ETH zürich

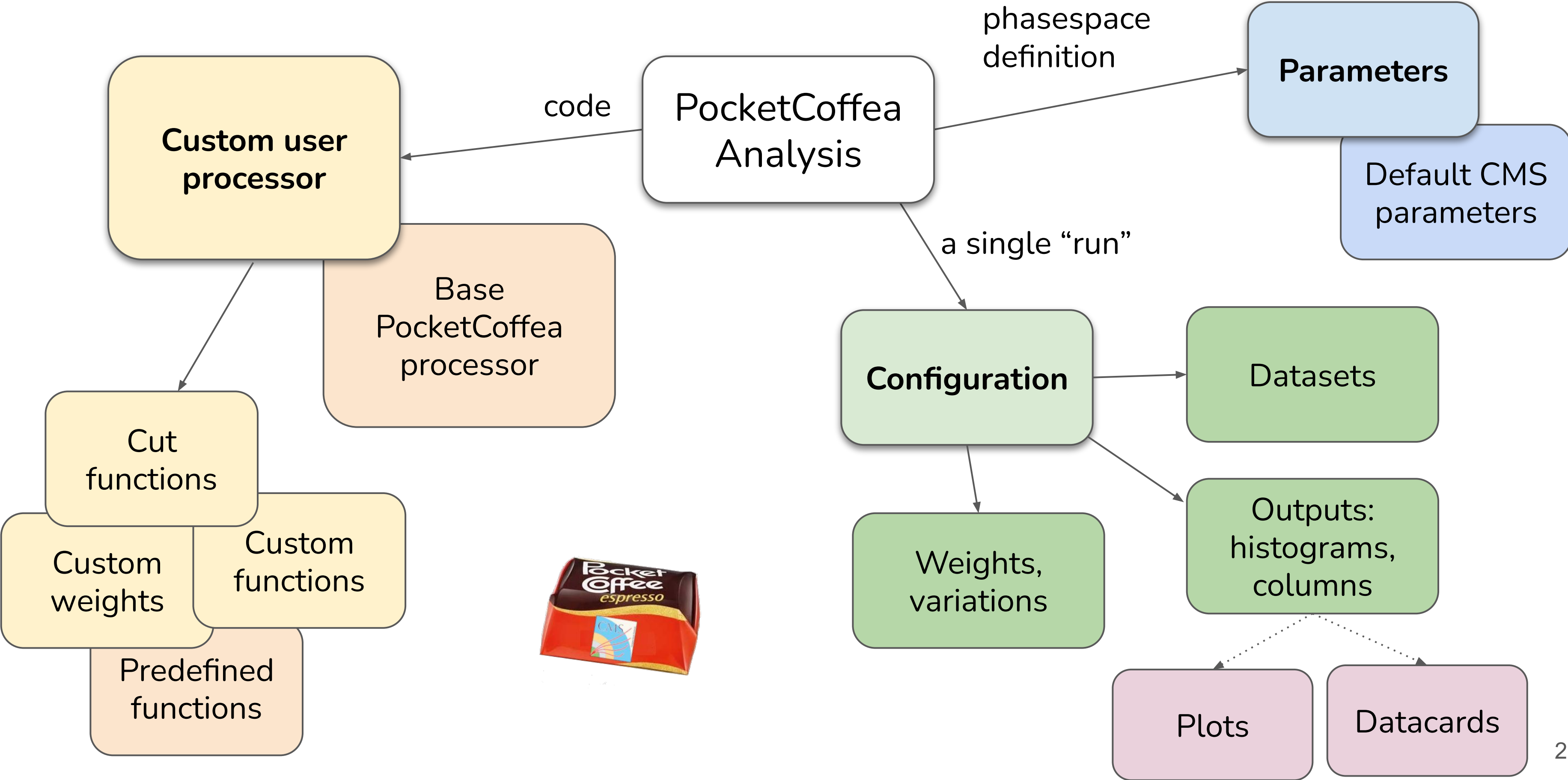
PocketCoffea declarative analysis in PyHEP

Davide Valsecchi (ETH Zurich),
Tommaso Tedeschi (INFN Perugia)

INFN Quasi-Interactive Analysis Workshop

09/01/2025

PocketCoffea: declarative analysis with Coffea



A customizable CMS analysis workflow

PocketCoffea implements a Coffea processor **containing all the standard steps of a CMS templated analysis:**

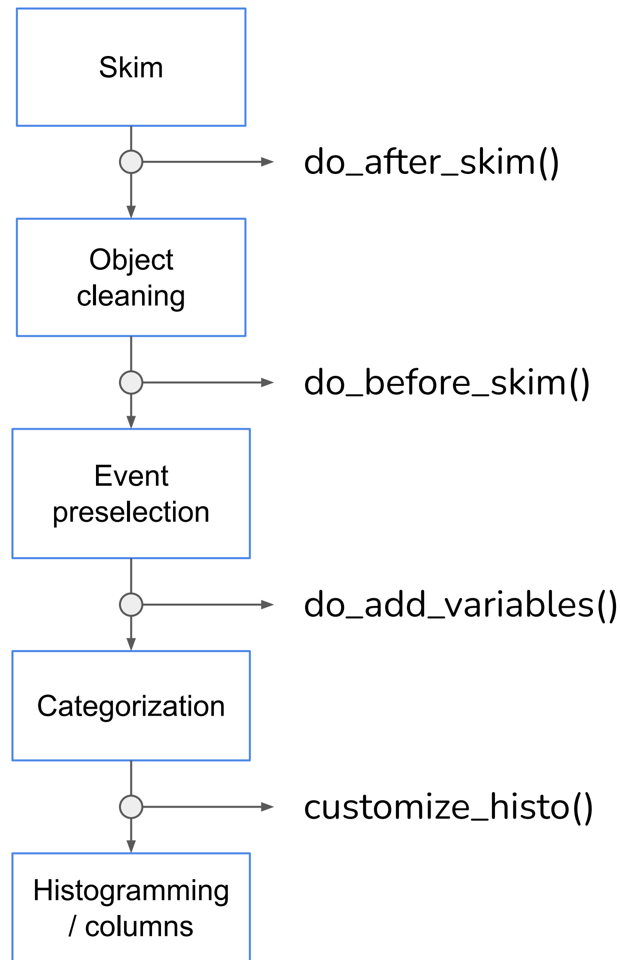
- skimming
- object preselection and calibration
- Event preselections
- Categorization
- Histogramming
- Ntuples export

→ **Fully configurable from a python dictionary** → see next slide

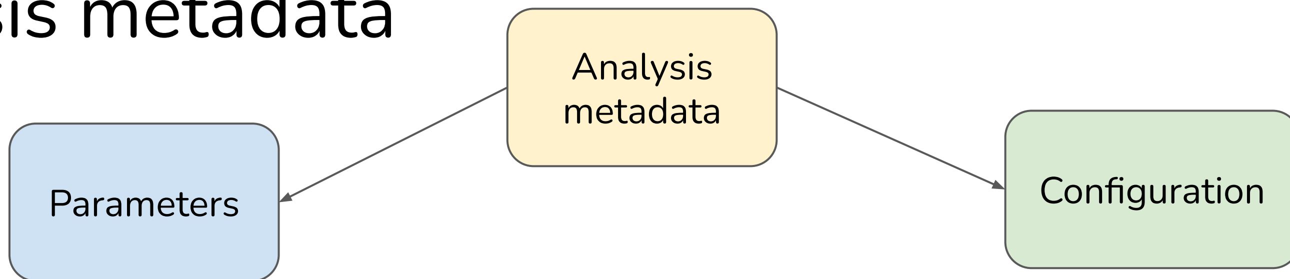
Custom workflow implemented as **derived processor class** with **predefinite entry points**

- Custom event cleaning
- Custom object selection
- Custom variables
- Custom weights computation

For these customizations, the user needs to **expand the** Base processor **code** and/or the libraries containing the parameters and cut functions



Analysis metadata



“Phasespace” definition:

- triggers, lumi, event flags
- objects working point, calibration version, scale factors

Once well defined and tested it can be **shared between groups** to have a common ground.

Handled with **composable yaml files**

Analysis “run” definition:

- list of datasets
- categories, weights, variations
- output: histograms and columns

An analysis lives in a set of configuration which needs to be run in a certain order (workflow management tools!)

Python configuration: *Configurator* object

Configuration

keep track of all the parameters in a **single config file**

- Define all the relevant parameters for **running**:
  - → input dataset to process
  - → workflow and output folder
  - → executor parameters
- Define **cutflow** by defining cut functions **dynamically**
  - → skimming, preselections and categorization
- Define **weights** to apply by category / by sample
- Define **variations** to store by category / by sample
- List **histograms** to be produced and parameters:
  - → customize binning, labels, etc.
- Additional **analysis-specific parameters** can be also defined

Working now to generalize the parameters in the framework to make it fully analysis independent.

input dataset

```
9
10  cfg =  {
11      "dataset" : {
12          "jsons": ["datasets/signal_ttHTobb_lxplus.json",
13                    "datasets/backgrounds_MC.json"],
14          "filter" : {
15              "samples": ["TTToSemiLeptonic"],
16              "samples_exclude" : [],
17              "year": ['2018']
18          }
19      },
20
21      # Input and output files
22      "workflow" : ttHbbBaseProcessor,
23      "output"   : "output/test_base",
24      "worflow_options" : {},
25
26      "run_options" : {
27          "executor"       : "dask/lxplus",
28          "workers"        : 1,
29          "scaleout"       : 50,
30          "queue"          : "microcentury",
31          "walltime"       : "00:40:00",
32          "mem_per_worker" : "4GB", # GB
33          "exclusive"      : False,
34          "chunk"          : 100000,
35          "retries"        : 50,
36          "treereduction"  : 10,
37          "max"            : None,
38          "skipbadfiles"   : None,
39          "voms"           : None,
40          "limit"          : None,
41          "adapt"          : False,
42      },
43
44      # Cuts and plots settings
45      "finalstate" : "semileptonic",
46      "skim": [get_nObj_min(4, 15., "Jet") ],
47      "preselections" : [semileptonic_presel_nobtag],
```

cuts

```
48      "categories": {
49          "baseline": [passthrough],
50          "1b" : [ get_nBtag(1, coll="BJetGood")],
51          "2b" : [ get_nBtag(2, coll="BJetGood")],
52          "3b" : [ get_nBtag(3, coll="BJetGood")],
53          "4b" : [ get_nBtag(4, coll="BJetGood")]
54      },
55
56
57
58      "weights": {
59          "common": {
60              "inclusive": ["genWeight","lumi","XS",
61                            "pileup",
62                            "sf_ele_reco", "sf_ele_id",
63                            "sf_mu_id","sf_mu_iso",
64                            "sf_btag",
65                            "sf_btag_calib", "sf_jet_puId",
66                            ],
67              "bycategory" : {
68                  "4b": [
69                      WeightCustom(
70                          name="example",
71                          function= lambda events, size, metadata: [("p
72                      )
73                  ]
74              }
75          },
76          "bysample": {
77          }
78      },
79
80      "variations": {
81          "weights": {
82              "common": {
83                  "inclusive": [  "pileup",
84                                  "sf_ele_reco", "sf_ele_id",
85                                  "sf_mu_id", "sf_mu_iso", "sf_jet_puId
86
87                               ],
88                  "bycategory" : {
89                      "3b": [ f"sf_btag_{b}" for b in btag_variations["
90                  }
91              },
```

histograms

```
98      "variables":
99      {
100
101         **jet_hists(coll="JetGood"),
102         **jet_hists(coll="BJetGood"),
103         **ele_hists(coll="ElectronGood"),
104         **muon_hists(coll="MuonGood"),
105         **count_hist(name="nJets", coll="JetGood",bins=10, start=4, stop=14),
106         **count_hist(name="nBJets", coll="BJetGood",bins=12, start=2, stop=14),
107         **jet_hists(coll="JetGood", pos=0),
108         **jet_hists(coll="JetGood", pos=1),
109         **jet_hists(coll="JetGood", pos=2),
110         **jet_hists(coll="JetGood", pos=3),
111         **jet_hists(coll="JetGood", pos=4),
112         **jet_hists(name="bjet",coll="BJetGood", pos=0),
113         **jet_hists(name="bjet",coll="BJetGood", pos=1),
114         **jet_hists(name="bjet",coll="BJetGood", pos=2),
115         **jet_hists(name="bjet",coll="BJetGood", pos=3),
116         **jet_hists(name="bjet",coll="BJetGood", pos=4),
```

Categorization

- Skimming, preselection, categorization is handled by **Cut** objects:
  - ⟶ Function that can be parametrized and reused.
- All the steps are configured from the python cfg file.
- Implemented different flavour of categorization
  - ⟶ Standard
  - ⟶ “Cartesian” combination of binnings (for differential analysis)
- Supporting both 1D and 2D masks:
  - ⟶ we can analyze the “jet” collection instead of working event-by-event

```
def count_objects_lt(events, params, year, sample):
    '''
    Count the number of objects in `params["object"]`
    keep only events with smaller (<) amount than `pa
    '''
    mask = ak.num(events[params["object"]], axis=1) <
    return ak.where(ak.is_none(mask), False, mask)
```

```
dilepton_presel = Cut(
    name="dilepton",
    params={
        "METbranch": {
            '2016': "MET",
            '2017': "MET",
            '2018': "MET",
        },
        "njet": 2,
        "nbjet": 1,
        "pt_leading_lepton": 25,
        "met": 40,
        "mll": 20,
        "mll_SFOS": {'low': 76, 'high': 106},
    },
    function=cuts_f.dilepton,
)
```

```
"categories": CartesianSelection(
    multicuts = [
        MultiCut(name="Njets",
                 cuts=[
                     get_nObj_eq(4, 15., "JetGood"),
                     get_nObj_eq(5, 15., "JetGood"),
                     get_nObj_min(6, 15., "JetGood"),
                 ],
                 cuts_names=["4j","5j","6j"]),
        MultiCut(name="Nbjet",
                cuts=[
                    Cut("jet_eta",{"coll":"JetGood", "eta_max":0.5}, jet_eta_cut, collection="JetGood")
                    Cut("jet_eta",{"coll":"JetGood", "eta_max":1}, jet_eta_cut, collection="JetGood"),
                    Cut("jet_eta",{"coll":"JetGood", "eta_max":1.5}, jet_eta_cut, collection="JetGood")
                ],
                 cuts_names=["jeta0.5", "jeta1","jeta1.5"])
    ],
    common_cats = StandardSelection({
        "inclusive": [passthrough],
        "4jets_40pt" : [get_nObj_min(4, 40., "JetGood")]
    })
),
```

Histogram configuration

```
# 2D plots
"jet_eta_pt_leading": HistConf(
    [
        Axis(coll="JetGood", field="pt", pos=0, bins=40, start=0, stop=1000,
             label="Leading jet $p_T$"),
        Axis(coll="JetGood", field="eta", pos=0, bins=40, start=-2.4, stop=2.4,
             label="Leading jet $\eta$"),
    ]
),
"jet_eta_pt_all": HistConf(
    [
        Axis(coll="JetGood", field="pt", bins=40, start=0, stop=1000,
             label="Leading jet $p_T$"),
        Axis(coll="JetGood", field="eta", bins=40, start=-2.4, stop=2.4,
             label="Leading jet $\eta$")
    ]
),
```

```
**jet_hists(coll="JetGood"),
**jet_hists(coll="BJetGood"),
**ele_hists(coll="ElectronGood"),
**muon_hists(coll="MuonGood"),
**count_hist(name="nJets", coll="JetGood",bins=10, start=4, stop=14),
**count_hist(name="nBJets", coll="BJetGood",bins=12, start=2, stop=14),
```

Histograms are assembled in the configuration and **filled automatically**

- Special axis attributes are used to defined what to use for filling:
- **coll:** collection “Electron”
- **field:** “p_T”
- **pos:** 0 (index of the object in the collection, if None, accumulate all objects)

Filling custom histograms all the time **without changing the processor code**

Useful **defaults** and **factory methods** are also provided

Arrays export

```
variables = {},
columns = {
    "common": {
        "inclusive": [],
        "bycategory": {
                "semilep_LHE": [
                    ColOut(
                        "Parton",
                        ["pt", "eta", "phi", "mass", "pdgId", "provenance"],
                        flatten=False
                    ),
                    ColOut(
                        "PartonMatched",
                        ["pt", "eta", "phi","mass", "pdgId", "provenance", "dRMatchedJet"],
                        flatten=False
                    ),
                    ColOut(
                        "JetGood",
                        ["pt", "eta", "phi", "hadronFlavour", "btagDeepFlavB", "btag_L", "btag_M", "btag_H"],
                        flatten=False
                    ),
                    ColOut(
                        "JetGoodMatched",
                        ["pt", "eta", "phi", "hadronFlavour", "btagDeepFlavB", "btag_L", "btag_M", "btag_H", "dRMatchedJet"],
                        flatten=False
                    ),
```

```
@dataclass
class ColOut:
    collection: str  # Collection
    columns: List[str]  # list of columns to export
    flatten: bool = True  # Flatten by default
    store_size: bool = True
    fill_none: bool = True
    fill_value: float = -999.0  # by default the None elements are filled
    pos_start: int = None  # First position in the collection to export. If None export from the first element
    pos_end: int = None  # Last position in the collection to export. If None export until the last element
```

Collections from the NanoAOD can be exported to parquet

- done with a very similar code as in HiggsDNA
- Configured from the configuration file: by sample, category

Planned improvements on the arrays export:

- Export systematic variations (trivial feature to add)
- Export only “changed” columns for systematic variations

Useful **defaults** and **factory methods** are also provided

Parameters

docs, code

The idea is to have some **CMS defaults** ready for the users and then add additional files or modify the existing one:

- Completely **free schema** (keeping a meaningful structure for default objects)
- Easy to **build on top** of another configuration
- Each analysis run **saves its own parameters** set for reuse
- **Composable**: many groups can share a set of common files without copy and paste errors
- Add parameters *programmatically*: e.g. by an workflow stop

```
18    # Loading default parameters
19    from pocket_coffea.parameters import defaults
20    default_parameters = defaults.get_default_parameters()
21    defaults.register_configuration_dir("config_dir", localdir+"/params")
22
23    parameters = defaults.merge_parameters_from_files(default_parameters,
24                                                      f"{localdir}/params/object_preselection.yaml",
25                                                      f"{localdir}/params/triggers.yaml",
26                                                      update=True)
```

Defaults

The defaults contain configs for UL Run2: live in pocket_coffea.parameters

- lumi and pileup
- Event flags
- Jet calibrations: different versions (with or w/o JER, with or w/o uncertainties)
- Jet scale factors
- Lepton scale factors

They do not include:

- trigger configuration
- objects preselections

These must be defined by each analysis (zmumu example)

```yaml
jet_scale_factors:
  btagSF:
    # DeepJet AK4 tagger shape SF
    '2016_PreVFP':
      file: /cvmfs/cms.cern.ch/rsync/cms-nanoAOD/
      name: "deepJet_shape"
    '2016_PostVFP':
      file: /cvmfs/cms.cern.ch/rsync/cms-nanoAOD/
      name: "deepJet_shape"
    '2017':
      file: /cvmfs/cms.cern.ch/rsync/cms-nanoAOD/
      name: "deepJet_shape"
    '2018':
      file: /cvmfs/cms.cern.ch/rsync/cms-nanoAOD/
      name: "deepJet_shape"
```

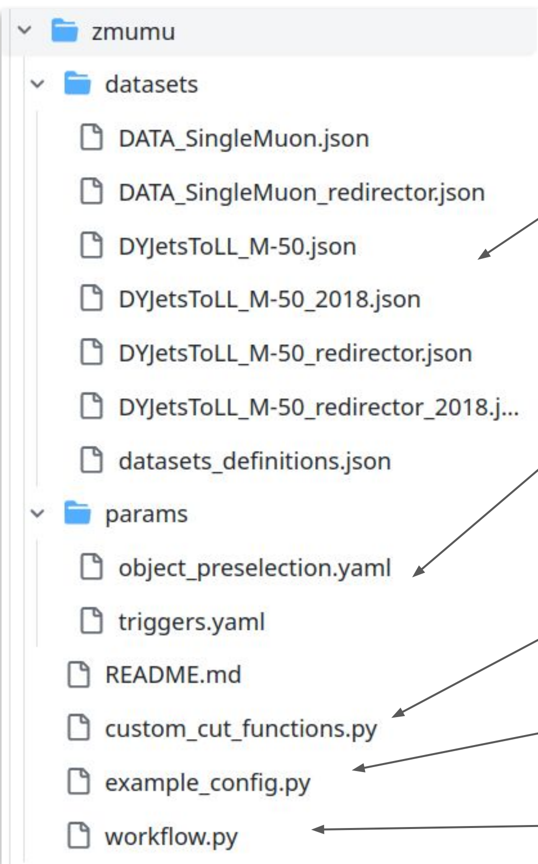
```yaml
object_preselection:
  Muon:
    pt: 15
    eta: 2.4
    iso: 0.25  #PFIsoLoose
    id: tightId

  Electron:
    pt: 15
    eta: 2.4
    iso: 0.06
    id: mvaFall17V2Iso_WP80

  Jet:
    dr_lepton: 0.4
    pt: 30
    eta: 2.4
    jetId: 2
    puId:
      wp: L
      value: 4
      maxpt: 50.0
```

```yaml
"2018":
  SingleEle:
    - Ele32_WPTight_Gsf
    - Ele28_eta2p1_WPTight_Gsf_HT150
  SingleMuon:
    - IsoMu24
```

Analysis example

During the CAT hackathon we prepared a very simple example on $Z \rightarrow \mu\mu$.

Demonstrate how to define an analysis in a independent repository.



datasets definition

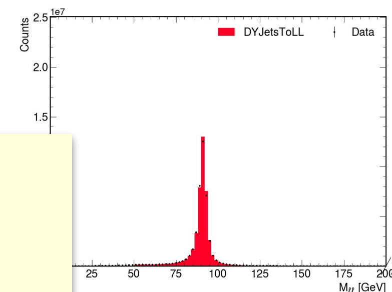
Custom parameters

custom functions

configuration

user processor

Analysis output




```
parameters_dump.yaml
config.json
configurator.pkl
slurm_log
output_all.coffea
```

- Full dump of the parameters
- Human-readable dump of the configuration files in json
- Pickled *Configurator* object for easy rerunning
- output parquet file

Reproducibility

To be able to reproduce the analysis for a given configuration:

- ➔ a pickled version of the configurator object is stored in the output folder
- ➔ the config.pkl can be loaded by the processor to run the analysis with the exact same parameters

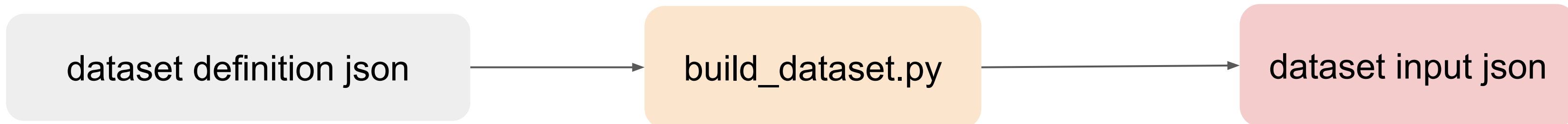
Human readable configuration dump

- → a json version of the analysis config is also saved
- → the file contains all metadata also about used functions
- → one can check all the analysis steps without looking at the code.


```
"finalstate": "semileptonic",
"skim": [
  {
    "name": "nJet_min4_pt15.0",
    "params": {
      "N": 4,
      "coll": "Jet",
      "minpt": 15.0
    },
    "function": {
      "name": "min_nObj_minPt",
      "module": "pocket_coffea.lib.cut_functions",
      "src_file": "/afs/cern.ch/work/d/dvalsecc/private/PocketCoffea/pocke
      "f_hash": 8751285677478
    },
    "id": "nJet_min4_pt15.0__6419684264282140959"
  }
],
"preselections": [
  {
    "name": "semileptonic_nobtag",
    "params": {
      "METbranch": {
        "2016": "MET",
        "2017": "METFixEE2017",
        "2018": "MET"
      },
      "njet": 4,
      "nbjet": 0,
      "pt_leading_electron": {
        "2016": 29,
        "2017": 30,
        "2018": 30
```

Datasets handling

The full list of files for each sample is needed as input of coffea.




```
"ttHTobb": {
    "sample": "ttHTobb",
    "json_output" : "datasets/signal_ttHTobb_lxplus.json",
    "storage_prefix": "/pnfs/psi.ch/cms/trivcat/",
    "files": [
        { "das_names": ["/ttHTobb_M125_TuneCP5_13TeV-powheg-pyt
          "metadata":{
              "year": "2018",
              "isMC": true
          }
        }
    ]
},
```

```
{
    "ttHTobb_2018": {
        "metadata": {
            "das_names": "['/ttHTobb_M125_TuneCP5_13TeV-powheg-pythia8/RunI
            "sample": "ttHTobb",
            "year": "2018",
            "isMC": "True",
            "nevents": "9668000",
            "size": "27889361259"
        },
        "files": [
            "root://xrootd-cms.infn.it///store/mc/RunIISummer20UL18NanoAOD
2500000/6BF93845-49D5-2547-B860-4F7601074715.root",
            "root://xrootd-cms.infn.it///store/mc/RunIISummer20UL18NanoAOD
250000/32D0D1A3-74EB-8146-9F3F-B392AB168FDD.root",
            "root://xrootd-cms.infn.it///store/mc/RunIISummer20UL18NanoAOD
2500000/831D3EA7-36DB-3D41-82D3-5E61A4A9BE04.root",
            "root://xrootd-cms.infn.it///store/mc/RunIISummer20UL18NanoAOD
2510000/AD4D540C-90C0-6A4F-B2AD-39F8CA9143FF.root",
```

handling in a compact way multiple years,
multiple samples

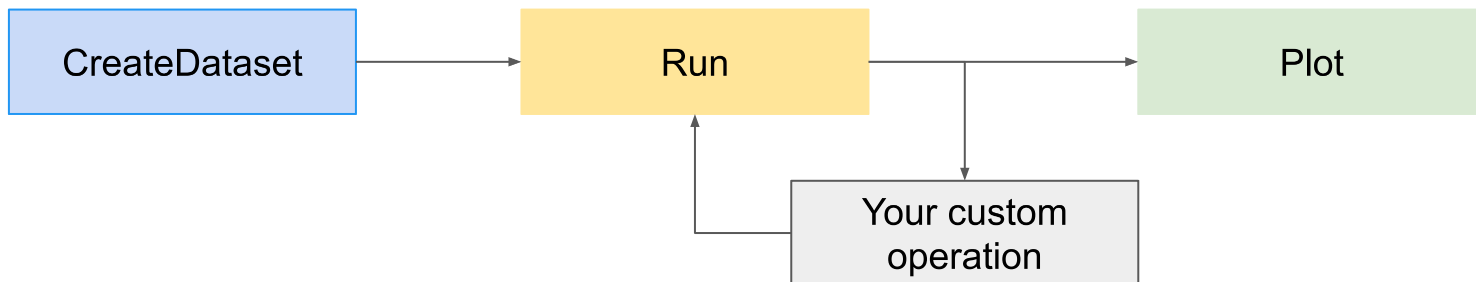
Full docs: https://pocketcoffea.readthedocs.io/en/latest/examples.html#dataset-creation

Workflow orchestration with Law

Law is a workflow management system to define and run complex analyses. PocketCoffea is used to configure and run single analysis step.

Now available: **law tasks for PocketCoffea analysis steps**

- alternative way of running your PocketCoffea configurations, chaining them for automation and clear dependencies
- Example: extract templates → compute SF → rerun templates → plot validations



Thanks to Felix Zinn for introducing this feature in 0.9.5!

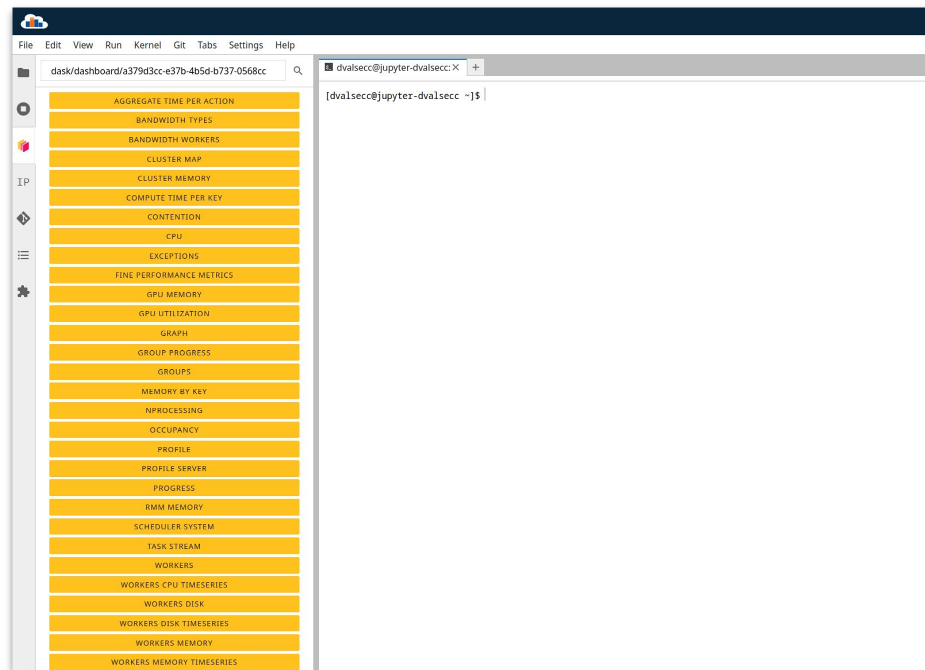
Analysis Facilities integration

Thanks to Dask, PocketCoffea can be scaled in many analysis facilities,

Supported analysis facilities:

- ✅ Purdue
- ✅ INFN AF
- ⚠️ Coffea casa
- ✅ Swan @ CERN

There will be a talk this afternoon with an example on the CMS INFN analysis facility



PocketCoffea @INFN AF

Produced a specific executor:

Takes care of uploading (asynchronously) proxyfile to the nodes and setting the environment properly

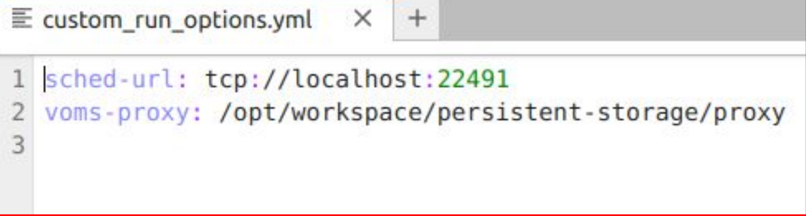
```python
class DaskExecutorFactory(ExecutorFactoryABC):

    def __init__(self, run_options, outputdir, **kwargs):
        self.outputdir = outputdir
        if "sched-url" not in run_options or run_options["sched-url"] is None:
            raise Exception("User need to specify `sched-url` in the custom run options! Please provide the URL of the Dask
        self.sched_url = run_options["sched-url"]
        if "voms-proxy" not in run_options or run_options["voms-proxy"] is None:
            raise Exception("User need to specify `voms-proxy` in the custom run options!")
        self.proxy_path = run_options["voms-proxy"]
        super().__init__(run_options, **kwargs)

    def setup(self):
        ''' Start the DASK cluster here'''
        # At INFN AF, the best way to handle DASK clusters is to create them via the Dask labextension and then connect the
        self.dask_client = Client(address=str(self.sched_url))
        self.dask_client.restart()
        try:
            self.dask_client.register_worker_plugin(UploadFile(self.proxy_path))
        except:
            print("Unable to upload proxyfile, it may be already uploaded")
        # get file name from path
        self.dask_client.register_worker_plugin(SetProxyPlugin(proxy_name=self.proxy_path.split("/")[-1]))

    def customized_args(self):
        args = super().customized_args()
        args["client"] = self.dask_client
        args["treereduction"] = self.run_options["tree-reduction"]
        args["retries"] = self.run_options["retries"]
        return args

    def get(self):
        return coffea_processor.dask_executor(**self.customized_args())

    def close(self):
        self.dask_client.close()
```

PocketCoffea @INFN AF

Produced a specific executor:

Takes care of uploading (asynchronously) proxyfile to the nodes and setting the environment properly

```python
class DaskExecutorFactory(ExecutorFactoryABC):

    def __init__(self, run_options, outputdir, **kwargs):
        self.outputdir = outputdir
        if "sched-url" not in run_options or run_options["sched-url"] is None:
            raise Exception("User need to specify `sched-url` in the custom run options! Please provide the URL of the Dask
        self.sched_url = run_options["sched-url"]
        if "voms-proxy" not in run_options or run_options["voms-proxy"] is None:
            raise Exception("User need to specify `voms-proxy` in the custom run options!")
        self.proxy_path = run_options["voms-proxy"]
        super().__init__(run_options, **kwargs)

    def setup(self):
        ''' Start the DASK cluster here'''
        # At INFN AF, the best way to handle DASK clusters is to create them via the Dask labextension and then connect the
        self.dask_client = Client(address=str(self.sched_url))
        self.dask_client.restart()
        try:
            self.dask_client.register_worker_plugin(UploadFile(self.proxy_path))
        except:
            print("Unable to upload proxyfi
        # get file name from path
        self.dask_client.register_worker_pl

    def customized_args(self):
        args = super().customized_args()
        args["client"] = self.dask_client
        args["treereduction"] = self.run_options["tree-reduction"]
        args["retries"] = self.run_options["retries"]
        return args

    def get(self):
        return coffea_processor.dask_executor(**self.customized_args())

    def close(self):
        self.dask_client.close()
```



PocketCoffea @INFN AF

Produced a specific executor:

Takes care of uploading (asynchronously) proxyfile to the nodes and setting the environment properly

```
class DaskExecutorFactory(ExecutorFactoryABC):

    class SetProxyPlugin(WorkerPlugin):

        def __init__(self, proxy_name='proxy'):
            self.proxy_name = proxy_name

        async def setup(self, worker: Worker):
            import os
            working_dir = worker.local_directory
            os.environ['X509_USER_PROXY'] = working_dir + '/' + self.proxy_name
            os.environ['X509_CERT_DIR']="/cvmfs/grid.cern.ch/etc/grid-security/certificates/"
            try:
                os.chmod(working_dir + '/' + self.proxy_name, 0o400)
            except:
                print("Unable to modify proxyfile permissions, they may be already modified")
        except:
            print("Unable to upload proxyfile, it may be already uploaded")
        # get file name from path
        self.dask_client.register_worker_plugin(SetProxyPlugin(proxy_name=self.proxy_path.split("/")[-1]))

    def customized_args(self):
        args = super().customized_args()
        args["client"] = self.dask_client
        args["treereduction"] = self.run_options["tree-reduction"]
        args["retries"] = self.run_options["retries"]
        return args

    def get(self):
        return coffea_processor.dask_executor(**self.customized_args())

    def close(self):
        self.dask_client.close()
```

f the Dask

onnect the

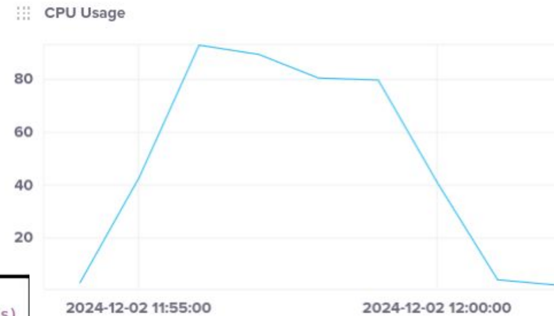
PocketCoffea's Z->μμ

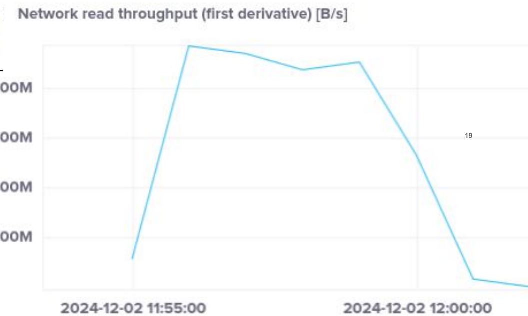
First tests with a Coffea-based analysis:

- 2018 Z->μμ using PocketCoffea configuration layer:
  - object preselections, skim, event selection, computation of basic scale factors with their variations and histogramming
- Dataset:
  - Data + DY: 1.2 TB
    - 1.2 bln events
    - **stored at Legnaro**
    - chunksize 4mln

Category	Events	Throughput (events/s)
Total	1180932962	3181431.42
Skimmed	822423975	2215608.81
Preselected	69283566	186649.81

Benchmark run **on the same 96 CPUs at Legnaro production tier2 site as RDF's benchmark**





With chunksize 4mln, user CPU usage at 80/90% and network read throughput at 400/500 MB/s:

- we are efficiently using the CPU, throughput is smaller wrt RDF's benchmark due to the different workload

PocketCoffea's 2018 TTBar flow

3 different benchmarks (run on **the same 96 CPUs at Legnaro** production tier2 site as RDF's benchmark) of increasing complexity:

Dataset: 2018 UL NanoAOD TTBar ~1TB, 476mln events, **stored at Legnaro**

Chunksize: 1mln

Small: objects and event preselection, scale factors with just 1 variation, no subsamples, 2 categories, 1 variation, 5 histograms

Medium: objects and event preselections, scale factors and their 7 variations, 3 subsamples, 4 categories, 5 histograms

Large: objects and event preselections, scale factors and their 7 variations, 3 subsamples, 14 categories, 5 histograms

CPU usage at 90%, throughput between 250-500 MB/s: the larger the workload, smaller reading throughput

Category	Events	Throughput (events/s)
Total	476408000	3080715.65
Skimmed	181836162	1175852.44
Preselected	63595247	411241.78

Category	Events	Throughput (events/s)
Total	476408000	2515867.48
Skimmed	181836162	960260.29
Preselected	63595247	335840.74

Category	Events	Throughput (events/s)
Total	476408000	1760817.70
Skimmed	181836162	672071.70
Preselected	63595247	235049.87

Demo!

hub: https://cms-it-hub.cloud.cnaf.infn.it/

image: `ghcr.io/comp-dev-cms-ita/jupyterlab:AF20-alma9-v0.0.10-rc9`

code: https://github.com/ttedeschi/workshop2025_demo/tree/main/PocketCoffea