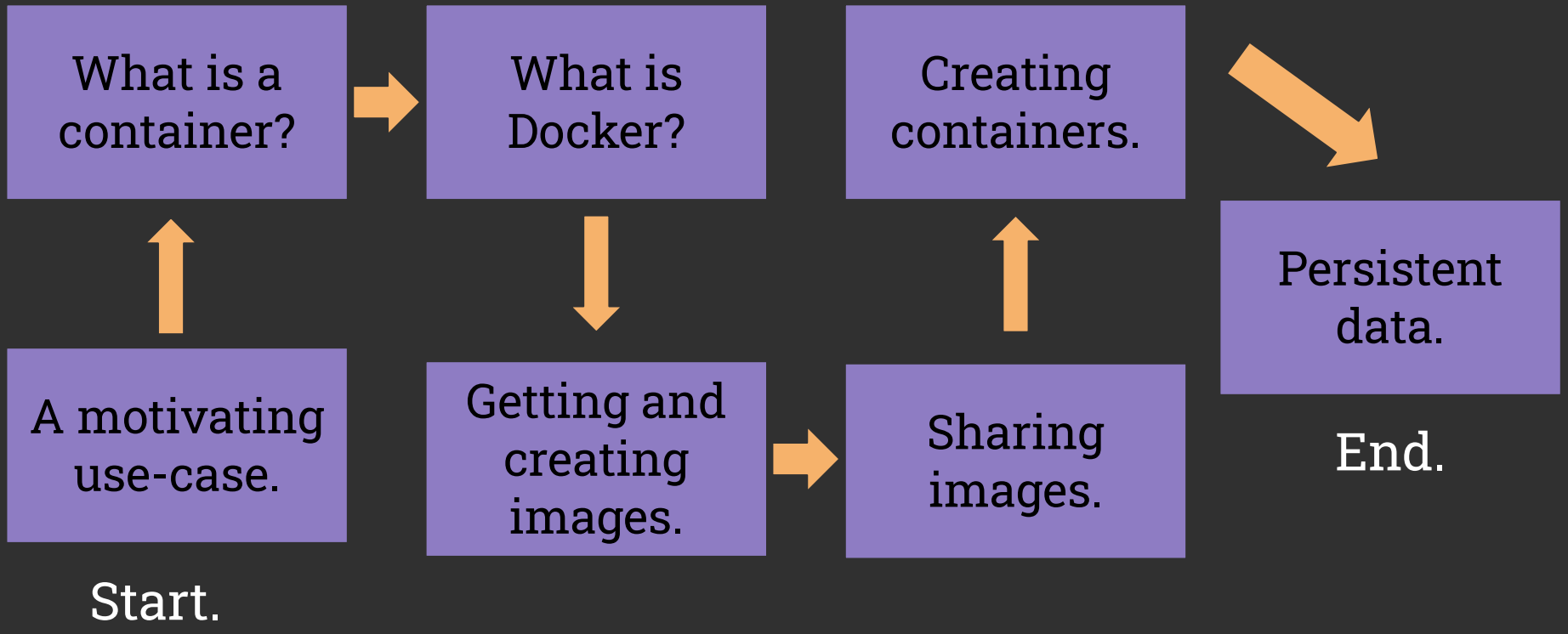
Break Free from Dependency Hell with Docker!

Containerizing Your Apps for Less Headaches.

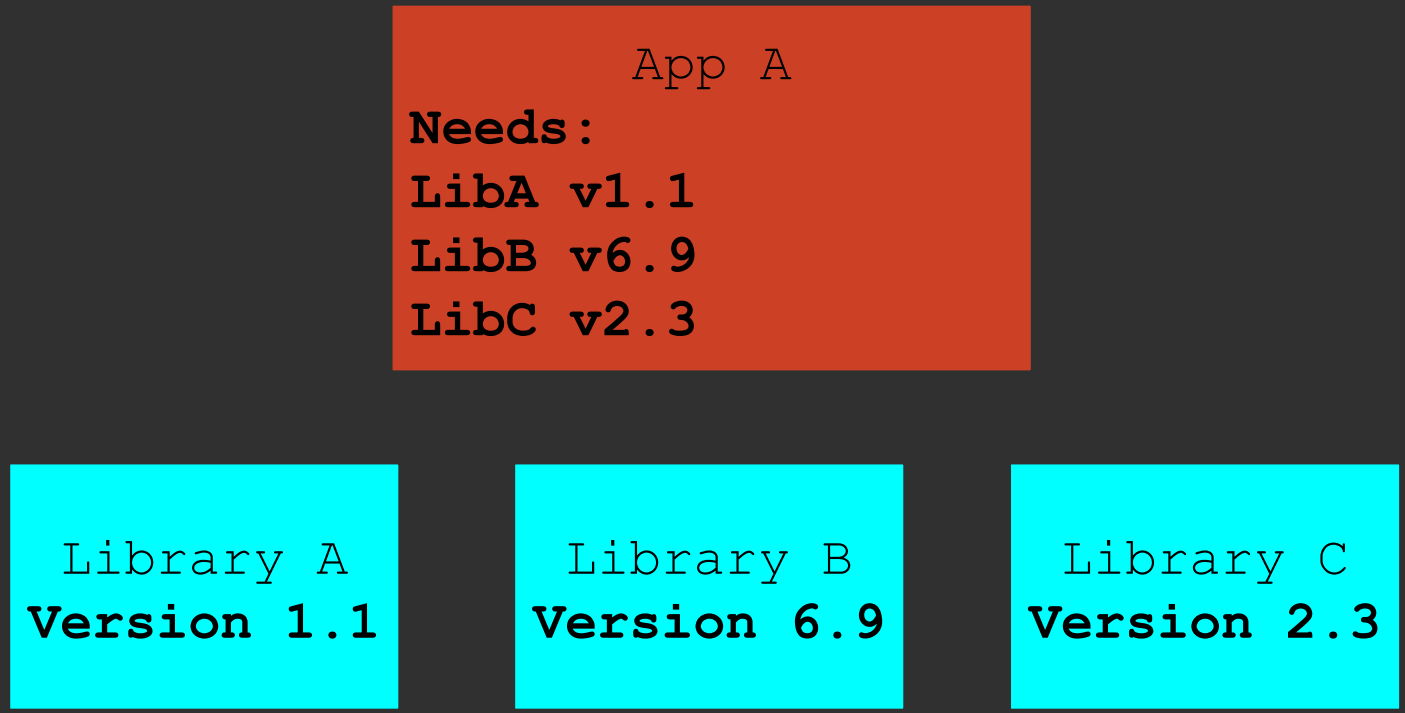
James Hurd

jameshurd@ku.edu

Roadmap


What is a container?
What is Docker?
Creating containers.
Persistent data.
End.
A motivating use-case.
Getting and creating images.
Sharing images.
Start.

Has this ever been you?



Perfect, all dependencies met. . .

App A

Needs:

LibA v1.1

LibB v6.9

LibC v2.3

App B

Needs:

LibA v1.2

LibB v6.9

LibC v2.3

Library A

Version 1.1

Library B

Version 6.9

Library C

Version 2.3

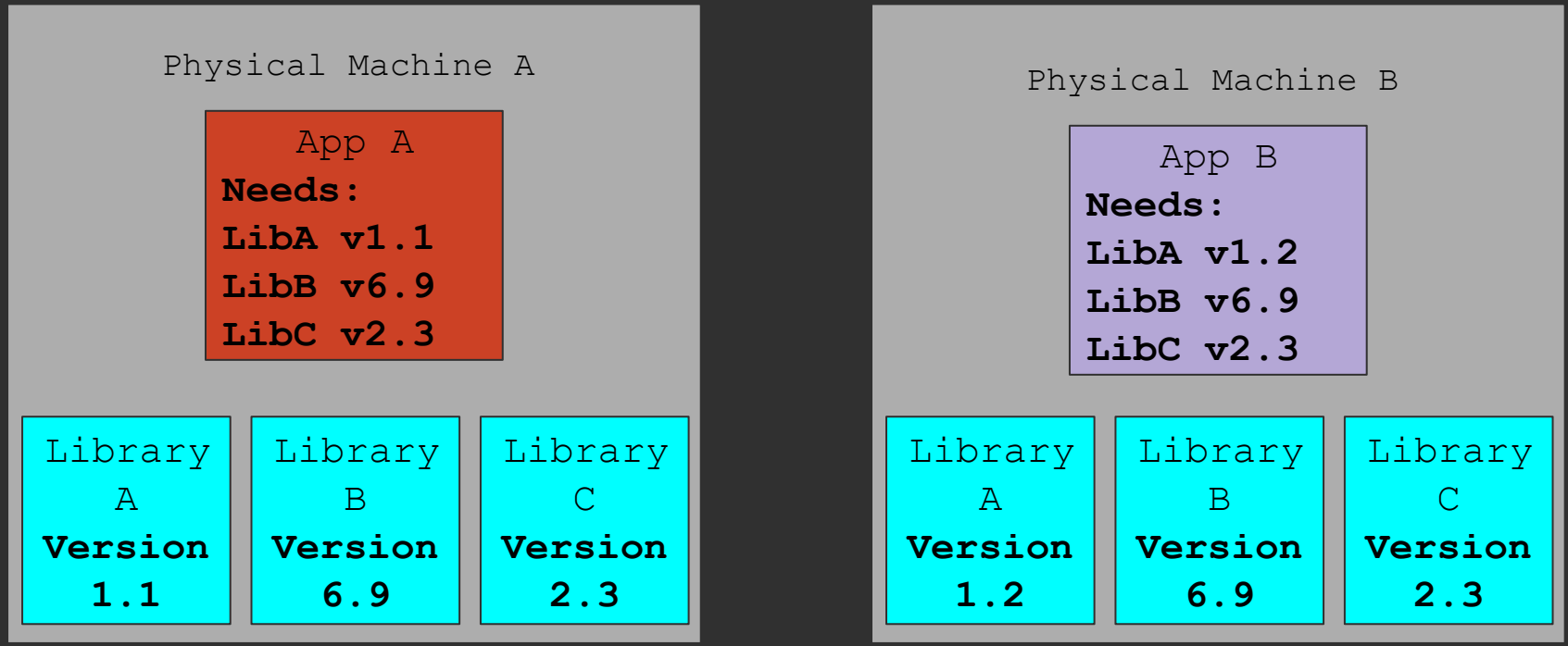
Noooo!!!!!!!!!!!!!!!!!! Can’t install App B!

App A and App B are **incompatible**.

Welcome to dependency hell.

How do we get out?

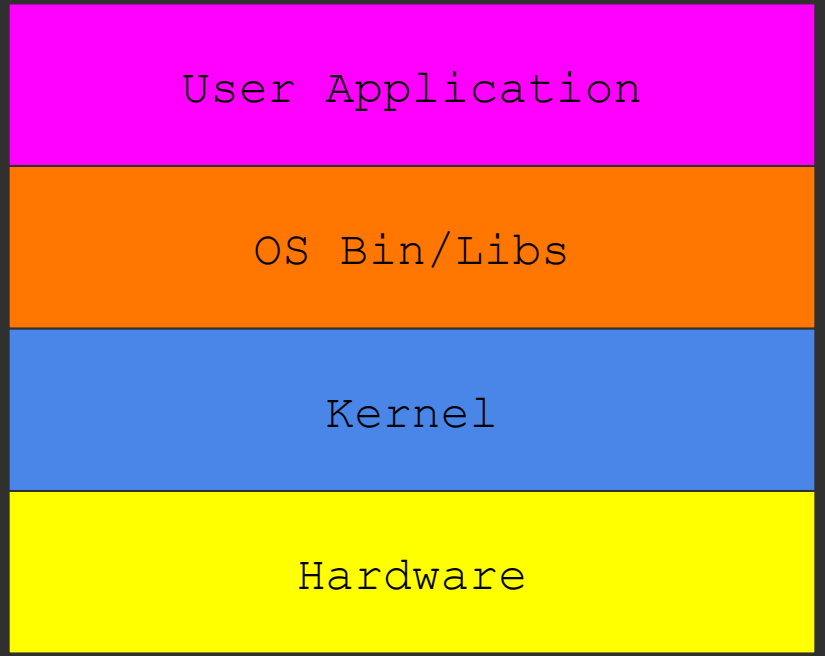
Option 1: Use Different Physical Machines

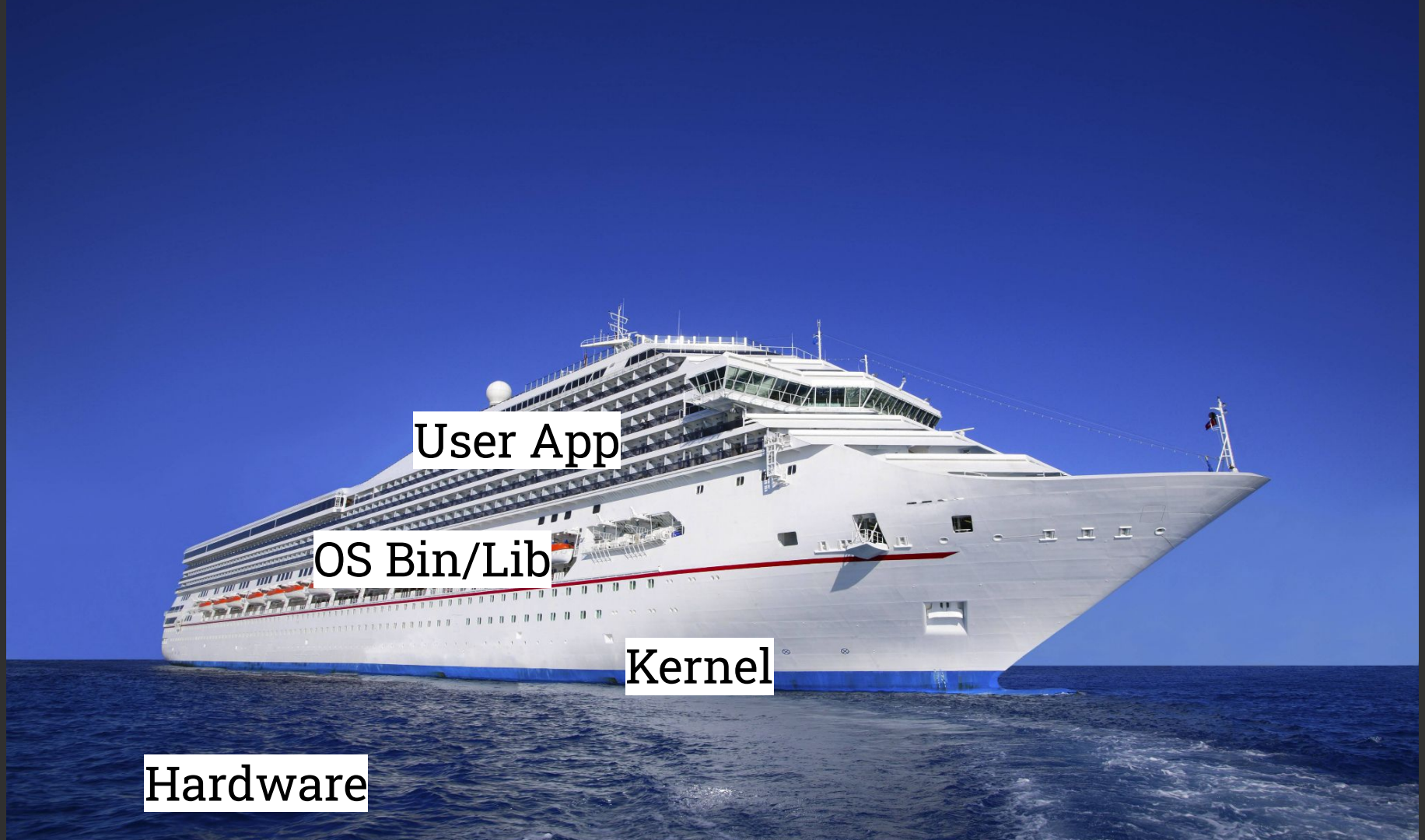


Option 1: Use Different Physical Machines

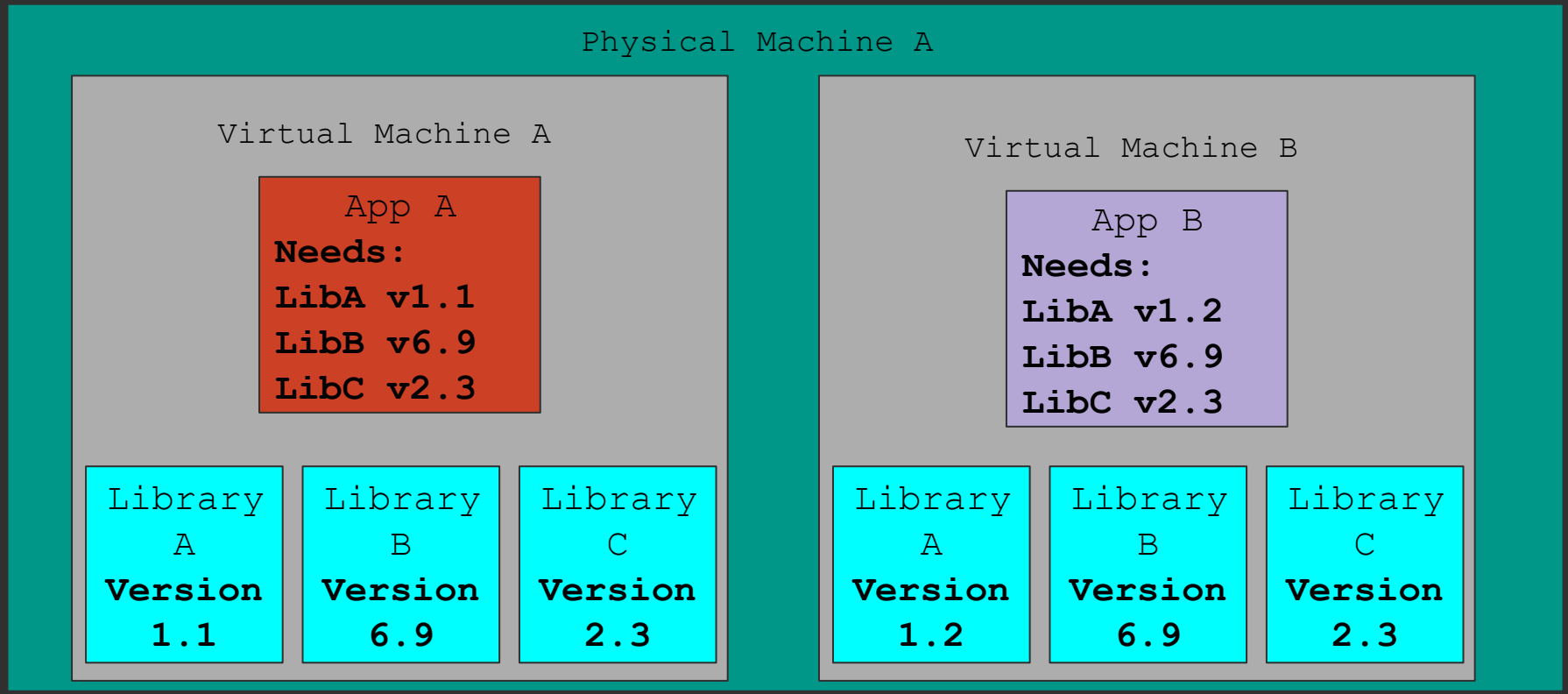
Sure, this technically works but...

- Not scalable.
- Too expensive.
- Takes up too much space.
- Uses a lot of energy.
- Just generally overkill.




User App
OS Bin/Lib
Kernel
Hardware

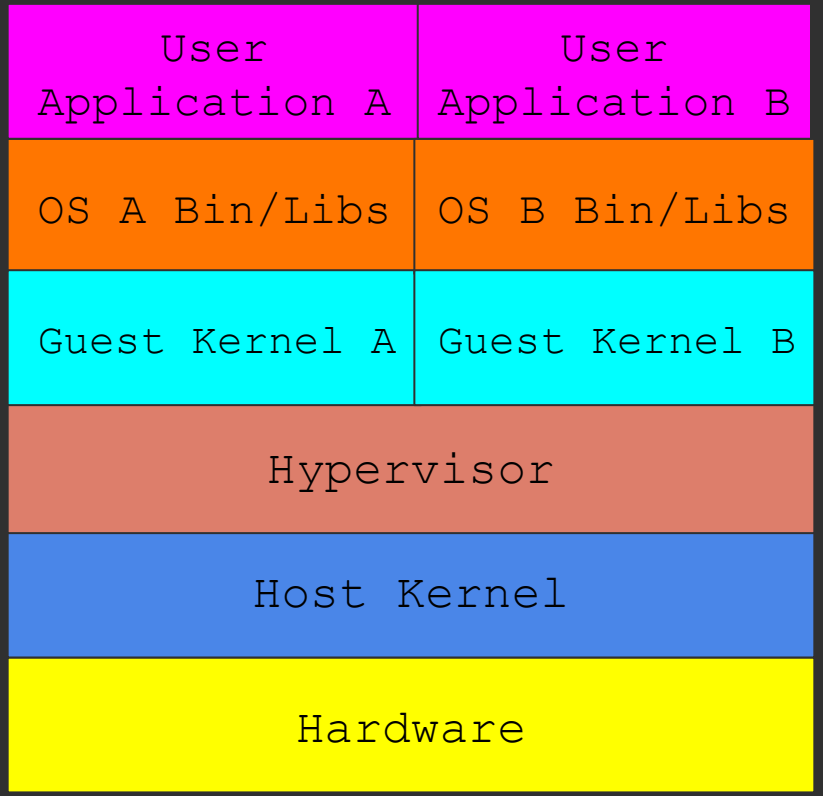
Option 2: Use Virtual Machines

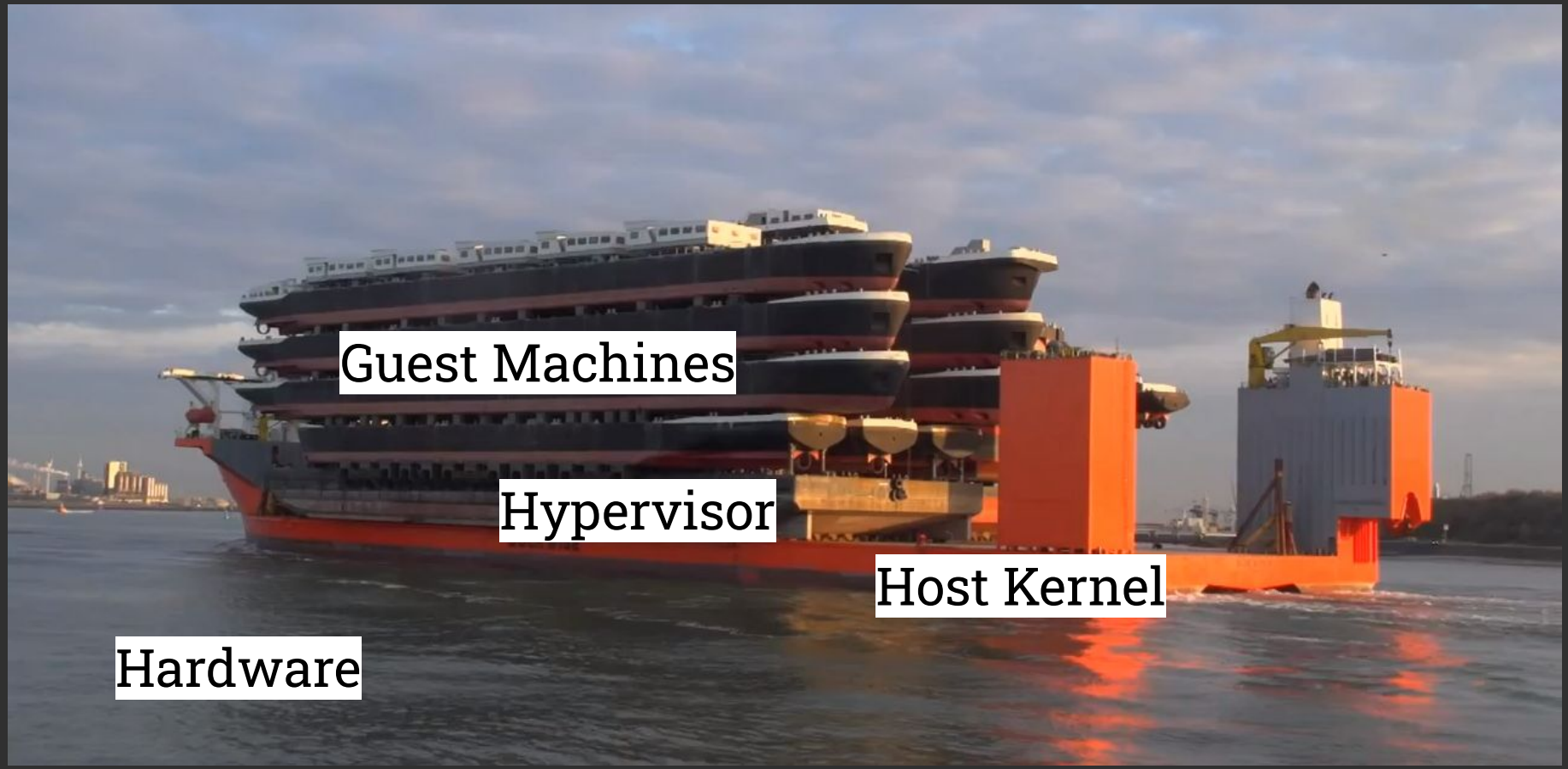


Virtualized System Environment

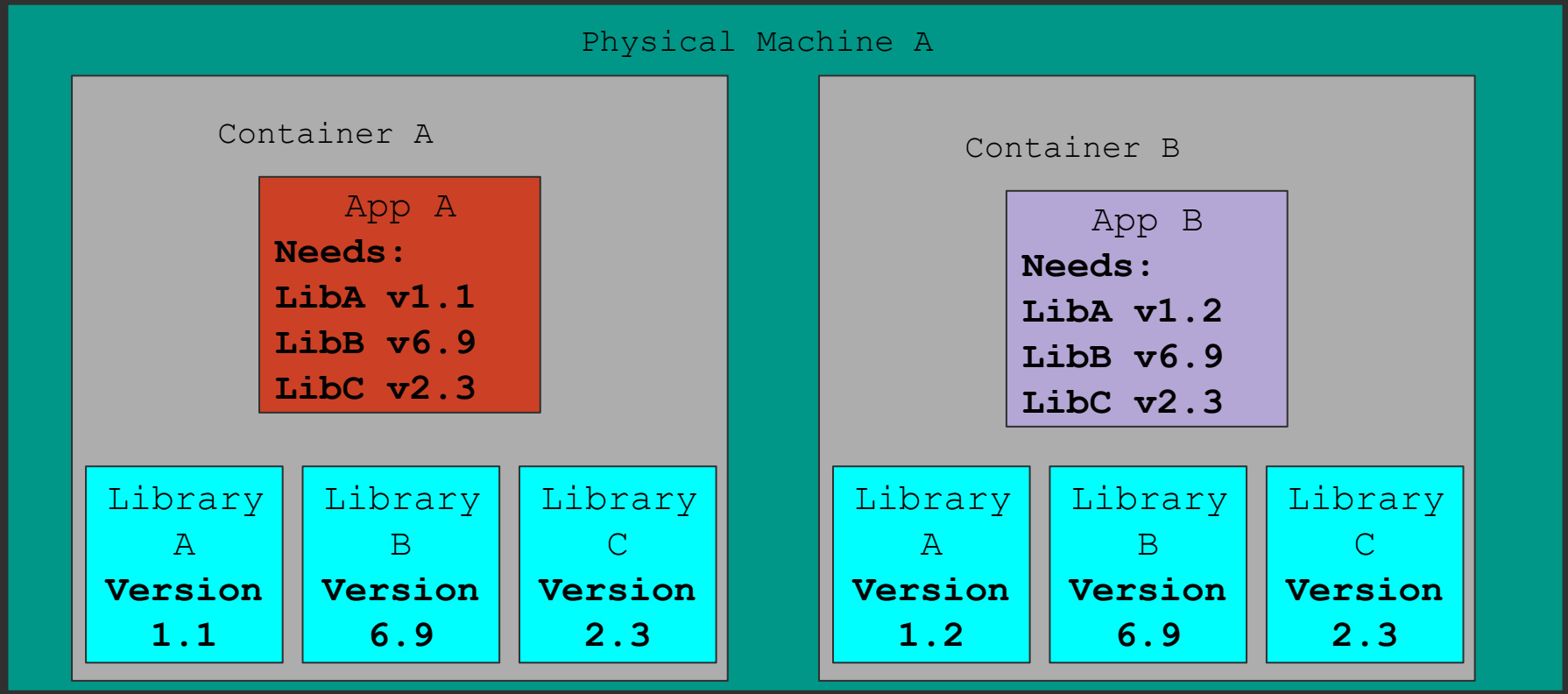
Better! However...

- Still not super scalable.
- Large overhead in compute-heavy apps.
  - e.g. Things using GPUs.
- Can be annoying to manage.
- Portability across hypervisors can be a problem.
- Resource intensive.




Guest Machines
Hypervisor
Host Kernel
Hardware

Option 3: Use Containers

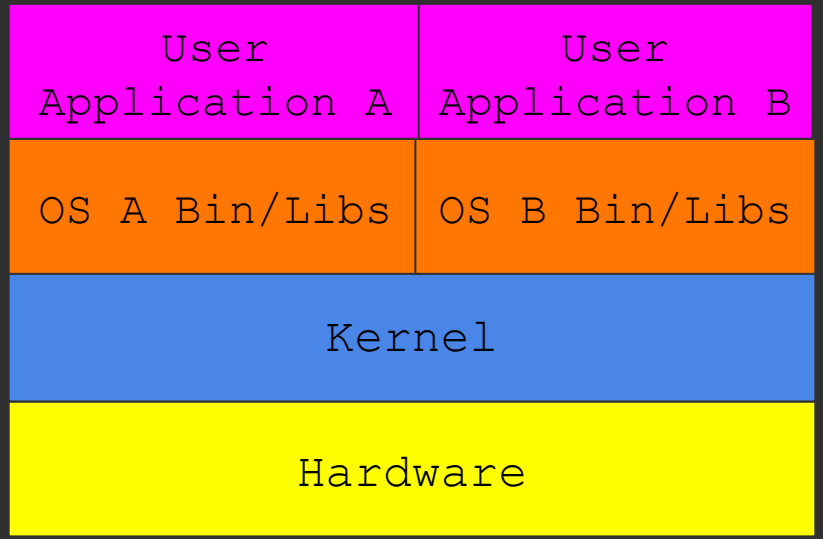


Containerized System Environment

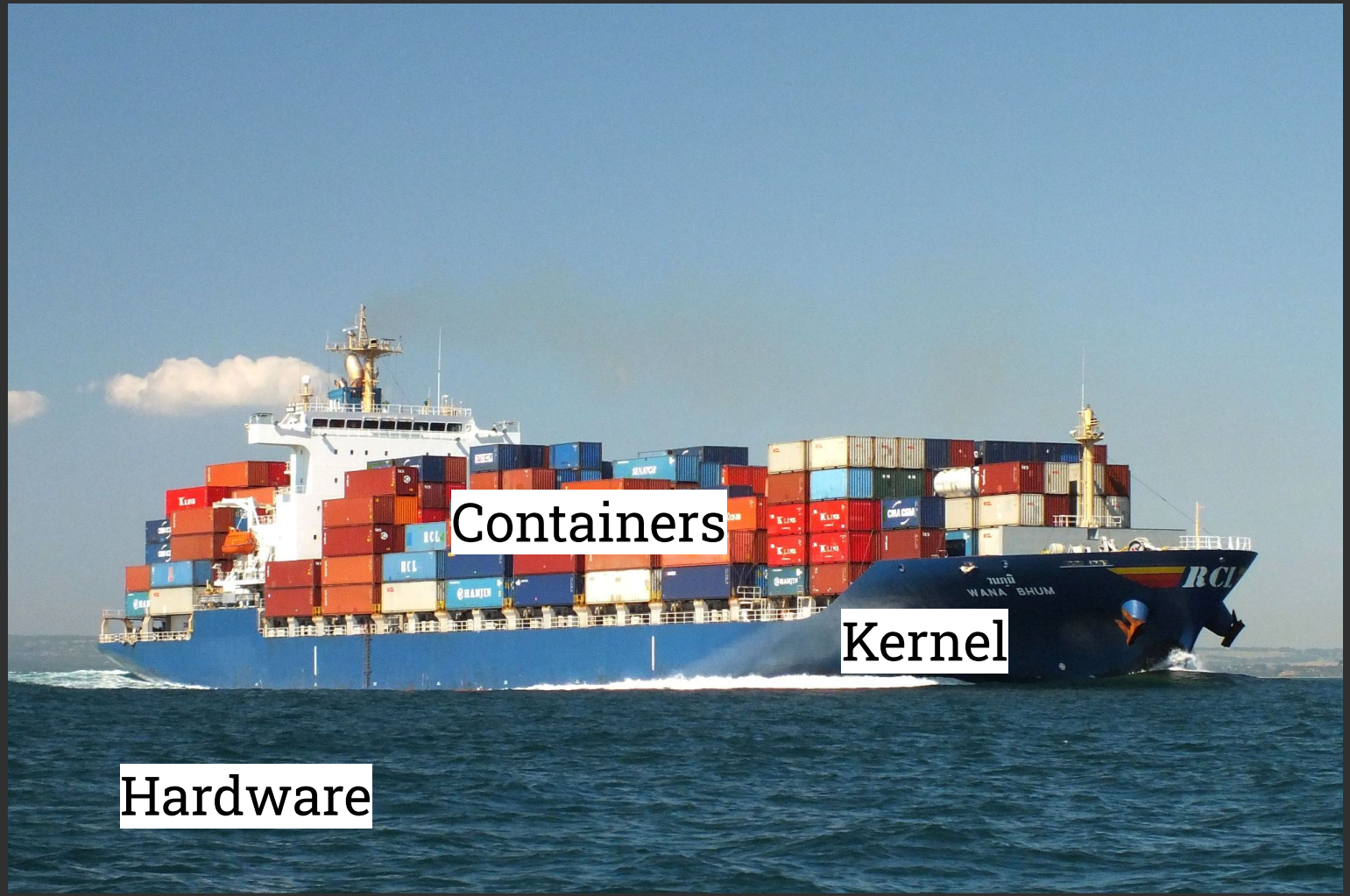
Much better!

- Isolated environments.
- Negligible overhead.
- Lightweight.
- Scalable.
- Very portable!
  - All you need is a container runtime.

*

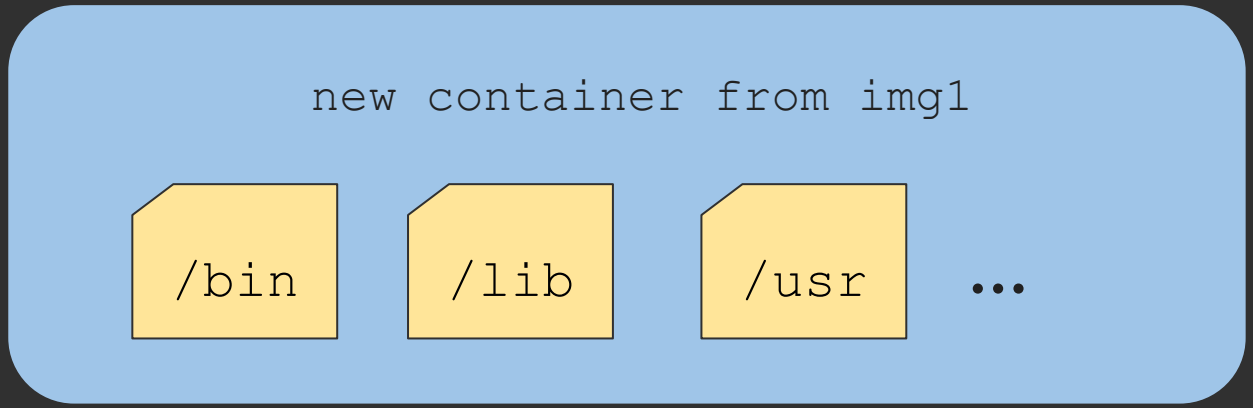


*A bit of a lie in some cases since some container runtimes talk through a daemon to do certain things like networking.


Containers
Kernel
Hardware

A Look Inside the Container

- Assuming your image is something UNIX-y.
- Any processes that run in this container will see this as its root.
  - Essentially fancy `chroot`.
- Processes running in here cannot access/see anything outside.
  - Possible because of kernel magic with `namespaces` and `cgroups`.



Containers are **NOT** virtual machines.

They are simply isolated processes.

Containers separate **application** from **infrastructure**.

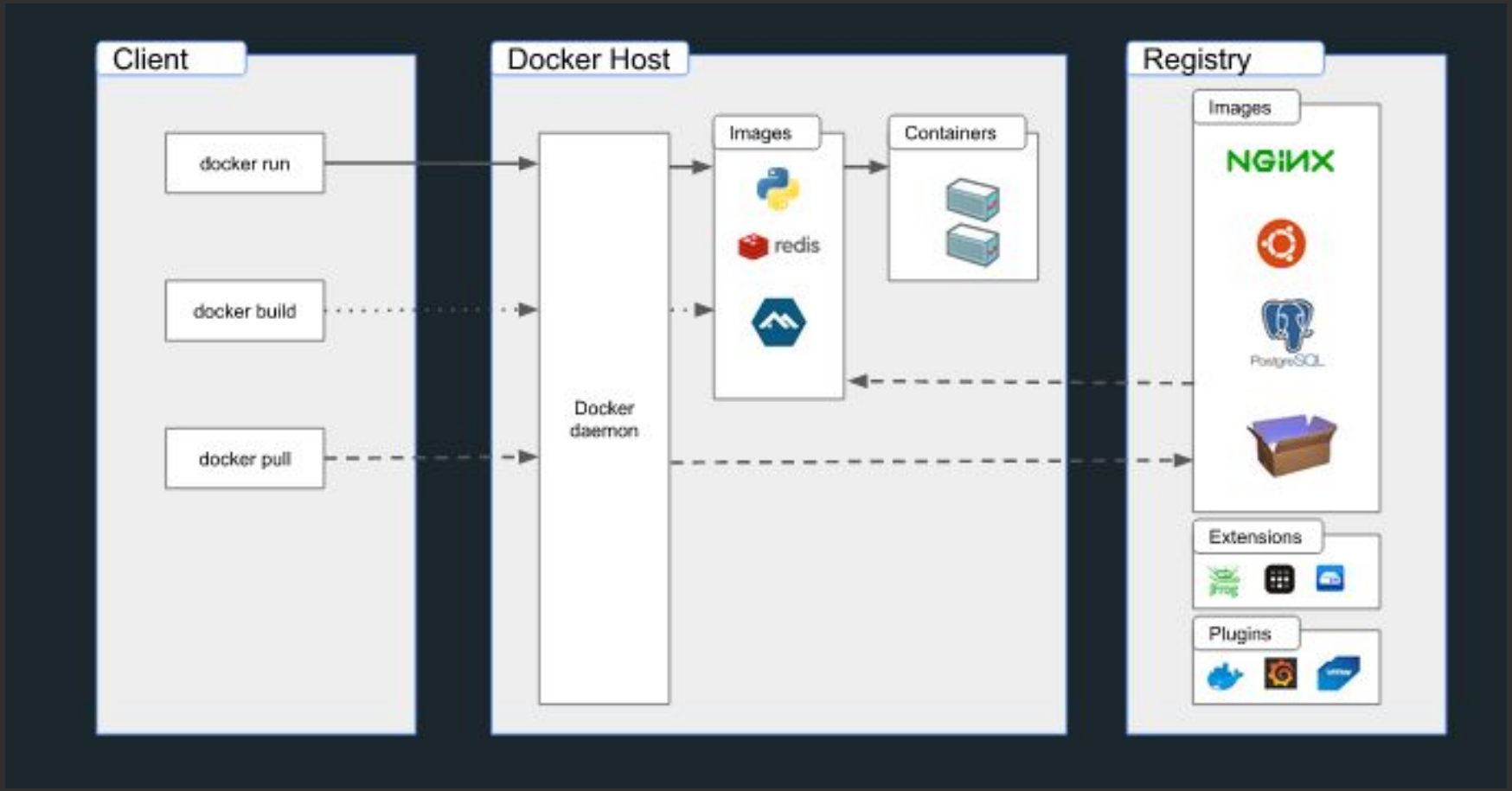

IT WORKS ON MY MACHINE
THEN WE'LL SHIP YOUR MACHINE
AND THAT IS HOW DOCKER WAS BORN

What is *Exactly* Docker?

- Docker is a container runtime.
- Docker daemon (`dockerd`) manages:
  - Storing Images.
  - Building Images.
  - Running Containers.
  - Networking (in some cases)
- Communicate with daemon through client (`docker`).
- Will need to download, link on `README`.

What is a Docker Image?

- Read-only
- Tells Docker how to create and spin up your container.
- The environment your program will be run in.
- Analogous to a VM snapshot or OS ISO.
- Create your own or pull from a registry.


Client
docker run
docker build
docker pull
Docker Host
Docker daemon
Images
redis
Containers
Registry
Images
NGINX
PostgreSQL
Extensions
Plugins

Notes on Linux and Windows Compatibility

Recall that containers share the host kernel.
Consequences for Docker:

- Windows images will not run on Linux
- Docker on Windows spawns a Linux VM in the background to run Linux images.

Still with me?

Questions?

Now let's see how to actually do stuff.

Getting Prebuilt Images

- dockerhub – registry of thousands of pre-created images.
  - Images made by users and official sources.
  - `hub.docker.com`
- Useful for getting “base” images.
- `docker pull img` – Gets `img` from dockerhub.
  - e.g. `docker pull ubuntu`
  - Defaults to latest release, so just pull to update.
  - Specify release `tag` with `docker pull img:tag`
- `docker images` – See images you have.
- `docker image prune` – Remove old/unused images.

Building Your Own Docker Image

- `Dockerfile`
  - Contains instructions on how to make an image.
  - Goes in the root of the app you want to containerize.
  - Top line: `# syntax=docker/dockerfile:1`
- `docker build -t name .`
  - To build app from `Dockerfile` in same directory.
  - Image identified by tag `name:latest`
- `docker tag old new`
  - Retag image with tag `old` to tag `new`.

Essential `Dockerfile` Commands

- `FROM img` – Specify base image.
- `WORKDIR dir` – Working directory in container for commands.
- `COPY src dest` – Copy files on the host at `src` to `dest` in image.
- `RUN command` – Run `command` inside the image.
  - Creates new layer in image.
- `CMD ["cmd", "arg1", "arg2", ...]` – Run `cmd` with `arg1`, `arg2`, ...
  - Docker will run last `CMD` in Dockerfile, used to launch app.
- `EXPOSE port` – Expose TCP port number `port` to the host.

```
example_app/
├── app.py
├── instance
└── templates
    └── index.html
```

```
$ tree example_app
```

```
# syntax=docker/dockerfile:1
FROM python
WORKDIR /app
COPY . .
RUN pip3 install flask
CMD ["flask", "run","--host=0.0.0.0"]
EXPOSE 5000
```

Dockerfile

Containerizing a simple Flask app:

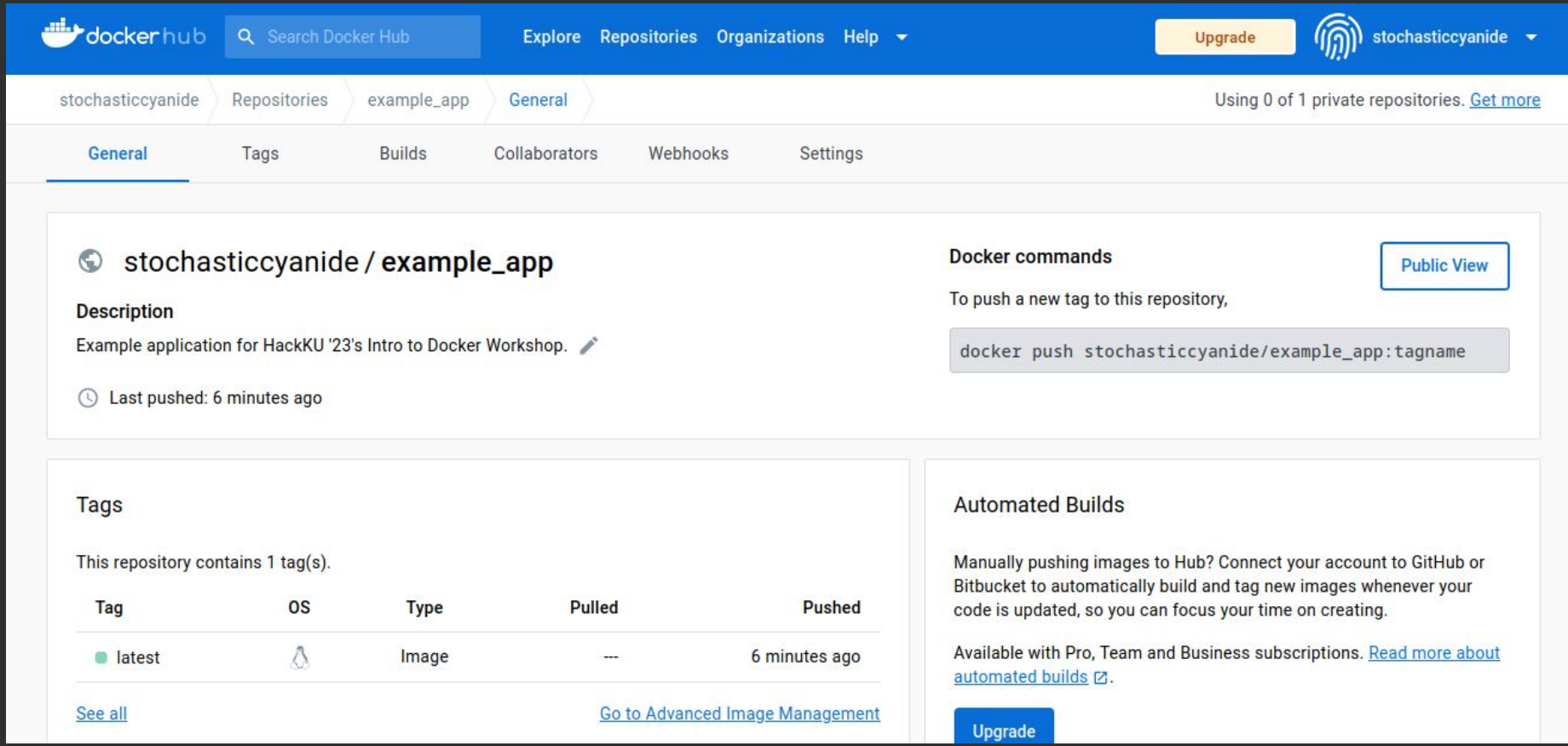
- Uses a `sqlite3` database.
- Hosts on port 5000.

```
$ docker build -t example_app .
```

Sharing your images.

- Create an account on dockerhub and create a new repository.
- `docker login -u your_username` – Log into dockerhub from client.
- Make sure the tag on your local image == name of the repository.
- `docker push tag`

```
[james@deck ex]$ docker tag example_app stochasticcyanide/example_app
[james@deck ex]$ docker push stochasticcyanide/example_app:latest
```


docker hub
Search Docker Hub
Explore
Repositories
Organizations
Help
Upgrade
stochasticcyanide
stochasticcyanide
Repositories
example_app
General
Using 0 of 1 private repositories. Get more
General
Tags
Builds
Collaborators
Webhooks
Settings
stochasticcyanide / example_app
Description
Example application for HackKU '23's Intro to Docker Workshop.
Last pushed: 6 minutes ago
Docker commands
Public View
To push a new tag to this repository,
docker push stochasticcyanide/example_app:tagname
Tags
This repository contains 1 tag(s).
Tag
OS
Type
Pulled
Pushed
latest
Image
---
6 minutes ago
See all
Go to Advanced Image Management
Automated Builds
Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.
Available with Pro, Team and Business subscriptions. Read more about automated builds.
Upgrade

Okay, how do I use these images?

What Is Docker Doing When I Run a Container?

```
$ docker run img1
```

dockerd

img1

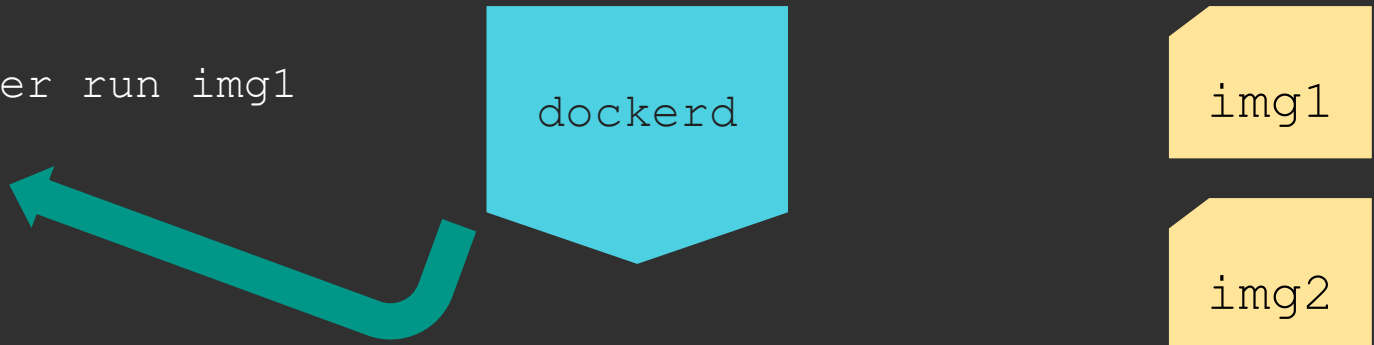
img2

img3

"Hey, can we run a container from `img1`?"

What Is Docker Doing When I Run a Container?

```
$ docker run img1
```



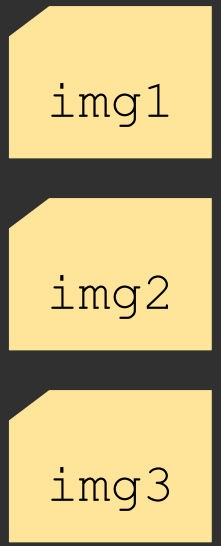
"Yeah sure, let me build it."

What Is Docker Doing When I Run a Container?

```
$ docker run img1
```

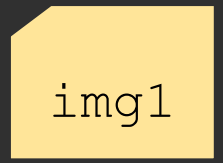
What Is Docker Doing When I Run a Container?

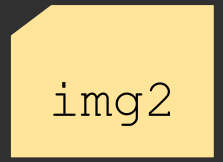
```
$ docker run img1
```

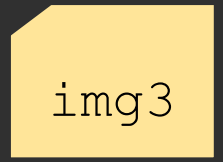


What Is Docker Doing When I Run a Container?

```
$ docker run img1
```







Important takeaway:
Docker builds a **new** container from image **each** run.

"Off you go child!"

```
new container from img1

Running CMD from Dockerfile
```

Running Your Container

- `docker run img_name` – Create a container from `img` and start it.
  - `-d` – Detach (i.e. run in background).
    - Outputs container ID.
  - `-p h:c` – Maps port `c` in container to port `h` on host.
- `docker ps` – Show running containers.
  - `-a` – see stopped containers too.
- `docker start id` – Start a stopped container with id `id`.
  - Will start detached.
- `docker exec id command` – Run `command` in container with id `id`.
  - Useful for checking on things/inspecting container.

Stopping and Updating Your Container

- `docker stop id` – Stop container with id `id`.
  - `id` can be extracted from `docker ps` if unknown.
- `docker restart id` – restart container with id `id`.
- `docker rm id` – Remove container with id `id`.
  - Deleting a container will delete all of its data!
- To update your app, simply rebuild and run the new image!
  - Make sure to stop and delete old container.

```
[james@deck ex]$ docker run -dp 5000:5000 example_app

6a32c28e0cd362e1b0f38076a4f3d201b5002c32a46e76d4ec81fd2bc2915b33
```

Container ID

```
[james@deck ex]$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

6a32c28e0cd3 example_app "flask run --host=0.…" 3 seconds ago Up 2
seconds 0.0.0.0:5000->5000/tcp, :::5000->5000/tcp lucid_bose
```

```
[james@deck ex]$ docker stop 6a32c28e0cd3

6a32c28e0cd362e1b0f38076a4f3d201b5002c32a46e76d4ec81fd2bc2915b33
```

```
[james@deck ex]$ docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
```

```
[james@deck ex]$ docker rm 6a32c28e0cd3

6a32c28e0cd362e1b0f38076a4f3d201b5002c32a46e76d4ec81fd2bc2915b33
```

Aaaaannnddd, gone.

Problems With Sharing/Persistent Data

- All data is deleted when a container is deleted.
  - Containerizing databases :(
- One container cannot read any data from another.
- This is all by design.

The Solution: Container Volume Mounts!

- Volume mounts connect filesystem paths in the container and host.
- `docker volume create name` – Create a volume called `name`.
- Pass `--mount type=volume,src=name,target=path` to `docker run`.
  - `src` – name of volume to mount.
  - `path` – absolute path to directory from container to include in volume.
- `docker volume inspect` – View details about volume.
- `docker volume rm name` – Delete volume `name`.
  - All containers using `name` must first be deleted!

```
[james@deck ex]$ docker volume create database
database
```

```
[james@deck ex]$ docker run -dp 5000:5000 --mount
type=volume,src=database,target=/app/instance example_app
```

```
[james@deck ex]$ docker volume inspect database

[
    {
    "CreatedAt": "2023-04-13T20:44:39-05:00",
    "Driver": "local",
    "Labels": null,
    "Mountpoint": "/var/lib/docker/volumes/database/_data",
    "Name": "database", "Options": null, "Scope": "local"
    }
]
```

Where the data is stored on host
(will probably need root perms to access).

```
[james@deck ex]$ sudo ls /var/lib/docker/volumes/database/_data
database.db
```

Now you know the basics of Docker!

docker.jameshurd.net

Thank you!

jameshurd@ku.edu

fin.