

PACKAGING & DOCKER

CS 346: Application
Development

DEPLOYMENT CHALLENGES

Our goal is **stable, consistent software** that we can deliver to our customers. Every version of the OS that we support, every different platform, every conceivable configuration should work as expected.

However, if our development / testing / deployment environments don't match exactly, our software might not run properly!

- You might be missing files e.g., a config file that you created manually on your development machine but forgot to include in the package.
- You may have incorrect versions of libraries e.g., different versions of DirectX installed.
- You may have a different version of the OS on your development and deployment systems.



“Things you should never say to a customer”.

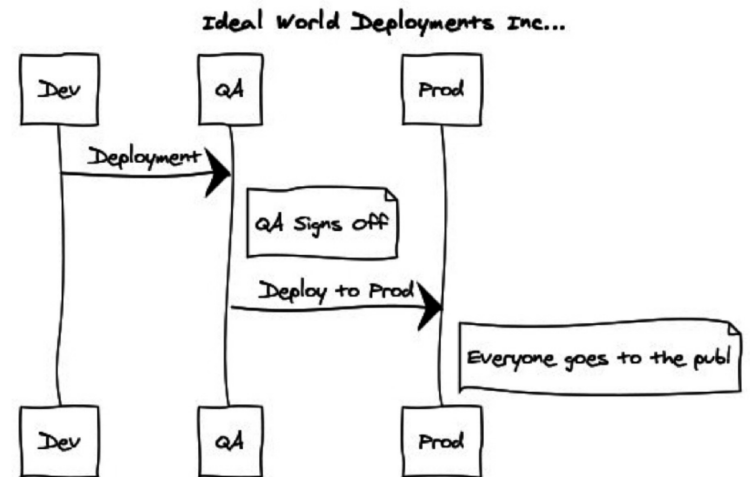
HOW CAN WE ACCOMPLISH THIS?

Multiple environments, that you keep in sync.

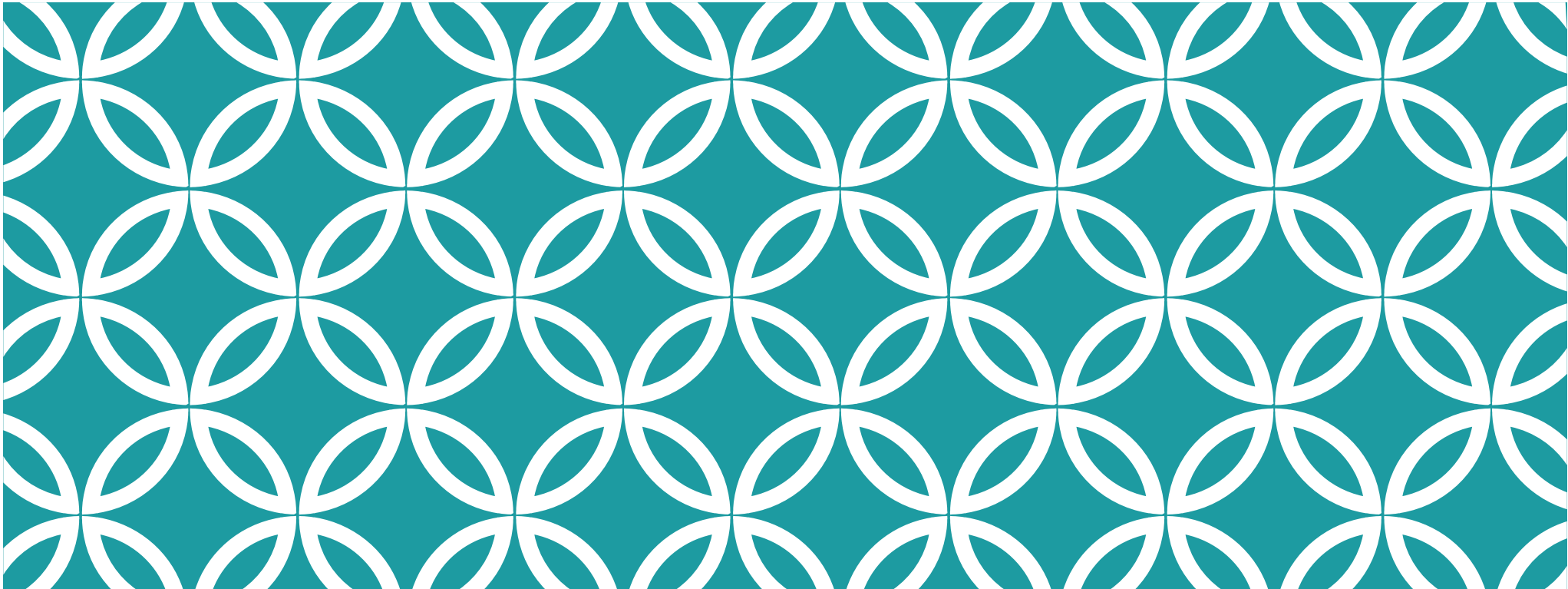
- **Development:** the machine that you use for builds.
- **Testing:** configurations representing each environment that you need to support (e.g. all supported versions of macOS, Windows and so on).
- **Deployment:** a system where you might deploy the software (live, running!)

How do you guarantee that each environment has the correct versions of every piece of software that your application uses or requires?

How do you handle this when you cannot control the deployment environment? e.g., an end-user's home computer?



<https://dzone.com/refcardz/deployment-automation-patterns>



INSTALLERS

CS 346 Application
Development

SOLUTION 1: INSTALLERS

One approach is to carefully control what is released to customers: create *installers* that handle the tasks for you.

An **installer** is a software application that is designed to install other software.

- You are probably familiar with these already! e.g., Setup installers on Windows.

What do they do?

- Create a folder where your application will exist e.g., c:\Program Files\MyApp
- Copy your executable program to that location, set permissions, register with the OS.
- Install system libraries, register with the OS if needed e.g., c:\Windows\System32
- Setup initial preferences including application icons.

These are very OS specific! Mac, Windows, Linux have different constraints.

GRADLE PACKAGING

Gradle will create installers for you.

1. Console applications
 - Tasks > Build > distZip
 - Creates a JAR file, and scripts to execute it (all in a ZIP file).
2. Desktop/JVM applications
 - Tasks > Compose Desktop > packageDistribution
 - Creates a Windows MSI, macOS DMG or Linux DEB.
3. Services are more complex...

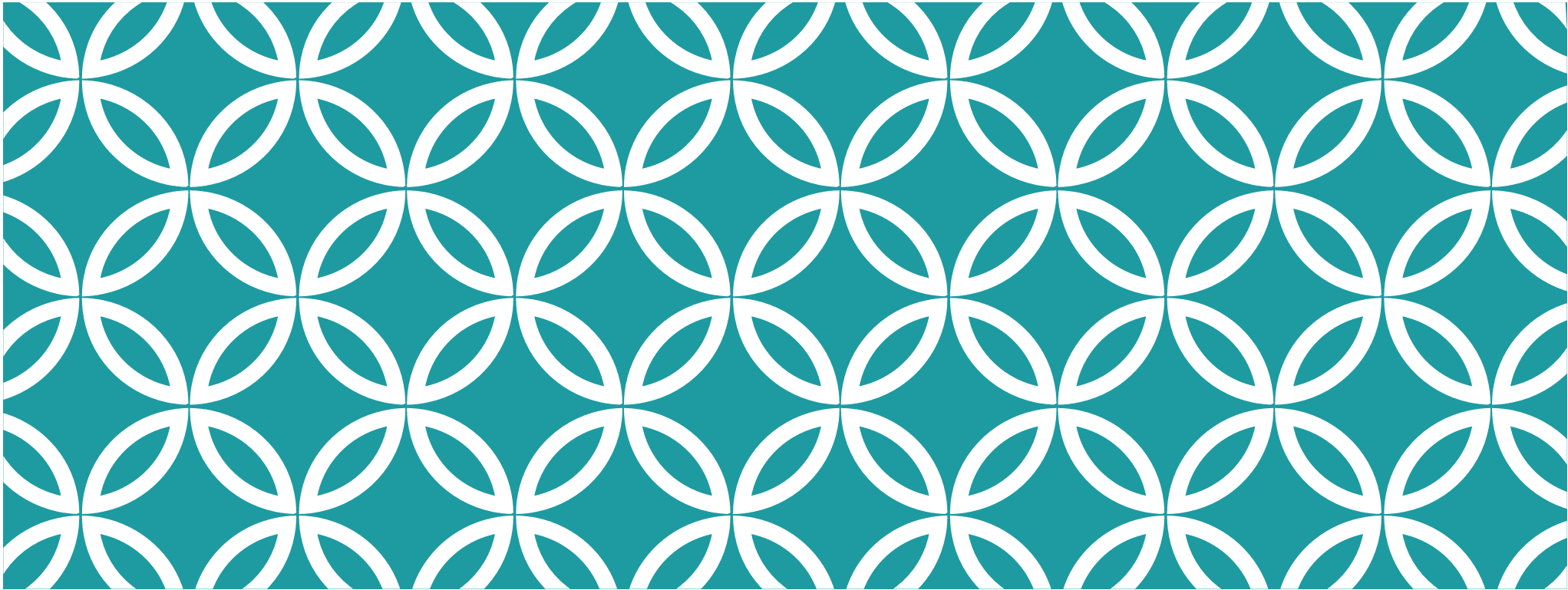
WHERE INSTALLERS FAIL

It's not uncommon for software to be deployed from an installer, and then have it fail. Often this happens because your operating conditions are different from the conditions under which you developed and tested.

Examples:

- You may have tested on a different version of the operating system than the user has, so your software may work differently on their system.
- Your application might rely on other software to be installed e.g., `sed` or a particular version of the bash shell that your installer cannot install.
- The runtime environment might need to be configured in a specific way for your application to run correctly e.g., environment variables holding private keys, like `AWS_SECRET='KASDJFTG_&JGJMHGF_!@GHHY@'`, or specific network configurations.

How do we fix this? *We specify and control the deployment environment.*



VIRTUALIZATION

CS 346 Application
Development

SOLUTION 2: VIRTUALIZATION

Virtualization uses software to create an **abstraction layer** over computer hardware, enabling the division of a single computer's hardware components—such as processors, memory and storage—into multiple virtual machines (VMs).

Each VM runs its own operating system (OS) and behaves like an independent computer, even though it is running on just a portion of the actual underlying computer hardware. – [IBM \(2024\)](#)

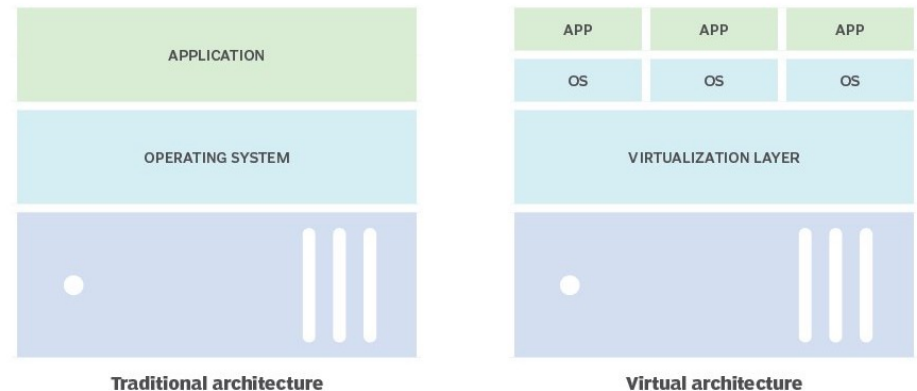


Diagram from [TechTarget \(2024\)](#)

VIRTUALIZATION

Benefits?

- Resource efficiency: Physical hardware can be shared across multiple operating environments.
- Easier management: Virtual “Machines” can be started up as needed (backed up, moved).
- Control: We can use this to specify the runtime environment!

Downsides?

- Fairly heavyweight. We’re hosting an OS specifically for our application.

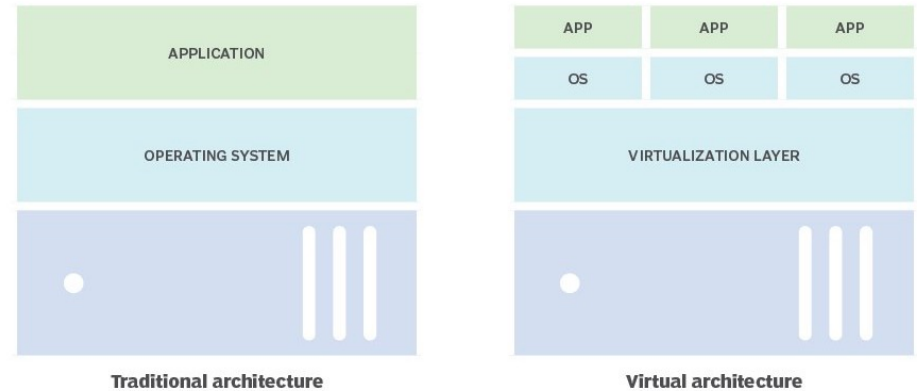


Diagram from [TechTarget \(2024\)](#)

CONTAINERIZATION

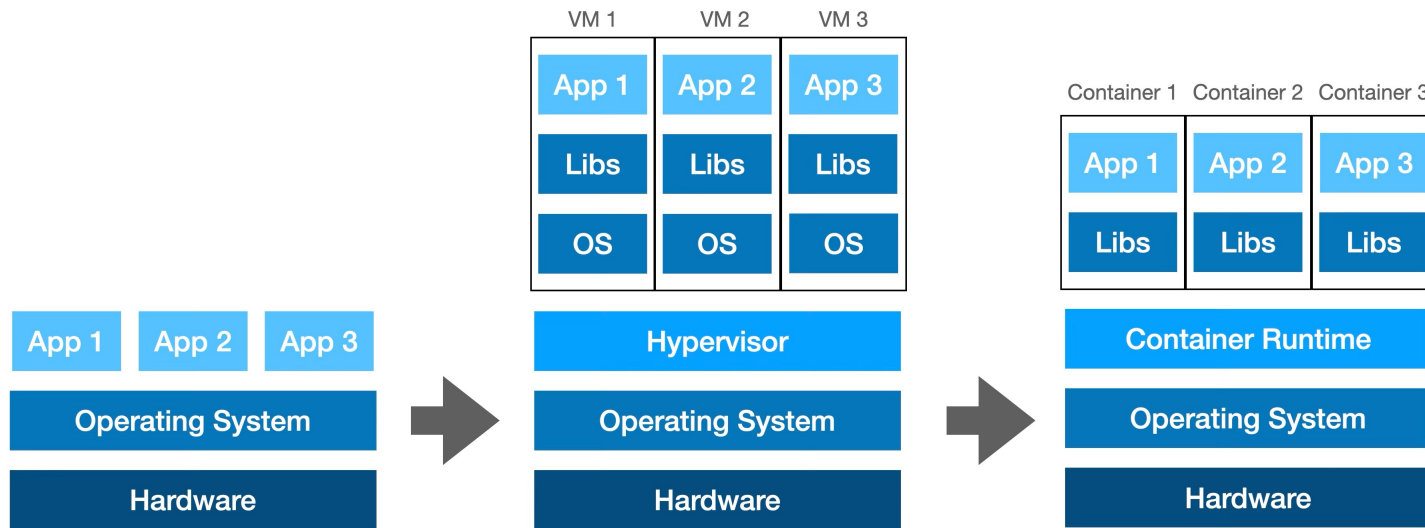


Fig 1. Standalone Hardware

Fig 2. Virtualization

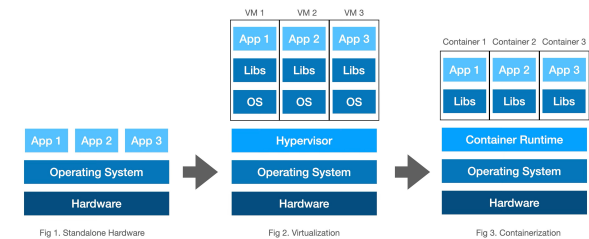
Fig 3. Containerization

One OS that hosts everything

Dedicated virtual machines

Lightweight containers?

CONTAINERIZATION



Standalone: Software is installed directly into the host operating system.

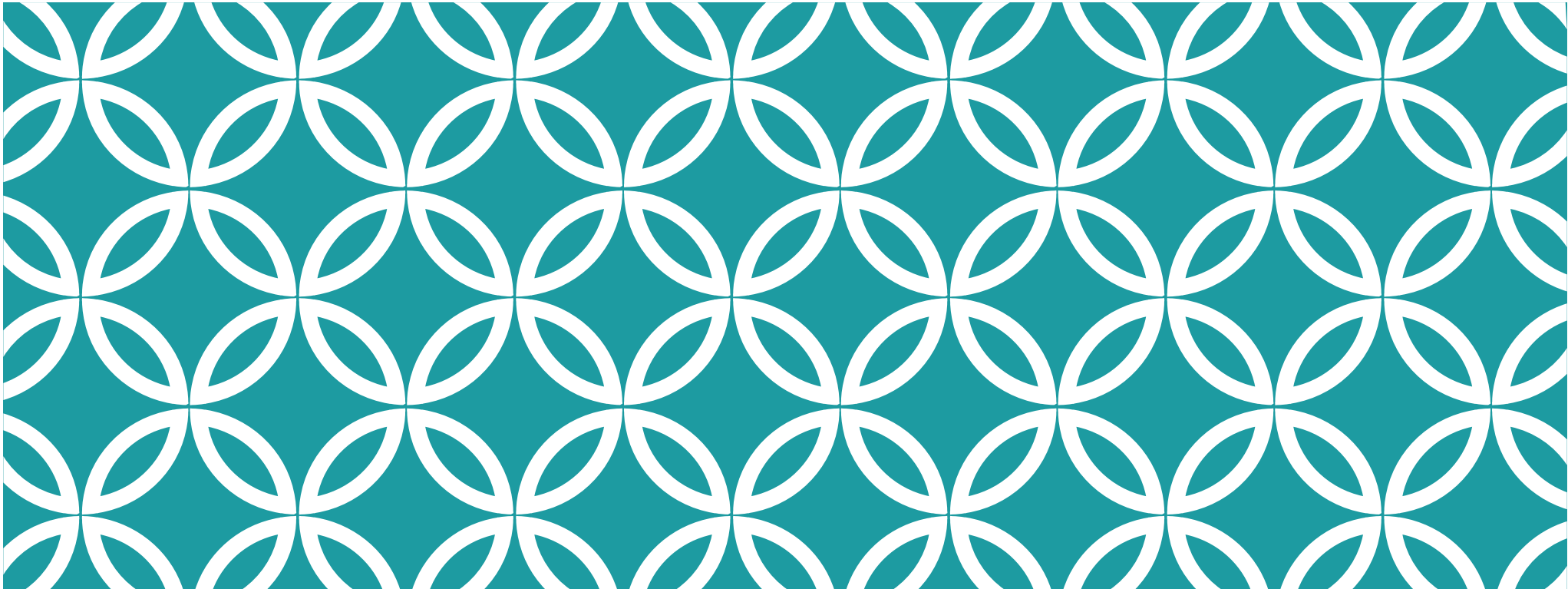
- The OS must allocate and manage resources for each application.
- Security concerns with a shared environment.

Virtualization: Multiple virtual machines, each one is an abstraction of a physical machine.

- Each virtual machine is running a complete OS, and allocated its own memory, CPU cycles etc.
- Can dictate how physical resources are shared across VMs e.g., 128 GB of RAM can be split.
- Provides isolation of each application into its own OS instance for improved security.

Container: An isolated environment for running an application.

- Containers run on the same underlying host OS; lightweight compared to virtual machines.
- The host OS schedules CPU, resources to the containers not VMs. Containers are just processes.
- Smaller, easy to start/stop; can be deployed on any physical and virtual machines, or even to the cloud.



DOCKER

CS 346 Application
Development

SOLUTION 3: CONTAINERIZATION (DOCKER)

Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications.

<https://docs.docker.com/get-started>

Docker is a **containerization** platform.

- Create images that bundle your application and its environment together.
- Provides an online hub where you can distribute these images to other people.
- Provides the runtime engine to execute images.

Docker is NOT meant for end-users!

- It's for people like us that need an efficient and consistent way to install servers/services.

INSTALLATION



Docker Desktop for Mac

A native application using the macOS sandbox security model which delivers all Docker tools to your Mac.



Docker Desktop for Windows

A native Windows application which delivers all Docker tools to your Windows computer.



Docker Desktop for Linux

A native Linux application which delivers all Docker tools to your Linux computer.

<https://docs.docker.com/get-started/get-docker/>

Installation is simple. You can install Docker from installers on their website or your favorite package manager.

- e.g., `brew install docker` on macOS.

```
$ docker version
```

```
Client: Docker Engine - Community
```

```
Version:          27.3.1  
API version:      1.47  
Go version:       go1.23.1  
Git commit:       ce1223035a  
Built:            Fri Sep 20 11:01:47 2024  
OS/Arch:          darwin/arm64  
Context:          desktop-linux
```

```
Server: Docker Desktop 4.34.3 (170107)
```

```
Engine:
```

```
Version:          27.2.0  
API version:      1.47 (minimum version 1.24)  
Go version:       go1.21.13  
Git commit:       3ab5c7d  
Built:            Tue Aug 27 14:15:41 2024  
OS/Arch:          linux/arm64  
Experimental:    false
```

```
containerd:
```

```
Version:          1.7.20  
GitCommit:        8fc6bcff51318944179630522a095cc9dbf9f353
```

```
runc:
```

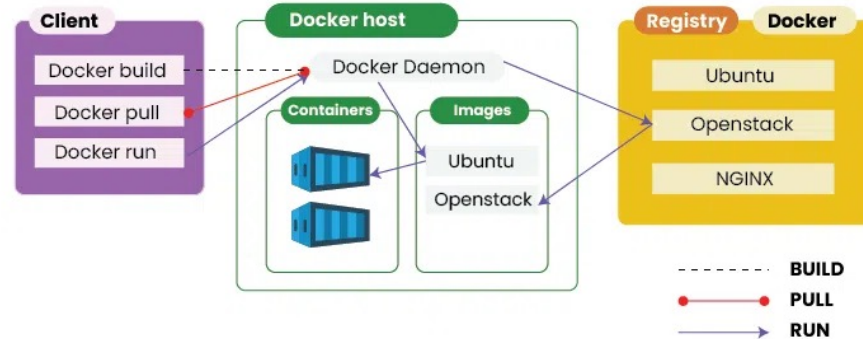
```
Version:          1.1.13  
GitCommit:        v1.1.13-0-g58aa920
```

```
docker-init:
```

```
Version:          0.19.0  
GitCommit:        de40ad0
```


ARCHITECTURE

Docker Architecture



<https://www.geeksforgeeks.org>

- The **Client** is the system on which you are running Docker commands i.e., host.
- The **Docker Host** is a background process that manages Docker runtime behavior.
 - An **Image** is a snapshot of your environment + application at a point in time.
 - A **Container** is a running instance of your Image.
- The **Registry** is an online repository where you can store images for others to use.

WORKFLOW

Step 1: Write a program

- Create a program that can be executed. For us, this will typically be a JAR file that we can run using ``java -jar filename.jar``.

Step 2: Write a Dockerfile

- Create a configuration file that describes your environment.

Step 3: Create a Docker Image

- Create an image which contains your environment (including dependencies) and executable at a point-in-time.

Step 4: Run your Docker Image

STEP 1: PROGRAM

Write a complex and useful application (Hello.kt in this example).

```
fun main() {  
    println("Hello Docker!")  
}
```

Compile it to a JAR file, and copy the JAR file to a new/empty directory.

```
$ kotlinc Hello.kt -include-runtime -d Hello.jar
```

```
$ java -jar Hello.jar
```

```
Hello Docker!
```

```
$ mkdir docker
```

```
$ cp Hello.jar docker/
```

STEP 2: DOCKERFILE

Create a file named `Dockerfile` in the same directory as your JAR file.

```
# start with this image, it includes a Linux kernel and Java JDK 17
FROM openjdk:17

# import your Hello.jar file, and host in the app subdir.
# at runtime, your filesystem will expose under /app subdir
COPY Hello.jar /app

# set /app as your working directory and `cd` to it
WORKDIR /app

# run the application
CMD java -jar Hello.jar
```

STEP 3: CREATE AN IMAGE

Build an image in this directory (which uses the Dockerfile)

```
$ cd docker  
$ docker build -t hello-docker .
```

- -t tells Docker to tag it with a version (defaults to latest).
- hello-docker is the name that will be assigned to our image.
- . indicates that it should include the current directory's contents in the image.

STEP 4: RUN IT

Check that it was created

```
$ docker images
$ REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
hello-docker   latest   a615e715b56d  7 second ago  455MB
```

Run it!

```
$ docker run hello-docker
Hello Docker!
```

STEP 5: PUBLISH IT (OPTIONAL)

1. Create an account on [Docker Hub](#) if you haven't already. Login.
2. Create a repository to hold your images.
3. Tag your local image with your username/repository.
4. Push your local image to that repository.

```
$ docker image ls
REPOSITORY      TAG          IMAGE ID      CREATED        SIZE
hello-docker    latest      f81c65fd07d3 3 minutes ago 455MB
```

```
$ docker tag f81c65fd07d3 jfavery/cs346
```

```
$ docker push jfavery/cs346:latest
```

WHEN TO USE THIS?

Docker containers are extremely common when publishing web services to the cloud! Publish a container and have AWS/Firebase/some service host and run the container.

One major consideration when doing this is that you need to map a port number (since you want network traffic directed to the host machine to be redirected to the running container. e.g., below.

```
# Dockerfile
FROM openjdk:17
VOLUME /tmp
EXPOSE 8080
ARG JAR_FILE=target/service-docker.jar
ADD ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```