High-Performance Data-Intensive Computing Systems Laboratory

Docker and Kubernetes

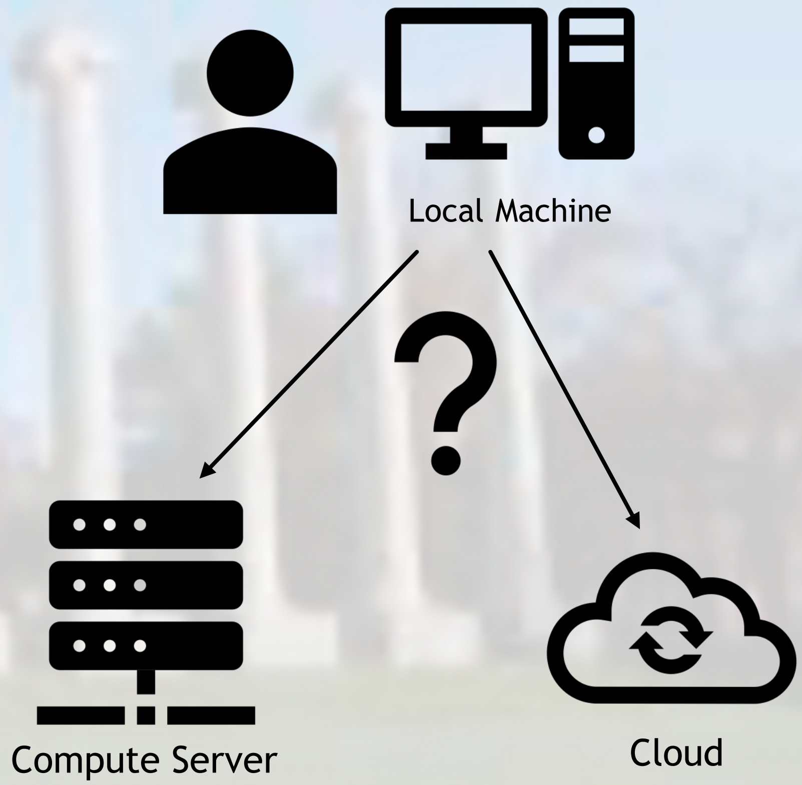
MORENet Technical Summit

20 Feb 2023

University of Missouri

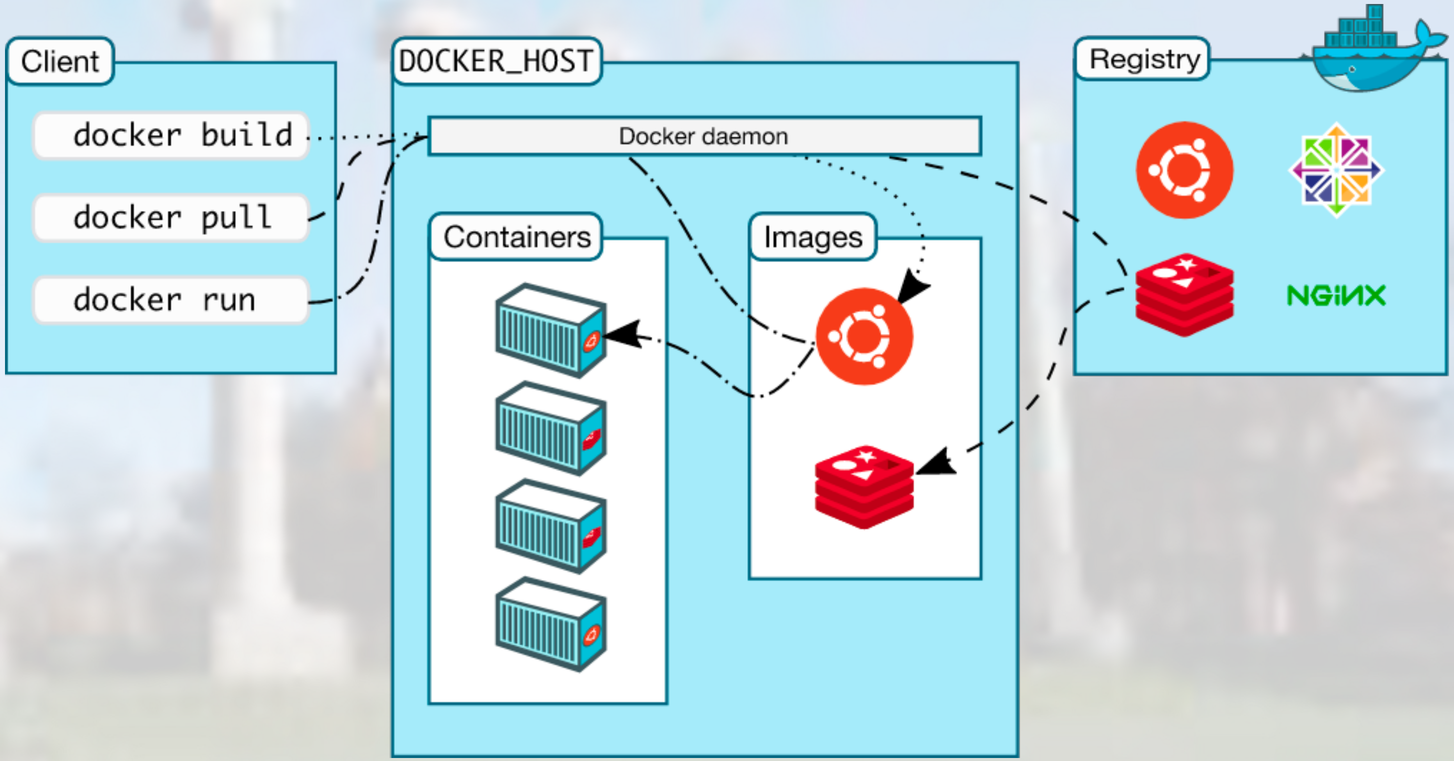
The Problem: Scalability & Reproducibility

- How do we ensure reliable portability of software developed on local development machines to other computational environments?
- How do we move code from local development on one machine to hundreds/thousands of machines?



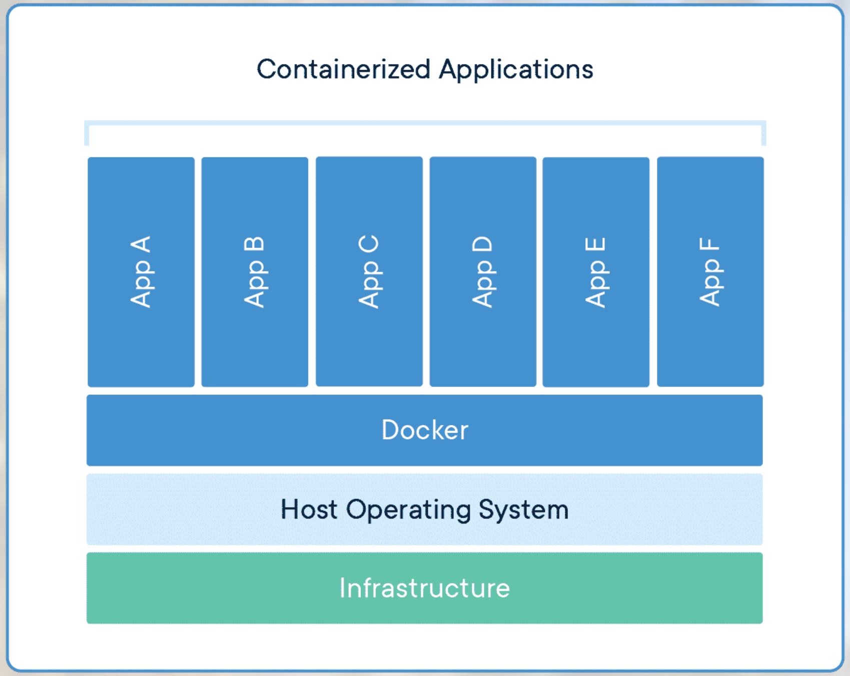
Docker

- **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 are how software is deployed in **Kubernetes**!*

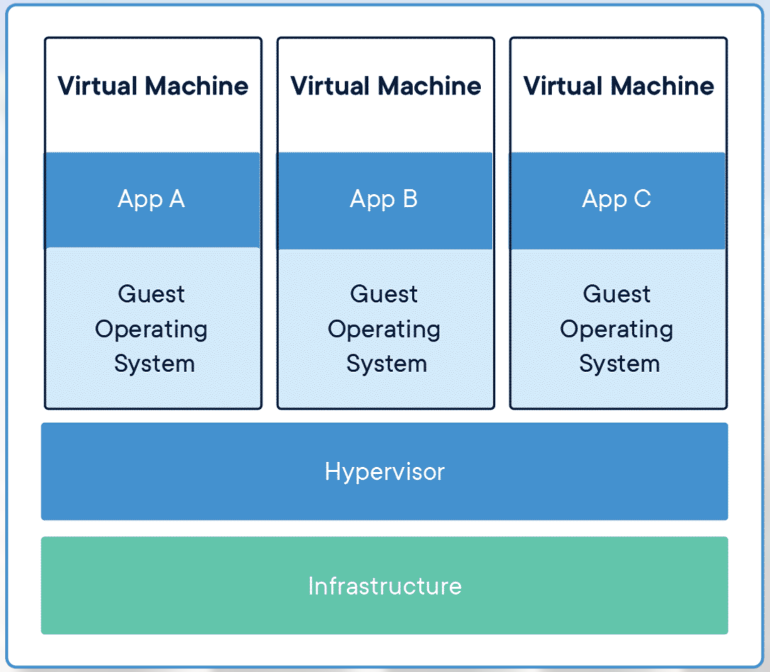


1. https://en.wikipedia.org/wiki/Docker_(software)
2. Image: https://docs.docker.com/get-started/overview/

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.

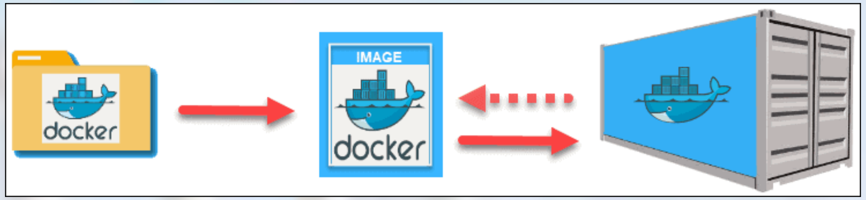


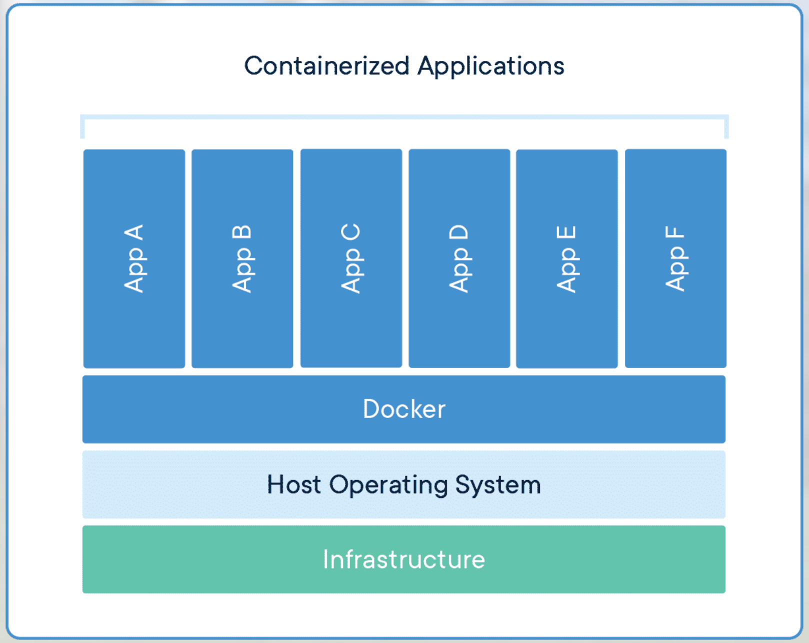
Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also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

https://www.docker.com/resources/what-container/

Key Docker Concepts

- **Dockerfile** - A list of commands and instructions describing how to build an Image
- **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.
- **Container** - An instantiated runtime of a docker image, containing all necessary software for a given application to run, both at the OS and language-specific level
  - Images become containers at runtime
- **Registry** - a service for storing container images





https://www.docker.com/resources/what-container/
https://phoenixnap.com/kb/docker-image-vs-container

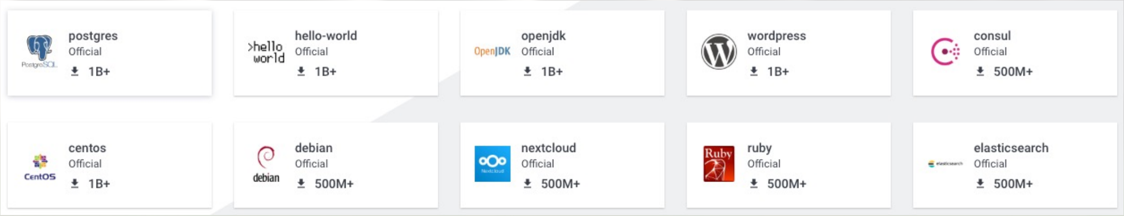
Sample Dockerfile

```
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
```

- ARG PYVERSION=3.8: Create a build argument for the python version that defaults to 3.8
- FROM python:${PYVERSION}: Start FROM an existing image in a Docker *registry*. In this case, python
- RUN mkdir -p /workspace, WORKDIR /workspace: Create a directory named /workspace and set it as the working directory
- COPY /requirements.txt /workspace: Copy a file named "requirements.txt" from the build directory to /workspace in the container
- RUN pip install -r ./requirements.txt: Use pip to install the requirements defined in requirements.txt
- COPY /*.py /workspace/: Copy all python files in the build directory to the /workspace directory in the container
- CMD /bin/bash: Set the default command to run when the container starts to /bin/bash

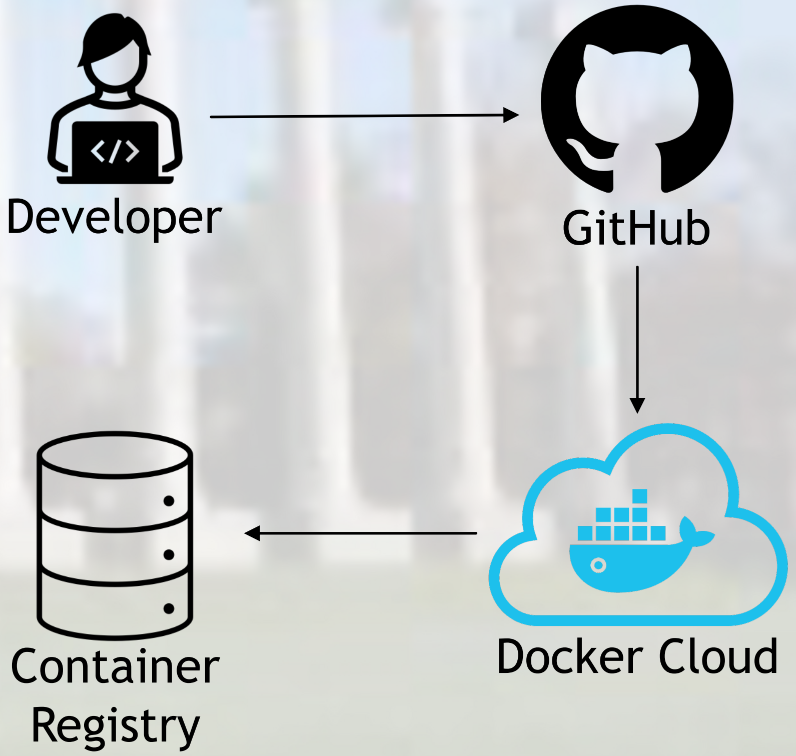
Docker Image Registries

- 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: nvcr.io/nvidia/pytorch:22.08-py3
- Security and Visibility
  - Container images published on registries can be public or private



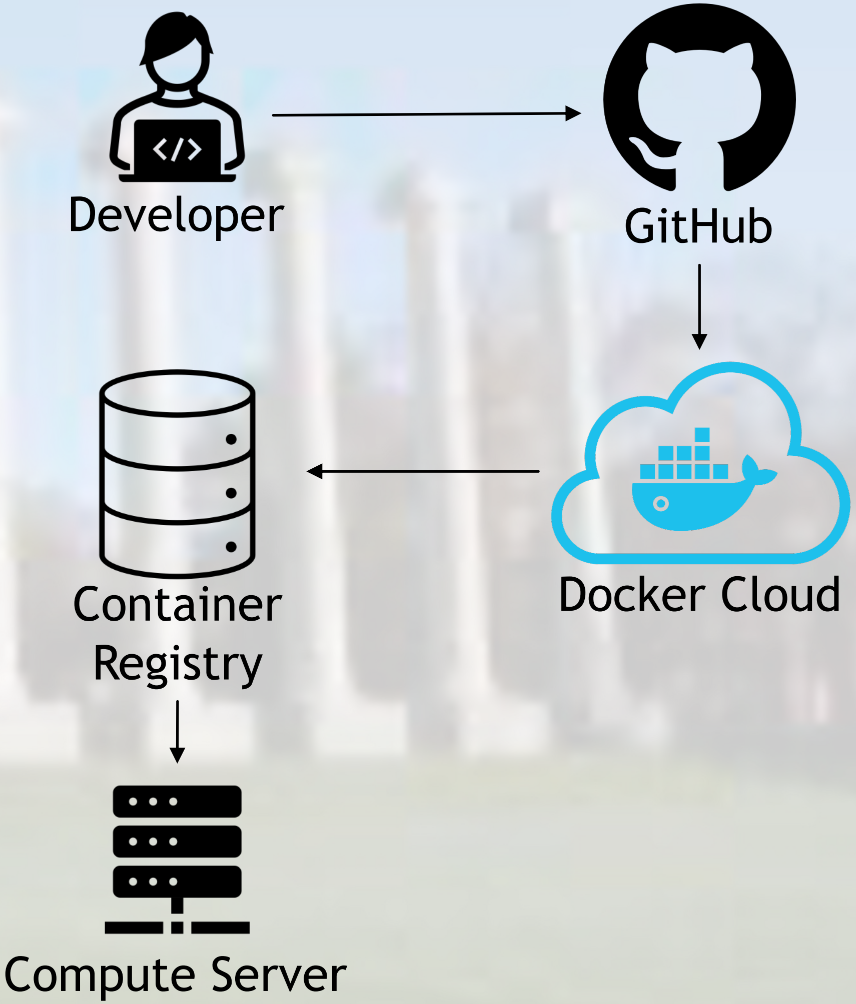
Automation and Docker Registries

- 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



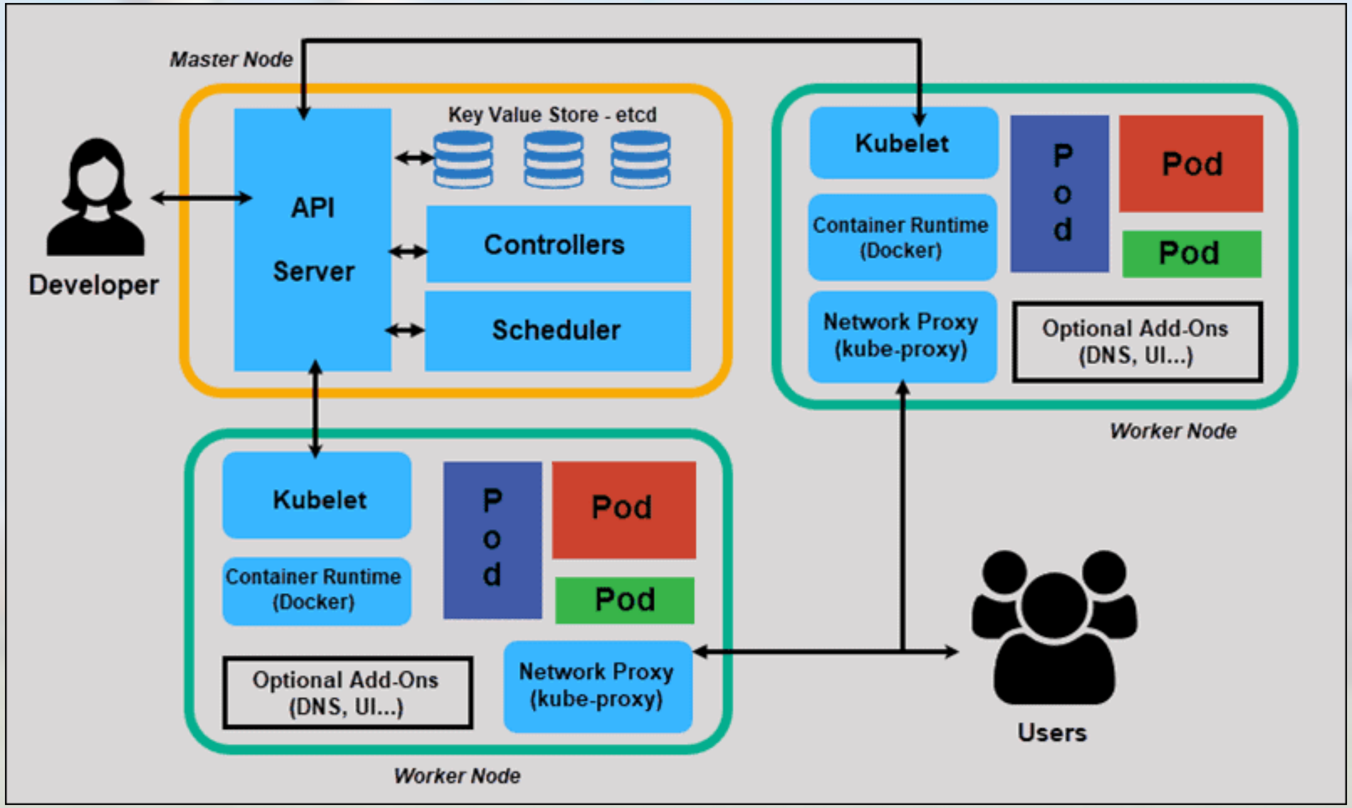
Deploying Software in Docker Containers

- Docker provides a reliable and portable way to manage software
- Docker registries provide a way to share docker images to any machine running docker
- Problems: Scalability & Orchestration
  - What happens when we want to deploy our software to multiple machines with differing amounts of resources running on large amounts of data?
  - What if we want multiple pieces of software to work *together* to solve a problem?



Kubernetes

- Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.[1]
- Kubernetes enables both simple and complex container orchestration



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

Key Kubernetes Concepts

- **Nodes** - a node is a physical machine where containers are deployed. Each node must run a container runtime.
- **Namespaces** - provide a way for K8s to partition cluster resources across multiple or many users in an exclusive way.
- **Pods** - 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
- **Jobs** – long running processes that require more than time and compute than pods
- **Services** - a set of pods working together

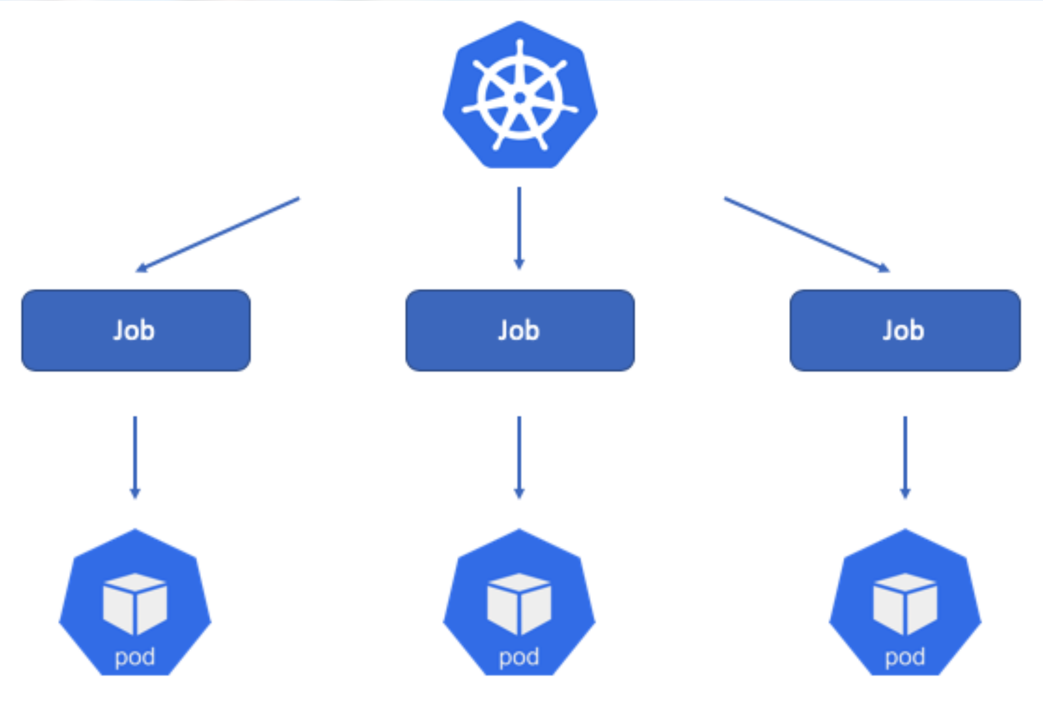


Image: https://aws.plainenglish.io/kubernetes-deep-dive-job-and-cronjob-5ffed1c5fa4e

Yet Another Markup Language (YAML)

XML	JSON	YAML

<Servers>
<Server>
<name>Server1</name>
<owner>John</owner>
<created>123456</created>
<status>active</status>
</Server>
</Servers>
	{
	Servers: [
	{
	name: Server1,
	owner: John,
	created: 123456,
	status: active
	}
	]
	}
		Servers:
		- name: Server1
		owner: John
		created: 123456
		status: active

- 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

Sample Kube Pod Spec

We begin by setting the API Version and the type of object we are creating (Pod), as well as the name of the pod

From here we are defining the container to run in this pod

Set the name of the container, the image the container should run, and the command that should run when the container begins

Here, we define the requested and maximum amount of resources our container needs to run, in this case that is 2 CPU cores and 4 GB of RAM

```
apiVersion: v1
kind: Pod
metadata:
  name: python3-pod
spec:
  containers:
    - name: python3-container
      image: python:3.8
      command: ["sleep", "infinity"]
      resources:
        limits:
          memory: 4Gi
          cpu: 2
        requests:
          memory: 4Gi
          cpu: 2
```

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

https://kubernetes.io/docs/reference/kubectl/

Automated Container Orchestration in Kubernetes

1. Users push code to VCS
2. CI/CD systems build Docker image and push to Container Registry
3. Users create pod/job using published Docker image
   - KubeCTL Command Line Tool
4. Kubernetes schedules the pod/job onto a node with required resources

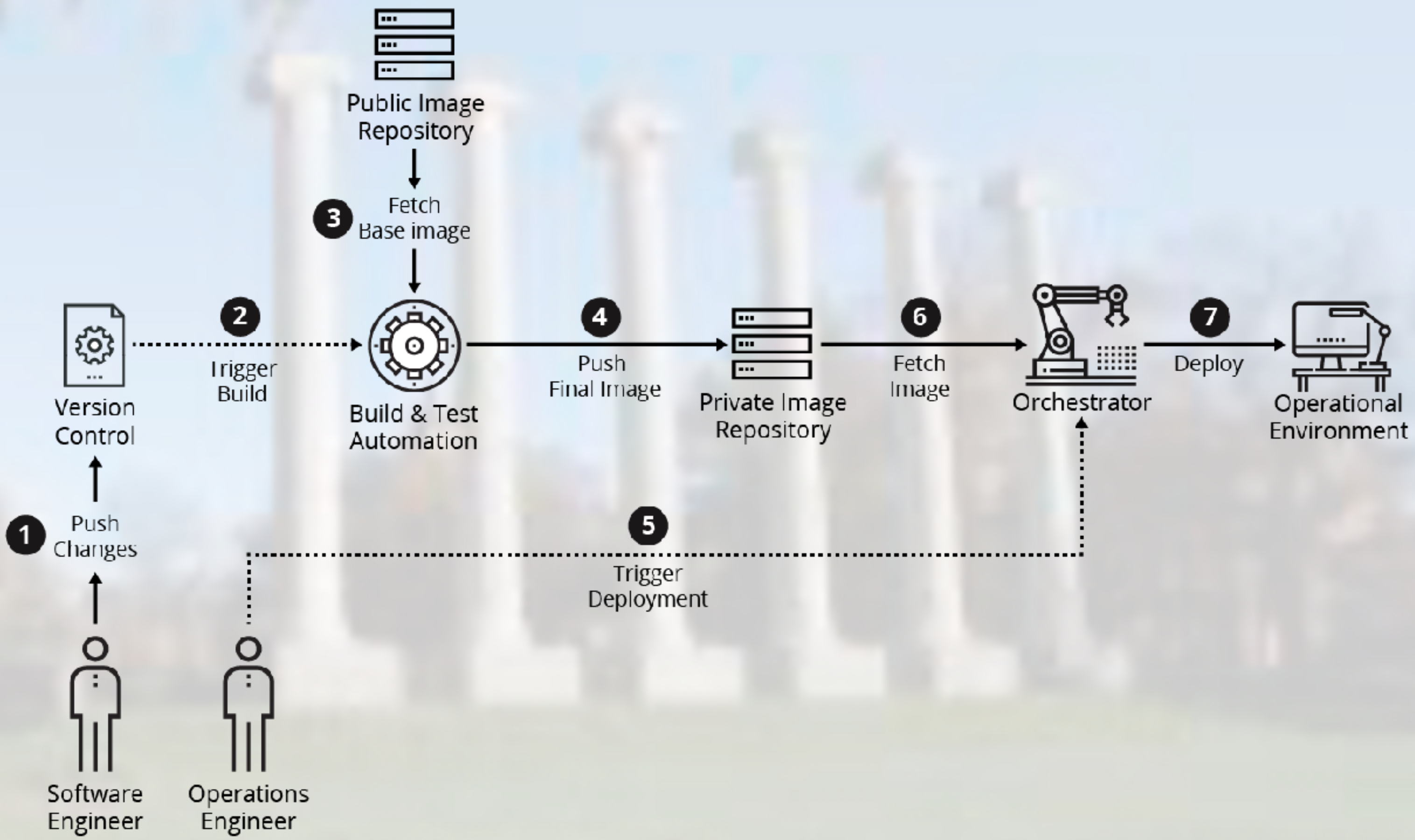
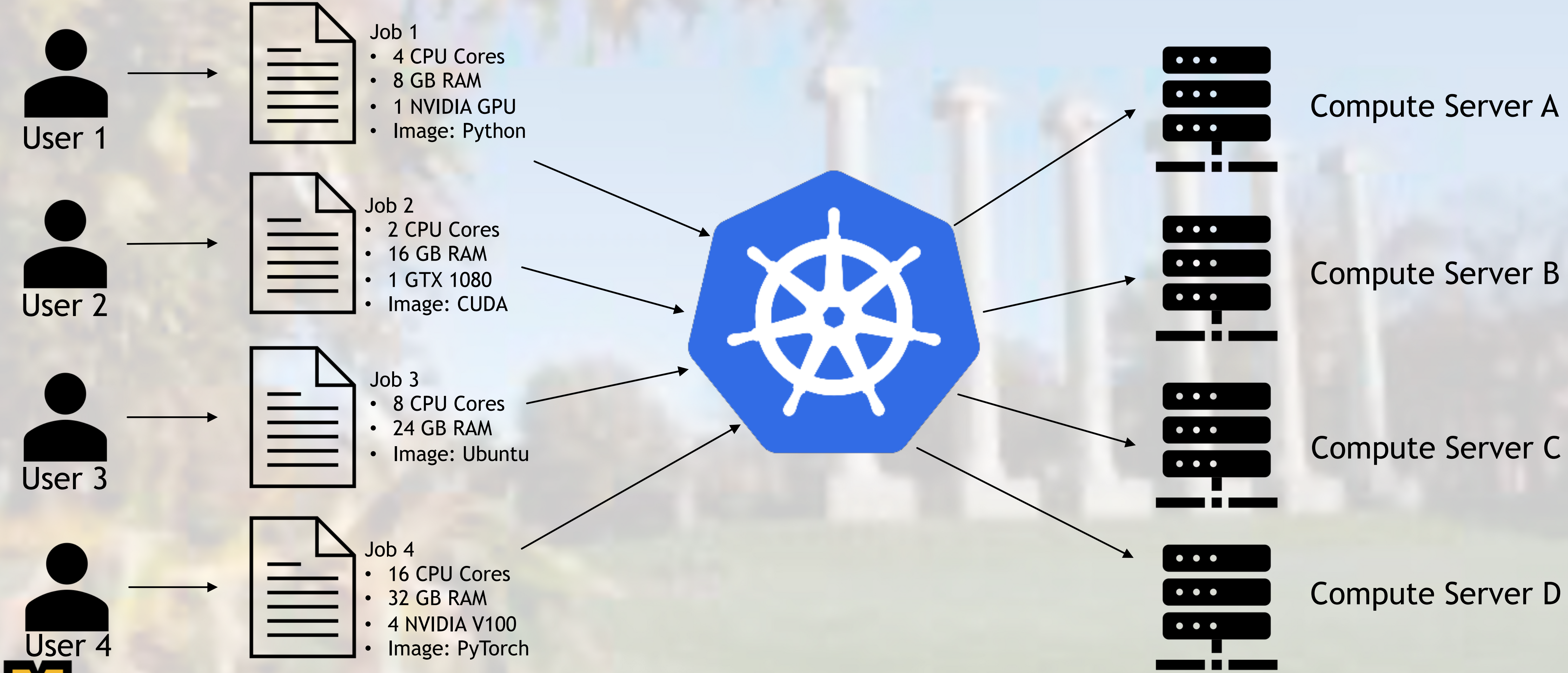


Figure 4: Model Container Supply Chain

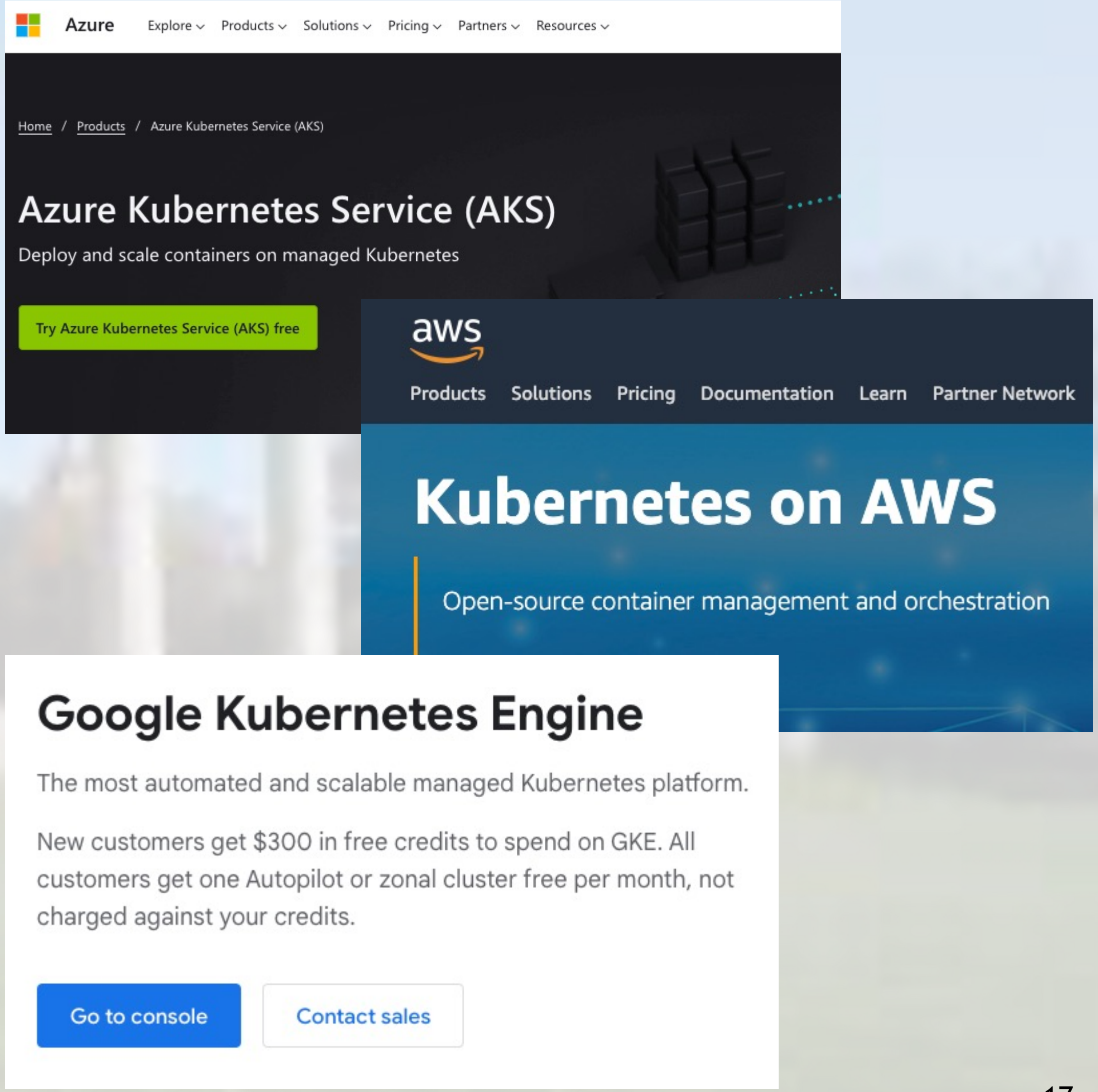
Image: https://insights.sei.cmu.edu/blog/7-quick-steps-to-using-containers-securely/

Deployments in Kubernetes



Kubernetes as Cloud Computing Platform

- K8s can be run on internal systems, to enable scheduling and orchestration of servers in a closed system
- K8s can also be used on Cloud Infrastructure as a Service (IaaS) services like:
  - Google Cloud Platform (GCP)
  - Amazon Web Services (AWS)
  - Microsoft Azure

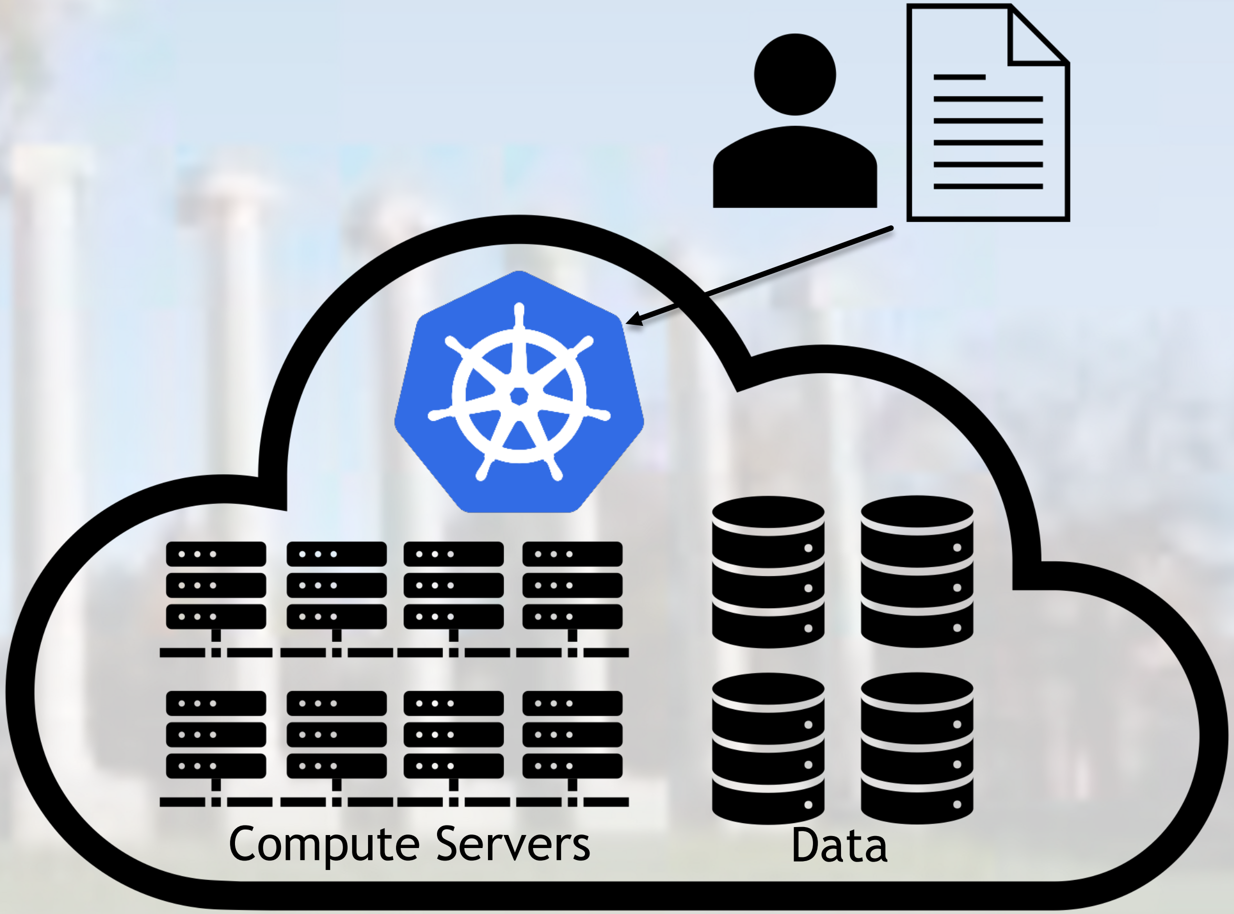


University of Missouri

https://aws.amazon.com/kubernetes/
https://cloud.google.com/kubernetes-engine
https://azure.microsoft.com/en-us/products/kubernetes-service/

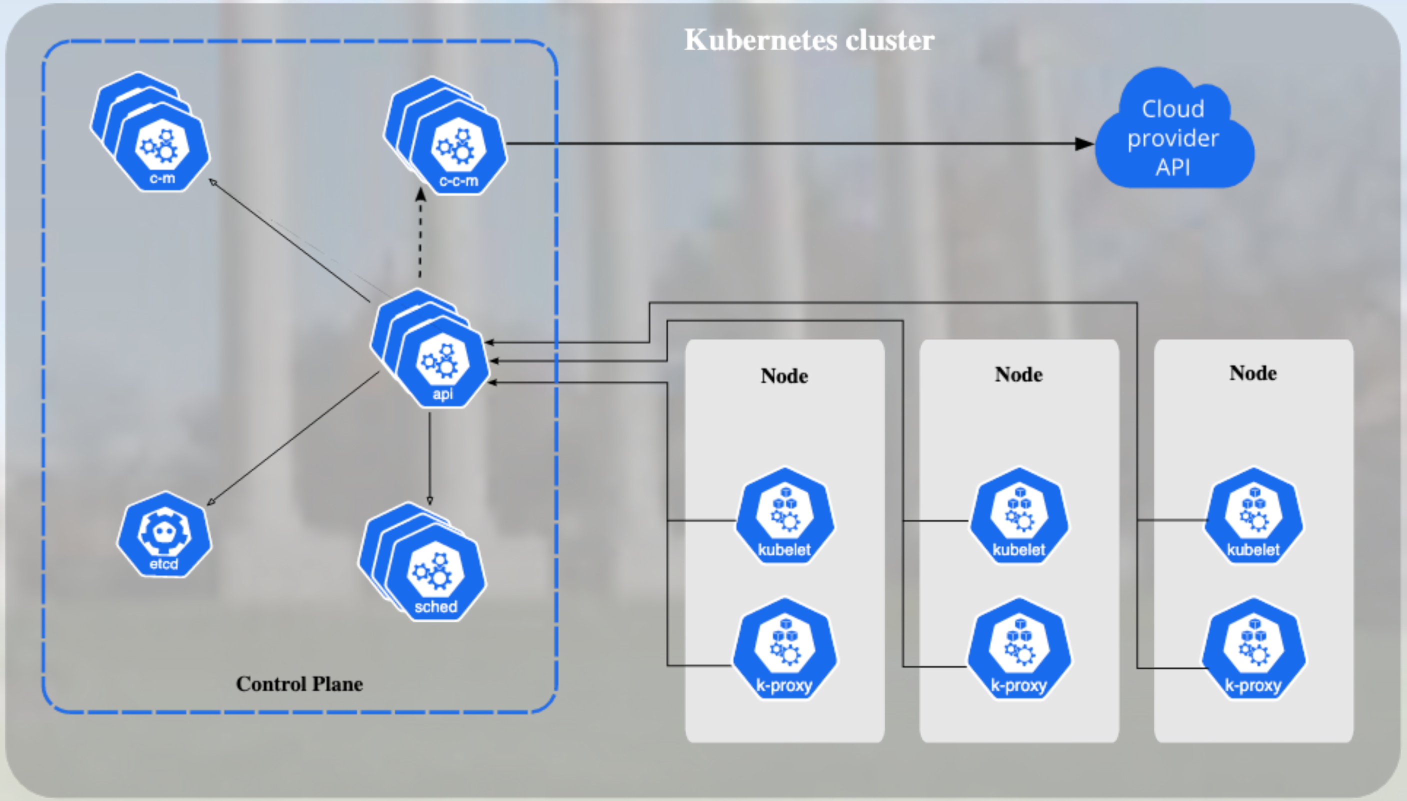
Why K8s in Cloud Services?

- Scalability
  - Cloud services offer an incredible amount of resources on demand
  - Integration of Cloud Compute Services with Cloud Storage (S3) enables efficient scaling of data intensive research
- Less Internal Overhead
  - No requirement of K8s setup and management for users

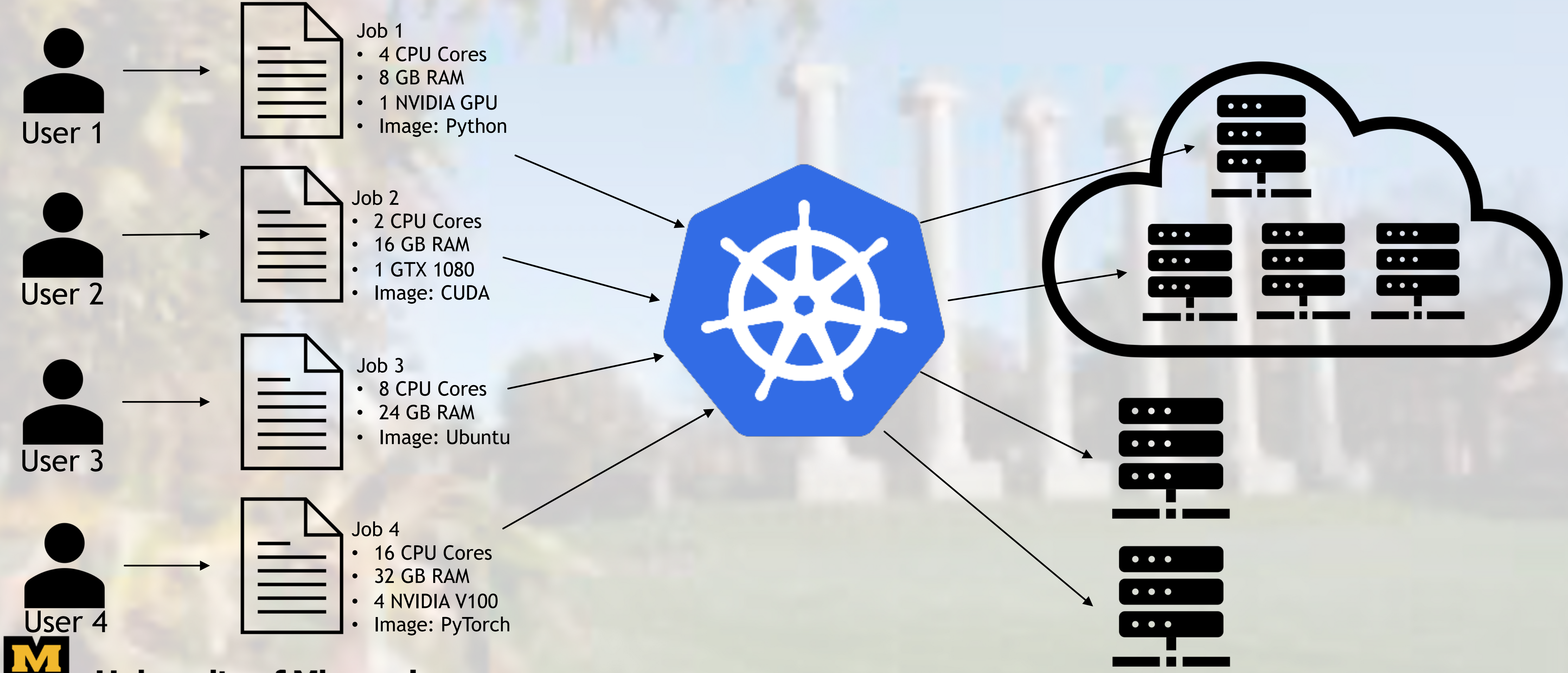


Kubernetes Cloud Controller Manager

- De-coupling of internal K8s cluster and cloud platform
- Enables hybrid systems where pods and jobs can run on either internal or cloud provided compute services
- Main Components:
  - Node controller
  - Route controller
  - Service controller



K8s Deployments in the Cloud



Example: Kubernetes in Action

Step 0: Creating Accounts & Logging In

- Create an account with an image registry
  - Docker Hub: https://hub.docker.com/
  - NRP GitLab: https://gitlab.nrp-nautilus.io/
- Download and install the Docker desktop client for your OS:
  - https://docs.docker.com/get-docker/
- Login to your image registry: `docker login [url]`
  - Enter you username and password to the image registry
- Gain access to a Kubernetes cluster and download the KubeCTL config file
  - Google Kubernetes Engine: https://cloud.google.com/kubernetes-engine
  - AWS Elastic Kubernetes Service: https://aws.amazon.com/eks/
  - NRP Nautilus: https://portal.nrp-nautilus.io/

Step 1: Building a Container

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
- K8s can work with many different container services, but for this example we use Docker
- Once we have built and pushed our container, we can see it online on Docker Hub:
  - https://hub.docker.com/repository/docker/jalexhurt/imagemagick/

Step 2: Build the K8s Spec YAML

pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: imagemagick-pod
spec:
  containers:
    - name: imagemagick-container
      image: jalexhurt/imagemagick
      command: ["sleep", "infinity"]
      resources:
        limits:
          memory: 4Gi
          cpu: 2
        requests:
          memory: 4Gi
          cpu: 2
```

- We specify the image we built and pushed in Step 1 as the `image` in the YAML
- Any image on a public registry can be used on the cluster
  - We do not have to always build and push our own container image
  - Many common frameworks / languages already have containers available (i.e., NodeJS, Python, etc.)
- The `sleep infinity` command will ensure our pod keeps running in the background so that we can connect to it
- For this minimalist example, we only ask for 2 CPU cores and 4 GB of RAM, but more resources can be requested and are available

Step 3: Create the Pod

- We use the KubeCTL tool with our newly built YAML file to create our pod on the cluster:

Bash
```
$ kubectl apply  -f  ./pod.yaml
```

- We can see our newly created pod and its status using KubeCTL tool:

Bash
```
$ kubectl get pods
```

- To get more in-depth information on a pod, we can use the describe command with the name of our pod:

Bash
```
$ kubectl describe pod imagemagick-pod
```

Step 4: Use the Pod

- Once the pod moves to the running state, we can attach to it and run commands inside the container with ImageMagick installed that we built in Step 1
- 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

```bash
$ kubectl exec –it  imagemagick-pod   --   /bin/bash
$ 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
```

Step 5: Delete the Pod

- Once we have finished using the Pod, we need to delete the pod to release the resources it is using
- Many clusters have policies on how long Pods can be running, since Pods typically do not have any kind of automatic exit strategy
- On the Nautilus cluster, after 6 hours, your Pod will be killed and the resources automatically released
- To delete the pod manually, we can again use KubeCTL:

Bash

```bash
$ kubectl delete pod imagemagick-pod
```