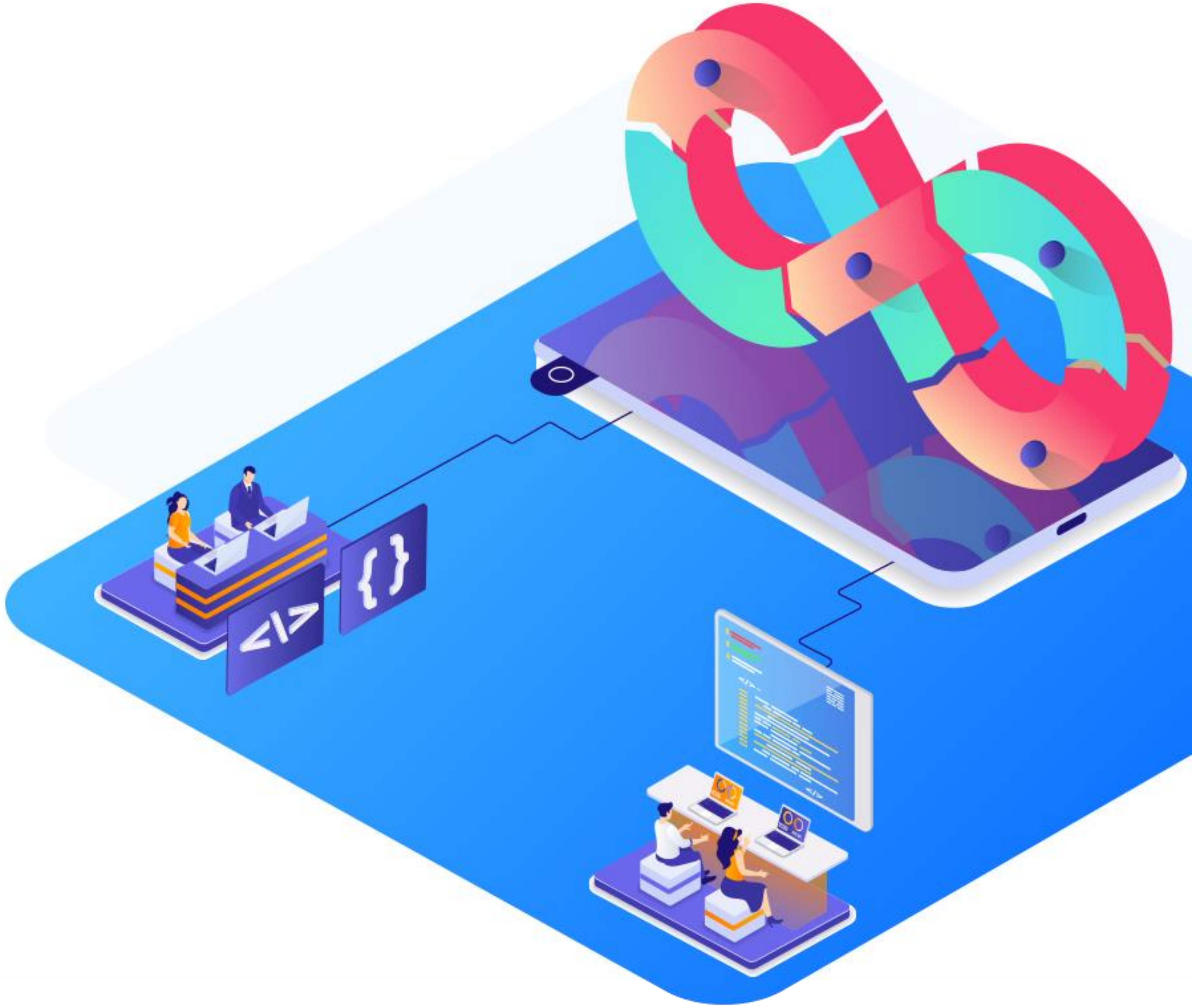
Containerization with Docker



Networking in Docker

Learning Objectives

By the end of this lesson, you will be able to:

- Implement different types of networks and their architecture for enhanced network design and functionality
- Manage the container network model (CNM) architecture and utilize different network drivers for effective management
- Configure and publish the ports for seamless communication between networked services and applications
- Access logs and troubleshoot services for maintaining system health and ensuring optimal performance

Docker Networking

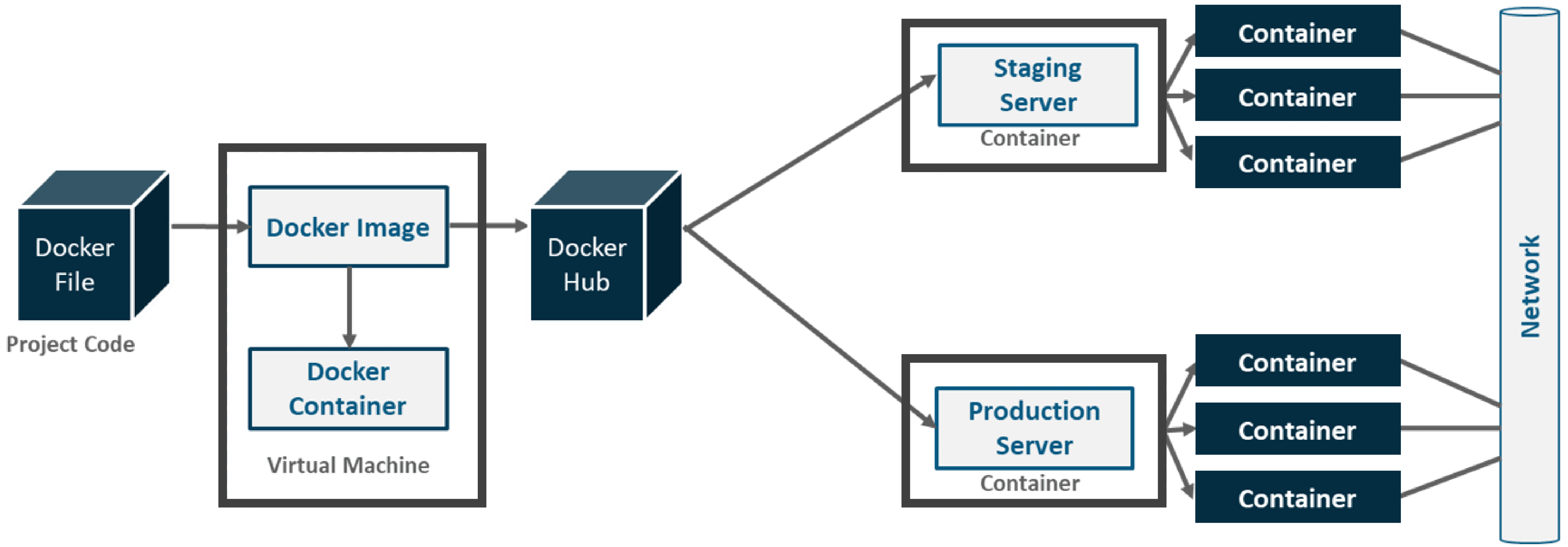
Docker Networking: Introduction

It refers to the ability for containers to connect and communicate with each other or to non-Docker workloads.

Docker provides different network modes such as Bridge, None, and Host to cater to various networking needs. This ensures the isolation, security, and connectivity of containerized applications.

Docker Networking: Workflow

The following image illustrates the Docker workflow, starting with a **Dockerfile** that contains project code. The **Dockerfile** creates a **Docker Image** stored in the **Docker Hub.**



This image can be deployed as **Docker containers** on staging and production servers. Each server can host multiple containers connected to a network.

Docker Networking: Key Concepts

Below are the essential key concepts that manage connectivity, IP allocation, and container communication in Docker:

Network drivers

- Controls how container networking is set up and managed
- Examples: Bridge, Overlay, and MACVLAN drivers

IP address management

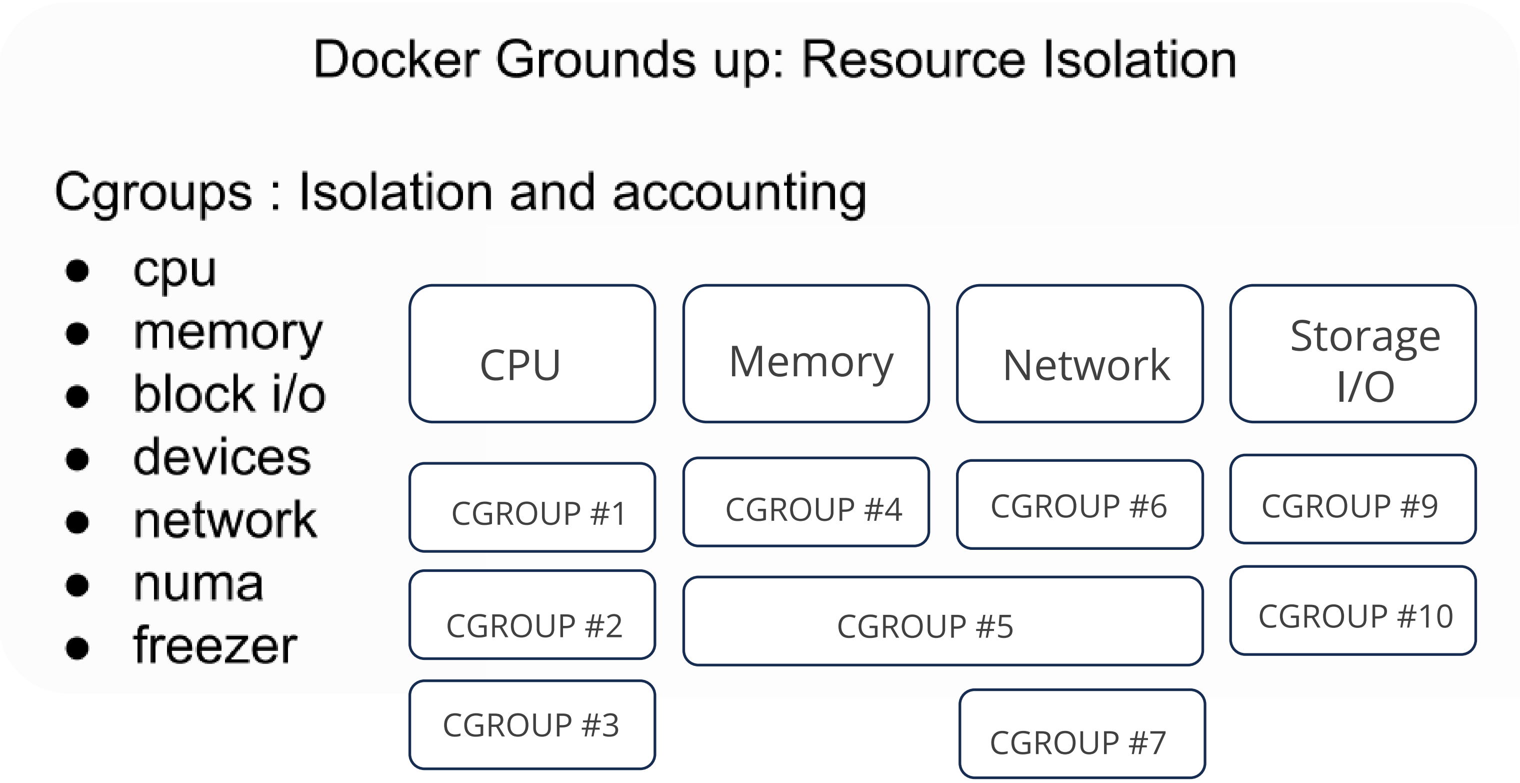
- Assigns IP addresses automatically to the containers
- Ensures containers can communicate within the network

DNS resolution

- Allows containers to find each other by name
- Makes communication between containers easier without using IP addresses

Docker Control Groups

Control groups (cgroups) are a feature of the Linux kernel that Docker uses to manage and limit the resources (like CPU, memory, disk I/O) that containers can use.



Docker Grounds up: Resource Isolation

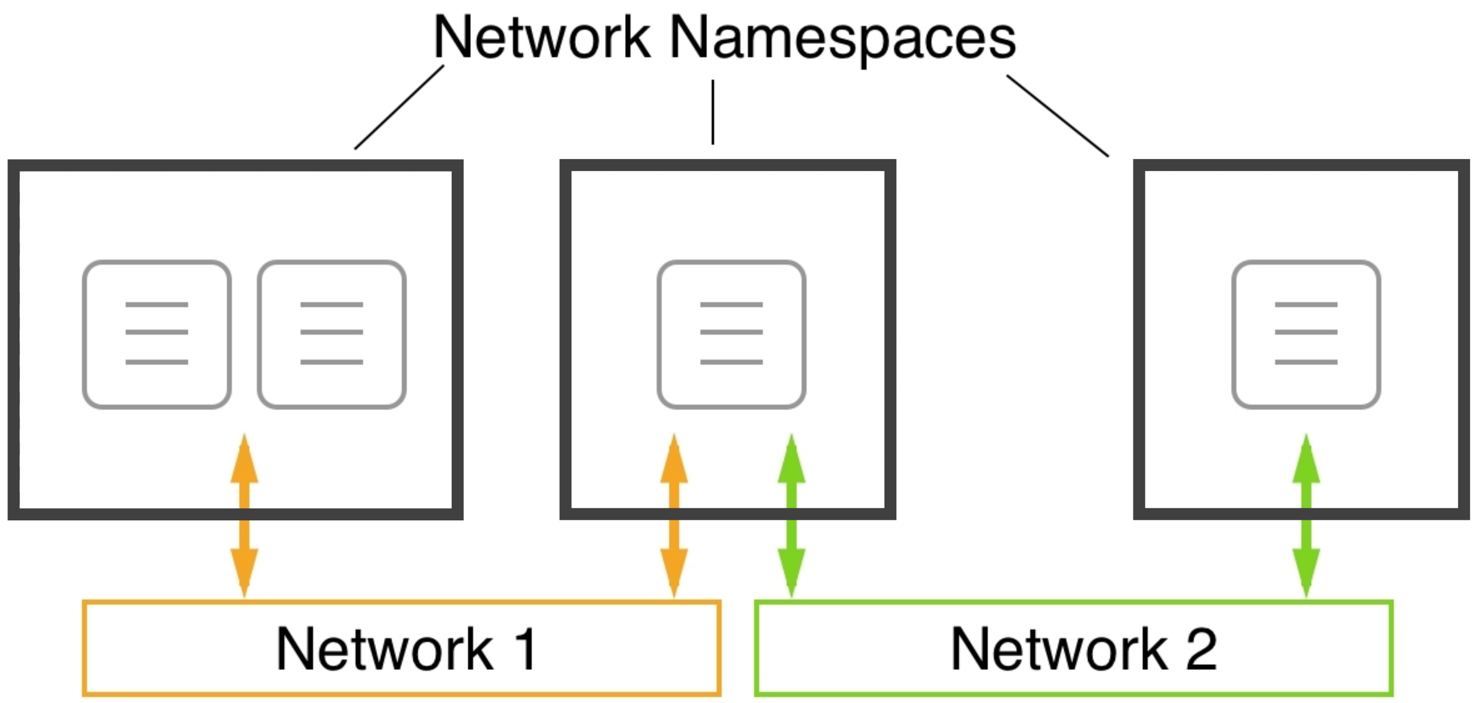
Cgroups : Isolation and accounting

- cpu
- memory
- block i/o
- devices
- network
- numa
- freezer

They ensure that a container does not consume more than its allocated resources, preventing it from affecting the performance of other containers or the host system.

Docker Namespace

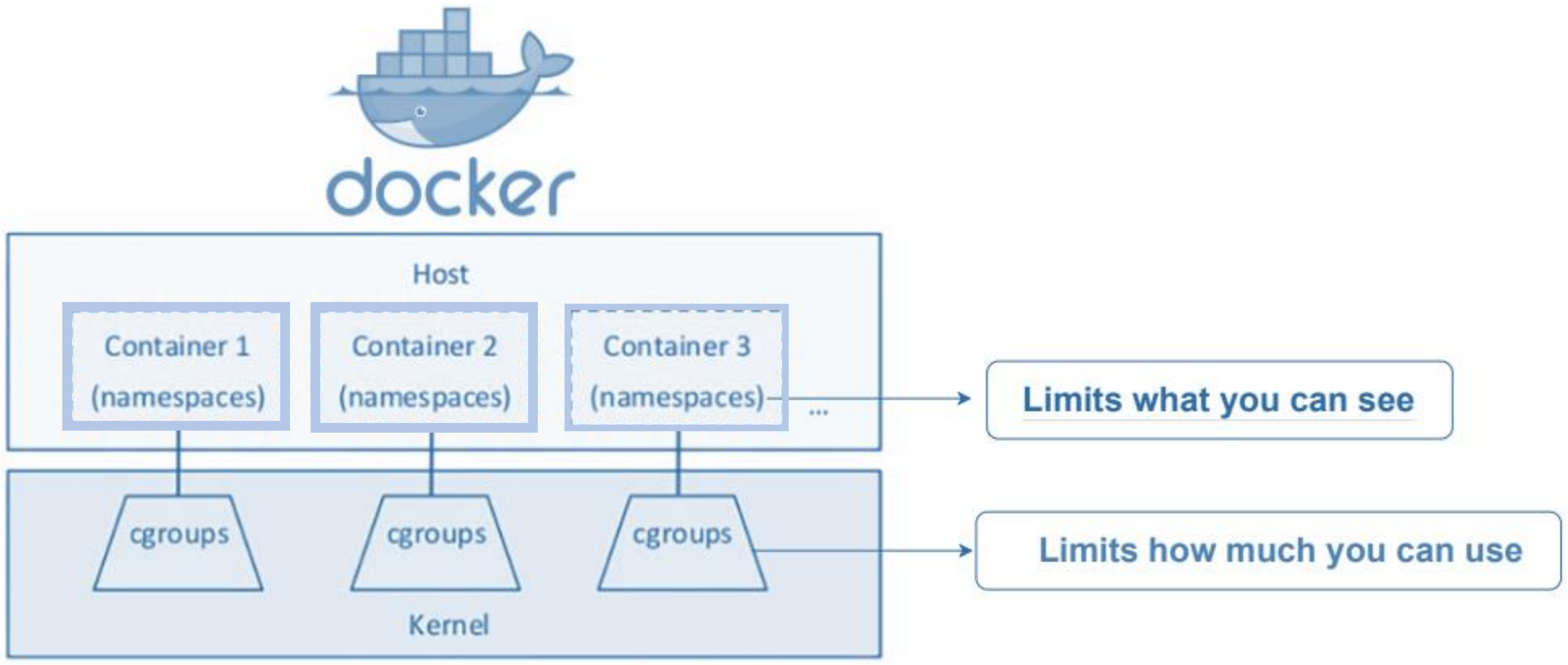
Namespaces are another feature of the Linux kernel that Docker leverages to isolate containers.



They isolate aspects like process IDs, network interfaces, and inter-process communication, ensuring containers operate independently from each other and the host.

Docker Namespace and Cgroups

Namespaces ensure process isolation, and cgroups manage resource allocation, forming the foundation of secure and efficient container management in Docker.



Types of Network

Bridge network

- Default network for standalone containers
- Isolated environment for containers on the same host

Use case: For internal communication between containers on the same machine

Host network

- No network isolation between the container and the host
- Containers use the host's network stack

Use case: For applications requiring high network performance

Overlay network

- Connects containers across multiple hosts
- Encrypted communication between containers

Use case: For multi-host Docker setups, commonly used with Docker swarm

Types of Network

MACVLAN network

- Assigns a MAC address to containers, making them appear as physical devices on the network
- Direct access to the physical network

Use case: For legacy applications requiring direct network access

None network

- Disables networking for the container
- Complete isolation from other containers and the host network

Use case: For tasks that do not require any network access

Quick Check

You are running multiple Docker containers on a server and must prevent any container from consuming too much CPU or memory. What would you use to enforce these limits?

A. Allocating IP addresses to containers

B. Managing container storage

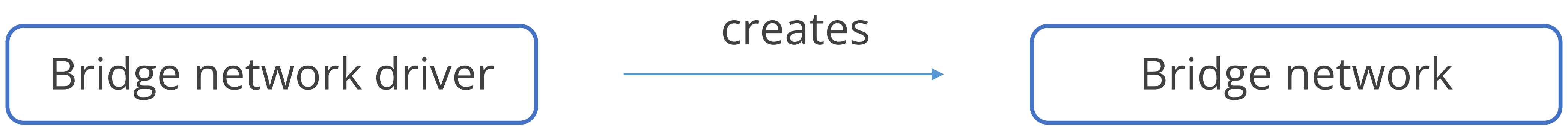
C. Limiting resource usage of containers

D. Ensuring process isolation

Overview of Bridge Network

Bridge Network: Introduction

It is a default Docker network present on any Linux host that runs a Docker engine.



Bridge network driver

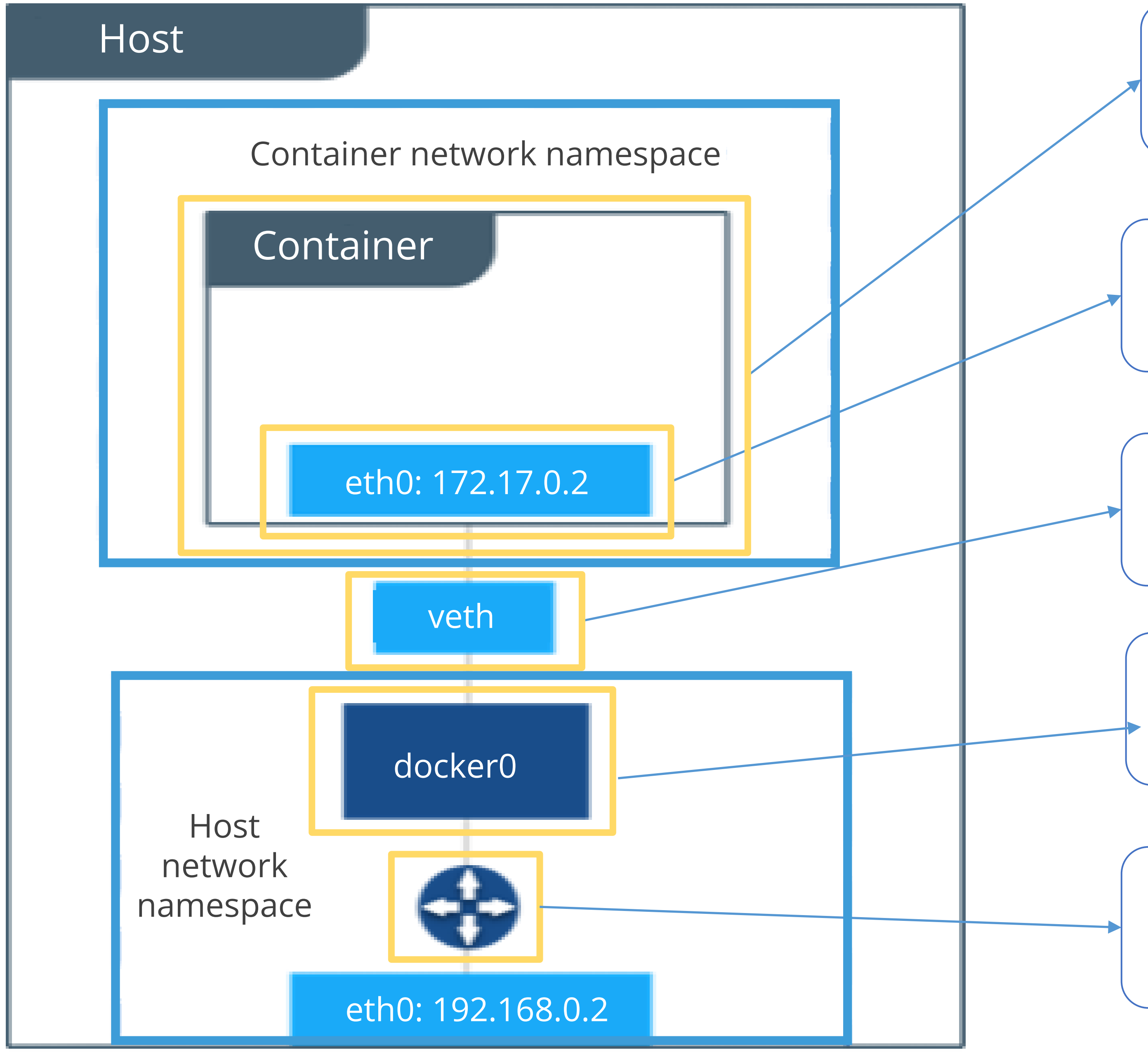
creates

Bridge network

Understanding correlated terms:

- A bridge is a Docker network.
- It is also a Docker network driver or template, which connect the bridge network.
- **docker0** is the kernel building block that implements the bridge network.

Working of Bridge Network



By default, the **Docker Engine** connects the container to the bridge network.

The bridge driver creates **eth0**, and the Docker native IPAM driver gives an address.

veth is a virtual ethernet interface, which connects bridge to the **eth0** interface inside the container.

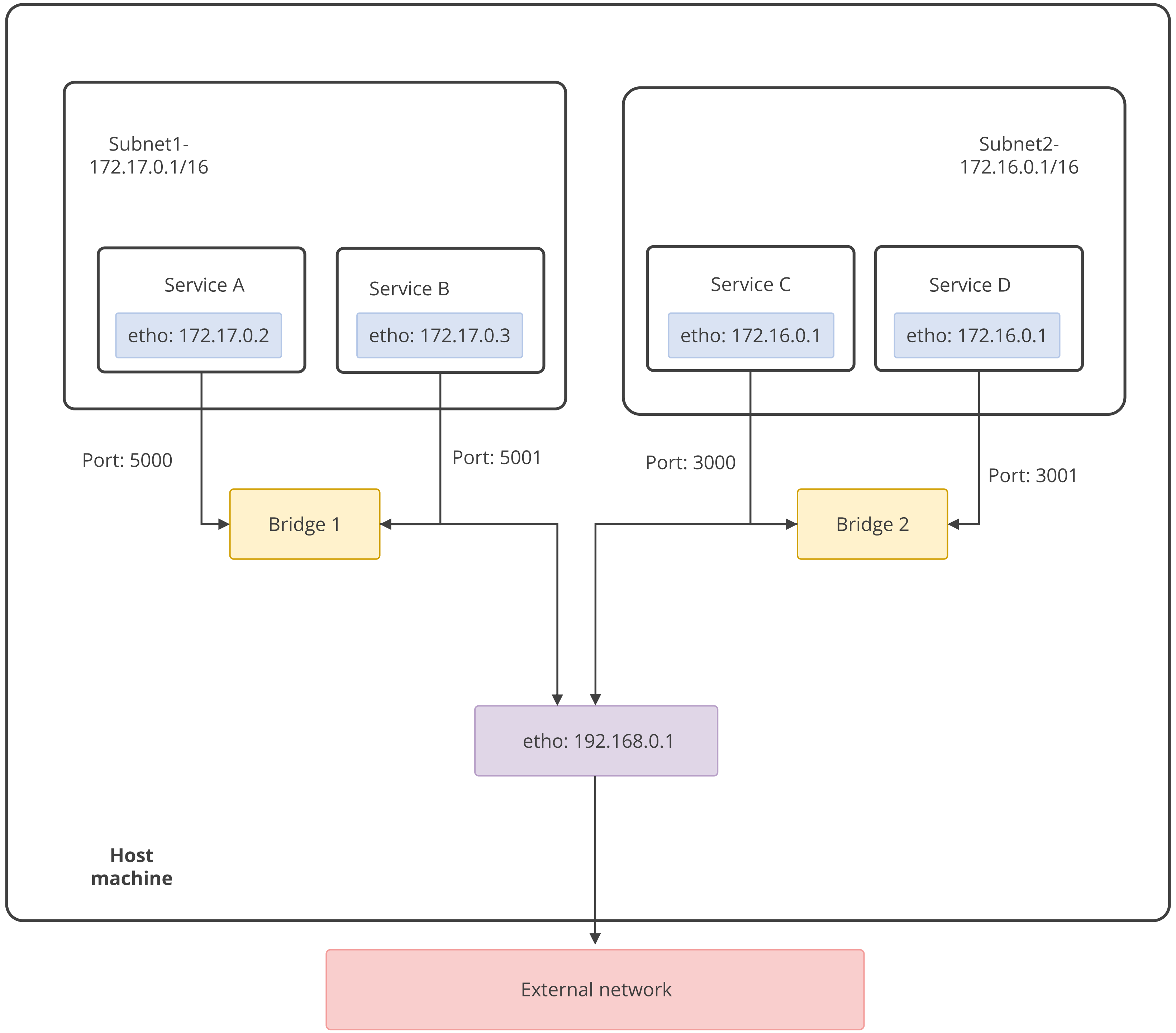
docker0 is a Linux bridge that exists in the host network namespace.

The routing table connects **docker0** and **eth0** on the external network.

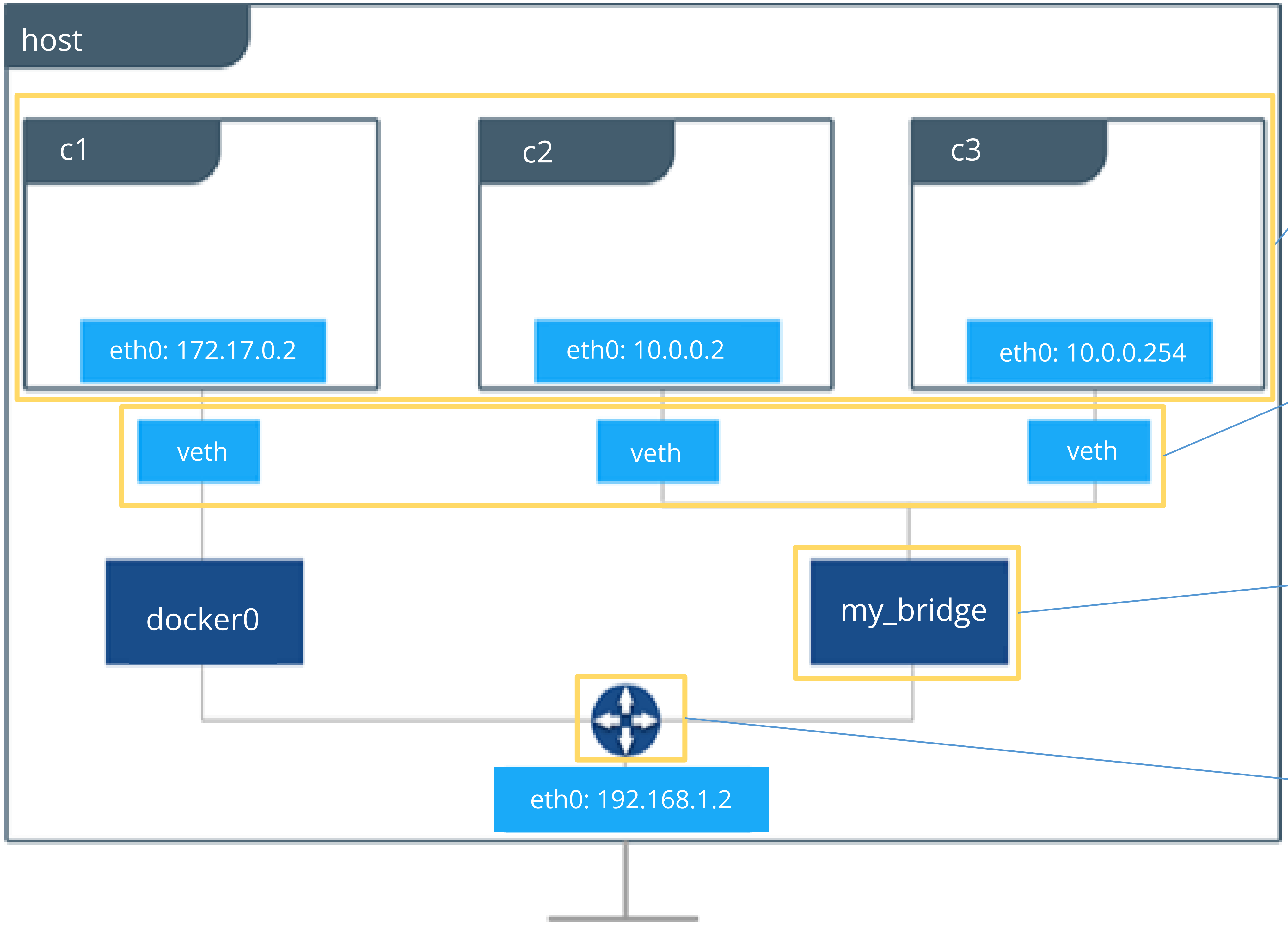
***Source**: https://success.docker.com/article/networking#dockerbrigenetworkdriver*

User-Defined Bridge Network

It allows for customized network configurations in Docker, going beyond the default network options.



User-Defined Bridge Network: Architecture



Containers

Virtual ethernet interfaces

The user-defined bridge that connects two containers via two interfaces

The routing table connects **docker0, my_bridge,** and **eth0** on the external network

Source: https://success.docker.com/article/networking#dockerbridgenetworkdriver

Creating a User-Defined Bridge Network

The user can define specific subnet ranges and gateways and even specify which containers can connect to which networks.

1. Creating a network:

```
docker network create <network_name>
```

2. Creating a network with a custom subnet and gateway:

```
docker network create --subnet=<subnet_ip_range> --gateway=<gateway_ip> <network_name>
```

3. Inspecting a network’s configuration:

```
docker network inspect <network_name>
```

Difference between Default and User-Defined Bridge Network

Feature	Default bridge	User-defined bridge
Isolation and interoperability	Limited isolation and interoperability between containers	Better isolation and interoperability, allowing more control over container communication
Automatic DNS resolution	Not available	Enabled, allowing containers to resolve each other by name automatically
Attachment and detachment of containers	Static containers cannot be attached or detached on the fly	Dynamic containers can be attached or detached from the network at any time
Configurable bridge creation	Not supported	Fully configurable, allowing custom network setups
Environment variables sharing	Linked containers can share environment variables	Does not support sharing environment variables between linked containers

Assisted Practice

Creating a Docker Bridge Network

Duration: 10 minutes

Problem Statement:

You have been assigned a task to create a default network in Docker and inspect it so that it can be established that the bridge driver is the default network driver.

Outcome:

By completing this demo, you will be able to create a Docker bridge network, verify its configuration, and confirm that the bridge driver is correctly set as the default network driver.

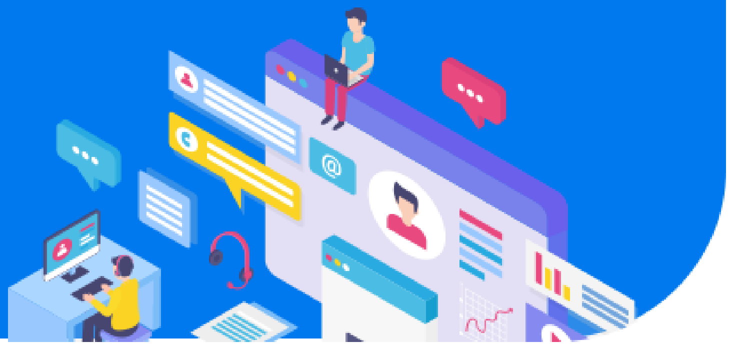
Note: Refer to the demo document for detailed steps:
01_Creating_a_Docker_Bridge_Network

Assisted Practice: Guidelines

Steps to be followed:

1. Create a default network in Docker
2. Inspect the network for the network driver

Assisted Practice



Creating User-Defined Bridge Network

Duration: 20 minutes

Problem Statement:

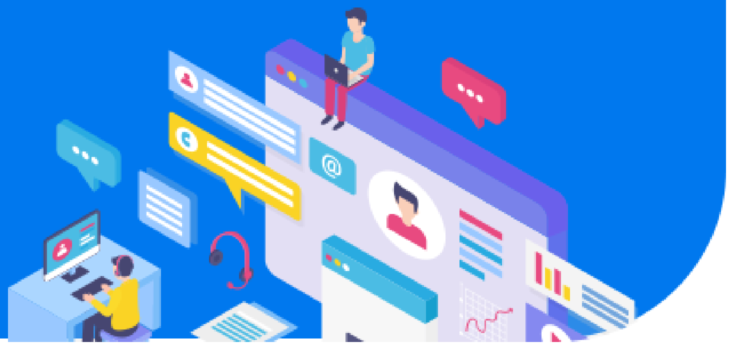
You have been asked to create a user-defined bridge network that can be used to connect multiple containers to a single network.

Outcome:

By completing this demo, you will be able to create a user-defined bridge network in Docker, connect multiple containers to it, and manage their communication effectively within the custom network.

Note: Refer to the demo document for detailed steps:
02_Creating_User-Defined_Bridge_Network

Assisted Practice: Guidelines



Steps to be followed:

1. Create and delete user-defined network
2. Connect a ngnix container to the my-net network
3. Connect the running my-nginx container to the my-net network
4. Inspect the container and disconnect it from the network

Quick Check

You need more control over your Docker container network settings than what the default bridge network offers. What is the main advantage of using a user-defined bridge network in this scenario?

A. Limited isolation

B. Static container attachment

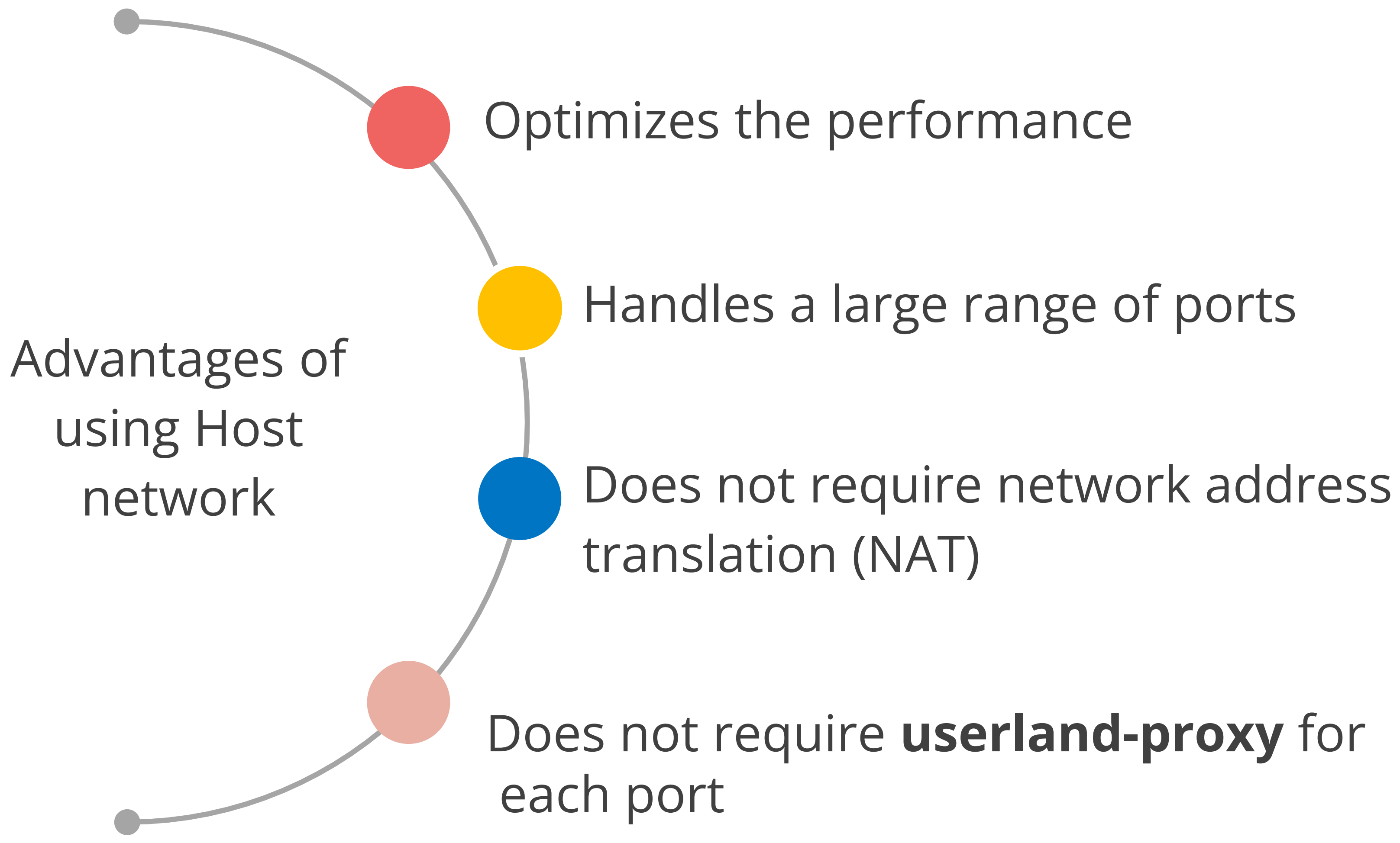
C. Configurable bridge creation

D. Environment variables sharing

Host Network

Host Network: Introduction

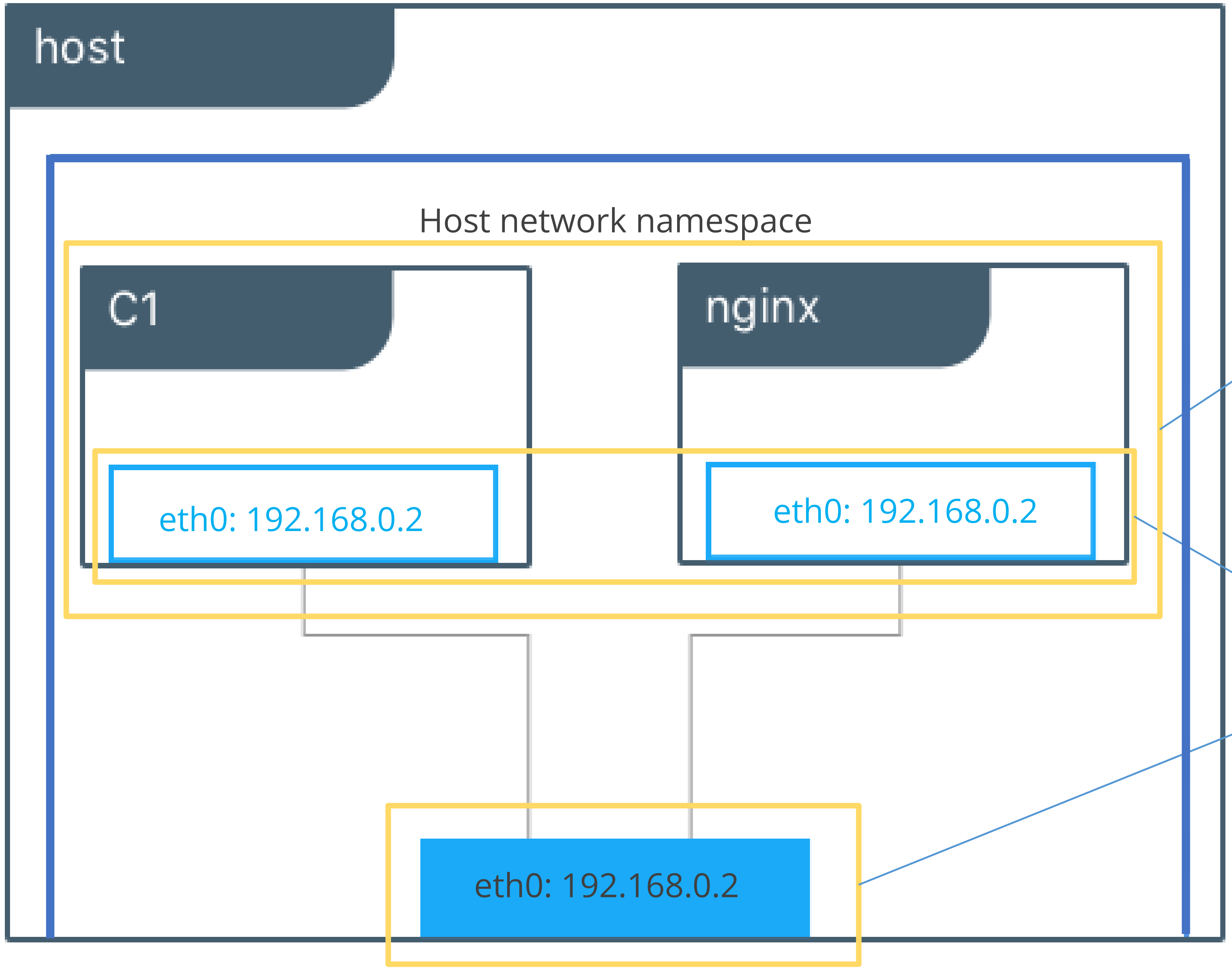
It is a network configuration option where a container shares the network namespace of the host system.



Advantages of using Host network

- Optimizes the performance
- Handles a large range of ports
- Does not require network address translation (NAT)
- Does not require **userland-proxy** for each port

Host Network: Architecture



These containers connect using a **localhost** on C1.

The containers C1 and nginx use the host network and share the same interface for **eth0**.

Source: https://success.docker.com/article/networking#dockerhostnetworkdriver

Using Host Network in Docker Swarm Services

The host network mode lets containers use the host's network, which is ideal for applications needing high performance and quick access to the host's resources.

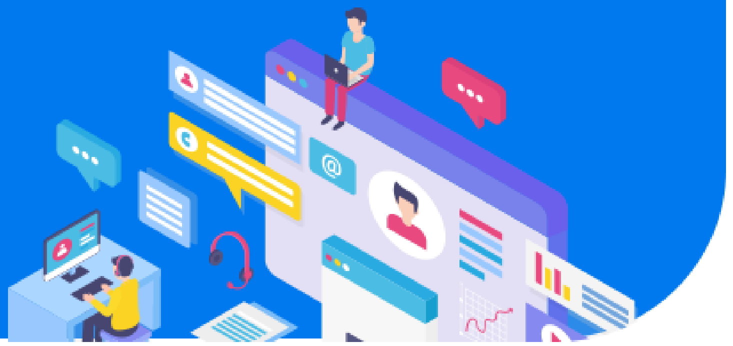
--network host: This is passed with command **docker service create** to use a host network for a swarm service.

Syntax

```
docker service create --network host <image_name>
```

This command creates a new service in Docker swarm using the host's network. The service’s containers will use the host network directly, meaning no network isolation.

Assisted Practice



Creating a Host Network

Duration: 20 minutes

Problem Statement:

You have been asked to create a host network so that your container gets its own IP address allocation and is not isolated from the host.

Outcome:

By completing this demo, you will be able to create a host network in Docker and link containers to it, enabling them to use the host’s network stack directly.

Note: Refer to the demo document for detailed steps:
03_Creating_a_Host_Network

Assisted Practice: Guidelines

Steps to be followed:

1. Initialize Docker swarm and create a standalone container
2. Inspect and verify container networking

Quick Check

How does the host network mode optimize performance in Docker?

A. By isolating containers on separate networks

B. By Attaching static container

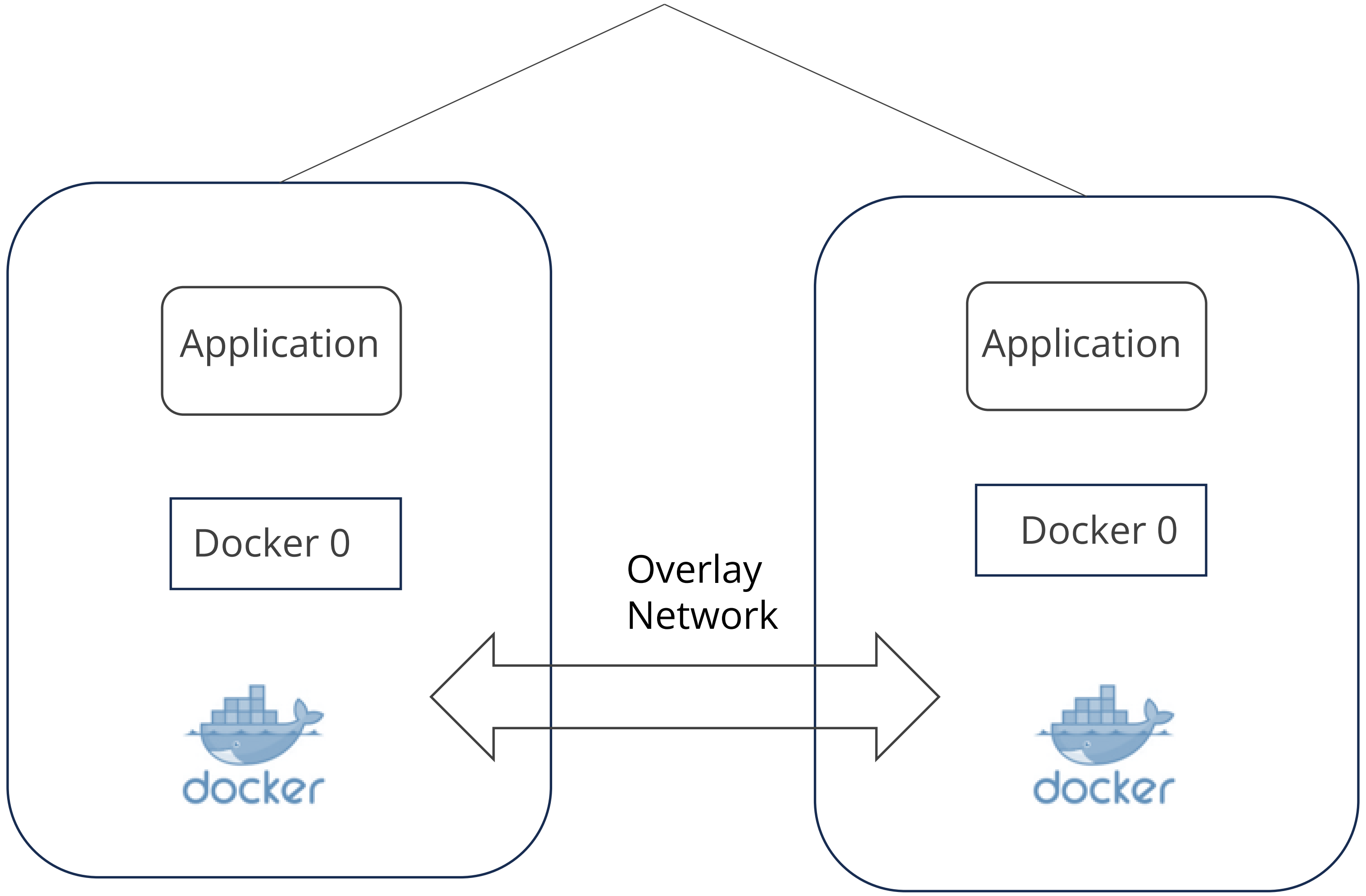
C. By using a dedicated virtual network interface

D. By enabling enhanced security measures

Overlay Network

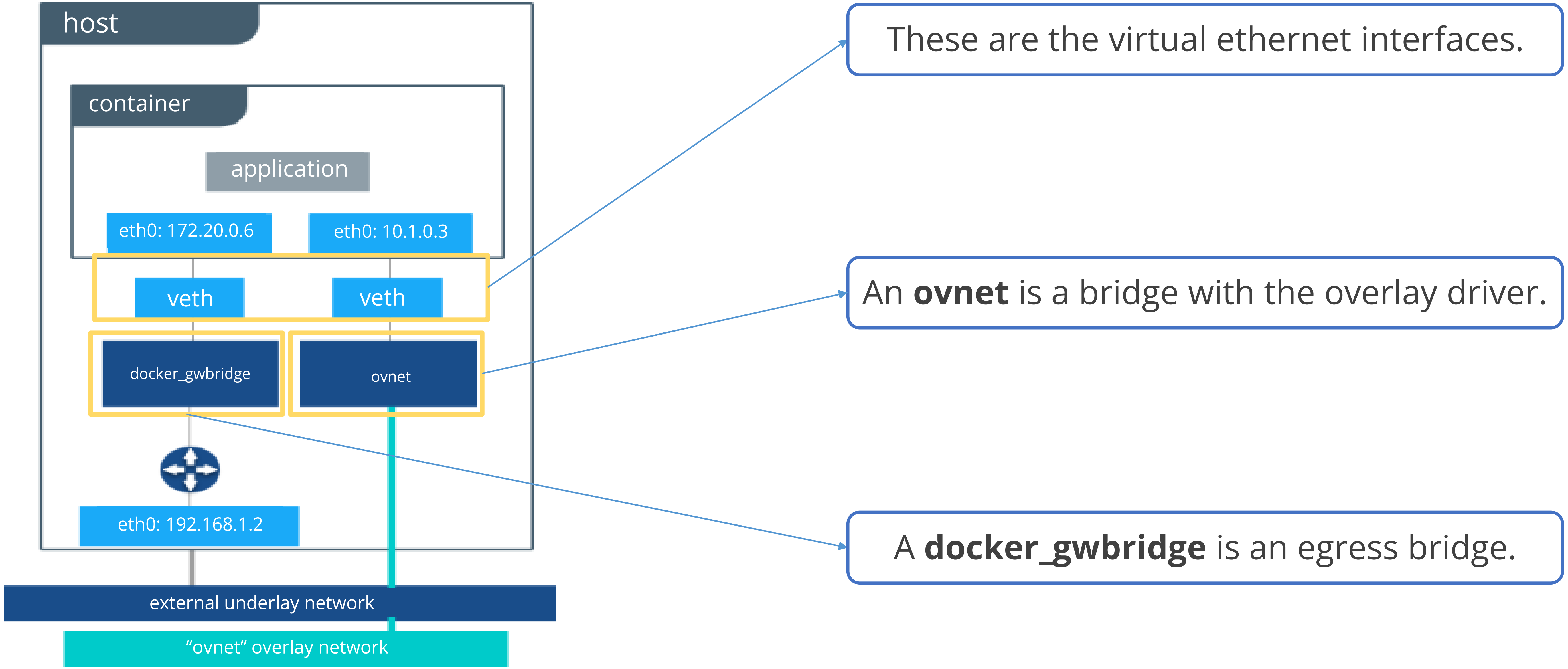
Overlay Network: Introduction

It creates a virtual network across multiple Docker hosts, letting containers on different hosts communicate as if they were on the same network.



Overlay Network: Architecture

Provisioning for an overlay network is automated by the Docker swarm control plane.



These are the virtual ethernet interfaces.

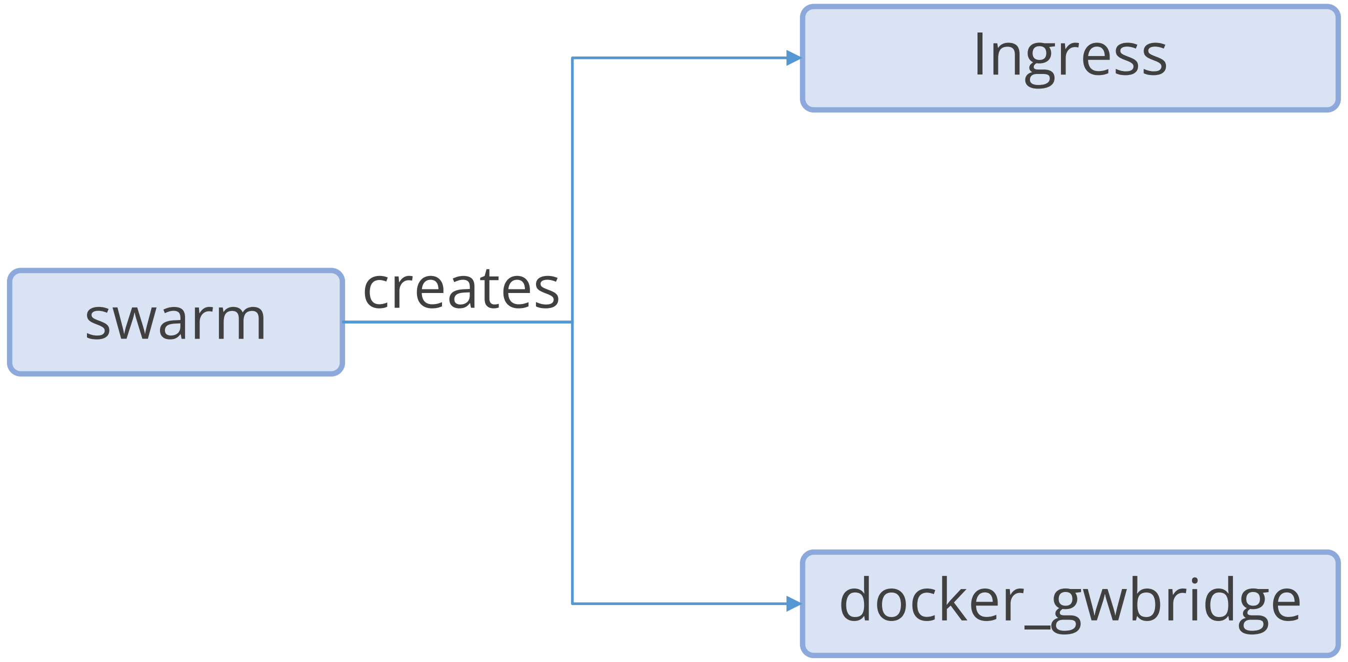
An **ovnet** is a bridge with the overlay driver.

A **docker_gwbridge** is an egress bridge.

Source: https://success.docker.com/article/networking#overlaydrivernetworkarchitecture

Overlay Network Components in Docker Swarm

In a Docker swarm, the overlay network relies on key components like the Ingress network and **docker_gwbridge**.



The Ingress network manages traffic and services within the swarm, while the **docker_gwbridge** connects all Docker daemons, enabling smooth communication across nodes.

Key Requirements in Setting Up an Overlay Network

Open ports:

- Open TCP port 2377 for cluster management communications
- Open TCP and UDP port 7946 for communication among nodes
- Open UDP port 4789 for overlay network traffic

Initialize Docker daemons:

As a swarm manager using **docker swarm init,** initialize the Docker daemons or join the Docker daemon to an existing swarm using **docker swarm join** before creating an overlay network

MACVLAN Network

MACVLAN Network: Introduction

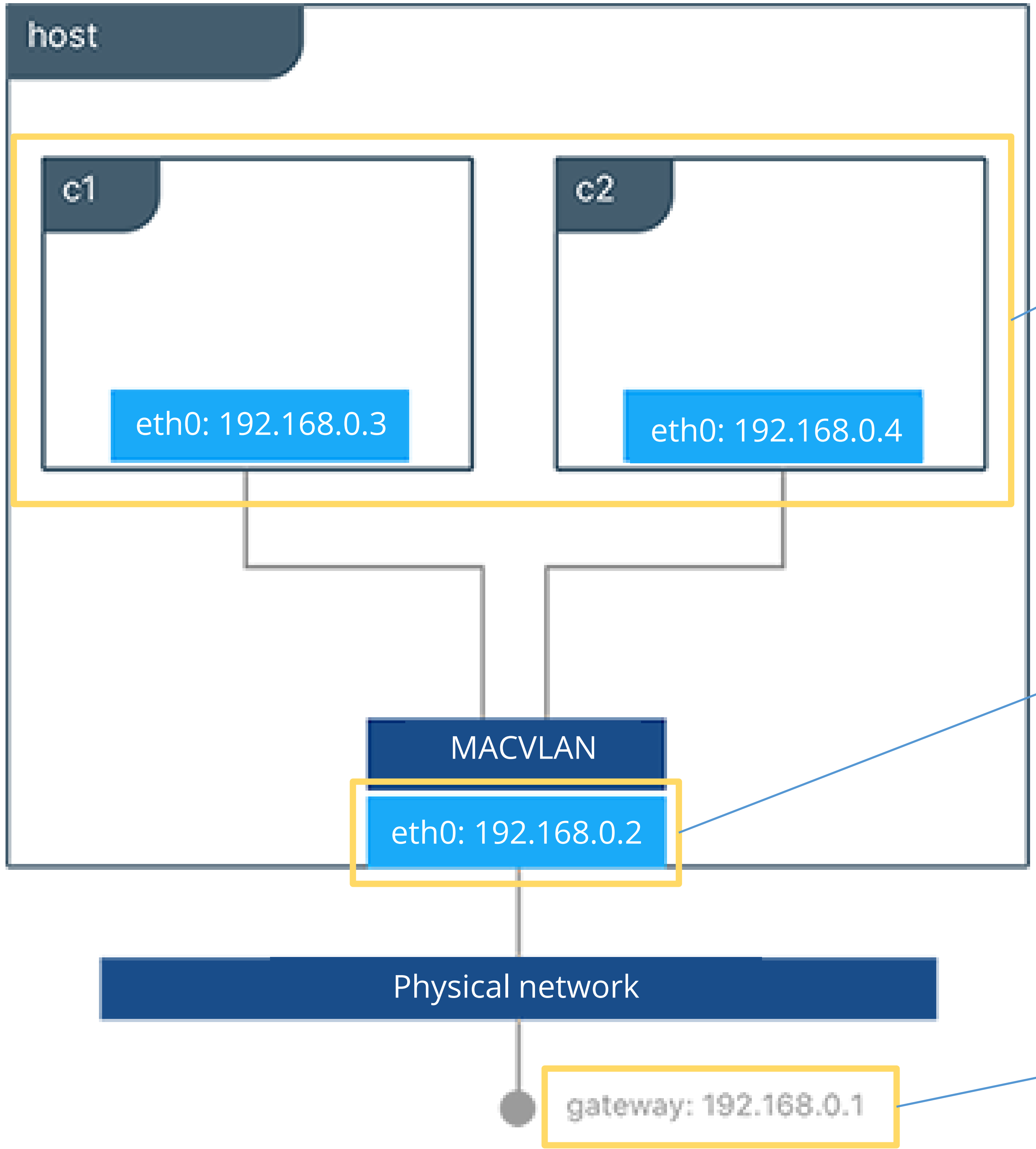
This network assigns MAC addresses to the virtual network interface of containers. This helps the legacy applications to connect to the physical network directly.

Precautionary measures:

Cut down the large number of unique MAC addresses to save the network from damage

Handle **promiscuous mode** via networking equipment to assign multiple MAC addresses to the single physical interface

MACVLAN Network: Architecture



The containers are formed on **mvnet** network.

An interface **eth0** on the host is bound to **mvnet** *(*a MACVLAN network*).*

An external gateway is required during MACVLAN network configuration.

https://success.docker.com/article/networking#MACVLAN

Advantages and Use Cases of MACVLAN Networking

Positive performance implications:

- MACVLAN has a simple and lightweight architecture.
- MACVLAN drivers provide direct access between the physical network and containers.
- MACVLAN containers receive routable IP addresses that are present on the subnet of the physical network.

Use cases of MACVLAN include:

- It implements low-latency applications that require quick response times and minimal delay.
- It designs the network so that containers are on the same subnet and use IP addresses as the external host network.

Quick Check

You're configuring a Docker network using MACVLAN and want to avoid potential network conflicts. Which precaution should you take?

A. Increase the number of MAC addresses

B. Use network address translation (NAT)

C. Disable promiscuous mode

D. Limit the number of unique MAC addresses

None Network

None Network: Introduction

The none network in Docker disables all networking for a container, resulting in a container with no network interface (**eth0**).

Useful when isolating a container from any network communication is required

Ideal for containers that need to be completely isolated from the network for security or testing purposes

Allows manual network configuration or specialized setups where no default networking is desired

Implementing the None Network in Docker

The none network runs tasks that don’t require network access, such as processing data internally or running tests that require complete isolation from other networks.

Form a container with none network:

Command:

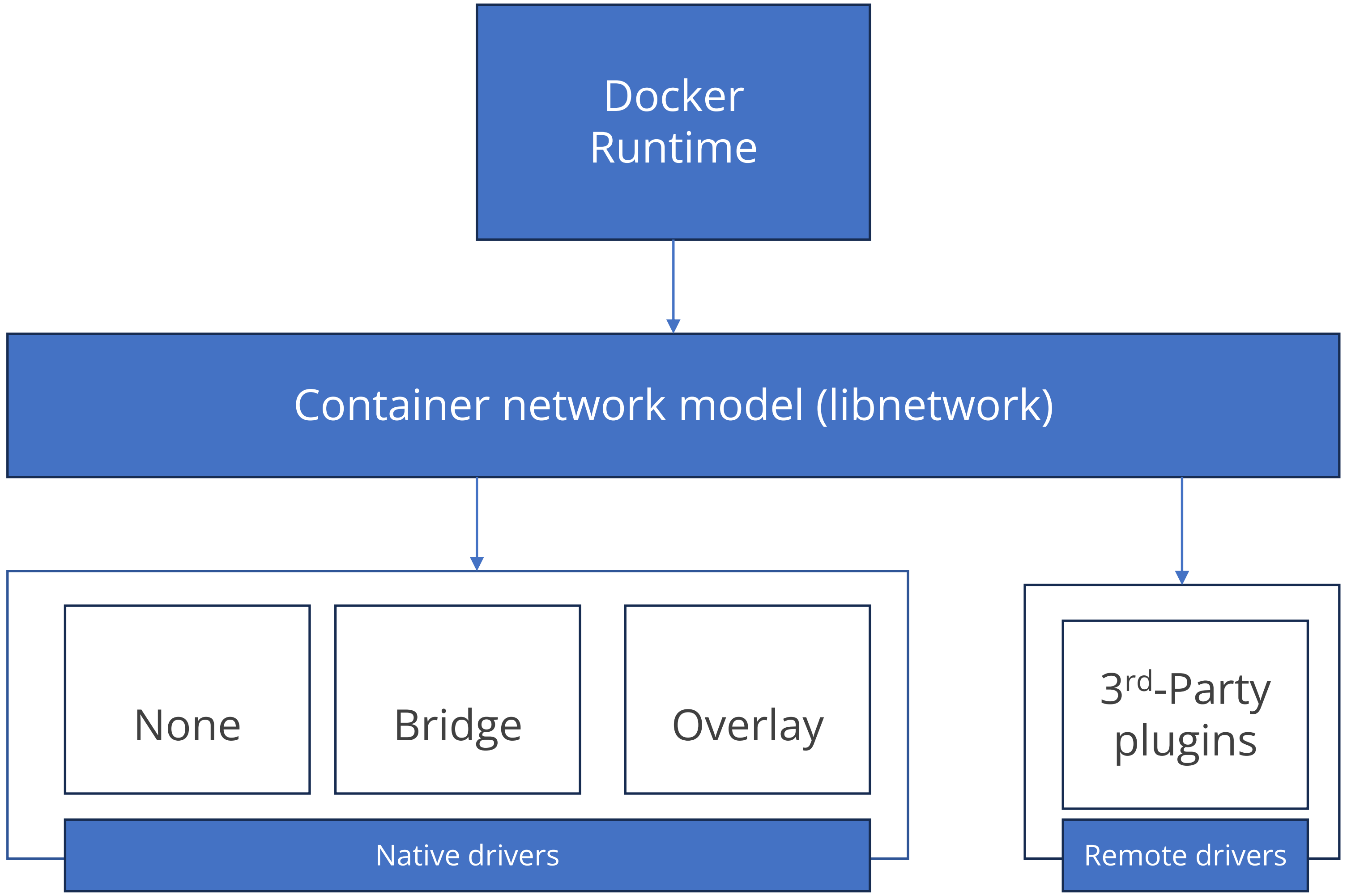
```
$ docker run --rm -dit \
  --network none \
  --name no-net-alpine \
  alpine:latest \
  ash
```

Using this will result in a container with no **eth0**

Overview of Container Networking Model

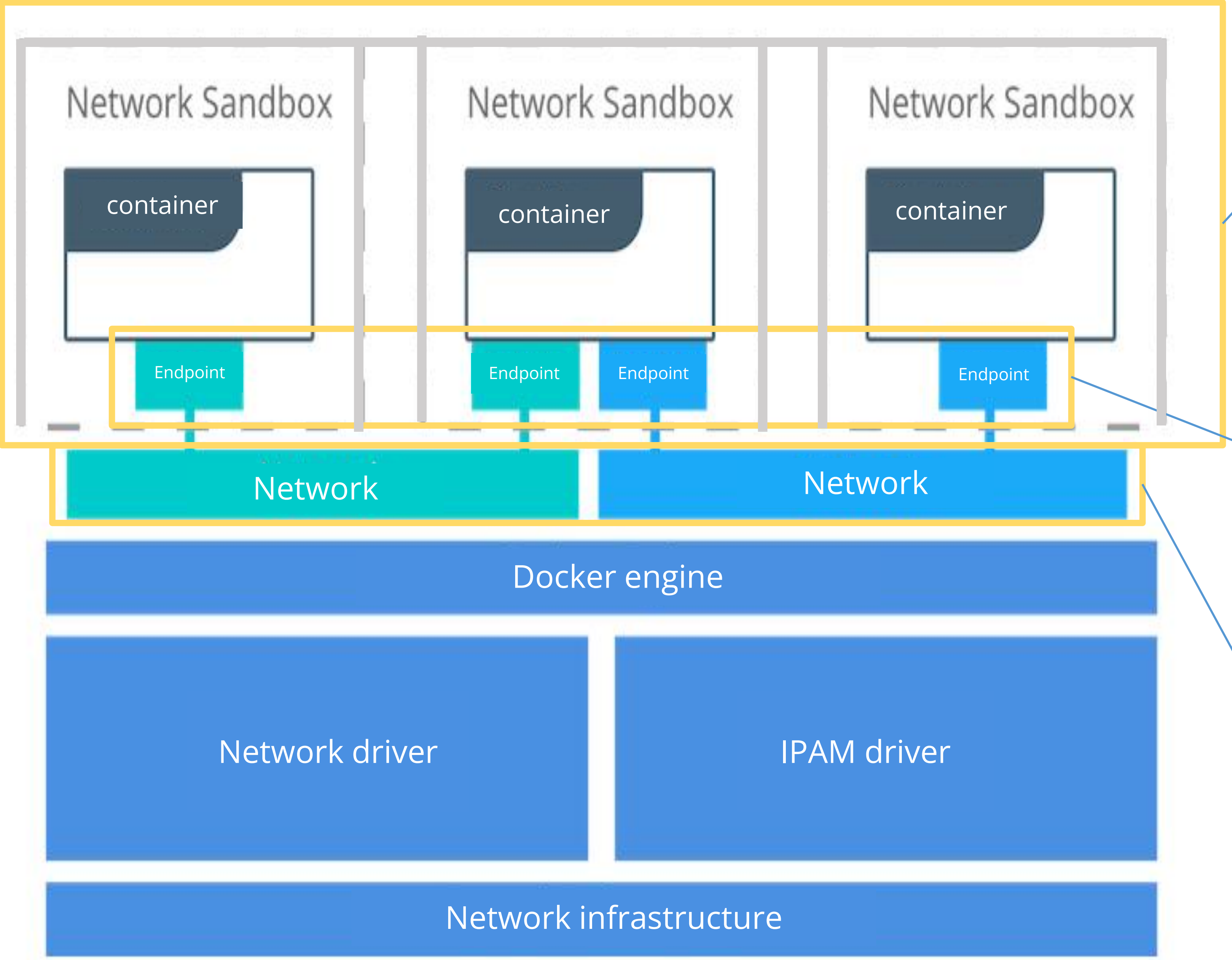
Container Networking Model (CNM)

It is a standard framework that defines how networking should be handled within containers.



Container Networking Model (CNM): Architecture

The CNM provides portability to applications across diverse infrastructures.



A sandbox is composed of a container's network stack configuration.

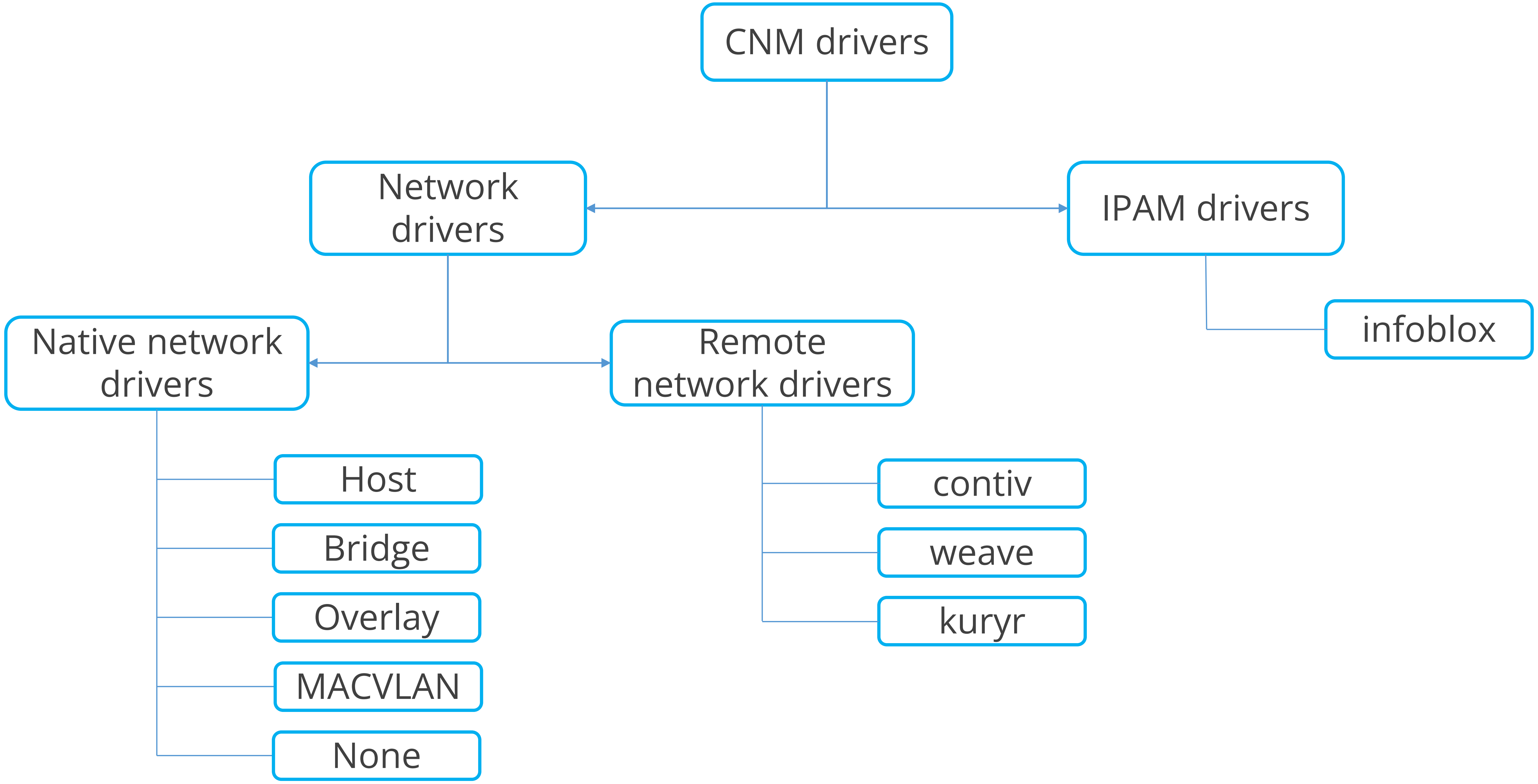
These endpoints join the sandboxes to a network.

A Network is the collection of endpoints.

Source: *https://success.docker.com/article/networking#thecontainernetworkingmodel*

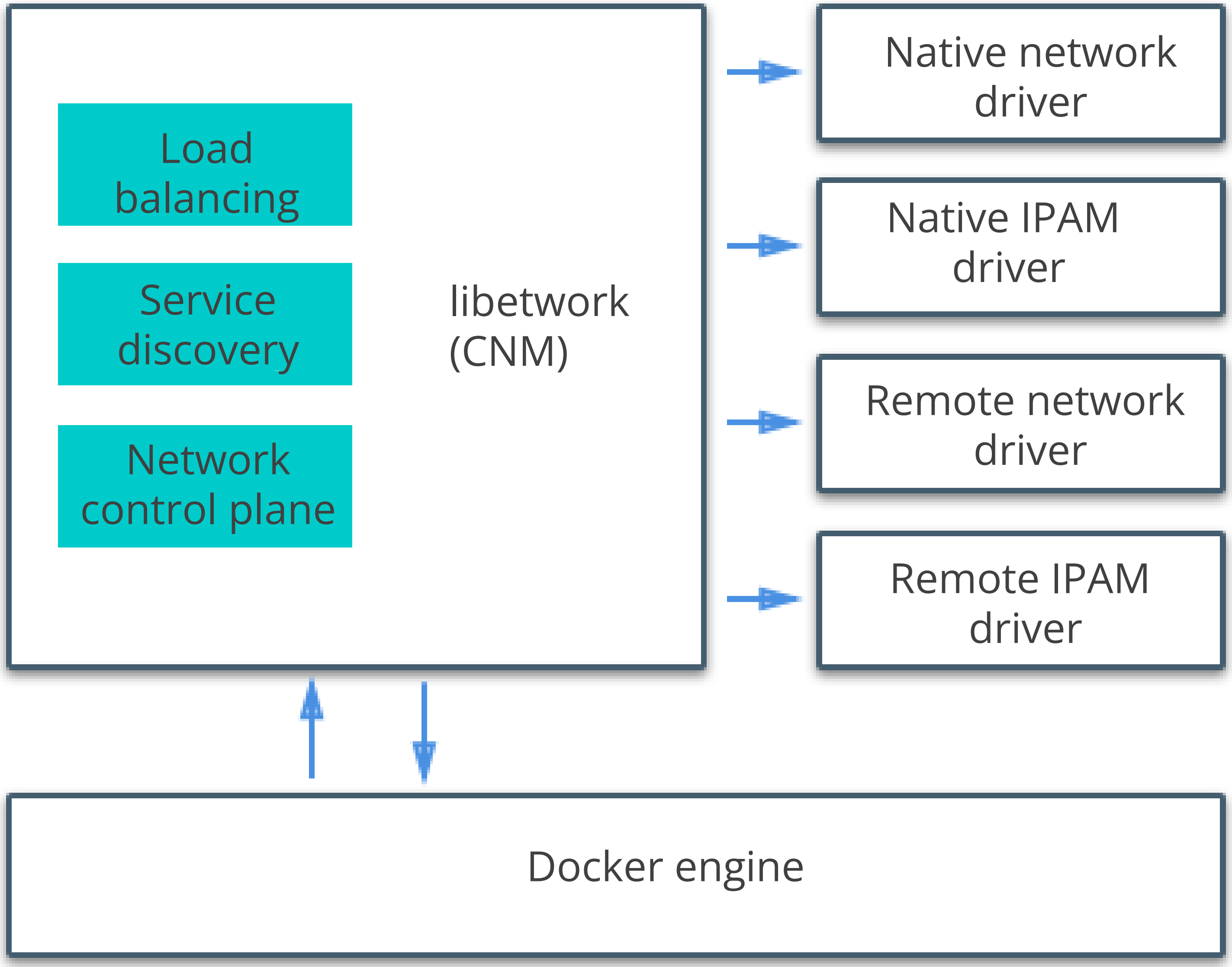
Container Networking Model: Drivers

It defines how network connectivity is managed for containers using various drivers.



Interaction between Docker Engine, CNM, and Network Drivers

The interaction between the Docker engine, the container networking model (CNM), and various network drivers enables efficient management of container networking, including load balancing and service discovery.



Source: https://success.docker.com/article/networking#thecontainernetworkingmodel

Quick Check

You're troubleshooting a Docker container's network connectivity issues. Which CNM component should you check to ensure the container runtime is functioning correctly?

A. Libnetwork

B. Docker runtime

C. 3rd – Party plugins

D. Native drivers

Use Cases of Network Drivers

Bridge Network Driver: Use Case

Problem statement:

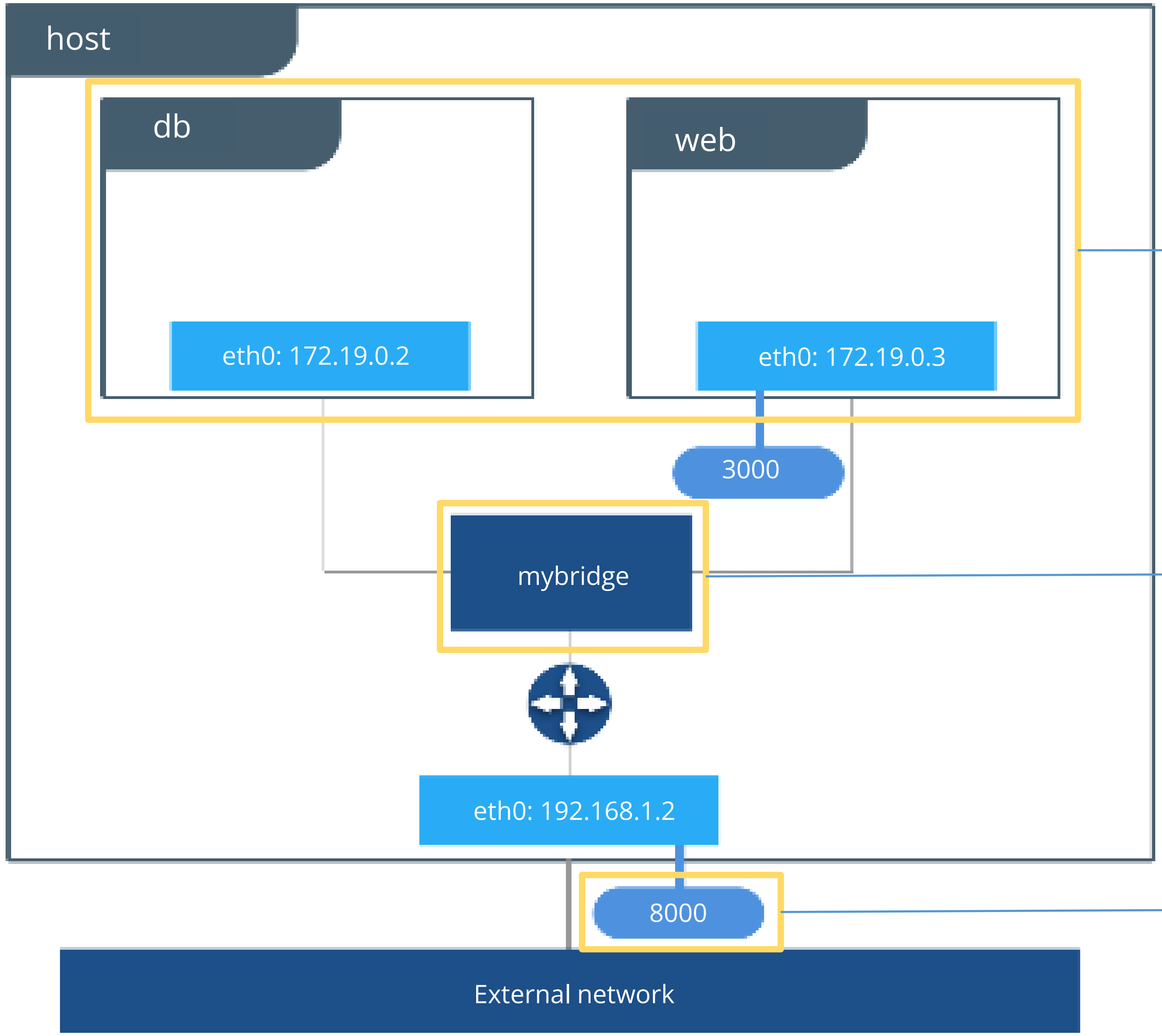
- PetSoft developed the **pets** application with two microservices, **db** (database) and **web** (frontend), both hosted on Docker containers.
- They required efficient container linking to enable seamless communication and service discovery within the local environment.

Challenge:

- PetSoft needed the web container to consistently connect to the **db** container using container names, avoiding IP issues during restarts.
- They also had to expose the **web** service to the external network, enabling users to access from outside the Docker environment.

Bridge Network Driver: Use Case

Solution



db and **web** are containers of an application called pets. This application is available on **<host-ip>:8000**.

mybridge is helping containers **web** to interact with **db** by its container name. This driver is a local scope driver.

An application pets is served on the host at port 8000.

Source: https://www.docker.com/blog/understanding-docker-networking-drivers-use-cases/

Bridge Network Driver: Use Case

Outcome:

- The bridge network allows the web to connect to **db** automatically using container names without manual updates.
- The bridge network facilitates the automatic discovery of the **db** service, ensuring consistent connectivity even after the container restarts due to Docker’s internal DNS.
- By keeping **db** internal and exposing only the web on port 8000, PetSoft ensures security while maintaining user access.

Overlay Network Driver: Use Case

Problem statement:

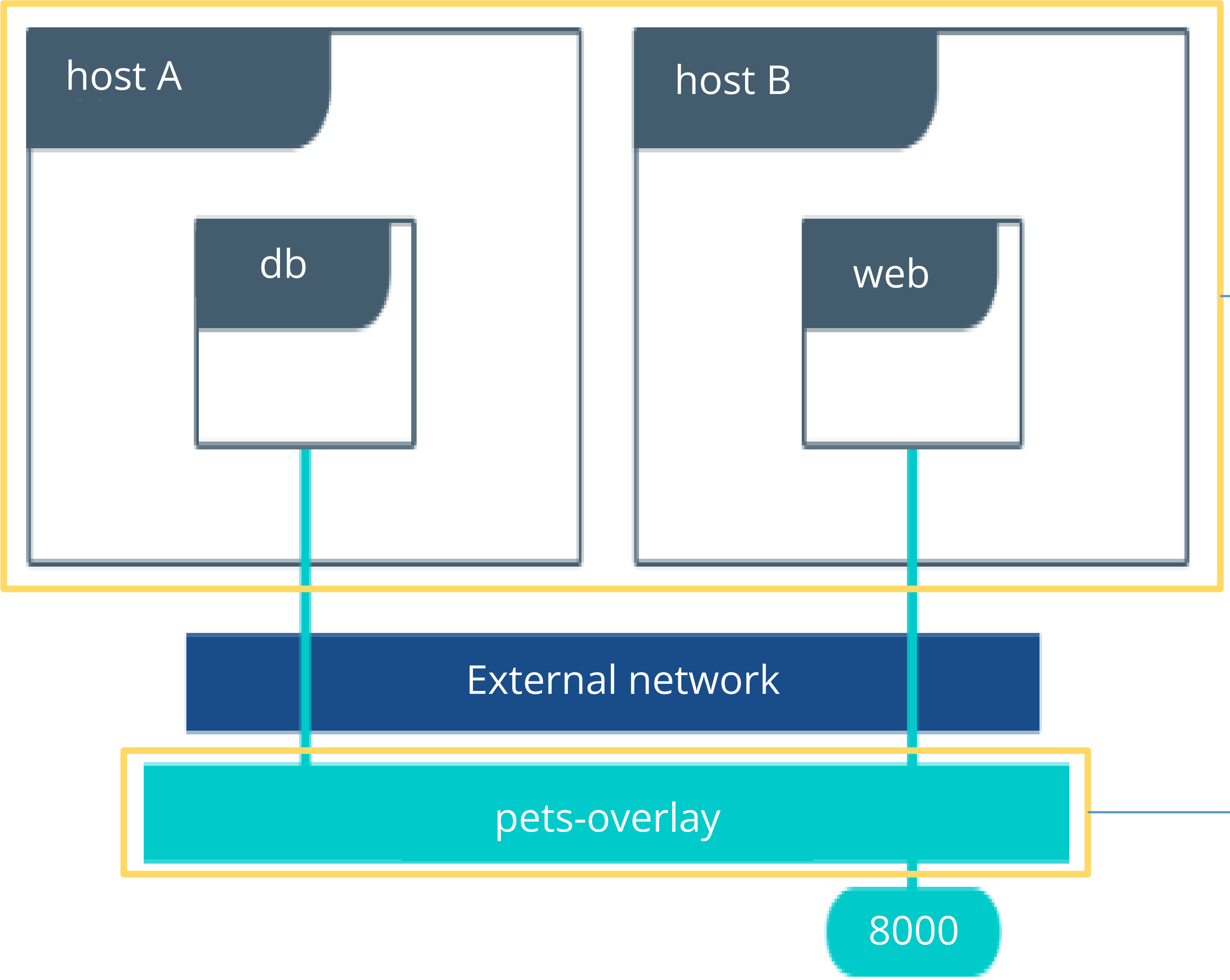
- As the application grew, PetSoft moved the **db** and **web** microservices to separate hosts.
- This shift created the need for secure and seamless communication between the microservices across different hosts.

Challenge:

- The primary challenge was enabling the **db** and **web** containers on separate hosts to communicate as if they were on the same network.
- PetSoft also needed to ensure security and performance while exposing the **web** service to the external network on a specific port.

Overlay Network Driver: Use Case

Solution



The application **pets** is deployed on **db** and **web**. These two containers are present on different hosts.

The overlay network, created by a universal control panel, allows the **db** and **web** to connect by ensuring security and performance while exposing the **web** service to the external network.

Source: https://www.docker.com/blog/understanding-docker-networking-drivers-use-cases/

Overlay Network Driver: Use Case

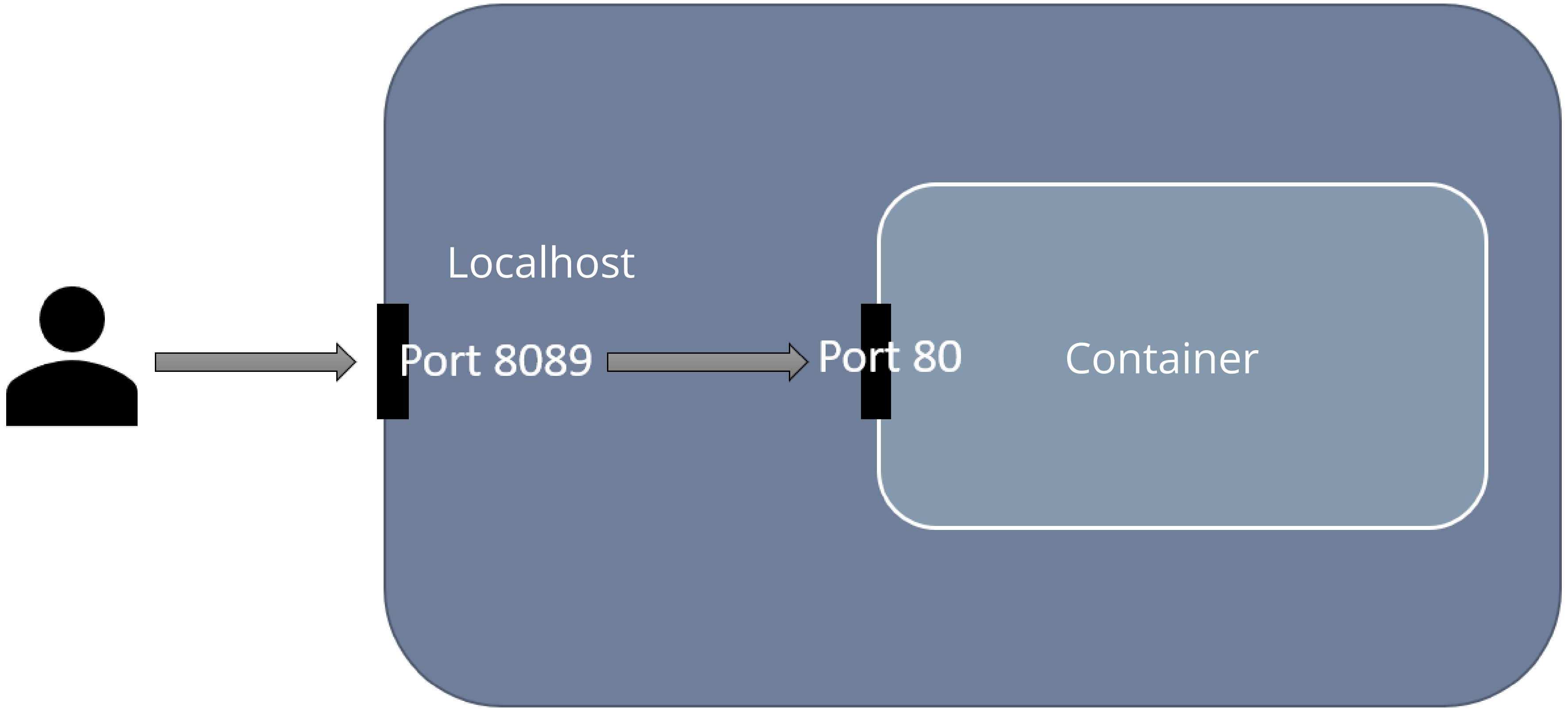
Outcome:

- The overlay network enables **db** and **web** to communicate easily, even on different hosts.
- The overlay network provides encrypted communication between containers, ensuring data security as it traverses between hosts.
- By leveraging the overlay network, PetSoft scales its application horizontally, adding more services across different hosts without compromising performance or connectivity.

Ports

Ports: Introduction

They are the logical endpoints in a network that allow communication between devices.



They serve as channels for different types of network traffic, enabling data to be sent and received between applications on a host system.

Importance of Ports in Containerization

Following are some of the importance of using ports in containerization:

1. Ports make containerized applications accessible from outside by binding container ports to host machine ports.

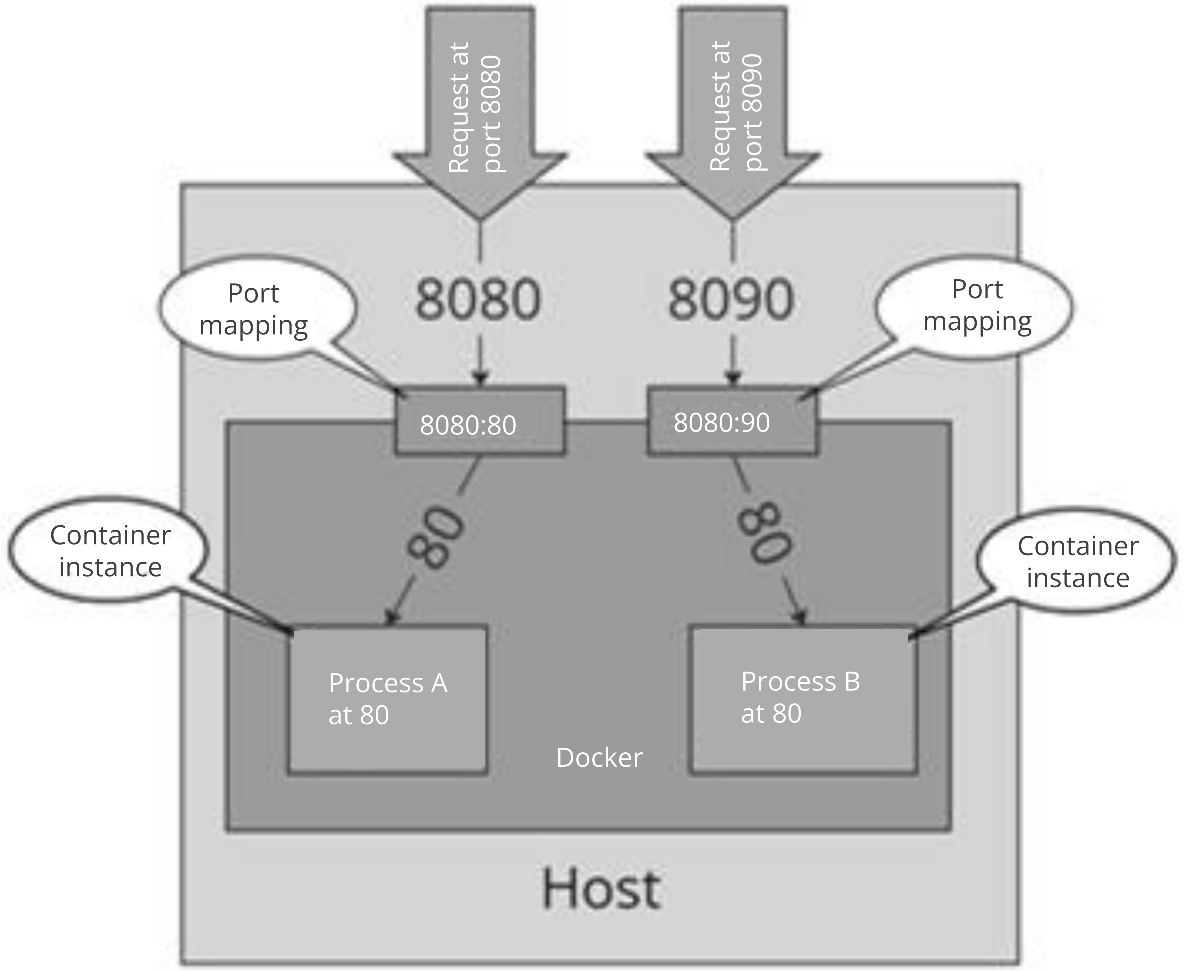
2. It enables service discovery and load balancing, distributing traffic across containers for improved performance.

3. It reduces security risks by limiting exposure and minimizing the application's attack surface.

4. It provides flexibility through custom bindings, preventing conflicts and enabling multiple services to run simultaneously.

Identifying Ports

Docker automatically assigns ports when running a container, but user can specify which ports should be used.



Use the **docker ps** command to identify port mappings between the host and the container, helping to understand how external traffic is directed

Identifying Ports

Role of port:

The host port is bound to the container’s port, allowing the container to connect to the external environment.

Use **docker ps** to find all the ports mapped:

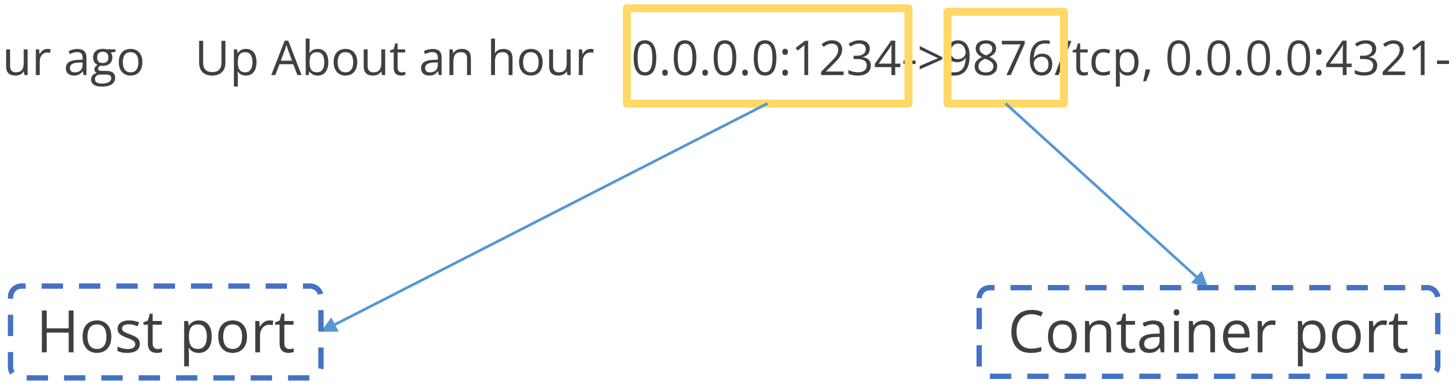
Command:

```
$ docker ps
```

Output:

```
CONTAINER ID     IMAGE          COMMAND       CREATED               STATUS              PORTS
b650456536c7     busybox:latest    top         About an hour ago   Up About an hour   0.0.0.0:1234->9876/tcp, 0.0.0.0:4321-
>7890/tcp

NAMES
 test
```

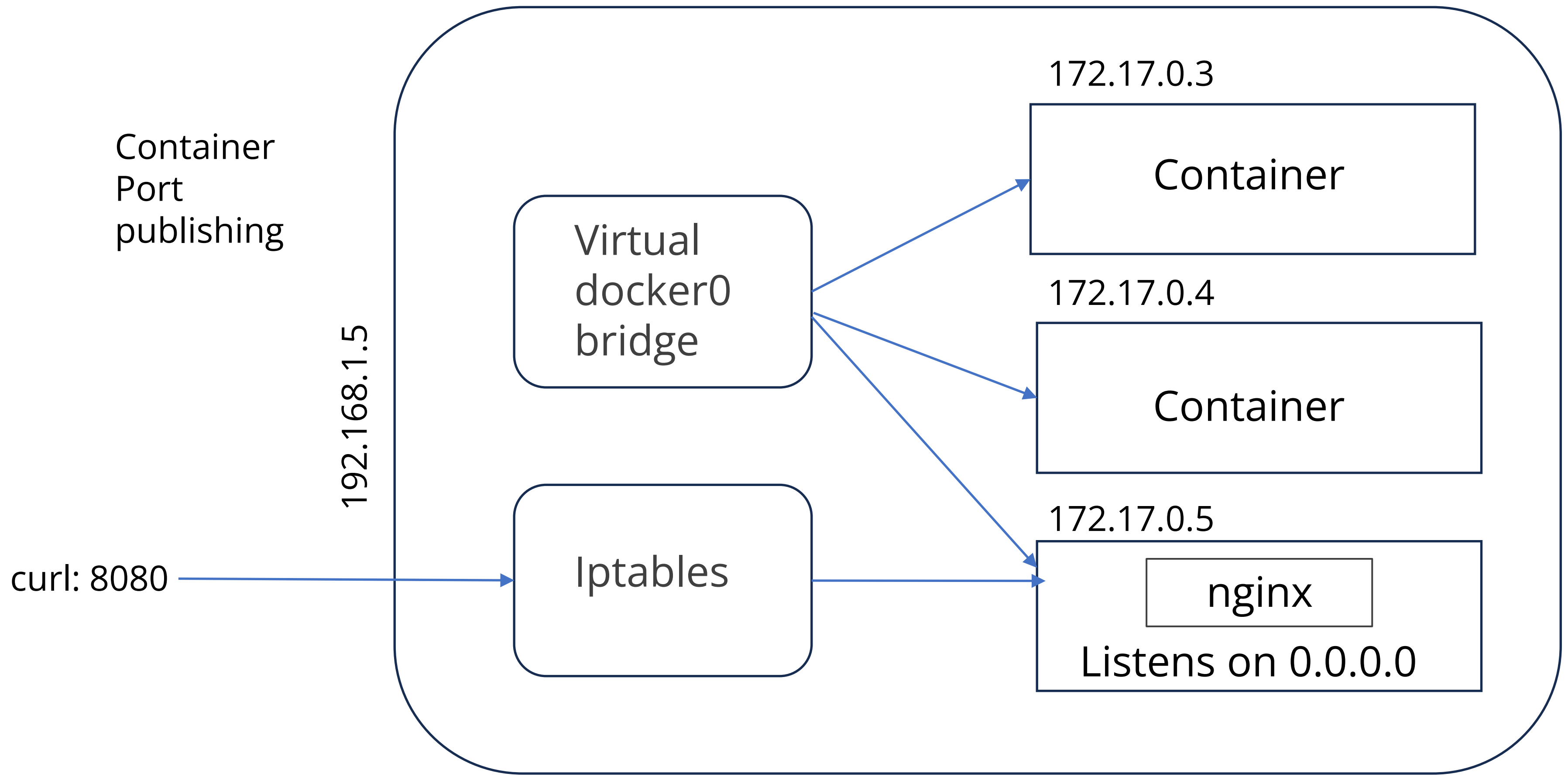


Host port

Container port

Publishing Ports

They map a container's internal port to a host machine port, allowing the service to be accessed externally in Docker.



Use the **docker ps** command to identify port mappings between the host and the container, helping to understand how external traffic is directed.

Publishing Swarm Service Ports

Ways to publish **swarm service port** to hosts that are present outside the swarm:

- Using the routing mesh
- Bypassing the routing mesh

Using the routing mesh:

Use **--publish <PUBLISHED-PORT>:<SERVICE-PORT>** flag to publish a service's port externally to the swarm.

Example:

```
$ docker service create --name my_web \
            --replicas 3 \
            --publish published=8080,target=80 \
            nginx
```

Publishing Swarm Service Ports

Bypassing the routing mesh:

Use the **mode=host** option to the **--publish** flag to publish a service’s port directly on the node where it is running.

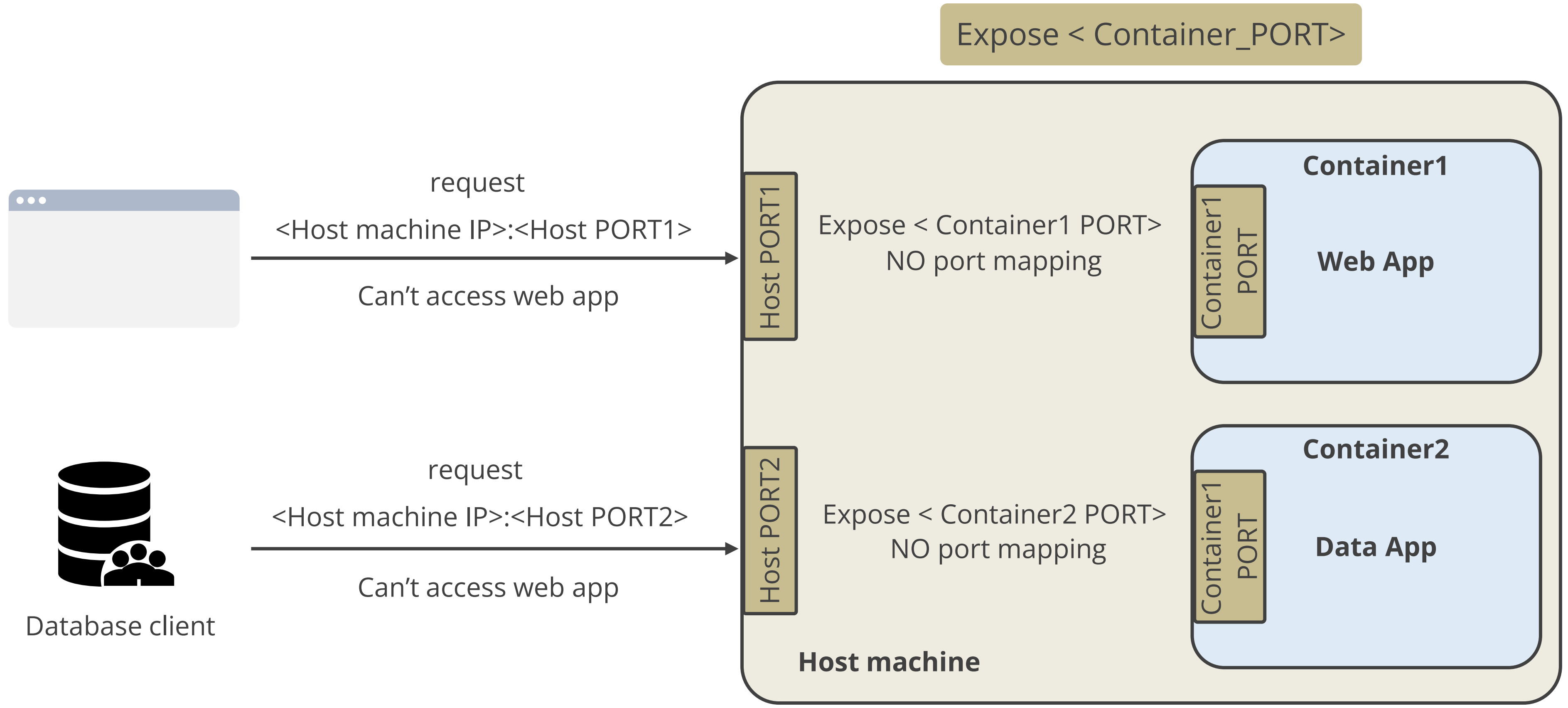
Example:

```
$ docker service create \
     --mode global \
     --publish mode=host,target=80,published=8080 \
     --name=nginx \
     nginx:latest
```

Exposing Ports

It makes a container's internal port available to other containers within the same network, facilitating inter-container communication.

EXPOSE



Ports are exposed using the **EXPOSE** instruction in the Dockerfile or the **--expose** flag at runtime, though this doesn't map them to the host.

Exposing Ports: Example

Exposing ports:

Using **--expose** exposes the ports or range of ports in the container.

Example: Let us expose port 80 without publishing the port

```
$ docker run --expose 80 ubuntu bash
```

Assisted Practice

Publishing Swarm Service Ports

Duration: 20 minutes

Problem Statement:

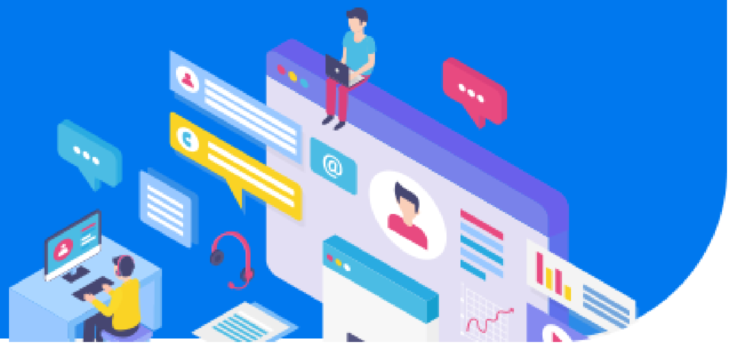
You are tasked with demonstrating how to publish swarm service ports for external access, using both Routing Mesh and direct node publishing to allow external connectivity to services within a Docker swarm.

Outcome:

By completing this demo, you will successfully publish Docker swarm service ports, enabling external access to services and demonstrating effective network configuration within the swarm environment.

Note: Refer to the demo document for detailed steps:
04_Publishing_Swarm_Service_Ports

Assisted Practice: Guidelines



Steps to be followed:

1. Publish swarm service port for external access

Quick Check

You need to verify which ports on your Docker container are mapped to your host machine. Which Docker command would you use?

A. docker inspect

B. docker run

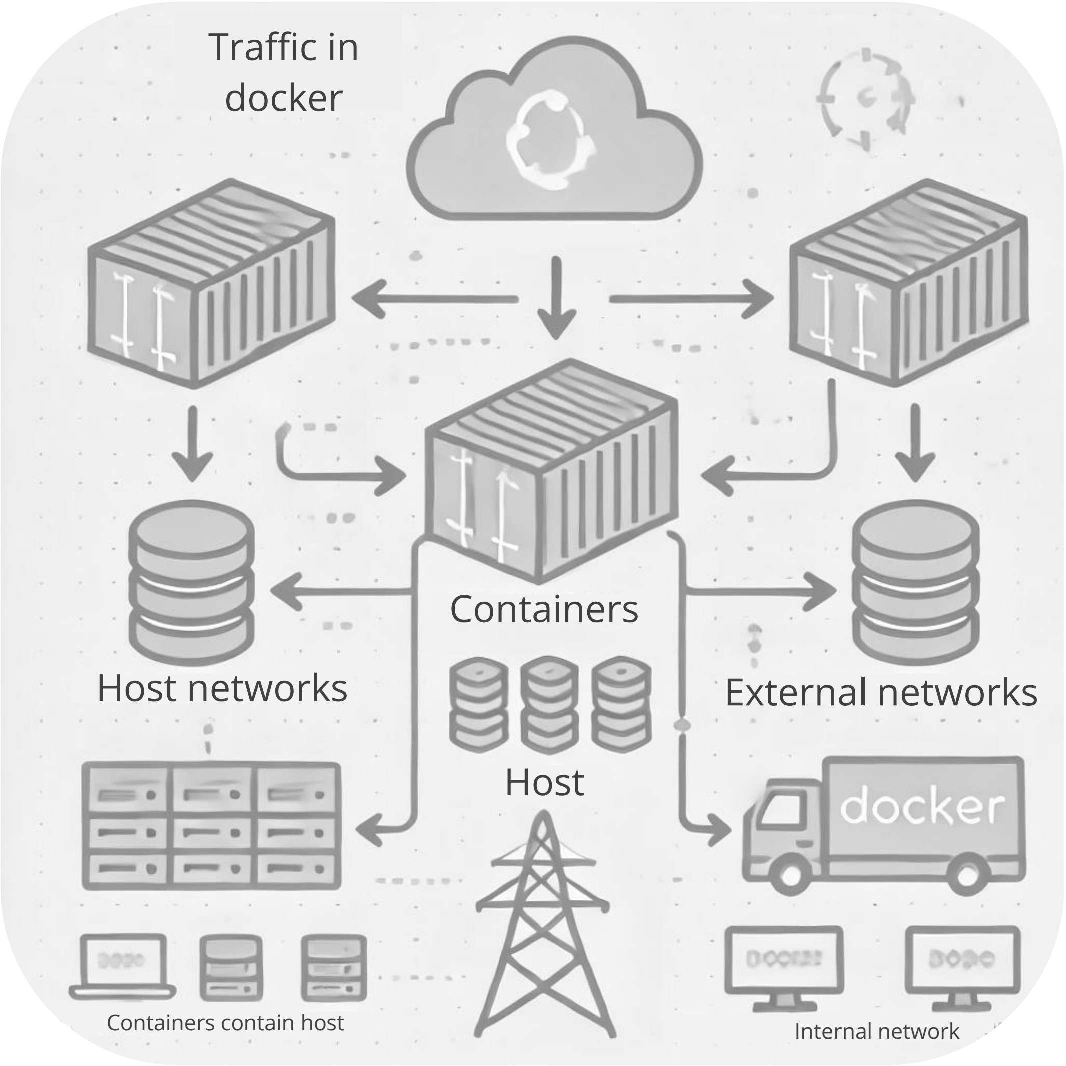
C. docker ps

D. docker exec

Overview of Traffic

Traffic: Introduction

It refers to the network data flow between containers, the host machine, and external networks within a Docker environment.



Managing and understanding traffic in Docker is essential for maintaining the performance, security, and reliability of containerized applications.

Traffic Monitoring in Docker

It involves observing and analyzing the network communications between containers and external networks.

Key metrics to monitor:

01

Network traffic volume
(bytes sent and received)

02

Packet rates
(packets sent and received)

03

Error rates in
network interfaces

04

Connection statistics, including established,
waiting, and failed connections

Logging in Docker

Traffic logging in Docker is crucial for maintaining visibility into the operations and security of containerized applications.

Docker supports several built-in logging drivers that allow for the collection and management of logs generated by Docker containers:

json-file	The default driver that writes logs in json format to files on the host, simplifying readability and parsing
syslog	Forwards logs to a local or remote syslog daemon, ideal for integrating with established logging frameworks
fluentd	Collects logs and sends them to multiple destinations, enhancing searchability and analysis

Inbound Traffic for Swarm Management

Below are the inbound traffic for Swarm management:

Swarm mode port	Purpose
TCP port 2377	Cluster management and raft sync communications
TCP and UDP port 7946	Communication between all nodes
UDP port 4789	Overlay network traffic

While using an overlay network with the encryption option, ensure that the IP protocol 50 (ESP) traffic is allowed

Traffic: Network Ports

Network ports and protocols that swarm cluster components listen on:

Cluster components	Port and protocols	Purpose
Swarm manager	Inbound 80/tcp (HTTP)	Allows **docker pull** commands to work
	Inbound 2375/tcp	Allows Docker Engine CLI commands to the Engine daemon
	Inbound 3375/tcp	Allows Engine CLI commands to the swarm manager
	Inbound 22/tcp	Allows remote management through SSH
Service discovery	Inbound 80/tcp (HTTP)	Allows **docker pull** commands to work
	Inbound **Discovery service port**	Requires setting to the port that the backend discovery service listens on
	Inbound 22/tcp	Allows remote management through SSH

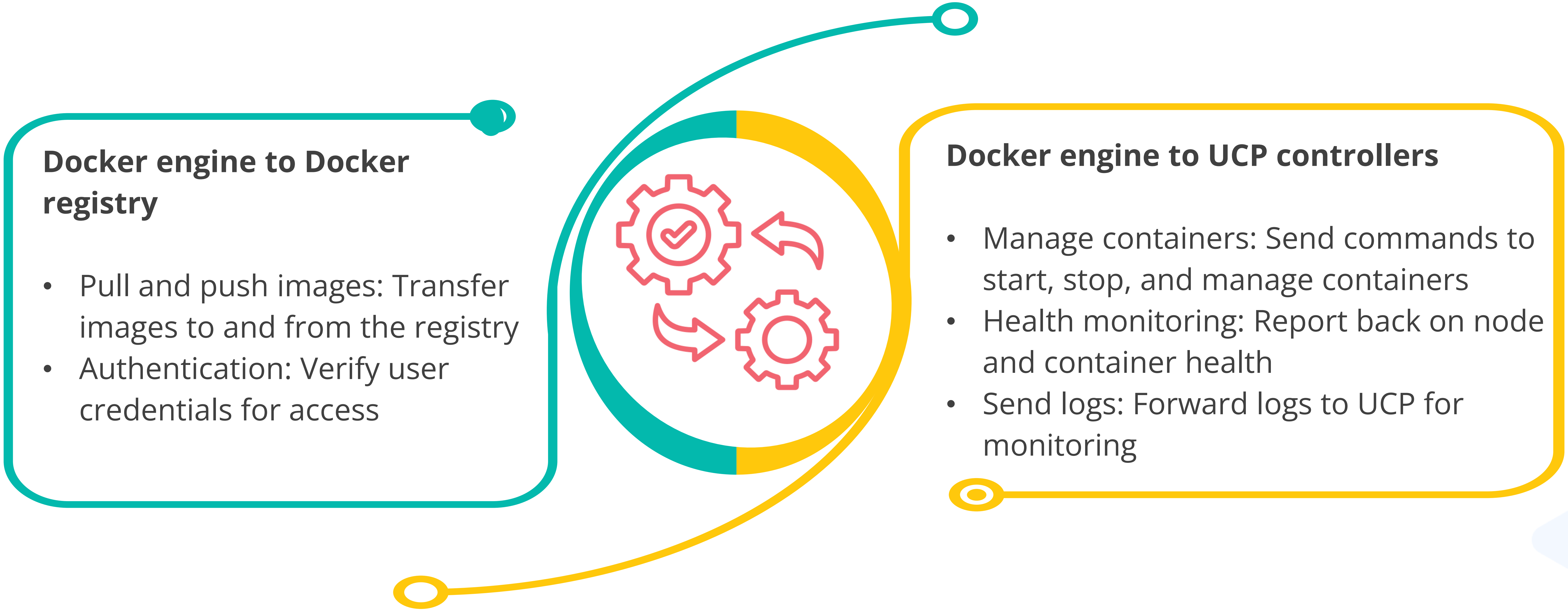
Traffic: Network Ports

Network ports and protocols that swarm cluster components listen on:

Cluster components	Port and protocols	Purpose
Swarm nodes	Inbound 80/tcp (HTTP)	Allows **docker pull** commands to work
	Inbound 2375/tcp	Allows Engine CLI commands to the Docker daemon
	Inbound 22/tcp	Allows remote management through SSH
Custom, cross-host container networks	Inbound 7946/tcp	Allows discovery of other container networks
	Inbound 7946/udp	Allows discovery of container networks
	Inbound <store-port>/tcp	It is a network key-value store service port
	4789/udp	It is required for the container overlay network
	ESP packets	It is required for encrypted overlay networks

Docker Communication Paths

Below are the types of traffic between the Docker engine, Docker registry, and UCP (Universal control plane) controllers:



Docker engine to Docker registry

- Pull and push images: Transfer images to and from the registry
- Authentication: Verify user credentials for access

Docker engine to UCP controllers

- Manage containers: Send commands to start, stop, and manage containers
- Health monitoring: Report back on node and container health
- Send logs: Forward logs to UCP for monitoring

Docker Communication Paths

Below are the types of traffic between the Docker engine, Docker registry, and UCP (Universal control plane) controllers:

UCP controllers to Docker registry

- Fetch images: Retrieve images for system use and user applications
- Sync settings: Keep configuration settings consistent

Between UCP controllers

- Stay in sync: Share state and configuration details
- Data backup: Replicate data for safety and redundancy

Troubleshooting Container and Engine Logs in Docker

In Docker, troubleshooting logs involves analyzing container-specific logs and Docker Engine logs to diagnose and resolve issues effectively.

Below are the key components in troubleshooting container and engine logs in Docker:

Accessing logs:

- Use the **docker logs <container_id>** command to retrieve real-time logs from a specific container
- Use the **--follow** flag to keep streaming logs in real-time, which is helpful for live monitoring and troubleshooting

Troubleshooting Container and Engine Logs in Docker

Engine-level logs:

- Locate the Docker engine logs on the host system, typically under **/var/log/docker.log**
- Review these logs to gain insights into Docker daemon activities, network issues, and container lifecycle events

Log management and rotation:

- Configure log drivers (e.g., json-file, syslog) for better log management and to avoid disk space issues
- Implement log rotation to prevent logs from consuming excessive storage using the **log-opts** option in Docker

Assisted Practice

Configuring Docker to Use External DNS

Duration: 20 minutes

Problem Statement:

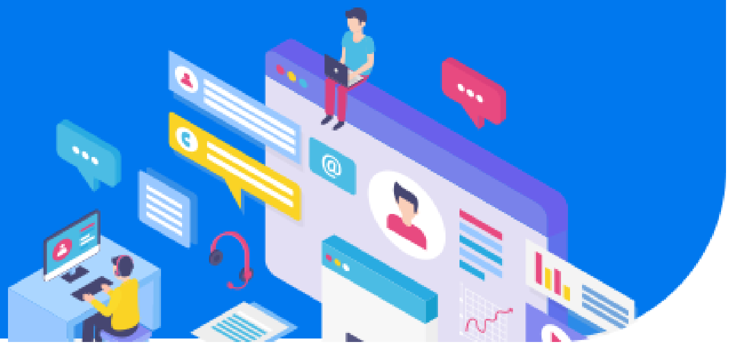
You have been asked to configure your Docker daemon to use external DNS so that it can be used to pull images from an external IP address.

Outcome:

By completing this demo, you will be able to configure Docker to use external DNS and pull images from external IP addresses effectively.

Note: Refer to the demo document for detailed steps: 05_Configuring_Docker_to_Use_External_DNS

Assisted Practice: Guidelines



Steps to be followed:

1. Configure the Docker Daemon config file to use external DNS

Assisted Practice

Troubleshooting Container and Engine Logs

Duration: 20 minutes

Problem Statement:

You have been asked to troubleshoot container and engine logs to resolve connectivity issues between the containers.

Outcome:

By completing this demo, you will be able to identify and resolve connectivity issues between Docker containers by analyzing effectively both container logs and Docker Engine logs, ensuring smooth communication between your containers.

Note: Refer to the demo document for detailed steps:
06_Troubleshooting_Container_and_Engine_Logs

Assisted Practice: Guidelines

Steps to be followed:

1. Set up Docker containers
2. Troubleshoot container and engine logs

Quick Check

You're reviewing the logs for your Docker container but want to confirm which logging driver is being used by default. Which is the default logging driver in Docker?

A. json-file

B. fluentd

C. Syslog

D. journald

Key Takeaways

- Docker networking ensures isolation, security, and connectivity for containerized applications.
- The bridge network serves as the default Docker network on any Linux host running the Docker Engine.
- MACVLAN network facilitates legacy applications by allowing them to connect to the physical network directly.
- The container networking model provides an overview of different networking models and their applications within Docker.
- The importance of ports in containerization highlights the critical role of ports in managing communication between containers and external systems.
- Use cases of network drivers discuss specific scenarios where different Docker network drivers are applied.

Securing Communication Network with Bridge Network

Duration: 60 min.

Project Agenda: To establish a secure and isolated communication channel between Docker containers for secure data exchange and improved application security

Description: Your company is experiencing a transition toward modernized containerized applications using Docker. To address this, you are undertaking a project that aims to set up a secure communication network for applications within Docker containers. This project involves creating a custom Docker bridge network, deploying multiple containers within this network, and demonstrating secure interaction between these containers.

LESSON-END PROJECT

Securing Communication Network with Bridge Network

Perform the following:

1. Create a Docker container
2. Deploy multiple containers
3. Establish communication between the containers in a custom network
4. Connect and verify the container network communication

Expected deliverables: Secure Docker bridge network enabling isolated communication between containers

Thank You