High-Performance Data-Intensive Computing Systems Laboratory

With Big Data Comes Big Compute: Scaling Machine Learning onto Public and Commercial Clouds with Kubernetes – Part 2

Introduction to Containers and Kubernetes

IEEE International Conference on Big Data

December 2023

University of Missouri

Part 2 – Learning Objectives

- Deploying Scikit Learn ML Jobs to Kubernetes
- Deploying GPU Jobs to Kubernetes for Training Computer Vision Models
- Scaling Computer Vision Models on Kubernetes with Job Automation

MU-HPDI/Nautilus – Reminder from Part 1

- Sample Dockerfiles
- Sample Kubernetes YAML File
- Wiki with detailed walkthroughs for:
  - Getting Started
  - Creating PVC
  - Creating Pods
  - Creating Jobs

- Tutorial Repository of Jupyter Project Pages, Code Samples, YAML, etc.

https://github.com/MU-HPDI/bigdata-2023

- Git Clone Link:

https://github.com/MU-HPDI/bigdata-2023.git

Over the short break you should have cloned done the examples and content for Part 2

High-Performance Data-Intensive Computing Systems Laboratory

Part 2-A

National Research Platform
Nautilus Research Cluster

Introduction to Containers and Kubernetes

IEEE International Conference on Big Data

December 2023

University of Missouri

Motivation

- Shallow ML modules often require extensive hyperparameter optimization to find optimal performance
- Deep Learning models require incredible amounts of compute to effectively train
- Using single developer machines or local on-prem resources often fail to scale effectively
- Kubernetes Clusters provide an efficient and scalable solution to training ML and deep learning models at scale
- This workshop will address the building and deployment of ML and deep learning containers to Nautilus to train deep networks

Part-2 Tutorial Outline

- Quick Containerization Review and Advanced:
  - Review of containerization
  - Building ML containers for Scikit-Learn
  - Building Optimized Containers for Deep Learning with PyTorch
  - Using Common Frameworks: Detectron2, MMDetection, and Ultralytics
- Kubernetes ML / Deep Learning:
  - Review of Kubernetes Architecture and Key Concepts
  - Introduction to NRP Nautilus HyperCluster
  - Migrating Data to NRP with S3
  - Deploying Scikit-Learn ML Jobs to Nautilus
  - Deploying GPU Jobs to Nautilus for Training Computer Vision Models

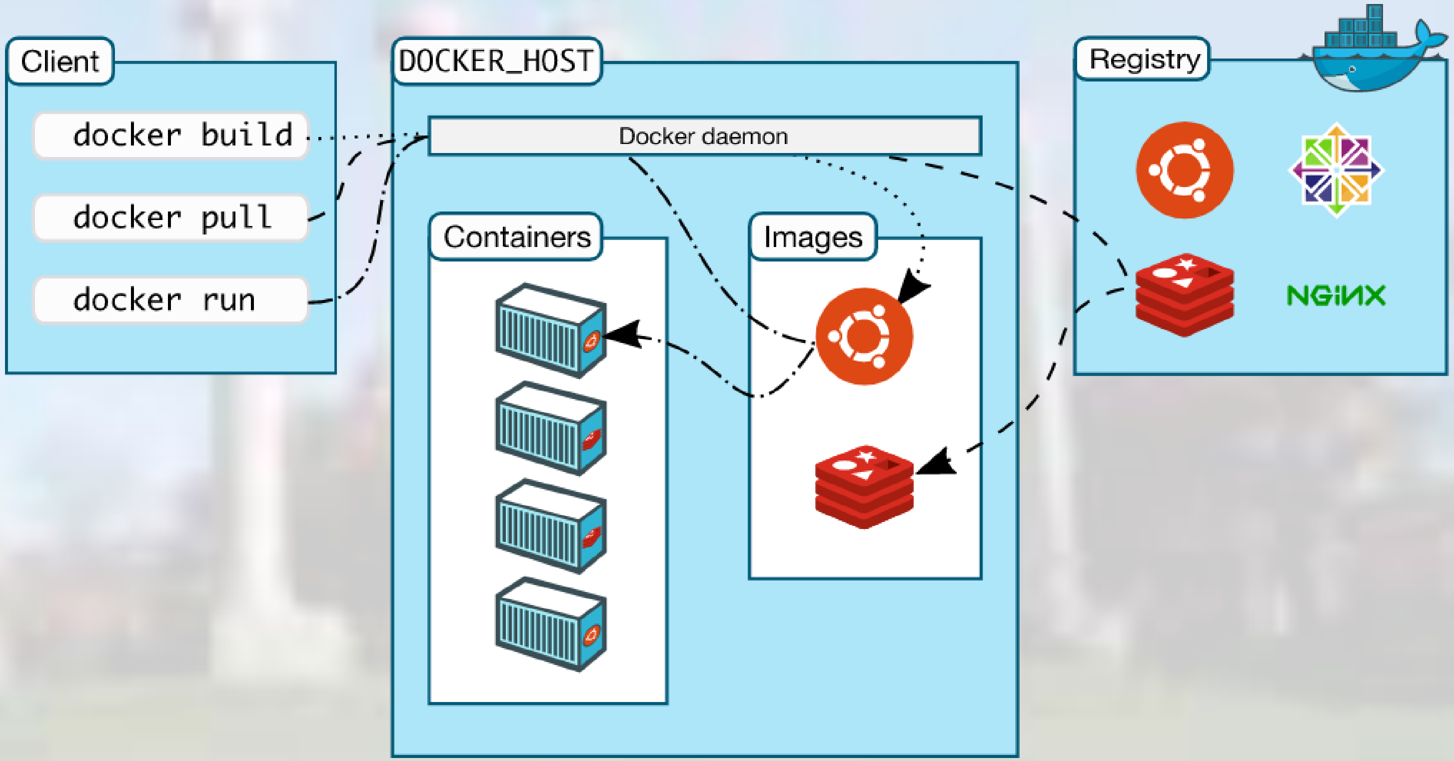
Containerization

Building Optimized Containers for Deep Learning with PyTorch

Containerization

Docker

- Docker
  - 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.
- Nautilus Gitlab
  - Offers the ability to create repo for your code
  - Offers the ability to create easily maintained and developed custom images using CI/CD feature

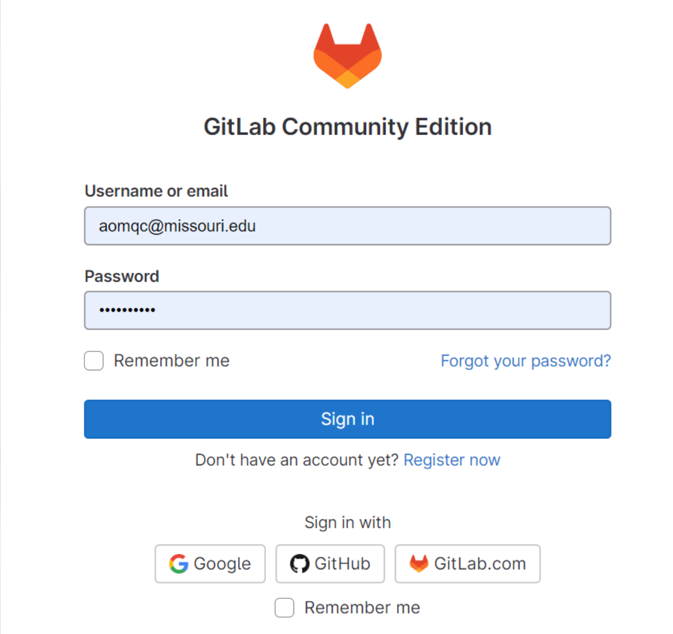


Containerization
Nautilus Gitlab

https://gitlab.nrp-nautilus.io/

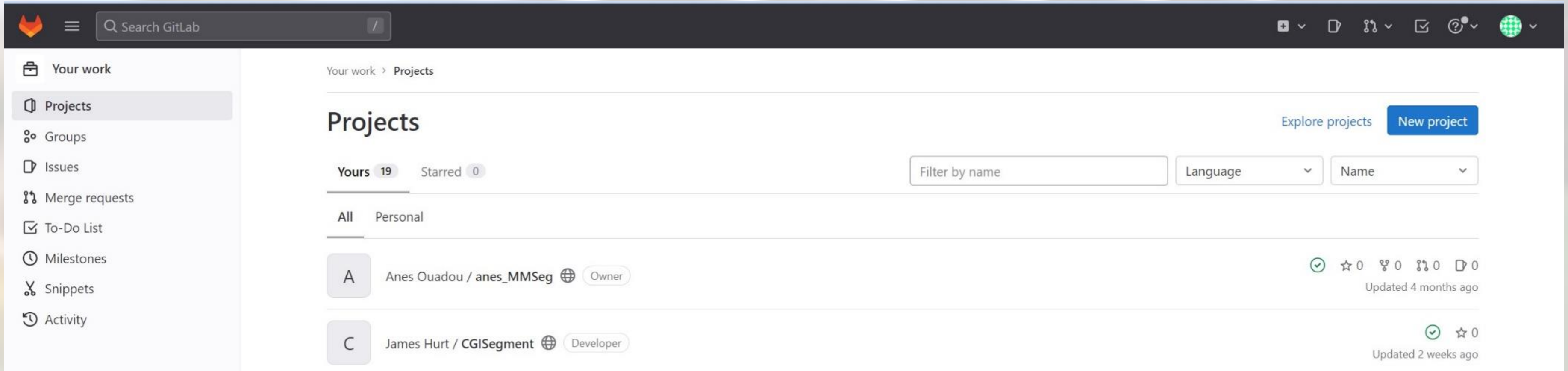
- Git Version Control System
- CI/CD services for building and storing container images to use on Nautilus

Containerization
Nautilus Gitlab



Create an account on Nautilus GitLab

Note: You may need to have your university system added by NRP Admins to use NRP GitLab



Nautilus Gitlab
create custom image

Create new repository

- Code & other files can be private, but the repository and the container registry MUST be public
- Open the repository in the Web IDE or clone it locally

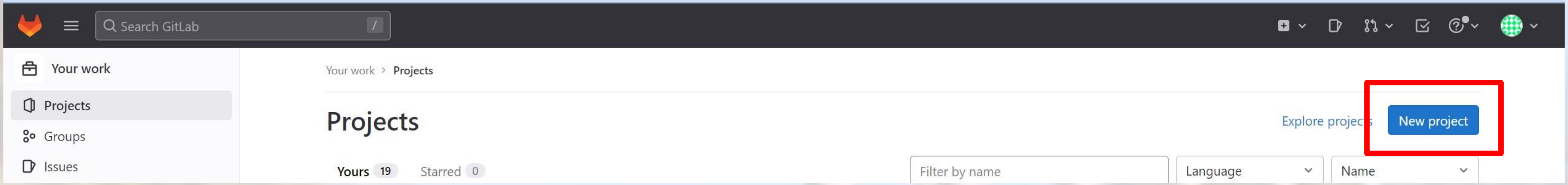
Create Gitlab CI YAML File

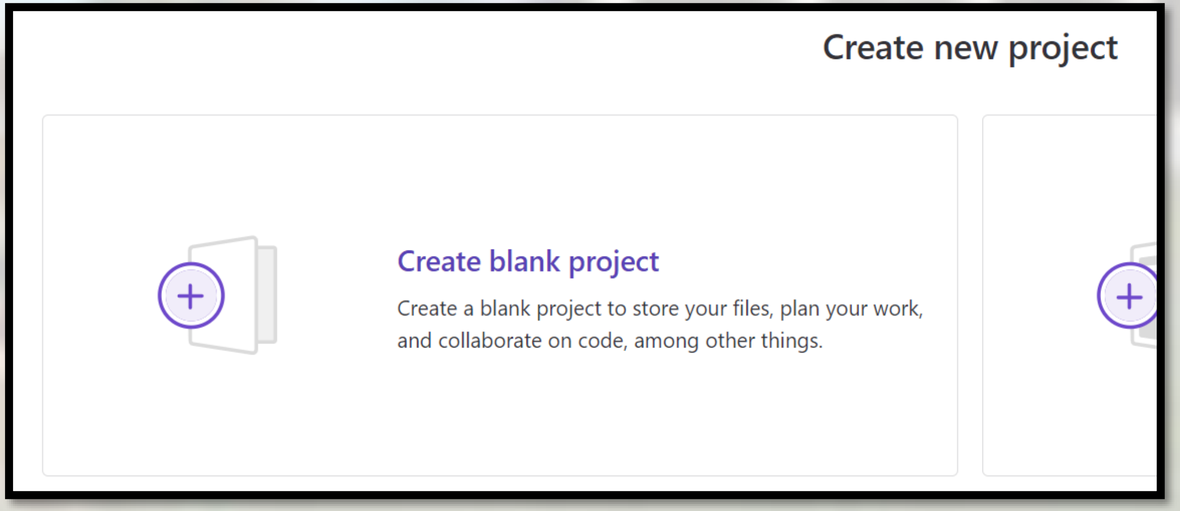
- Create a file called .gitlab-ci.yml
- Copy its content from this link: [.gitlab-ci.yml]
- Commit and push to your repository

Create Dockerfile

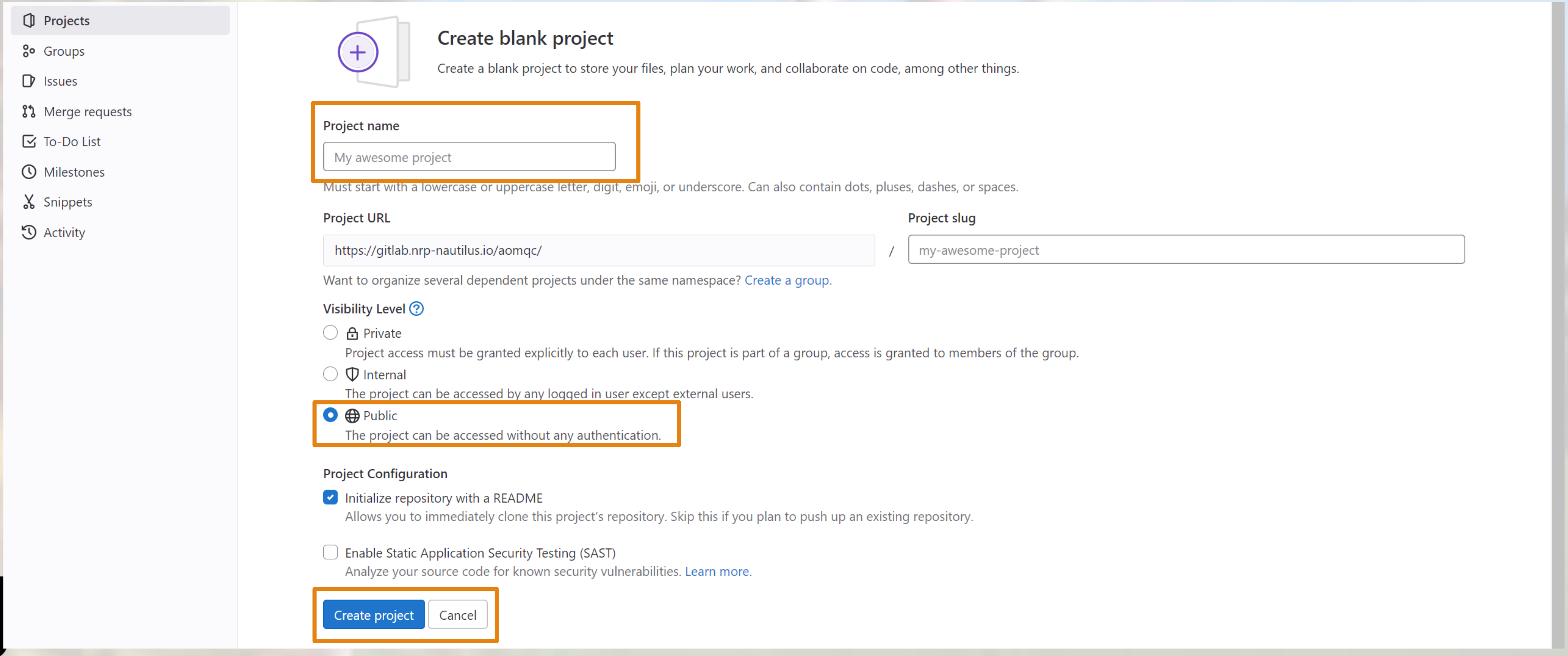
- Write the appropriate commands to build the image you need
- Use a community container as your base
- Commit and push to your repository

Nautilus Gitlab
create custom image



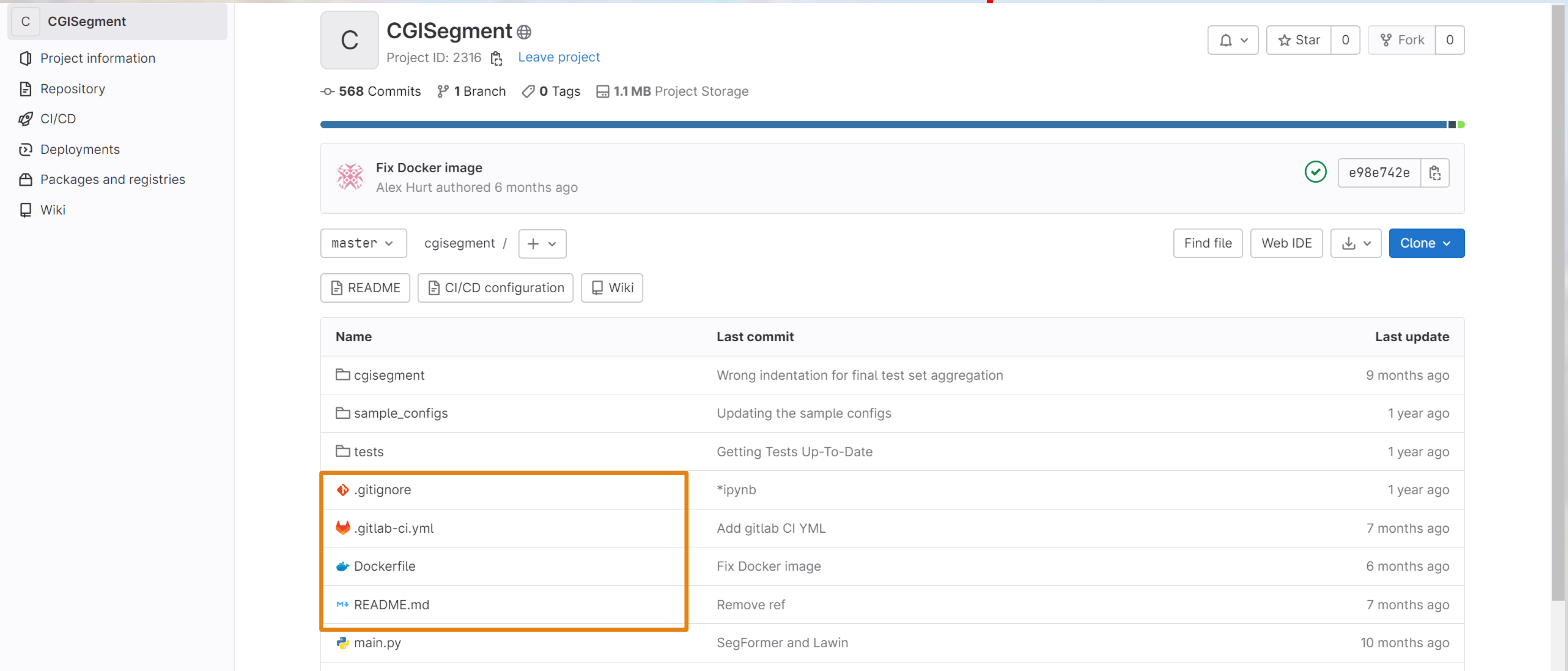


Nautilus Gitlab
create custom image



Nautilus Gitlab

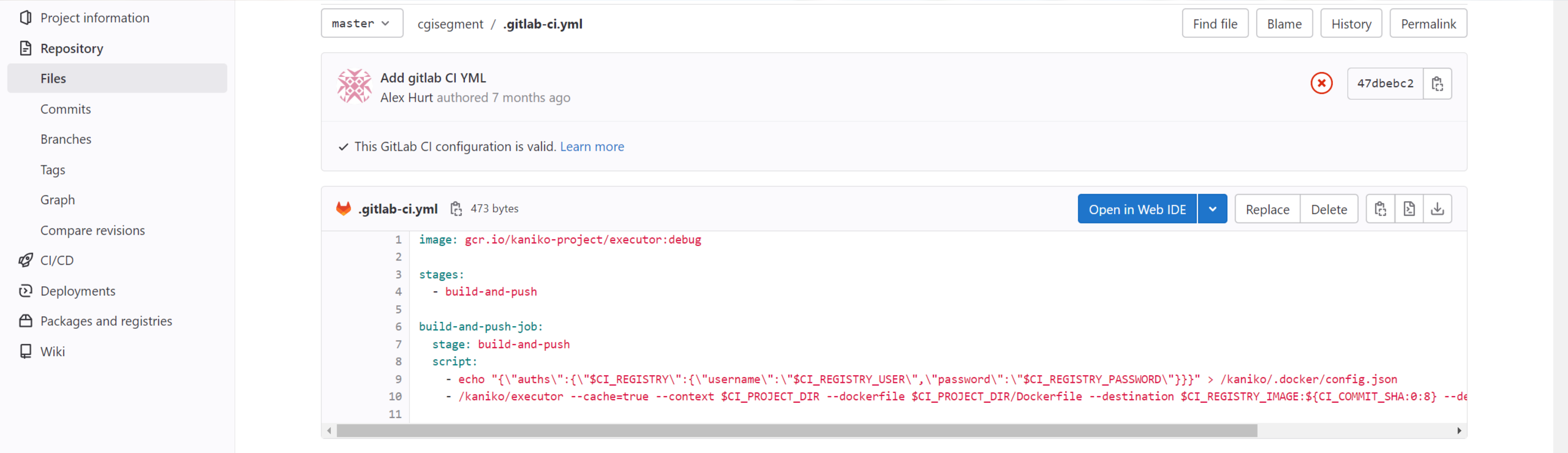
create custom image



Nautilus Gitlab
create custom image

Note: The content of .gitlab-ci.yml in the screenshot below is incorrect. The correct contents of this file can be found at this link:

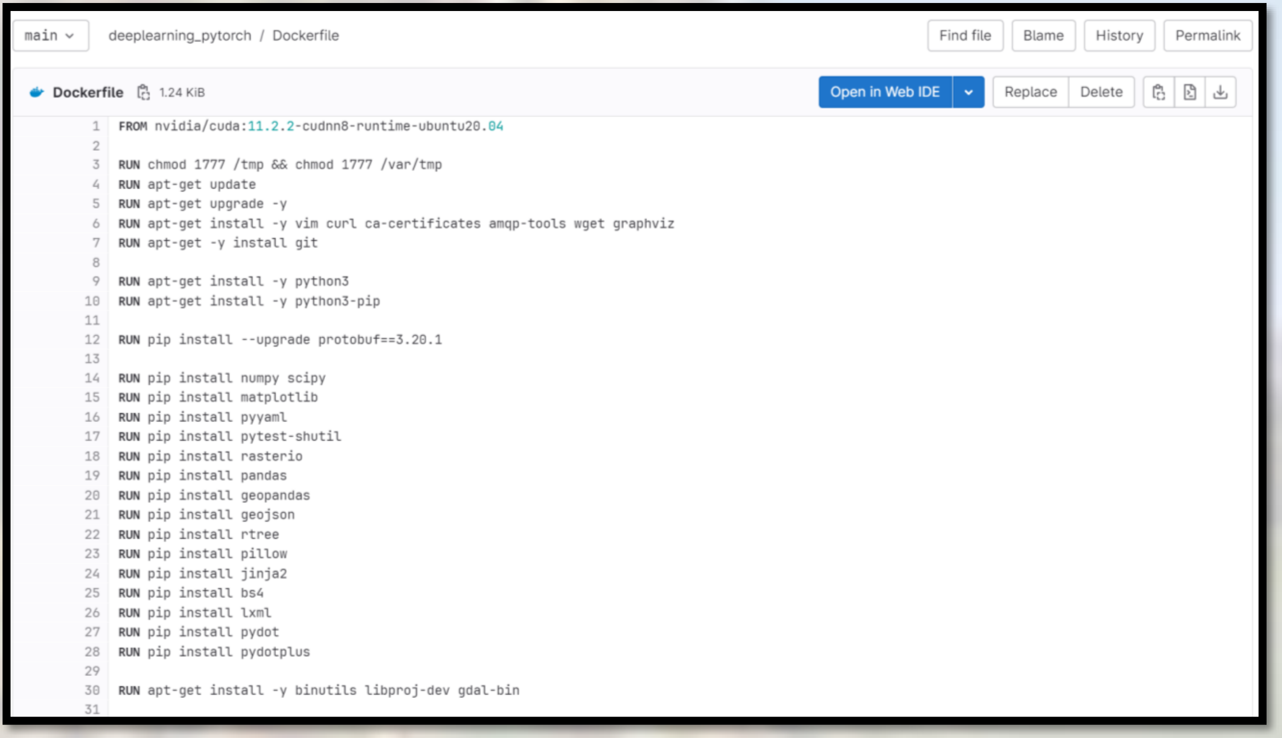
https://ucsd-prp.gitlab.io/userdocs/development/gitlab/#step-3-continuous-integration-automation



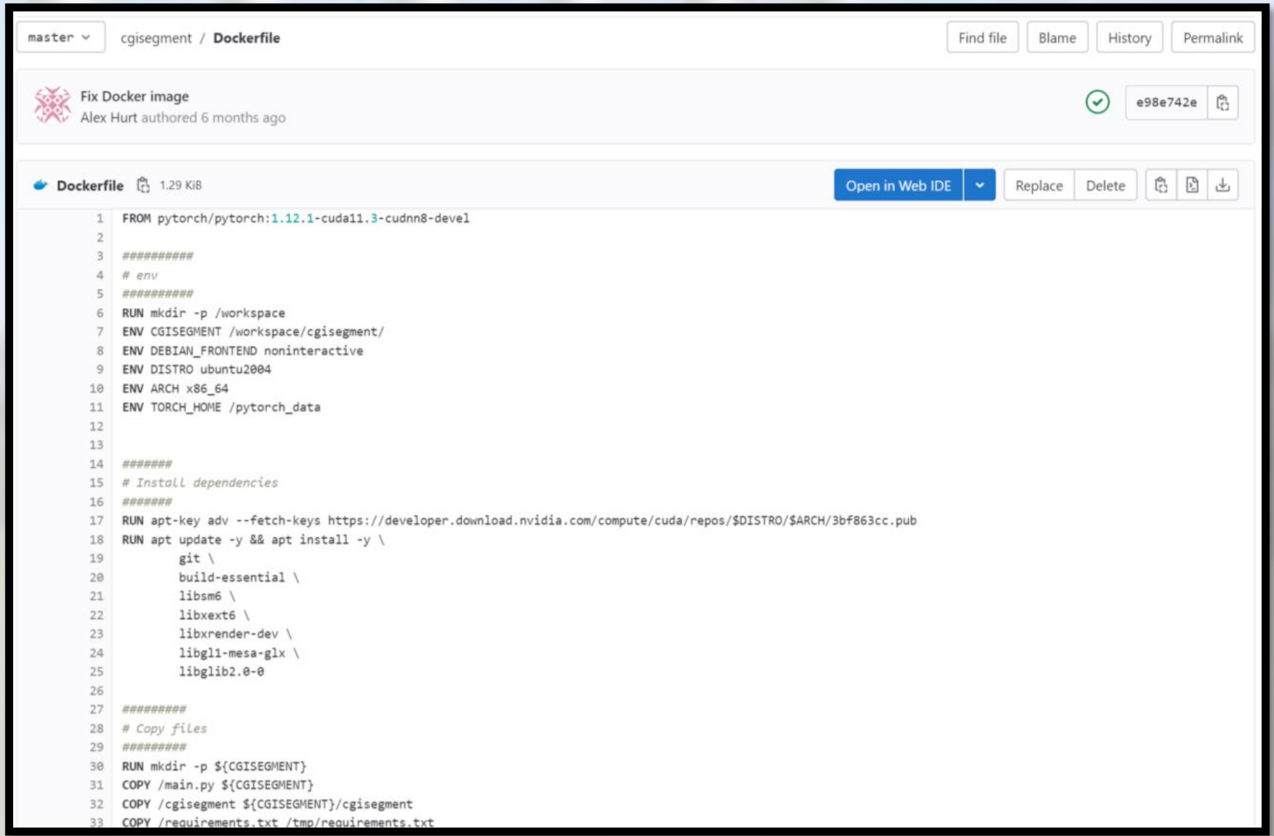
Nautilus Gitlab

create custom image

Two different styles of writing Docker file


main / deeplearning_pytorch / Dockerfile — Dockerfile 1.24 KiB

```
FROM nvidia/cuda:11.2.2-cudnn8-runtime-ubuntu20.04

RUN chmod 1777 /tmp && chmod 1777 /var/tmp
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y vim curl ca-certificates amqp-tools wget graphviz
RUN apt-get -y install git

RUN apt-get install -y python3
RUN apt-get install -y python3-pip

RUN pip install --upgrade protobuf==3.20.1

RUN pip install numpy scipy
RUN pip install matplotlib
RUN pip install pyyaml
RUN pip install pytest-shutil
RUN pip install rasterio
RUN pip install pandas
RUN pip install geopandas
RUN pip install geojson
RUN pip install rtree
RUN pip install pillow
RUN pip install jinja2
RUN pip install bs4
RUN pip install lxml
RUN pip install pydot
RUN pip install pydotplus

RUN apt-get install -y binutils libproj-dev gdal-bin

```


master / cgisegment / Dockerfile — Fix Docker image, Alex Hurt authored 6 months ago, e98e742e — Dockerfile 1.29 KiB

```
FROM pytorch/pytorch:1.12.1-cuda11.3-cudnn8-devel

##########
# env
##########
RUN mkdir -p /workspace
ENV CGISEGMENT /workspace/cgisegment/
ENV DEBIAN_FRONTEND noninteractive
ENV DISTRO ubuntu2004
ENV ARCH x86_64
ENV TORCH_HOME /pytorch_data


#######
# Install dependencies
#######
RUN apt-key adv --fetch-keys https://developer.download.nvidia.com/compute/cuda/repos/$DISTRO/$ARCH/3bf863cc.pub
RUN apt update -y && apt install -y \
        git \
        build-essential \
        libsm6 \
        libxext6 \
        libxrender-dev \
        libgl1-mesa-glx \
        libglib2.0-0

#########
# Copy files
#########
RUN mkdir -p ${CGISEGMENT}
COPY /main.py ${CGISEGMENT}
COPY /cgisegment ${CGISEGMENT}/cgisegment
COPY /requirements.txt /tmp/requirements.txt
```

Nautilus Gitlab
create custom image

- To utilize the GPUs in a cluster, GPU-enabled image is a MUST
- GPU containers can be built from community developed and published base containers, such as nvidia/cuda and the NVIDIA NGC
- CUDA GPU base images comes in three variations:
  - base
  - runtime
  - devel

Nautilus Gitlab
create custom image

base:

Contains the bare minimum (libcudart) to deploy a pre-built CUDA application.

Use this image if you want to manually select which CUDA packages you want to install.

runtime:

extends the base image by adding all the shared libraries from the CUDA toolkit.

Use this image if you have a pre-built application using multiple CUDA libraries.

devel:

extends the runtime image by adding the compiler toolchain, the debugging tools, the headers and the static libraries.

Use this image to compile a CUDA application from sources.

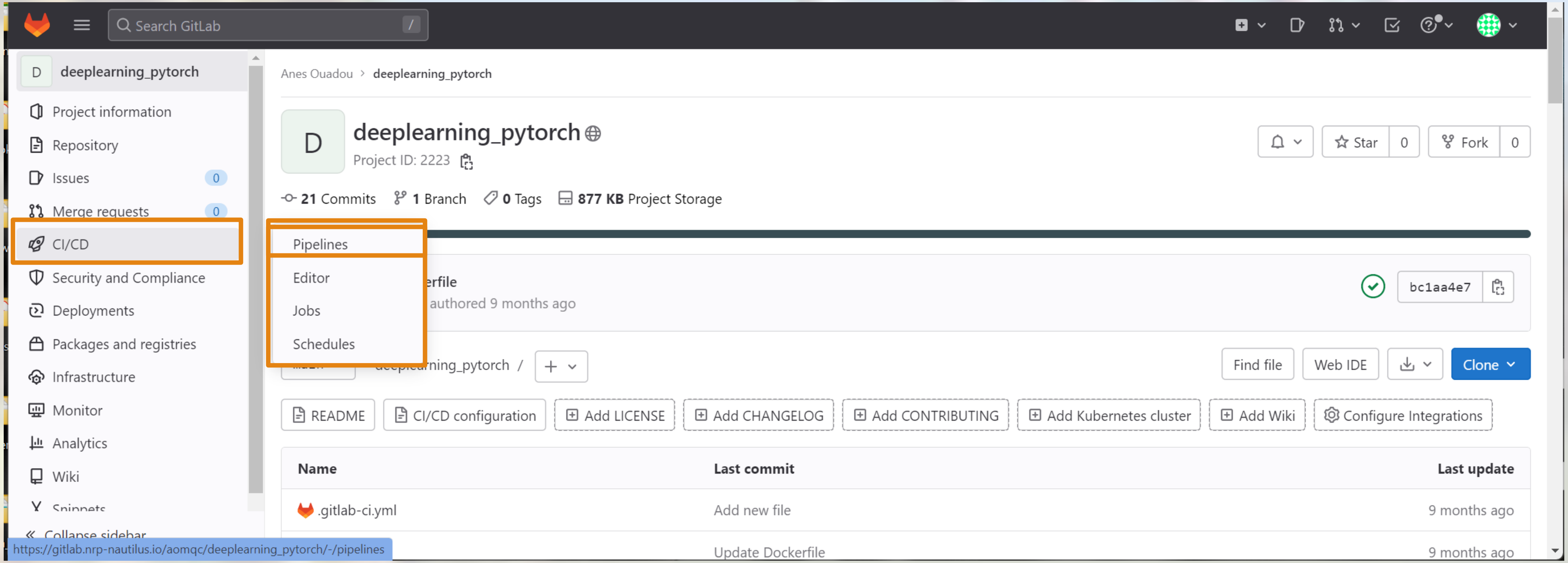
Nautilus Gitlab

create custom image

- NGC container
  - created by NVDIA
  - optimized tensor library for deep learning using GPUs and CPUs.
  - Specifically created for PyTorch based applications.
  - It facilitates the development of all kind of applications:
    - natural language processing (NLP)
    - recommenders
    - computer vision

Nautilus Gitlab

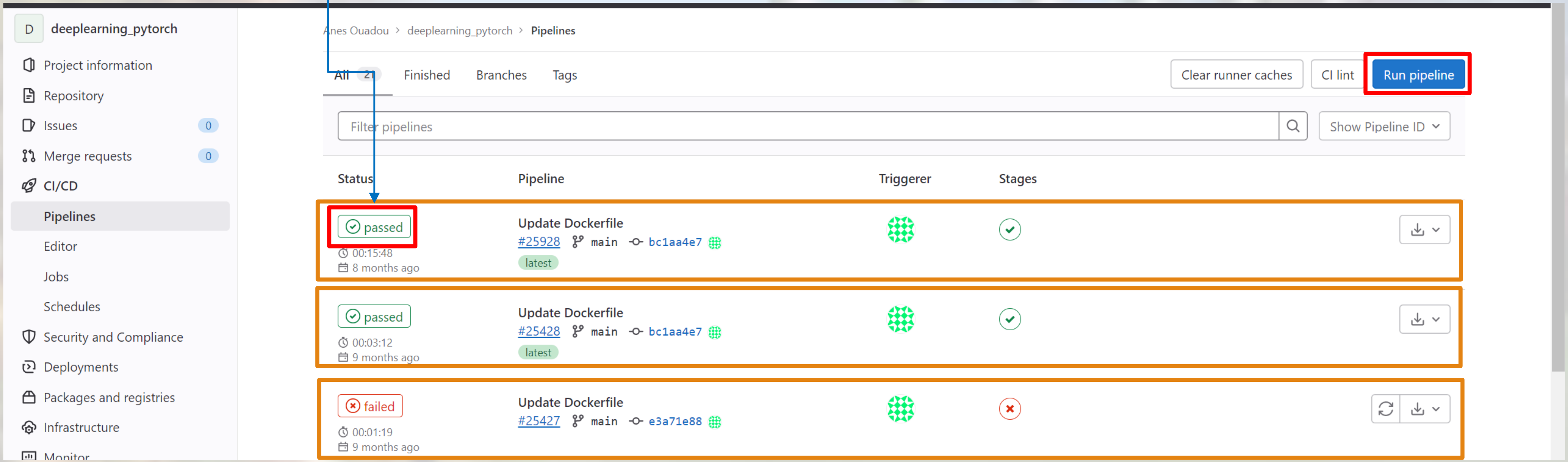
create custom image



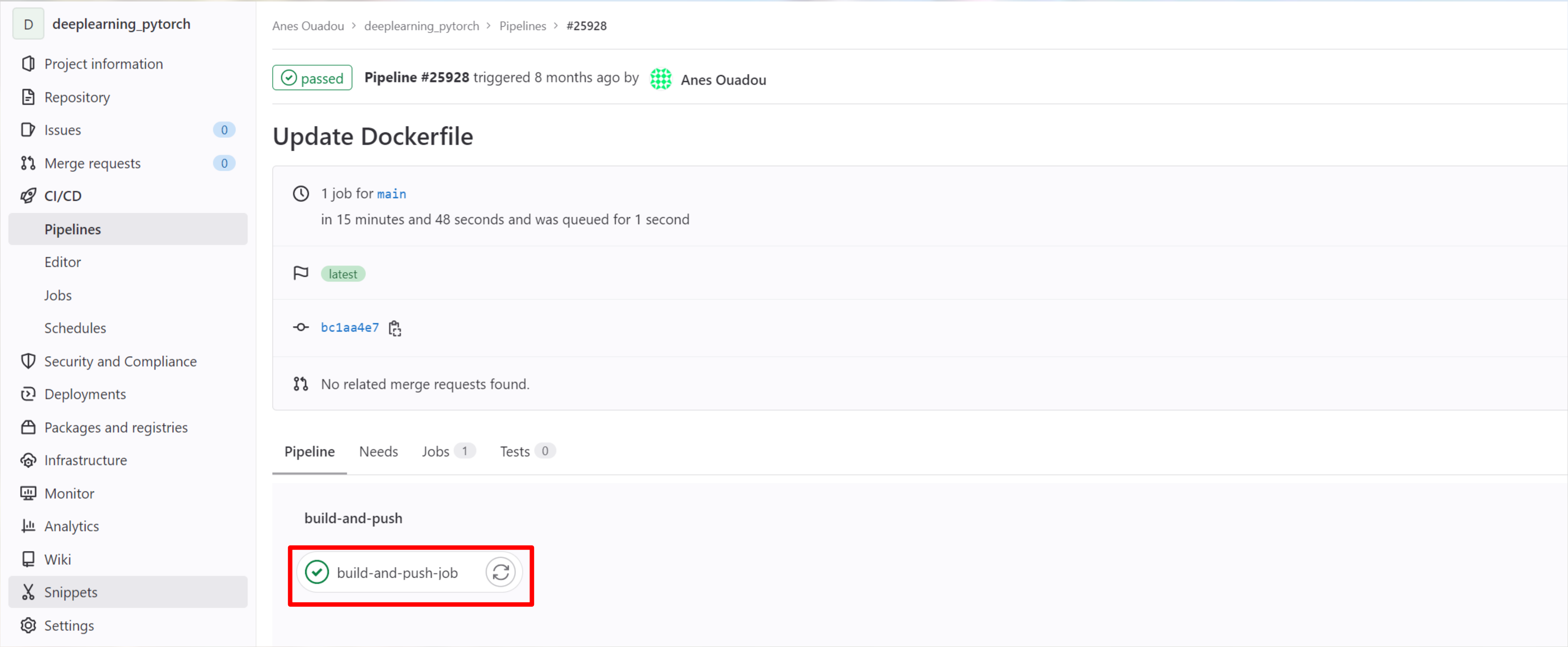
Nautilus Gitlab

create custom image

The image is being built the status is:
Running

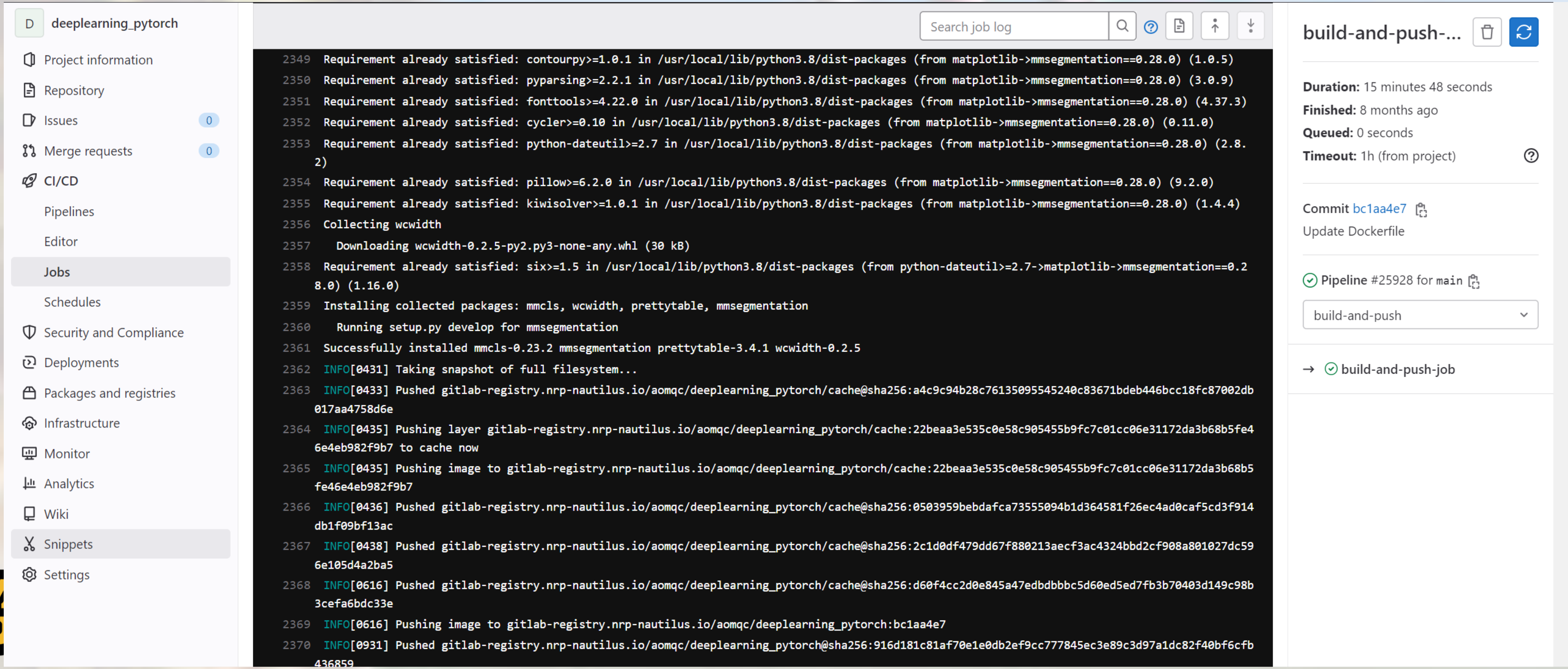


Nautilus Gitlab
create custom image



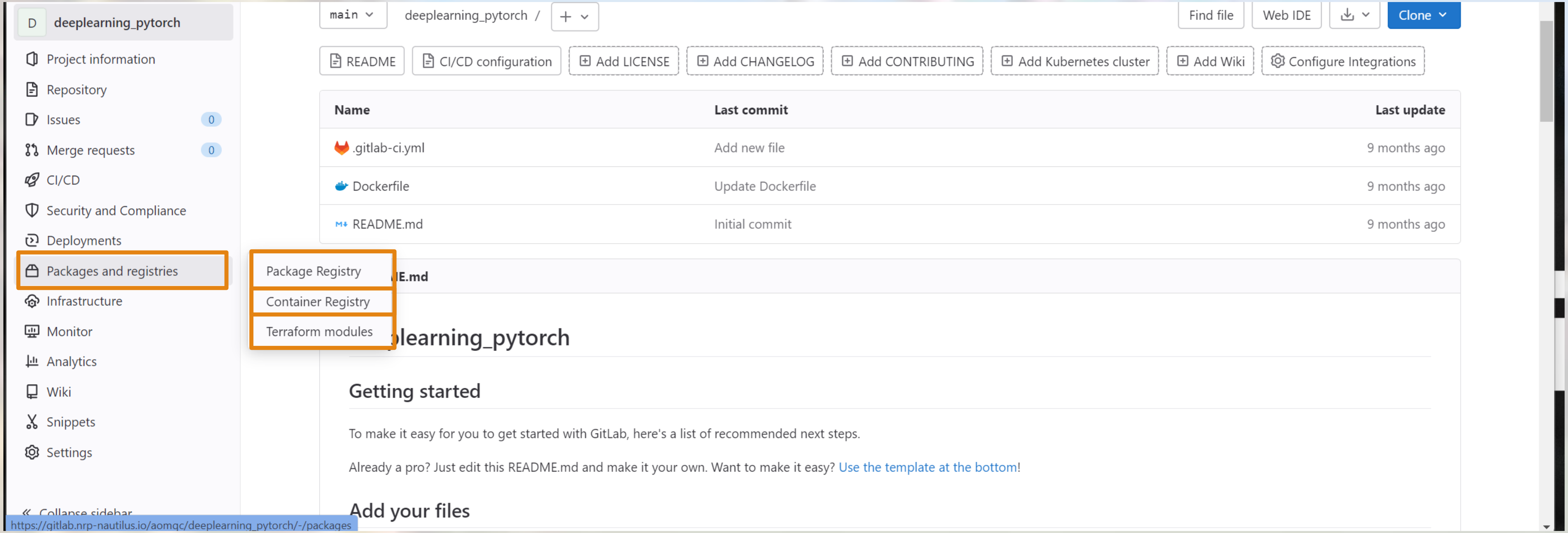
Nautilus Gitlab

create custom image



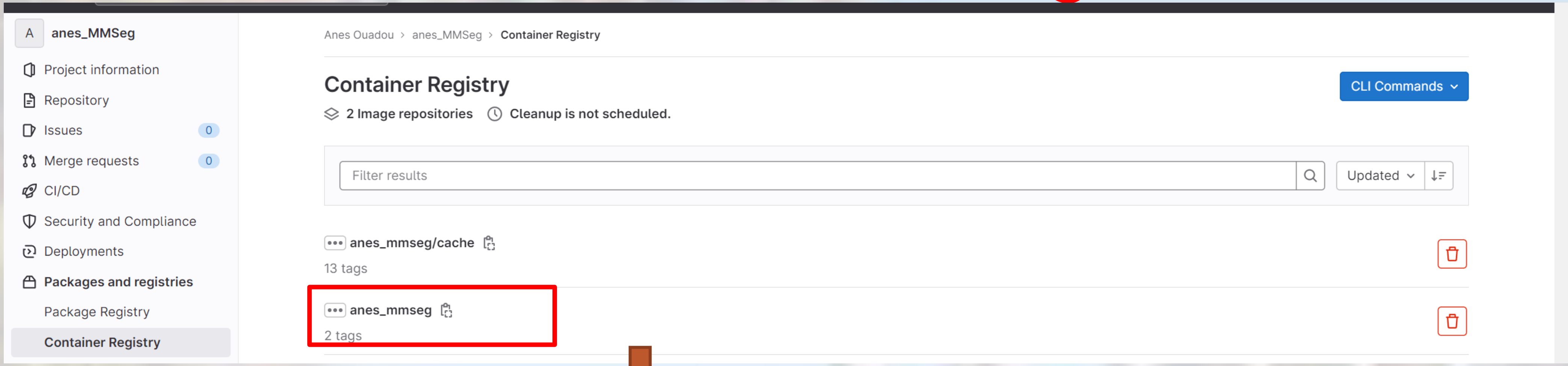
Nautilus Gitlab

create custom image



Nautilus Gitlab

create custom image



Nautilus Gitlab
create custom image

- The list of images will always have one image labeled latest and at least one image with a randomly generated tag

gitlab-registry.nrp-nautilus.io/aomqc/anes_mmseg:latest

gitlab-registry.nrp-nautilus.io/aomqc/anes_mmseg:4f76327d

Nautilus gitlab instance address	username	Image name	Image tag
gitlab-registry.nrp-nautilus.io/	aomqc/	anes_mmseg:	latest
gitlab-registry.nrp-nautilus.io/	aomqc/	anes_mmseg:	4f76327d

Note: Never use the latest tag in your YAML files, as it causes collision issues on nodes in the cluster

Using Common Deep Learning Frameworks

Open MMLab, FAIR, and Ultralytics

Deep Learning Frameworks

- Many companies and research labs are developing open source Deep Neural Network frameworks to perform Computer Vision tasks
  - Open MMLab
  - Facebook AI Research
  - Ultralytics
  - Google
- We can build optimized Docker containers with these frameworks and use them as a baseline for research
- To use a given framework on Nautilus, we only need a Dockerfile → From the Dockerfile, we can build a container image that can be deployed on the cluster

ultralytics

Meta AI

MMLab Frameworks: MMDetection, MMClassification, MMSegmentation

- Open MMLab has developed a set of extensible libraries for performing key Computer Vision tasks:
  - Classification: MMClassification
  - Object Detection: MMDetection
  - Semantic Segmentation: MMSegmentation
- These libraries can serve as an excellent starting point for many CV applications
- Very large model zoo with community trained models and benchmarks
- Highly extensible and configurable frameworks

MMDetection

MMSegmentation

MMClassification

MMDetection Dockerfile

- Dockerfile contains all steps to create a fully functional MMDetection Python environment
- Add your code to the container or use this Dockerfile as a base in a multi-stage build

MU-HPDI/Nautilus

University of Missouri

```
FROM nvcr.io/nvidia/pytorch:22.06-py3

# arg
ARG MMDET_VERSION=2.25.0
ARG MMCV_VERSION=1.5.2
ARG MMCLS_VERSION=0.23.1
ARG BUILD_DIR="/build"

# env variables
ENV MPLCONFIGDIR /tmp
ENV TORCH_HOME /tmp
ENV MMCV_WITH_OPS 1
ENV FORCE_CUDA 1
ENV CUDA_HOME /usr/local/cuda

# create the build dir
RUN mkdir -p ${BUILD_DIR}

RUN apt update -y && apt install -y libpng-dev libjpeg-dev libgl-dev wget && rm -rf /var/lib/apt/lists/*
# install mm cv
# https://mmcv.readthedocs.io/en/latest/get_started/build.html#build-on-linux-or-macos
RUN wget https://github.com/open-mmlab/mmcv/archive/refs/tags/v${MMCV_VERSION}.tar.gz -O /tmp/mmcv.tar.gz
RUN tar -xzf /tmp/mmcv.tar.gz
RUN mv ./mmcv-${MMCV_VERSION} ${BUILD_DIR}/mmcv
# install it
RUN pip install -r ${BUILD_DIR}/mmcv/requirements/optional.txt
RUN pip install -v ${BUILD_DIR}/mmcv


# install mm detection
# https://github.com/open-mmlab/mmdetection/blob/v2.25.0/docs/en/get_started.md#customize-installation
RUN wget https://github.com/open-mmlab/mmdetection/archive/refs/tags/v${MMDET_VERSION}.tar.gz  -O /tmp/mmdet.tar.gz
RUN tar -xzf /tmp/mmdet.tar.gz
RUN mv ./mmdetection-${MMDET_VERSION} ${BUILD_DIR}/mmdet
# install it
RUN pip install ${BUILD_DIR}/mmdet
```

Detectron2

- Detectron2 is an open source framework for Object Detection and Instance Segmentation developed by FAIR
- The original Mask R-CNN model was developed and published via this framework
- Highly extensible with custom components and highly configurable
- High quality pretrained models and dataset benchmarks

Meta AI

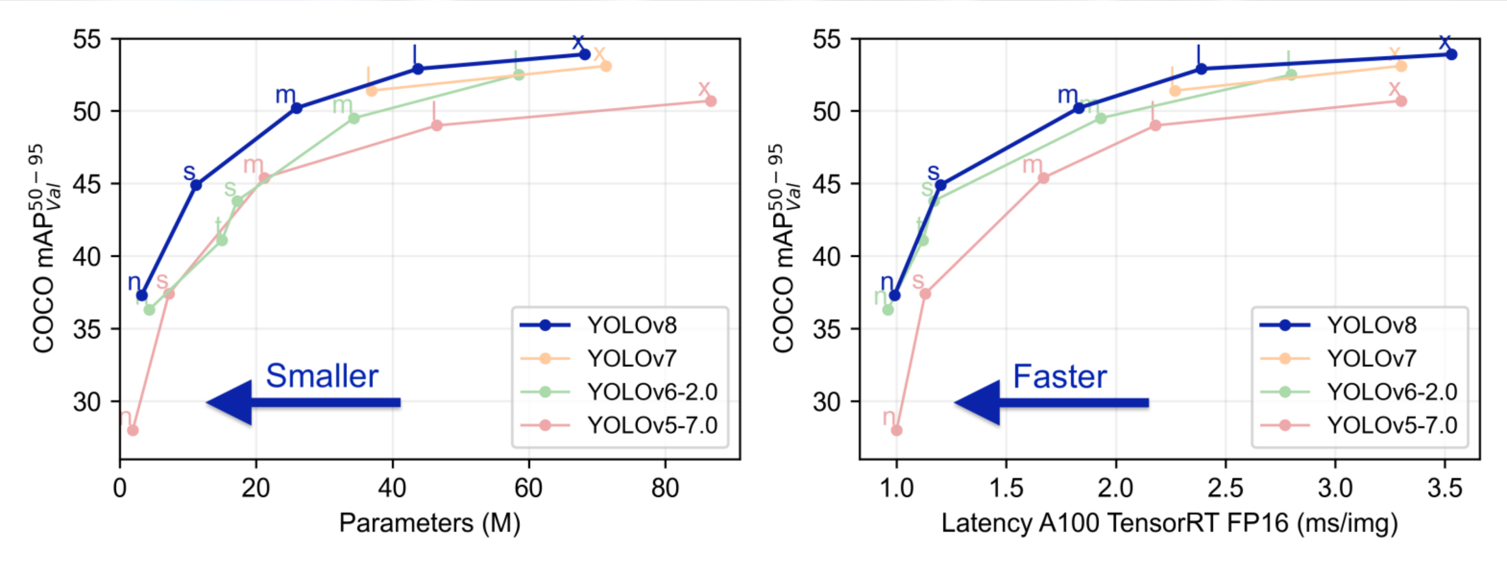
Detectron2 Dockerfile

- Dockerfile contains all steps to create a fully functional Detectron2 Python environment
- Add your code to the container or use this Dockerfile as a base in a multi-stage build

```
FROM nvcr.io/nvidia/pytorch:21.06-py3

# install system reqs
RUN apt update && apt install -y vim libgl-dev
RUN apt-get install --reinstall ca-certificates # for git

# env variables
ENV MPLCONFIGDIR /tmp
ENV TORCH_HOME /tmp
ENV FVCORE_CACHE /tmp


##########
# Detectron 2
#########
RUN git clone -b 'v0.4' --single-branch --depth 1 --recursive https://github.com/facebookresearch/detectron2.git /workspace/detectron2
RUN pip install -v  /workspace/detectron2/
```

Ultralytics

- Developer of most used Open-Source implementation of YOLOv3, YOLOv5, and more recently, YOLOv8
- You Only Look Once (YOLO) is a common real-time object detection architecture with many variations over the years
  - YOLO
  - YOLO9000 (a.k.a. YOLOv2)
  - YOLOv3
  - YOLOv5
  - YOLOv8



Ultralytics Dockerfile

- Dockerfile contains all steps to create a fully functional YOLOv3, YOLOv5, or YOLOv8 Python environment
- Add your code to the container or use this Dockerfile as a base in a multi-stage build

ultralytics/ultralytics

University of Misso

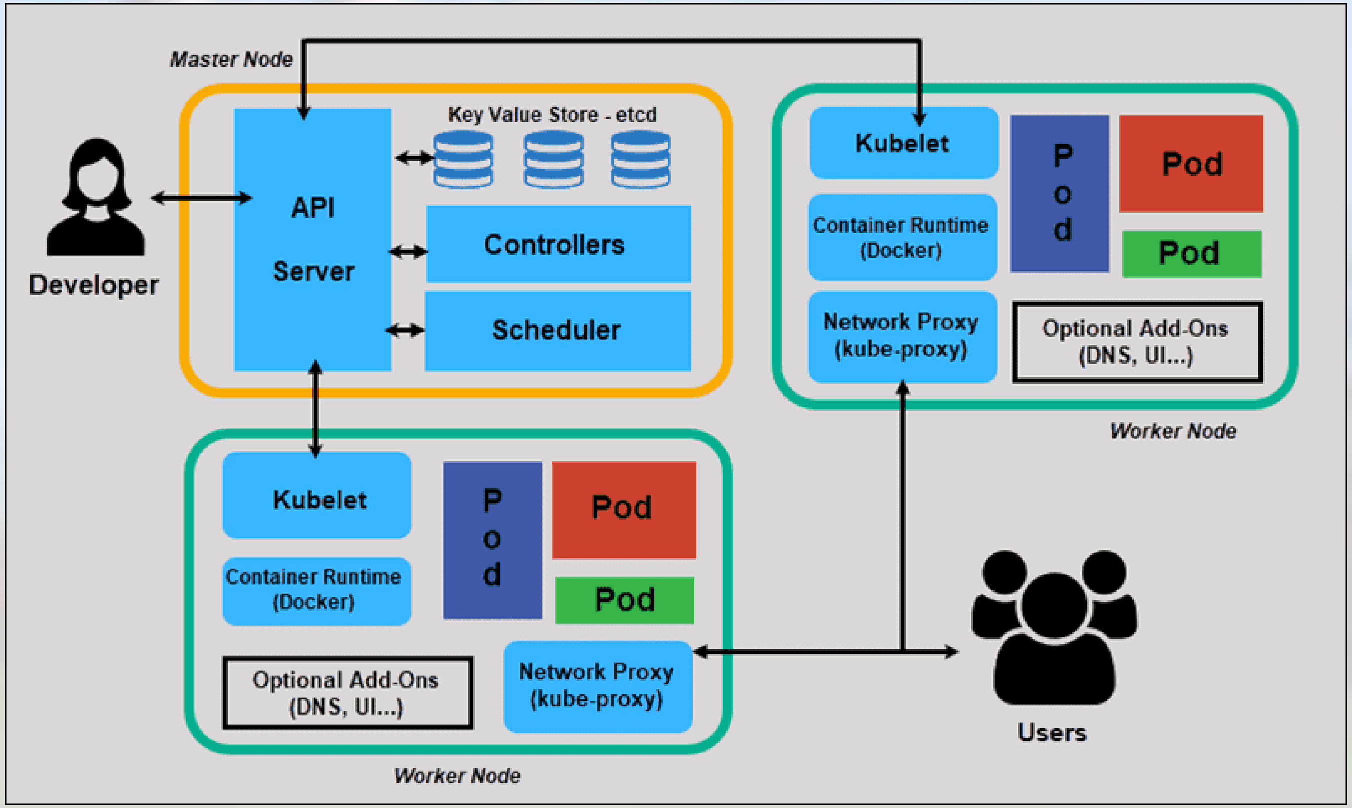
```
# Ultralytics YOLO 🚀, AGPL-3.0 license
# Builds ultralytics/ultralytics:latest image on DockerHub https://hub.docker.com/r/ultralytics/ultralytics
# Image is CUDA-optimized for YOLOv8 single/multi-GPU training and inference

# Start FROM PyTorch image https://hub.docker.com/r/pytorch/pytorch or nvcr.io/nvidia/pytorch:23.03-py3
FROM pytorch/pytorch:2.0.0-cuda11.7-cudnn8-runtime

# Downloads to user config dir
ADD https://ultralytics.com/assets/Arial.ttf https://ultralytics.com/assets/Arial.Unicode.ttf /root/.config/Ultralytics/

# Install linux packages
# g++ required to build 'tflite_support' package
RUN apt update \
    && apt install --no-install-recommends -y gcc git zip curl htop libgl1-mesa-glx libglib2.0-0 libpython3-dev gnupg g++
# RUN alias python=python3

# Security updates
# https://security.snyk.io/vuln/SNYK-UBUNTU1804-OPENSSL-3314796
RUN apt upgrade --no-install-recommends -y openssl tar

# Create working directory
RUN mkdir -p /usr/src/ultralytics
WORKDIR /usr/src/ultralytics

# Copy contents
# COPY . /usr/src/app  (issues as not a .git directory)
RUN git clone https://github.com/ultralytics/ultralytics /usr/src/ultralytics
ADD https://github.com/ultralytics/assets/releases/download/v0.0.0/yolov8n.pt /usr/src/ultralytics/

# Install pip packages
RUN python3 -m pip install --upgrade pip wheel
RUN pip install --no-cache -e . albumentations comet tensorboard thop pycocotools

# Set environment variables
ENV OMP_NUM_THREADS=1
```

NRP Nautilus (Review)

Kubernetes Architecture and Key Concepts

Kubernetes

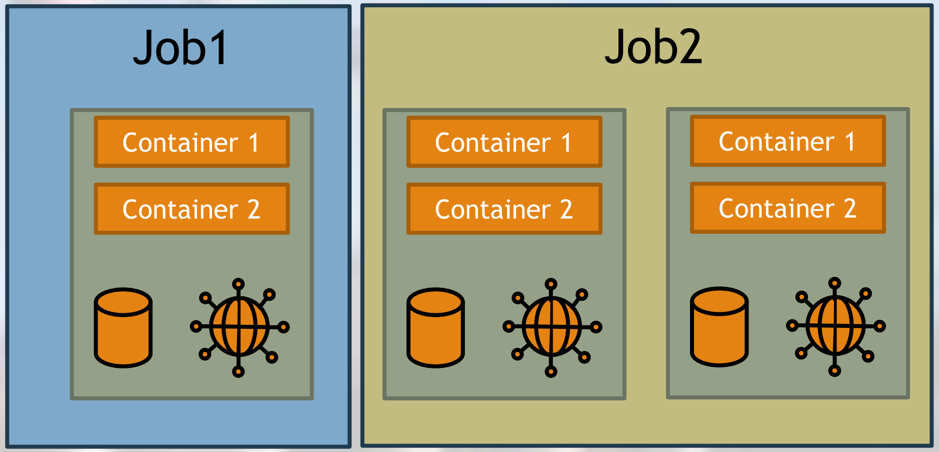
- Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.[1]
- Kubernetes enables both simple and complex container orchestration
- Kubernetes cluster has two main components
  - Master node
  - Worker node

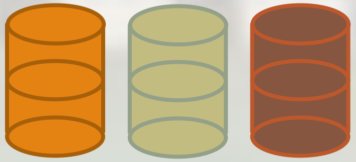


1. https://kubernetes.io/
2. Image: https://phoenixnap.com/kb/understanding-kubernetes-architecture-diagrams
3. Logo: https://commons.wikimedia.org/wiki/File:Kubernetes_logo_without_workmark.svg

Key Kubernetes Concepts
ReplicaSet and Deployment

- **Pods** - are the basic scheduling unit of K8s.
- **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]
- **Jobs** - creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate.[1]
- **Persistent volume -** is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using storage classes



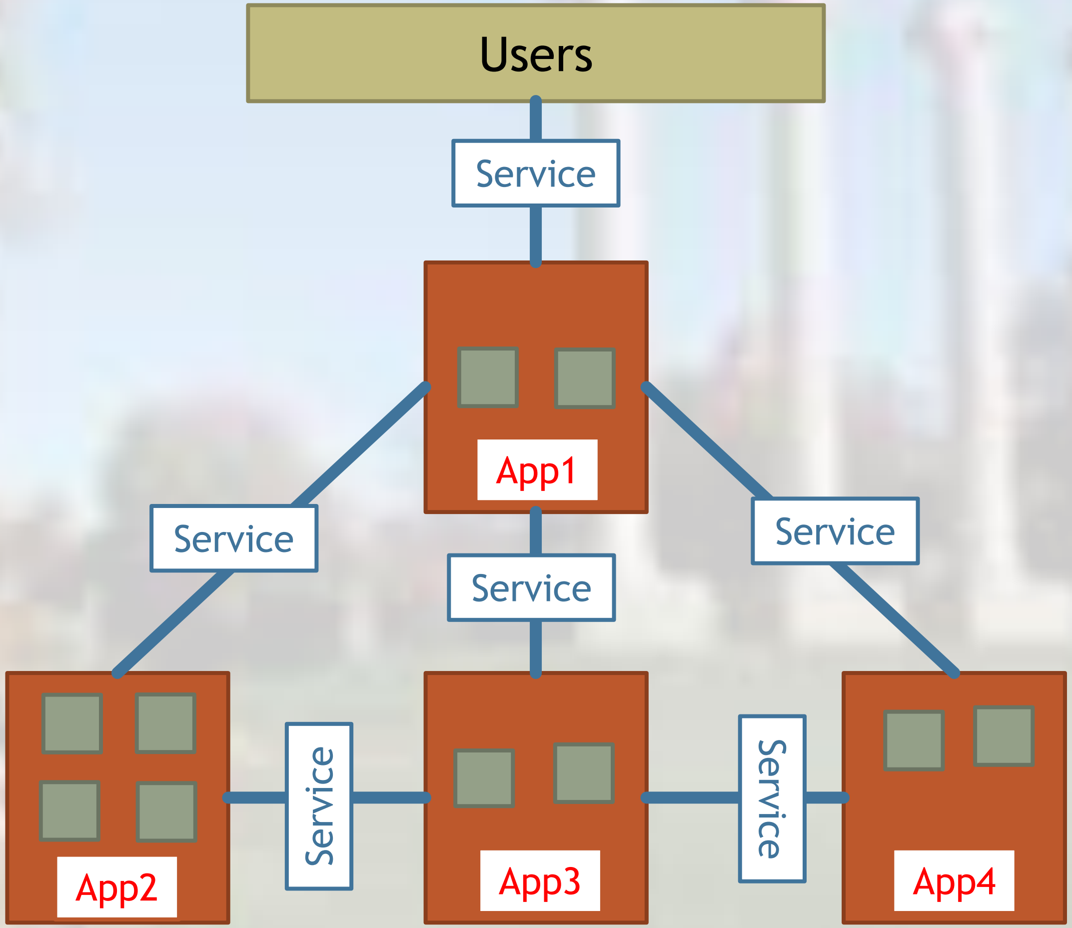


- Different classes
  - cephfs
  - NFS storage
- Different access mode
  - ReadWriteOnce
  - ReadOnlyMany
  - ReadWriteMany

1. https://kubernetes.io/

Key Kubernetes Concepts
Services

Each Pod has a unique IP address which changes every time a Pod is dead and restarted, this render communication hard



Services enable communication between application running in pods within the cluster and with outside users if necessary

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

Config file (YAML)
Pod with 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 10 GB of RAM

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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: pod-name-sso
spec:
  containers:
  - name: pod-name-sso
    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
```

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: 14,462
  - RAM: 69 TB
  - NVIDIA GPUs: 1150

University of Missouri

Nautilus Logo: https://portal.nrp-nautilus.io/
Docker Logo: https://developers.redhat.com/blog/2014/05/15/practical-introduction-to-docker-containers
Kubernetes Logo: https://commons.wikimedia.org/wiki/File:Kubernetes_logo_without_workmark.svg

Migrating Data to Nautilus

Deep Learning & Data

- Deep Learning algorithms require a vast amount of representative training data to effectively train
  - The more complex the network architecture, the more quality data is required
- Previous portions of this workshop have covered creating the containers to train the algorithms, but we have not yet covered how to stage data to Nautilus
  - All data for use on Nautilus will need to be moved to *persistent volumes* on the cluster
- Three ways to move data to a persistent volume on Nautilus:
  - Using KubeCTL
  - Using Commercial Cloud Storage
  - **Using Nautilus S3 Storage** ***← Recommended***

Migrating Data to Nautilus: KubeCTL

- The command line Kubernetes tool, KubeCTL, has functionality to copy data to and from running pods
  - We can use this copy utility to move data to the cluster

- Advantages:
  - No additional installation or setup
  - No need to stage data in cloud storage
  - Requires only the KubeCTL command line tool

- Disadvantages:
  - Only small amounts (< 100 MB) of data can be moved per kubectl copy command
  - KubeCTL copy is *slow* at < 100 Mbps upload speeds

- How to:

```
kubectl cp localpath podname:/path/on/pod
```

Migrating Data to Nautilus: Using Commercial Cloud Storage

- We can use commercial cloud storage, such GCP Buckets or AWS S3 to move data to Nautilus

- Advantages:
  - Flexibility of cloud platform
  - Very fast transfer speeds
  - Capable of virtually any amount of data

- Disadvantages
  - Cost
  - Setup of Cloud Storage
  - Installation of Cloud Interface
  - Staging of data in cloud

- How to:
  - Create cloud bucket with Commercial Cloud Vendor (i.e., GCP)
  - Copy data to Bucket: gsutil cp localpath gs://bucketName
  - Create Google Cloud Pod on Nautilus
  - Copy data from Bucket: gsutil cp gs://bucketName /path/on/pod

Google Cloud

Amazon S3

MU-HPDI/Nautilus

Migrating Data to Nautilus: Using Nautilus S3 Storage

MU-HPDI/Nautilus

- NRP provides S3 bucket storage for free to Nautilus users upon request
- Advantages:
  - Free
  - High-throughput link to Nautilus cluster
  - Integration with S3-compatible software
  - Capability to handle large amounts of data
- Disadvantages:
  - Must request access
  - Setup of cloud integration
  - Staging of data into Nautilus S3
- How To:
  - Request access to cloud storage to receive Access ID and Keys
  - Install rclone or similar tool at data source and copy data from source to Nautilus S3
  - Create rclone or similar tool pod on Nautilus cluster and copy from S3 to Persistent Volume

High-Performance Data-Intensive Computing Systems Laboratory

Part 2-B

Deploying Scikit-Learn Jobs to Nautilus for Machine Learning

Introduction to Containers and Kubernetes

IEEE International Conference on Big Data

December 2023

University of Missouri

Prerequisites

- To run jobs on Nautilus, the following prerequisites must be met:
  - You have access to Nautilus and have been assigned a namespace
  - You have a container published to a public registry with the necessary code to perform the ML task
  - You have the data for the ML task on a persistent volume in the cluster

Access

ML Container

Data

Access the MachineLearning_K8s Notebook

Follow along activity

Job Specification Format

- We will step through each of these sub-keys in the following slides
- Keep in mind key distinctions in the format between pods and jobs: the *template* key

```
spec:
  backoffLimit
  template:
    spec:
      restartPolicy
      containers:
        - name
          image
          workingDir
          command
          imagePullPolicy
          ports
          env
          requests
          volumeMounts
      volumes:
        - name
          persistentVolumeClaim
        - name
          emptyDir
      affinity
```

MU-HPDI/Nautilus

Building the Job Specification: MetaData and Setup

- We initialize our job specification file by setting the API Version and setting our resource type to a Job
- We then set a Job Name
- Next, we begin the job specification and set a backoff limit
- We then create our template key followed by another specification within the template
- Finally, we set the restart policy

```
apiVersion: batch/v1
kind: Job

metadata:
  name: my-job-name

spec:
  backoffLimit: 0
  template:
    spec:
          restartPolicy: Never
```

High-Performance Data-Intensive Computing Systems Laboratory

Part 2-C

Deploying GPU Jobs to Nautilus for Computer Vision Applications

Introduction to Containers and Kubernetes

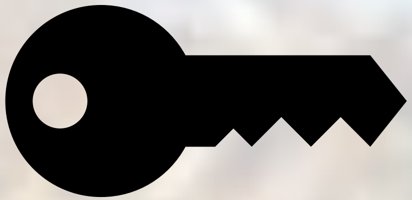
IEEE International Conference on Big Data

December 2023

University of Missouri

Prerequisites – Review + GPU differences

- To run GPU jobs on Nautilus, the following prerequisites must be met:
  - You have access to Nautilus and have been assigned a namespace
  - You have a **GPU enabled** container published to a public registry with the necessary code to perform the CV task
  - You have the data for the CV task on a persistent volume in the cluster



Access

GPU Container

Data

Job Specification Format (Review)

- We will step through each of these sub-keys in the following slides
- Keep in mind key distinctions in the format between pods and jobs: the *template* key
- Full example of running a GPU job and GPU pod for deep learning available on GitHub

MU-HPDI/Nautilus

```
spec:
 backoffLimit
 template:
  spec:
   restartPolicy
   containers:
    - name
     image
     workingDir
     command
     imagePullPolicy
     ports
     env
     requests
     volumeMounts
   volumes:
    - name
     persistentVolumeClaim
    - name
     emptyDir
   affinity
```

Building the Job Specification (Review): MetaData and Setup

- We initialize our job specification file by setting the API Version and setting our resource type to a Job
- We then set a Job Name
- Next, we begin the job specification and set a backoff limit
- We then create our template key followed by another specification within the template
- Finally, we set the restart policy

```yaml
apiVersion: batch/v1
kind: Job

metadata:
  name: my-job-name

spec:
  backoffLimit: 0
  template:
    spec:
        restartPolicy: Never
```

```yaml
containers:
 - name: my-container-name
   image: my-image
   workingDir: /path/to/mydir
   imagePullPolicy: IfNotPresent
   command: ["python", "train.py", "/data/train_cfg.py"]
   ports:
    - containerPort: 8880
   env:
    - name: CUDA_VISIBLE_DEVICES
      value: "0,1,2,3"
   resources:
    limits:
      memory: 64Gi
      cpu: 32
      nvidia.com/gpu: 4
    requests:
      memory: 8Gi
      cpu: 8
      nvidia.com/gpu: 4
```

Building the Job Spec: Creating the Container

- For our job, we will use a single container
- We set the name (name) and the container image to use for this container (image)
  - Container image must be publicly accessible
- We set the working directory when the container starts (workingDir)
- We set the command to run when the container starts (command)
- We indicate the ports that should be open on the container (ports) as well as any environment variables we want to be present at runtime (env)
- Finally, we set the minimum (requests) and maximum (limits) amount of resources our container will need (resources)

Building the Job Spec: Mounting Data

- We need to set the Volume Mounts to the container (volumeMounts)
- We then need to specify the volumes that we will use under the spec key
  - The name of the volume mount in the volumeMounts must match the name of one of the specified volumes
  - The claim name is the PVC name given at PVC creation time
- We add an additional volume mount to deep learning containers utilizing PyTorch's Distributed Data Parallel: a shared volume mount to allow for IPC

```
spec:
 template:
  spec:
   containers:
     - name: my-container
      ...
      volumeMounts:
       - mountPath: /data
        name: my-pvc
       - mountPath: /dev/shm
        name: dshm
    volumes:
     - name: my-pvc
      persistentVolumeClaim:
       claimName: my-pvc
     - name: dshm
      emptyDir:
       medium: Memory
```

Building the Job Spec: Setting Node Affinity

- Node affinity allows us to specify what characteristics we need for the node that is assigned for our job
- Most commonly, node affinity is used to set the type(s) of GPUs for jobs
- Best practice: take the ***least*** powerful GPU for what you need

```yaml
spec:
 template:
  spec:
   containers:
     - name: my-container
      ...
      volumeMounts:
       - mountPath: /data
        name: my-pvc
       - mountPath: /dev/shm
        name: dshm
   volumes:
    - name: my-pvc
     persistentVolumeClaim:
      claimName: my-pvc
    - name: dshm
     emptyDir:
      medium: Memory
   affinity:
    nodeAffinity:
     requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
       - matchExpressions:
         - key: nvidia.com/gpu.product
          operator: In
          values:
           - NVIDIA-GeForce-RTX-3090
```

Starting the Job

- Once you have built a full Kubernetes Job Specification YAML file, you can begin your job with KubeCTL:

```
kubectl apply –f myJob.yaml
```

- Common Pitfalls:
  - Not ensuring that volume mount names and volume names match
  - Private container repositories
  - Using the latest tag on your deep learning container
  - Building a CPU only container for GPU code
- Your job may sit at PENDING state if:
  - You requested too many resources
  - You requested too powerful a GPU
- Your job may fail to start if:
  - There are any permission issues related to pulling the image
  - There is any exception or error thrown when the container attempts to start

Access the DeepLearning_K8s Notebook

Follow along activity

High-Performance Data-Intensive Computing Systems Laboratory

Part 2-D
Automating Jobs on Nautilus using Bash and Python

Introduction to Containers and Kubernetes

IEEE International Conference on Big Data

December 2023

University of Missouri

Automating GPU Jobs on Nautilus 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 }}-pretr
        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:

'''
```

Automating GPU Jobs on Nautilus

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
        fileout.write(output_file)
        fileout.close()
```

Automating GPU Jobs on Nautilus

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

```
@ECHO OFF

Rem This batch file executes kubectl commands to create training jobs

::echo %kubectl%
SET exp_list=2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18

(for %%a in  (%exp_list%) do (
    echo %%a

    kubectl create -f experiments\exp%%a/job_exp%%a_deeplab_img.yaml
    kubectl create -f experiments\exp%%a/job_exp%%a_deeplab_tci.yaml
    kubectl create -f experiments\exp%%a/job_exp%%a_deeplab_img_pretrained.yaml
    kubectl create -f experiments\exp%%a/job_exp%%a_deeplab_tci_pretrained.yaml

    kubectl create -f experiments\exp%%a/job_exp%%a_unet_img.yaml
    kubectl create -f experiments\exp%%a/job_exp%%a_unet_tci.yaml
    kubectl create -f experiments\exp%%a/job_exp%%a_unet_img_pretrained.yaml
    kubectl create -f experiments\exp%%a/job_exp%%a_unet_tci_pretrained.yaml
))

echo "batch complete"
```

```
@ECHO OFF

Rem This batch file executes kubectl commands to delete training jobs

::echo %kubectl%
SET exp_list=1 2 3 4 5 6 7 8

(for %%a in  (%exp_list%) do (
    echo %%a

    kubectl delete -f experiments_2\exp%%a/job_exp%%a_deeplabv3_img.yaml
    kubectl delete -f experiments_2\exp%%a/job_exp%%a_deeplabv3_tci.yaml
    kubectl delete -f experiments_2\exp%%a/job_exp%%a_deeplabv3plus_img.yaml
    kubectl delete -f experiments_2\exp%%a/job_exp%%a_deeplabv3plus_tci.yaml

    kubectl delete -f experiments_2\exp%%a/job_exp%%a_unet_img.yaml
    kubectl delete -f experiments_2\exp%%a/job_exp%%a_unet_tci.yaml
    kubectl delete -f experiments_2\exp%%a/job_exp%%a_unetplus_img.yaml
    kubectl delete -f experiments_2\exp%%a/job_exp%%a_unetplus_tci.yaml
))

echo "batch complete"
```

Automating GPU Jobs on Nautilus

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: $CONTAINAER_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 Nautlius HyperCluster.
  - https://github.com/MU-HPDI/Nautilus-Job-Launcher

- Installation:
  - Use the latest .whl pushed to GitLab's PyPI repository:

```
pip3 install --extra-index-url https://gitlab.nrp-nautilus.io/api/v4/projects/2953/packages/pypi/simple nautiluslauncher
```

  - You can clone this repository and use pip to install it:

```
pip3 install 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

Automating GPU Jobs on Nautilus

Nautilus Job Launcher

- Configuration requires three keys:
- Namesapce (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

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

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

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

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=i, 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()
```

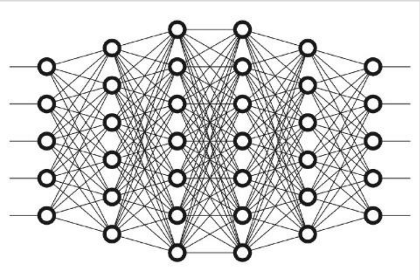
Deep Learning on Nautilus: Semantic segmentation

Nautilus

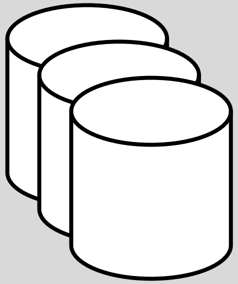
- Iterations of Training Completed: 515,550
- Number of images Processed: 7,070,400
- Trainable Parameters Optimized: 23 millions per model
- The time it took to prepare the experimental set up and to run all the training sessions in parallel is 12 hours
- The actual time it would have take to train is 21 days 12 hours 45 minutes

Nautilus

Deep Learning on Nautilus: Transformer Research

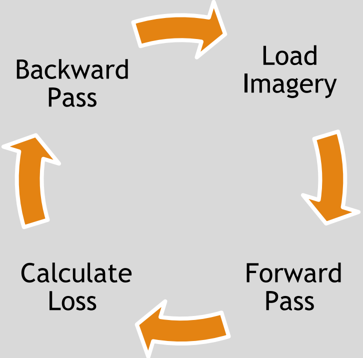


9 Deep Neural Architectures

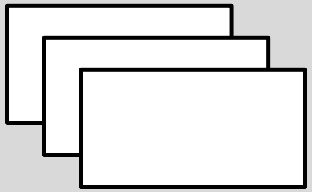


3 Open Source HR-RSI Datasets

27 Trained Models



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



124.74 TB Imagery Processed

Wall Clock Time: 76 days, 10 hours

https://www.freecodecamp.org/news/want-to-know-how-deep-learning-works-heres-a-quick-guide-for-everyone-1aedeca88076/
https://commons.wikimedia.org/wiki/File:PyTorch_logo_icon.svg
https://www.clipartmax.com/download/m2K9A0m2b1m2i8i8_4-o-clock-clip-art-clock-face-clip-art/

MU-HPDI/Nautilus – A resource for exploration

- Sample Dockerfiles
- Sample Kubernetes YAML File
- Wiki with detailed walkthroughs for:
  - Getting Started
  - Creating PVC
  - Creating Pods
  - Creating Jobs

Check back occasionally, we are adding more content and recipes for scaling

Machine Learning and Deep Learning