

Accelerating Data Science Workflows with Kubernetes

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 Kubernetes: Container Orchestration
- ▶ Kubernetes Architecture: Control Plane vs. Worker Node
- ▶ KubeCTL: Interfacing with K8s
- ▶ K8s Resource Types
 - ▶ PVCs, Pods, Jobs, & Services
- ▶ Provisioning Resources
 - ▶ YAML Specification Files
- ▶ Data Science Workflows in K8s
 - ▶ SKLearn
 - ▶ Jupyter
 - ▶ GPU-enabled Workflows: PyTorch
- ▶ Automation with Kubernetes
 - ▶ Using 3rd Party Libraries
 - ▶ Using the K8s Python Client

Workshop Outline

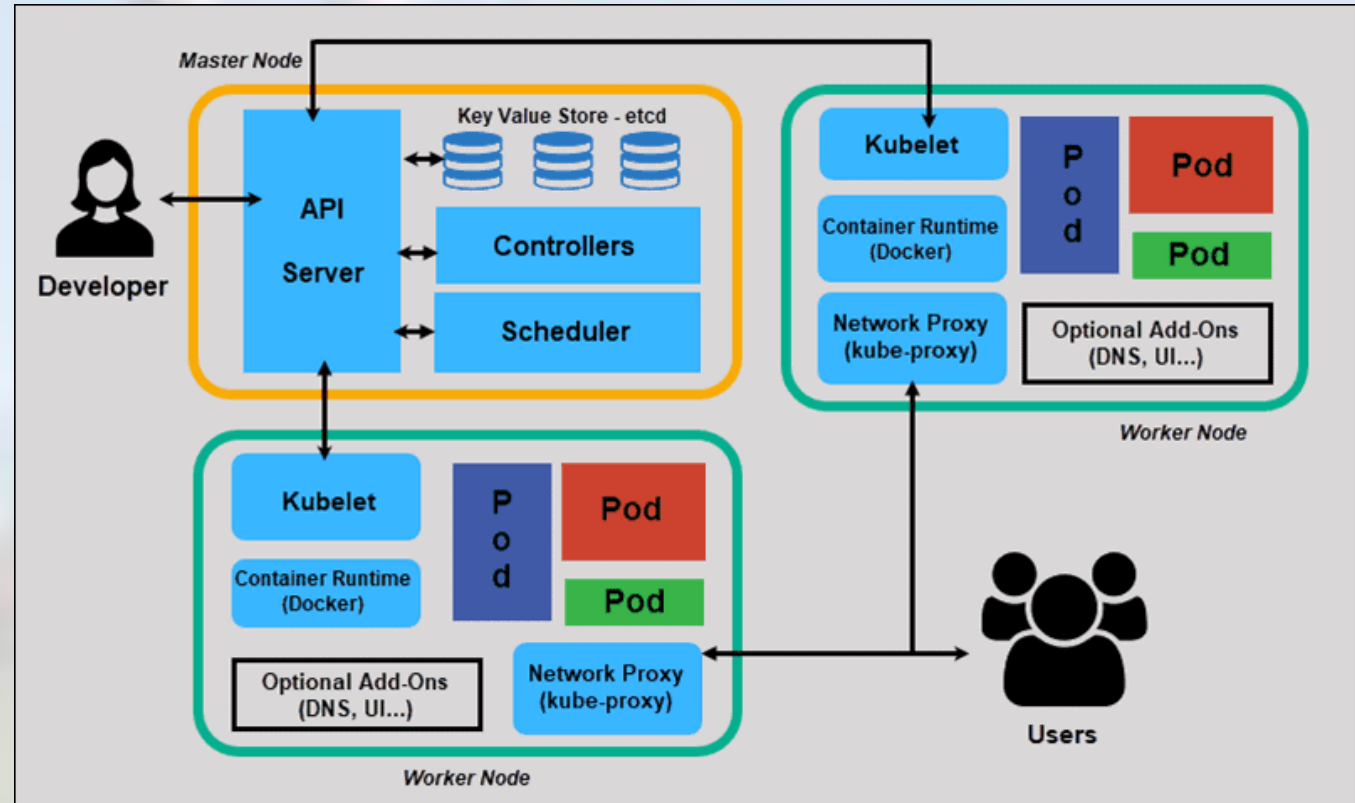
- ▶ Introduction to Kubernetes **(Slides)**
 - ▶ Container Orchestration
 - ▶ Kubernetes Architecture
 - ▶ Kubernetes Resource Types
- ▶ Interfacing with K8s: KubeCTL and YAML **(Slides)**
 - ▶ YAML Definition
 - ▶ KubeCTL introduction and syntax
- ▶ Introduction to K8s **(Hands-On)**
 - ▶ The National Research Platform
 - ▶ Creating persistent storage in K8s
 - ▶ Spawning a Pod
 - ▶ Creating a Job
- ▶ Running Data Science Workflows **(Hands-On)**
 - ▶ CPU-jobs: Sci-Kit Learn
 - ▶ Interactive Pods: Jupyter
 - ▶ GPU-jobs: PyTorch
- ▶ Automation with Kubernetes **(Slides)**
 - ▶ Using 3rd Party Libraries: Jinja2
 - ▶ Using the Kubernetes Python Client
- ▶ Automating K8s Job Creation **(Hands-On)**
 - ▶ Using the Kubernetes Python Client

Introduction to Kubernetes

What is Kubernetes and what is it used for

Kubernetes

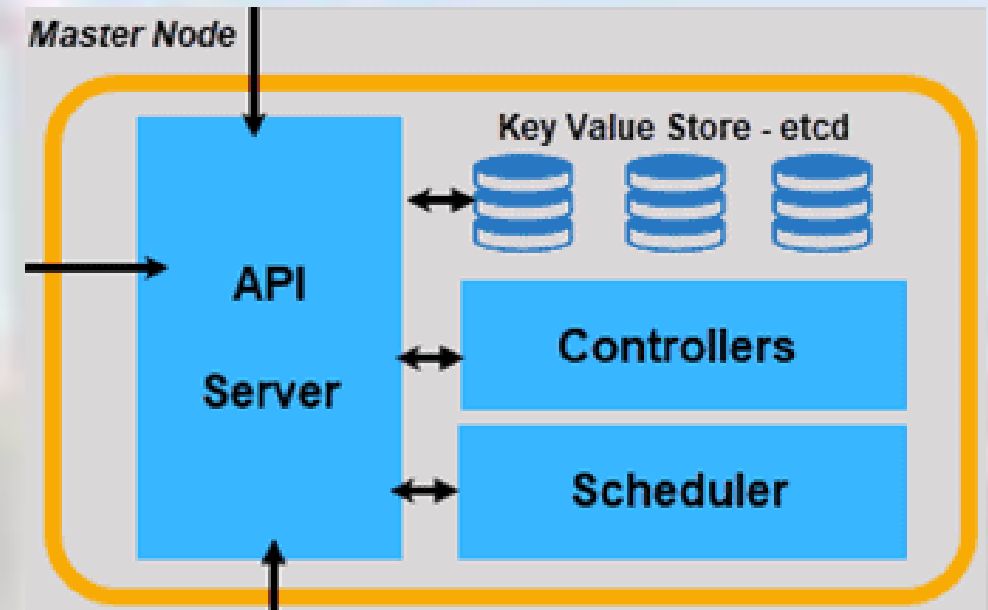
- ▶ Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.¹
- ▶ Kubernetes enables both simple and complex container orchestration
- ▶ Kubernetes cluster has two main components
 - ▶ Master node
 - ▶ Worker node



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

Kubernetes Master node

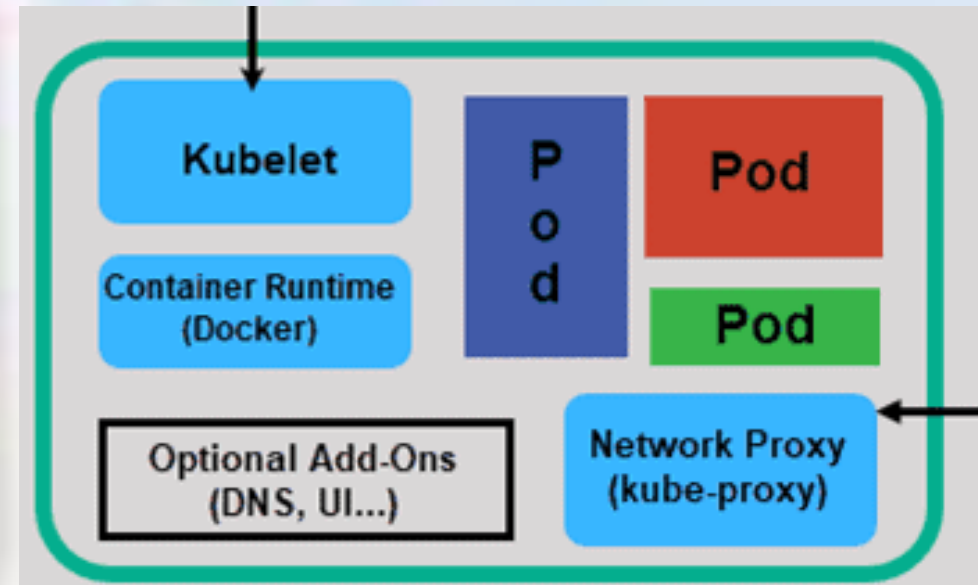
- ▶ Also known as the Control Plane, it is responsible for managing the state of the cluster
- ▶ API server: interface between master node and the rest of the cluster
- ▶ etcd: distributed key-value store that stores the cluster's persistent information
- ▶ Scheduler: responsible for scheduling pods 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



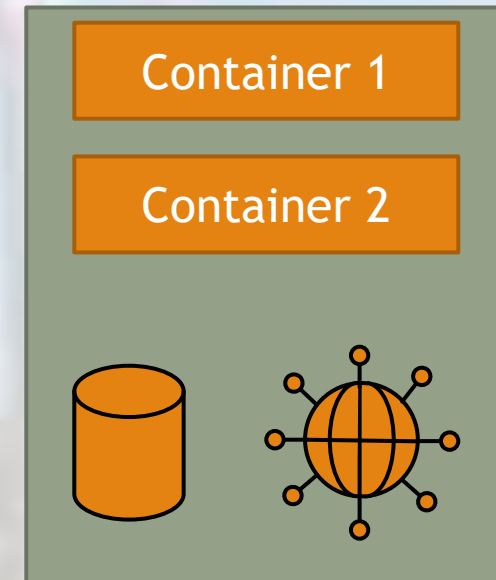
Kubernetes concepts

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.¹
- ▶ **Deployment** - is a higher-level concept that manages ReplicaSets and provides declarative updates to Pods along with a lot of other useful features.¹



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.¹
- ▶ 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

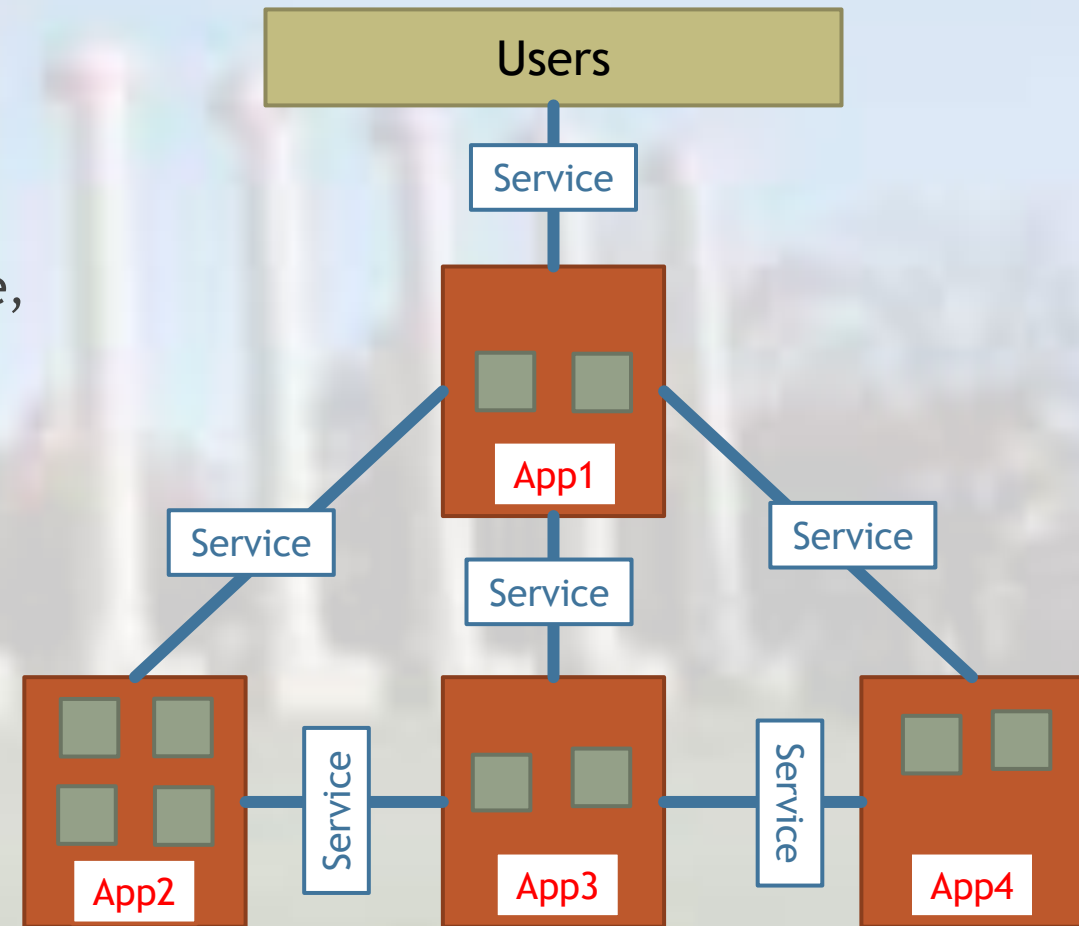
- ▶ To maintain the data generated a persistent volume (storage) is needed
- ▶ A **persistent volume** is storage on the cluster that has been provisioned by an administrator or dynamically provisioned using storage classes.¹
- ▶ There exists different classes of persistent volumes such as:
 - ▶ cephfs
 - ▶ 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> <Servers> <Server> <name>Server1</name> <owner>John</owner> <created>123456</created> <status>active</status> </Server> </Servers> </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

Synopsis of YAML language¹

- ▶ Comments in YAML begins with the (#) character and they must be separated from other tokens by whitespaces.
- ▶ Indentation of whitespace is used to denote structure.
- ▶ Tabs are not included as indentation for YAML files.
- ▶ Lists are important
 - ▶ List members are denoted by a leading hyphen (-).
 - ▶ List members are enclosed in square brackets and separated by commas.
 - ▶ YAML always requires colons and commas used as list separators followed by space with scalar values.
- ▶ Associative arrays are represented using colon (:) in the format of key value pair. They are enclosed in curly braces {}.

1. https://www.tutorialspoint.com/yaml/yaml_basics.htm

Structure of YAML file

- ▶ Each YAML file consists of specific parts, each part is dedicated to hold a type of information that enables us to communicate our needs to the Kubernetes cluster.
- ▶ **kind** - tells Kubernetes the type of the resource: pod, job, service, storage
- ▶ **metadata** - data about the resource
 - ▶ Name is most common attribute to set
- ▶ **spec** - the attributes specified in this section depends on the kind
 - ▶ We will cover common spec attributes

Pod without mounted volume

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
```

Job without mounted volume

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

From here we are defining the pod that is to be started by this job

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: job
metadata:
  name: python3-job
spec:
  template:
    containers:
      - name: python3-pod
        image: python:3.8
        command: ["sleep", "infinity"]
    resources:
      limits:
        memory: 4Gi
        cpu: 2
      requests:
        memory: 4Gi
        cpu: 2
```

Persistent Volume (PVC) Creation

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

From here we are defining the class of the persistent volume we are using

Here we define the type of access mode of the persistent volume we are creating

Here, we define the requested size of the persistent volume 50 GB of storage

To increase the size of the persistent volume storage we only need to modify this value
We can only increase but we cannot decrease

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: your_name
spec:
  storageClassName: rook-cephfs
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 50Gi
```

Pod with PVC

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 10 GB of RAM

Information of the mounted volume and how it is defined within the pod

```
apiVersion: v1
kind: Pod

metadata:
  name: pod-name-ss0

spec:
  containers:
    - name: pod-name-ss0
      image: python:3.8
      command: ["sh", "-c", "echo 'Im a new pod' && sleep infinity"]

      resources:
        limits:
          memory: 12Gi
          cpu: 2
        requests:
          memory: 10Gi
          cpu: 2

      volumeMounts:
        - mountPath: /data
          name: anes-pv

  volumes:
    - name: anes-pv
      persistentVolumeClaim:
        claimName: anes-pv
```

Job with mounted volume

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

From here we are defining the pod that is to be started by this job

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

Information of the mounted volume and it is defined within the job

```
apiVersion: v1
kind: job
metadata:
  name: python3-job
spec:
  template:
    containers:
      - name: python3-pod
        image: python:3.8
        command: ["sleep", "infinity"]
        resources:
          limits:
            memory: 4Gi
            cpu: 2
          requests:
            memory: 4Gi
            cpu: 2
        volumeMounts:
          - name: canada
            mountPath: /Canada
    volumes:
      - name: canada
        persistentVolumeClaim:
          claimName: canada2019-3
```


Interfacing with Kubernetes: KubeCTL installation

- ▶ <https://kubernetes.io/docs/tasks/tools/>
- ▶ This link provide options for installing kubectl with:
 - ▶ Linux
 - ▶ macOS
 - ▶ windows

Interfacing with Kubernetes:

KubeCTL installation (Linux)

there are three options to installing kubectl on Linux

- [Install kubectl binary with curl on Linux](#)
- [Install using native package management](#)
- [Install using other package management](#)

Interfacing with Kubernetes:

KubeCTL installation (Windows)

- ▶ There are two options to install Kubectl on windows
 - [Install kubectl binary with curl on Windows](#)
 - [Install on Windows using Chocolatey, Scoop, or winget](#)

Interfacing with Kubernetes: KubeCTL installation (MacOS)

► There are many options to install Kubectl on MacOS

- [Install kubectl binary with curl on macOS](#)
- [Install with Homebrew on macOS](#)
- [Install with Macports on macOS](#)

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
```

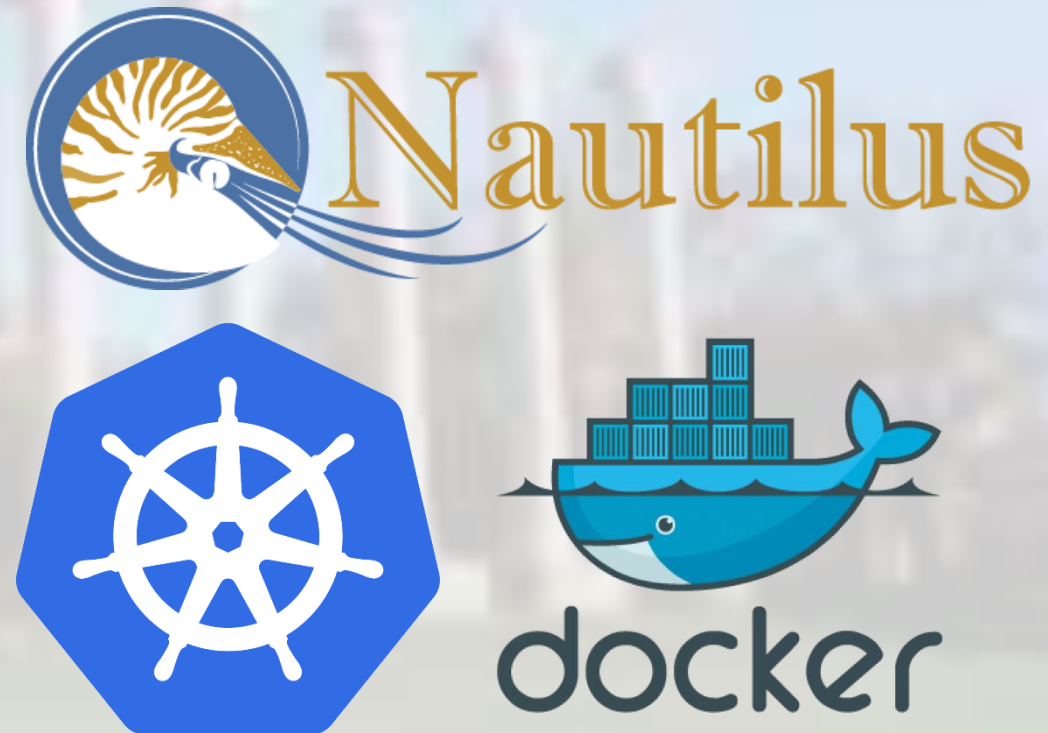

National Research Platform Nautilus Hyper Cluster

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 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]

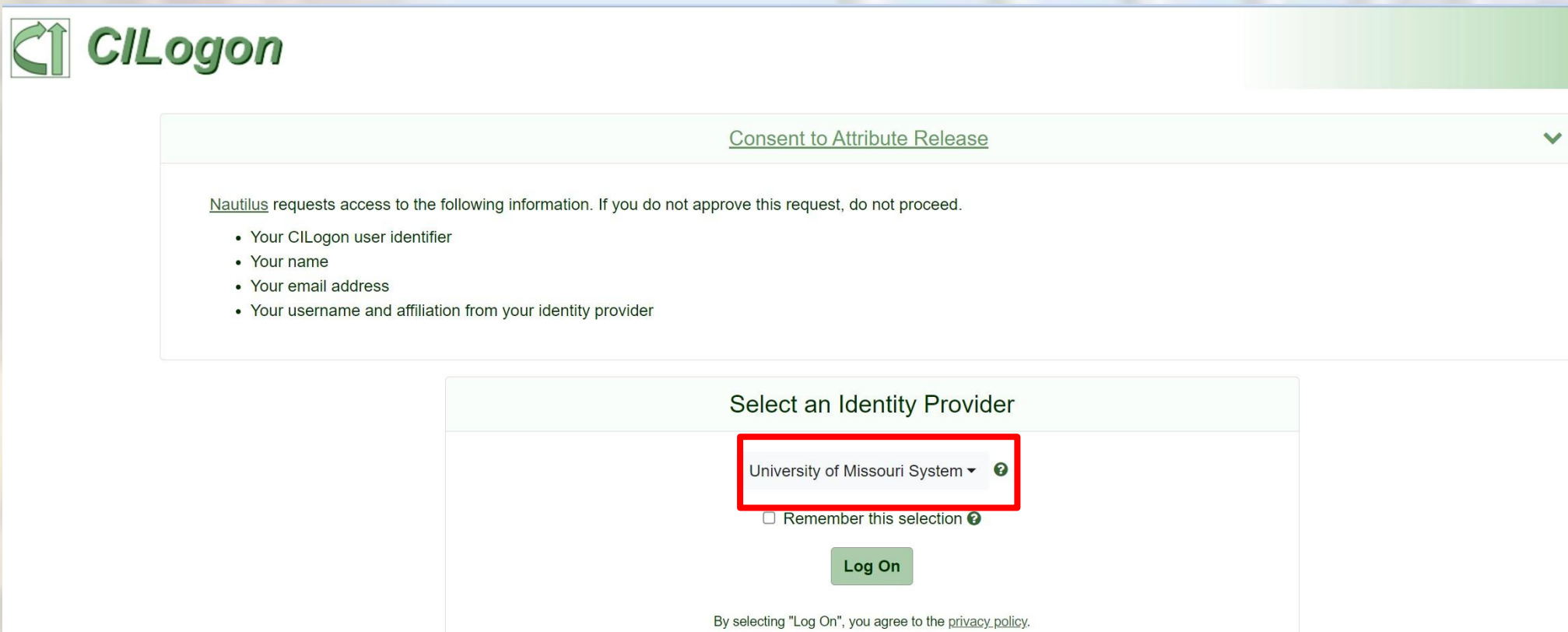
Documentation: <https://docs.nationalresearchplatform.org/>

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



The image shows the CILogon login interface. At the top left is the CILogon logo. Below it is a green bar with the text "Consent to Attribute Release" and a dropdown arrow. Underneath is a section titled "Nautilus requests access to the following information. If you do not approve this request, do not proceed." with a list of permissions: "Your CILogon user identifier", "Your name", "Your email address", and "Your username and affiliation from your identity provider". Below this is a section titled "Select an Identity Provider" with a dropdown menu showing "University of Missouri System" and a question mark icon. There is a checkbox for "Remember this selection" and a "Log On" button. At the bottom, a small text line reads "By selecting 'Log On', you agree to the [privacy policy](#)."

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

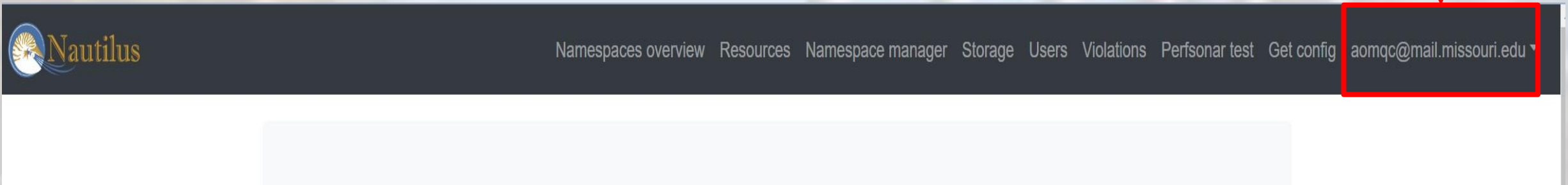
University of Missouri System ▼ ?

☐ 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

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

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

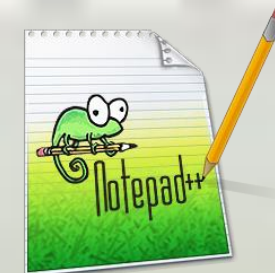
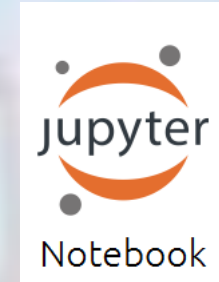
Automating Jobs in K8s using Bash and Python

Automating GPU Jobs in K8s using Bash and Python

- ▶ Nautilus is set up for parallel computing allowing for the running of multiple jobs at the same time
- ▶ Automation of jobs handling (submission, deletion) is key for the smooth operation
- ▶ There are multiple ways to automate the job handle processes
- ▶ We present here two ways:
 - ▶ jinja + bash
 - ▶ Nautilus Job Launcher library

jinja & bash

- ▶ We need a Python and/or Jupyter environment with these libraries:
 - ▶ yaml: to read/write yaml files
 - ▶ jinja2: to create and update templates that can be used to generate yaml files
 - ▶ os: to generate directories
- ▶ Text editor such as Notepad++ to write bash files



jinja & bash

- ▶ from jinja2 import Template
- ▶ Define template
 - ▶ It needs to be a multi line string
 - ▶ The variables to be updated are denoted by double braces `{{.}}`
 - ▶ The name of variable between the braces is used as reference
- ▶ `j2_template1 = Template(template1)`

```
template1 = '''apiVersion: batch/v1
kind: Job
metadata:
  name: anes-job-train-exp-{{ exp_num }}-{{ network }}-{{ data_type }}-pretrain
spec:
  template:
    spec:
      containers:
      - name: anes-pod-train-exp-{{ exp_num }}-{{ network }}-{{ data_type }}-pretrain
        image: gitlab-registry.nrp-nautilus.io/jhurt/cgisegment:e98e742e
        command: ["/bin/sh", "-c"]
        args:
        - python3 main.py --task train --output_dir /canada2019-3/{{ sourcedir }}/e
        volumeMounts:
        - name: canada2019-3
          mountPath: /canada2019-3
      resources:
        limits:
          memory: 12Gi
          cpu: "4"
          nvidia.com/gpu: 2
        requests:
          memory: 12Gi
          cpu: "4"
          nvidia.com/gpu: 2
      volumes:
      - name: canada2019-3
        persistentVolumeClaim:
          claimName: canada2019-3
        restartPolicy: OnFailure
      backoffLimit:
      ...
```

jinja & bash

- ▶ We use a loop to auto generate the files
- ▶ We need to define variables in a dictionary where:
 - ▶ **Keys**: variable names as defined in the template
 - ▶ **Values**: values of the variables for this iteration
- ▶ Apply values to the template using:


```
output_file = j2_template1.render(data)
```
- ▶ Save the yaml file to the appropriate location

```
for exp in list(range(8)):
    exp_num = exp + 1

    if os.path.exists('{}/exp{}'.format(source_dir, exp_num)):
        shutil.rmtree('{}/exp{}'.format(source_dir, exp_num))
    os.mkdir('{}/exp{}'.format(source_dir, exp_num))

    for folder in folders_list:

        parts      = folder.split('_')
        network    = parts[0]
        data_type  = parts[1]

        data = {'sourcedir':source_dir,
                'exp_num':exp_num,
                'network':network,
                'data_type':data_type,
                'outputdir':dict1[folder][0],
                'configfile':dict1[folder][1]}

        output_file = j2_template1.render(data)

        fileout = open('{}/exp{}/job_exp{}_{}_{}.yaml'.format(source_dir, exp_num,
                                                                network,
                                                                data_type,
                                                                folder))
        fileout.write(output_file)
        fileout.close()
```

jinja & bash

- ▶ Now that all yaml files have been generated we need bash files to
 - ▶ Submit jobs
 - ▶ Delete jobs after they finish
- ▶ We will write a bash file for each operation
 - ▶ Bash for job submission
 - ▶ Bash for deletion of completed jobs
- ▶ Execute bash file in the terminal

```

1  @ECHO OFF
2
3  Rem This batch file executes kubectl commands to create training jobs
4
5  ::echo %kubectl%
6  SET exp_list=2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
7
8  (for %%a in (%exp_list%) do (
9      echo %%a
10
11     kubectl create -f experiments\exp%%a/job_exp%%a_deeplab_img.yaml
12     kubectl create -f experiments\exp%%a/job_exp%%a_deeplab_tci.yaml
13     kubectl create -f experiments\exp%%a/job_exp%%a_deeplab_img_pretrained.yaml
14     kubectl create -f experiments\exp%%a/job_exp%%a_deeplab_tci_pretrained.yaml
15
16     kubectl create -f experiments\exp%%a/job_exp%%a_unet_img.yaml
17     kubectl create -f experiments\exp%%a/job_exp%%a_unet_tci.yaml
18     kubectl create -f experiments\exp%%a/job_exp%%a_unet_img_pretrained.yaml
19     kubectl create -f experiments\exp%%a/job_exp%%a_unet_tci_pretrained.yaml
20 ))
21
22 echo "batch complete"
    
```

```

1  @ECHO OFF
2
3  Rem This batch file executes kubectl commands to delete training jobs
4
5  ::echo %kubectl%
6  SET exp_list=1 2 3 4 5 6 7 8
7
8  (for %%a in (%exp_list%) do (
9      echo %%a
10
11     kubectl delete -f experiments_2\exp%%a/job_exp%%a_deeplabv3_img.yaml
12     kubectl delete -f experiments_2\exp%%a/job_exp%%a_deeplabv3_tci.yaml
13     kubectl delete -f experiments_2\exp%%a/job_exp%%a_deeplabv3plus_img.yaml
14     kubectl delete -f experiments_2\exp%%a/job_exp%%a_deeplabv3plus_tci.yaml
15
16     kubectl delete -f experiments_2\exp%%a/job_exp%%a_unet_img.yaml
17     kubectl delete -f experiments_2\exp%%a/job_exp%%a_unet_tci.yaml
18     kubectl delete -f experiments_2\exp%%a/job_exp%%a_unetplus_img.yaml
19     kubectl delete -f experiments_2\exp%%a/job_exp%%a_unetplus_tci.yaml
20 ))
21
22 echo "batch complete"
    
```

jinja & bash

- ▶ We can use bash in addition to Powershell and Jinja2 to automate K8s job launch
- ▶ Creation of template Kube Spec YAML with environment variables (preceded by \$)
- ▶ Bash scripting combined with environment variables to set the Dataset and/or Model to train and automatically launch the job

```
spec:
  template:
    spec:
      containers:
        - name: myContainer
          image: $CONTAINER_IMAGE
          workingDir: $WORKDIR
```

Template YAML

```
Dirs="mydir1 mydir2 mydir3 mydir4"
Container="ubuntu:20.04"

for Dirpath in $Dirs; do
  CONTAINER_IMAGE=$Container WORKDIR=$Dirpath envsubst < template.yml | kubectl apply -f -
done
```

Bash Script

Nautilus Job Launcher

- ▶ This Nautilus Job Launcher is an open-source Python library that enables automation of launching jobs on the NRP Nautilus HyperCluster.

- ▶ <https://github.com/MUAMLL/Nautilus-Job-Launcher>

- ▶ Installation:

- ▶ Install from GitHub:

```
pip3 install git+https://github.com/MUAMLL/Nautilus-Job-Launcher
```

Automating GPU Jobs on Nautilus

Nautilus Job Launcher

- ▶ The Nautilus Launcher can be used as
 - ▶ an application at the command line that will kick off jobs from a YAML config file
 - ▶ it can be utilized as a library integrated into other Python applications.
- ▶ You must have your Kubernetes **config** file in **~/.kube/config** to use this library!

Automating GPU Jobs on Nautilus

Nautilus Job Launcher

- ▶ Command line: The job launcher is invoked as a library and uses a configuration file (YAML):

```
python3 -m nautiluslauncher -c cfg.yaml
```

- ▶ You can choose to perform a dryrun by passing a `--dryrun` flag:

```
python3 -m nautiluslauncher -c cfg.py --dryrun
```

- ▶ `cfg.yaml`: this file contains the required configuration for the Job launcher library to work

Nautilus Job Launcher

- ▶ Configuration requires three keys:
- ▶ Namespace (**required**):
 - ▶ the namespace on the Nautilus cluster you'd like to use
- ▶ Jobs (**required**):
 - ▶ list of dictionaries that define all of the parameters for each job
- ▶ Defaults (**optional**):
 - ▶ It is a starting place for all jobs in your config.
 - ▶ All jobs will use the defaults as the beginning configuration and then whatever is placed in each job will be added to **or override** what is present in the defaults key

```
defaults:
  container: python:3.8
  workingDir: /mydir

jobs:
-
  container: python:3.7
-
  workingDir: /mydir2
-
  container: python:3.7
  workingDir: /mydir2
```

| Key | Description | Default | Type |
|-------------------|--|----------|----------------|
| job_name | The name of the job | required | str |
| image | The container image to use | required | str |
| command | The command to run when the job starts | required | str/list[str] |
| workingDir | Working directory when the job starts | None | str |
| env | The environment variables | None | dict[str, str] |
| volumes | The volumes to mount | None | dict[str, str] |
| ports | The container ports to expose | None | list[int] |
| gpu_types | The types of GPUs required | None | list[str] |
| min_cpu | Minimum # of CPU Cores | 2 | int |
| max_cpu | Max # of CPU cores | 4 | int |
| min_ram | Min GB of RAM | 4 | int |
| max_ram | Max GB of RAM | 8 | int |
| gpu | # of GPUs | 0 | int |
| shm | When true, add shared memory mount | false | bool |

Automating GPU Jobs on Nautilus

Nautilus Job Launcher

- ▶ Library usage:
- ▶ The Job launcher can be integrated with user's application/library
- ▶ This can be done in different ways:
 - ▶ import Job launcher into the user's scripts.
 - ▶ utilize a dictionary to configure your jobs and integrate that into your application
 - ▶ from a YAML file

Automating GPU Jobs on Nautilus

Nautilus Job Launcher

import Job Launcher into the user's scripts.

```
from nautiluslauncher import Job, NautilusAutomationClient

client = NautilusAutomationClient("mynamespace")
images = ["python:3.6", "python:3.7", "python:3.8"]
for i, img in enumerate(images):
    j = Job(job_name=f"test_python_{i}", image=img, command=["python", "-c", "print('hello world')"])
    client.create_job(j)
```

Automating GPU Jobs on Nautilus

Nautilus Job Launcher

Utilize a dictionary to configure your jobs

```
from nautiluslauncher import NautilusJobLauncher

my_jobs = {
    "namespace": "mynamespace",
    "jobs": [
        {"image": "python:3.6", "command": ["python", "-c", "print('hello world')"], "job_name": "myjob1"},
        {"image": "python:3.7", "command": ["python", "-c", "print('hello world')"], "job_name": "myjob2"},
        {"image": "python:3.8", "command": ["python", "-c", "print('hello world')"], "job_name": "myjob3"}
    ]
}

launcher = NautilusJobLauncher(my_jobs)
launcher.run()
```

Automating GPU Jobs on Nautilus

Nautilus Job Launcher

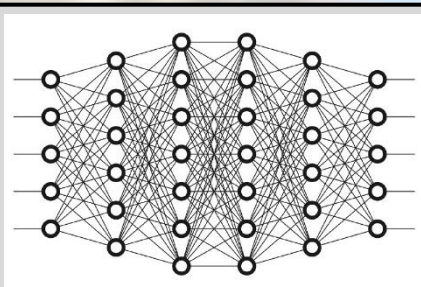
from a YAML file

```
from nautiluslauncher import NautilusJobLauncher

my_file = "myCfg.yaml"

launcher = NautilusJobLauncher.from_config(my_file)
launcher.run()
```

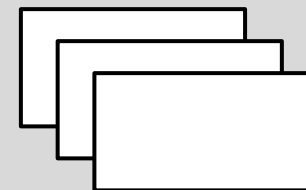
Deep Learning on Nautilus: Transformer Research



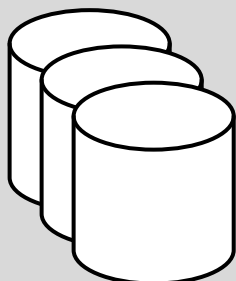
9 Deep Neural Architectures



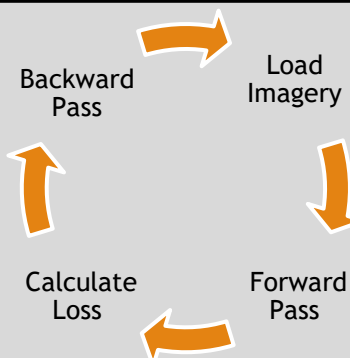
27 Trained Models



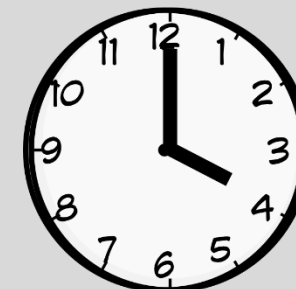
124.74 TB
Imagery
Processed



3 Open Source
HR-RSI Datasets



- 8,100 Epochs
- 30M iterations
- 1.7B parameters



Wall Clock Time:
76 days, 10 hours



MUAMLL/SDSS2025

- ▶ NRP Portal

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

- ▶ JupyterHub Instance:

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

- ▶ Tutorial Repository for this Tutorial:

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

- ▶ Git Clone Command:

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