columnflow

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

CLUSTER OF EXCELLENCE
QUANTUM UNIVERSE

SPONSORED BY THE
Federal Ministry
of Education
and Research

columnflow: Fully automated analyses via flow of columns over distributed resources

Marcel Rieger
for the c/f Team

ACAT 2024

15.3.2024

columnflow: Fully automated analysis via flow of columns over distributed resources

Marcel Rieger
on behalf of the cf-Team

General idea

- Python-based framework for nano-like inputs
- End-to-end **orchestration** & **automation**
- **No reliance** on single local cluster or local storage
- Adapt to any remote cluster and storage system
  - ▻ HTCondor, Slurm, CMS-CRAB, LSF
  - ▻ Store via `file://`, `xrootd://`, `gsiftp://`, `webdav://`
- **Persistent intermediate outputs**
  - ▻ Debugging, reuse, sharing across groups

Key concepts

- Experiment **agnostic core**
  - ▻ Organize experiment-specific recipes in extensions
- Use awkward arrays as interface, parquet as file format
  - ▻ Give **users full control** over processing tools (NumPy, TensorFlow, coffea-nano-format, pandas, ...)
- High degree of **code-reuse** and collaboration
- Define **workflows** with luigi + law, metadata with order
- Control and execution via **CLI**, **scripts** and **notebooks**

Automation stack

luigi → law (luigi analysis workflow) → columnflow → analysis code

workflow engine (originally by Spotify) | layer for HEP & scale-out (experiment independent) | framework (experiment independent) | analysis code

Parallelization over ...

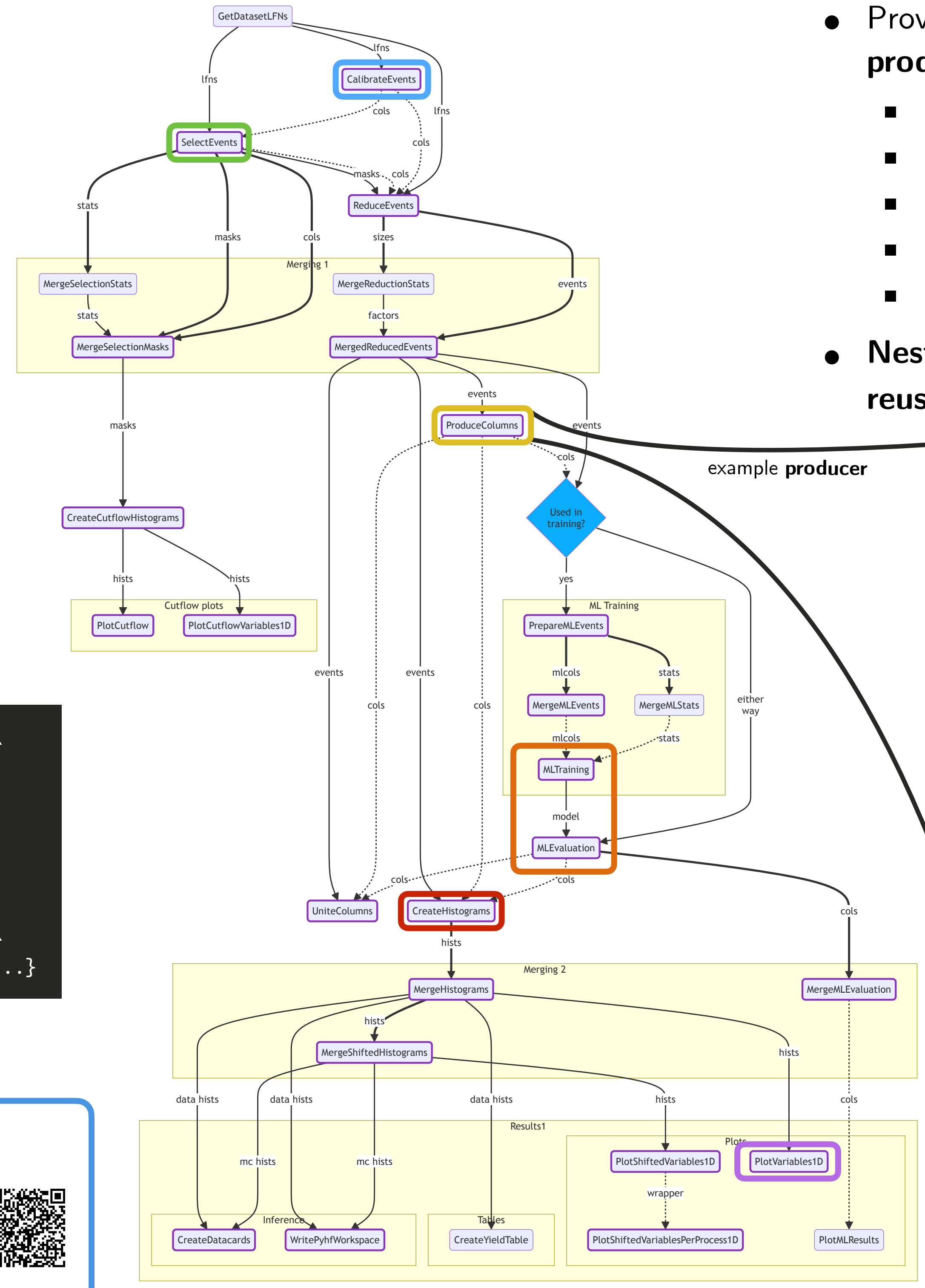
- Campaigns & datasets
- Files
- Systematics

▻ Typically $\mathcal{O}$(10k) 60min jobs, **however**, on **standard resources**

▻ HTCondor, CRAB, ...

Example graph*

(* Just a suggestion, can be easily altered or amended by analyses)



example **producer**

```python
@producer(
    uses={
        "nMuon", "Muon.pt", "Muon.eta",
    },
    produces={
        "muon_weight", "muon_weight_up", "muon_weight_down",
    },
    # only allowed on mc
    mc_only=True,
)
def muon_weights(
    self: Producer,
    events: ak.Array,
    muon_mask: ak.Array | type(Ellipsis) = Ellipsis,
    **kwargs,
) -> ak.Array:
    """ ... Creates muon weights using the correctionlib. ... """

    # flat absolute eta and pt views
    abs_eta = flat_np_view(abs(events.Muon.eta[muon_mask]), axis=1)
    pt = flat_np_view(events.Muon.pt[muon_mask], axis=1)

    # loop over systematics
    for syst, postfix in [
        ("sf", ""),
        ("systup", "_up"),
        ("systdown", "_down"),
    ]:
        sf_flat = self.muon_sf_corrector(self.year, abs_eta, pt, syst)

        # add the correct layout to it
        sf = layout_ak_array(sf_flat, events.Muon.pt[muon_mask])

        # create the product over all muons per event
        weight = ak.prod(sf, axis=1, mask_identity=False)

        # store it
        events = set_ak_column(events, f"muon_weight{postfix}", weight,

    return events
```

Simple customization

- Provide simple functions, **producers**, to create
  - **calibrated** (*updated*) **columns**
  - **selection masks**
  - **new columns**
  - **ML training & evaluation**
  - **variables**
- **Nesting** enables for easy **reuse** and **capsulation**
- Using bare **awkward arrays**
- Implementation and **choice of tools** fully **up to user**

Graph execution

- **Single command** can trigger the full pipeline from **inputs to plots**
- **Example**

```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets ttbar,dy \
    --calibrators jec,jer \
    --selector full \
    --producers muon_weights \
    --variables jet*_{eta,pt} \
    --workflow {crab,htcondor,...}
```

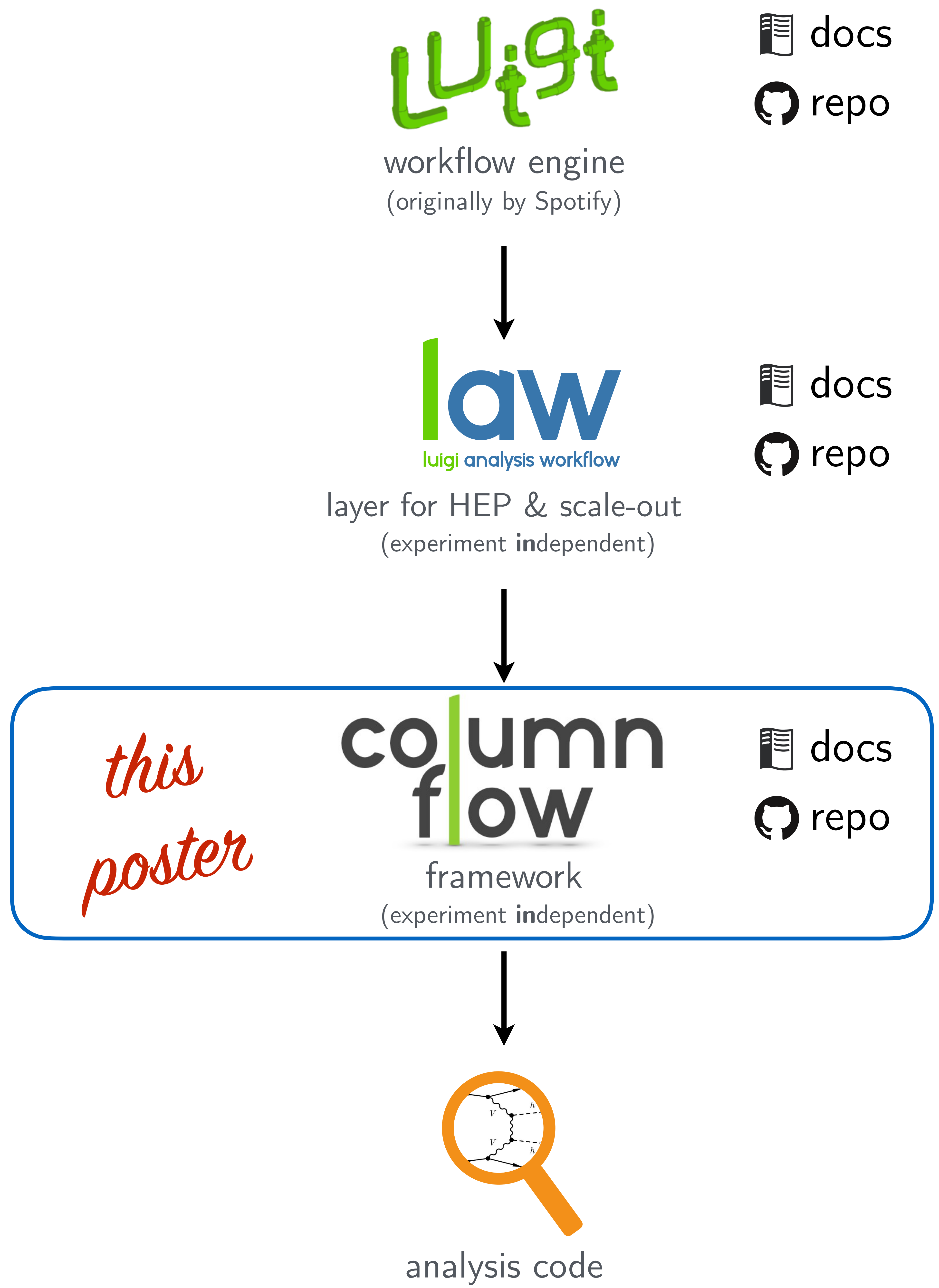
Documentation

github.com/columnflow
columnflow.readthedocs.io

General idea

- Python-based framework for nano-like inputs
- End-to-end **orchestration** & **automation**
  - ▻ From events to plots in a single command
- **No reliance** on single local cluster or local storage
- Adapt to any remote cluster and storage system
  - ▻ HTCondor, Slurm, CMS-CRAB, LSF
  - ▻ Store via `file://`, `xrootd://`, `gsiftp://`, `webdav://`
- **Persistent intermediate outputs**
  - ▻ Debugging, reuse, sharing across groups

Automation stack



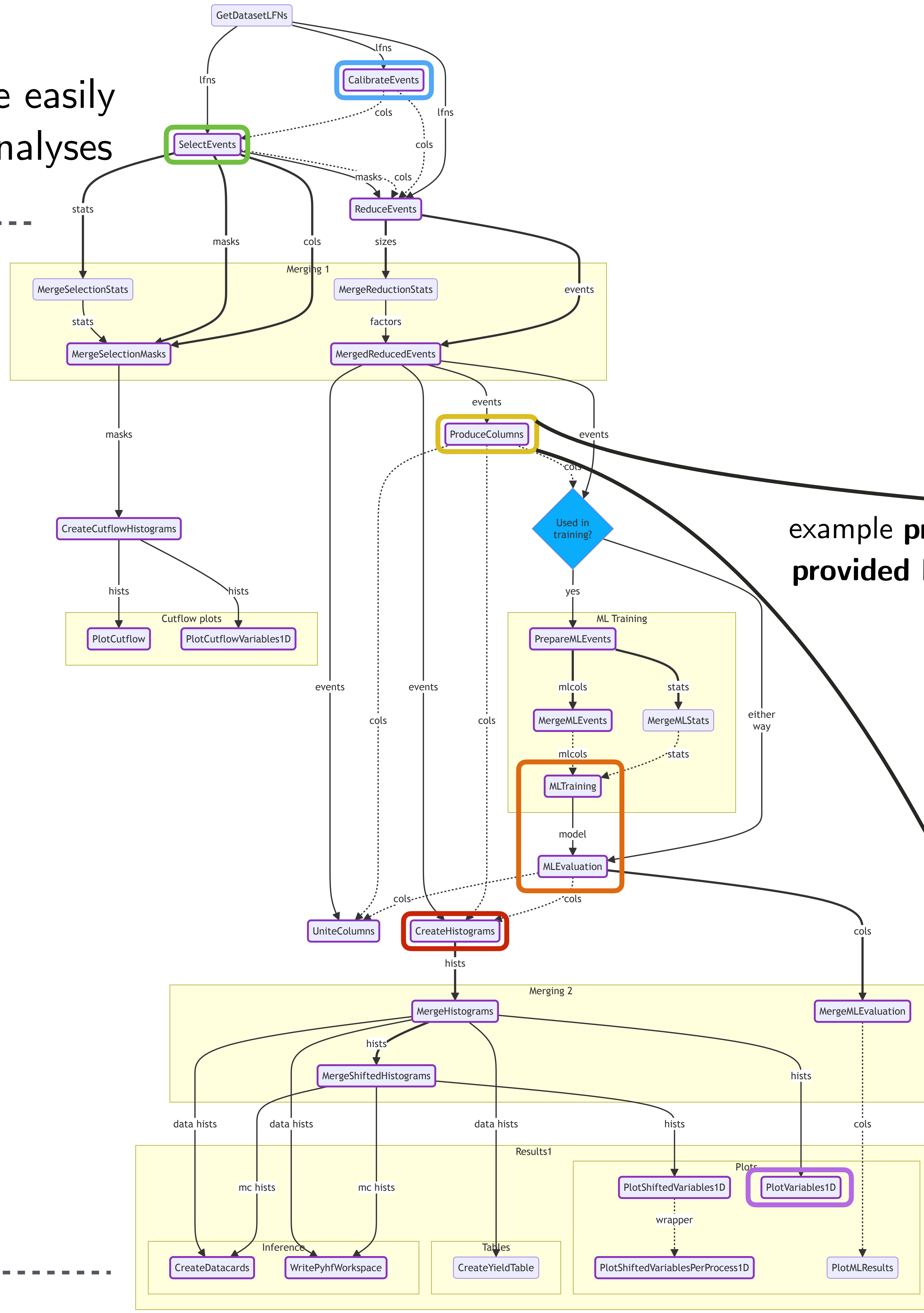
Key concepts

- Experiment-**agnostic core**
- Use awkward arrays as interface, parquet as file format
  - ▻ Give **users full control** over tools used (NumPy, TensorFlow, coffea-nano-format, pandas, ...)
- Define **workflows** with luigi + law, **metadata** with order
- **Capsulation of standard recipes**
  - ▻ High degree of **code-reuse** & collaboration

Example graph

Just a suggestion, can be easily altered or amended by analyses

Nano inputs



Plots & results

Simple customization

- Provide simple functions, **producers**, to create
  - calibrated (*updated*) columns
  - selection masks
  - new columns
  - ML training & evaluation
  - variables
- **Nesting** enables for easy **reuse** and **capsulation**

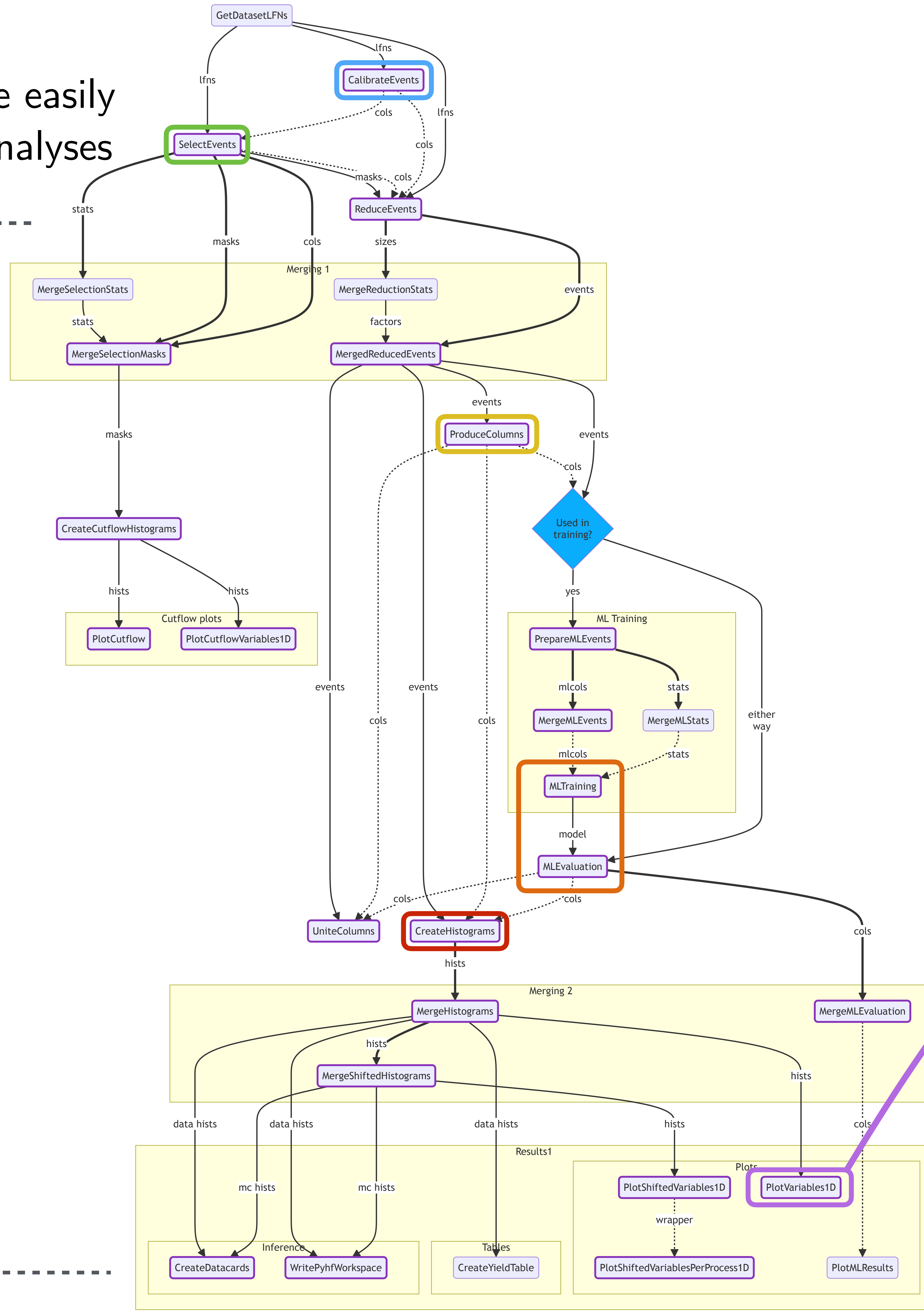
example **producer provided by user**

```python
@producer(
    uses={
        "nMuon", "Muon.pt", "Muon.eta",
    },
    produces={
        "muon_weight", "muon_weight_up", "muon_weight_down",
    },
    # only allowed on mc
    mc_only=True,
)
def muon_weights(
    self: Producer,
    events: ak.Array,
    muon_mask: ak.Array | type(Ellipsis) = Ellipsis,
    **kwargs,
) -> ak.Array:
    """ Creates muon weights using the correctionlib. """

    # flat absolute eta and pt views
    abs_eta = flat_np_view(abs(events.Muon.eta[muon_mask]), axis=1)
    pt = flat_np_view(events.Muon.pt[muon_mask], axis=1)

    # loop over systematics
    for syst, postfix in [
        ("sf", ""),
        ("systup", "_up"),
        ("systdown", "_down"),
    ]:
        sf_flat = self.muon_sf_corrector(self.year, abs_eta, pt, syst)

        # add the correct layout to it
        sf = layout_ak_array(sf_flat, events.Muon.pt[muon_mask])

        # create the product over all muons per event
        weight = ak.prod(sf, axis=1, mask_identity=False)

        # store it
        events = set_ak_column(events, f"muon_weight{postfix}", weight)

    return events
```

Example graph

Just a suggestion, can be easily altered or amended by analyses

Nano inputs



Plots & results

Simple customization

- Provide simple functions, **producers**, to create
  - **calibrated** (*updated*) **columns**
  - **selection masks**
  - **new columns**
  - **ML training & evaluation**
  - **variables**
- **Nesting** enables for easy **reuse** and **capsulation**

Graph execution

- **Single command** can trigger the full pipeline from **inputs** to **plots**, or any intermediate task
- **Example**

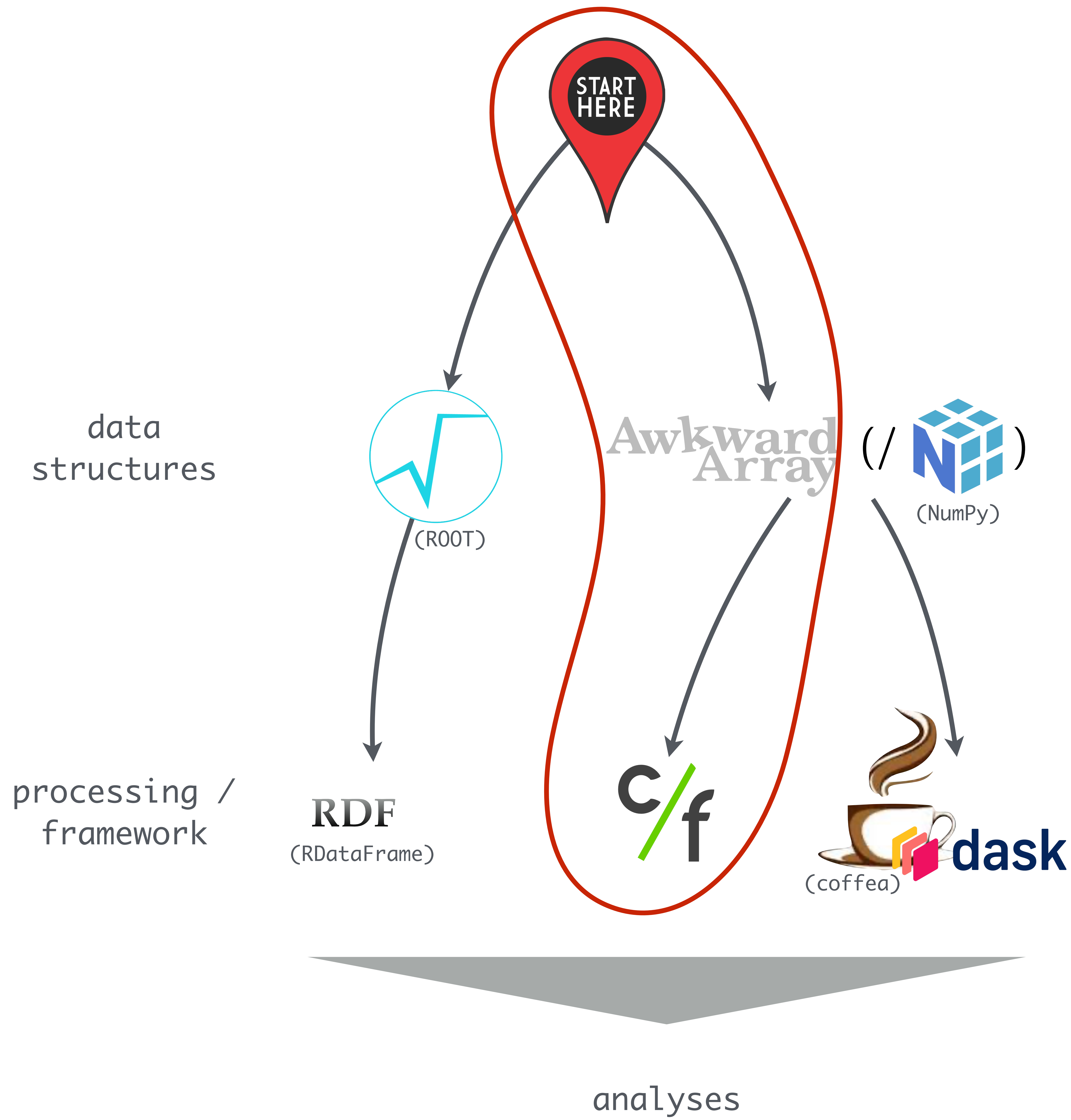
```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets ttbar,dy \
    --calibrators jec,jer \
    --selector full \
    --producers muon_weights \
    --variables jet*_{eta,pt} \
    --workflow {crab,htcondor,...}
```

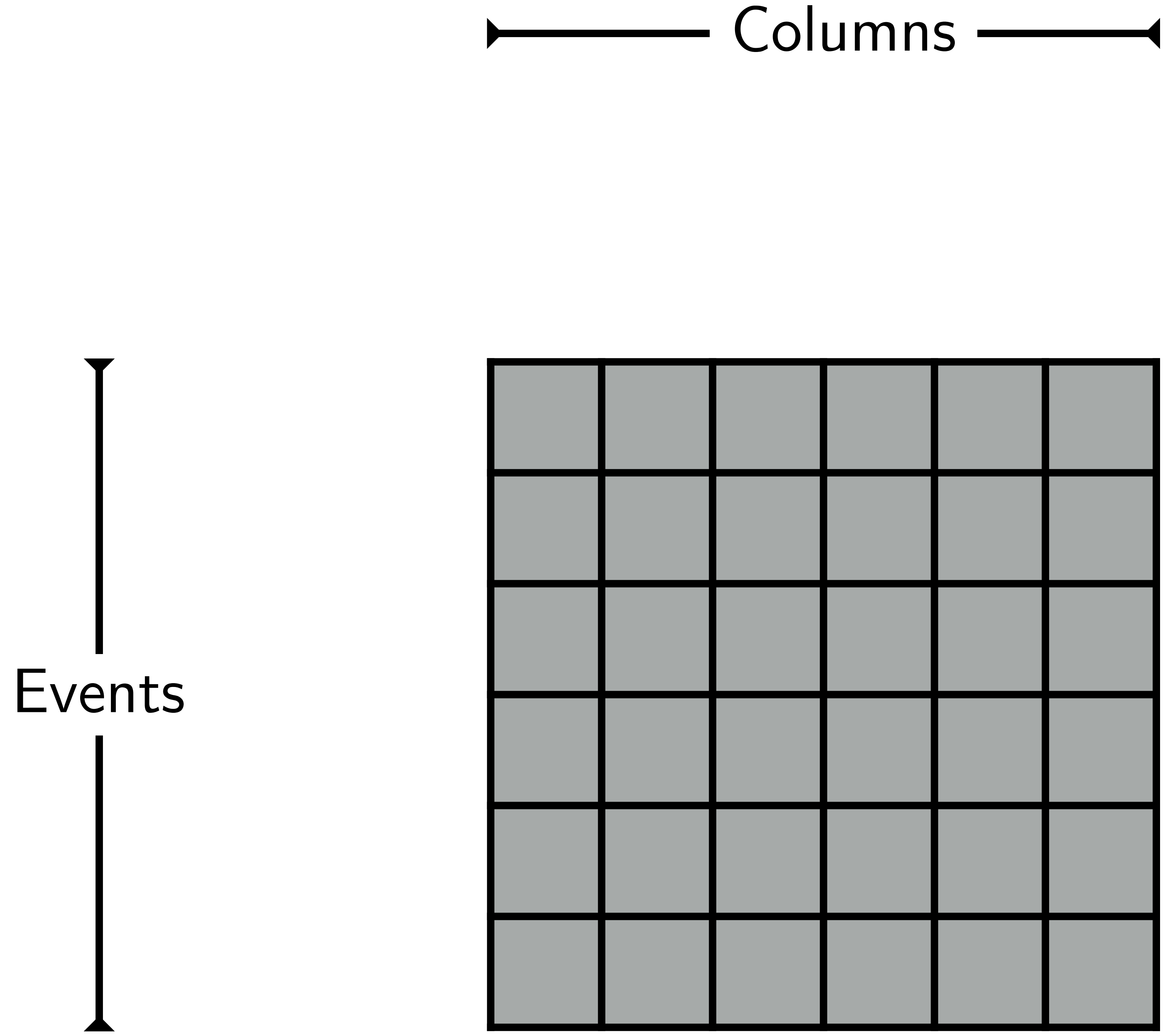
Backup

columnflow in depth

- **Python framework for vectorized, columnar HEP analysis with flat (nano-like) inputs**
  - Mostly experiment agnostic **core**, plenty of CMS-related **specializations** on top
  - Using awkward arrays + coffea nano-scheme, parquet as file format
  - Workflows with luigi/law, metadata definition using order

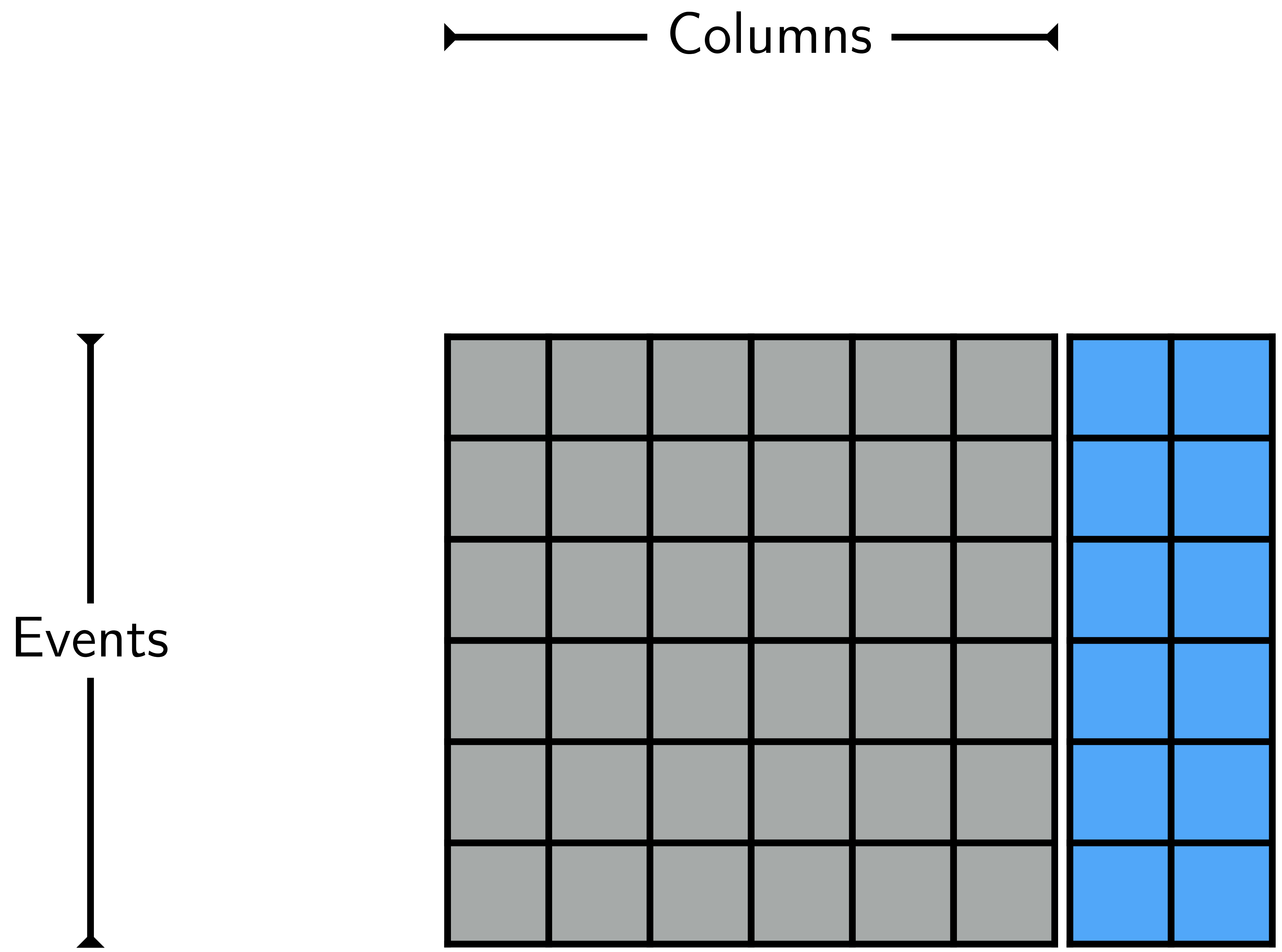
- **Our initial wishlist**
  - End-to-end **orchestration & automation**
    - ▻ One command can trigger the entire workflow
  - Highly parallel execution on **any remote batch system**
    - ▻ HTCondor, Slurm, LSF, WLCG, CMS-CRAB, ...
  - Seamless integration of **any remote storage system**
    - ▻ *Storage:* `file://`, `xrootd://`, `gsiftp://`, `webdav://`, ...
  - No reliance on custom, local hardware
    - ▻ We need to be able to invite external collaborators
    - ▻ Reduction in speed (!) to be compensated with high parallelism
  - **Persistent** intermediate outputs
    - ▻ Easy reuse across groups, ML applications, working with students ...


START HERE
data structures
(ROOT)
Awkward Array
(/
)
(NumPy)
processing / framework
RDF
(RDataFrame)
(coffea)
dask
analyses



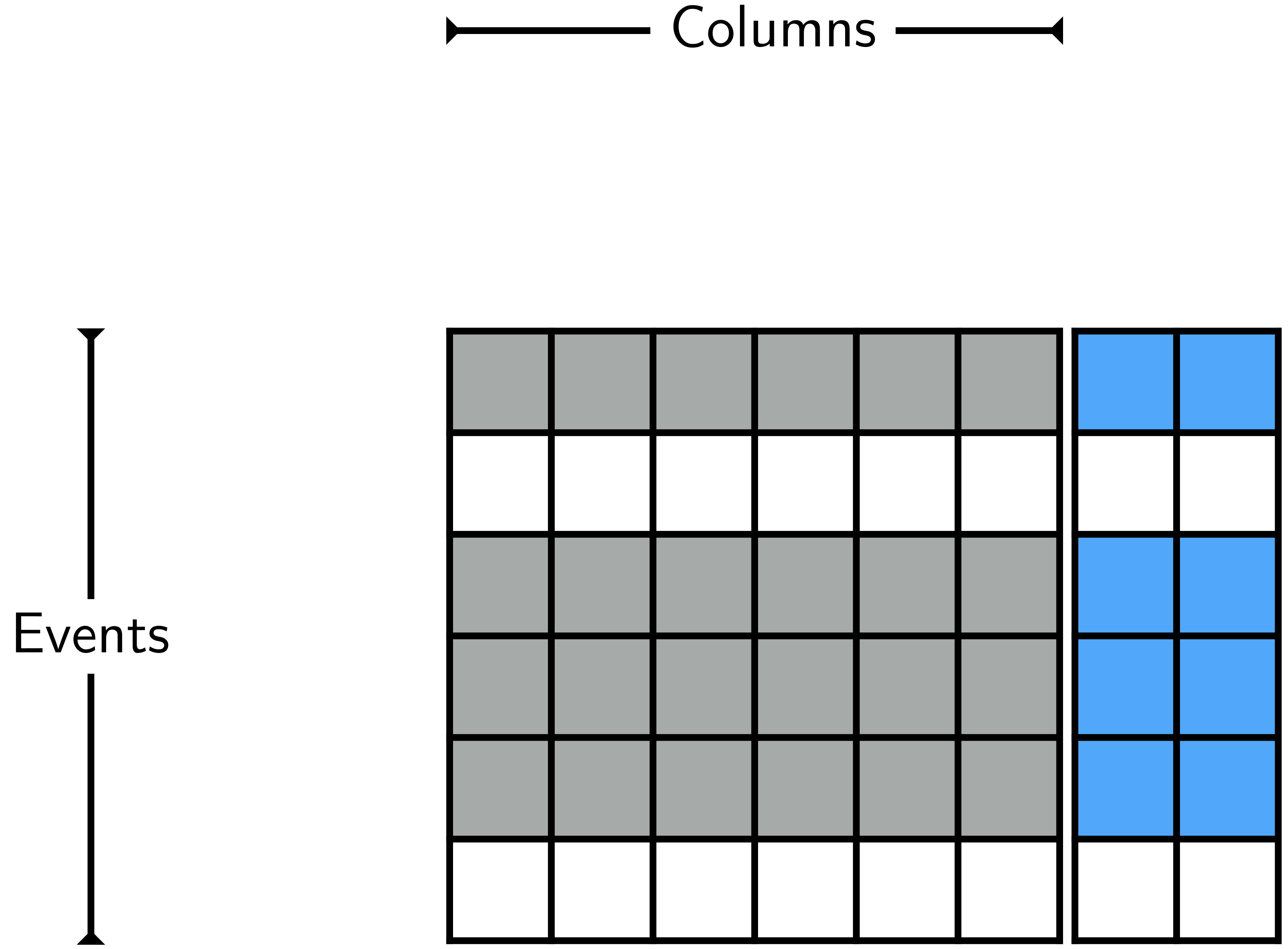
Operations

- Extension
- Selection (*creating* masks)
- Reduction (*applying* masks)
- Extension
- Merge



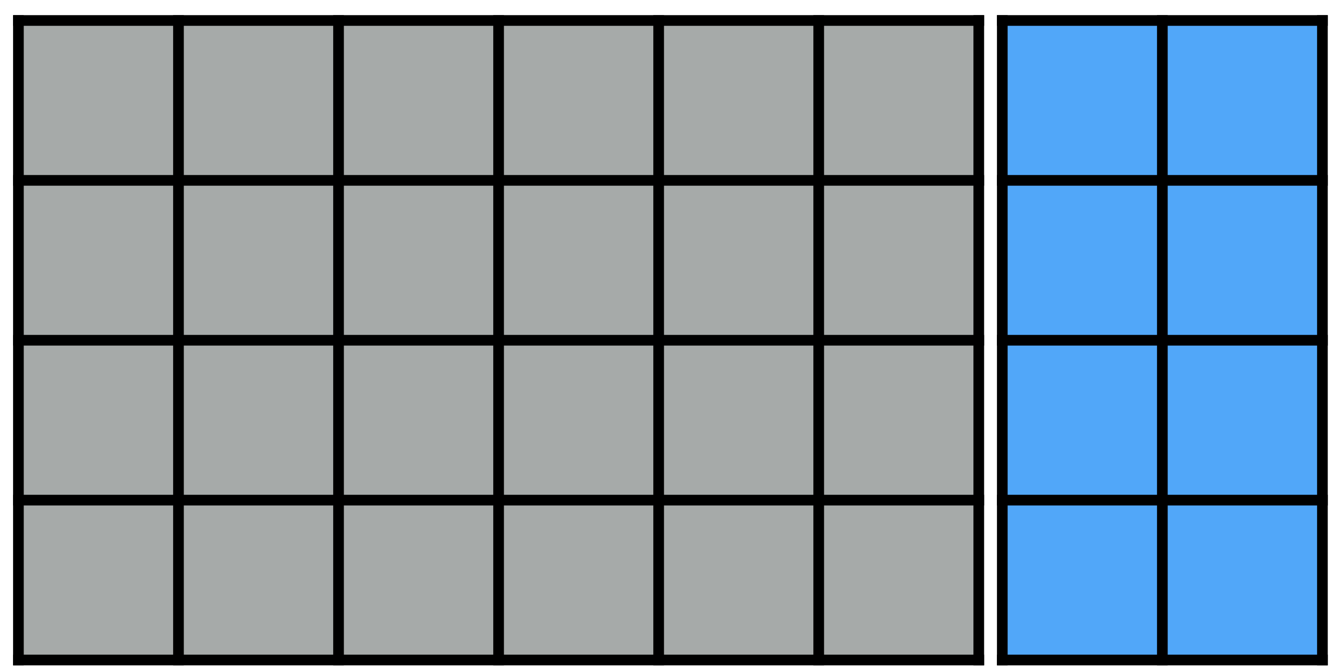
Operations

- ☑ Extension
- ☐ Selection (*creating* masks)
- ☐ Reduction (*applying* masks)
- ☐ Extension
- ☐ Merge



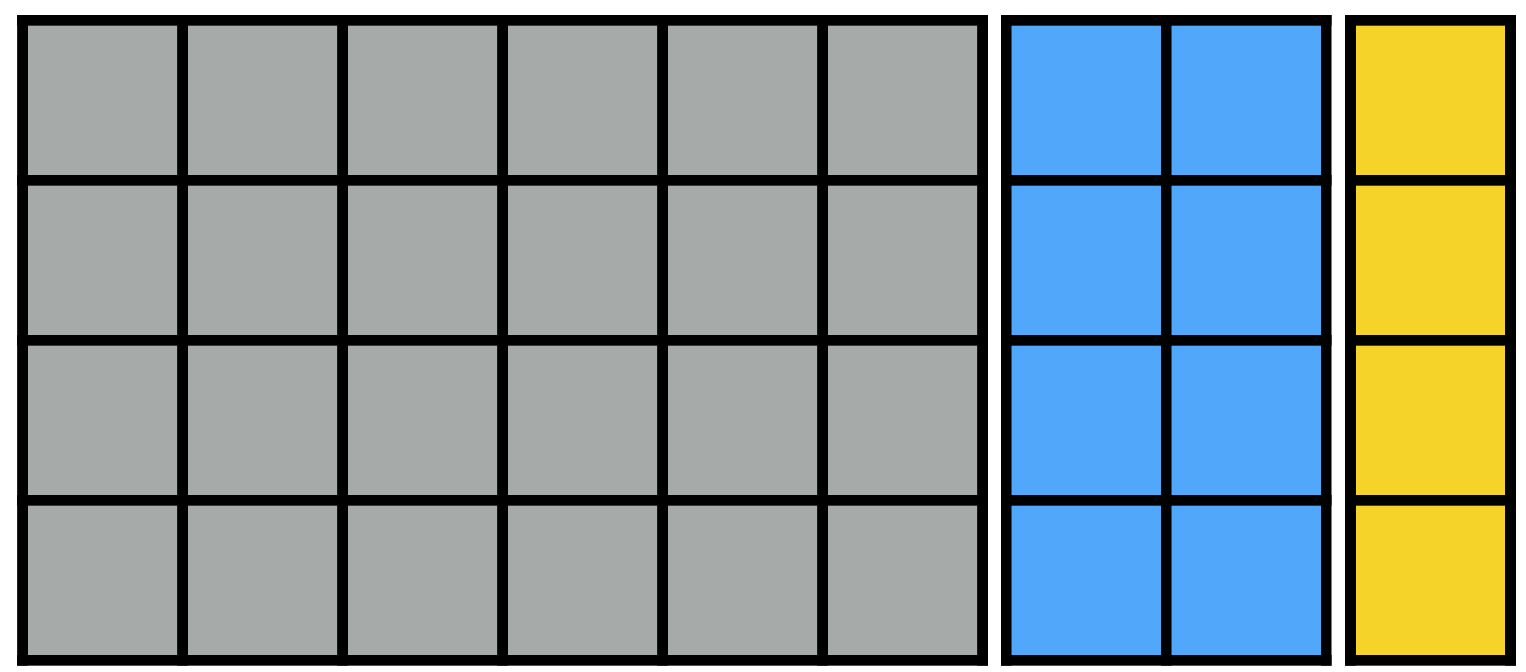
Operations

- [x] Extension
- [x] Selection (*creating* masks)
- [ ] Reduction (*applying* masks)
- [ ] Extension
- [ ] Merge



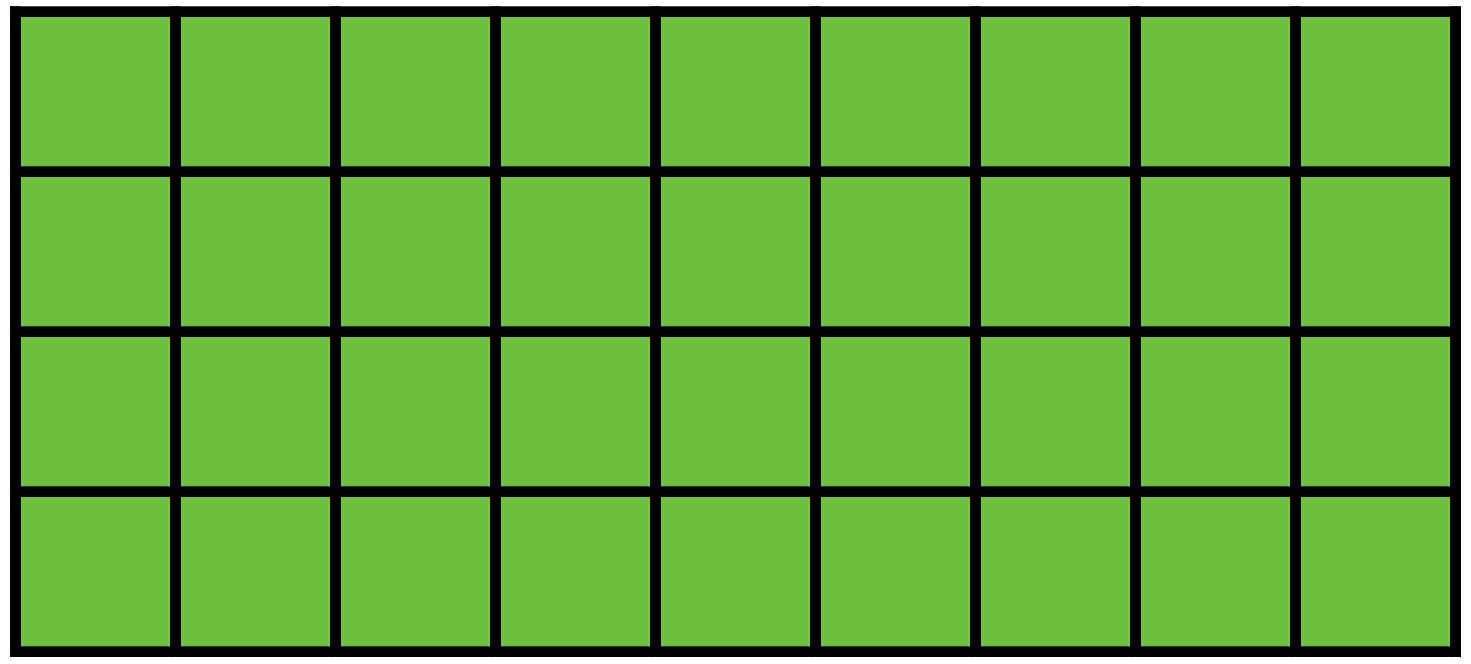
Operations

- [x] Extension
- [x] Selection (*creating* masks)
- [x] Reduction (*applying* masks)
- [ ] Extension
- [ ] Merge



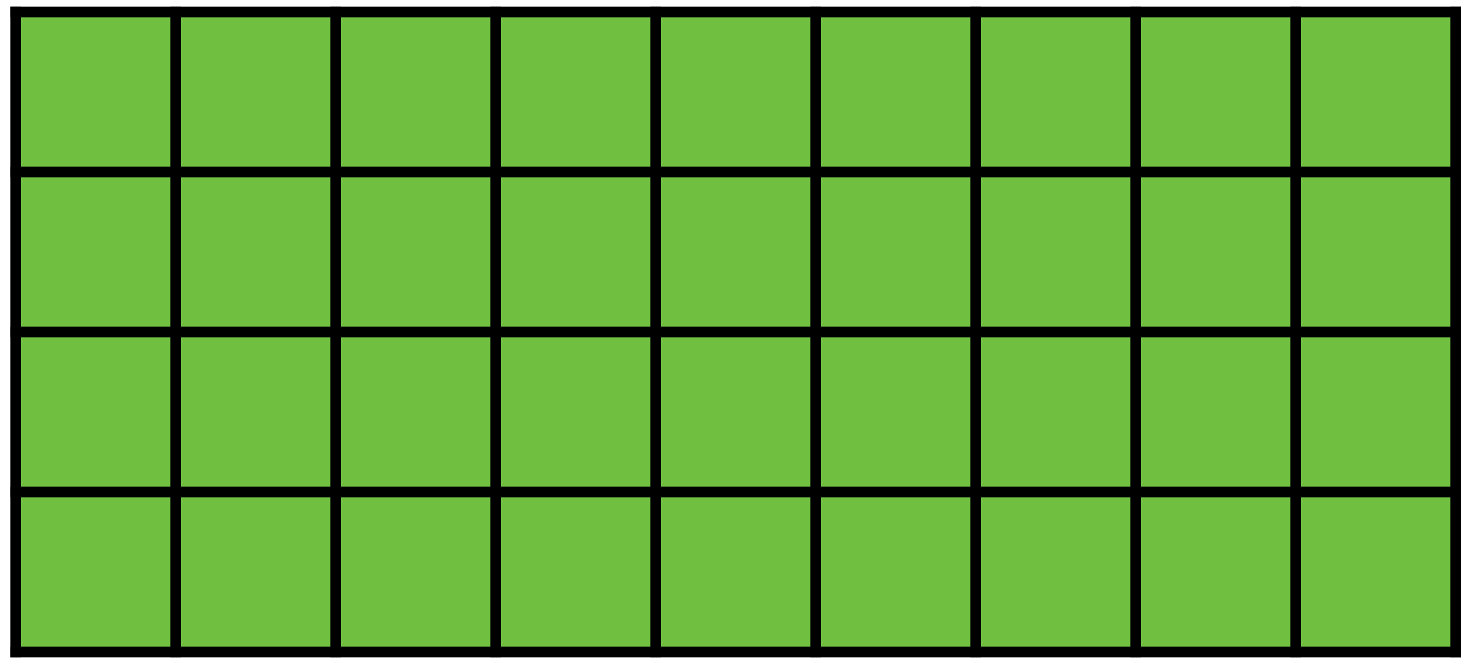
Operations

- ☑ Extension
- ☑ Selection (*creating* masks)
- ☑ Reduction (*applying* masks)
- ☑ Extension
- ☐ Merge



Operations

- ☑ Extension
- ☑ Selection (*creating* masks)
- ☑ Reduction (*applying* masks)
- ☑ Extension
- ☑ Merge

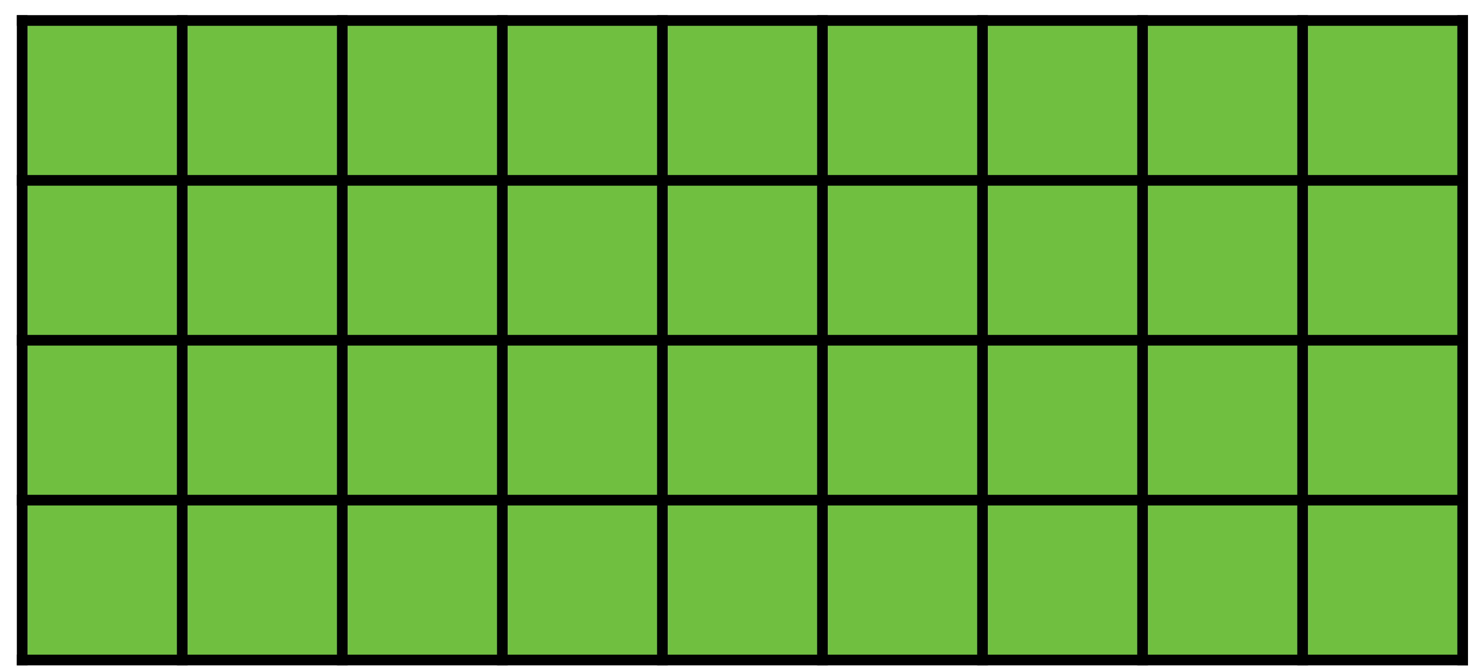


- **In-memory**
  - Trivial
  - NumPy / awkward array provide all necessary tools and helpers

- **Across a large scale analysis with persistent intermediate files**

Operations

- ☑ Extension
- ☑ Selection (*creating* masks)
- ☑ Reduction (*applying* masks)
- ☑ Extension
- ☑ Merge



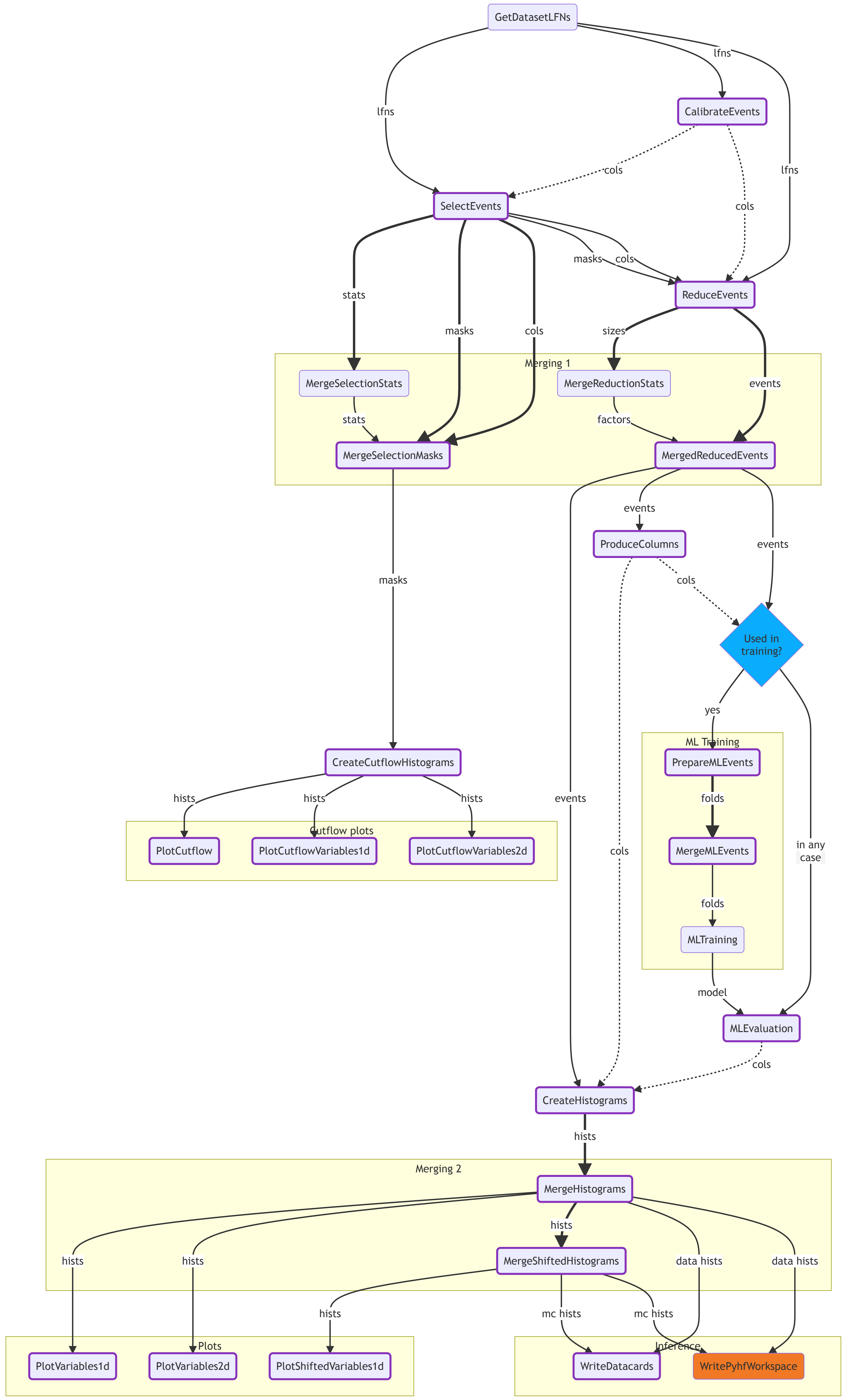
Operations

- ✓ Extension
- ✓ Selection (*creating* masks)
- ✓ Reduction (*applying* masks)
- ✓ Extension
- ✓ Merge

- **In-memory**
  - Trivial
  - NumPy / awkward array provide all necessary tools and helpers

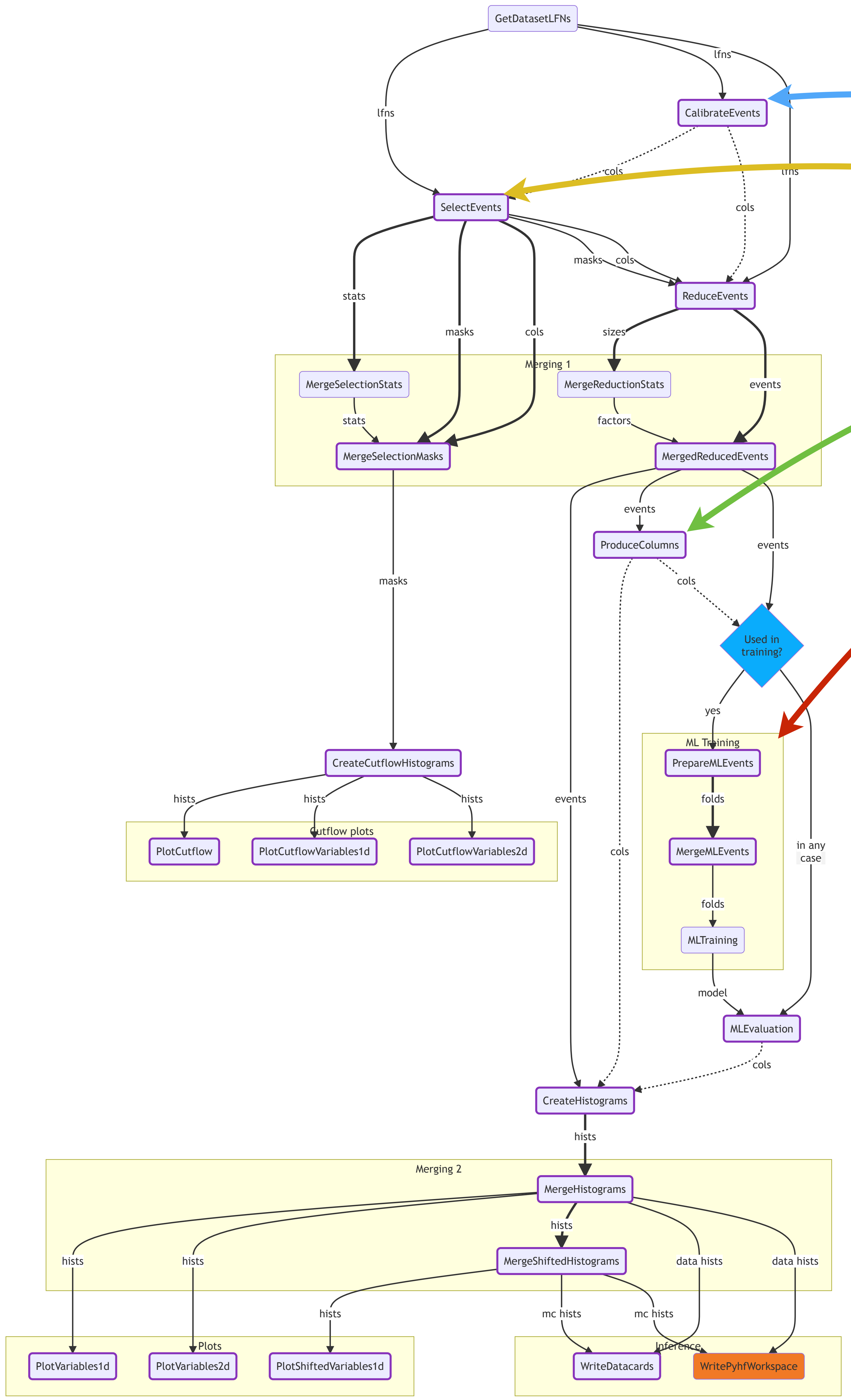
- **Across a large scale analysis with persistent intermediate files**
  - represent input files
    - Typically $\mathcal{O}(1k - 10k)$
    - High parallelism, only **single-core** requirement
    - Chunked reading with IO offloading to threads
  - and represent columns, potentially stored in **additional files** and **same event order**
    - Flexible decisions by analyses whether to store columns and when to load them
    - Can be written & read in multi-threaded IO
    - Only write merged when necessary

- 1️⃣ **Fully orchestrated workflow**
  - Only a *suggestion*, but able to model majority of analyses
  - Can be altered or created from scratch by analyses



live task graph

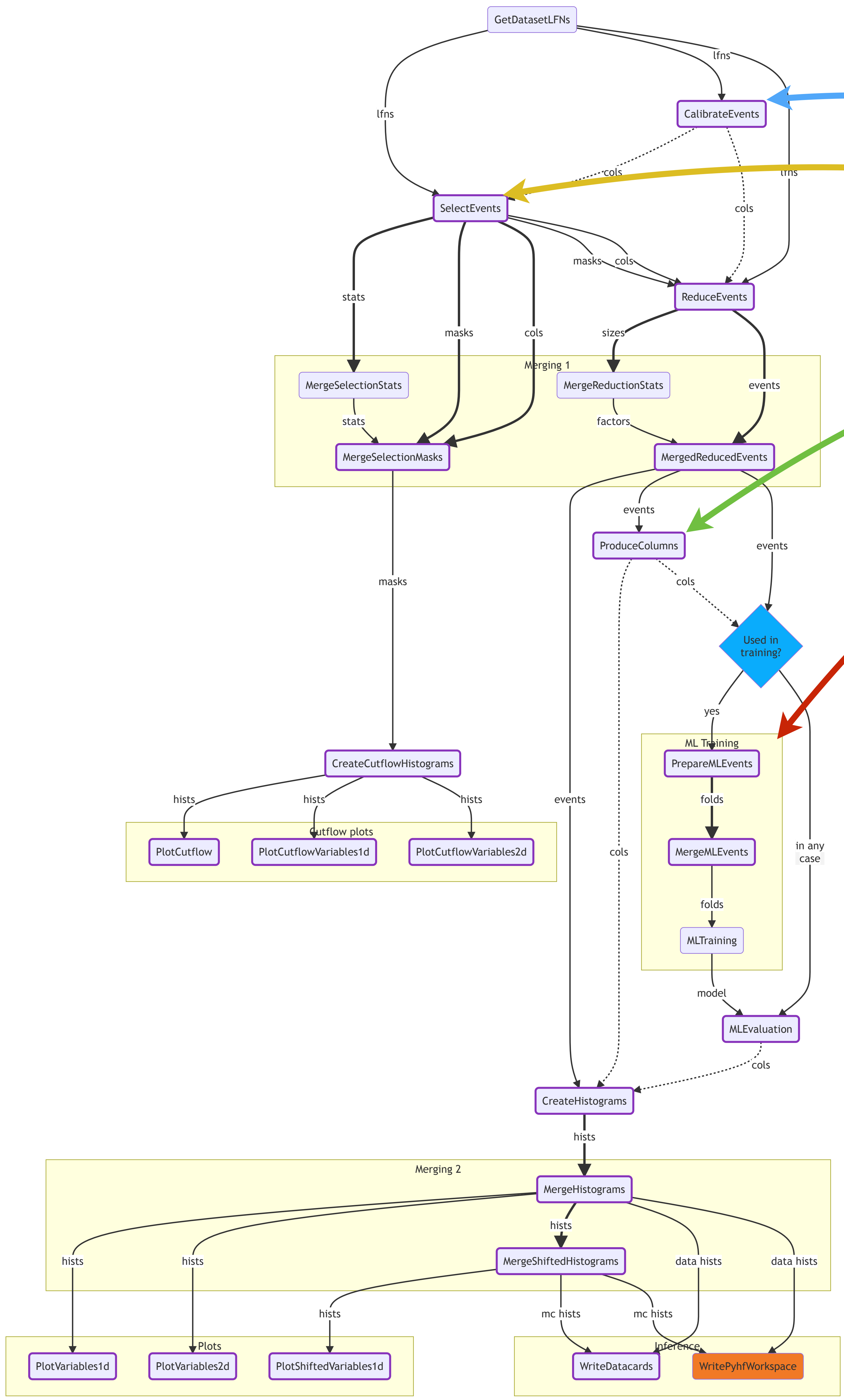
- **1 Fully orchestrated workflow**
  - Only a *suggestion*, but able to model majority of analyses
  - Can be altered or created from scratch by analyses



live task graph

- **2 Tools for on-demand column retrieval / production**
  - Configurable functions creating new columns at certain points of the workflow
  - Can be selected at execution time, e.g. `btag_weight`, `pu_weight`
  - Carry information on **used** and **produced** columns, → open & save only necessary columns (see backup)

- 1️⃣ **Fully orchestrated workflow**
  - Only a *suggestion*, but able to model majority of analyses
  - Can be altered or created from scratch by analyses



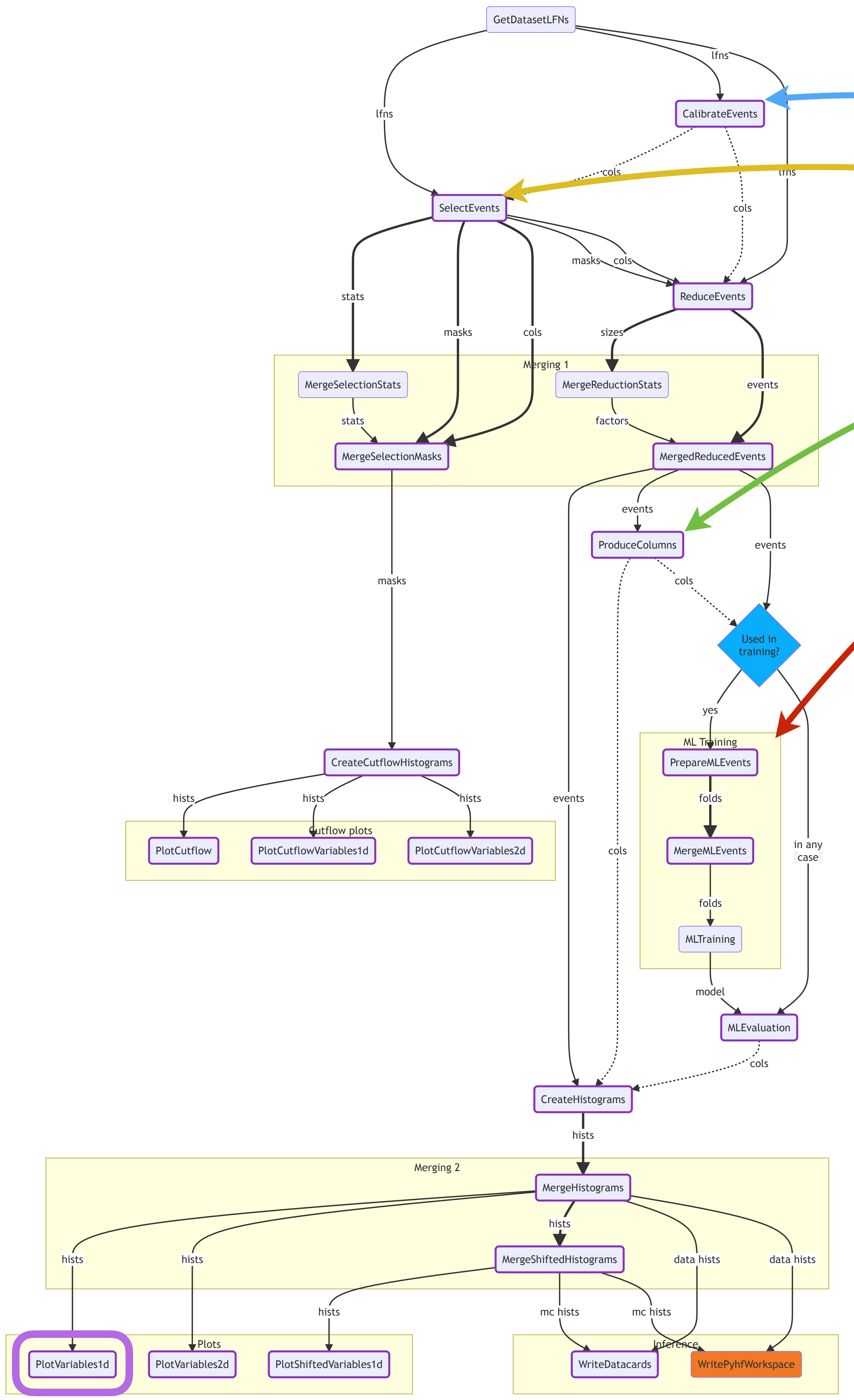
live task graph

- 2️⃣ **Tools for on-demand column retrieval / production**
  - Configurable functions creating new columns at certain points of the workflow
  - Can be selected at execution time, e.g. `btag_weight`, `pu_weight`
  - Carry information on **used** and **produced** columns, → open & save only necessary columns (see backup)

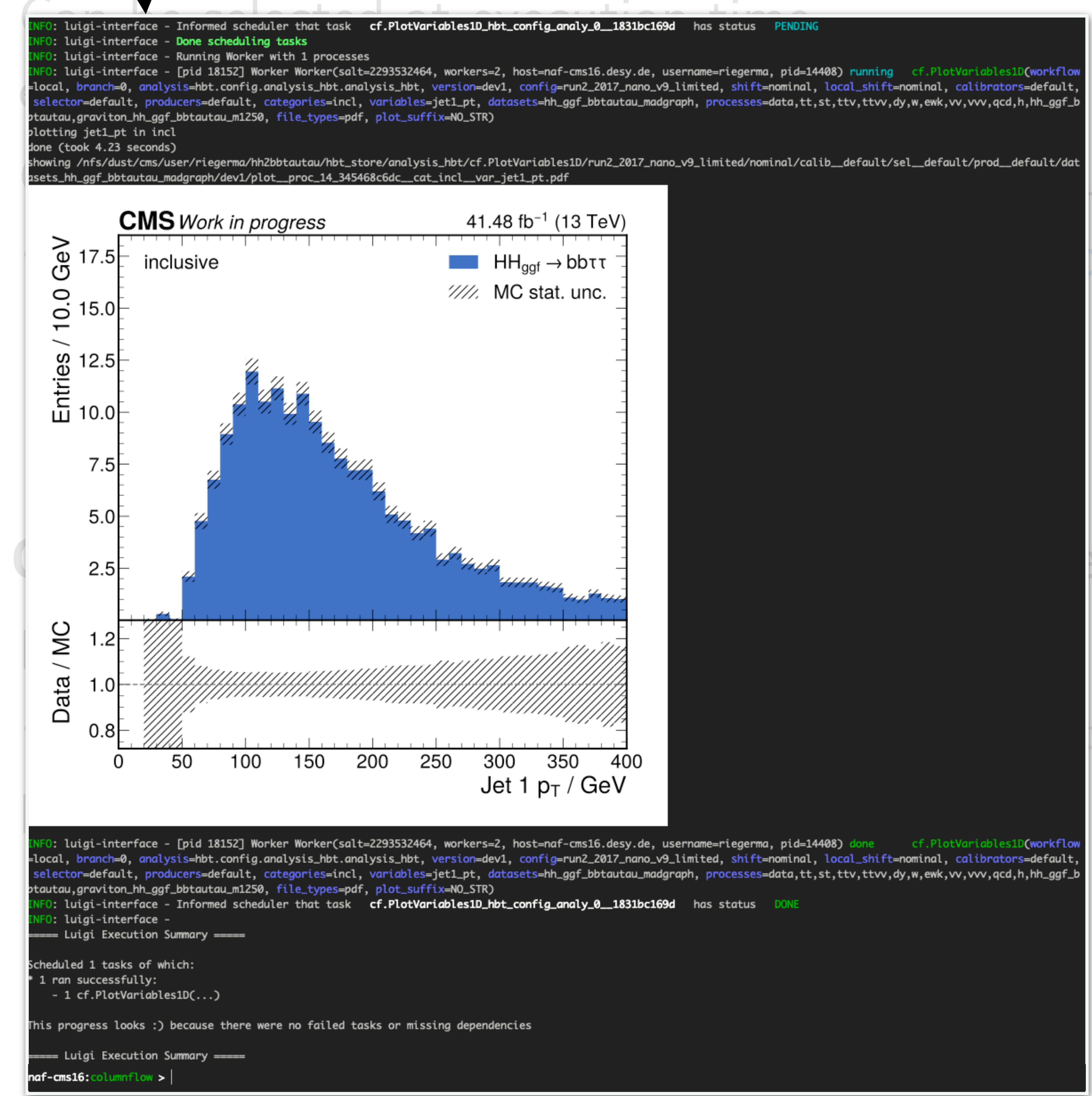
- 3️⃣ **Collection of standardized column producers (CMS)**
  - Mostly SF and weight production using `correctionlib` → `jec`, `jer`, `tec`, `e_sf`, `mu_sf`, `trigger_sf`, `btag_sf`, ...
  - Plug-in mechanism for analyses

- 1 **Fully orchestrated workflow**
  - Only a *suggestion*, but able to model majority of analyses
  - Can be altered or created from scratch by analyses

```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets hh_bbtautau \
    --calibrators jec \
    --selector full \
    --producers all_weights \
    --variables jet1_pt
```



live task graph



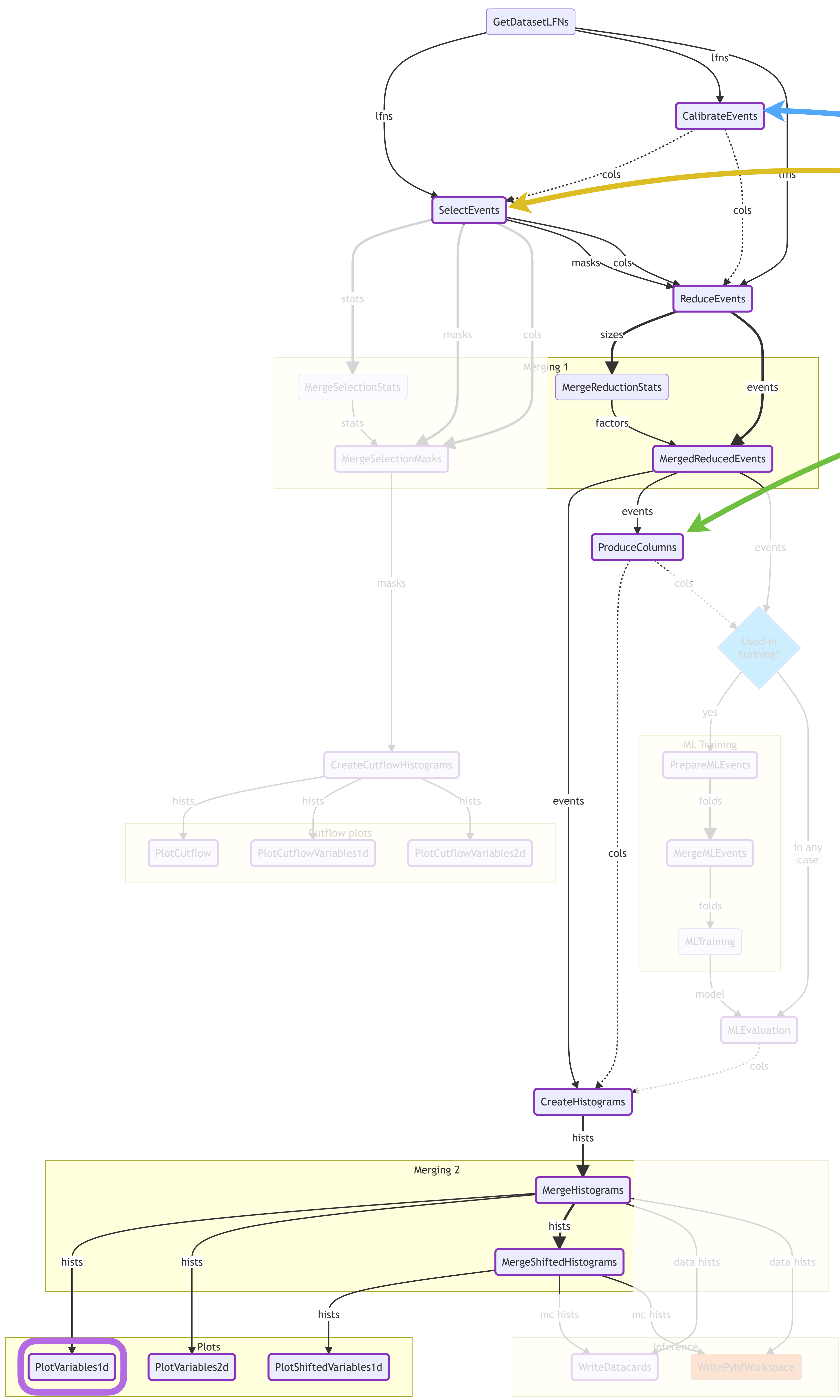
Single producer

```python
@producer(
    uses={
        "nMuon", "Muon.pt", "Muon.eta",
    },
    produces={
        "muon_weight", "muon_weight_up", "muon_weight_down",
    },
    # only allowed on mc
    mc_only=True,
)
def muon_weights(
    self: Producer,
    events: ak.Array,
    muon_mask: ak.Array | type(Ellipsis) = Ellipsis,
    **kwargs,
) -> ak.Array:
    """ ... Creates muon weights using the correctionlib. ... """

    # flat absolute eta and pt views
    abs_eta = flat_np_view(abs(events.Muon.eta[muon_mask]), axis=1)
    pt = flat_np_view(events.Muon.pt[muon_mask], axis=1)

    # loop over systematics
    for syst, postfix in [
        ("sf", ""),
        ("systup", "_up"),
        ("systdown", "_down"),
    ]:
        sf_flat = self.muon_sf_corrector(self.year, abs_eta, pt, syst)

        # add the correct layout to it
        sf = layout_ak_array(sf_flat, events.Muon.pt[muon_mask])

        # create the product over all muons per event
        weight = ak.prod(sf, axis=1, mask_identity=False)

        # store it
        events = set_ak_column(events, f"muon_weight{postfix}", weight, value_type=np.float32)

    return events
```

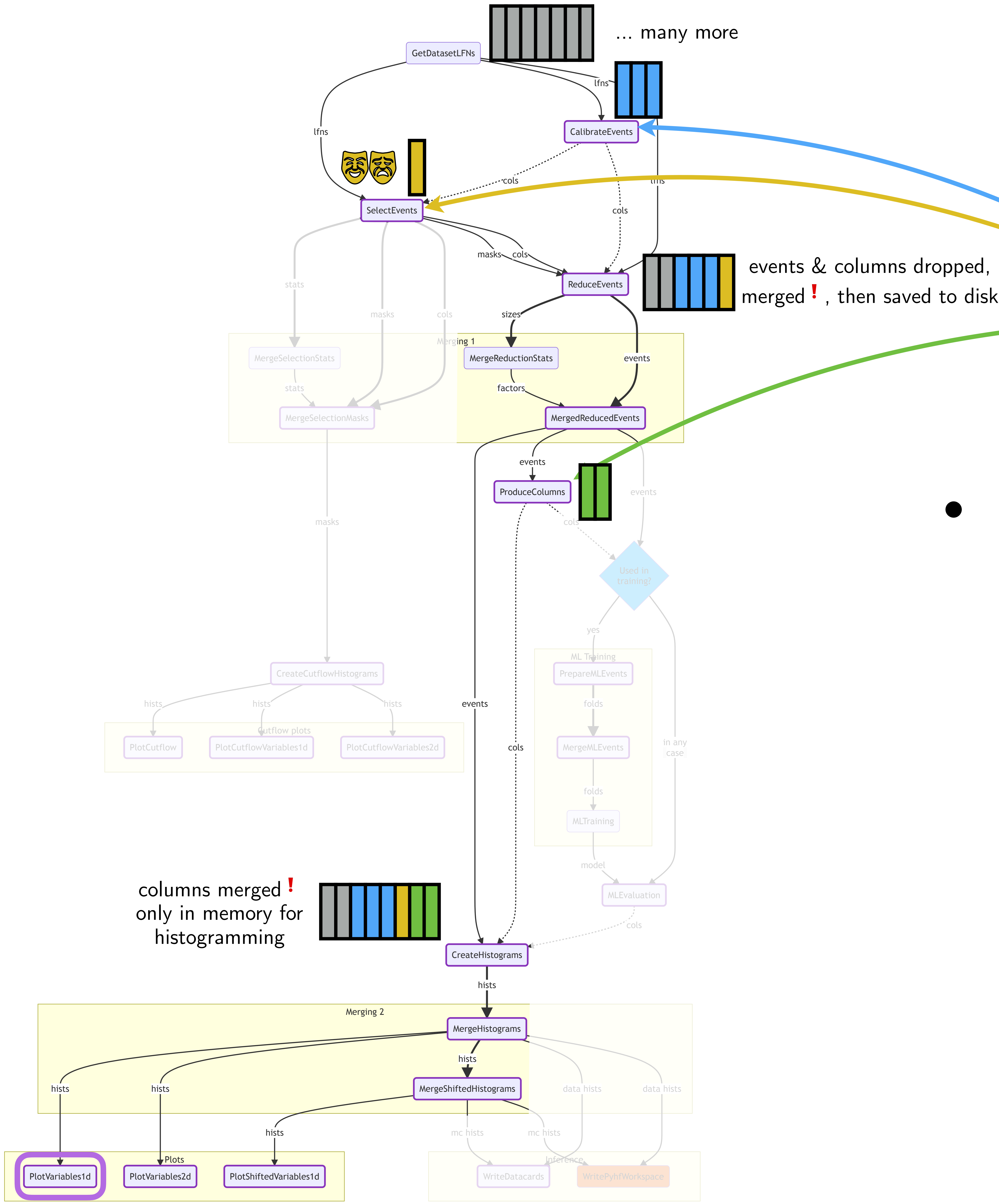
Nested producer

```python
@producer(
    uses={
        category_ids, features, normalization_weights, normalized_pdf_weight,
        normalized_murmuf_weight, normalized_pu_weight, normalized_btag_weights,
        tau_weights, electron_weights, muon_weights, trigger_weights,
    },
    produces={
        category_ids, features, normalization_weights, normalized_pdf_weight,
        normalized_murmuf_weight, normalized_pu_weight, normalized_btag_weights,
        tau_weights, electron_weights, muon_weights, trigger_weights,
    },
)
def default(self: Producer, events: ak.Array, **kwargs) -> ak.Array:
    # category ids
    events = self[category_ids](events, **kwargs)

    # features
    events = self[features](events, **kwargs)

    # mc-only weights
    if self.dataset_inst.is_mc:
        # normalization weights
        events = self[normalization_weights](events, **kwargs)

        # normalized pdf weight
        events = self[normalized_pdf_weight](events, **kwargs)

        # normalized renorm./fact. weight
        events = self[normalized_murmuf_weight](events, **kwargs)

        # normalized pu weights
        events = self[normalized_pu_weight](events, **kwargs)

        # btag weights
        events = self[normalized_btag_weights](events, **kwargs)

        # tau weights
        events = self[tau_weights](events, **kwargs)

        # electron weights
        events = self[electron_weights](events, **kwargs)

        # muon weights
        events = self[muon_weights](events, **kwargs)

        # trigger weights
        events = self[trigger_weights](events, **kwargs)

    return events
```

flow of columns



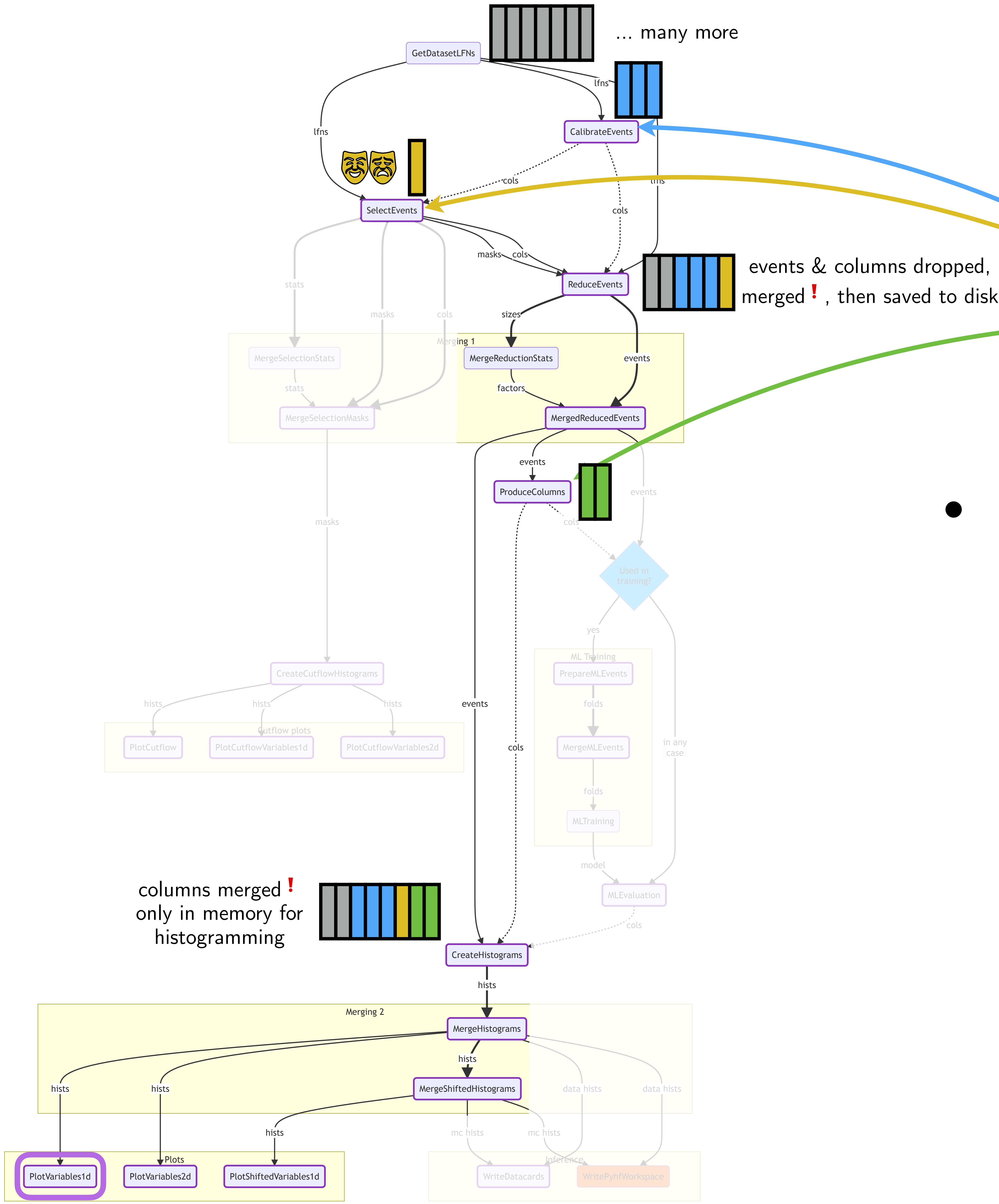

```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets hh_bbtautau \
    --calibrators jec \
    --selector full \
    --producers all_weights \
    --variables jet1_pt
```




```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets hh_bbtautau \
    --calibrators jec \
    --selector full \
    --producers all_weights \
    --variables jet1_pt
```

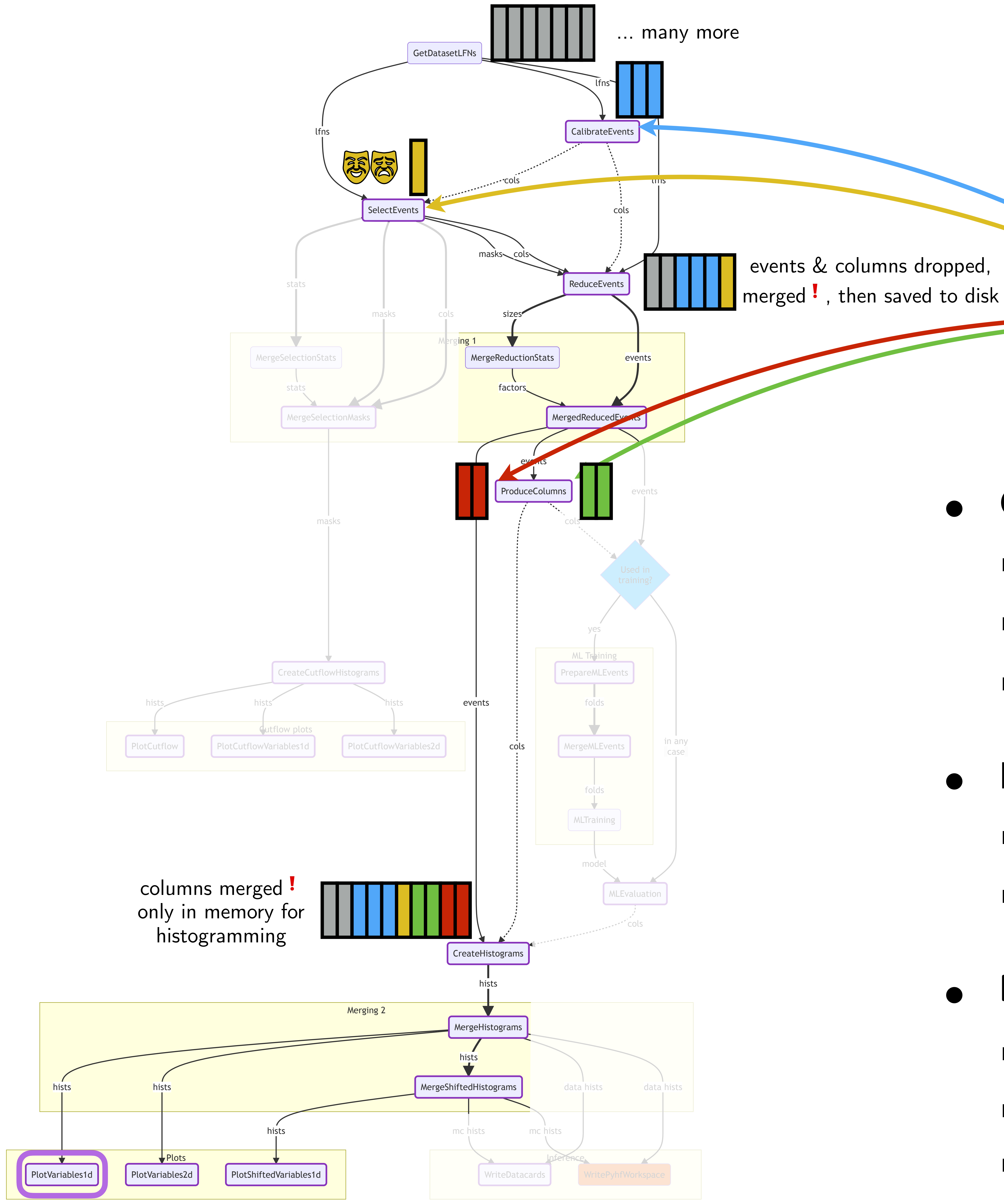
- Columns are
  - **produced** on demand
  - **read** only if required
  - **overlayed** & **aliased** to mimic coherent array !

flow of columns




```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets hh_bbtautau \
    --calibrators jec \
    --selector full \
    --producers all_weights,event_shape \
    --variables subjettiness
```

- Columns are
  - **produced** on demand
  - **read** only if required
  - **overlayed** & **aliased** to mimic coherent array !




```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets hh_bbtautau \
    --calibrators jec \
    --selector full \
    --producers all_weights,event_shape \
    --variables subjettiness
```

- Columns are
  - **produced** on demand
  - **read** only if required
  - **overlayed** & **aliased** to mimic coherent array !
- Existing columns
  - are **not reproduced**
  - can be **shared across groups**
- **NB**
  - Task ≠ jobs → jobs can run **multiple** tasks
  - Example producers in backup
  - IO description in backup

Base Stack

micromamba with conda-forge packages

→ contains all required **non-python** packages,

rarely updated

(`python3.9`, `bash/zsh`, `git`, `gfal2`)

Mamba

CONDA-FORGE

"cf" Sandbox

Relocatable python virtual env

→ All **python** packages needed to run tasks,

moderately updated

(`luigi`, `law`, `pyyaml`)

python virtualenv

Task sandboxes

Any type: venv, cmssw subshell, docker, ...

→ **Python** packages to run **a specific task**,

frequently updated

(`e.g. awkward`, `numpy`, `tensorflow`, `...`)

python virtualenv

CMSSW
(subshell)

...

Example: muon weight producer (as shown earlier)

```
@producer(
    uses={
        "nMuon", "Muon.pt", "Muon.eta",
    },
    produces={
        "muon_weight", "muon_weight_up", "muon_weight_down",
    },
    # only allowed on mc
    mc_only=True,
)
def muon_weights(
    self: Producer,
    events: ak.Array,
    muon_mask: ak.Array | type(Ellipsis) = Ellipsis,
    **kwargs,
) -> ak.Array:
    """ ... Creates muon weights using the correctionlib. ... """

    # flat absolute eta and pt views
    abs_eta = flat_np_view(abs(events.Muon.eta[muon_mask]), axis=1)
    pt = flat_np_view(events.Muon.pt[muon_mask], axis=1)

    # loop over systematics
    for syst, postfix in [
        ("sf", ""),
        ("systup", "_up"),
        ("systdown", "_down"),
    ]:
        sf_flat = self.muon_sf_corrector(self.year, abs_eta, pt, syst)

        # add the correct layout to it
        sf = layout_ak_array(sf_flat, events.Muon.pt[muon_mask])

        # create the product over all muons per event
        weight = ak.prod(sf, axis=1, mask_identity=False)

        # store it
        events = set_ak_column(events, f"muon_weight{postfix}", weight, value_type=np.float32)

    return events
```

Writing your own producer (calibrator, selector, ...) (1)

`@producer` decorator will create a class `muon_weights`

`uses` declares columns that should be **read**

`produces` declares columns to be **written**

Additional flags enable during checks, e.g.

- `mc_only` (`bool`), `data_only` (`bool`)
- `nominal_only` (`bool`), `shifts_only` (`set[str]`)

Wrapped function becomes the main **callable** of the class & always should at least accept `events` and `**kwargs`

Use `set_ak_column` to conveniently add new columns

Return all `events`
(selectors: return also a `SelectionResult`)

Example: muon weight producer (as shown earlier)

```python
@producer(
    uses={
        "nMuon", "Muon.pt", "Muon.eta",
    },
    produces={
        "muon_weight", "muon_weight_up", "muon_weight_down",
    },
    # only allowed on mc
    mc_only=True,
)
def muon_weights(
    self: Producer,
    events: ak.Array,
    muon_mask: ak.Array | type(Ellipsis) = Ellipsis,
    **kwargs,
) -> ak.Array:
    """ Creates muon weights using the correctionlib. """

    # flat absolute eta and pt views
    abs_eta = flat_np_view(abs(events.Muon.eta[muon_mask]), axis=1)
    pt = flat_np_view(events.Muon.pt[muon_mask], axis=1)

    # loop over systematics
    for syst, postfix in [
        ("sf", ""),
        ("systup", "_up"),
        ("systdown", "_down"),
    ]:
        sf_flat = self.muon_sf_corrector(self.year, abs_eta, pt, syst)

        # add the correct layout to it
        sf = layout_ak_array(sf_flat, events.Muon.pt[muon_mask])

        # create the product over all muons per event
        weight = ak.prod(sf, axis=1, mask_identity=False)

        # store it
        events = set_ak_column(events, f"muon_weight{postfix}", weight, value_type=np.float32)

    return events
```

`@producer` decorator will create a class `muon_weights`

`uses` declares columns that should be **read**

`produces` declares columns to be **written**

Additional flags enable during checks, e.g.

- `mc_only` (`bool`), `data_only` (`bool`)
- `nominal_only` (`bool`), `shifts_only` (`set[str]`)

Wrapped function becomes the main **callable** of the class & always should at least accept `events` and `**kwargs`

Use `set_ak_column` to conveniently add new columns

Return all `events`
(selectors: return also a `SelectionResult`)

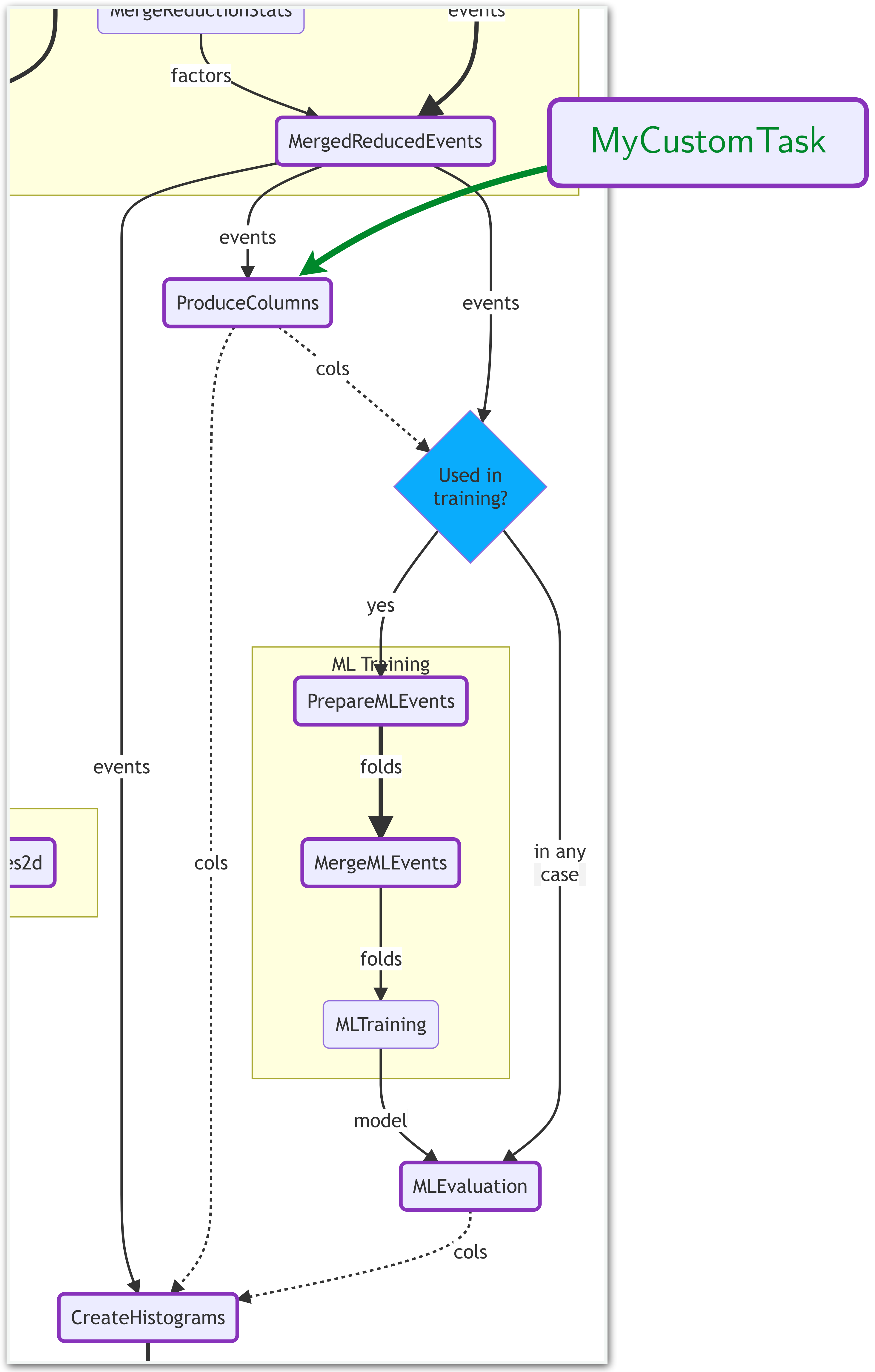
Where does the `muon_sf_corrector` come from?

Example: muon weight producer (as shown earlier)

```python
@producer(
    uses={
        "nMuon", "Muon.pt", "Muon.eta",
    },
    produces={
        "muon_weight", "muon_weight_up", "muon_weight_down",
    },
    # only allowed on mc
    mc_only=True,
)
def muon_weights(
    self: Producer,
    events: ak.Array,
    muon_mask: ak.Array | type(Ellipsis) = Ellipsis,
    **kwargs,
) -> ak.Array:
    """ Creates muon weights using the correctionlib. """

    # flat absolute eta and pt views
    abs_eta = flat_np_view(abs(events.Muon.eta[muon_mask]), axis=1)
    pt = flat_np_view(events.Muon.pt[muon_mask], axis=1)

    # loop over systematics
    for syst, postfix in [
        ("sf", ""),
        ("systup", "_up"),
        ("systdown", "_down"),
    ]:
        sf_flat = self.muon_sf_corrector(self.year, abs_eta, pt, syst)

        # add the correct layout to it
        sf = layout_ak_array(sf_flat, events.Muon.pt[muon_mask])

        # create the product over all muons per event
        weight = ak.prod(sf, axis=1, mask_identity=False)

        # store it
        events = set_ak_column(events, f"muon_weight{postfix}", weight, value_type=np.float32)

    return events
```

- From previous slide: "Wrapped function becomes the main **callable** of the class"
  → Called for every chunk of events during processing

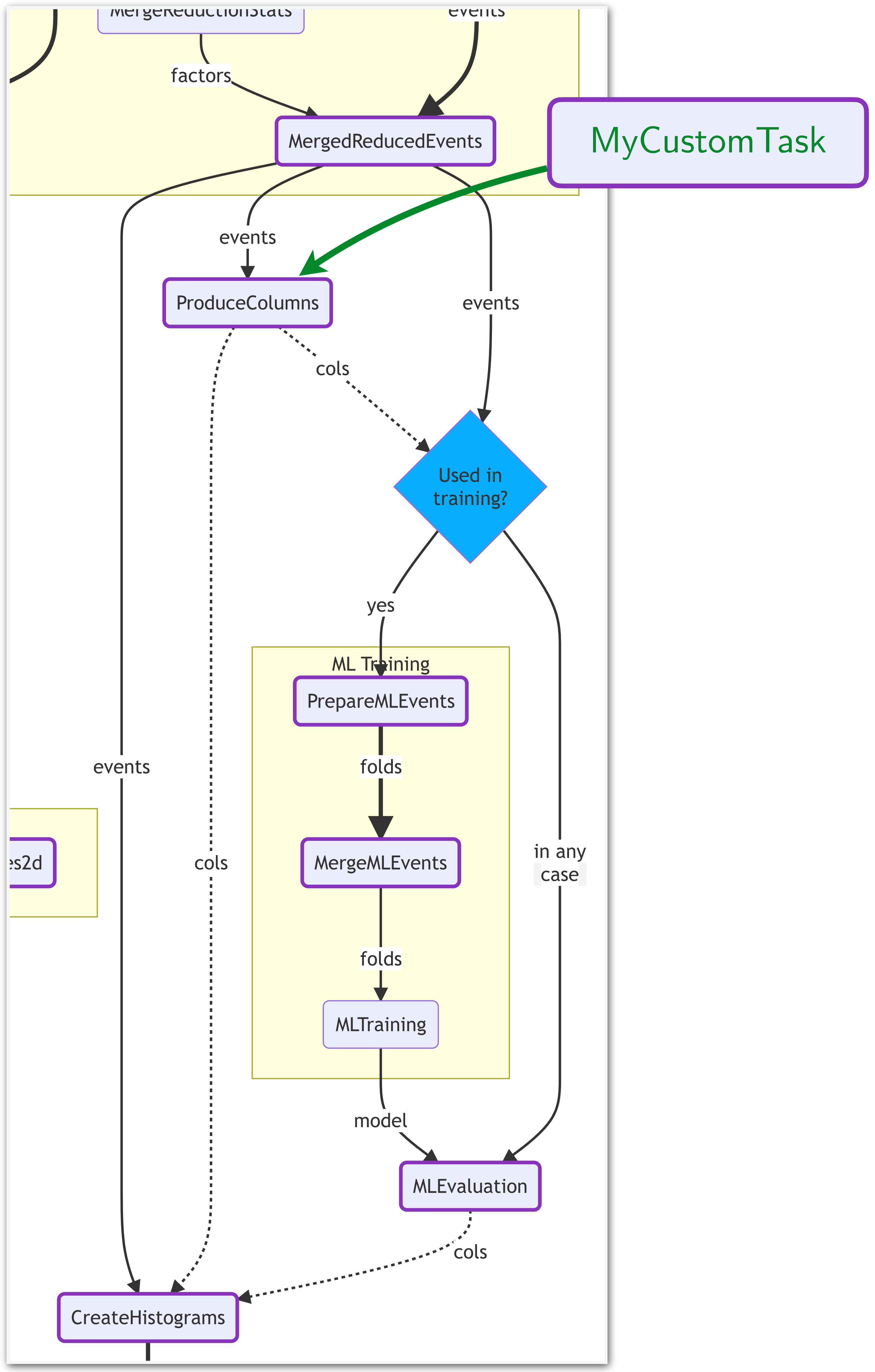
- **But**
  - How to setup objects **before** the actual event processing?
  - How to define a **custom dependency**?
    (i.e., task(s) on whose outputs the producer depends)



- From previous slide: "Wrapped function becomes the main **callable** of the class"
  → Called for every chunk of events during processing

- **But**
  - How to setup objects **before** the actual event processing?
  - How to define a **custom dependency**?
    (i.e., task(s) on whose outputs the producer depends)

- **Three additional hooks**
  - `init(self) -> None`
    - Method called as soon as producer registered by a task
    - Receives important task variables via `self` (requested dataset, shift, ...)
  - `requires(self, reqs: dict) -> None`
    - Method called when task declares its dependcies
    - Allows injecting custom dependencies into `reqs` that will be resolved by luigi
  - `setup(self, reqs: dict, inputs: dict, reader_targes: dict) -> None`
    - Method called in task's `run()` **once** before loop over event chunks
    - Receives `reqs` defined before and corresponding `inputs`
    - Allows setting up objects to be used in main callable



- `init(self) -> None`

```python
@jer.init
def jer_init(self: Calibrator) -> None:
    if self.propagate_met:
        self.uses |= {
            "MET.pt", "MET.phi",
        }
        self.produces |= {
            "MET.pt", "MET.phi", "MET.pt_jer_up", "MET.pt_jer_down", "MET.phi_jer_up",
            "MET.phi_jer_down", "MET.pt_unsmeared", "MET.phi_unsmeared",
        }
```

from calibration/cms/jets.py

- `requires(self, reqs: dict) -> None`

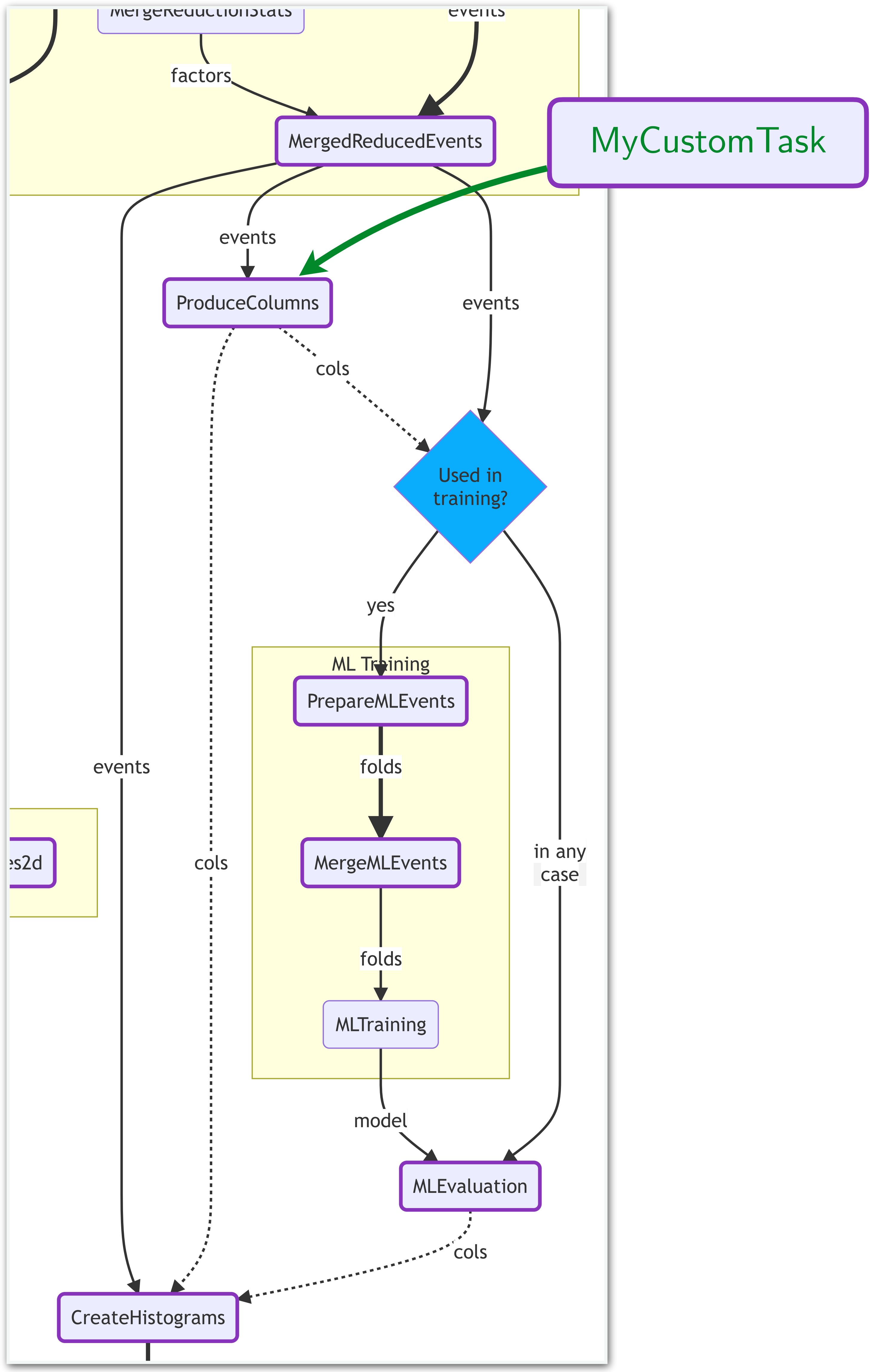
```python
@muon_weights.requires
def muon_weights_requires(self: Producer, reqs: dict) -> None:
    from columnflow.tasks.external import BundleExternalFiles
    reqs["external_files"] = BundleExternalFiles.req(self.task)
```

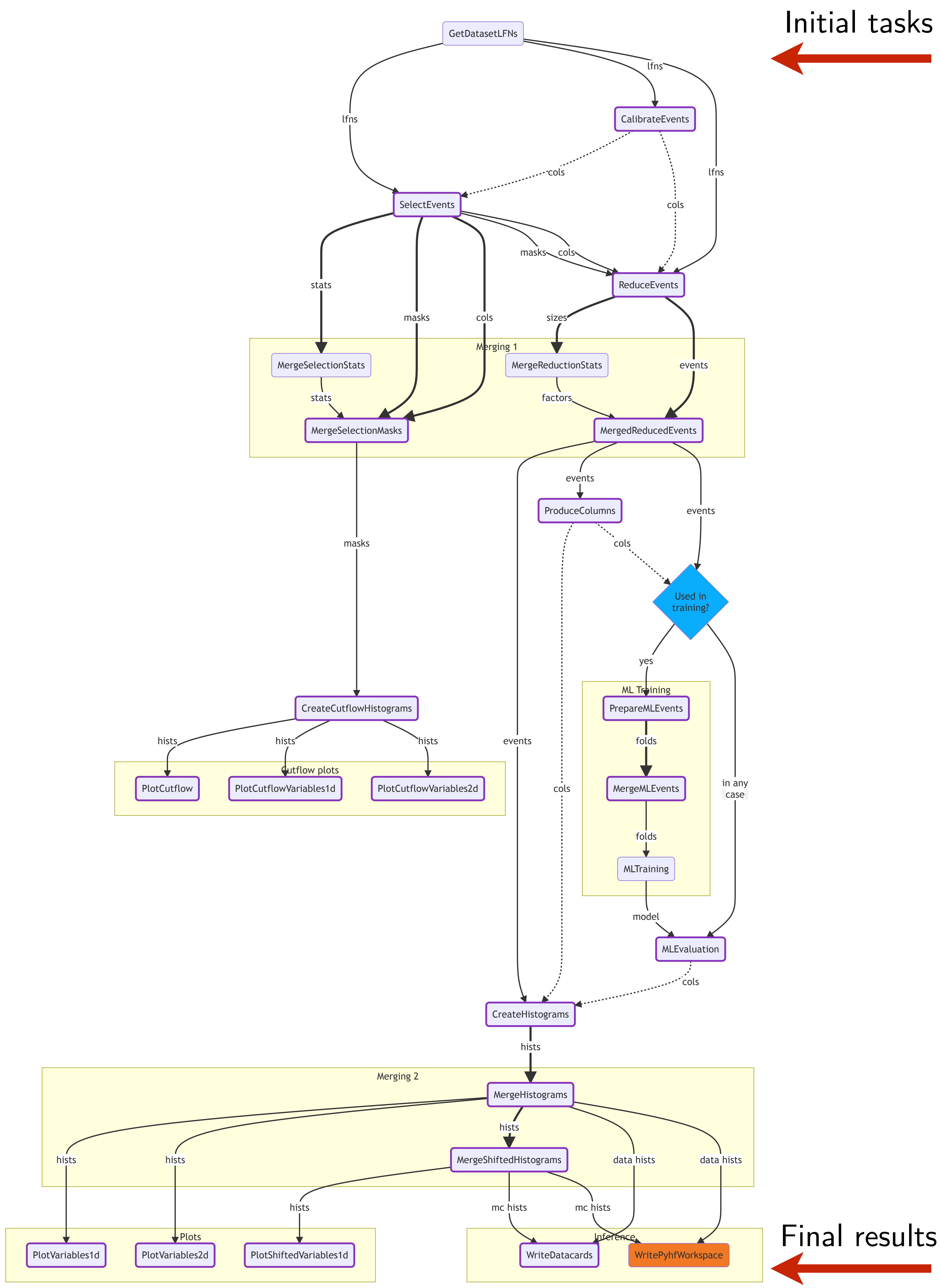
from production/cms/muon.py

- `setup(self, reqs: dict, inputs: dict, reader_targes: dict) -> None`

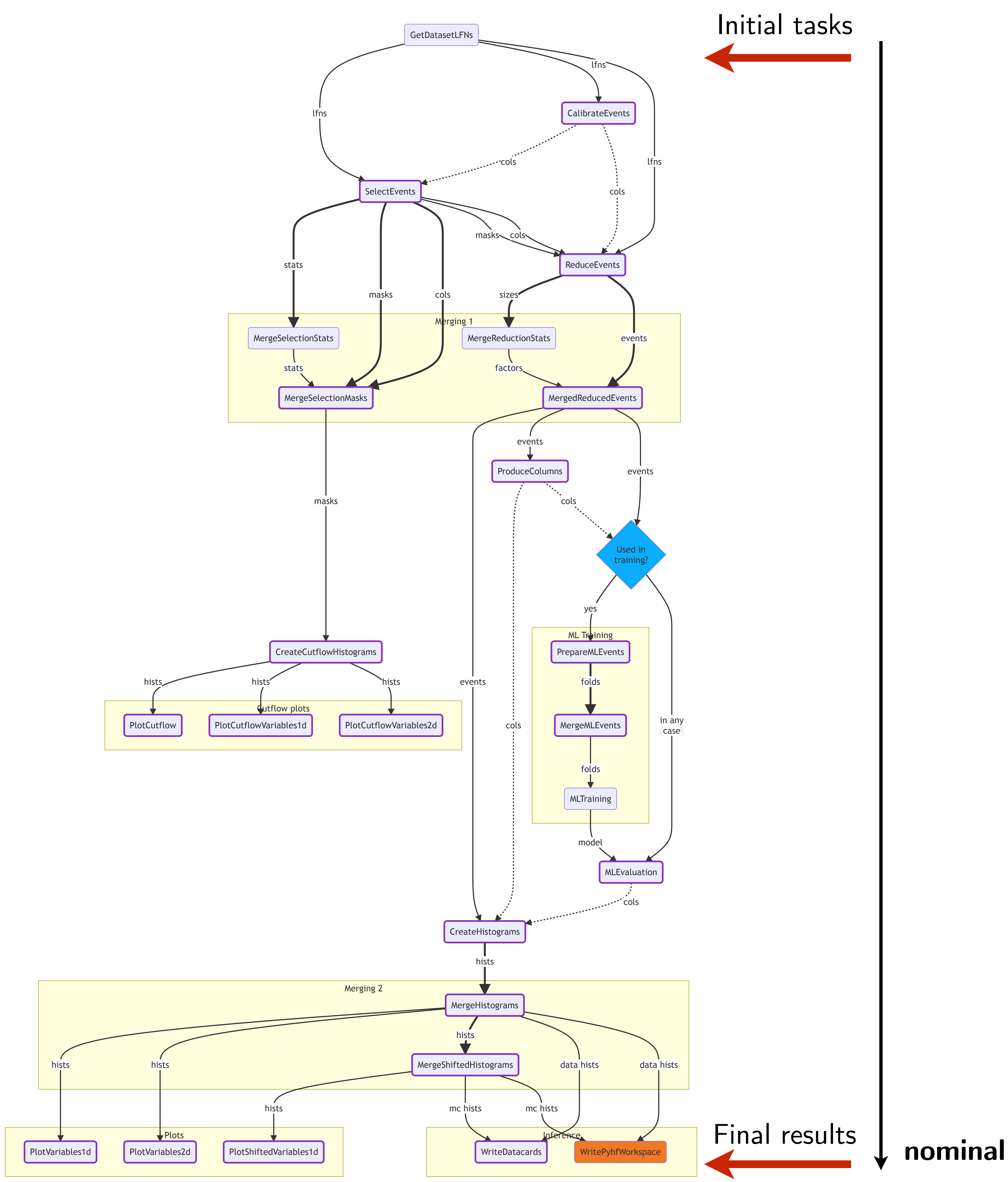
```python
@muon_weights.setup
def muon_weights_setup(self: Producer, reqs: dict, inputs: dict, reader_targets: InsertableDict) -> None:
    bundle = reqs["external_files"]

    # create the corrector
    import correctionlib
    correctionlib.highlevel.Correction.__call__ = correctionlib.highlevel.Correction.evaluate
    correction_set = correctionlib.CorrectionSet.from_string(
        self.get_muon_file(bundle.files).load(formatter="gzip").decode("utf-8"),
    )
    corrector_name, self.year = self.get_muon_config()
    self.muon_sf_corrector = correction_set[corrector_name]
```

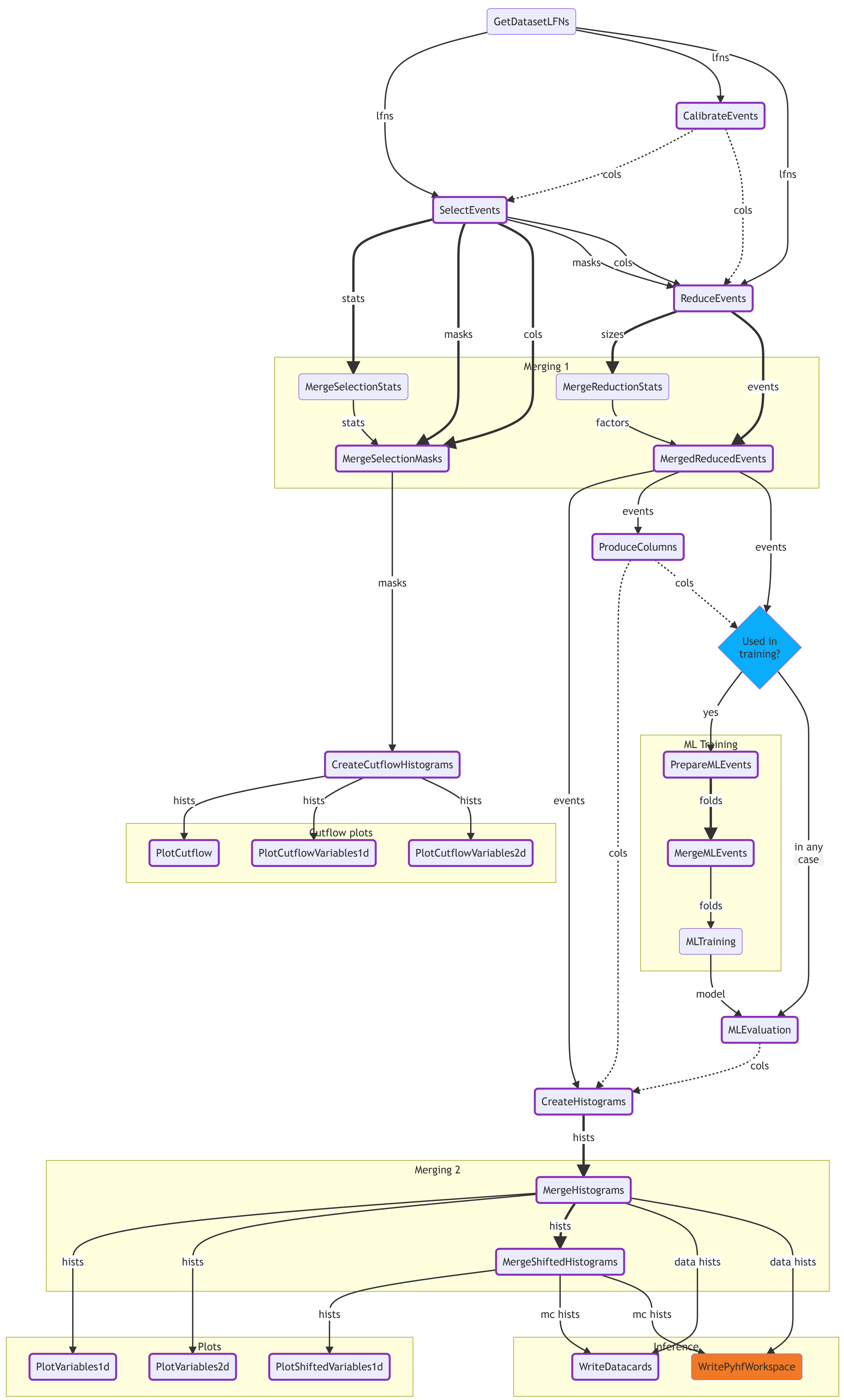
from production/cms/muon.py




Initial tasks
GetDatasetLFNs
lfns
CalibrateEvents
cols
SelectEvents
masks
stats
ReduceEvents
sizes
events
Merging 1
MergeSelectionStats
MergeReductionStats
factors
MergeSelectionMasks
MergedReducedEvents
ProduceColumns
Used in training?
yes
in any case
ML Training
PrepareMLEvents
folds
MergeMLEvents
MLTraining
model
MLEvaluation
CreateCutflowHistograms
hists
Cutflow plots
PlotCutflow
PlotCutflowVariables1d
PlotCutflowVariables2d
CreateHistograms
Merging 2
MergeHistograms
MergeShiftedHistograms
data hists
mc hists
Plots
PlotVariables1d
PlotVariables2d
PlotShiftedVariables1d
inference
WriteDatacards
WritePyhfWorkspace
Final results

Conceptual handling of systematic uncertainties





Initial tasks

Final results

nominal **tune(up|down)** **jec(up|down)** **pileup(up|down)** ...

Key idea

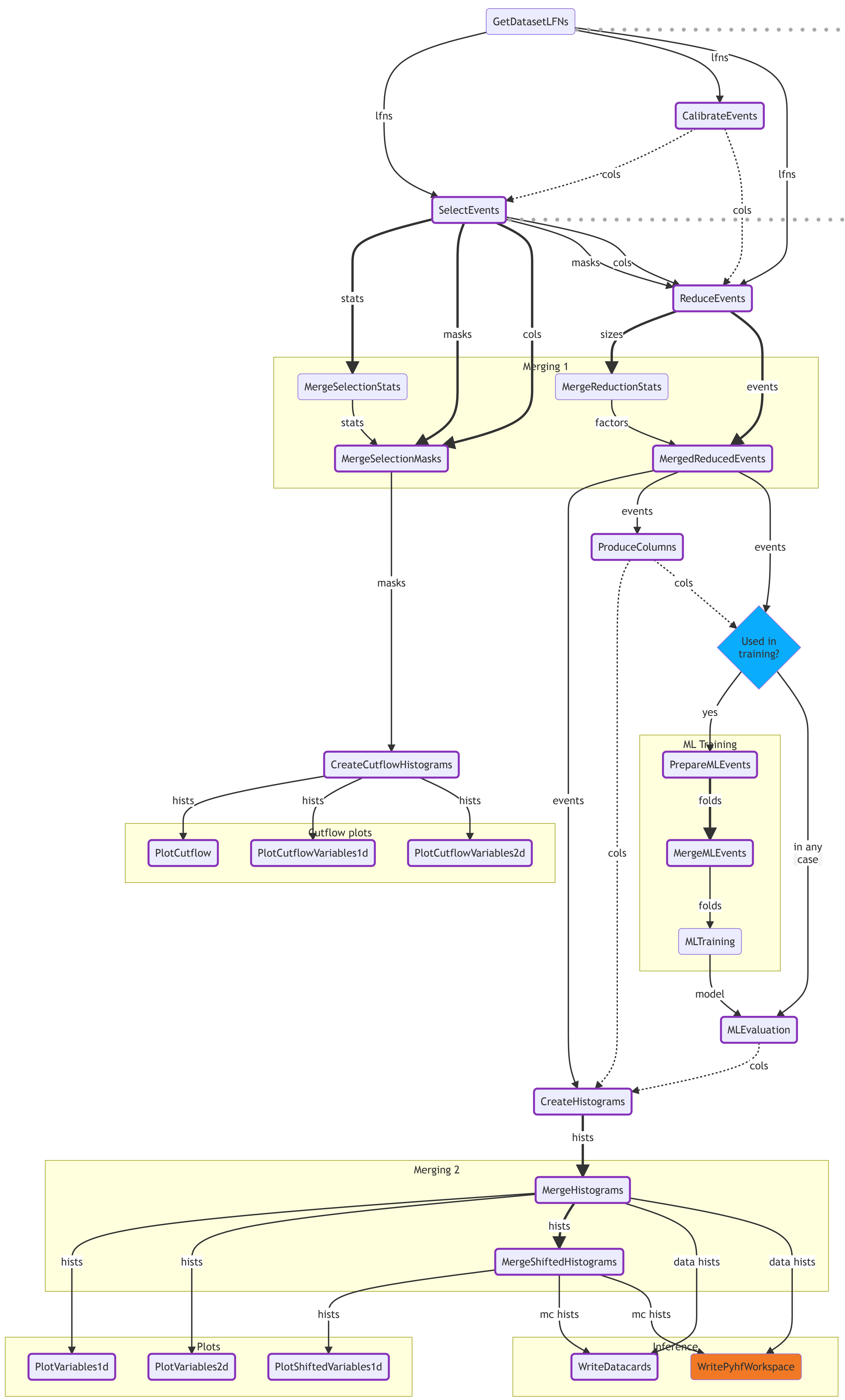
Tasks *know* which uncertainties

- *they implement*
- they *depend on* (through upstream tasks)

Key idea

Tasks *know* which uncertainties

- *they implement*
- they *depend on* (through upstream tasks)



Initial tasks

reuses all **nominal** outputs above **SelectEvents**

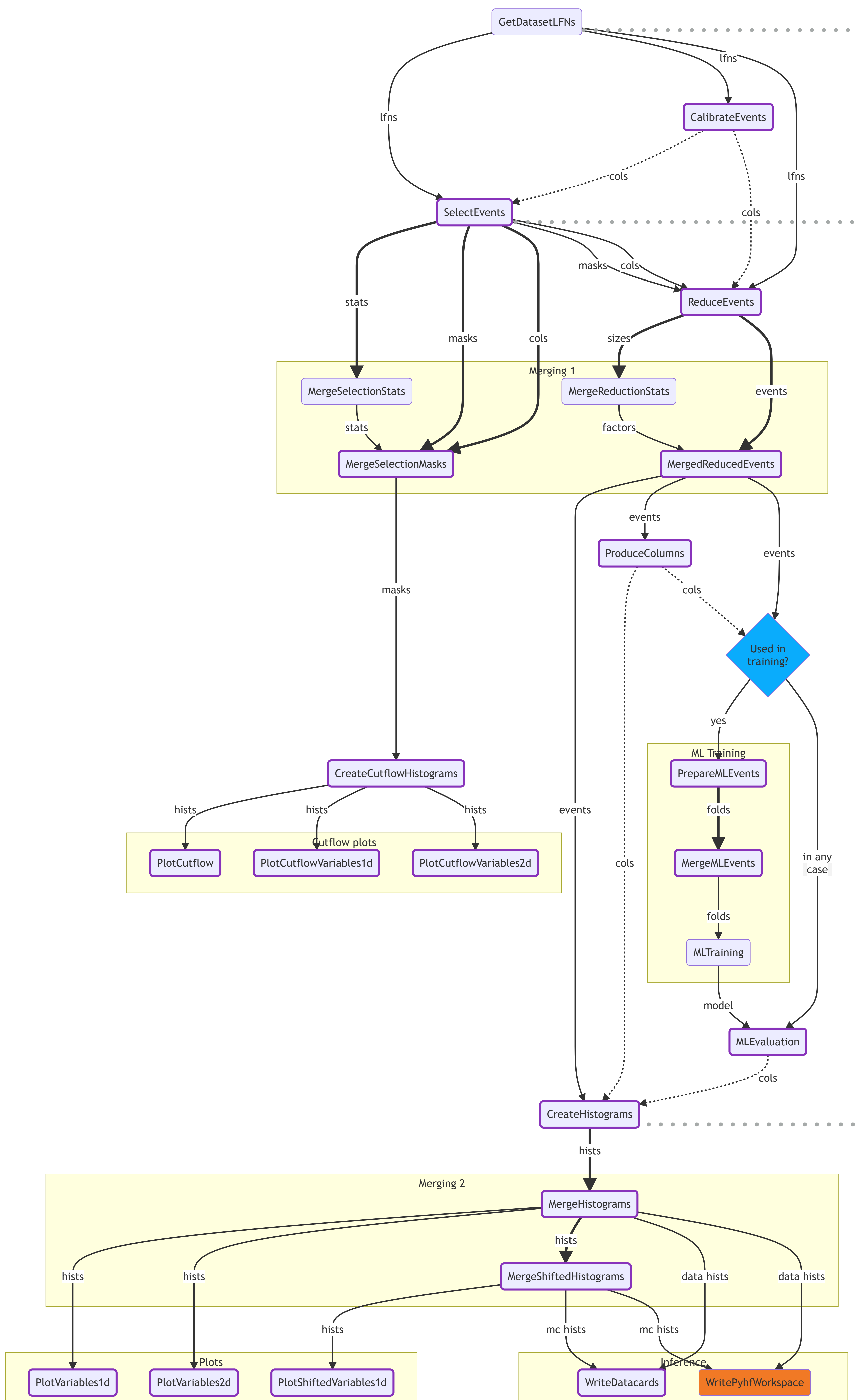
Final results

nominal | **tune(up|down)** | **jec(up|down)** | **pileup(up|down)** | …

Key idea

Tasks *know* which uncertainties

- *they implement*
- they *depend on* (through upstream tasks)



Initial tasks

reuses all **nominal** outputs above **SelectEvents**

reuses all **nominal** outputs above **CreateHistograms**

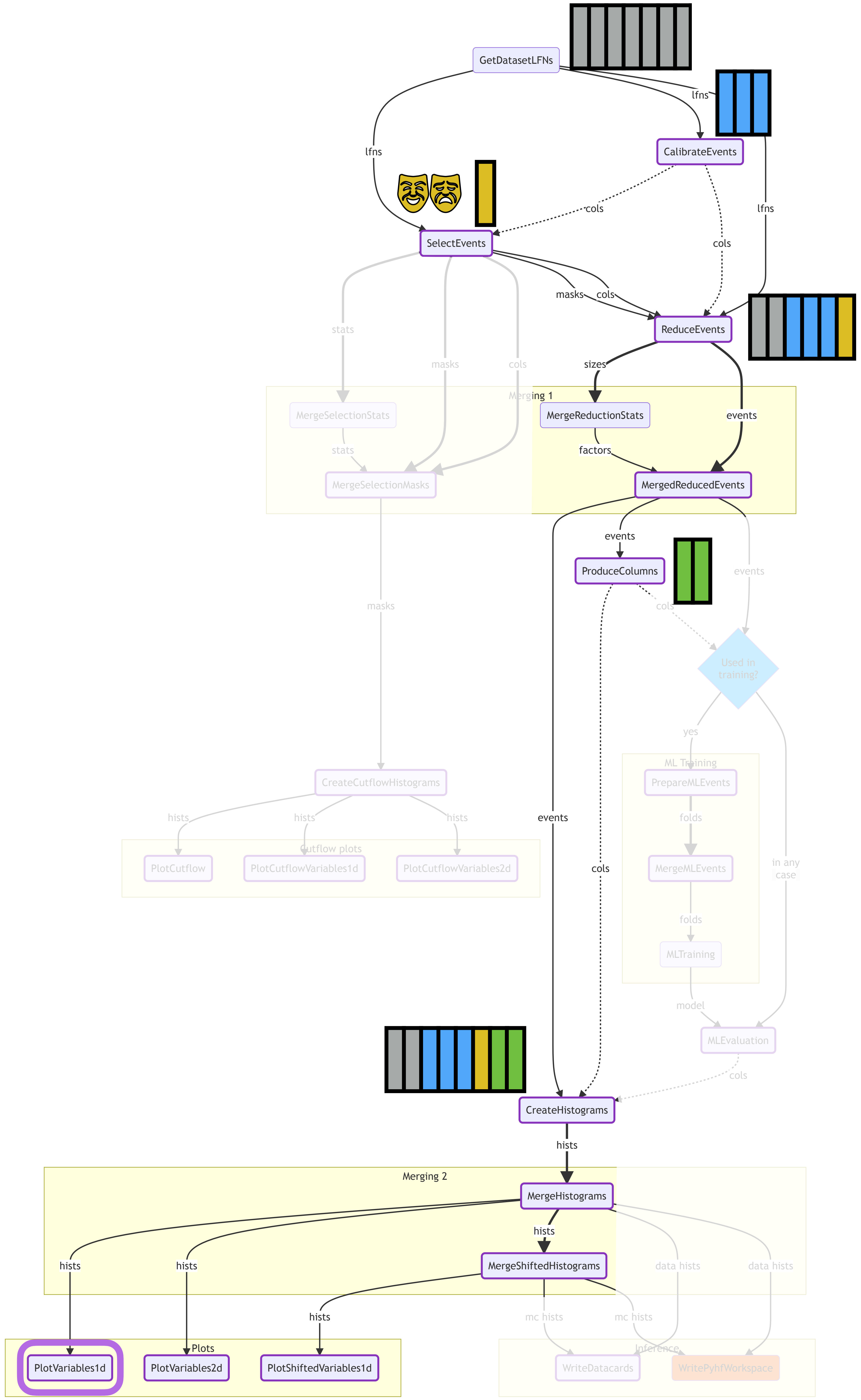
Final results

nominal | **tune(up|down)** | **jec(up|down)** | **pileup(up|down)** ...

Key idea

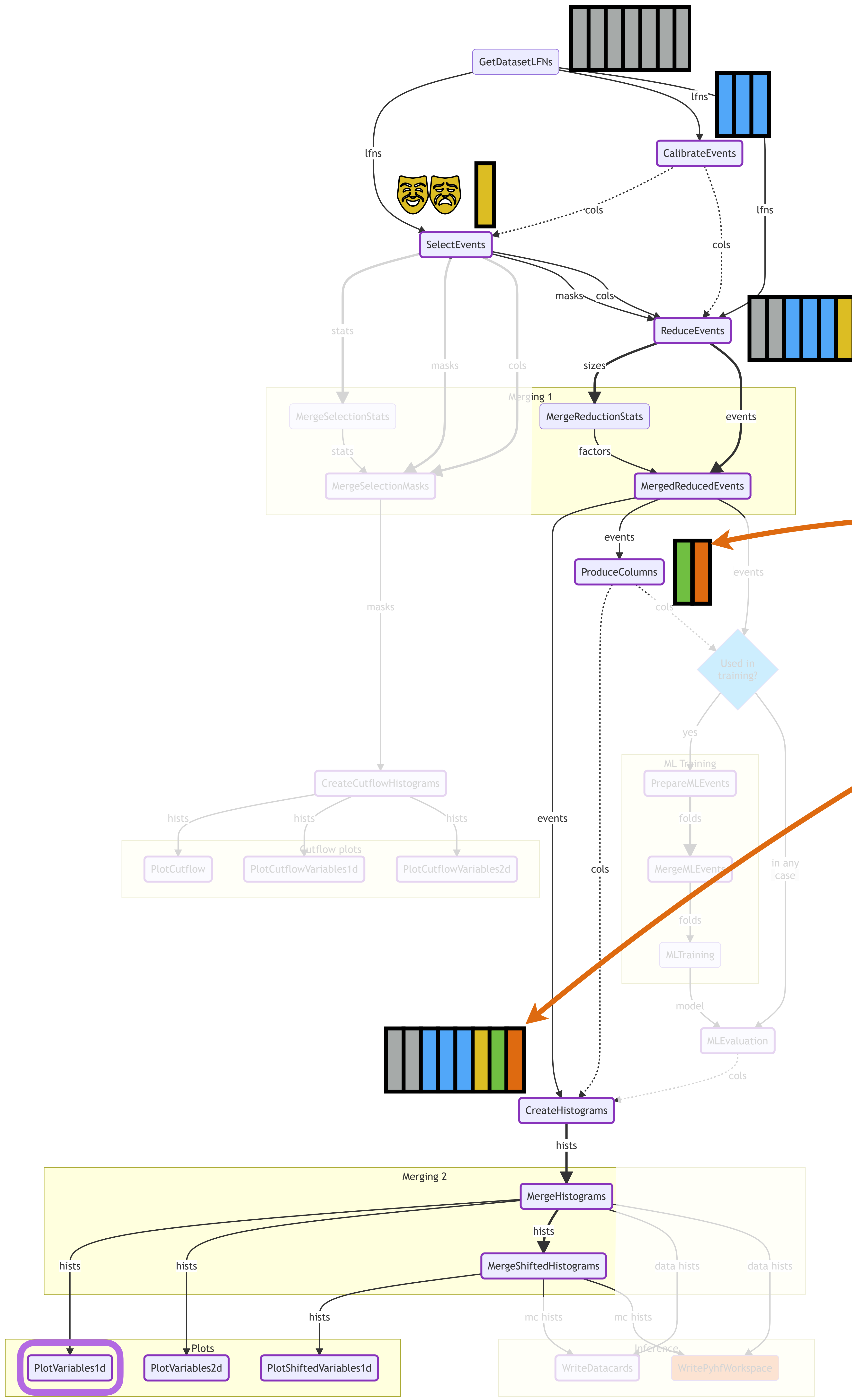
Tasks *know* which uncertainties

- *they implement*
- they *depend on* (through upstream tasks)




```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets hh_bbtautau \
    --calibrators jec \
    --selector full \
    --producers all_weights \
    --variables jet1_pt \
    --shift nominal
```

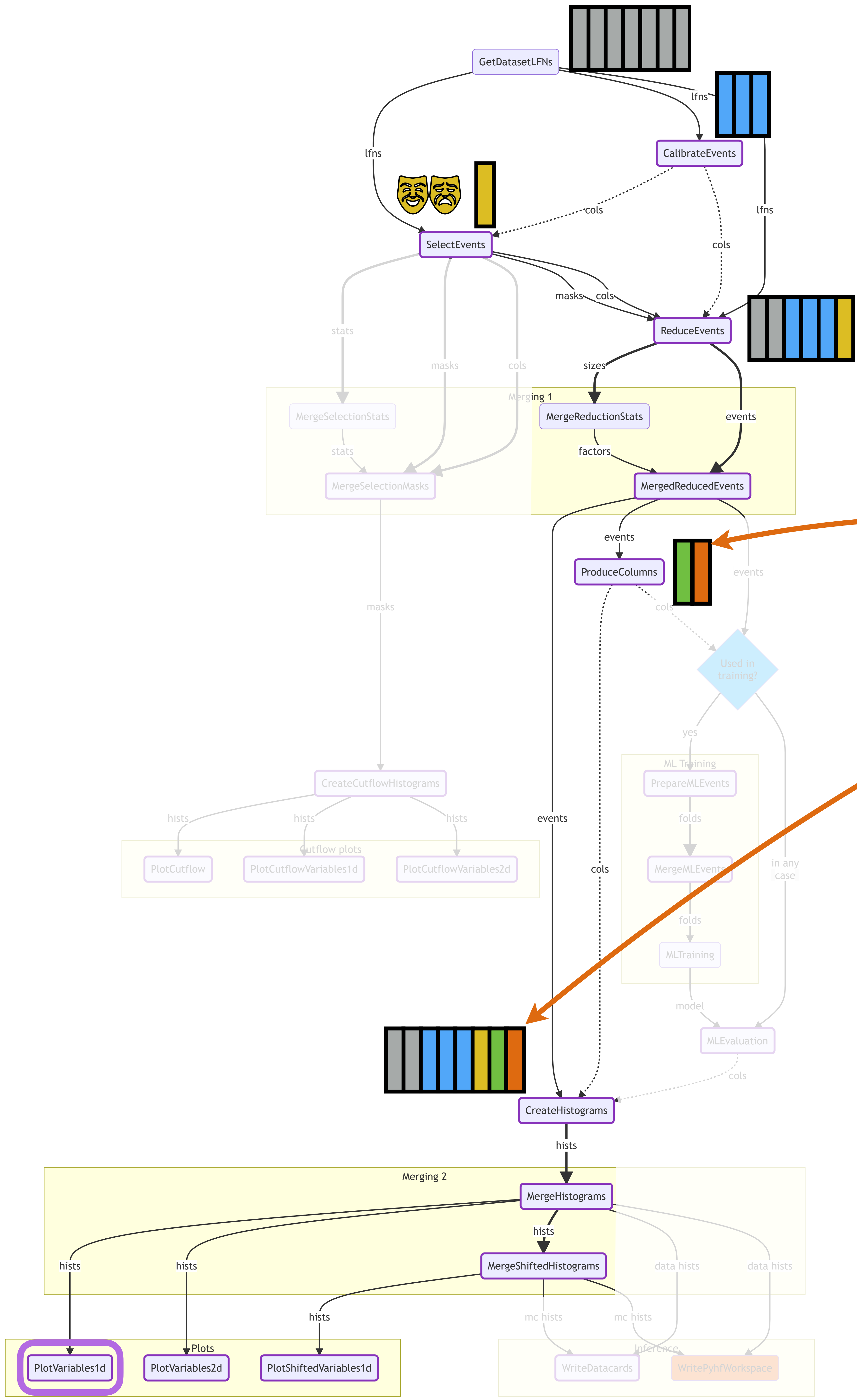
usually the default




```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets hh_bbtautau \
    --calibrators jec \
    --selector full \
    --producers all_weights \
    --variables jet1_pt \
    --shift tauid_up
```

- Handling of systematics
  - fully outsourced to **task dependency resolution**
  - **efficient**, no unnecessary computations
  - executable with **high parallelism**

flow of **columns**: systematic uncertainties




```
> law run cf.PlotVariables1D \
    --version dev1 \
    --datasets hh_bbtautau \
    --calibrators jec \
    --selector full \
    --producers all_weights \
    --variables jet1_pt \
    --shift tauid_up
```

- Handling of systematics
  - fully outsourced to **task dependency resolution**
  - **efficient**, no unnecessary computations
  - executable with **high parallelism**

- It's the analyzer's choice
  - **where** varied columns are produced
  - if they are already part of *normal* columns
    - Computationally *trivial*: produce right away in
    - Computationally *demanding*: produce in parallel

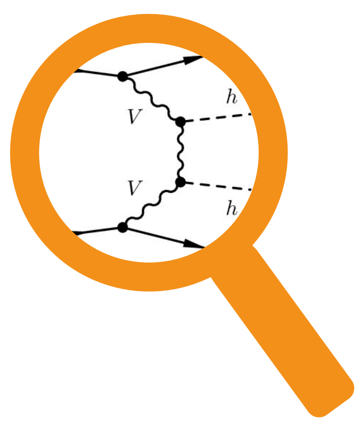
Orchestration software stack

luigi → law → columnflow → analysis

workflow engine
(originally by Spotify)

layer for HEP & scale-out
(experiment independent)

"framework"
(experiment independent*)

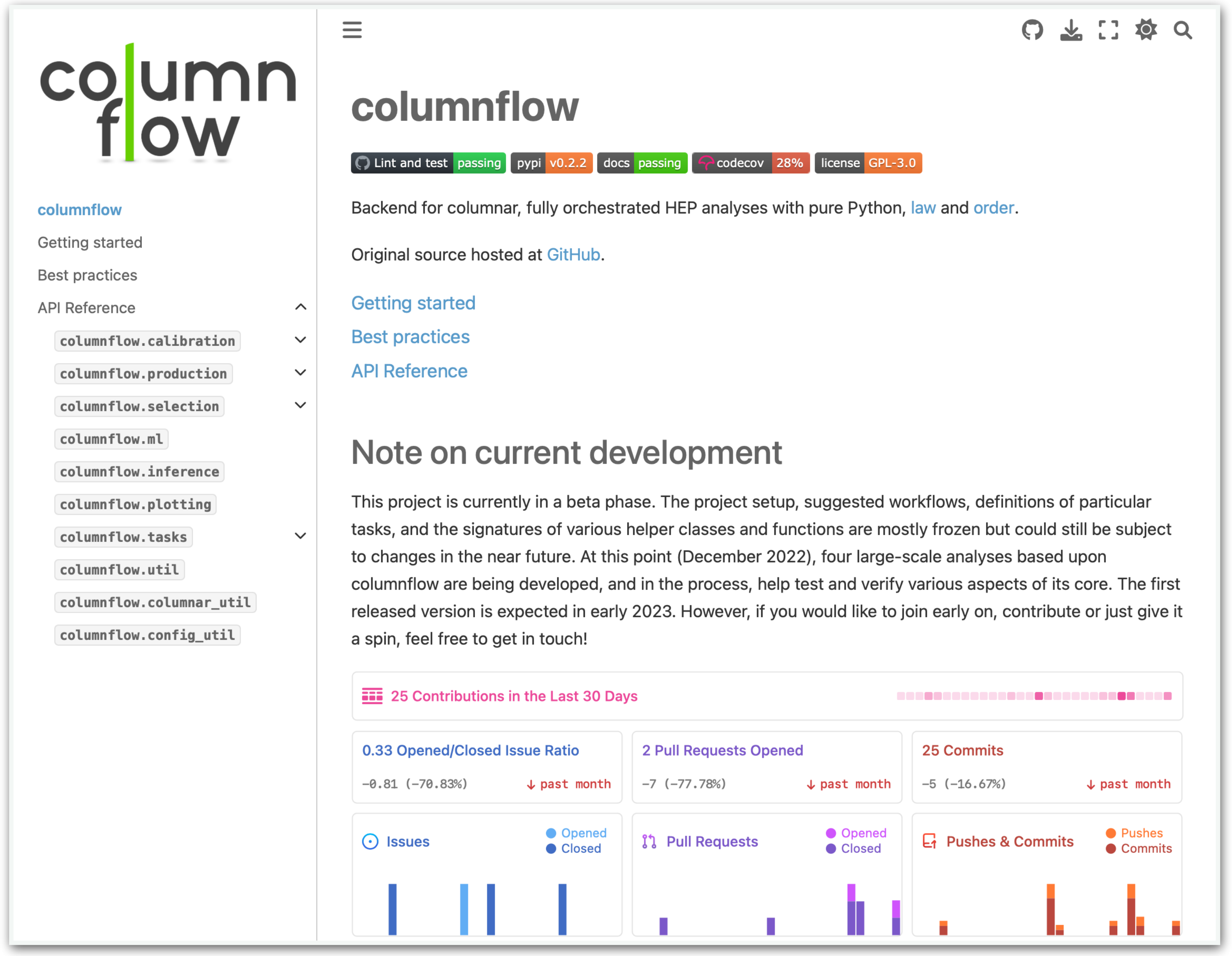


analysis

* soon

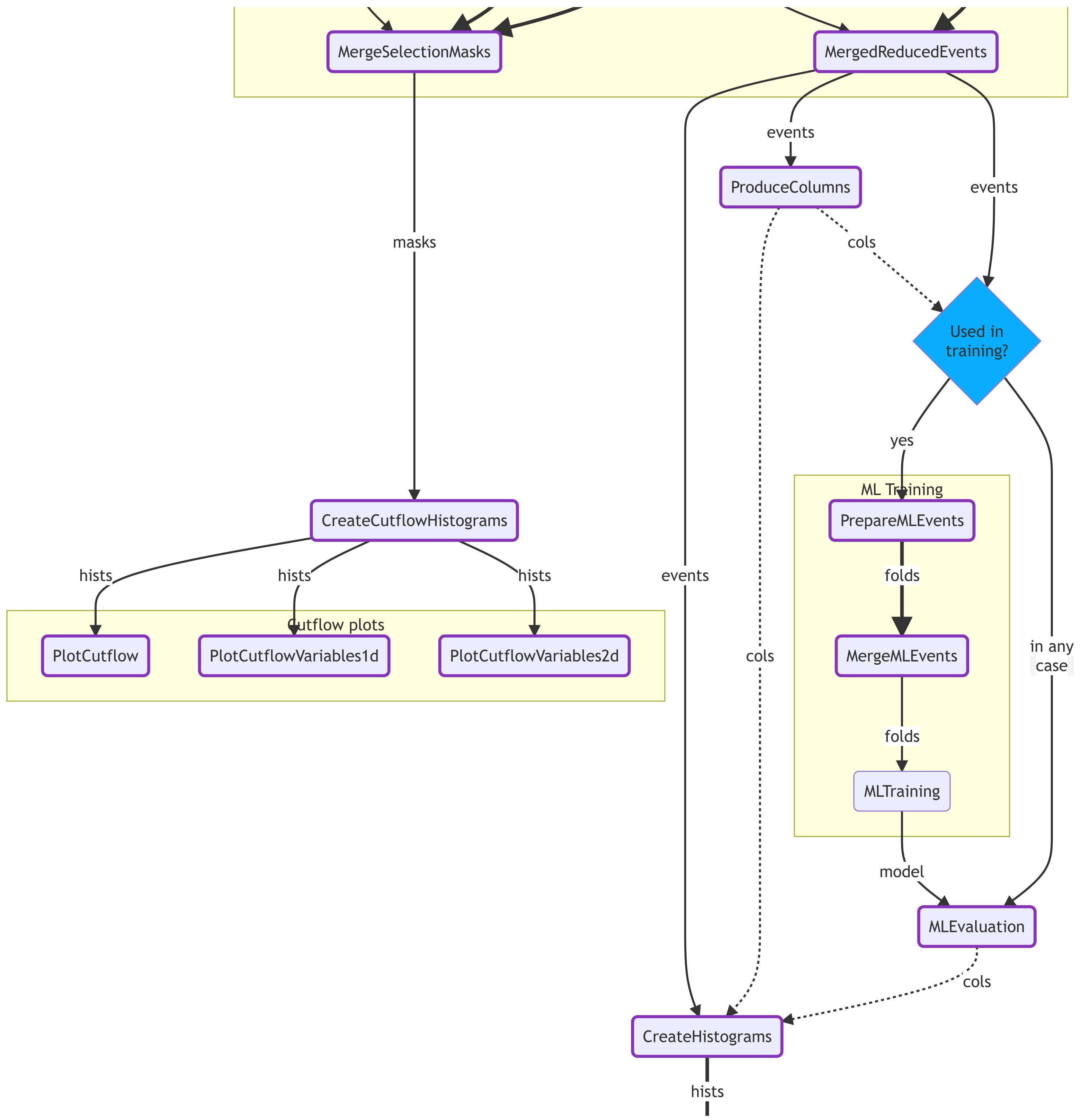
- **Python framework for vectorized, columnar HEP analysis with nano-like inputs**
  - Mostly experiment agnostic core
  - Fully orchestrated & automated
  - Intermediate outputs
  - Efficient through on-demand column production & retrieval
  - Able to incorporate **any remote resource**

- Checks 15/17 "ideal workflow"items of CMS ATTF report (Sec. 4, backup)
  - Vast Python (HEP) community and tool landscape is key



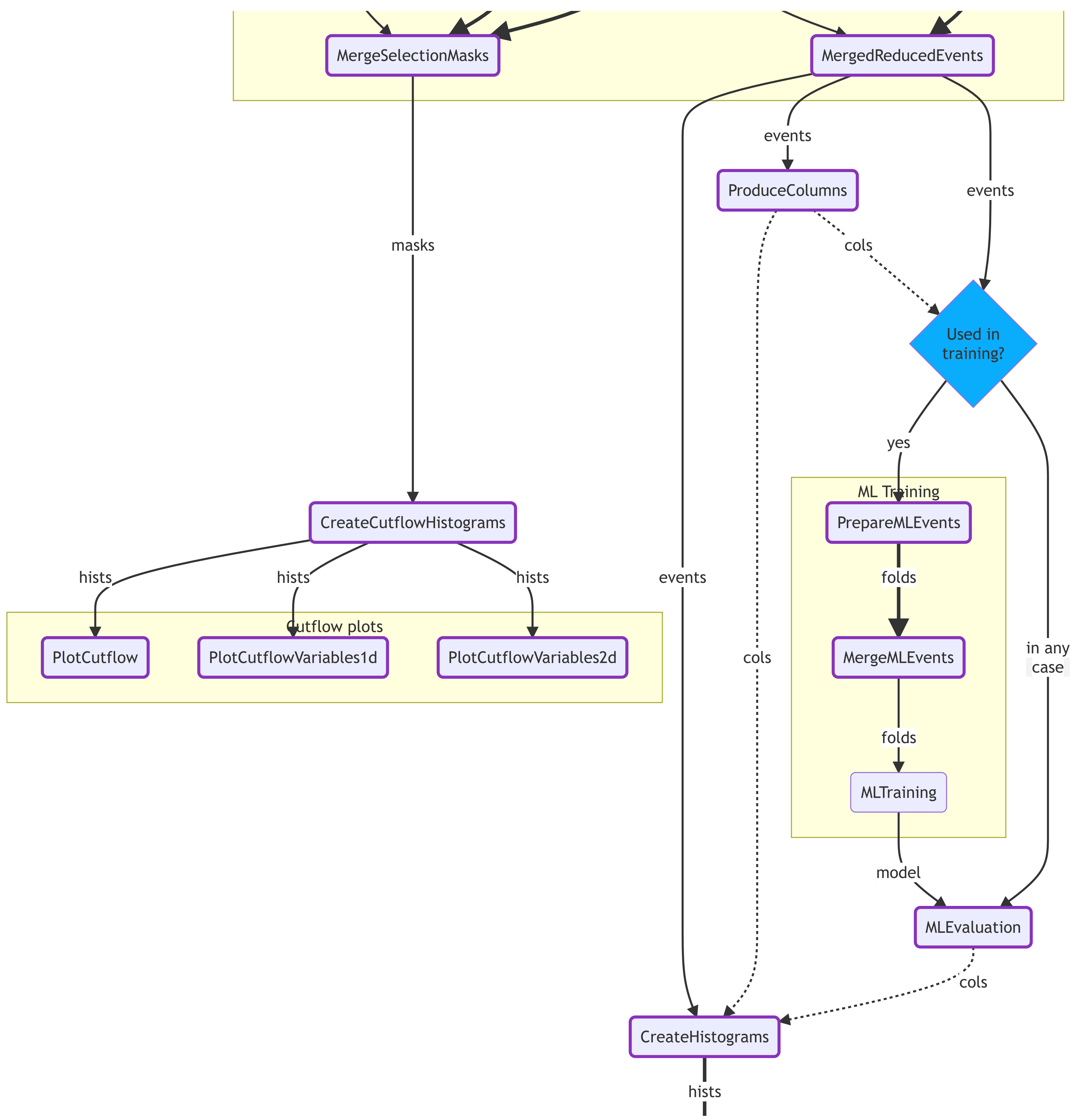
- **Currently pushing for extensive documentation release**
- Feedback still highly appreciated !
- github.com/columnflow, columnflow.rtfd.io

columnflow technicalities



- **Case 1**: Create histograms
  - `law run cf.CreateHistograms --dataset tt \`
    `--producers my_features --variables jet*`
  - Loads default columns from "MergeReducedEvents" **plus** columns created by a producer called "my_features"

```python
@producer(
    uses={"Jet.pt", "Jet.phi"},
    produces={"Jet.px", "Jet.py"},
)
def my_features(self: Producer, events: ak.Array, **kwargs) -> ak.Array:
    events = set_ak_column_f32(events, "Jet.px", events.Jet.pt * np.cos(events.Jet.phi))
    events = set_ak_column_f32(events, "Jet.py", events.Jet.pt * np.sin(events.Jet.phi))

    return events
```



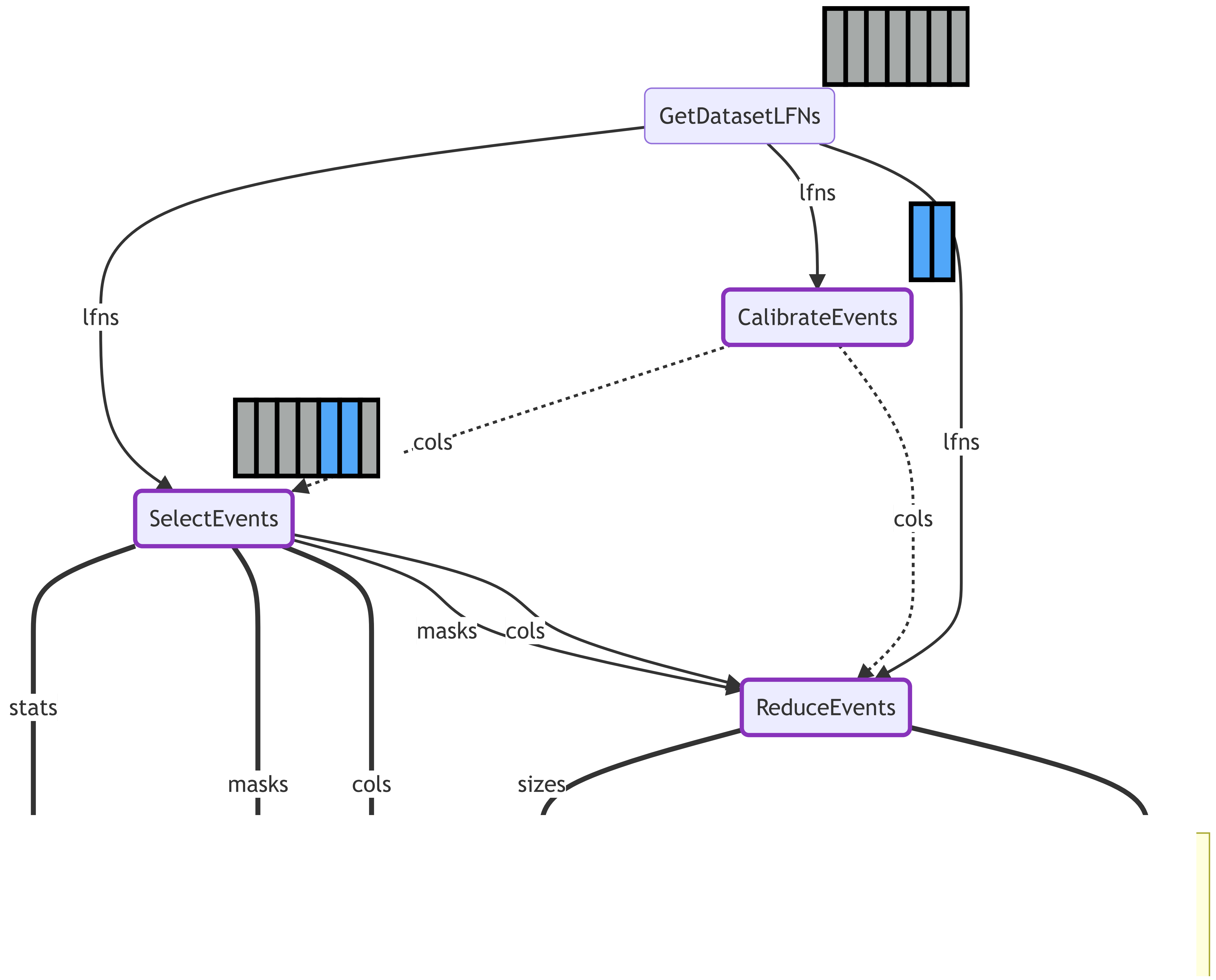
- **Case 1**: Create histograms
  - `law run cf.CreateHistograms --dataset tt \`
    `--producers my_features --variables jet*`
  - Loads default columns from "MergeReducedEvents" **plus** columns created by a producer called "my_features"

```
@producer(
    uses={"Jet.pt", "Jet.phi"},
    produces={"Jet.px", "Jet.py"},
)
def my_features(self: Producer, events: ak.Array, **kwargs) -> ak.Array:
    events = set_ak_column_f32(events, "Jet.px", events.Jet.pt * np.cos(events.Jet.phi))
    events = set_ak_column_f32(events, "Jet.py", events.Jet.pt * np.sin(events.Jet.phi))

    return events
```

- **Case 2**: Create different histograms
  - `law run cf.CreateHistograms --dataset tt \`
    `--producers my_features,event_shapes --variables jet*`
  - Loads default columns from "MergeReducedEvents" **plus** columns created producer**s** "my_features" **and** "event_shapes"

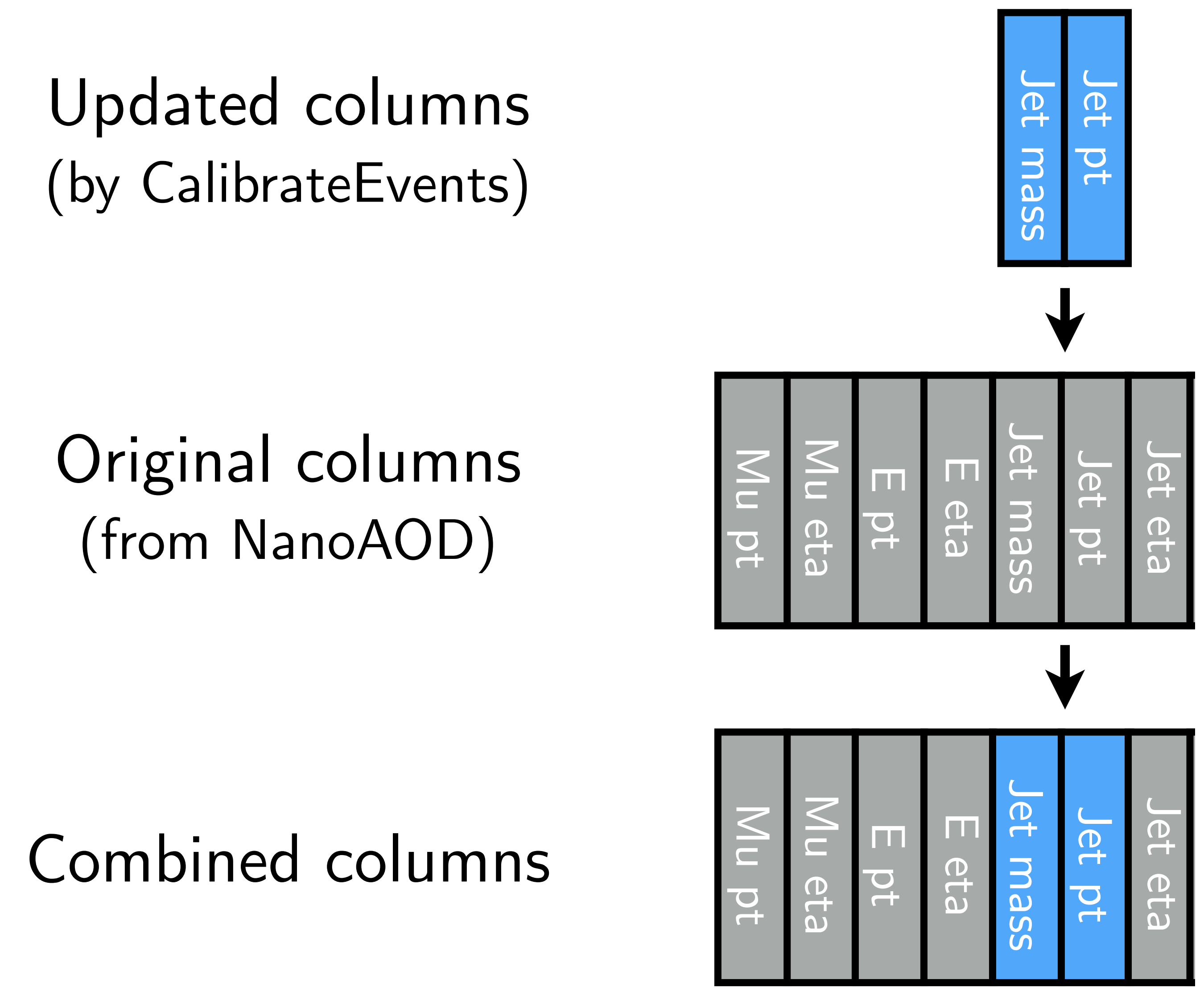
```
@producer(
    uses={"..."},
    produces={"..."},
)
def event_shapes(self: Producer, events: ak.Array, **kwargs) -> ak.Array:
    events = set_ak_column_f32(events, "fox_wolfram1", ...)
    events = set_ak_column_f32(events, "subjettiness", ...)
    ...

    return events
```

  - Only processes "event_shapes", reuses columns from "my_features"

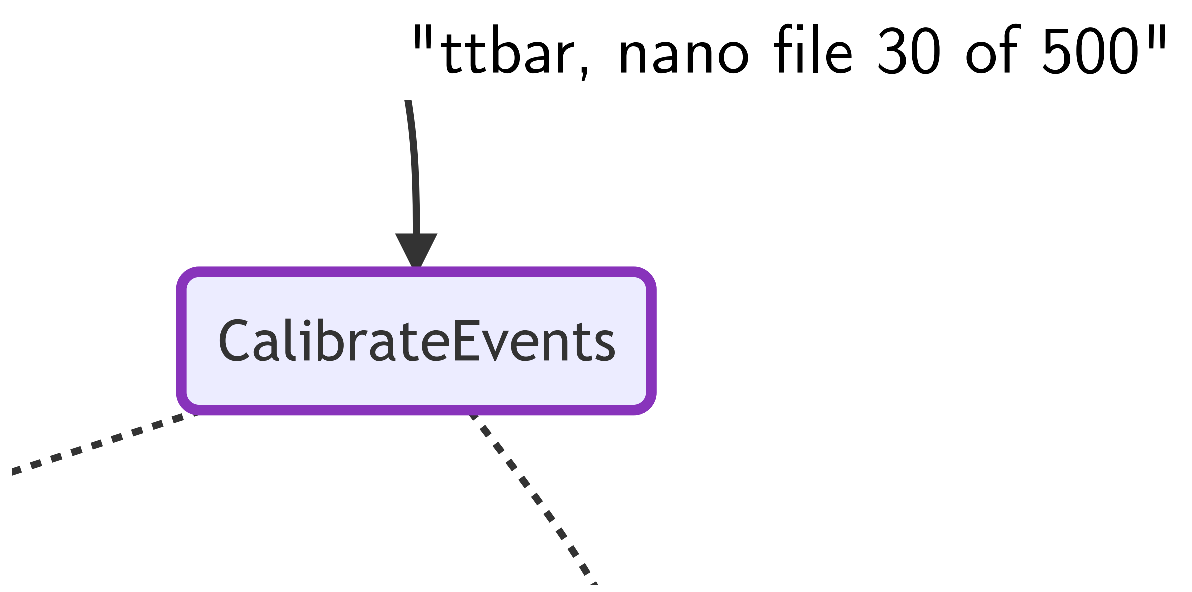


Layering of columns

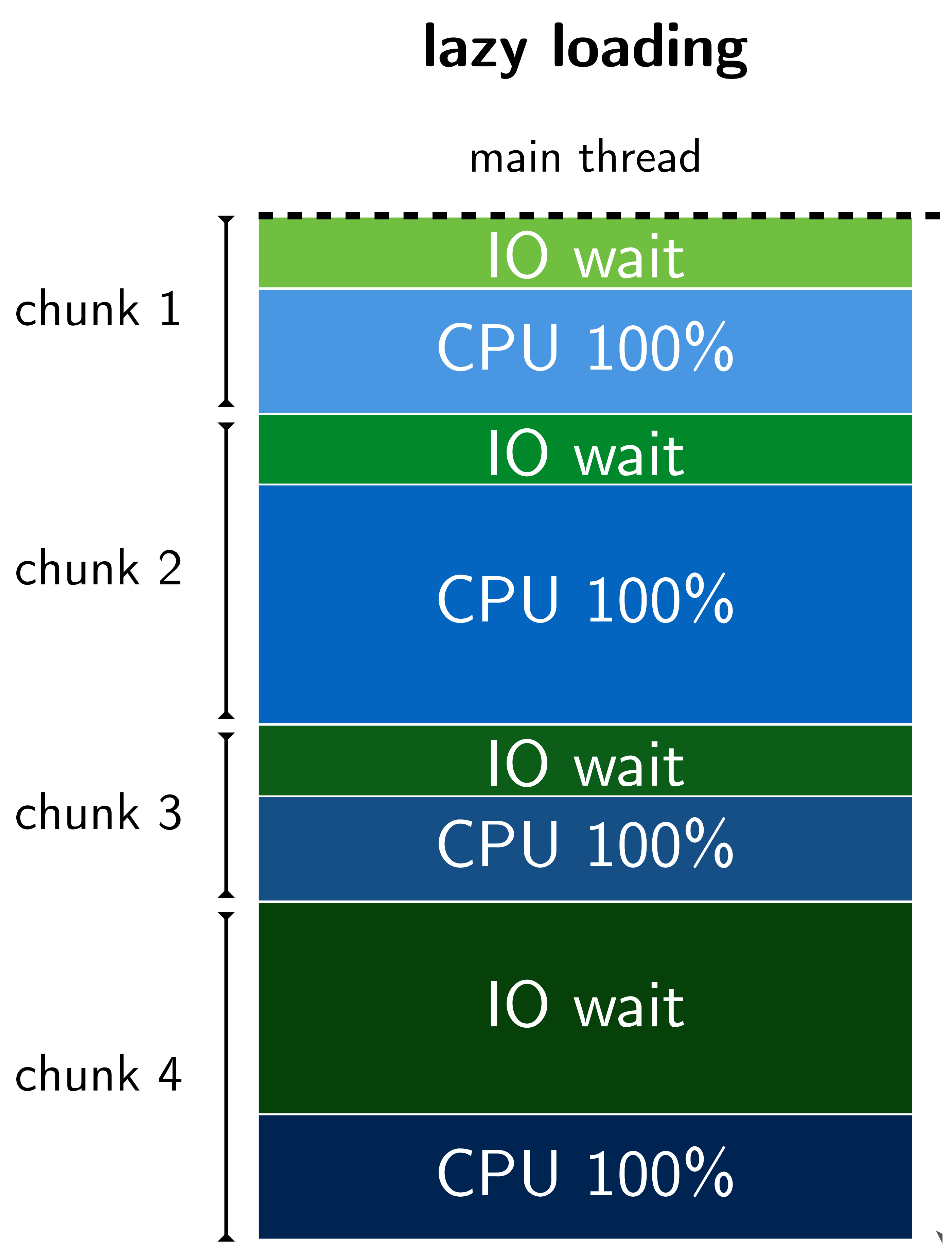
e.g. in SelectEvents



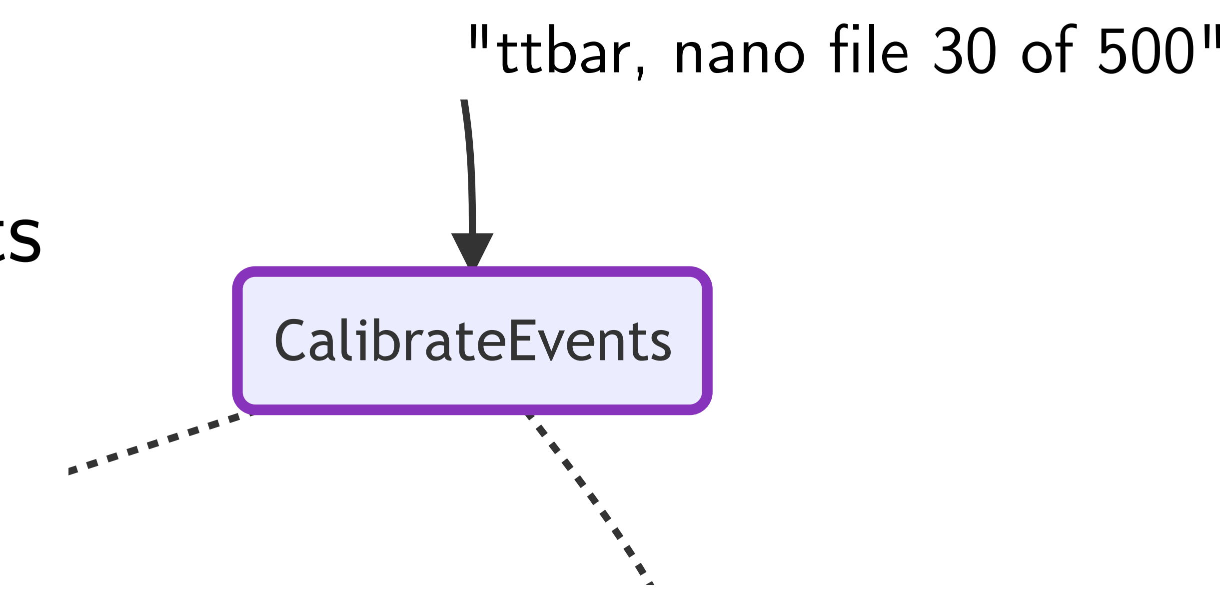
- Each task handles a **single input** in **one* process** (* or more if needed)
  - Single input = potentially multiple files with **different columns** for the **same** events
  - Orchestration allows processing on **any resource**
  - **Highly parallel** when running over **all inputs**



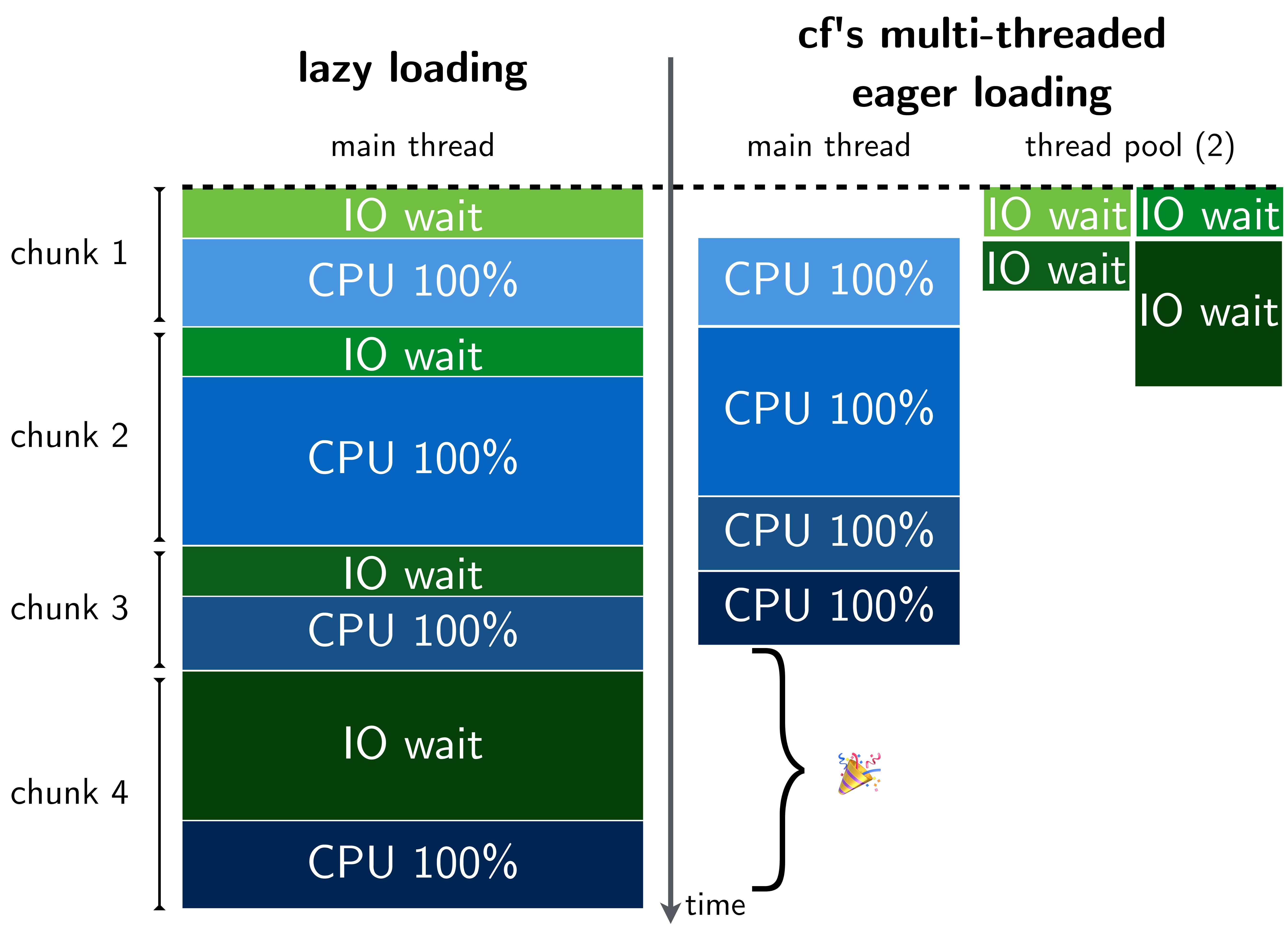
- Loop over event chunks in **single thread**, **offload IO waits** to thread pool



- Each task handles a **single input** in **one* process** (* or more if needed)
  - Single input = potentially multiple files with **different columns** for the **same** events
  - Orchestration allows processing on **any resource**
  - **Highly parallel** when running over **all inputs**



- Loop over event chunks in **single thread**, **offload IO waits** to thread pool



Straight-forward integration of `dask_awkward`
→ Map chunks to partitions
→ `compute()` partitions in thread-pool
→ *Single-node* dask graph
→ Provide result to main thread

"Ideal workflow" in ATTF report

data processing

- ☑ **F1.1** Executable in "one go"
- ☑ **F1.2** Output intermediate results on demand
- ☑ **F1.3** Identify and rerun only necessary components
- ☑ **F1.4** Composition of columns to easy reuse / sharing
- ☑ **F1.5** Reproducibility via CI/CD
- ☑ **F1.6** Version checkpointing
- ☑ **F1.7** Support for custom NANO input

analysis description

- ⊡ **F2.1** Non-imperative paradigm
- ☑ **F2.2** Physics object representation for NANO objects
- ☑ **F2.3** Seamless handling of systematic uncertainties
- ☑ **F2.4** Automatic datacard writing
- ⊡ **F2.5** Analysis results in different formats (datacards, pyhf workspace, HEPData, ...)
- ☐ **F2.6** Export to / import from dedicated, static workflow language
- ☑ **F2.7** Workflow configuration separated from analysis code
- ☑ **F2.8** Multidimensional histograms

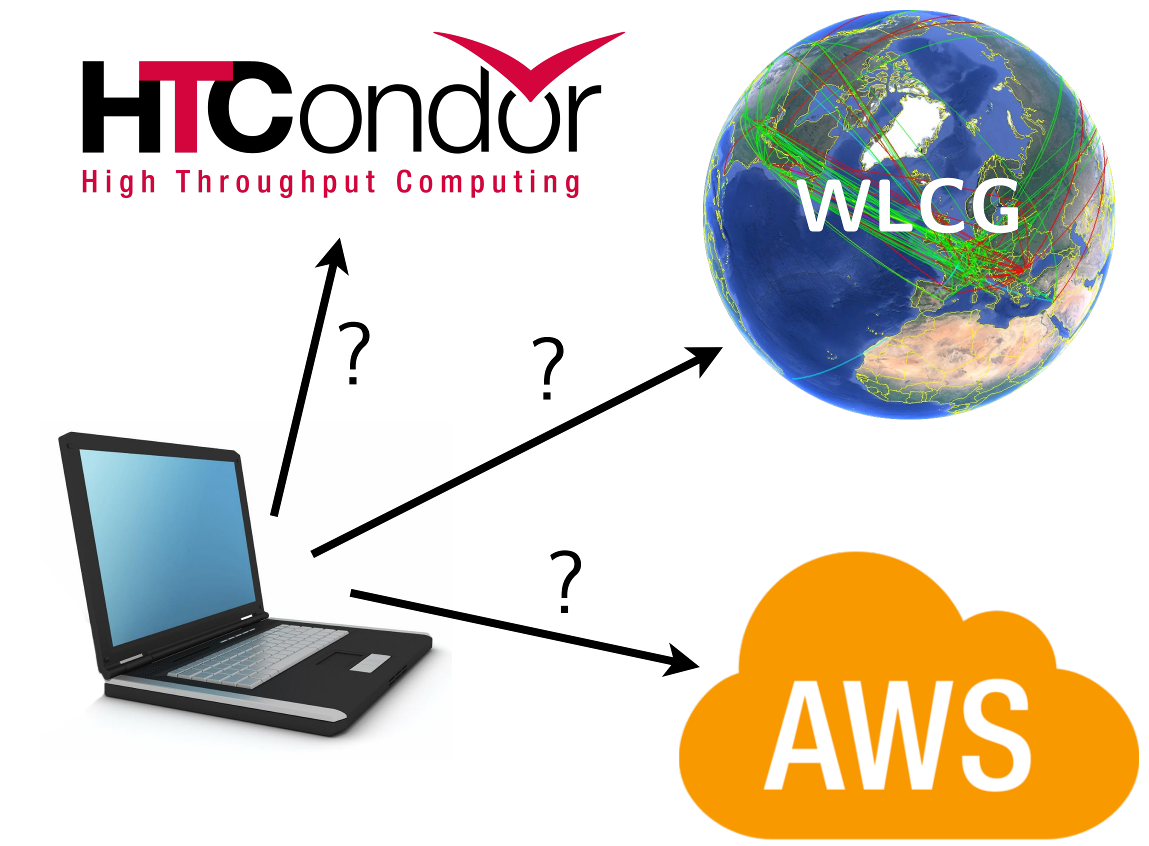
resources

- ☑ **F3.1** Resource agnosticism
- ☑ **F3.2** Easily scalable (local, multi-core, batch)

law & luigi

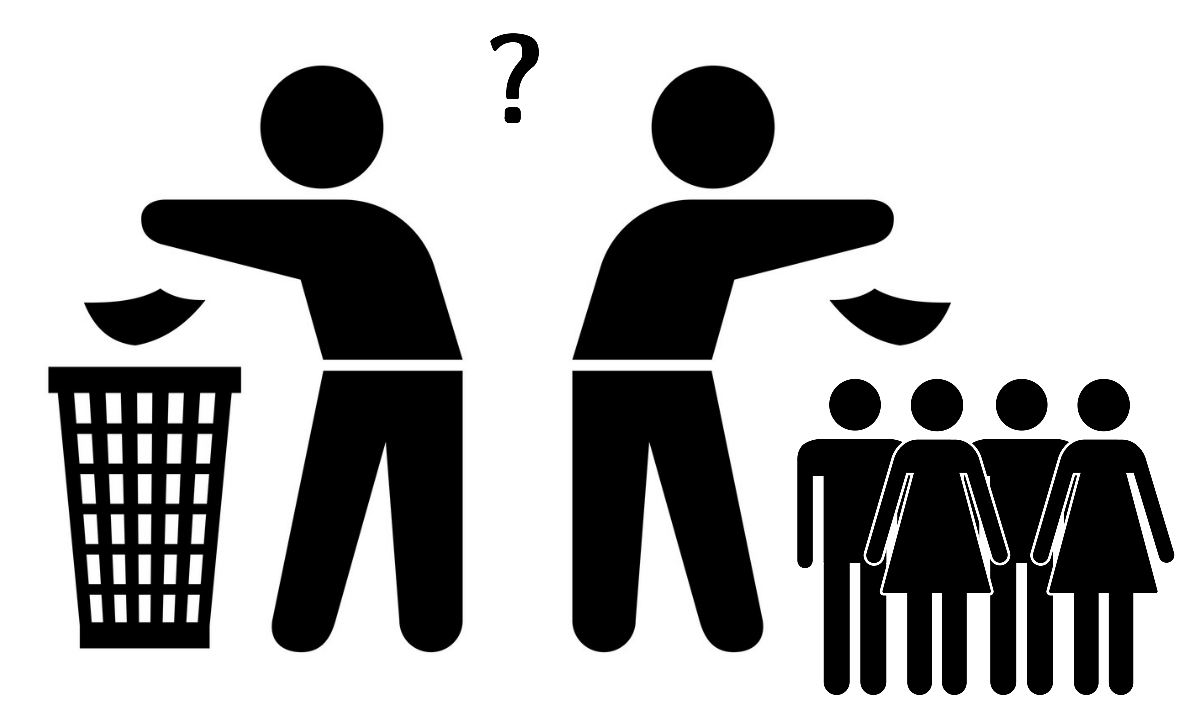
Motivational questions

- **Portability**: Does the analysis depend on ...
  - where it runs?
  - where it stores data?
    - ▻ Execution/storage should **not** dictate code design!

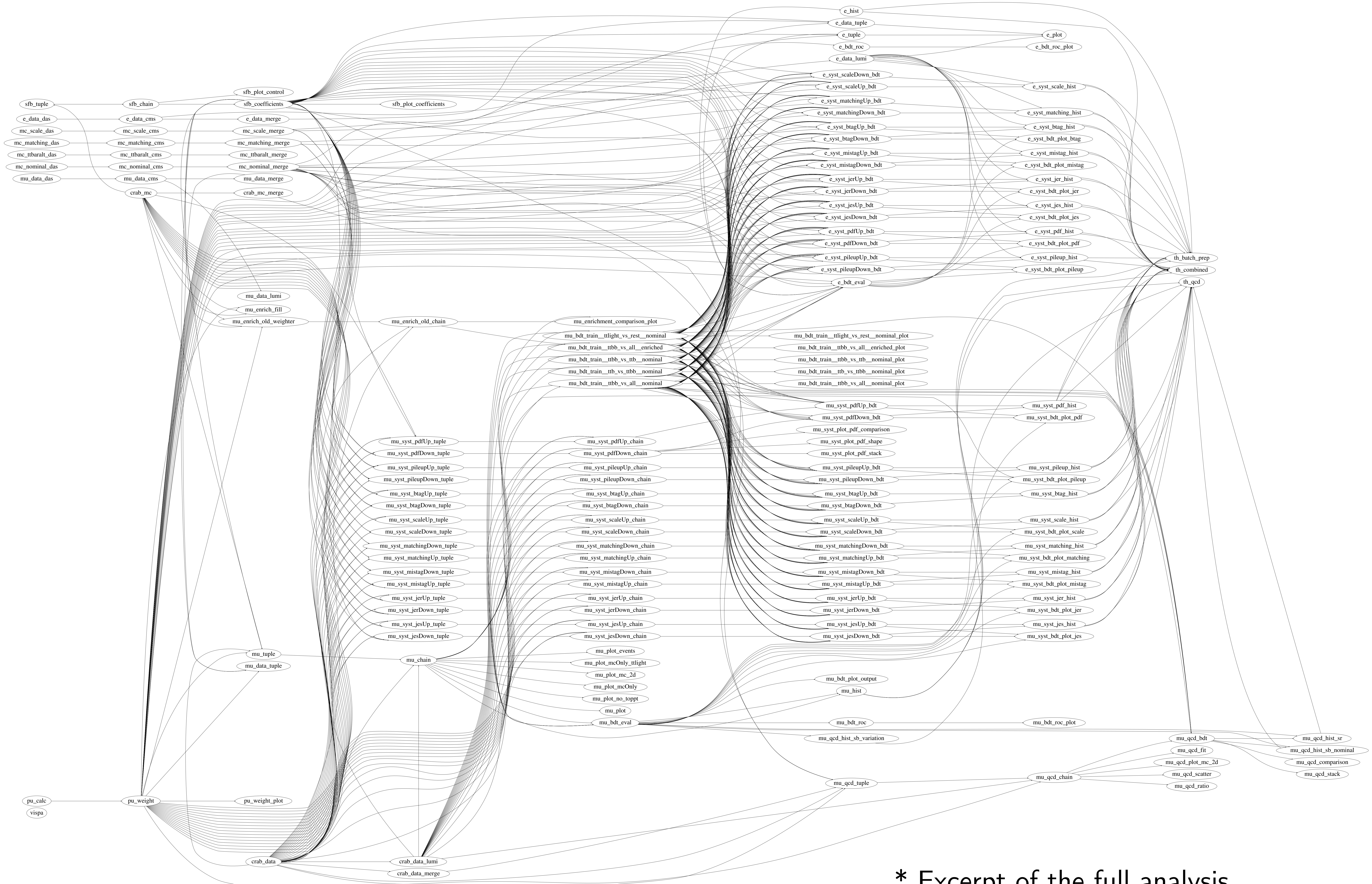


- **Reproducibility**: When a postdoc / PhD student leaves, ...
  - can someone else run the analysis?
  - is there a loss of information? Is a new *framework* required?
    - ▻ Dependencies often **only** exist in the physicists head!

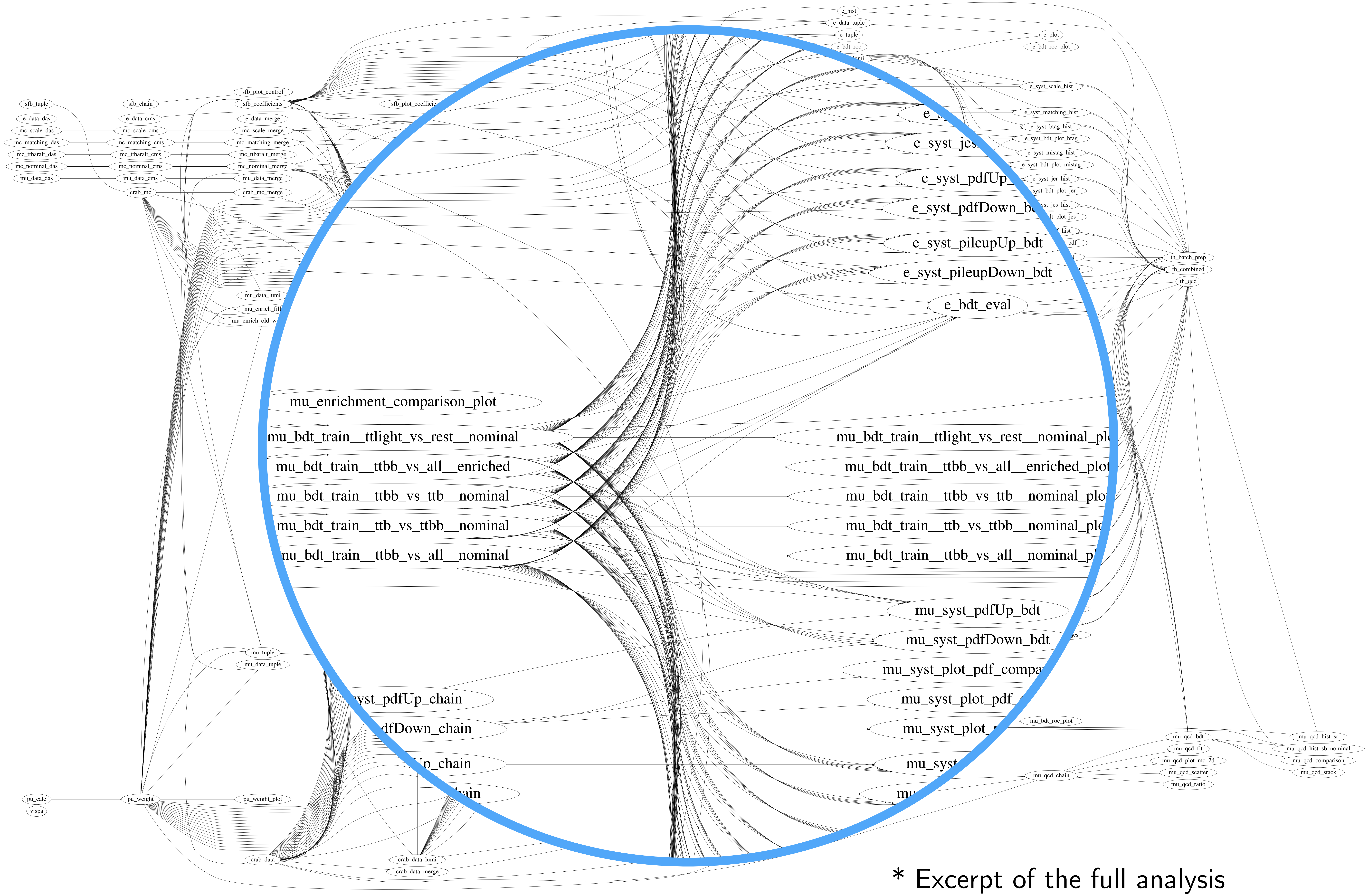
- **Preservation**: After an analysis is published ...
  - are people investing time to preserve their work?
  - can it be repeated after O(years)?
    - ▻ Daily working environment should provide preservation features **out-of-the-box**!



- Personal experience: ⅔ of "analysis" time for technicalities, ⅓ left for physics
  → **Physics output doubled if it were the other way round?**



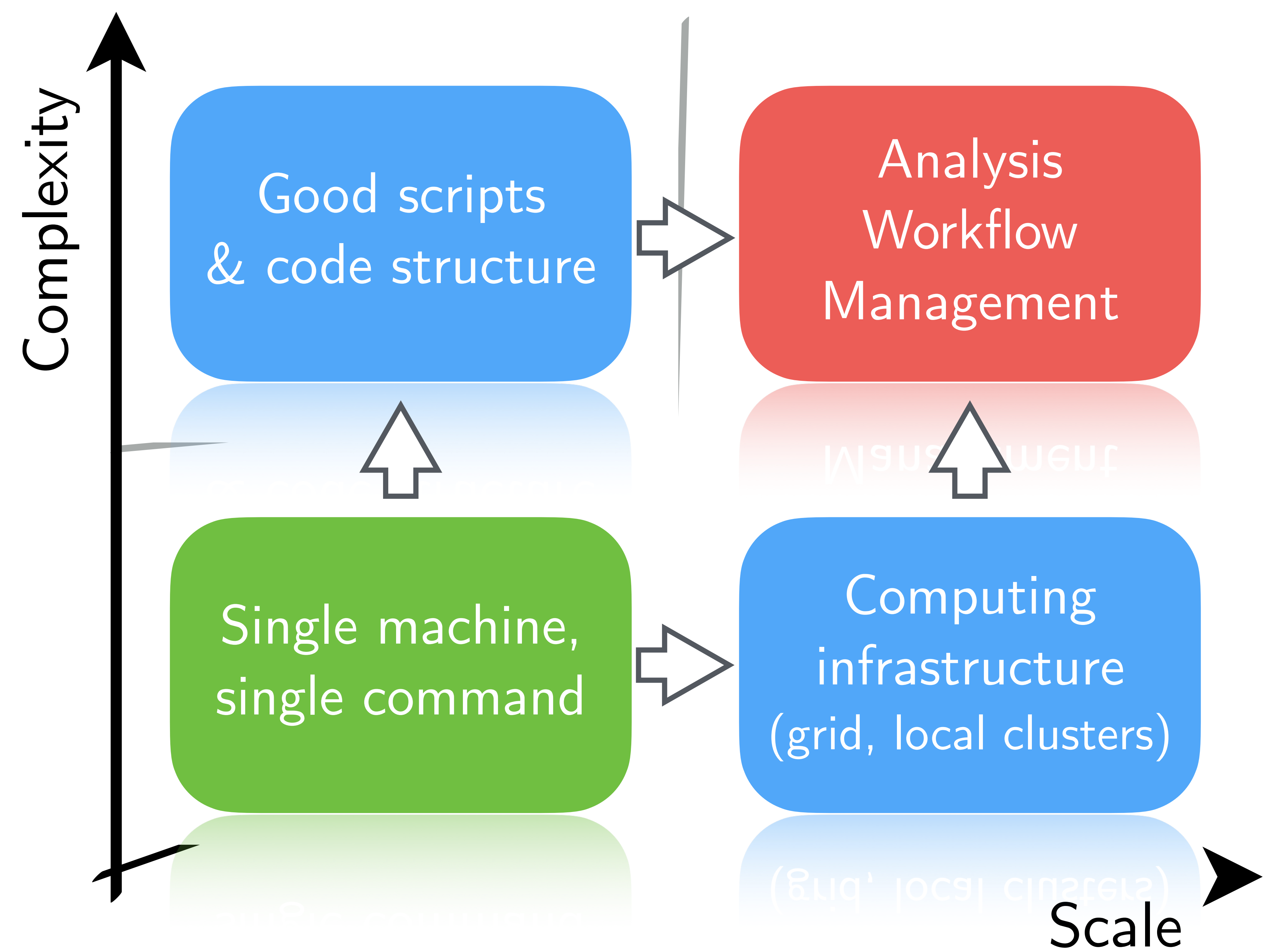
* Excerpt of the full analysis



* Excerpt of the full analysis

Landscape of HEP analyses

- Most analyses are both **large and complex**
  - Structure & requirements between workloads mostly undocumented
  - Manual execution & steering of jobs, bookkeeping of data across SEs, data revisions, …
  - → Error-prone & time-consuming



- **In the following**
  - → Approach **complexity** with luigi
  - → Enabling **large-scale** with law

Tailored systems

- Structure “iterative”, a-priori unknown
- Dynamic workflows, fast R&D cycles
- DAG with arbitrary dependencies
- Incorporate *any* existing infrastructure
- Use custom software, everywhere

Wishlist for end-user analyses

- Structure known in advance
- Workflows static & recurring
- One-dimensional design
- Special production infrastructure
- Homogeneous software requirements

→ Requirements for HEP analyses mostly orthogonal github.com/spotify/luigi

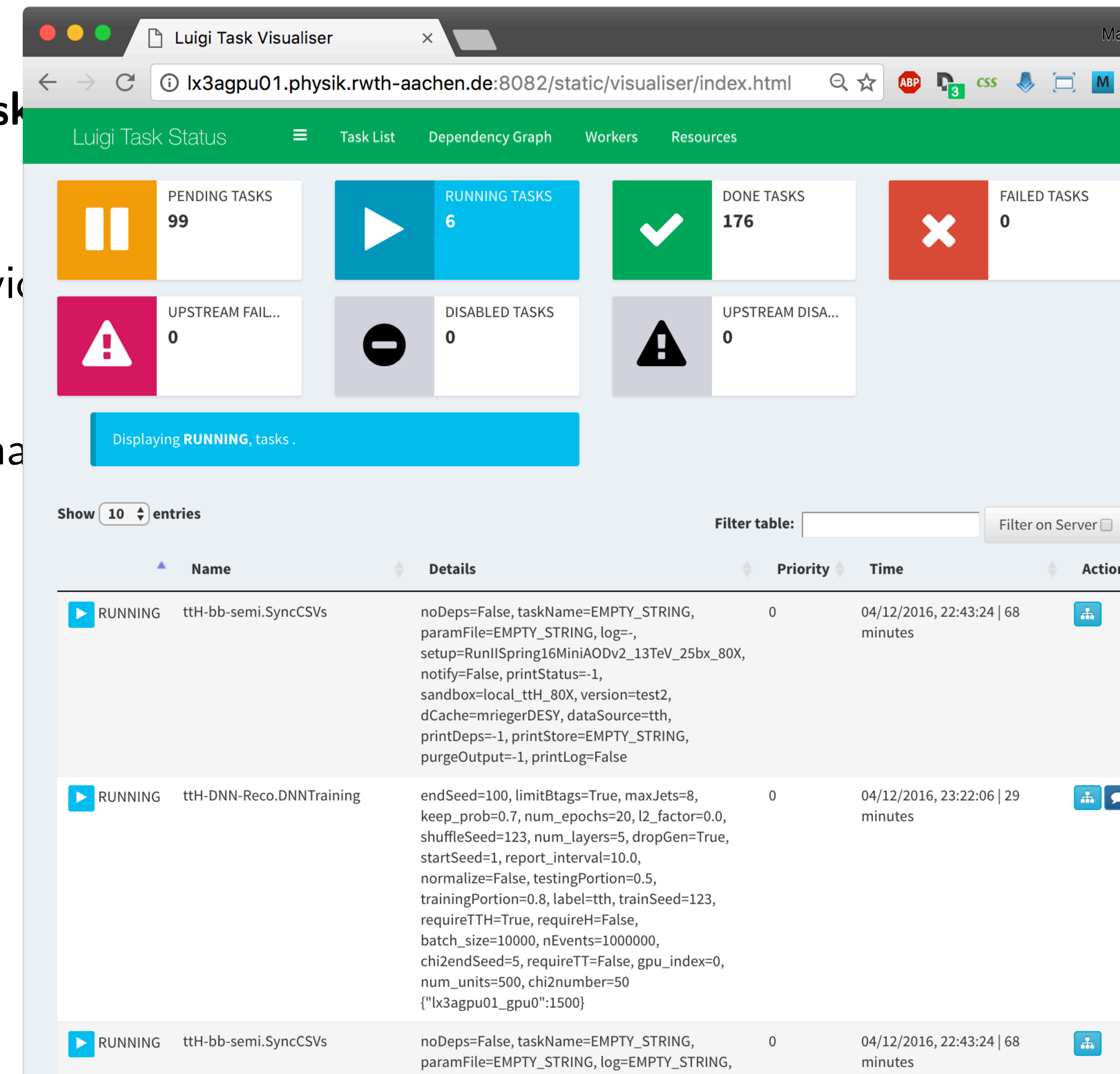
Watch 493 Unstar 15.2k Fork 2.3k

- Python package for building complex pipelines
- Development started at Spotify, now open-source and community-drive

Building blocks

1. Workloads defined as **Task** classes that can **require** other **Task**
2. Tasks produce output **Targets**
3. **Parameters** customize tasks & control runtime behavi

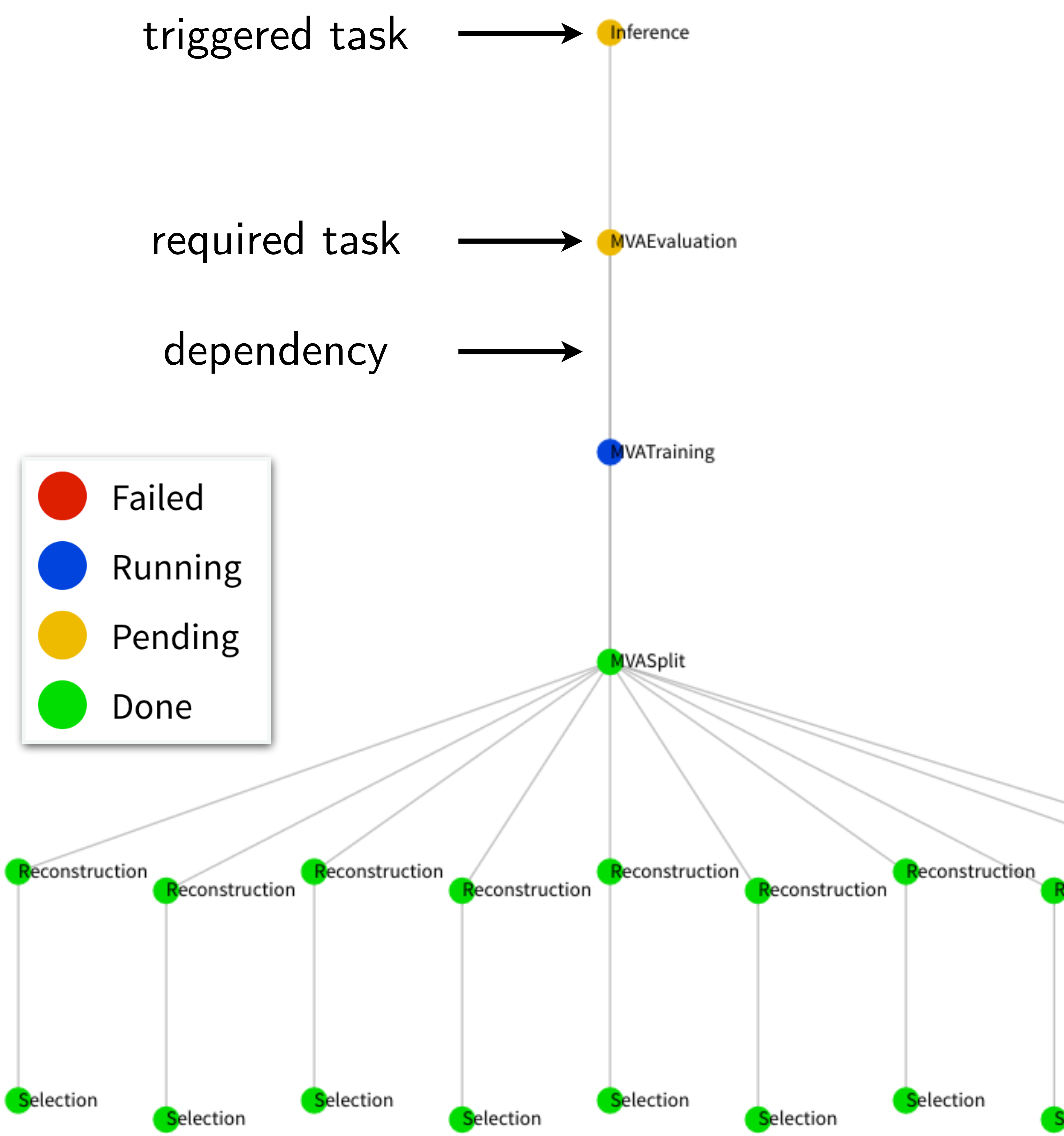
- Web UI with two-way messaging (task → UI, UI → task), automatic error ha command line interface, ...



- Luigi's execution model is make-like

1. Create dependency tree for triggered task
2. Determine tasks to actually run:
   - Walk through tree (top-down)
   - For each path, stop if all output targets of a task

- Only processes what is really necessary
- Scalable through simple structure
- Error handling & automatic re-scheduling



* in this case, the task is considered `complete`

```python
# reco.py

import luigi

from my_analysis.tasks import Selection


class Reconstruction(luigi.Task):

    dataset = luigi.Parameter(default="ttH")


    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return luigi.LocalTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

```
> python reco.py Reconstruction --dataset ttbar
```

```python
# reco.py

import luigi

from my_analysis.tasks import Selection


class Reconstruction(luigi.Task):

    dataset = luigi.Parameter(default="ttH")


    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return luigi.LocalTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

`Parameter` object on class-level

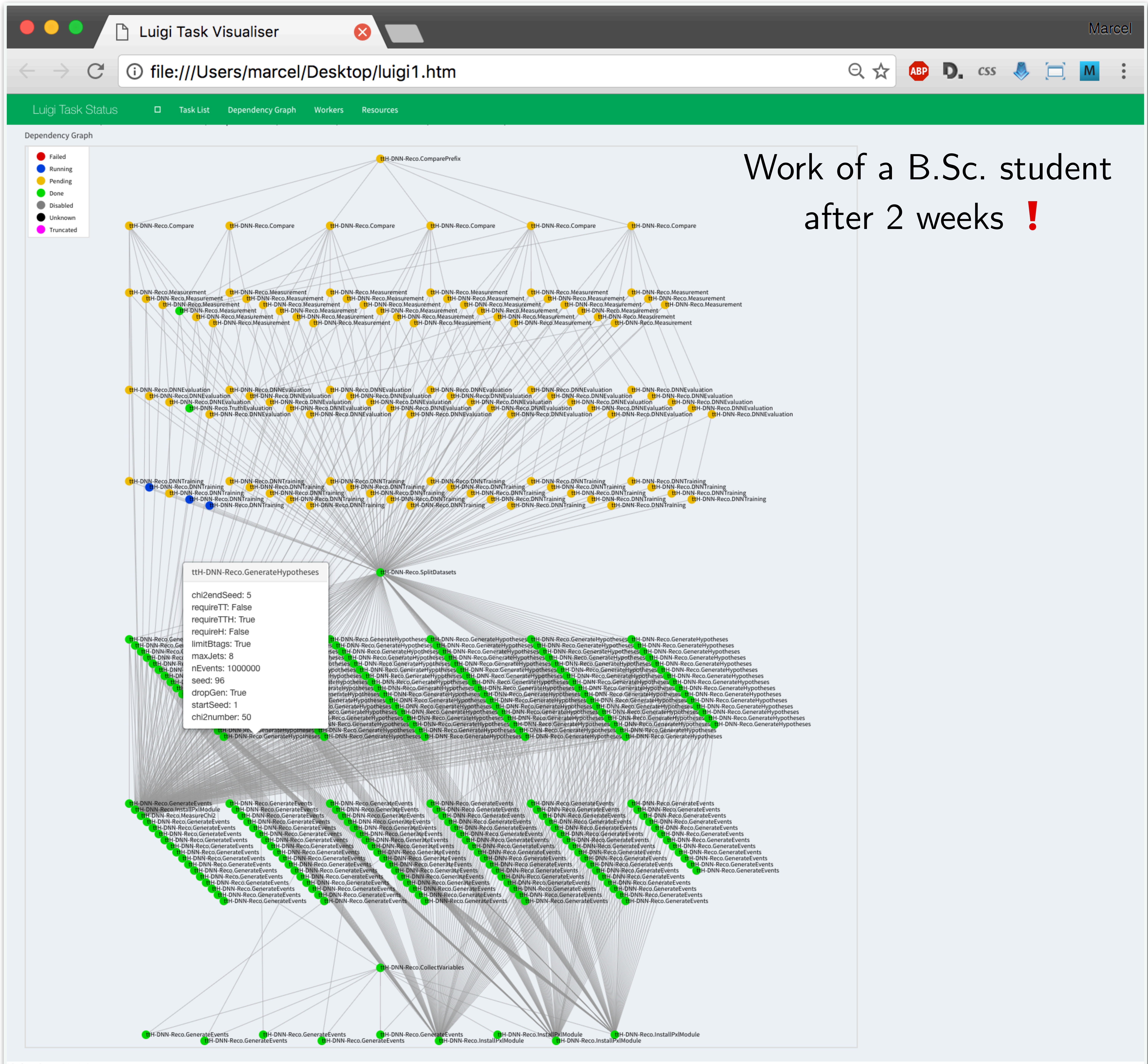
`string` on instance-level

luigi's local file target:
- `path: string`
- `exists(): bool`
- `remove()`
- `open(): fd`
- `...`

Encoding parameters into output target path

```
> python reco.py Reconstruction --dataset ttbar
```

Example dependency trees



Work of a B.Sc. student after 2 weeks **!**

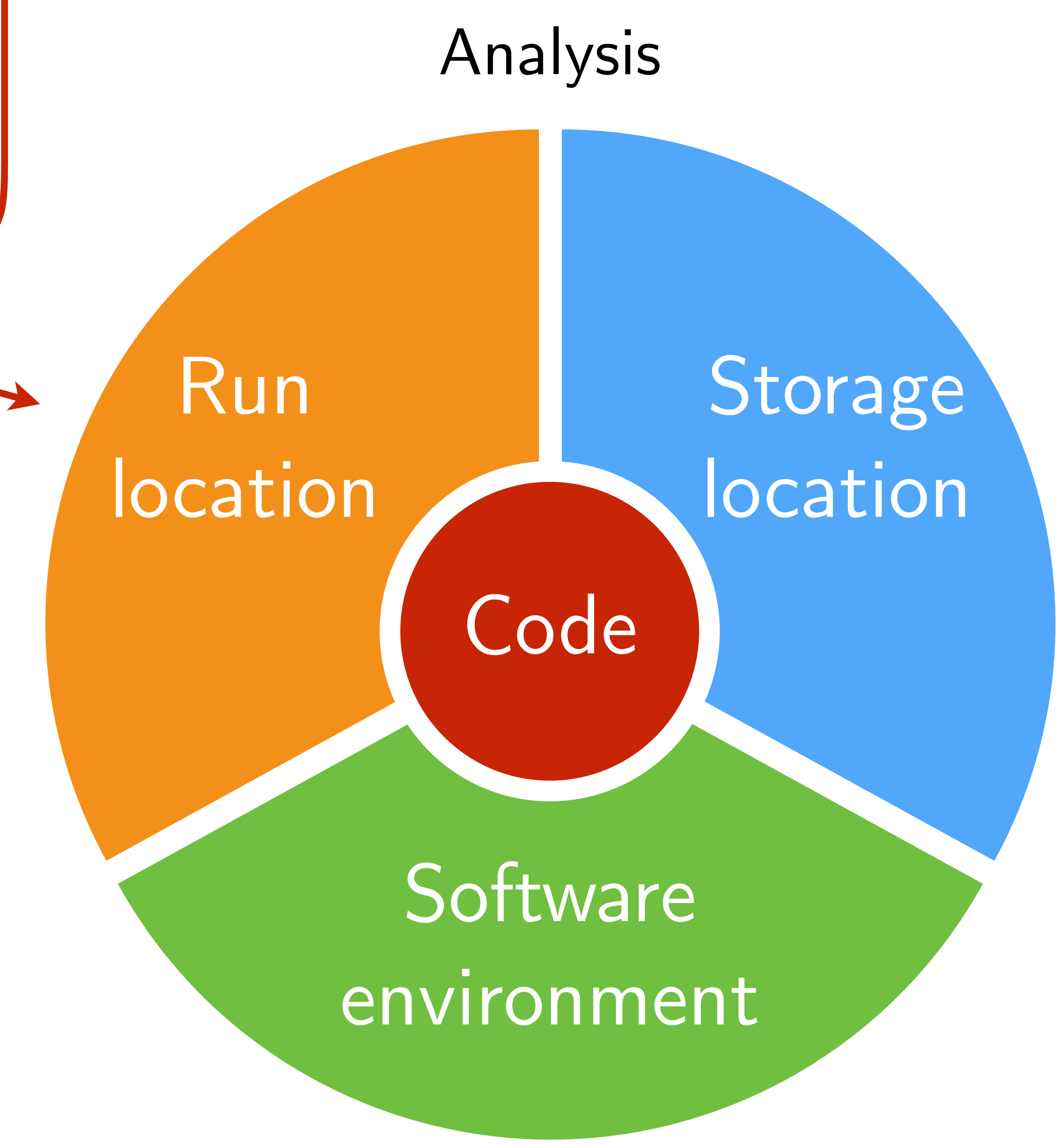
- law: extension **on top** of *luigi* (i.e. it does not replace *luigi*)

- Software design follows 3 primary goals:

1. Experiment-agnostic core (in fact, not even related to physics)
2. Scalability on HEP infrastructure (but not limited to it)
3. Decoupling of **run locations**, **storage locations** & **software environments**
   - Not constrained to specific resources
   - All components interchangeable

- Toolbox to follow an **analysis design pattern**
  - No constraint on language or data structures
  - → Not a *framework*

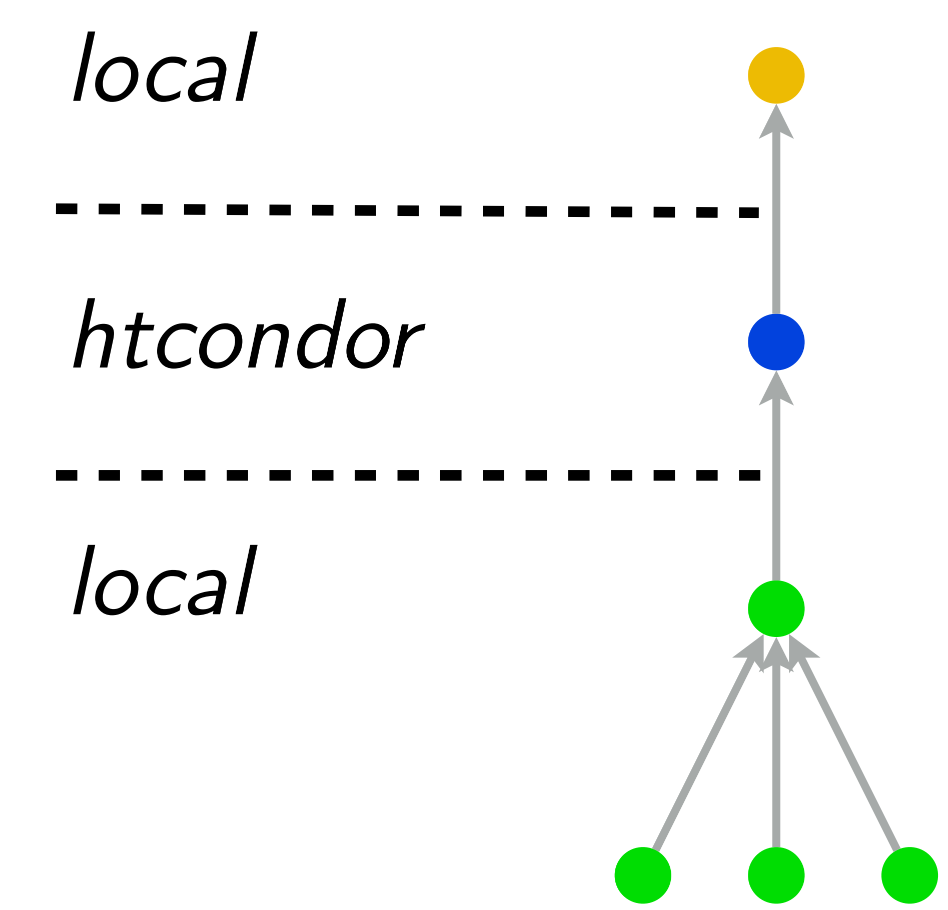
- **Most used** workflow system for analyses in CMS
  - O(20) analyses, O(60-80) people
  - Central groups, e.g. HIG, TAU, BTV



1. **Job submission**

- Idea: submission built into tasks, **no need to write extra code**
- Currently supported job systems: HTCondor, LSF, gLite, ARC, Slurm, CMS-CRAB
- Mandatory features such as automatic resubmission, flexible task ↔ job matching,
  job files fully configurable at submission time, internal job staging when queues are saturated, ...
- From the htcondor_at_cern example:

```
lxplus129:law_test > law run CreateChars --workflow htcondor
INFO: [pid 30564] Worker Worker(host=lxplus129.cern.ch, username=mrieger) running
                  CreateChars(branch=-1, start_branch=0, end_branch=26, version=v1)
going to submit 26 htcondor job(s)
submitted 1/26 job(s)
submitted 26/26 job(s)
14:35:40: all: 26, pending: 26 (+26), running: 0 (+0),   finished: 0 (+0),   retry: 0 (+0), failed: 0 (+0)
...
14:37:10: all: 26, pending: 0 (+0),   running: 26 (+26), finished: 0 (+0),   retry: 0 (+0), failed: 0 (+0)
14:37:40: all: 26, pending: 0 (+0),   running: 10 (-16), finished: 16 (+16), retry: 0 (+0), failed: 0 (+0)
14:38:10: all: 26, pending: 0 (+0),   running: 0  (+0),  finished: 26 (+10), retry: 0 (+0), failed: 0 (+0)
INFO: [pid 30564] Worker Worker(host=lxplus129.cern.ch, username=mrieger) done!

lxplus129:law_test >
```



Job status polling example from CMS HH combination

```
16:04:23: all: 3321, pending: 2821 (+2821), running: 426 (+426), finished: 74 (+74), retry: 0 (+0), failed: 0 (+0)
16:04:37: all: 3321, pending: 2829 (+2829), running: 5 (+5), finished: 487 (+487), retry: 0 (+0), failed: 0 (+0)
16:06:15: all: 3321, pending: 2827 (-2), running: 6 (+1), finished: 488 (+1), retry: 0 (+0), failed: 0 (+0)
16:06:17: all: 3321, pending: 2813 (-8), running: 424 (-2), finished: 84 (+10), retry: 0 (+0), failed: 0 (+0)
16:08:11: all: 3321, pending: 2820 (-7), running: 8 (+2), finished: 493 (+5), retry: 0 (+0), failed: 0 (+0)
16:08:26: all: 3321, pending: 2810 (-3), running: 422 (-2), finished: 89 (+5), retry: 0 (+0), failed: 0 (+0)
16:09:44: all: 3321, pending: 2819 (-1), running: 9 (+1), finished: 493 (+0), retry: 0 (+0), failed: 0 (+0)
16:10:03: all: 3321, pending: 2808 (-2), running: 420 (-2), finished: 93 (+4), retry: 0 (+0), failed: 0 (+0)
16:12:26: all: 3321, pending: 2817 (-2), running: 5 (-4), finished: 499 (+6), retry: 0 (+0), failed: 0 (+0)
16:12:46: all: 3321, pending: 2802 (-6), running: 422 (+2), finished: 97 (+4), retry: 0 (+0), failed: 0 (+0)
16:15:11: all: 3321, pending: 2811 (-6), running: 7 (+2), finished: 503 (+4), retry: 0 (+0), failed: 0 (+0)
16:15:39: all: 3321, pending: 2796 (-6), running: 420 (-2), finished: 105 (+8), retry: 0 (+0), failed: 0 (+0)
16:17:18: all: 3321, pending: 2806 (-5), running: 10 (+3), finished: 505 (+2), retry: 0 (+0), failed: 0 (+0)
16:17:49: all: 3321, pending: 2792 (-4), running: 415 (-5), finished: 114 (+9), retry: 0 (+0), failed: 0 (+0)
16:19:34: all: 3321, pending: 2800 (-6), running: 11 (+1), finished: 510 (+5), retry: 0 (+0), failed: 0 (+0)
16:20:15: all: 3321, pending: 2788 (-4), running: 413 (-2), finished: 120 (+6), retry: 0 (+0), failed: 0 (+0)
16:21:26: all: 3321, pending: 2795 (-5), running: 13 (+2), finished: 513 (+3), retry: 0 (+0), failed: 0 (+0)
16:21:53: all: 3321, pending: 2784 (-4), running: 411 (-2), finished: 126 (+6), retry: 0 (+0), failed: 0 (+0)
16:23:47: all: 3321, pending: 2791 (-4), running: 14 (+1), finished: 516 (+3), retry: 0 (+0), failed: 0 (+0)
16:24:10: all: 3321, pending: 2779 (-5), running: 411 (+0), finished: 131 (+5), retry: 0 (+0), failed: 0 (+0)
16:26:05: all: 3321, pending: 2705 (-86), running: 92 (+78), finished: 524 (+8), retry: 0 (+0), failed: 0 (+0)
16:26:33: all: 3321, pending: 2683 (-96), running: 502 (+91), finished: 136 (+5), retry: 0 (+0), failed: 0 (+0)
16:29:08: all: 3321, pending: 2690 (-15), running: 87 (-5), finished: 544 (+20), retry: 0 (+0), failed: 0 (+0)
16:29:21: all: 3321, pending: 2647 (-36), running: 530 (+28), finished: 144 (+8), retry: 0 (+0), failed: 0 (+0)
16:30:39: all: 3321, pending: 2651 (-39), running: 46 (-41), finished: 624 (+80), retry: 0 (+0), failed: 0 (+0)
16:30:54: all: 3321, pending: 2621 (-26), running: 550 (+20), finished: 150 (+6), retry: 0 (+0), failed: 0 (+0)
16:32:02: all: 3321, pending: 2634 (-17), running: 35 (-11), finished: 652 (+28), retry: 0 (+0), failed: 0 (+0)
16:32:26: all: 3321, pending: 2608 (-13), running: 555 (+5), finished: 158 (+8), retry: 0 (+0), failed: 0 (+0)
16:33:29: all: 3321, pending: 2630 (-4), running: 30 (-5), finished: 661 (+9), retry: 0 (+0), failed: 0 (+0)
16:34:18: all: 3321, pending: 2597 (-11), running: 561 (+6), finished: 163 (+5), retry: 0 (+0), failed: 0 (+0)
16:35:16: all: 3321, pending: 2621 (-9), running: 26 (-4), finished: 674 (+13), retry: 0 (+0), failed: 0 (+0)
16:36:06: all: 3321, pending: 2586 (-11), running: 560 (-1), finished: 175 (+12), retry: 0 (+0), failed: 0 (+0)
16:37:39: all: 3321, pending: 2612 (-9), running: 23 (-3), finished: 686 (+12), retry: 0 (+0), failed: 0 (+0)
16:39:19: all: 3321, pending: 2577 (-9), running: 559 (-1), finished: 185 (+10), retry: 0 (+0), failed: 0 (+0)
16:39:32: all: 3321, pending: 2603 (-9), running: 19 (-4), finished: 699 (+13), retry: 0 (+0), failed: 0 (+0)
16:41:04: all: 3321, pending: 2566 (-11), running: 556 (-3), finished: 199 (+14), retry: 0 (+0), failed: 0 (+0)
16:41:25: all: 3321, pending: 2593 (-10), running: 23 (+4), finished: 705 (+6), retry: 0 (+0), failed: 0 (+0)
```

2. Remote targets

luigi analysis workflow

- Idea: work with remote files **as if they were local**
- Remote targets built on top of GFAL2 Python bindings
  - ▻ Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - ▻ API **identical** to local targets
  - ! Actual remote interface **interchangeable** (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, **local caching** (backup), configurable protocols, round-robin, ...

"FileSystem" configuration

```
# law.cfg

[wlcg_fs]
base: root://eosuser.cern.ch/eos/user/m/mrieger

...
```

- Base path prefixed to all paths using this "fs"
- Configurable per file operation (stat, listdir, ...)
- Protected against removal of parent directories

2. Remote targets

- Idea: work with remote files **as if they were local**
- Remote targets built on top of GFAL2 Python bindings
  - ▻ Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - ▻ API **identical** to local targets
  - ! Actual remote interface **interchangeable** (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, **local caching** (backup), configurable protocols, round-robin, ...

Conveniently reading remote files

```python
# read a remote json file
target = law.WLCGFileTarget("/file.json", fs="wlcg_fs")

with target.open("r") as f:
    data = json.load(f)
```

2. Remote targets

- Idea: work with remote files **as if they were local**
- Remote targets built on top of GFAL2 Python bindings
  - Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - API **identical** to local targets
  - ! Actual remote interface **interchangeable** (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, **local caching** (backup), configurable protocols, round-robin, ...

Conveniently reading remote files

```python
# read a remote json file
target = law.WLCGFileTarget("/file.json", fs="wlcg_fs")

# use convenience methods for common operations
data = target.load(formatter="json")
```

2. Remote targets

- Idea: work with remote files **as if they were local**
- Remote targets built on top of GFAL2 Python bindings
  - Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - API **identical** to local targets
  - ! Actual remote interface **interchangeable** (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, **local caching** (backup), configurable protocols, round-robin, ...

Conveniently reading remote files

```
# same for root files with context guard
target = law.WLCGFileTarget("/file.root", fs="wlcg_fs")

with target.load(formatter="root") as tfile:
    tfile.ls()
```

2. Remote targets

- Idea: work with remote files **as if they were local**
- Remote targets built on top of GFAL2 Python bindings
  - ▻ Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - ▻ API **identical** to local targets
  - ! Actual remote interface **interchangeable** (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, **local caching** (backup), configurable protocols, round-robin, ...

Conveniently reading remote files

```python
# multiple other "formatters" available
target = law.WLCGFileTarget("/model.pb", fs="wlcg_fs")

graph = target.load(formatter="tensorflow")
session = tf.Session(graph=graph)
```

2. Remote targets

- Idea: work with remote files **as if they were local**
- Remote targets built on top of GFAL2 Python bindings
  - ▻ Supports all WLCG protocols (XRootD, WebDAV, GridFTP, dCache, SRM, ...) + DropBox
  - ▻ API **identical** to local targets
  - **!** Actual remote interface **interchangeable** (GFAL2 is just a good default, fsspec integration easily possible)
- Mandatory features: automatic retries, **local caching** (backup), configurable protocols, round-robin, ...

```python
def run(self):
    # get the input to this task, which is a *.gz file
    # (the output of the requirements)
    inp = self.input()

    # create the correction set
    import correctionlib
    correction_set = correctionlib.CorrectionSet.from_string(
        inp.load(formatter="gzip"),
    )

    ...
```

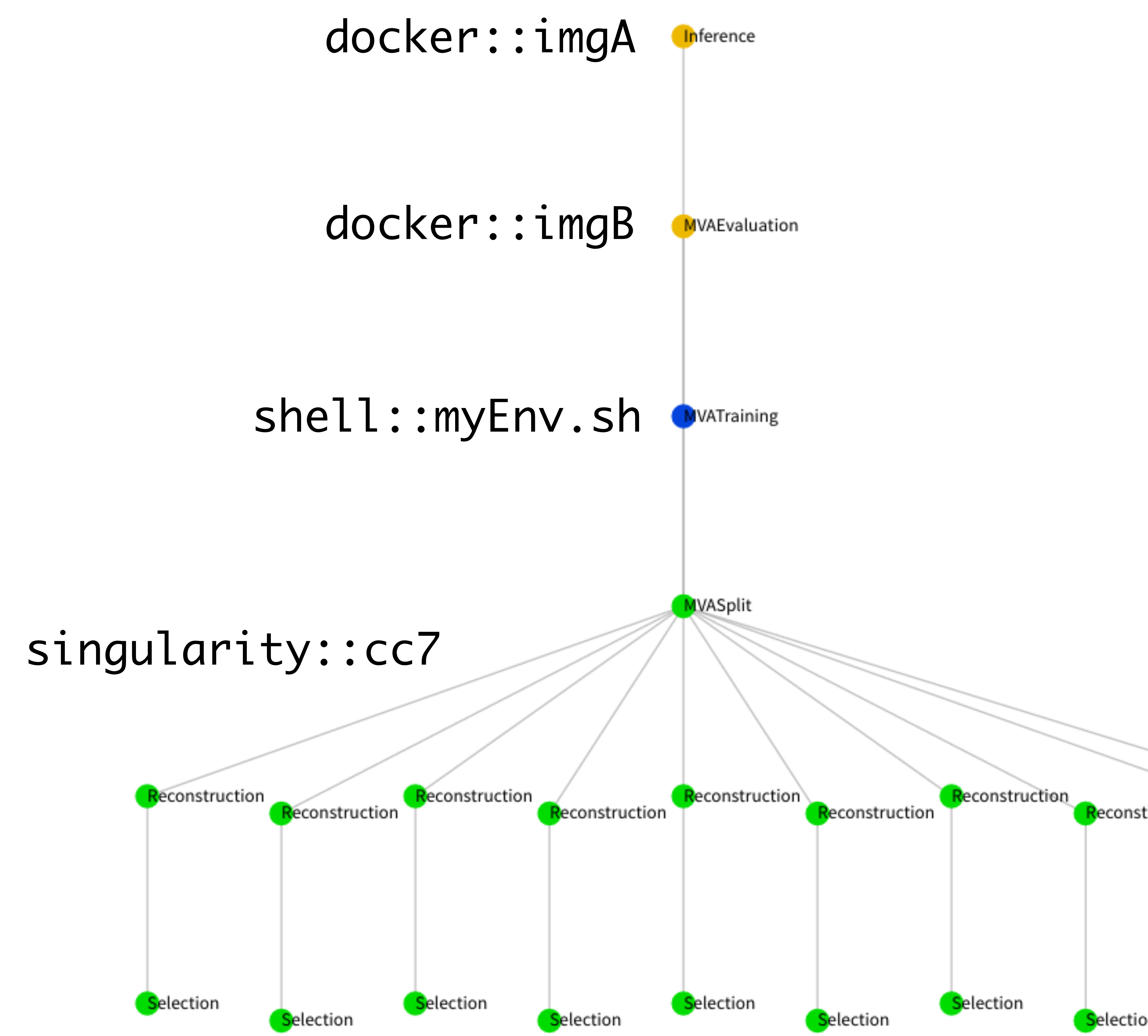
downloads the file if it is remote → *puts it into the local cache for later use* → *opens the file and decompresses the content*

3. Environment sandboxing

- Diverging software requirements between typical workloads is a great feature / challenge / problem

- Introduce sandboxing:
  - Run entire task in **different environment**

- Existing sandbox implementations:
  - Sub-shell with init file (e.g. for CMSSW)
  - Virtual envs
  - Docker images
  - Singularity images



```python
# reco.py

import luigi

from my_analysis.tasks import Selection


class Reconstruction(luigi.Task):

    dataset = luigi.Parameter(default="ttH")


    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return luigi.LocalTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

- [x] luigi task
- [ ] law task
- [ ] Run on HTCondor
- [ ] Store on EOS
- [ ] Run in docker

Example ☞

```
> python reco.py Reconstruction --dataset ttbar
```

```python
# reco.py

import luigi
import law
from my_analysis.tasks import Selection


class Reconstruction(law.Task):

    dataset = luigi.Parameter(default="ttH")


    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return law.LocalFileTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

- ☑ luigi task
- ☑ law task
- ☐ Run on HTCondor
- ☐ Store on EOS
- ☐ Run in docker

Example ☞

```
> law run Reconstruction --dataset ttbar
```

```python
# reco.py

import luigi
import law
from my_analysis.tasks import Selection


class Reconstruction(law.Task, law.HTCondorWorkflow):

    dataset = luigi.Parameter(default="ttH")


    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return law.LocalFileTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

- ☑ luigi task
- ☑ law task
- ☑ Run on HTCondor
- ☐ Store on EOS
- ☐ Run in docker

Example ☞

```
> law run Reconstruction --dataset ttbar --workflow htcondor
```

```python
# reco.py

import luigi
import law
from my_analysis.tasks import Selection


class Reconstruction(law.Task, law.HTCondorWorkflow):

    dataset = luigi.Parameter(default="ttH")


    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return law.WLCGFileTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

- ☑ luigi task
- ☑ law task
- ☑ Run on HTCondor
- ☑ Store on EOS
- ☐ Run in docker

Example ☞

```
> law run Reconstruction --dataset ttbar --workflow htcondor
```

```python
# reco.py

import luigi
import law
from my_analysis.tasks import Selection


class Reconstruction(law.SandboxTask, law.HTCondorWorkflow):

    dataset = luigi.Parameter(default="ttH")
    sandbox = "docker::cern/cc7-base"

    def requires(self):
        return Selection(dataset=self.dataset)

    def output(self):
        return law.WLCGFileTarget(f"reco_{self.dataset}.root")

    def run(self):
        inp = self.input()  # output() of requirements
        outp = self.output()

        # perform reco on file described by "inp" and produce "outp"
        ...
```

- ✔ luigi task
- ✔ law task
- ✔ Run on HTCondor
- ✔ Store on EOS
- ✔ Run in docker

Example ☞

```
> law run Reconstruction --dataset ttbar --workflow htcondor
```

- **CLI**

  ```
  > law run Reconstruction --dataset ttbar --workflow htcondor
  ```

  - Full auto-completion of tasks and parameters

- **Scripting**
  - Mix task completeness checks, job execution & input/output retrieval with custom scripts
  - Easy interface to existing tasks for prototyping

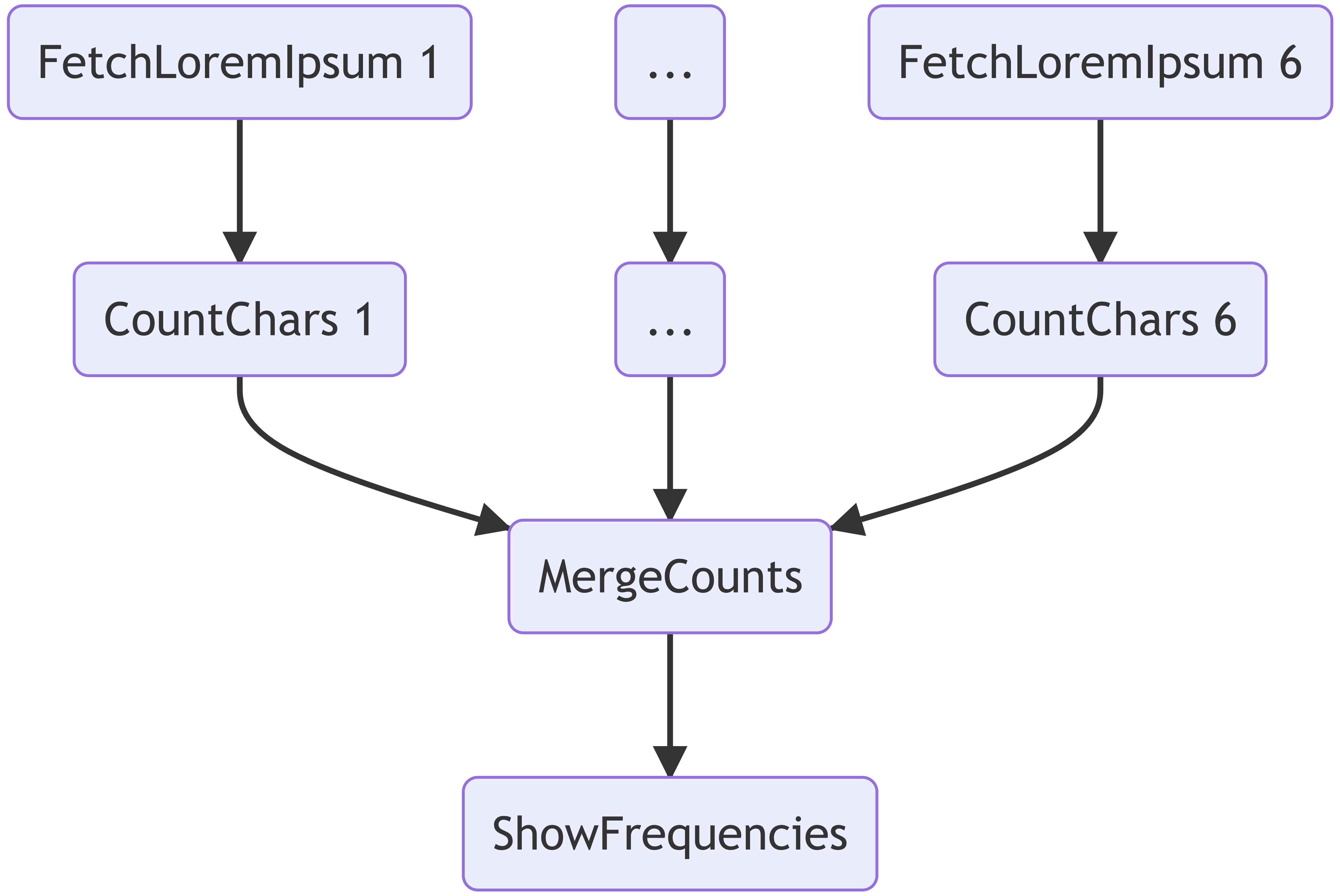
```python
from analysis.tasks import Selection
import akward as ak

# create the task and ensure it's complete
task = Selection(dataset="ttH_bb", version="v3", shift="nominal")
task.law_run()

# read the selected events (a .parquet file)
events = task.output().load(formatter="awkward")

# get the number of jets per event
n_jets = ak.num(events.Jet, axis=1)
print(n_jets)
```

- **Notebooks**

```
In [5]: %law run ShowFrequencies --print-status -1

print task status with max_depth -1 and target_depth 0

0 > ShowFrequencies(slow=False)
│
└──1 > MergeCounts(slow=False)
   │      LocalFileTarget(fs=local_fs, path=$DATA_PATH/chars_merged.json)
   │        existent
   │
   ├──2 > CountChars(file_index=1, slow=False)
   │  │      LocalFileTarget(fs=local_fs, path=$DATA_PATH/chars_1.json)
   │  │        existent
   │  │
   │  └──3 > FetchLoremIpsum(file_index=1, slow=False)
   │            LocalFileTarget(fs=local_fs, path=$DATA_PATH/loremipsum_1.txt)
   │              existent
   │
```

launch binder

- Print character frequencies in the "loremipsum" placeholder text (from examples/loremipsum)
  - Fetch 6 paragraphs as txt files from some server
  - Count character frequencies and save them in json
  - Merge into a single json file
  - Print frequencies

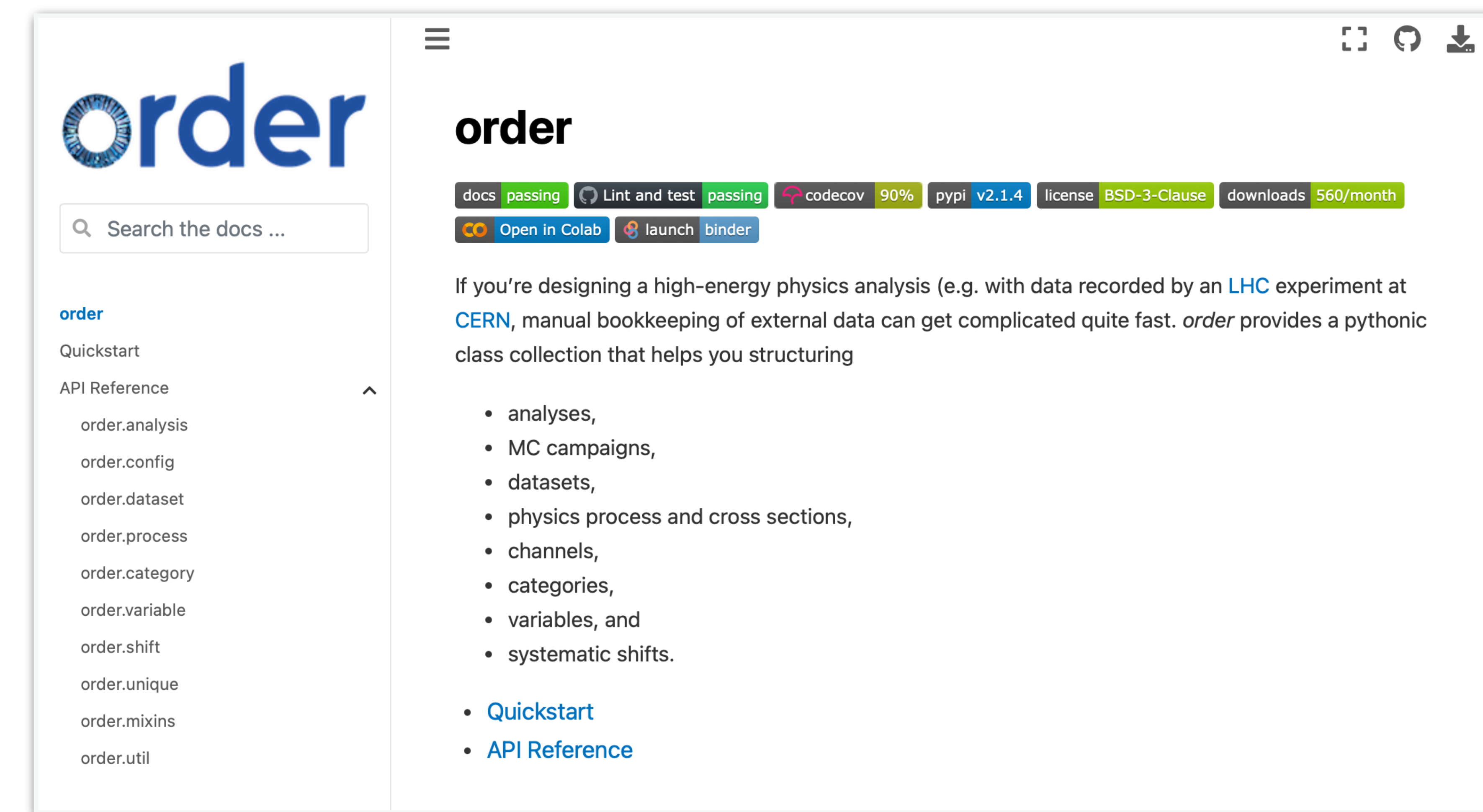


- Sowing CLI usage in the following, but launch binder for the notebook version order

- Pythonic class collection to help structuring CMS metadata
- Provides **programmatic access to** and **relations between various entities**

Name	Purpose
`Analysis`	Represents the central object of a physics analysis.
`Campaign`	Provides data of a well-defined range of data-taking, detector alignment, MC settings, datasets, etc.
`Config`	Holds analysis information related to a campaign instance (most configuration happens here!).
`Dataset`	Definition of a dataset, produced for / measured in a campaign.
`Process`	Phyiscs process with cross sections for multiple center-of-mass energies, labels, etc.
`Channel`	Analysis channel, often defined by a particular decay resulting in distinct final state objects.
`Category`	Category definition, (optionally) within the phase-space of an analysis channel.
`Variable`	Generic variable description providing expression and selection statements, titles, binning, etc.
`Shift`	Represents a systematic shift with a name, direction and type.

documentation



- **Examples**

```
In [3]: dataset_ttH.get_process("ttH").get_xsec(ecm=13)
```

Out[3]: $0.5071\ ^{+0.0294118}_{-0.0466532}$ (scale)

```
In [12]: cfg.get_variable("jet1_px").get_full_title(root=True)
Out[12]: 'jet1_px;Leading jet p_{x} / GeV;Entries / 20.0 GeV'
```

- Heavily used throughout **columnflow**, common objects (datasets and cross-sections) centralized in /uhh-cms/cmsdb
- **Note**: Moving code-base to CMS-wide project via CAT group, datasets & cross-sections to be managed centrally 🎉