Next-Gen Analysis Tools are Now!
(Biased towards columnar analysis tools / coffea)

Lindsey Gray

LPC Physics Forum

29 Sept. 2022

Overview

- Today's "Physics" Forum is going to be organized into two parts
  - "Physics" because this is about mechanically how we do our data science

- Part 1: What's awkward array, uproot, coffea / what is provided / extensibility
  - Basics, histogramming, corrections, piping things to combine / ML

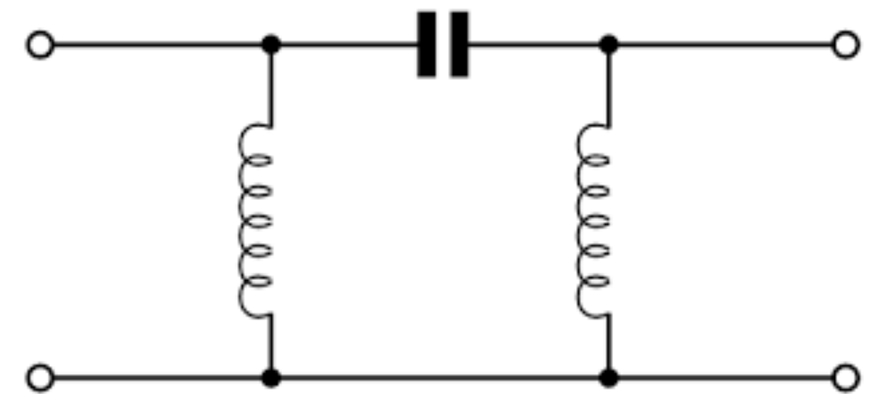
- Part 2: More open-ended discussion on why it's useful to change over (early)
  - Mechanically in terms of computation
  - Entering into design as users
  - Professional development concerns
  - Why should you try new things now as opposed to later?

Intro to Columnar Analysis / Coffea

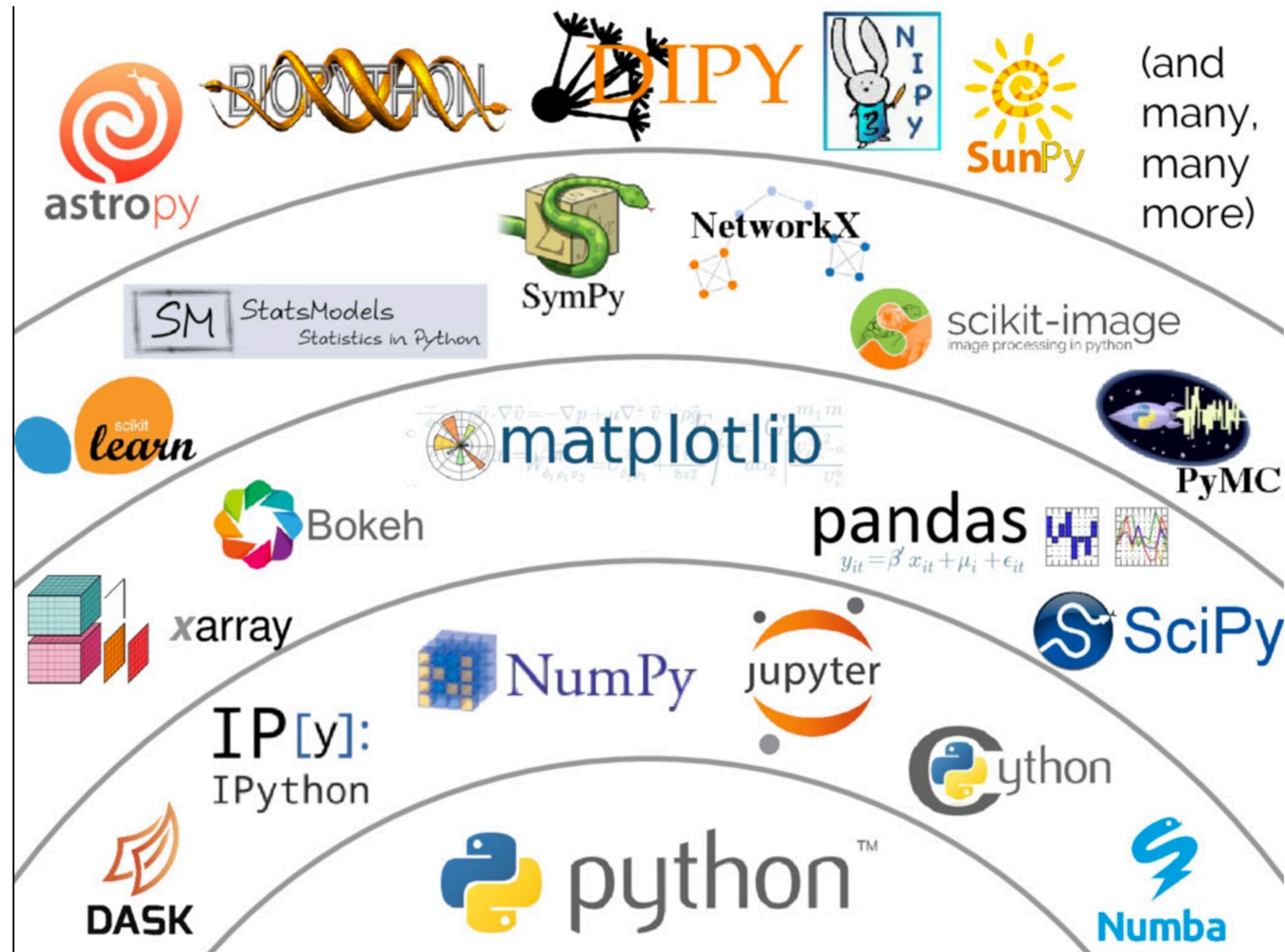
Fermilab

Impedance Mismatches

- ROOT File <-> Machine Learning
- Big data <-> PyROOT
- HEP Physicist <-> Industry



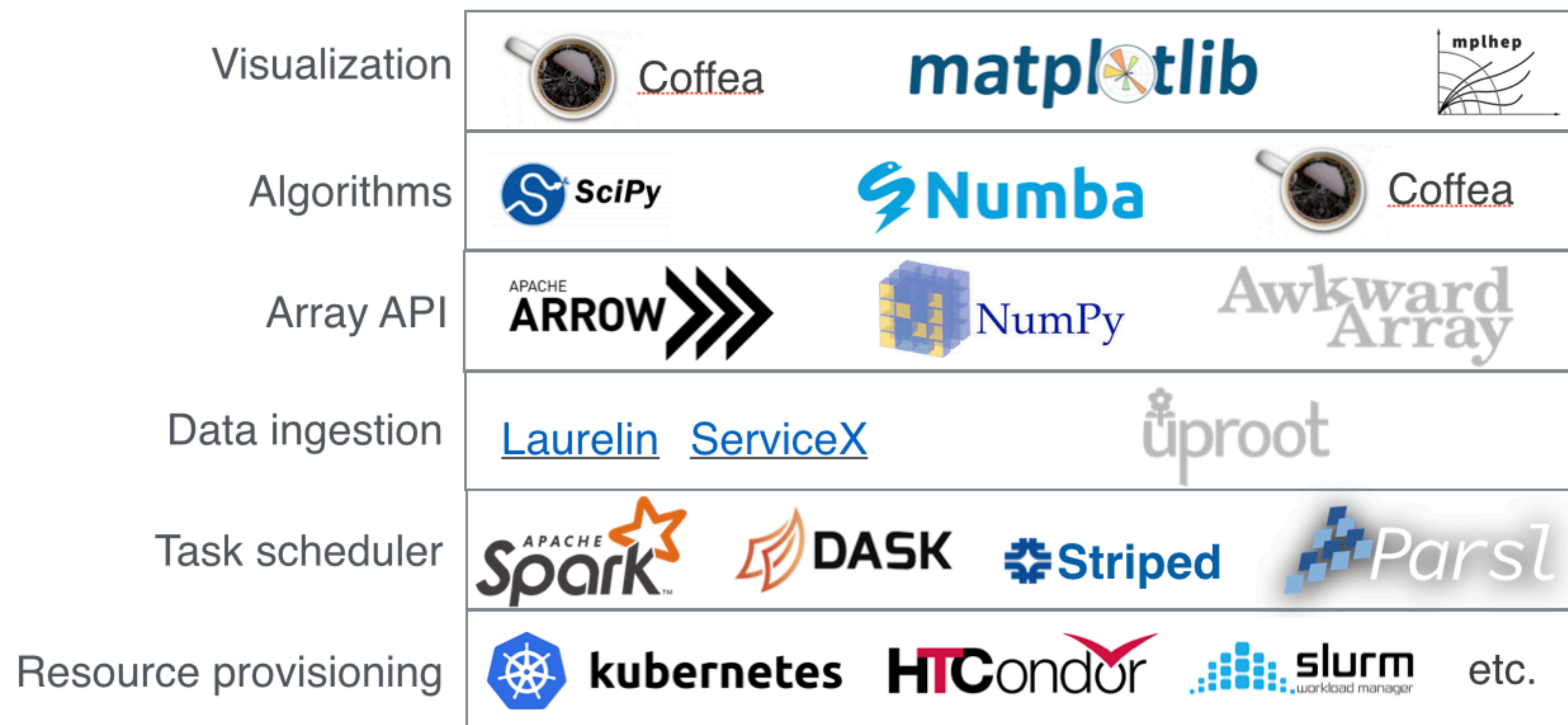
Scientific Python



Fermilab

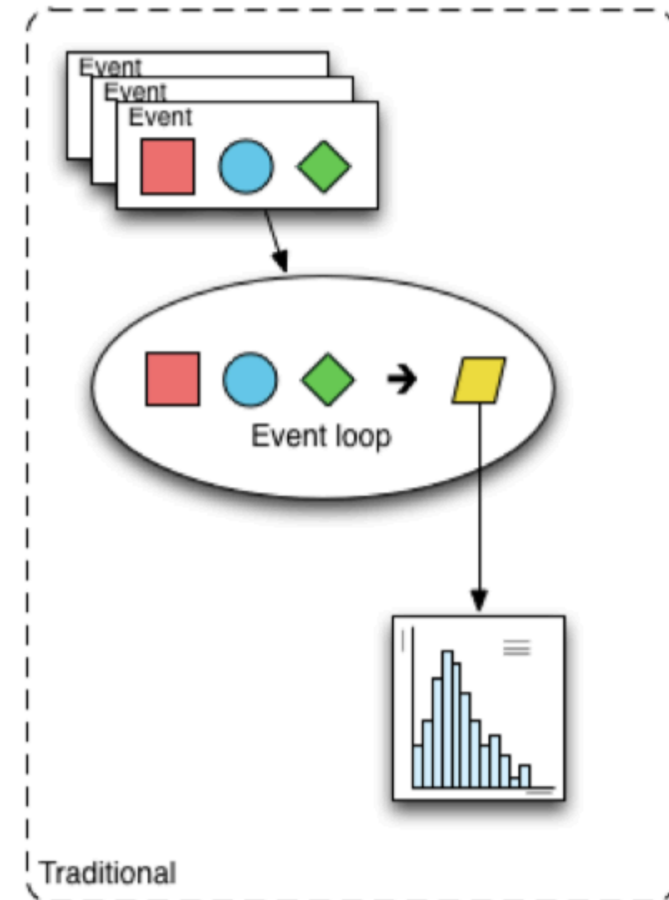
Coffea is

- A package in the scientific python ecosystem
  - `$ pip install coffea`
- A user interface for columnar analysis
  - With missing pieces of the stack filled in
- A minimum viable product
  - We are data analyzers too #dogfooding
- A really strong glue

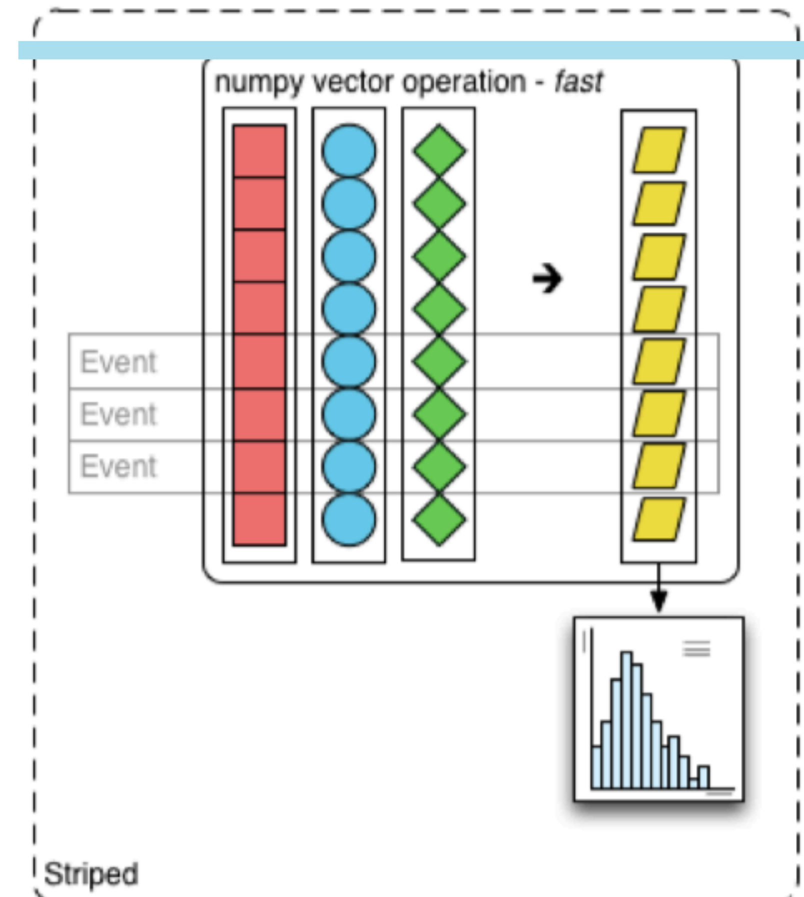


What is columnar analysis?

- Event loop analysis:
  - Load relevant values for a specific event into local variables
  - Evaluate several expressions
  - Store derived values
  - Repeat (explicit outer loop)



- Columnar analysis:
  - Load relevant values for many events into contiguous arrays
  - Evaluate several **array programming** expressions
    - Implicit *inner* loops
  - Store derived values



Ferm

Concrete example

```
void MyClass::Loop() {
 size_t nEvents;
 // load...

 for (Long64_t iEvent=0; iEvent<nEvents; iEvent++) {
  double MET_pt;
  int nElectron;
  double * Electron_pt;
  double * Electron_eta;
  // load...

  if ( MET_pt > 100. ) continue;

  for(size_t iEl=0; iEl<nElectron; ++iEl) {
   if ( Electron_pt[iEl] > 30. ) {
    hist->Fill(Electron_eta[iEl]);
   }
  }
 }
}
```

Event loop

Fermilab

Concrete example

```
void MyClass::Loop() {
  size_t nEvents;
  // load...

  for (Long64_t iEvent=0; iEvent<nEvents; iEvent++) {
    double MET_pt;
    int nElectron;
    double * Electron_pt;
    double * Electron_eta;
    // load...

    if ( MET_pt > 100. ) continue;

    for(size_t iEl=0; iEl<nElectron; ++iEl) {
      if ( Electron_pt[iEl] > 30. ) {
        hist->Fill(Electron_eta[iEl]);
      }
    }
  }
}
```

Event loop

```
void MyClass::Loop() {
  size_t nEvents;
  double * MET_pt;
  int * nElectron;
  size_t nElectron_flat;
  double * Electron_pt;
  double * Electron_eta;
  // load...

  bool * eventmask = allocate(nEvents);
  for (size_t i=0; i<nEvents; i++)
    eventmask[i] = MET_pt[i] > 100.;

  bool * entrymask = allocate(nElectron_flat);
  for (size_t i=0; i<nElectron_flat; ++i)
    entrymask[i] = Electron_pt[i] > 30.;

  bool * entrymask2 = allocate(nElectron_flat);
  size_t * parents = get_parents(nEvents, nElectron);
  for (size_t i=0; i<nElectron_flat; ++i)
    entrymask2[i] = eventmask[parents[i]] & entrymask[i];

  double * take_result = allocate(nElectron_flat);
  size_t idx = 0;
  for (size_t i=0; i<nElectron_flat; ++i)
    if ( entrymask2[i] )
      take_result[idx++] = Electron_eta[i];

  for (size_t i=0; i<idx; i++)
    hist->Fill(take_result[i]);
}
```

Columnar

Fermilab

Concrete example

```
void MyClass::Loop() {
  size_t nEvents;
  // load...

  for (Long64_t iEvent=0; iEvent<nEvents; iEvent++) {
    double MET_pt;
    int nElectron;
    double * Electron_pt;
    double * Electron_eta;
    // load...

    if ( MET_pt > 100. ) continue;

    for(size_t iEl=0; iEl<nElectron; ++iEl) {
      if ( Electron_pt[iEl] > 30. ) {
        hist->Fill(Electron_eta[iEl]);
      }
    }
  }
}
```

Event loop

```
cut = (events.MET.pt < 100.) & (events.Electron.pt > 30.)
hist.fill(eta=events.Electron.eta[cut].flatten())
```

Columnar

Scaling out

- User is provided data frame of columns they wish to process
- User fills a defined set of accumulators
  - Histograms, dictionaries of counts, appendable arrays, …
- Coffea executor takes care of the rest
  - Local machine, dask, spark, parsl (and condor)

```
from coffea import hist, processor

class MyProcessor(processor.ProcessorABC):
    def __init__(self, flag=False):
        self._flag = flag
        self._accumulator = processor.dict_accumulator({
            # Define histograms
        })

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

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

        # PHYSICS GOES HERE

        return output

    def postprocess(self, accumulator):
        return accumulator

p = MyProcessor()
```

coffea executor

ROOT files
Parquet files
…

map

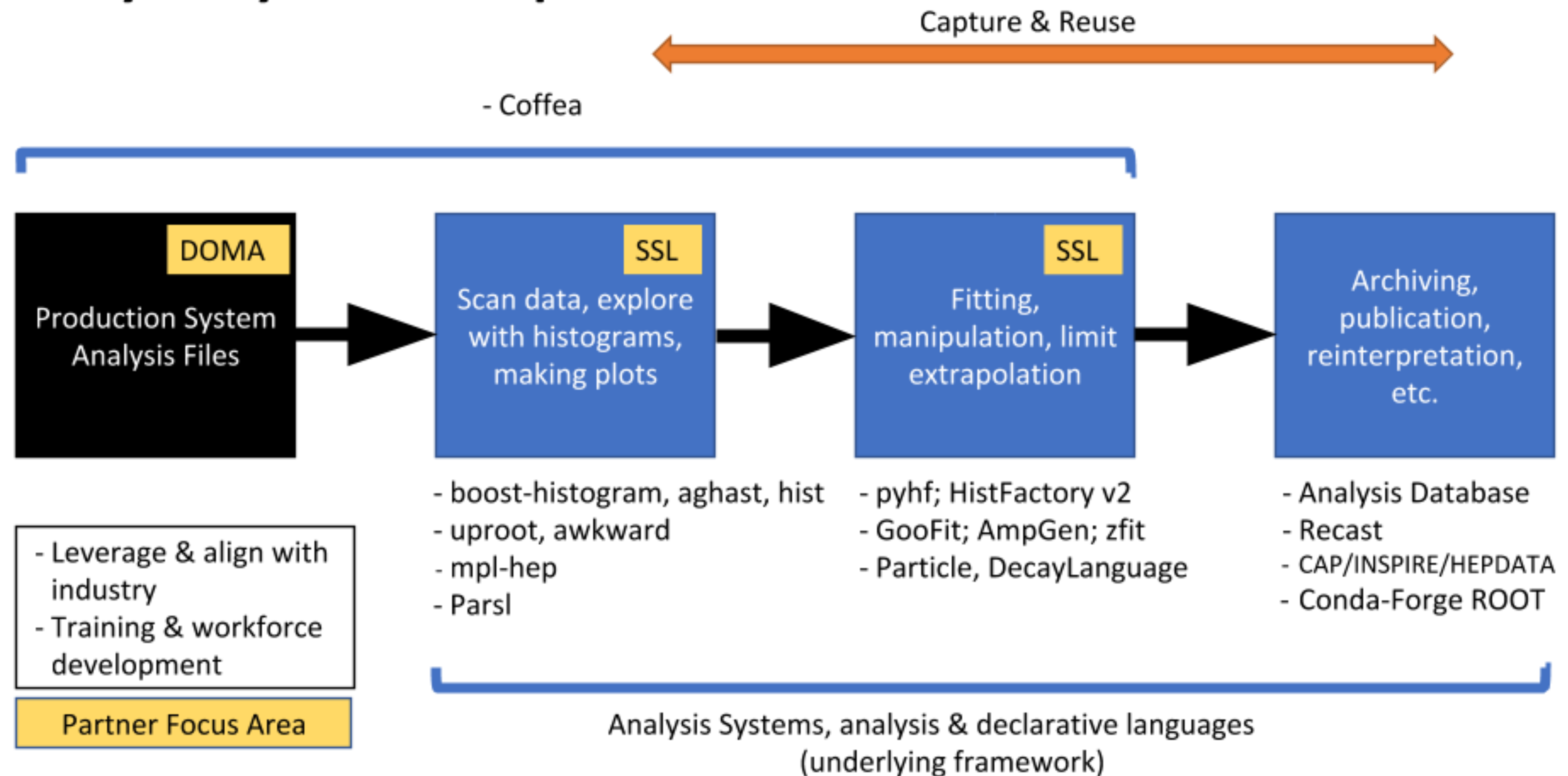
coffea processor

reduce

Histograms
Event lists
…

In context

- IRIS-HEP Analysis systems group
- https://iris-hep.org/as.html



nanoevents - data efficient, lazy access root files

```python
from coffea import processor, hist

class MyZPeak(processor.ProcessorABC):
    def __init__(self):
        self._histo = hist.Hist(
            "Events",
            hist.Cat("dataset", "Dataset"),
            hist.Bin("mass", "Z mass", 60, 60, 120),
        )

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

    # we will receive a NanoEvents instead of a coffea DataFrame
    def process(self, events):
        out = self.accumulator.identity()
        mmevents = events[
            (ak.num(events.Muon) == 2)
            & (ak.sum(events.Muon.charge, axis=1) == 0)
        ]
        zmm = mmevents.Muon[:, 0] + mmevents.Muon[:, 1]
        out.fill(
            dataset=events.metadata["dataset"],
            mass=zmm.mass,
        )
        return out

    def postprocess(self, accumulator):
        return accumulator
```

- Awkward array based toolkit for transfer-efficient operations with root files
- Generalized object oriented access to ROOT file columns as physics objects
  - https://coffeateam.github.io/coffea/notebooks/nanoevents.html
- Watch the snazzy YouTube tutorial from PyHep 2020
  - https://www.youtube.com/watch?v=McKSS_WjLwU

Distributed Computing with Coffea

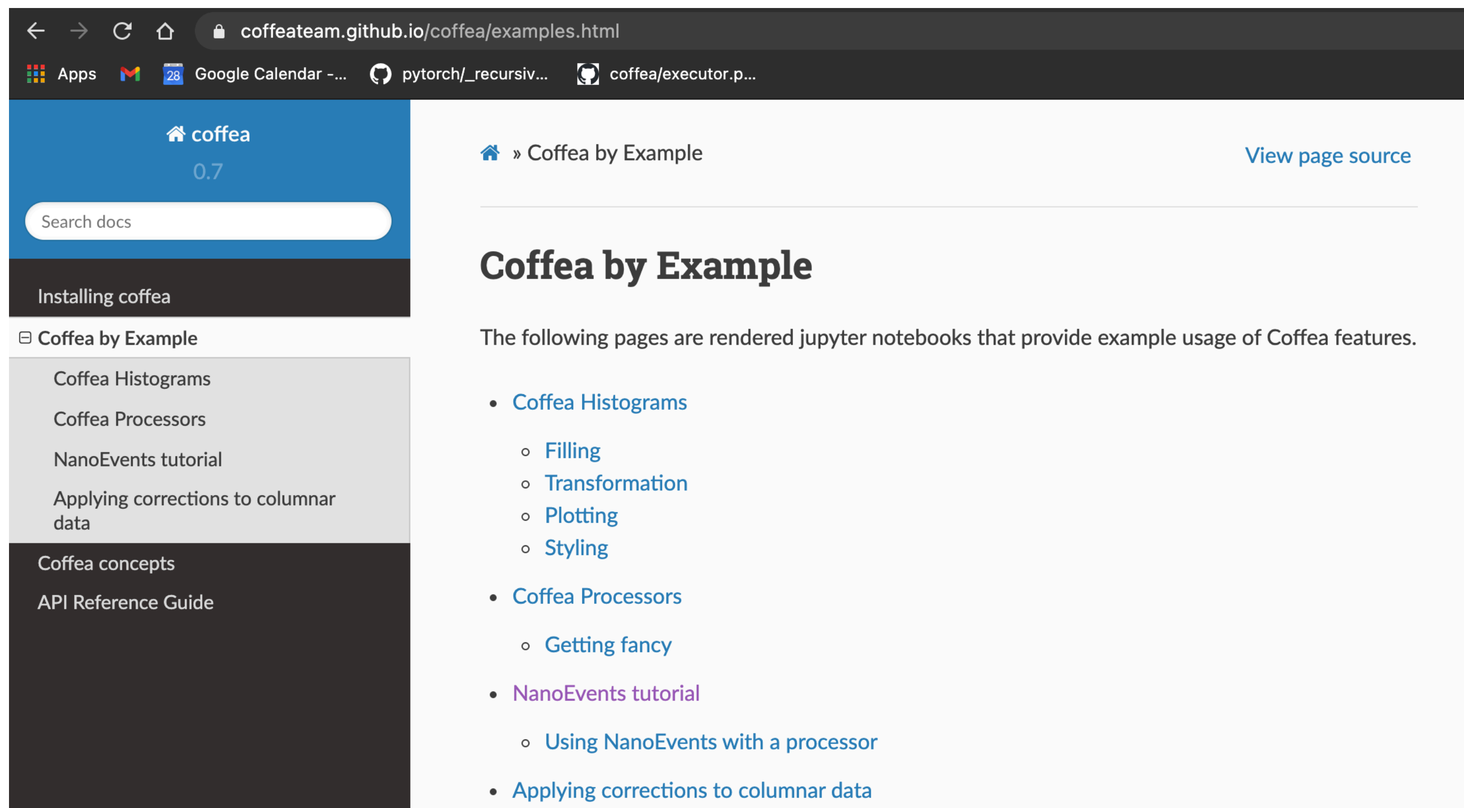
- We actively support and maintain a variety of execution engines
  - Dask, Apache Spark, Parsl, WorkQueue - choose your poison
  - As well as prototypes aimed at HL-LHC: ServiceX and SkyHook
  - If you've got one that you like - add it and we'll happily support it if the tests pass

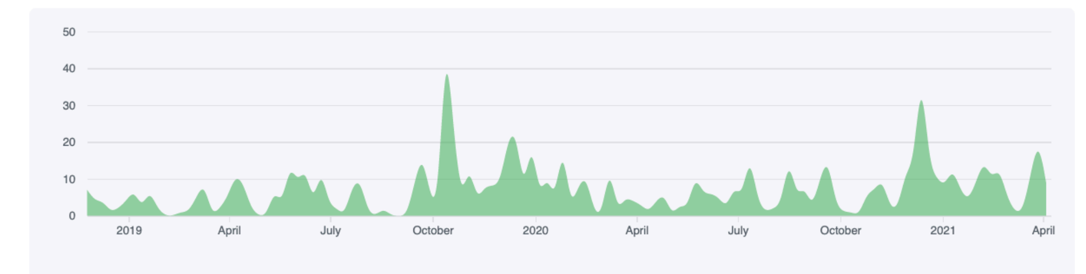
- Each execution engine supports a variety of clusters
  - Not maintained by coffea but rather those execution engines' projects
  - HTCondor, slurm, pbs, specialized batch systems for supercomputers
  - We have successfully run coffea + dask/parsl analyses on super computers and a variety of difficult condor setups (FNAL, CERN, weird nested kubernetes stuff)
  - Flexibility in, e.g., Dask to setup our own flavors of cluster interfaces when needed

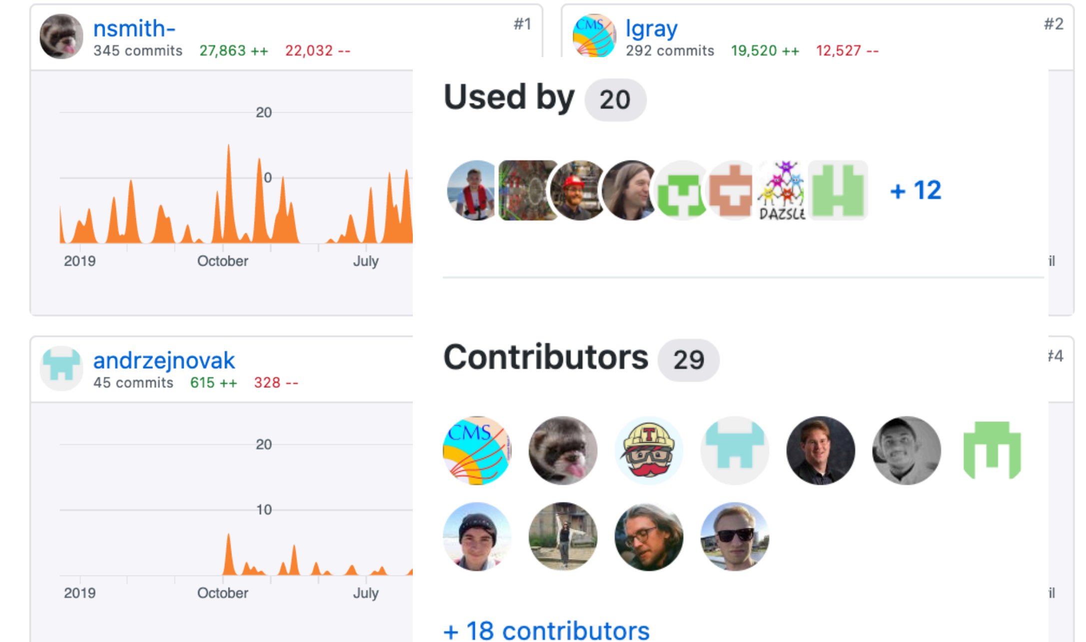
- Containerized - batch-ready coffea(-dask) singularity images on cvmfs

- Suffices to say that analyses written in coffea are highly portable
  - Doesn't care what batch system you use and nowhere is such an assumption made
  - The only constraint is that your batch system does not have 5-minute enqueueing times

Coffea documentation and support







- Extensive documentation of code base in multiple forms
  - Basic documentation website
  - Jupyter Notebooks
  - YouTube videos
- Significant use by other projects, large contributor base
  - Intend to keep this project going for a long time
  - 65 direct forks of the coffea repository
  - Standard open-source core + community supported model

Coffea Corrections and ML

```
from coffea.btag_tools import BTagScaleFactor

btag_sf = BTagScaleFactor("data/DeepCSV_102XSF_V1.btag.csv.gz", "medium")

print("SF:", btag_sf.eval("central", events.Jet.hadronFlavour, abs(events.Jet.eta), events.Jet.pt))
print("systematic +:", btag_sf.eval("up", events.Jet.hadronFlavour, abs(events.Jet.eta), events.Jet.pt))
```

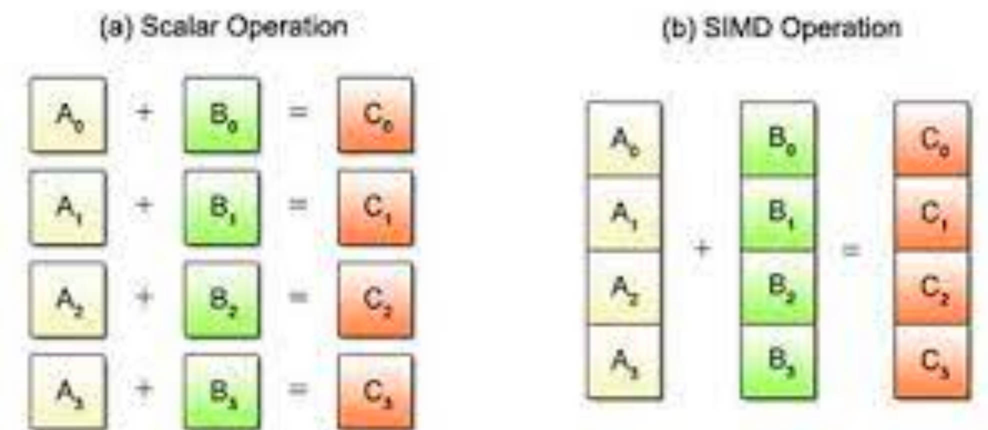
```
import torch
import awkward as ak

model = torch.load('/some/model.pt')
x = ak.Array([[1., 2., 3.], [4., 5.], [6.]])
# find the x with largest probability in a given event, assuming 'model' is trained to do that
probs = ak.softmax(ak.unflatten(model(torch.tensor(ak.flatten(x))).numpy(), ak.num(x)), axis=1)
```

- Via awkward arrays coffea can process most if not all tabular/columnar data
  - Can ingest parquet, root by default and extensible to any reasonable file format
- Often we want to apply corrections to our data or make variations to estimate the effect of systematics
  - Coffea has tools that make bookkeeping easy for this task for all corrections used by CMS
  - This will be upgraded to correctionlib once it is ready (N. Smith is an author on both)
- All ML toolkits natively use flat columnar data
  - Awkward arrays are compatible with all ML tools by default, and is differentiable

Discussion and Talking Points

1 - Why to change from the computational perspective

- Modern CPUs all have vector processors embedded within them that you can access with specialized instructions
  - This means that one instruction can operate on multiple pieces of data give that data is organized correctly in the various tiers of memory within a computer
  - Awkward array and RDF both attempt to organize data such that these instructions can be issues for computation gain while doing your analysis
  - As such, your programs will achieve faster throughput for less input effort by following patterns that facilitate the appropriate organization in memory



- A curiosity: this same pattern is the fundamental operating pattern of GPUs
  - It is therefore how most ML data is organized, and as ML continues to become more prevalent in analysis design keeping things organized similar helps us a ton!
  - Using numpy-like idioms (like awkward array) can help you write programs which can be used on GPUs with little change to code for immediate improvements in processing speed (see hepaccelerate)

2 - Entering into design as a user

- Think critically about your analysis workflow
  - This recent presentation is a spicy and useful take on physicists in the broader context of modern data science (h/t Nick Smith)
  - In the long run you want to do more science and write less code
  - It's always worth it to constructively criticize the systems you are using, but it requires a very different mindset from how we tend to work
    - This kind of thinking towards what can be made more efficient led directly to the creation of packages like uproot, awkward array, dask, etc.
  - As much as we want to get to the science as quickly as possible, hastily declaring things not to work isn't very helpful, and getting things done is more than "just write the script"

- Try things out and see if it improves the situation at least for you!
  - If it does - share it!
- Talk to people working on analysis software your ideas and tests early and often!
  - Siloing knowledge because of perceived advantage is fractious in a scientific collaboration
  - Folks managing higher level packages can offer perspective for further improvement or a platform to promote your visibility
  - The less time we collectively spend retreading the same ideas, the faster we can advance

3 - Professional Development Concerns

- As a field we want to train the next generation of successful professors and staff scientists
  - However, that is not the end point for the majority of those we train (not a bad thing!)
  - But - we do not corner the market on data science techniques as we did in the past
    - Actually, there's been a fair amount of innovation from non-academic scientists (not just HEP) that led to numpy, pandas, dask, etc. that's similar to the way that we'd like to evolve within HEP analysis software
  - HEP can easily be considered a subfield of data science these days

- It's useful to speak in the same concepts to the wider field about how you view your data
  - This lets your draw upon innovation well outside your colleagues in HEP
  - Allows you to concisely demonstrate your abilities with data analysis to people you will interview with, regardless of their background (~everybody knows numpy brackets)
  - Reduces technical aspects of bootcamps, improved focus on changes in what you actually do

4 - So, why is it good to be an early adopter?

- Quite plainly, it's nice to go faster sooner
  - The tools and usage patterns are fairly well known and there's enough of a community that people can help each other out.
  - Hence, you can benefit rather quickly from the faster turn around time the computational improvements afford, and characterize your knowledge as a physicist in a language more and different data scientists understand

- More importantly, you can help shape the tool you want to use
  - Everyone has different preferences but we're all doing the same thing, there are patterns to take advantage of towards collective improvement
  - Expressing your preferences to people designing the software helps them deduce usage patterns and interfaces that are actually effective
    - Where is automation important? What's the right level of bookkeeping at different stages of processing? How to extract information about processing steps and at what level of detail?, etc. can only be truly refined by talking to a broad spectrum of users since we're all inventive in fairly different ways

- Finally, you get to own more of the project and make unique contributions
  - It's not just one person's vision for analysis that matters and more often than not tools can be shared, especially given a well defined common interface