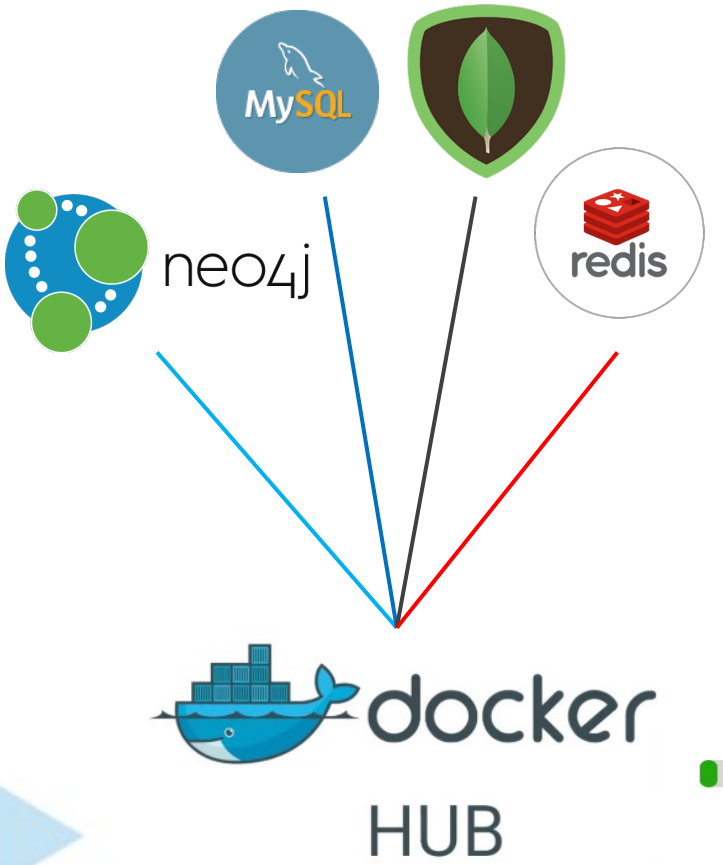
Docker Engine
A program for creating containers and running them.
Run on a physical server or virtual machine.
User can operate with command。

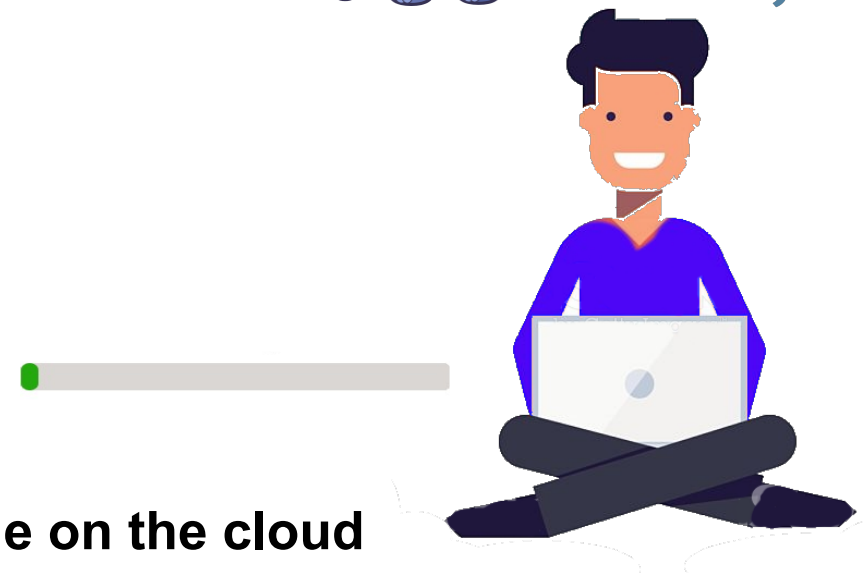
Docker architecture

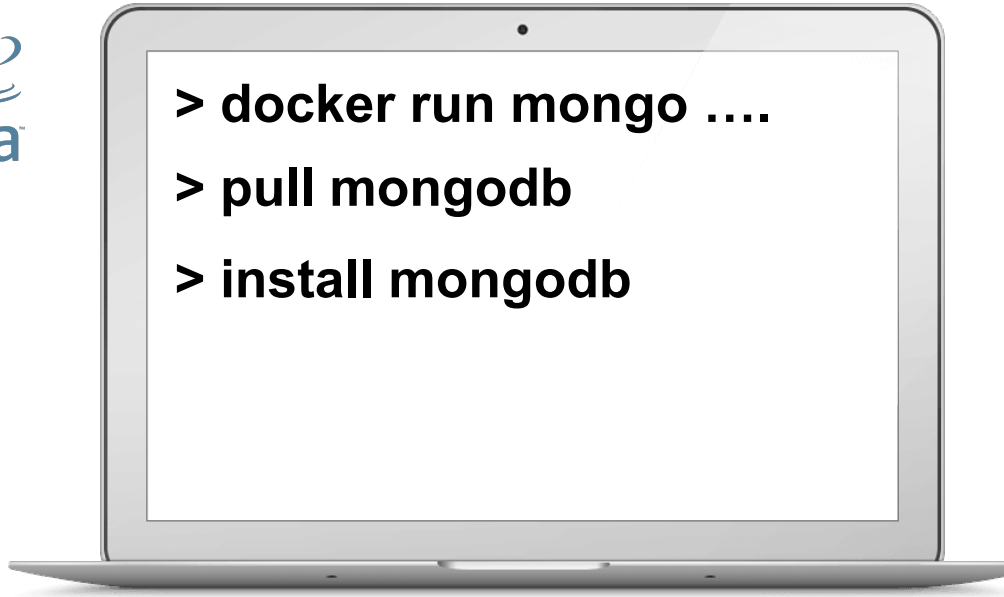


Docker :

Docker makes it really easy to install and run software without worrying about setup or dependencies

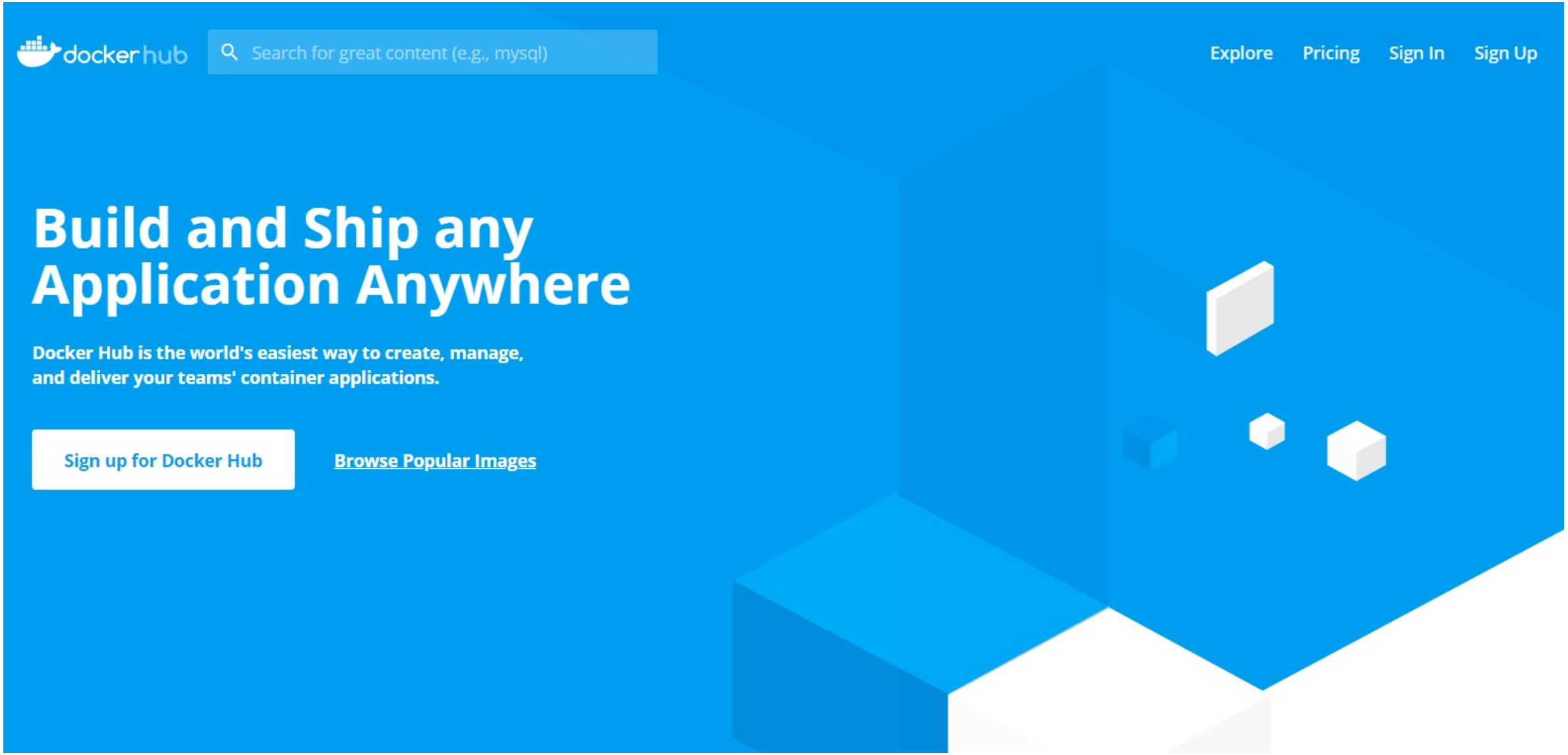




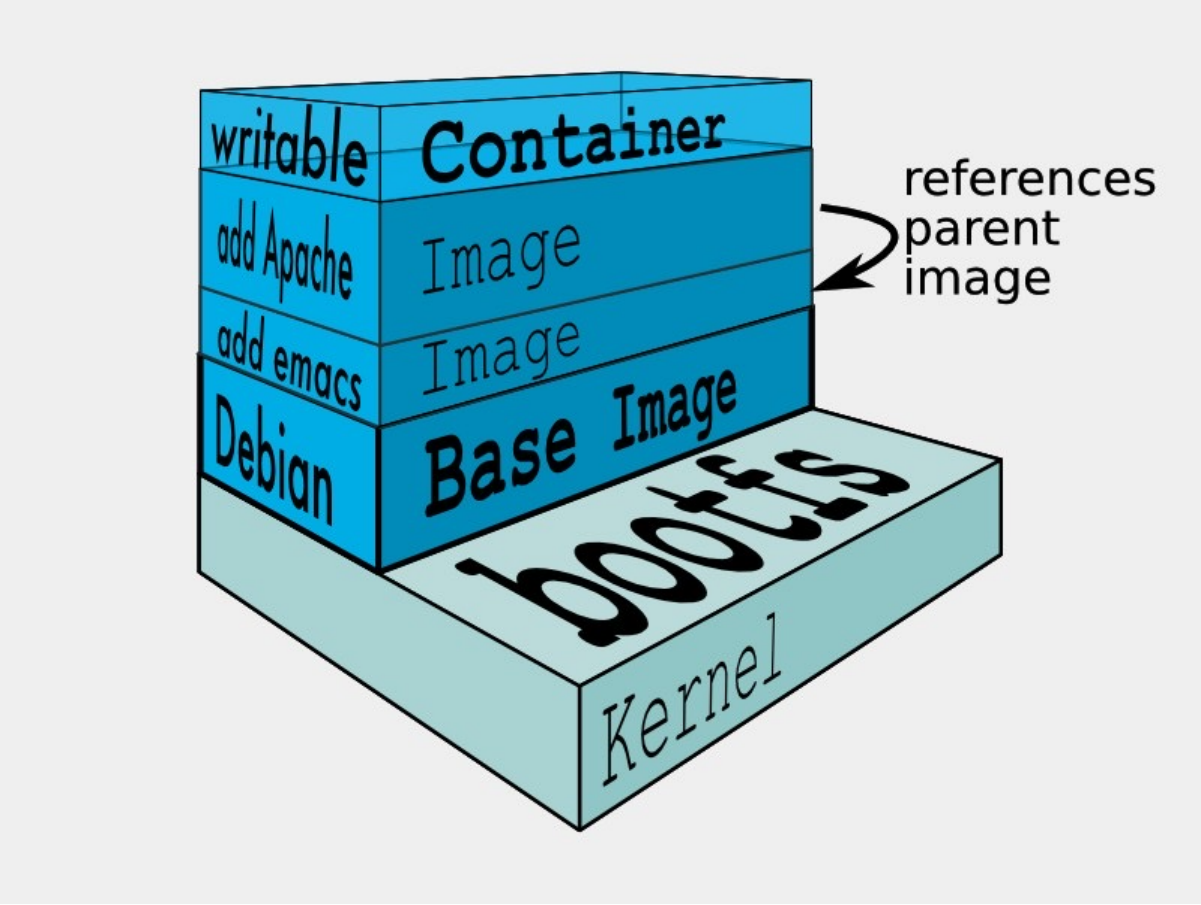


Common images available on the cloud

hub.docker.com

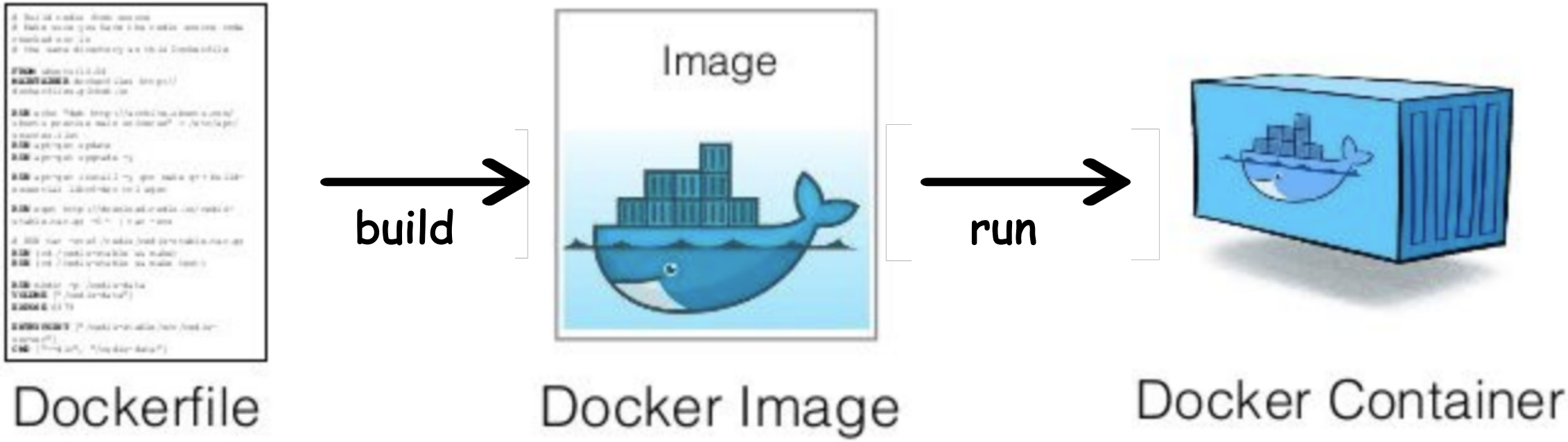


Layers of Docker container



Any RUN commands you specify in the Dockerfile creates a new layer for the container. In the end when you run your container, Docker combines these layers and runs your containers. Layering helps Docker to reduce duplication and increases the re-use.

Dockerfile



Common **Dockerfile** instructions start with RUN, ENV, FROM, MAINTAINER, ADD, and CMD, among others.

FROM - Specifies the base image that the Dockerfile will use to build a new image.

MAINTAINER - Specifies the Dockerfile Author Name and his/her email.

ARG - 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.

WORKDIR - The WORKDIR instruction sets the working directory for any RUN, CMD, ENTRYPOINT, COPY and ADD instructions that follow it in the Dockerfile.

VOLUME - creates a mount point with the specified name and marks it as holding externally mounted volumes from native host or other containers.

ENV - Sets the environment variables. For this example : JAVA_HOME

COPY - It only lets you copy in a local file or directory from your host into the Docker image itself.

ADD - lets you do that too, but it also supports 2 other sources. First, you can use a URL instead of a local file / directory. Secondly, you can extract a tar file from the source directly into the destination..

EXPOSE - This instruction exposes specified port to the host machine.

RUN - Runs any UNIX command to build the image.

RUN <commands>	***/bin/sh -c <commands>***
RUN ["executable", "params", "more params"]	***The exec form is parsed as a JSON array***

CMD - Provides the facility to run commands at the start of container. This can be overridden upon executing the docker run command.

ENTRYPOINT - allows you to configure a container that will run as an executable.

Dockerfile best practice

Incremental build time

Tip #1: Order matters for caching

```
FROM debian
COPY . /app
RUN apt-get update
RUN apt-get -y install openjdk-8-jdk ssh vim
COPY . /app
CMD ["java", "-jar", "/app/target/app.jar"]
```

Tip #2: More specific COPY to limit cache busts

```
FROM debian
RUN apt-get update
RUN apt-get -y install openjdk-8-jdk ssh vim
COPY . /app
COPY target/app.jar /app
CMD ["java", "-jar", "/app/target/app.jar"]
```

Tip #3: Identify cacheable units such as apt-get update & install

```
FROM debian
RUN apt-get update
RUN apt-get -y install openjdk-8-jdk ssh vim
RUN apt-get update \
 && apt-get -y install \
    openjdk-8-jdk ssh vim
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Reduce Image size

Tip #4: Remove unnecessary dependencies

```
FROM debian
RUN apt-get update \
 && apt-get -y install --no-install-recommends \
    openjdk-8-jdk ~~ssh vim~~
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Tip #5: Remove package manager cache

```
FROM debian
RUN apt-get update \
 && apt-get -y install --no-install-recommends \
    openjdk-8-jdk \
 && rm -rf /var/lib/apt/lists/*
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Maintainability

Tip #6: Use official images when possible

```
FROM debian
RUN apt-get update \
 && apt-get -y install --no-install-recommends \
    openjdk-8-jdk\
 && rm -rf /var/lib/apt/lists/*
FROM openjdk
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Tip #7: Use more specific tags

```
FROM openjdk:latest
FROM openjdk:8
COPY target/app.jar /app
CMD ["java", "-jar", "/app/app.jar"]
```

Tip #8: Look for minimal flavors

REPOSITORY	TAG	SIZE
openjdk	8	624MB
openjdk	8-jre	443MB
openjdk	8-jre-slim	204MB
openjdk	8-jre-alpine	83MB

Reproducibility

Tip #9: Build from source in a consistent environment

The source code is the source of truth from which you want to build a Docker image. The Dockerfile is simply the blueprint.

```
~~FROM openjdk:8-jre-alpine~~
FROM maven:3.6-jdk-8-alpine
WORKDIR /app
~~COPY app.jar /app~~
COPY pom.xml .
COPY src ./src
RUN mvn -e -B package
CMD ["java", "-jar", "/app/app.jar"]
```

Tip #10: Fetch dependencies in a separate step

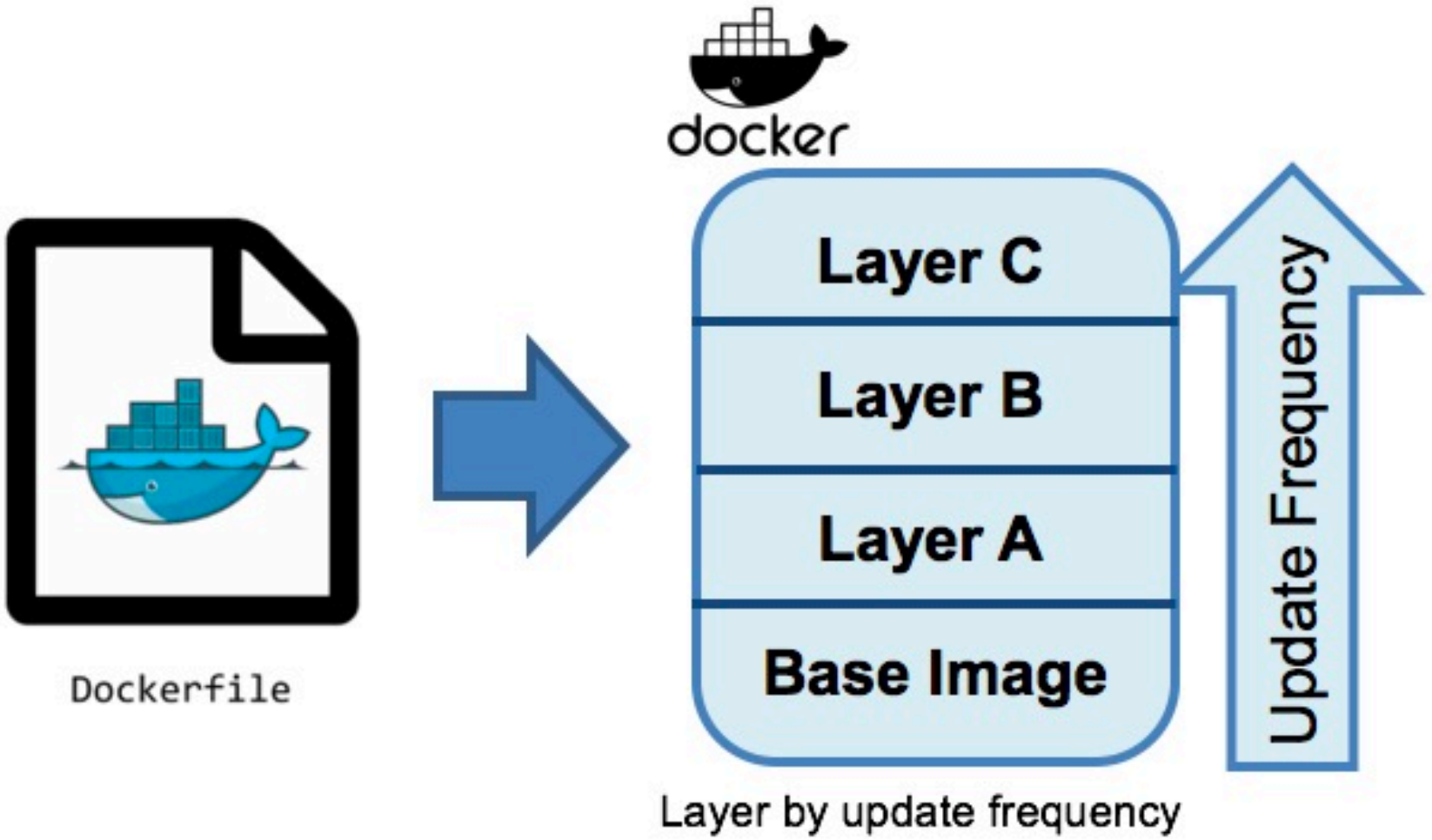
```
FROM maven:3.6-jdk-8-alpine
WORKDIR /app
COPY pom.xml .
RUN mvn -e -B dependency:resolve
COPY src ./src
RUN mvn -e -B package
CMD ["java", "-jar", "/app/app.jar"]
```

Reproducibility

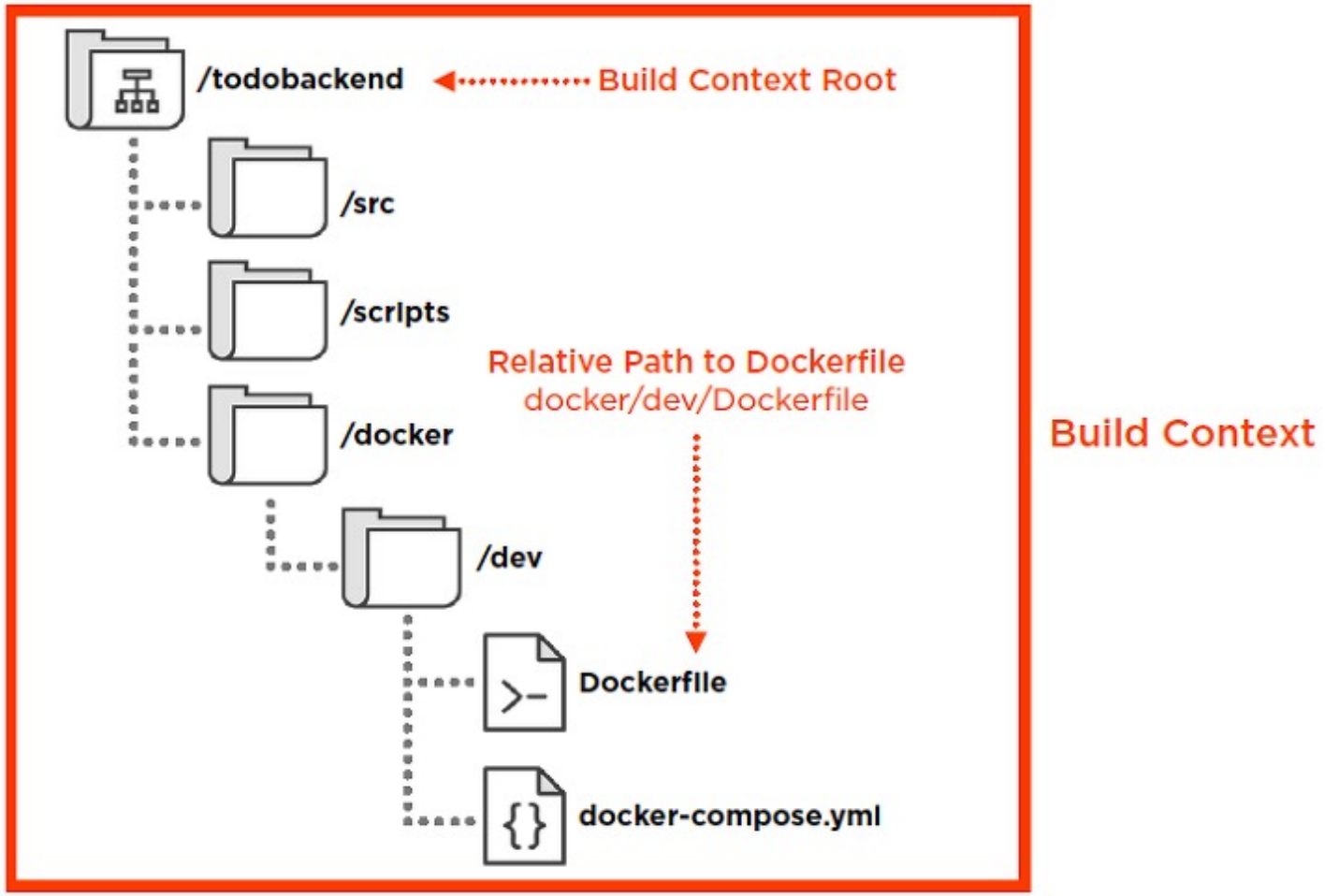
Tip #11: Use multi-stage builds to remove build dependencies (recommended Dockerfile)

```
FROM maven:3.6-jdk-8-alpine AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn -e -B dependency:resolve
COPY src ./src
RUN mvn -e -B package
CMD ["java", "-jar", "/app/app.jar"]

FROM openjdk:8-jre-alpine
COPY --from=builder /app/target/app.jar /
CMD ["java", "-jar", "/app.jar"]
```


docker
Layer C
Layer B
Layer A
Base Image
Update Frequency
Dockerfile
Layer by update frequency

Understand build context



Exclude with .dockerignore

```
# comment
*/temp*
*/*/temp*
temp?
```

This file causes the following build behavior:

Rule	Behavior
`# comment`	Ignored.
`*/temp*`	Exclude files and directories whose names start with `temp` in any immediate subdirectory of the root. For example, the plain file `/somedir/temporary.txt` is excluded, as is the directory `/somedir/temp`.
`*/*/temp*`	Exclude files and directories starting with `temp` from any subdirectory that is two levels below the root. For example, `/somedir/subdir/temporary.txt` is excluded.
`temp?`	Exclude files and directories in the root directory whose names are a one-character extension of `temp`. For example, `/tempa` and `/tempb` are excluded.