

---

**cofea-casa**

***Release 2023.11.16+39.g015f705***

**cofea-casa, UNL**

**Dec 02, 2023**



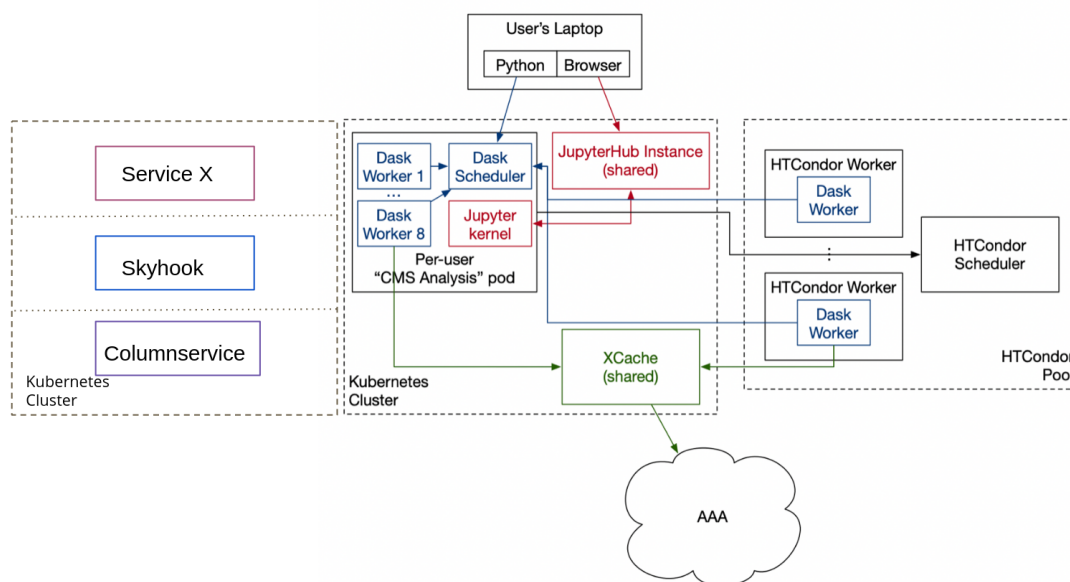
## GETTING STARTED

<b>1</b>	<b>First Steps at Coffea-Casa @ UNL</b>	<b>3</b>
<b>2</b>	<b>Installing Custom Packages on Coffea-Casa</b>	<b>21</b>
<b>3</b>	<b>Performance Metrics on Coffea-Casa</b>	<b>23</b>
<b>4</b>	<b>Interfacing With HTCondor Workers</b>	<b>27</b>
<b>5</b>	<b>Troubleshooting Common Issues</b>	<b>29</b>
<b>6</b>	<b>Coffea-Casa Setup Without Dask Labextension</b>	<b>31</b>
<b>7</b>	<b>How to Configure Dask Labextension Cluster</b>	<b>33</b>
<b>8</b>	<b>Deployment of Coffea-casa Analysis Facility at your Tier 2/Tier 3 grid site or Cluster</b>	<b>35</b>
<b>9</b>	<b>coffea_casa module API</b>	<b>37</b>
<b>10</b>	<b>Community Support and Help</b>	<b>39</b>
	<b>Index</b>	<b>41</b>

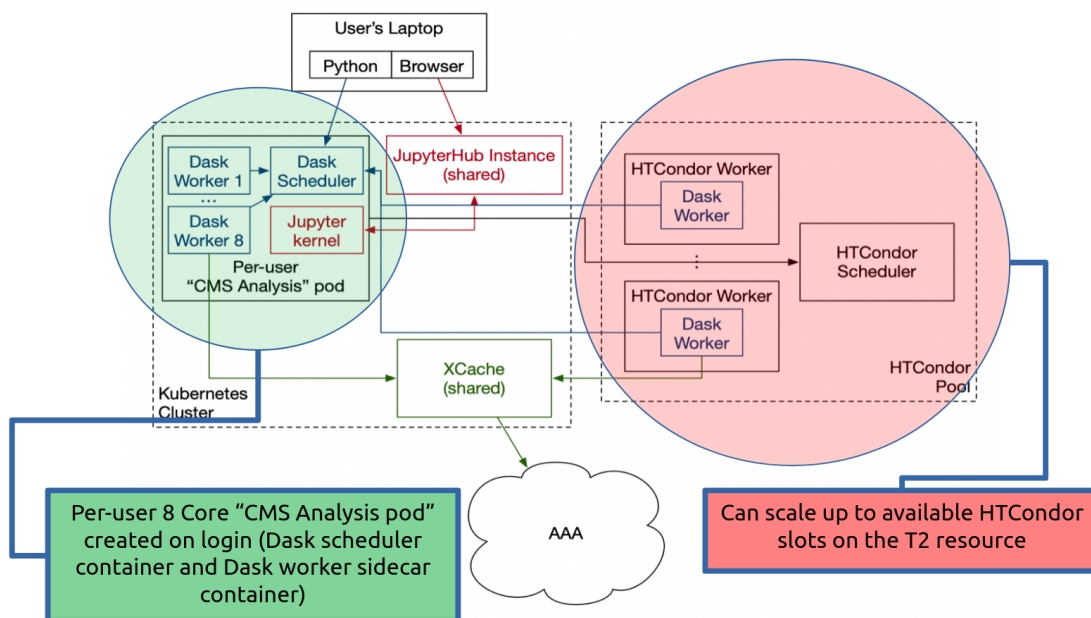


**Coffea-casa** is a prototype of an analysis facility that provides services for “low latency columnar analysis,” enabling rapid processing of data in a column-wise fashion.

Coffea-casa Analysis Facility (AF) services, based on *Dask* and *Jupyter Notebook* technologies, aim to dramatically lower time for analysis and provide an easily-scalable and user-friendly computational environment that will simplify, facilitate, and accelerate the delivery of HEP results.



The facility is built on top of a Kubernetes cluster and integrates with dedicated resources, with resources allocated via fair share through the local HTCondor system and Nebraska Tier-2.



**Note:** **Coffea-casa** is a prototype and is currently in active development: if you had noticed a bug or would like to leave

us feedback, we invite you to open an issue directly on GitHub: [<https://github.com/CoffeaTeam/coffea-casa/issues>](https://github.com/CoffeaTeam/coffea-casa/issues)

---

## FIRST STEPS AT COFFEA-CASA @ UNL

### 1.1 Prerequisites

The primary mode of analysis with coffea-casa is [coffea](#). Coffea provides plenty of examples to users in its [documentation](#). Further resources, meant to run specifically on coffea-casa, can be found in the sidebar under the “Gallery of Coffea-casa Examples” section or the appropriate repository [here](#).

Knowledge of [Python](#) is assumed. The standard environment for coffea analyses is within [Jupyter Notebooks](#), which allow for dynamic, block-by-block execution of code. Coffea-casa employs the [JupyterLab](#) interface. JupyterLab is designed for hosting Jupyter Notebooks on the web and permits the usage of additional features within its environment, including Git access, compatibility with cluster computing tools, and much, much more.

If you aren’t familiar with any of these tools, please click on the links above and get acquainted with how they work before delving into coffea-casa.

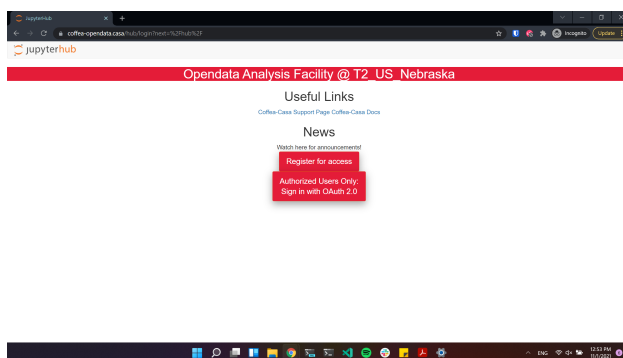
### 1.2 Access

---

**Important:** For CMS or opendata files, please see the relevant sections for coffea-casa at T2 Nebraska. For ATLAS files, see coffea-casa at UChicago.

---

There are two access points to the Coffea-casa AF @ T2 Nebraska. The site at <https://coffea-opendata.casa> is for Opendata and can be accessed through any CILogon identity provider, though it will not be able to process any files that require authentication.



---

**Important:** Remember that to access this instance you need to register: click “Register for access”. (We have limited resources available and can’t provide access to everyone under CILogon).

---

The other at <https://coffea.casa> is for CMS data and can be accessed through the CMS AuthZ instance; this site is capable of handling all CMS files and uses tokens for authentication.



Another coffea-casa instance exists for the AF @ UChicago, which is meant to be used with ATLAS data. You can access it at <https://coffea.af.uchicago.edu>.



See the appropriate section below if you need help going through the registration process for either access point.

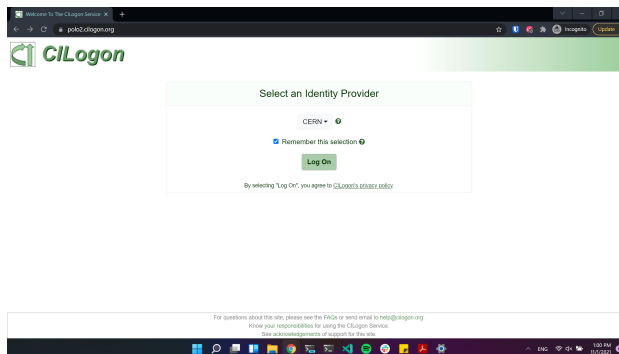
## 1.2.1 Opendata CILogon Authentication Instance

---

**Important:** This section applies only to the Opendata Coffea-Casa instance.

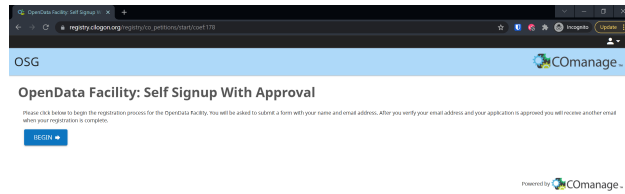
---

Currently Opendata Coffea-Casa supports any CILogon identity provider. Select your identity provider:

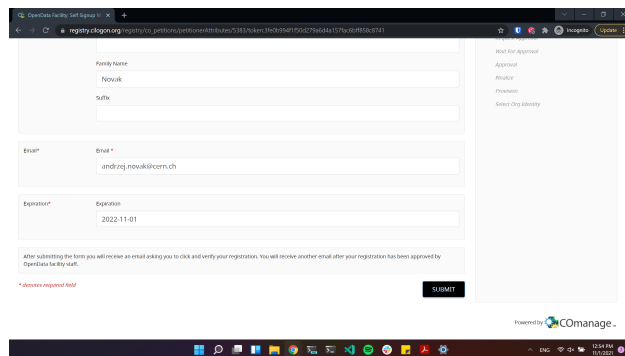


For accessing Opendata Coffea-Casa, we are offering a self-signup registration form with approval.

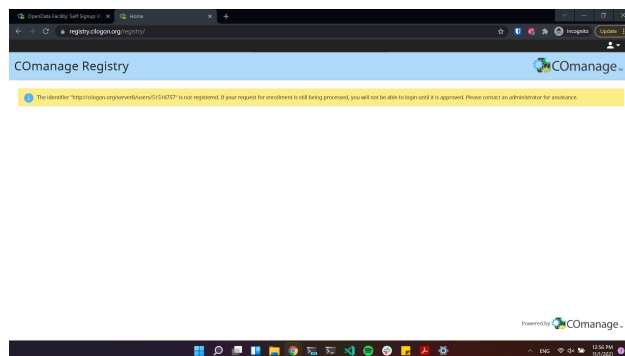




Click to proceed to the next stage:



Click to proceed to the next stage:



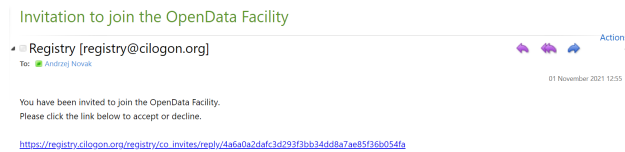
If you see the next window, it means that the registration request was sent successfully!

---

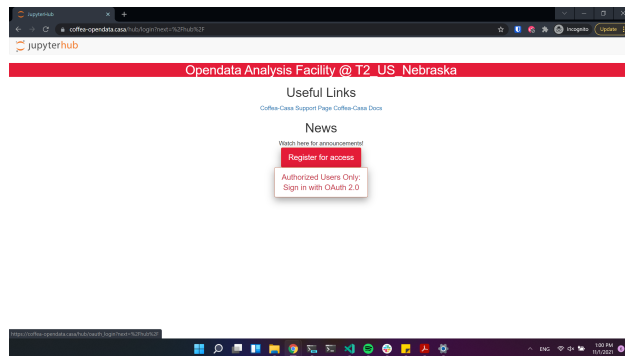
**Important: After this step please wait until you get approved by an administrator!**

---

After your request is approved, you will receive an email, where you will simply need to click a link:



Voila! Now you can login to Opendata Coffea-Casa. Click on “Authorized Users Only: Sign in with OAuth 2.0” to do so:



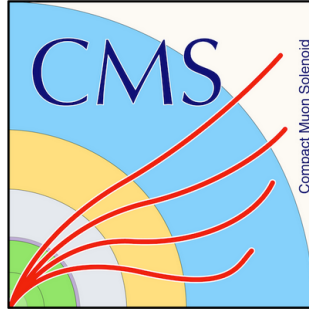
## 1.2.2 CMS AuthZ Authentication Instance

---

**Important:** This section applies only to the CMS Coffea-Casa instance.

---

Currently Coffea-Casa Analysis Facility @ T2 Nebraska supports any member of CMS VO organisation. To access it please sign in or sign up using **Apply for an account**.



Welcome to **cms**

Sign in with

CERN SSO

Not a member?

Apply for an account

You have been successfully authenticated as

**CN=Oksana**

**Shadura,CN=728983,CN=oshadura,OU=Users,OU=Organic**

**Units,DC=cern,DC=ch**

This certificate is not linked to any account in this organization

**Approval Required for *gitops coffea-casa production***

This client was dynamically registered  
[approve.dynamically-registered-containers](#)

more information  
You will be redirected to the following page if you click Approve:  
[https://cmsaf-jh.unl.edu/hub/oauth\\_callback](https://cmsaf-jh.unl.edu/hub/oauth_callback)

Access to:

- ☒ log in using your identity
- ☒ basic profile information
- ☒ email address
- ☒ physical address
- ☒ telephone number
- ☒ Select the WLCG JWT profile
- ☒ Read access to CMS data
- ☒ Write access to CMS data
- ☒ Stage access to CMS data

Remember this decision:

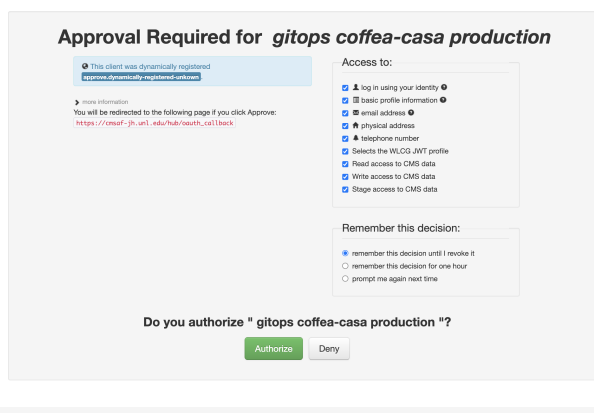
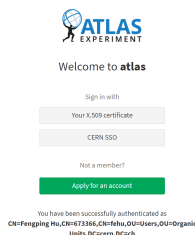
- ☒ remember this decision until I revoke it
- ☐ remember this decision for one hour
- ☐ prompt me again next time

Do you authorize "gitops coffea-casa production"?

### 1.2.3 ATLAS AuthZ Authentication Instance

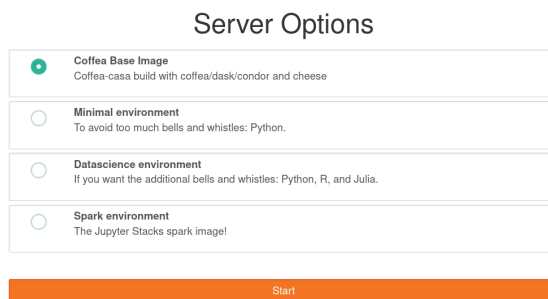
Currently Coffea-Casa Analysis Facility @ UChicago can support any member of ATLAS.

Sign in with your ATLAS CERN credential:

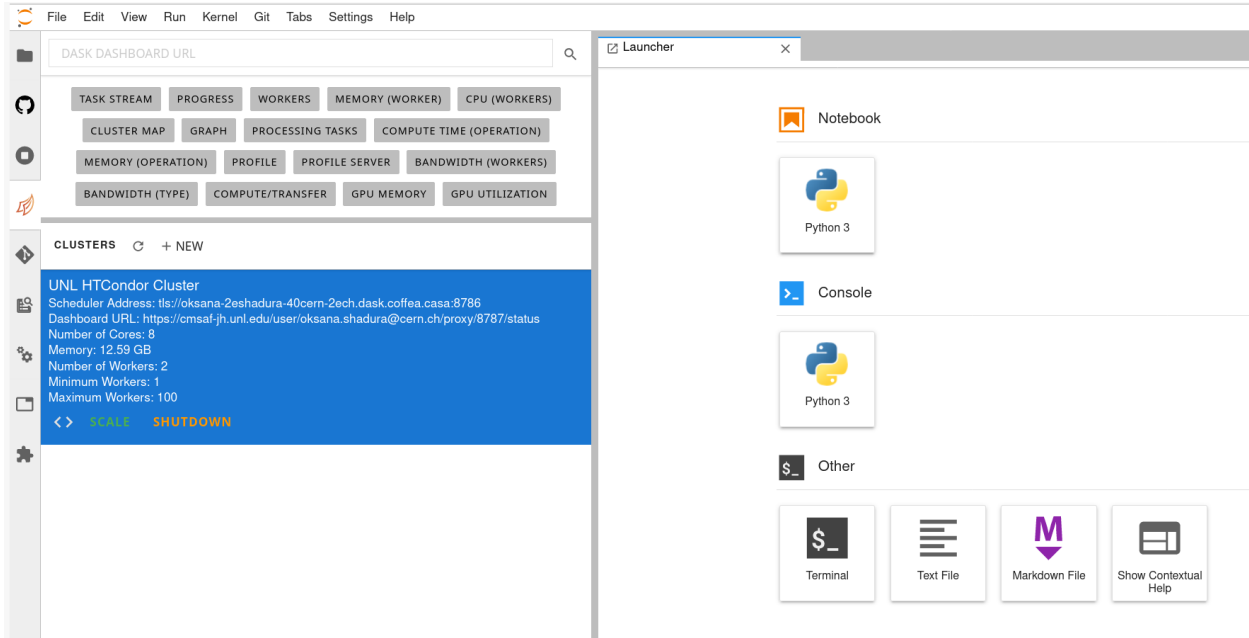


### 1.3 Docker Image Selection

The default image is preloaded with *coffea*, *Dask*, and *HTCondor* and you should select it:



This will forward you to your own personal Jupyterhub instance running at Analysis Facility @ T2 Nebraska:

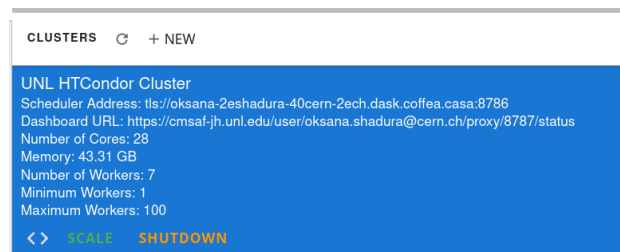


## 1.4 Cluster Resources in Coffea-Casa Analysis Facility @ T2 Nebraska

By default, the Coffea-casa Dask cluster should provide you with a scheduler and workers, which you can see by clicking on the colored Dask icon in the left sidebar.

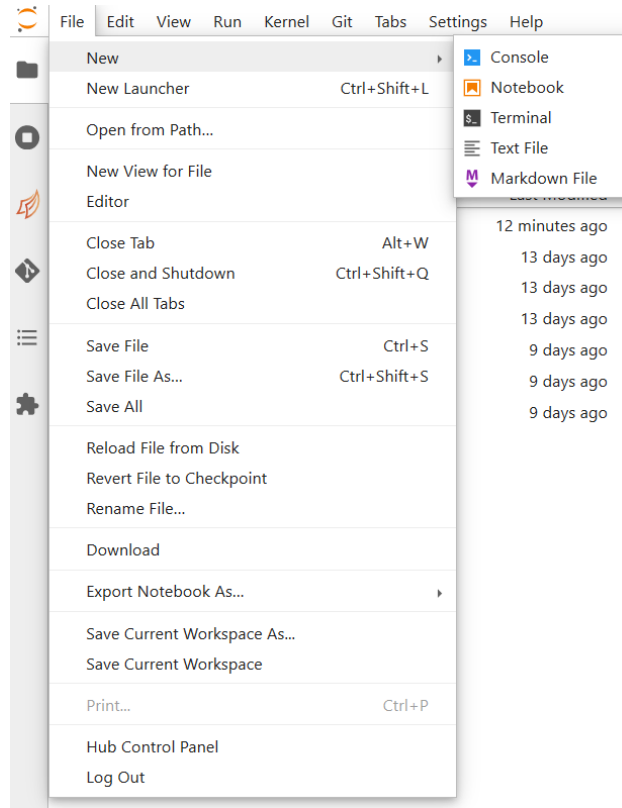


As soon as you start your computations, you will notice that available resources at the Opendata Coffea-Casa Analysis Facility @ T2 Nebraska autoscale depending on the resources available in the HTCondor pool at Nebraska Tier 2.



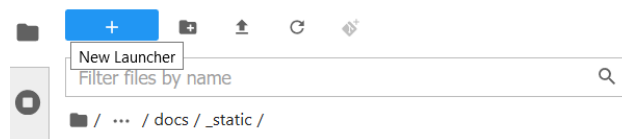
## 1.5 Opening a New Console or File

There are three ways by which you can open a new tab within coffea-casa. Two are located within the **File** menu at the very top of the JupyterLab interface: *New* and *New Launcher*.



The *New* dropdown menu allows you to open the console or a file of a specified format directly. The *New Launcher* option creates a new tab with buttons that permit you to launch a console or a new file, exactly like the interface you are shown when you first open coffea-casa.

The final way is specific to the **File Browser** tab of the sidebar.



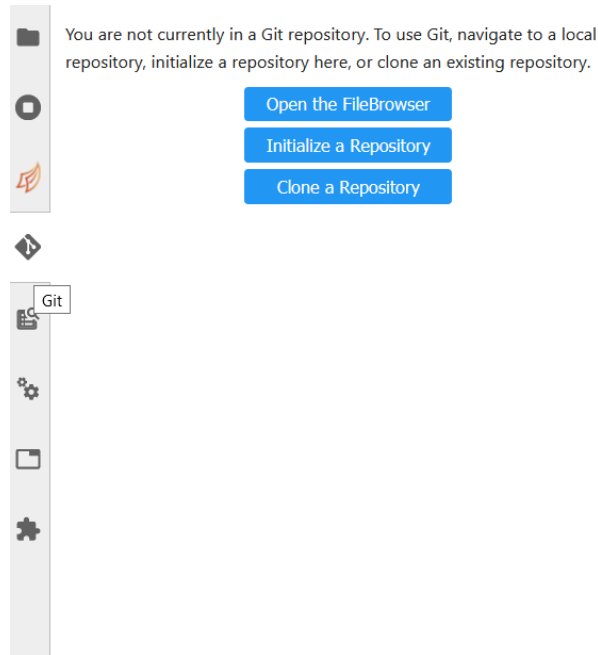
This behaves exactly like the *New Launcher* option above.

**Note:** Regardless of the method you use to open a new file, the file will be saved to the current directory of your **File Browser**.

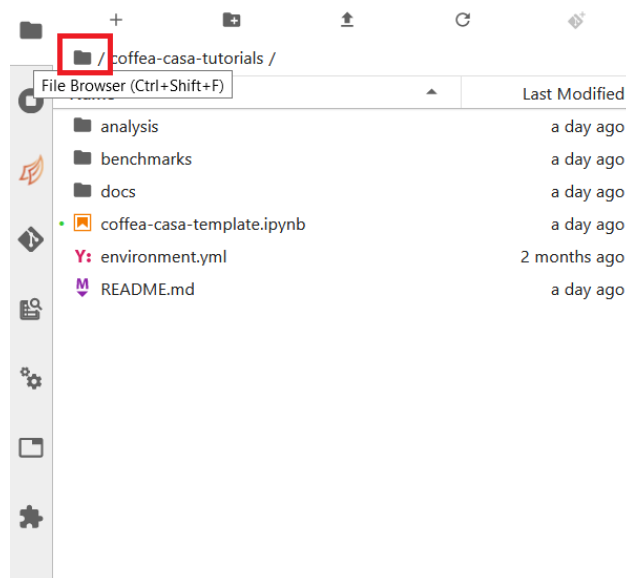
## 1.6 Using Git

Cloning a repository in the Coffea-casa Analysis Facility @ T2 Nebraska is simple, though it can be a little confusing because it is spread across two tabs in the sidebar: the *File Browser* and the *Git* tabs.

In order to clone a repository, first go to the Git tab. It should look like this:



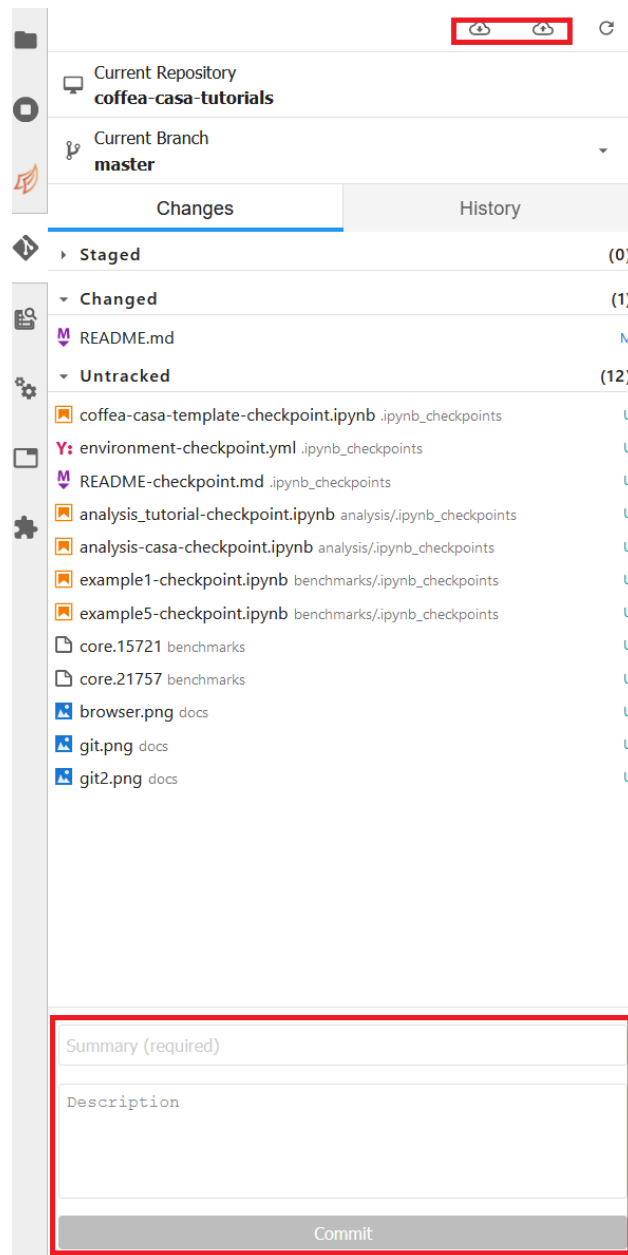
Simply click the appropriate button (initialize a repository, or clone a repository) and you'll be hooked up to GitHub. This should then take you to the *File Browser* tab, which is where you can see all of the repositories you have cloned in your JupyterLab instance. The File Browser should look like this:



If you wish to change repositories, simply click the folder button to enter the root directory. If you are in the root directory, the Git tab will reset and allow you to clone another repository.

If you wish to commit, push, or pull from the repository you currently have active in the File Browser, then you can

return to the Git tab. It should change to look like this, so long as you have a repository open in the File Browser:



The buttons in the top right allow for pulling and pushing respectively. When you have edited files in a directory, they will show up under the *Changed* category, at which point you can hit the + to add them to a commit (at which point they will show up under *Staged*). Filling out the box at the bottom of the sidebar will file your commit, and prepare it for you to push.



## 1.7 Using XCache

**Important:** This section applies only to the CMS Coffea-Casa instance.

When we use CMS data, we generally require certificates or we will be faced with authentication errors. Coffea-casa handles the issue of certificates internally through xcache tokens so that its users do not explicitly have to import their certificates, though this dynamic requires adjustment of the redirector portion of the path to the root file requested.

Let's say we wish to request the file:

```
root://cmsxrootd.fnal.gov//store/data/Run2018A/DoubleMuon/NANOAOB/02Apr2020-v1/30000/
↳0555868D-6B32-D249-9ED1-6B9A6AABDAF7.root
```

Then we would replace the `cmsxrootd.fnal.gov` redirector with the `xcache` redirector:

```
root://xcache//store/data/Run2018A/DoubleMuon/NANOAOB/02Apr2020-v1/30000/0555868D-6B32-
↳D249-9ED1-6B9A6AABDAF7.root
```

Now, we will be able to access our data.

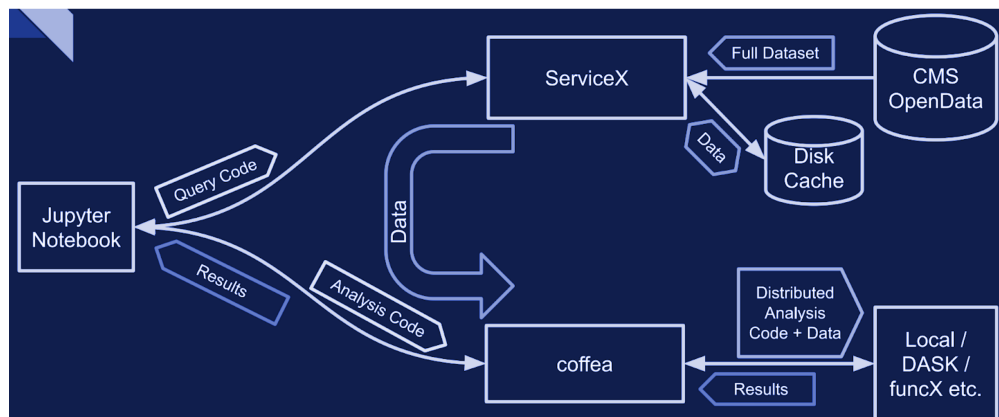
In addition to handling authentication, XCache will cache files so that they are able to be pulled more quickly in subsequent runs of the analysis. It should be expected, then, that the first analysis run with a new coffea-casa file will run slower than ones which follow afterwards.

## 1.8 ServiceX

**Important:** This section applies only to the ATLAS Coffea-Casa instance.

**Important:** This section applies only to the ATLAS Coffea-Casa instance at UChicago. The instances at T2 Nebraska are capable of handling ServiceX requests through uproot, but the feature is still at an experimental stage. Ask an administrator for more information on accessing ServiceX on the T2 Nebraska instances.

When dealing with very large datasets it is often better to do initial data filtering and augmentation using [ServiceX](#). ServiceX transformations produce their output as an Awkward Array. The array can then be used in a regular Coffea processor. Here is a schema explaining the workflow:



There are two different UC AF-deployed ServiceX instances. The only difference between them is the type of input data they are capable of processing. **Uproot** processes any kind of “flat” ROOT files, while **xAOD** processes only Rucio registered xAOD files.

To use them one has to register and get approved. Sign in will lead you to a Globus registration page where you may choose to use an account connected to your institution:

Once approved, you will be able to see the status of your requests in the dashboard:

Title	Submitted by	Start time	Finish time	Status	Files completed	Workers	Actions
Untitled	Ilija Vukotic	2021-11-01 22:18:09	2021-11-01 22:19:13	Complete	1 of 1	-	
Untitled	Ilija Vukotic	2021-11-01 22:18:09	2021-11-01 22:19:05	Complete	3 of 3	-	
Untitled	Ilija Vukotic	2021-11-01 22:18:09	2021-11-01 22:18:58	Complete	4 of 4	-	
Untitled	Ilija Vukotic	2021-11-01 22:18:09	2021-11-01 22:18:55	Complete	4 of 4	-	
Untitled	Ilija Vukotic	2021-10-25 15:13:39	2021-10-25 15:13:50	Complete	1 of 1	-	
Untitled	Ilija Vukotic	2021-10-25 15:12:56	2021-10-25 15:13:07	Complete	1 of 1	-	

For your code to be able to authenticate your requests you need to download a servicex.yaml file, which should be placed in your working directory. The file is downloaded from your profile page:

For an example analysis using ServiceX and Coffea look [here](#).

## 1.9 Opendata Example

In this example (which corresponds to [ADL Benchmark 1](#)), we'll try to run a simple analysis example on the Coffea-Casa Analysis Facility. We will use the `coffea_casa` wrapper library, which allows use of pre-configured settings for HTCondor configuration and Dask scheduler/worker images.

Our goal in this *toy* analysis is to plot the missing transverse energy (*MET*) of all events from a sample dataset; this data was converted from 2012 CMS Open Data (17 GB, 54 million events), and is available in public EOS (`root://eospublic.cern.ch//eos/root-eos/benchmark/Run2012B_SingleMu.root`).

First, we need to import the coffea libraries used in this example:

```
import numpy as np
%matplotlib inline
from coffea import hist
import coffea.processor as processor
import awkward as ak
from coffea.nanoevents import schemas
```

To select the aforementioned data in a coffea-friendly syntax, we employ a dictionary of datasets, where each dataset (key) corresponds to a list of files (values):

```
fileset = {'SingleMu' : ["root://eospublic.cern.ch//eos/root-eos/benchmark/Run2012B_
↳ SingleMu.root"]}]
```

Coffea provides the `coffea.processor` module, where users may write their analysis code without worrying about the details of efficient parallelization, assuming that the parallelization is a trivial map-reduce operation (e.g., filling histograms and adding them together).

```
# This program plots an event-level variable (in this case, MET, but switching it is as
↳ easy as a dict-key change). It also demonstrates an easy use of the book-keeping
↳ cutflow tool, to keep track of the number of events processed.

# The processor class bundles our data analysis together while giving us some helpful
↳ tools. It also leaves looping and chunks to the framework instead of us.
class Processor(processor.ProcessorABC):
    def __init__(self):
        # Bins and categories for the histogram are defined here. For format, see https://
↳ /coffeateam.github.io/coffea/stubs/coffea.hist.hist_tools.Hist.html && https://
↳ coffeateam.github.io/coffea/stubs/coffea.hist.hist_tools.Bin.html
        dataset_axis = hist.Cat("dataset", "")
        MET_axis = hist.Bin("MET", "MET [GeV]", 50, 0, 100)

        # The accumulator keeps our data chunks together for histogramming. It also
↳ gives us cutflow, which can be used to keep track of data.
        self._accumulator = processor.dict_accumulator({
            'MET': hist.Hist("Counts", dataset_axis, MET_axis),
            'cutflow': processor.defaultdict_accumulator(int)
        })

    @property
    def accumulator(self):
        return self._accumulator
```

(continues on next page)

(continued from previous page)

```

def process(self, events):
    output = self.accumulator.identity()

    # This is where we do our actual analysis. The dataset has columns similar to
    ↪ the TTree's; events.columns can tell you them, or events.[object].columns for deeper
    ↪ depth.
    dataset = events.metadata["dataset"]
    MET = events.MET.pt

    # We can define a new key for cutflow (in this case 'all events'). Then we can
    ↪ put values into it. We need += because it's per-chunk (demonstrated below)
    output['cutflow']['all events'] += ak.size(MET)
    output['cutflow']['number of chunks'] += 1

    # This fills our histogram once our data is collected. The hist key ('MET=') will
    ↪ be defined in the bin in __init__.
    output['MET'].fill(dataset=dataset, MET=MET)
    return output

def postprocess(self, accumulator):
    return accumulator

```

With our data in our fileset variable and our processor ready to go, we simply need to connect to the Dask Labextension-powered cluster available within the Coffea-Casa Analysis Facility @ T2 Nebraska. This can be done by dragging the scheduler into the notebook, or by manually typing the following:

```

from dask.distributed import Client
client = Client("tls://localhost:8786")

```

Then we bundle everything up to run our job, making use of the Dask executor. We must point it to our client as defined above. In the Runner, we specify that we want to make use of the NanoAODSchema (as our input file is a NanoAOD).

```

executor = processor.DaskExecutor(client=client)
run = processor.Runner(executor=executor,
                      schema=schemas.NanoAODSchema,
                      savemetrics=True
                      )

output, metrics = run(fileset, "Events", processor_instance=Processor())

```

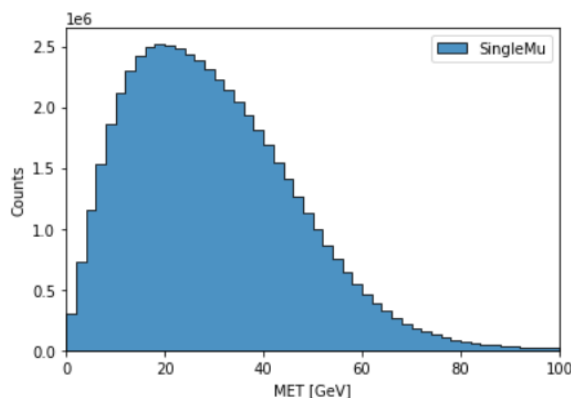
The final step is to generate a 1D histogram from the data output to the 'MET' key. fill\_opts are optional arguments to fill the graph (default is a line).

```

hist.plot1d(output['MET'], overlay='dataset', fill_opts={'edgecolor': (0,0,0,0.3), 'alpha'
↪ ': 0.8})

```

As a result you should see the following plot:



## 1.10 CMS Example

**Important:** This section applies only to the CMS Coffea-Casa instance.

Now we will try to run a short example, using CMS data, which corresponds to plotting the [dimuon Z-peak](#). We use dimuon data which consists of ~3 million events at ~2.7 GB which belongs to the /DoubleMuon/Run2018A-02Apr2020-v1/NANOAOD dataset.

We import some common coffea libraries used in this example:

```
import numpy as np
from coffea import hist
from coffea.analysis_objects import JaggedCandidateArray
import coffea.processor as processor
%matplotlib inline
```

To select the aforementioned data in a coffea-friendly syntax, we employ a dictionary of datasets, where each dataset (key) corresponds to a list of files (values):

```
fileset = {'DoubleMu' : ['root://xcache//store/data/Run2018A/DoubleMuon/NANOAOD/
→ 02Apr2020-v1/30000/0555868D-6B32-D249-9ED1-6B9A6AABDAF7.root',
                        'root://xcache//store/data/Run2018A/DoubleMuon/NANOAOD/02Apr2020-
→ v1/30000/07796DC0-9F65-F940-AAD1-FE82262B4B03.root',
                        'root://xcache//store/data/Run2018A/DoubleMuon/NANOAOD/02Apr2020-
→ v1/30000/09BED5A5-E6CC-AC4E-9344-B60B3A186CFA.root']}]
```

Coffea provides the `coffea.processor` module, where users may write their analysis code without worrying about the details of efficient parallelization, assuming that the parallelization is a trivial map-reduce operation (e.g., filling histograms and adding them together).

```
class Processor(processor.ProcessorABC):
    def __init__(self):
        dataset_axis = hist.Cat("dataset", "Dataset")
        dimu_mass_axis = hist.Bin("dimu_mass", "$\mu\mu$ Mass [GeV]", 50, 20, 120)

        self._accumulator = processor.dict_accumulator({
            'dimu_mass': hist.Hist("Counts", dataset_axis, dimu_mass_axis),
        })
```

(continues on next page)

(continued from previous page)

```

@property
def accumulator(self):
    return self._accumulator

def process(self, events):
    output = self.accumulator.identity()

    dataset = events.metadata["dataset"]

    mu = events.Muon
    # Select events with 2 muons whose charges cancel out (Zs are charge-neutral).
    dimu_neutral = mu[(ak.num(mu) == 2) & (ak.sum(mu.charge, axis=1) == 0)]
    # Add together muon pair p4's, find dimuon mass.
    dimu_mass = (dimu_neutral[:, 0] + dimu_neutral[:, 1]).mass
    # Plot dimuon mass.
    output['dimu_mass'].fill(dataset=dataset, dimu_mass=dimu_mass)
    return output

def postprocess(self, accumulator):
    return accumulator

```

With our data in our fileset variable and our processor ready to go, we simply need to connect to the Dask Labextension-powered cluster available within the Coffea-Casa Analysis Facility @ T2 Nebraska. This can be done by dragging the scheduler into the notebook, or by manually typing the following:

```

from dask.distributed import Client
client = Client("tls://localhost:8786")

```

Then we bundle everything up to run our job, making use of the Dask executor. To do this, we must point to a client within `executor_args`.

```

executor = processor.DaskExecutor(client=client)
run = processor.Runner(executor=executor,
                      schema=schemas.NanoAODSchema,
                      )

output = run(fileset, "Events", processor_instance=Processor())

```

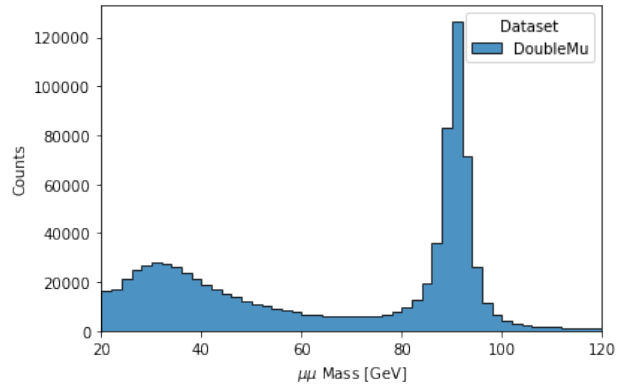
The final step is to generate a 1D histogram from the data output to the 'MET' key. `fill_opts` are optional arguments to fill the graph (default is a line).

```

hist.plot1d(output['dimu_mass'], overlay='dataset', fill_opts={'edgecolor': (0,0,0,0.3),
↪ 'alpha': 0.8})

```

As a result you should see the following plot:



## 1.11 ATLAS Examples

---

**Important:** This section applies only to the ATLAS Coffea-Casa instance.

---

The notebooks about columnar data analysis with DAOD\_PHYSLITE [here](https://github.com/nikoladze/agc-tools-workshop-2021-physlite) `<https://github.com/nikoladze/agc-tools-workshop-2021-physlite>` may be useful as a reference.





## INSTALLING CUSTOM PACKAGES ON COFFEA-CASA

Coffea-casa provides support for users to install their own packages through pip. In cases where your notebook does not send jobs to workers, you can simply install your packages to the scheduler through the terminal. If you wish for your workers to make use of custom packages, however, you'll need to go through some extra steps to ensure your packages are installed on all workers. Installation to the scheduler is recommended in either case.

### 2.1 Installations on the Scheduler

Let's run through an example: we want to install `pytest` to our scheduler, and we don't need it to be distributed across our workers. In that case, we can open up a new terminal in coffea-casa by going to the **File** tab and starting a new launcher. A cursory `pip freeze` demonstrates that `pytest` is *not* installed by default, and so we have to add it ourselves. We can do this through `pip install pytest`, which will tell us that the package is being installed into `/opt/conda/lib/python3.8/site-packages`. A follow-up `pip freeze` indicates that `pytest` is now in our list of packages:

```
pyrsistent @ file:///home/conda/feedstock_root/build_artifacts/pyrsistent_1610146798212/work
PySocks @ file:///tmp/build/80754af9/pysocks_1605305779399/work
pytest==6.2.3
python-dateutil==2.8.1
python-editor==1.0.4
python-json-logger @ file:///home/conda/feedstock_root/build_artifacts/python-json-logger_1602545356084/work
```

More complicated pip installations are also available. Refer to the [pip install docs](#) for more information.

### 2.2 Installations on the Workers

If you install a package to your scheduler but attempt to run it on your workers, you will quickly run into `ModuleNotFound` errors. This is because your workers don't have the package! Luckily, a remedy to this problem is fairly simple. Again, let's assume that we wish to install `pytest` and that this time we wish to use it during processing on our workers. We first get it onto our scheduler (as above), and then we tell our workers to install it:

```
from dask.distributed import Client, PipInstall

dependencies = [
    "pytest",
]
client = Client("tls://localhost:8786")
client.register_worker_plugin(PipInstall(packages=dependencies))
```

To use this code for your own purposes, you merely have to put in your dependencies in the `dependencies` list. This should support all installation formats that pip does, and you can add multiple packages by expanding the list. For example, to install a second package from a GitHub repository, you could specify:

```
dependencies = [  
    "pytest",  
    "topcoffea@git+https://github.com/TopEFT/topcoffea.git",  
]
```

and your workers should have both `pytest` and `topcoffea` installed onto them.

## 2.3 Sending Files to Workers (Without Pip)

If you have a file that you wish to import without pip, and you need to send it to your workers, then you can do so by initializing your client and providing it the following line:

```
client.upload_file('foo.py')
```

Where `foo.py` is replaced by the file you are attempting to import.

## PERFORMANCE METRICS ON COFFEA-CASA

### 3.1 Available Features

At some point you will probably want to measure the performance of your analysis on coffea-casa. Luckily, both Dask and coffea come equipped with the capabilities to give you more information about different aspects of your analysis run. These include, broadly:

- `run_uproot_job` has a `savemetrics` arg which can provide basic info such as the number of entries and the process time
- the Dask dashboard includes a variety of info, and can be used interactively while your analysis is running
- the Dask performance report is a snapshot of the Dask dashboard that can be saved for later review

### 3.2 Coffea Metrics

The simplest metrics that can be obtained stem from coffea. They include the number of bytes read, the names of all columns, the number of entries, the processing time (summed across all cores), and the number of chunks. These can be accessed by adding the `savemetrics: True` argument to your `Runner` function. For example:

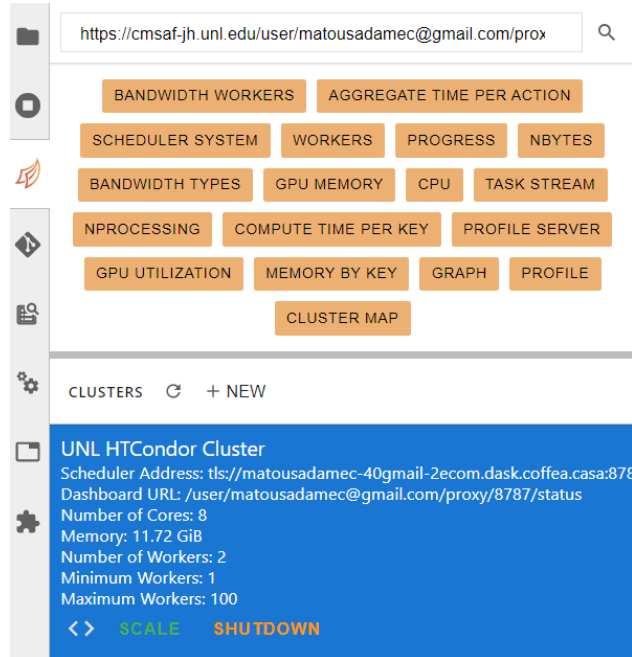
```
run = processor.Runner(executor=executor,
                      schema=schemas.NanoAODSchema,
                      savemetrics=True)

output, metrics = run(fileset, "Events", processor_instance=Processor())
```

It should be noted that the introduction of this argument changes the format of your output by converting it into a tuple. Within this tuple, `output[0]` will contain everything that `output` did without `savemetrics` on, while `output[1]` will contain the metrics. You can retrieve the “standard” behavior by taking the output of `run()` as two variables, as we did above, because Python is capable of parsing tuple outputs into multiple variables.

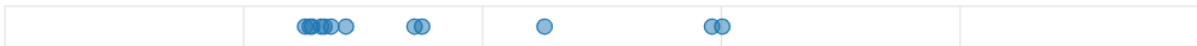
### 3.3 Dask Interactive Dashboard

The Dask interactive dashboard can be accessed in the same side-pane where your cluster information is displayed. The buttons at the top should be orange when your analysis is running, though they may be grey if the scheduler is inactive or has been shutdown.

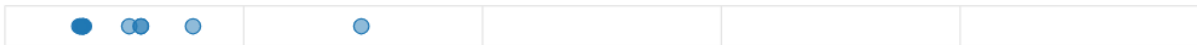


Among the tabs, you're most likely to find useful information under *workers* and *task stream*. The workers tab gives you information about which workers are running and how many resources each is using:

#### CPU Use (%)



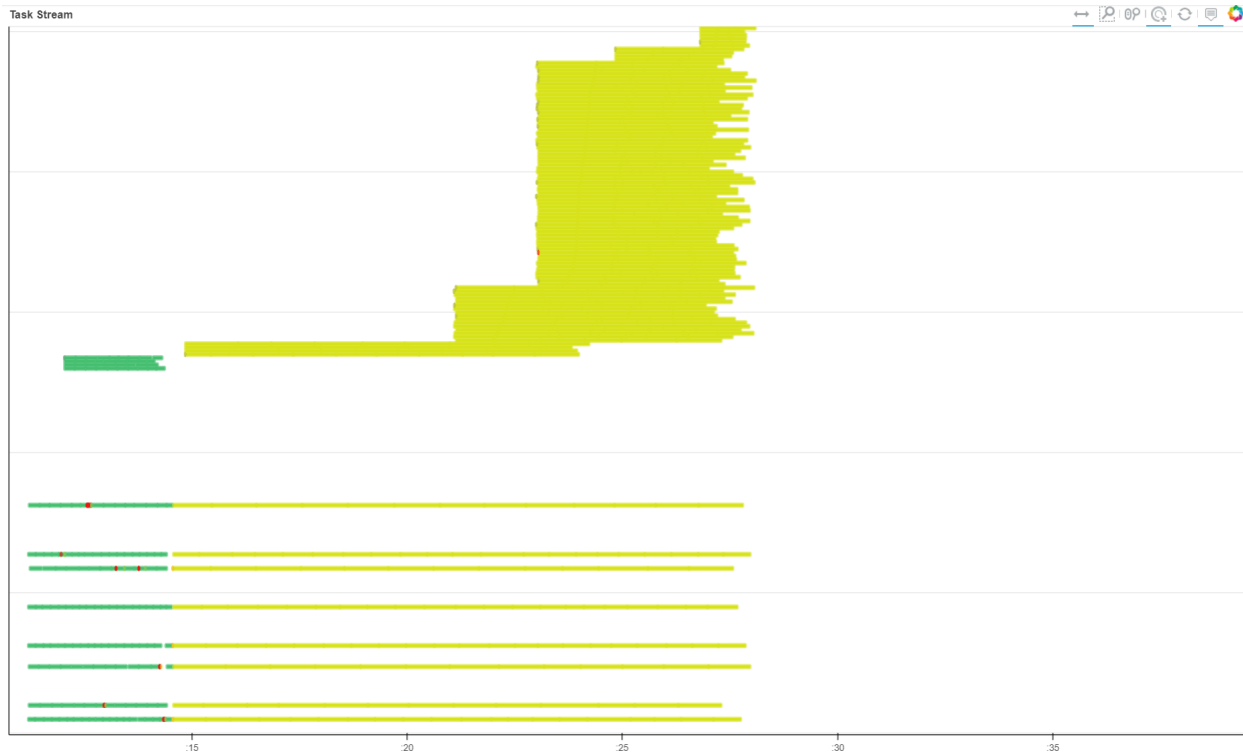
#### Memory Use (%)



name	address	nthreads	cpu	memory	memory_limit	memory %	num_fds	read_bytes	write_bytes
Total (11)		44	139.7 %	6.58 GiB	63.22 GiB	10.4 %	401	12 MiB	201 KiB
htcondor--1648i	tls://red-c6005.i	4	180.8 %	922.79 MiB	5.72 GiB	15.7 %	44	1 MiB	53 KiB
htcondor--1648i	tls://red-c0805.i	4	240.3 %	666.52 MiB	5.72 GiB	11.4 %	35	6 MiB	36 KiB
htcondor--1648i	tls://red-c6005.i	4	109.3 %	386.47 MiB	5.72 GiB	6.6 %	35	2 KiB	1 KiB
htcondor--1648i	tls://red-c0805.i	4	236.9 %	668.19 MiB	5.72 GiB	11.4 %	35	3 MiB	20 KiB
htcondor--1648i	tls://red-c6005.i	4	114.2 %	381.33 MiB	5.72 GiB	6.5 %	35	80 B	80 B
htcondor--1648i	tls://red-c6005.i	4	102.1 %	383.35 MiB	5.72 GiB	6.5 %	35	336 B	1 KiB
htcondor--1648i	tls://red-c6005.i	4	102.9 %	370.58 MiB	5.72 GiB	6.3 %	35	172 B	744 B
htcondor--1648i	tls://red-c6005.i	4	107.0 %	365.26 MiB	5.72 GiB	6.2 %	35	176 B	763 B
htcondor--1648i	tls://red-c6005.i	4	100.6 %	375.85 MiB	5.72 GiB	6.4 %	35	144 B	625 B
htcondor--1648i	tls://red-c6005.i	4	105.9 %	384.48 MiB	5.72 GiB	6.6 %	35	175 B	757 B
kubernetes-wor	tls://matousada	4	137.2 %	1.79 GiB	6.00 GiB	29.8 %	42	1 MiB	86 KiB

The task stream tab gives you information about what each thread is currently working on. This can give insight into,


for example, preprocessing versus processing times, the length of an average processing thread, and a visual depiction of anomalies in thread run-time. The graph can look a little intimidating because we typically have a lot of threads:



Put simply, however, the green parts of the task stream represent the preprocessing step and the yellow parts represent processing. The “verticality” represents the number of threads currently running. Because autoscaling is on, this number varies, with some shutting down (the yellow lines that end short) and some starting up (the new lines stacked at the top).

### 3.4 Dask Performance Report

The Dask performance report is much like the dashboard, though it provides a static snapshot which is generated after a run completes. This snapshot is in the form of an HTML file, which looks something like this when rendered:

 Distrust HTML

---

Summary Task Stream Worker Profile (compute) Worker Profile (administrative) Scheduler Profile (administrative) Bandwidth (Workers) Bandwidth (Types)

---

## Dask Performance Report

Select different tabs on the top for additional information

**Duration: 450.12 s**

### Tasks Information

- number of tasks: 113
- compute time: 14757.91 s
- deserialize time: 795.26 ms
- transfer time: 2.71 s

### Scheduler Information

- Address: `tls://192.168.134.117:8786`
- Workers: 14
- Threads: 56
- Memory: 80.39 GiB
- Dask Version: 2021.04.0
- Dask Distributed Version: 2.16.0+395.g61e46849

It should be noted that the information presented here is somewhat more limited. We don't have access to the nice worker tab, but we do still have access to the task stream. In order to generate a performance report for your analysis, you have to wrap your `run_uproot_job` in such a way:

```
from dask.distributed import performance_report
with performance_report(filename="dask-report.html"):
    output = run(fileset, "Events", processor_instance=Processor())
```

The file will be saved in the working directory, unless you specify a direct path along with the file name.

## 3.5 Suggestions for Improving Performance

With a better understanding of our performance, it's natural to wonder how it could be improved. While autoscaling should pinpoint the ideal number of workers for an analysis run, it could be imperfect, and testing scaling with a manually set amount of workers could be nice check. If disparities between the autoscaling amount and the true ideal amount exist, they should be reported as an issue so that the coffea-casa team try to better optimize the system.

Another factor which can dramatically impact performance is the `chunksize` in `run_uproot_job`. In general, it appears that a lower `chunksize` results in quicker runtimes, but there is a lower bound beyond which performance begins to drop. Optimizing `chunksize` should be a first stop for addressing performance issues if autoscaling is satisfactory.

Lastly, if you are appealing to regular Python operations within your processor (i.e., not Awkward or NumPy), try to wrap them with [Numba](#).

## INTERFACING WITH HTCONDOR WORKERS

### 4.1 Locating Workers

Dask deploys workers through HTCondor. Information about these workers can be located through the terminal using the `condor_q` command. This will display information about all workers which have run on coffea-casa, but you can typically find your workers by approximating their starting time and date. For a more reliable method, you can run the command `condor_q -af:h Owner JobStartDate JobId DaskSchedulerAddress | grep -E "(username|Owner)"` where `username` is replaced by your coffea-casa username. This username is identical to the name in your terminal command line which follows `cms-jovyan@jupyter` (e.g. `cms-jovyan@jupyter-matousadamec-40gmail-2ecom:~$` has username `matousadamec-40gmail-2ecom`).

### 4.2 Accessing Workers

Each of the workers listed after `condor_q` is executed has an ID associated with it. This can be found either under the `BATCH_NAME` or `JOBS_ID` columns, both of which should be identical (up to the decimal). To connect to a specific worker, you can use the `condor_ssh_to_job ID` command, replacing `ID` with your worker's ID. Upon a successful connection, your terminal should indicate the worker you have connected to. An example of what this looks like is provided below:

```
cms-jovyan@jupyter-matousadamec-40gmail-2ecom:~$ condor_ssh_to_job 16767555
Welcome to slot1_6@red-c6113.unl.edu!
Your condor job is running with pid(s) 5863.
/bin/bash: /opt/conda/lib/libtinfo.so.6: no version information available (required by /bin/bash)
(base) cms-jovyan@cms-jovyan-16767555:/var/lib/condor/execute/dir_5542$
```

From here, you can execute terminal commands as usual, but you are now “within” the worker. Of particular interest here will be the log files `_condor_stderr` and `_condor_stdout`, which will tell you any errors or print statements executed during the worker's runtime. You should also be able to see an `xcache_token` if you are wanting to use CMS data (as you should be running on an instance where `xcache` is enabled).

## 4.3 Killing Workers

To kill a job, go to the scheduler terminal. Find the ID of the job you are seeking to kill through `condor_q`, and then use `condom_rm ID` (replacing ID with your job's ID). This may be useful if your job becomes stuck during processing.



## TROUBLESHOOTING COMMON ISSUES

In general, it is advised that you restart your coffea-casa server before doing further troubleshooting, so that you can ensure your instance is up-to-date. You can do this by going into the File menu, accessing the Hub Control Panel, and pressing the big red “Stop My Server” button.



After the server is shut down, you will get a series of linear prompts to start it back up again. If the problem persists, then it's time for a deeper investigation!

### 5.1 Accessibility Issues

**The coffea-casa server won't load, I get an error when trying to access the page, or I'm told there are certificate issues.**

There's a plethora of issues which seem specific to certain web browsers. If you run into any of these, please attempt to open coffea-casa in a different browser. Should this still fail, open a new issue.

If opening coffea-casa in a new browser solves the issue, you are still encouraged to provide information within [this issue](#) to help us gather data.

**Running a manually-configured Dask cluster gives me a dashboard link, but the dashboard link does not work.**

This is expected behavior. If you go into the Dask sidebar of JupyterLab, however, the orange keys should still work and give you access to the information you'd find within the dashboard. If the keys are grey or any other problems arise, please submit an issue.

### 5.2 Runtime Issues

**The terminal appears to terminate without an error, or I have noticed strange “core.####” files within my file browser.**

If your terminal is terminating without errors, please check for the aforementioned core files within your file browser. If they are present, then you are generating core dumps. In either case, report an issue on GitHub specifying what you are trying to do, which step is going wrong, and whether you are getting core dumps. This will help us pinpoint what's going wrong.

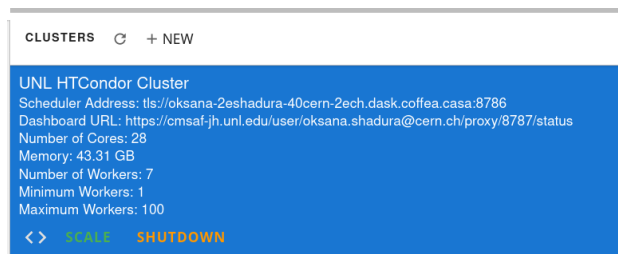
**I have installed a package through the terminal, but I still get ModuleNotFound errors when attempting to run my processor.**

Ensure that you have installed your package onto the workers as well. A guide for this can be found [here in our documentation](#).

## COFFEA-CASA SETUP WITHOUT DASK LABEXTENTION

### 6.1 Preparations

Please shut down UNL HTCondor Cluster (powered by Dask Labextension and available by default), by pushing the button labeled Shutdown:



### 6.2 Instantiating Your Own CoffeaCasaCluster

The next snippet will set up a cluster by instantiating a Dask Client with *CoffeaCasaCluster*, scaled to use 10 jobs:

```
from distributed import Client
from coffea_casa import CoffeaCasaCluster

cluster = CoffeaCasaCluster()
cluster.scale(10)
client = Client(cluster)
```

You can use an adaptive mechanism for Dask job autoscaling. This will scale Dask workers automatically based on scheduler activity:

```
from distributed import Client
from coffea_casa import CoffeaCasaCluster

cluster = CoffeaCasaCluster()
cluster.adapt(minimum=4, maximum=10)
client = Client(cluster)
```

**Note:** Don't forget to shutdown your Coffea-casa cluster before starting a new one:

```
cluster.close()
```

## 6.3 CoffeaCasaCluster

The default *CoffeaCasaCluster* constructor settings:

```
{
'protocol': 'tls://',
'security': Security(require_encryption=True,
                      tls_ca_file='/etc/cmsaf-secrets/ca.pem',
                      tls_client_cert='/etc/cmsaf-secrets/hostcert.pem',
                      tls_client_key='/etc/cmsaf-secrets/hostcert.pem',
                      tls_scheduler_cert='/etc/cmsaf-secrets/hostcert.pem',
                      tls_scheduler_key='/etc/cmsaf-secrets/hostcert.pem',
                      tls_worker_cert='/etc/cmsaf-secrets/hostcert.pem',
                      tls_worker_key='/etc/cmsaf-secrets/hostcert.pem'),
'log_directory': 'logs',
'silence_logs': 'DEBUG',
'scheduler_options': {'port': 8786,
'dashboard_address': '8787',
'protocol': 'tls',
'external_address': 'tls://oksana-2eshadura-40cern-2ech.dask.coffea.casa:8786'},
'job_extra': {'universe': 'docker',
'docker_image': 'coffeateam/coffea-casa-analysis:0.2.23',
'container_service_names': 'dask',
'dask_container_port': 8786,
'transfer_input_files': '/etc/cmsaf-secrets/ca.pem, /etc/cmsaf-secrets/
↪hostcert.pem, /etc/cmsaf-secrets/xcache_token',
'encrypt_input_files': '/etc/cmsaf-secrets/ca.pem, /etc/cmsaf-secrets/
↪hostcert.pem, /etc/cmsaf-secrets/xcache_token',
'transfer_output_files': '',
'when_to_transfer_output': 'ON_EXIT',
'should_transfer_files': 'YES',
'Stream_Output': 'False',
'Stream_Error': 'False',
'+DaskSchedulerAddress': '"tls://oksana-2eshadura-40cern-2ech.dask.coffea.
↪casa:8786"'}}
```

which you can easily adjust just passing appropriate arguments to *CoffeaCasaCluster* constructor:

```
cluster = CoffeaCasaCluster(cores=1, memory="10 GiB")
```

or

```
cluster = CoffeaCasaCluster(job_extra = {'docker_image': 'coffeateam/coffea-casa-
↪analysis:latest'})
```

---

**Note:** Coffea-casa is using communication through the TLS protocol. You will not be able to disable TLS!

---

To learn how to use Dask Labextension, please check *How to Configure Dask Labextension Cluster*.

## HOW TO CONFIGURE DASK LABEXTENSION CLUSTER

The Dask JupyterLab extension package provides a JupyterLab extension to manage Dask clusters, as well as to embed Dask's dashboard plots directly into JupyterLab panes.

The `~/.config/dask/jobqueue-coffea-casa.yaml` or `/etc/dask/jobqueue-coffea-casa.yaml` files are usually the default configuration files used for CoffeaCasaCluster:

Example of a file:

```
jobqueue:
  coffea-casa:

    # Dask worker options, taken from https://github.com/dask/dask-jobqueue/tree/master/
    ↪dask_jobqueue
    cores: 4                # Total number of cores per job
    memory: "6 GiB"         # Total amount of memory per job
    processes: null         # Number of Python processes per jobs
    worker-image: "coffeateam/coffea-casa-analysis:0.xx.xx"

    # Communication settings
    interface: null         # Network interface to use like eth0 or ib0
    death-timeout: 60       # Number of seconds to wait if a worker can not find a
    ↪scheduler
    local-directory: null   # Location of fast local storage like /scratch or $TMPDIR
    extra: []

    # HTCondor Resource Manager options
    disk: "5 GiB"          # Amount of disk per worker job
    env-extra: []
    job-extra: {}          # Extra submit attributes
    log-directory: null
    shebang: "#!/usr/bin/env condor_submit -spool"

    # Scheduler options
    scheduler-options: {}
    name: dask-worker
```

To configure a cluster that is launched using it, you should adjust the Dask configuration file, typically stored at `~/.config/dask/labextension.yaml` or `/etc/dask/labextension.yaml`.

```
labextension:
  factory:
    module: 'coffea_casa'
```

(continues on next page)

(continued from previous page)

```

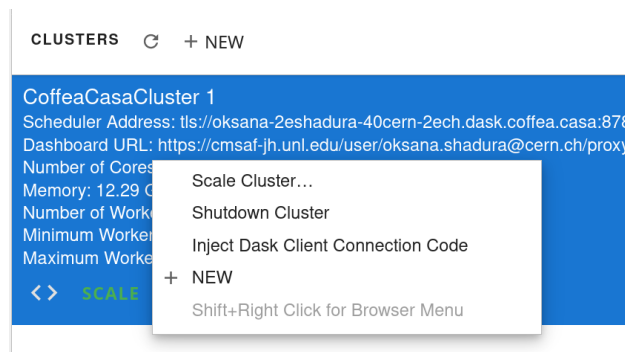
class: 'CoffeaCasaCluster'
args: []
kwargs: {}

default:
  workers: 1
  adapt:
    minimum: 5
    maximum: 10

```

Users can edit `kwargs: {}` to change a `CoffeaCasaCluster` constructor directly (see more details in [Coffea-Casa Setup Without Dask Labextension](#)).

To get an address of scheduler that will be used during client connection, try right-clicking on the cluster in the sidebar:



And then pressing *Inject Dask Client Connection Code*, as is shown in example below:

```

from dask.distributed import Client
client = Client("tls://oksana-2eshadura-40cern-2ech.dask.coffea.casa:8786")
client

```

or, more simply:

```

from dask.distributed import Client
client = Client("tls://localhost:8786")
client

```

**DEPLOYMENT OF COFFEA-CASA ANALYSIS FACILITY AT YOUR  
TIER 2/TIER 3 GRID SITE OR CLUSTER**





## **COFFEA\_CASA MODULE API**

```
class coffea_casa.CoffeaCasaCluster(*, security=None, worker_image=None, scheduler_options=None,  
                                     scheduler_port=8786, dashboard_port=8787, nanny_port=8001,  
                                     **job_kwargs)
```

This is a subclass expanding settings for launch Dask via HTCondorCluster over HTCondor in US.CMS facility.

```
classmethod security()
```

Return the Dask Security object used by CoffeaCasa.



## COMMUNITY SUPPORT AND HELP

Coffea-casa is deployed at CMS Nebraska Tier 2 grid site and developed by a group of developers from University of Nebraska-Lincoln, University of Nebraska Holland Computing Center, University of Wisconsin-Madison and Morgridge Institute.

We gratefully acknowledge the National Science Foundation which supported this work through NSF grant #1836650.

### 10.1 Discussion

Conversation happens in the following places:

1. **Bug reports and feature requests** are managed through [GitHub issues](#)
2. **Ask questions:** You can ask questions by sending email to [e-group](#), adding a GitHub issue, or contacting us in IRIS-HEP Slack channel ([#coffea-casa](#)).

---

**Important:** Coffea-casa at Nebraska technical support through [GitHub Discussions](#).

---

### 10.2 Tech Support

[Here](#) is the link to tech support.



## INDEX

### C

CoffeaCasaCluster (*class in coffea\_casa*), [37](#)

### S

security() (*coffea\_casa.CoffeaCasaCluster class method*), [37](#)