

DOCKER FUNDAMENTALS



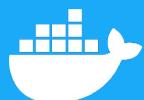
HOW WE TEACH

- Docker believes in learning by doing.
- The course is lab driven
- Work together!
- Ask questions at any time.



SESSION LOGISTICS

- 2 days duration
- Mostly exercises
- Regular breaks



ASSUMED KNOWLEDGE AND REQUIREMENTS

- Familiarity with Bash or Powershell
- Bash Cheat Sheet: <http://bit.ly/2mTQr8l>
- Powershell Cheat Sheet: <https://bit.ly/2EPHxze>



YOUR LAB ENVIRONMENT

- You have been given several instances for use in exercises.
- Ask instructor for username and password if you don't have them already.



COURSE LEARNING OBJECTIVES

By the end of this course, learners will be able to:

- Assess the advantages of a containerized software development & deployment
- Use Docker engine features necessary for running containerized applications
- Utilize Swarm and Kubernetes orchestrators to deploy, maintain, and scale a distributed application



INTRODUCING DOCKER



WHAT WE WANT

Ideal software should:

- be modular and flexible (devs)
- be easy to migrate (devops)
- be easy to scale, monitor and lifecycle (ops)
- mitigate vulnerabilities (security)
- run cheap (business)



WITHOUT CONTAINERIZATION

DB + API + Frontend

DB Config + API Config + Frontend Config

DB Dependencies + API Dependencies + Frontend Dependencies

Base Filesystem

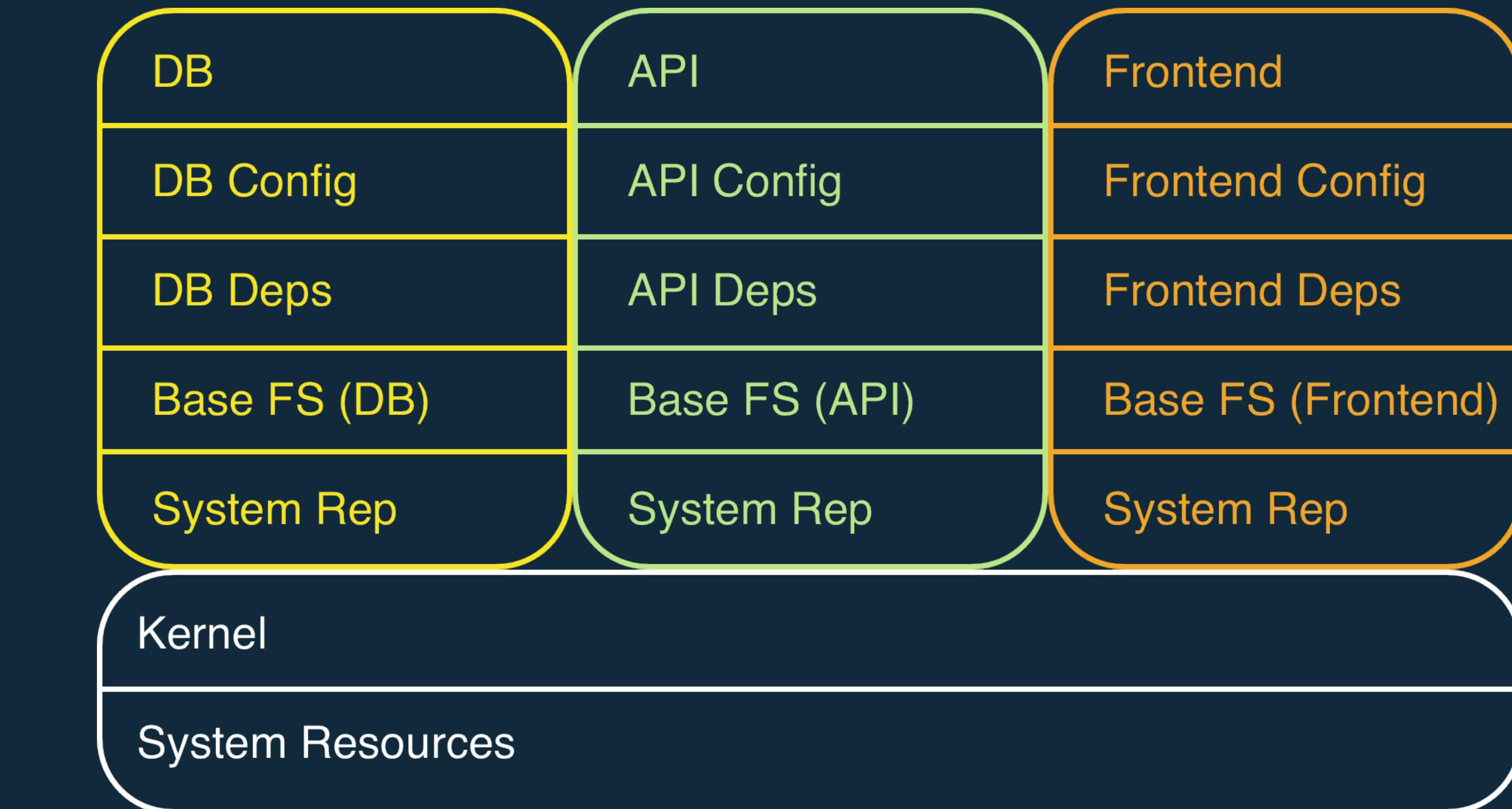
System Representation (FS, ports, process tree, users...)

Kernel

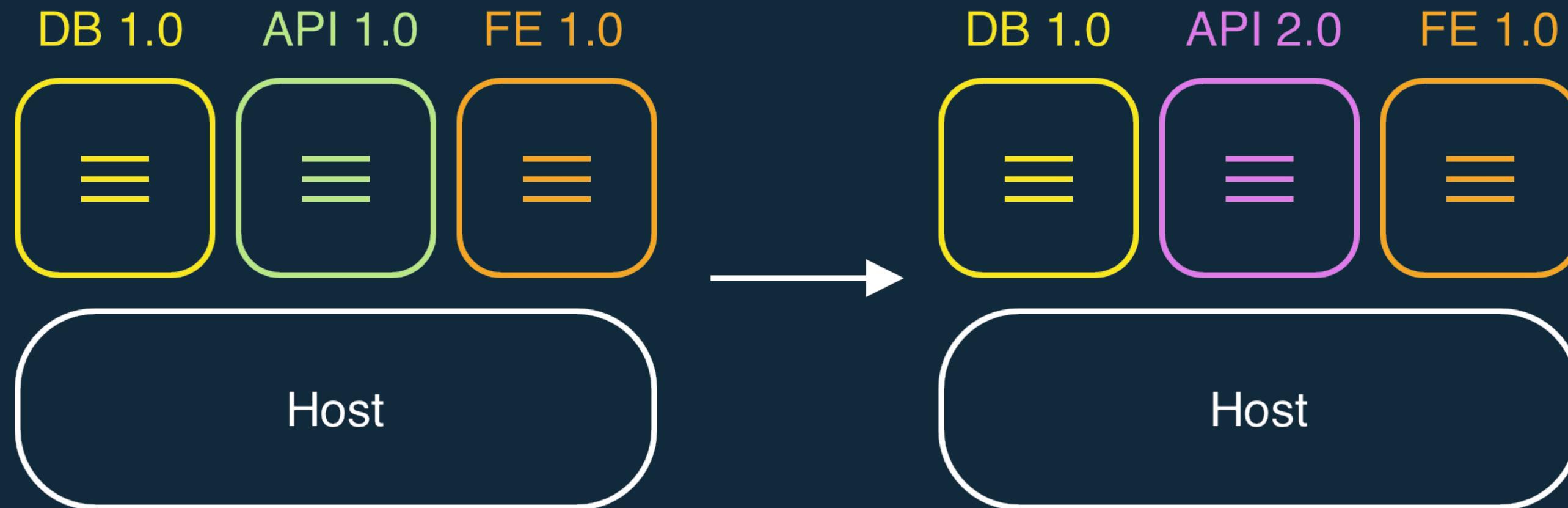
System Resources



WITH CONTAINERIZATION



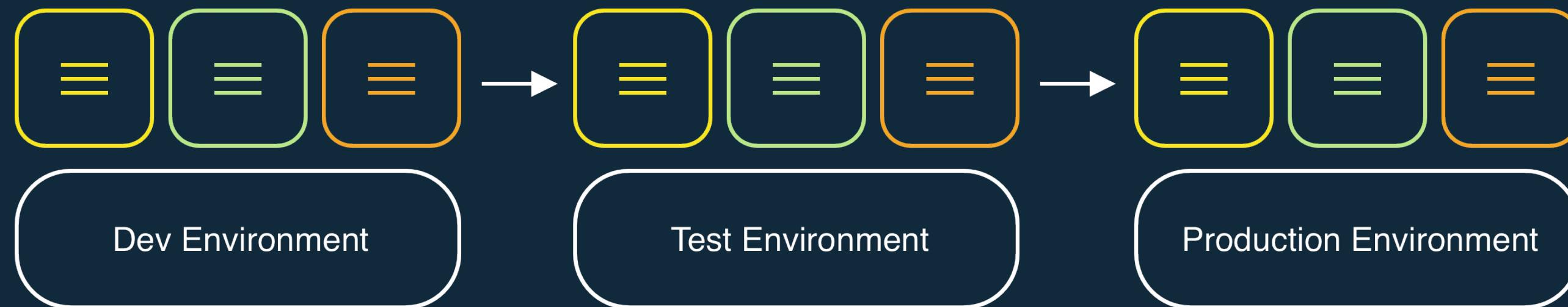
RAPID DEVELOPMENT



Containers can be removed and replaced with a minimum of impact on their neighbors, increasing developer choice and speed.



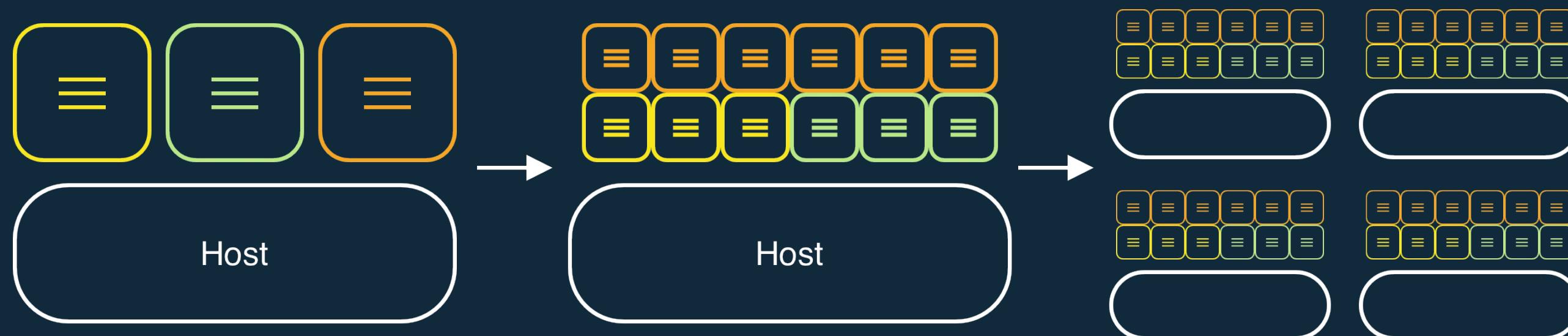
SMOOTH MIGRATION



Containers carry their environment and dependencies with them, simplifying and minimizing requirements on the hosts that run them.



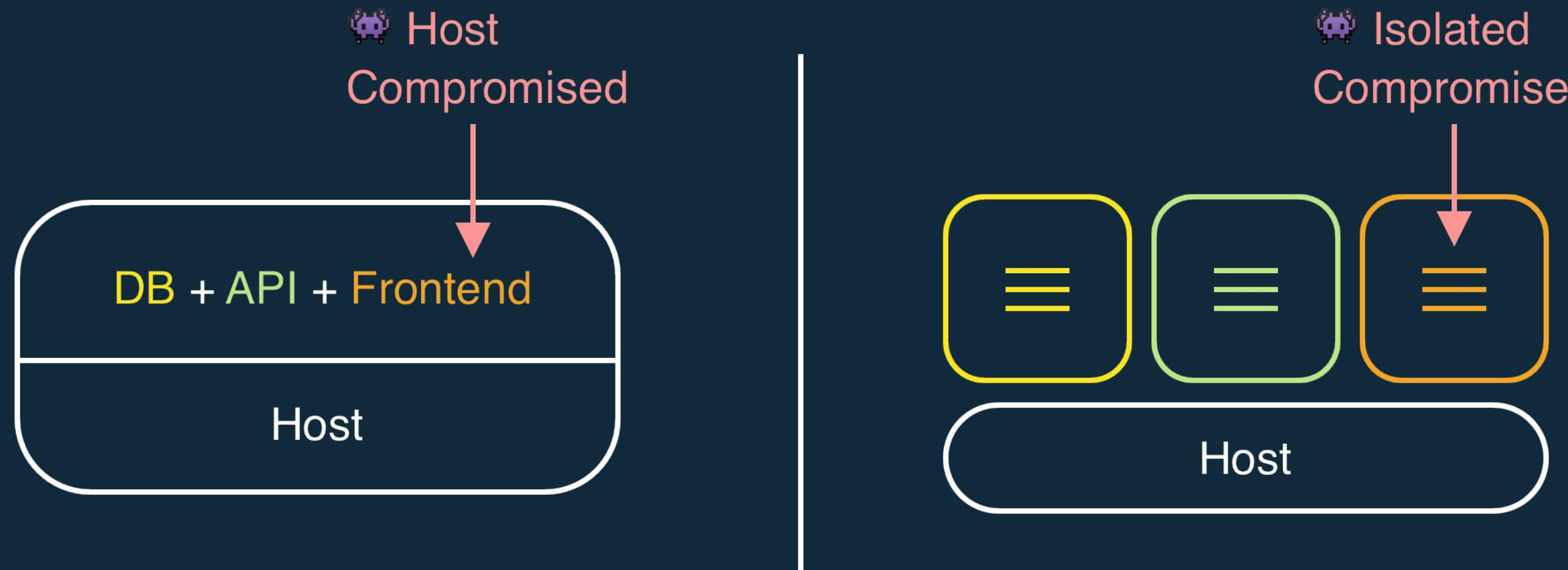
SIMPLE SCALE & MAINTENANCE



Weak coupling between containers minimizes side effects when scaling and simplifies monitoring.



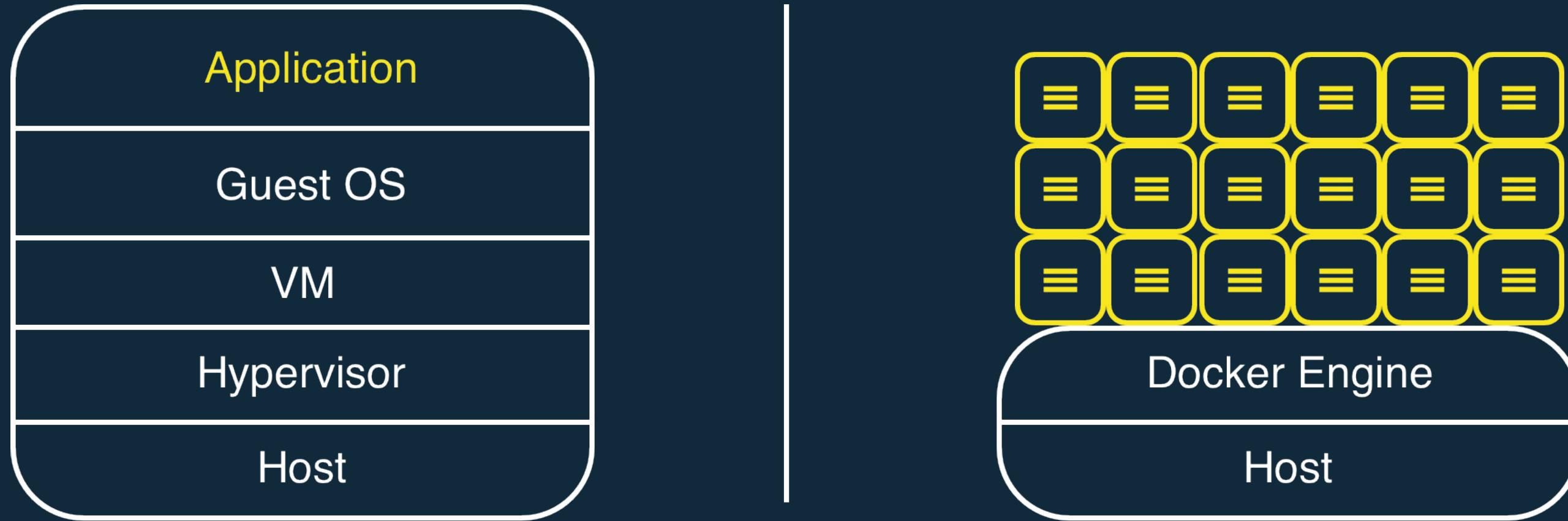
SECURE BY DEFAULT



Containers have private system resources, so a compromise in one does not affect the rest.



APPLICATION DENSITY

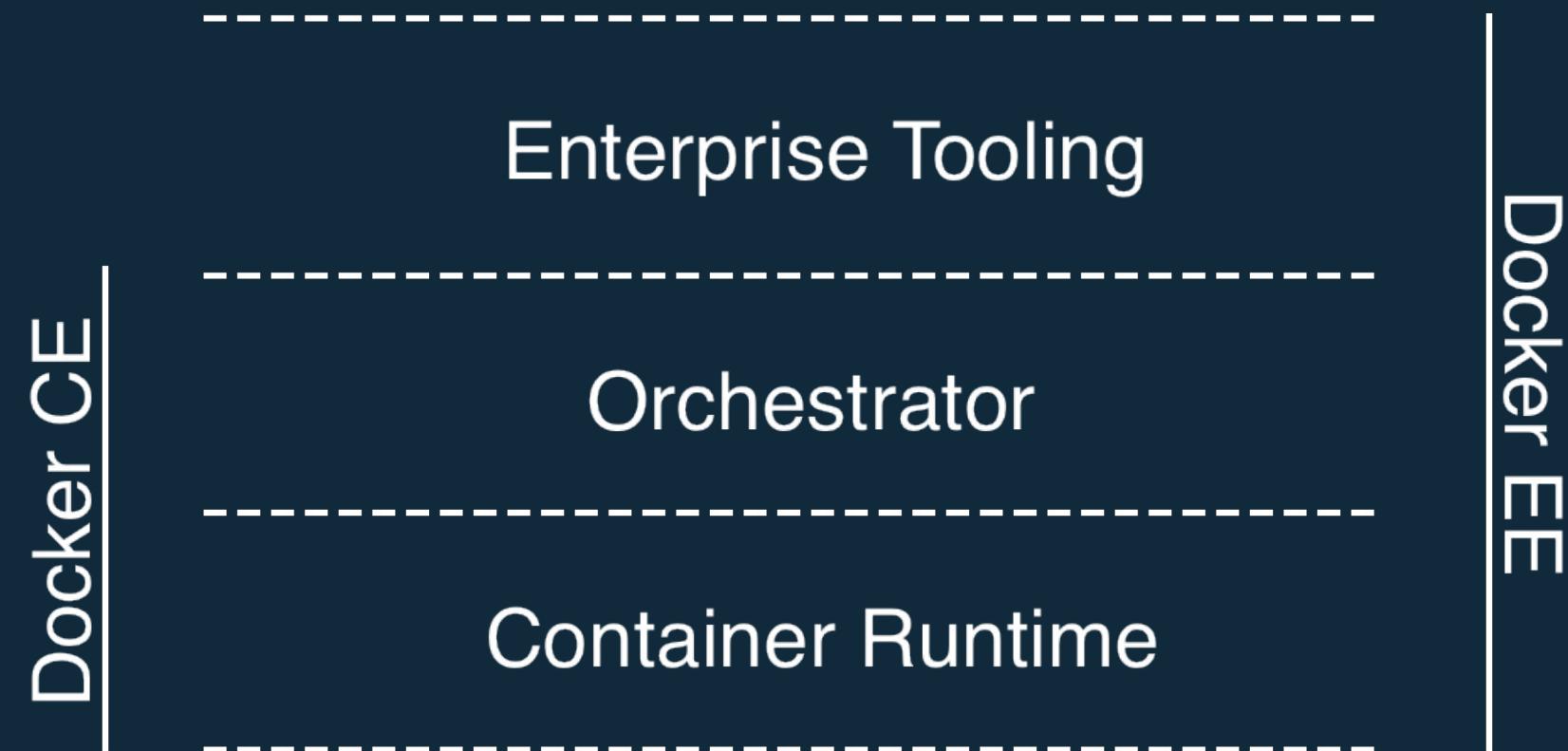


Virtual Machines

Containers save datacenter costs by running many more application instances than virtual machines can on the same physical hosts.



THE CONTAINERIZATION STACK



 CONTAINERIZATION FUNDAMENTALS

DISCUSSION: RUNNING CONTAINERS

What assurances would you need to run a process on an arbitrary host?

Consider:

- Hostile environments
- Required resources



LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Describe what a container is in terms of processes and isolation tools
- Use the key commands for interacting with Docker containers



CONTAINERS ARE PROCESSES

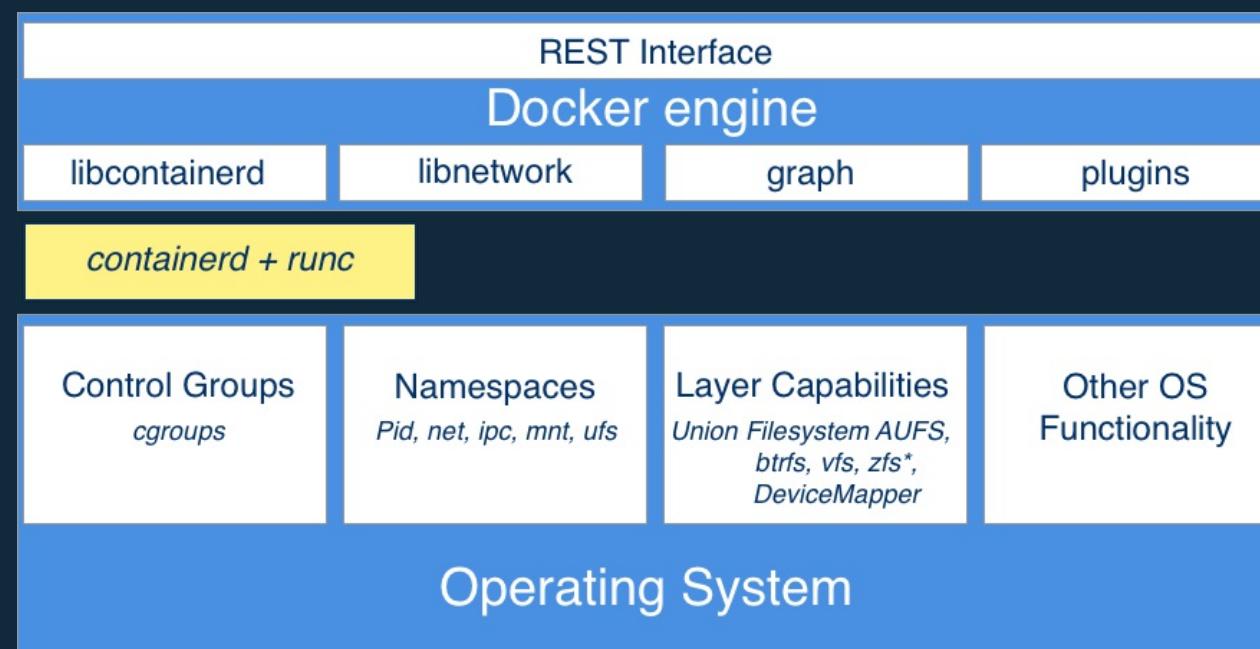
Containers are processes sandboxed by:

- Kernel namespaces
- Control Groups
- Root privilege management & syscall restrictions (Linux)
- VM isolation (Windows)

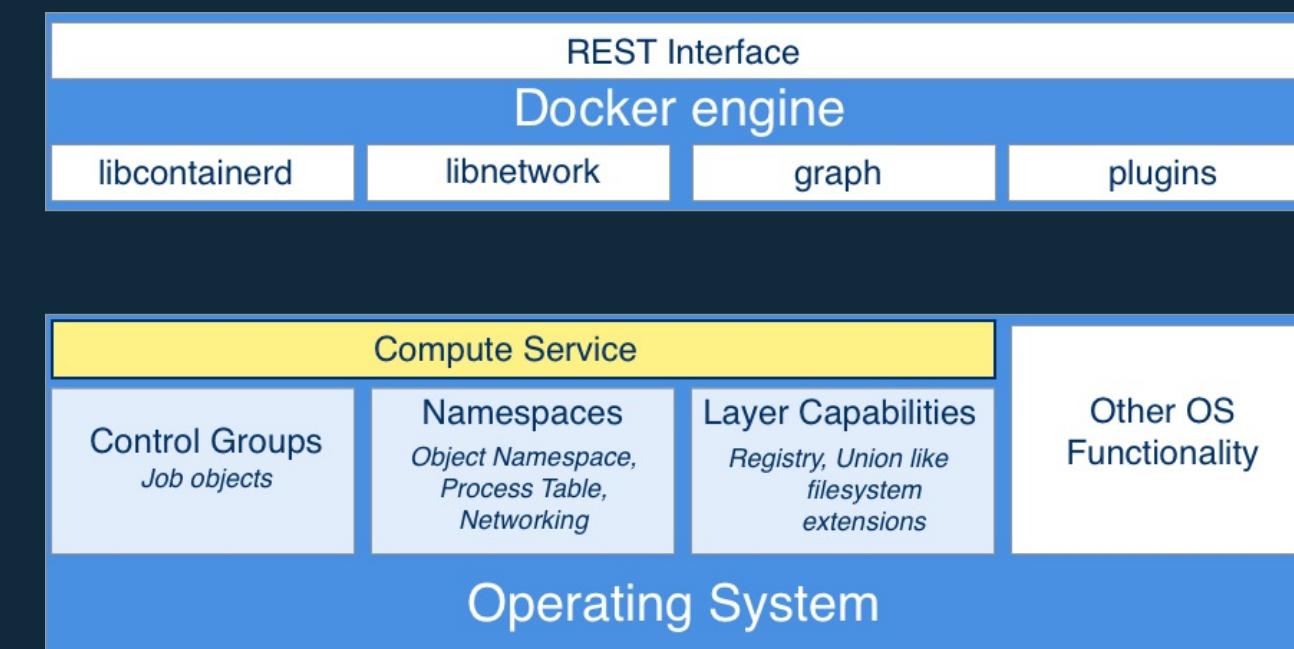


ARCHITECTURE

Architecture in Linux



Architecture in Windows

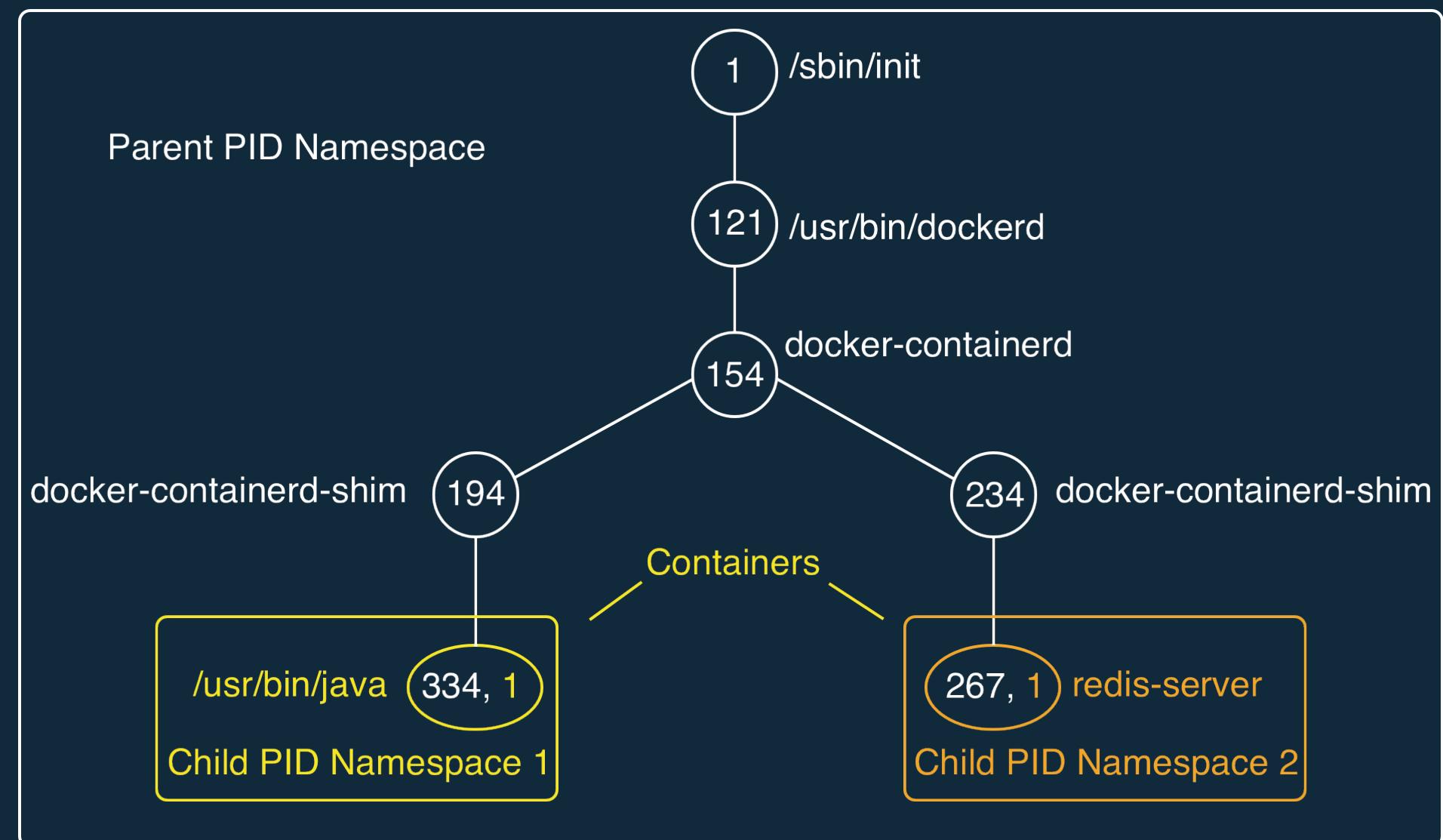


LINUX KERNEL NAMESPACES

- Default:
 - Process IDs
 - Network stacks
 - Inter-process communications
 - Mount points
 - Hostnames
- Optional:
 - User IDs



LINUX PID KERNEL NAMESPACE

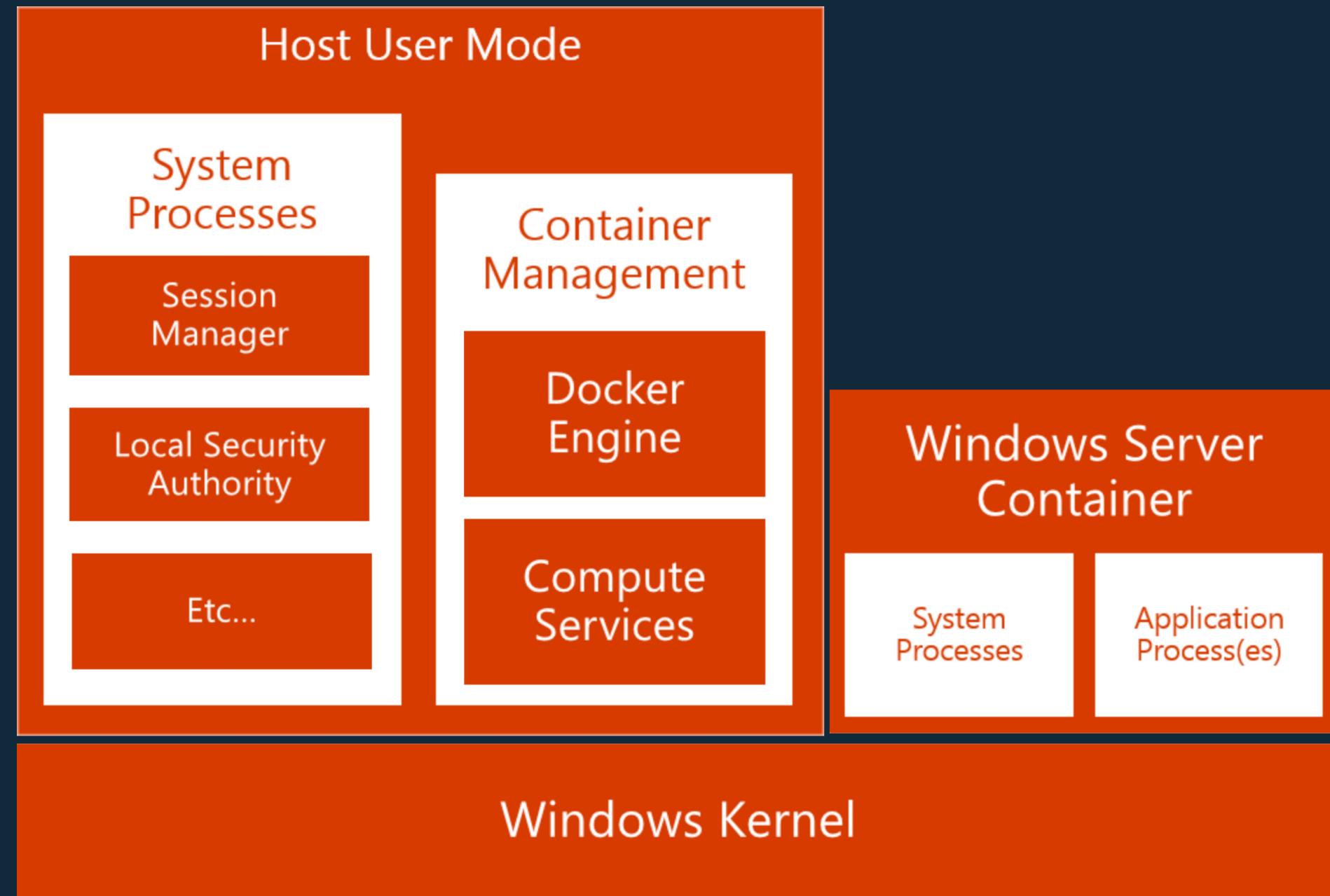


OPTIONAL LINUX ISOLATION FEATURES

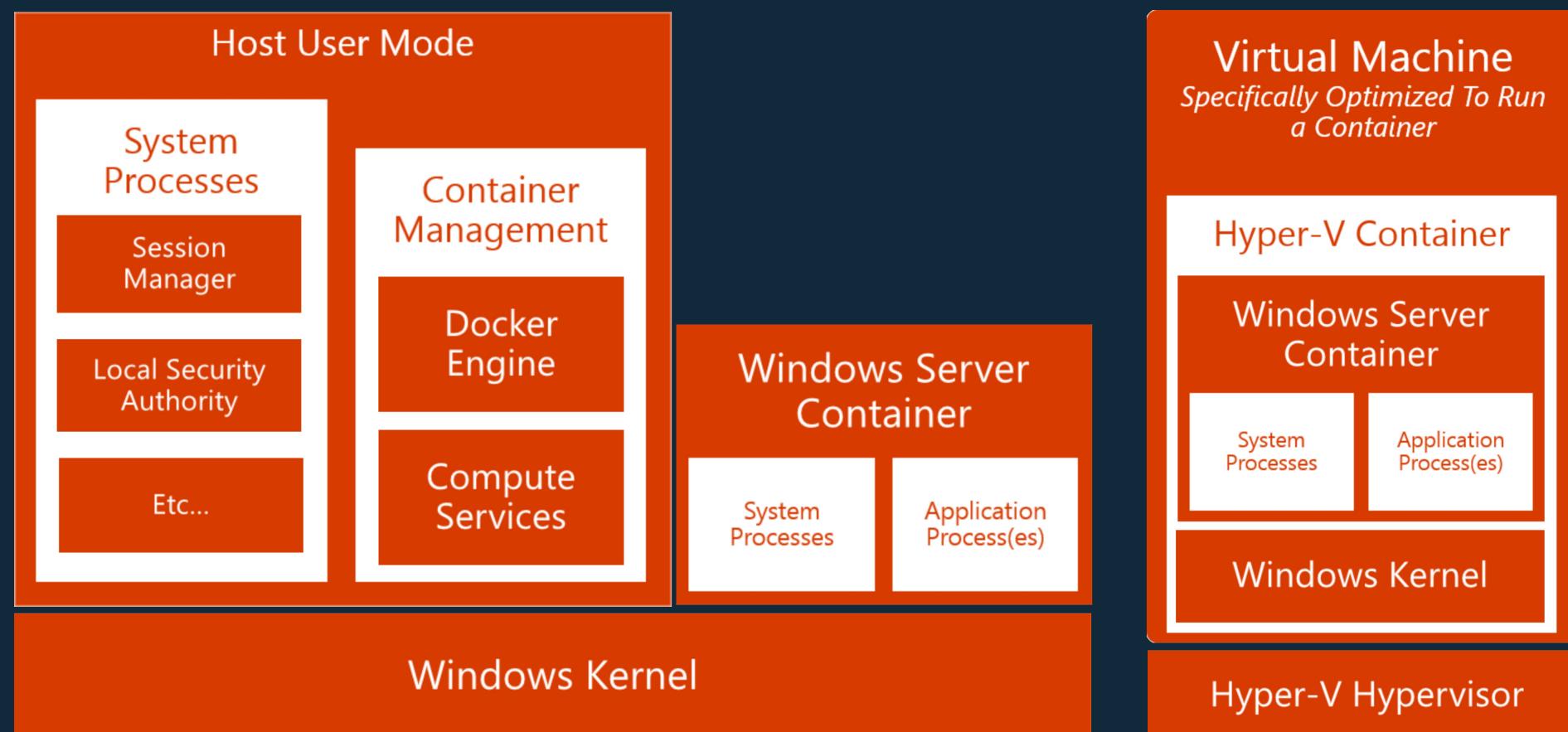
- Control groups: limit memory & CPU
- Root privilege management: whitelist root powers
- System call management: whitelist available system calls
- Linux Security Modules: mandatory filesystem access control



WINDOWS: HOST KERNEL CONTAINERS



WINDOWS: HYPER-V CONTAINERS





INSTRUCTOR DEMO: PROCESS ISOLATION

See the demo:

- Process Isolation

in the Docker Fundamentals Exercises book.





EXERCISE: CONTAINER BASICS

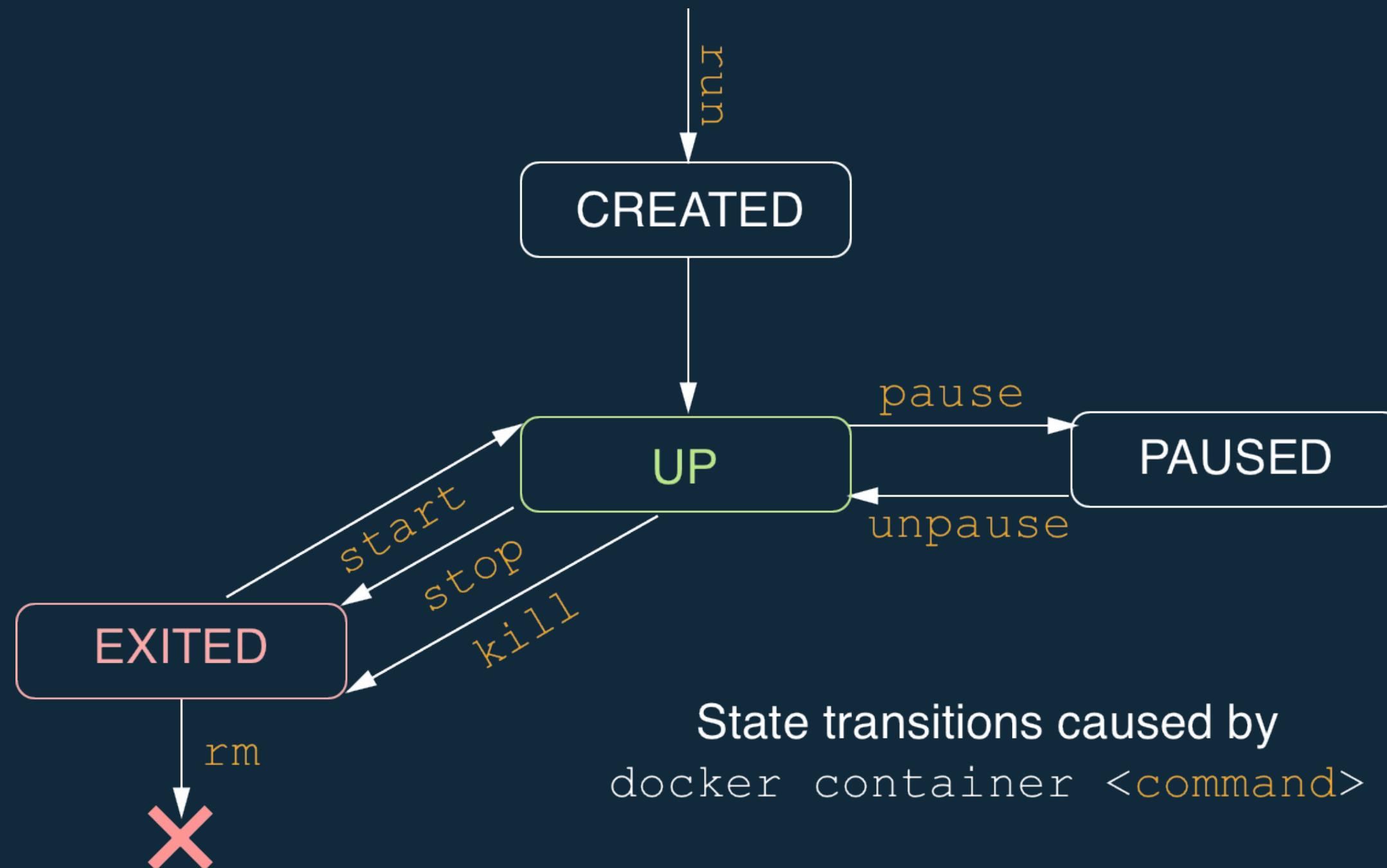
Work through:

- Running and Inspecting a Container
- Interactive Containers
- Detached Containers and Logging
- Starting, Stopping, Inspecting and Deleting Containers

In the Docker Fundamentals Exercises book.



CONTAINER LIFECYCLE



CONTAINER LOGS

- STDOUT and STDERR for a containerized process
- **docker container logs <container name>**



CONTAINER BASICS TAKEAWAYS

- Single process constrained by kernel namespaces, control groups and other technologies
- Private & ephemeral filesystem and data



FURTHER READING

- List of container commands: <http://dockr.ly/2iLBV2I>
- Getting started with containers: <http://dockr.ly/2gmxKWB>
- Start containers automatically: <http://dockr.ly/2xB8sMI>
- Limit a container's resources: <http://dockr.ly/2wqN5Nn>
- Keep containers alive during daemon downtime: <http://dockr.ly/2emLwb5>
- Isolate containers with a user namespace: <http://dockr.ly/2gmyKdf>
- Intro to Windows Containers: <https://dockr.ly/2CTYhYb>





CREATING IMAGES



DISCUSSION: PROVISIONING FILESYSTEMS

What are some potential difficulties with provisioning entire filesystems for containers? How can we avoid these problems?



LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Create images via several methods
- Describe the filesystem structure underlying an image
- Understand the performance implications of different image design decisions
- Correctly tag and namespace images for distribution on a registry



WHAT ARE IMAGES?

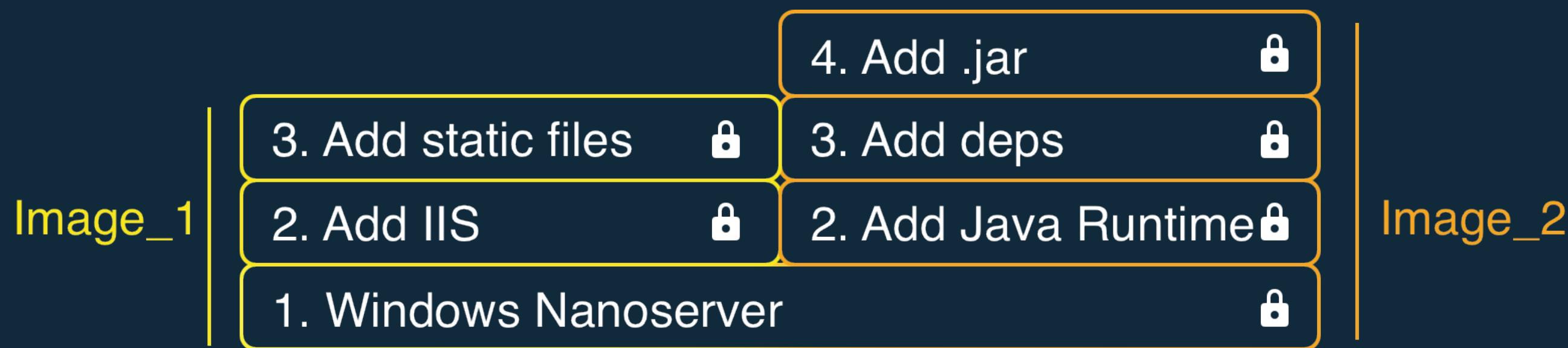
- A filesystem for container process
- Made of a stack of immutable layers
- Start with a base image
- New layer for each change

Image =
layered FS

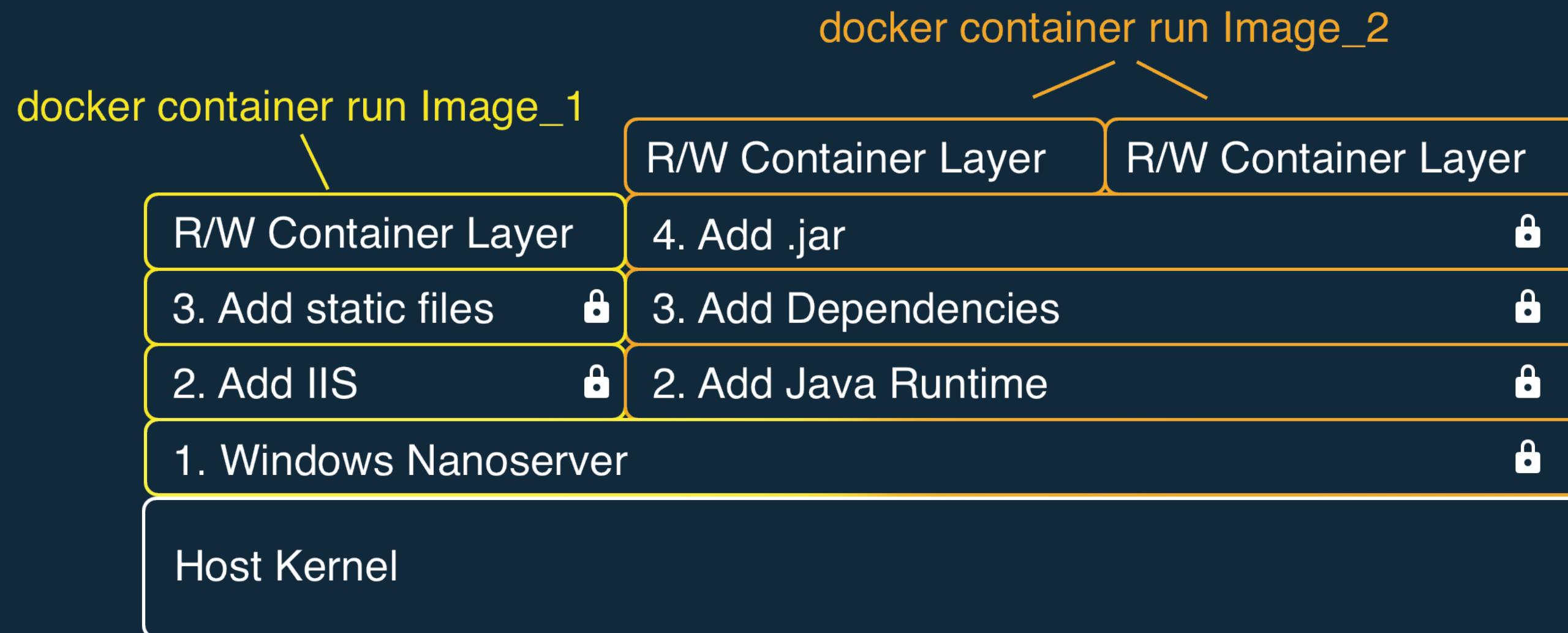
3. Add static files 
2. Add IIS 
1. Windows Nanoserver 



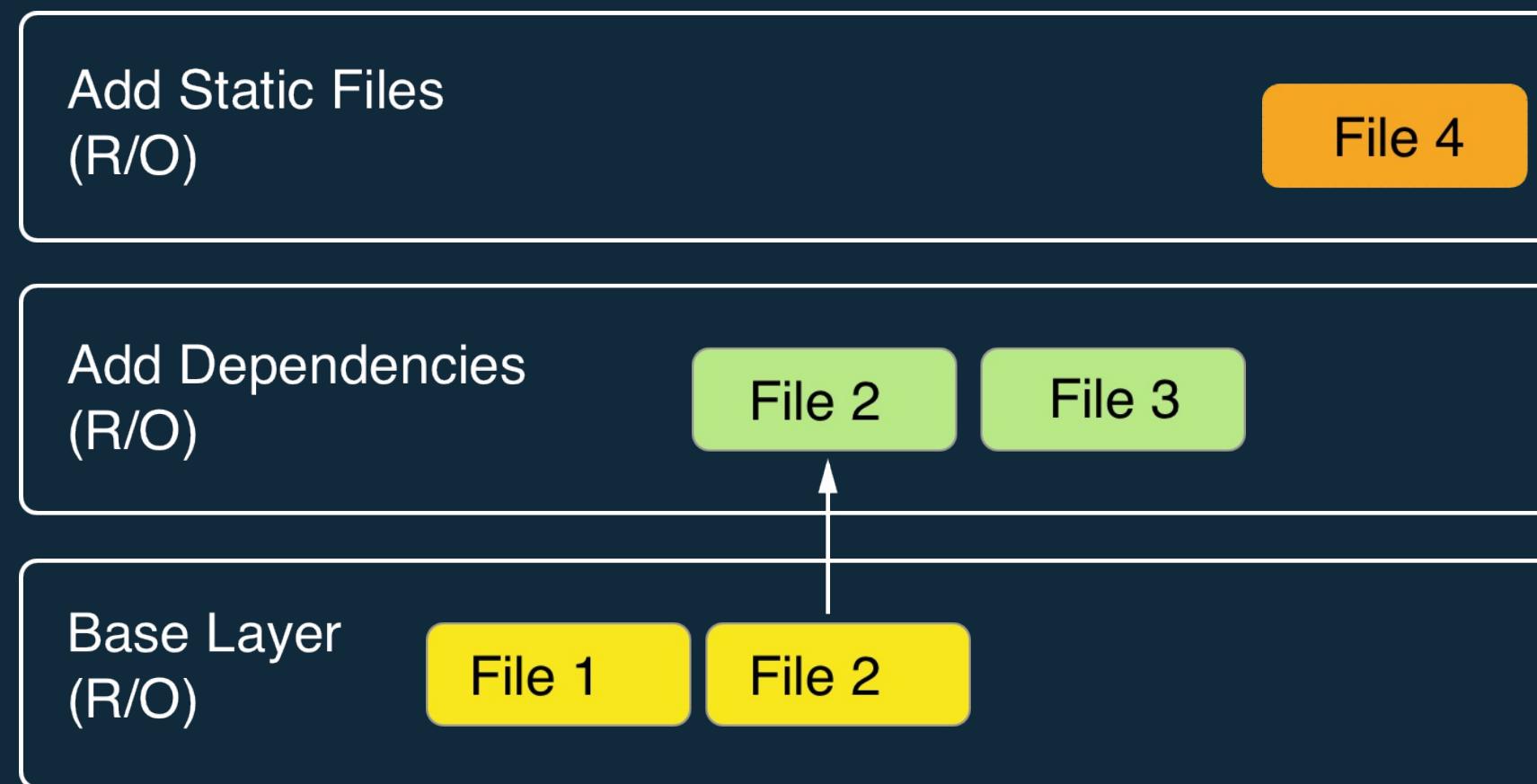
SHARING LAYERS



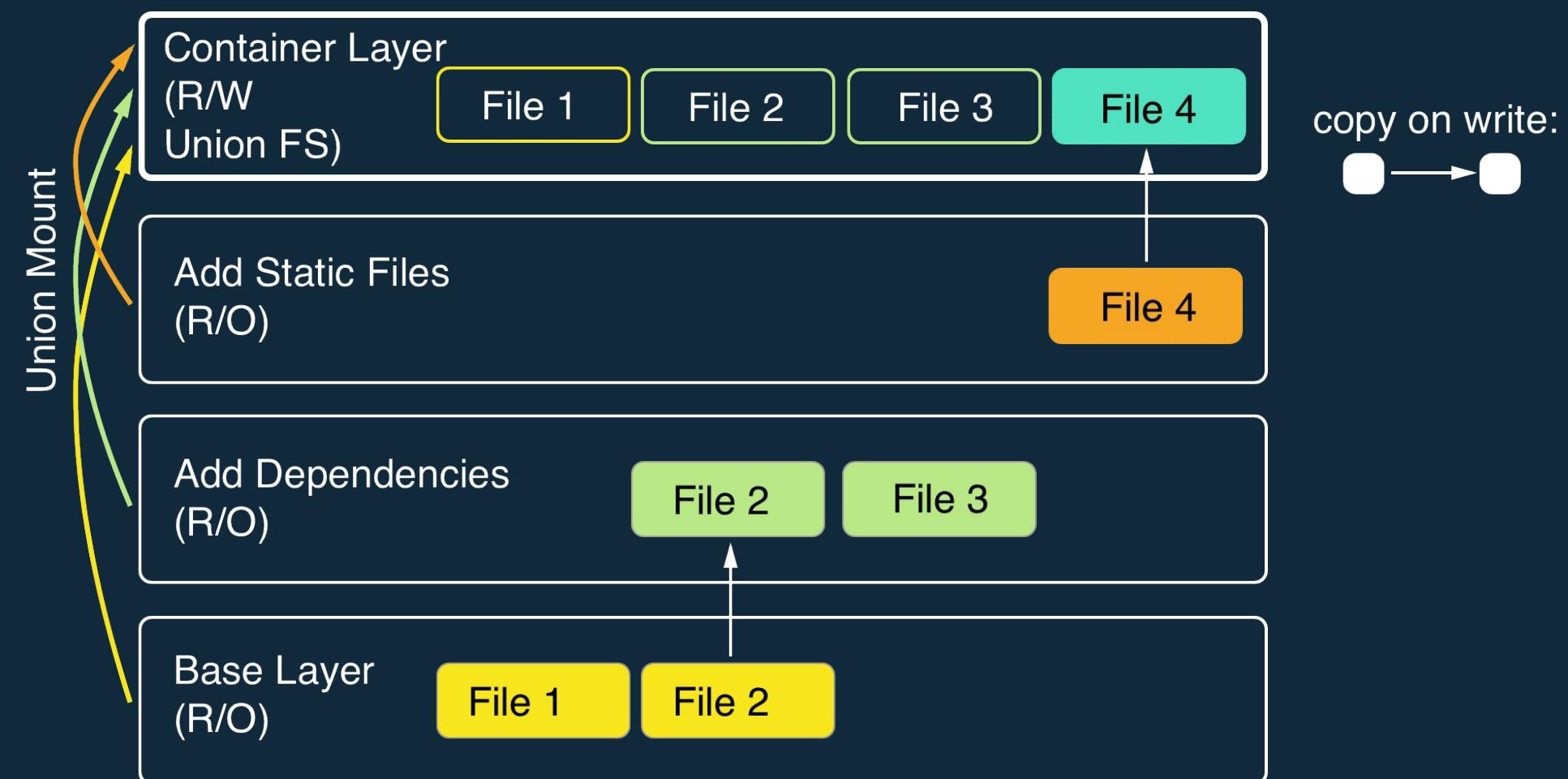
THE WRITABLE CONTAINER LAYER



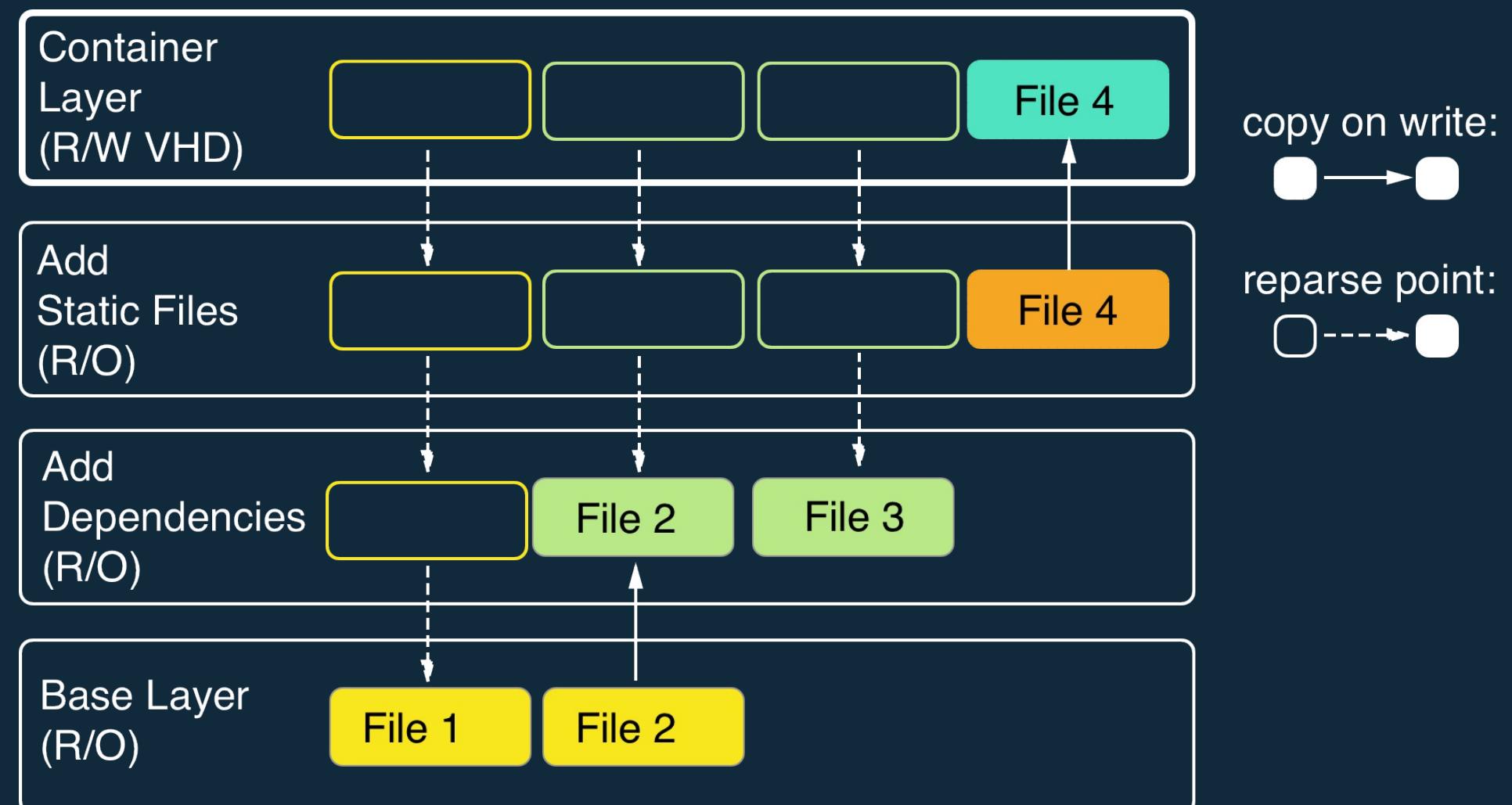
IMAGES: COPY ON WRITE



LINUX CONTAINERS: UNION FS



WINDOWS CONTAINERS: LINKED FS



CREATING IMAGES

Three methods:

- Commit the R/W container layer as a new R/O image layer.
- Define new layers to add to a starting image in a Dockerfile.
- Import a tarball into Docker as a standalone base layer.



COMMITTING CONTAINER CHANGES

- **docker container commit**
saves container layer as new R/O image layer
- Pro: build images interactively
- Con: hard to reproduce or audit; **avoid this** in practice.



DOCKERFILES

- Content manifest
- Provides image layer documentation
- Enables automation (CI/CD)



DOCKERFILES

- **FROM** command defines base image.
- Each subsequent command adds a layer or metadata
- **docker image build ...** builds image from Dockerfile

Linux containers:

```
# Comments begin with the pound sign
FROM ubuntu:16.04
RUN apt-get update && apt-get install -y wget
ADD /data /myapp/data
...
...
```

Windows containers:

```
# Comments begin with the pound sign
FROM microsoft/nanoserver:latest
RUN Install-Module -Name Nuget -Force
ADD c:\\myapp\\data c:\\app\\data
...
...
```





EXERCISES: CREATING IMAGES

Work through:

- Interactive Image Creation
- Creating Images with Dockerfiles (1/2)

in the Docker Fundamentals Exercises book.





INSTRUCTOR DEMO: CREATING IMAGES

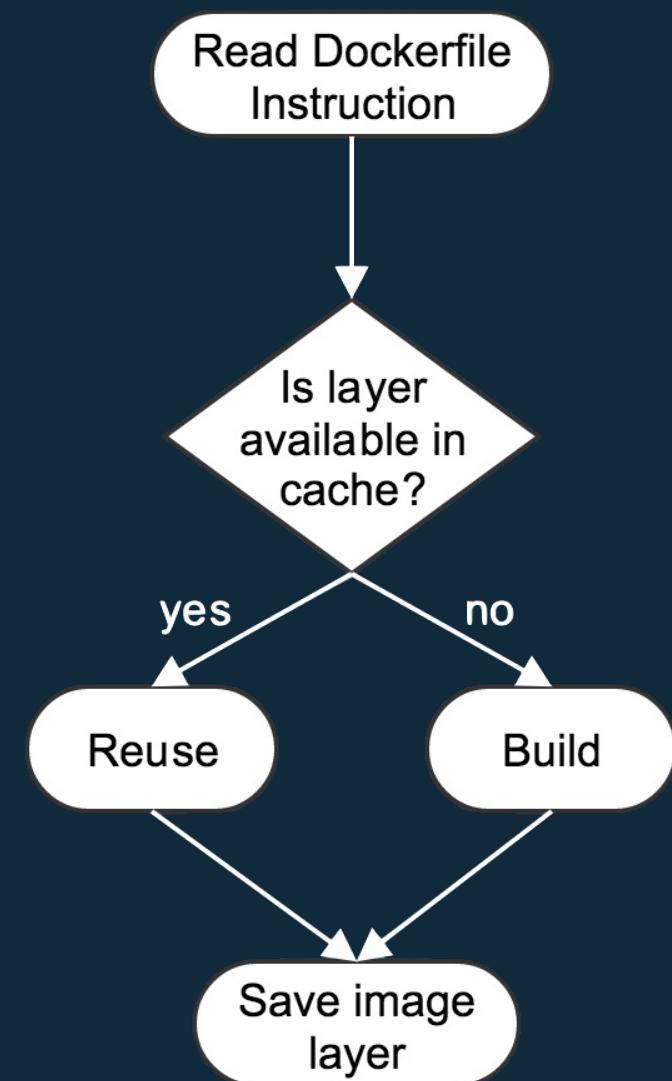
See the demo:

- Creating Images

In the Docker Fundamentals Exercises book.



BUILD CACHE



- After completion, the resulting image layer is labeled with a hash of the content of all current image layers in the stack.



CMD AND ENTRYPOINT

- Recall all containers run a process as their PID 1
- **CMD** and **ENTRYPOINT** allow us to specify default processes.



CMD AND ENTRYPOINT

- **CMD** alone: default command and list of parameters.
- **CMD + ENTRYPOINT**: **ENTRYPOINT** provides command, **CMD** provides default parameters.
- **CMD** overridden by command arguments to **docker container run**
- **ENTRYPOINT** overridden via
--entrypoint flag to **docker container run**.



SHELL VS. EXEC FORMAT

Linux containers:

```
# Shell form  
CMD sudo -u ${USER} java ...  
  
# Exec form  
CMD ["sudo", "-u", "jdoe", "java", ...]
```

Windows containers:

```
# shell form  
CMD c:\\Apache24\\bin\\httpd.exe -w  
  
# exec form  
CMD ["c:\\Apache24\\bin\\httpd.exe", "-w"]
```

Note the "\\\" in the expressions





EXERCISE: DOCKERFILES (2/2)

Work through:

- Creating Images with Dockerfiles (2/2)

In the Docker Fundamentals Exercises book.



COPY AND ADD COMMANDS

COPY copies files from build context to image:

```
COPY <src> <dest>
```

ADD can also **untar*** and **fetch URLs**.

* *Linux containers only!*

In both cases

- create checksum for files added
- log checksum in build cache
- cache invalidated if checksum changed



DOCKERFILE COMMAND ROUNDUP

- `FROM`: base image to start from (usually OS)
- `RUN`: run a command in the environment defined so far
- `CMD` and `ENTRYPOINT`: define default behavior
- `COPY` and `ADD`: copy files into container

Many more Dockerfile commands are available; see the docs at
<https://docs.docker.com/engine/reference/builder/>



ADVANCED DOCKERFILE CONSTRUCTION

How can we build images that are:

- Lightweight
- Secure
- Minimal build times



THE SCRATCH IMAGE

- An "empty" image
- Can't be pulled
- Doesn't create a layer
- Used for building images not based on any pre-existing image
- Linux only

```
FROM scratch

ADD centos-7-docker.tar.xz /

LABEL org.label-schema.schema-version="1.0" \
      org.label-schema.name="CentOS Base Image" \
      org.label-schema.vendor="CentOS" \
      org.label-schema.license="GPLv2" \
      org.label-schema.build-date="20181205"

CMD ["/bin/bash"]
```



MULTI-STAGE BUILDS

Linux containers:

Hello World, in C:

```
FROM alpine:3.5
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
ADD hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
CMD /app/bin/hello
```

Windows containers:

Hello World, in Go:

```
FROM golang:nanoserver
COPY . /code
WORKDIR /code
RUN go build hello.go
CMD ["\"\\code\\hello.exe"]
```

builds to:

Bills-MBP:demo billmills\$ docker image ls hwc				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hwc	latest	142c29686b6a	15 hours ago	184 MB



MULTI-STAGE BUILDS

Hello World, lightweight:

Linux containers:

```
# Full SDK version (built and discarded)
FROM alpine:3.5 AS build
RUN apk update && \
    apk add --update alpine-sdk
RUN mkdir /app
WORKDIR /app
ADD hello.c /app
RUN mkdir bin
RUN gcc -Wall hello.c -o bin/hello
```

```
# Lightweight image returned as final product
FROM alpine:3.5
COPY --from=build /app/bin/hello /app/hello
CMD /app/hello
```

Windows containers:

```
FROM golang:nanoserver as gobuild
COPY . /code
WORKDIR /code
RUN go build hello.go

FROM microsoft/nanoserver
COPY --from=gobuild /code/hello.exe /hello.exe
EXPOSE 8080
CMD ["\"hello.exe\"]
```

Builds to:

```
Bills-MBP:demo billmills$ docker image ls hwc
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
hwc            latest       5d925fcf9c96  39 seconds ago  4MB
```



BUILD TARGETS

Dockerfile

```
FROM <base image> as base
...
FROM <foo image> as foo
...
FROM <bar image> as bar
...
FROM alpine:3.4
...
COPY --from foo ...
COPY --from bar ...
...
```

Building the image

docker image build --target <name> ...



BUILDKIT

- Speed-optimized builder, enable via `export DOCKER_BUILDKIT=1`
- Parallelizes multi-stage builds
- Custom frontends
- 2x - 9x build speedup
- Linux only as of 18.09.0-ee





EXERCISE: MULTI-STAGE BUILDS

Work through:

- Multi-Stage Builds

In the Docker Fundamentals Exercises book.



IMAGE CONSTRUCTION BEST PRACTICES

- Start with official images
- Use multi-stage builds to drop compilers, SDKs...
- More layers leverage the cache...
- ...but fewer layers perform better.



DEVELOPMENT: MORE LAYERS

Bad caching:

```
FROM python:3.5-alpine
RUN mkdir /app
COPY /mypy /app/
RUN pip install -r app/reqs.txt
...
```

Good caching:

```
FROM python:3.5-alpine
RUN mkdir /app
COPY /mypy/reqs.txt /app/
RUN pip install -r app/reqs.txt
COPY /mypy /app/
...
...
```



PRODUCTION: LESS LAYERS

- To collapse ALL image layers:
- Or build with **--squash** flag (experimental): compress all non-base layers
- Combine **container export** with **--squash** for one shareable base layer + one application-specific upper layer

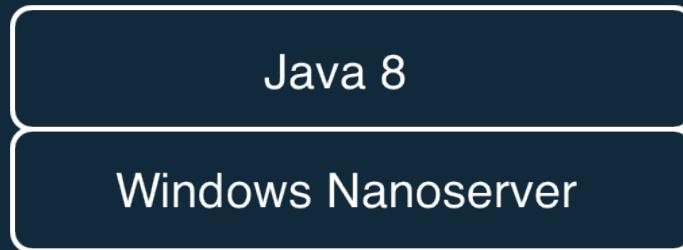
```
docker container run -d --name demo mytallimage:1.0  
docker container export demo > image.tar  
cat image.tar | docker image import - myflatimage:1.0
```



BEST PRACTICE: PATCHING & UPDATES



BAD



GOOD



IMAGE TAGS

- Optional string after image name, separated by :
- **:latest** by default
- Same image with two tags shares same ID, image layers:

```
$ docker image ls centos*
REPOSITORY      TAG          IMAGE ID       CREATED        SIZE
centos          7            8140d0c64310    7 days ago    193 MB
$ docker image tag centos:7 centos:mytag
$ docker image ls centos*
REPOSITORY      TAG          IMAGE ID       CREATED        SIZE
centos          7            8140d0c64310    7 days ago    193 MB
centos          mytag        8140d0c64310    7 days ago    193 MB
```



IMAGE NAMESPACES

Images exist in one of three namespaces:

- Root (**ubuntu, nginx, mongo, mysql, ...**)
- User / Org (**jdoe/myapp:1.1, microsoft/nanoserver:latest, ...**)
- Registry (**FQDN/jdoe/myapp:1.1, ...**)



IMAGE TAGGING & NAMESPACING

- Tag on build:

```
docker image build -t myapp:1.0 .
```

- Retag an existing image:

```
docker image tag myapp:1.0 me/myapp:2.0
```

- Note **docker image tag** can set both tag and namespace.
- Names and tags are just pointers to image ID
- Image ID corresponds to immutable content addressable storage



SHARING IMAGES

- Docker Hub
 - Provides certified commercial and free software distributed as Docker Images
 - Shares community-generated images and content





EXERCISE: MANAGING IMAGES

Work through:

- Managing Images

In the Docker Fundamentals Exercises book.



IMAGE CREATION TAKEAWAYS

- Images are built out of read-only layers.
- Dockerfiles specify image layer contents.
- Key Dockerfile commands: **FROM**, **RUN**, **COPY** and **ENTRYPOINT**
- Images must be namespaced according to where you intend on sharing them.



FURTHER READING 1/2

Linux Containers:

- Best practices for writing Dockerfiles: <http://dockr.ly/22WijO>
- Use multi-stage builds: <http://dockr.ly/2ewcUY3>
- More about images, containers, and storage drivers: <http://dockr.ly/1TuWndC>
- Details on image layering: <https://bit.ly/2AHX7iW>
- Graphdriver plugins: <http://dockr.ly/2eIVCab>
- Docker Reference Architecture: An Intro to Storage Solutions for Docker CaaS: <http://dockr.ly/2x8sBw2>
- How to select a storage driver: <http://dockr.ly/2eDu8y0>
- Use the AUFS storage driver: <http://dockr.ly/2jVc1Zz>
- User guided caching in Docker: <http://dockr.ly/2xKafPf>



FURTHER READING 2/2

WINDOWS CONTAINERS:

- Dockerfile on Windows: <http://bit.ly/2waNvsS>
- Optimize Windows Dockerfiles: <http://bit.ly/2whpfn7>
- Windows Container Samples:
 - <http://bit.ly/2wCrPXY>
 - <http://bit.ly/2ghRr5o>
- Powershell Tricks: <http://bit.ly/2wb7Azn>
- Multi-stage builds for Windows containers: <http://bit.ly/2iBRmdN>
- The SHELL command: <http://dockr.ly/2whvyqZ>





DOCKER VOLUMES



DISCUSSION: MANAGING DATA

If a container generates a lot of data, where should it be stored? What if you need to provision data to a container?



LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Define a volume and identify its primary use cases
- Describe the advantages and potential security risks of mounting volumes and host directories into containers



VOLUME USECASES

Volumes provide a R/W path **separate** from the layered filesystem.

- **Mount** data at container startup
- **Persist** data when a container is deleted
- **Share** data between containers
- **Speed up** I/O by circumventing the union filesystem



BASIC VOLUMES

- **Named:** managed by Docker; filesystem independent; user-specified identifier
- **Anonymous:** managed by Docker; filesystem independent; randomly-generated identifier
- **Host mounted:** mount a specific path on the host; DIY management





INSTRUCTOR DEMO: VOLUMES

See the demo:

- Basic Volume Usage

In the Docker Fundamentals Exercises book.



VOLUMES IN DOCKERFILES

- VOLUME instruction creates a mount point
- Can specify arguments in a JSON array or string
- Cannot map volumes to host directories
- Volumes are initialized when the container is executed

Linux containers:

```
FROM nginx:latest
...
# string example
VOLUME /myvolume

# string example with multiple volumes
VOLUME /www/website1 /www/website2

# JSON example
VOLUME ["myvol1", "myvol2"]
...
```

Windows containers:

```
FROM microsoft/iis:nanoserver
...
# string examples
VOLUME c:\\data
VOLUME c:\\data2 c:\\data3

# JSON examples
VOLUME ["c:\\\\data4", "c:\\\\data5"]
VOLUME ["c:/data6", "c:/data7"]
...
```



VOLUMES AND SECURITY

- Point of ingress to the host and other containers
- Don't mount things unnecessarily
- Use the `:ro` flag
- Linux: in-memory `tmpfs` mounts available





EXERCISE: VOLUMES USECASE

Work through:

- Database Volumes

In the Docker Fundamentals Exercises book.



DOCKER VOLUME TAKEAWAYS

- Volumes persist data beyond the container lifecycle
- Volumes bypass the copy on write system (better for write-heavy containers)



FURTHER READING

- How to use volumes: <http://dockr.ly/2vRZBDG>
- Troubleshoot volume errors: <http://dockr.ly/2vyjvbP>
- Docker volume reference: <http://dockr.ly/2ewrlew>



CONTAINERIZATION FUNDAMENTALS

CONCLUSION: ANY APP, ANYWHERE.

- Containers are isolated processes
- Images provide filesystem for containers
- Volumes persist data





DOCKER NETWORKING BASICS



DISCUSSION: PORTABLE NETWORKS

Network traffic must by definition traverse a network outside its originating container. How can we make inter-container communication as portable and secure as containers themselves?



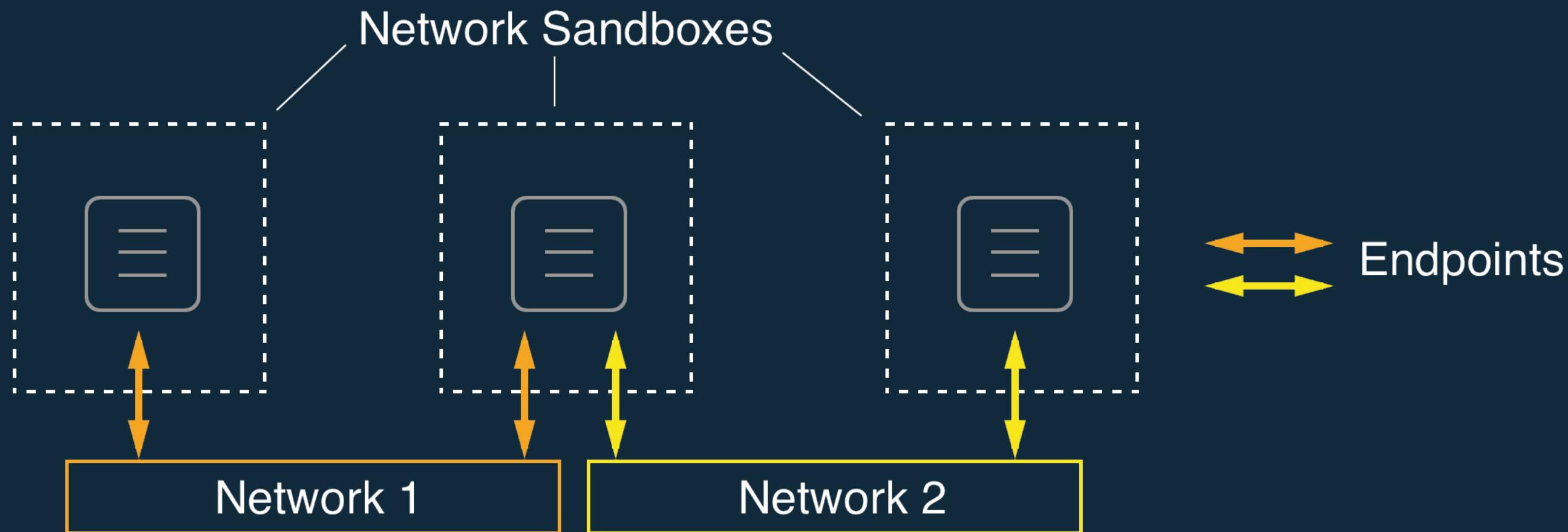
LEARNING OBJECTIVES

By the end of this module, learners will be able to:

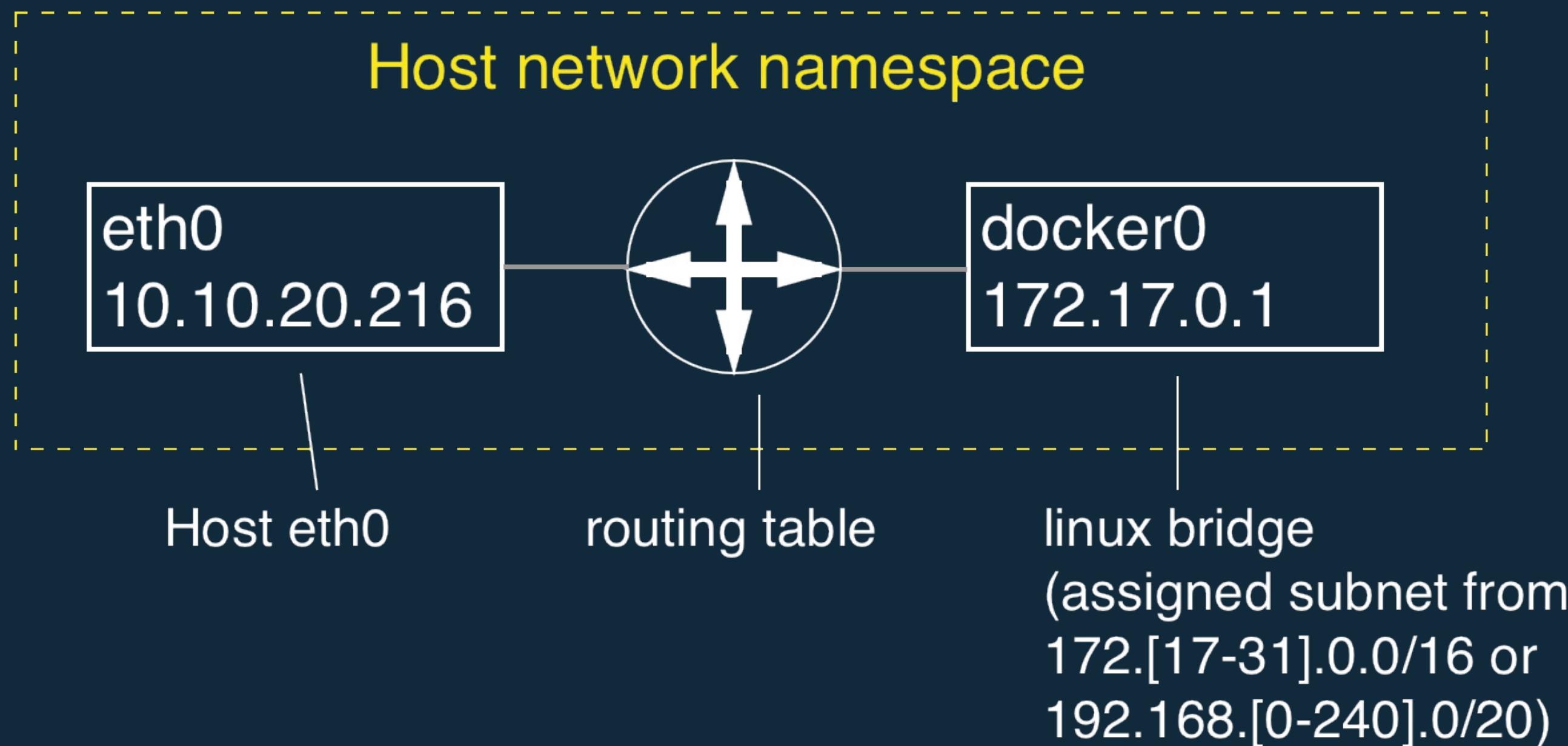
- Describe Docker's container network model and its security implications
- Describe the basic technologies that underwrite single host networks
- Understand how Docker manipulates a host's firewall rules to control container traffic



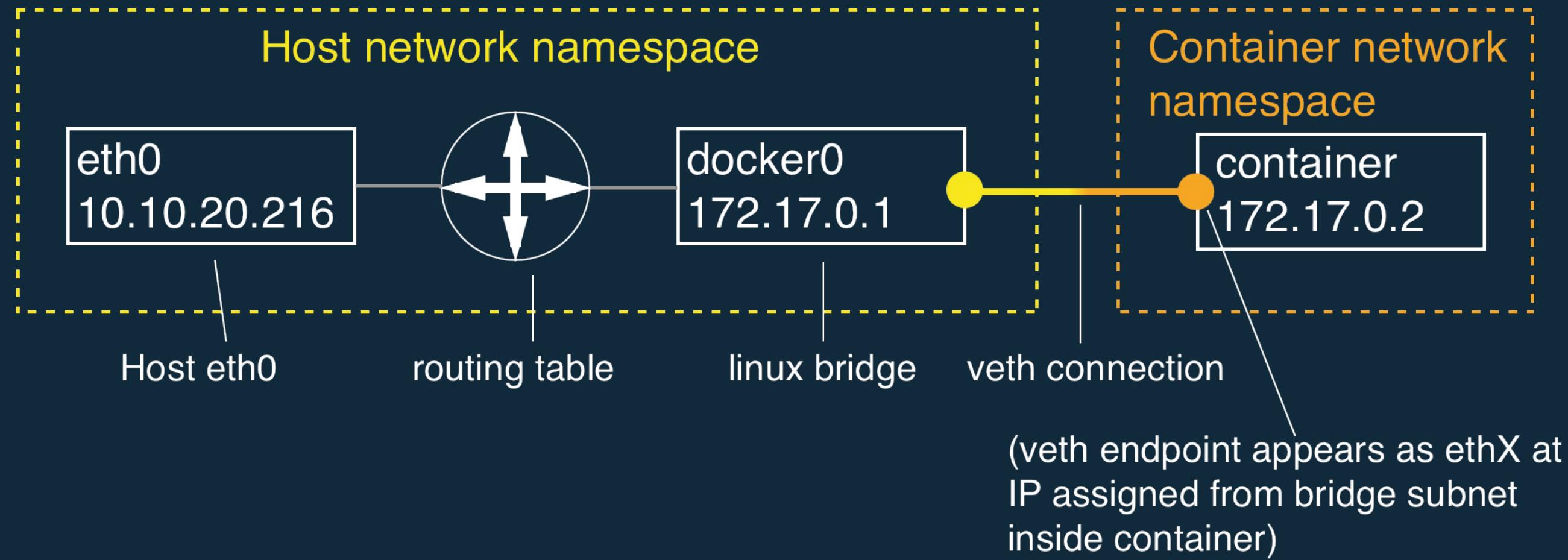
THE CONTAINER NETWORK MODEL



LINUX: DEFAULT SINGLE-HOST NETWORK



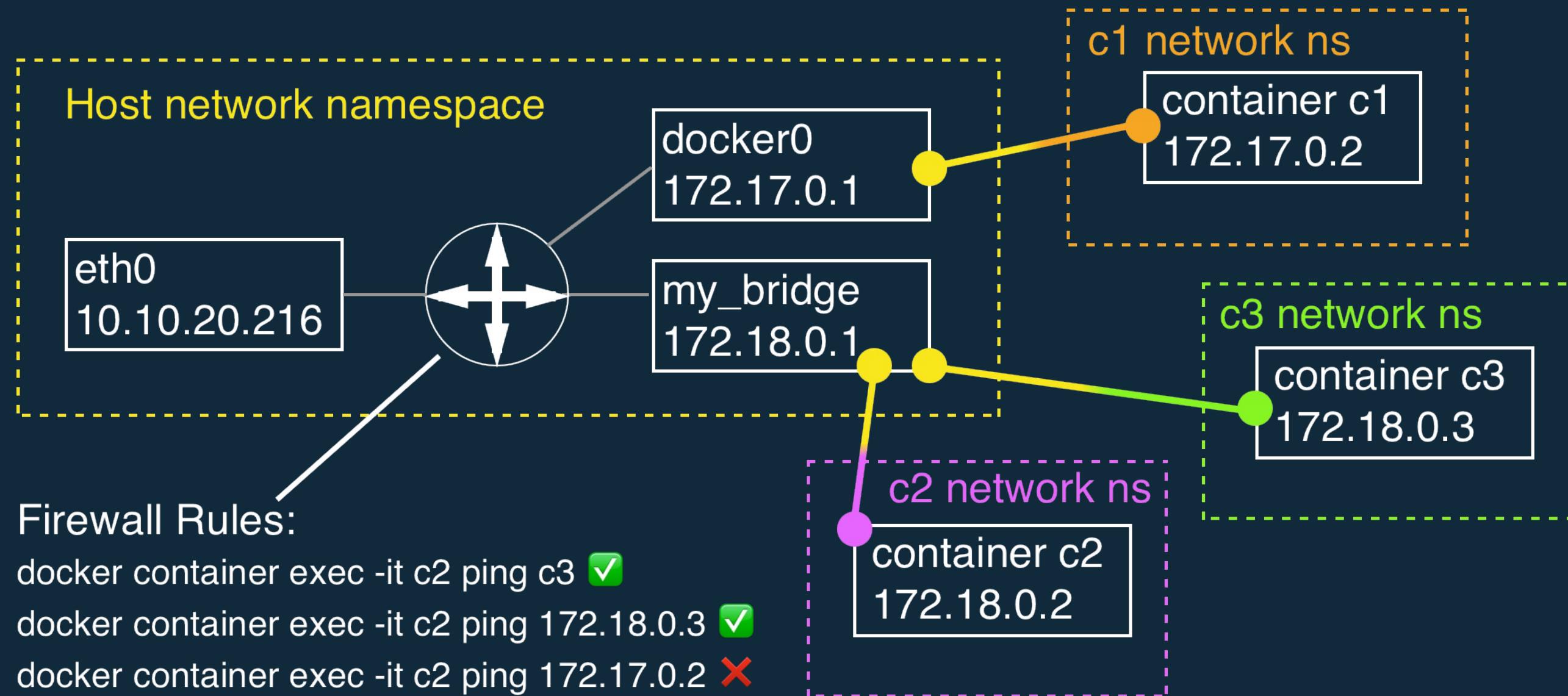
LINUX: DEFAULT CONTAINER NETWORKING



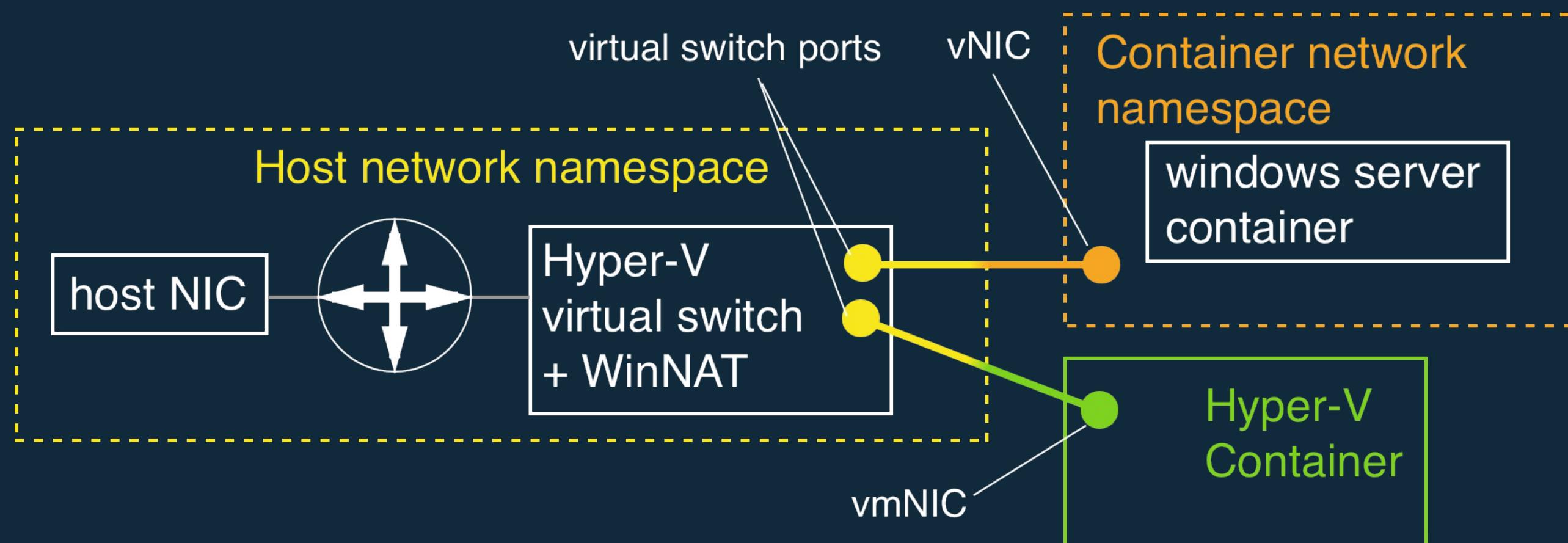
Quiz: identify the sandbox, endpoint and network corresponding to the container networking model objects in this diagram.



LINUX: USER-DEFINED BRIDGES & FIREWALLS



WINDOWS: THE NAT NETWORK



EXPOSING CONTAINER PORTS

- Containers have no public IP address by default.
- Can forward host port -> container port
- Mapping created manually or automatically.
- Port mappings visible via

docker container ls or

docker container port





INSTRUCTOR DEMO: SINGLE HOST NETWORKS

See the demo:

- Single Host Networks

In the Docker Fundamentals Exercises book.





EXERCISE: SINGLE HOST NETWORKS

Work through:

- Introduction to Container Networking
- Container Port Mapping

in the Docker Fundamentals Exercises book.



DOCKER NETWORKING TAKEAWAYS

- Single host networks follow the container networking model:
 - Sandbox: Network namespaces
 - Endpoint: veth (linux) / vNIC (win)
 - Network: bridge (linux) / NAT + Hyper-V switch (win)
- Containers resolve each other by DNS lookup when explicitly named and attached to custom networks
- Docker software defined networks are firewalled from each other by default



FURTHER READING

- Docker Reference Architecture: Designing Scalable, Portable Docker Container Networks: <https://dockr.ly/2q3O8jq>
- Network containers: <http://dockr.ly/2x1BYgW>
- Docker container networking: <http://dockr.ly/1QnT6y8>
- Understand container communication: <http://dockr.ly/2iSrH00>

WINDOWS CONTAINERS:

- Windows Container Networking: <https://bit.ly/2JdIJ5V>
- Windows NAT (WinNAT) — Capabilities and limitations: <http://bit.ly/2vC23xS>
- Published Ports On Windows Containers Don't Do Loopback: <http://bit.ly/2whltu4>
- Cheatsheet to replace "netsh", "nslookup" and more with Powershell: <http://bit.ly/2iB7pbw>





INTRODUCTION TO SWARM MODE



DISCUSSION: ANY APP, ANYWHERE?

Containers are portable. What does this imply about the best ways to manage a containerized data center?



LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Create a swarm and schedule workload on it via services and stacks
- Provision configuration to services via configs and secrets
- Control scheduling decisions governing where containers are placed in the swarm
- Configure service discovery and routing for swarm services
- Manage application lifecycle on swarm



ORCHESTRATOR GOALS

Top-line goal: operate a datacenter like a pool of compute resources (not individual machines). This requires:

- Add / remove compute resources securely and easily
- Schedule containers across the cluster transparently
- Streamline container-to-container communication (service discovery, load balancing and routing)



ORCHESTRATOR ARCHITECTURE

Orchestrators rely on three networking planes:

- Management plane: cluster & workload maintenance
- Control plane: service discovery
- Data plane: routing



SWARM MANAGEMENT PLANE

Management plane: Raft consensus

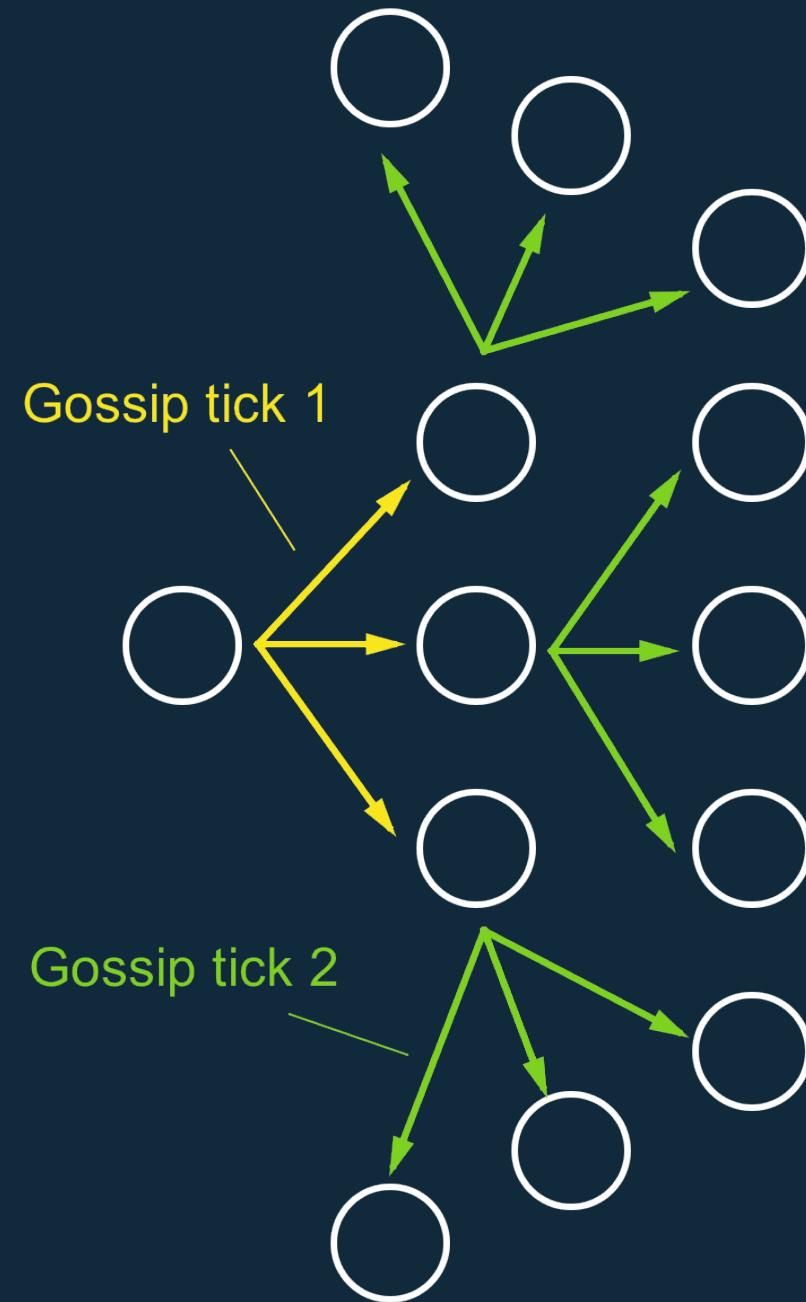
- High availability guaranteed via multiple manager nodes
- All decisions made by majority consent
- Consequence: must have odd number of managers

Note: all management plane communications are mutually TLS encrypted by default.



SWARM CONTROL PLANE

Control plane: gossip network

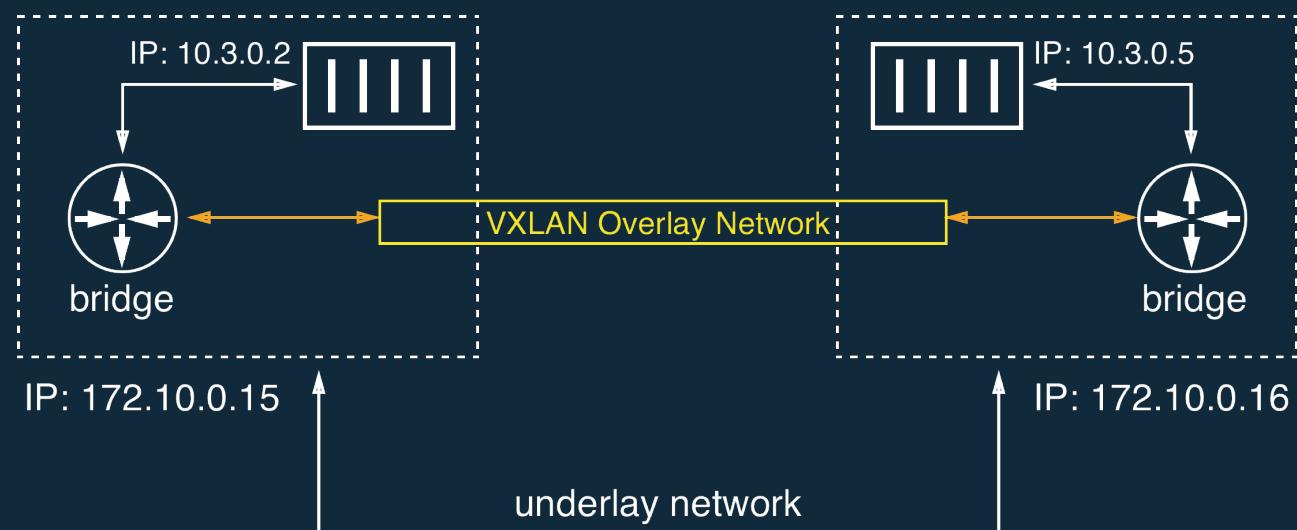


- Gossip model: random peers share knowledge of state of the cluster every tick
- Scheduling info propagates peer-to-peer (see SWIM protocol)
- Consequence: p2p control plane scales like $O(n)$ but is 'eventually consistent': must retry failed lookups.



SWARM DATA PLANE

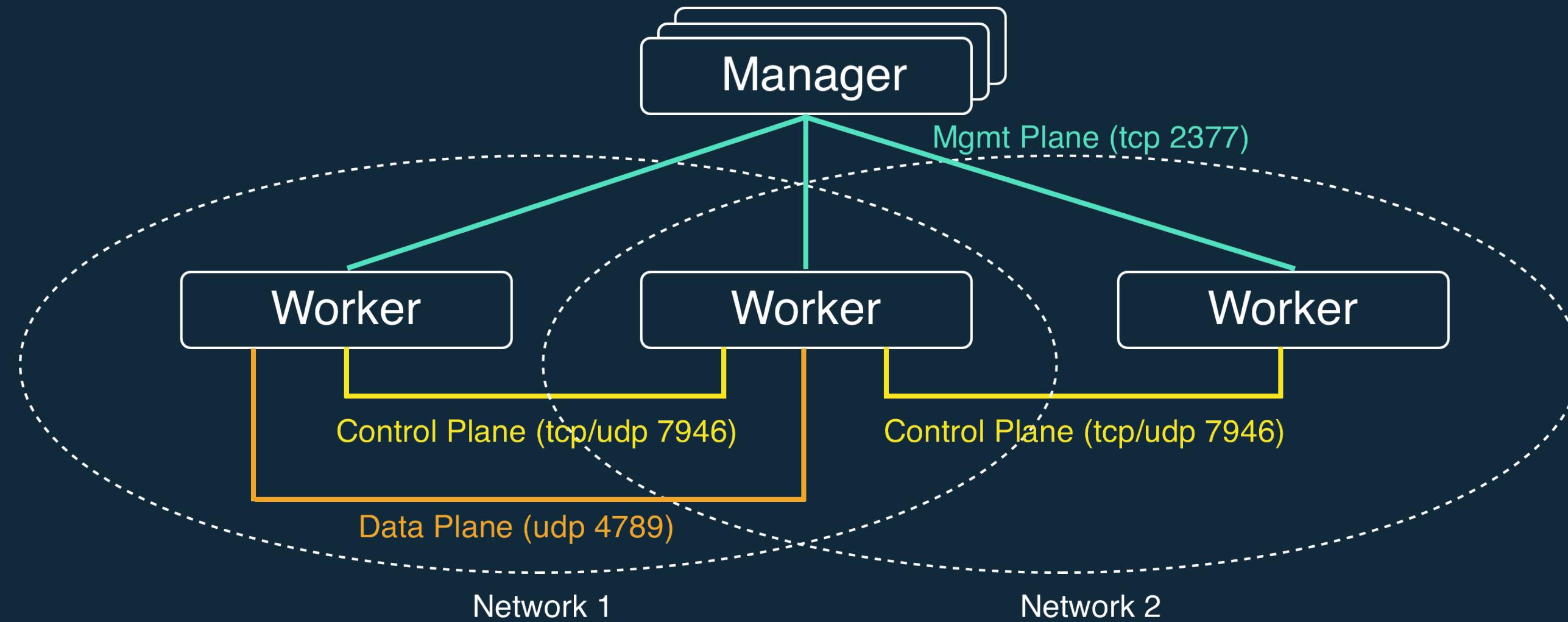
Data plane: VXLAN tunnel



- Encapsulates L2 (MAC address) packet with L3 (IP) header
- Based on UDP (not TCP): no dropped packet guarantees



SWARM NETWORK TOPOLOGY



Note: fixed IPs for managers, dynamic IPs OK for workers

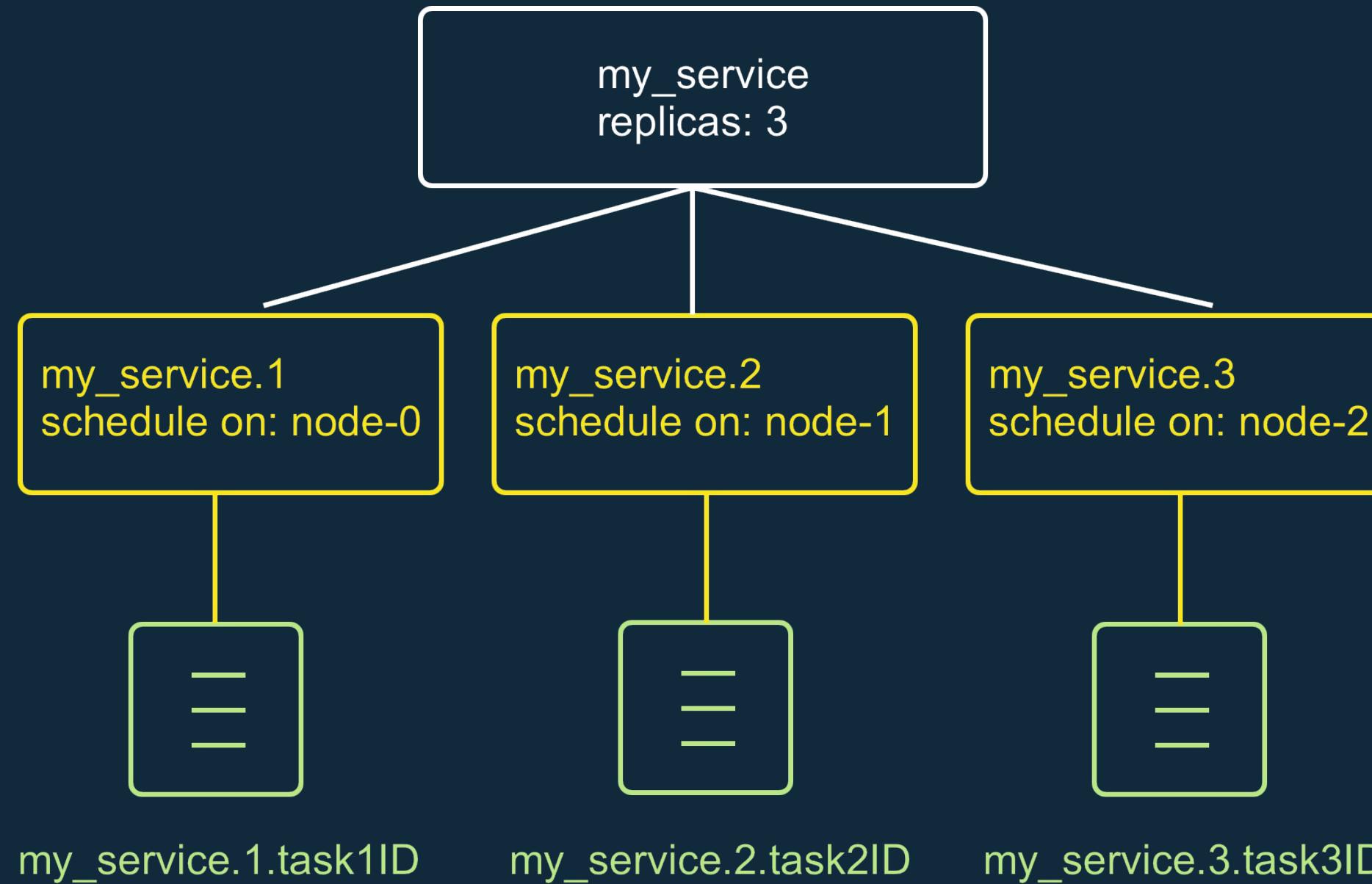


DOCKER SERVICES

- Goal: declare and (re)configure many similar containers all at once
- Goal: scale apps by adding containers seamlessly
- A service defines the **desired state** of a group of identically configured containers.
- Docker provides **transparent service discovery** for Services.



SERVICES VS. TASKS VS. CONTAINERS



services:
docker service ls

tasks:
docker service ps \
my_service

containers:
docker container ls





INSTRUCTOR DEMO: SELF-HEALING SWARM

See the demo:

- Self-Healing Swarm

In the Docker Fundamentals Exercises book.





EXERCISE: SWARMS & SERVICES

Work through:

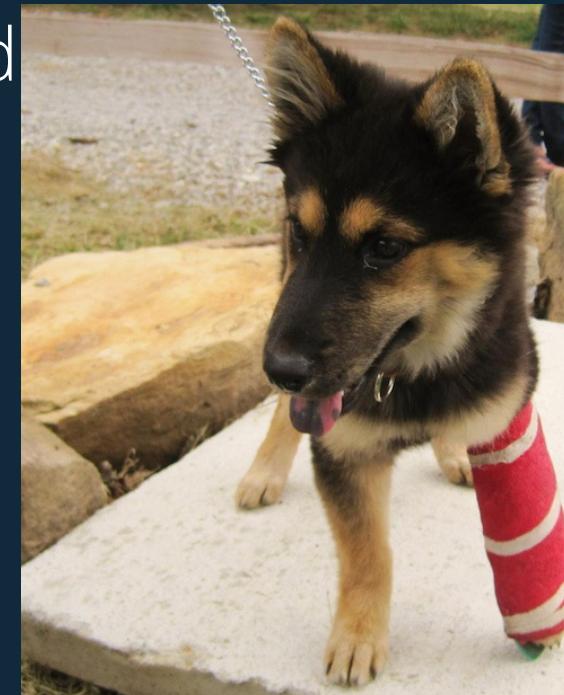
- Creating a Swarm
- Starting a Service
- Node Failure Recovery

in the Docker Fundamentals Exercises book.



PETS VERSUS LIVESTOCK

- Swarm reschedules exited containers automatically
- When a container becomes unhealthy, **kill** it and get a new one.



VMs



Containers

Dog photo [jeffreyw](#); Livestock photo [Paul Asman, Jill Lenoble](#); images CC-BY 2.0



SCHEDULING CONTROL

Ops may want to influence container scheduling decisions to accommodate things like:

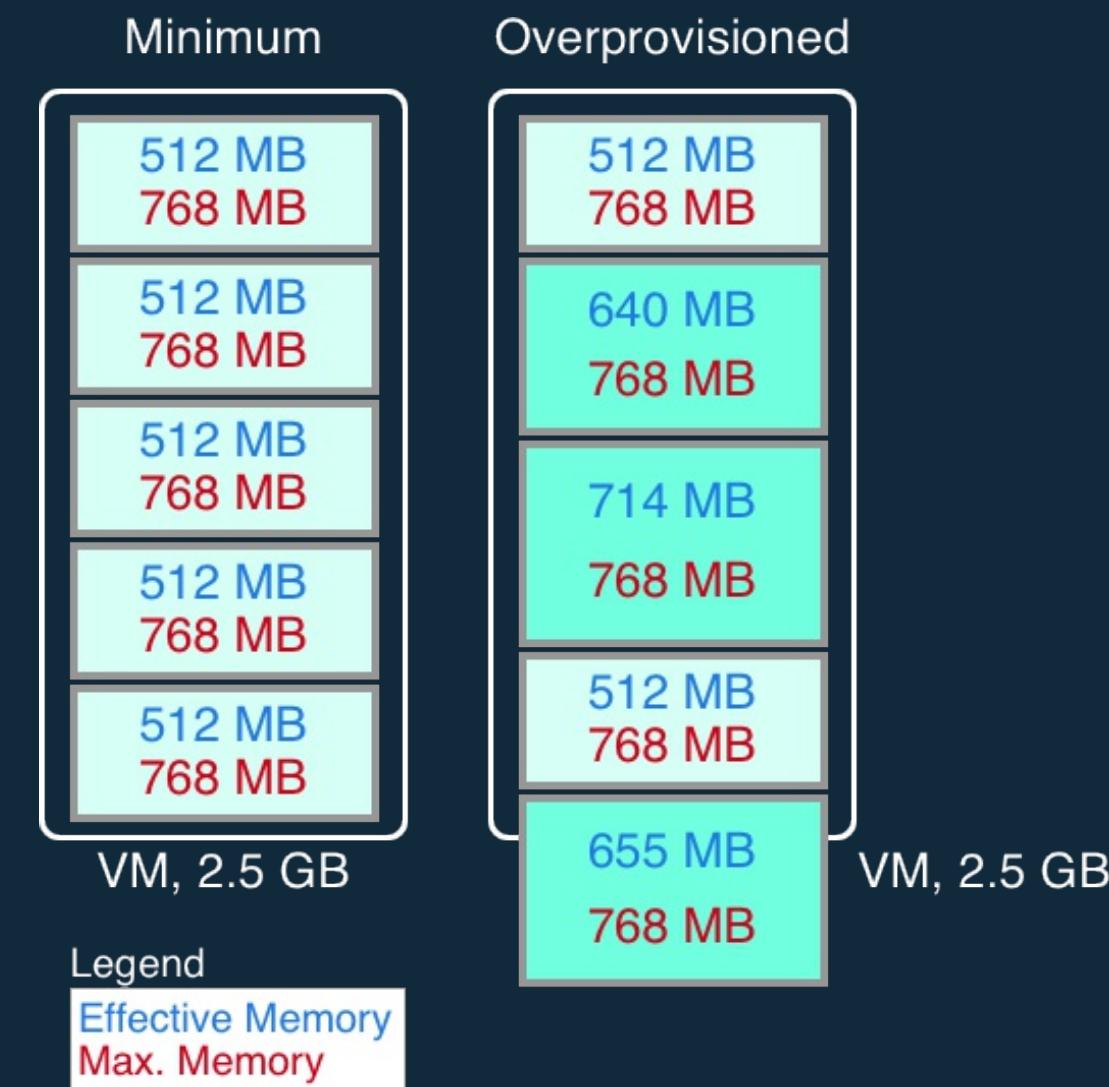
- Resource constraints
- Hardware requirements
- Reserved / decommissioned nodes
- Daemon-type services



RESOURCE-AWARE SCHEDULING

A typical scenario:

- Nearly all node memory consumed
- Memory overconsumption causes one node failure
- Containers from failed node are rescheduled elsewhere
- New hosts now overprovisioned, crash
- Result: cascading cluster failure



RESOURCE-AWARE SCHEDULING

- CPU and memory allocation controls provide two possible constraints:
 - **reservation**: amount of resources to reserve for this container
 - **limit**: do not allow container to consume more than this
- limit \leq reservation to avoid overprovisioning after scheduling has completed
- See **docker system info** for total allocatable resources
- Best practice: always specify resource constraints



NODE LABELING

- Node label: arbitrary key/value pairs applied to nodes
- A node can bear any number of labels
- Scheduling options:
 - Hard requirement: service must schedule containers on nodes with a particular label (ex: hardware requirements)
 - Topological spread: service will be spread out across all values of a given label (ex: spread containers out across racks, voltage zones etc)



REPLICATED VS. GLOBAL SCHEDULING

- Replicated: service requests a scalable number of replicas, scheduled according to flexible constraints (exactly what we've seen so far)
- Global: exactly one container runs on every node (appropriate for daemon-like services, ex. logging, monitoring, disk management)





EXERCISE: SWARM SCHEDULING

Work through:

- Swarm Scheduling
in the Docker Fundamentals Exercises book.



PROVISIONING CONFIGURATION

- Configuration changes between environments
- Want to provision **modular** configuration at run time
- Solutions:
 - **.env** files: environment variables
 - docker config: plaintext config files
 - docker secrets: encrypted config files



LOW-SECURITY CONFIGURATIONS

- .env files:
 - list of variables and values in a text file
 - provisioned to the service on creation
 - declared as environment variables in each running container for that service
- docker config:
 - File stored in Swarm's raft database
 - provisioned to service as a 'config' object
 - File appears at specified path in every service container



HIGH-SECURITY CONFIG: DOCKER SECRETS

- Need to provision secure info to containers: passwords, keys, access tokens
- Must have this info available to managers
- But, NEVER want this information sitting or transmitted unencrypted on host disk.
- Solution: docker secrets
 - Encrypted on managers in raft db
 - Encrypted in transit to workers by Swarm's default mutual TLS
 - Mounted from in-memory tmpfs on worker, not disk (linux only), to files in container filesystem



DOCKER STACKS

- yaml manifest describing all elements of application (services, configs, secrets, networks, volumes etc.)
- Stand up & tear down an app in one command
- Superset of the docker-compose specification: <https://dockr.ly/2iHUpex>



STACK FILES

```
version: "3.7"

services:
  db:
    image: postgres:9.6
    env_file:
      - myvars.env
    configs:
      - source: initscript
        target: /docker-entrypoint-initdb.d/init.sh
    secrets:
      - password
    environment:
      - POSTGRES_PASSWORD_FILE=/run/secrets/password

  configs:
    initscript:
      file: ./db-init.sh

  secrets:
    password:
      external: true
```

- Top level keys: basic Docker objects
- Second-level keys: instances of objects
- nth-level keys: properties of parent object





EXERCISE: SWARM CONFIGS

Work through:

- Provisioning Swarm Configuration
in the Docker Fundamentals Exercises book.



SERVICE NETWORKING

Two questions when networking services:

- Is the destination service **stateful** or **stateless**?
- Is the request origin **internal** or **external** to your swarm?



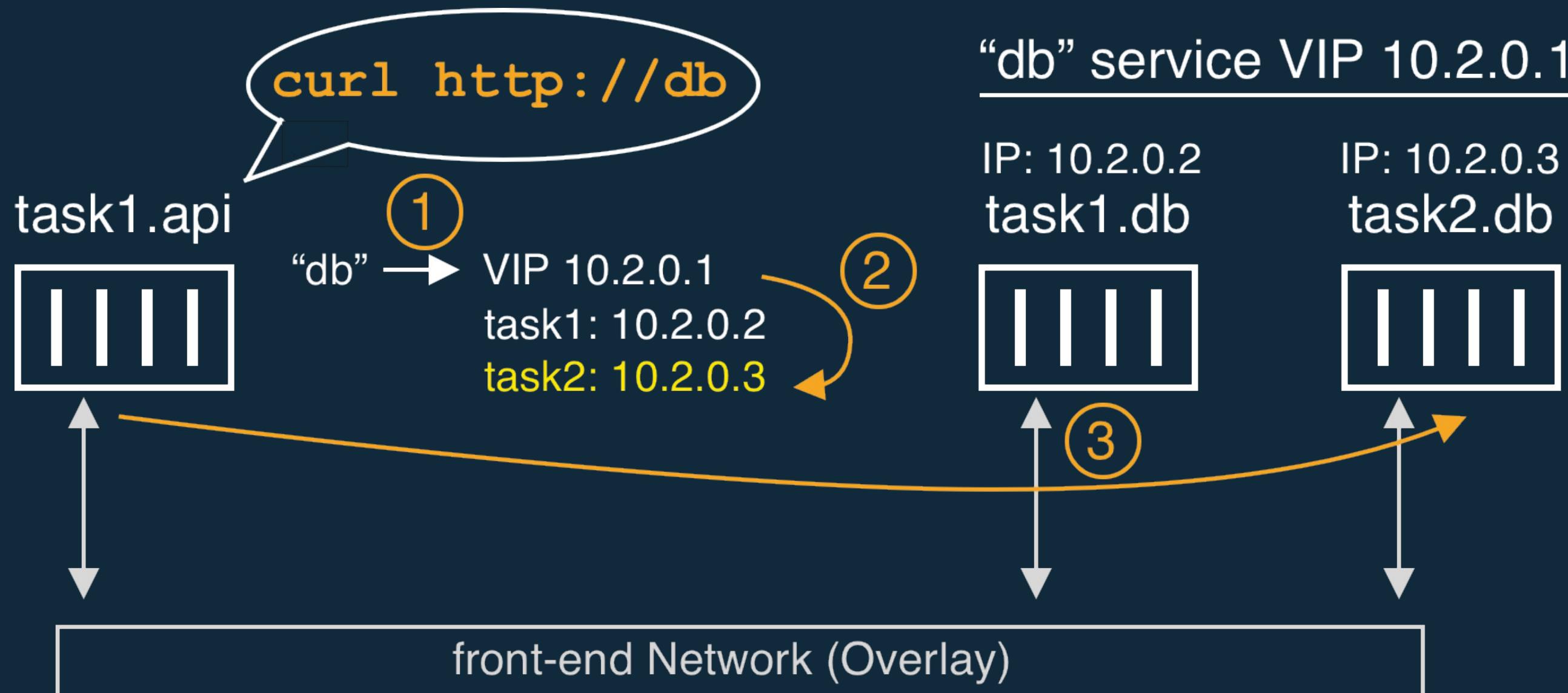
SERVICE NETWORKING

- Internal origin: DNS lookup
- External origin: Port mapping
- Stateless destination: VIP routing
- Stateful destination: container IP routing



INTERNAL / STATELESS

Solution: DNS + VIP

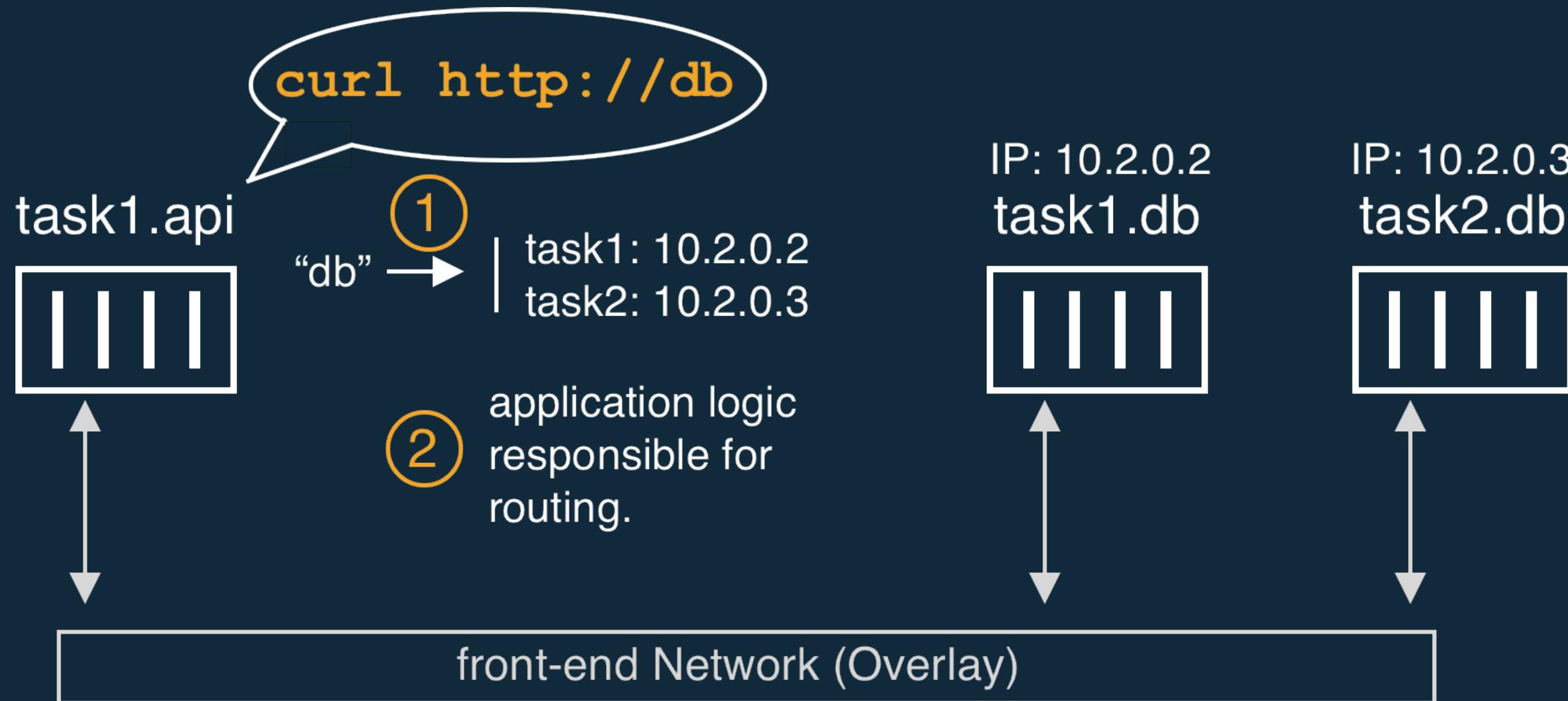


```
docker service create --name db --replicas 2 \
--network front-end mydb:latest
```



INTERNAL / STATEFUL

Solution: DNSRR Endpoints

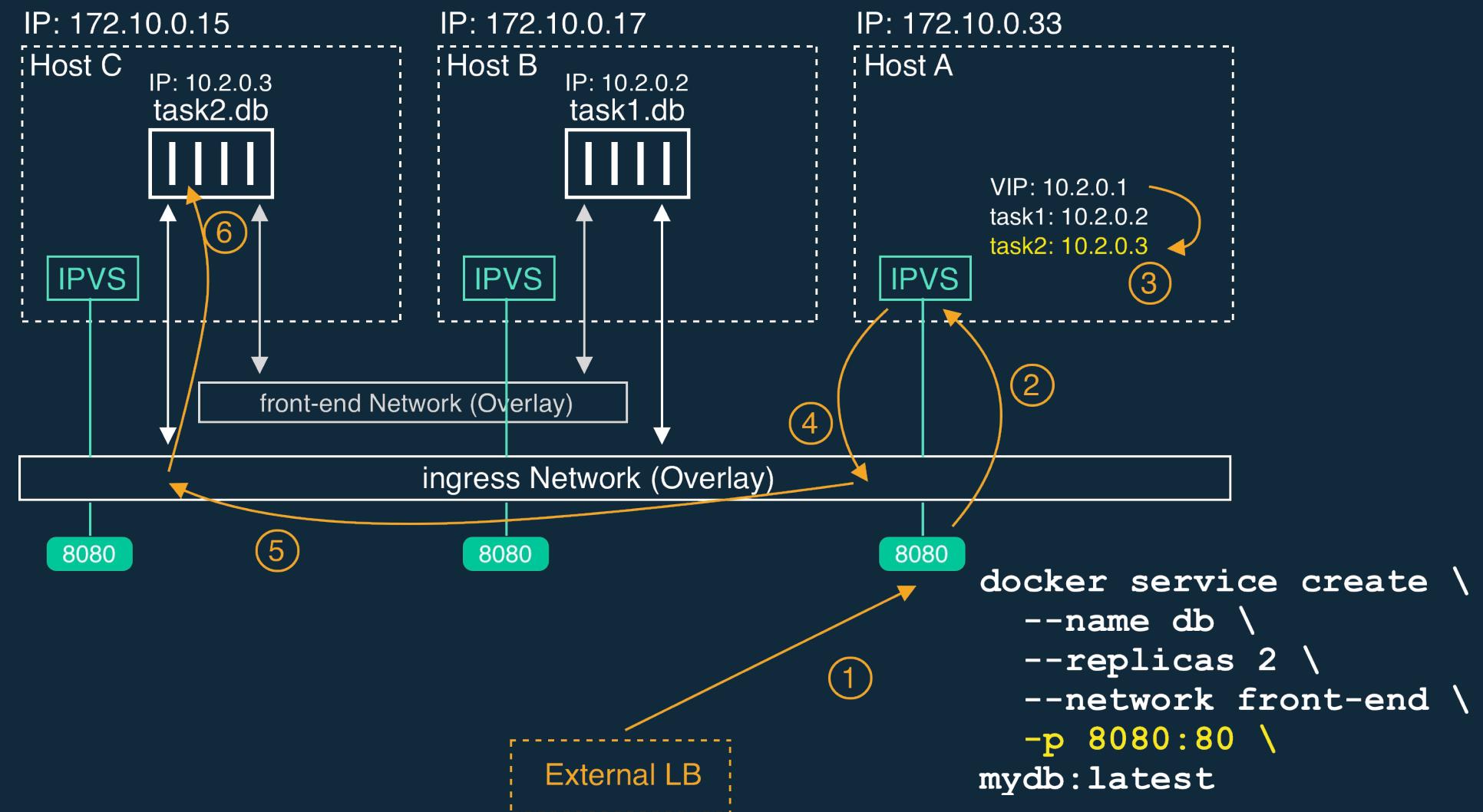


```
docker service create --name db --replicas 2 \
--network front-end --endpoint-mode dnsrr mydb:latest
```



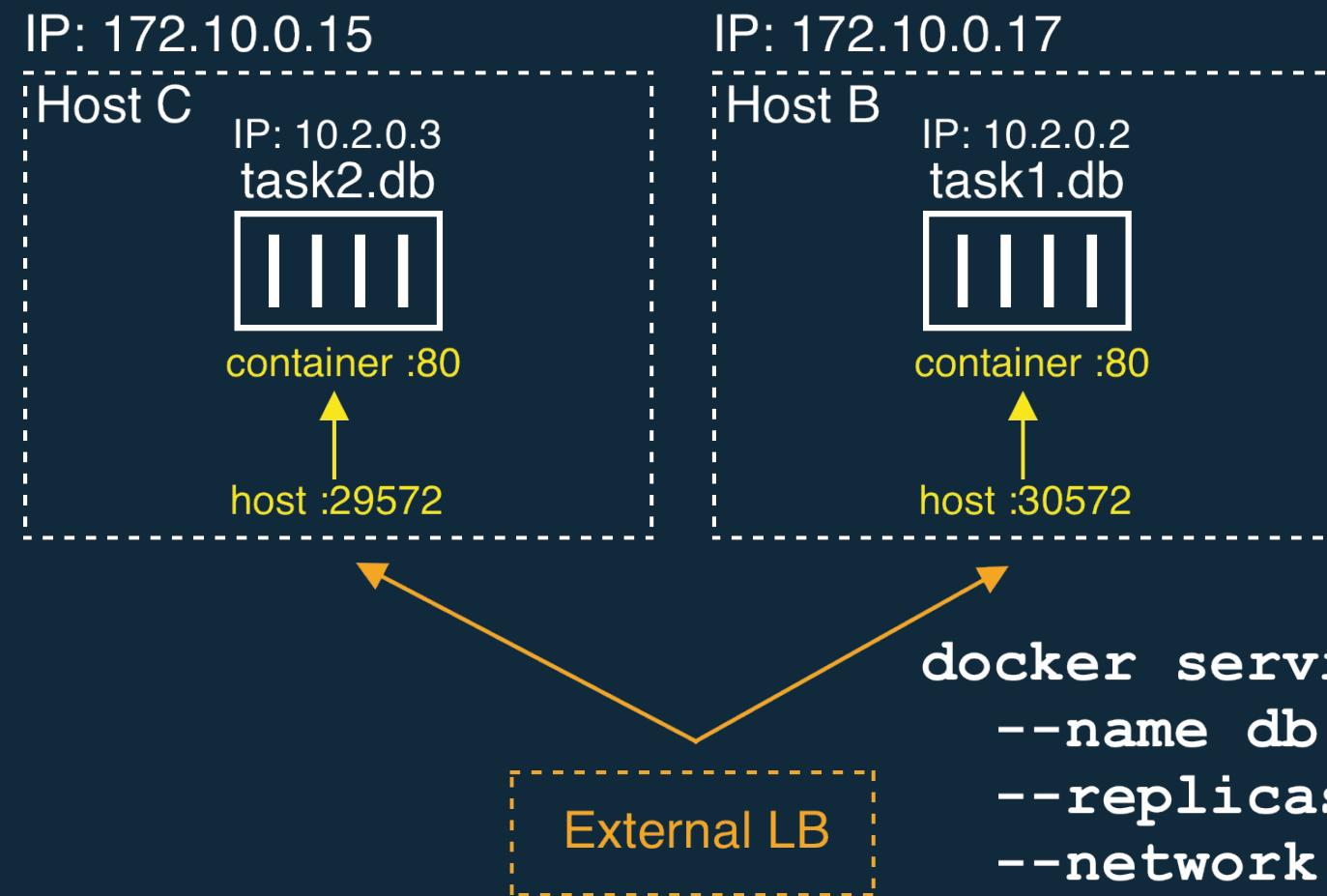
EXTERNAL / STATELESS

Solution: Swarm L4 Routing Mesh



EXTERNAL / STATEFUL

Solution: **--publish mode=host**



```
docker service create \
--name db \
--replicas 2 \
--network front-end \
--publish mode=host,target=80 \
mydb:latest
```





EXERCISE: SWARM ROUTING

Work through:

- Routing to Services

in the Docker Fundamentals Exercises book.



MANAGING APPLICATIONS

We have all the components we need to deploy a full application on Swarm, but:

- How will we scale it?
- How will we roll out new software versions without downtime?
- How can we automatically and gracefully roll back failed updates?



OUR APPLICATION: DOCKERCOINS



(DockerCoins 2016 logo courtesy of [@XtlCnslt](#) and [@ndeloof](#). Thanks!)

- It is a DockerCoin miner!
- Dockercoins consists of 5 services working together:



MANAGING MICROSERVICES

- Can scale individual, bottlenecked components
 - Spend resources exactly where needed
 - But, need monitoring hooks to assess specific bottlenecks
- Can upgrade components 'independently'
 - Patches and upgrades to one component don't get blocked by other teams' release cycles
 - Still have to be careful that interface between components is stable or backwards compatible
- Not building for microservices? Can still take advantage of all scaling, roll-out and roll-back features of any orchestrator.



SERVICE UPDATE & ROLLBACK

- Rolling updates / rollback
- Configurable parallelism, delay
- Auto-trigger rollbacks on deployment failure
- Replacement container start / stop order





EXERCISE: SWARM APPLICATION UPGRADES

Work through:

- Updating Applications
in the Docker Fundamentals Exercises book.



SWARM MODE TAKEAWAYS

- Orchestration facilitates treating datacenter as a pool of compute for containers
- Applications defined as stacks consisting of modular components: services, configs, secrets, networks etc
- Swarm networking planes build in default mTLS encryption for managers, simple DNS-based service discovery for applications
- Rich set of operational control features for scheduling control, resource allocation, and application upgrades



FURTHER READING

- Getting started with Docker swarm: <http://dockr.ly/2vUFWTA>
- Swarm mode overview: <http://dockr.ly/2jh11Vd>
- Get started with multi-host networking: <http://dockr.ly/2gtqRms>
- Manage sensitive data with Docker secrets: <http://dockr.ly/2vUNbuH>
- Stack file reference: <https://dockr.ly/2iHUpex>





INTRODUCTION TO KUBERNETES



DISCUSSION: CONTAINER COMMUNICATION

Suppose you have two services you want to guarantee can reach each other on the same host, to eliminate network latency. How would you do this in Swarm?



LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Understand the components and roles of Kubernetes masters and nodes
- Identify and explain the core Kubernetes objects (Pod, ReplicaSet, Deployment, Service, Volumes)
- Explore the Kubernetes networking model
- Contrast Kubernetes and Swarm



KUBERNETES VS. SWARM

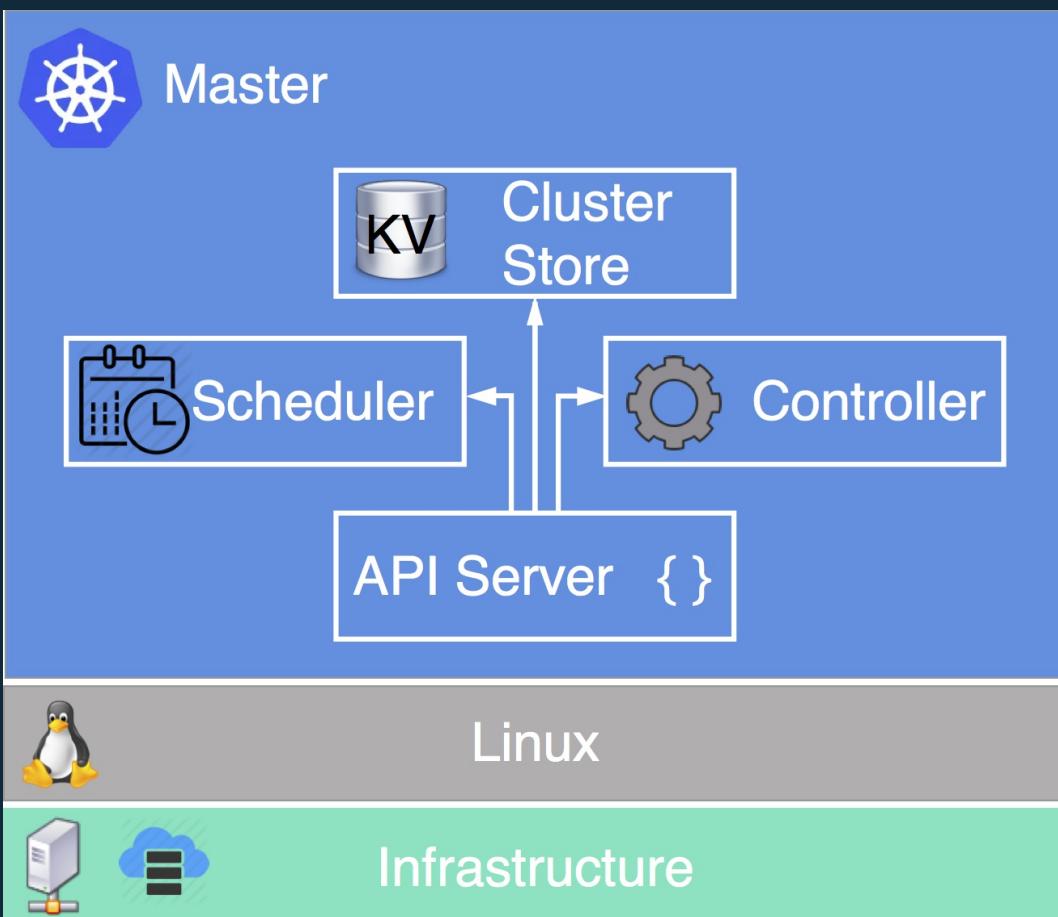
- Kubernetes: modular, prioritizes communication model
- Swarm: monolithic, prioritizes security model



KUBERNETES MASTER

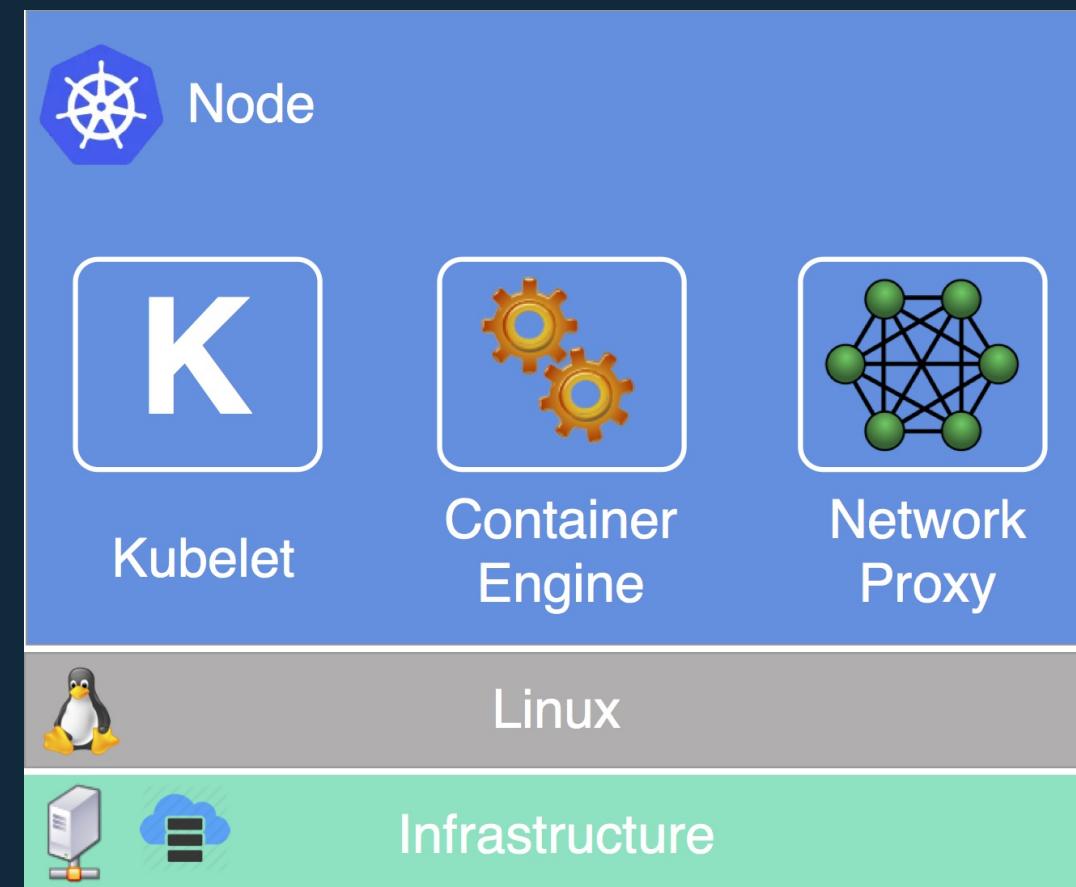
Important Components

- **API Server:** Frontend into Kubernetes control plane
- **Cluster Store:** Config and state of cluster
- **Controller Manager:** Assert desired state
- **Scheduler:** Assigns workload to nodes

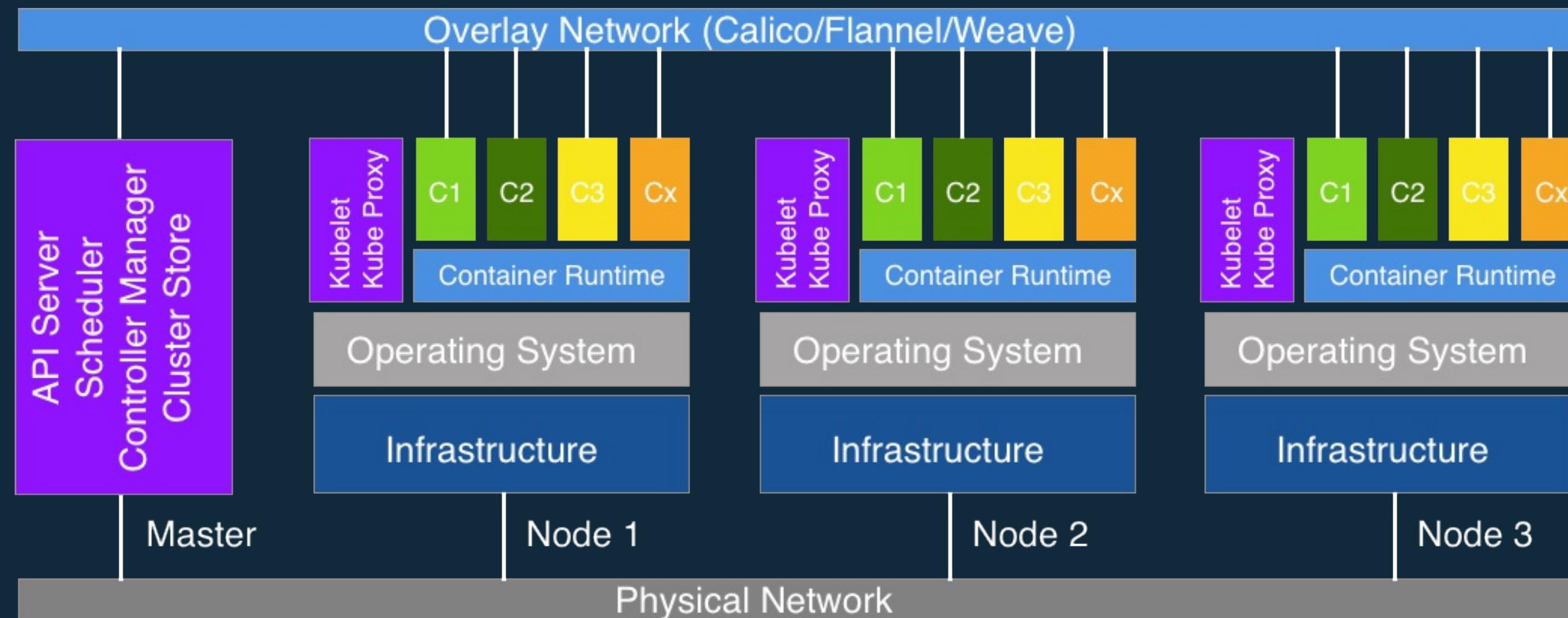


KUBERNETES NODE

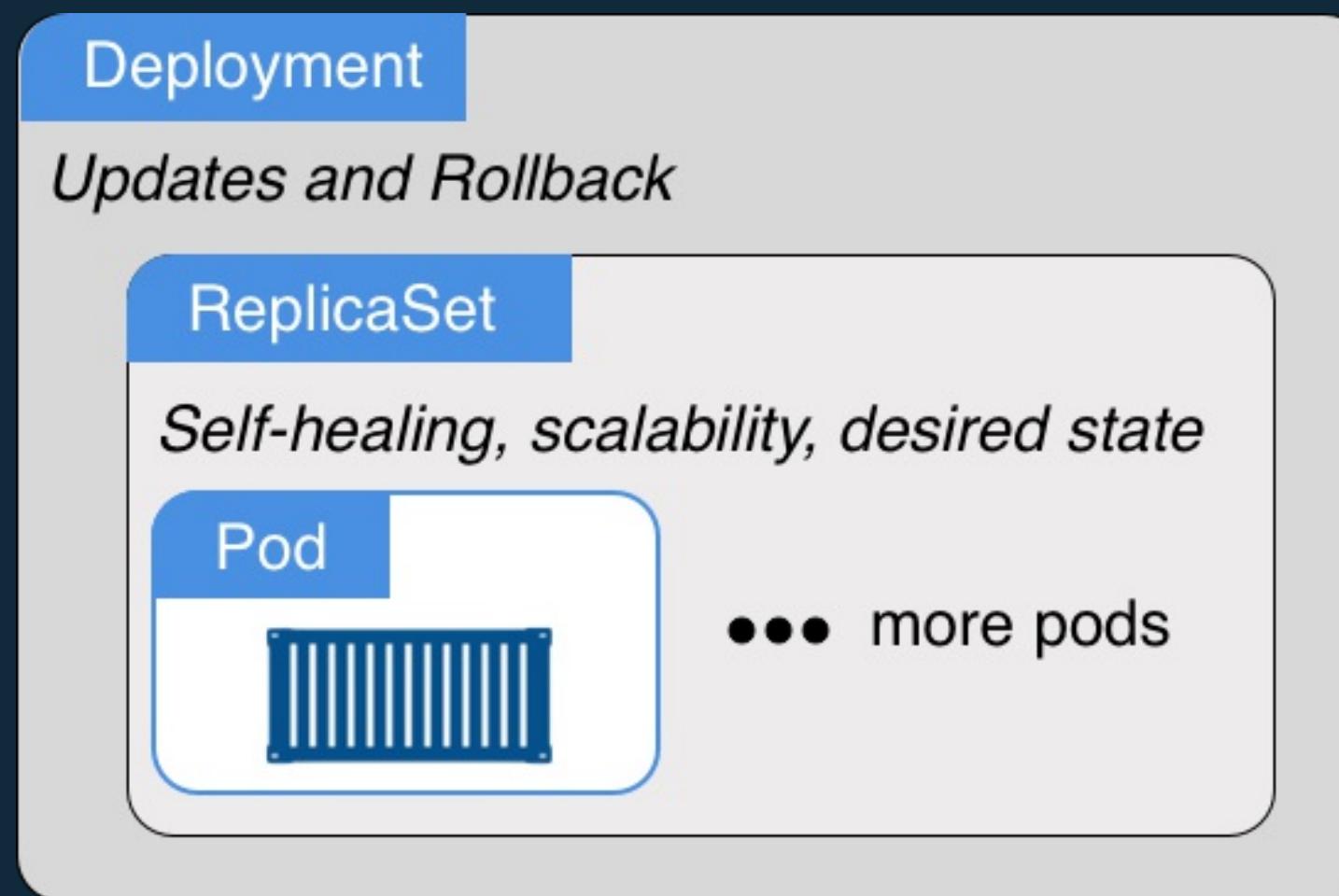
- Kubelet: Kubernetes Agent
- Container Engine: Host Containers
- Network Proxy: Networking & Load Balancing



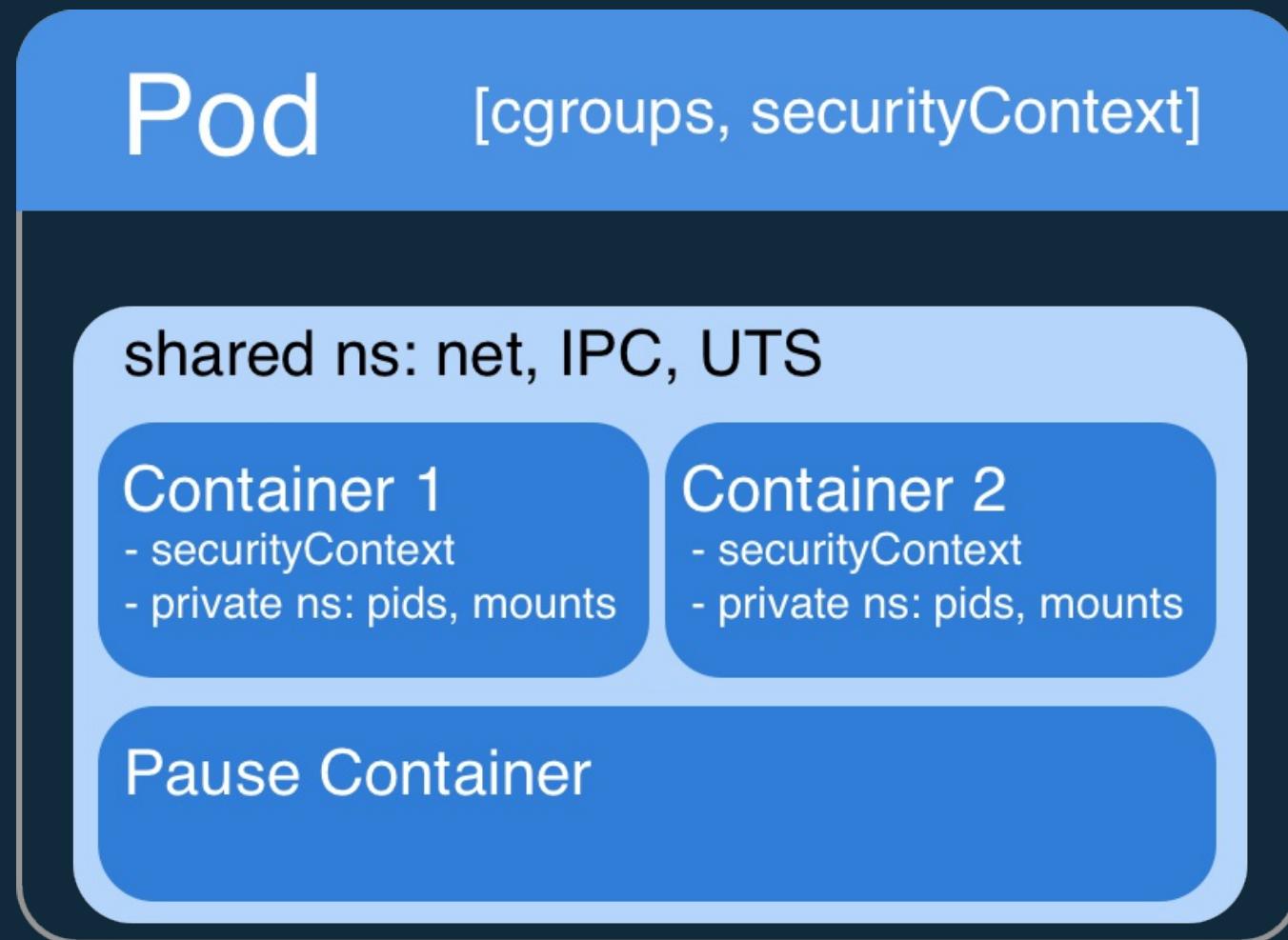
ARCHITECTURE



KUBERNETES ORCHESTRATION OBJECTS



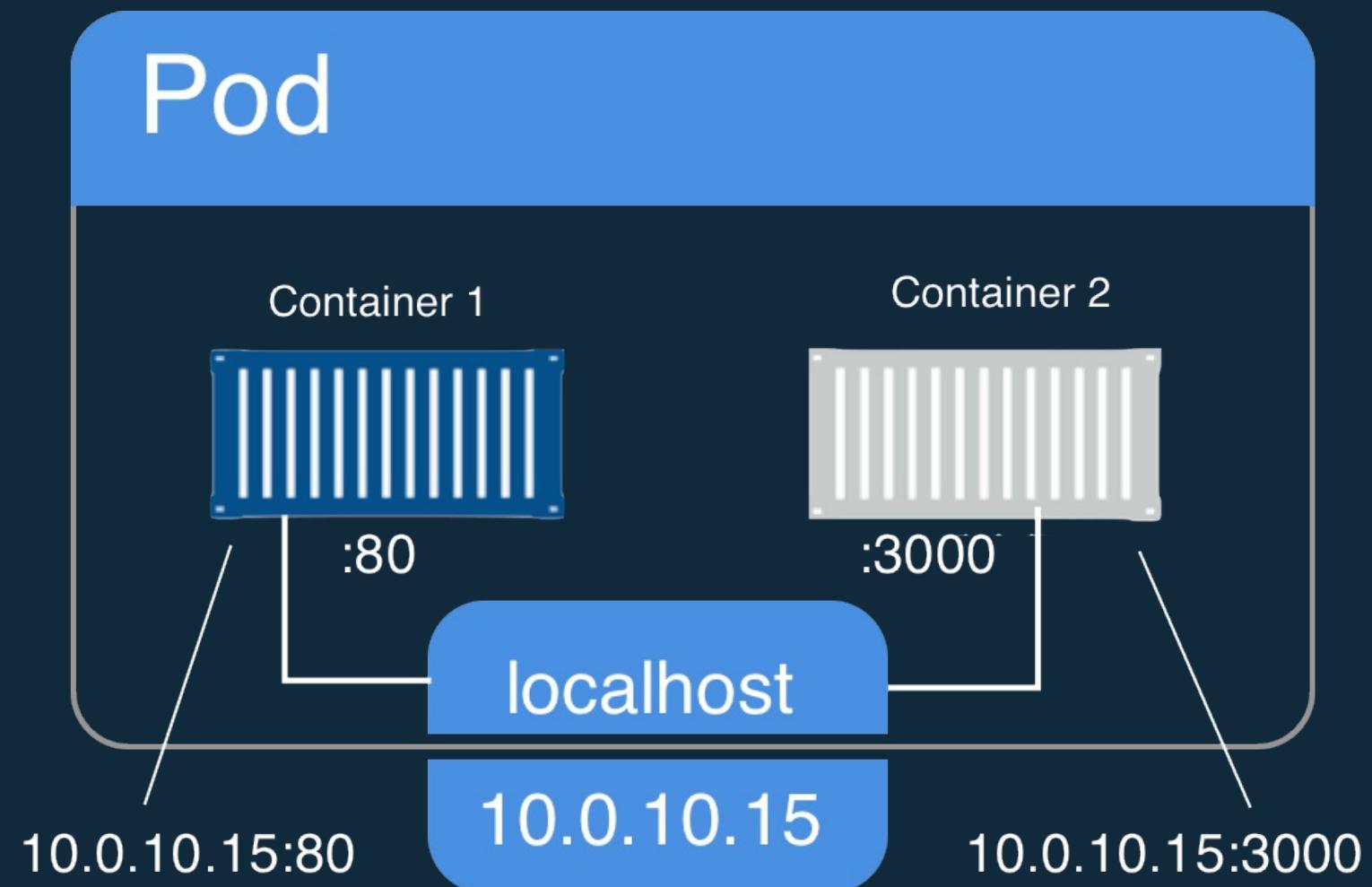
PODS



Logical host supporting interacting processes



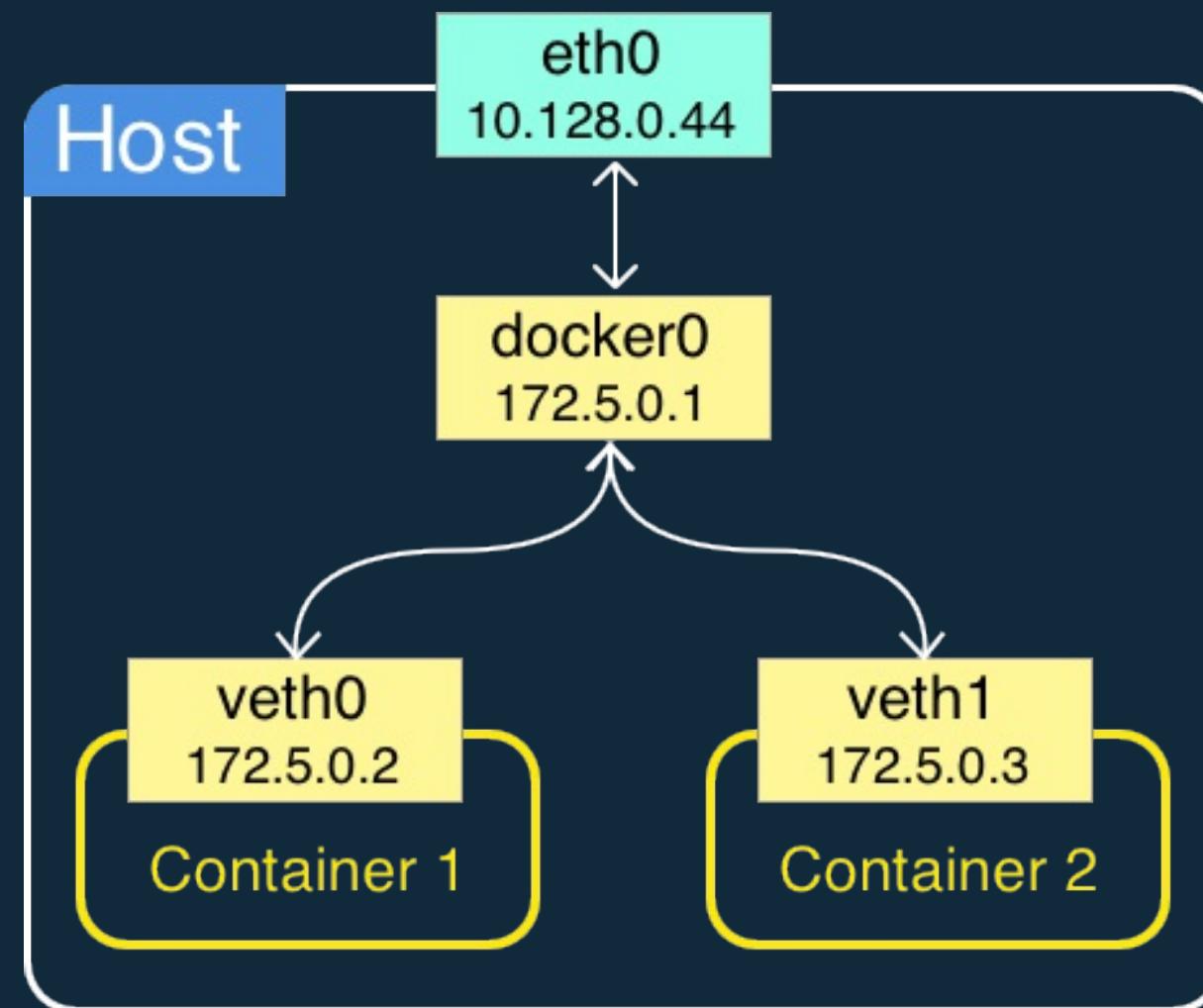
PODS



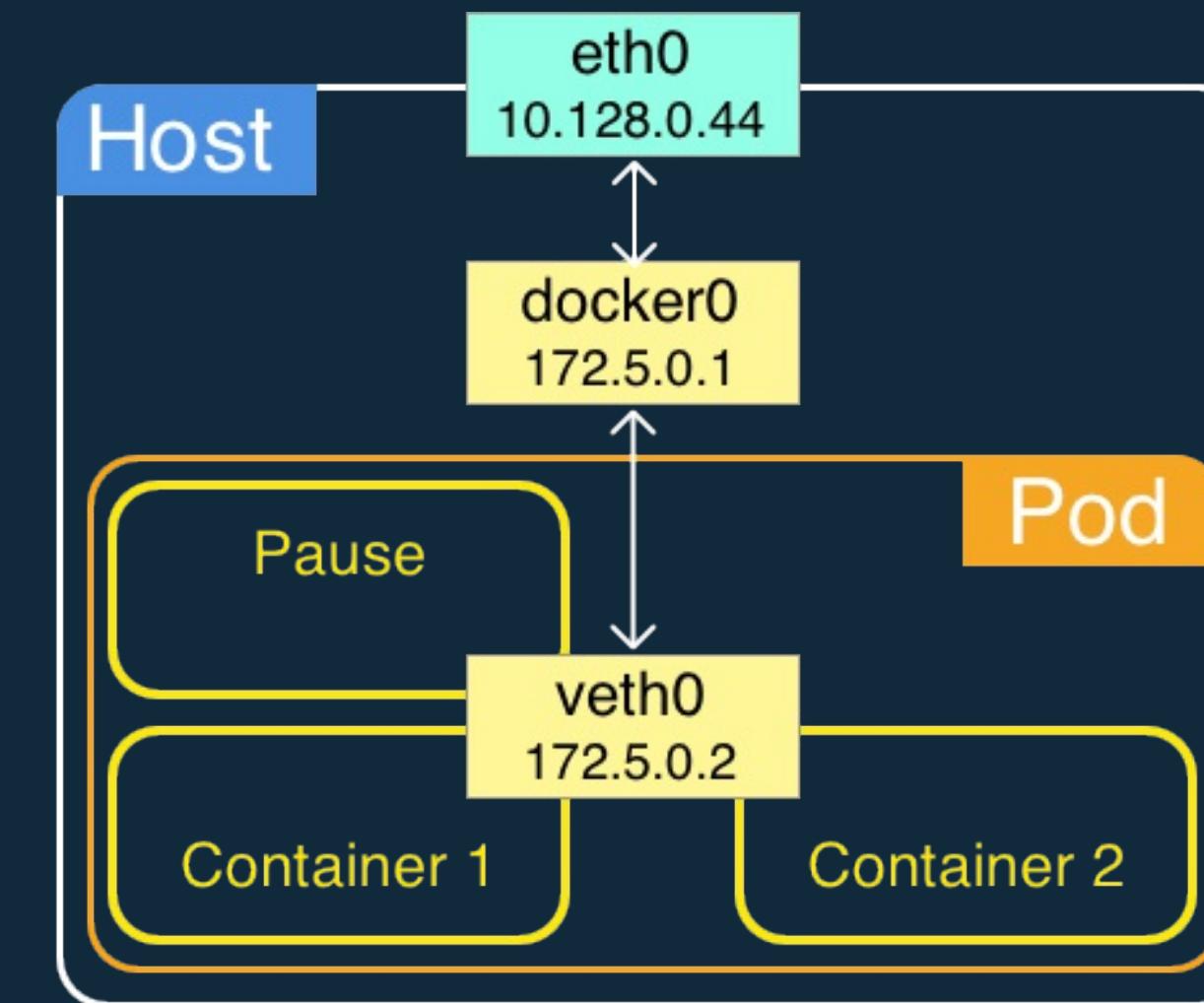
- Use **localhost** for intra-pod communication
- All containers in a pod share same IP



POD NETWORKING



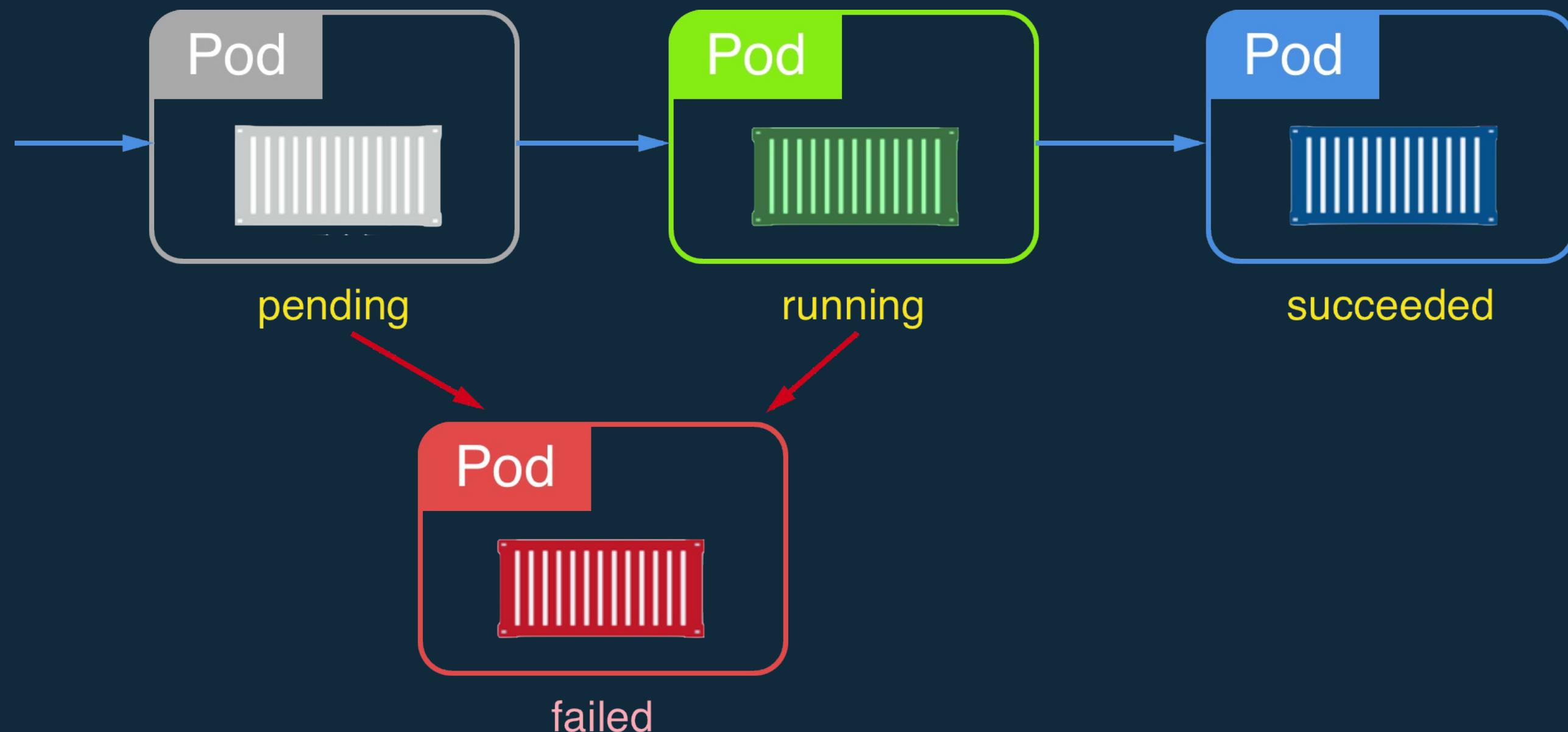
Docker Bridge Network



Kubernetes Network



POD LIFECYCLE





INSTRUCTOR DEMO: KUBERNETES BASICS

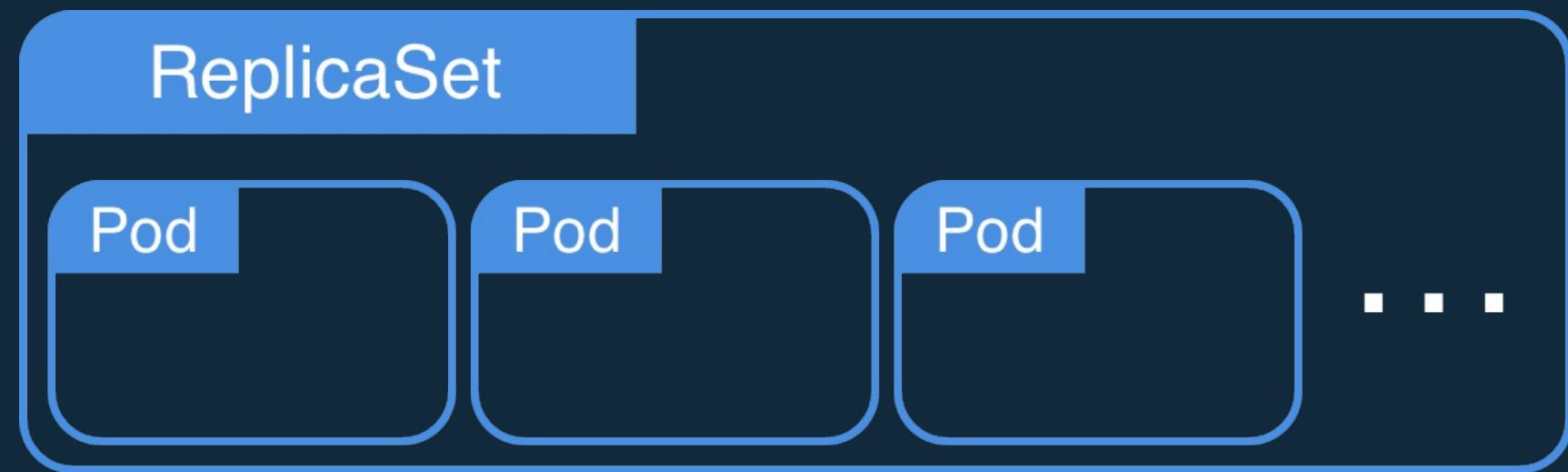
See the demo:

- Kubernetes Basics

In the Docker Fundamentals Exercises book.



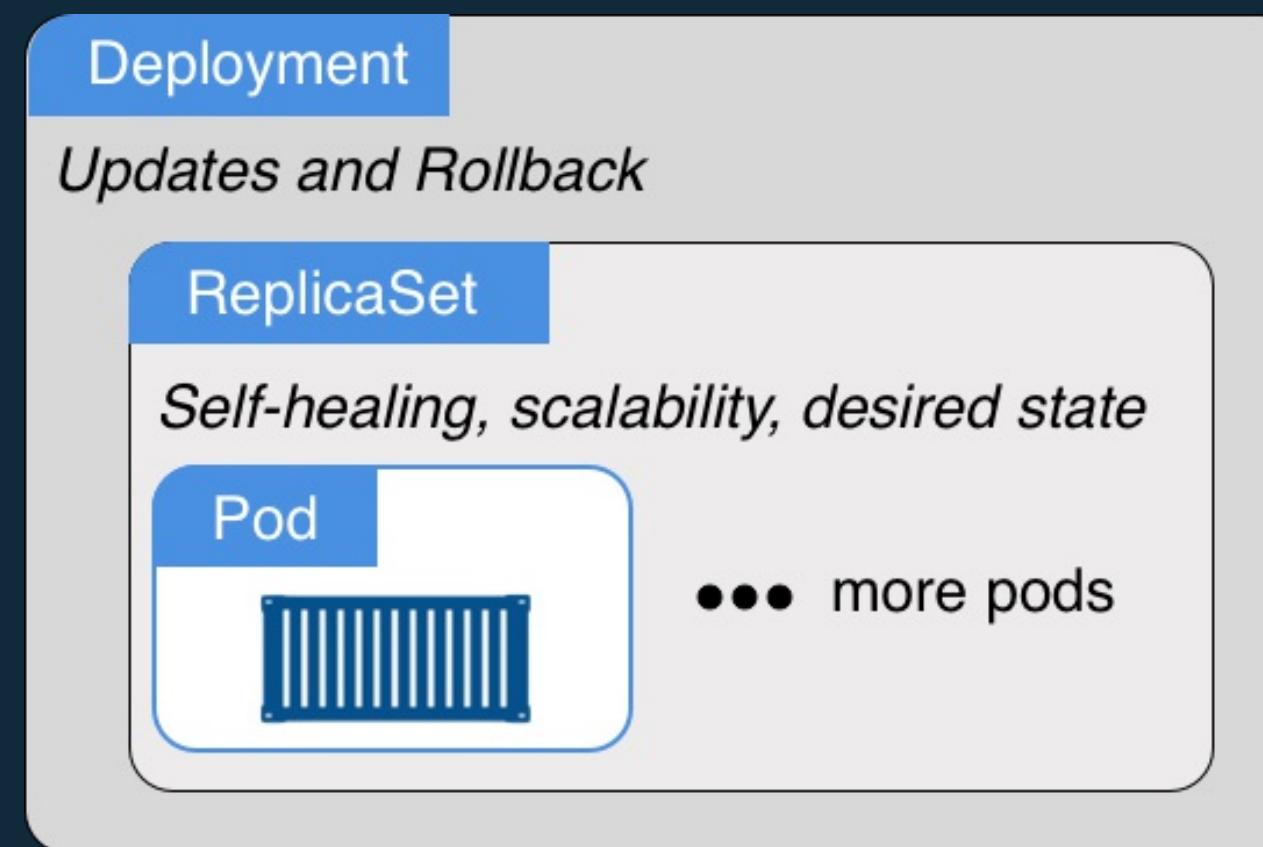
REPLICASET



- Scaling
- Keep-alive



DEPLOYMENT



- Build on top of **ReplicaSets**
- Add configurable Updates and Rollback
- Older Versions of ReplicaSets stick around for easy Rollback



CONFIGURATION

Applications typically need environment-specific config:

- Environment variables
- Configuration files
- Non-sensitive info (ports, usernames, endpoints)
- Sensitive info (passwords, access tokens, keys)

Config should be **decoupled** from pod definition and **portable** across the cluster.



CONFIGMAPS

- Collections of key/value pairs, or text files
- Provisioned to containers via env vars or volume mounts
- Appropriate for low/no security config



SECRETS

- Defined and provisioned similarly to configMaps (env vars or volume mounts)
- Intended for secure info:
 - Provisioned in a tmpfs, never written to disk
 - Encrypted by UCP on masters automatically
- Warning: secrets are recoverable with **kubectl get secrets** from masters, and potentially with **docker container inspect** from host workers



STORAGE VOLUMES

- Volumes: same lifecycle as pod (compare to persistent Docker volumes)
- PersistentVolumes: 'immortal' volume (similar to Docker)
- Storage backends exposed by container storage interface drivers





EXERCISE: KUBERNETES ORCHESTRATION

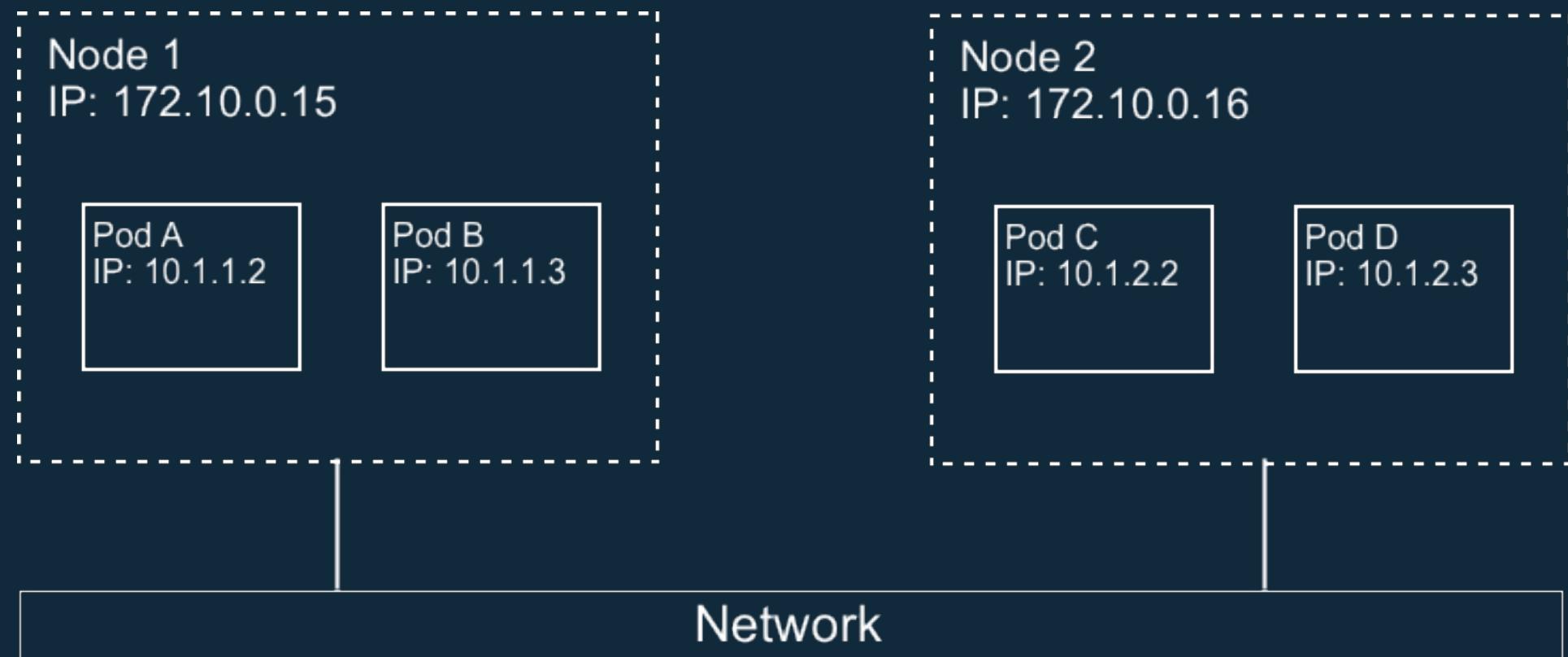
Work through:

- Installing Kubernetes
- Kubernetes Orchestration
- Provisioning Kube Config

in the Docker Fundamentals Exercises book.



KUBERNETES NETWORK MODEL



Requirements

- Pod <--> Pod without NAT
- Node <--> Pod without NAT
- Pod's peers find it at the same IP it finds itself
- Creates a **flat network**, like VMs



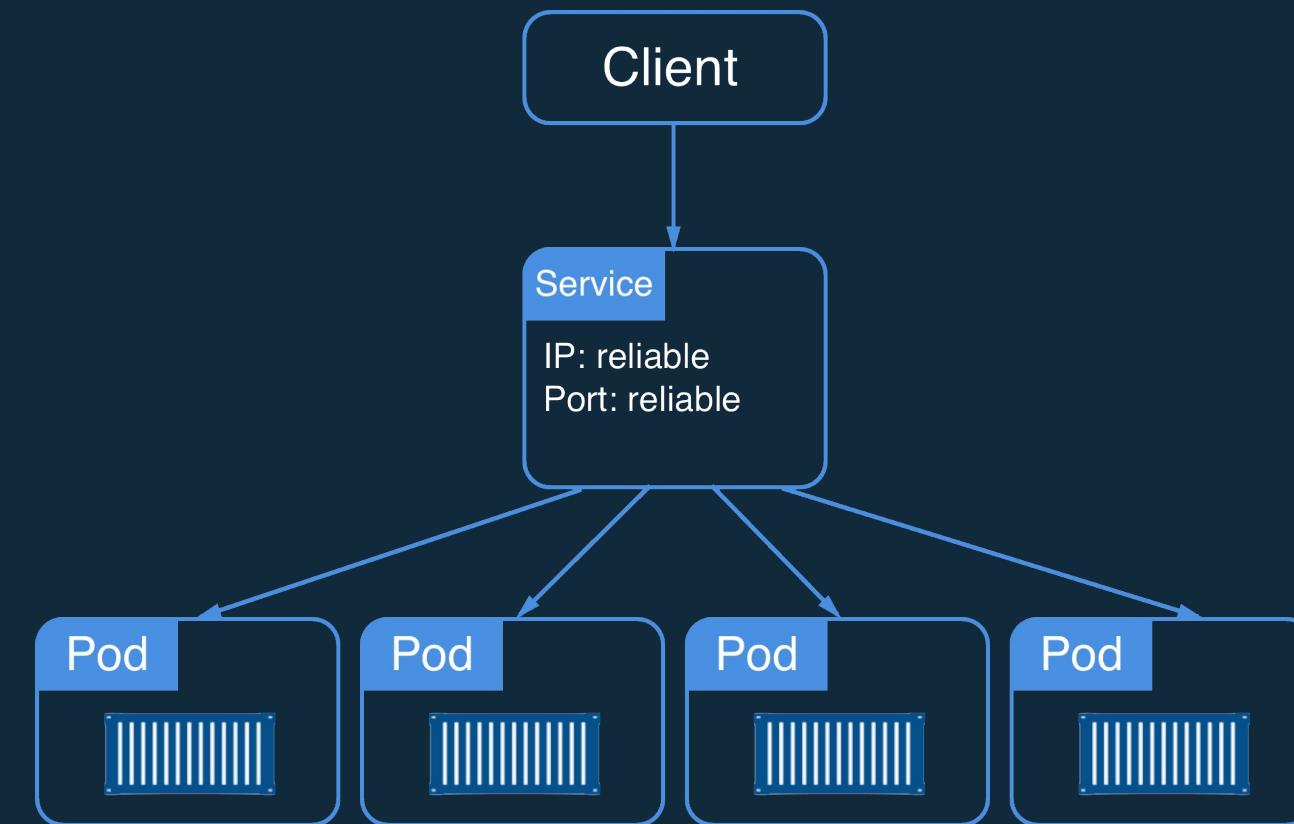
SERVICE

Problem:

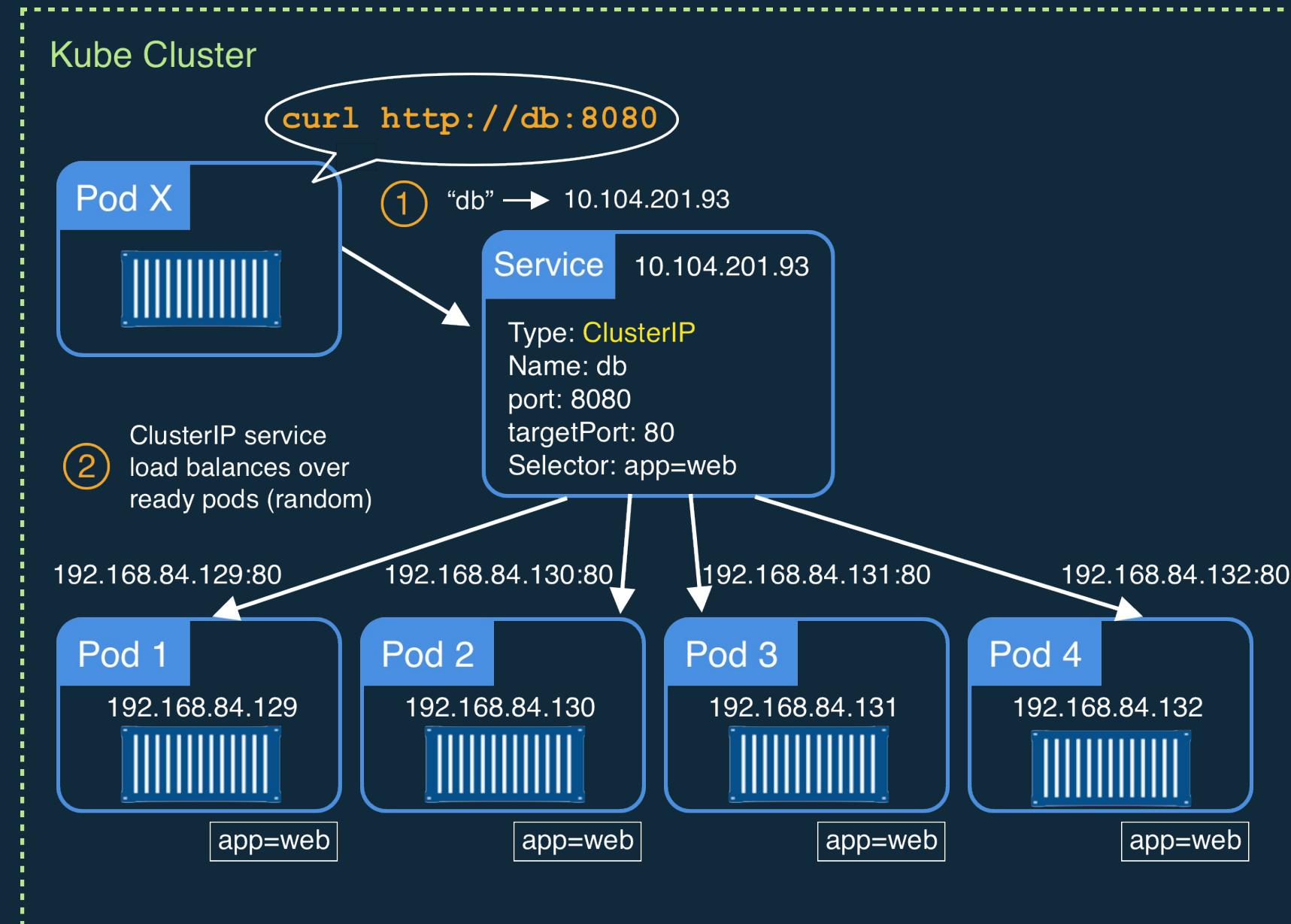
- Pods are mortal
- Pods are never resurrected
- Pod IP Addr cannot be relied on

Solution:

- Service defines:
 - Logical set of Pods
 - Policy how to access them



CLUSTERIP SERVICES

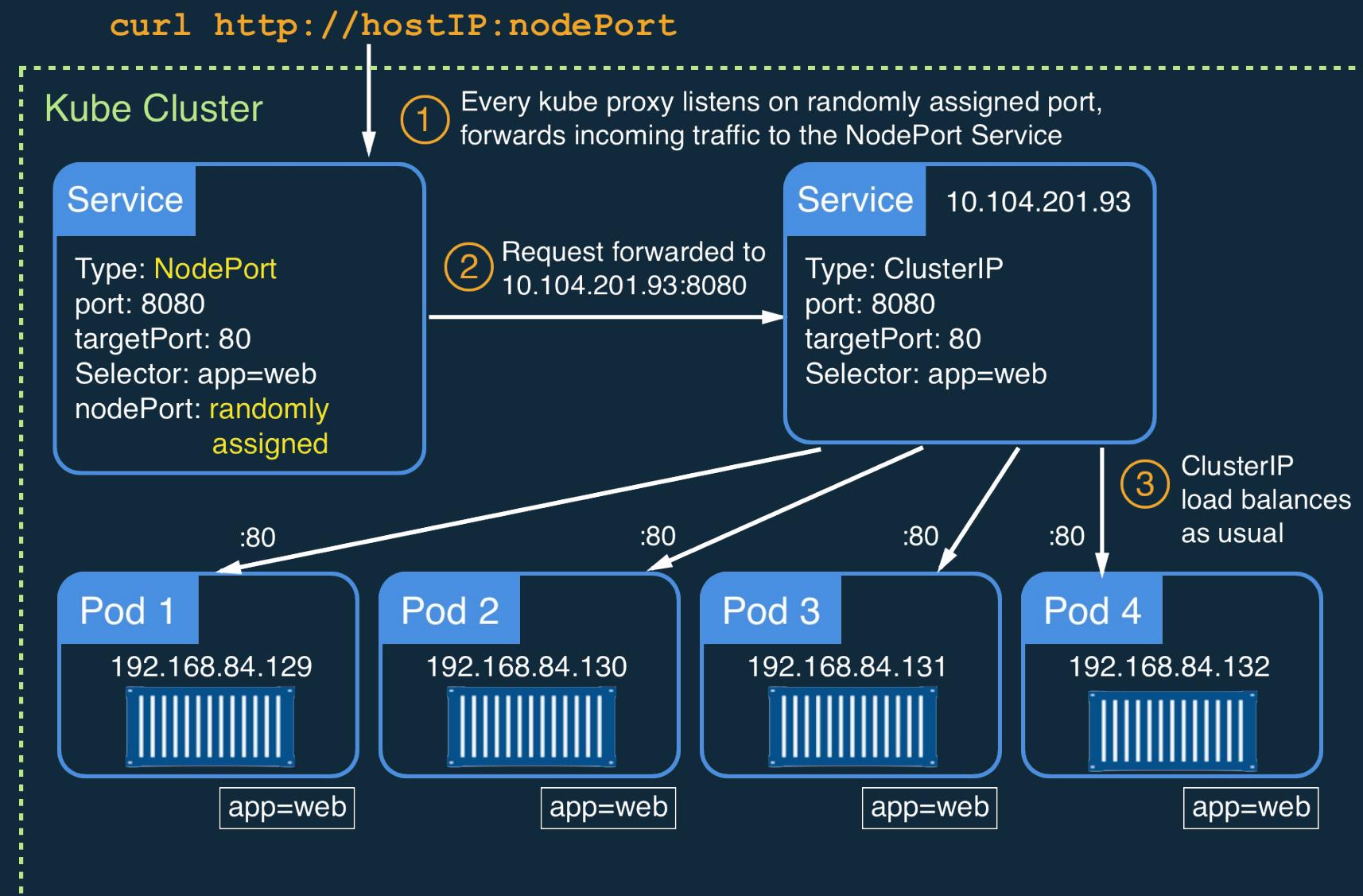


- Usecase:

- Cluster internal origin
- Stateless destination
- Similar to Swarm VIP



NODEPORT SERVICES



- Usecase:

- Cluster external origin
- Stateless destination
- Similar to Swarm L4 mesh



KUBERNETES LABEL SELECTORS

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
```

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: nginx
  ports:
    - port: 8000
      targetPort: 80
```



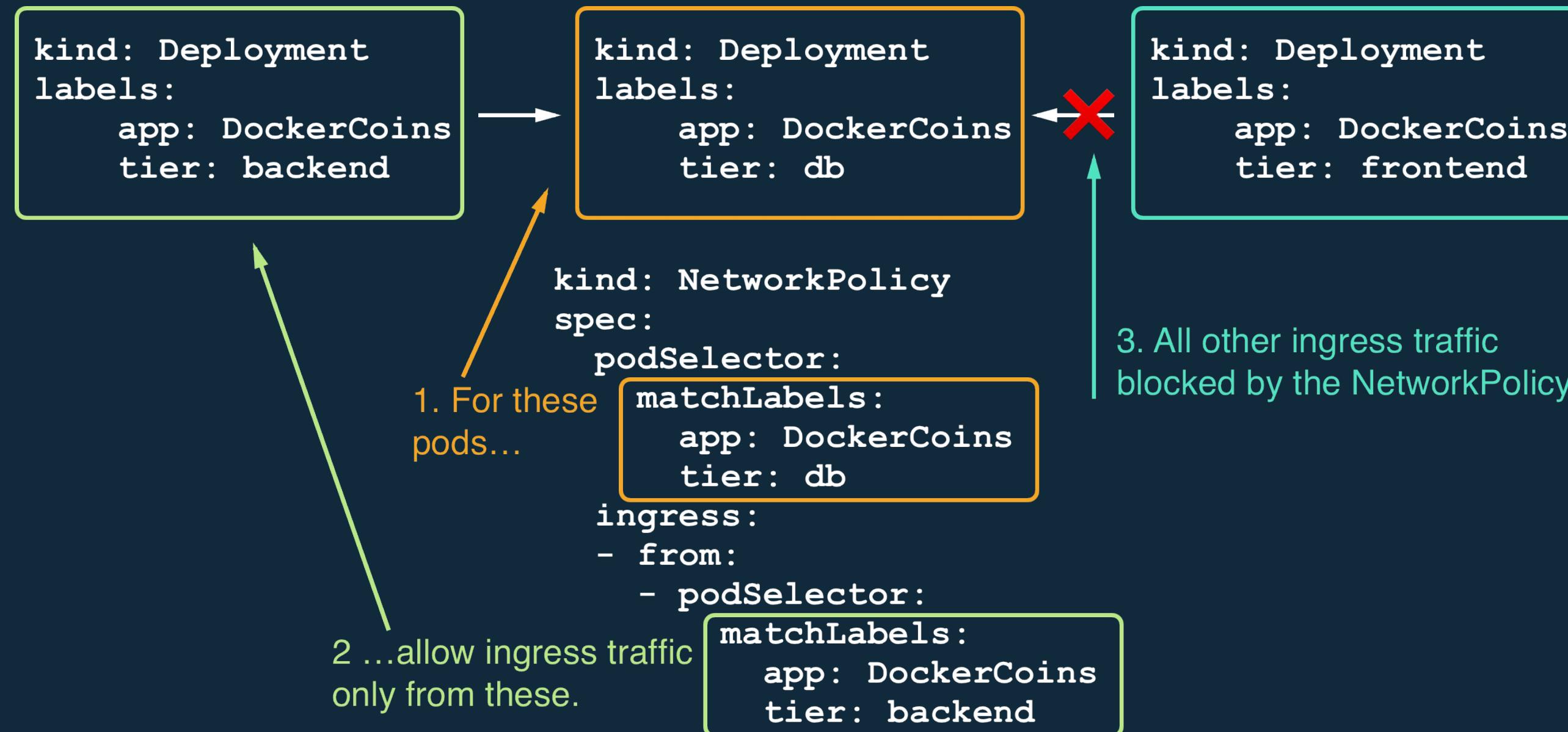
NETWORK POLICIES

- Network policies control traffic
- Traffic allowed by default
- Traffic denied if network policy exist but no rule allows it
- Independent ingress & egress rules

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ...
  namespace: ...
...
spec:
  podSelector: ...
  ingress:
    - ...
    - ...
  egress:
    - ...
    - ...
```



SAMPLE NETWORK POLICIES

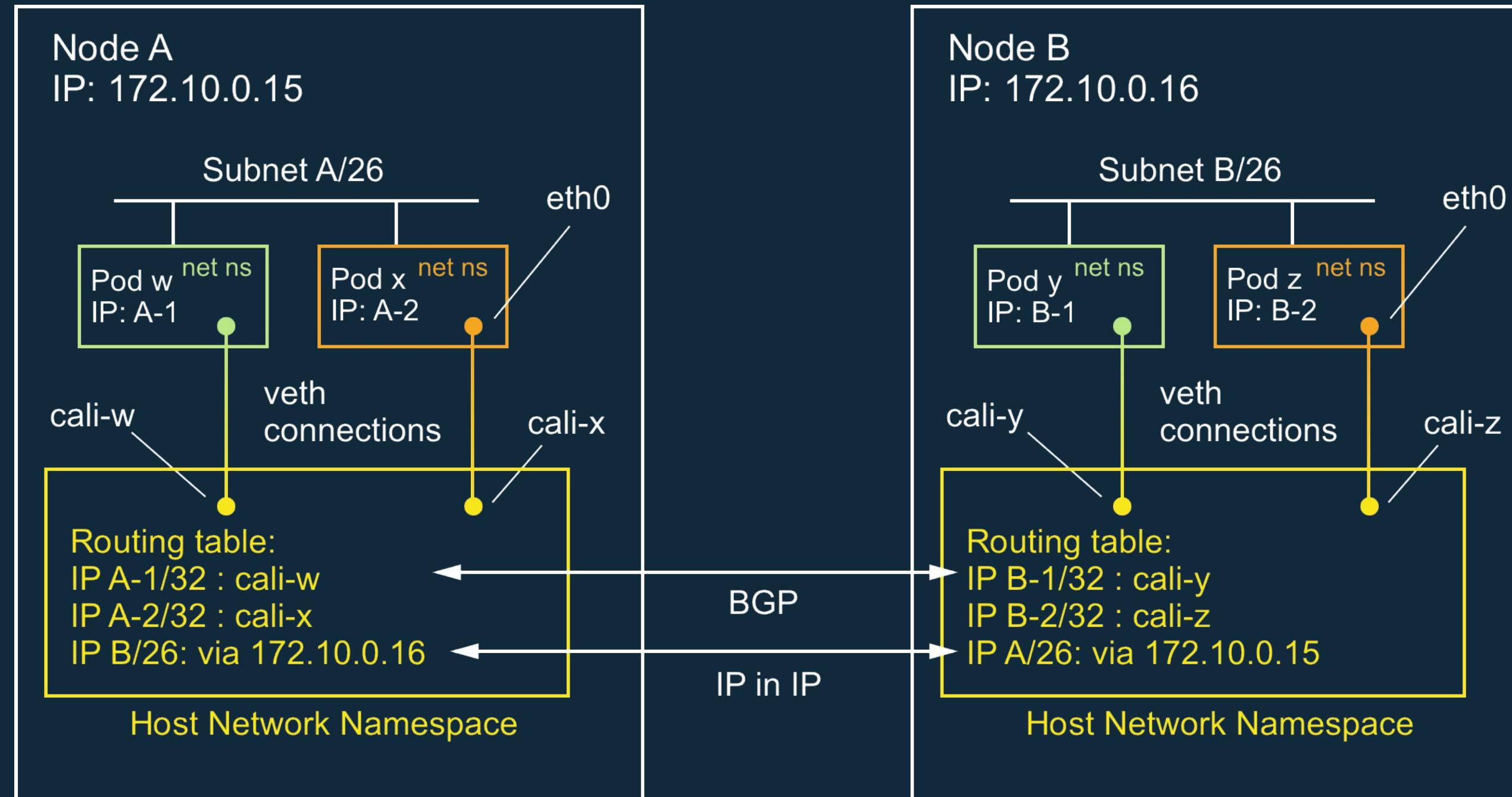


KUBERNETES NETWORKING PLANES

- Management:
 - Master to Master: etcd Raft
 - Master to Node: apiserver (TCP 443) <-> kubelet (TCP 10250)
- Data & Control:
 - BYO networking
 - See Cluster DNS, <http://bit.ly/2DMmdyt>



CALICO





EXERCISE: KUBERNETES NETWORKING

Work through:

- Kubernetes Networking

In the Docker Fundamentals Exercises book.



COMPARING SWARM & KUBERNETES

Swarm	Kubernetes
Swarm	Cluster
Node	Cluster Member
Manager	Master
Worker	(Worker) Node
Container	Container **
Task	Pod
Service	ReplicaSet
Service	Deployment
Stack	Stack **
VIP	ClusterIP Service
Routing Mesh	NodePort Service
Network	Network Policies



KUBERNETES TAKEAWAYS

- Kubernetes is an alternative to Swarm
- Kube provides more flexibility in its orchestration objects at the cost of more config
- Kube prioritizes inter-container communication, compared to Swarm's prioritization of security.



FURTHER READING

- Docker & Kubernetes: <https://www.docker.com/kubernetes>
- Official Kubernetes Docs: <https://kubernetes.io/docs>
- Tutorials: <http://bit.ly/2yLGn61>
- Interactive Tutorials: <https://bit.ly/2rdwlVZ>
- Understanding Kubernetes Networking: <http://bit.ly/2kdl1qQ>
- Kubernetes the Hard Way: <http://bit.ly/29Dq4wC>



FUNDAMENTAL ORCHESTRATION TAKEAWAYS

- Distributed Application Architecture orchestrates one or more containers across one or more nodes
- Orchestrators abstract away the differences between processes and between nodes
- Orchestrators enhance scalability and stability



BONUS TOPICS





DOCKER SYSTEM COMMANDS



LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Execute clean-up commands
- Locate Docker system information



CLEAN-UP COMMANDS

- docker system df

TYPE	TOTAL	ACTIVE	SIZE	RECLAIMABLE
Images	39	2	9.01 GB	7.269 GB (80%)
Containers	2	2	69.36 MB	0 B (0%)
Local Volumes	0	0	0 B	0 B

- docker system prune

more limited...

- docker image prune [--filter "foo=bar"]
- docker container prune [--filter "foo=bar"]
- docker volume prune [--filter "foo=bar"]
- docker network prune [--filter "foo=bar"]



INSPECT THE SYSTEM

docker system info

```
Containers: 2
Running: 2
Paused: 0
Stopped: 0
Images: 105
Server Version: 17.03.0-ee
Storage Driver: overlay2
Backing Filesystem: extfs
Supports d_type: true
Native Overlay Diff: true
Logging Driver: json-file
Cgroup Driver: cgroups
Plugins:
Volume: local
Network: bridge host ipvlan macvlan null overlay
Swarm: active
NodeID: ybmqksh6fm627armruq0e8id1
Is Manager: true
ClusterID: 2rbf1dv6t5ntro2fxbry6ikr3
Managers: 1
Nodes: 1
Orchestration:
Task History Retention Limit: 5
Raft:
Snapshot Interval: 10000
Number of Old Snapshots to Retain: 0
Heartbeat Tick: 1
...
```



SYSTEM EVENTS

Start observing with ...

docker system events

Generate events with ...

docker container run --rm alpine echo 'Hello World!'

```
2017-01-25T16:57:48.553596179-06:00 container create 30eb630790d44052f26c1081...
2017-01-25T16:57:48.556718161-06:00 container attach 30eb630790d44052f26c1081...
2017-01-25T16:57:48.698190608-06:00 network connect de1b2b40f522e69318847ada3...
2017-01-25T16:57:49.062631155-06:00 container start 30eb630790d44052f26c1081d...
2017-01-25T16:57:49.164526268-06:00 container die 30eb630790d44052f26c1081dbf...
2017-01-25T16:57:49.613422740-06:00 network disconnect de1b2b40f522e69318847a...
2017-01-25T16:57:49.815845051-06:00 container destroy 30eb630790d44052f26c108...
```





EXERCISE: SYSTEM COMMANDS

Work through:

- Cleaning up Docker Resources
- Inspection Commands

in the Docker Fundamentals Exercises book.



DISCUSSION

- What is the origin of dangling image layers?
- What are some potential pitfalls to automating system cleanup with prune commands, and how to avoid them?
- Questions?



FURTHER READING

- System commands reference: <http://dockr.ly/2eMR53i>





DOCKER PLUGINS (LINUX)



LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Discuss the purpose and format of Docker plugins
- Witness the value of plugins by deploying a plugin

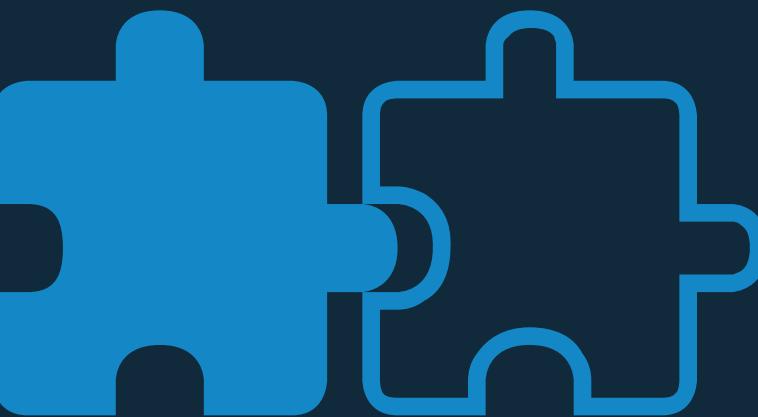


PLUGINS

- Extend the Docker platform
- Distributed as Docker images
- Hosted on hub.docker.com

List plugins on system:

```
$ docker plugin ls
ID          NAME           DESCRIPTION          ENABLED
bee424413706  vieux/sshfs:latest  sshFS plugin for Docker  true
```



INSTALL A PLUGIN

```
$ docker plugin install vieux/sshfs
Plugin "vieux/sshfs" is requesting the following privileges:
- network: [host]
- mount: [/var/lib/docker/plugins/]
- device: [/dev/fuse]
- capabilities: [CAP_SYS_ADMIN]
Do you grant the above permissions? [y/N] y
latest: Pulling from vieux/sshfs
a23658ccfda2: Download complete
Digest: sha256:d33ffa08df2a4fa87f83fded4ac3ac2e3a96d296c55aa1a0afce33fb2cc7b9b0
Status: Downloaded newer image for vieux/sshfs:latest
Installed plugin vieux/sshfs
```



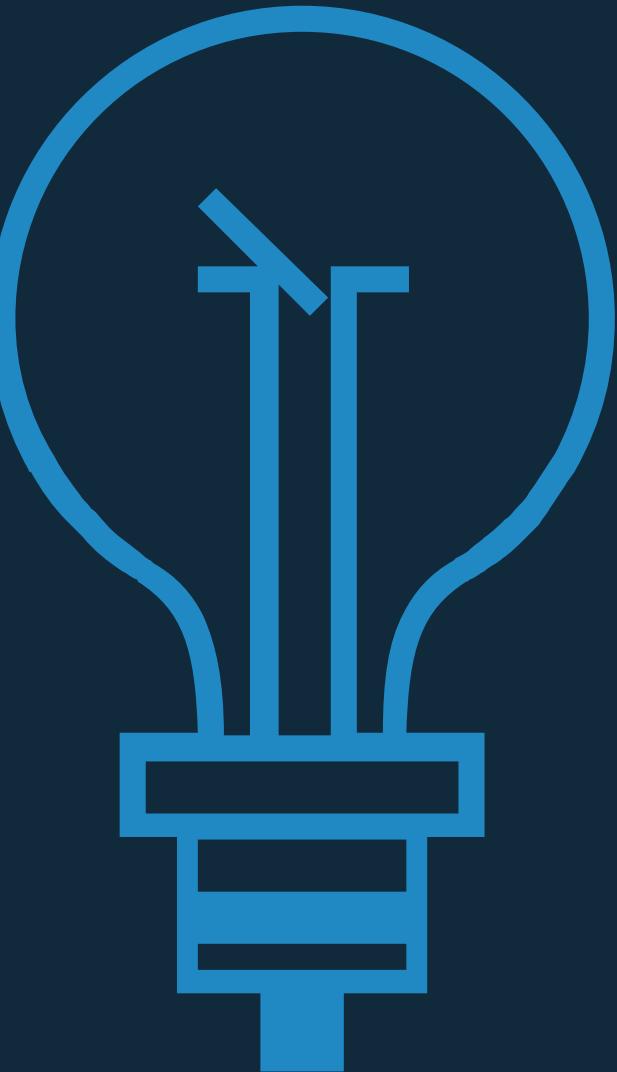
USING THE PLUGIN

Create a volume...

```
docker volume create -d vieux/sshfs \
-o sshcmd=<user@host:path> \
-o password=<password> \
sshvolume
```

Use the volume...

```
docker container run -v sshvolume:/demo -it centos:7 bash
```





EXERCISE: PLUGINS

Work through:

- Docker Plugins

In the Docker Fundamentals Exercises book.



DISCUSSION

- What sort of volume plugins would make sense for your applications in your datacenter?
- Questions?



FURTHER READING

- How to implement a plugin: <http://dockr.ly/2vRIEcG>
- Create an authorization plugin: <http://dockr.ly/2wuxfBB>
- List of Docker Engine plugins: <http://dockr.ly/2vyXERe>
- Docker network driver plugins: <http://dockr.ly/2wqiill>
- Volume plugins: <http://dockr.ly/2vSbU2W>
- Plugin configuration reference: <http://dockr.ly/2er4bm2>
- Plugin API reference: <http://dockr.ly/2ewofHi>





INTRODUCTION TO DOCKER COMPOSE



DISCUSSION: PROCESSES VS. APPLICATIONS

Containers and images describe individual processes. What will we need to describe entire applications?



LEARNING OBJECTIVES

By the end of this module, learners will be able to:

- Design scalable Docker services
- Leverage Docker's built in service discovery mechanism
- Write a compose file describing an application



DISTRIBUTED APPLICATION ARCHITECTURE

- Applications consisting of one or more containers across one or more nodes
- Docker Compose facilitates multi-container design **on a single node.**

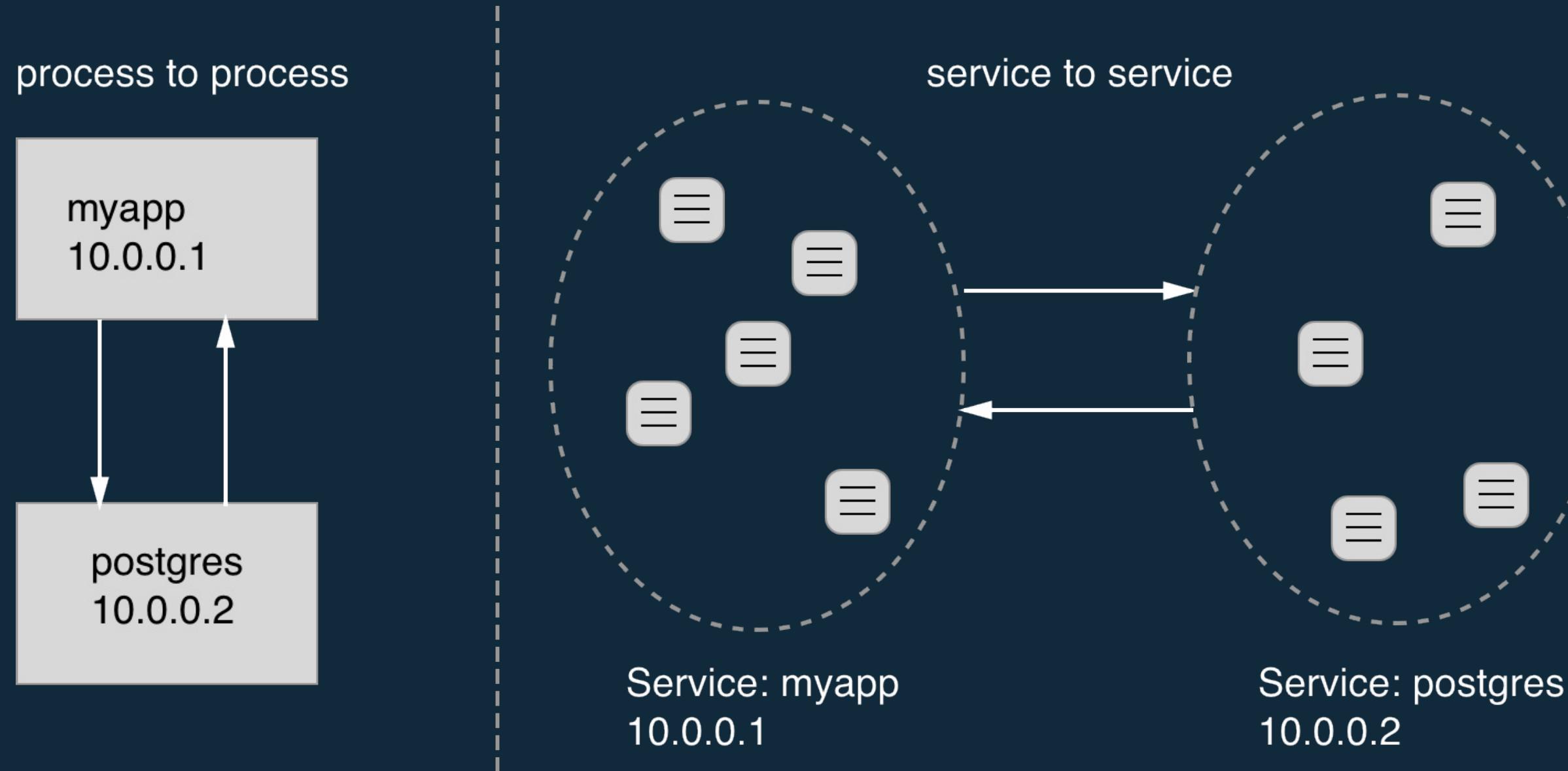


DOCKER SERVICES

- Goal: declare and (re)configure many similar containers all at once
- Goal: scale apps by adding containers seamlessly
- A service defines the **desired state** of a group of identically configured containers.
- Docker provides **transparent service discovery** for Services.



SERVICE DISCOVERY



Services are assigned a **Virtual IP** which spreads traffic out across the underlying containers automatically.



OUR APPLICATION: DOCKERCOINS



(DockerCoins 2016 logo courtesy of [@XtlCnslt](#) and [@ndeloof](#). Thanks!)

- It is a DockerCoin miner!
- Dockercoins consists of 5 services working together:





INSTRUCTOR DEMO: DOCKER COMPOSE

See the demo:

- Docker Compose

In the Docker Fundamentals Exercises book.





EXERCISE: COMPOSE APPS

Work through

- Starting a Compose App
- Scaling a Compose App

in the Docker Fundamentals Exercises book.



DOCKER COMPOSE TAKEAWAYS

- Docker Compose makes single node orchestration easy
- Compose services makes scaling applications easy
- Bottleneck identification important
- Syntactically: **docker-compose.yml** + API



FURTHER READING

- Docker compose examples: <http://dockr.ly/1FL2VQ6>
- Overview of docker-compose CLI: <http://dockr.ly/2wtQLZT>
- **docker-compose.yaml** reference: <http://dockr.ly/2iHUpex>
- Docker Compose and Windows: <http://bit.ly/2watrqk>



DOCKER FUNDAMENTALS

Please take our feedback survey:

<http://bit.ly/2GwsqLA>

Get in touch: training@docker.com

success.docker.com/training



YOU'RE ON YOUR WAY TO BECOMING DOCKER CERTIFIED!

- Study up with our Study Guides at <http://bit.ly/2yPzAdb>
- Take it online 24 hours a day
- Results delivered immediately
- Benefits include:
 - Digital certificate
 - Online verification
 - Private LinkedIn group
 - Exclusive events

SUCCESS.DOCKER.COM/CERTIFICATION

