Software Engineering

Prof. Dr. Christoph Schober

Department of Applied Computer Science,
Deggendorf Institute of Technology

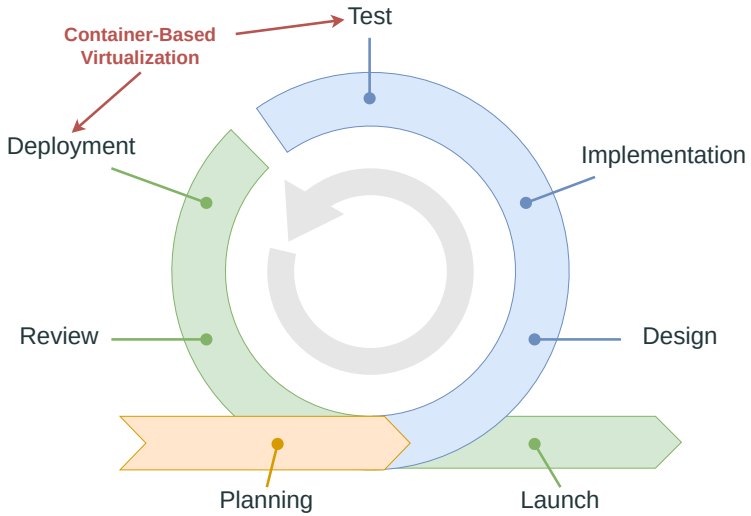
christoph.schober@th-deg.de

Container-based Virtualization

Examples

Find code examples for this lecture under:
`https://mygit.th-deg.de/schober-teaching/examples/docker-examples`

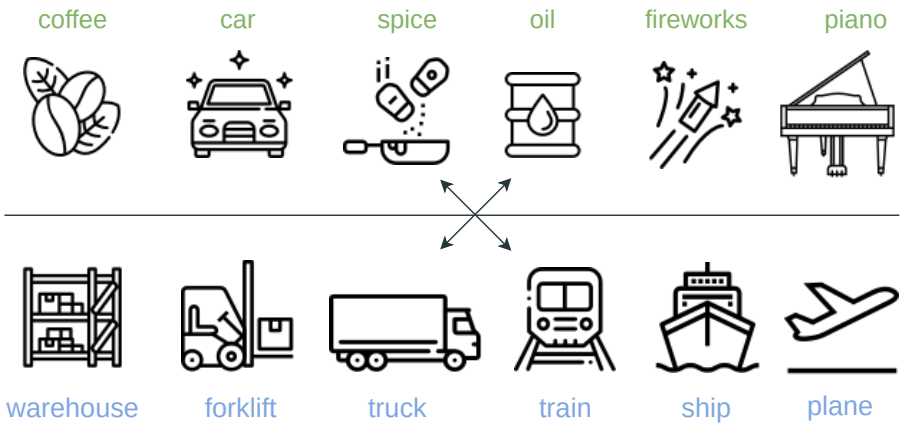
Context


Container-Based
Virtualization
Test
Deployment
Implementation
Review
Design
Planning
Launch

Container-based Virtualization

1. Motivation

Cargo transport before 1950:



Challenge: Multiplicity of goods

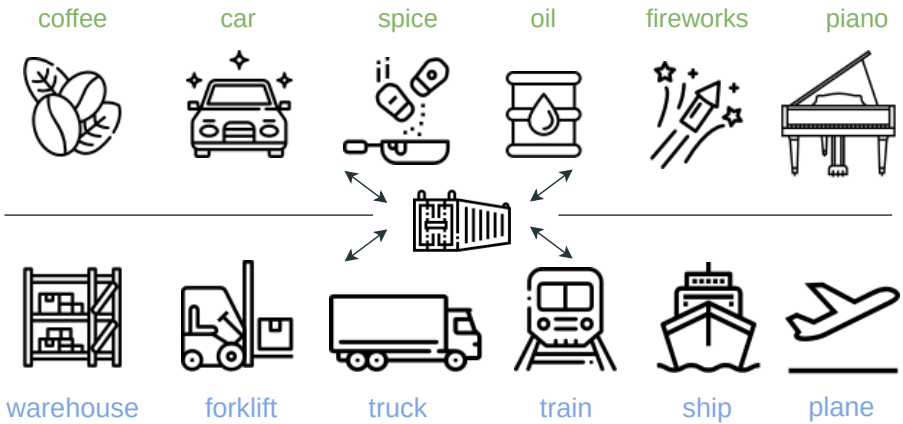
- Rapidly increasing freight traffic worldwide.
- Goods come in different shapes and sizes.
- There are interactions between goods (e.g., coffee & spices).

Challenge: Multiplicity of storage and transport methods

- Under-utilization of space in transportation.
- Transshipment to other means of transport is cumbersome and dangerous.

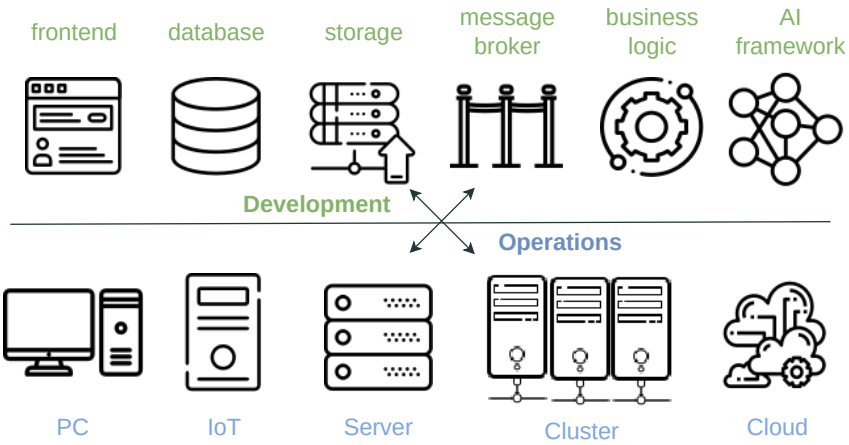
→ Cargo transport pre 1950 was pretty costintensive.

Cargo transport from 1950:



Container revolution: The introduction of intermodal containers which can be loaded, unloaded, stacked, and transported efficiently over long distances.

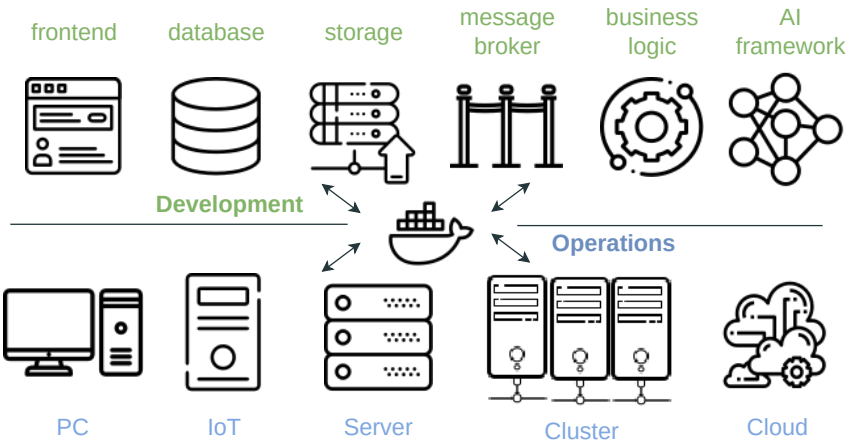
Challenge in Software Engineering: **The DevOps barrier**:



Challenges:

- Dependencies between components within the software.
- Dependencies between the software and external services.
- Increased expenditures on installing external services.
- Increased expenditures on maintaining external services.
- Heterogeneous nature of the operations infrastructure.
- Limited portability of complex software solutions.

Challenge in Software Engineering: **The DevOps barrier**:



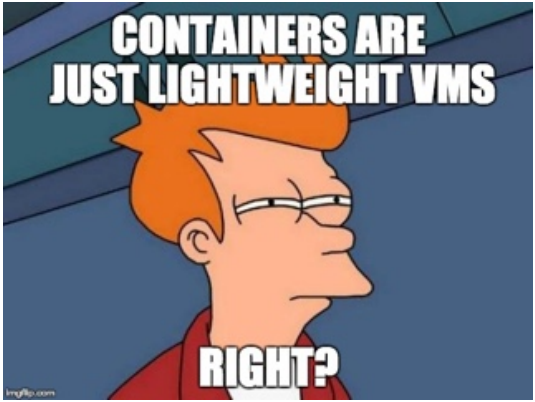
Container-based Virtualization

2. Core Concepts

Container-Based Virtualization

Virtualization technique based on runtime environments (called containers) running on top of a single operating system kernel.

- Virtualizes an operating system rather than the underlying hardware.
- Used to virtually package and isolate applications for deployment.
- Provides a means to easily and universally run software on a variety of hard- and software combinations (including most cloud providers).

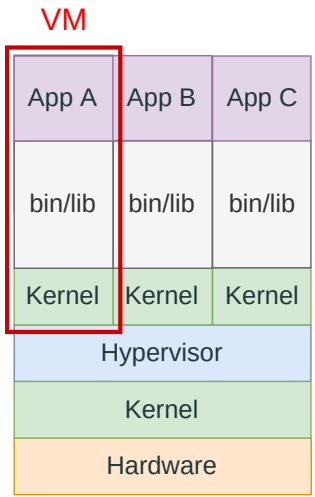


→ NNNOOOOOOOOOOOOOOOOOO !!!!

Virtual Machines (VMs):

- A single (type-2) *hypervisor* virtualizes hardware in the form of a virtual machine
- A VM runs a full guest OS, which is separate and independent from the host OS.
- Software running inside the guest OS is fully isolated from other virtual machines or guest OSes (the same applies for bins/libs).

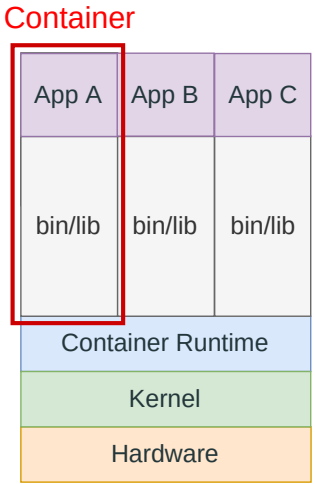
→ Convenient way to securely operate software.



Container:

- A *container runtime* runs containers using the kernel of the host operating system.
- A container does **not include** a kernel.
- Software running inside the guest OS is fully isolated from other virtual machines or guest OSes (the same applies for bins/libs).

→ Convenient and efficient way to securely operate software.



Linux kernel features enabling container-based virtualization:

- **Namespaces:**[1] Partitions kernel resources such that one set of processes sees one set of resources (e.g., PIDs, UIDs, IPC, mount points).

  → Establishes a sandbox, i.e., *limits how much a process can see*.

- **Control Groups (cgroups):**[2] Limits, accounts for, and isolates the resource usage (CPU, memory, disk, I/O, network, etc.) of a process or namespace.

  → Controls the sandbox, i.e., *limits how much a process can use*.

[1] https://man7.org/linux/man-pages/man7/namespaces.7.html
[2] https://man7.org/linux/man-pages/man7/cgroups.7.html

Benefits of containers compared to VMs:

- The startup time is under 1 second (as opposed to minutes).
- The size of a minimal container is a few MB (as opposed to GB).
- Close to bare-metal runtime performance (i.e., speed).
- Resource sharing according to the best-effort service model.

Drawbacks of containers compared to VMs:

- Require a Linux kernel.
- Weaker isolation.
- Graphical applications don't work well.

Container-based Virtualization

3. The Docker Platform

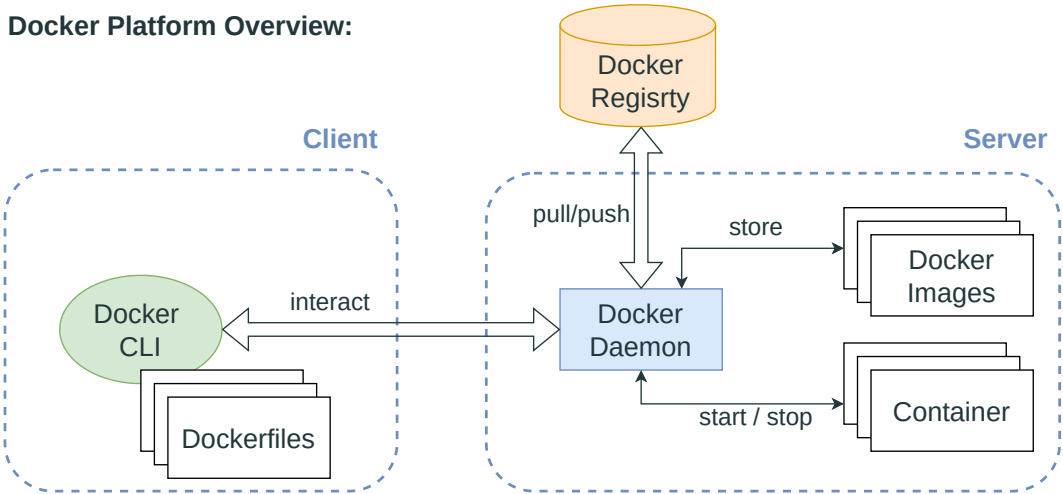
Docker: Still the de-facto standard container runtime.

- 13m+ developers, 7m+ applications
- Based on the *Open Container Initiative* (OCI) format

Alternative container runtimes:

- FreeBSD Jails
- Linux Containers (LXC)
- CRI-O
- Podman

Docker Platform Overview:



Components of the Docker platform:

- **Docker Image**: Standalone, immutable, and executable package of software that includes everything to run a container (code, runtime, tools, etc.).
- **Container**: Runnable instance of an image.
- **Docker Daemon**: Process on the host computer that manages Docker objects such as containers and images (e.g., creates, runs, removes containers).
- **Docker CLI**: Command-line interface to interact with the Docker daemon.
- **Dockerfiles**: Text document specifying the commands to assemble an image.
- **Regisrty**: Repository in which images are stored and distributed.

First Steps:

- Installation manual: `https://docs.docker.com/get-docker/`
- Hello World with Docker:

```
1 docker run hello-world
```

Documentation:

- A comprehensive documentation is available at: `https://docs.docker.com/engine/reference/commandline/docker/`.

The Docker Hub: Centralized repository of Docker images.

- Cloud-based service provided by the Docker.
- Allows developers to store, manage, and distribute Docker.
- Default registry for Docker images used by millions of developers.
- *Warning:* Everyone can publish Docker images at the Docker hub.

Link: `https://hub.docker.com`

Running example:

- Let us consider a small application to calculate the GCD.
- The source code is written in Java and available in iLearn.
- The code shall run on the **Amazon Corretto**[3] JDK.

→ The goal is to apply container-based virtualization to realize the scenario.

[3] Production-ready OpenJDK distribution offering runtime performance improvements, bug fixes, security patches and **long-term support** at no additional costs.

Container-based Virtualization

4. The Docker CLI

Pull an image [4] (from the Docker Hub[5]):

```
1 docker pull [OPTIONS] NAME[:<TAG>|@DIGEST]
```

Convention: Tags define the version of the service. The tag `latest` is the default tag and is supposed to reference the latest version.

Examples:

- Pull the Corretto image:

```
1 docker pull amazoncorretto                    # default tag 'latest'
2 docker pull amazoncorretto:17                 # version tag
3 docker pull amazoncorretto@sha256:762d7c...   # digest
```

[4] https://docs.docker.com/engine/reference/commandline/pull/
[5] https://hub.docker.com

Create a container[6]:

```
1 docker create [OPTIONS] IMAGE [COMMAND] [ARG..]
```

Example:

- Create a container from the Corretto image:

```
1 docker create -it --name jdk amazoncorretto:17 bash
```

- The option `--name` assigns a name to the container.
- The options `-it` keep `STDIN` open and allocate a pseudo-TTY device.

[6] https://docs.docker.com/engine/reference/commandline/create/

Start a container [7]:

```
1 docker start [OPTIONS] CONTAINER [CONTAINER...]
```

Example:

- Start the previously created Corretto container:

```
1 docker start jdk
```

[7] https://docs.docker.com/engine/reference/commandline/start/

List existing containers[8]:

```
1 docker ps [OPTIONS]
```

Examples:

- List running containers:

```
1 docker ps
```

- List all containers (incl. the stopped ones):

```
1 docker ps -a
```

[8] https://docs.docker.com/engine/reference/commandline/ps/

Attach the standard streams of the terminal emulator to a running container[9]

```
1 docker attach [OPTIONS] CONTAINER
```

Example:

- Attach to the container named `jdk`:

```
1 docker attach jdk
```

[9] https://docs.docker.com/engine/reference/commandline/attach/

Run a command in a running container[10]

```
1 docker exec [OPTIONS] CONTAINER COMMAND [ARG..]
```

Examples:

- Run the command `java --version` inside the container `jdk`:

```
1 docker exec jdk java --version
```

- Start a `bash` shell inside the container `jdk` and attach to it:

```
1 docker exec -it jdk bash
```

[10] https://docs.docker.com/engine/reference/commandline/exec/

Stop a container[11]

```
1 docker stop [OPTIONS] CONTAINER [CONTAINER...]
```

Example:

- Stop the container `jdk`:

```
1 docker stop jdk
```

[11] https://docs.docker.com/engine/reference/commandline/stop/

Delete a container[12]

```
1 docker rm [OPTIONS] CONTAINER [CONTAINER...]
```

Example:

- Delete the container `jdk`:

```
1 docker rm jdk
```

[12] https://docs.docker.com/engine/reference/commandline/rm/

Run a command in a new container[13]

```
1 docker run [OPTIONS] IMAGE [COMMAND] [ARG..]
```

Note: The *run* command behaves the same as *create*, followed by *start*.

Examples:

- Create a Corretto container, run a command, and delete it:

```
1 docker run --rm amazoncorretto:17 java --version
```

- Create an interactive Corretto container and detach immediately.

```
1 docker run -dit --name jdk amazoncorretto:17
```

[13] https://docs.docker.com/engine/reference/commandline/run/

Fetch the logs of a container[14]

```
1 docker logs [OPTIONS] CONTAINER
```

Example:

- Create a Corretto container, write ”Hi” to the console, spawn a new bash shell. Afterwards check the logs of the container (should contain ”Hi”).

```
1 docker run --rm -dit --name jdk amazoncorretto:17 bash -c "echo 'Hi'; bash"
2 docker logs jdk
```

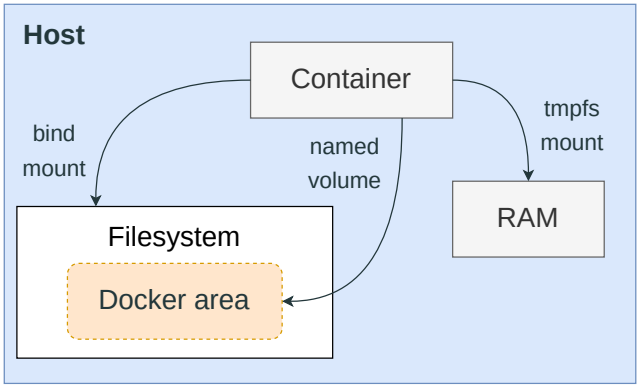
[14] https://docs.docker.com/engine/reference/commandline/log/

Container-based Virtualization

5. Persistent Storage

Warning: Changes made to the filesystem in a container are volatile. Such changes are not persisted beyond the lifecycle of the container.

Mechanisms for persisting data generated by Docker containers:



Bind Mount[15]

Mounts a file or directory on the *host* machine into a container.

Example:

```
1 docker run --rm -v /home/schober/workspaces/gcd-in-java:/src amazoncorretto:17 \
2     java /src/GCD.java
```

- The option `-v` specifies the bind mount in the form of `src:dest`.
- An alternative syntax is provided by the option `--mount`.

[15] `https://docs.docker.com/storage/bind-mounts/`

Named Volumes[16]

A named storage area located in the host filesystem and managed by Docker.

Volumes have several advantages over bind mounts:

- Volumes are easier to back up or migrate than bind mounts.
- Volumes work on both Linux and Windows containers.
- Volumes can be safely shared among multiple containers.
- Provide better performance than bind mounts from Mac and Windows hosts.

[16] `https://docs.docker.com/engine/reference/commandline/volume/`

Create a volume:

```
1 docker volume create [OPTIONS] [VOLUME]
```

List all volumes:

```
1 docker volume ls [OPTIONS]
```

Show the details of a single volume:

```
1 docker volume inspect [OPTIONS] VOLUME
```

Delete a volume:

```
1 docker volume rm [OPTIONS] VOLUME
```

Example:

```
# create the named volume
docker volume create gcd-data

# create a new Corretto container and detach
docker run --rm -dit --name jdk -v gcd-data:/src -w /src amazoncorretto:17

# copy the source file from the host filesystem to the container
docker cp GCD.java jdk:/src

# run the java application using Corretto
docker exec jdk java GCD.java

# stop and delete the container (--rm option)
docker stop jdk
```

TempFS Mount

Volume that is stored in the host systems memory only, and never written to the hosts filesystem (Linux only).

Example:

```
1 docker run --rm -dit --name jdk --mount type=tmpfs,destination=/secret \
2     amazoncorretto:17
```

→ Used during the lifetime of a container to store non-persistent state or sensitive information (such as passwords or keys).

Container-based Virtualization

6. Application Configuration

Environment Variable

A named value that is stored in the operating system's environment and can be accessed and modified by programs running on the system.

- Used to store configuration settings, system paths, and other data.
- Allow for more flexible configuration of applications.
- Provide a way to pass information from the host system to a Docker container.

Declare a user-defined environment variable[17]:

```
1 MY_FOO=bar  # MY_FOO is the variable name, BAR is its value
```

Declare a user-defined environment variable that is visible from subshells:

```
1 export MY_FOO=bar
```

Reading an environment variable:

```
1 echo $MY_FOO
```

Delete an user-defined environment variables:

```
1 unset MY_FOO
```

[17] Variable names should have all capital letters with underscores as separators.

Best Practice: Software applications running in Docker containers are typically configured using environment variables. Examples:

- Adding options to the JVM:

```
1 docker run --rm -dit -e JAVA_TOOL_OPTIONS=-Xmx1g amazoncorretto:17
```

- Running a PostgreSQL container:

```
1 docker run -dit --name postgres \
2   -e POSTGRES_DB=db1 \
3   -e POSTGRES_USER=schober \
4   -e POSTGRES_PASSWORD=secret \
5 postgres:14.2
```

Container-based Virtualization

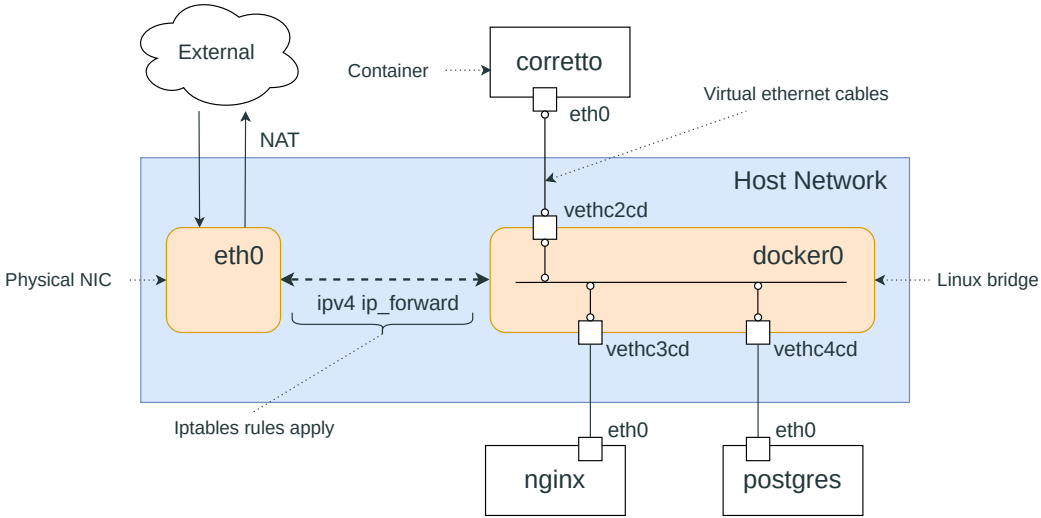
7. Networking

Docker Networking

Mechanism that allows communication between Docker containers and the host system or other external networks.

- Uses a Linux bridge[18] network by default (interface `docker0`).
- When a container is started, it is assigned a virtual interface (`veth...`), and connected to the default bridge.
- Each container runs its own network stack and is assigned a unique IP address within the bridge network.

[18] A kernel module that behaves like a network switch.


External
Container
corretto
eth0
Virtual ethernet cables
NAT
Host Network
vethc2cd
Physical NIC
eth0
docker0
Linux bridge
ipv4 ip_forward
vethc3cd
vethc4cd
Iptables rules apply
eth0
eth0
nginx
postgres

Portforwarding: Expose a port on the host system and forward traffic to that port to a specific port on a container. Examples:

- Corretto:

```
docker run --rm -dit -p 8001:8080 amazoncorretto:17
```

- PostgreSQL:

```
docker run -dit --name postgres \
  -p 5000:5432 \
  -e POSTGRES_DB=db1 \
  -e POSTGRES_USER=schober \
  -e POSTGRES_PASSWORD=secret \
postgres:14.2
```

Host network mode: Allows a container to use the network stack of the host system instead of creating its own isolated network.

Example:

```
1 docker run --rm -dit --network=host amazoncorretto:17
```

If a container is started in that mode, it is directly connected to the physical network interface.

Warning: This mode poses a (significant) security risk.

Custom networks: The docker daemon supports adding virtual networks (bridges), which provide a secure and flexible war of connecting containers.

Example:

```
1 docker network create my-network
2 docker run --rm -dit --network=my-network amazoncorretto:17
3 docker run --rm -dit --network=my-network postgres:14.2
```

When a container is started on a custom network, it is assigned an IP address within the subnet of the network, which is separate from the default bridge.

Container-based Virtualization

8. Dockerfiles

Dockerfile

A text file that contains a series of instructions for building a Docker image.

Example:

```
1  FROM amazoncorretto:17
2  LABEL maintainer=christoph.schober@th-deg.de
3
4  RUN yum update -y && yum install -y ca-certificates && yum clean all
5  WORKDIR /opt/gcd
6  COPY GCD.java .
7  RUN javac GCD.java
8
9  ENTRYPOINT ["java"]
10 CMD ["GCD"]
```

Build[19] a Docker image from a Dockerfile:

```
1 docker build [OPTIONS] PATH | URL | -
```

Example:

```
1 docker build . -t gcd:latest
```

Naming and Tagging:

- The argument `-t` names and optionally tags the image in `name:tag` format.
- Tags usually indicate the version of the service running inside the container.
- If no tag is provided, the default tag `latest` is used.

[19] `https://docs.docker.com/engine/reference/commandline/build/`

Dockerfiles are separated into *build stages*, initialized with the `FROM`[20] statement:

```
1 FROM IMAGE [AS NAME]
```

The referenced Docker image is called *base image*, as subsequential statements perform modifications to it.

Note: Dockerfiles always start with a `FROM` statement. Example:

```
1 FROM amazoncorretto:17 AS jdk
```

[20] `https://docs.docker.com/engine/reference/builder/#from`

The statement LABEL[21] adds metadata to the image.

```
1 LABEL KEY=VALUE [KEY=VALUE..]
```

Plea: Do it better than 90% of all developers and annotate your images.

Example:

```
1 LABEL maintainer=christoph.schober@th-deg.de
2 LABEL version=2.7.1 revision=r0
3 LABEL description="one nice image"
```

[21] https://docs.docker.com/engine/reference/builder/#label

The statement RUN[22] executes a Linux shell command that may change the file system of the Docker image:

```
1 RUN COMMAND
```

Example (creates a file):

```
1 RUN echo "Hello World" > /opt/file.txt
```

Example (installs the package `curl` with `apt`):

```
1 RUN apt-get update && apt-get install -y curl
```

[22] https://docs.docker.com/engine/reference/builder/#run

The statement `COPY`[23] copies files and directories from the host system into the Docker image, which changes the file system of the Docker image:

```
1 COPY [OPTIONS] SRC [SRC...] DEST
```

Example (copies the file `main.py` to `/app` inside the Docker image):

```
1 COPY main.py /app
```

Example (copies the directory `src` to `/app` inside the Docker image):

```
1 COPY src /app
```

Note: Paths may be relative to the working directory of `docker build`.

[23] `https://docs.docker.com/engine/reference/builder/#copy`

The statement **ENV**[24] sets environment variables within the Docker image:

```
1 ENV KEY=VALUE [KEY=VALUE...]
```

Example:

```
1 ENV JAVA_TOOL_OPTIONS=-Xmx1g
```

Note: Environment variables specified during container instantiation have higher precedence.

[24] https://docs.docker.com/engine/reference/builder/#env

The statement `ARG`[25] defines build-time variables passed in as arguments to the `docker build` command:

```
1 ARG KEY[=VALUE]
```

Example:

```
1 ARG CORRETTO_VERSION=17
2 FROM amazoncorretto:${CORRETTO_VERSION}
```

Build command:

```
1 $ docker build . -t gcd --build-arg CORRETTO_VERSION=20
```

[25] https://docs.docker.com/engine/reference/builder/#arg

The statement `ENTRYPOINT`[26] sets the primary command that will be run when a Docker container is started from the image:

```
1 ENTRYPOINT COMMAND [PARAMS...]
```

Example (`exec`-format, recommended):

```
1 ENTRYPOINT ["java", "-d", "/opt/gcd"]
```

Example (`shell`-format):

```
1 ENTRYPOINT "java -d /opt/gcd"
```

[26] https://docs.docker.com/engine/reference/builder/#entrypoint

The statement `CMD`[27] is supposed to specify default arguments for the primary command:

```
1 CMD COMMAND [PARAMS...]
```

Example:

```
1 ENTRYPOINT ["java"]
2 CMD ["-d", "/opt/gcd", "GCD"]
```

Note: `CMD` can also specify a default command that will run when a Docker container is started from the image (e.g., in case no entrypoint is specified).

[27] `https://docs.docker.com/engine/reference/builder/#cmd`

Difference between `ENTRYPOINT` and `CMD`:

- When a container is run, any additional arguments passed on the command line are appended to the `ENTRYPOINT` command:

```
1 docker run --rm gcd --version  # runs java --version
```

- The statement `CMD` provides a default value for the command-line argument set in the example.

In summary, `ENTRYPOINT` sets the primary command for the container, while `CMD` provides default arguments for that command or specifies a default command if no `ENTRYPOINT` is defined.

Further reading: See the reference documentation[28] for all 32 Dockerfile statements (as of 20.10).

[28] `https://docs.docker.com/engine/reference/builder/`

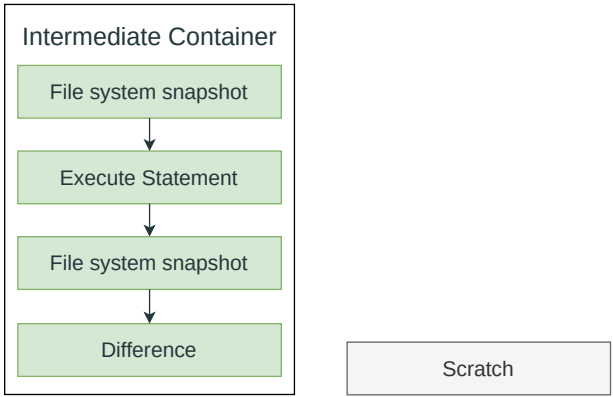
Docker image build process:

```
FROM amazoncorretto:17

RUN yum update -y && yum install -y
ca-certificates && yum clean all

COPY GCD.java /opt

CMD ["java", "/opt/GCD.java"]
```



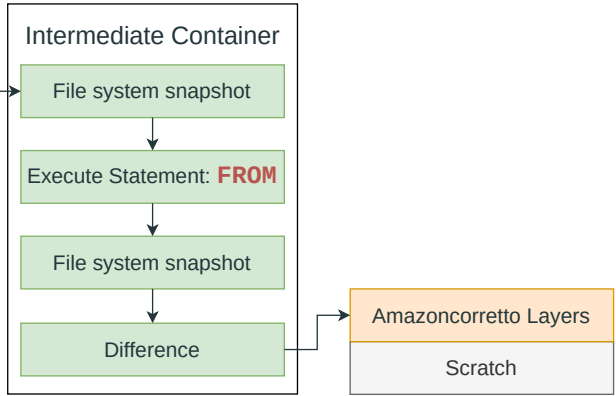
Docker image build process:

```
FROM amazoncorretto:17

RUN yum update -y && yum install -y
ca-certificates && yum clean all

COPY GCD.java /opt

CMD ["java", "/opt/GCD.java"]
```



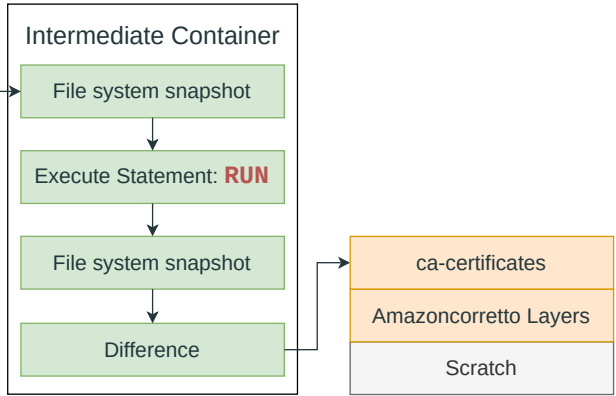
Docker image build process:

```
FROM amazoncorretto:17

RUN yum update -y && yum install -y
ca-certificates && yum clean all

COPY GCD.java /opt

CMD ["java", "/opt/GCD.java"]
```



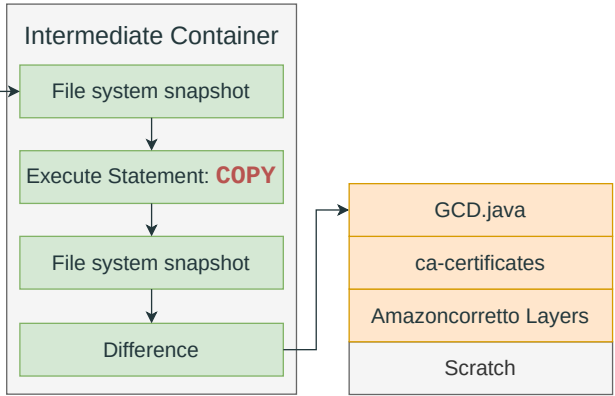
Docker image build process:

```
FROM amazoncorretto:17

RUN yum update -y && yum install -y
ca-certificates && yum clean all

COPY GCD.java /opt

CMD ["java", "/opt/GCD.java"]
```



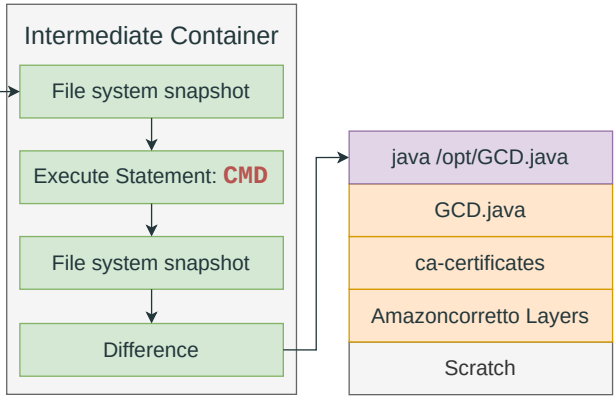
Docker image build process:

```
FROM amazoncorretto:17

RUN yum update -y && yum install -y
ca-certificates && yum clean all

COPY GCD.java /opt

CMD ["java", "/opt/GCD.java"]
```



Best practice: Chain shell commands to optimize file system usage.

- Instead of:

```
1 RUN yum update -y
2 RUN yum install -y ca-certificates
3 RUN yum clean all
```

- Use a single `RUN` statement:

```
1 RUN yum update -y && yum install -y ca-certificates && yum clean all
```

Benefit: Package manager cache is not included in the Docker image.

Best practice: Use Alpine[29]-based base-images.

- The majority of distribution images have critical CVEs.
- Alpine Linux is a security-oriented, lightweight distribution.
- Based on `musl` libc and `busybox` (be careful, no `glibc`).
- Alpine 3.14.9 has a size of 5.61MB (!)

Benefit: Reduces the attack surface for CVEs.

[29] `https://alpinelinux.org`

Best practice: Use *multi-stage* builds:

- The resulting Docker image results from the last build stage.
- Put any activity that is not required to run the service of the Docker image into a separate build stage, e.g. compile source code.

Benefit: Unnecessary files (e.g., build tools) are not included in the Docker image.

Example:

```
1  # STAGE 1
2  FROM python:3.11-alpine AS builder
3  LABEL maintainer=christoph.schober@th-deg.de
4
5  RUN apk --no-cache --update add build-base
6  RUN pip install numpy --user
7
8  # STAGE 2
9  FROM python:3.11-alpine
10 COPY --from=builder /root/.local /root
```

→ Reduces the size of the Docker image by factor ≈ 4.

Best practice: Do not use images from the Docker hub.

- The majority of Docker images suffer from a large amount of critical CVEs.
- Anyone can upload a Docker image to the Docker hub (insecure by design).
- What if a Docker hub account is compromised and an attacker manages to replace a popular Docker image with a malicious one?

→ If that is not possible, at least use *Docker Official Images* only.

Container-based Virtualization

9. Docker Compose

Docker Compose

Tool for defining and running multi-container Docker applications, allowing for easy orchestration and management of complex environments.

Benefits:

- Declarative configuration of multiple Docker containers in a YAML file.
- Better organization and readability of the configuration.
- Provides a simple way to start, stop, and restart all containers at once.
- Manages environment variables, volumes, and networking between containers.

Configuration file: `docker-compose.yml`

Basic structure:

```
1 version: ""      # either "2" or "3" ("2" recommended for single node systems)
2 services: {}     # Specification of the container stack
3 volumes: {}      # Specification of named volumes (optional)
4 networks: {}     # Specification of custom networks (optional)
```

See the reference documentation[30] for details.

[30] `https://docs.docker.com/compose/compose-file/compose-file-v2/`

A `docker-compose.yml` example creating a container running PostgreSQL:

```yaml
version: "2"
services:
  postgresql:
    image: postgres:14.2
    ports:
      - 5432:5432
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: secret
    volumes:
      - pgdata:/var/lib/postgresql/data
volumes:
  pgdata:
```

A `docker-compose.yml` building the `cat-of-the-day` example:

```
version: "2"
services:
  cat-of-the-day:
    image: cat-of-the-day:latest
    build: .
    ports:
      - 80:5000
```

See: `https://mygit.th-deg.de/aw-public/docker-examples/-/tree/master/compose-cat-of-the-day`

Useful commands:

- Create and start the multi-container application:

```
1 docker compose up -d
```

- Stop and remove the multi-container application:

```
1 docker compose down
```

- Watch the logs of a particular `services`:

```
1 docker compose logs SERVICE
```

- Execute a command inside a partiuclar `services`:

```
1 docker compose exec SERVICE COMMAND
```

Container-based Virtualization

10. Summary

Summary

You should have acquired the competencies to

- Explain the difference between a VM and a container.
- Know which kernel features enable container-based virtualization.
- Describe the components of the Docker platform.
- Manage Docker containers (create, start, stop, remove).
- Persist data generated by containers.
- Create a secure networking infrastructure for containers.
- Build Docker Images from Dockerfiles.
- Set up complex multi-container applications using Docker compose.