忙しい人のためのDocker入門

2018年Winterバージョン

@zembustu 前佛雅人
2018年12月20日
スクエアfreeセミナー

@zembutsu
前佛 雅人

Blog: https://pocketstudio.net　@zembutsu　zembutsu

SAKURA internet

さくらインターネット株式会社
技術本部ミドルウェアグループ
Technology Evangelist / Developer Advocate

- エバンジェリストチーム
- 石狩市の小学校プログラミング教育支援プロジェクト
- 仮想化基盤チーム

お持ち帰り
Key Takeaways

Containerization　Namespace・Cgroup

プロセスを簡単にコンテナ化（isolate）し、

「プロセス・ファイルシステム・ネットワーク・等々」に対して

Build Ship Run

簡単かつ素早く開発・移動・実行できるプラットフォームが Docker

1 Dockerコンテナは実行に必要な全てをパッケージして、簡単に動かせる

- アプリケーションを簡単に開発し、移動し、実行するためのプログラムとプラットフォームを提供するのがDocker
- クライアント・サーバ型

containerd

https://docker.com

2 Dockerイメージは複数のイメージ・レイヤとメタ情報の積み重なり

- イメージ・レイヤ（image layer）は読み込み専用
- 親子関係がある
- イメージに対する変更はCopy on Write（CoW）処理が走る
- コンテナ実行にはイメージが必要で、Docker Hubから得られる
- コンテナ実行時のみ、読み書きが可能なレイヤを追加

3 コンテナのプロセスはデフォルトでisolate（隔離・分離）された状態

- namespace（名前空間）でプロセス空間やファイルシステムやネットワーク等を分ける技術と、cgroups（コントロール・グループ）でリソースの利用上限を指定
- コンテナはポートをデフォルトで開かない
- ネットワークはブリッジ、ホスト、noneの3種類
- ボリュームはコンテナ間でファイルシステムを共有できる。名前付き（named）とホスト・ボリューム

From “Cloud Computing”,
To “Cloud Native”

クラウド・ネイティヴの時代

仮想化からクラウド・ネイティブへ

From Virtualization to Cloud Native

CLOUD NATIVE COMPUTING FOUNDATION

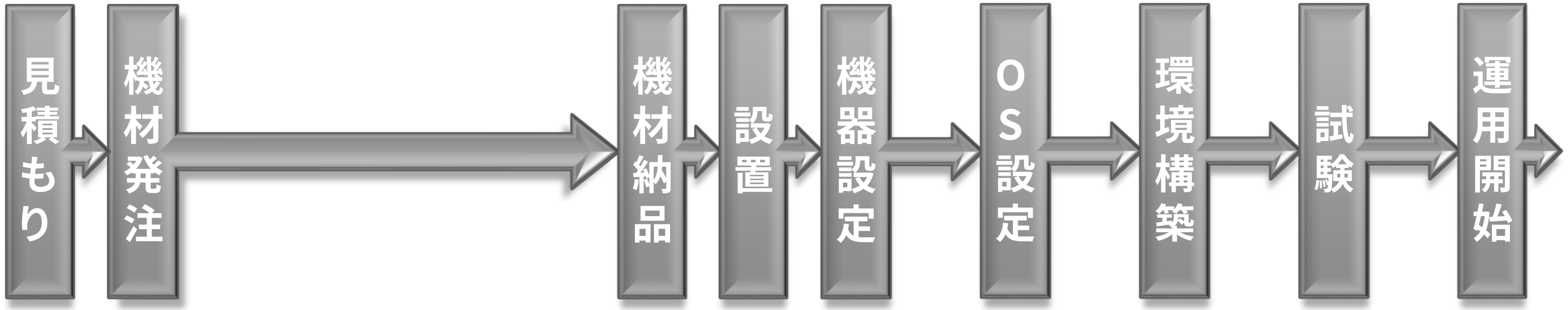
kubernetes

- クラウド・ネイティブ・コンピューティングはオープンソースのソフトウェアを積み重ね、次のために用います：
  - アプリケーションをマイクロサービス(*microservices*)に分割し、
  - 各パーツ自身をコンテナにパッケージし、
  - リソース利用を最適化するために、動的に統合/オーケストレート(*orchestrate*)する

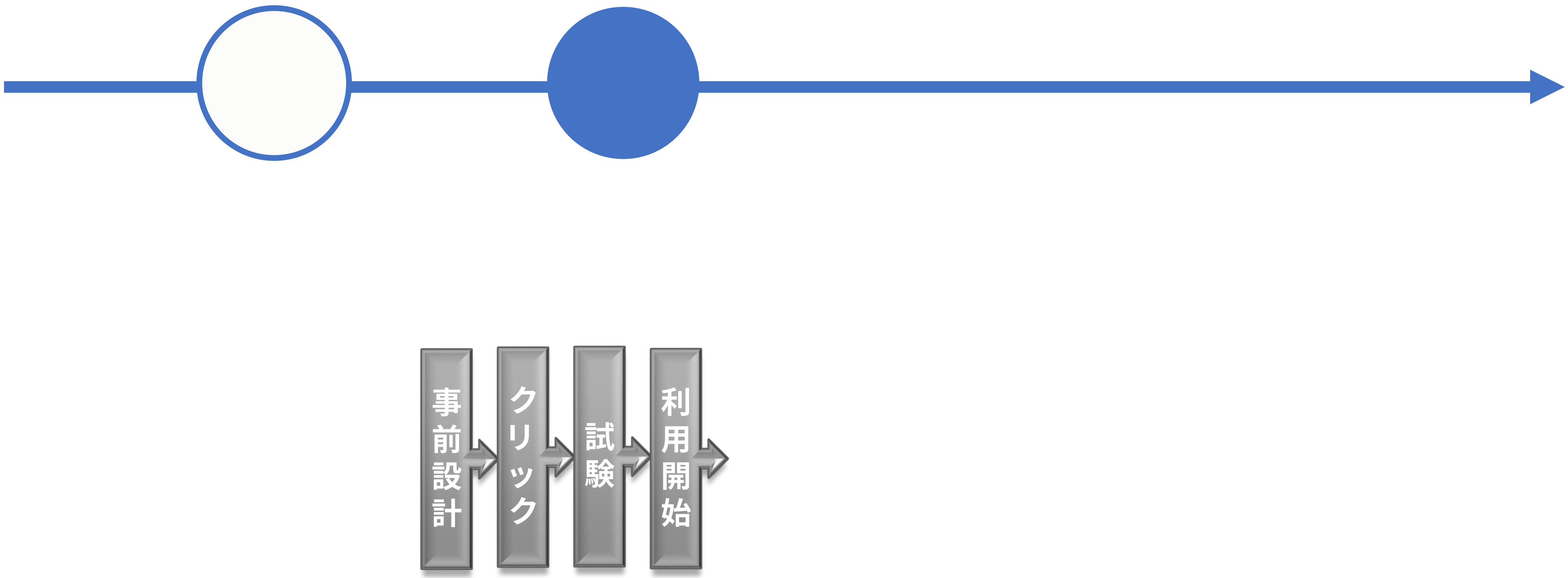
2000	2001	2006	2009	2010	2011	2013	2015
非仮想化ハードウェア	仮想化	IaaS	PaaS	オープンソース IaaS	オープンソース PaaS	コンテナ	クラウドネイティブ
Sun microsystems	vmware	aws	HEROKU	openstack	CLOUD FOUNDRY	docker	CLOUD NATIVE COMPUTING FOUNDATION

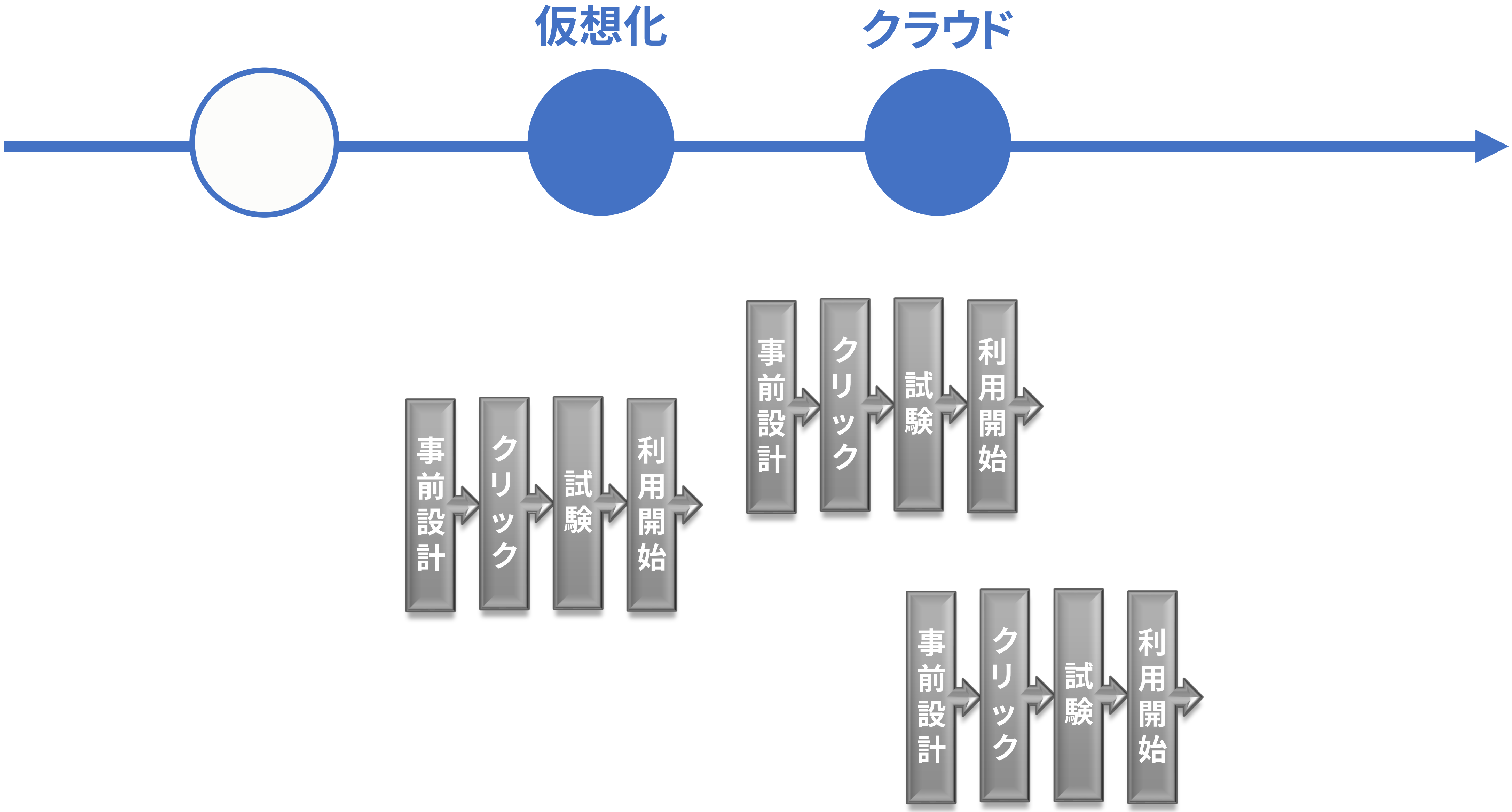
※ https://github.com/cncf/presentations からの引用

物理サーバ時代

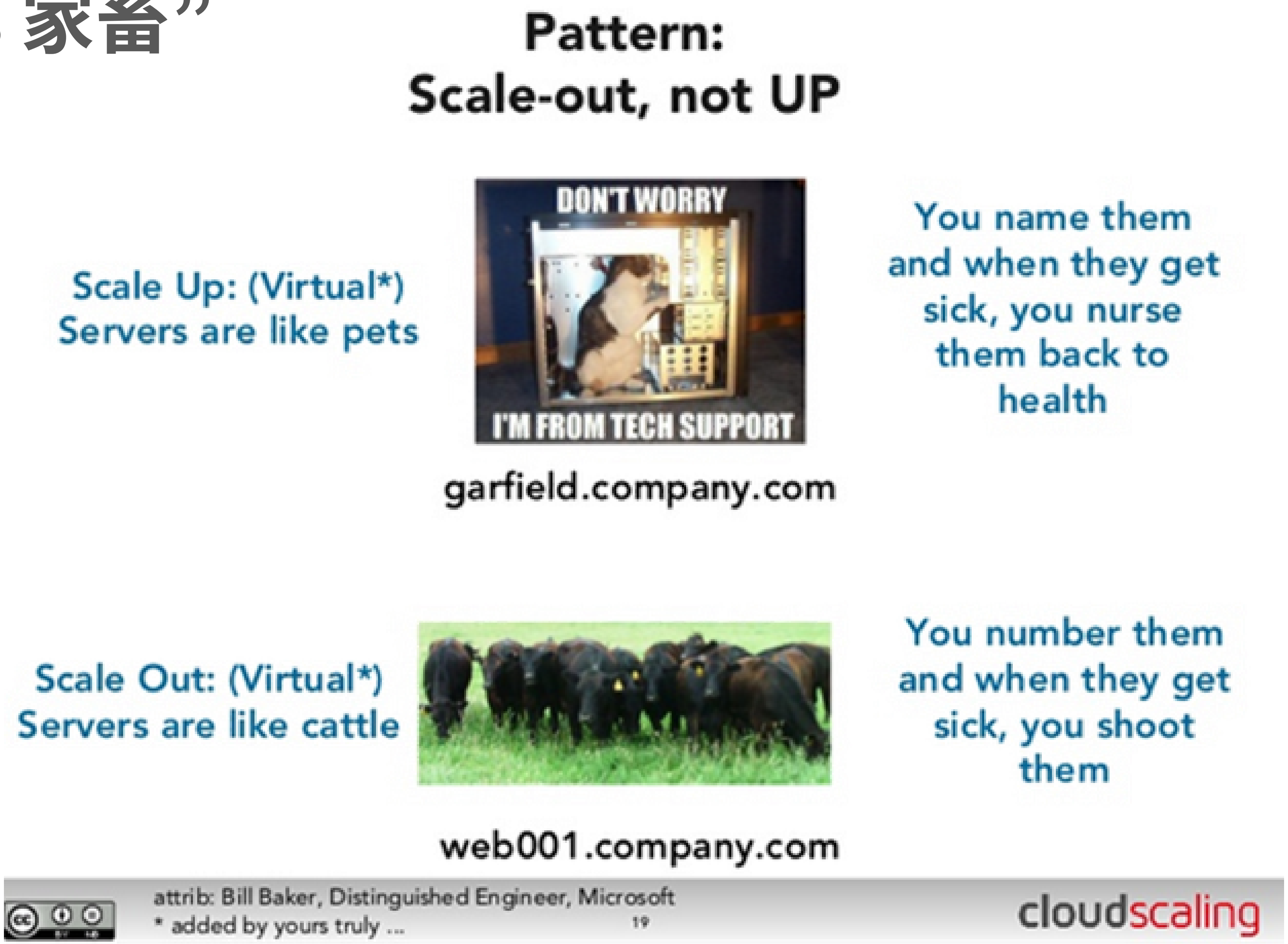


仮想化




仮想化
クラウド
事前設計
クリック
試験
利用開始
事前設計
クリック
試験
利用開始
事前設計
クリック
試験
利用開始

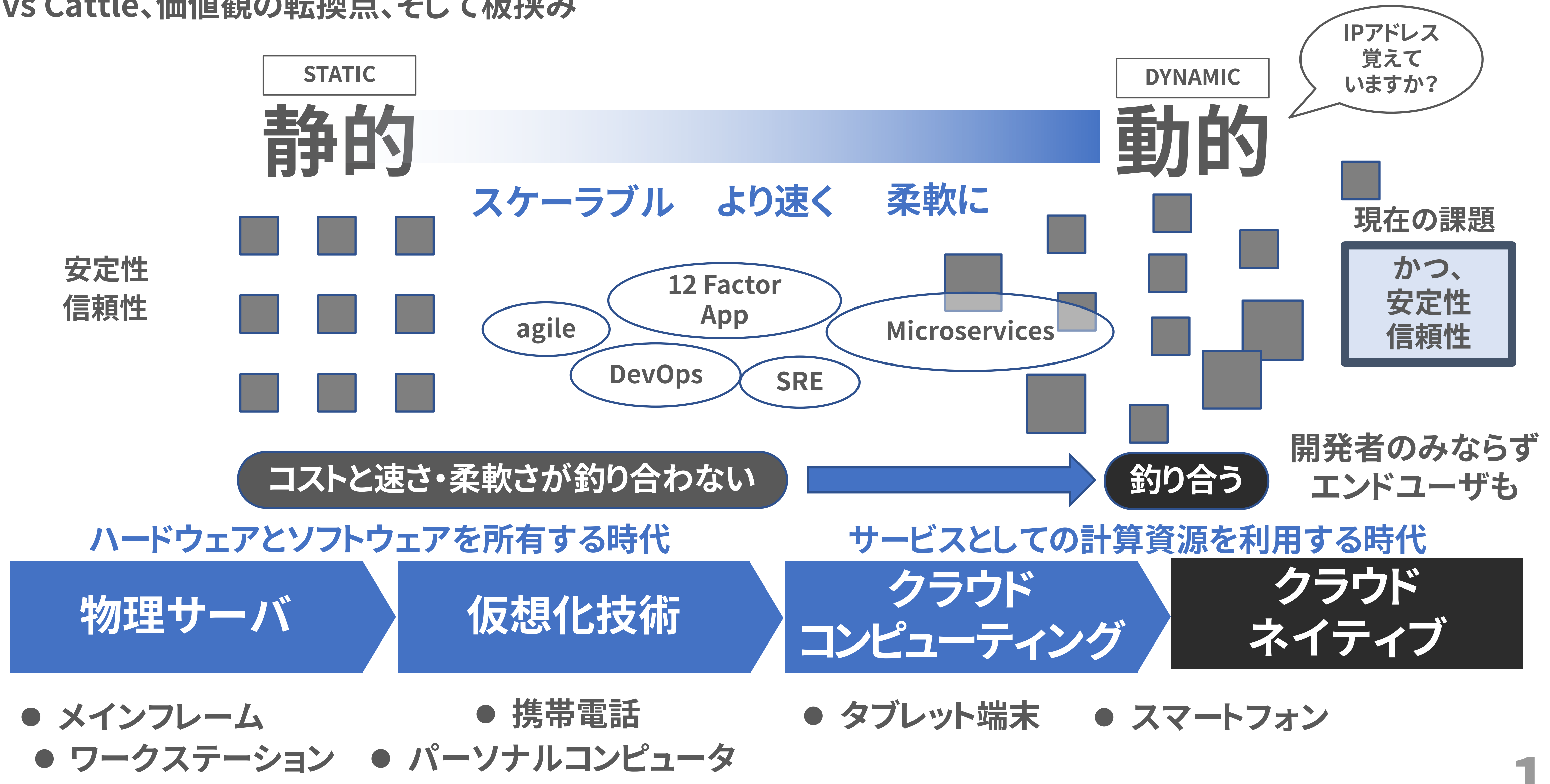
“ペット vs 家畜”



Architectures for open and scalable clouds
http://www.slideshare.net/randybias/architectures-for-open-and-scalable-clouds

システム基盤も、サービス側も変わり続ける前提

Pet vs Cattle、価値観の転換点、そして板挟み



Cloud Nativeの概念は以前から
クラウド・ネイティブ

- Service oriented architecture
- Redundancy
- Statelessness
- NoSQL
- Eventual consistency

Netflix Development Patterns for Scale,
Performance & Availability (DMG206) | AWS re:Invent 2013
https://www.slideshare.net/AmazonWebServices/dmg206

- Availability 可用性 — マイクロサービス
- Automation 自動化 — デプロイ / 管理
- Acceleration 加速 — CI/CD / OODA
- Anywhere! どこでも — コンテナはどこでも移動

これらのパターンであり、ソフトウェアが必要

【参考】https://github.com/cncf/presentations/blob/master/2016-software-circus/what-is-cloud-native/what-is-cloud-native.pdf

Cloud Native Computing Foundation (CNCF)

ベンダロックイン無くクラウドを移動できるように

CLOUD NATIVE
COMPUTING FOUNDATION

2015年設立　プロジェクトを支援

THE LINUX FOUNDATION

2000年設立。Linuxを中心としたオープンソースのエコシステムを築くため、コンピュータ業界を中心に自動車業界など、50以上のサブプロジェクトを持つ。幅広く業界との調整や標準化のために努める非営利団体。

オープンソースのソフトウェアを積み重ねて:

- **コンテナ化**

  アプリケーションやプロセス等の各パートをコンテナ内にパッケージ化し、再利用性、透明性、、リソースを分離

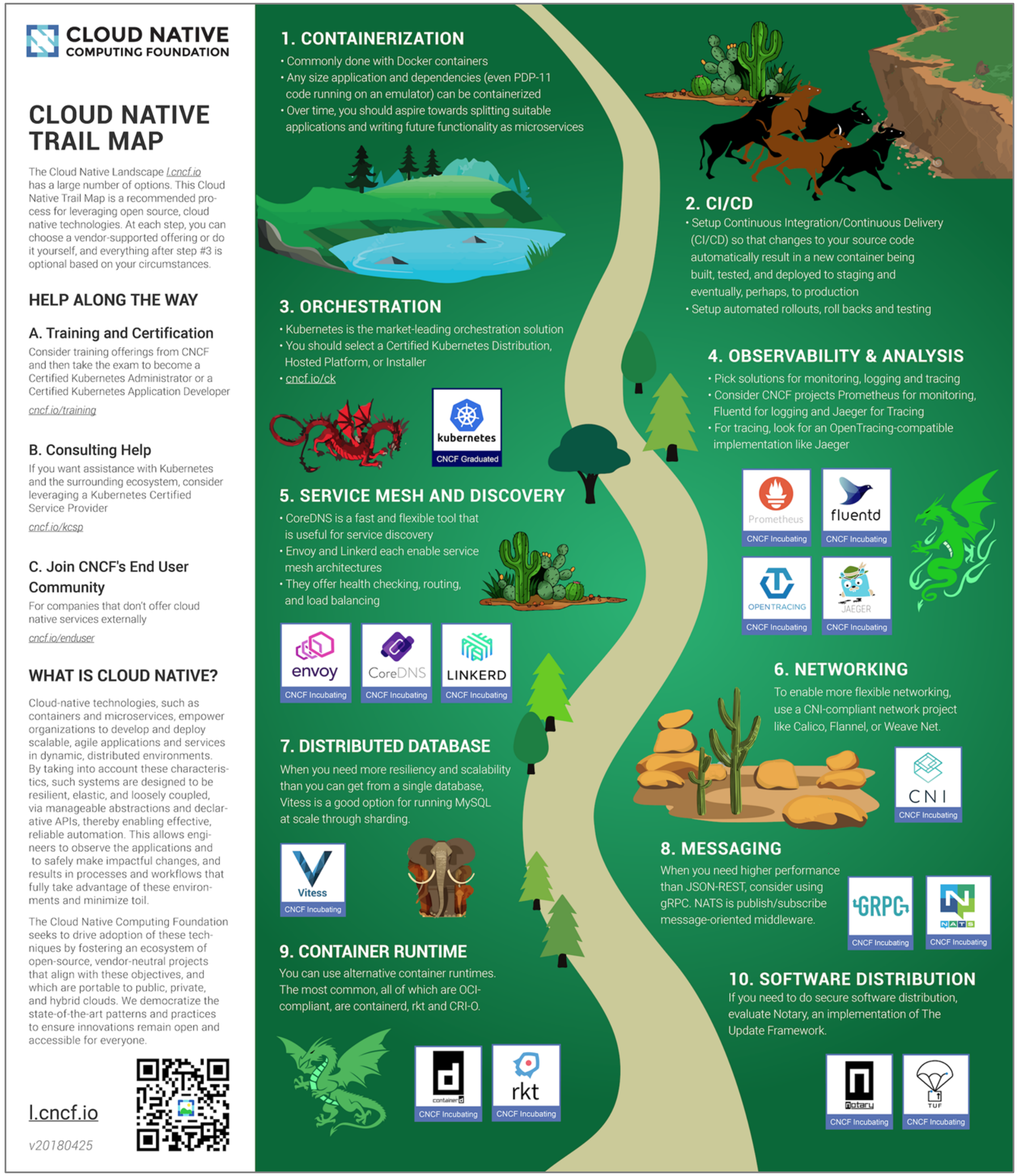
- **動的なオーケストレーション**

  コンテナを活発にスケジュールし、
  リソース利用率の最適化を管理

- **マイクロサービス指向**

  アプリケーションをマイクロサービスに分割し、
  全体的な敏捷性(agility)とメンテナンス性を極めて向上

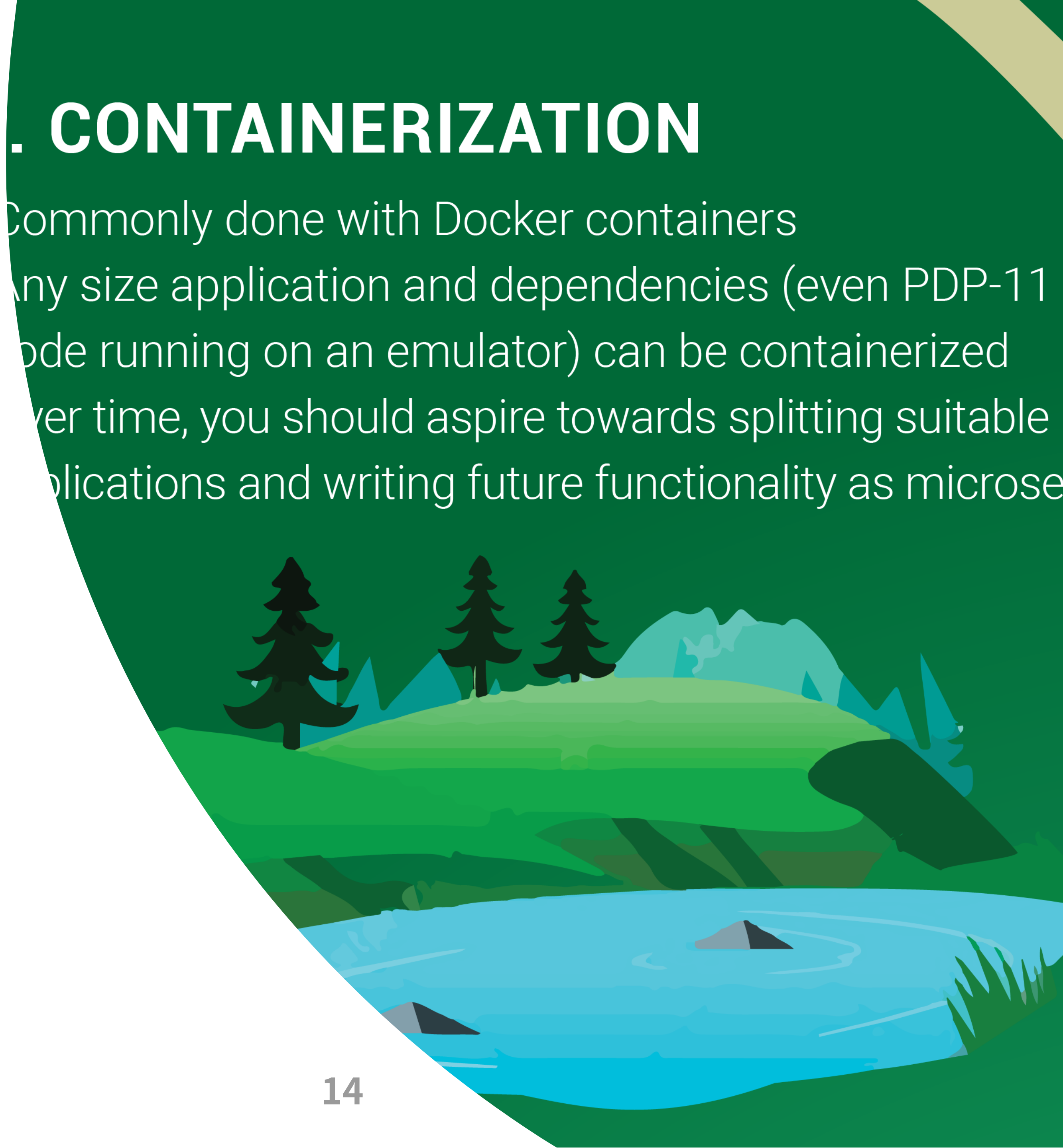
【参考】CNCF About https://www.cncf.io/about/faq/



コンテナやマイクロサービスのようなクラウドネイティブ技術は、組織が開発とデプロイをスケーラブルに、アジャイル・アプリケーションとサービスを動的に、かつ分散環境で行えるようになります。これらの特長の考慮した設計のシステムであれば、回復力のある（resilient）、弾力的（elastic）で、疎結合（loosely coupled）となり、管理可能な抽象化と宣言型の API を通すので、つまり、効率的で信頼性のある自動化をもたらすのです。これにより、エンジニアはアプリケーションを監視できたり、影響の大きな変更も簡単に行えたりします。そして**プロセスとワークフローは結果的に、これらの環境を最大限に活かし、苦痛を最小化するのです。**

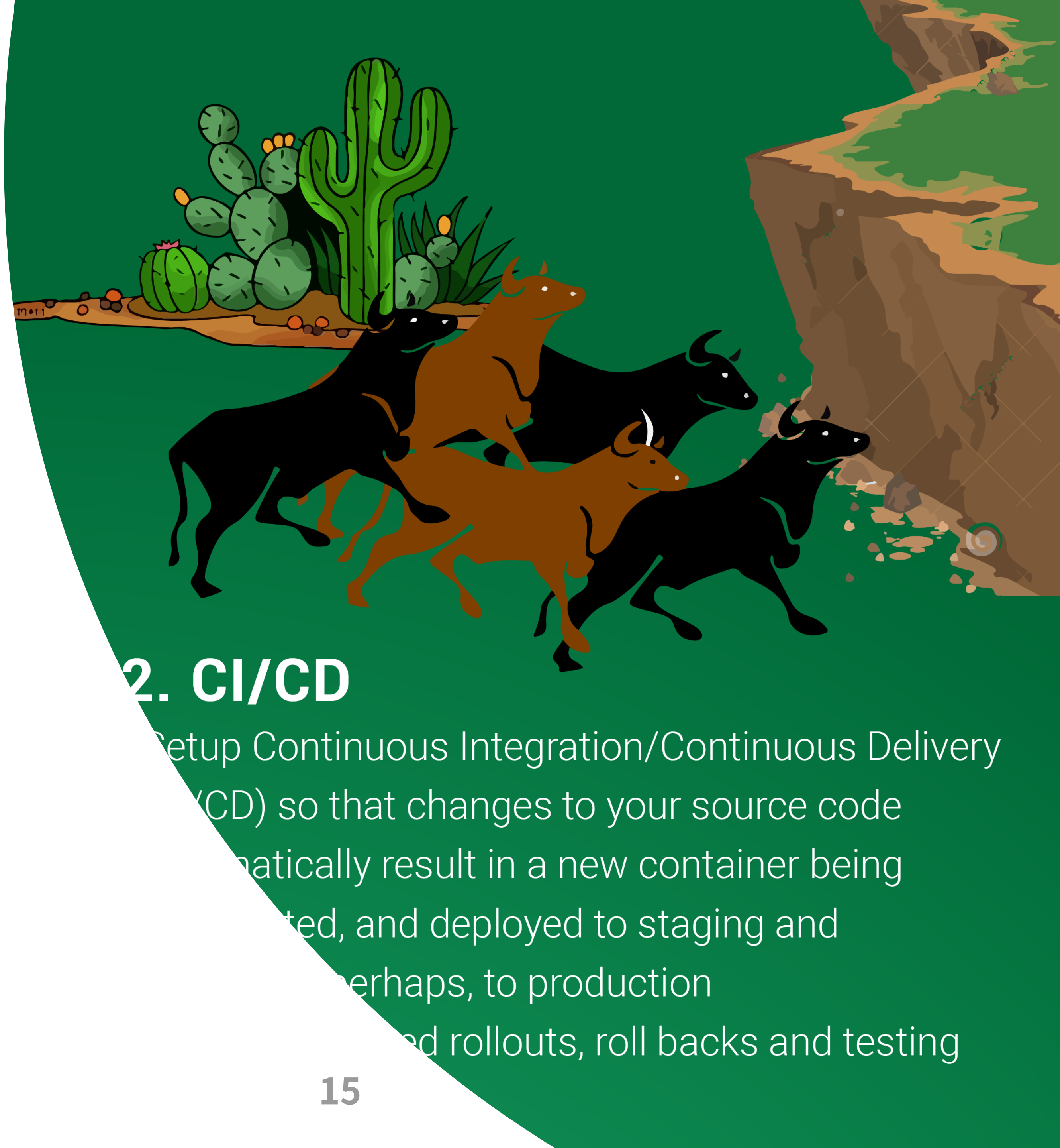
1. コンテナ化

- Docker コンテナで一般的に行う
- あらゆる大きさのアプリケーションと依存関係（エミュレータ上で実行する PDP-11 のコードですら）をコンテナ化できる
- 時間がたてば、適切なアプリケーション分割を求めるようになり、次世代の機能をマイクロサービスとして書くでしょう



2. CI/CD

- 継続的インテグレーション/継続的デリバリ(CI/CD)をセットアップすると、ソースコードに対する変更の結果、自動的に新しいコンテナが構築、テストされ、定期的にテスト環境に展開するだけでなく、本番環境にすら展開する。
- ロールアウト、ロールバック、テストを自動的に行うようセットアップする。



3. オーケストレーション

- Kubernetes は業界トップのオーケストレーション・ソリューションです。
- 認証 Kubernetes ディストリビューション、対応(Hosted)プラットフォーム、インストーラーから選択すべきです。

Cloud Native Landscape

v20180525

インタラクティブなランドスケープは l.cncf.io をご覧ください

Greyed logos are not open source

App Definition and Development

Database and Data Warehouse

Streaming

Source Code Management

Application Definition and Image Build

Continuous Integration / Continuous Delivery (CI/CD)

Orchestration & Management

Scheduling & Orchestration

Coordination & Service Discovery

Service Management

Runtime

Cloud-Native Storage

Container Runtime

Cloud-Native Network

Provisioning

Host Management / Tooling

Infrastructure Automation

Container Registries

Secure Images

Key Management

Cloud

Public

Private

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.

l.cncf.io

CLOUD NATIVE Landscape

CLOUD NATIVE COMPUTING FOUNDATION

Redpoint

Amplify

Platforms

Certified Kubernetes - Distribution

Certified Kubernetes - Hosted

Certified Kubernetes - Installer

Non-Certified Kubernetes

PaaS/Container Service

Observability & Analysis

Monitoring

Logging

Tracing

Serverless

Special

Kubernetes Certified Service Provider

Kubernetes Training Partner

Cloud Native 参照アーキテクチャ

kubernetes

containerd

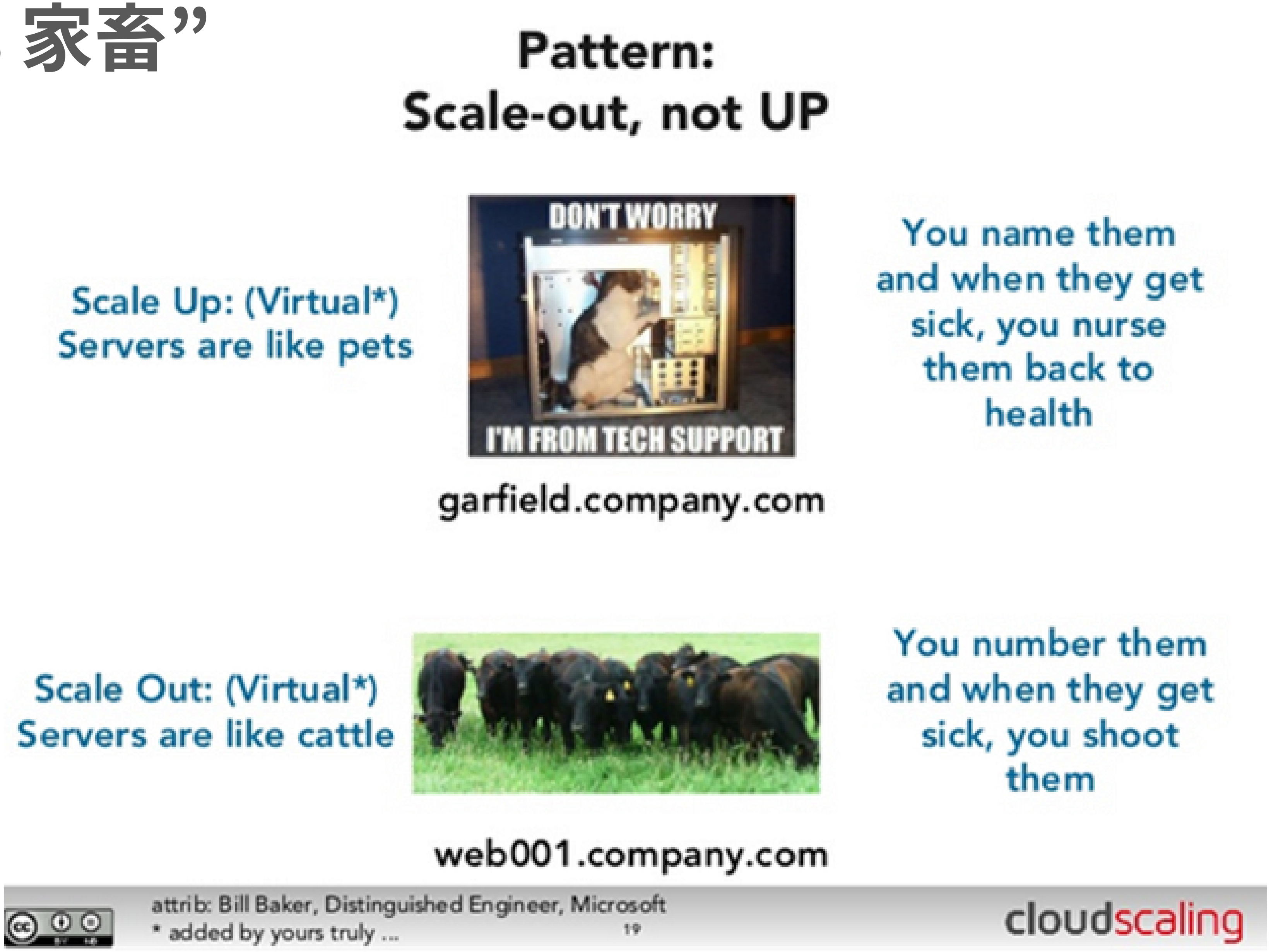
Application Definition / Development	マイクロサービス・パターン
Orchestration & Management	分散オーケストレーションと管理
Runtime	コンテナ化
Provisioning	
Networking	インフラ ※ CNCFプロジェクトが定義する範囲外
Compute	
Storage	

【参考】https://github.com/cncf/presentations/blob/master/2016-software-circus/what-is-cloud-native/what-is-cloud-native.pdf

Container

Dockerとは何なのか？

“ペット vs 家畜”



Architectures for open and scalable clouds
http://www.slideshare.net/randybias/architectures-for-open-and-scalable-clouds

Docker≠コンテナ

Docker

Dockerイメージとして

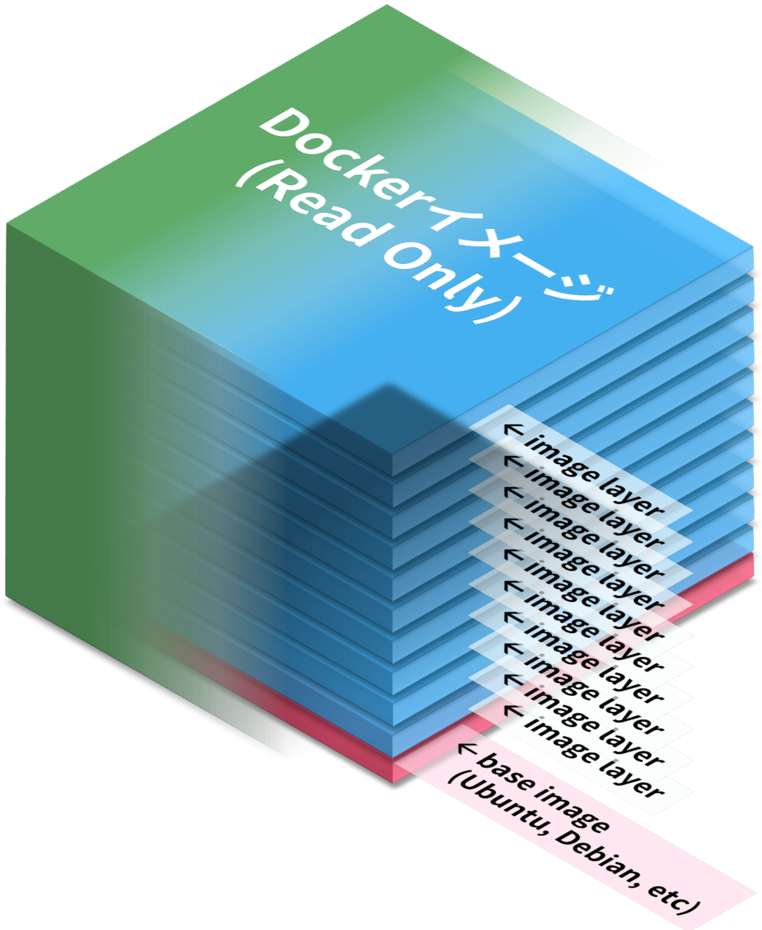
全ての依存関係をパッケージ化して、コンテナとして動かす

Linuxファイルシステムを

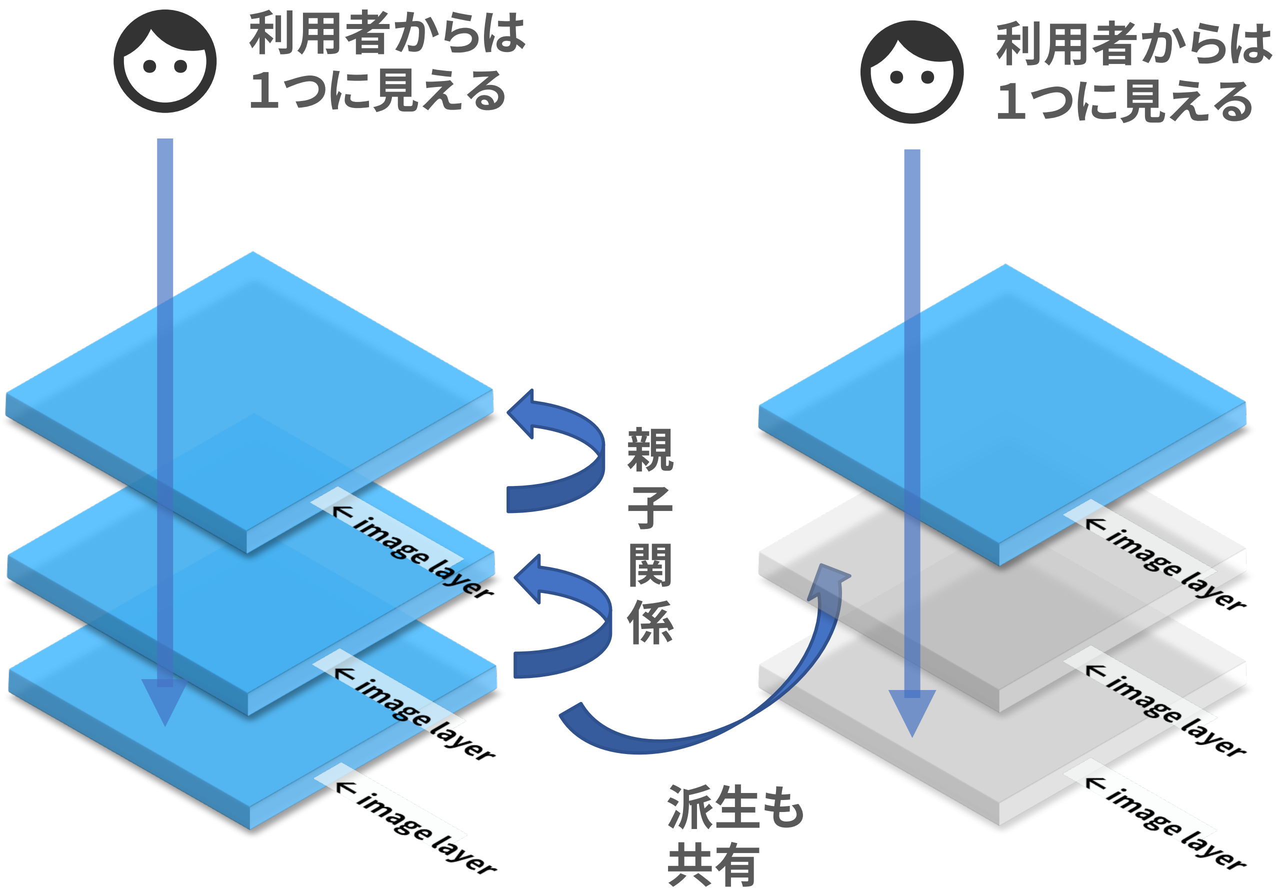
“Docker allows you to package an application with all of its dependencies into a standardized unit for software development.”

www.docker.com

Dockerイメージはイメージ・レイヤの積み重ね

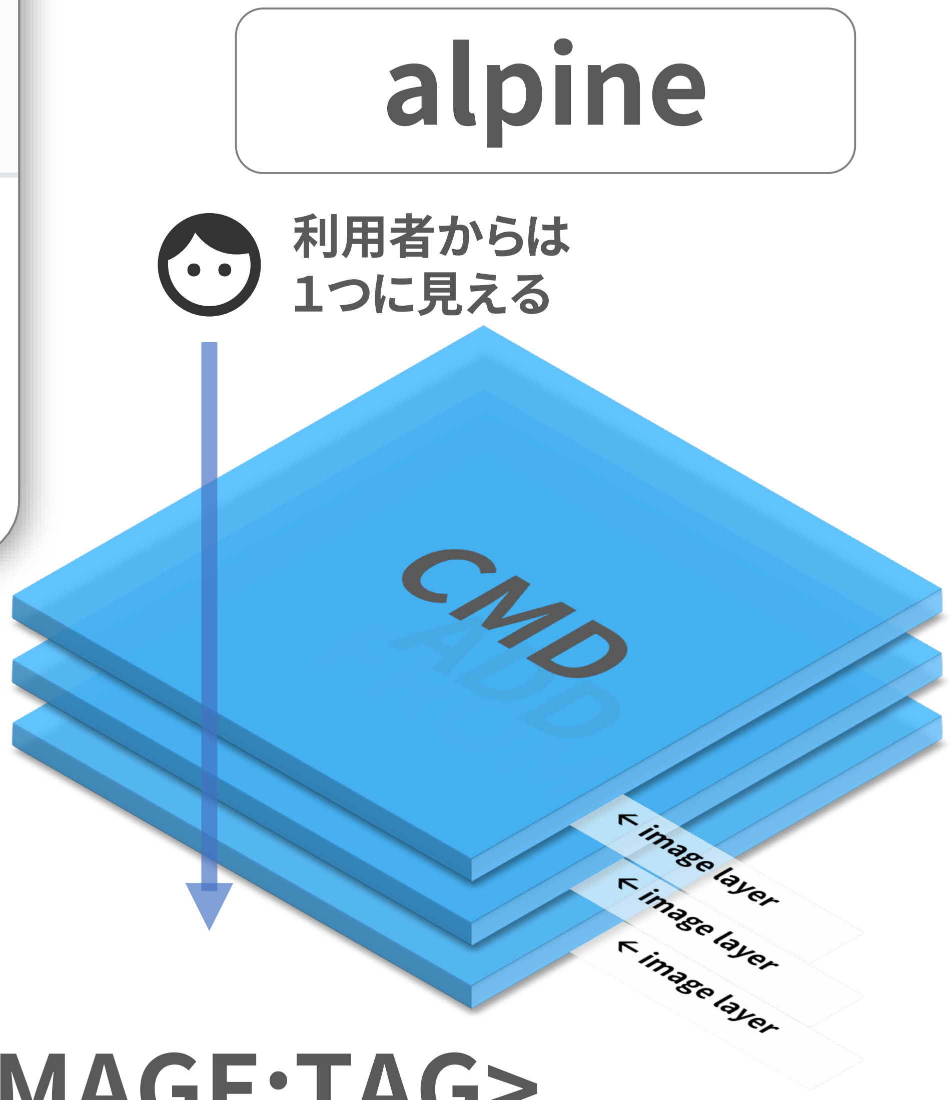


プログラムやライブラリと
メタ情報（実行するプログラムやポートなど）



だから高速に移動できる・開発を高速化できる
リソースを有効に使える “lightweight” な性質

```
4 lines (3 sloc) | 49 Bytes
1  FROM scratch
2  ADD rootfs.tar.xz /
3  CMD ["/bin/sh"]
```



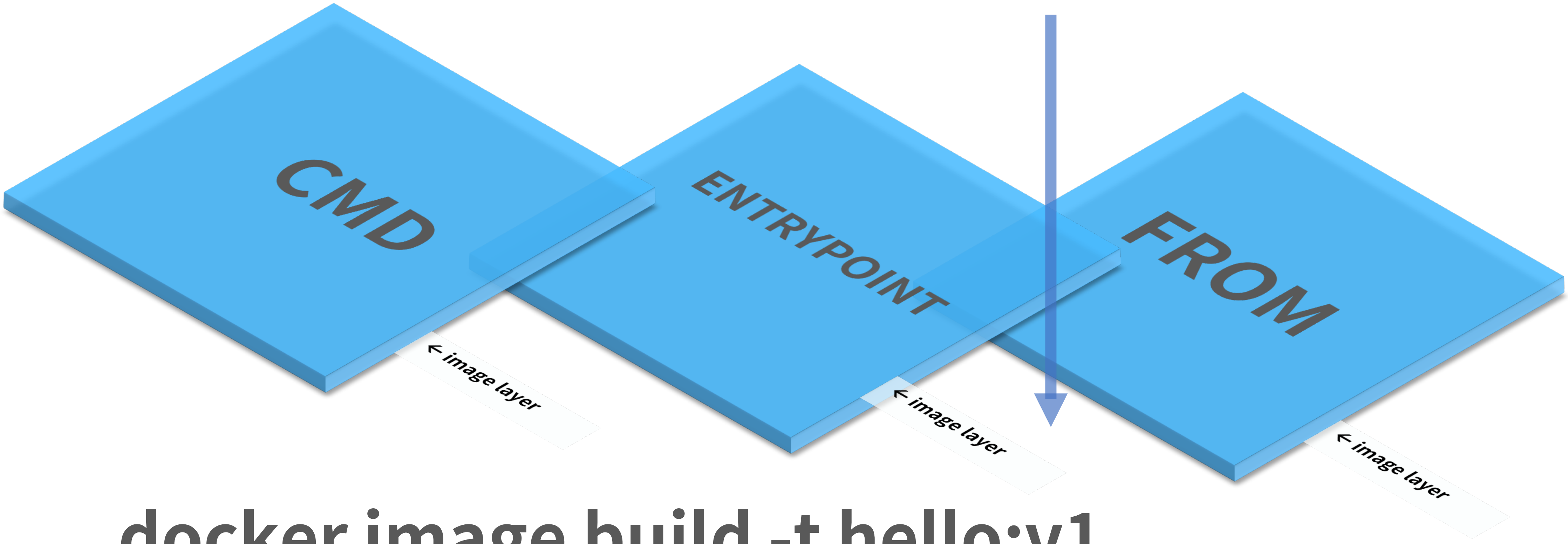
docker image build -t <IMAGE:TAG> .

(docker build)

```
FROM alpine
ENTRYPOINT ["/bin/echo"]
CMD ["こんにちは!こんにちは!"]
```

hello:v1

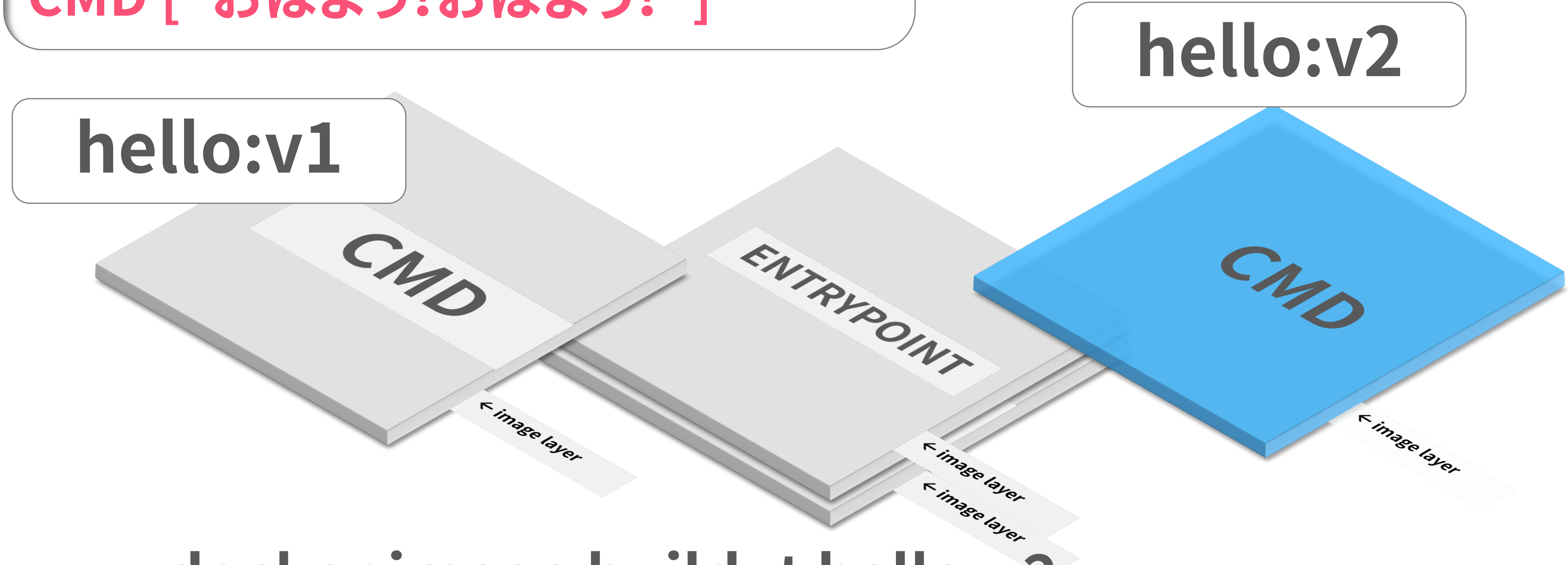
利用者からは
1つに見える



docker image build -t hello:v1 .

(docker build)

```
FROM alpine
ENTRYPOINT ["/bin/echo"]
CMD [“おはよう!おはよう!"]
```

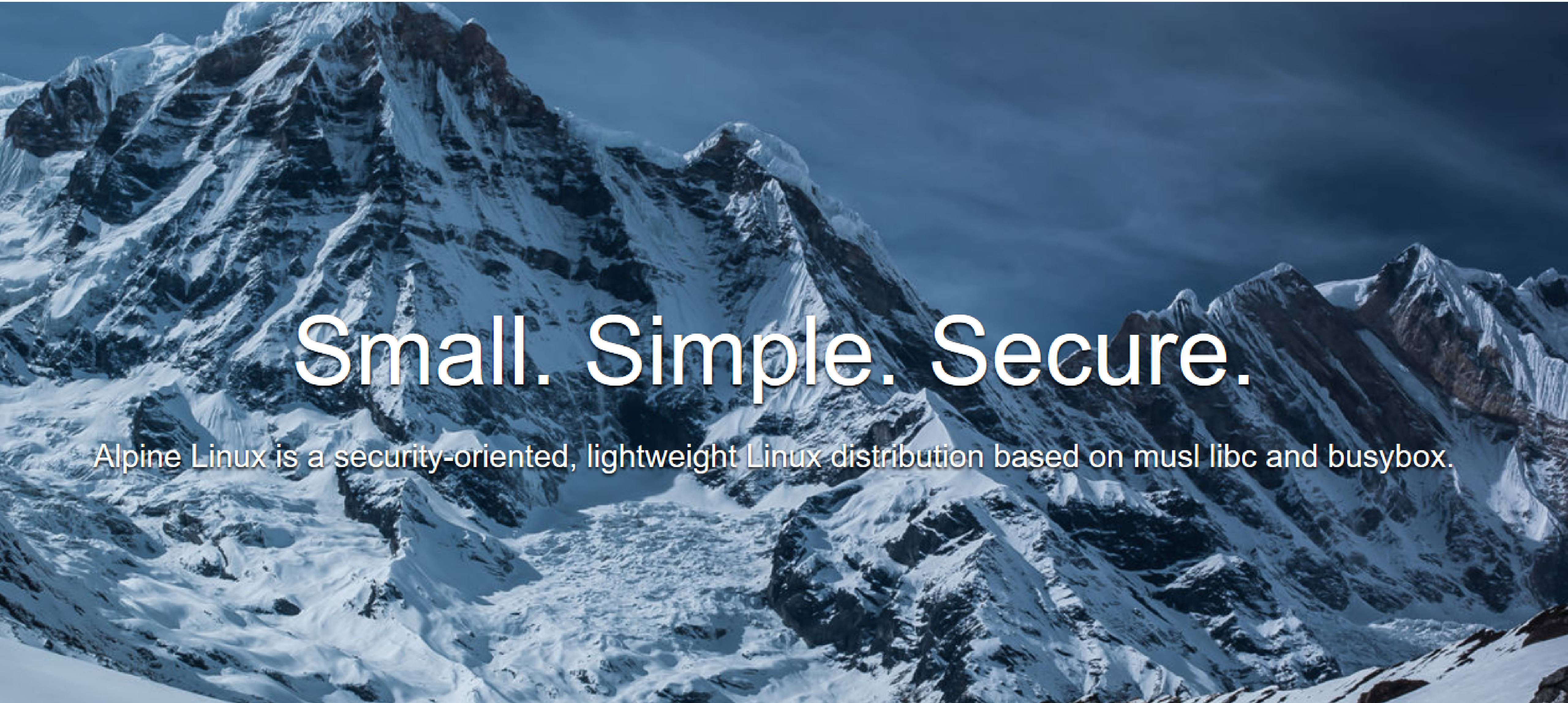


docker image build -t hello:v2 .

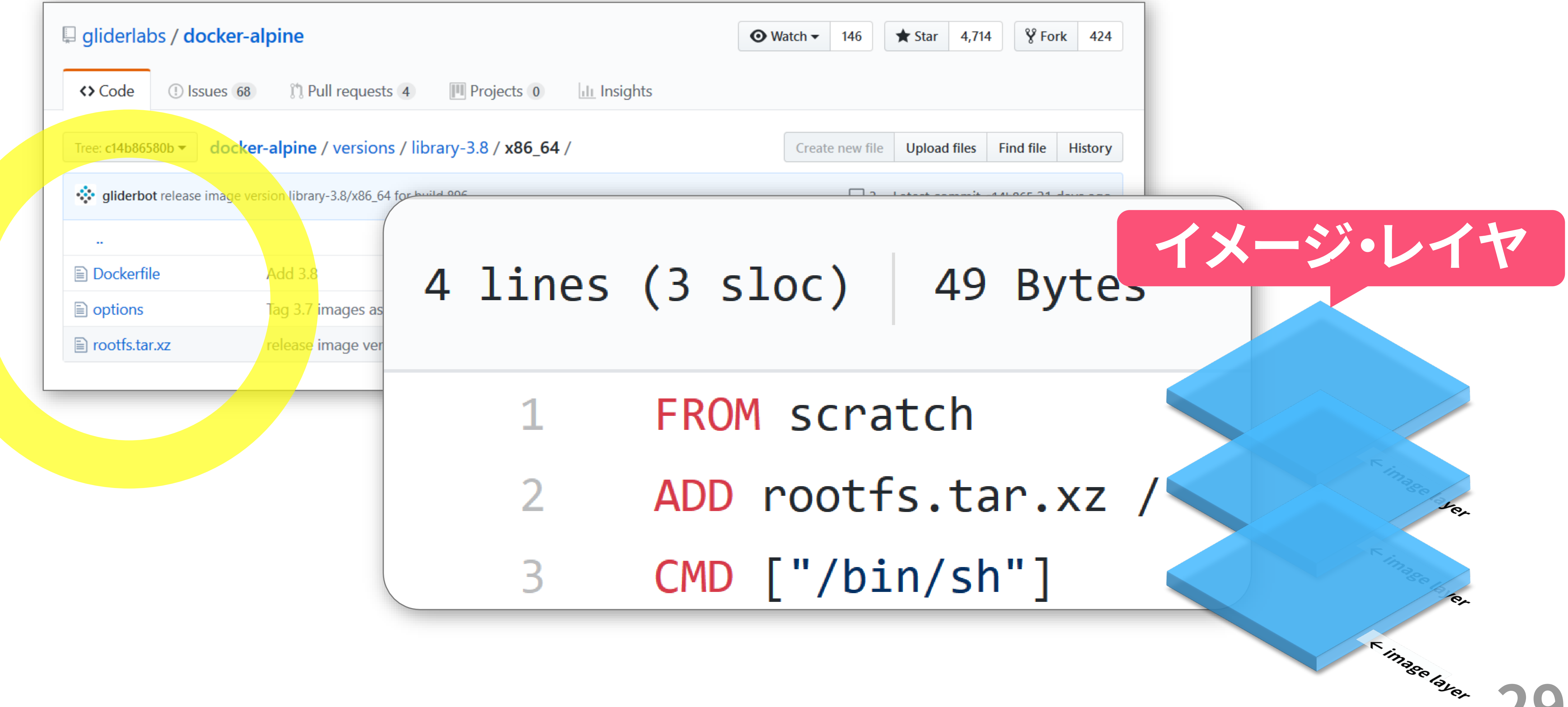
(docker build)

Alpine Linux

https://www.alpinelinux.org



gliderlabs / docker-alpine



docker image pull (docker pull)


```
$ docker pull hello-world
Using default tag: latest
latest: Pulling from library/hello-world
d1725b59e92d: Pull complete
Digest: sha256:0add3ace90ecb4adbf7777e9aacf18357296e799f81cabc9fde470971e499788
Status: Downloaded newer image for hello-world:latest
```

Docker

Dockerイメージとして

全ての依存関係をパッケージ化して、コンテナとして動かす

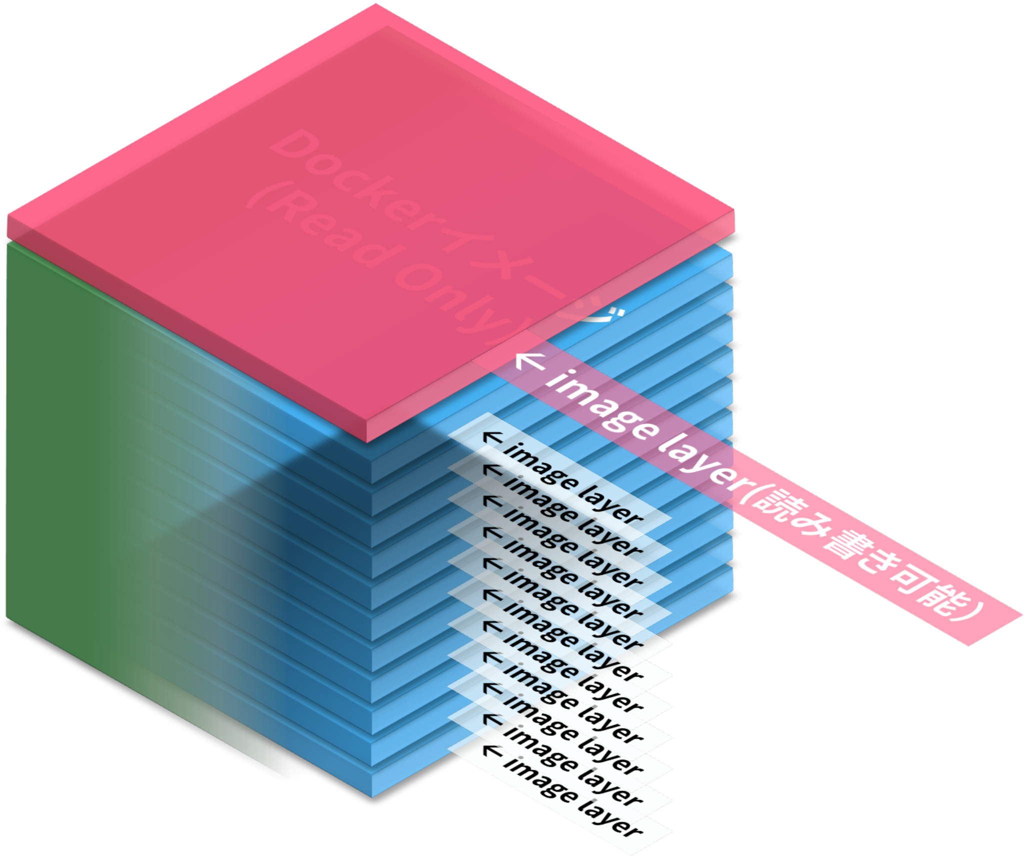
Linuxファイルシステムを

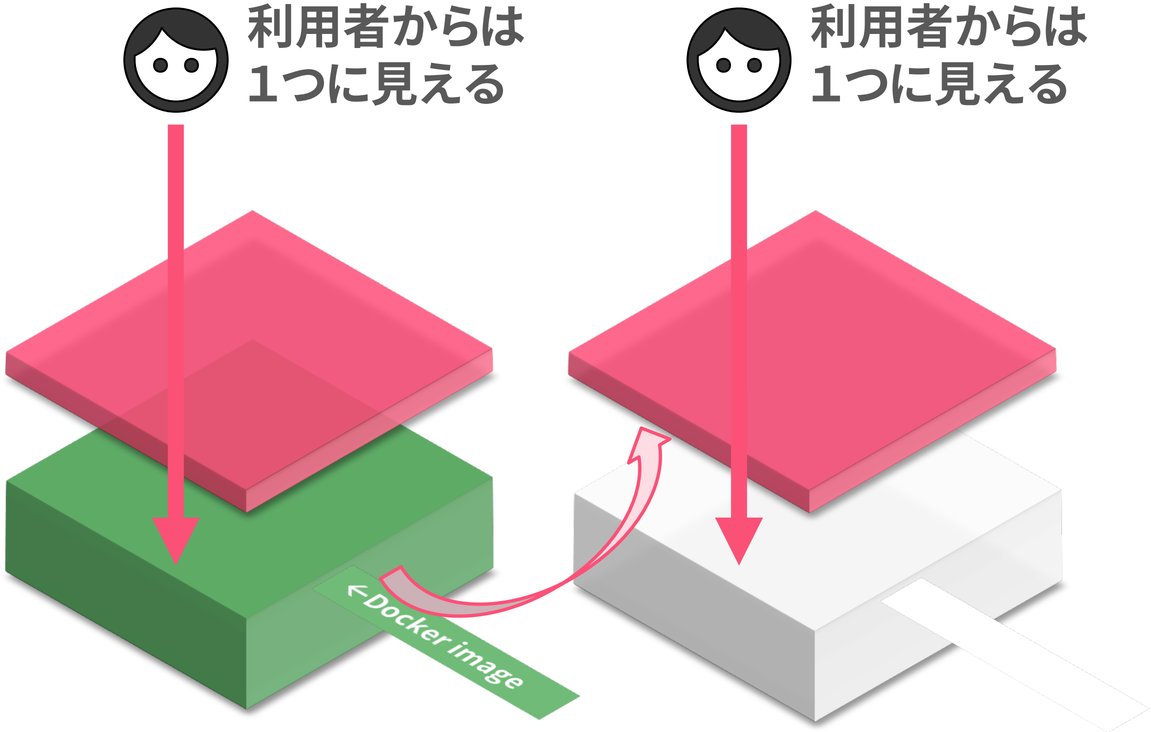
“Docker allows you to package an application with all of its dependencies into a standardized unit for software development.”

www.docker.com


TERMINAL BURCHARDKAI
HHLA
Hapag-Lloyd
HOUSTON EXPRESS
HAMBURG

DockerコンテナはDockerイメージを実行





元のレイヤに対する変更情報を記録
Copy on Write の性質

だから高速に移動できる・開発を高速化できる
リソースを有効に使える “lightweight” な性質

コンテナ
Technology

Docker
Specification

技術と仕様

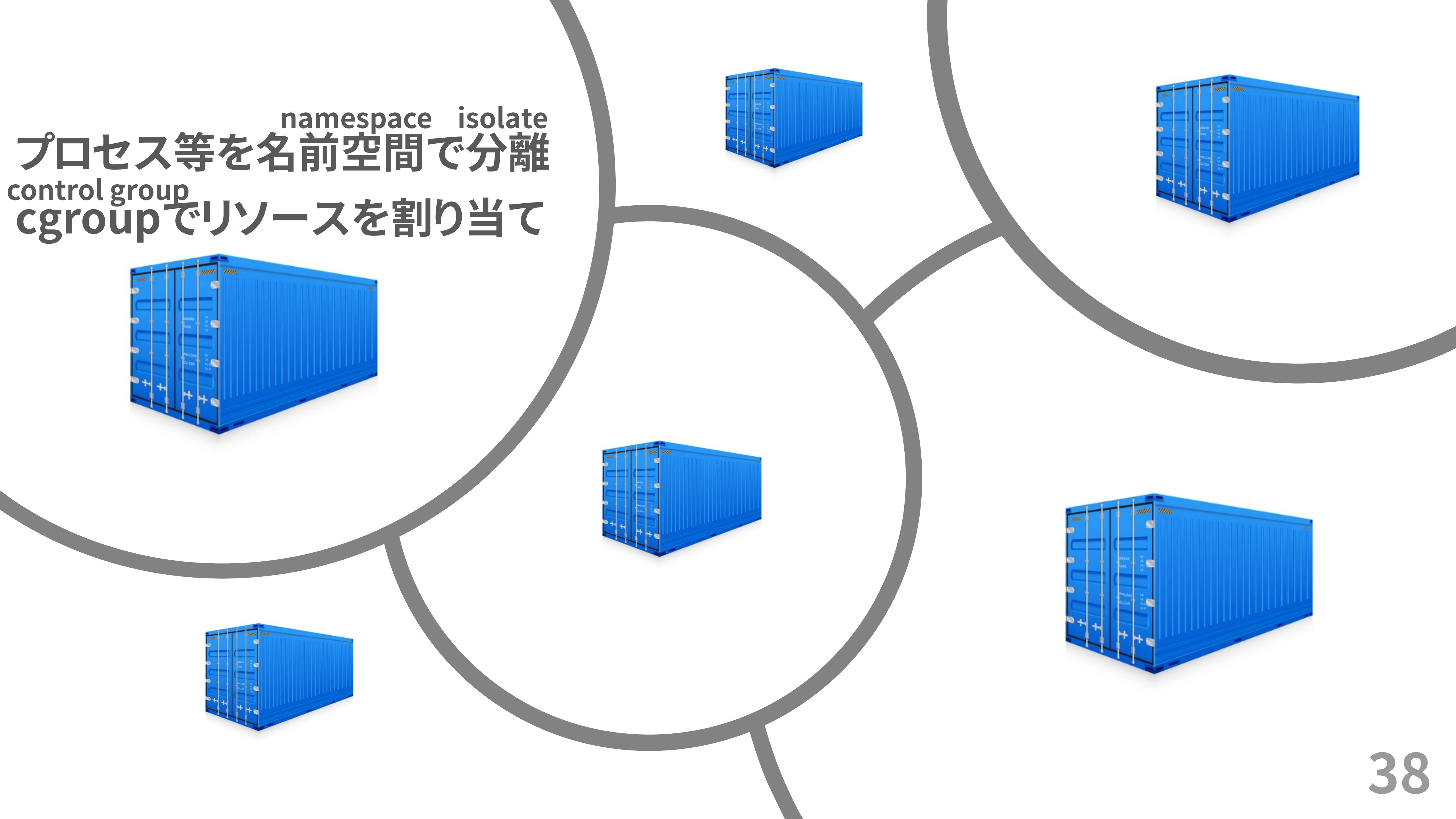
アプリケーションを isolate(分離)する

insulatus→isolated→isolate

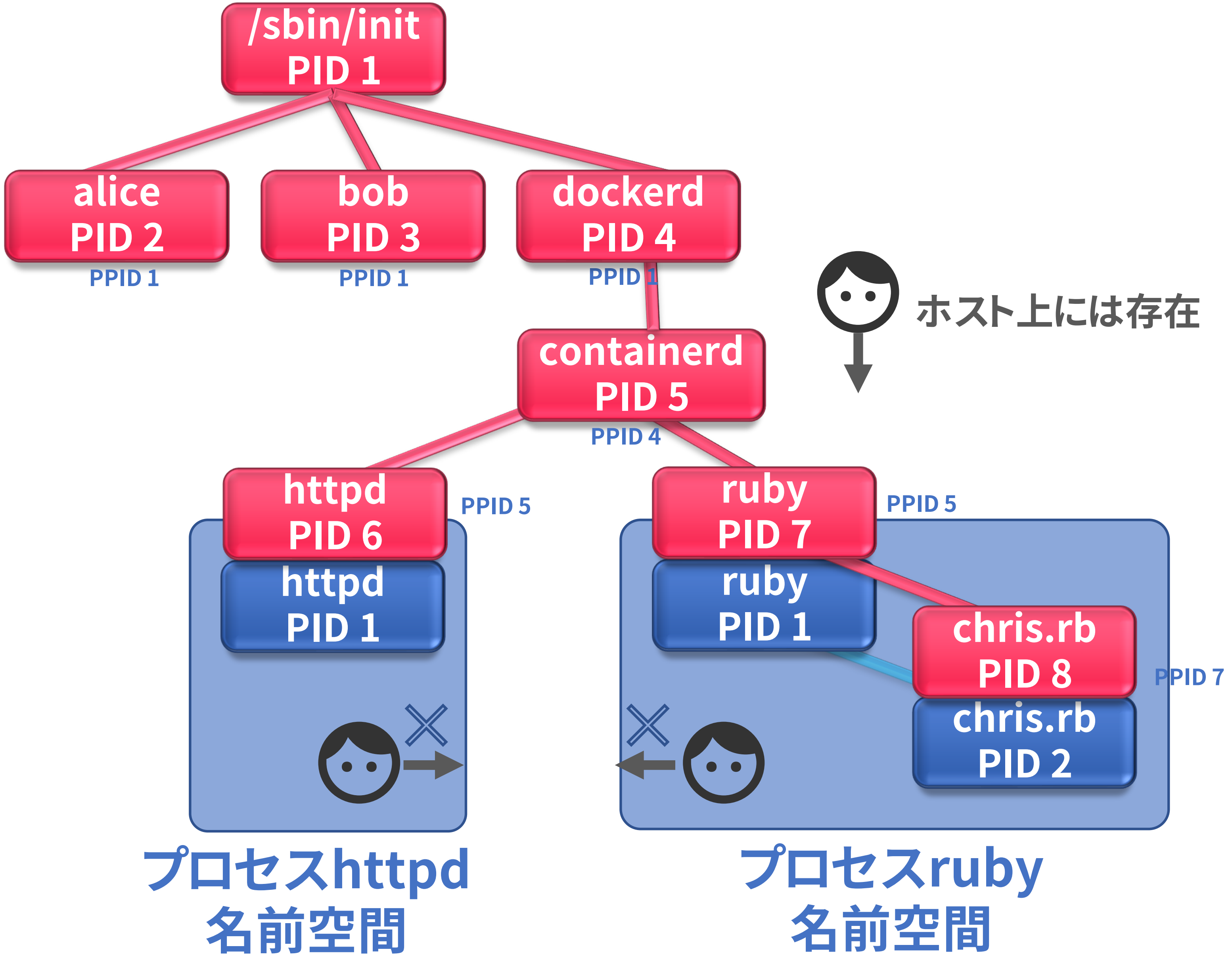
isle island

namespace isolate
プロセス等を名前空間で分離

control group
cgroupでリソースを割り当て

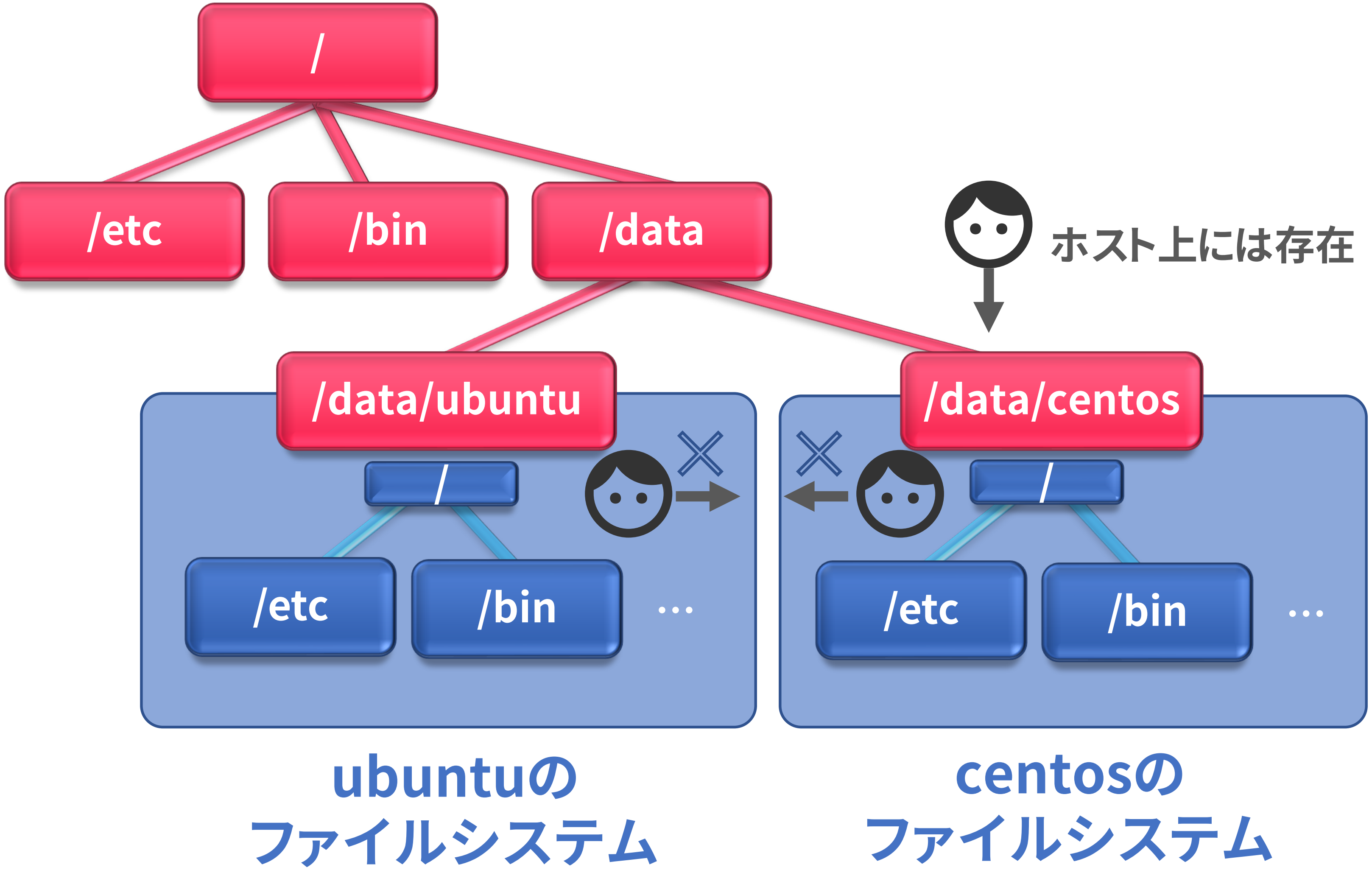


PID名前空間

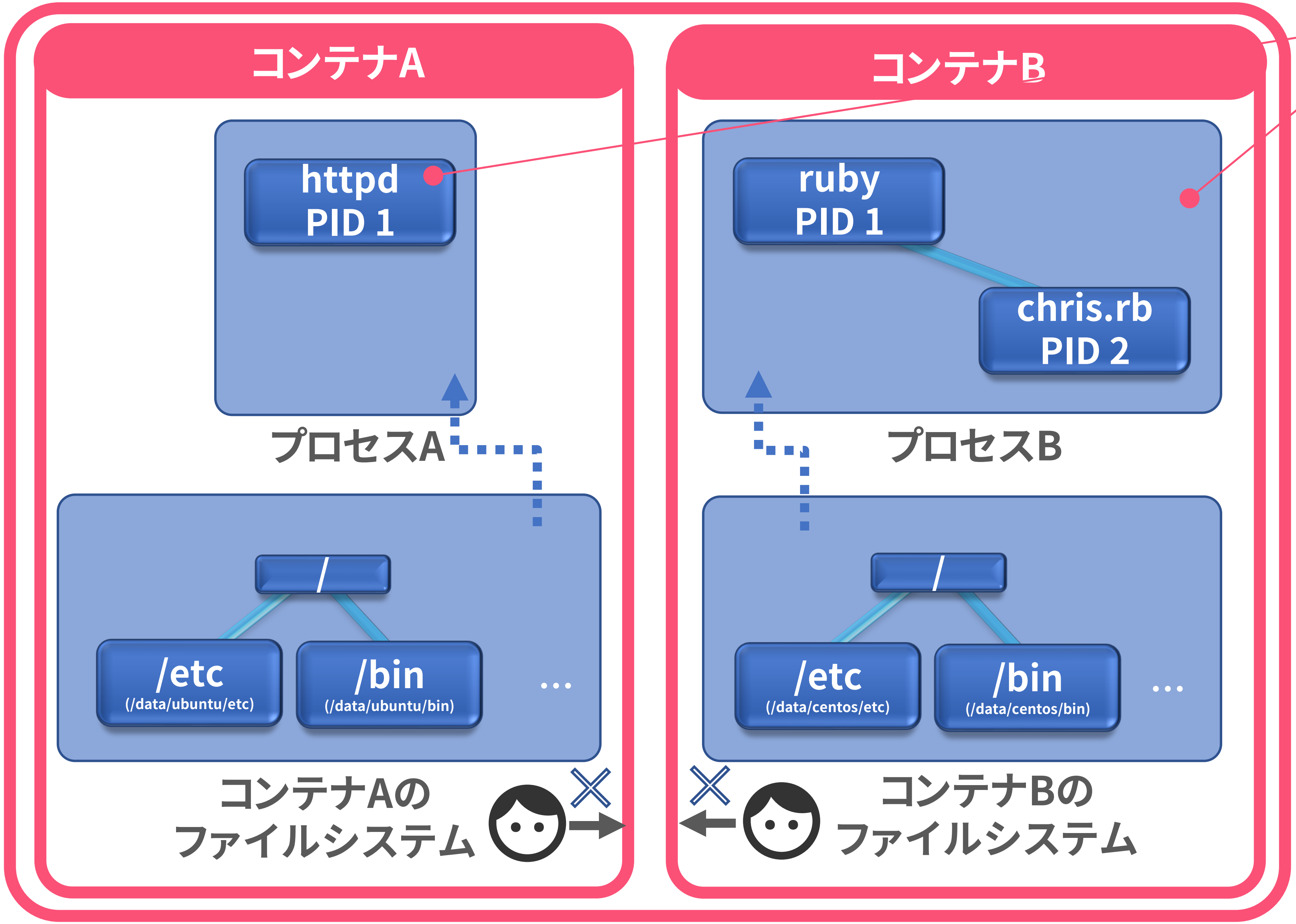


/sbin/init
PID 1
alice
PID 2
PPID 1
bob
PID 3
PPID 1
dockerd
PID 4
PPID 1
ホスト上には存在
containerd
PID 5
PPID 4
httpd
PID 6
PPID 5
httpd
PID 1
ruby
PID 7
PPID 5
ruby
PID 1
chris.rb
PID 8
PPID 7
chris.rb
PID 2
プロセスhttpd
名前空間
プロセスruby
名前空間

ファイルシステムを分ける (chroot)



コンテナはプロセスの状態



名前空間の isolate

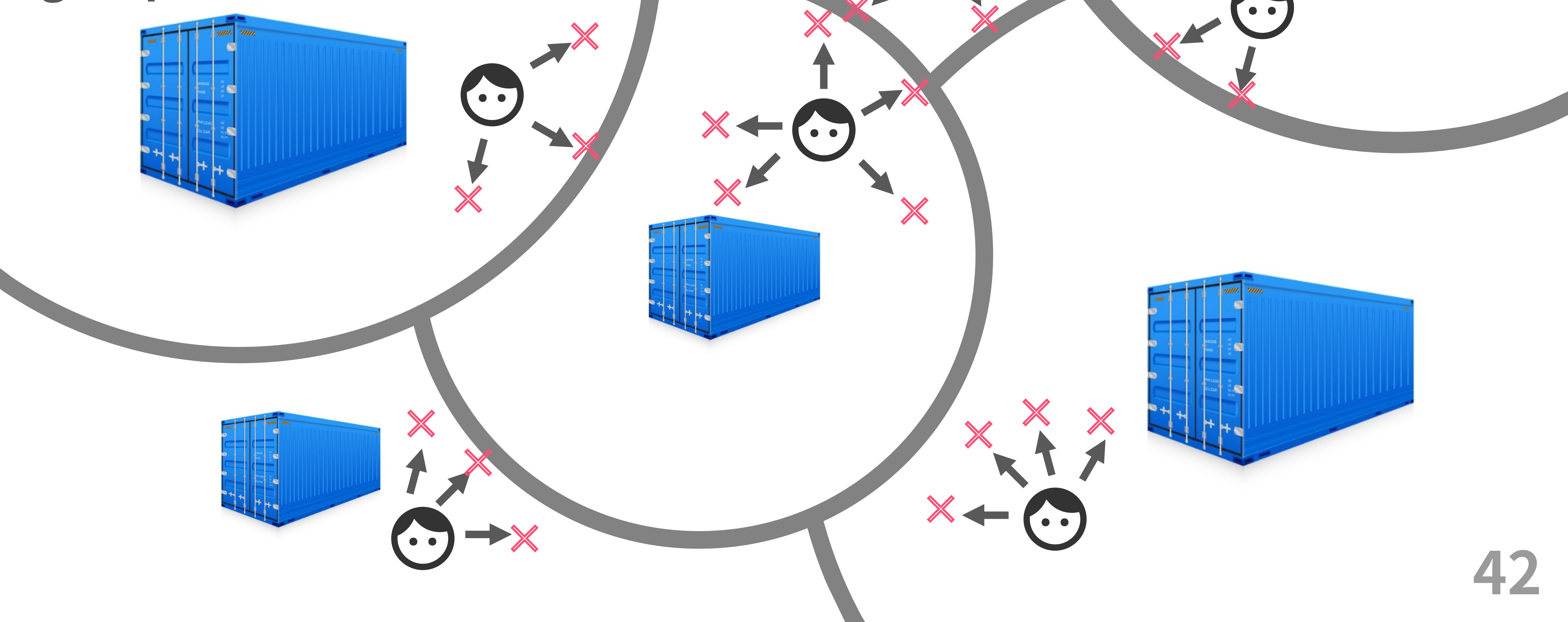
- ・プロセス
- ・ファイルシステム
- ・ネットワーク
- ・ホスト名
- ・UID・GID
- ・プロセス間通信、等

cgroupでリソース制限

- ・CPU
- ・メモリ
- ・I/O
- ・ディスク・クォータ、等

namespace isolate
プロセス等を名前空間で分離

control group
cgroupでリソースを割り当て



Docker

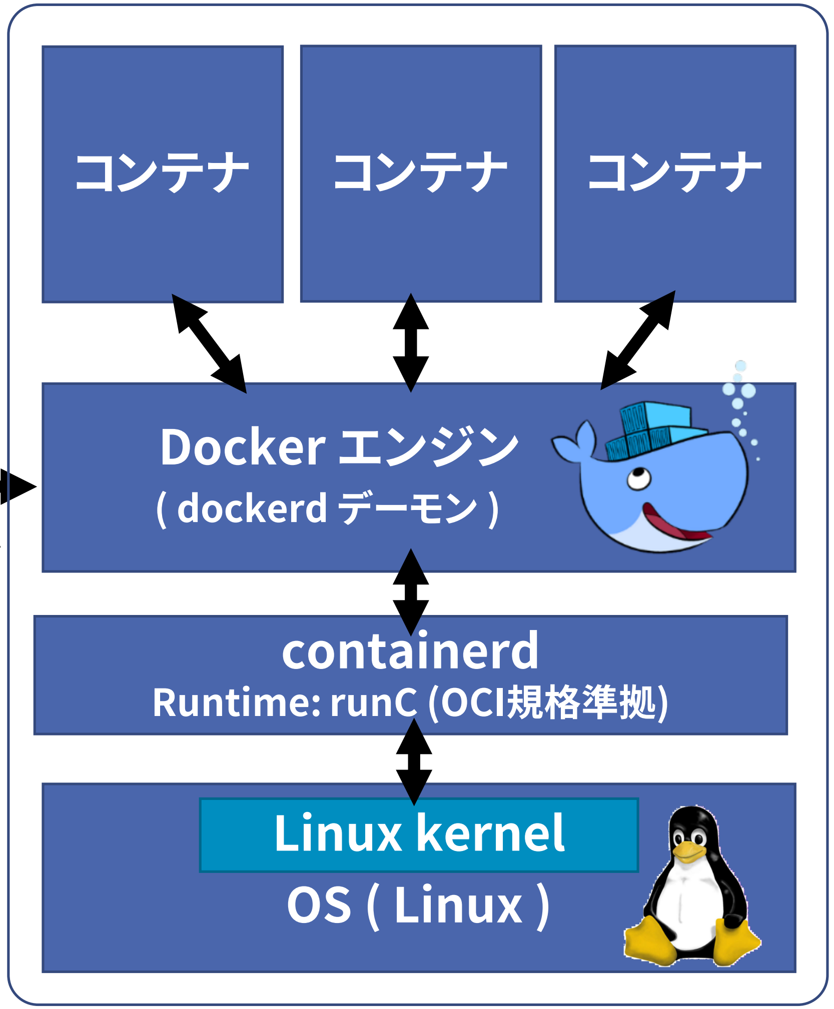
全ての依存関係をパッケージ化して、コンテナとして動かす

"Docker allows you to package an application with all of its dependencies into a standardized unit for software development."

www.docker.com

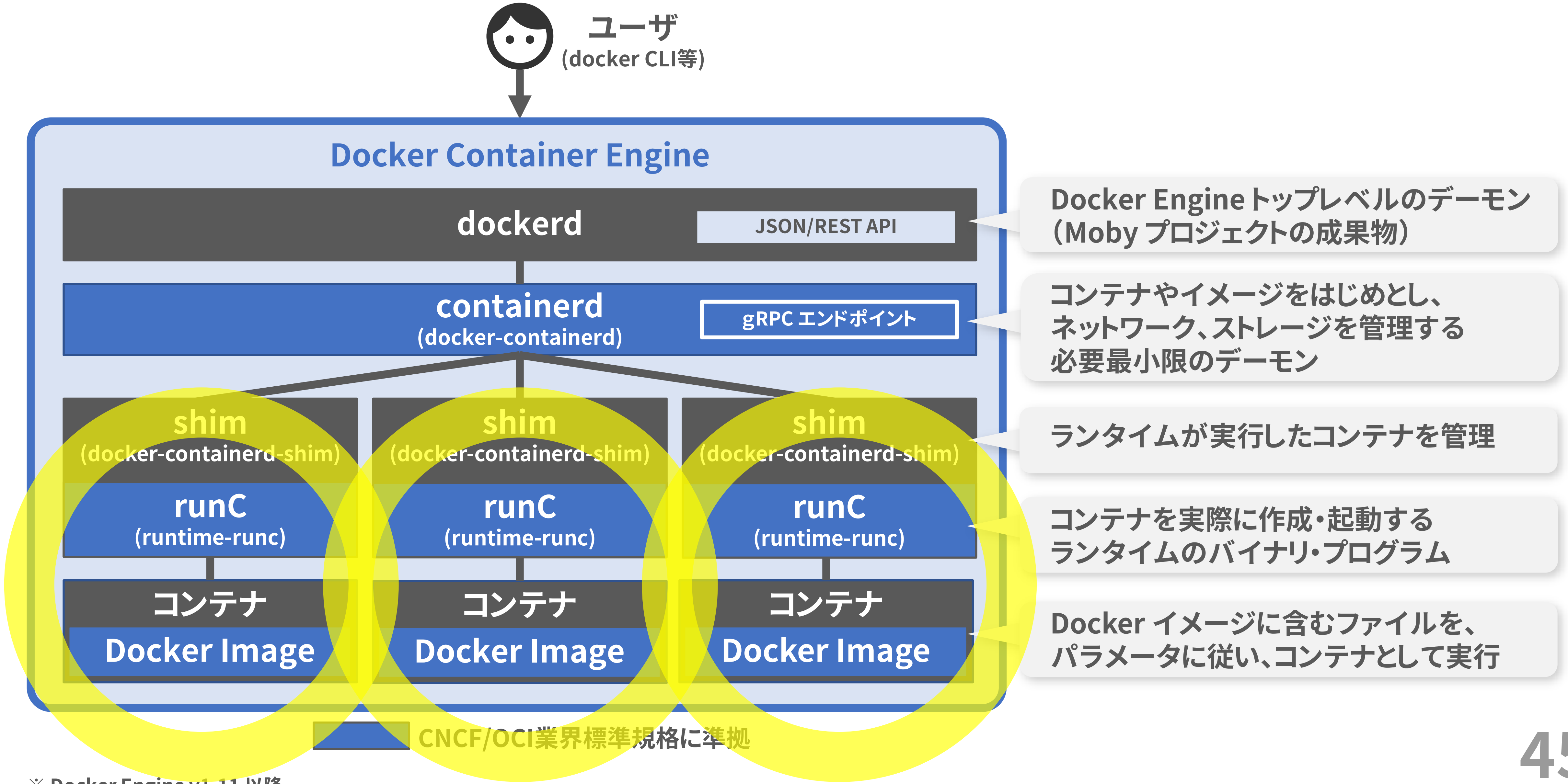
Dockerはサーバ・クライアント型モデル

- *docker*コマンド
  Linux, Mac OS X, Windows
- Kitematic (GUI)
  Mac OS X, Windows
- Docker Compose
- Docker Swarm



物理/仮想サーバ

Docker Engineのアーキテクチャ



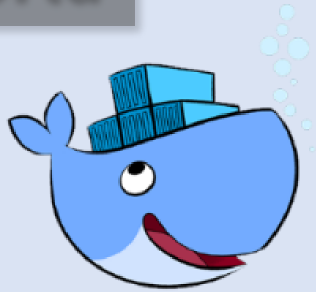
※ Docker Engine v1.11 以降

docker クライアント

docker エンジン

$ docker container run hello-world

run

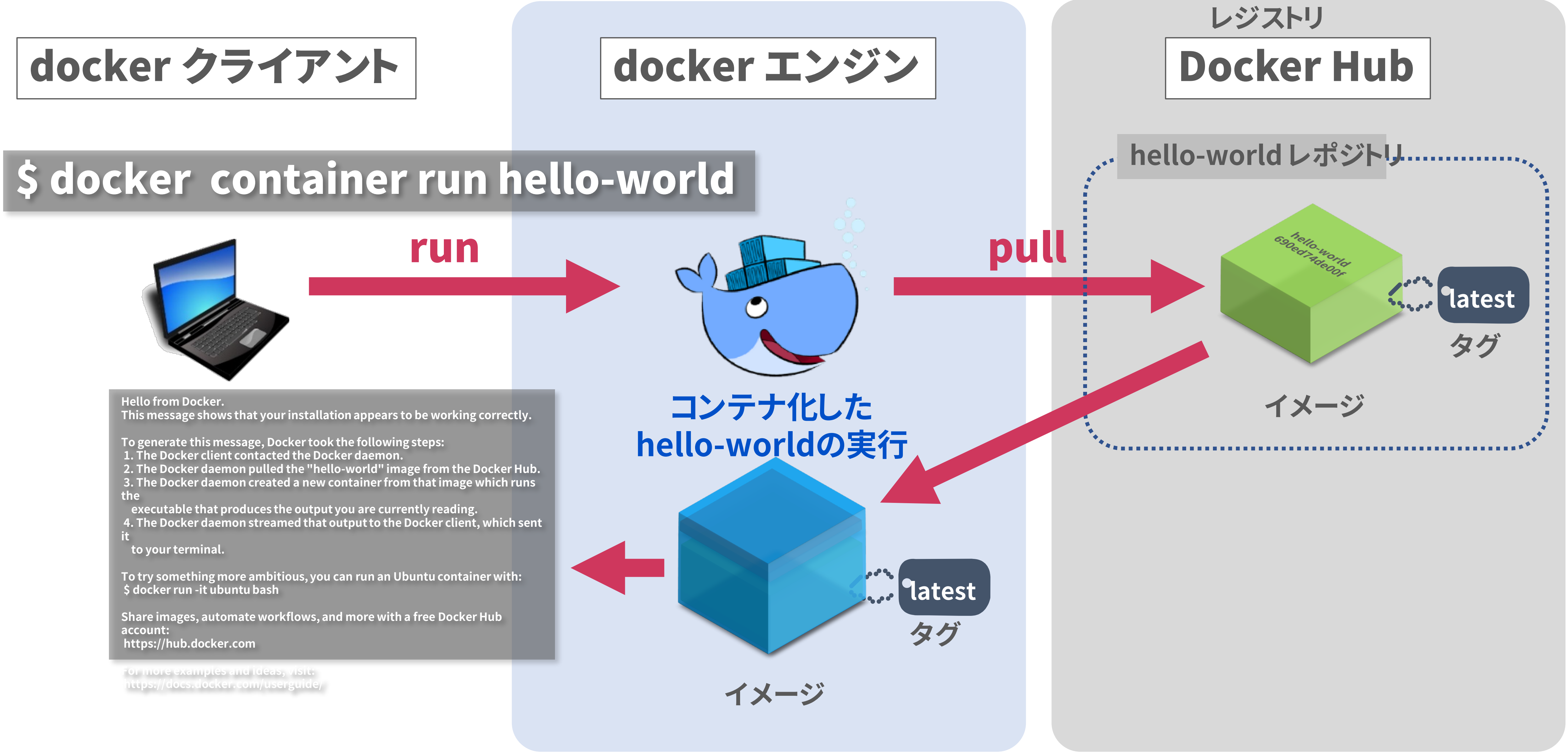


OSに対してdocker プログラムの
実行を伝える

Docker に対してコンテナに読み込む
イメージの情報を伝える

docker　container run　hello-world

Docker コンテナを作成・実行
するためのサブコマンド


docker クライアント
docker エンジン
レジストリ
Docker Hub
$ docker container run hello-world
run
pull
hello-world レポジトリ
hello-world
690ed74de00f
latest
タグ
イメージ
コンテナ化した
hello-worldの実行
Hello from Docker.
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
to your terminal.
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com
For more examples and ideas, visit:
https://docs.docker.com/userguide/
latest
タグ
イメージ

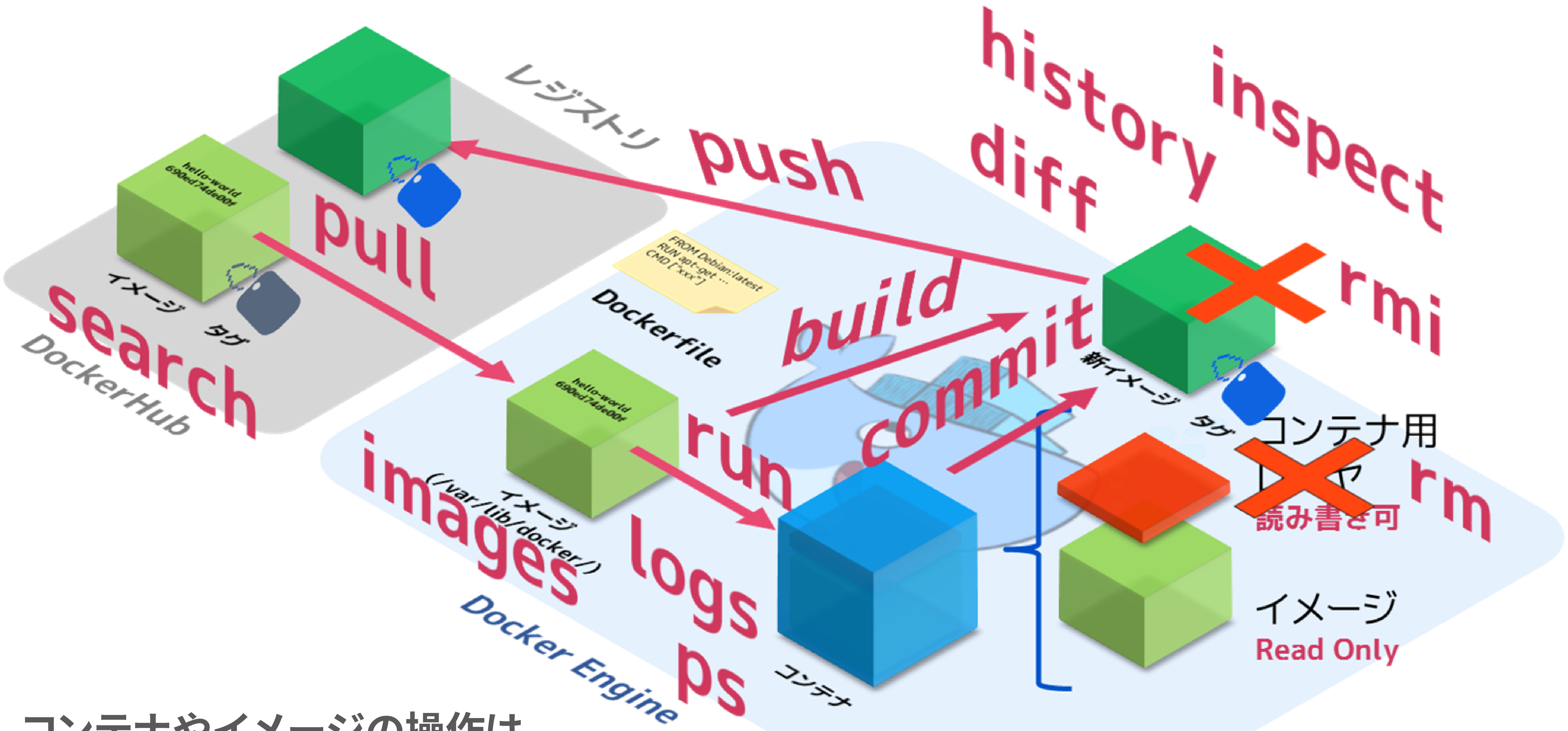
```
$ docker container run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
03f4658f8b78: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:8be990ef2aeb16dbcb9271ddfe2610fa6658d13f6dfb8bc72074cc1ca36966a7
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
(省略)
Share images, automate workflows, and more with a free Docker Hub account:
 https://hub.docker.com

For more examples and ideas, visit:
 https://docs.docker.com/userguide/
```

hello-worldイメージを使ったコンテナの実行(run)を命令

ローカルにhello-worldイメージがない

公式イメージlibraryにある”hello-world”を取得

最新の”hello-world:latest”イメージを取得完了



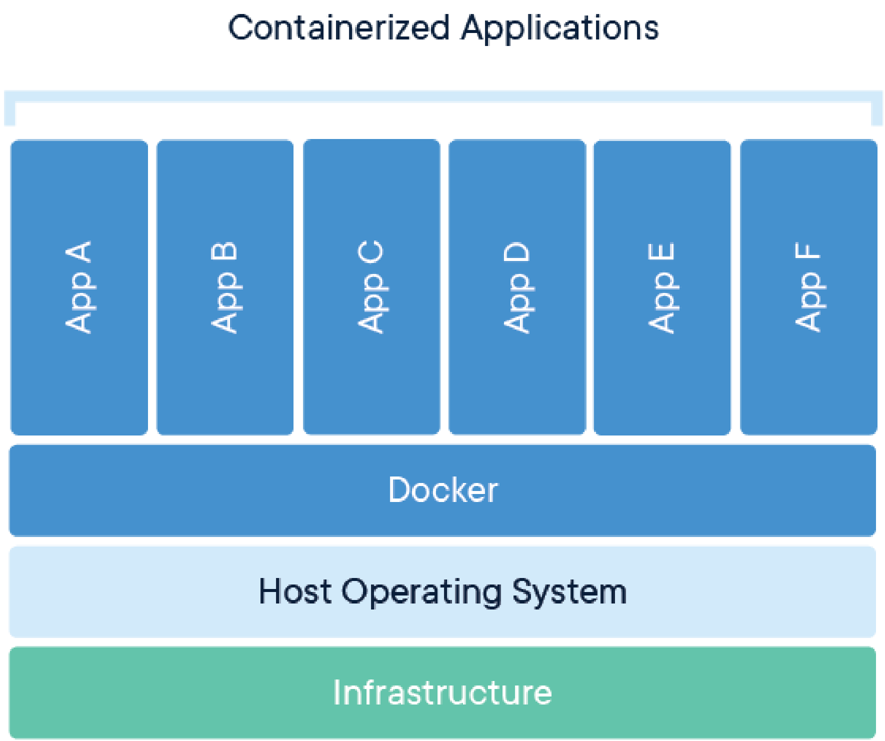
コンテナやイメージの操作は
コンテナIDやコンテナ・イメージIDを
中心に操作

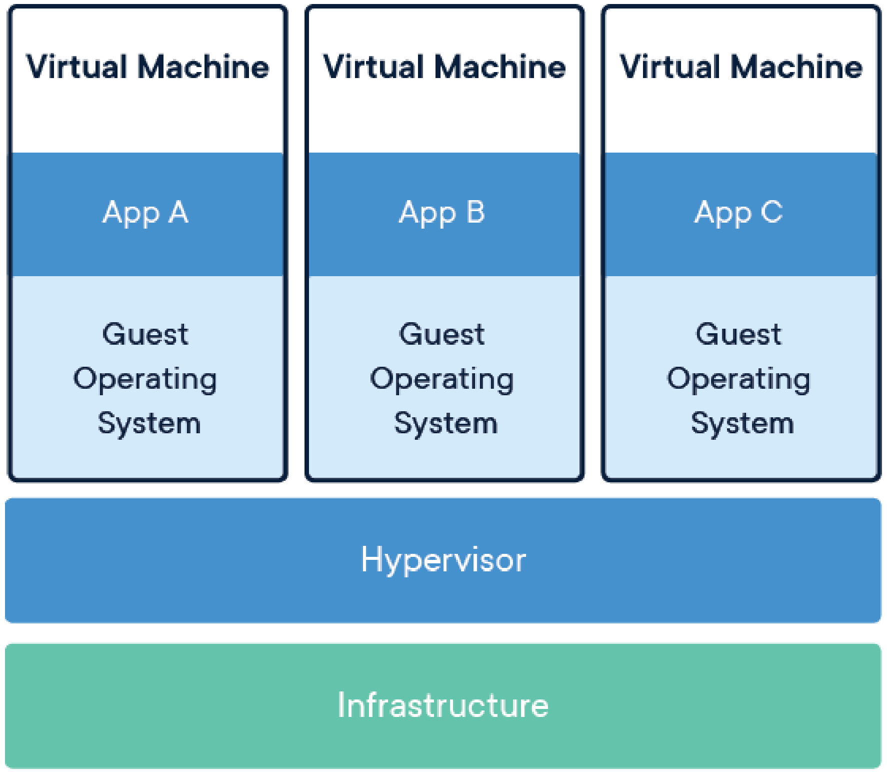
だがちょっとまって欲しい

仮想化とは違うの?

Docker≠コンテナ型仮想化

- アプリケーション・コンテナ
- システム・コンテナ(コンテナ型仮想化)

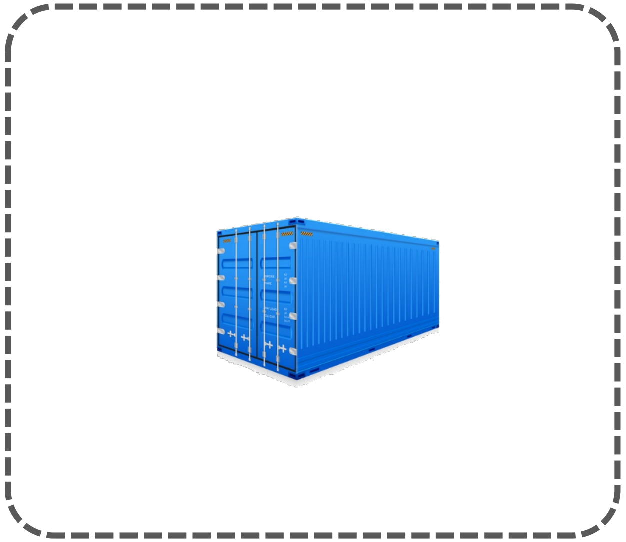




https://www.docker.com/resources/what-container network / volume

Dockerネットワークとボリューム

Dockerのネットワークも基本的にisolate(分離・孤立)



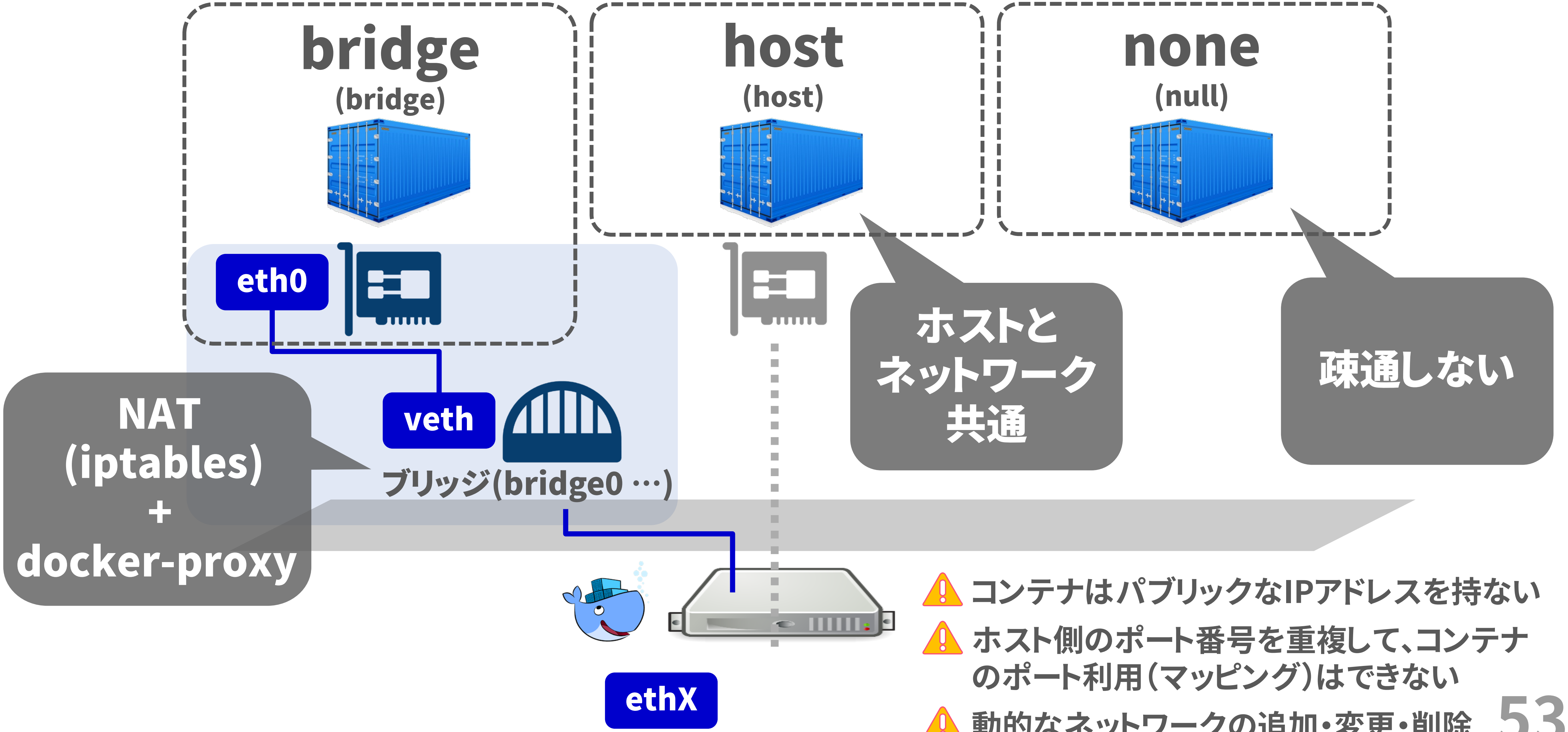
コンテナは、プロセス空間等だけでなく、ネットワークも分離可能

docker network create –subnet 192.168.0.0/24

コンテナ(状態のプロセス)はデフォルトではポートを開かないため、利用時は起動前に明示する必要

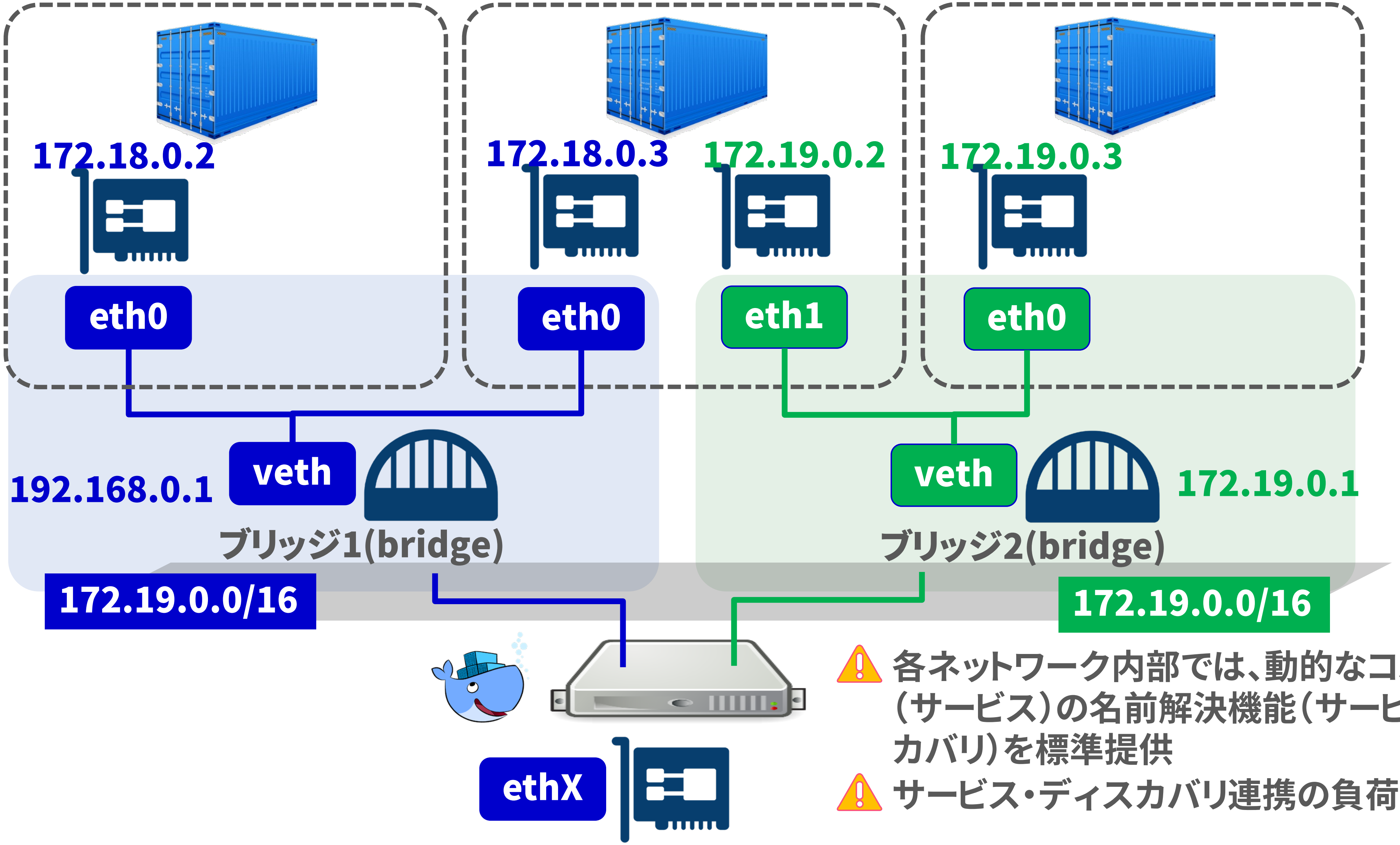
docker run -p [host:container]
docker run -P

3つのネットワーク・モデル



- ⚠ コンテナはパブリックなIPアドレスを持たない
- ⚠ ホスト側のポート番号を重複して、コンテナのポート利用(マッピング)はできない
- ⚠ 動的なネットワークの追加・変更・削除

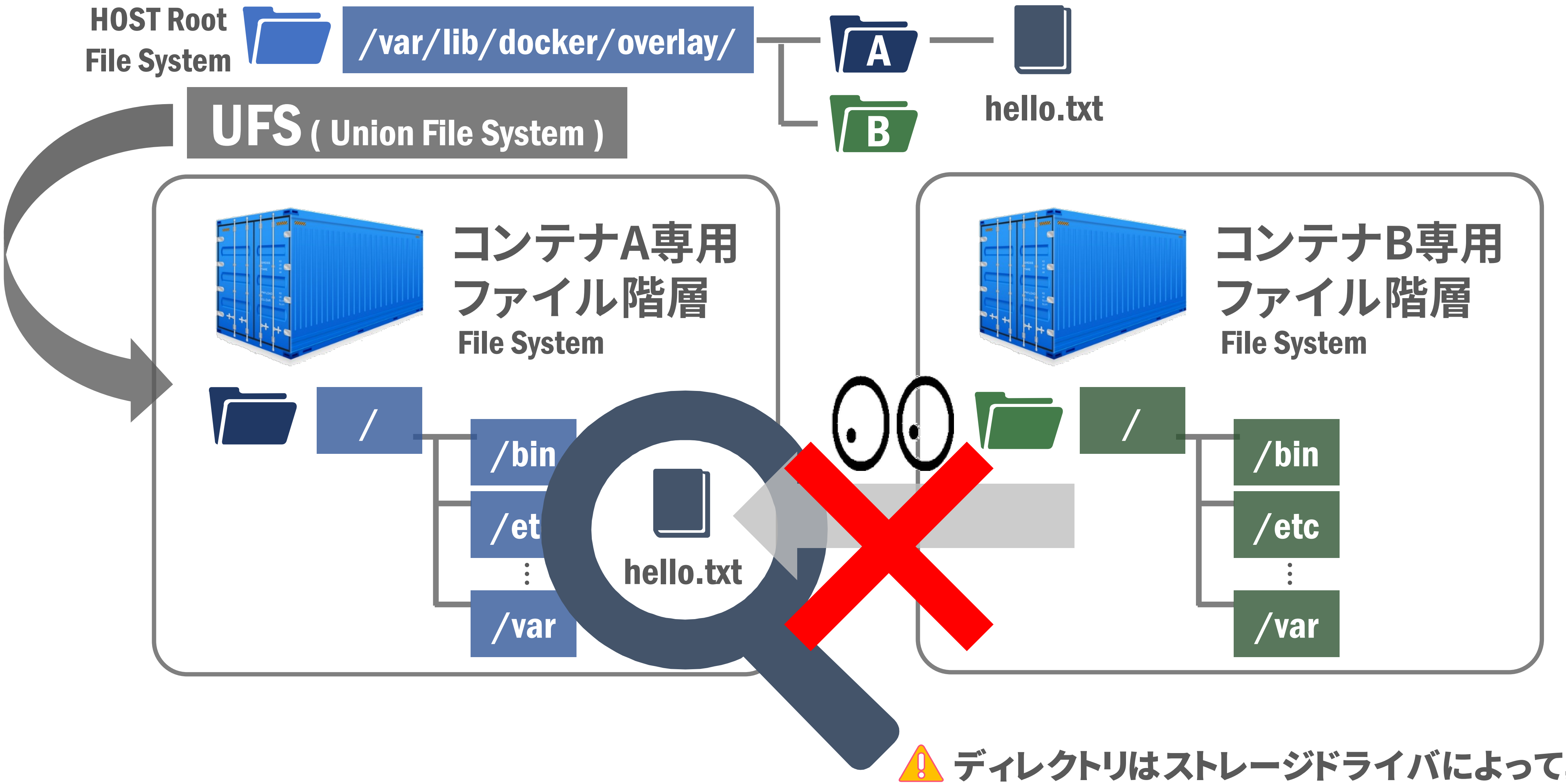
コンテナは複数のネットワーク(ブリッジ)に接続できる



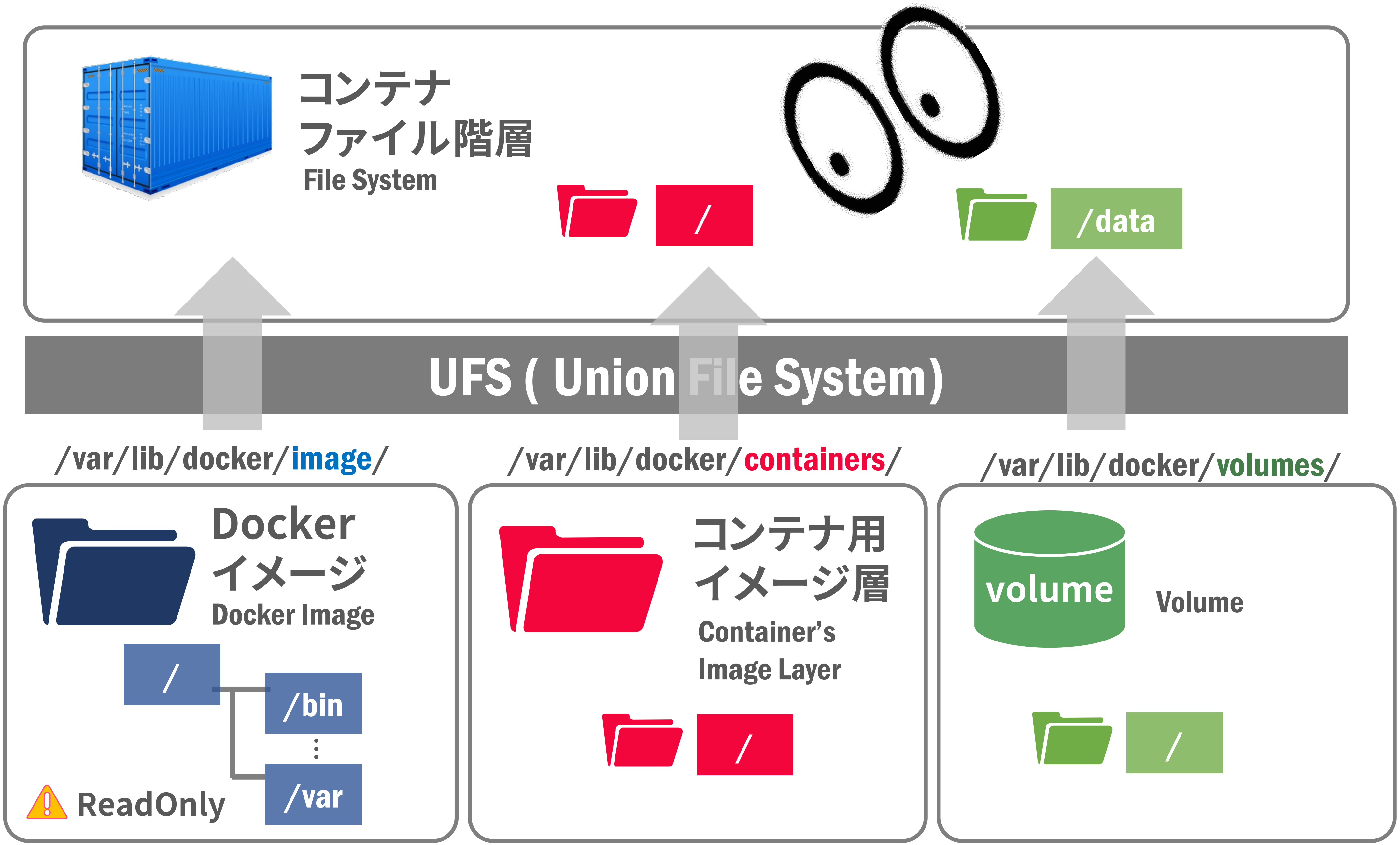
⚠ 各ネットワーク内部では、動的なコンテナ名(サービス)の名前解決機能(サービス・ディスカバリ)を標準提供

⚠ サービス・ディスカバリ連携の負荷分散

コンテナとファイルシステムのisolate(隔離)



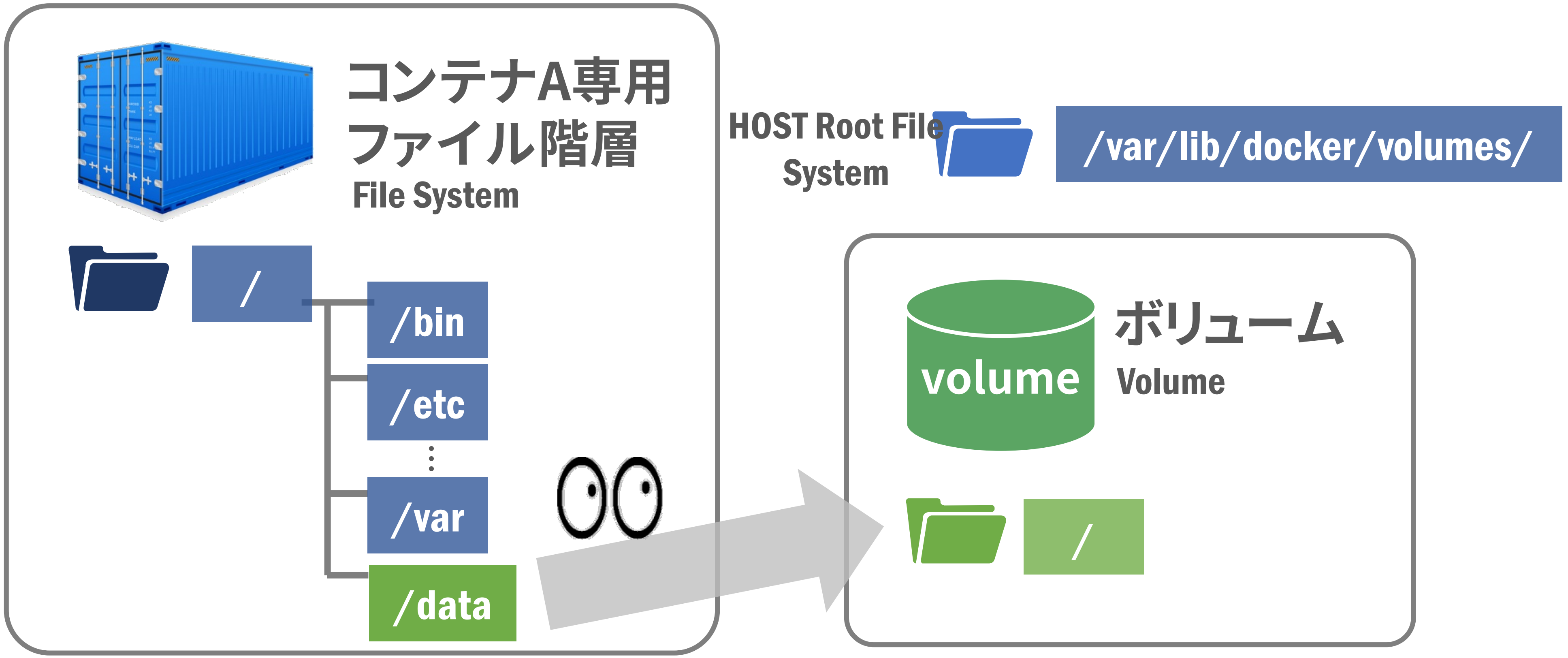
⚠ ディレクトリはストレージドライバによって異なる



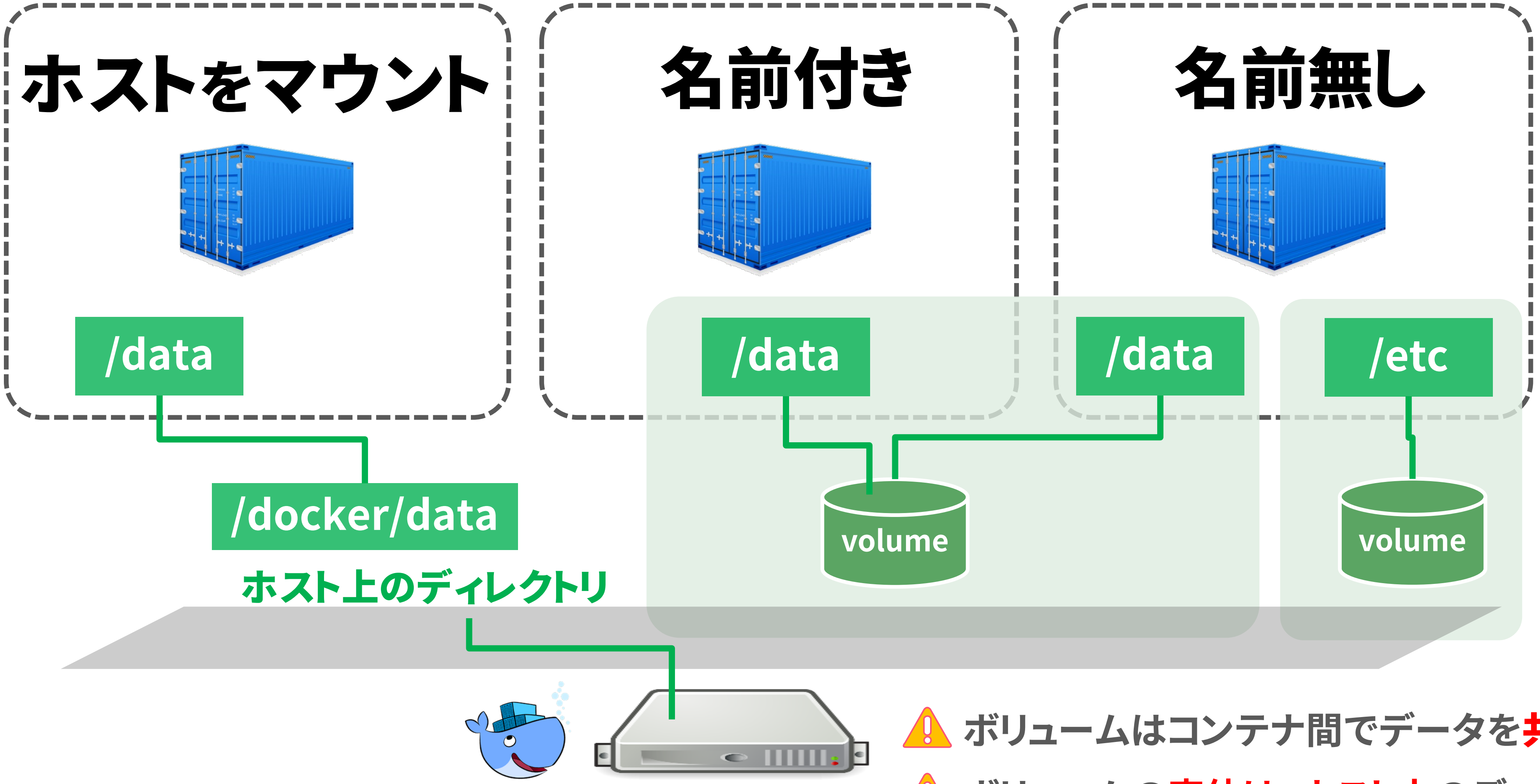
コンテナ
ファイル階層
File System
/
/data
UFS (Union File System)
/var/lib/docker/image/
/var/lib/docker/containers/
/var/lib/docker/volumes/
Docker
イメージ
Docker Image
/
/bin
/var
ReadOnly
コンテナ用
イメージ層
Container's
Image Layer
/
volume
Volume
/

ボリュームはコンテナ間で共有可能

コンテナからはUFSを通してデータ領域が見える
ストレージ・ドライバのオーバヘッドを受けない
複数のコンテナでボリュームを共有できる



ボリュームは3種類



⚠ ボリュームはコンテナ間でデータを**共有できる**

⚠ ボリュームの**実体は、ホスト上**のディレクトリ
/var/lib/docker/volumes

docker container run

ping のコンテナを
作ってみよう・使ってみよう

“ping” を実行するコンテナ

```
$ docker container run -it alpine
/ # cat /etc/issue
Welcome to Alpine Linux 3.8
Kernel ¥r on an ¥m (¥l)
/ # ping -c 3 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=120 time=1.124 ms
64 bytes from 8.8.8.8: seq=1 ttl=120 time=1.045 ms
64 bytes from 8.8.8.8: seq=2 ttl=120 time=0.979 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.979/1.049/1.124 ms

/ # exit
```

= docker run alpine ¥
ping -c 3 8.8.8.8

※1行で

foreground　　background
attach と detach

$ docker run alpine ping -c 100 8.8.8.8

PING 8.8.8.8 (8.8.8.8): 56 data bytes

64 bytes from 8.8.8.8: seq=0 ttl=120 time=7.260 ms

64 bytes from 8.8.8.8: seq=1 ttl=120 time=1.037 ms
(snip)
64 bytes from 8.8.8.8: seq=96 ttl=120 time=0.972 ms

64 bytes from 8.8.8.8: seq=97 ttl=120 time=1.032 ms

64 bytes from 8.8.8.8: seq=98 ttl=120 time=1.019 ms

64 bytes from 8.8.8.8: seq=99 ttl=120 time=1.052 ms

--- 8.8.8.8 ping statistics ---

100 packets transmitted, 100 packets received, 0% packet loss

round-trip min/avg/max = 0.911/1.116/6.986 ms

$ docker run -d alpine ping -c 100 8.8.8.8

de1dd8c69adebbe9df269bc38b57e3fe3549eb85140acb2c1d2a4eb51bc74160

docker container ps

$ docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
de1dd8c69ade	alpine	"ping -c 100 8.8.8.8"	43 seconds ago	Up 42 seconds		romantic_hawking

$ docker logs de

```
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=120 time=6.986 ms
64 bytes from 8.8.8.8: seq=1 ttl=120 time=0.956 ms
(snip)
64 bytes from 8.8.8.8: seq=98 ttl=120 time=1.075 ms
64 bytes from 8.8.8.8: seq=99 ttl=120 time=1.056 ms

--- 8.8.8.8 ping statistics ---
100 packets transmitted, 100 packets received, 0% packet loss
round-trip min/avg/max = 0.866/1.106/7.671 ms
```

docker container ps

```
$ docker ps
CONTAINER ID      IMAGE           COMMAND            CREATED        STATUS
```

```
$ docker ps –al
CONTAINER ID      IMAGE           COMMAND            CREATED        STATUS              PORTS
NAMES
de1dd8c69ade      alpine          "ping -c 100 8.8.8.8"   4 minutes ago      Exited (0) 2 minutes ago
romantic_hawking
```

```
$ docker rm ${docker ps -alq}
```

docker container exec

```
$docker run -d alpine ping -c 100 8.8.8.8

$docker exec -it $(docker ps -alq) /bin/sh
/ # ps ax
PID   USER     TIME  COMMAND
    1 root      0:00 ping -c 100 8.8.8.8
    6 root      0:00 /bin/sh
   11 root      0:00 ps ax
/ # pstree
ping
/ # exit
```

Dockerfile

```
FROM alpine:latest
ENTRYPOINT ["ping","-c","3"]
```

docker image build

```
# docker image build -t myping:1.0 .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM alpine:latest
 ---> 11cd0b38bc3c
Step 2/2 : ENTRYPOINT ["ping","-c","3"]
 ---> Running in bd71f7c0d243
Removing intermediate container bd71f7c0d243
 ---> b8c3422e54d1
Successfully built b8c3422e54d1
Successfully tagged myping:1.0
$ docker image ls
REPOSITORY        TAG               IMAGE ID          CREATED           SIZE
myping            1.0               485a295c402e      28 seconds ago    4.41MB
```

docker container run

```
$ docker run -it  myping:1.0 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=120 time=8.663 ms
64 bytes from 8.8.8.8: seq=1 ttl=120 time=1.004 ms
64 bytes from 8.8.8.8: seq=2 ttl=120 time=0.960 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.960/3.542/8.663 ms

$ docker run -it  myping:1.0
BusyBox v1.28.4 (2018-05-30 10:45:57 UTC) multi-call binary.

Usage: ping [OPTIONS] HOST

Send ICMP ECHO_REQUEST packets to network hosts
(snip)
```

Dockerfile

```
FROM alpine:latest
ENTRYPOINT ["ping","-c","3"]
```

Dockerfile

```
FROM alpine:latest
ENTRYPOINT ["ping","-c","3"]
CMD ["8.8.8.8"]
```

```
$ docker image build -t myping:1.1 .
Sending build context to Docker daemon  2.048kB
Step 1/3 : FROM alpine:latest
 ---> 11cd0b38bc3c
Step 2/3 : ENTRYPOINT ["ping","-c","3"]
 ---> Using cache
 ---> 2e704637972a
Step 3/3 : CMD ["8.8.8.8"]
 ---> Running in 61c8e7b682a4
Removing intermediate container 61c8e7b682a4
 ---> 0acf32d5be78
Successfully built 0acf32d5be78
Successfully tagged myping:1.1
```

docker container run

```
$ docker run -it  myping:1.1
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: seq=0 ttl=120 time=7.684 ms
64 bytes from 8.8.8.8: seq=1 ttl=120 time=1.153 ms
64 bytes from 8.8.8.8: seq=2 ttl=120 time=0.994 ms

--- 8.8.8.8 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.994/3.277/7.684 ms

$ docker run -it  myping:1.1 8.8.4.4
PING 8.8.4.4 (8.8.4.4): 56 data bytes
64 bytes from 8.8.4.4: seq=0 ttl=120 time=1.277 ms
64 bytes from 8.8.4.4: seq=1 ttl=120 time=1.252 ms
64 bytes from 8.8.4.4: seq=2 ttl=120 time=1.270 ms

--- 8.8.4.4 ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 1.252/1.266/1.277 ms
```

Dockerfile

```
FROM alpine:latest
ENTRYPOINT ["ping","-c","3"]
CMD ["8.8.8.8"]
```

docker network

```
$ docker network create --subnet 192.168.0.1/24 mynet
44dd2cdd6120c5265ecacfece5bd66077de1e6ab67f453601b8058eb0f65757f

$ docker network ls
NETWORK ID     NAME     DRIVER   SCOPE
df54d986479d   bridge   bridge   local
11dc9b54f1cc   host     host     local
44dd2cdd6120   mynet    bridge   local
a786e4ae4841   none     null     local
```

```
$ docker run -it --net=mynet alpine
/ # ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
53: eth0@if54: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue state UP
    link/ether 02:42:c0:a8:00:02 brd ff:ff:ff:ff:ff:ff
    inet 192.168.0.2/24 brd 192.168.0.255 scope global eth0
       valid_lft forever preferred_lft forever
```

Port mapping

- 「docker run -d -P nginx」は自動的にマッピング

ホスト側　コンテナ側

- 「docker run -d -p 8080:80 nginx」は手動

$ docker ps

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
d11fc51679e2	nginx	"nginx -g 'daemon of…"	18 seconds ago	Up 16 seconds	0.0.0.0:32773->80/tcp	focused_heisenberg
33ce5561ec1a	nginx	"nginx -g 'daemon of…"	19 seconds ago	Up 17 seconds	0.0.0.0:32772->80/tcp	gallant_ramanujan
a00d140e2500	nginx	"nginx -g 'daemon of…"	19 seconds ago	Up 18 seconds	0.0.0.0:32771->80/tcp	adoring_euclid
5e0f1c10b6f7	nginx	"nginx -g 'daemon of…"	20 seconds ago	Up 19 seconds	0.0.0.0:32770->80/tcp	flamboyant_meitner
cde6f341cacf	nginx	"nginx -g 'daemon of…"	26 seconds ago	Up 25 seconds	0.0.0.0:32769->80/tcp	cranky_rosalind

nmap -sT -O 127.0.0.1

volume

```
$ docker run -it alpine
/ # date
Sat Aug 25 02:36:47 UTC 2018
```

ホスト側　コンテナ側

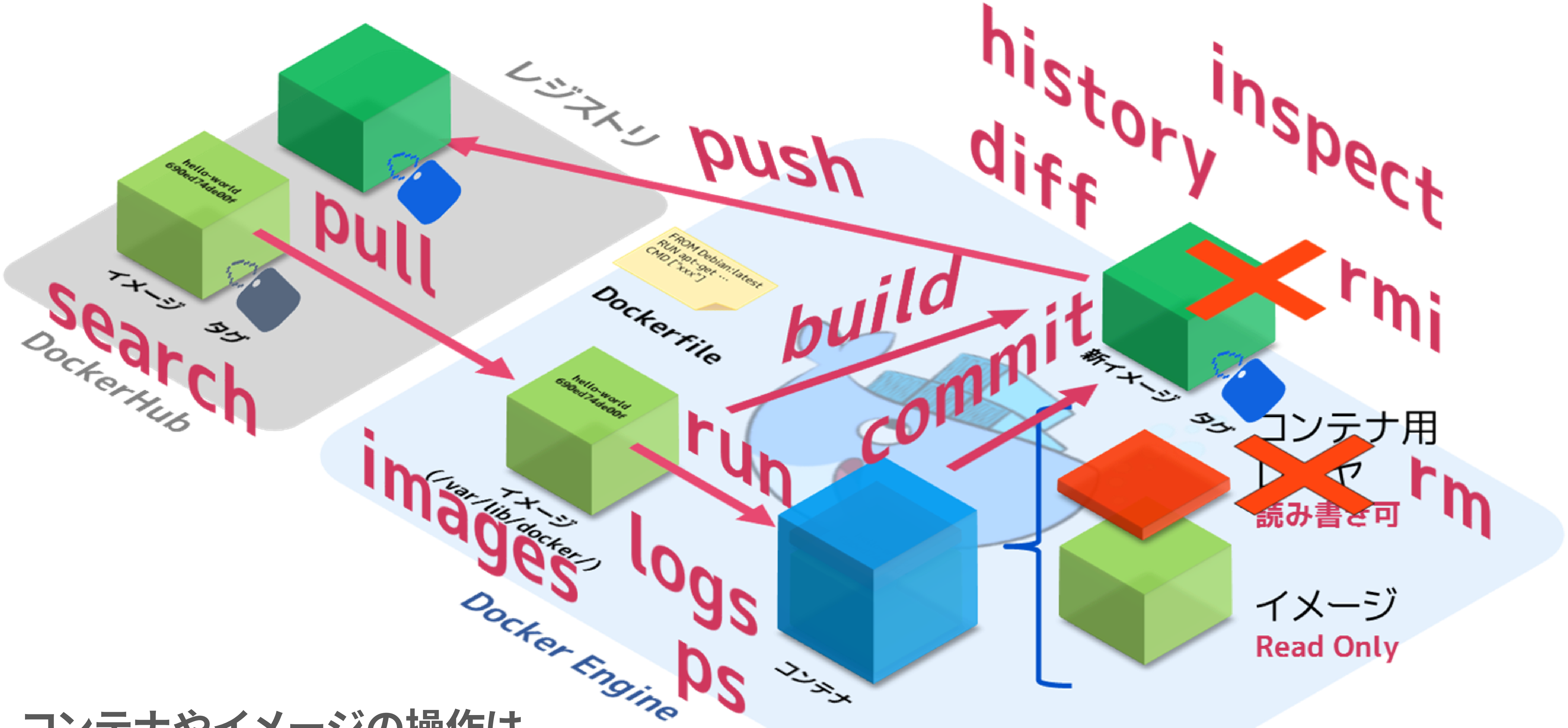
```
$ docker run -it -v /etc/localtime:/etc/localtime alpine
/ # date
Sat Aug 25 11:37:38 JST 2018
```

お掃除系の便利コマンド

- docker rm $(docker ps -aq)
- docker rmi $(docker images -aq)

- docker container prune
- docker image prune
- docker network prune
- docker volume prune

- docker system prune
- docker system df



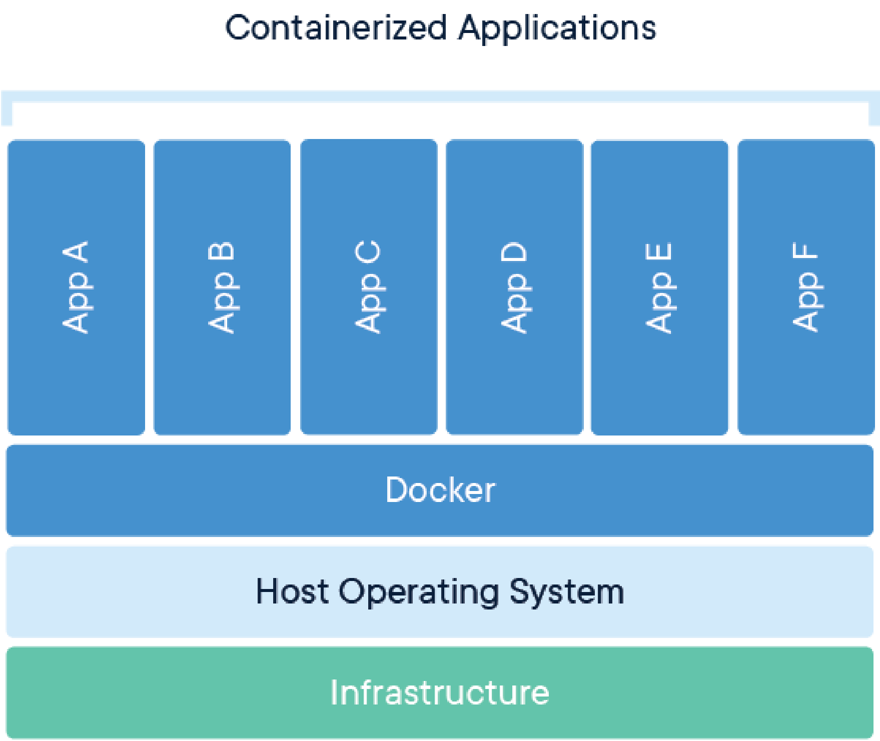
コンテナやイメージの操作は
コンテナIDやコンテナ・イメージIDを
中心に操作

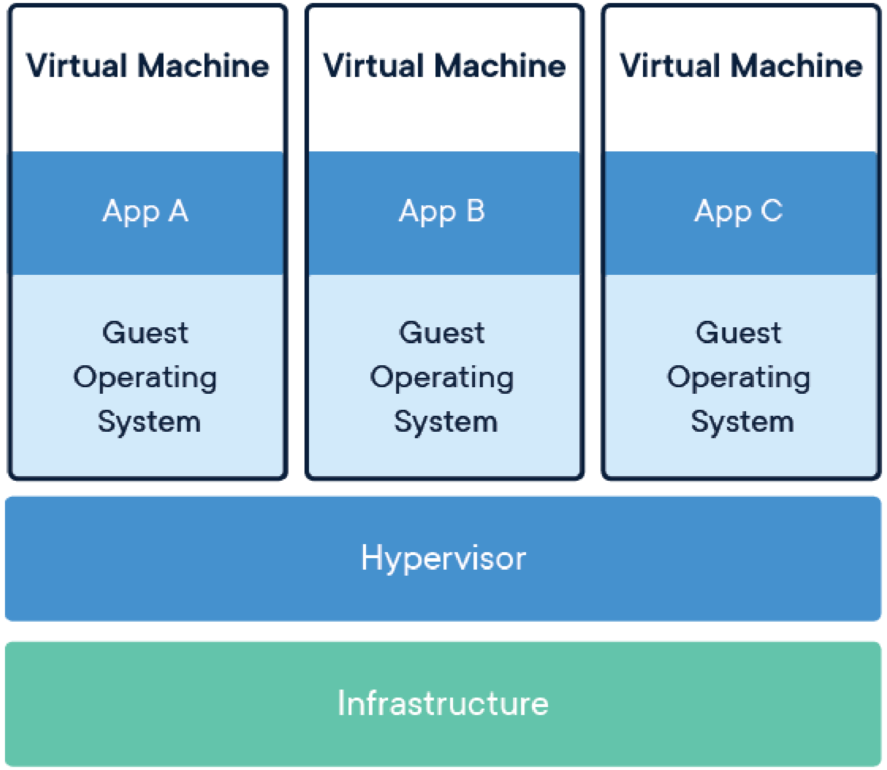
だがちょっとまって欲しい

仮想化とは違うの？

Docker≠コンテナ型仮想化

- アプリケーション・コンテナ
- システム・コンテナ（コンテナ型仮想化）





https://www.docker.com/resources/what-container

Orchestration

オーケストレーションと Kubernetes

Cloud Native 参照アーキテクチャ

kubernetes
containerd

Application Definition / Development	マイクロサービス・パターン
Orchestration & Management	分散オーケストレーションと管理
Runtime	コンテナ化
Provisioning	
Networking	インフラ ※ CNCFプロジェクトが定義する範囲外
Compute	
Storage	

【参考】https://github.com/cncf/presentations/blob/master/2016-software-circus/what-is-cloud-native/what-is-cloud-native.pdf

アプリケーションやサービスの
オーケストレーション
Orchestration

複数のホスト・システム上を横断するアプリケーションをスケール（拡大・縮小）できる機能
※コンテナに依存しない

設定ファイルをベースにサービスを定義・維持

- Kubernetes
- Docker Engine (swarm mode)
  + Docker Compose
- Rancher
- Nomad

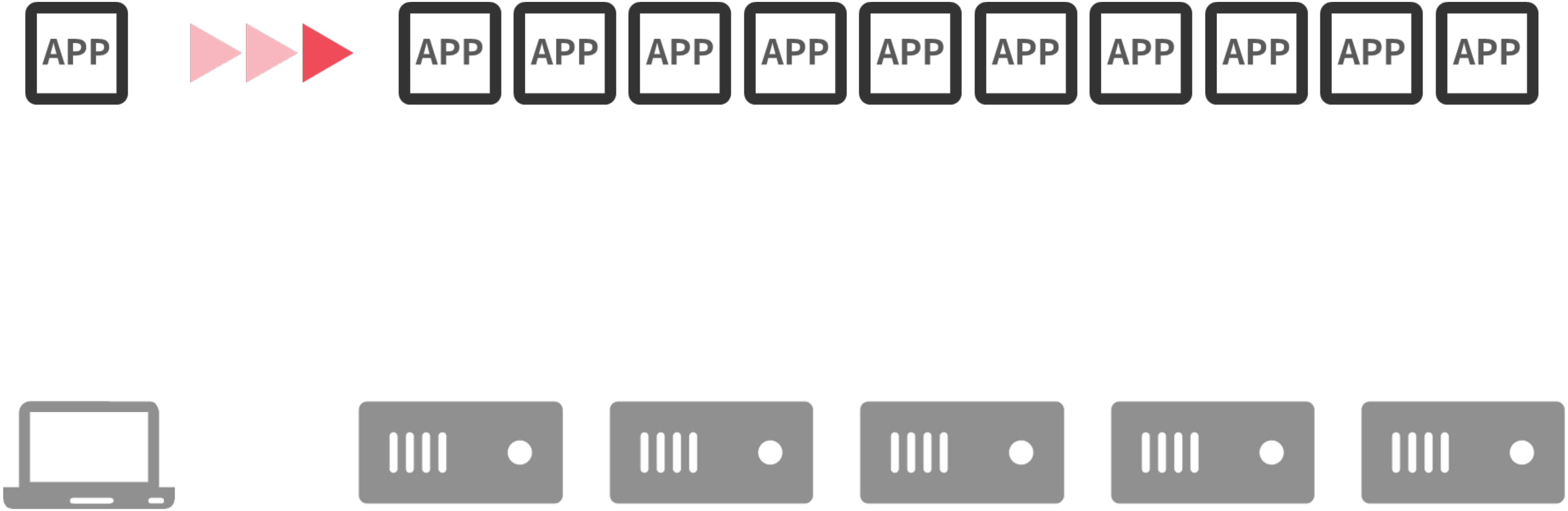
計算資源の抽象化

(主に(コンテナ)やジョブの
スケジューリング
Scheduling

- Marathon, chronos
- Docker swarm
- Deis
- fleet

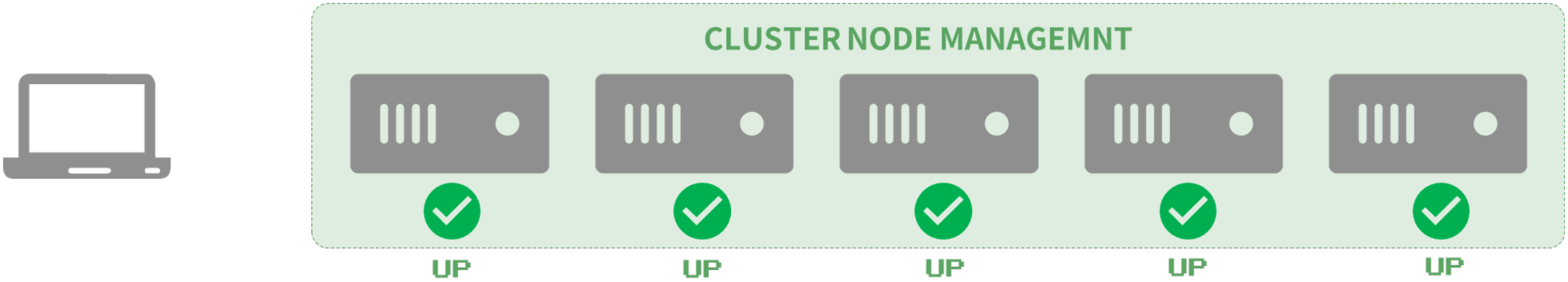
ホスト計算資源の
クラスタ管理
Cluster Management

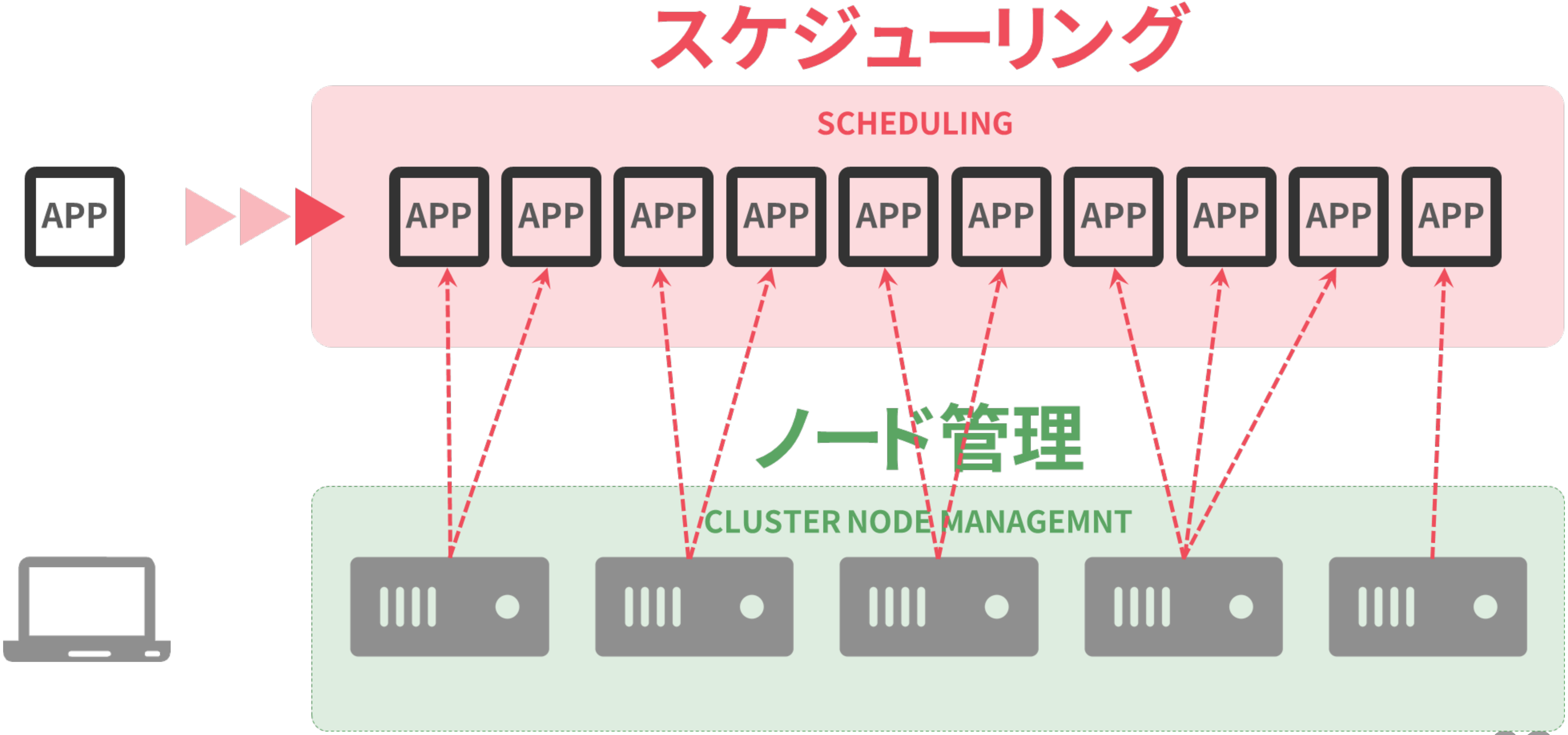
- Apache Mesos
- DCOS (Mesosphere)


APP
APP
APP
APP
APP
APP
APP
APP
APP
APP
APP

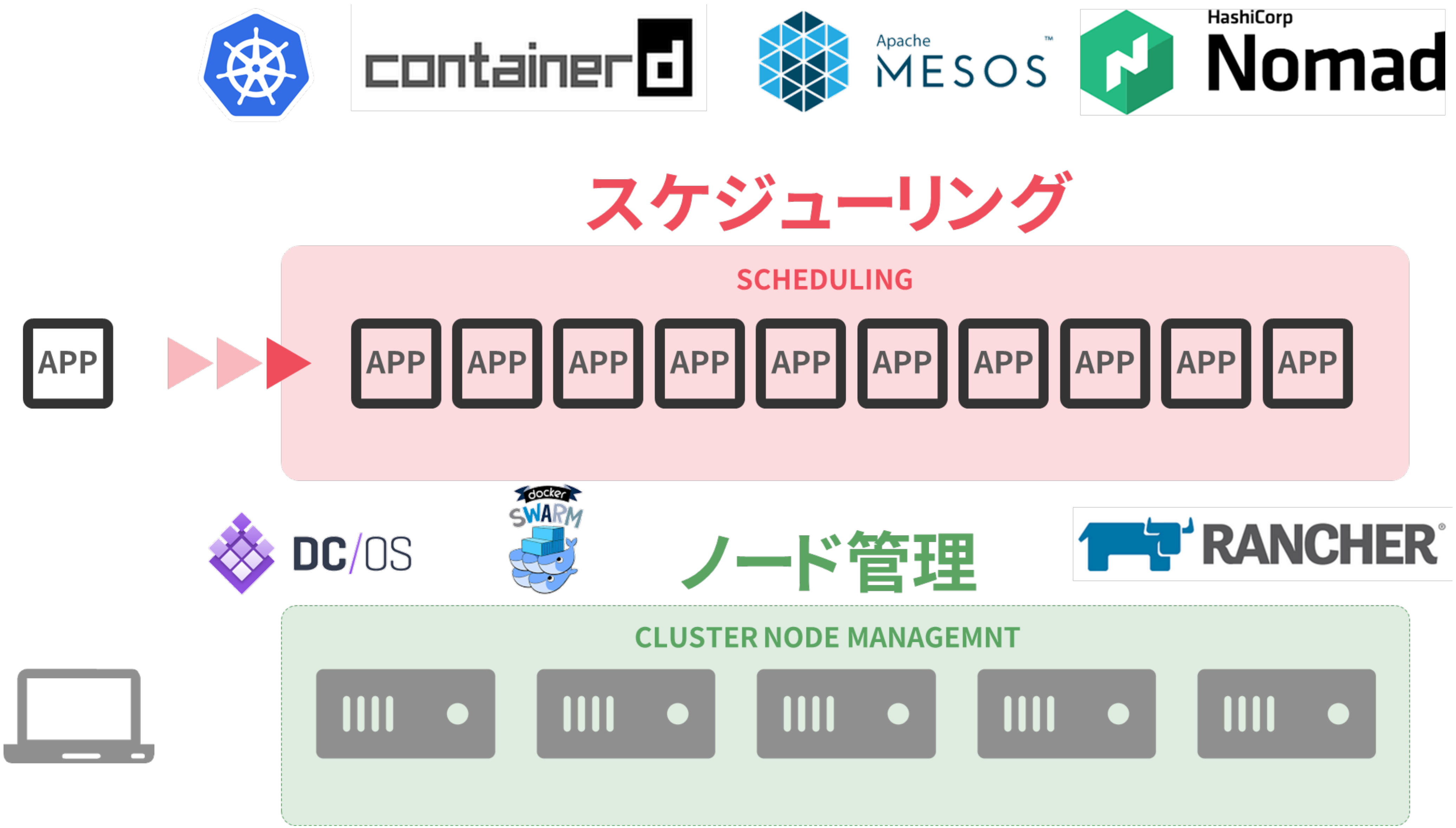

APP
APP
APP
APP
APP
APP
APP
APP
APP
APP
APP

ノード管理

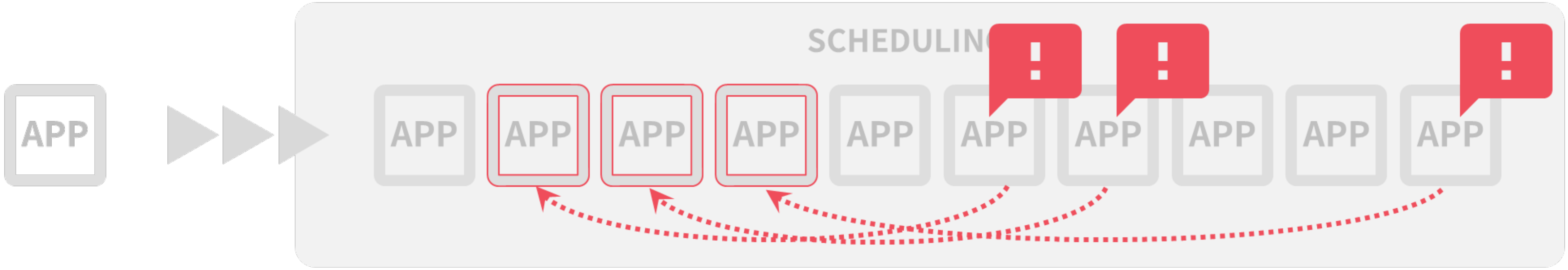

CLUSTER NODE MANAGEMNT
UP
UP
UP
UP
UP



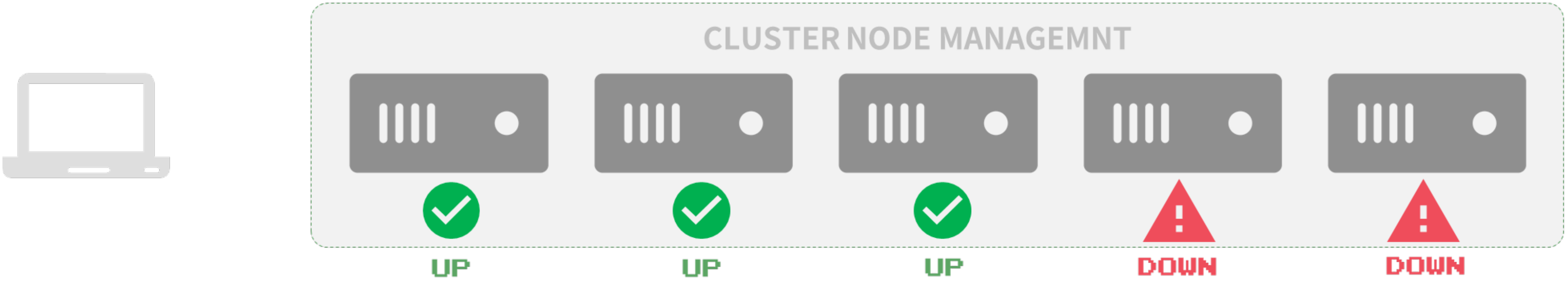
スケジューリング
SCHEDULING
APP
APP APP APP APP APP APP APP APP APP APP
ノード管理
CLUSTER NODE MANAGEMNT


containerd
Apache MESOS
HashiCorp Nomad
スケジューリング
SCHEDULING
APP
APP APP APP APP APP APP APP APP APP APP
DC/OS
docker SWARM
ノード管理
RANCHER
CLUSTER NODE MANAGEMNT

スケジューリング



ノード管理



アプリケーションやサービスの
オーケストレーション
Orchestration

複数のホスト・システム上を横断するアプリケーションをスケール（拡大・縮小）できる機能
※コンテナに依存しない

設定ファイルをベースにサービスを定義・維持

計算資源の抽象化

- Kubernetes
- Docker Engine (swarm mode)
  + Docker Compose
- Rancher
- Nomad

(主に(コンテナ)やジョブの
スケジューリング
Scheduling

- Marathon, chronos
- Docker swarm
- Deis
- fleet

ホスト計算資源の
クラスタ管理
Cluster Management

- Apache Mesos
- DCOS (Mesosphere)

従来のオーケストレーション

【参考】https://www.slideshare.net/Docker/container-orchestration-from-theory-to-practice/7

宣言型サービス・モデルのオーケストレーション

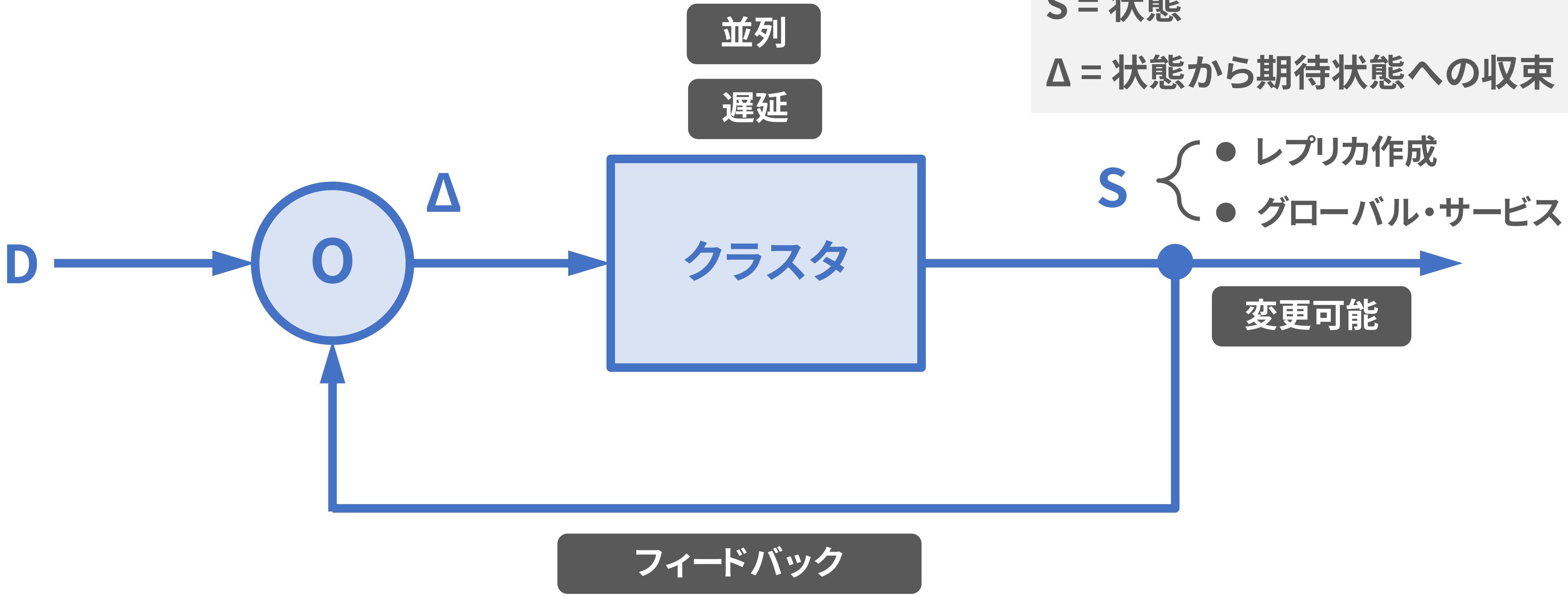
Declarative service model

D = 期待状態

O = オーケストレータ

S = 状態

Δ = 状態から期待状態への収束

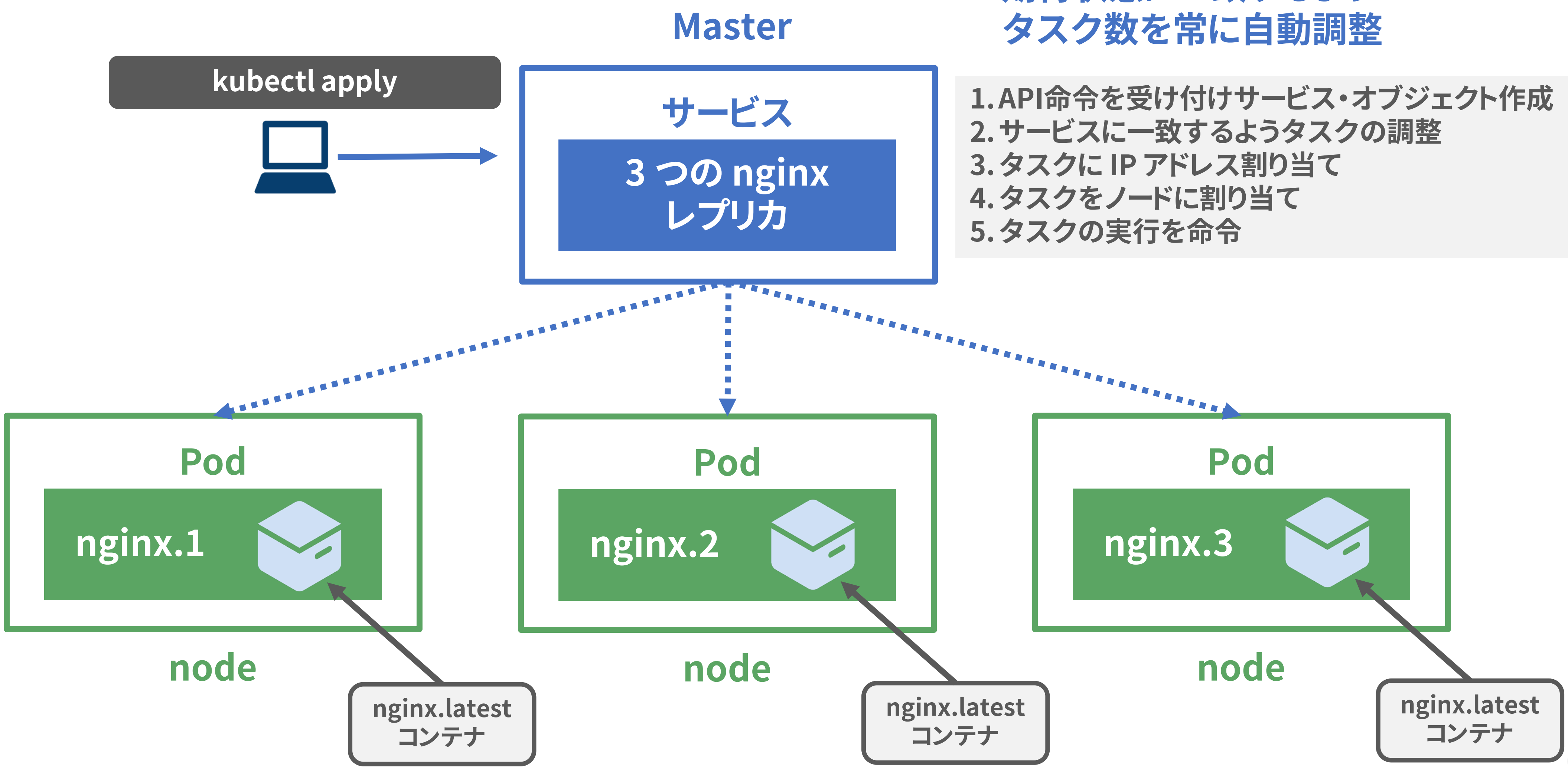


【参考】 https://www.slideshare.net/Docker/container-orchestration-from-theory-to-practice/7

サービスとタスクの概念

「コンテナ」の起動ではない

- 期待状態に一致するよう
  タスク数を常に自動調整



1. API命令を受け付けサービス・オブジェクト作成
2. サービスに一致するようタスクの調整
3. タスクに IP アドレス割り当て
4. タスクをノードに割り当て
5. タスクの実行を命令

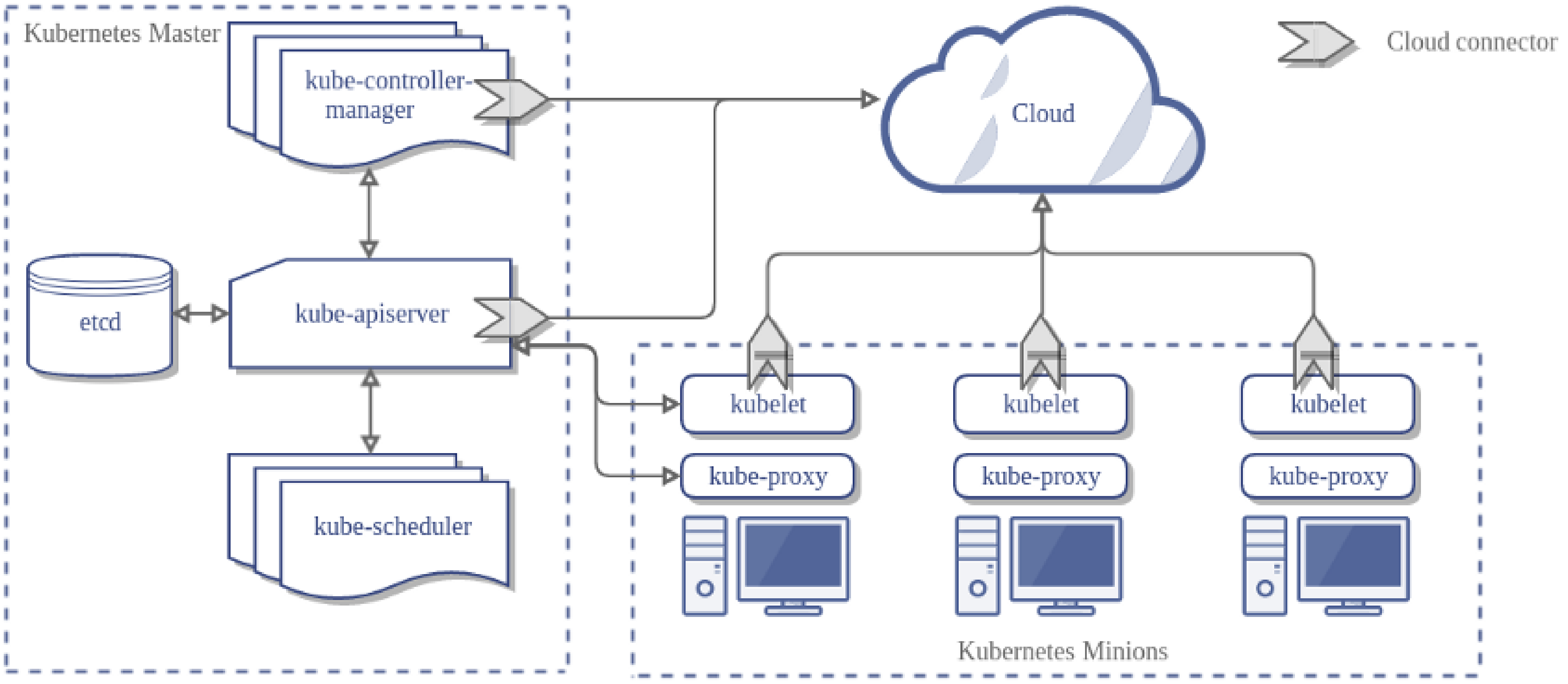
【参考】 https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#services-tasks-and-containers kubernetes

κυβερνήτης

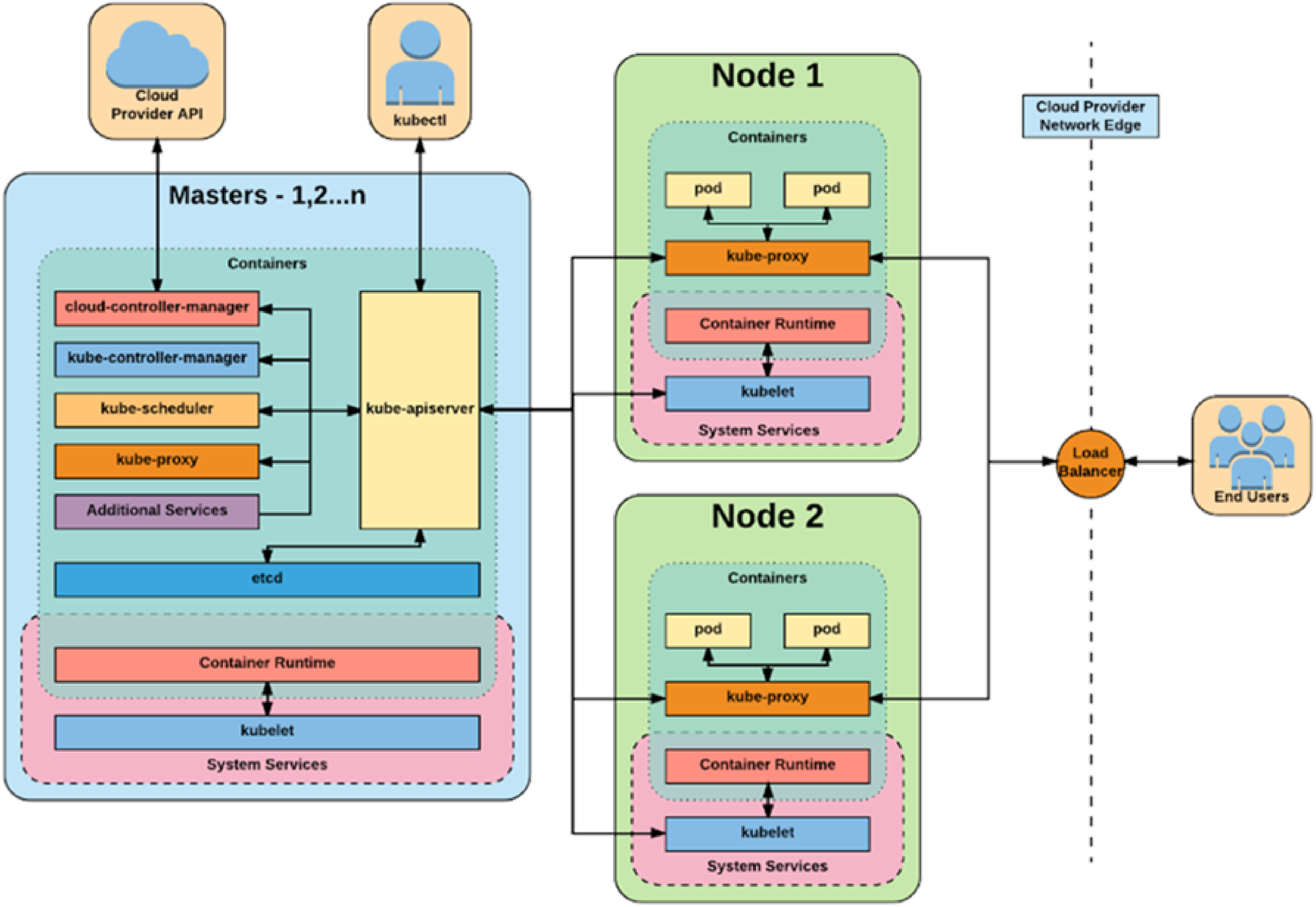
/ky.ber.nɛ̌ː.tɛːs/ → /ky.βerˈni.tis/ → /ci.verˈni.tis/

a captain, a steersman,
a pilot, a navigator

https://en.wiktionary.org/wiki/%CE%BA%CF%85%CE%B2%CE%B5%CF%81%CE%BD%CE%AE%CF%84%CE%B7%CF%82



Kubernetes Master
kube-controller-manager
etcd
kube-apiserver
kube-scheduler
Cloud
Cloud connector
kubelet
kube-proxy
kubelet
kube-proxy
kubelet
kube-proxy
Kubernetes Minions

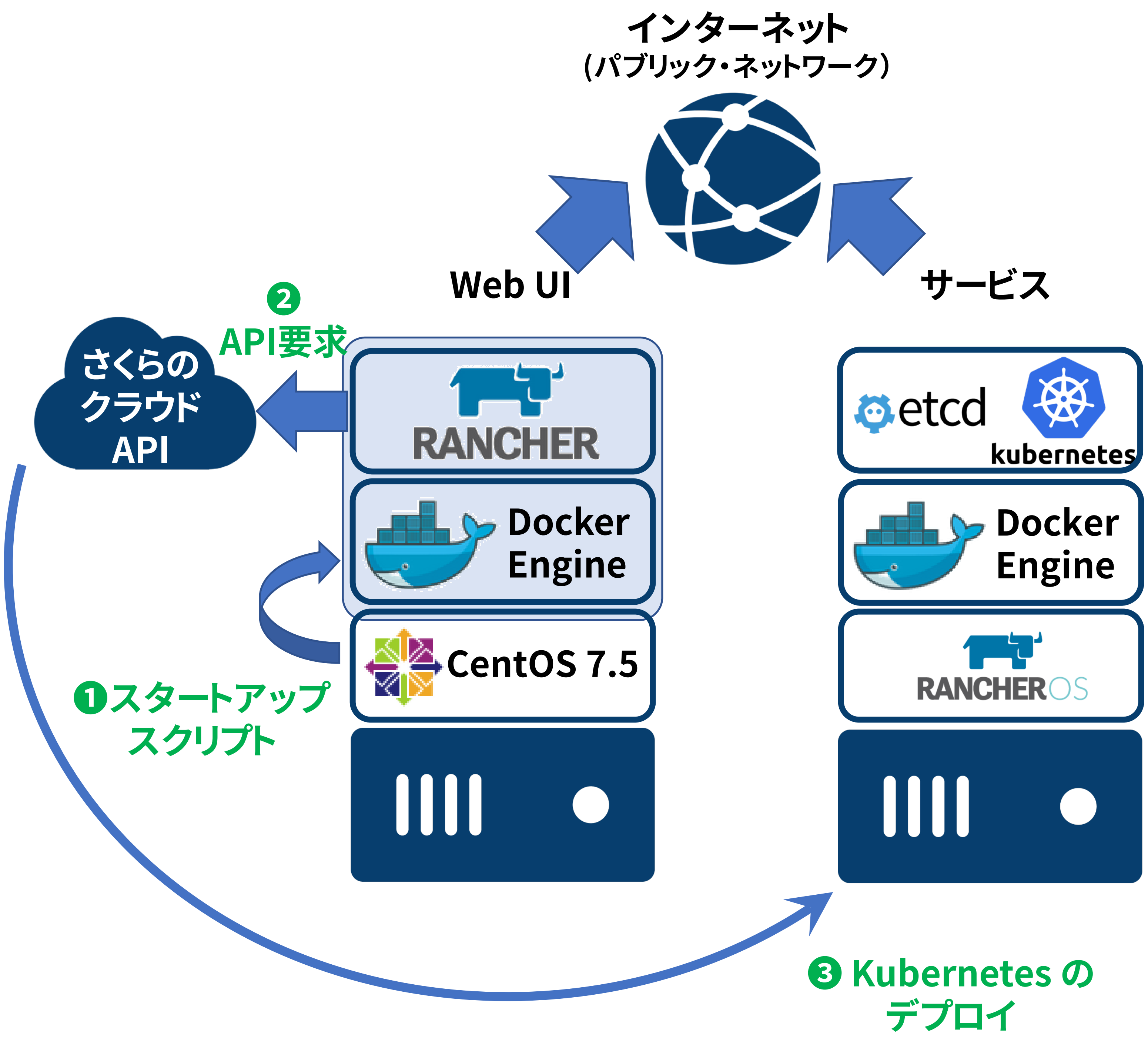


[Kubernetes - A Comprehensive Overview]() by Bob Killen (December 11, 2017)

Rancher

Rancher

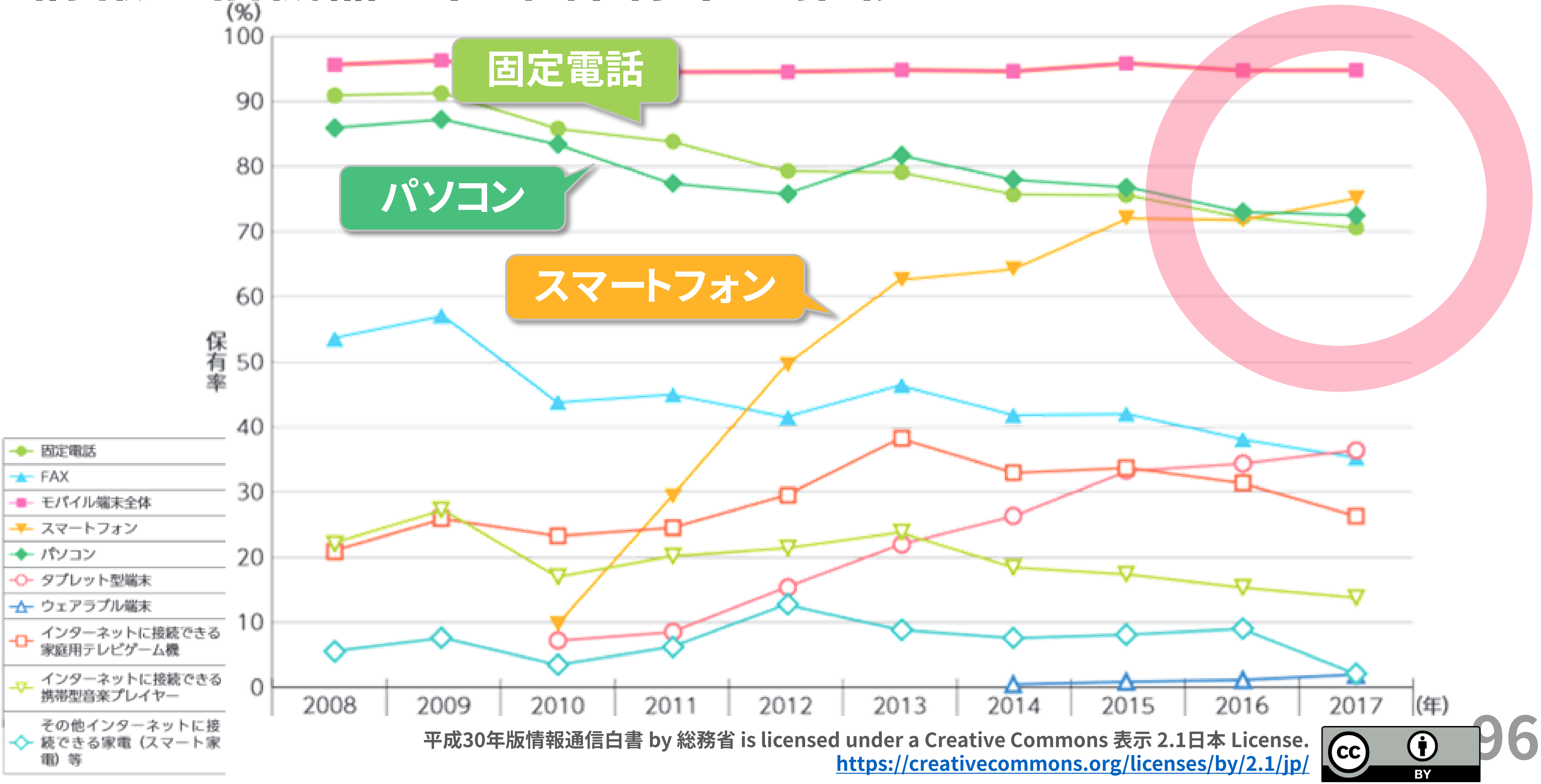
- kubernetes 構成要素やクラスタを統合管理
  - 複数のクラスタをまとめて管理(監視・ログの集約)
  - クラウドへのデプロイや、既存のクラスタの取り込み
  - CI/CD パイプライン
  - ユーザ認証・管理
  - GUI
- https://rancher.com
  - オープンソース
  - Rancher Labs 社、2014年設立
  - サポート体制あり



インターネット
(パブリック・ネットワーク)
Web UI
サービス
❷
API要求
さくらの
クラウド
API
RANCHER
Docker
Engine
CentOS 7.5
❶スタートアップ
スクリプト
etcd
kubernetes
Docker
Engine
RANCHEROS
❸ Kubernetes の
デプロイ

まとめ

情報通信機器の世帯保有率の推移



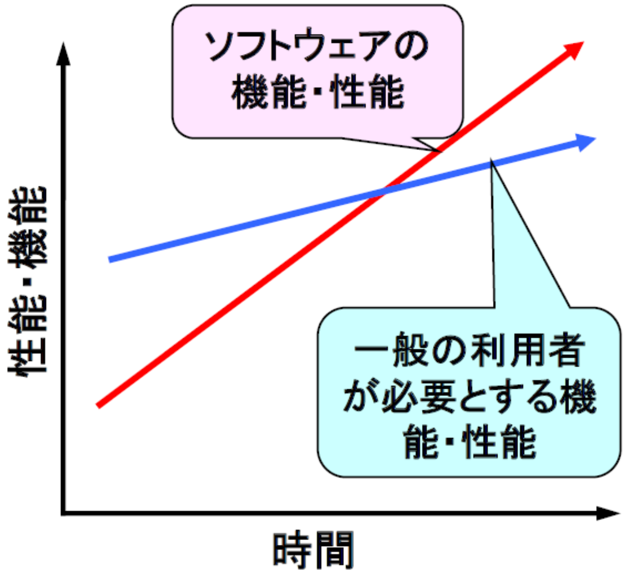
ソフトウェアやサービス開発の前提が変わる

- パッケージ販売モデルの終焉と、終わりなき開発・改善
- オーバーシューティング

オーバーシューティング現象

■ ソフトウェアの過剰性能、過剰機能、（過剰品質）

ソフトウェアの革新に限界はないが、
利用者のニーズを超えてしまっているのではないか



「パソコンの実利用年数が3年から5年になり、そして10年に伸び、ソフトのバージョンアップをしなくなる日が近づいている」
前川『ネットビジネス最前線』（1998）

（出典）ソフトウェアビジネスの未来、前川徹、2011年

キーワード

サービス・アプリケーション層

Robotic Process Automation
ロボティック・プロセス・オートメーション

コーディングが不要

- ロボティック・プロセス・オートメーション（Robotic Process Automation, RPA）とは、認知技術（ルールエンジン・機械学習・人工知能等）を活用した、主にホワイトカラー業務の効率化・自動化の取組みである。
- 人間の補完として業務を遂行できることから、仮想知的労働者（Digital Labor）とも言われている。また、デスクトップ作業のみに絞ったものをロボティック・デスクトップ・オートメーションと呼び、RPAと区別することもある。

計算資源の基盤・プラットフォーム層

Serverless
FaaS (Function as a Service)

サーバの管理が不要

- サーバレス・コンピューティングはアプリケーションを構築・実行するための概念を言及するもので、サーバ管理は扱わない。
→<u>アプリケーションの洗練されたデプロイ・モデル（方式）</u>
- サーバレス・コンピューティングとは、コードを実行するためにサーバを持たないのを意味するのではないし、運用エンジニアが不要になるのも意味しない。
→<u>開発者はビジネスロジックに応じたアプリケーションを書くのに集中でき、運用担当者はビジネスにクリティカルなタスクに集中できる。</u>

お持ち帰り
Key Takeaways

1 コンテナはサーバ仮想化技術の延長線上ではない

- 従来のハードウェア仮想化による計算資源の効率的な利用が目的ではなく「クラウド・ネイティブ」による更なる活用のため
- スマートフォンの普及は常時利用・継続開発が必要なシステムに
- 労働生産性向上は、開発や運用スタイルの変化も促す

https://cncf.io

2 コンテナはプロセスをisolate(分離)して動かす複数の技術

- アプリケーションの移植性を高める実装として広く普及した Docker
- namespace(名前空間)でプロセス空間やファイルシステムやネットワーク等を分ける技術と、cgroups(コントロール・グループ)でリソースの利用上限を指定
- コンテナ実行にはイメージが必要

https://docker.com

3 オーケストレーションは複数のコンテナ等の計算資源を束ねる

- クラスタを構成するためのノード管理と、スケジューリング(処理を実行するために必要なノードを選び出し、計画した後に実行)をする
- 期待状態(Desired State)を常に自動的に維持する仕組みを内蔵
- 継続してサービスを実行するために様々な機能を実装

https://kubernetes.io

Q&A

- 何か気になるところはありますか？
- https://slideshare.net/zembutsu
- Dockerドキュメント日本語訳
  http://docs.docker.jp
- 公式ドキュメント
  https://docs.docker.com