

Building Containerized Applications for Data Science

Presenter

► Dr. J. Alex Hurt

- Research Assistant Professor
 - Computer Vision, Geospatial Data Analysis, & High-Performance Computing
- University of Missouri
 - Center for Geospatial Intelligence
 - Dept. of Electrical Engineering and Computer Science
- Several years of experience both using and teaching Docker and K8s concepts to researchers and educators around the country as part of NSF-funded GP-ENGINE project (<https://gp-engine.umsystem.edu>)
- Lab GitHub: <https://github.com/MUAMLL>
 - Repo for this Tutorial: <https://github.com/MUAMLL/SDSS2025>

Learning Objectives

- ▶ Introduction to Containerization
 - ▶ Fundamental Containerization Concepts
- ▶ Docker as a Container Runtime
 - ▶ Common Docker Runtime Commands
 - ▶ Using the Docker API
- ▶ Using Community Containers
 - ▶ Pulling and Running Containers
- ▶ Building Containers
 - ▶ Writing a Dockerfile
 - ▶ Using Custom Containers
- ▶ Advanced Docker Usage (*Time Permitting*)
 - ▶ Multi-Stage Builds
 - ▶ Using Continuous Integration
 - ▶ Microservices

Workshop Outline

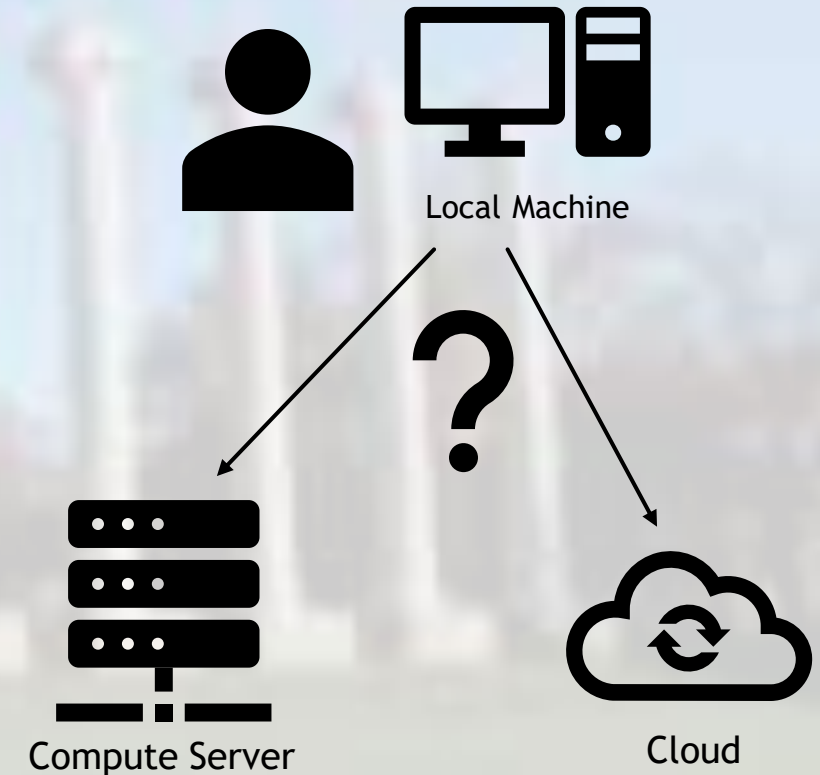
- ▶ Introduction to Software Containerization with Docker **(Slides)**
 - ▶ Containerization basics
 - ▶ Docker concepts
- ▶ Introduction to Docker **(Hands-On)**
 - ▶ The National Research Platform
 - ▶ Pulling and Using Community Containers on the NRP
- ▶ Docker Container Image Building and Publishing **(Slides)**
 - ▶ Writing Dockerfiles
 - ▶ Building and Using Custom Containers with Registries
- ▶ Building Dockerfiles and Using Custom Docker Containers **(Hands-On)**
 - ▶ Write your own Dockerfile
 - ▶ Build and use a custom container from your own Dockerfile
- ▶ Advanced Concepts with Docker **(Slides)**
 - ▶ Multi-Stage Builds
 - ▶ Using Continuous Integration
 - ▶ Microservices

Introduction

What is Docker and what problem is Docker trying to solve?

The Problem: ???

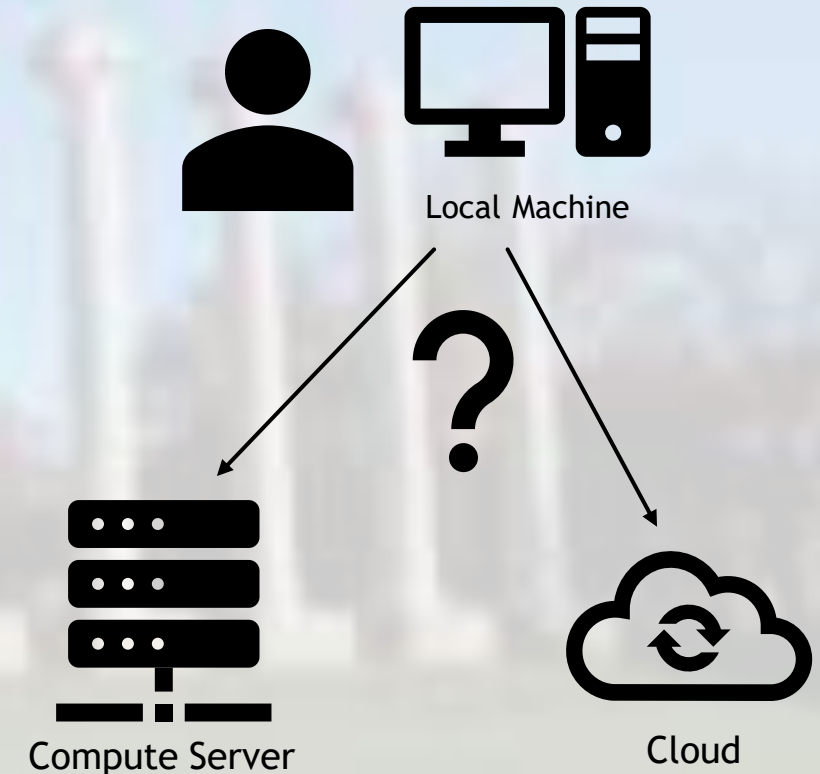
- ▶ What challenges have you had with using data-science software or software in general?
- ▶ Share with your neighbors and try to come up with a list of common challenges or annoyances.



The Problem:

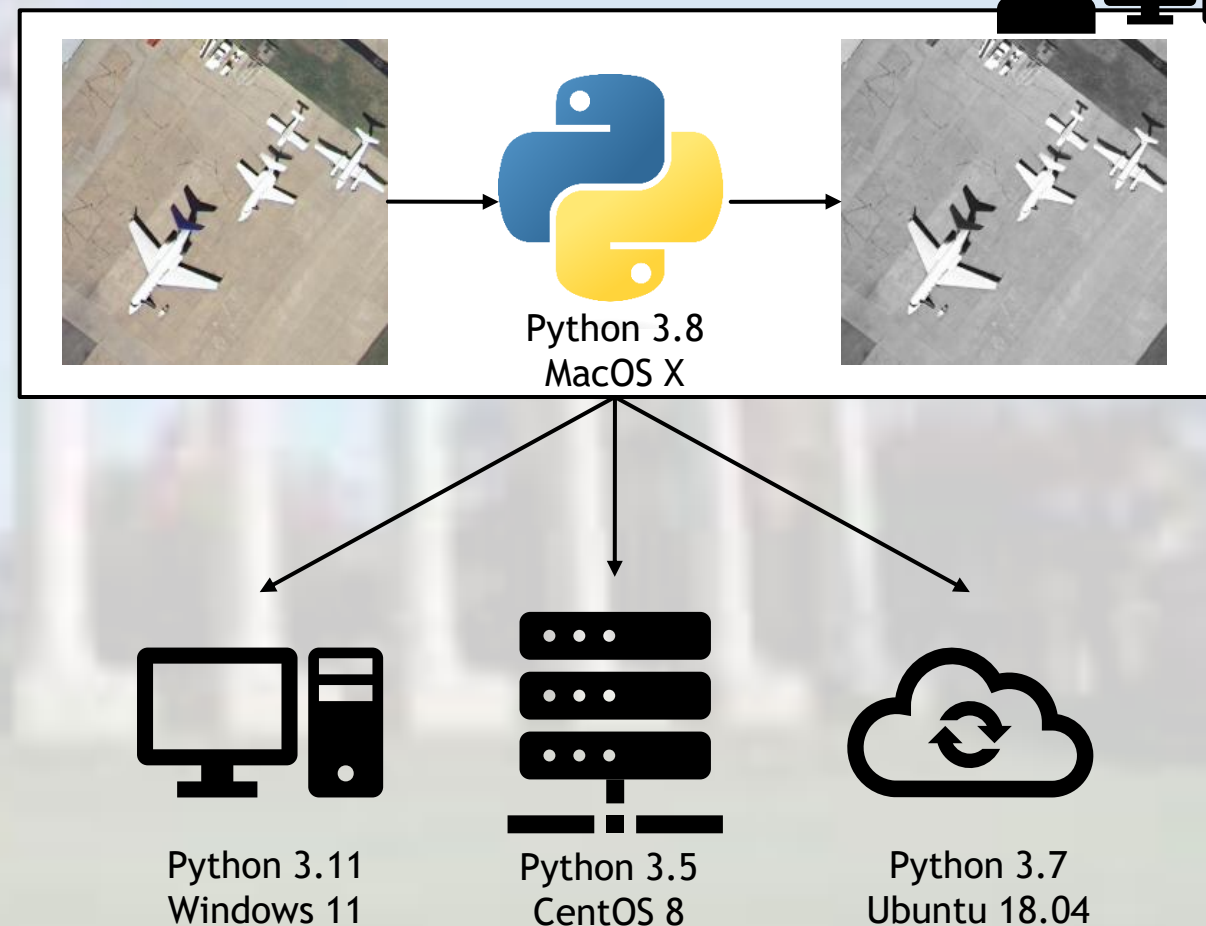
Scalability & Reproducibility

- ▶ Development often occurs on local development machines
- ▶ How do we ensure reliable portability of the software developed on local development machines to other computational environments?
- ▶ How do we move code from local development to running on large servers on large swaths of data?



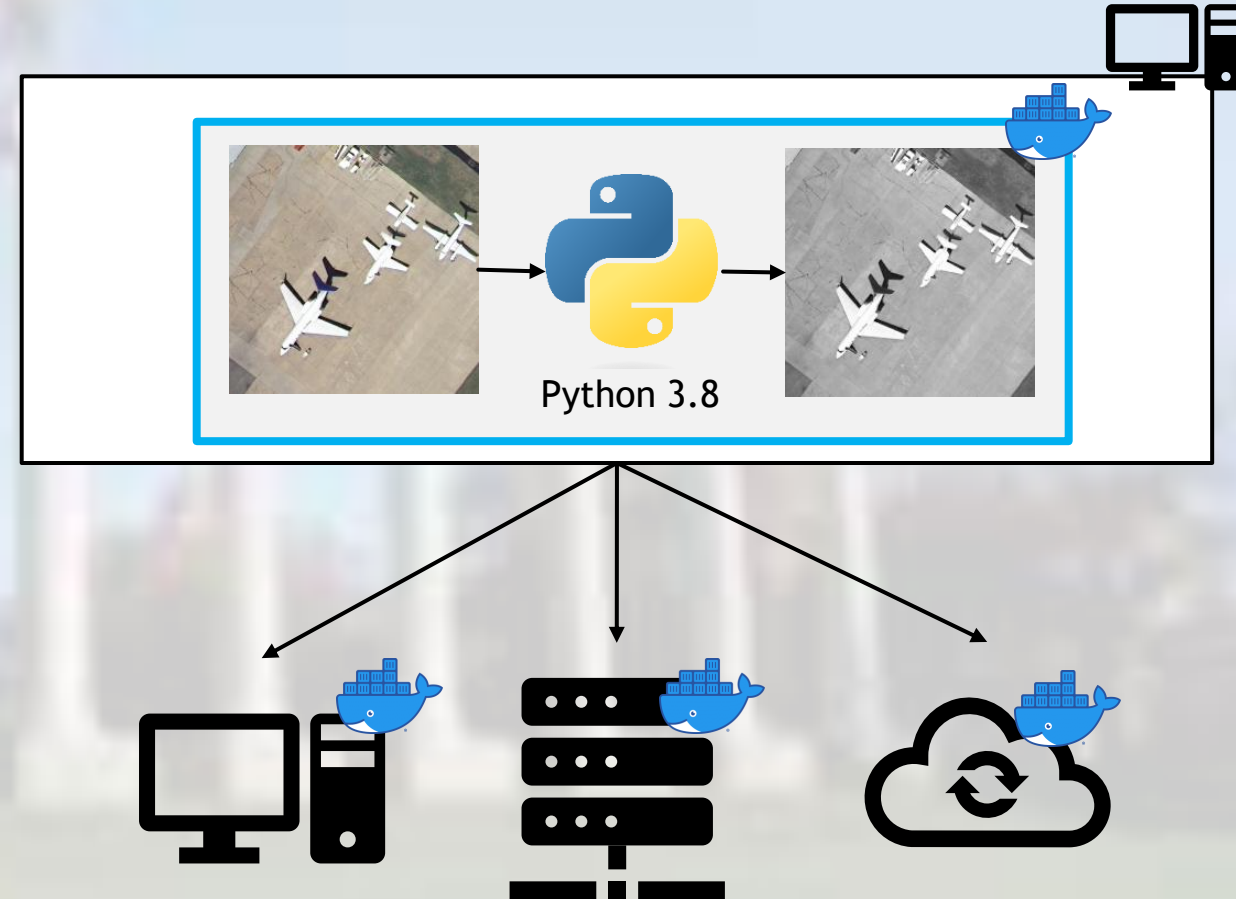
Example: Python Application for Image Processing

- ▶ I've built an application on my local machine to enable efficient image processing using Python 3.8
- ▶ My application requires certain Python packages **AND** system packages
- ▶ The specific versions of Python and the system packages are required
- ▶ How do I move my application to co-worker's computer? What about to a compute server? To the cloud?
 - ▶ Each of these locations will have their own installed system libraries, python installations, and python packages



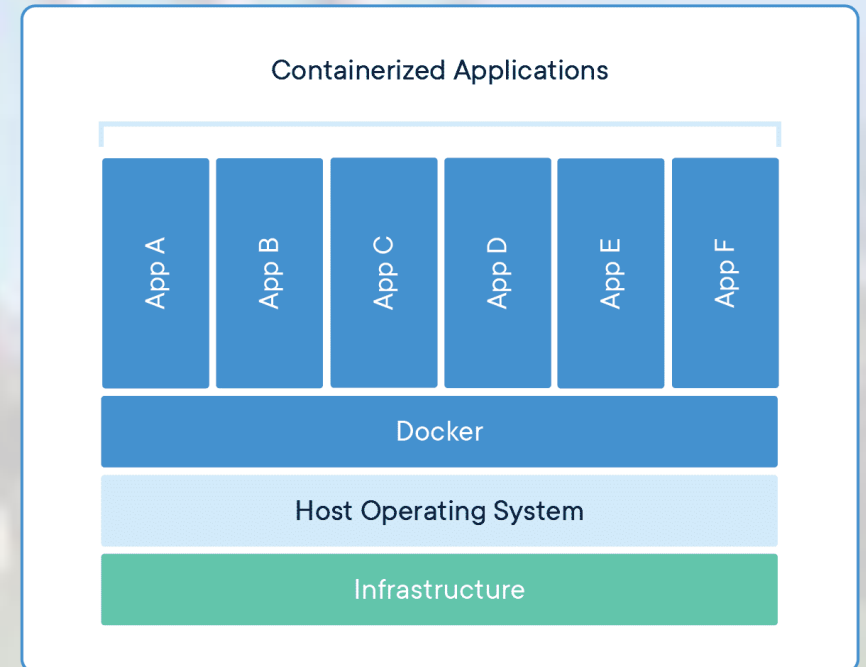
Containerization

- ▶ Solution: Package all of our requirements, at system level and at the language-specific level, into a software package, called a **container image**, that can be run anywhere
- ▶ Each location now only needs a **container runtime** and the software package



Container Runtimes

- ▶ Container runtimes are software components that run containers on a host operating system
- ▶ Every machine that we want to run containers on needs a container runtime
- ▶ There are multiple container runtimes:
 - ▶ Docker: most common
 - ▶ Podman: RHEL/CentOS replacement for Docker. Does not require root access
 - ▶ Singularity: Common in HPC applications
- ▶ We will discuss Docker here, as its concepts are generalizable to other runtimes



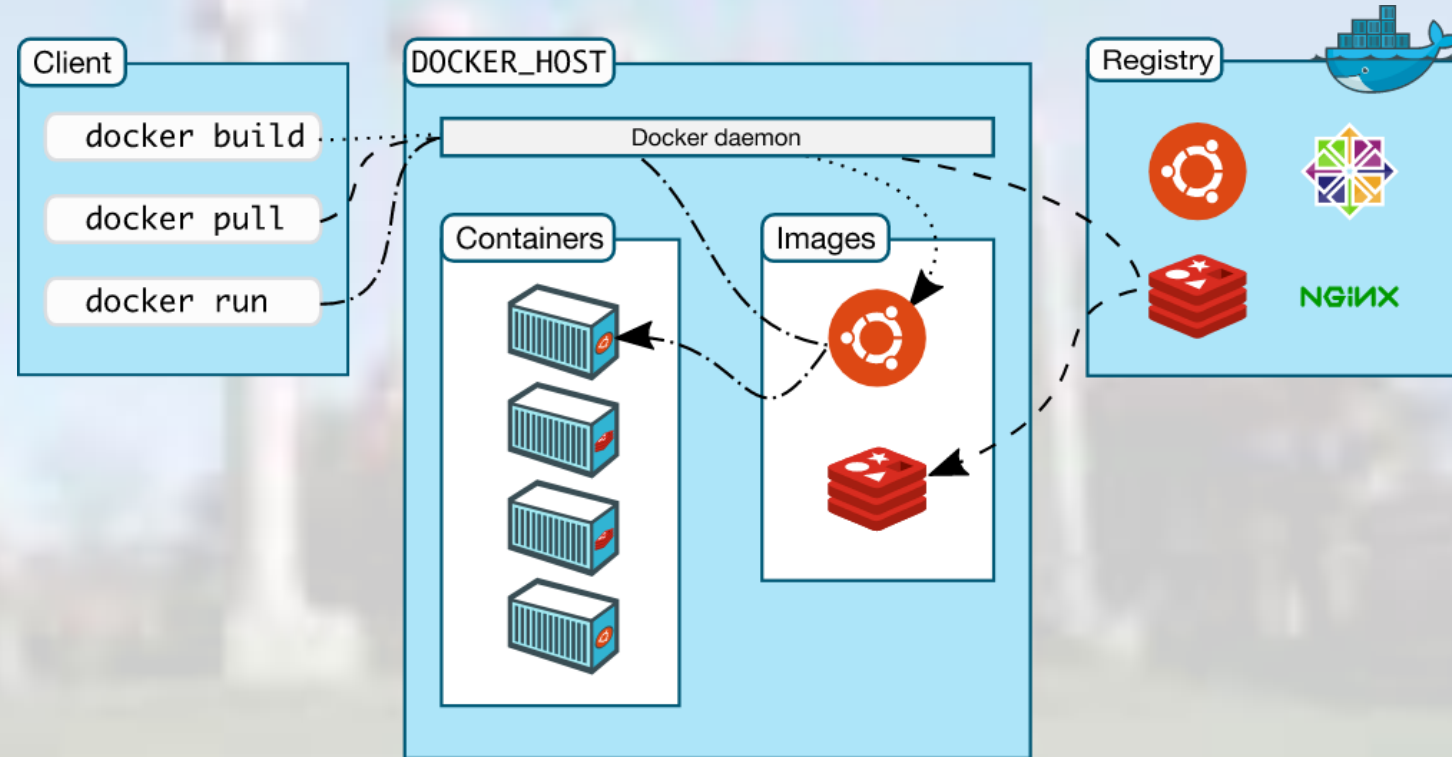
Docker

► What is Docker?

- Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.¹
- You can think of Docker containers as mini-VMs that contain all the packages, both at the OS and language-specific level, necessary to run your software.

► Why Docker?

- Docker enables predictable and reliable deployment of software.
- Docker containers are portable!
 - local development computers, compute clusters, internal compute servers, cloud infrastructure, and more!
- *Docker containers enable reliable portability of software to nearly any compute environment*



1. [https://en.wikipedia.org/wiki/Docker_\(software\)](https://en.wikipedia.org/wiki/Docker_(software))
 2. Image: <https://docs.docker.com/get-started/overview/>

Understanding the Docker Architecture

► Host / Daemon / Runtime

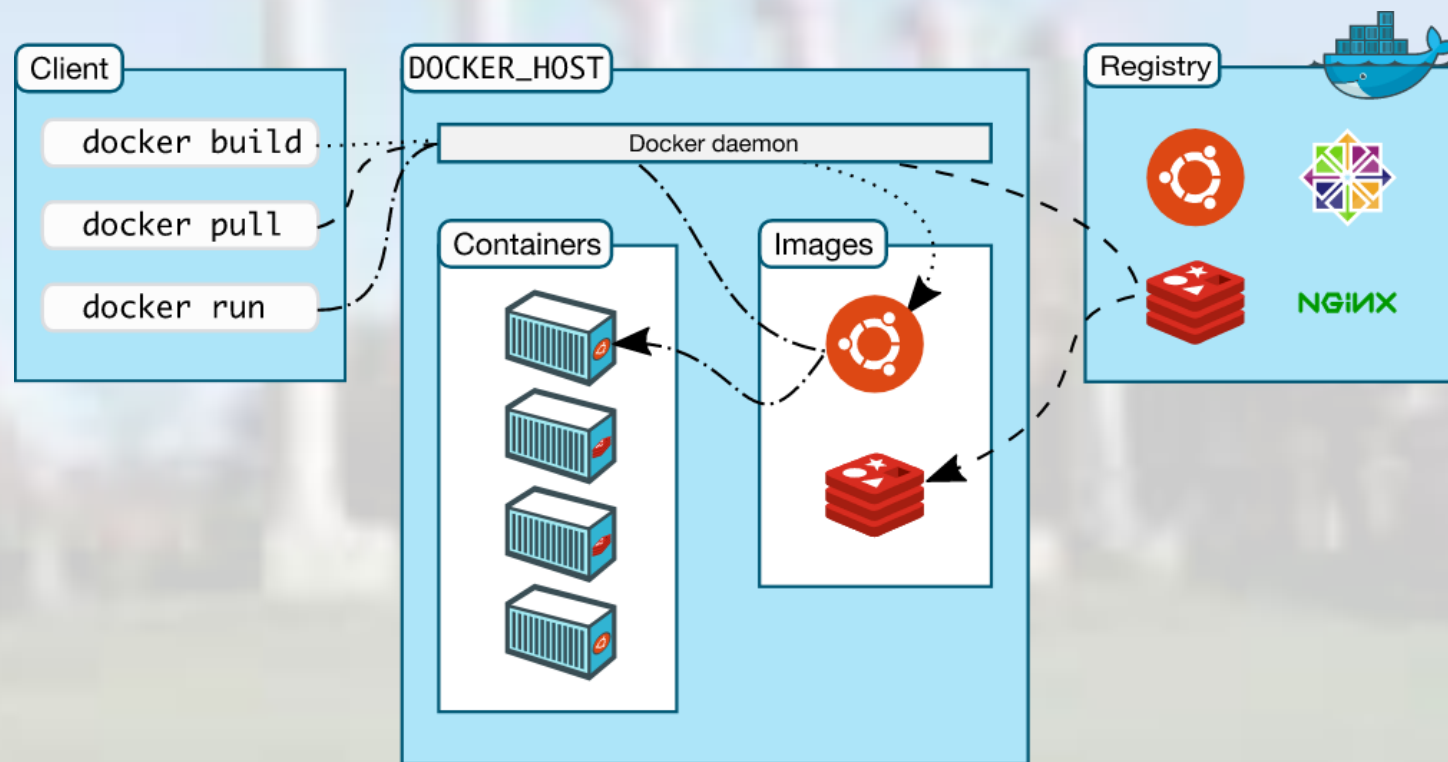
- Responsible for maintaining the running and stopped **containers** and the built or pulled in **container images**
- Interfaces with the Operating System and provides the **file and process isolation** to running containers

► Client

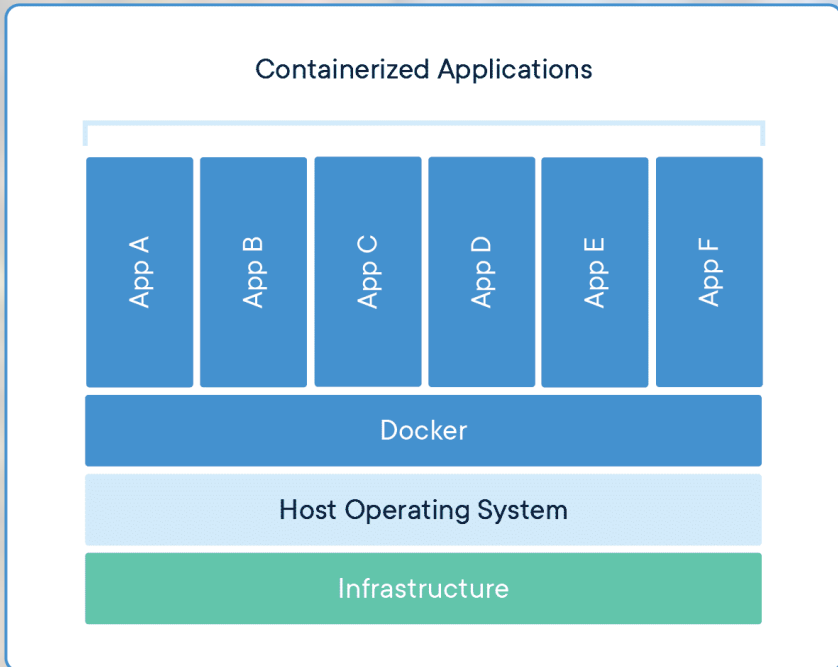
- How users and developers interact with the Docker runtime
- Each runtime provides a set of commands to its client that allows the users and developers running the runtime to perform needed operations: pull, push, run, build, etc.

► Registry

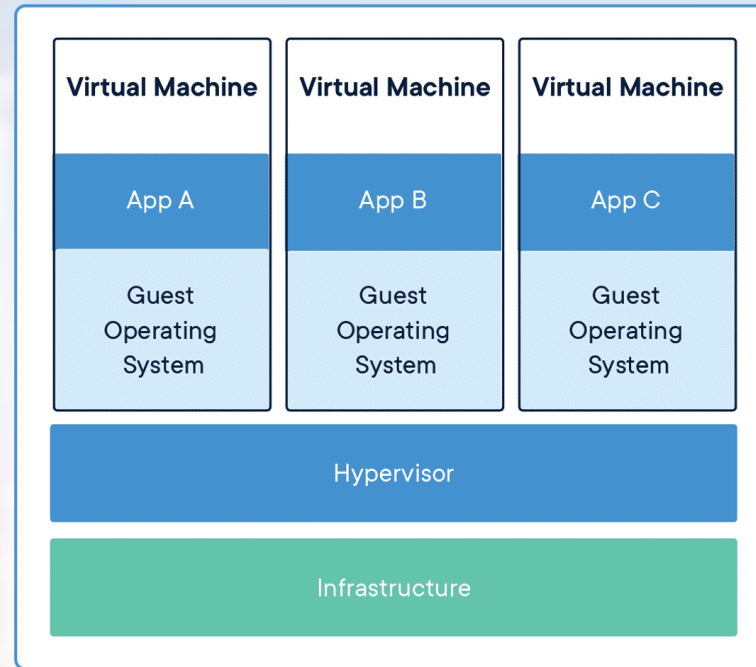
- The mechanism used to share containers between users, organizations, and compute environments
- Provides network-capable storage of container images



Docker: Containers vs Virtual Machines



Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.



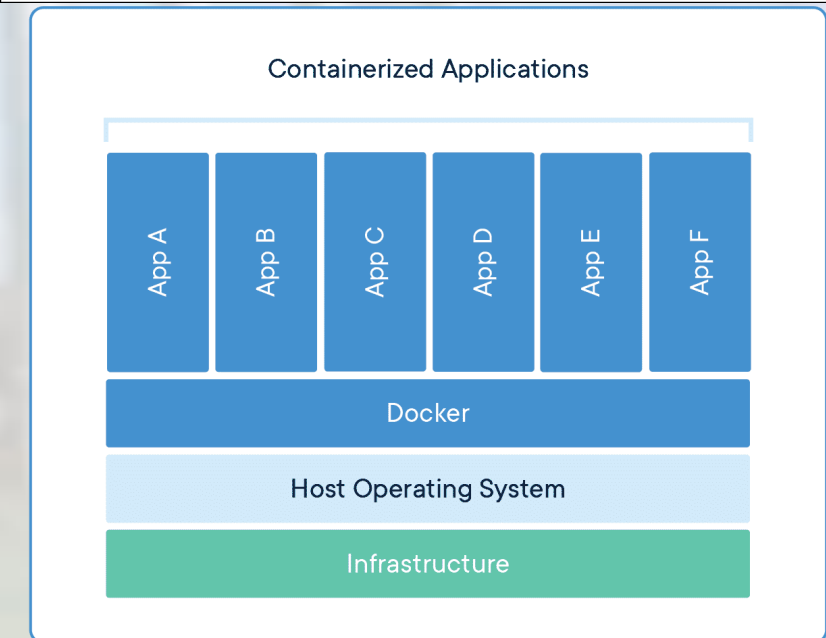
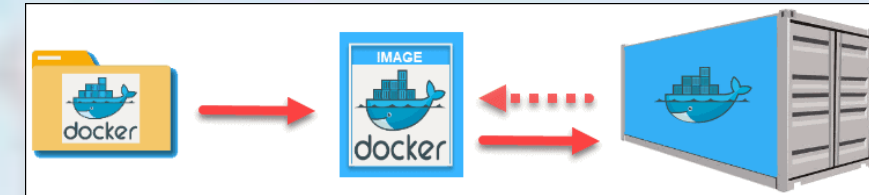
Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can also be slow to boot.

Docker Concepts

Key concepts: containers, container images, Dockerfiles, container registries

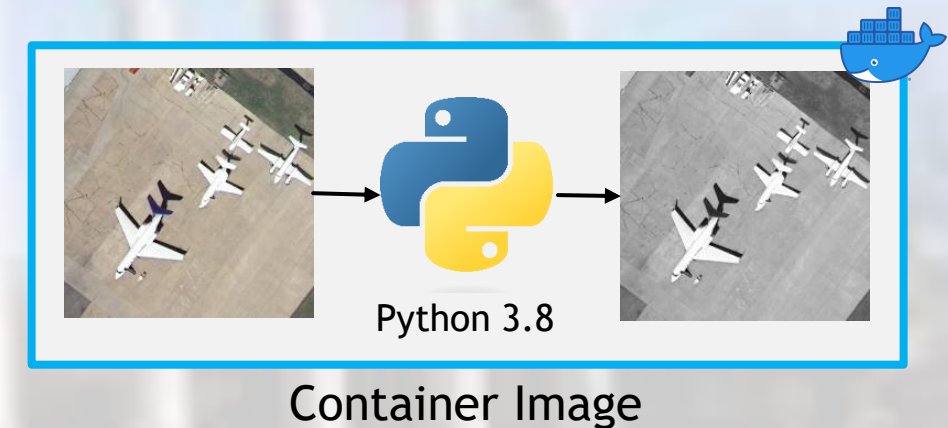
Key Docker Concepts

- ▶ **Container** - A container is a standard unit of software that packages up code and all of its dependencies, so the application runs reliably from one computing environment to another.
- ▶ **Image** - Standard unit of software that packages up code and its dependencies so the application runs reliably from one computing environment to another.
 - ▶ Includes everything needed to run an application: code, runtime, system tools, system libraries and settings.
- ▶ **Dockerfile** - A list of commands and instructions describing how to build an Image
- ▶ **Registry** - a service for storing container images



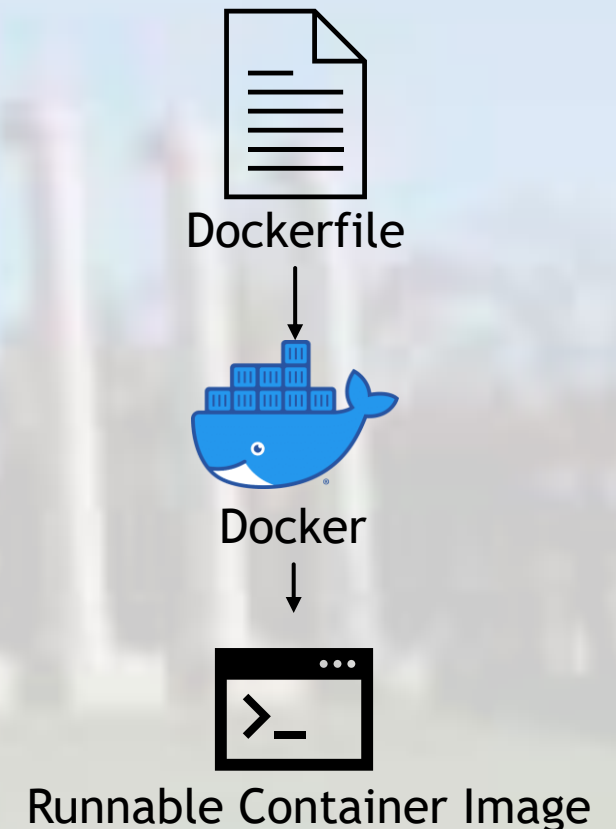
Containers vs Container Images

- ▶ The terms containers and images are often used interchangeably when discussing Docker, but there are some key distinctions
- ▶ Container images are software packages that include all necessary software to run the application/library
- ▶ Containers are an instantiation of container images
 - ▶ Images become containers at runtime
 - ▶ Analogous to relationship between a script and the process or a class and an object



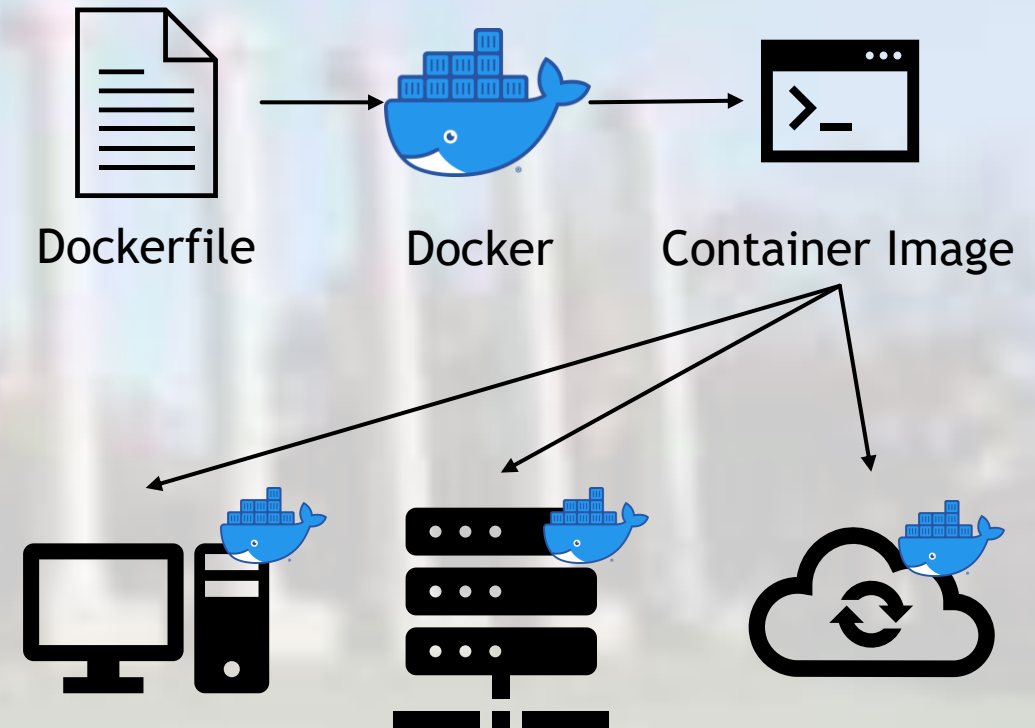
Dockerfiles

- ▶ Recall: Containers and container images enable reliable portability of software
- ▶ We need a way to build container images in a distributed and standardized way, so that anyone with a container runtime can build and run our software package (container image)
- ▶ Dockerfiles are the template, or recipe, for how a container image will be built
- ▶ Dockerfiles use a custom format and set of commands to describe how a container image will be built



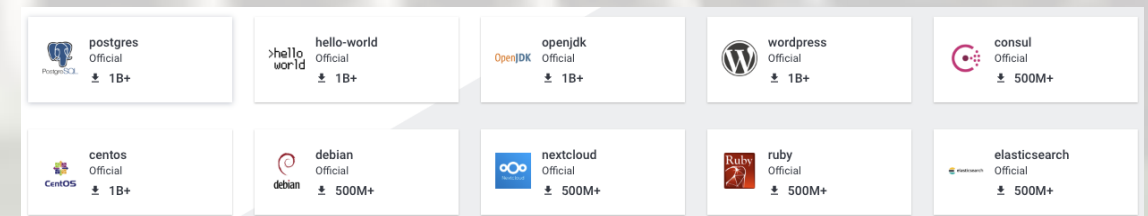
Distributing and Sharing Containers

- ▶ Container runtimes enable reliable portability of software by building container images
- ▶ We can utilize container images built and published by the container community utilizing base images
- ▶ How do we share and distribute container images that we have built?
 - ▶ Container Image Registries



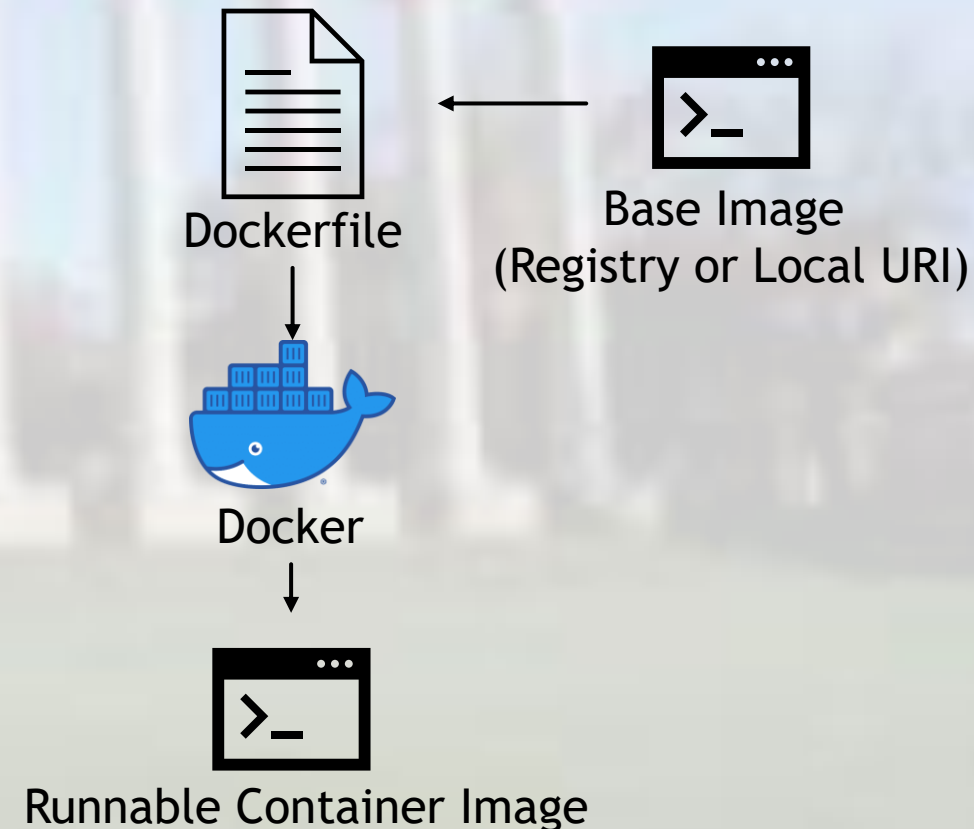
Docker Image Registries

- ▶ Container Registries are web-enabled storage locations for Docker container images
 - ▶ Similar to Google Drive for documents and spreadsheets
- ▶ Each image published on a registry contains a name and a tag:
 - ▶ `python:3.8` → Python is the name of image and 3.8 is the tag
- ▶ If a URL is not specified, the default registry used is Docker Hub
 - ▶ <https://hub.docker.com/>
- ▶ Other Container Registries
 - ▶ Docker can work with third party container registries when given full URL to the image
 - ▶ Example: `nvcv.io/nvidia/pytorch:22.08-py3`
- ▶ Security and Visibility
 - ▶ Container images published on registries can be public or private



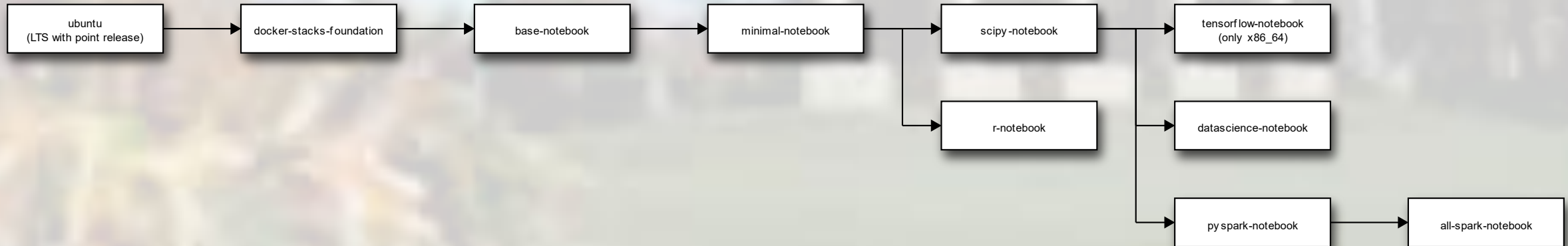
Understanding Docker Base Images

- ▶ Container images are built from a **base container**
- ▶ The base container determines what software, packages, and commands are available to the container
 - ▶ We cannot run apt in a container that uses Rocky as a base, but we can in a container that uses Debian / Ubuntu as a base
- ▶ Understanding the base used for a container is crucial as a user as it tells us how to interact with the software in the running container created from the container image
- ▶ Base images are shared and derived from either open-source registries or from locally built images
- ▶ Common Bases:
 - ▶ Linux Operating Systems: Ubuntu, Rocky, Alpine
 - ▶ Software Stacks: Python3, NodeJS, CUDA
 - ▶ Frameworks: PyTorch, Jupyter, ReactJS



Using Community Published Container Images

- ▶ Container images are extensible!
 - ▶ Containers can be built from other images
- ▶ We can then build hierarchical docker containers, where each container has a specific set of dependencies and packages installed
- ▶ Base images enable rapid and reproducible building of even complex docker containers by leveraging the open source, published docker images



Example: Jupyter Docker Images¹

1. <https://jupyter-docker-stacks.readthedocs.io/en/latest/using/selecting.html>

Understanding Container Image URIs

- ▶ Container image URIs follow a fixed structure that gives us information about its source, the software available, and often some kind of versioning information
- ▶ The structure of all container image URIs is:

`[REGISTRY URI]/[namespace or user]/[image name]:[tag]`

- ▶ However, many parts of this URI are optional and have defaults:
 - ▶ REGISTRY URI defaults to the Docker Hub registry
 - ▶ If user or namespace is omitted, then the container image is coming from the official library
 - ▶ Tag default to latest
- ▶ Examples:

`ubuntu:18.04`

`jalexhurt/imagemagick`

`gitlab-registry.nrp-nautilus.io/jhurt/mmdet-v3:016bcbcd`

Understanding Container Image URIs

`[REGISTRY URI]/[namespace or user]/[image name]:[tag]`

`ubuntu:18.04`

Registry: Docker Hub
Namespace: None - Official Library
Name: Ubuntu
Tag: 18.04

`jalexhurt/imagemagick`

Registry: Docker Hub
Namespace: jalexhurt
Name: imagemagick
Tag: latest

`gitlab-registry.nrp-nautilus.io/jhurt/mmdet-v3:016bcbcd`

Registry: NRP GitLab
Namespace: jhurt
Name: mmdet-v3
Tag: 016bcbcd

Docker Client API

Common Commands and Interactive Docker Usage including pulling, building, and running containers

Common Docker Commands: System Information

- ▶ To see all running containers:

```
docker ps
```

- ▶ To see all container images on our system:

```
docker images
```

- ▶ Delete a container:

```
docker rm [container]
```

- ▶ Delete a container image:

```
docker rmi [image]
```

Common Docker Commands: Pulling Containers

- ▶ To download or update a container image from a registry, we use the docker pull command:

```
docker pull [Image Name]
```

- ▶ We can specify simply the name of the image, and by default, Docker Hub will be used as the registry and latest will be used as the tag. Therefore the following two commands are the same:

```
docker pull ubuntu
```

```
docker pull docker.io/library/ubuntu:latest
```

- ▶ We can specify a name + tag to pull a specific tag from Docker Hub:

```
docker pull python:3.8
```

- ▶ Finally, we can specify a full URL to use a custom registry:

```
docker pull nvcr.io/nvidia/pytorch:23.02-py3
```

Common Docker Commands: : Running Containers

- ▶ Running a container: docker run

```
docker run [options] [Image Name] [command]
```

- ▶ If the container image is not found on your system, docker will search for that name and tag on Docker Hub and download it before running it

- ▶ Interactive running a container on Docker Hub:

```
docker run -it ubuntu:18.04
```

- ▶ Any options not specified will be set to default, which are set by either the Docker image itself (through the Dockerfile) or by the Docker daemon

- ▶ Overriding the startup command:

```
docker run -it python:3.8 /bin/bash
```

- ▶ Running a container in the background:

```
docker run -d [Image Name]
```

- ▶ Mounting a local directory to the container:

```
docker run -it -v /mylocaldir:/containerdir ubuntu:20.04 /bin/bash
```

- ▶ Exposing host ports to the container:

```
docker run -p 8888:8888 -d nginx
```

Common Docker Commands: Building Containers

- ▶ To build a container image, we use the docker build command:

```
docker build [options] [context path]
```

- ▶ To build a container image from a Dockerfile in the current directory:

```
docker build .
```

- ▶ It is common to set a name and tag for the image you are building:

```
docker build -t myimage:latest .
```

- ▶ When using more than 1 Dockerfile, we specify which one with another flag:

```
docker build -t myimage:latest -f Dockerfile.py3 .
```

- ▶ If we are using a single Dockerfile with multiple build stages, we specify that as well:

```
docker build -t myimage:latest --target stage2 .
```

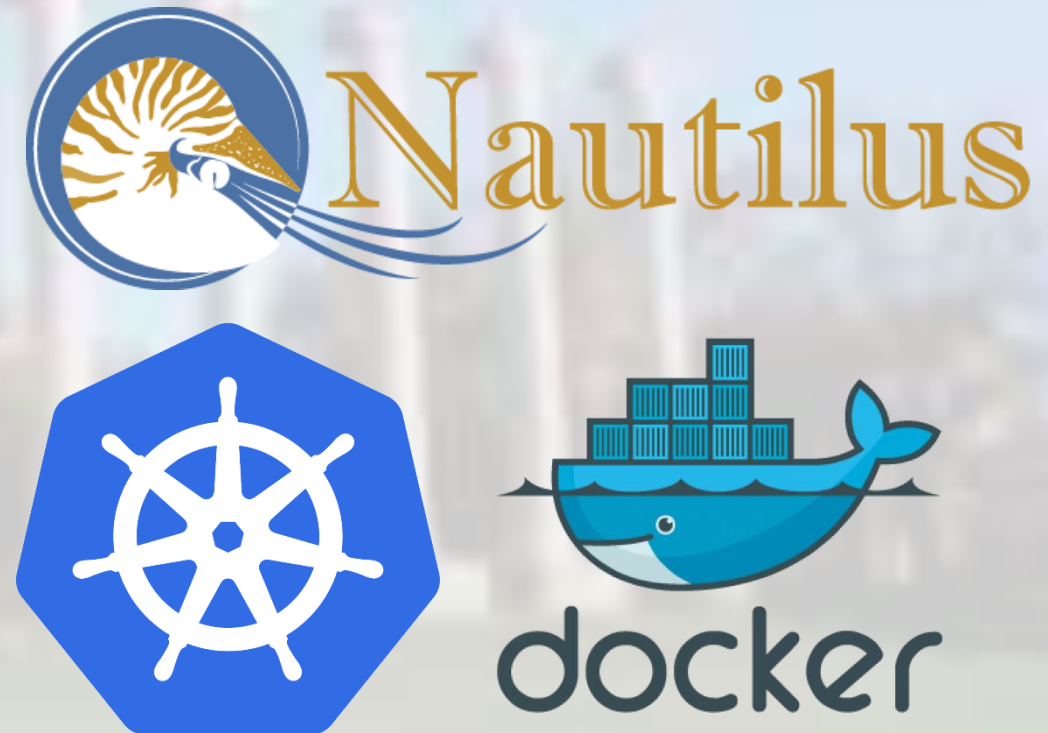

National Research Platform Nautilus Hyper-Cluster

A quick note on Kubernetes Clusters, the NRP, Commercial Clouds, and other K8s Clusters

- ▶ All commercial cloud providers support Containers and Kubernetes
- ▶ The concepts and examples in this tutorial may require minor modifications to adapt to other environments
- ▶ We are using the US National Research Platform solely for demonstration and tutorial purposes
- ▶ All Container and Kubernetes concepts are portable to commercial clouds or other research Kubernetes platforms

NSF NRP Nautilus HyperCluster

- ▶ The NSF Nautilus HyperCluster is an NSF-funded Kubernetes cluster with vast resources that can be utilized for various research purposes:
 - ▶ Prototyping research code
 - ▶ S3 cloud storage for data and models
 - ▶ Accelerated small-scale research compute
 - ▶ Scaling research compute for large scale experimentation
- ▶ Resources Available:
 - ▶ CPU Cores: 9,769
 - ▶ RAM: 167 TB
 - ▶ NVIDIA GPUs: 1342
- ▶ More information:
<https://nationalresearchplatform.org>



Hands-On: Docker API & Using Containers

<https://user-dev.nrp-nautilus.io>

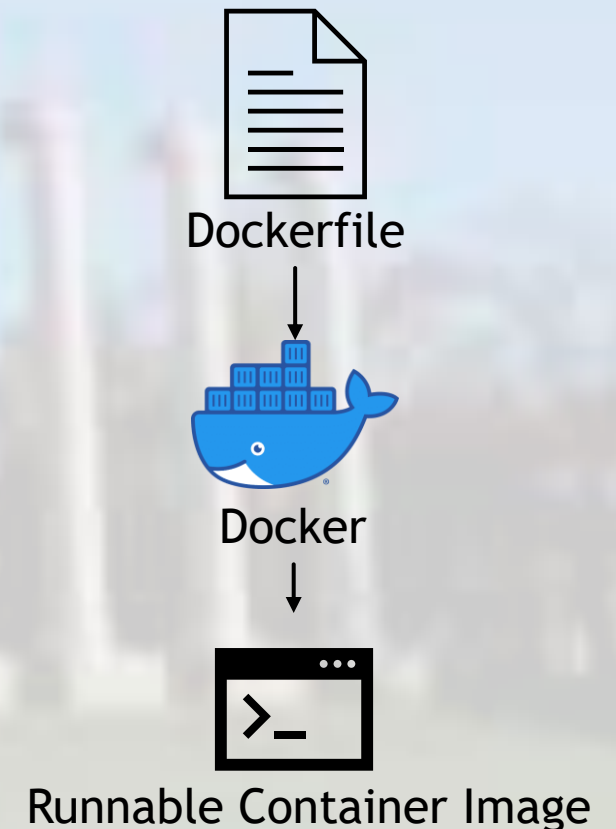
```
git clone https://github.com/MUAMLL/SDSS2025
```

Building Your Own Container

Dockerfiles

Dockerfiles

- ▶ Recall: Containers and container images enable reliable portability of software
- ▶ We need a way to build container images in a distributed and standardized way, so that anyone with a container runtime can build and run our software package (container image)
- ▶ Dockerfiles are the template, or recipe, for how a container image will be built
- ▶ Dockerfiles use a custom format and set of commands to describe how a container image will be built



Dockerfile: Common Commands

- ▶ Dockerfiles function via a set of directives and their respective arguments:

DIRECTIVE arg1 arg2 ...

- ▶ FROM

- ▶ The FROM command defines the starting point for the Docker build process. If you are building custom software from the ground up, you may be setting this to a Linux operating system, such as ubuntu.
- ▶ If you are working in a framework or programming language, this may also be a certain version of that framework, such as python:3.8, node:8, pytorch:1.8, etc.
- ▶ The docker image specified in the from command must be present either on the public docker hub repository, or be visible to the build context via a full URL.

- ▶ ENV & ARG

- ▶ The ENV and ARG commands define environment variables during the build process. There are 2 key differences between them:
- ▶ ENV commands persist to the final container, so ENV key value will persist to the launched container, while ARG key value will not
- ▶ ARG commands can be overridden at build time, which allows for templating of Dockerfiles
- ▶ The ENV command is very useful for tasks like ensuring your executable is present on the PATH of the container, while ARG is useful for things like ensuring you are building the correct version of the software.

Dockerfile: Common Commands

► CMD and ENTRYPOINT

- The CMD and ENTRYPOINT commands both set the command to be run when the container starts via a docker run command.
- There is 1 key difference between them: a CMD can be overridden via either a downstream Dockerfile (i.e., someone uses your container in their FROM command) or via the command line during container startup. An ENTRYPOINT cannot be as easily overridden, and requires a special flag to be overridden.
- Best practice is to set a CMD unless you have reasoning behind using ENTRYPOINT

► COPY

- The COPY command is how local files are copied into the container. Recall that Docker containers are mini virtual machines, meaning that it has its own filesystem. In order to build and run software in docker, we often need to copy our code and scripts into the container. To do this, we use the COPY command

► RUN

- The RUN command tells the docker build process to run a command inside the container. This could be everything from installing a package with apt (RUN apt install) to creating and removing files in the container.
- The commands available for the RUN command are determined by the base container, i.e. apt will only be available in Debian based containers.

Sample Dockerfile

```
ARG PYVERSION=3.8
```

```
FROM python:${PYVERSION}
```

```
RUN mkdir -p /workspace
```

```
WORKDIR /workspace
```

```
COPY /requirements.txt /workspace
```

```
RUN pip install -r ./requirements.txt
```

```
COPY /*.py /workspace/
```

```
CMD /bin/bash
```

Create a build argument for the python version that defaults to 3.8

Start FROM an existing image in a Docker *registry*. In this case, python

Create a directory named /workspace and set it as the working directory

Copy a file named “requirements.txt” from the build directory to /workspace in the container

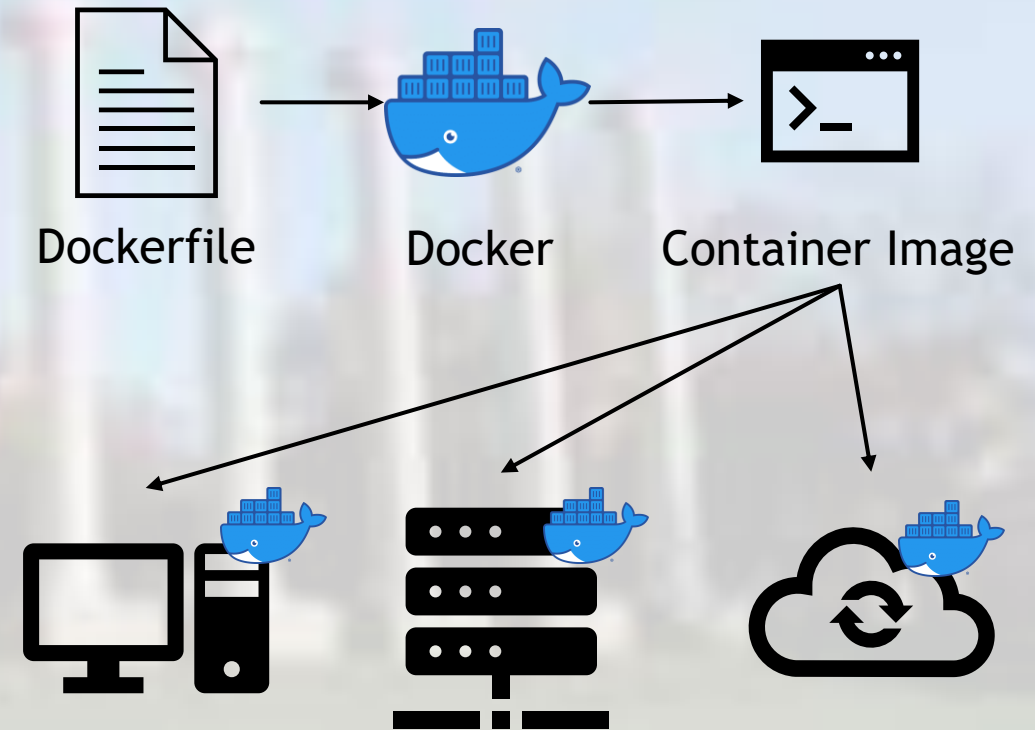
Use pip to install the requirements defined in requirements.txt

Copy all python files in the build directory to the /workspace directory in the container

Set the default command to run when the container starts to /bin/bash

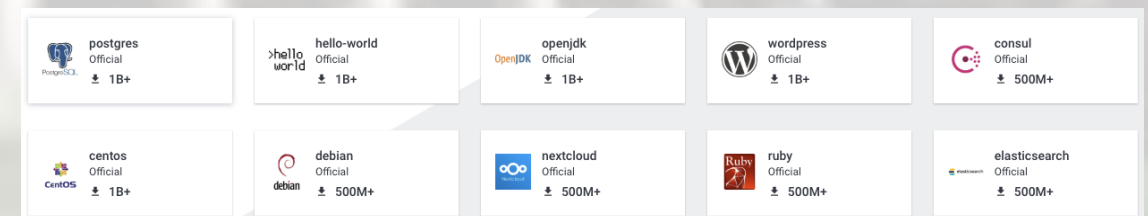
Distributing and Sharing Containers

- ▶ Container runtimes enable reliable portability of software by building container images
- ▶ We can utilize container images built and published by the container community utilizing base images
- ▶ How do we share and distribute container images that we have built?
 - ▶ Container Image Registries



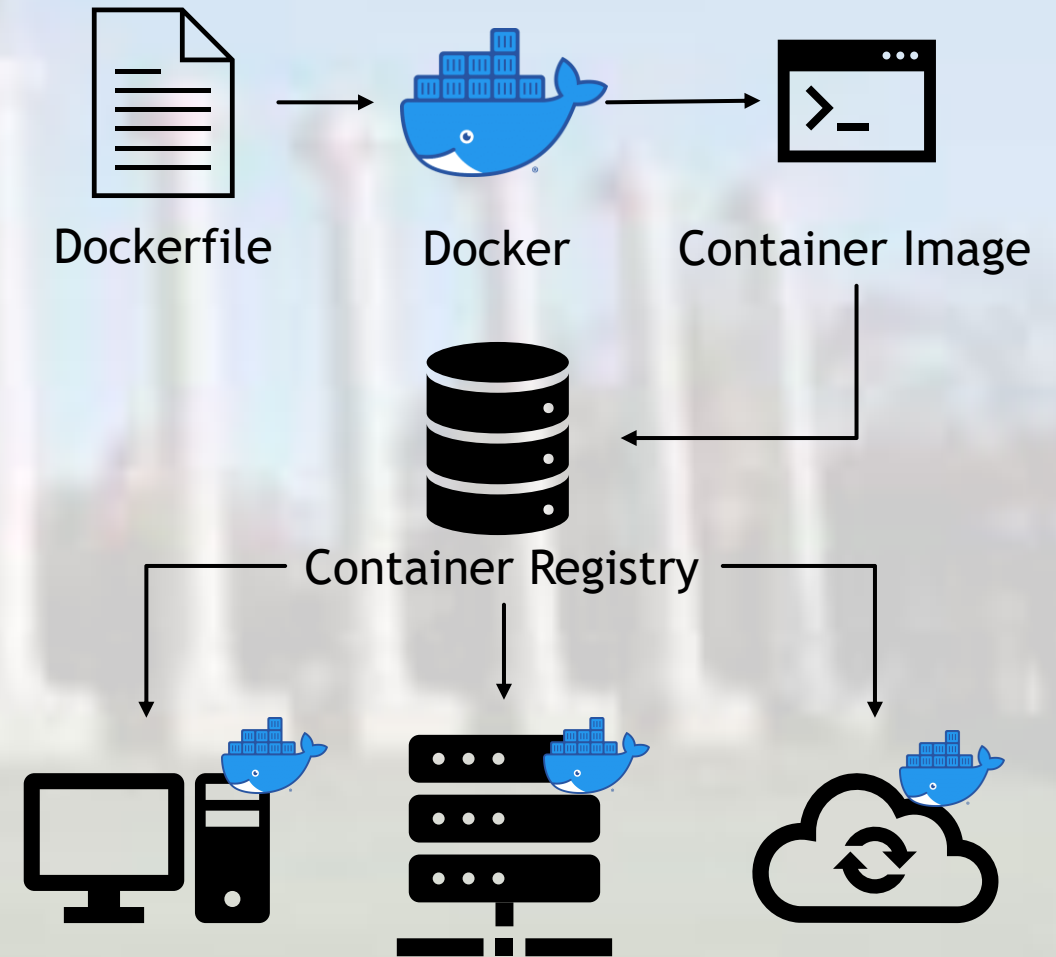
Docker Image Registries

- ▶ Container Registries are web-enabled storage locations for Docker container images
 - ▶ Similar to Google Drive for documents and spreadsheets
- ▶ Each image published on a registry contains a name and a tag:
 - ▶ python:3.8 → Python is the name of image and 3.8 is the tag
- ▶ If a URL is not specified, the default registry used is Docker Hub
 - ▶ <https://hub.docker.com/>
- ▶ Other Container Registries
 - ▶ Docker can work with third party container registries when given full URL to the image
 - ▶ Example: `nvcv.io/nvidia/pytorch:22.08-py3`
- ▶ Security and Visibility
 - ▶ Container images published on registries can be public or private



Revisiting our Example

- ▶ We can publish our container image to a registry, and then pull down our container image in each computing environment in which we want to use it
- ▶ We can utilize public registries, like Docker Hub, or private ones, to control who we allow to pull down our container



Docker in Action: ImageMagick

Dockerfile

```
FROM ubuntu:20.04

RUN apt update -y && \
    apt install -y imagemagick wget

RUN mkdir -p /workspace
WORKDIR /workspace

CMD /bin/bash
```

Bash

```
$ docker build . -t imagemagick

$ docker tag  imagemagick jalexhurt/imagemagick

$ docker push  jalexhurt/imagemagick
```

- ▶ We can combine Docker with VCS & CI/CD systems to automate this process, but for this example we do each step manually
- ▶ Once we have built and pushed our container, we can see it online on Docker Hub:
 - ▶ <https://hub.docker.com/r/jalexhurt/imagemagick>

Using the Container

- ▶ Once we have pulled the container, we can run it and use the software packaged inside!
- ▶ In this example, we download an image from the COCO dataset
- ▶ Using ImageMagick, we then:
 - ▶ read its properties
 - ▶ resize and convert it to PNG
 - ▶ read the properties of the newly created PNG



Bash

```
$ docker run -it jalexhurt/imagemagick:latest  
$ wget http://farm8.staticflickr.com/7441/9539317874_8b108e4489_z.jpg -O image.jpg  
$ identify image.jpg  
$ convert -resize 512x384 image.jpg image.png  
$ identify image.png
```

Hands-On: Building Custom Containers

<https://user-dev.nrp-nautilus.io>

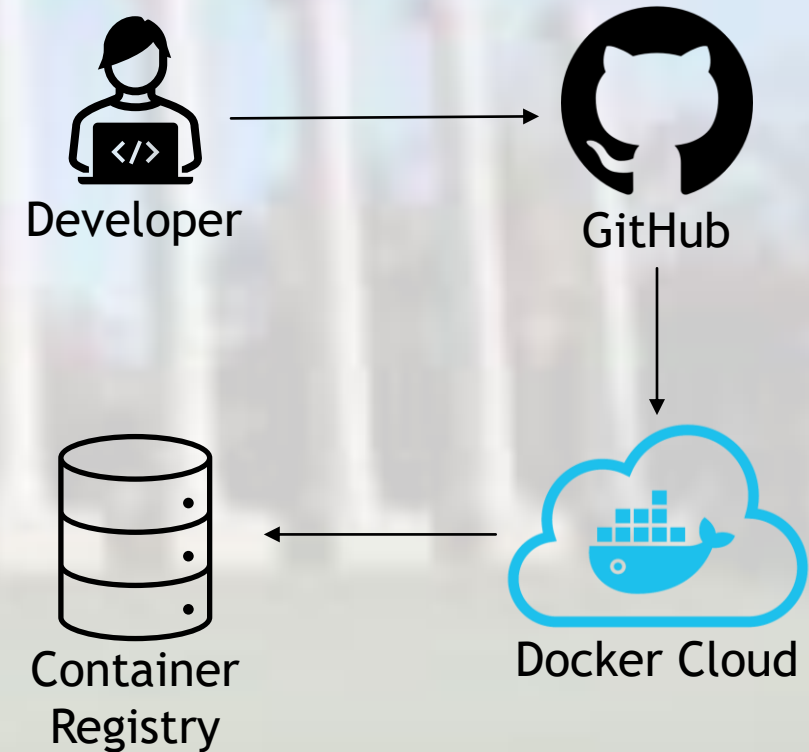
```
git clone https://github.com/MUAMLL/SDSS2025
```


Advanced Docker Concepts

Automation, Microservices, Docker Compose,
Multi-Stage Builds, and Cloud Computing

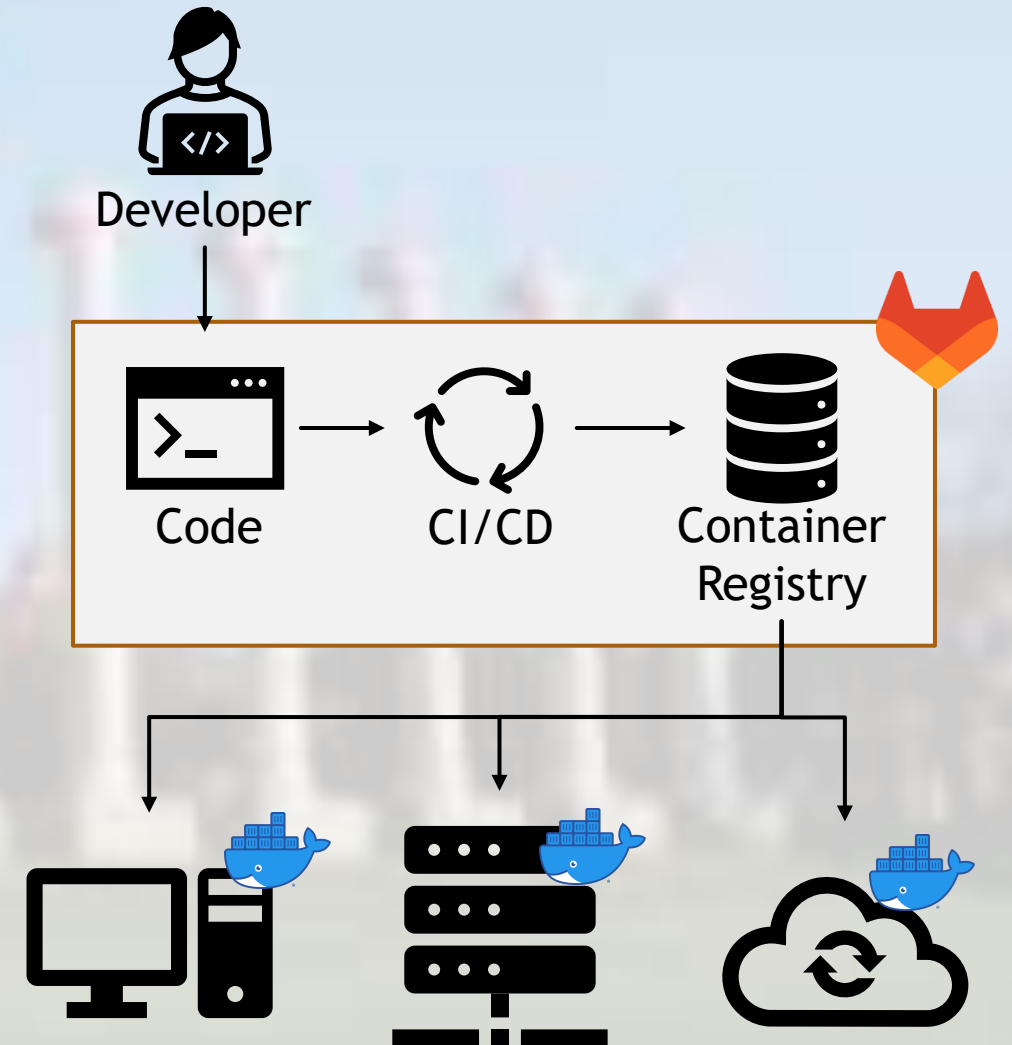
Automation and Docker Registries

- ▶ Recall: Dockerfiles are a set of instructions to build an image
- ▶ Continuous Integration / Continuous Deployment (CI/CD) systems can enable automation of publishing docker images
- ▶ Common CI/CD Services:
 - ▶ Gitlab CI
 - ▶ Docker Cloud
 - ▶ Jenkins



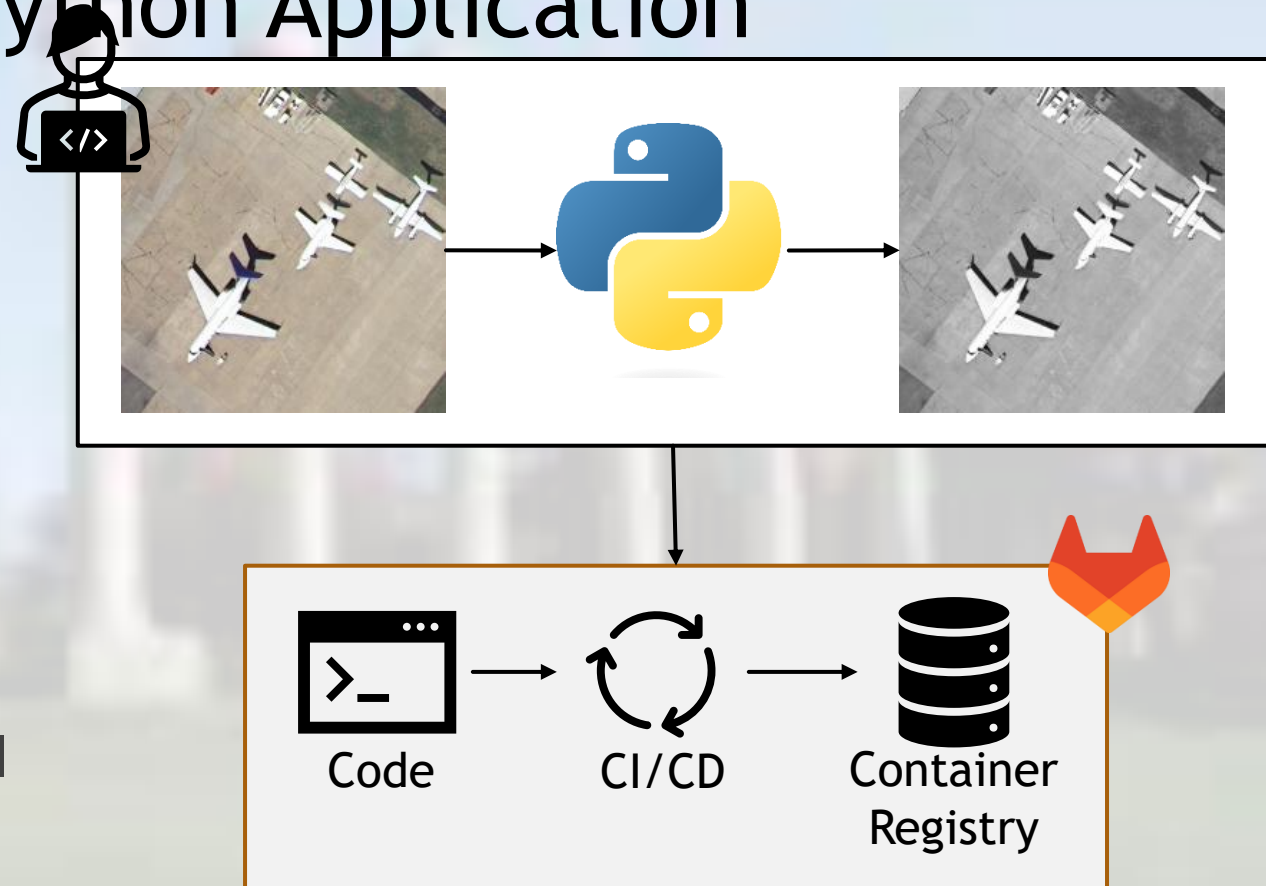
Automation with GitLab

- ▶ GitLab is a common tool for CI/CD with Docker because it contains VCS, CI/CD and a built-in image registry with access restrictions
 - ▶ Only those added to the repository can access the code, CI/CD build instructions, and the container registry
- ▶ In our previous example, the VCS, CI/CD, and Registry providers were all independent, but GitLab contains all three of these on a per-repository or per-group basis



Automation in Action: GitLab CI + Registry for Python Application

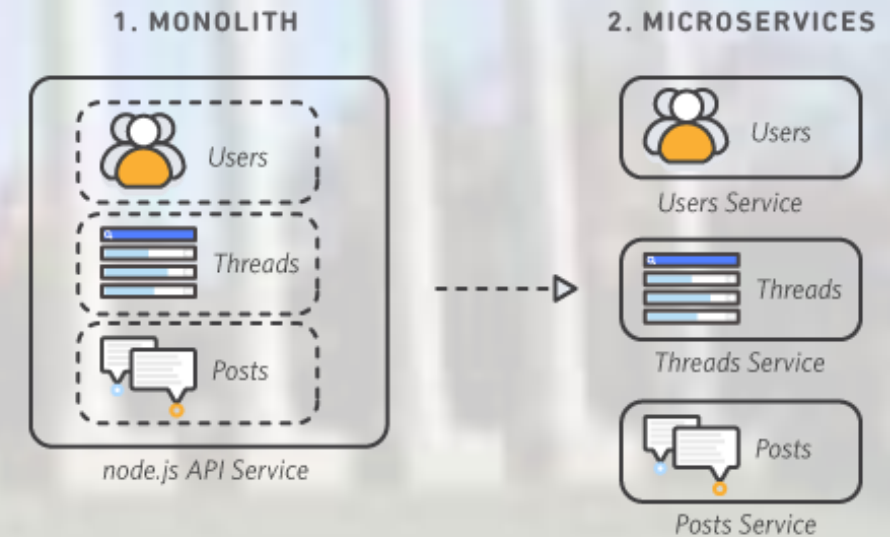
- ▶ We configure GitLab to utilize CI/CD and the Container Registry to automatically build and push a container each time we push a new commit to the repository
- ▶ Each commit has its own tag, and latest is automatically updated to point to the most recent commit
- ▶ Let's view the repository, the Gitlab CI config, and the container registry!



<https://gitlab.nrp-nautilus.io/jhurt/python-ci>

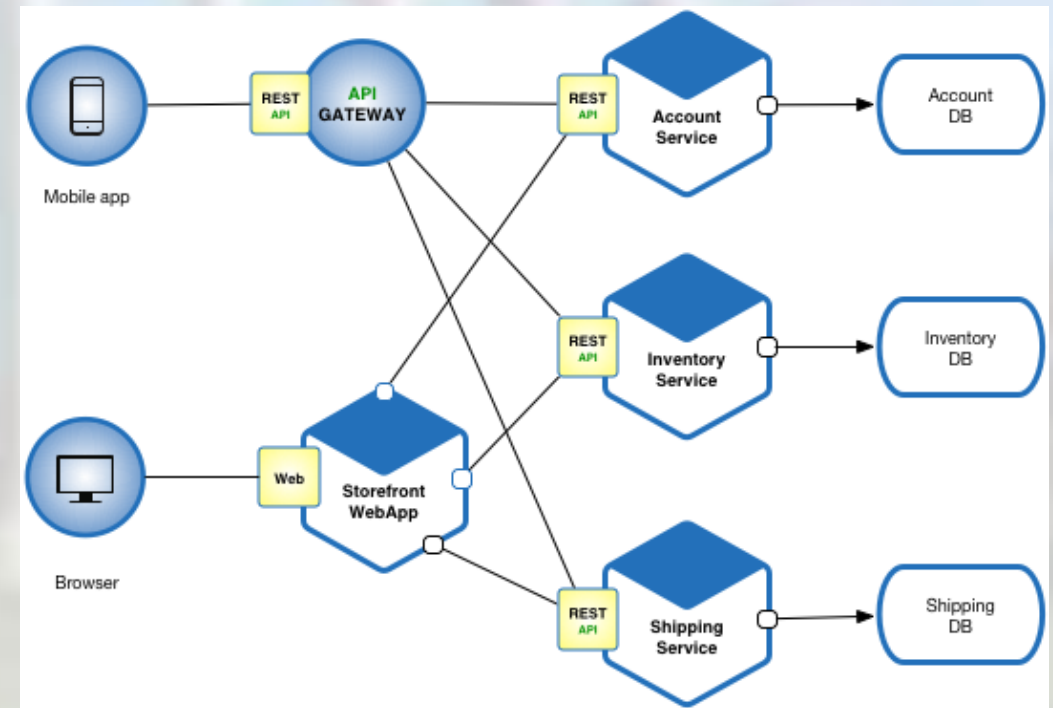
Microservices

- ▶ What are microservices?
 - ▶ Microservices are an architectural and/or organizational approach to software development where software is composed of small independent services that communicate over well-defined APIs
- ▶ Why do we want to use microservices?
 - ▶ Each service can be owned by self-contained teams or developers
 - ▶ Microservices architectures make applications easier to scale and faster to develop
 - ▶ Enable innovation and accelerate development of new features



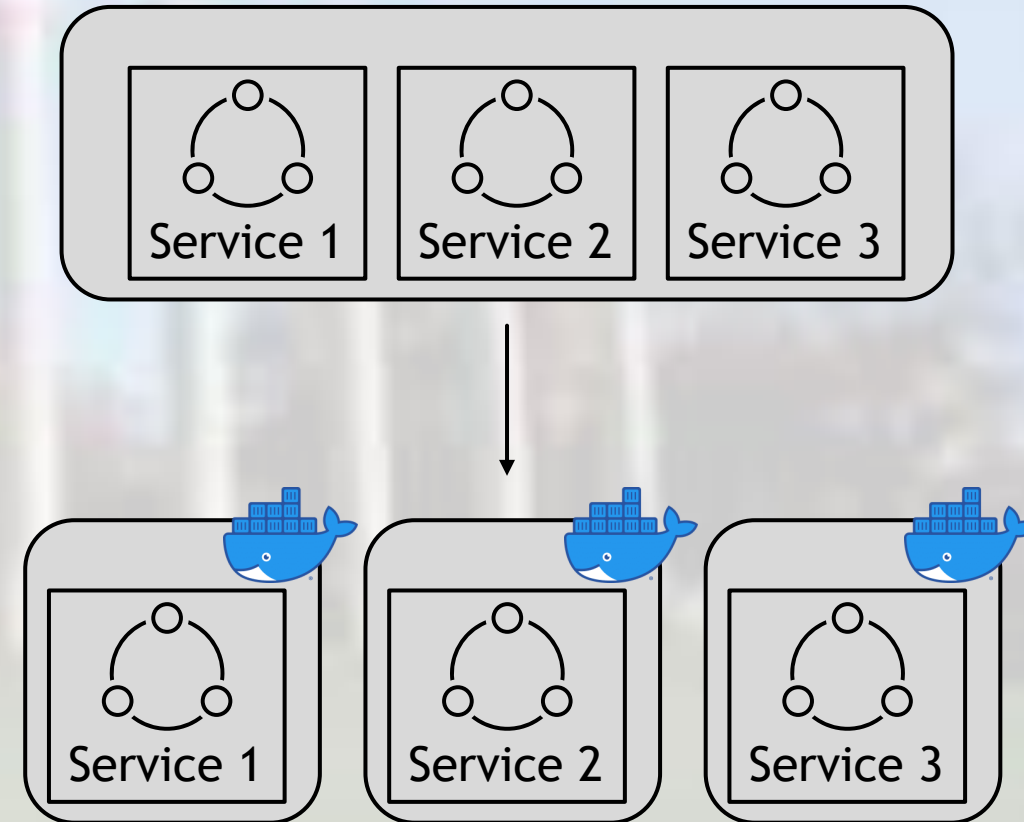
Example Microservice Architecture: E-Commerce Web Application

- ▶ Each service:
 - ▶ Runs independently and interacts with other services via APIs
 - ▶ Bears responsibility for different aspects and features of the overall application
 - ▶ Developed, tested, and deployed by different developers/teams
- ▶ Microservices are loosely coupled
 - ▶ Must be capable of interacting with the other microservices in the overall architecture



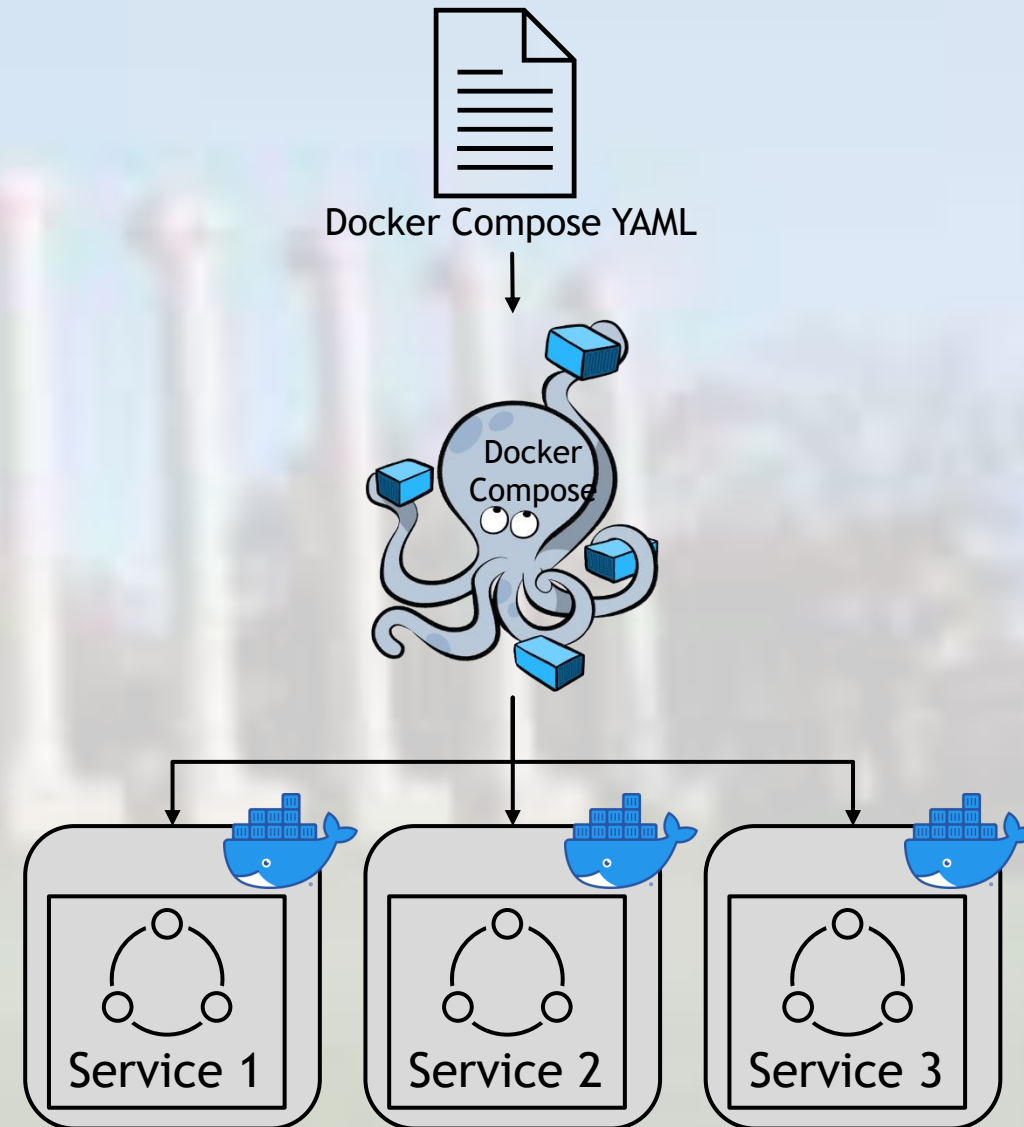
Microservices in Docker

- ▶ Each microservice in an application has its own system dependencies, software dependencies and source code → Docker
- ▶ By packaging microservices into Docker containers, we create a development, testing and deployment cycle that is:
 - ▶ Efficient
 - ▶ Reproducible
 - ▶ Portable



Utilizing Multiple Containers: Docker Compose

- ▶ Microservices and other software design paradigms can be developed and even deployed using multiple docker containers
- ▶ One common use case for this is a client-server model, such as a web server
- ▶ Docker compose is a tool that can enable robust management of multiple containers using YAML specified configuration



Multi-Stage Builds

- ▶ In some use-cases, we need to change the base container of our image but keep much of the work performed thus far in the image building process
- ▶ Alternatively, say there are libraries we need to build our library or application, but we do not need those libraries at runtime
- ▶ For both of these use-cases, we can utilize **multi-stage** builds to build multiple container images from a single Dockerfile

Multi-Stage Builds Example 1: Compiling a C++ Executable

- ▶ Suppose we have built an application in C++, and we want to distribute this application using Docker
- ▶ We need multiple build tools to generate the final executable, but we do not need those tools to be distributed with our final executable

```
FROM gcc:9 as build
RUN apt update && apt install -y cmake
RUN mkdir -p /app
COPY / /app

RUN mkdir -p /app/build
WORKDIR /app/build
RUN cmake .. && make

FROM gcc:9 as dist

RUN mkdir -p /workspace

COPY --from=build /app/build/myexecutable /workspace

CMD /workspace/myexecutable
```


Using Multi-Stage Builds

- Once we have built a Dockerfile utilizing a multi-stage build, we can specify which stage to build:

docker build

--file Dockerfile

--target dist

--tag myContainer:dist .

```
FROM gcc:9 as build
RUN apt update && apt install -y cmake
RUN mkdir -p /app
COPY / /app

RUN mkdir -p /app/build
WORKDIR /app/build
RUN cmake .. && make

FROM gcc:9 as dist

RUN mkdir -p /workspace

COPY --from=build /app/build/myexecutable /workspace

CMD /workspace/myexecutable
```

Multi-Stage Builds Example 2: Development vs. Production

- ▶ We may want to build a development container with additional tools to help us during the development process that we do not want to deploy to production
- ▶ We can use a multi-stage build to install only what we need for each
- ▶ We can again utilize the `--target` parameter to docker build to specify which stage to build.
- ▶ Multi-stage builds can be further extended to CI/CD and automation pipelines and/or docker-compose architectures and microservices

```
FROM jupyter/datascience-notebook:python-3.10.9 as dev
```

```
RUN mkdir -p /app
```

```
COPY / /app
```

```
RUN pip install /app/requirements.txt
```

```
CMD /bin/bash
```

```
FROM python:3.10 as prod
```

```
COPY --from=dev /app /app
```




















```
RUN pip install /app/requirements.txt
```

```
CMD /app/myscript.py
```

Docker Containers in the Cloud

- ▶ Containerization can be used to ensure portability to many kinds of computing environments, including cloud computing
- ▶ All major commercial cloud platforms offer services to deploy containers:
 - ▶ Amazon Web Services (AWS)
 - ▶ Google Cloud Platform (GCP)
 - ▶ Microsoft Azure

AWS Containers services

Sub-category	Use case	AWS service
Container orchestration	Run containerized applications or build microservices	 Amazon Elastic Container Service (ECS)
	Manage containers with Kubernetes	 Amazon Elastic Kubernetes Service (EKS)
Compute options	Run containers without managing servers	 AWS Fargate
	Run containers with server-level control	 Amazon Elastic Compute Cloud (EC2)
	Run fault-tolerant workloads for up to 90 percent off	 Amazon EC2 Spot Instances
Tools & services with containers support	Quickly launch and manage containerized applications	 AWS Copilot
	Share and deploy container software, publicly or privately	 Amazon Elastic Container Registry (ECR)
	Application-level networking for all your services	 AWS App Mesh
	Cloud resource discovery service	 AWS Cloud Map
	Package and deploy Lambda functions as container images	 AWS Lambda
	Build and run containerized applications on a fully managed service	 AWS App Runner
	Run simple containerized applications for a fixed, monthly price	 Amazon Lightsail
	Containerize and migrate existing applications	 AWS App2Container
On-premises	Run containers on customer-managed infrastructure	 Amazon ECS Anywhere
	Create and operate Kubernetes clusters on your own infrastructure	 Amazon EKS Anywhere
Enterprise-scale container management	Automated management for container and serverless deployments	 AWS Proton
	A fully managed, turnkey app platform	 Red Hat OpenShift Service on AWS (ROSA)
Open-source	Run the Kubernetes distribution that powers Amazon EKS	 Amazon EKS Distro
	Containerize and migrate existing applications	 AWS App2Container



MUAMLL/SDSS2025

- ▶ NRP Portal

<https://portal.nrp-nautilus.io>

- ▶ Coder Instance:

<https://user-dev.nrp-nautilus.io/>

- ▶ Tutorial Repository:

<https://github.com/MUAMLL/SDSS2025>

- ▶ Git Clone Command:

```
git clone https://github.com/MUAMLL/SDSS2025.git
```