

# Do Now:

- ▶ Sign into “Coder” - <http://user-dev.nrp-nautilus.io/>
  - ▶ Link is available in the Attendee Guide
  - ▶ If you signed in, put up a **yellow** sticky note
  - ▶ If you have an error or issue, put up a **red** sticky note

# GP-ENGINE - Migrating AI/ML workflows to Nautilus

Introduction to Containers and Kubernetes

Support provided by NSF OAC#2322218

January 2025

# Schedule - Day 1

- ▶ 1:00 - 1:30 -> Setup and Introduction
- ▶ 1:30 - 3:00 -> Introduction to Containers
- ▶ 3:00 - 3:20 -> Break
- ▶ 3:20 - 5:00 -> Continue Containers Lesson

# Schedule - Day 2

- ▶ 9:00 - 9:30 -> Arrive and Setup
- ▶ 9:30 - 10:45 -> Introduction to NRP
- ▶ 10:45 - 11:00 -> Break
- ▶ 11:00 - 12:00 -> Continue NRP Lesson
- ▶ 12:00 - 12:15 -> Future Directions and Q/A

# Learning Objectives

- ▶ Containerization and Docker
- ▶ Kubernetes Architecture and Concepts
- ▶ Hands-on with Nautilus HyperCluster
- ▶ Deploying Pods and Jobs in Kubernetes using Nautilus

# Workshop Outline

- ▶ Software Containerization with Docker
  - ▶ Containerization basics
  - ▶ Docker concepts: Pods, Jobs, etc.
  - ▶ Hands on with Docker
  - ▶ Advanced Concepts with Docker
- ▶ Container management with Kubernetes
  - ▶ Introduction to Kubernetes
  - ▶ Kubernetes concepts
  - ▶ Kubernetes usage
- ▶ NRP Nautilus HyperCluster
  - ▶ Cluster introduction, background and resources
  - ▶ Case Study: How MU utilizes Nautilus
  - ▶ Hands on with Kubernetes in Nautilus



# Part 1

# Software Containerization with Docker

Introduction to Containers and Kubernetes

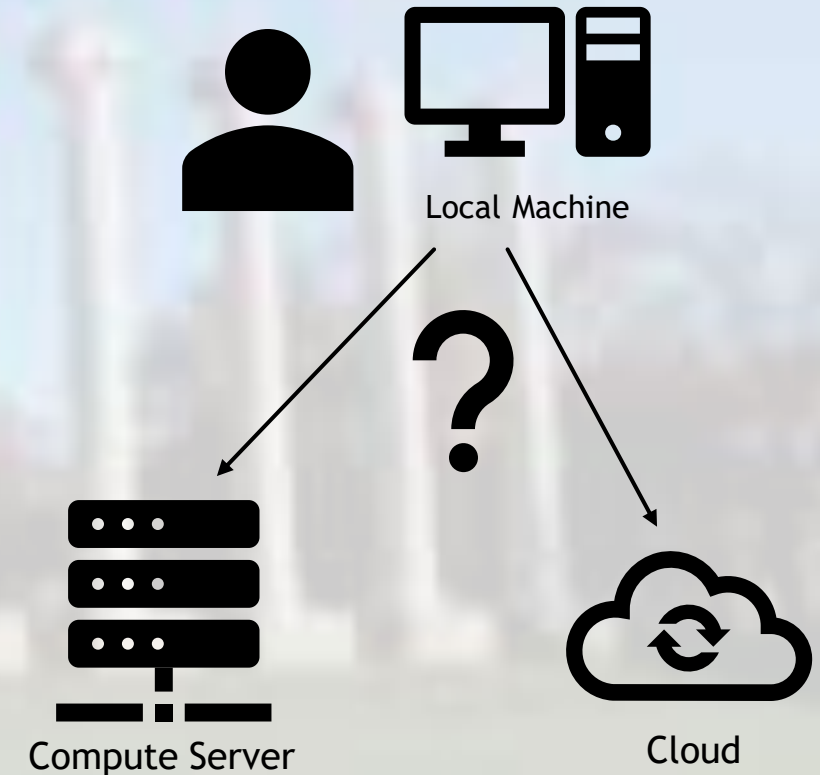
# Introduction

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



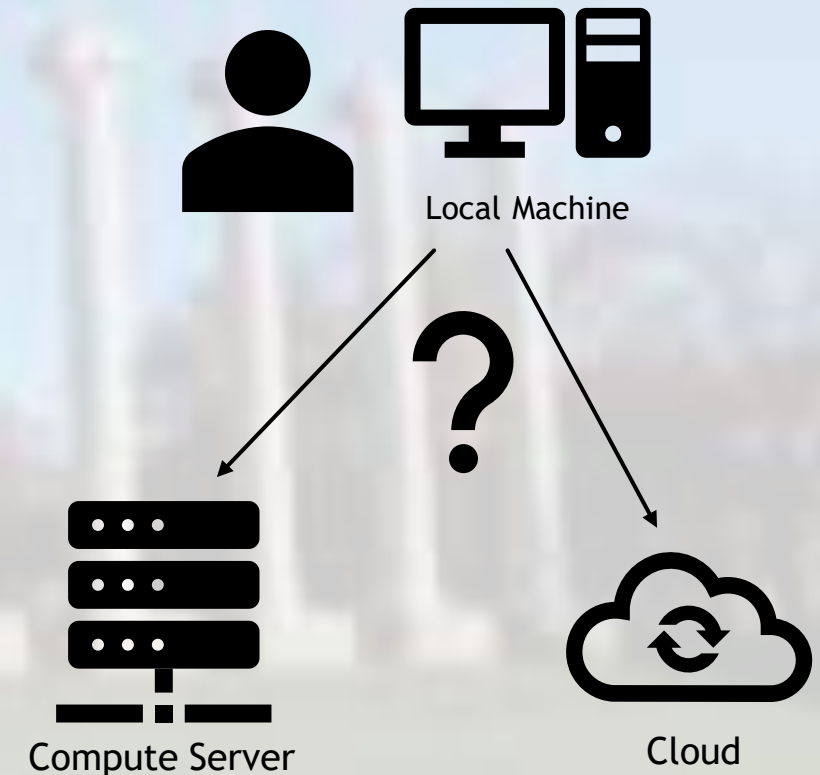
# The Problem: ???

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



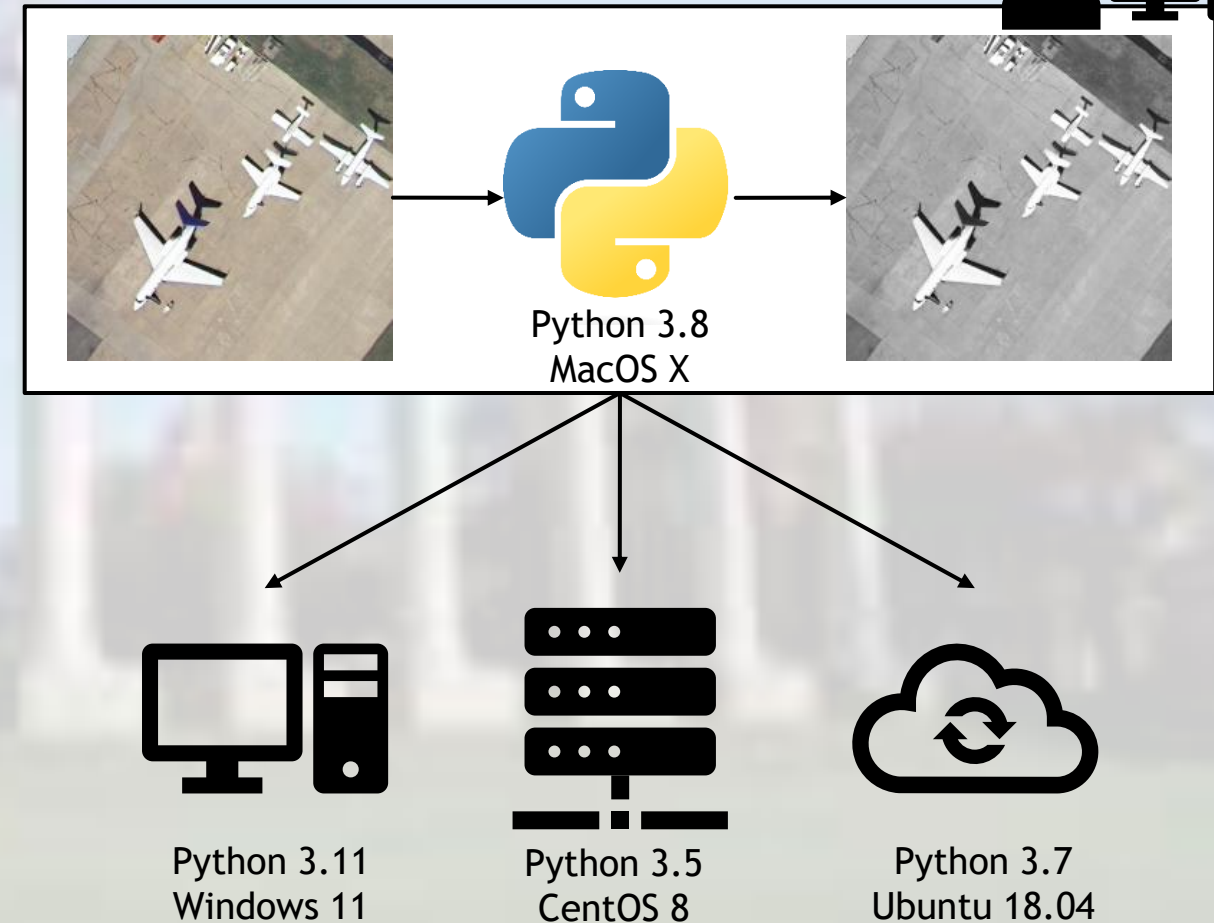
# The Problem: Scalability & Reproducibility

- ▶ 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?



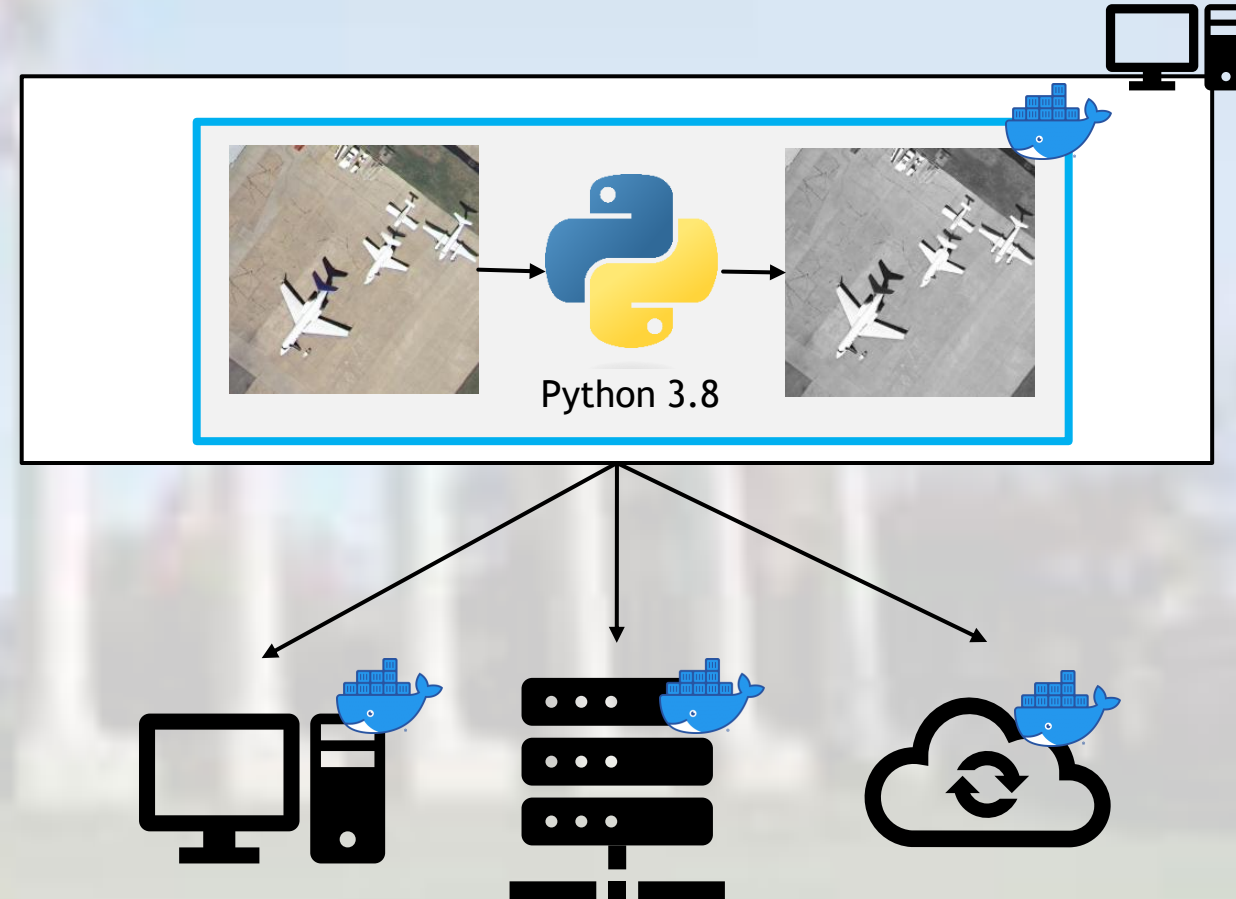
# Example: Python Application for Image Processing

- ▶ 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



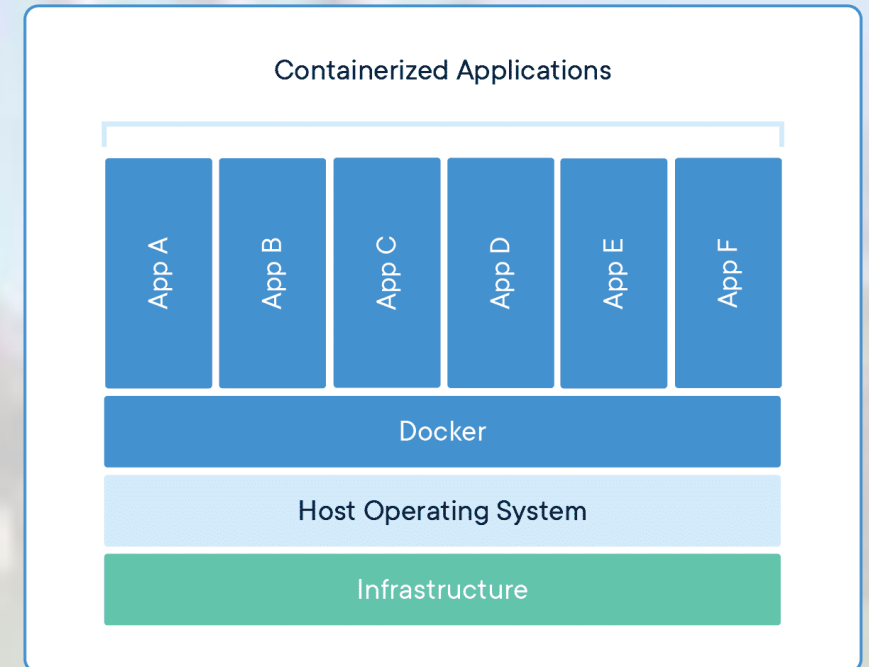
# Containerization

- 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





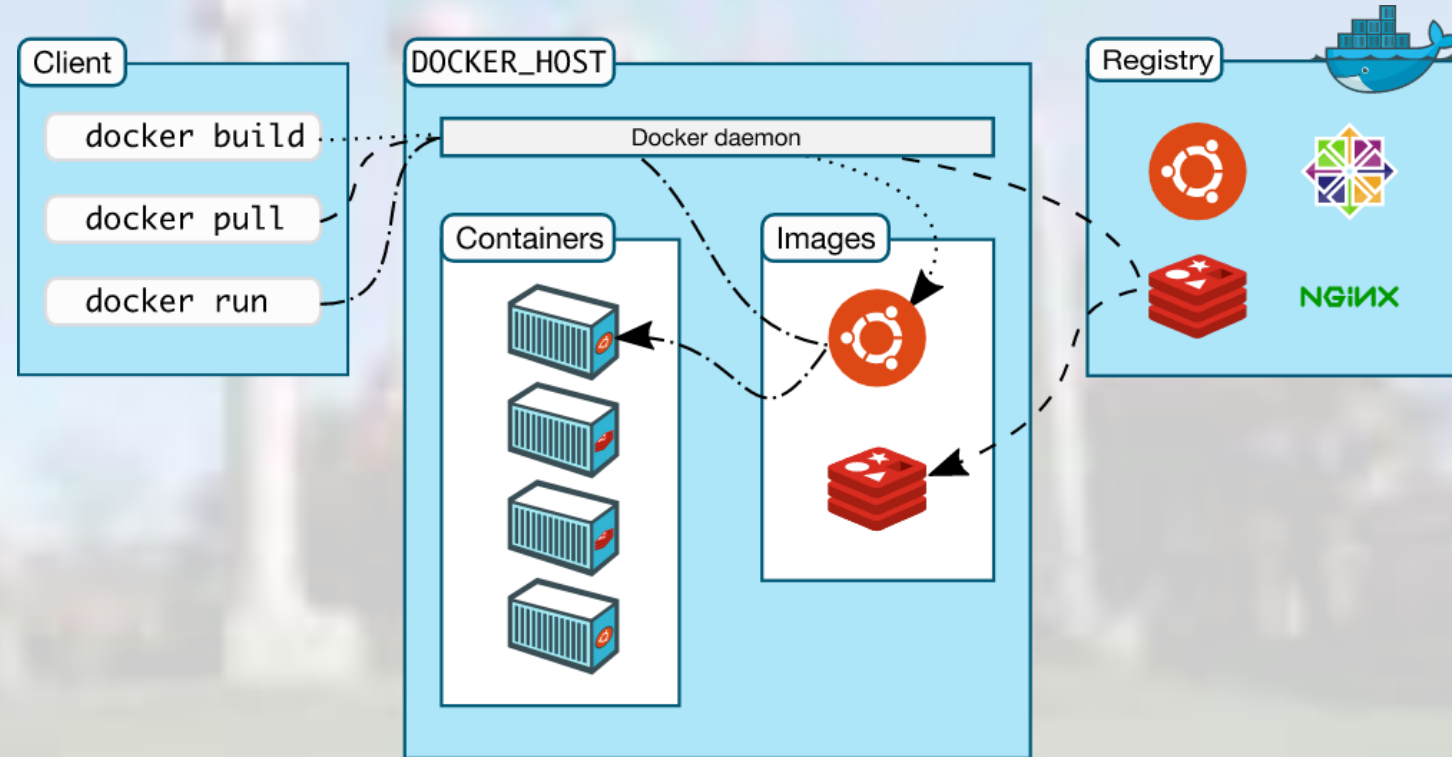
# Docker

## ► 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.<sup>1</sup>
- 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?

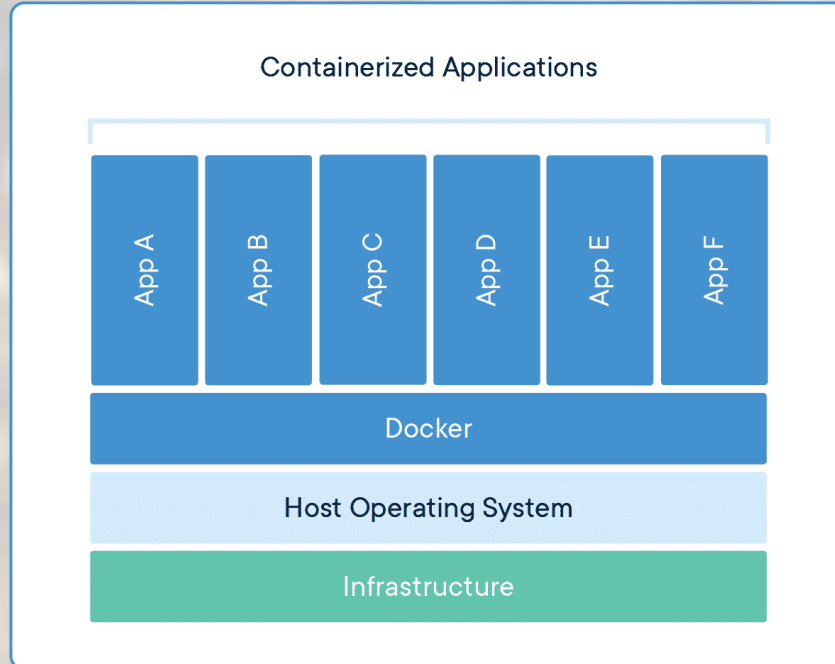
- Docker enables predictable and reliable deployment of software.
- Docker containers are portable!
  - local development computers, compute clusters, internal compute servers, cloud infrastructure, and more!
- *Docker containers enable reliable portability of software to nearly any compute environment*



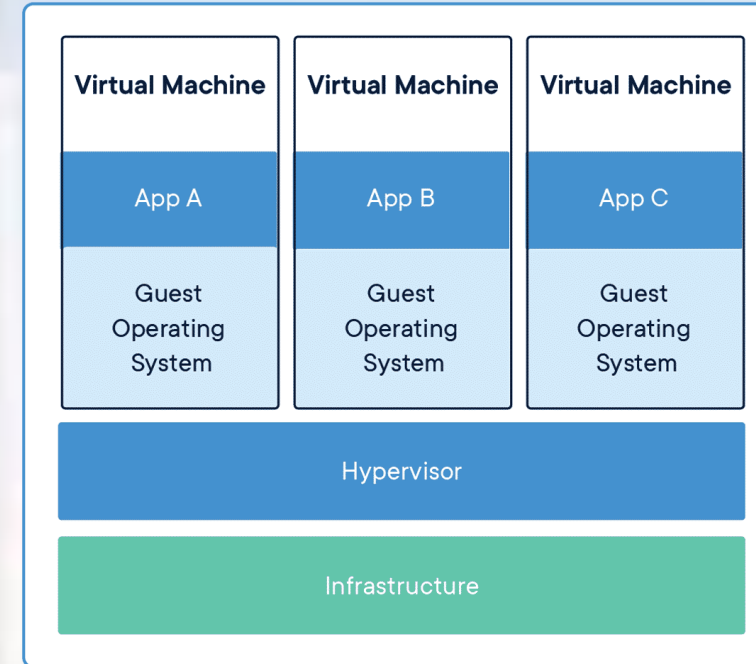
1. [https://en.wikipedia.org/wiki/Docker\\_\(software\)](https://en.wikipedia.org/wiki/Docker_(software))  
 2. Image: <https://docs.docker.com/get-started/overview/>



# 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.



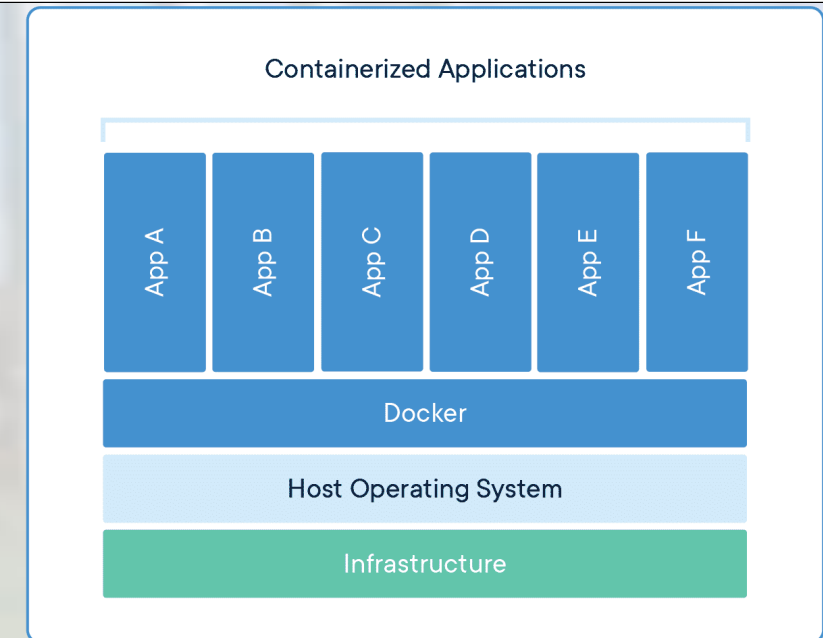
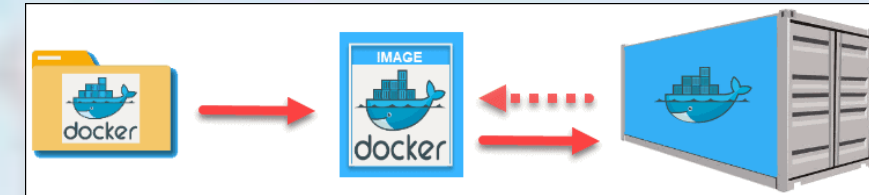
**Virtual machines (VMs)** are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries - taking up tens of GBs. VMs can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

# Docker Concepts

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

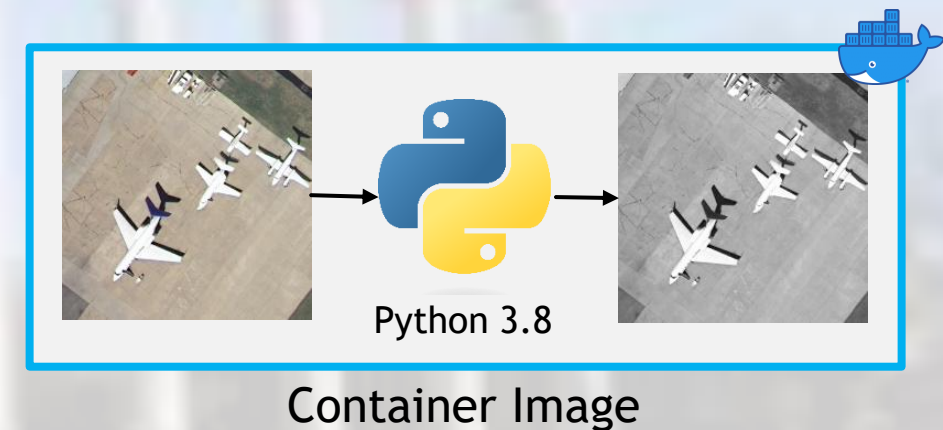
# Key Docker Concepts

- ▶ **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



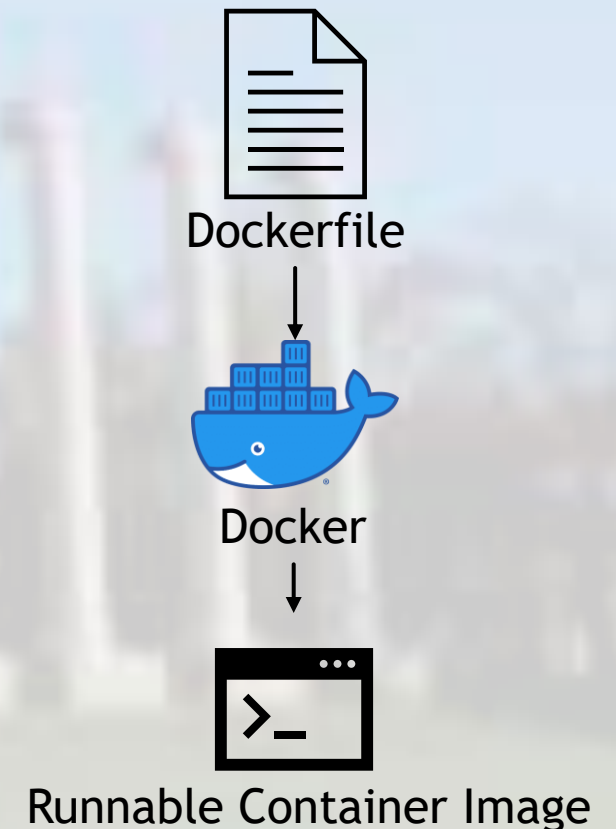
# Containers vs Container Images

- ▶ 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

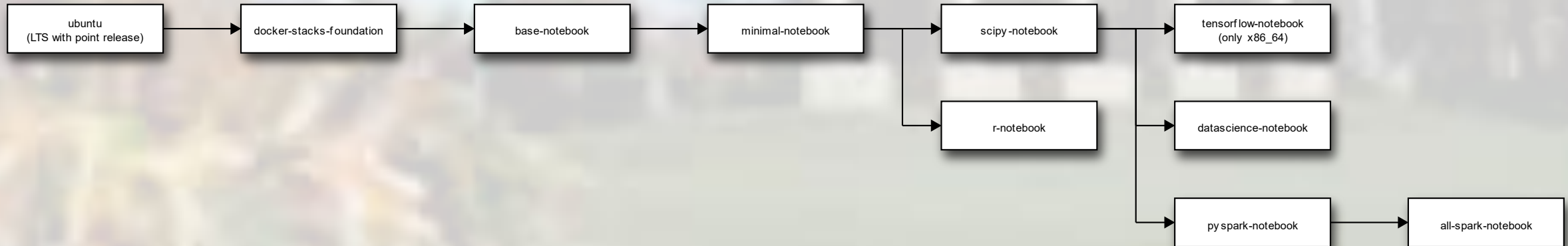
- ▶ Recall: Containers and container images enable reliable portability of software
- ▶ We need a way to build container images in a distributed and standardized way, so that anyone with a container runtime can build and run our software package (container image)
- ▶ Dockerfiles are the template, or 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





# 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 Images<sup>1</sup>

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



# 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

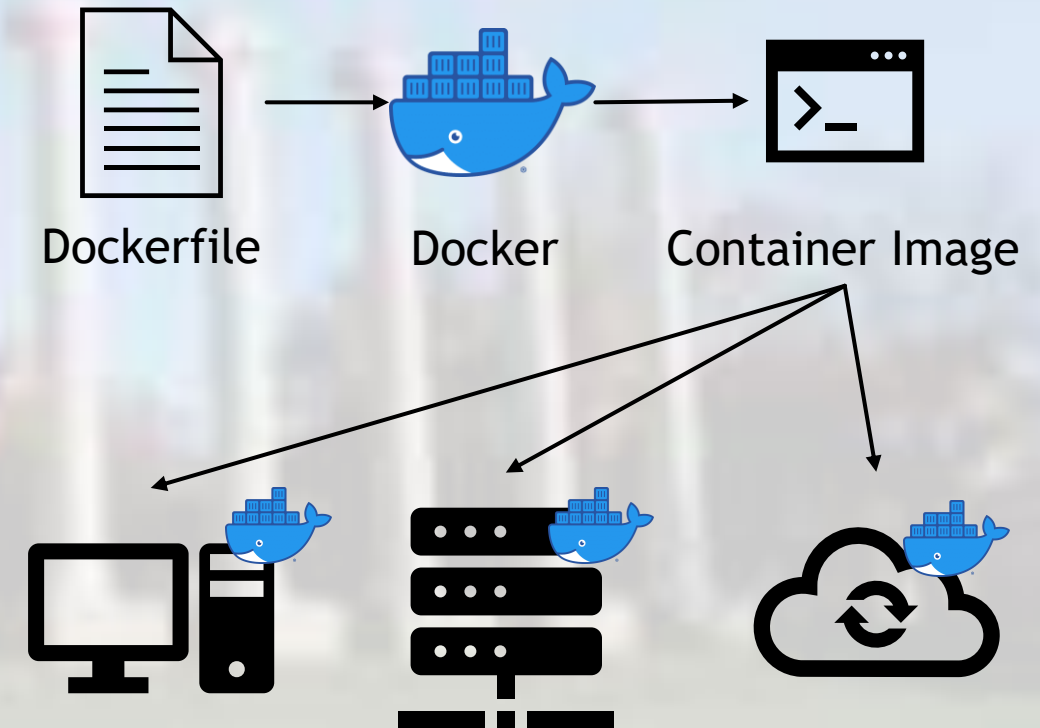
Use pip to install the requirements defined in requirements.txt

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

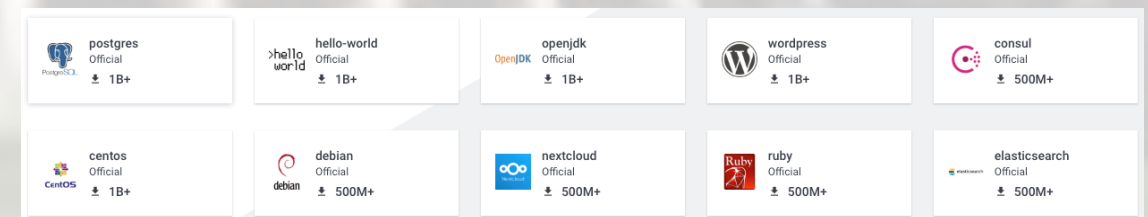
# Distributing and Sharing Containers

- ▶ 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

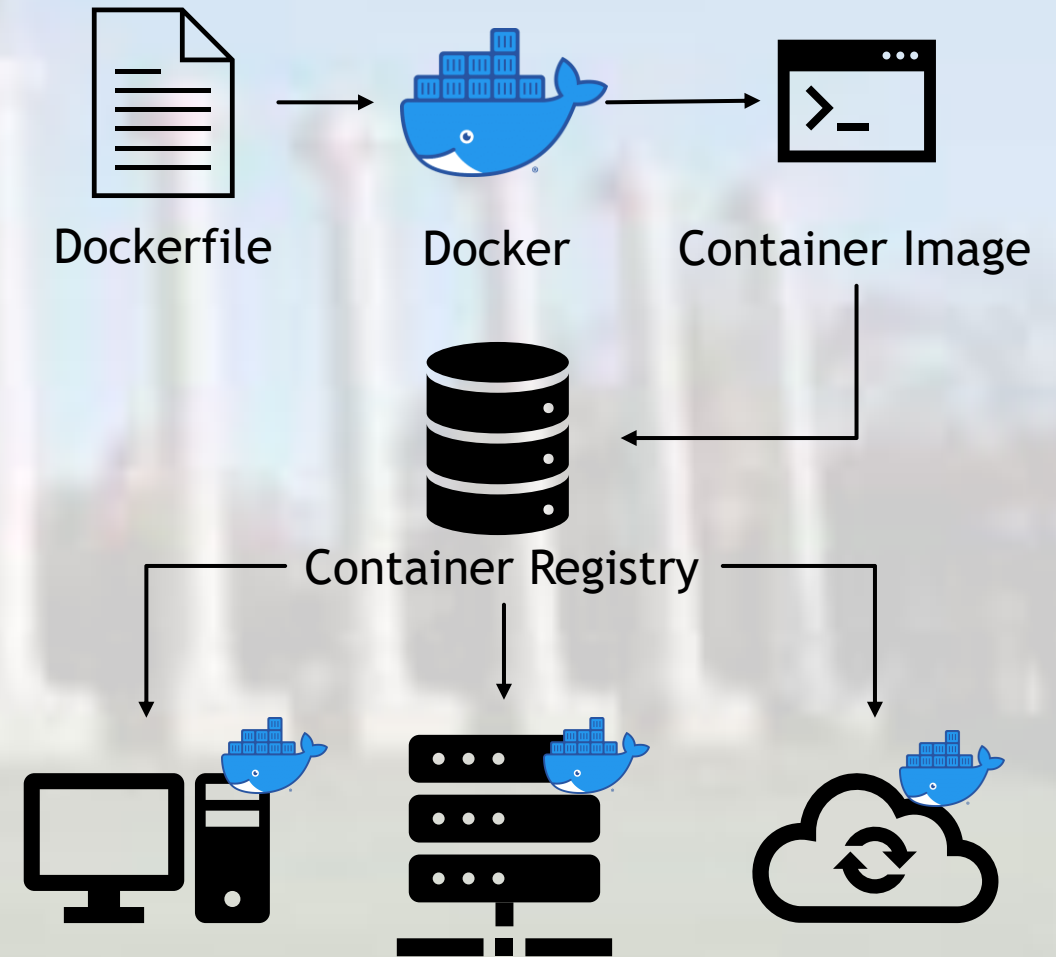
- ▶ Container Registries are web-enabled storage locations for Docker container images
  - ▶ Similar to Google Drive for documents and spreadsheets
- ▶ Each image published on a registry contains a name and a tag:
  - ▶ python:3.8 → Python is the name of image and 3.8 is the tag
- ▶ If a URL is not specified, 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

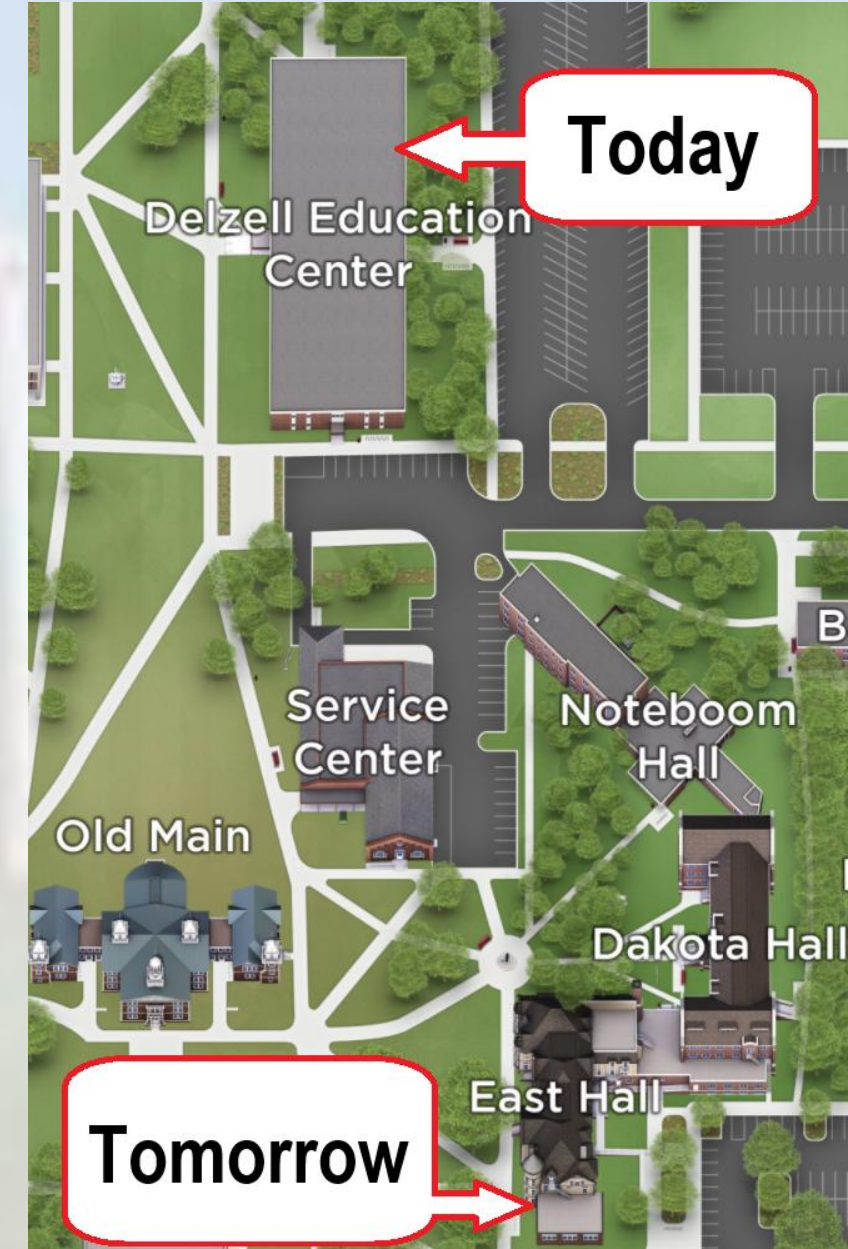
- ▶ We can publish our container image to a registry, and then pull down our container image in each computing environment in which we want to use it
- ▶ We can utilize public registries, like Docker Hub, or private ones, to control who we allow to pull down our container





# REMINDER!!!

## We are in a different building tomorrow!



## Do Now:

- ▶ Sign into “Jupyter” - <http://gp-engine.nrp-nautilus.io/>
- ▶ Sign into “Nautilus” - <https://portal.nrp-nautilus.io/>
- ▶ Links are available in the Attendee Guide
  - ▶ If you signed in, put up a **yellow** sticky note
  - ▶ If you have an error or issue, put up a **red** sticky note

# Part 2

# Container management with Kubernetes

Introduction to Containers and Kubernetes

# Subsection Outline

- ▶ Introduction to Kubernetes
- ▶ Kubernetes concepts
- ▶ Kubernetes usage
- ▶ Kubernetes hands on

# Introduction to Kubernetes

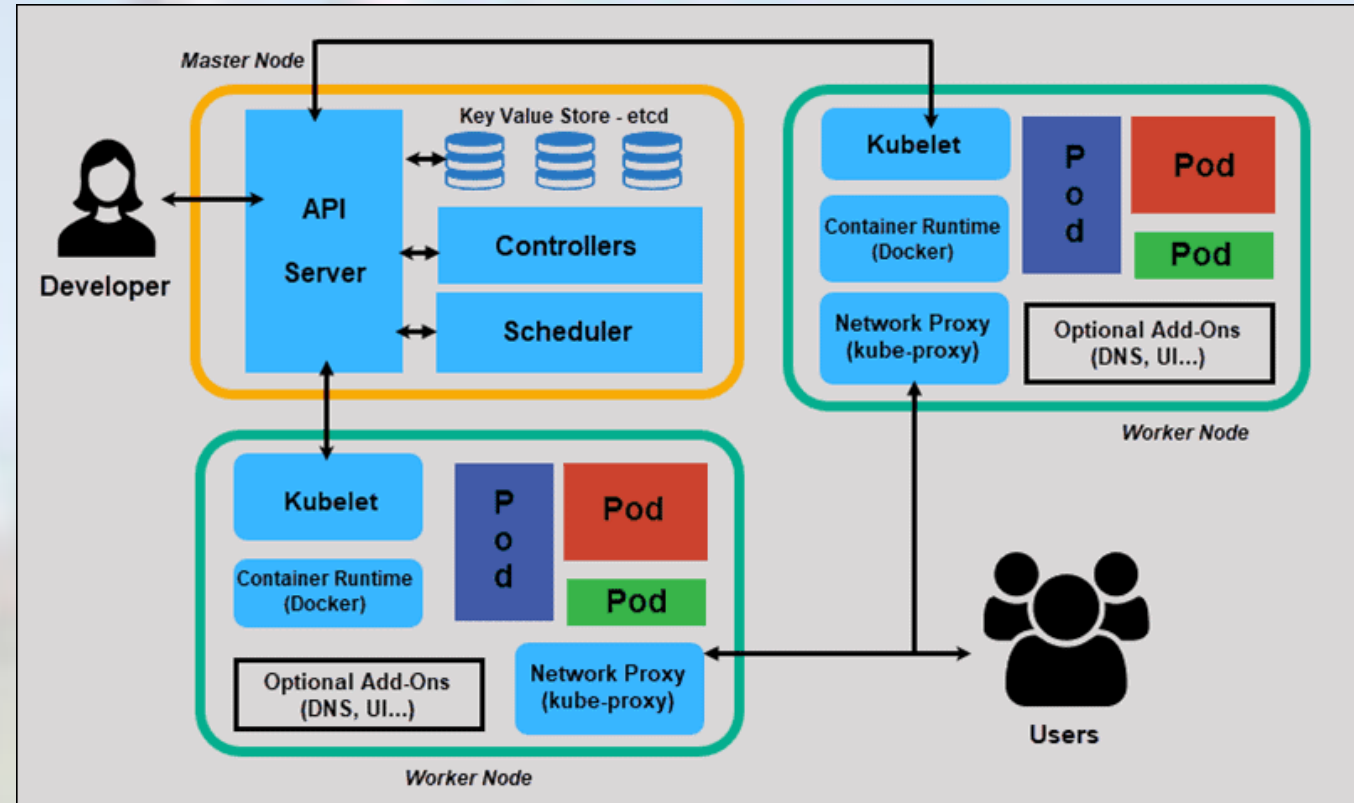
What is Kubernetes and what is it used for



# Kubernetes



- ▶ Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.<sup>1</sup>
- ▶ Kubernetes enables both simple and complex container orchestration
- ▶ Kubernetes cluster has two main components
  - ▶ Master node
  - ▶ Worker node



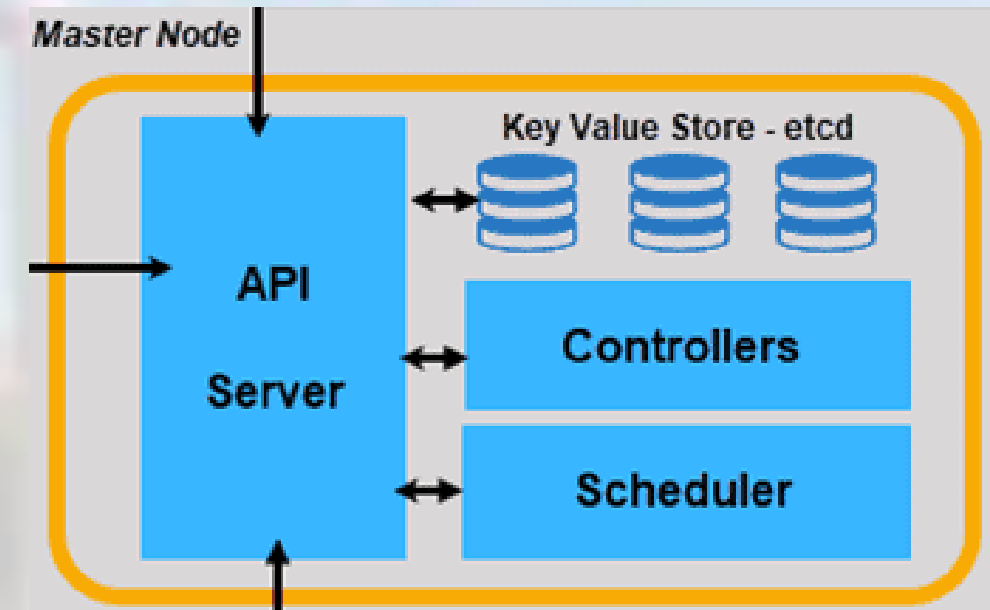
1. <https://kubernetes.io/>  
 2. Image: <https://phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams>  
 3. Logo: [https://commons.wikimedia.org/wiki/File:Kubernetes\\_logo\\_without\\_workmark.svg](https://commons.wikimedia.org/wiki/File:Kubernetes_logo_without_workmark.svg)



# Kubernetes

## Master node

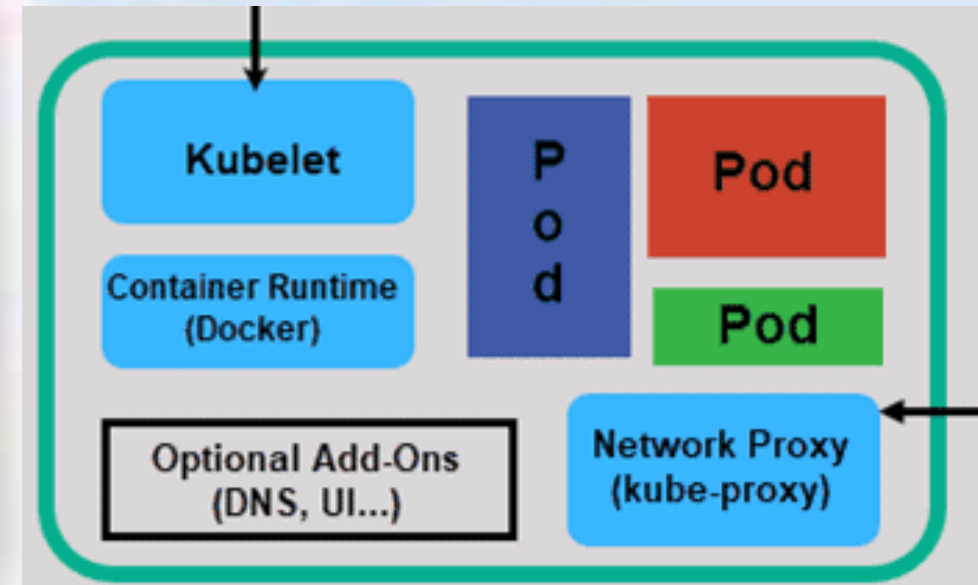
- ▶ Also known as the Control Plane, it is responsible for managing the state of the cluster
- ▶ API server: interface between master node and the rest of the cluster
- ▶ etcd: distributed key-value store that stores the cluster's persistent information
- ▶ Scheduler: responsible for scheduling pods onto the working nodes
- ▶ Controller manager: responsible for running controllers that manages the state of the clusters such as replication controller and deployment controller



# Kubernetes

## Worker node

- ▶ The physical machine where the operations takes place, it can run one or multiple pods
- ▶ Kubelet: daemon that runs each working node
- ▶ Container runtime: is responsible for pulling images from the registry, starting and stopping containers, and managing the container resources
- ▶ kube-proxy: responsible for routing traffic to the correct pod and provides load balancing so that the traffic is distributed evenly between the pods



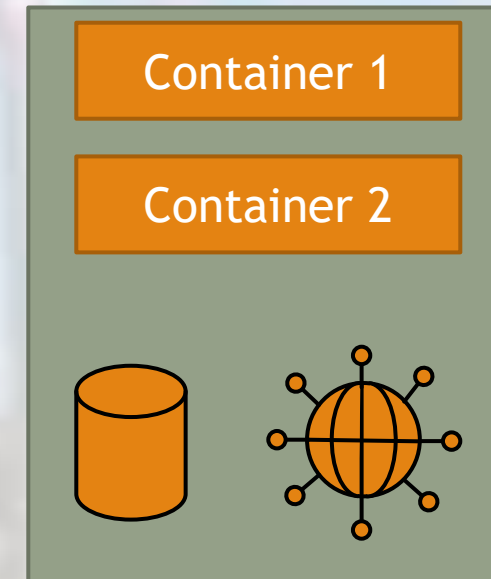
# Kubernetes concepts

Node, pod, persistent volume, job, deployment, service

# Key Kubernetes Concepts

## Pod

- ▶ Pods are the basic scheduling unit of K8s.
- ▶ Pods consist of one or more containers running inside. Each pod has a unique IP address to enable micro services or applications
- ▶ Pods can run custom scripts (initcontainer) at runtime to initialize the pod
- ▶ Pods generally have limitations on allocated resources and max runtime
- ▶ Pods are stateless, meaning all data uploaded or generated by the pod is deleted when the pod terminates



# Key Kubernetes Concepts

## ReplicaSet and Deployment

- ▶ **ReplicaSet** - its purpose is to maintain a stable set of replica Pods running at any given time.<sup>1</sup>
- ▶ **Deployment** - is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features.<sup>1</sup>



It is recommended to use  
Deployment instead of ReplicaSets

1. <https://kubernetes.io/>



# Key Kubernetes Concepts

## Jobs

- ▶ A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate.<sup>1</sup>
- ▶ A job has virtually access to unlimited resources and can run for extended periods of time
- ▶ A job may consist of one pod or multiple pods working in parallel
- ▶ Deleting a job will automatically delete its corresponding pod
- ▶ A job can create a new pod(s) if any of its pod(s) is deleted or failed for any reason.
- ▶ Similar to pods, jobs are stateless

1. <https://kubernetes.io/>

# Key Kubernetes Concepts

## Persistent volume

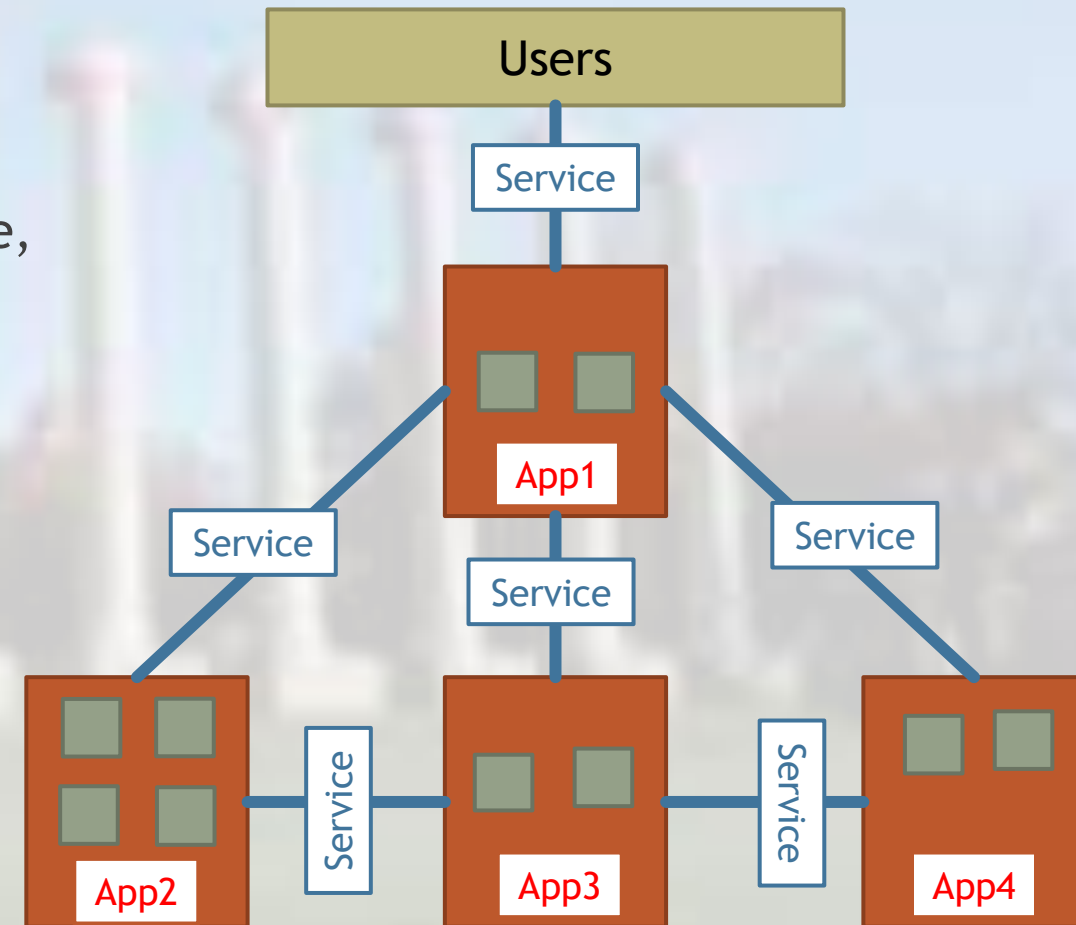
- ▶ To maintain the data generated a persistent volume (storage) is needed
- ▶ A **persistent volume** is storage on the cluster that has been provisioned by an administrator or dynamically provisioned using storage classes.<sup>1</sup>
- ▶ There exists different classes of persistent volumes such as:
  - ▶ cephfs
  - ▶ Fibre Channel storage
  - ▶ NFS storage
- ▶ There are different access modes:
  - ▶ ReadWriteOnce
  - ▶ ReadOnlyMany
  - ▶ ReadWriteMany
  - ▶ ReadWriteOncePod

1. <https://kubernetes.io/>

# Key Kubernetes Concepts

## Services

- ▶ Applications running within distinct pods communicate over the network using the unique IP address assigned to each pod
- ▶ Each Pod has a unique IP address assigned at runtime, which changes every time a Pod is restarted, making reliable communication less simple
- ▶ Services enable communication between applications running in pods within the cluster and with outside users if necessary
- ▶ There are four type of services supported by K8s
  - ▶ ClusterIP
  - ▶ NodePort
  - ▶ LoadBalancer
  - ▶ Ingress



# Kubernetes usage

Basics of YAML language, kubectl installation and usage

# Yet Another Markup Language (YAML)

XML	JSON	YAML
<pre>&lt;Servers&gt;   &lt;Server&gt;     &lt;name&gt;Server1&lt;/name&gt;     &lt;owner&gt;John&lt;/owner&gt;     &lt;created&gt;123456&lt;/created&gt;     &lt;status&gt;active&lt;/status&gt;   &lt;/Server&gt; &lt;/Servers&gt;</pre>	<pre>{   Servers: [     {       name: Server1,       owner: John,       created: 123456,       status: active     }   ] }</pre>	<pre>Servers: -   name: Server1     owner: John     created: 123456     status: active</pre>

- ▶ YAML is a key-value pair file format, similar to JSON and XML
- ▶ Kubernetes operations are performed using **YAML** files, known as a Spec file
  - ▶ Creating Persistent Storage
  - ▶ Creating Pods
  - ▶ Creating Jobs
  - ▶ Deploying services



# Interfacing with Kubernetes: KubeCTL

- ▶ With a published Docker image and prepared YAML Spec file, KubeCTL enables interaction with Kubernetes:

```
kubectl [command] [TYPE] [NAME] [flags]
```

where:

- ▶ **command:** Specifies the operation that you want to perform on one or more resources, for example create, get, describe, delete
- ▶ **TYPE:** Specifies the resource type, such as pod or job
- ▶ **NAME:** Specifies the name of the resource, or the path to a Spec file
- ▶ **flags:** Specifies optional flags, such as --server to specify the address and port of the API server

# Interfacing with Kubernetes: KubeCTL cheat sheet

## ► To create pod

```
kubectl create -f pod.yaml
```

```
kubectl apply -f pod.yaml
```

## ► To create job

```
kubectl create -f job.yaml
```

```
kubectl apply -f job.yaml
```

# Interfacing with Kubernetes: KubeCTL cheat sheet

## ► To check pod status

```
kubectl get pods
```

```
kubectl describe pod pod-name
```

## ► To check job status

```
kubectl get jobs
```

```
kubectl describe job job-name
```

## ► To debug pod

```
kubectl logs pod-name
```

# Interfacing with Kubernetes: KubeCTL cheat sheet

## ▶ Access Pod interactively

```
kubectl exec -it pod-name -- /bin/bash
```

## ▶ Copy data from Nautilus to local machine

```
kubectl cp pod-name:path/to/data local/path/
```

## ▶ Copy data to Nautilus from local machine

```
kubectl cp local/path/ pod-name:path/to/data
```

## ▶ Exit interactive Pod mode

Press ctrl+D

# Interfacing with Kubernetes: KubeCTL cheat sheet

## ► To delete pod

```
kubectrl delete -f pod.yaml
```

```
Kubectrl delete pod-name
```

## ► To delete job

```
kubectrl delete -f job.yaml
```

```
Kubectrl delete job-name
```



# Interfacing with Kubernetes: KubeCTL cheat sheet

## ▶ To create persistent volume

```
kubectl create -f pvc.yaml
```

```
kubectl apply -f pvc.yaml
```

## ▶ To increase the size of persistent volume

```
kubectl apply -f pvc.yaml
```

## ▶ To delete persistent volume

```
kubectl delete -f pvc.yaml
```

```
kubectl delete pvc-name
```

# Part 3

## National Research Platform

### Nautilus Research Cluster

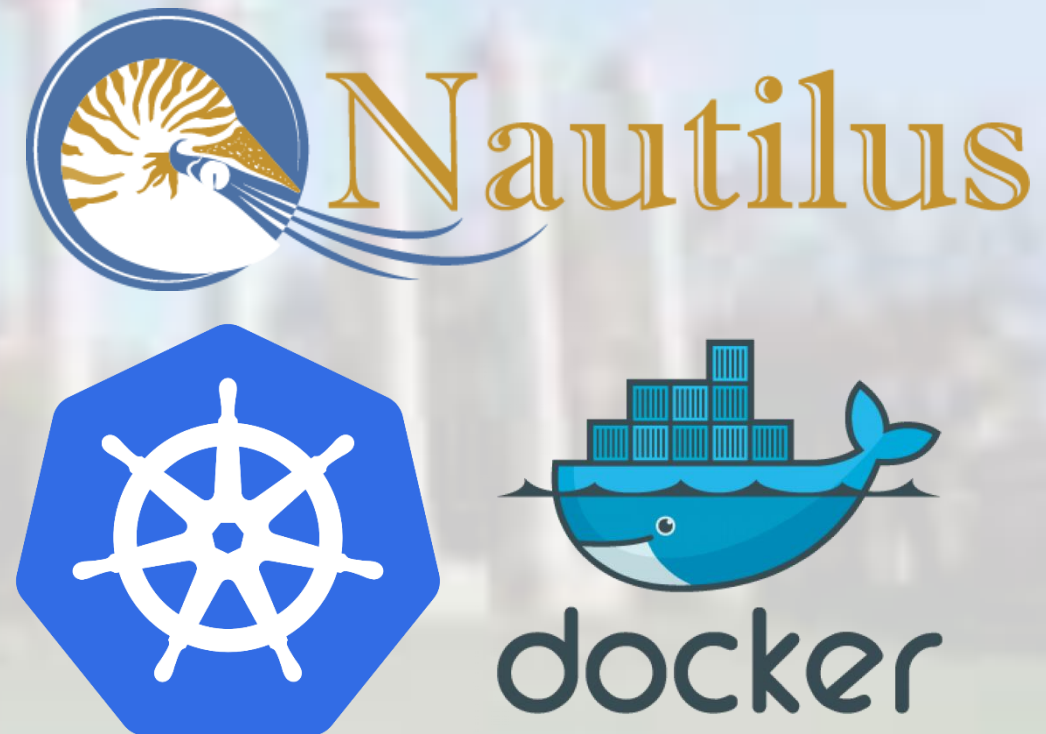
Introduction to Containers and Kubernetes

# 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 a 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



# Access to Nautilus

## Nautilus Access

### MU-managed Jupyter Hub

<https://gp-engine.nrp-nautilus.io/>

#### Advantages

No manual account  
creation

Ease of use

#### Disadvantages

Limited customizability

### Direct Access with KubeCTL

#### Disadvantages

Account and namespace  
creation

KubeCTL installation

#### Advantages

High scalability and  
customizability



# Access to Nautilus

- ▶ Follow the steps in getting started
  - ▶ <https://ucsd-prp.gitlab.io/userdocs/start/get-access/>
- ▶ Step1: Access Nautilus portal at <https://portal.nrp-nautilus.io>
- ▶ Step 2: Click on login



Namespaces overview Resources **Login**

## NRP Kubernetes portal

Here you can get an account in National Research Platform kubernetes portal by logging in with your university's credentials and requesting access in [matrix]

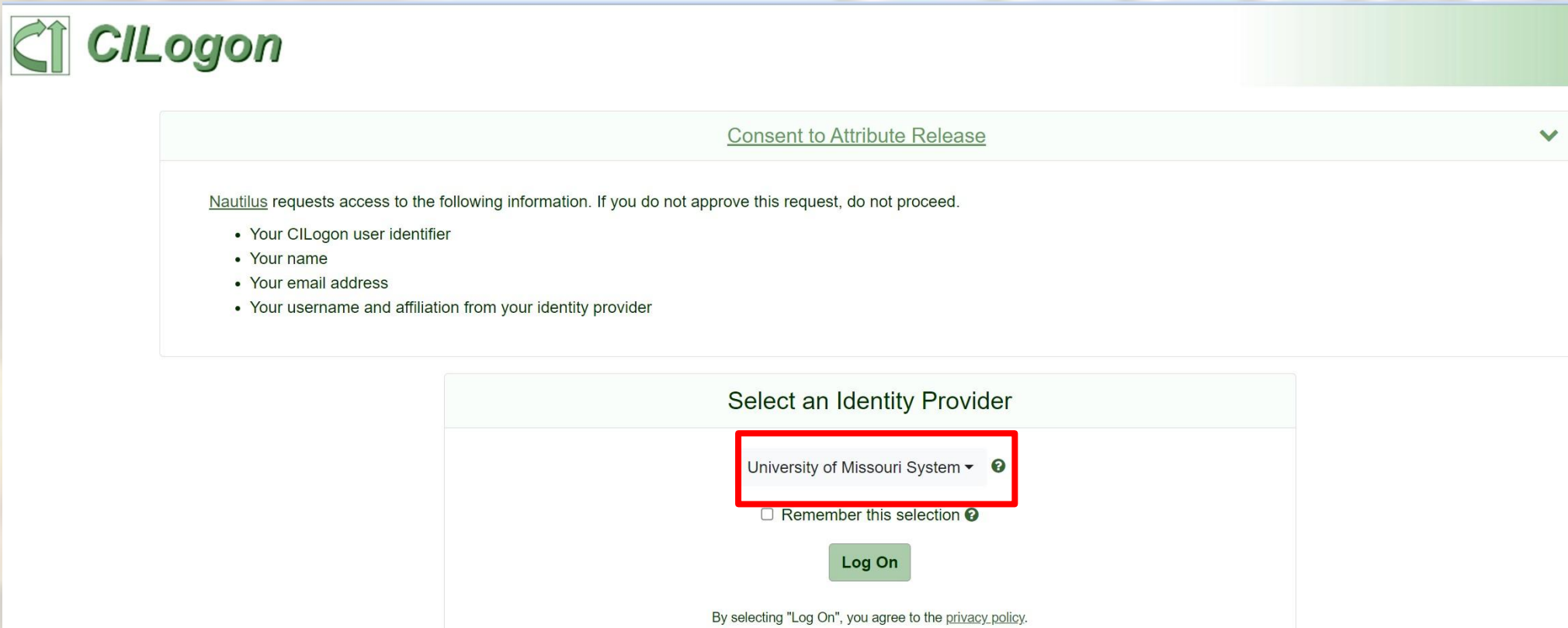
Documentation: <https://docs.nationalresearchplatform.org/>

You can easily join your node in our cluster - request instructions in [matrix] #general channel.

The National Research Platform currently has no storage that is suitable for HIPAA, PID, FISMA, FERPA, or protected data of any kind. Users are not permitted to store such data on NRP machines.

# Access to Nautilus

- ▶ Follow the steps in getting started
  - ▶ <https://ucsd-prp.gitlab.io/userdocs/start/get-access/>
- ▶ Step 3: Select identity provider - Either your institution, ORCID, GitHub, or Google



**CILogon**

[Consent to Attribute Release](#) ▼

Nautilus requests access to the following information. If you do not approve this request, do not proceed.

- Your CILogon user identifier
- Your name
- Your email address
- Your username and affiliation from your identity provider

Select an Identity Provider

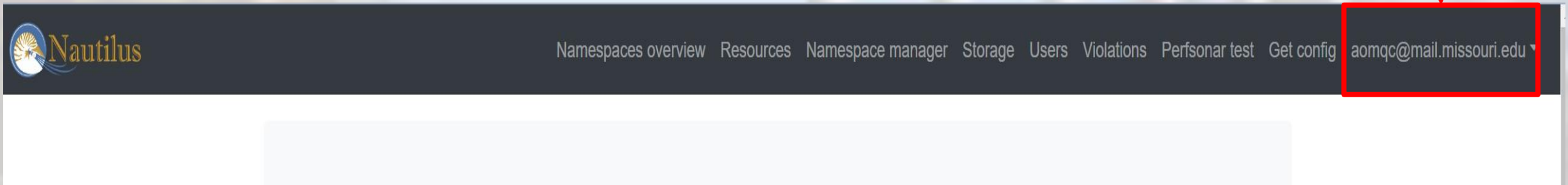
University of Missouri System ▼ ⓘ

☐ Remember this selection ⓘ

**Log On**

By selecting "Log On", you agree to the [privacy policy](#).

- ▶ Follow the steps in getting started
  - ▶ <https://ucsd-prp.gitlab.io/userdocs/start/get-access/>
- ▶ Step 4: Contact a Nautilus Namespace Admin
  - ▶ Email needs to be visible



- ▶ You need to be manually added to a namespace
  - ▶ As admins, we can add you to existing namespace or create a namespace for you

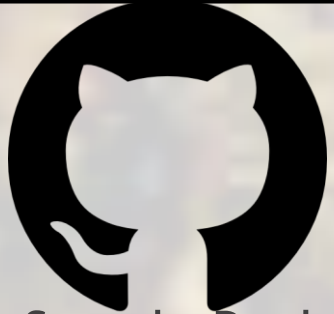
# Hands on Kubernetes

Life cycle of pods and jobs

# Hands on Kubernetes

- ▶ Demonstration of pod life cycle (without persistent volume)
  - ▶ Creation, interactive access, simple operation, closing, and deletion
- ▶ Creation of persistent volume
- ▶ Demonstration of pod life cycle (with persistent volume)
  - ▶ Creation, interactive access, simple operation, closing, and deletion
- ▶ Pod monitoring and debugging
- ▶ Demonstration of job life cycle





## MUAMLL/Nautilus

- ▶ Sample Dockerfiles
- ▶ Sample Kubernetes YAML File
- ▶ Wiki with detailed walkthroughs for:
  - ▶ Getting Started
  - ▶ Creating PVC
  - ▶ Creating Pods
  - ▶ Creating Jobs
- ▶ NRP Portal  
<https://portal.nrp-nautilus.io>
- ▶ JupyterHub Instance:  
<https://gp-engine.nrp-nautilus.io/>
- ▶ Tutorial Repository of Jupyter Project Pages, Code Samples, YAML, etc.  
<https://github.com/MUAMLL/gp-engine-tutorials>
- ▶ Git Clone Command:  

```
git clone https://github.com/MUAMLL/gp-engine-tutorials.git
```

**Over a short break, we will ensure everyone has cloned this Repo into their JupyterLab environment**