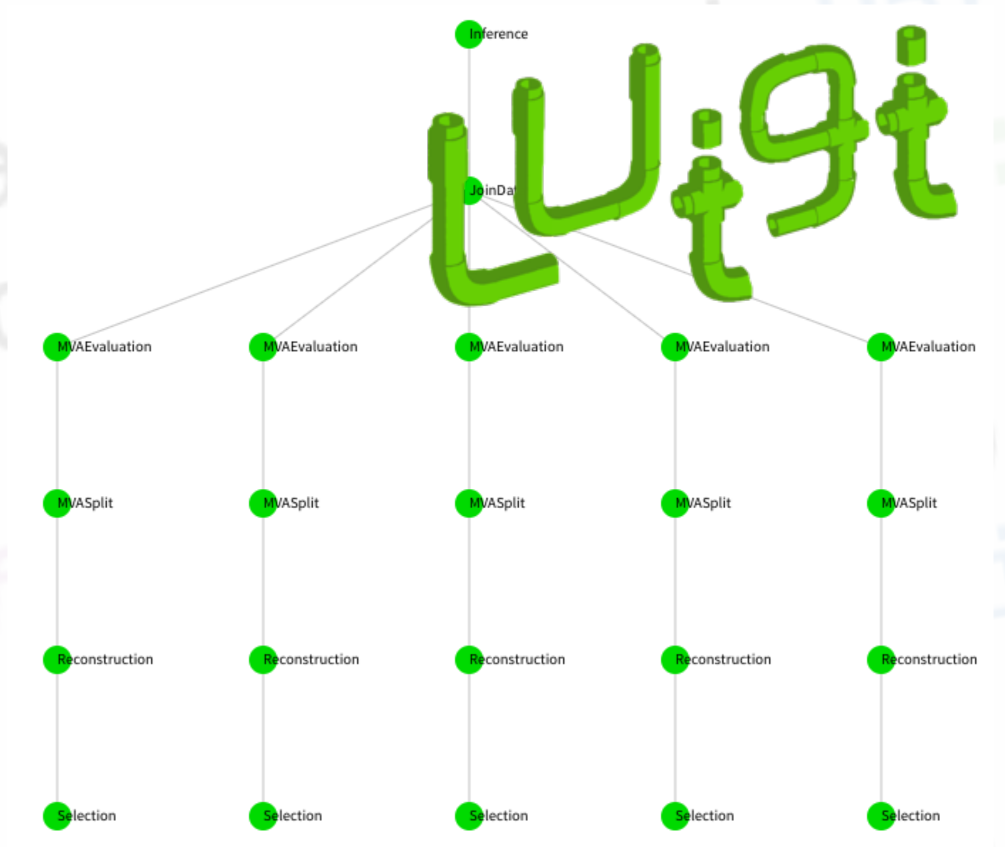


law

luigi analysis workflow

SPONSORED BY THE

UHH Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

CLUSTER OF EXCELLENCE
QUANTUM UNIVERSE

FSP CMS
Erforschung von
Universum und Materie

Federal Ministry
of Education
and Research

luigi analysis workflow

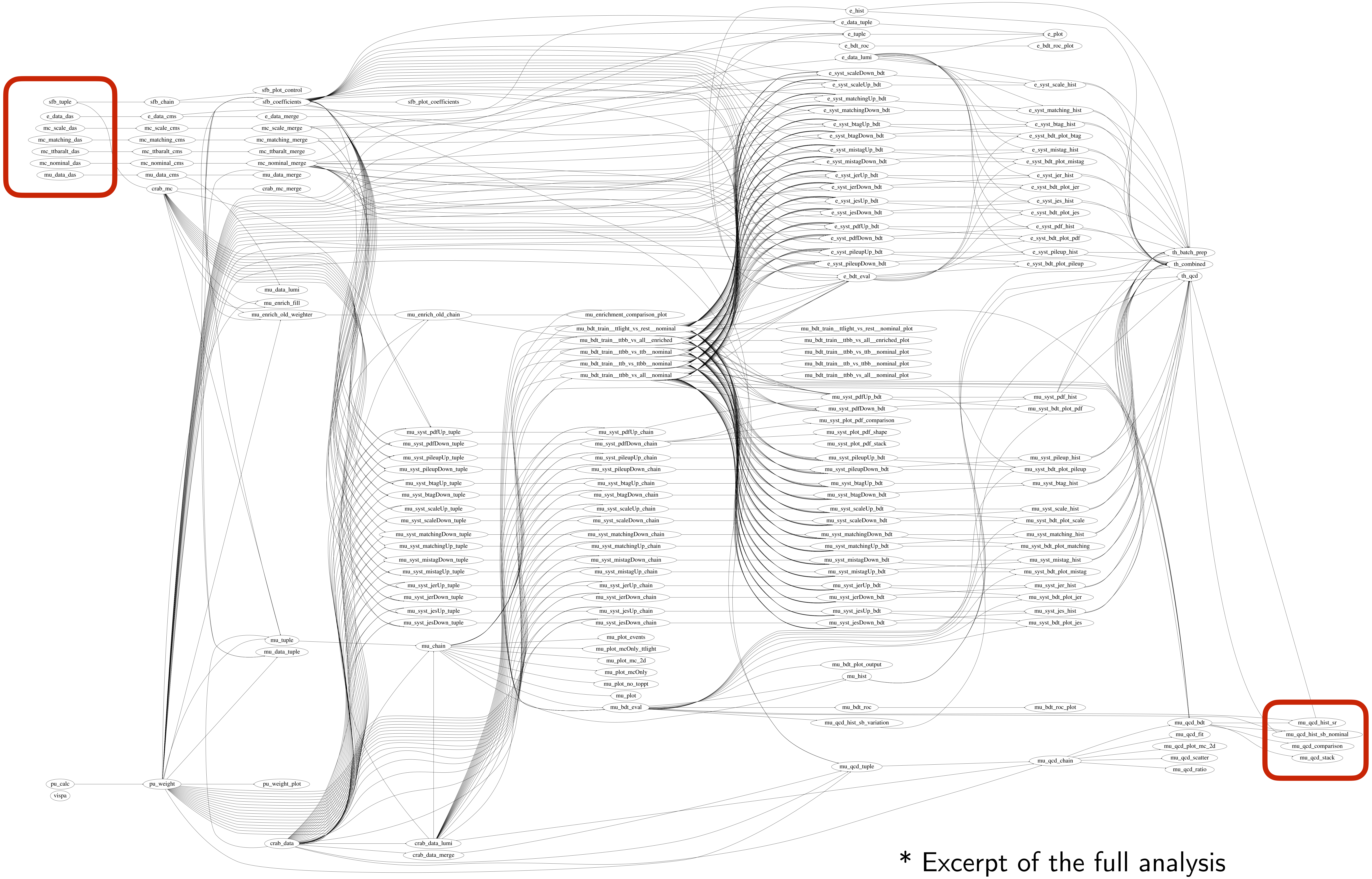
Marcel Rieger

2nd PyHEP.dev Workshop

28.8.2024

Example: ttbb measurement visualization

Entry points

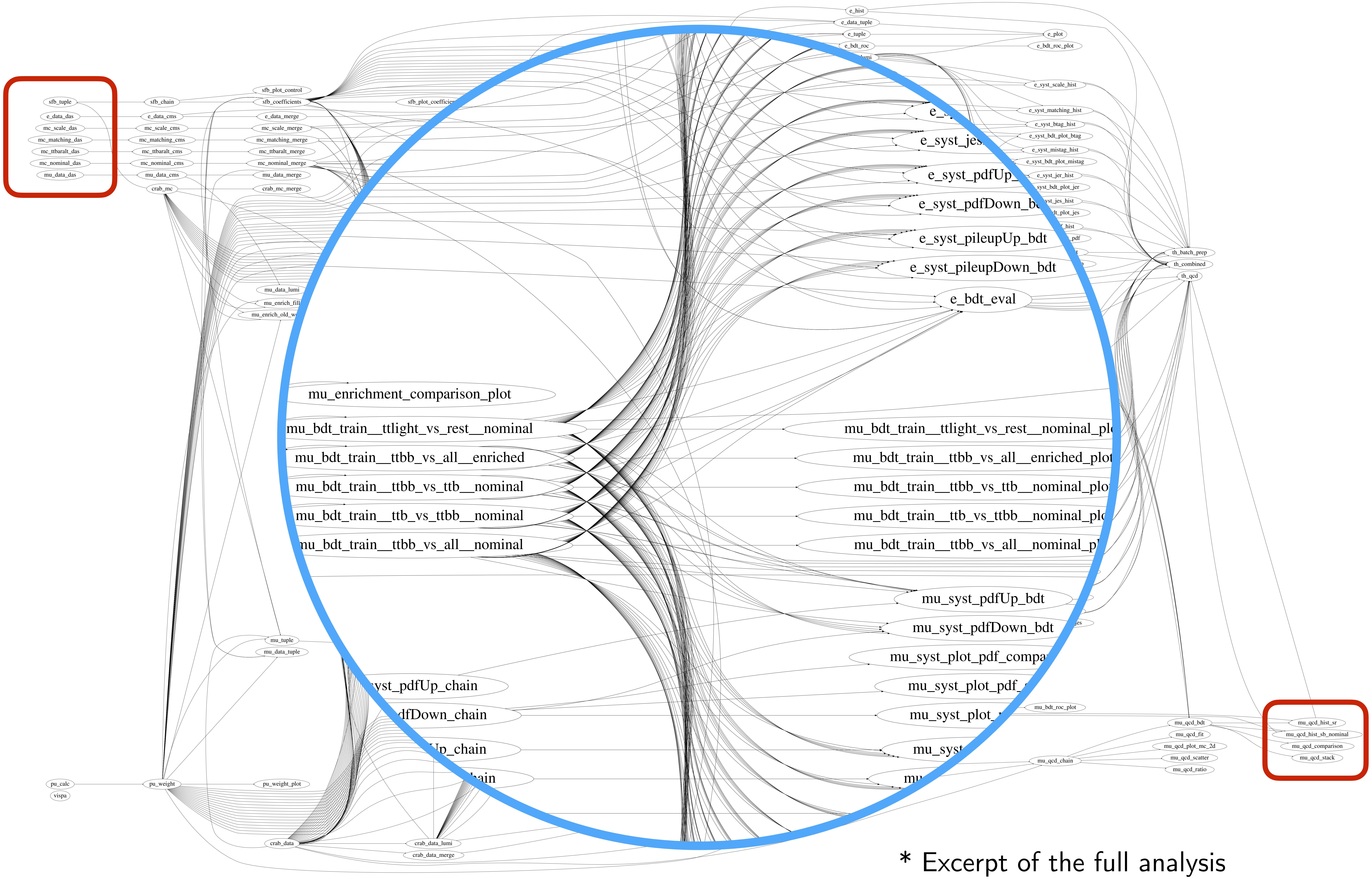


Results

* Excerpt of the full analysis

Example: ttbb measurement visualization

Entry points



Results

* Excerpt of the full analysis

Example: Analysis Grand Challenge (with ML)

Task

Result

Entry point

Data access

Selection

Separate events into subsamples and calculate training feature set

Model fitting and hyperparameter optimization

Select best models for inference

Columnar data

Event selection

Obtain BDT Input Features and Labels

Training

Model Storage

Intermediate result

Dependency

Modularity & interfaces

Declarative configuration

Entry point

Columnar data

Event selection, systematic uncertainties

Obtain BDT Input Features

Machine Learning Inference

Histograms

Statistical model building

Workspace

Statistical analysis

Result, diagnostics

Result

Data access

Selection & systematic uncertainties

Inference as a service

Post-processing, e.g. smoothing

From workspaces to likelihoods

Fit results and diagnostics

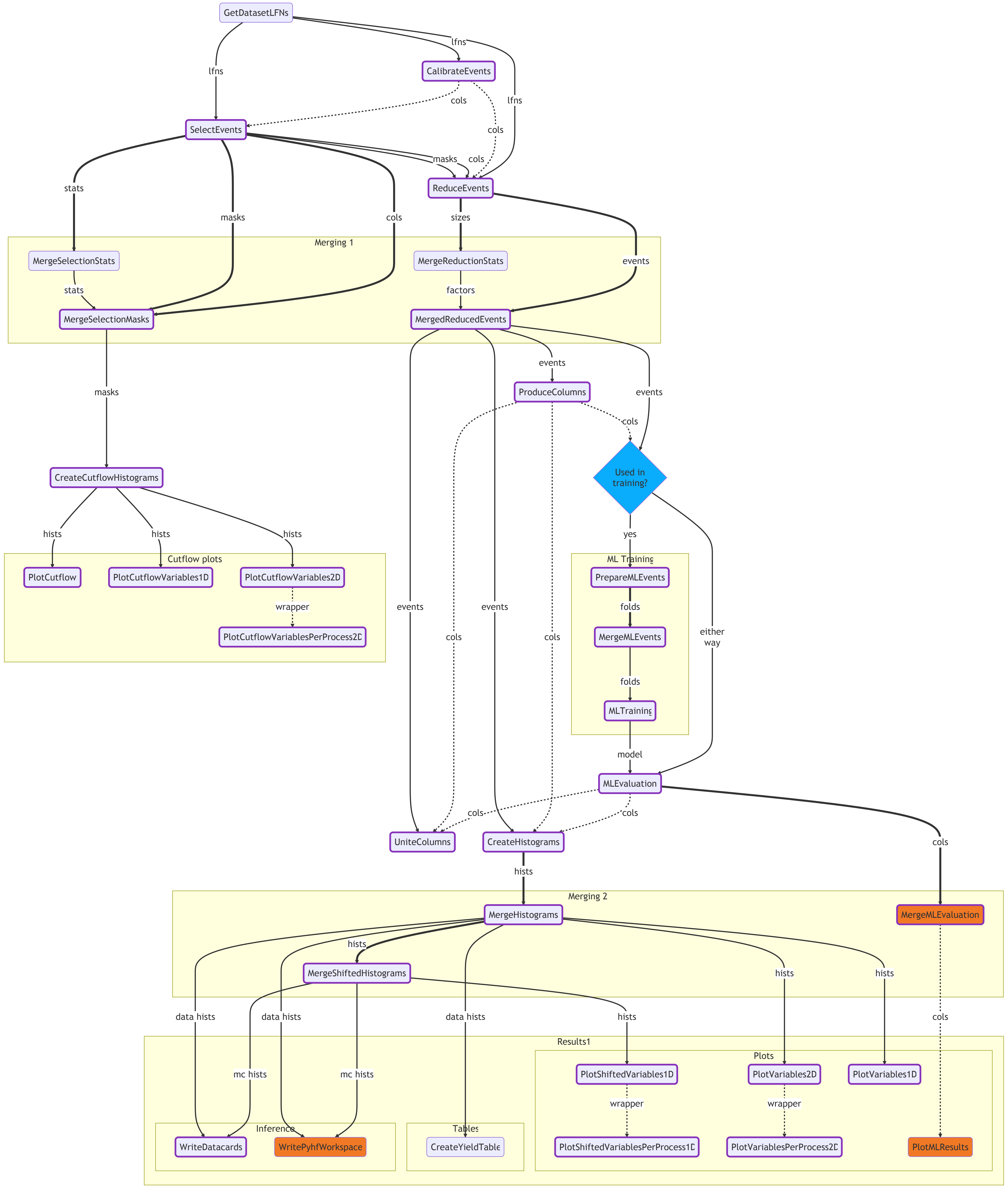
Reusability and preservation

Example: Fully orchestrated LHC Run 2 + 3 analysis with columnflow

Event processing

workflow

Plots & inference



Source

Luigi

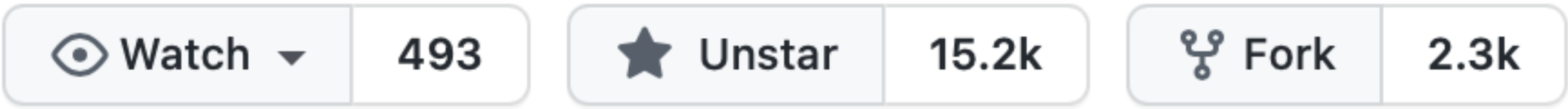
- Python package for building complex pipelines
- Development started at Spotify, now open-source and community-driven

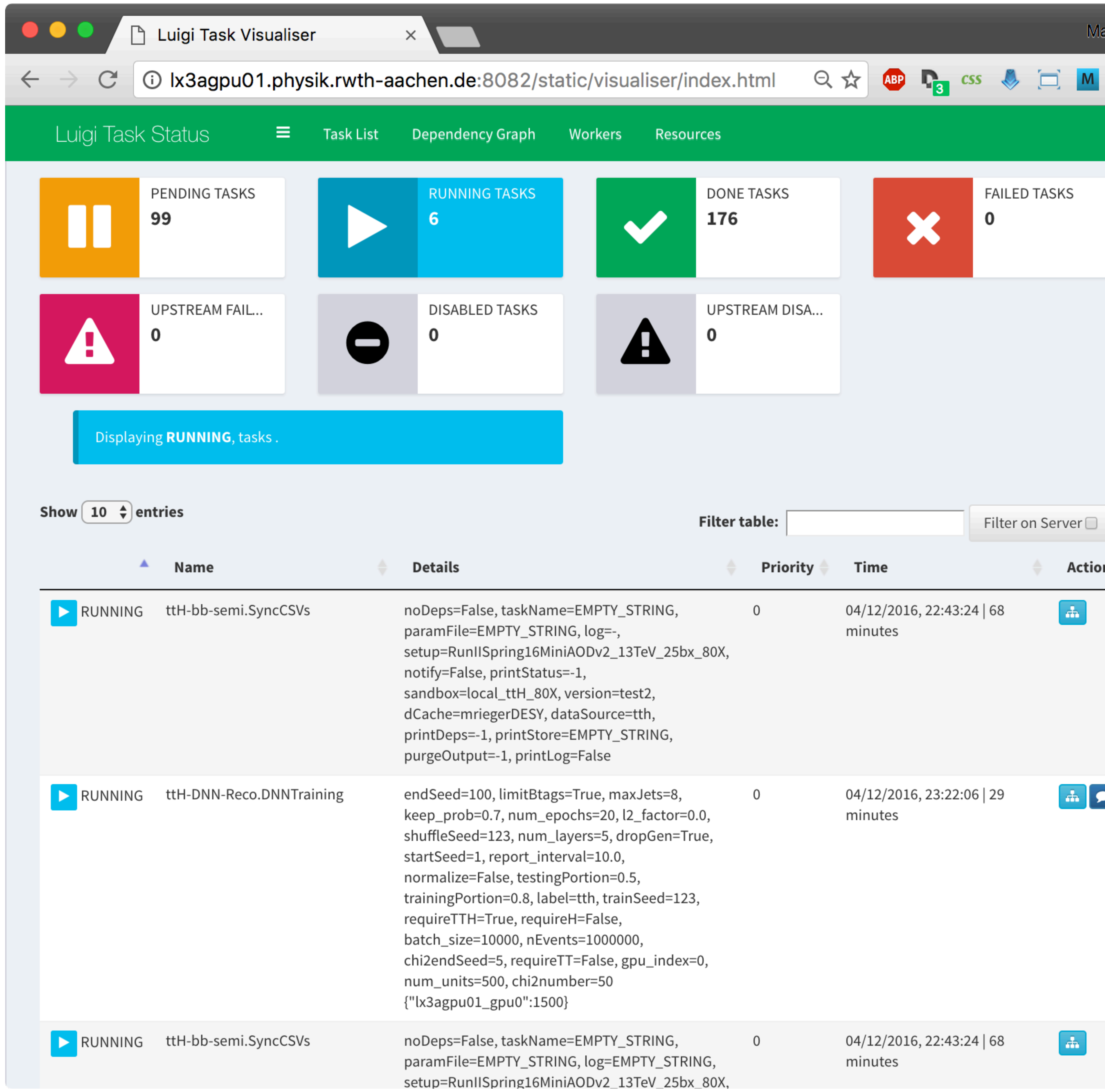
Building blocks

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

- Web UI with two-way messaging (task → UI, UI → task), automatic error handling, task history browser, collaborative features, command line interface, …
- Great documentation 📖

github.com/spotify/luigi





PENDING TASKS 99

RUNNING TASKS 6

DONE TASKS 176

FAILED TASKS 0

UPSTREAM FAIL... 0

DISABLED TASKS 0

UPSTREAM DISA... 0

Displaying **RUNNING**, tasks .

Show 10 entries

Filter table: Filter on Server □

	Name	Details	Priority	Time	Actio
▶ RUNNING	ttH-bb-semi.SyncCSVs	noDeps=False, taskName=EMPTY_STRING, paramFile=EMPTY_STRING, log=-, setup=RunIISpring16MiniAODv2_13TeV_25bx_80X, notify=False, printStatus=-1, sandbox=local_ttH_80X, version=test2, dCache=mriegerDESY, dataSource=tth, printDeps=-1, printStore=EMPTY_STRING, purgeOutput=-1, printLog=False	0	04/12/2016, 22:43:24 | 68 minutes	

▶ RUNNING	ttH-DNN-Reco.DNNTraining	endSeed=100, limitBtags=True, maxJets=8, keep_prob=0.7, num_epochs=20, l2_factor=0.0, shuffleSeed=123, num_layers=5, dropGen=True, startSeed=1, report_interval=10.0, normalize=False, testingPortion=0.5, trainingPortion=0.8, label=tth, trainSeed=123, requireTTH=True, requireH=False, batch_size=10000, nEvents=1000000, chi2endSeed=5, requireTT=False, gpu_index=0, num_units=500, chi2number=50
		{"lx3agpu01_gpu0":1500}
			0	04/12/2016, 23:22:06 | 29 minutes	

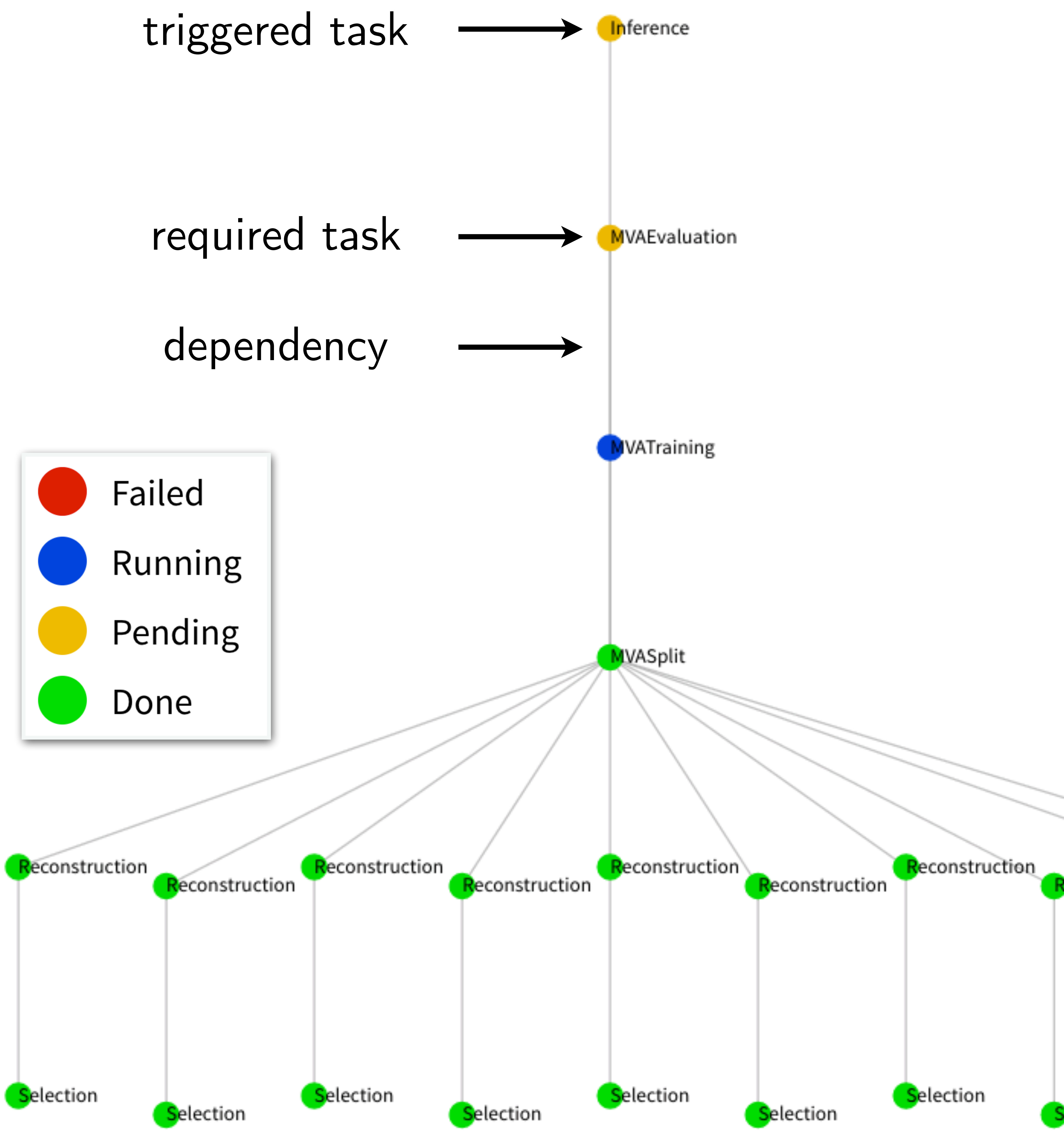
▶ RUNNING	ttH-bb-semi.SyncCSVs	noDeps=False, taskName=EMPTY_STRING, paramFile=EMPTY_STRING, log=EMPTY_STRING, setup=RunIISpring16MiniAODv2_13TeV_25bx_80X,	0	04/12/2016, 22:43:24 | 68 minutes	

make - like execution system

- 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 exist*

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



triggered task →

required task →

dependency →

● Failed

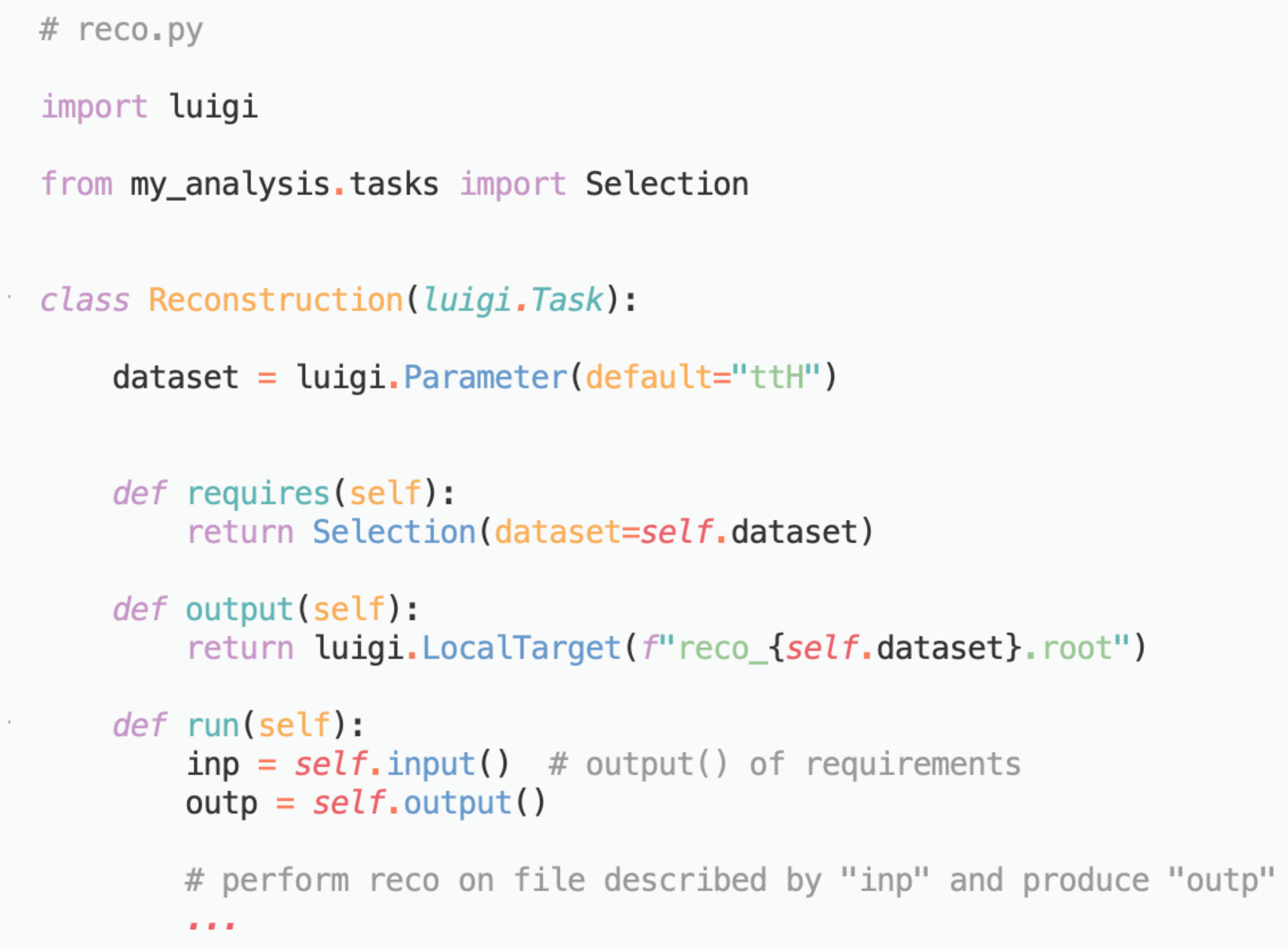
● Running

● Pending

● Done

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

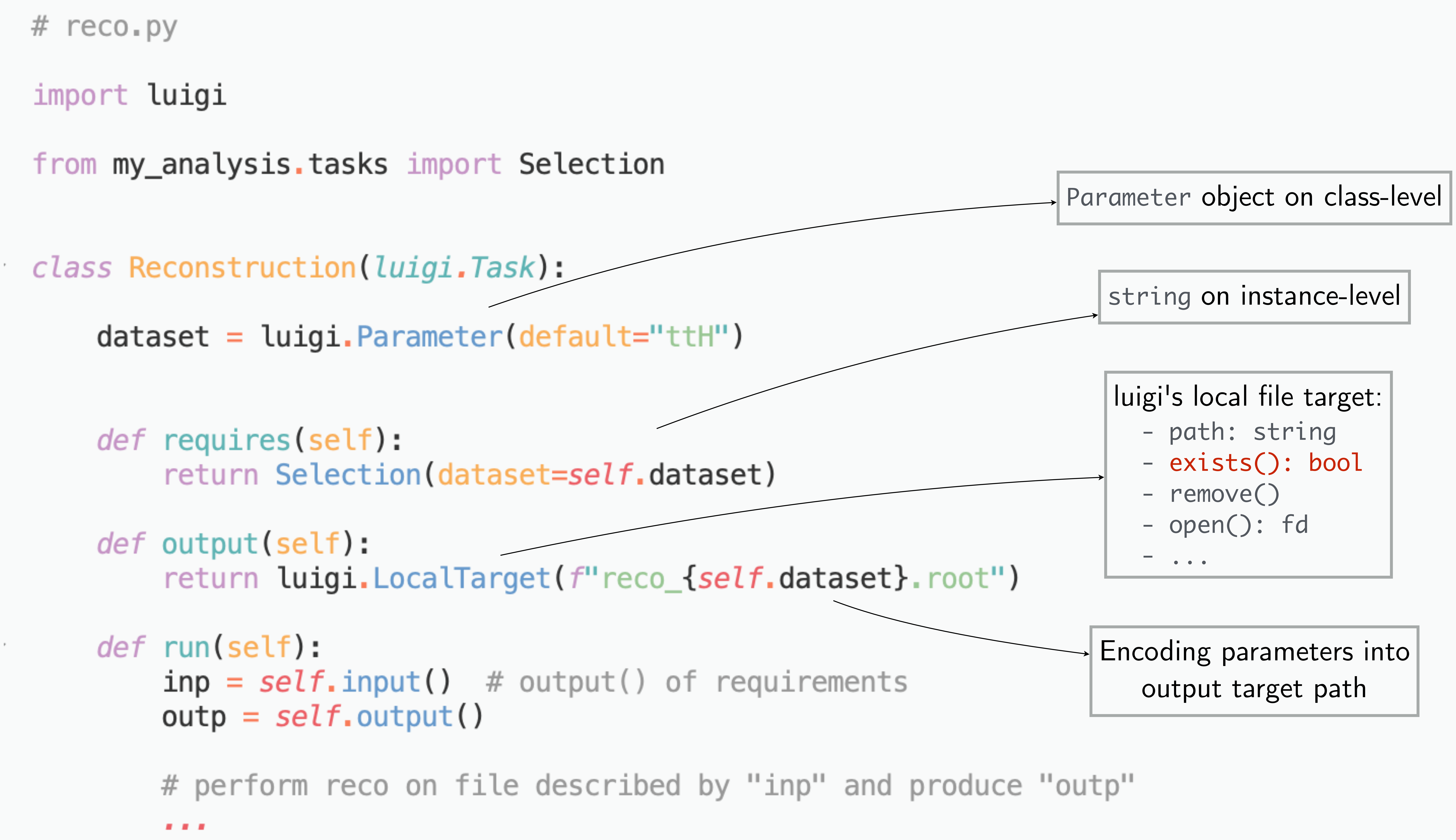
Luigi in a nutshell



```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
```

Luigi in a nutshell



```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

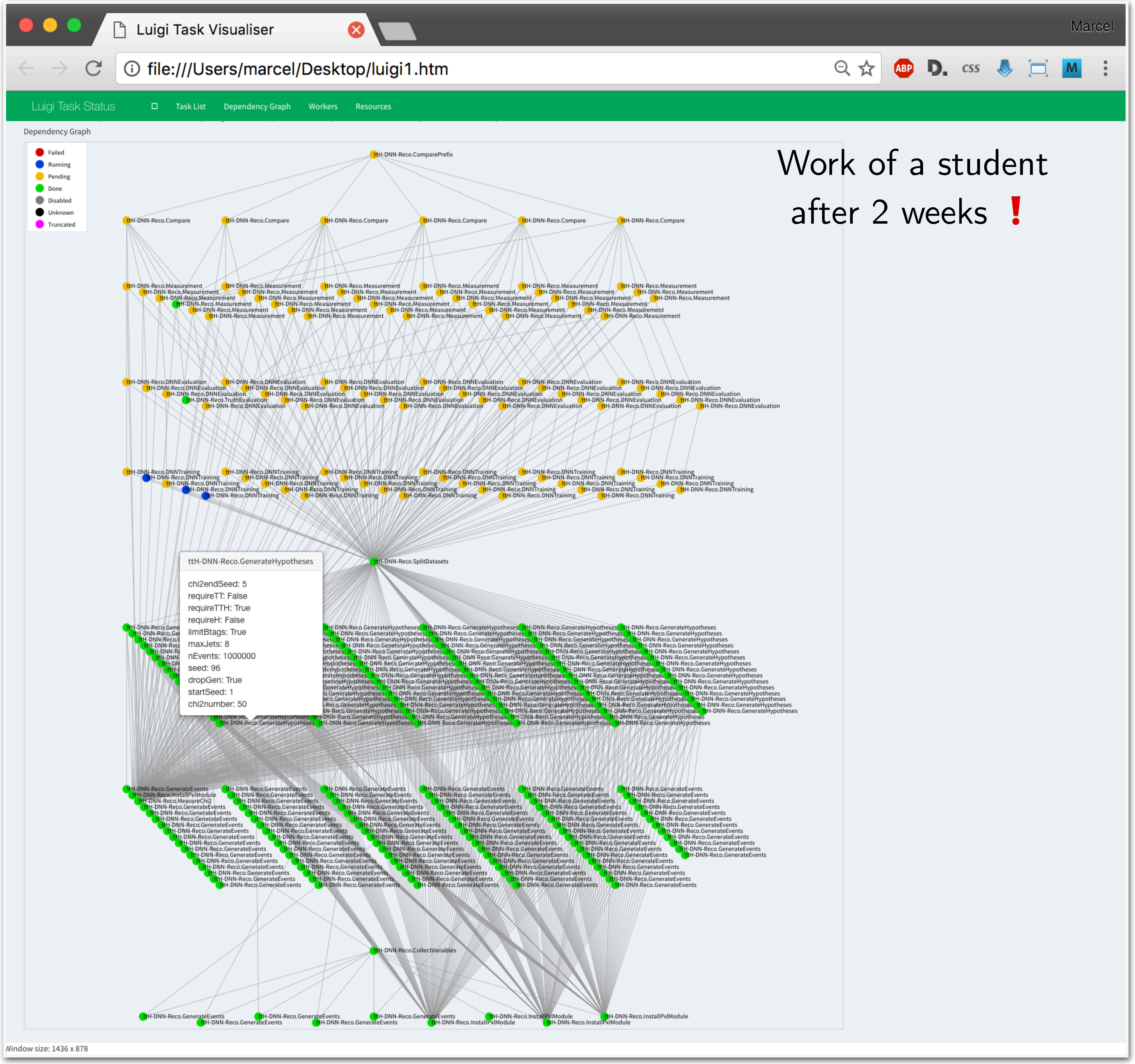
luigi's local file target:

- path: string
- exists(): bool
- remove()
- open(): fd
- ...

Encoding parameters into output target path

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

Example dependency trees



Luigi Task Visualiser

file:///Users/marcel/Desktop/luigi1.htm

Luigi Task Status □ Task List Dependency Graph Workers Resources

Dependency Graph

- 🟥 Failed
- 🟦 Running
- 🟨 Pending
- 🟩 Done
- ● Disabled
- ● Unknown
- ● Truncated

ttH-DNN-Reco.GenerateHypotheses

chi2endSeed: 5
requireTT: False
requireTTH: True
requireH: False
limitBtags: True
maxJets: 8
nEvents: 1000000
seed: 96
dropGen: True
startSeed: 1
chi2number: 50

Window size: 1436 x 878

Marcel
ttH-DNN-Reco.ComparePrefix
ttH-DNN-Reco.Compare
ttH-DNN-Reco.Measurement
ttH-DNN-Reco.DNNEvaluation
ttH-DNN-Reco.TruthEvaluation
ttH-DNN-Reco.DNNTraining
ttH-DNN-Reco.SplitDatasets
ttH-DNN-Reco.GenerateHypotheses
ttH-DNN-Reco.GenerateEvents
ttH-DNN-Reco.InstallPxlModule
ttH-DNN-Reco.MeasureChi2
ttH-DNN-Reco.CollectVariables

Work of a student after 2 weeks !

law

luigi analysis workflow

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

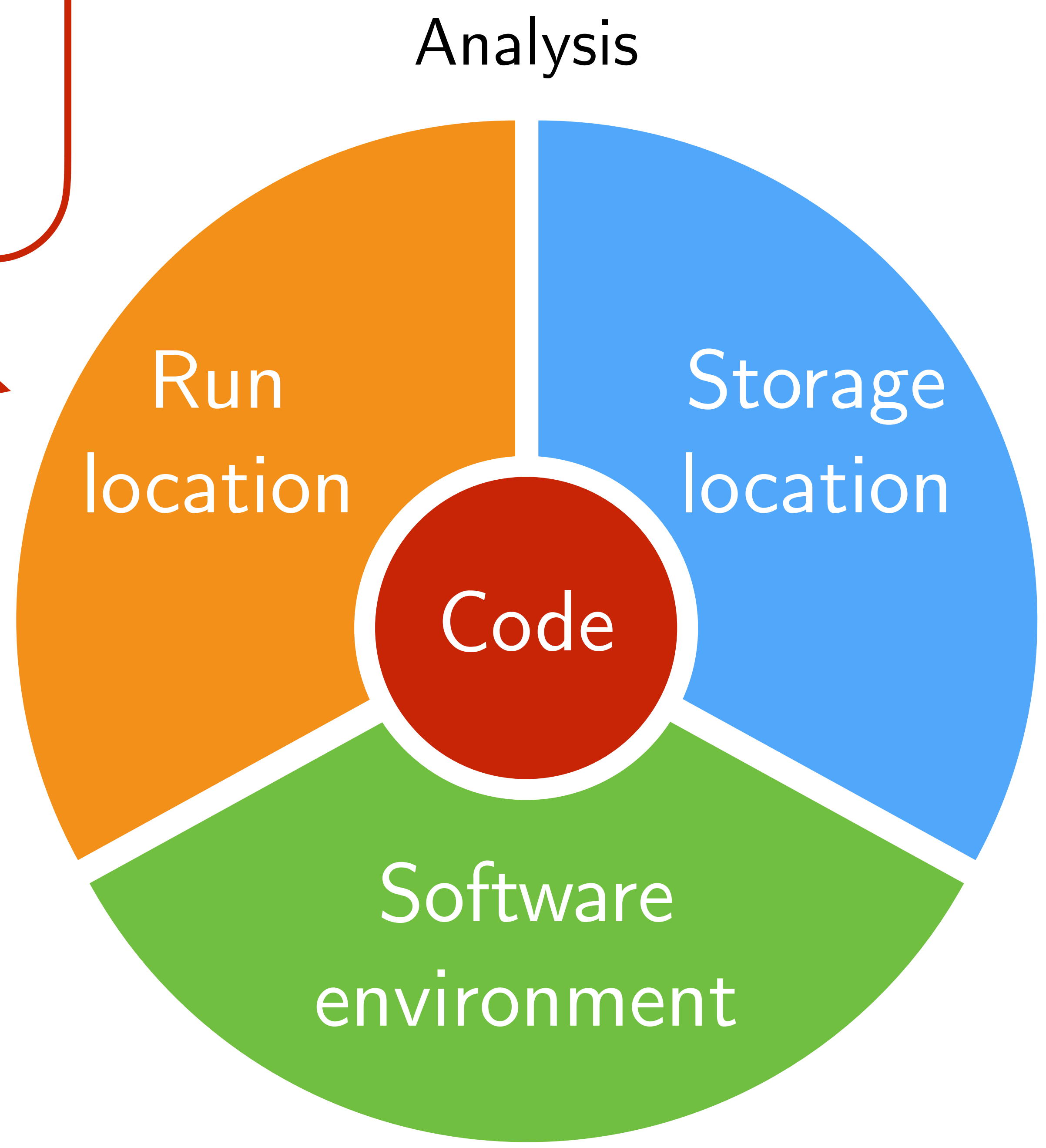
- Software design follows 3 primary goals:

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(40) analyses, O(150) people
  - Central groups, e.g. HIG, TAU, BTV, ...
- Also used outside CMS and outside HEP

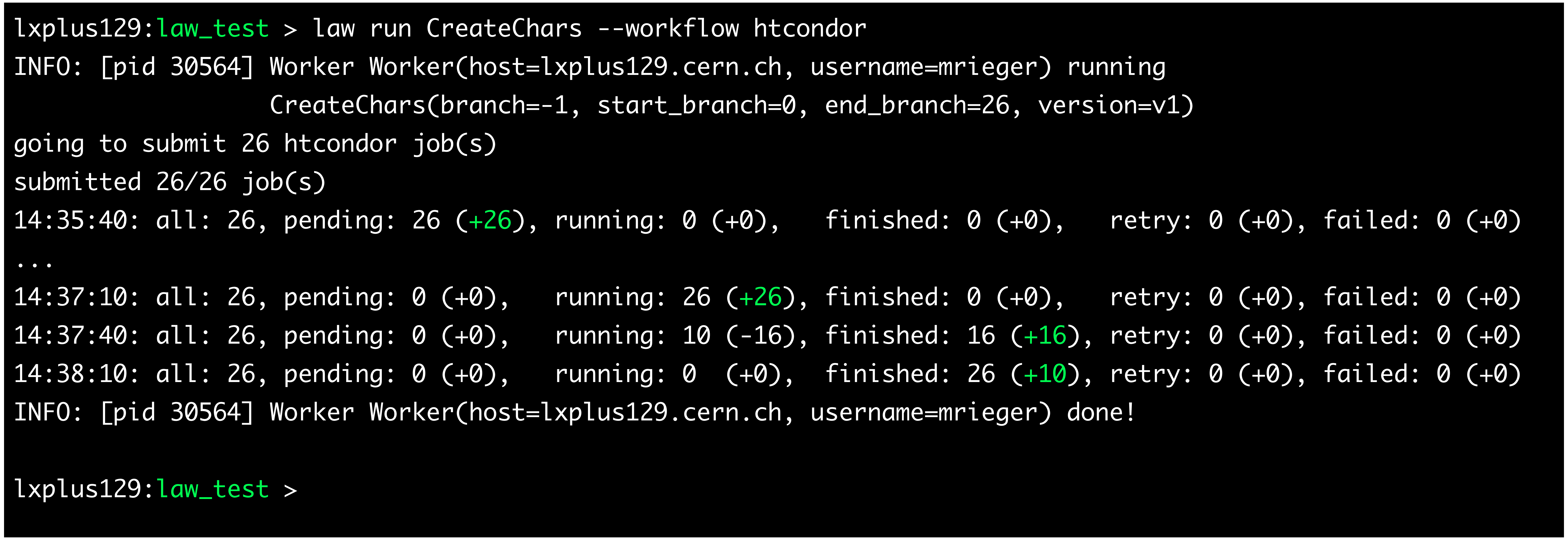
law
luigi analysis workflow



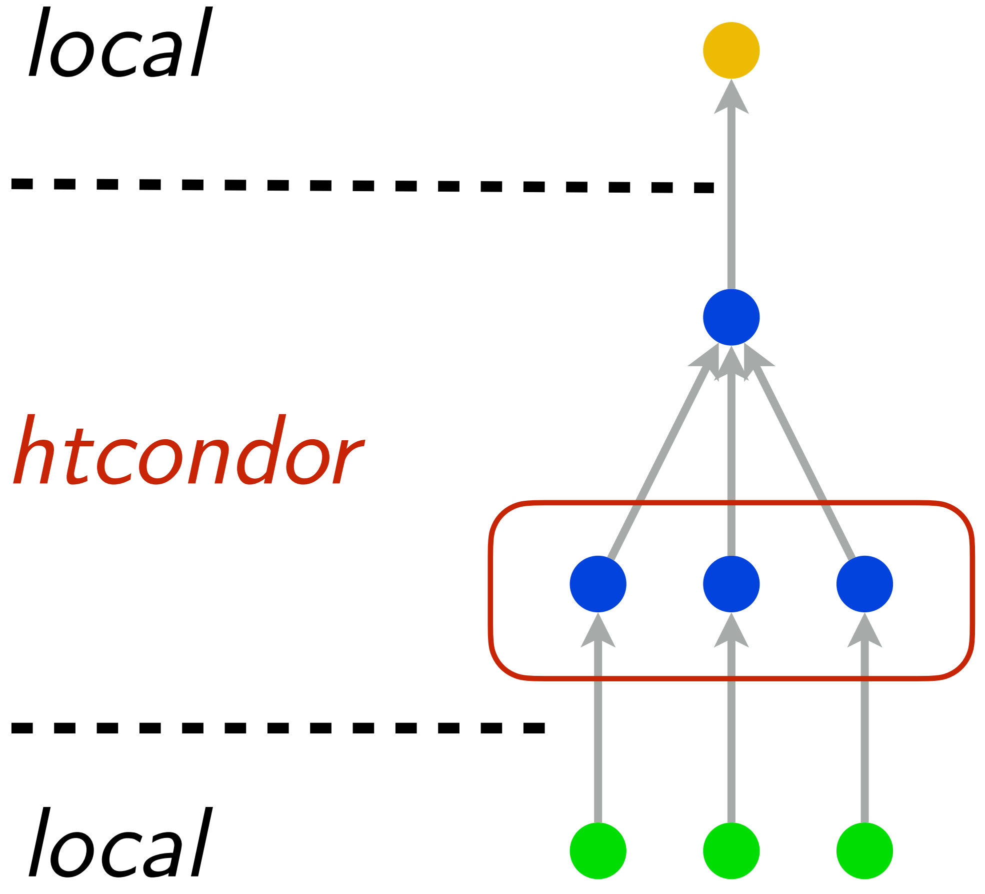
Law features (1)

1. Job submission

- Idea: submission built into tasks, **no need to write extra code**
- Currently supported job systems: HTCondor, Slurm, LSF, CMS-CRAB, gLite, ARC
- Mandatory features such as automatic resubmission, flexible task ↔ job matching, job files fully configurable, choice between *single job* and *job cluster* submission new
- 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 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 >
```

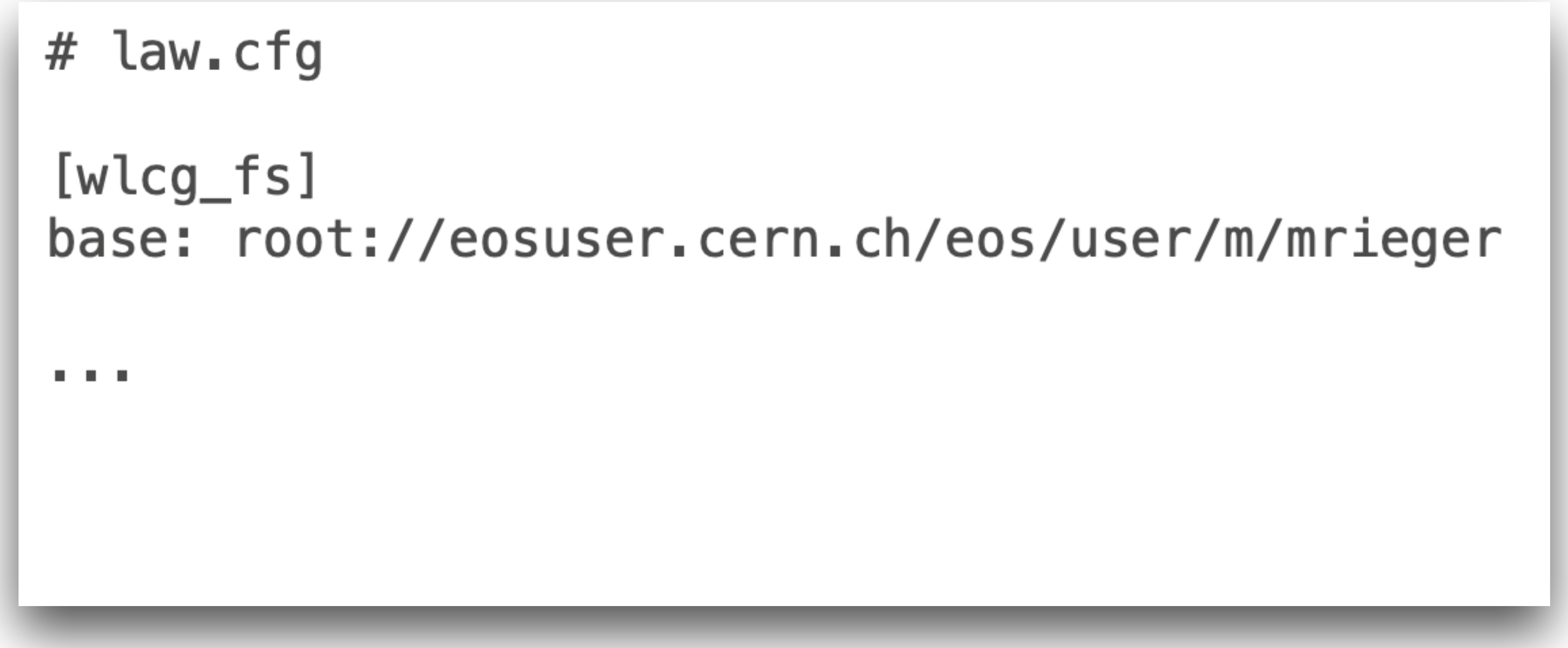


Law features (2)

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, ...

“FileSystem” configuration



```
# law.cfg

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

...
```

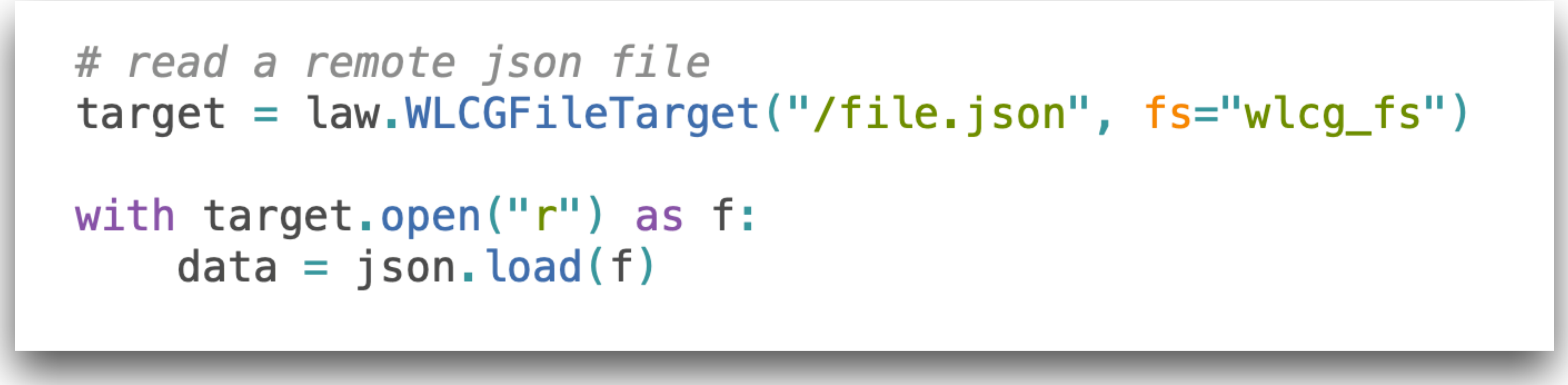
- Base path prefixed to all paths using this “fs”
- Configurable per file operation (stat, listdir, ...)
- Protected against removal of parent directories

Law features (2)

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



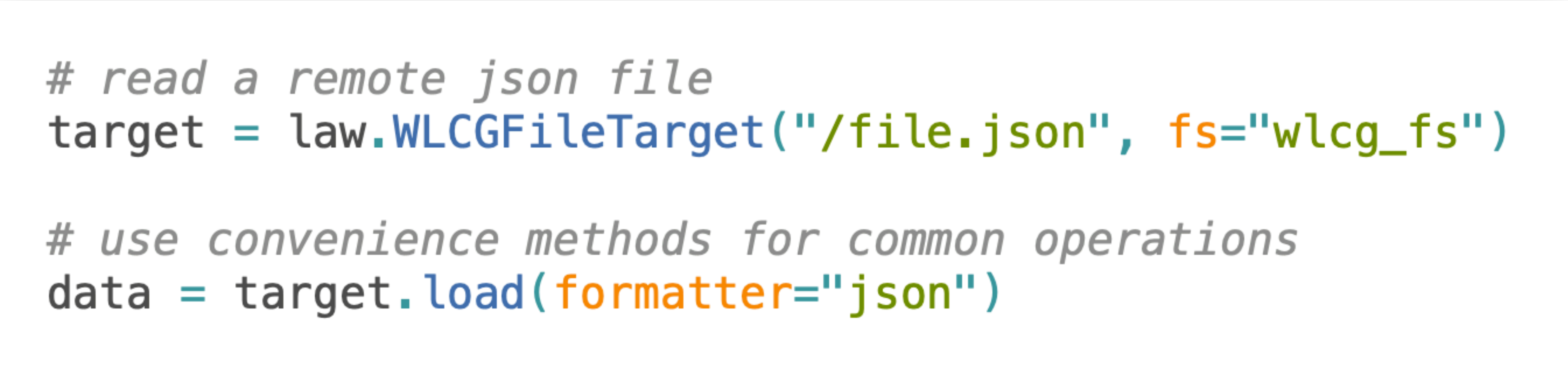
```
# read a remote json file
target = law.WLCGFileTarget("/file.json", fs="wlcg_fs")

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

Law features (2)

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



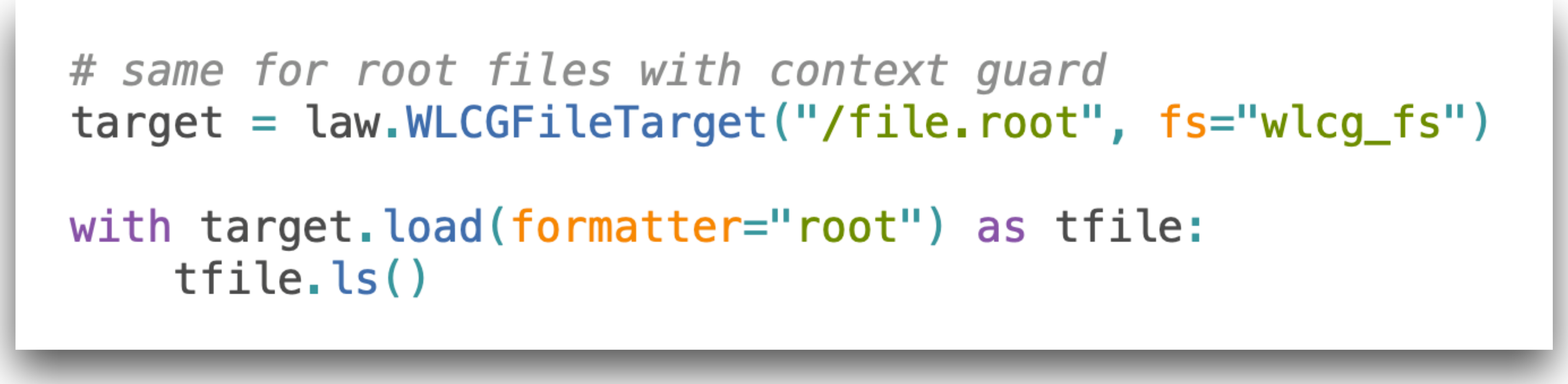
```
# read a remote json file
target = law.WLCGFileTarget("/file.json", fs="wlcg_fs")

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

Law features (2)

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

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

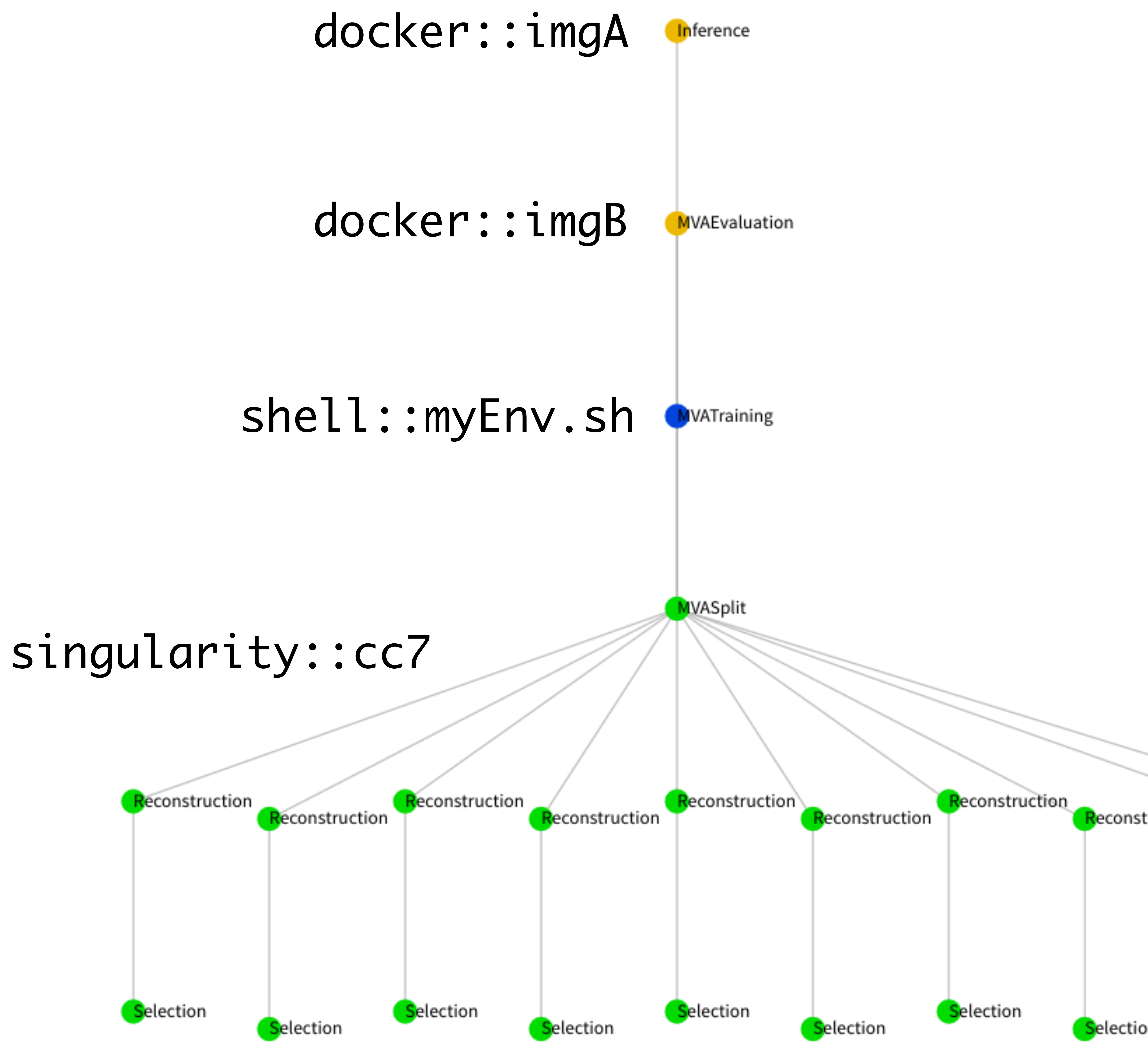
Law features (3)

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



docker::imgA

docker::imgB

shell::myEnv.sh

singularity::cc7

Law in action

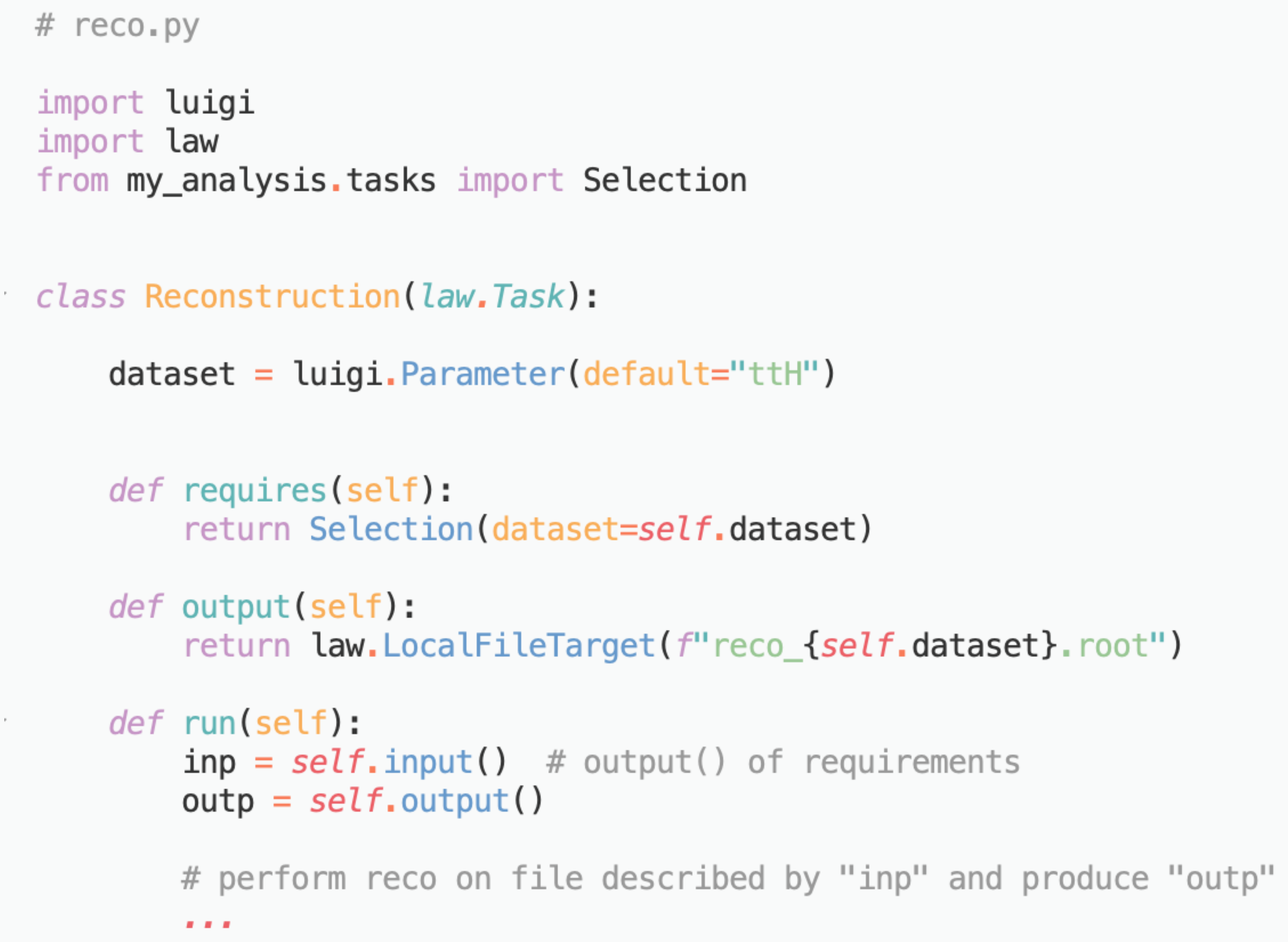
```python
# reco.py

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"
        ...
```

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

Example ☞

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

Law in action



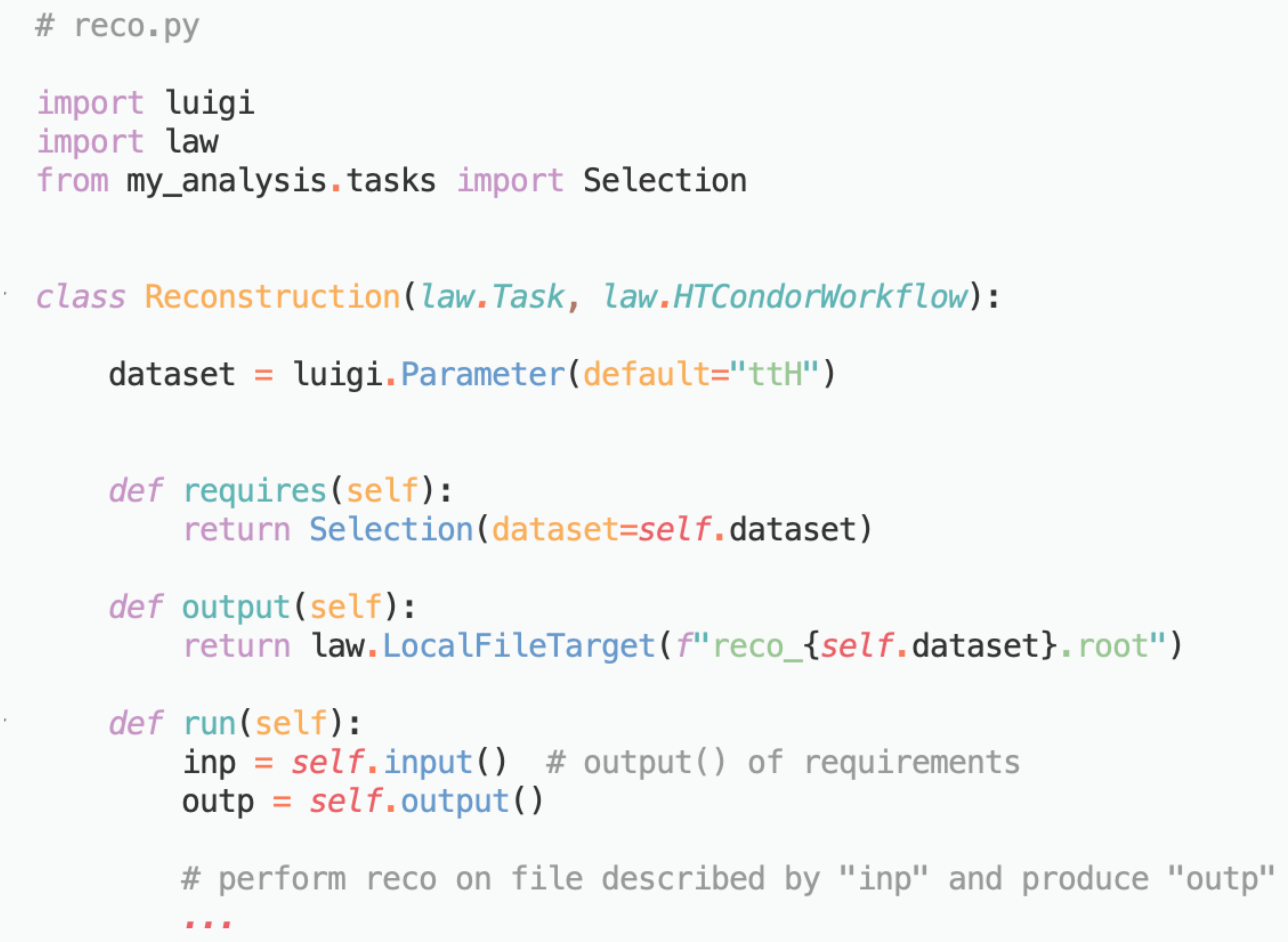
```
# reco.py

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
```

15 Law in action



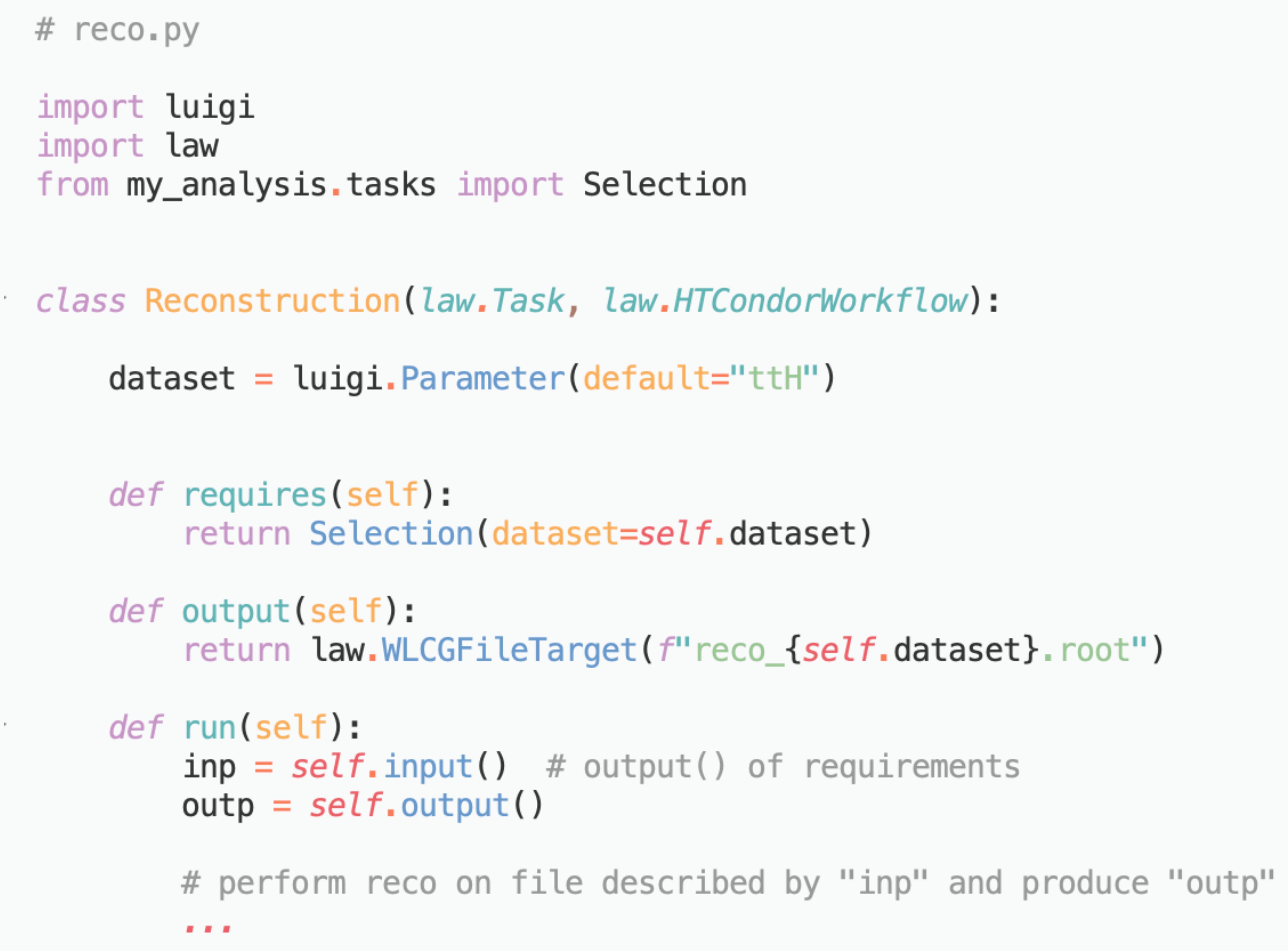
```python
# reco.py

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
```

Law in action



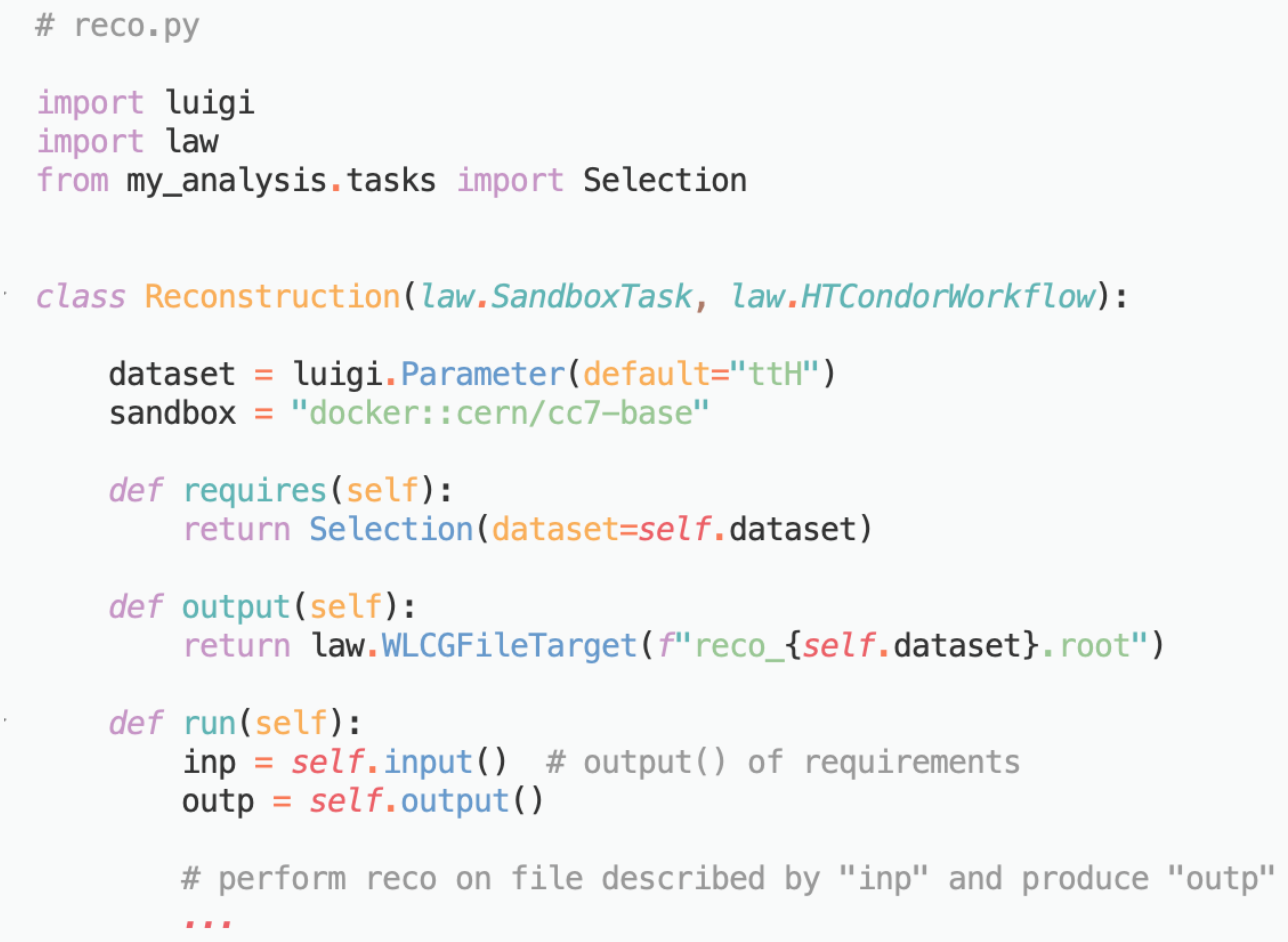
```
# reco.py

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
```

Law in action



```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
```

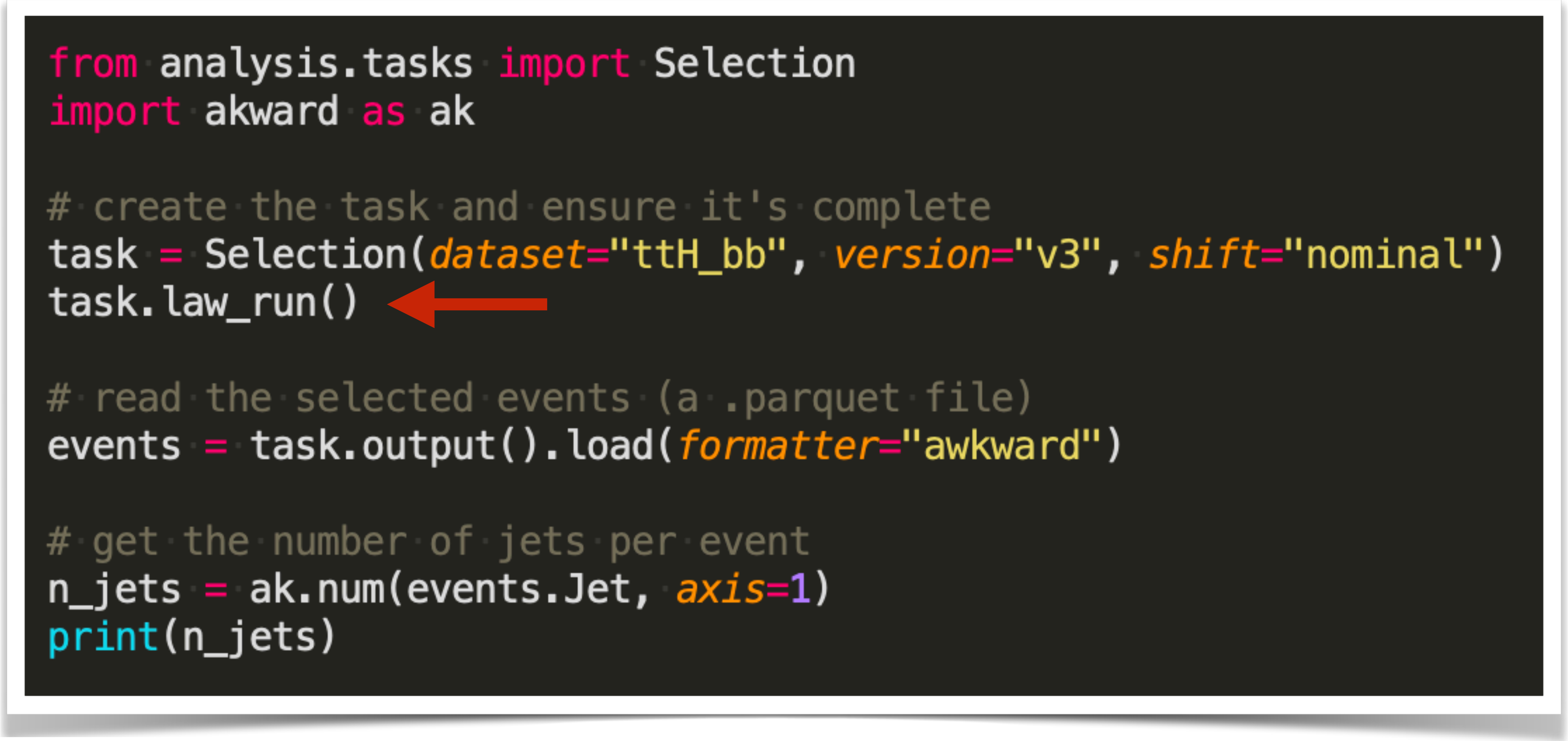
Triggers: CLI, scripting and notebooks

- **CLI**

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

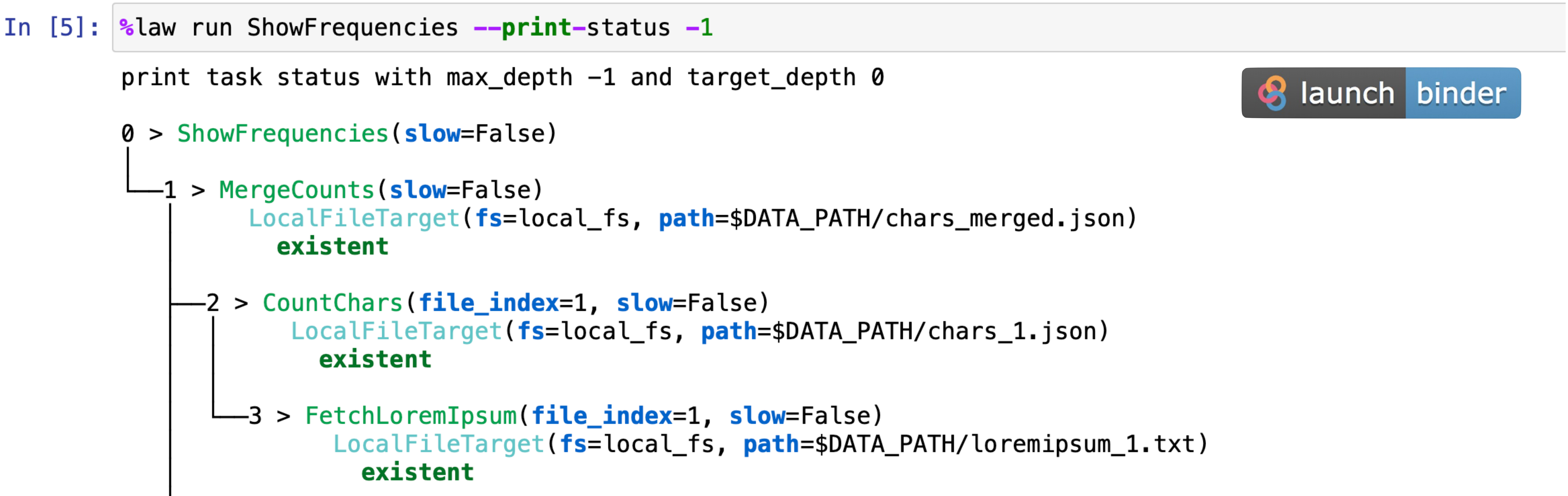
  - Full auto-completion of tasks and parameters

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



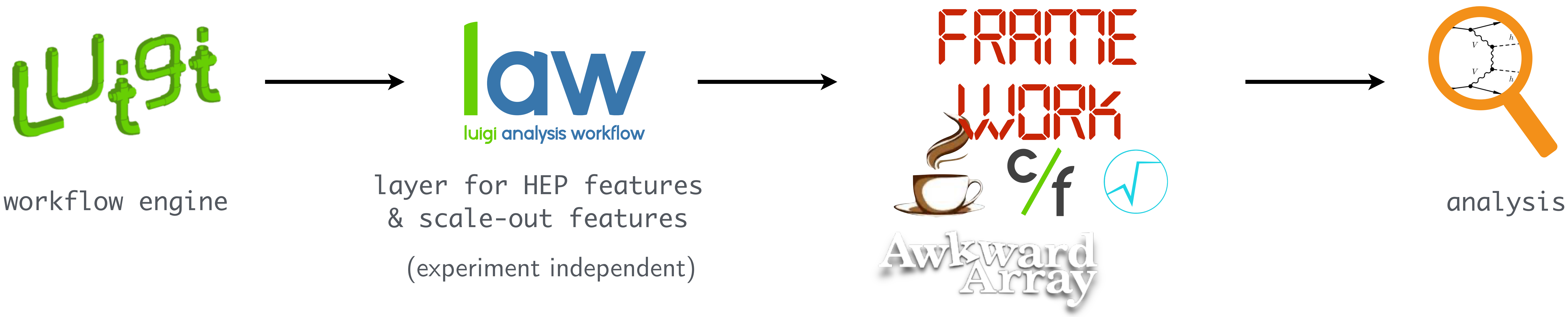
```python
from analysis.tasks import Selection
import akward as ak

# 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
   │
```

Automation/analysis stack



workflow engine

layer for HEP features
& scale-out features

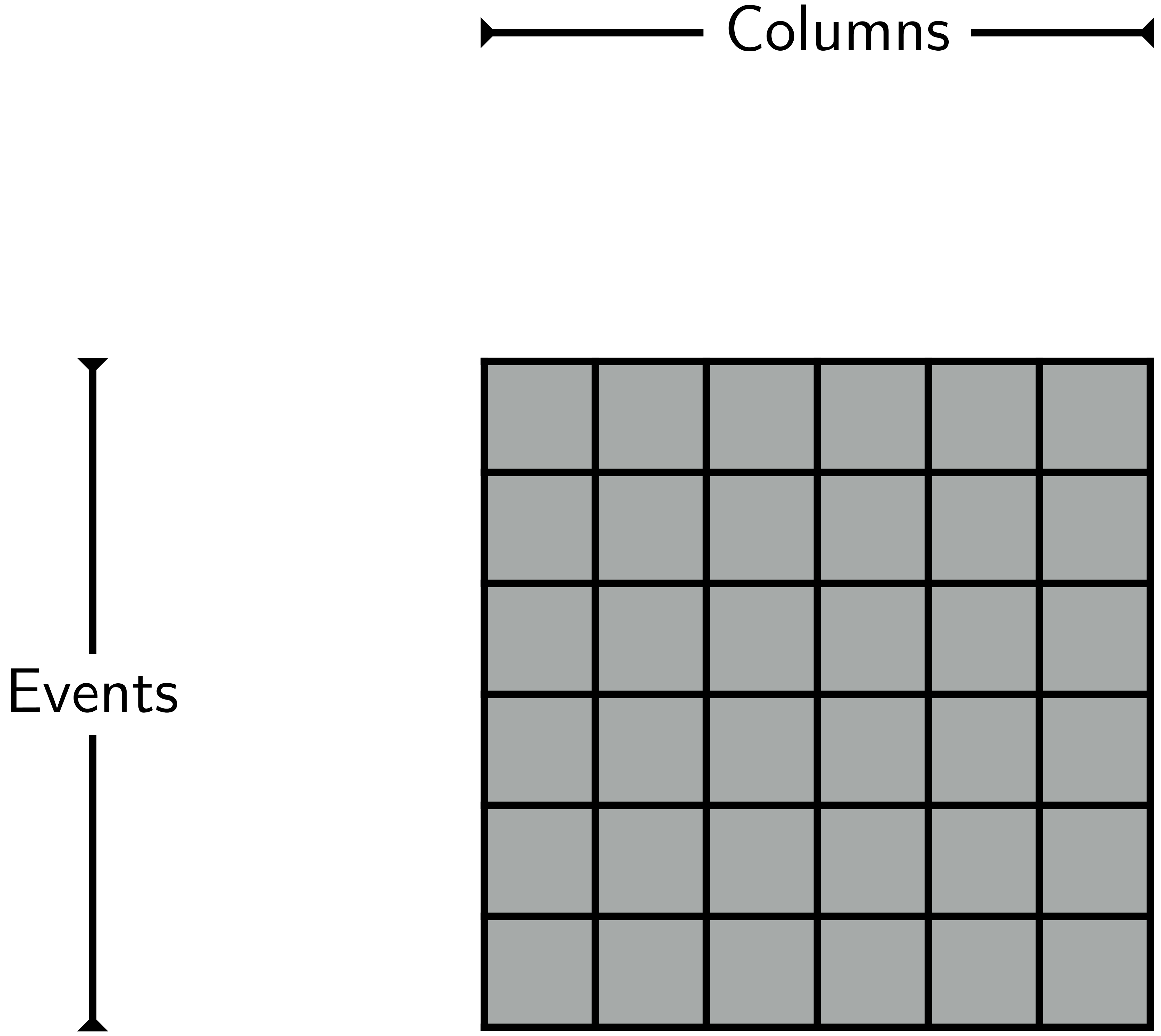
(experiment independent)

analysis

Example 1

columnflow

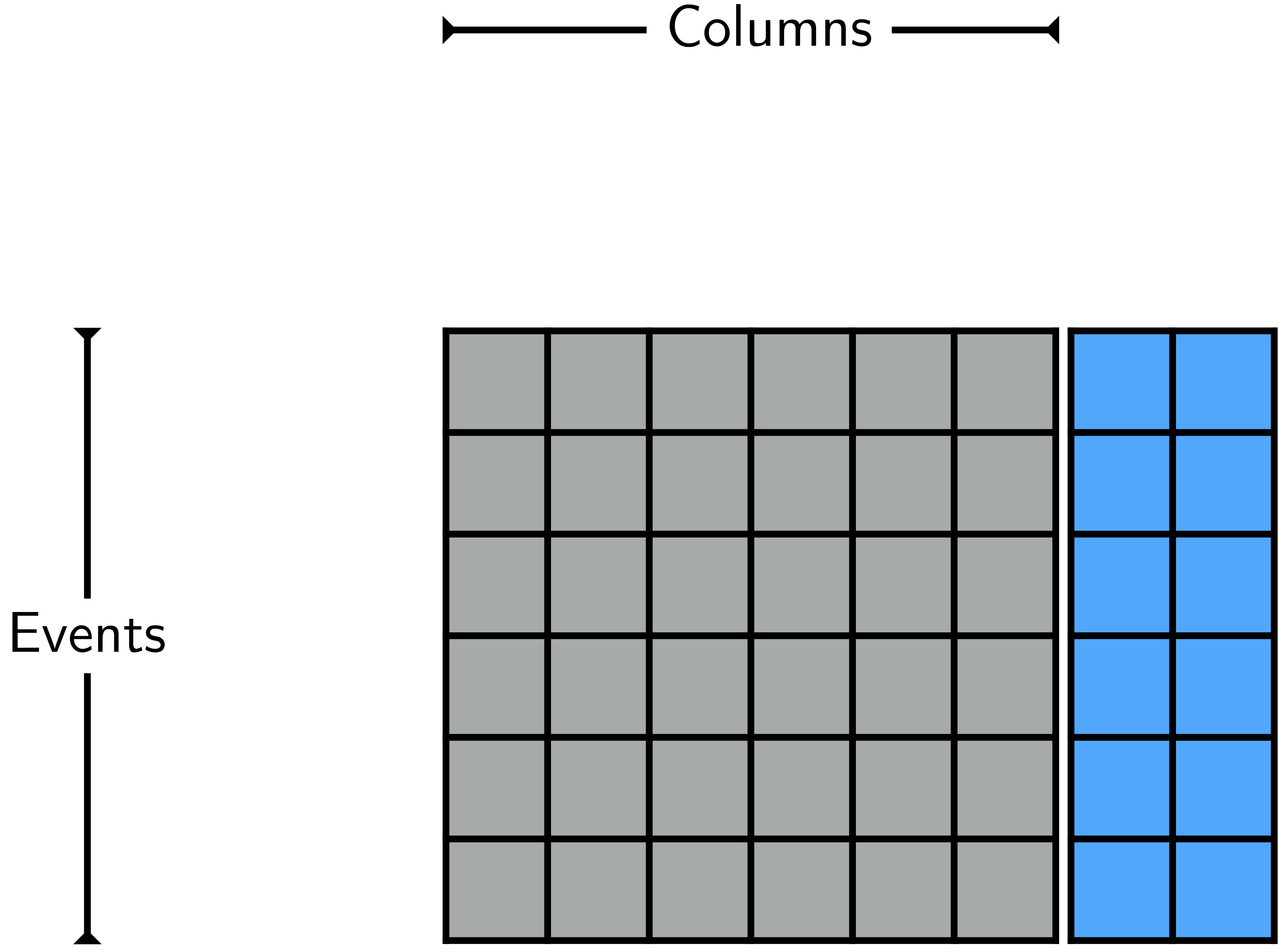
Columnar operations



Operations

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

Columnar operations



Operations

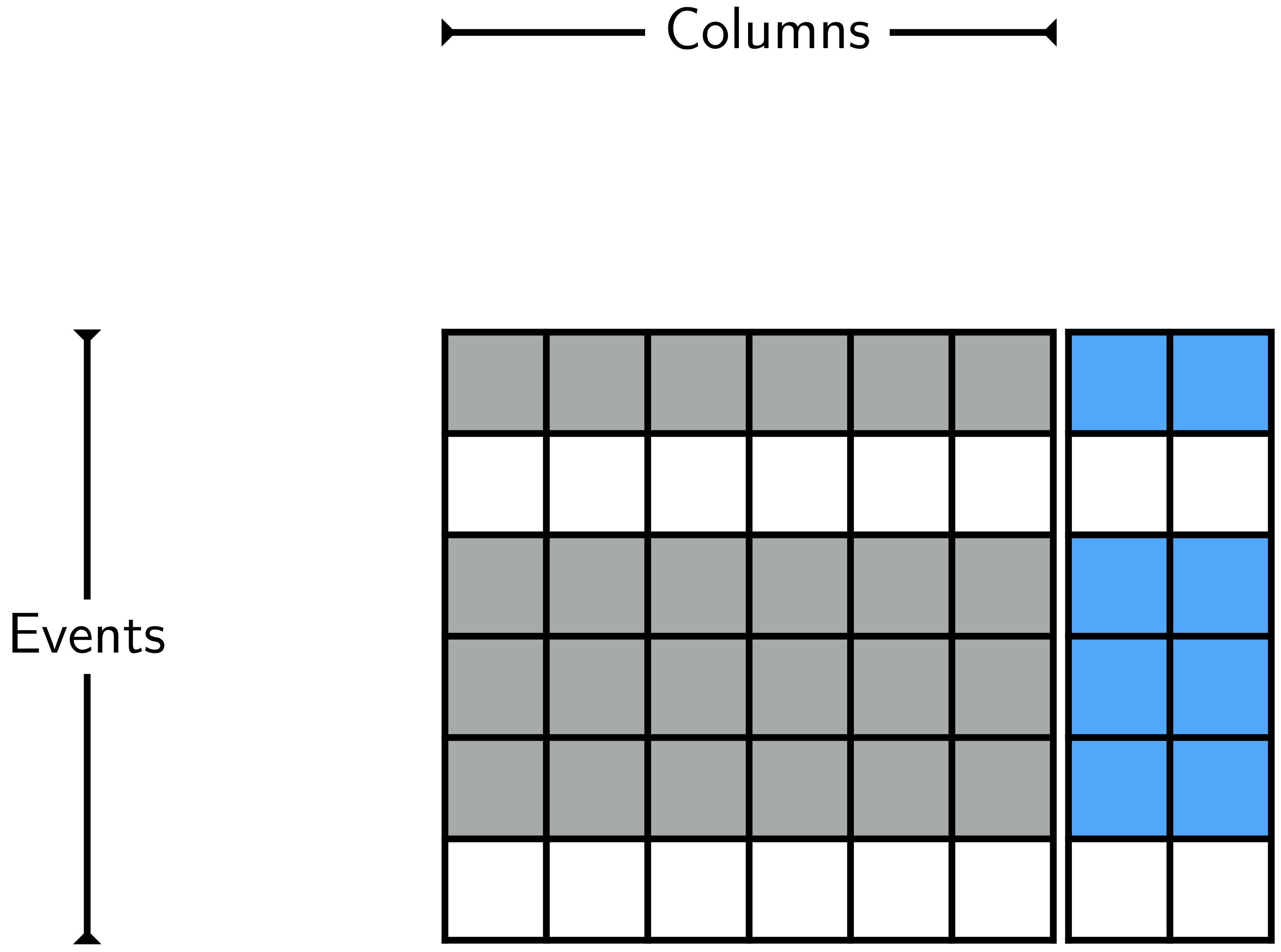
☑ Extension

□ Selection (*creating* masks)

□ Reduction (*applying* masks)

□ Extension

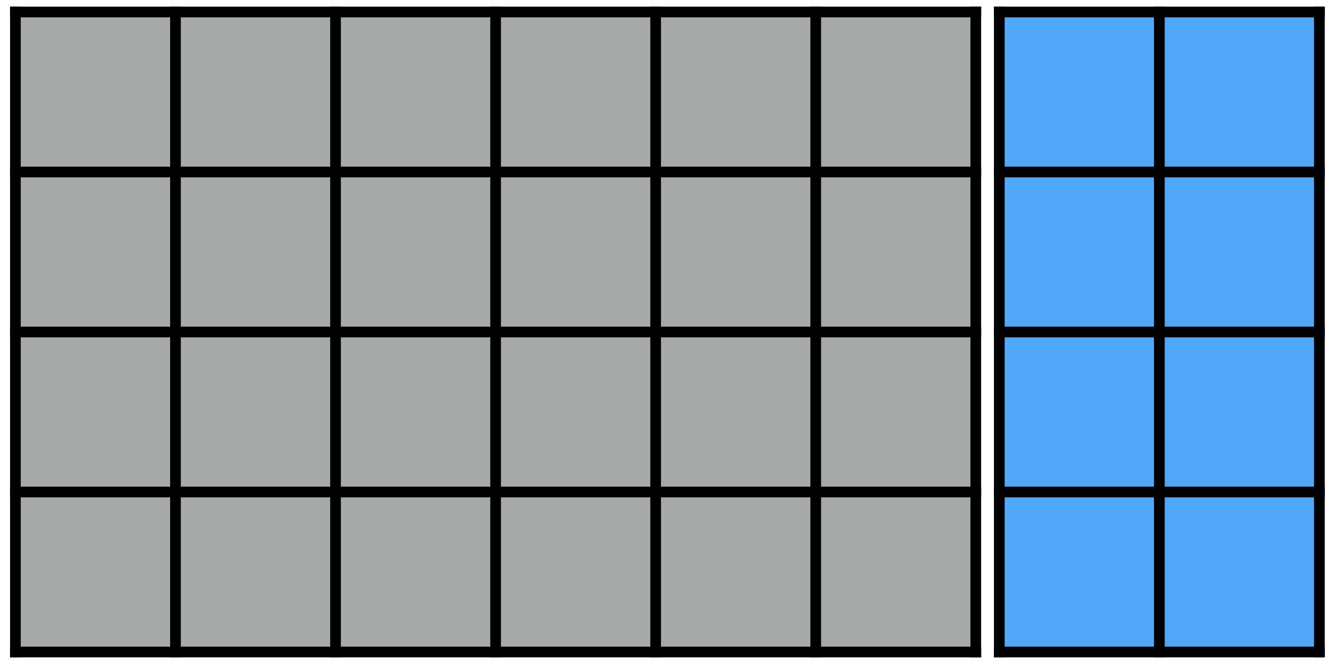
□ Merge



Operations

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

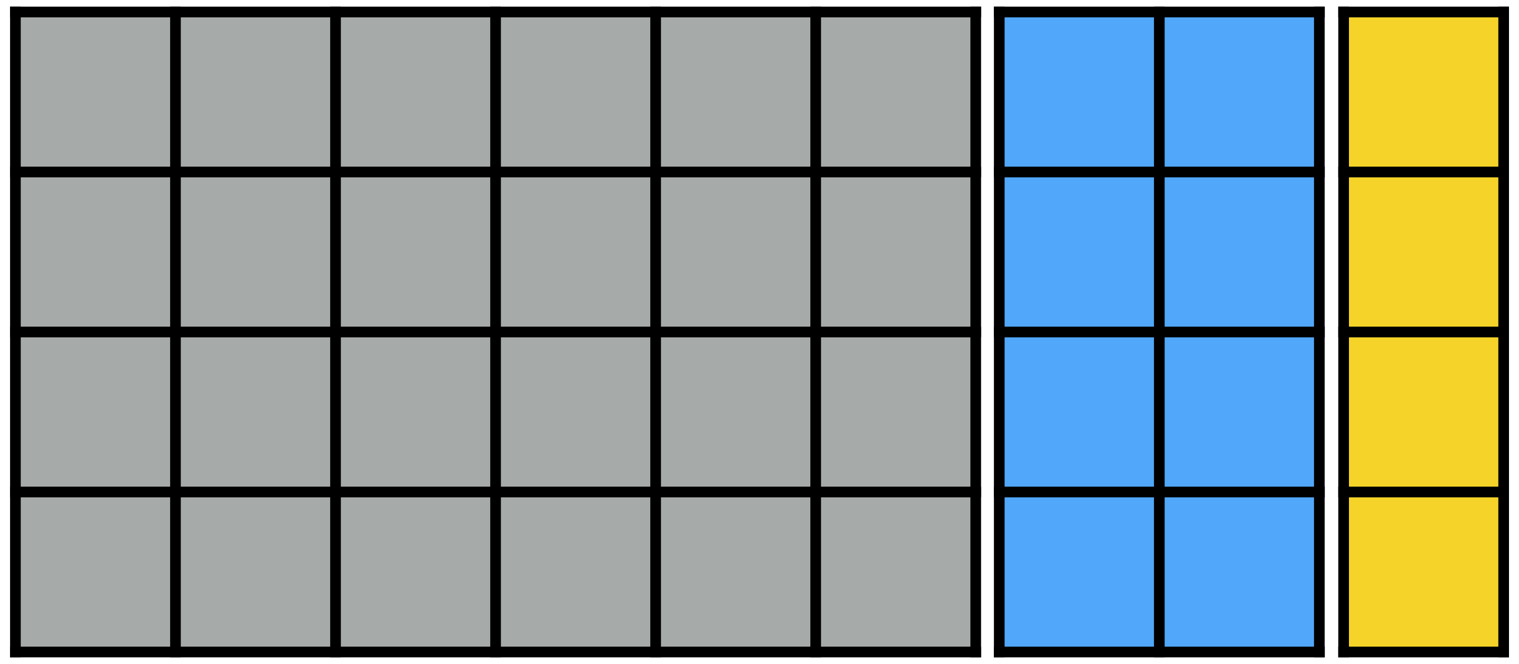
Columnar operations



Operations

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

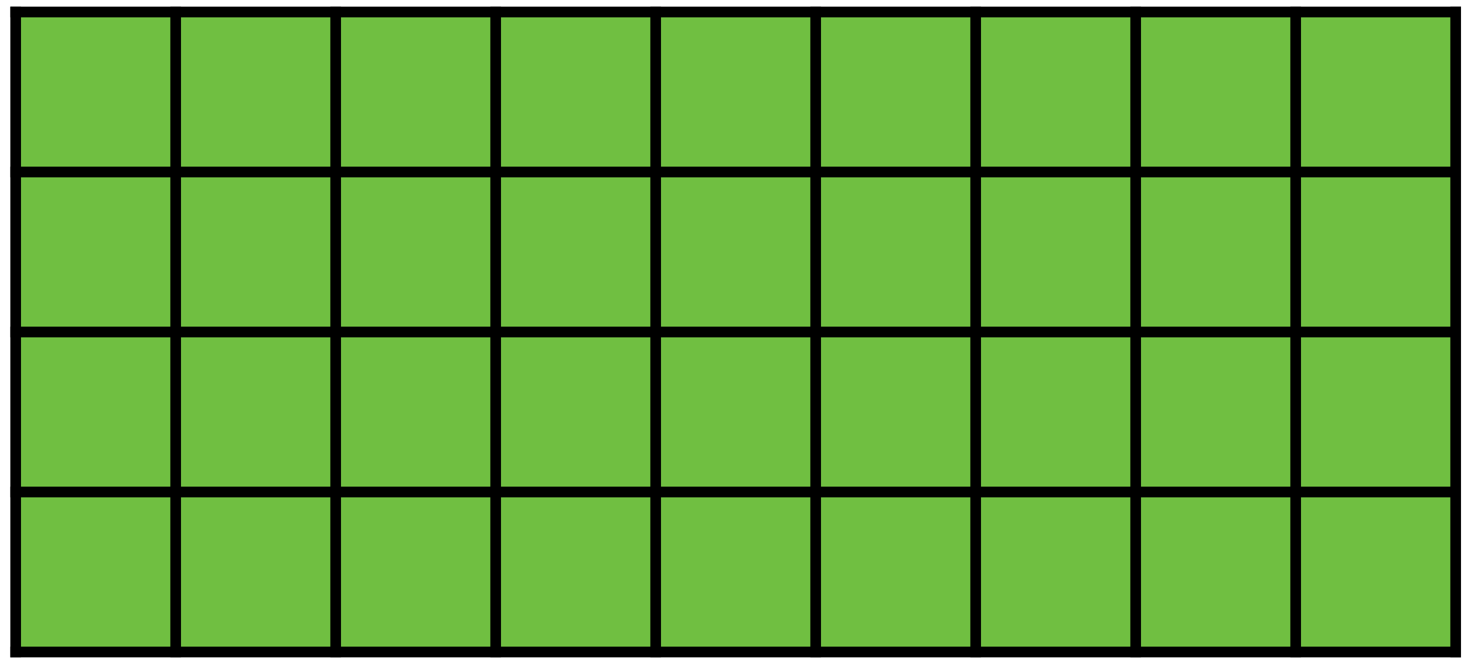
Columnar operations



Operations

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

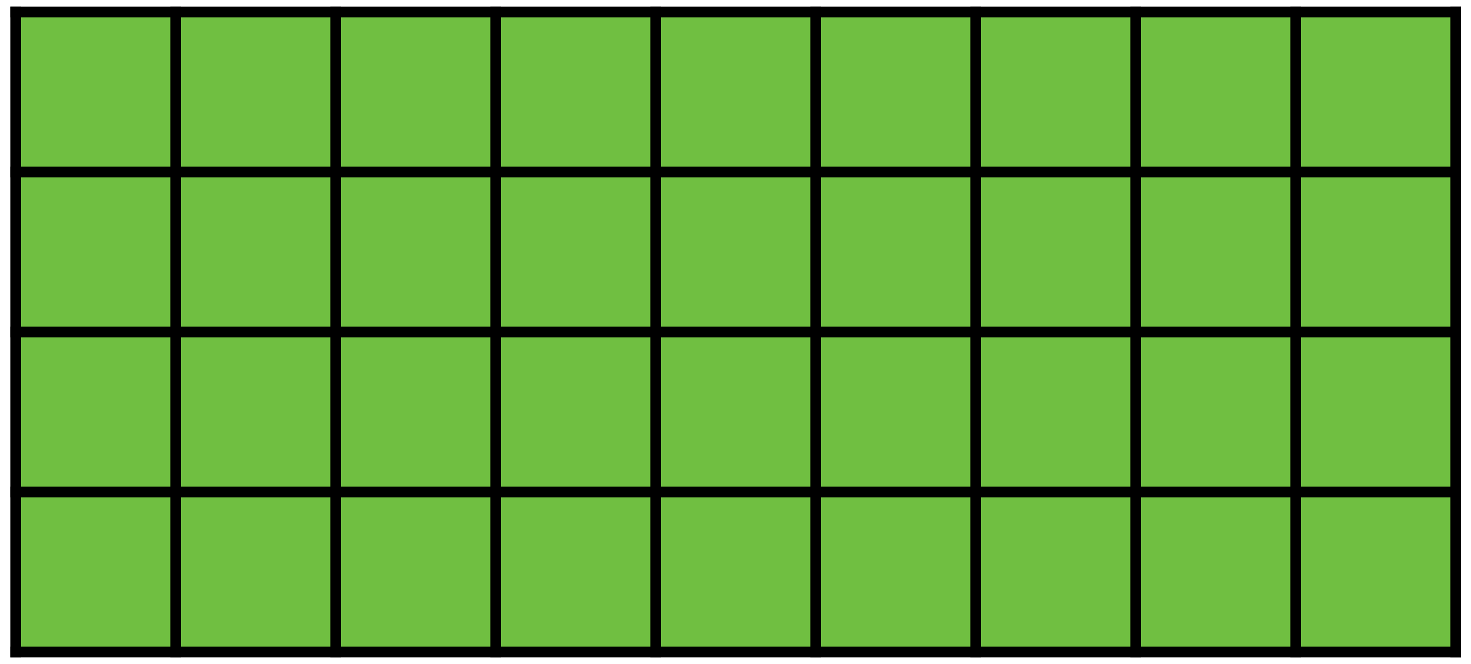
Columnar operations



Operations

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

Columnar operations



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

- **Non-trivial with persistent intermediate files** across a large scale analysis

Operations

☑ Extension

☑ Selection (*creating* masks)

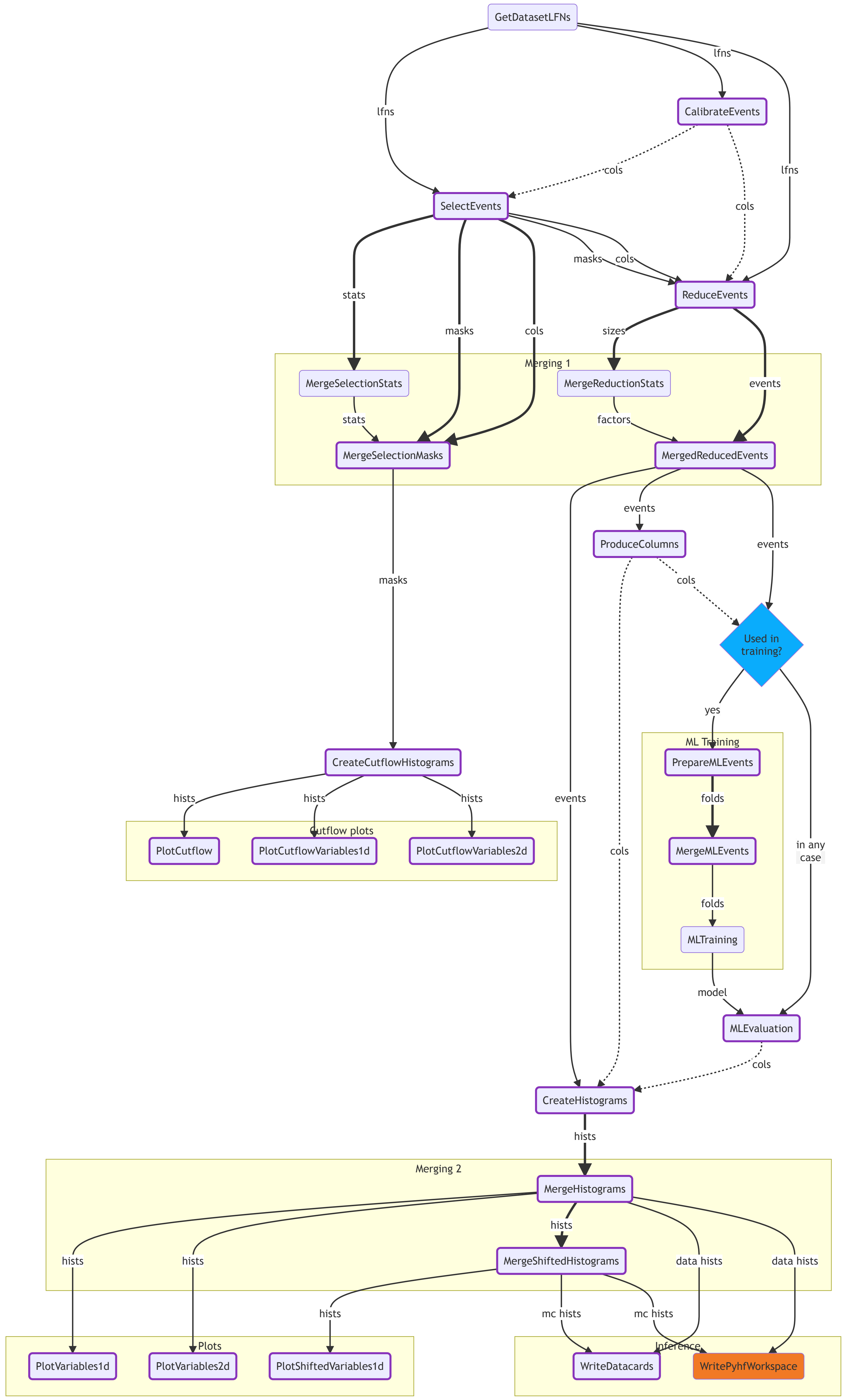
☑ Reduction (*applying* masks)

☑ Extension

☑ Merge

Core ingredients

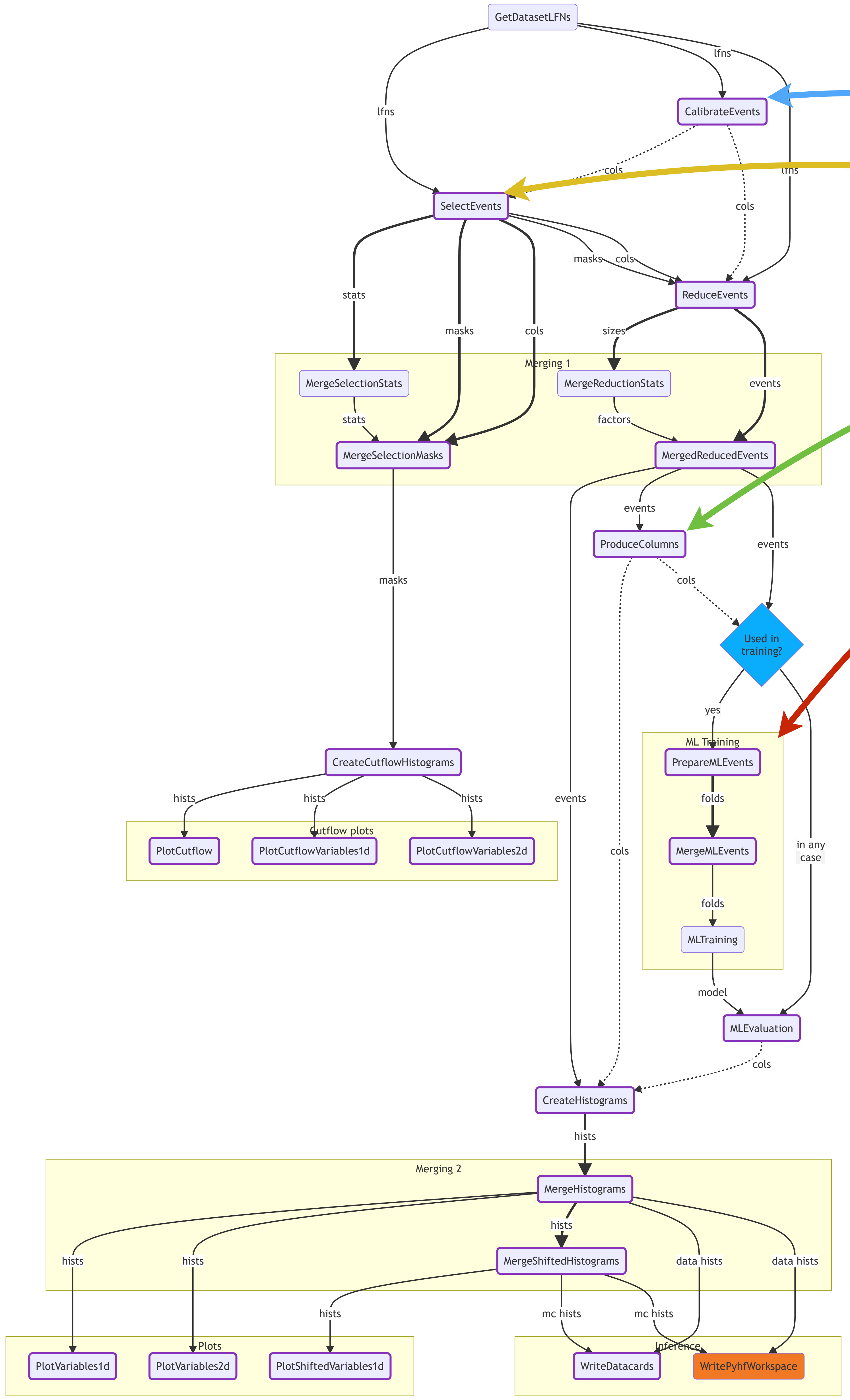
- 1️⃣ **Fully orchestrated workflow**
  - Only a *suggestion*, but able to model majority of analyses
  - Can be altered or created from scratch by analyses



live task graph

Core ingredients

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

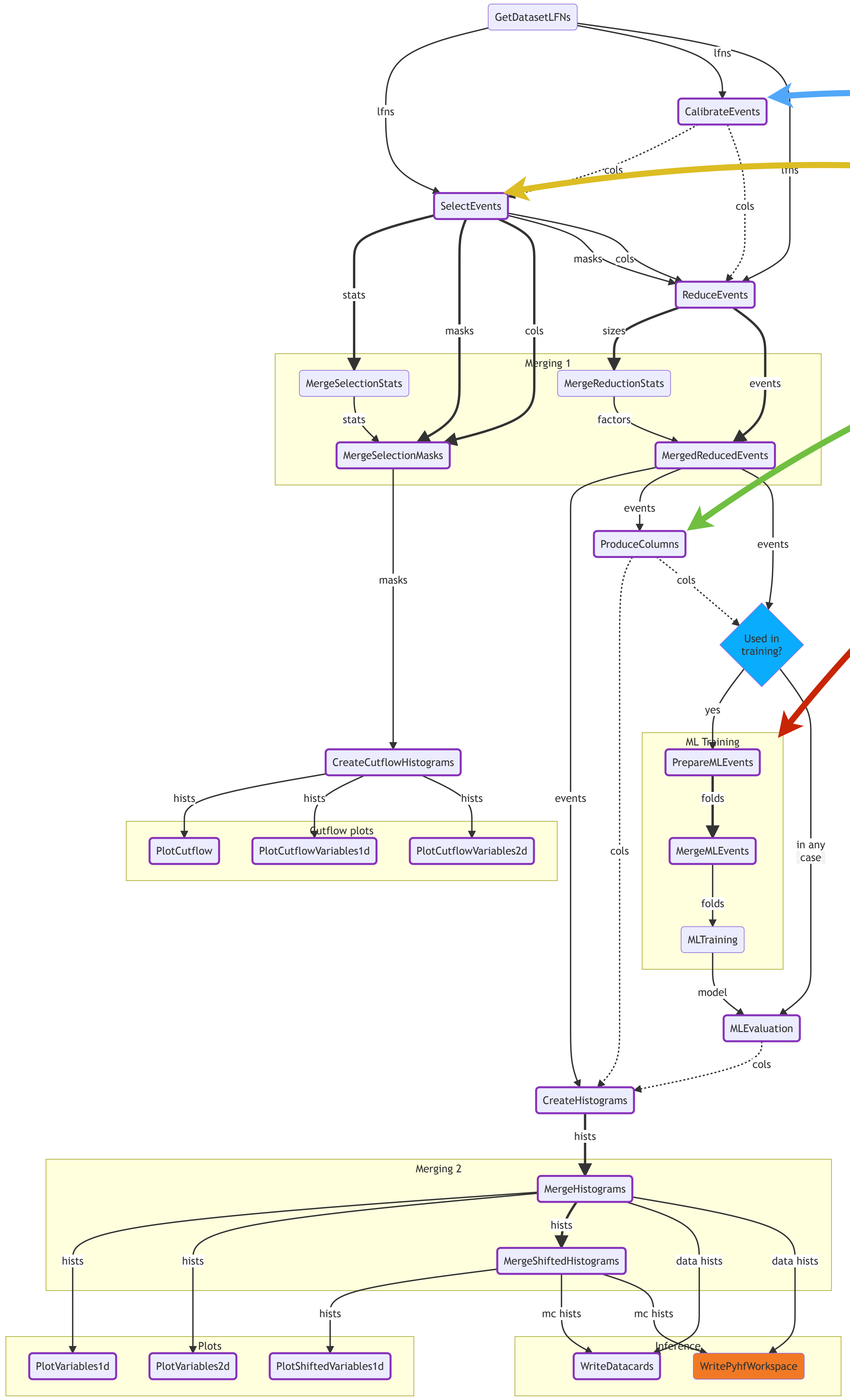


- 2️⃣ **Tools for on-demand column retrieval / production**
  - Configurable functions creating new columns at certain points of the workflow
  - Can be selected at execution time, e.g. `btag_weight`, `pu_weight`
  - Carry information on **used** and **produced** columns, → open & save only necessary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

live task graph

Core ingredients

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

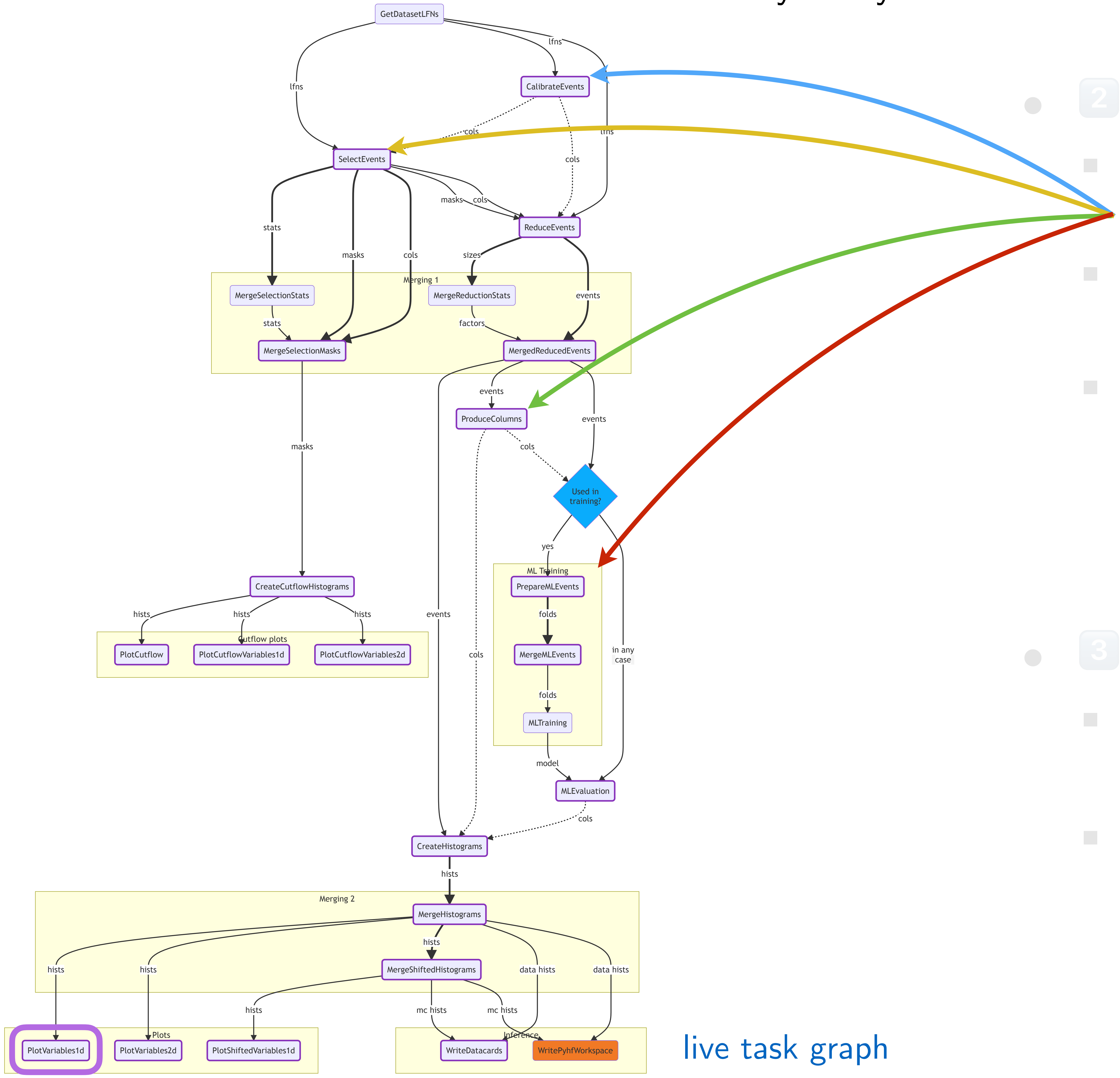


live task graph

- 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)

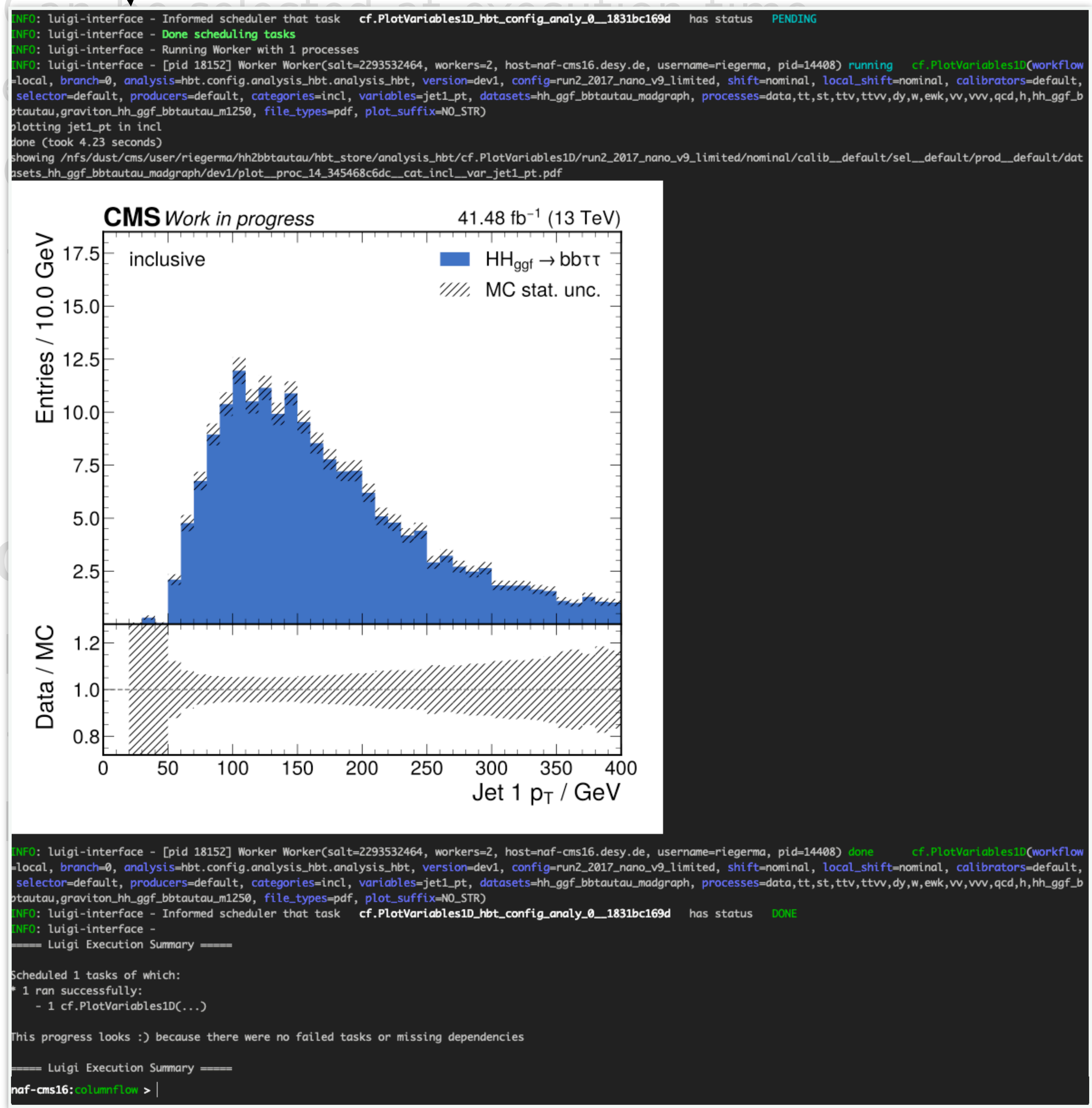
- 3 **Collection of standardized column producers (CMS)**
  - Mostly SF and weight production using `correctionlib` → `jec`, `jer`, `tec`, `e_sf`, `mu_sf`, `trigger_sf`, `btag_sf`, ...
  - Plug-in mechanism for analyses

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



live task graph

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



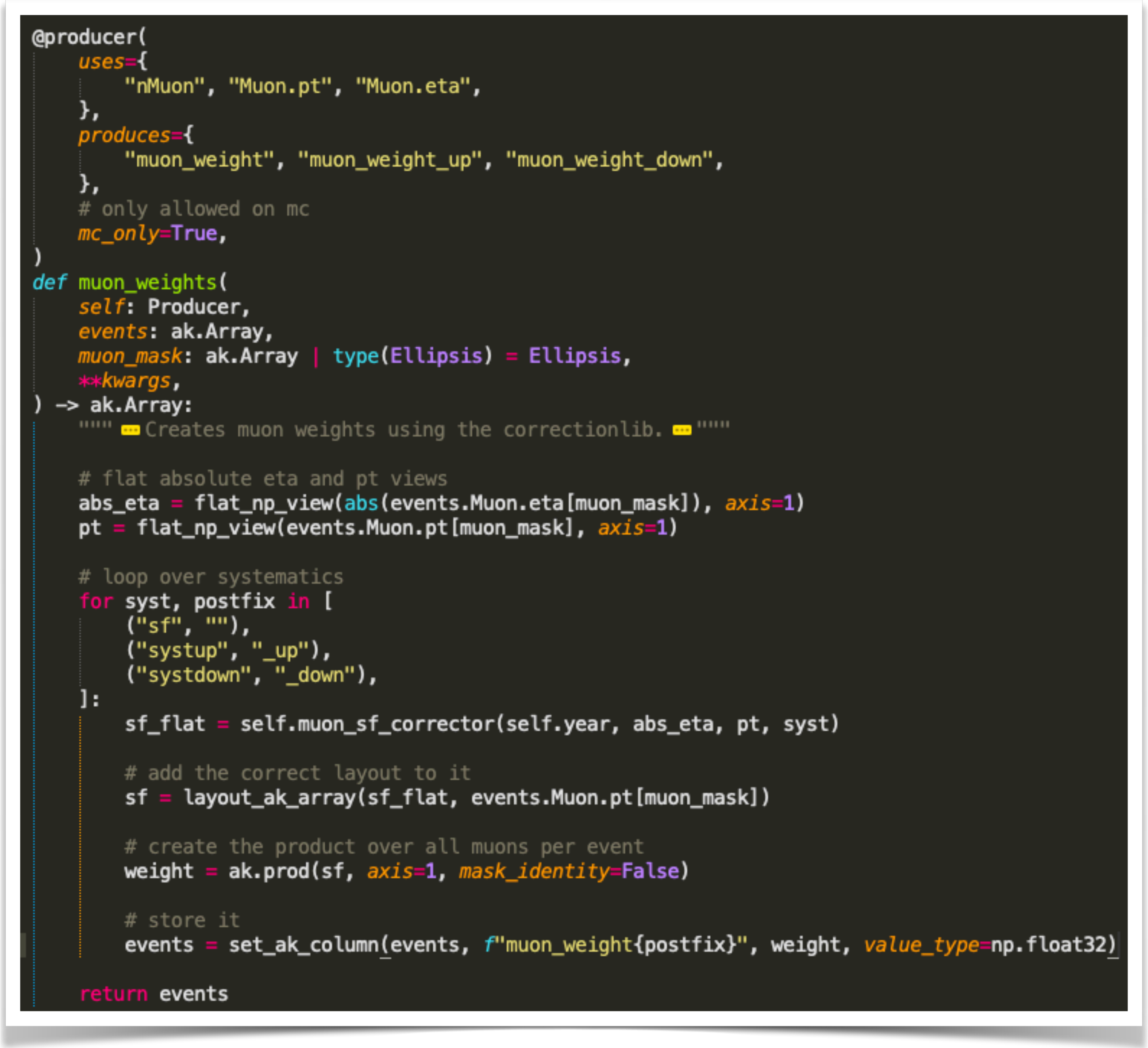
```
INFO: luigi-interface - Informed scheduler that task   cf.PlotVariables1D_hbt_config_analy_0__1831bc169d   has status   PENDING
INFO: luigi-interface - Done scheduling tasks
INFO: luigi-interface - Running Worker with 1 processes
INFO: luigi-interface - [pid 18152] Worker Worker(salt=2293532464, workers=2, host=naf-cms16.desy.de, username=riegerma, pid=14408) running   cf.PlotVariables1D(workflow
=local, branch=0, analysis=hbt.config.analysis_hbt.analysis_hbt, version=dev1, config=run2_2017_nano_v9_limited, shift=nominal, local_shift=nominal, calibrators=default,
 selector=default, producers=default, categories=incl, variables=jet1_pt, datasets=hh_ggf_bbtautau_madgraph, processes=data,tt,st,ttv,ttvv,dy,w,ewk,vv,vvv,qcd,h,hh_ggf_b
btautau,graviton_hh_ggf_bbtautau_m1250, file_types=pdf, plot_suffix=NO_STR)
plotting jet1_pt in incl
done (took 4.23 seconds)
showing /nfs/dust/cms/user/riegerma/hh2bbtautau/hbt_store/analysis_hbt/cf.PlotVariables1D/run2_2017_nano_v9_limited/nominal/calib__default/sel__default/prod__default/dat
asets_hh_ggf_bbtautau_madgraph/dev1/plot__proc_14_345468c6dc__cat_incl__var_jet1_pt.pdf
```

```
INFO: luigi-interface - [pid 18152] Worker Worker(salt=2293532464, workers=2, host=naf-cms16.desy.de, username=riegerma, pid=14408) done      cf.PlotVariables1D(workflow
=local, branch=0, analysis=hbt.config.analysis_hbt.analysis_hbt, version=dev1, config=run2_2017_nano_v9_limited, shift=nominal, local_shift=nominal, calibrators=default,
 selector=default, producers=default, categories=incl, variables=jet1_pt, datasets=hh_ggf_bbtautau_madgraph, processes=data,tt,st,ttv,ttvv,dy,w,ewk,vv,vvv,qcd,h,hh_ggf_b
btautau,graviton_hh_ggf_bbtautau_m1250, file_types=pdf, plot_suffix=NO_STR)
INFO: luigi-interface - Informed scheduler that task   cf.PlotVariables1D_hbt_config_analy_0__1831bc169d   has status   DONE
INFO: luigi-interface -
===== Luigi Execution Summary =====

Scheduled 1 tasks of which:
* 1 ran successfully:
    - 1 cf.PlotVariables1D(...)

This progress looks :) because there were no failed tasks or missing dependencies

===== Luigi Execution Summary =====

naf-cms16:columnflow > |
```

- 2 Tools fo val / production
  - Config olumns
    at certain points of the workflow
  - Can be selected at execution time
  - umns,
    ackup)
- 3 C rs (CMS)
  - ctionlib
  - ag_sf, ...

Example: single and nested producers

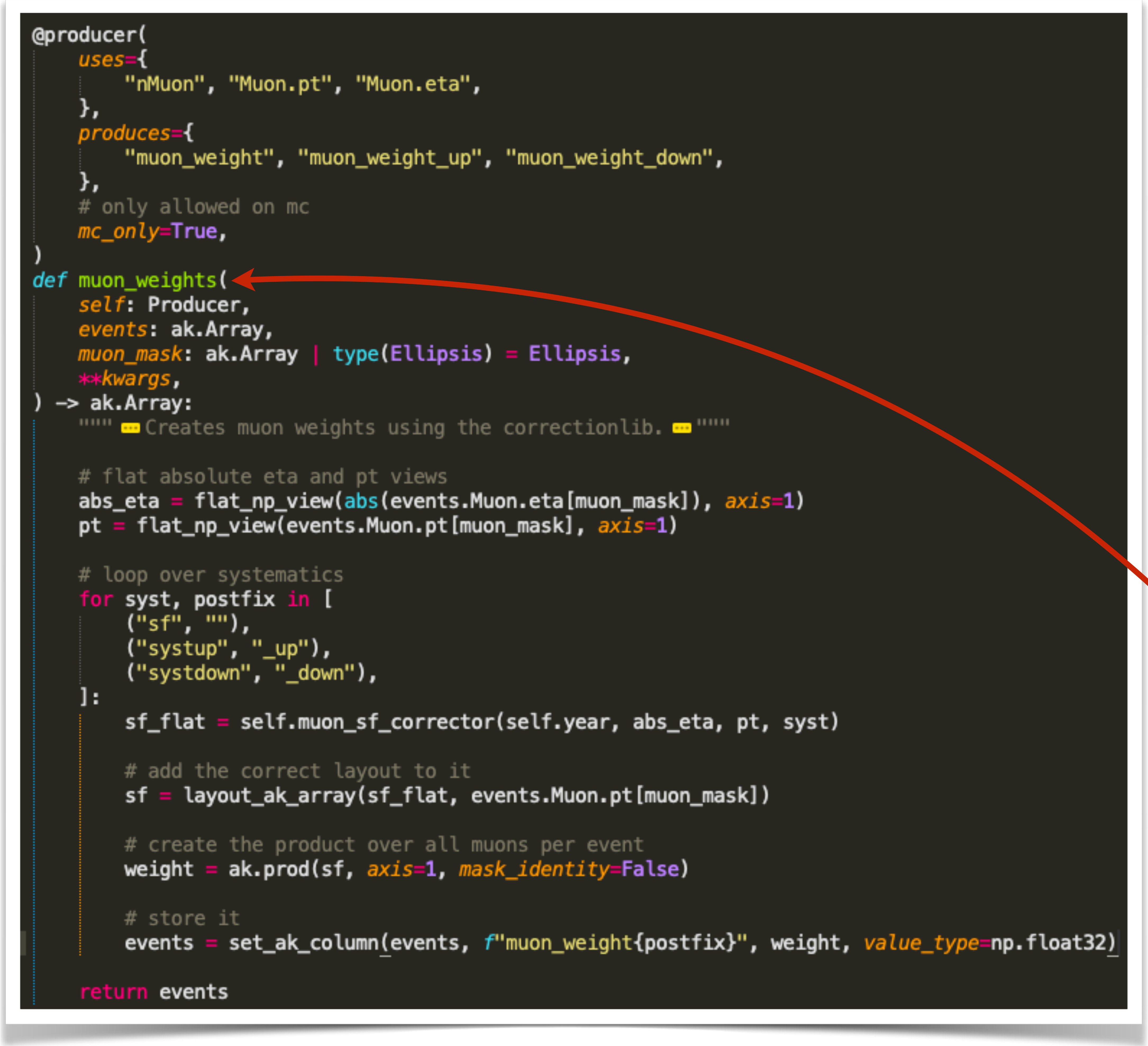
Single producer



```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
```

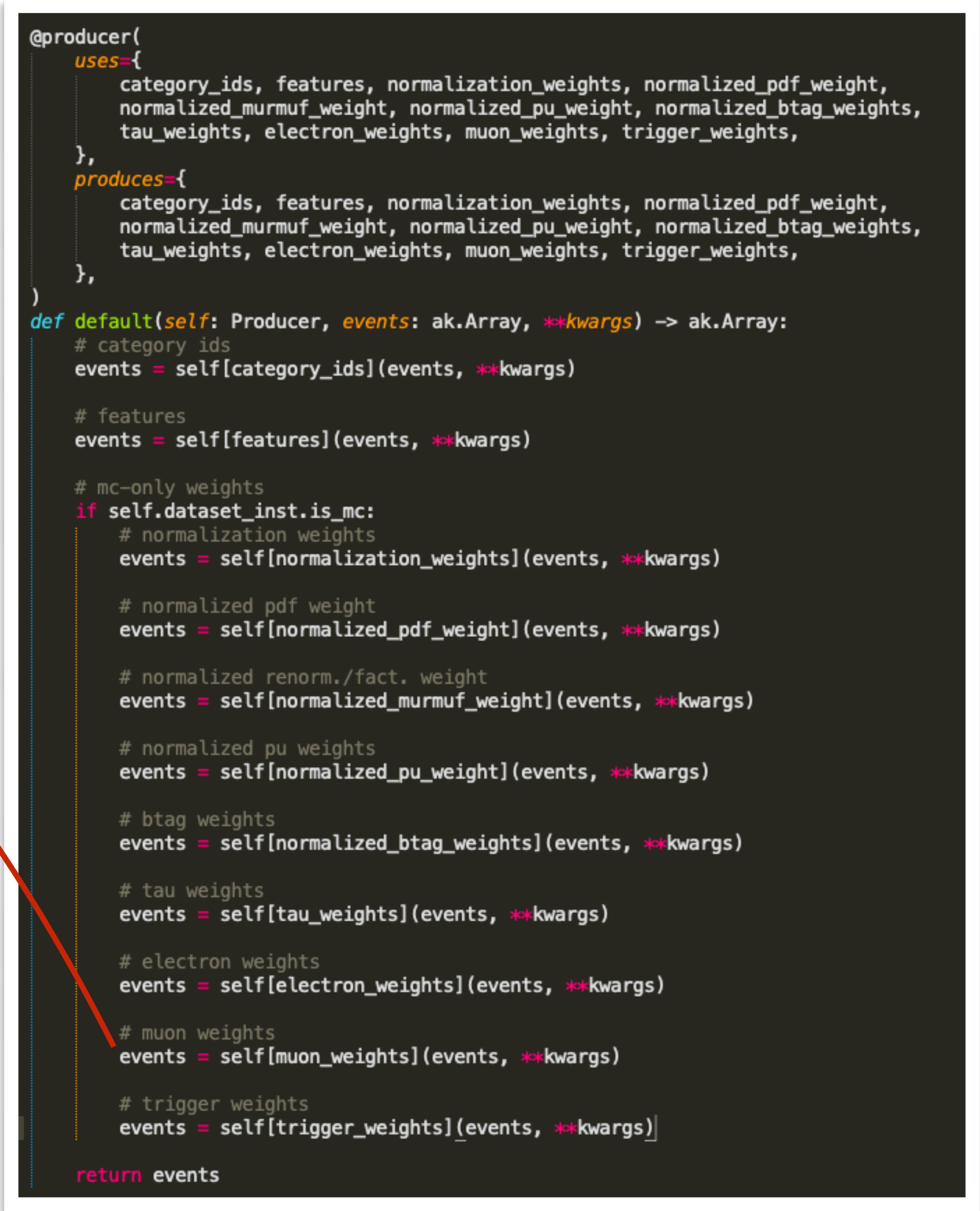
Example: single and nested producers

Single producer



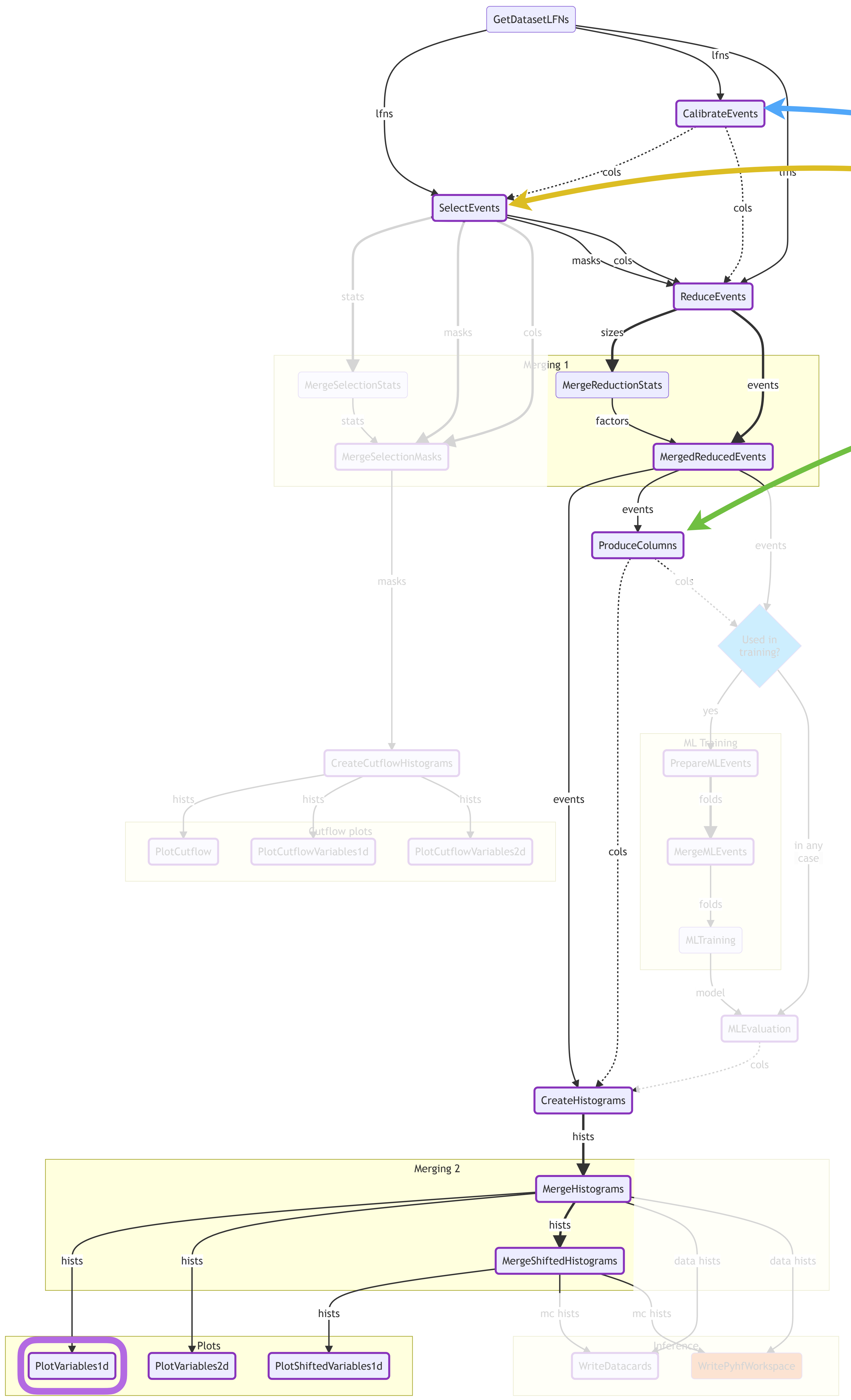
```
@producer(
    uses={
        "nMuon", "Muon.pt", "Muon.eta",
    },
    produces={
        "muon_weight", "muon_weight_up", "muon_weight_down",
    },
    # only allowed on mc
    mc_only=True,
)
def muon_weights(
    self: Producer,
    events: ak.Array,
    muon_mask: ak.Array | type(Ellipsis) = 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



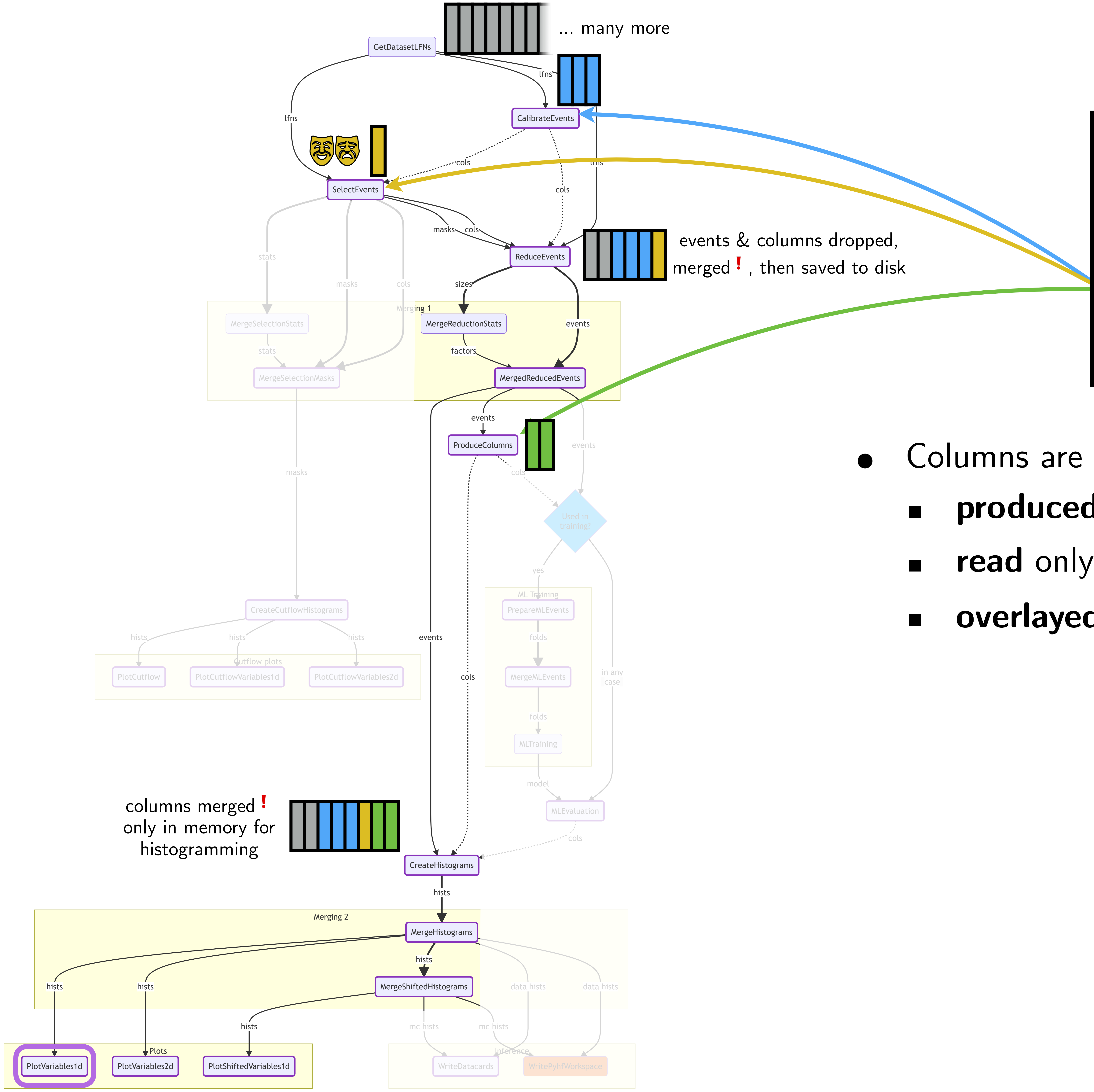
```
@producer(
    uses={
        category_ids, features, normalization_weights, normalized_pdf_weight,
        normalized_murmuf_weight, normalized_pu_weight, normalized_btag_weights,
        tau_weights, electron_weights, muon_weights, trigger_weights,
    },
    produces={
        category_ids, features, 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
```

flow of columns



... many more

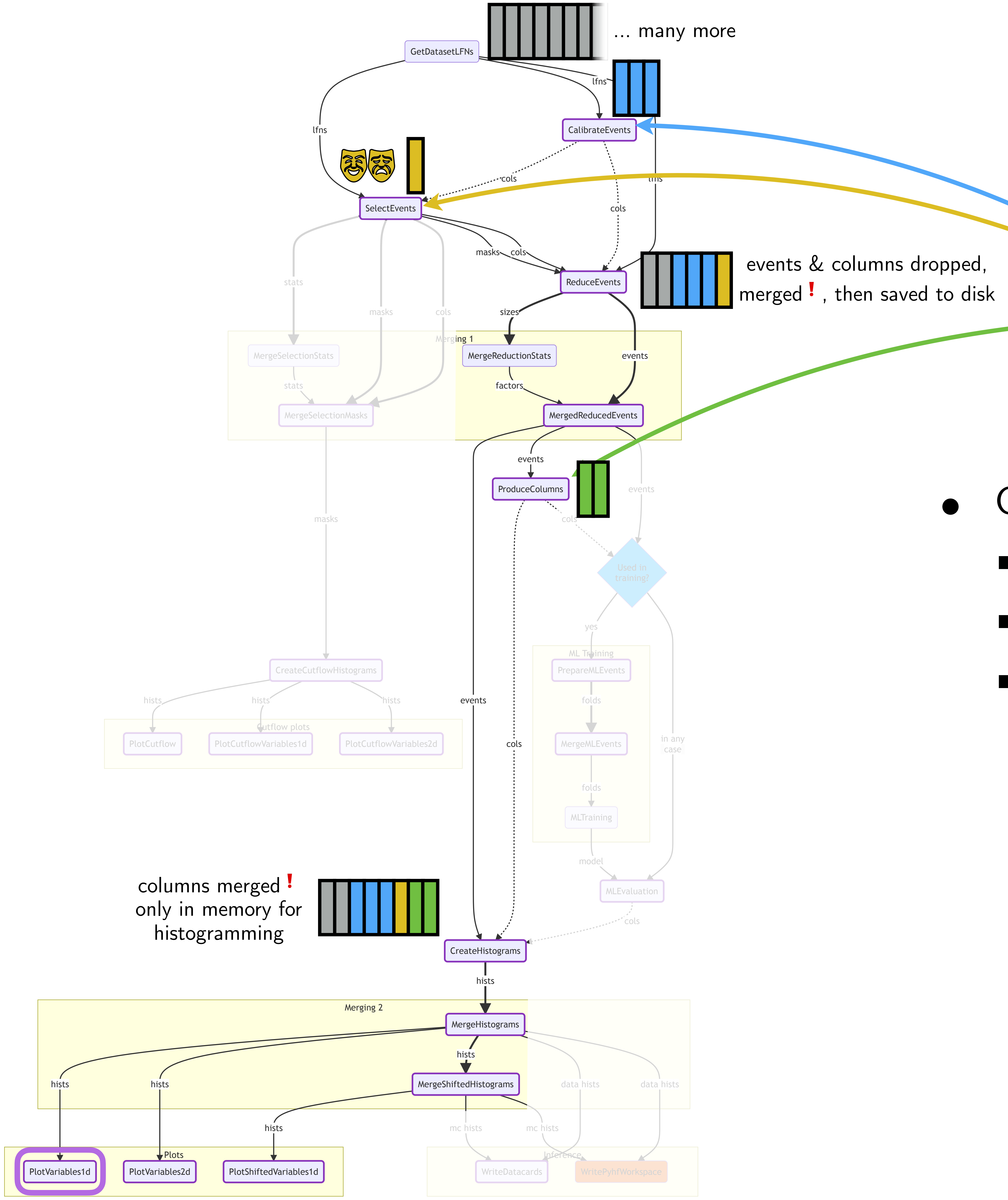
events & columns dropped, merged !, then saved to disk

columns merged !
only in memory for
histogramming

```
> 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



... many more

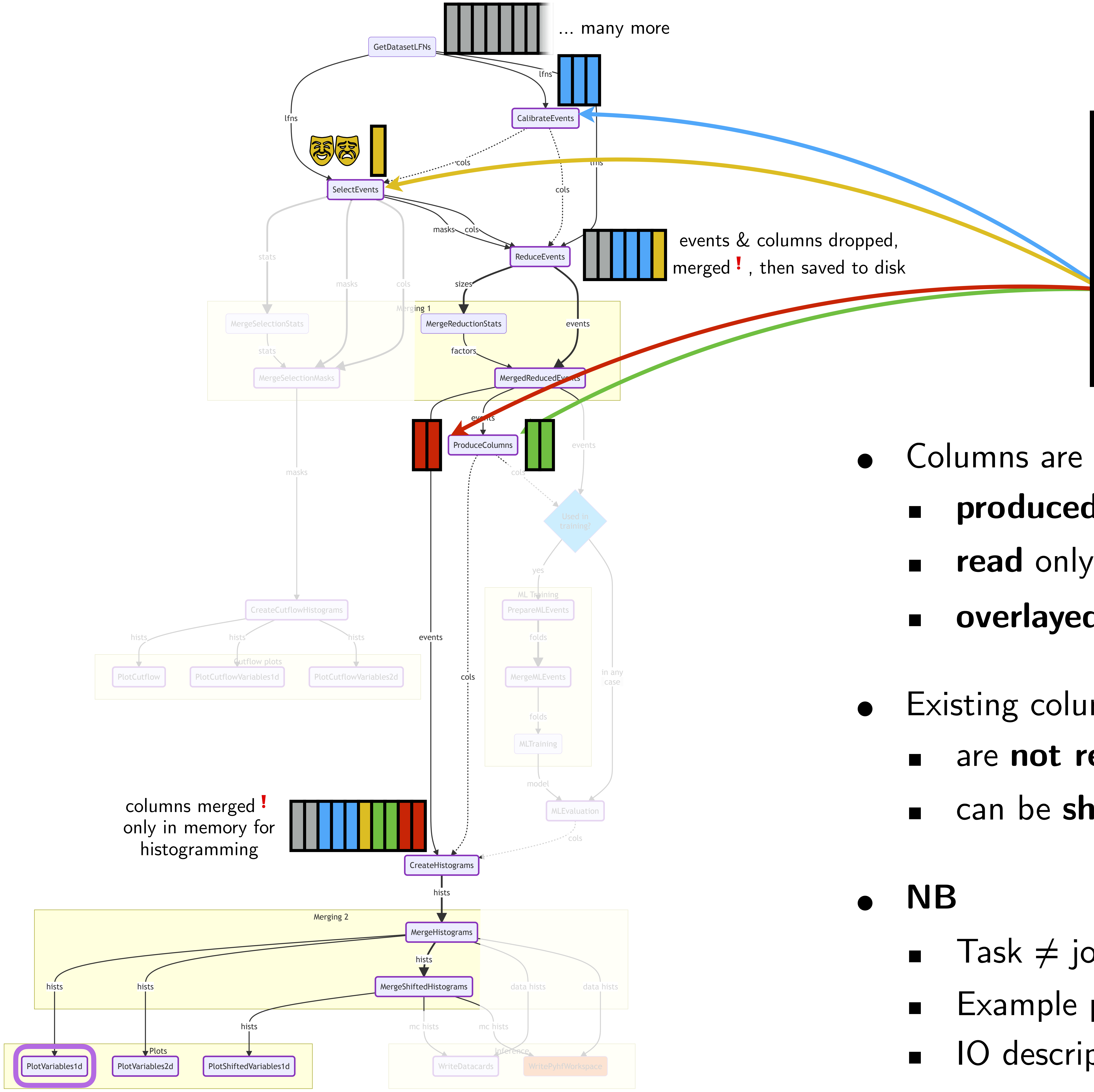
events & columns dropped, merged !, then saved to disk

columns merged ! only in memory for histogramming

```
> 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 !

flow of columns



... many more

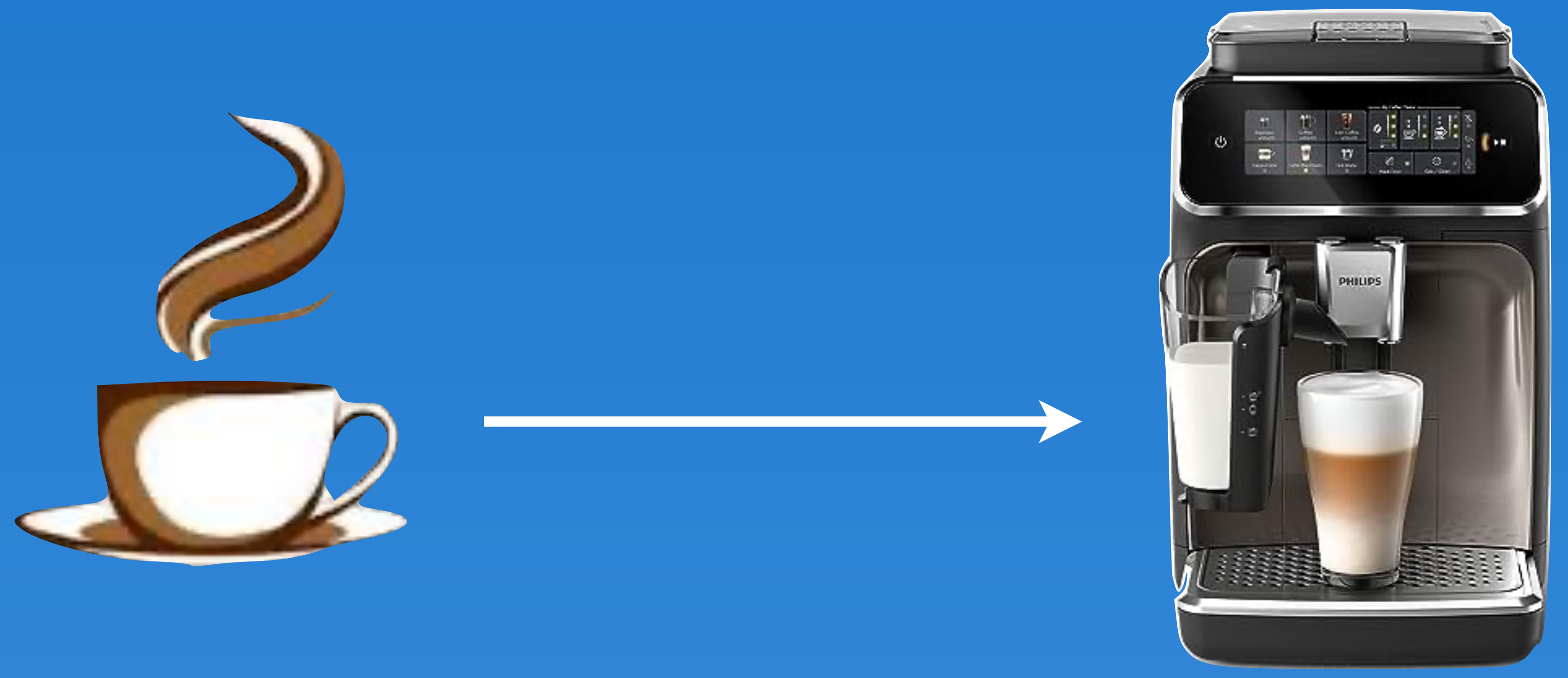
events & columns dropped, merged **!** , then saved to disk

columns merged **!** only in memory for histogramming

```
> 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

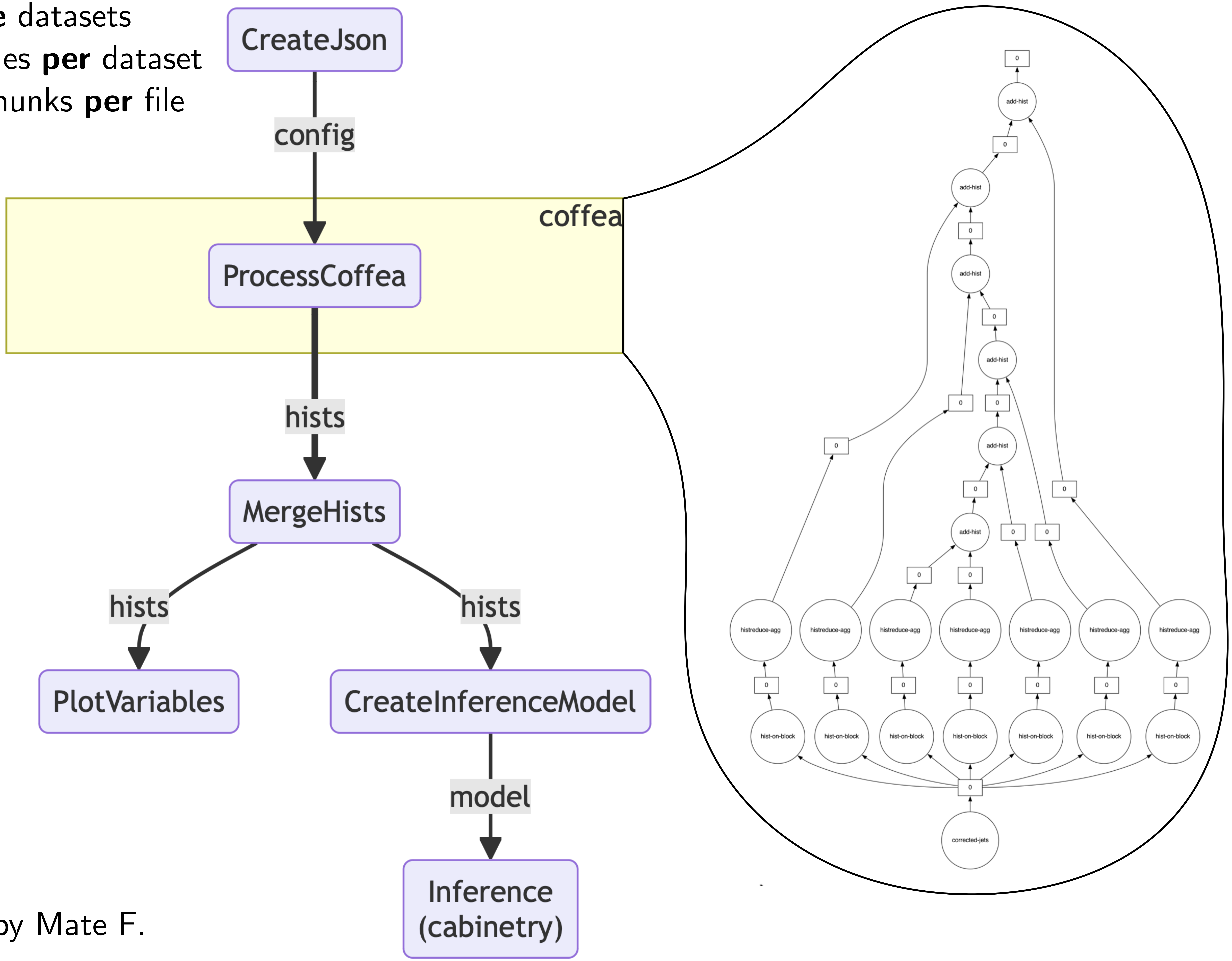
Example 2



(`coffea-maker` ?)

(`coffeaflow` ?)

- **multiple** datasets
- **many** files **per** dataset
- **many** chunks **per** file



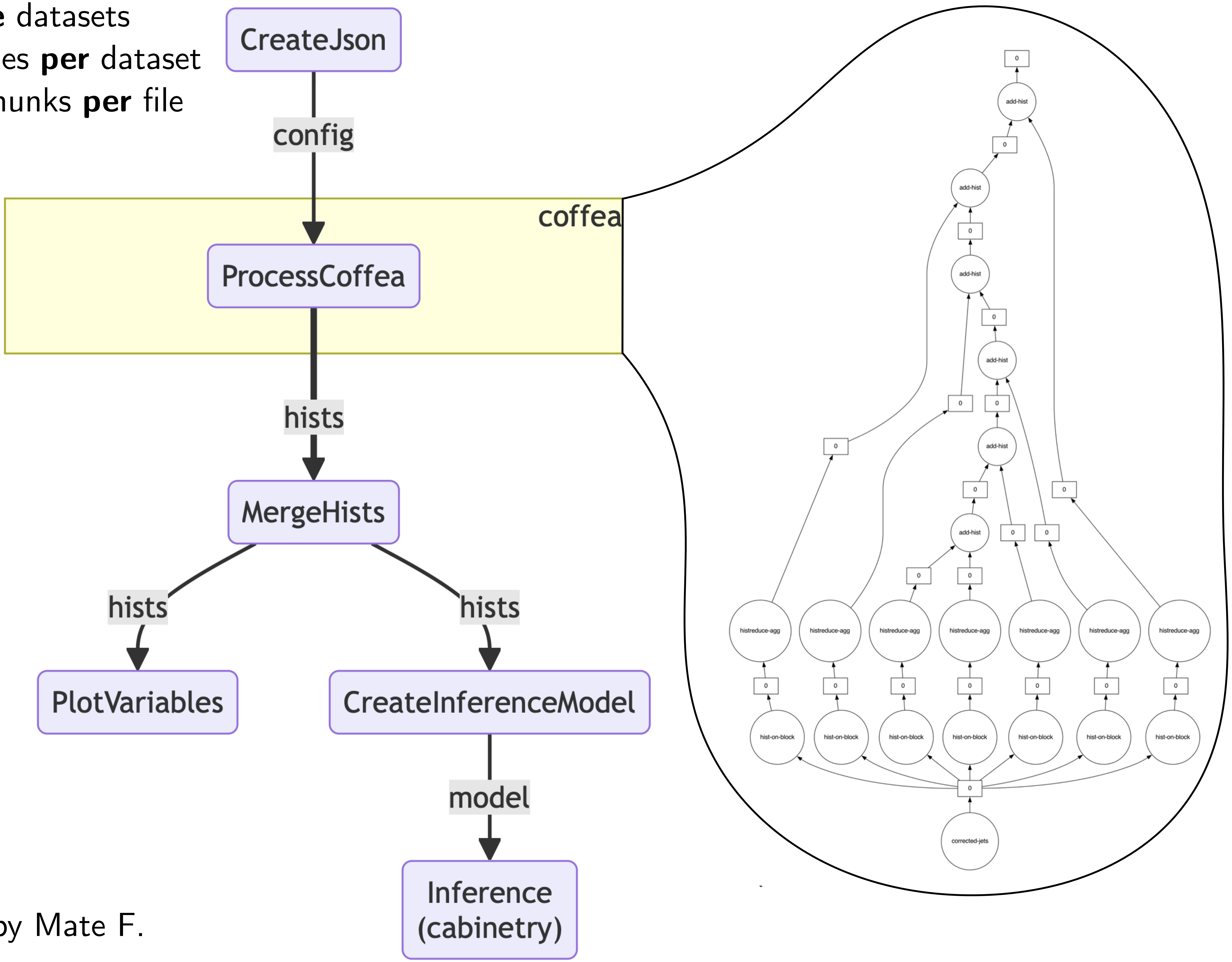
AGC implementation prototyped by Mate F.

Coffea in a (higher-level) workflow

- **multiple** datasets
- **many** files **per** dataset
- **many** chunks **per** file

Can depend on

```
--dataset NAME
--file-index I
--chunk-index J
...
```

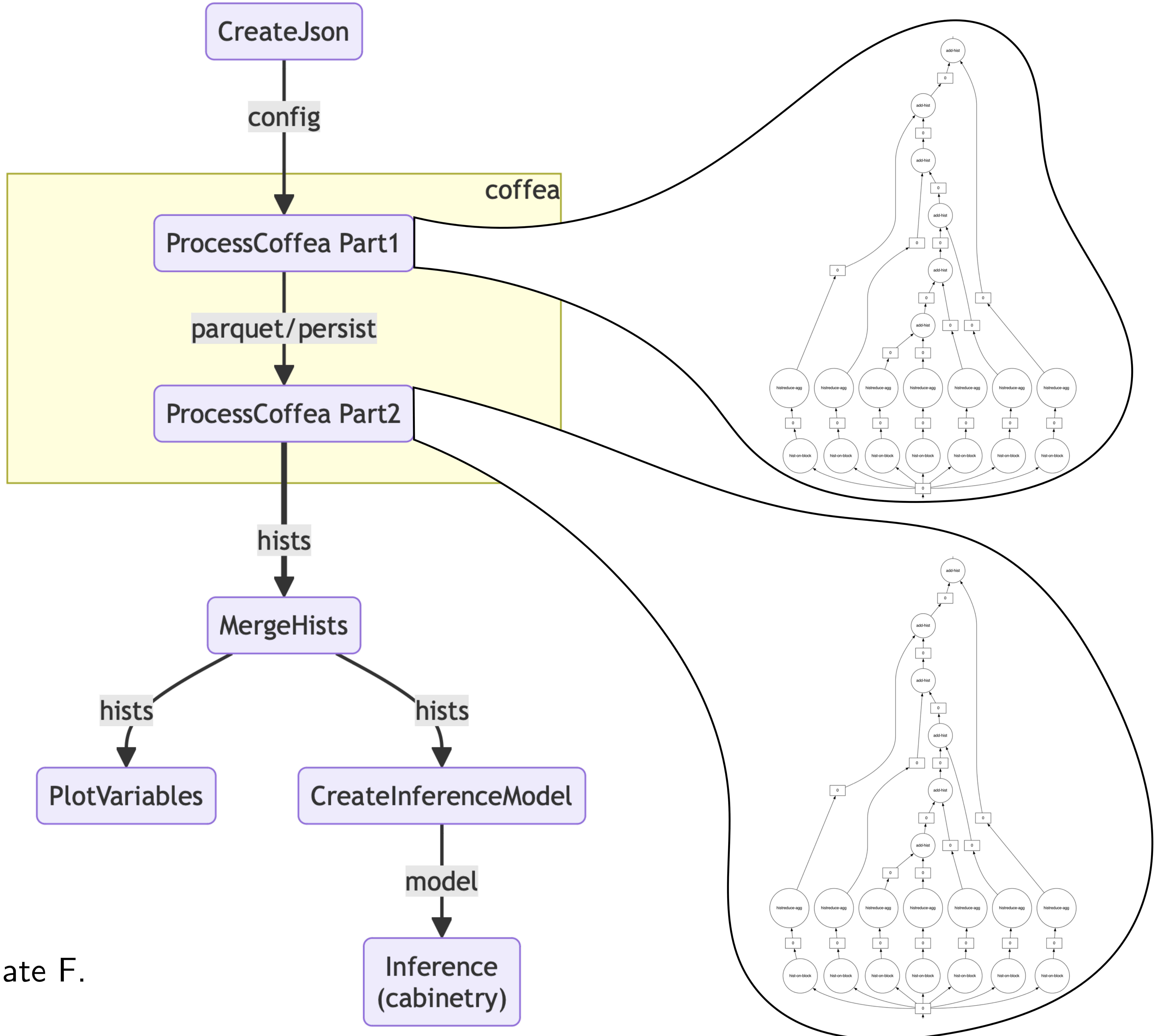


AGC implementation prototyped by Mate F.

Coffea in a (higher-level) workflow

Can depend on

```
--dataset NAME
--file-index I
--chunk-index J
...
```



AGC implementation prototyped by Mate F.

Backup

"Real-life considerations"

- **Example 1**
  - *Situation*
    - Definition of dependencies between "steps" must be highly flexible
    - E.g.: "Pileup in dataset **X** in era **Y** is known to be buggy, so use a different weight treatment"
  - *Consequence*

"Real-life considerations"

- **Example 1**
  - *Situation*
    - Definition of dependencies between "steps" must be highly flexible
    - E.g.: "Pileup in dataset **X** in era **Y** is known to be buggy, so use a different weight treatment"
  - *Consequence*
    - Modeling of the "dependency tree" must be highly customizable (*one might even say "hackable"*)
    - For people to gain trust in what they are doing, this should be possible in a language they already know ...

"Real-life considerations"

- **Example 1**
  - *Situation*
    - Definition of dependencies between "steps" must be highly flexible
    - E.g.: "Pileup in dataset **X** in era **Y** is known to be buggy, so use a different weight treatment"
  - *Consequence*
    - Modeling of the "dependency tree" must be highly customizable (*one might even say "hackable"*)
    - For people to gain trust in what they are doing, this should be possible in a language they already know ...

- **Example 2**
  - *Situation*
    - Be able to use $>$ **1** cluster, or large central batch systems (e.g. CRAB) (*avoiding single points of failure*)
    - Full automation, reprocessing only what's necessary
  - *Consequence*

"Real-life considerations"

- **Example 1**
  - *Situation*
    - Definition of dependencies between "steps" must be highly flexible
    - E.g.: "Pileup in dataset **X** in era **Y** is known to be buggy, so use a different weight treatment"
  - *Consequence*
    - Modeling of the "dependency tree" must be highly customizable (*one might even say "hackable"*)
    - For people to gain trust in what they are doing, this should be possible in a language they already know ...

- **Example 2**
  - *Situation*
    - Be able to use $> \mathbf{1}$ cluster, or large central batch systems (e.g. CRAB) (*avoiding single points of failure*)
    - Full automation, reprocessing only what's necessary
  - *Consequence*
    - Files need to be stored remotely as a necessity
    - Transfer between sites must be automatic and opportunistic (*otherwise it's not "automated"*)
    - Checking which files already exist must be highly performant (*to not send O(10k) network requests*)

"Real-life considerations"

- Example 1
  - *Situation*
    - ▻ Definition of dependencies between "steps" must be highly flexible

- **There is no free lunch**
  - Our HEP resources (clusters, grid, storage elements, software environments) are very **inhomogeneous**
  - A **realistic** workflow engine
    - ▻ can make some good, yet simple assumptions based on known best-practices

      BUT
    - ▻ it should **always** allow users to transparently **change decisions** & **configure every single aspect!**

  - *Consequence*
    - ▻ Files need to be stored remotely as a necessity
    - ▻ Transfer between sites must be automatic and opportunistic (*otherwise it's not "automated"*)
    - ▻ Checking which files already exist must be highly performant (*to not send O(10k) network requests*)

Other "workflow" engines

RAY + dask

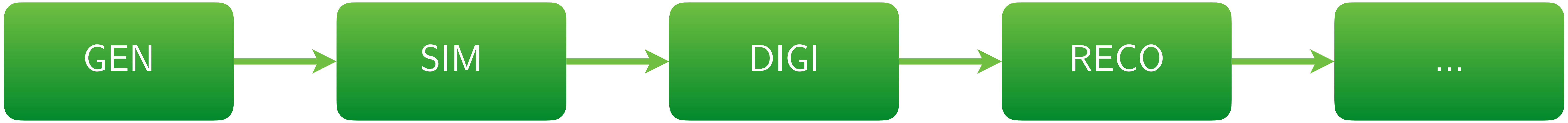
Apache Airflow

snakemake

luigi

- **Metrics for comparison**
  - Low-level array processing vs. high-level embedding
  - Pythonic usage
  - Usage Overhead (requires a DB, server, custom hardware, ...)
  - Built-in features
  - Configurability
  - ...

Existing WMS: MC production



Tailored systems

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

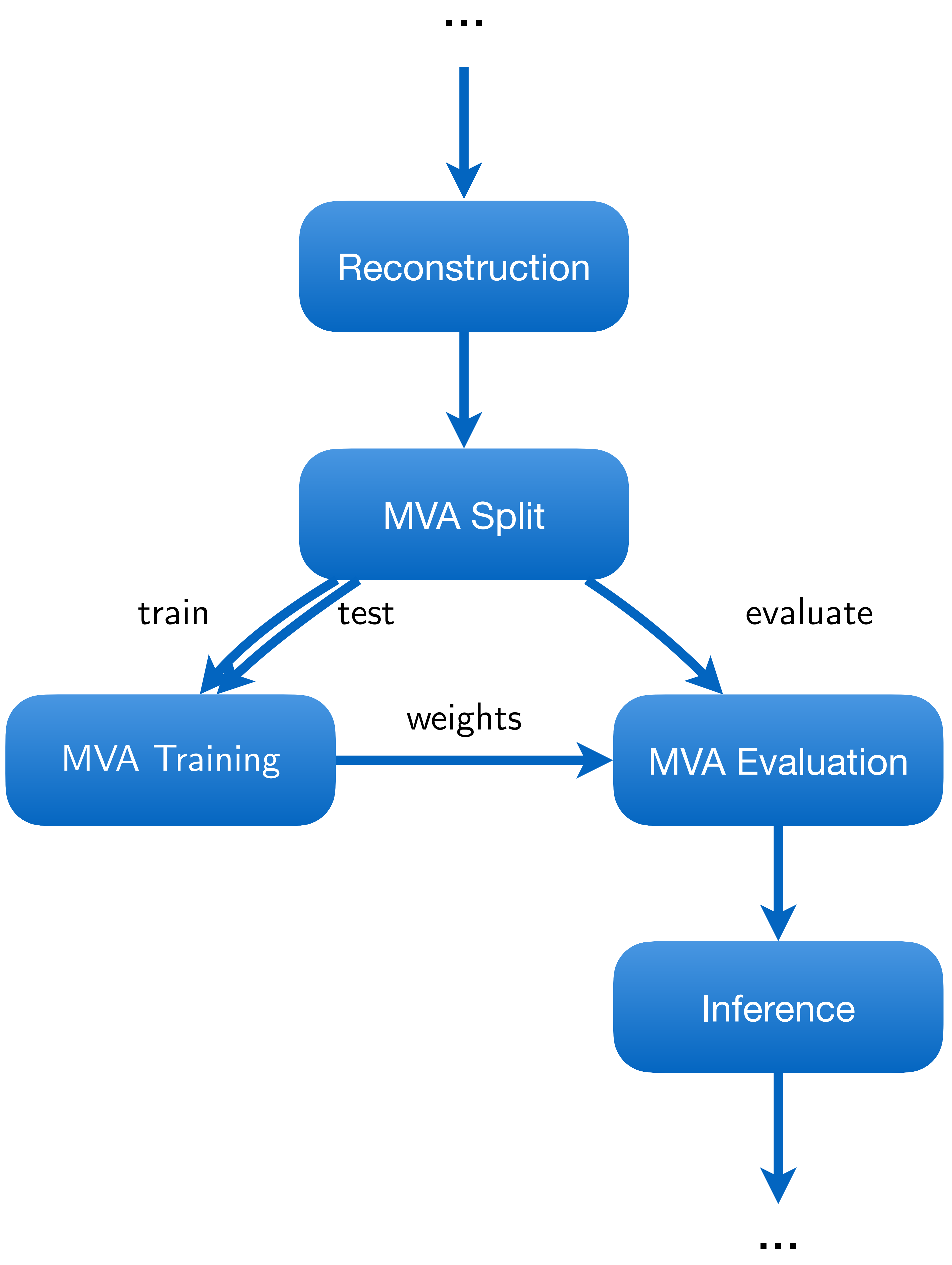
Wishlist for end-user analyses

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

→ Requirements for HEP analyses mostly orthogonal

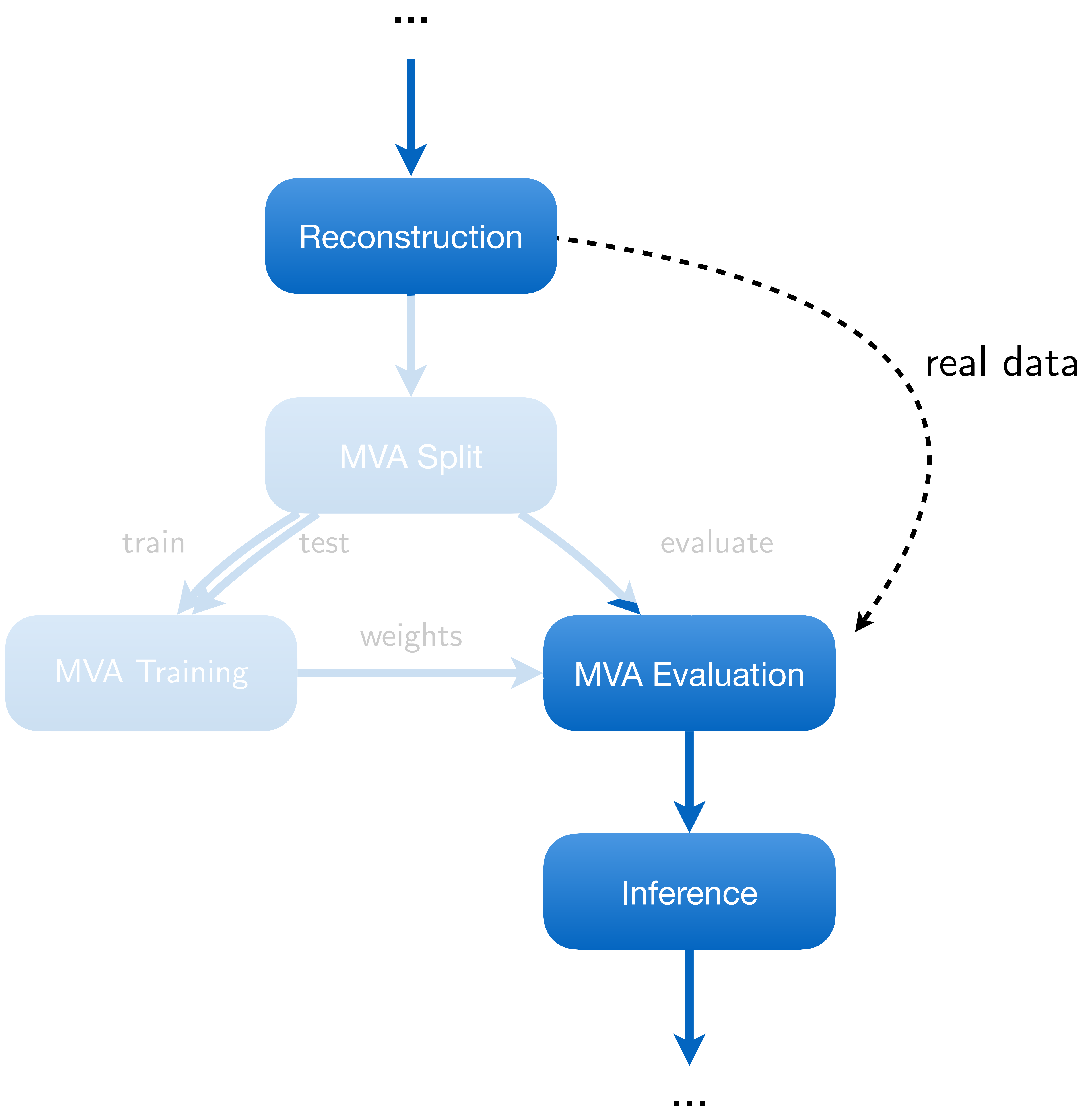
A typical example: ML workflow with uncertainties

Nominal MC



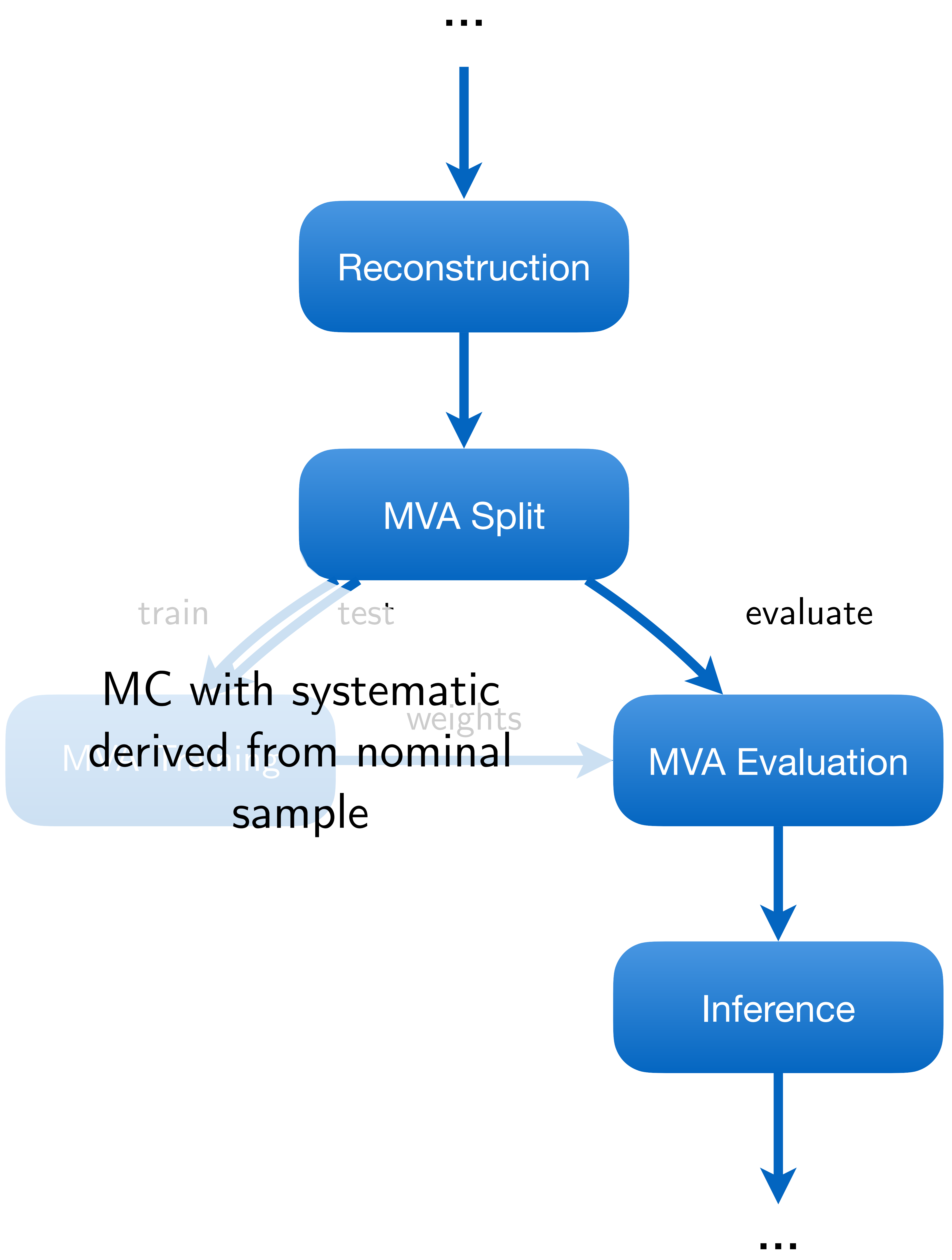
A typical example: ML workflow with uncertainties

Data



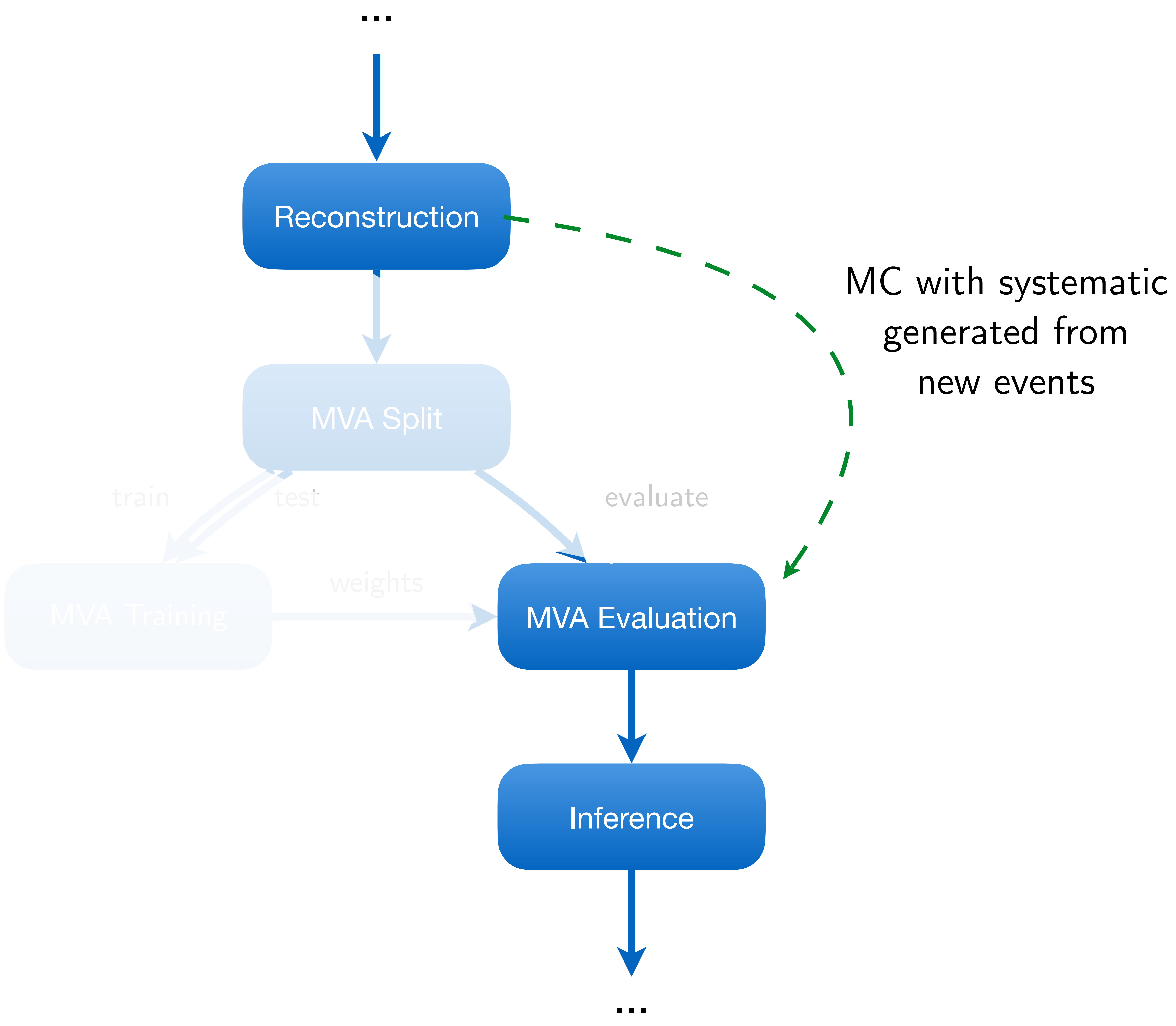
A typical example: ML workflow with uncertainties

MC, Syst. I

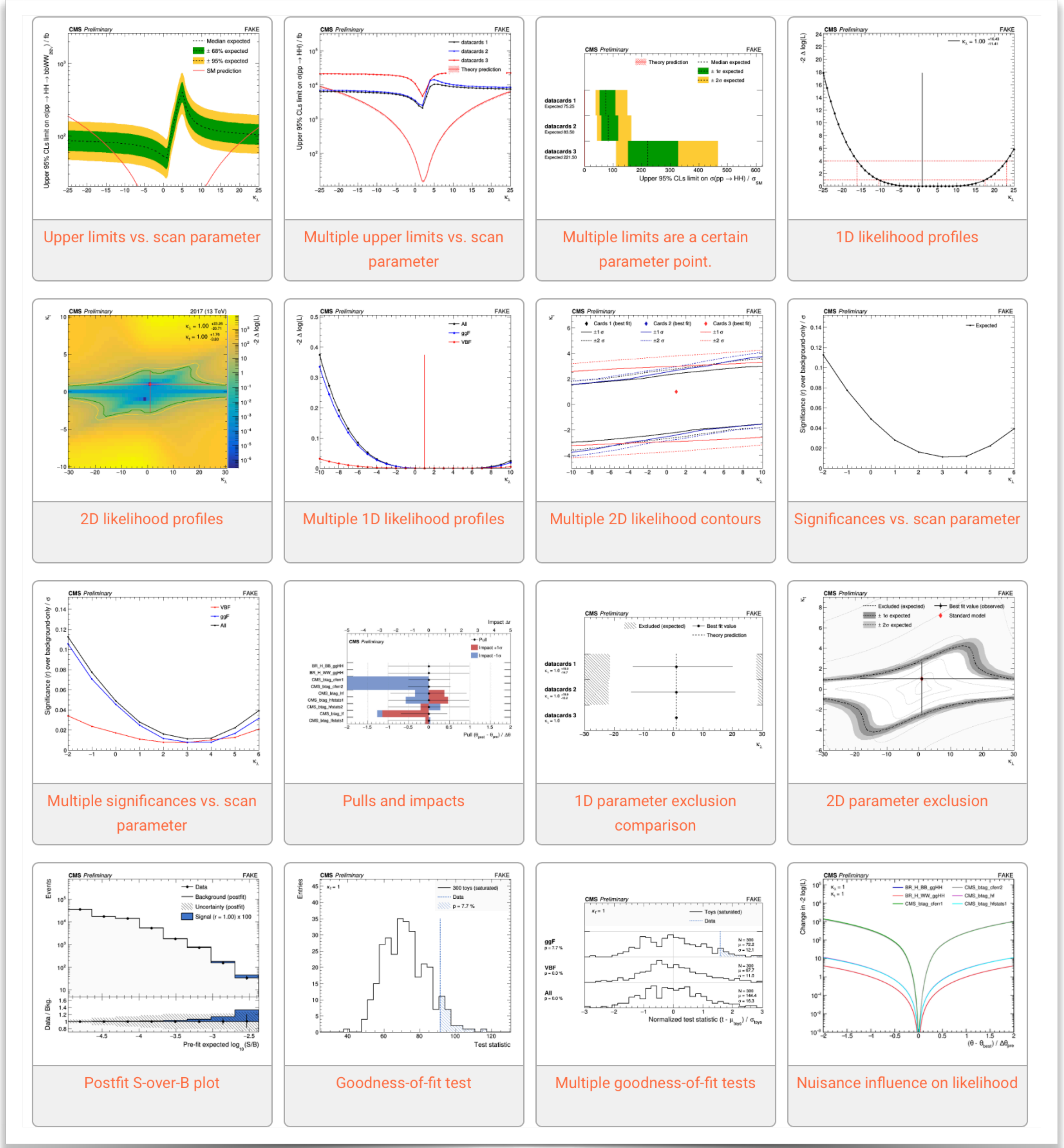


MC with systematic derived from nominal sample

MC, Syst. II



MC with systematic generated from new events



Upper limits vs. scan parameter

Multiple upper limits vs. scan parameter

Multiple limits are a certain parameter point.

1D likelihood profiles

2D likelihood profiles

Multiple 1D likelihood profiles

Multiple 2D likelihood contours

Significances vs. scan parameter

Multiple significances vs. scan parameter

Pulls and impacts

1D parameter exclusion comparison

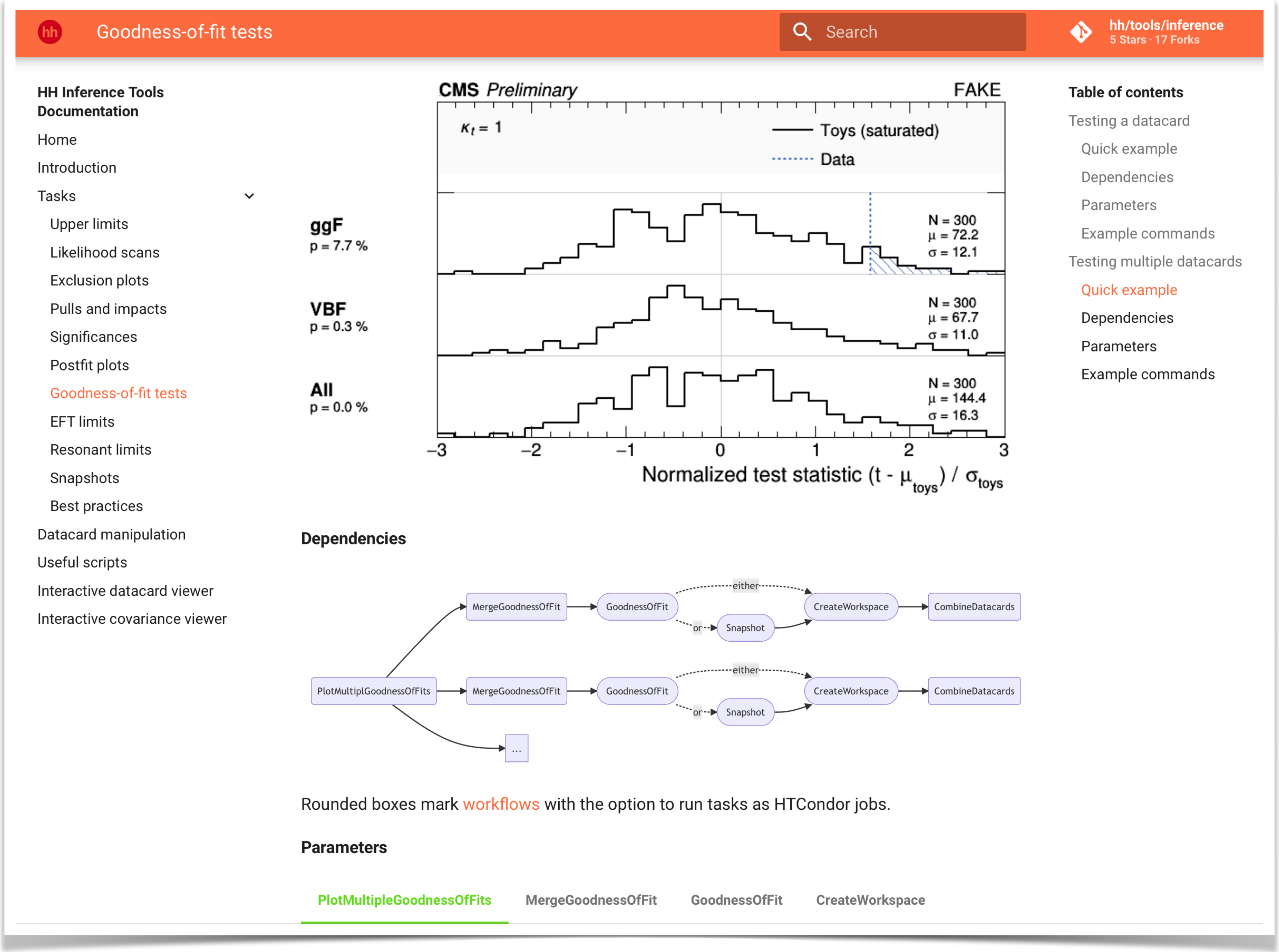
2D parameter exclusion

Postfit S-over-B plot

Goodness-of-fit test

Multiple goodness-of-fit tests

Nuisance influence on likelihood



Goodness-of-fit tests

Search

hh/tools/inference
5 Stars · 17 Forks

HH Inference Tools Documentation

- Home
- Introduction
- Tasks
  - Upper limits
  - Likelihood scans
  - Exclusion plots
  - Pulls and impacts
  - Significances
  - Postfit plots
  - Goodness-of-fit tests
  - EFT limits
  - Resonant limits
  - Snapshots
  - Best practices
- Datacard manipulation
- Useful scripts
- Interactive datacard viewer
- Interactive covariance viewer

Dependencies

Rounded boxes mark workflows with the option to run tasks as HTCondor jobs.

Parameters

PlotMultipleGoodnessOfFits | MergeGoodnessOfFit | GoodnessOfFit | CreateWorkspace

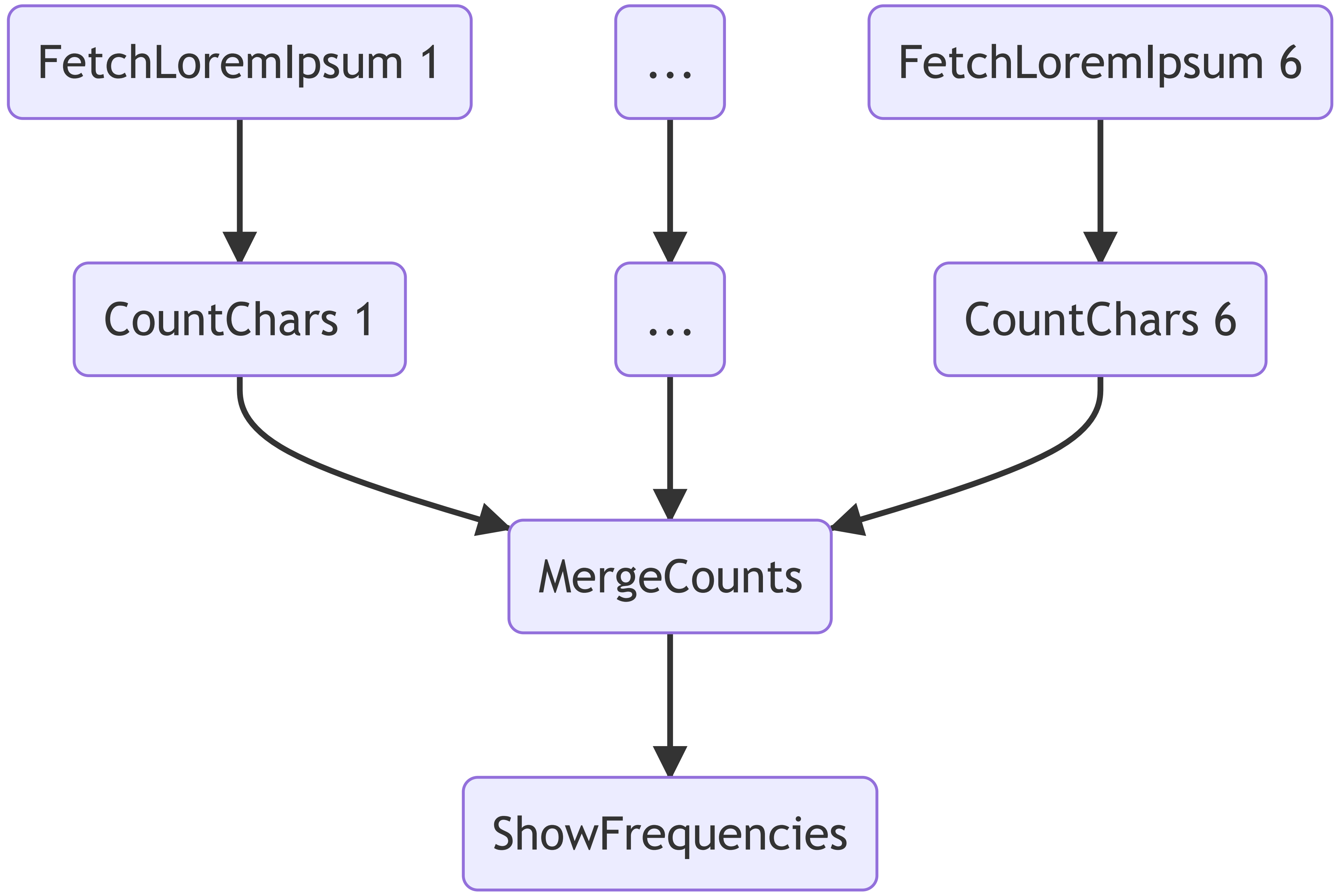
Table of contents

- Testing a datacard
  - Quick example
  - Dependencies
  - Parameters
  - Example commands
- Testing multiple datacards
  - Quick example
  - Dependencies
  - Parameters
  - Example commands

cms-hh.web.cern.ch/cms-hh/tools/inference

Hands-on!

- 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



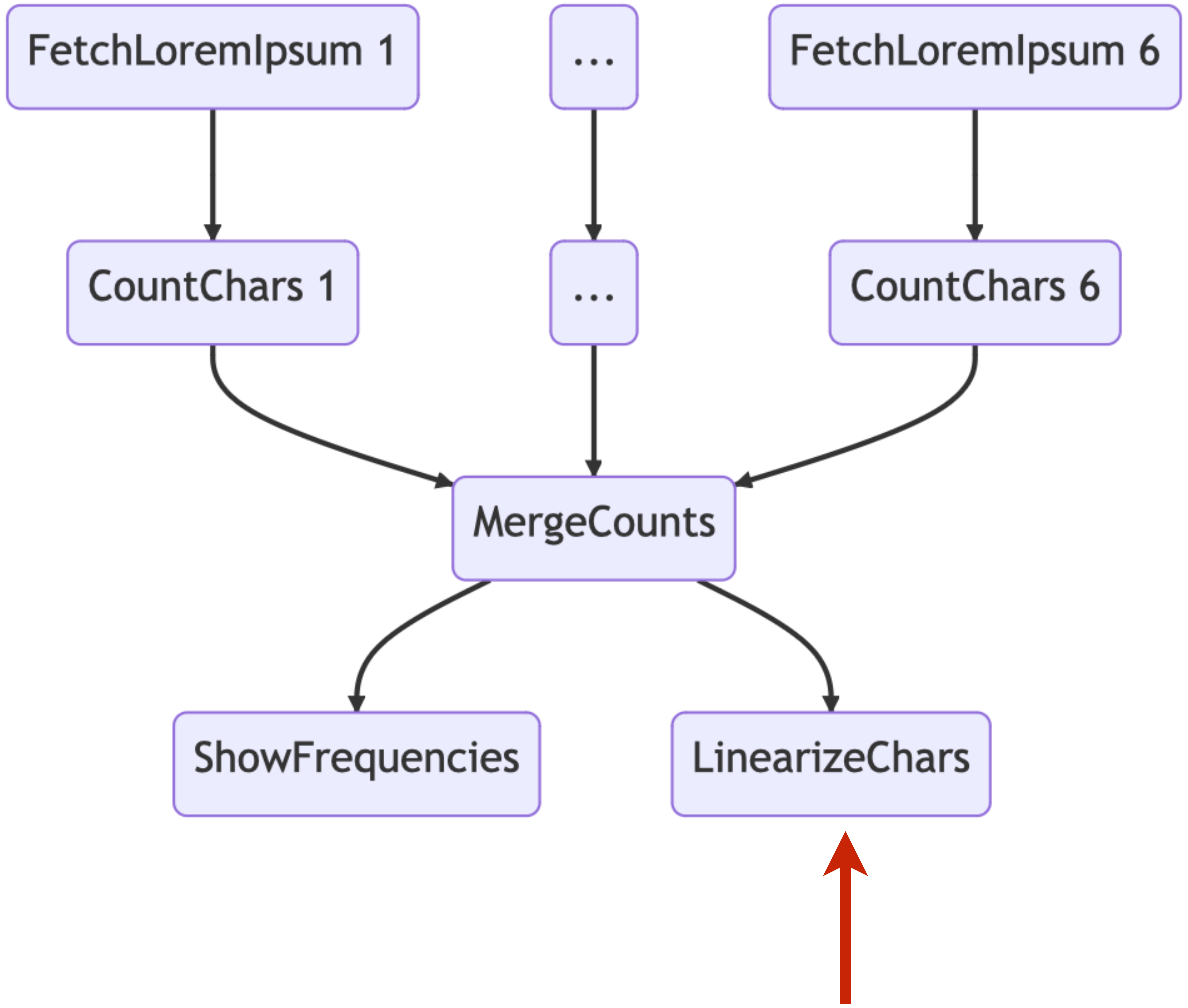
(graphs via mermaid.live)

- launch binder for the notebook version

- Additional example: Workflow using CERN HTCondor

Things to try

- **Interactive parameters**
  - Append `--print-status RECURSION_LEVEL[,TARGET_LEVEL]`
  - Append `--print-deps RECURSION_LEVEL`
  - Append `--remove-output RECURSION_LEVEL[,MODE],[RESTART]`
  - Append `--fetch-output RECURSION_LEVEL[,MODE],[DIRECTORY]`
- **Parallelize**
  - Append `--workers 4`
- **Add a task**
  - LinearizeChars
    - Create an ordered string "aaaaabbbccdddeeeeeeeeee..." from all existing characters and save it in a text file



Workflows: General ideas

- **Many tasks exhibit the same overall structure and/or purpose**
  - *"Run over N existing files" / "Generate N events/toys" / "Merge N into M files"*
  - All these tasks can **profit from the same features**
    - ▻ *"Only process file x and/to y", "Remove outputs of "x, y & z",*
      "Process N files, but consider the task finished once M < N are done", "..."

→ Calls for a generic container object that provides guidance and features for these cases

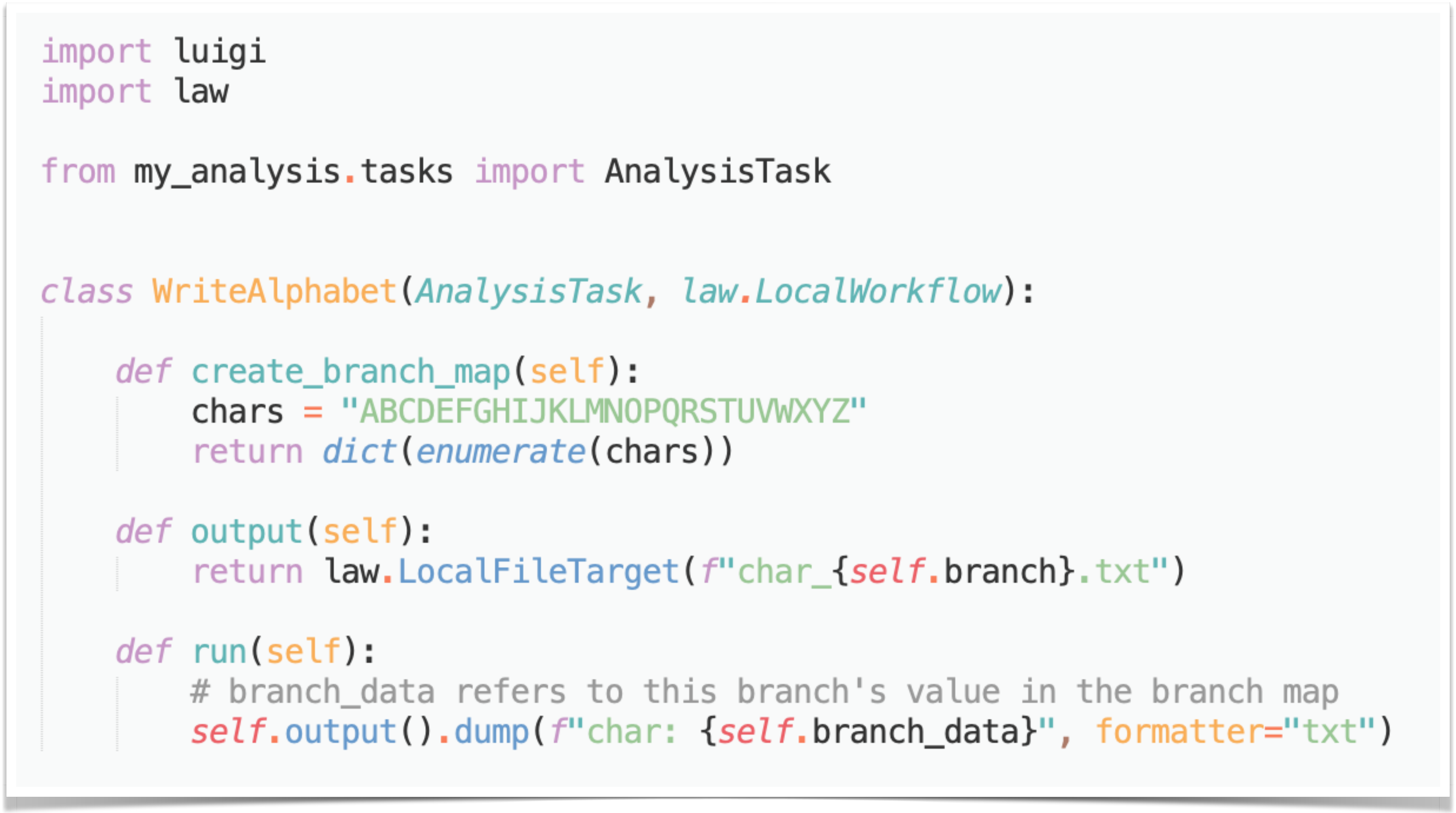
- **Workflow "containers"**
  - Task that introduces a parameters called `--branch b` (`luigi.IntParameter`)
    - ▻ `b >= 0`: Instantiates particular tasks called "branches"; `run()` will (e.g.) process file `b`
    - ▻ `b = -1`: Instantiates the workflow container itself; `run()` will run* **all branch tasks**

    * How branch tasks are run is implemented in different workflow types: **local** or several **remote ones**

- **Practical advantages**
  - Convenience: same features available in all workflows (see next slides)
  - **Scalability and versatility for remote workflows**
    - ▻ Jobs: Better control of jobs, submission, task-to-job matching ... (see next slides)
    - ▻ Luigi: Central scheduler breaks when pinged by O(10k) tasks every few seconds
    - ▻ Remote storage: Allows batched file operations instead of file-by-file requests

Workflows: example usage

- Tasks that each write a single character into a text file
- Character assigned to them though the branch map as their "branch data"



```python
import luigi
import law

from my_analysis.tasks import AnalysisTask


class WriteAlphabet(AnalysisTask, law.LocalWorkflow):

    def create_branch_map(self):
        chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        return dict(enumerate(chars))

    def output(self):
        return law.LocalFileTarget(f"char_{self.branch}.txt")

    def run(self):
        # branch_data refers to this branch's value in the branch map
        self.output().dump(f"char: {self.branch_data}", formatter="txt")
```

Workflows: remote workflows & jobs

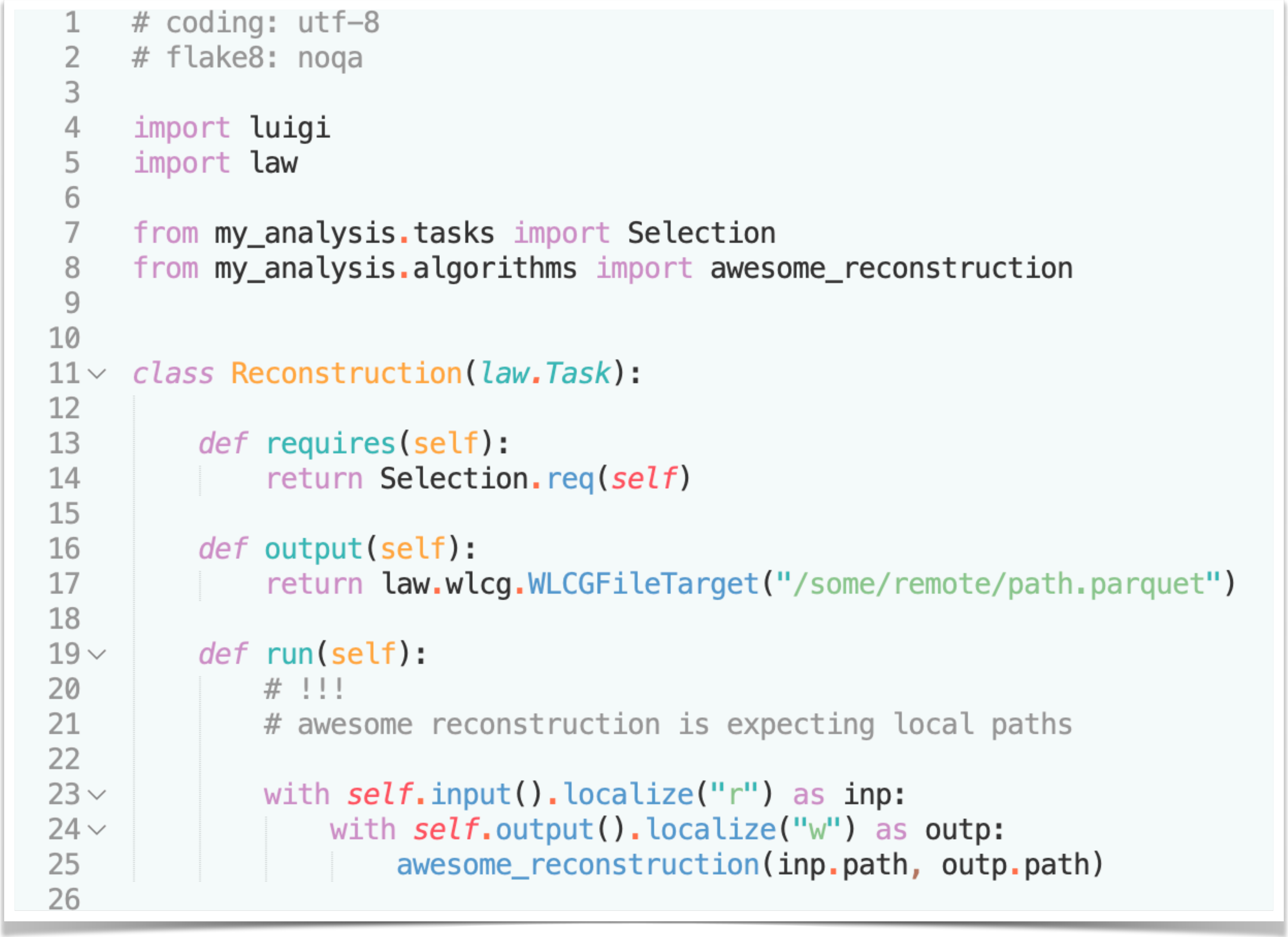
- **6 remote workflow implementations come with law**
  - htcondor, glite, lsf, arc, slurm, cms-crab (in PR#150)
  - Based on generic "job manager" implementations in contrib packages

- **Job managers fully decoupled from most law functionality**
  - Simple extensibility
  - No "auto-magic" in submission files, rather minimal and configurable through tasks
  - Usable also without law

- **Most important features**
  - Job submission functionality "declared" via task class inheritance
  - Provision of software and job-specific requirements through `workflow_requires()`
  - Control over remote jobs through parameters:

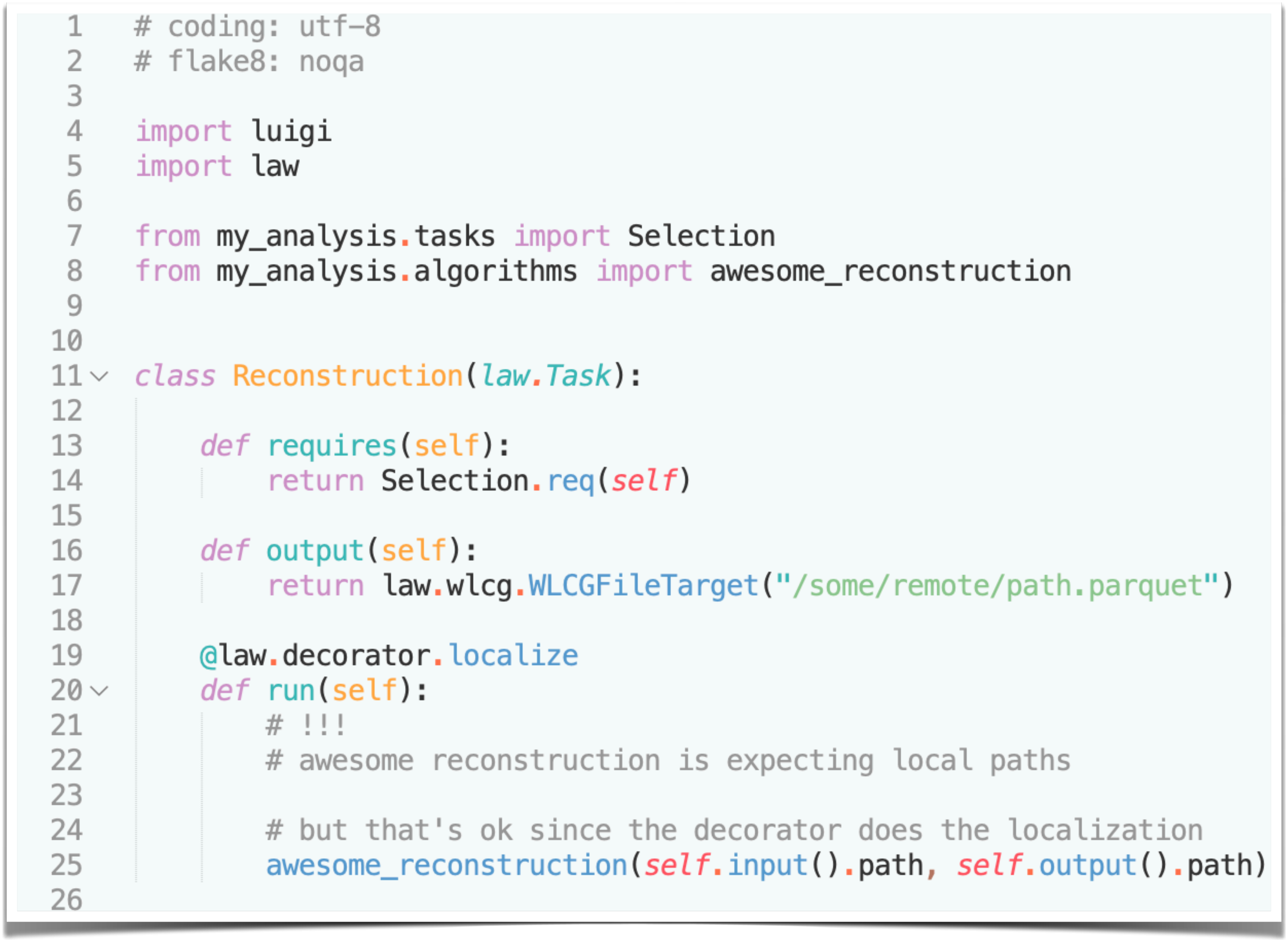
▻	`--branch`	`--branches`	:	granular control of which tasks to process
▻	`--acceptance`	`--tolerance`	:	defines when a workflow is complete / failed
▻	`--poll-interval`	`--walltime`	:	controls the job status polling interval and runtime
▻	`--tasks-per-job`	`--parallel-jobs`	:	control of resource usage at batch systems

Effective remote targets — "Localization"



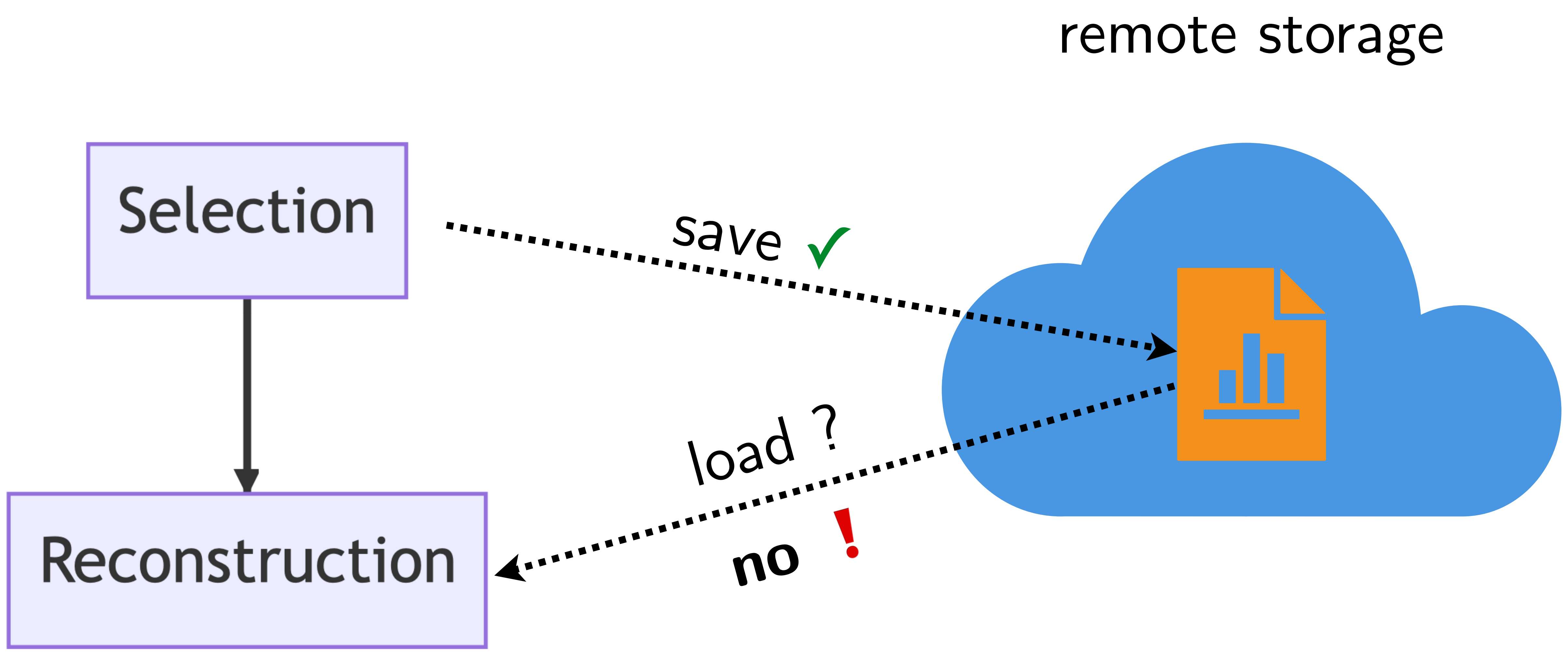
```python
# coding: utf-8
# flake8: noqa

import luigi
import law

from my_analysis.tasks import Selection
from my_analysis.algorithms import awesome_reconstruction


class Reconstruction(law.Task):

    def requires(self):
        return Selection.req(self)

    def output(self):
        return law.wlcg.WLCGFileTarget("/some/remote/path.parquet")

    def run(self):
        # !!!
        # awesome reconstruction is expecting local paths

        with self.input().localize("r") as inp:
            with self.output().localize("w") as outp:
                awesome_reconstruction(inp.path, outp.path)

```

Effective remote targets — "Localization"

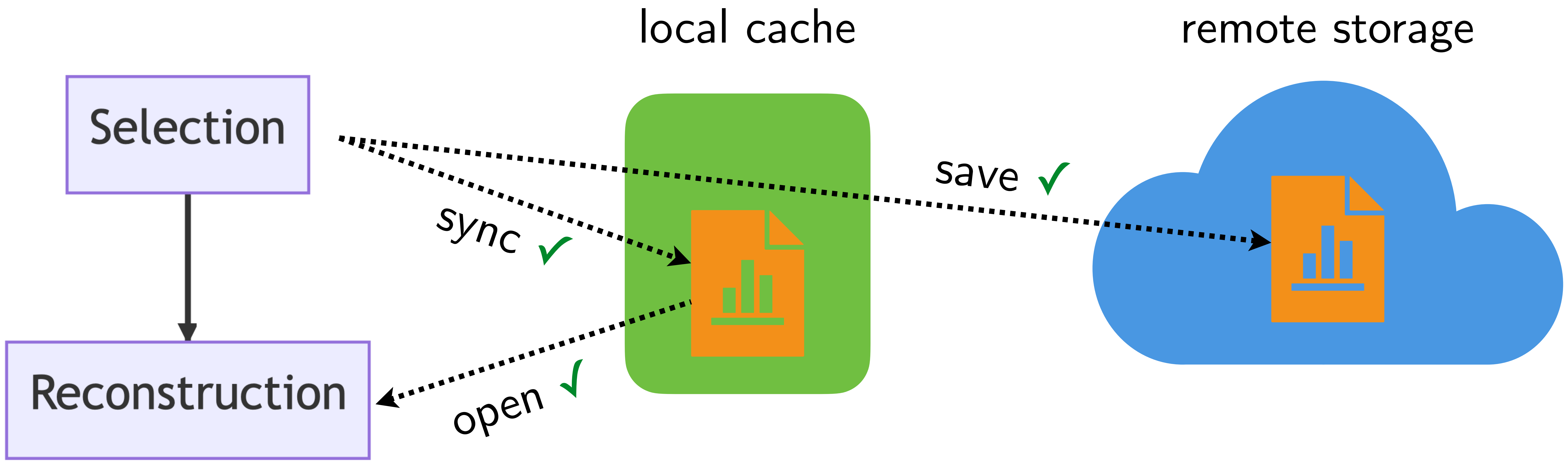


```python
# coding: utf-8
# flake8: noqa

import luigi
import law

from my_analysis.tasks import Selection
from my_analysis.algorithms import awesome_reconstruction


class Reconstruction(law.Task):

    def requires(self):
        return Selection.req(self)

    def output(self):
        return law.wlcg.WLCGFileTarget("/some/remote/path.parquet")

    @law.decorator.localize
    def run(self):
        # !!!
        # awesome reconstruction is expecting local paths

        # but that's ok since the decorator does the localization
        awesome_reconstruction(self.input().path, self.output().path)

```

- **Local cache for remote targets**

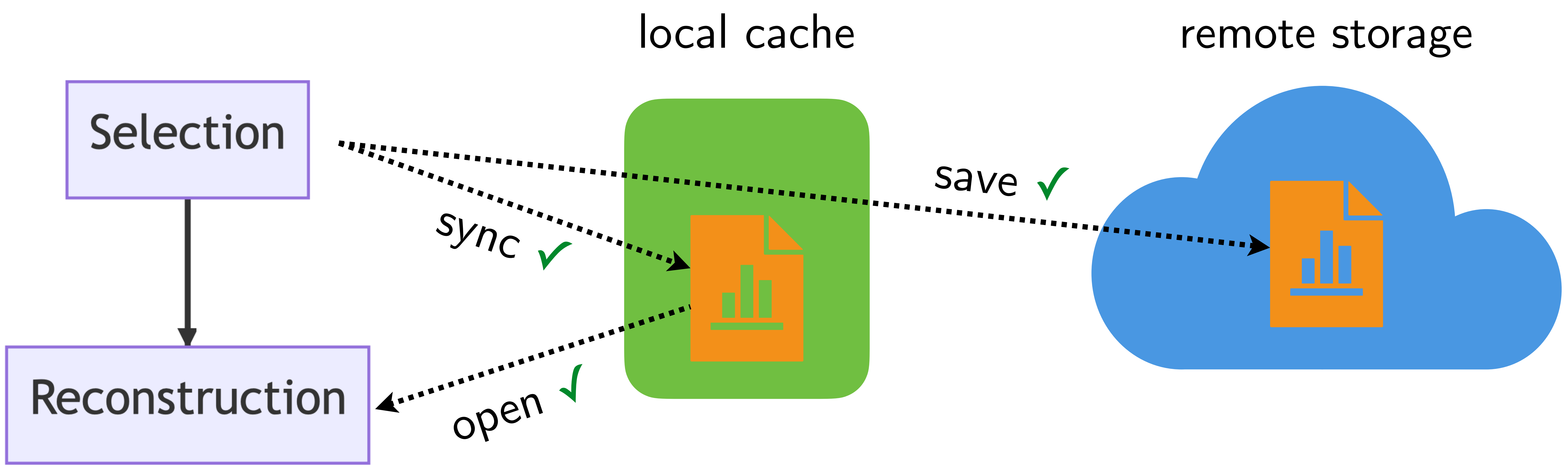


- **Local cache for remote targets**



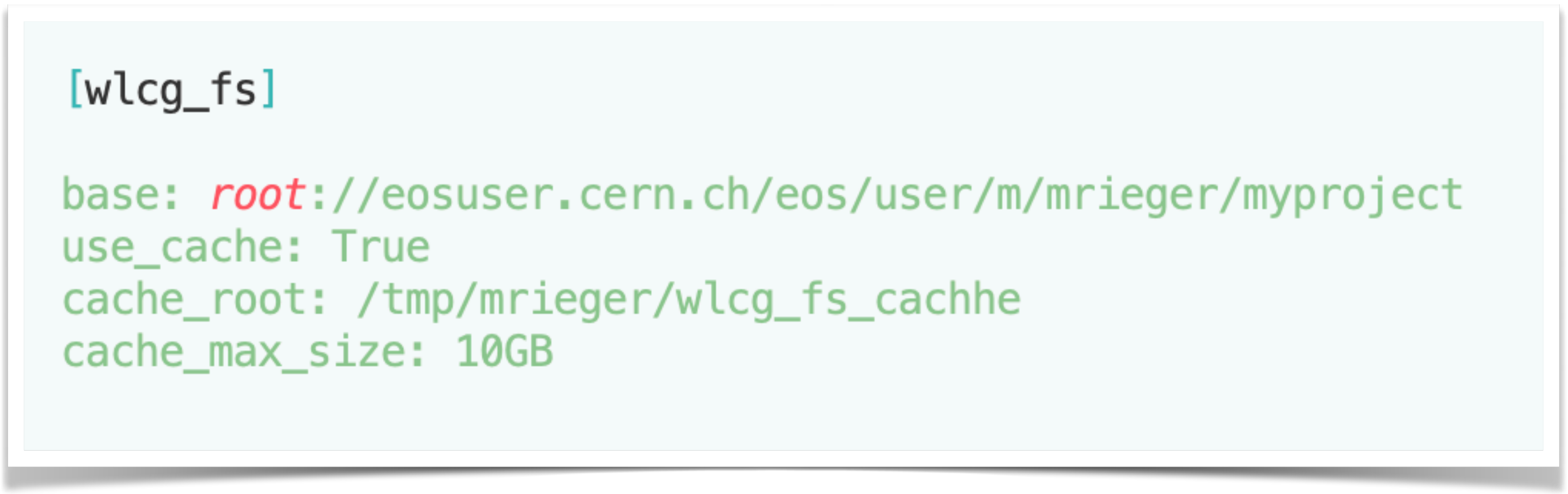
Effective remote targets — Caching

- **Local cache for remote targets**



- **Simple configuration**
  - When enabled, all operations on remote targets are cached

law.cfg



```
[wlcg_fs]

base: root://eosuser.cern.ch/eos/user/m/mrieger/myproject
use_cache: True
cache_root: /tmp/mrieger/wlcg_fs_cachhe
cache_max_size: 10GB
```

"Realistic" HEP workflow management

- **Consider this example again**

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

  - $\mathcal{O}$(500 - 4k) files, stored either locally or remotely
  - Any workflow engine will first check if things need to be rerun
    - $\mathcal{O}$(500 - 4k) file requests (**via network**)!
    - Prepare for admins to find you 👀
  - *What* law *does*
    - Reconstruction is a workflow
    - Workflows output a so-called **TargetCollection**'s, containing all outputs of its branch tasks
    - **TargetCollection**'s can check if their files are located in the same directory
    - If they do, perform a single (remote) **listdir** and compare basenames → **single request**

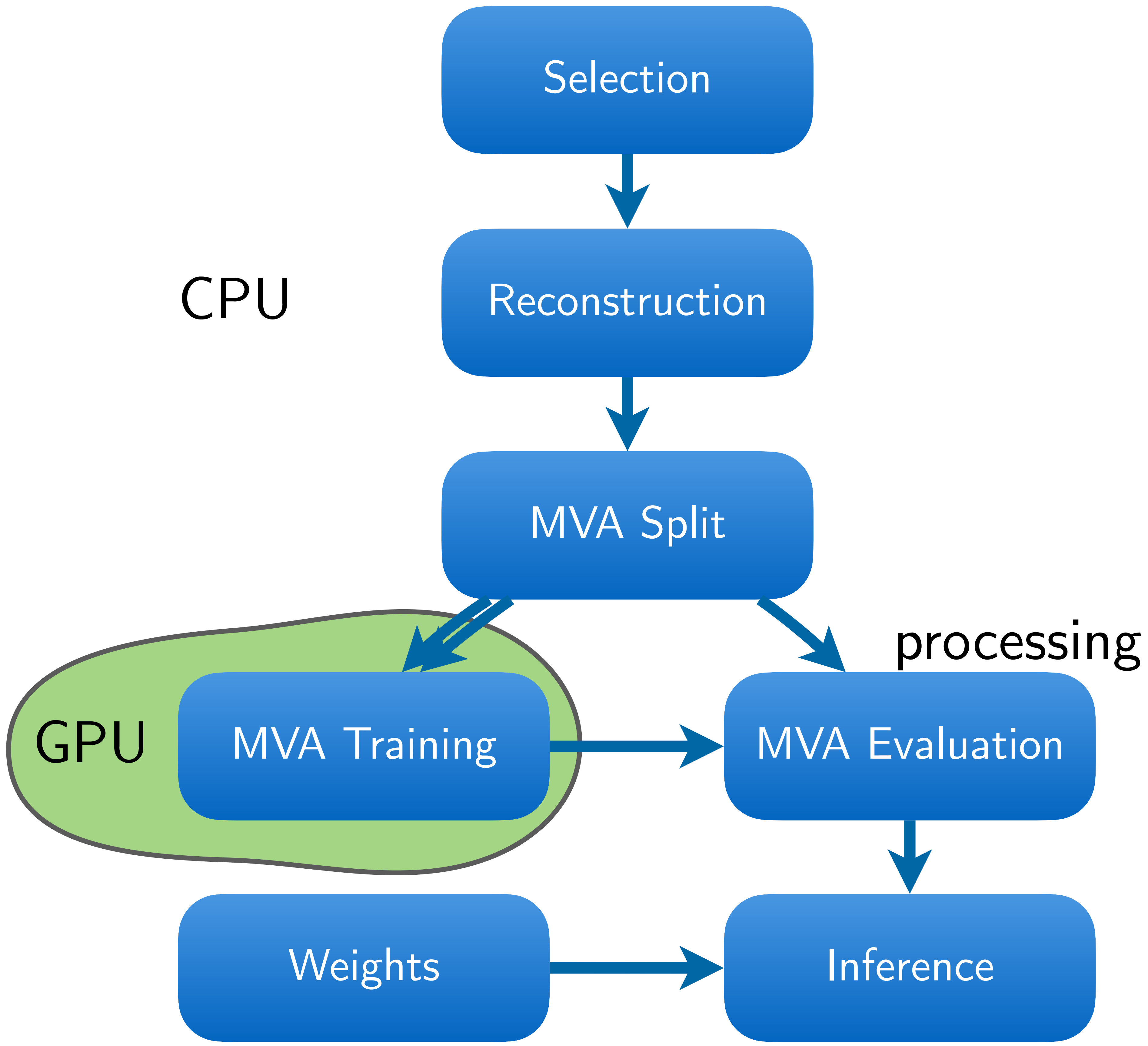
- **There is no free lunch**
  - Our HEP resources (clusters, grid, storage elements, software environments) are very **inhomogeneous**
  - A **realistic** workflow engine
    - can make some good, yet simple assumptions based on known best-practices

      BUT
    - it should **always** allow users to transparently **change decisions** & **configure every single aspect!**

Abstraction: analysis workflows

- Workflow, decomposable into particular workloads
- Workloads related to each other by common interface
  - In/outputs define directed acyclic graph (DAG)
- Alter default behavior via parameters
- Computing resources
  - Run location (CPU, GPU, WLCG, ...)
  - Storage location (local, dCache, EOS, ...)
- Software environment
- Collaborative development and
- Reproducible intermediate and

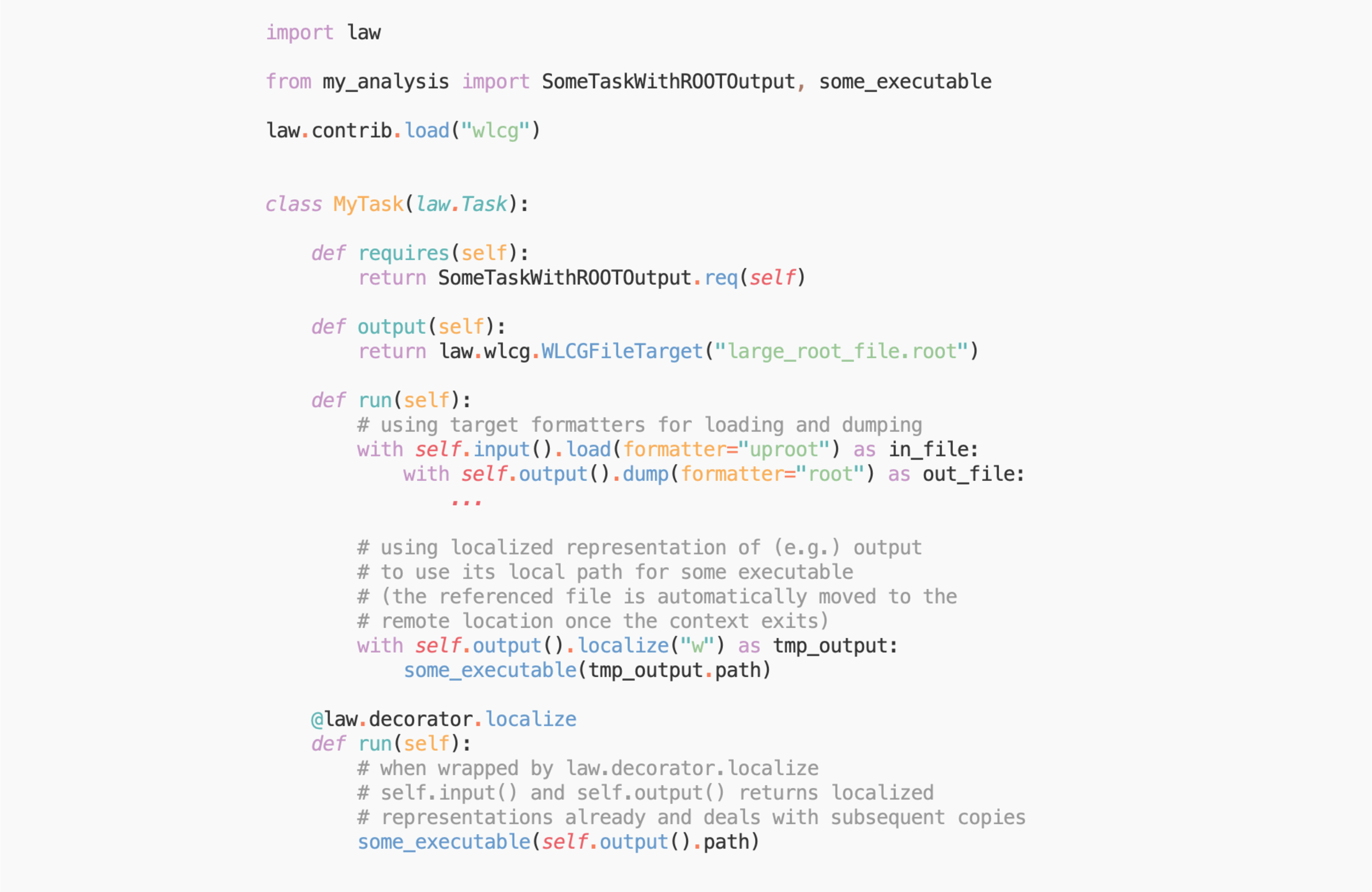
Example



→ Reads like a checklist for analysis workflow management

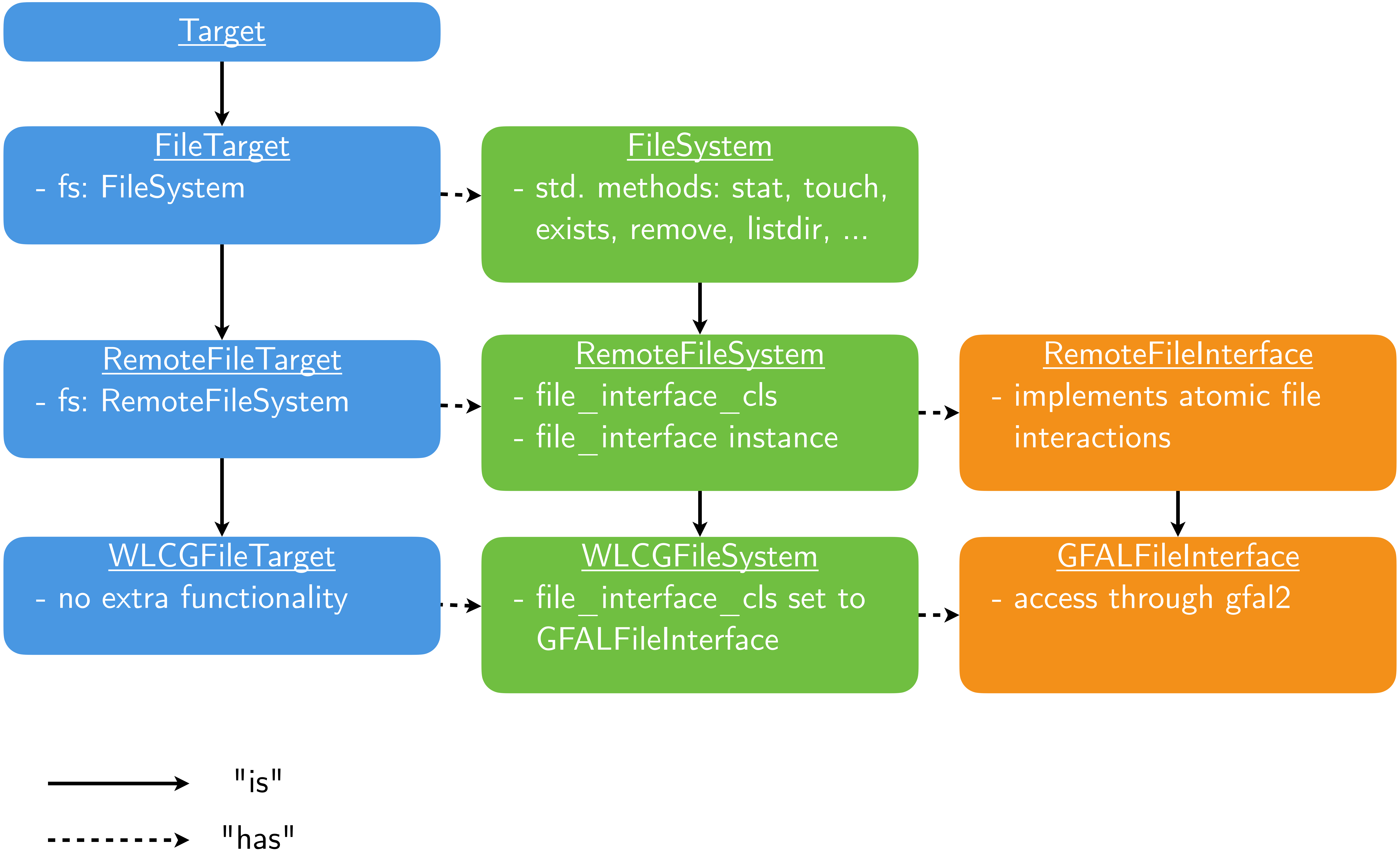
(Remote) targets

Working with remote targets



```python
import law

from my_analysis import SomeTaskWithROOTOutput, some_executable

law.contrib.load("wlcg")


class MyTask(law.Task):

    def requires(self):
        return SomeTaskWithROOTOutput.req(self)

    def output(self):
        return law.wlcg.WLCGFileTarget("large_root_file.root")

    def run(self):
        # using target formatters for loading and dumping
        with self.input().load(formatter="uproot") as in_file:
            with self.output().dump(formatter="root") as out_file:
                ...

        # using localized representation of (e.g.) output
        # to use its local path for some executable
        # (the referenced file is automatically moved to the
        # remote location once the context exits)
        with self.output().localize("w") as tmp_output:
            some_executable(tmp_output.path)

    @law.decorator.localize
    def run(self):
        # when wrapped by law.decorator.localize
        # self.input() and self.output() returns localized
        # representations already and deals with subsequent copies
        some_executable(self.output().path)
```

Remote target implementation details



Target

FileTarget
- fs: FileSystem

FileSystem
- std. methods: stat, touch, exists, remove, listdir, ...

RemoteFileTarget
- fs: RemoteFileSystem

RemoteFileSystem
- file_interface_cls
- file_interface instance

RemoteFileInterface
- implements atomic file interactions

WLCGFileTarget
- no extra functionality

WLCGFileSystem
- file_interface_cls set to GFALFileInterface

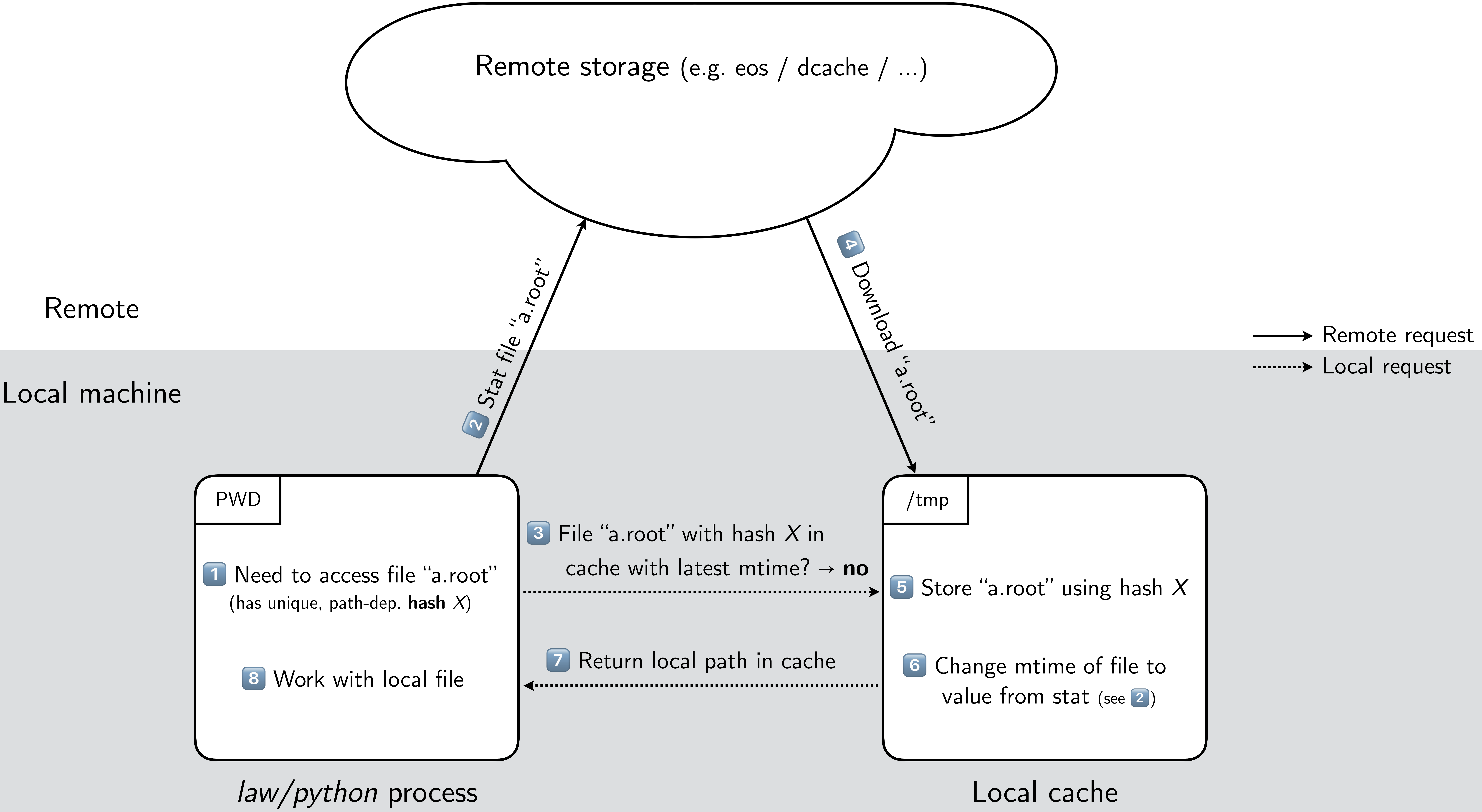
GFALFileInterface
- access through gfal2

→ "is"

- - - → "has"

Local caching (1)

Configuration ☞



Remote storage (e.g. eos / dcache / ...)

Remote

Local machine

2 Stat file "a.root"

4 Download "a.root"

→ Remote request

⋯→ Local request

PWD

1 Need to access file "a.root"
(has unique, path-dep. **hash** X)

8 Work with local file

3 File "a.root" with hash X in cache with latest mtime? → **no**

7 Return local path in cache

/tmp

5 Store "a.root" using hash X

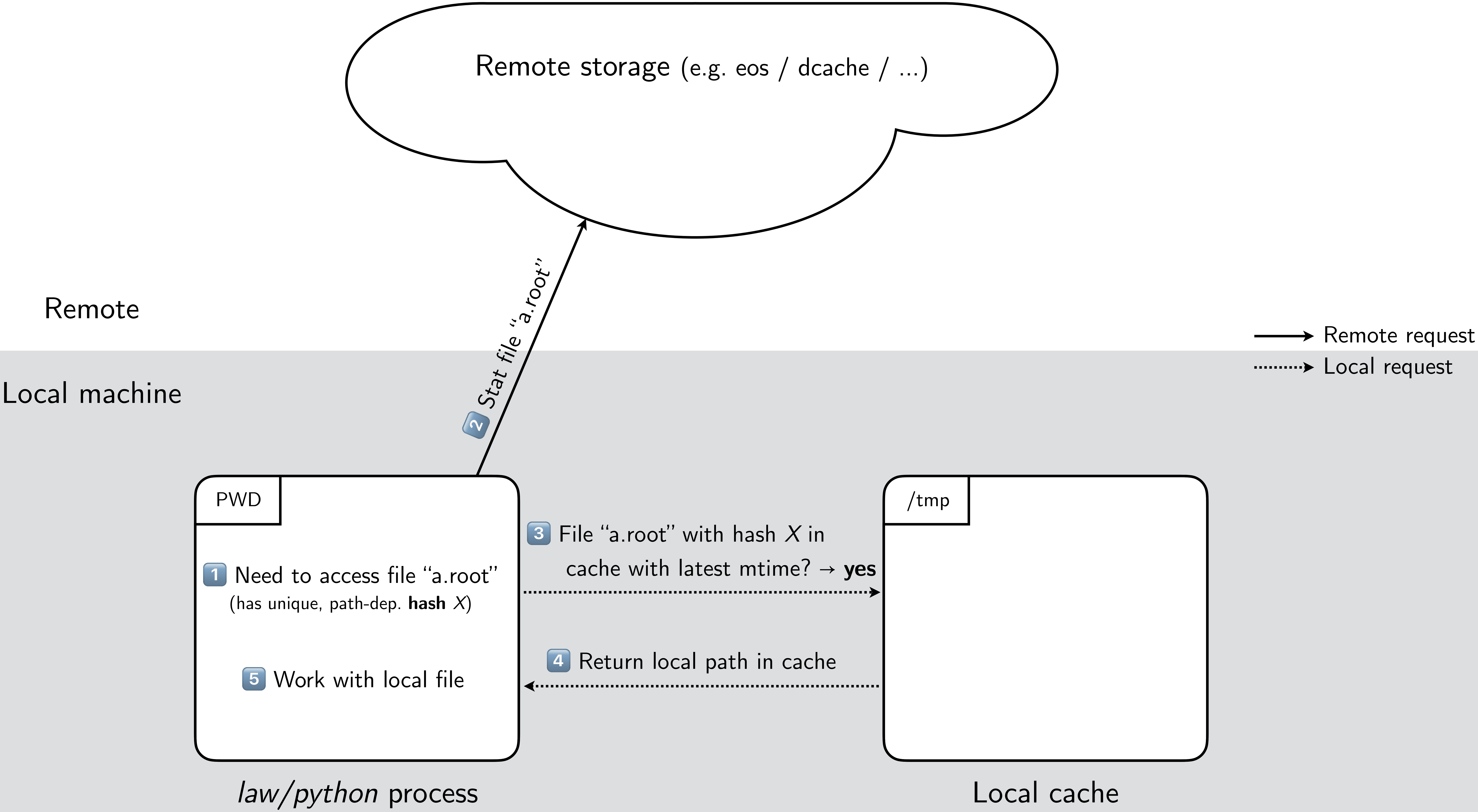
6 Change mtime of file to value from stat (see 2)

law/python process

Local cache

Local caching (2)

Configuration ☞



Remote storage (e.g. eos / dcache / ...)

Remote

Local machine

→ Remote request

⇢ Local request

2 Stat file "a.root"

PWD

1 Need to access file "a.root"
(has unique, path-dep. **hash** *X*)

5 Work with local file

3 File "a.root" with hash *X* in cache with latest mtime? → **yes**

4 Return local path in cache

/tmp

law/python process

Local cache

Workflows

Workflows: General ideas

- **Many tasks exhibit the same overall structure and/or purpose**
  - *"Run over N existing files" / "Generate N events/toys" / "Merge N into M files"*
  - All these tasks can **profit from the same features**
    - *"Only process file x and/to y", "Remove outputs of "x, y & z",*
      "Process N files, but consider the task finished once M < N are done", "..."

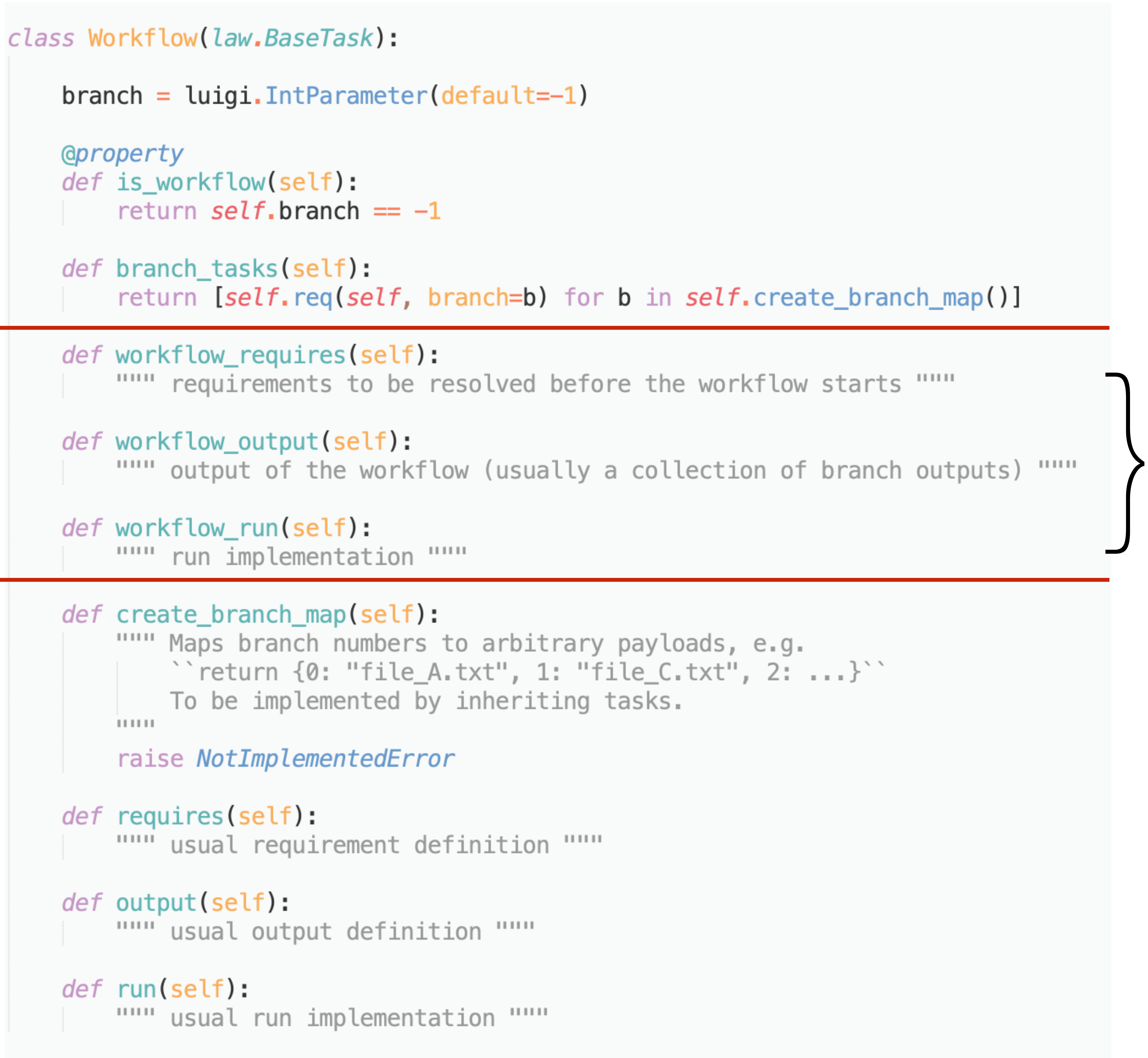
  → Calls for a generic container object that provides guidance and features for these cases

- **Workflow "containers"**
  - Task that introduces a parameters called `--branch b` (`luigi.IntParameter`)
    - `b >= 0`: Instantiates particular tasks called "branches"; `run()` will (e.g.) process file `b`
    - `b = -1`: Instantiates the workflow container itself; `run()` will run* all branch tasks

    * How branch tasks are run is implemented in different workflow types: local or several remote ones

- **Practical advantages**
  - Convenience: same features available in all workflows (see next slides)
  - **Scalability and versatility for remote workflows**
    - Jobs: Better control of jobs, submission, task-to-job matching ... (see next slides)
    - Luigi: Central scheduler breaks when pinged by O(10k) tasks every few seconds
    - Remote storage: allows batched file operations instead of file-by-file requests

Workflows: example implementation



Common

```python
class Workflow(law.BaseTask):

    branch = luigi.IntParameter(default=-1)

    @property
    def is_workflow(self):
        return self.branch == -1

    def branch_tasks(self):
        return [self.req(self, branch=b) for b in self.create_branch_map()]
```

Workflow specific

```python
    def workflow_requires(self):
        """ requirements to be resolved before the workflow starts """

    def workflow_output(self):
        """ output of the workflow (usually a collection of branch outputs) """

    def workflow_run(self):
        """ run implementation """
```

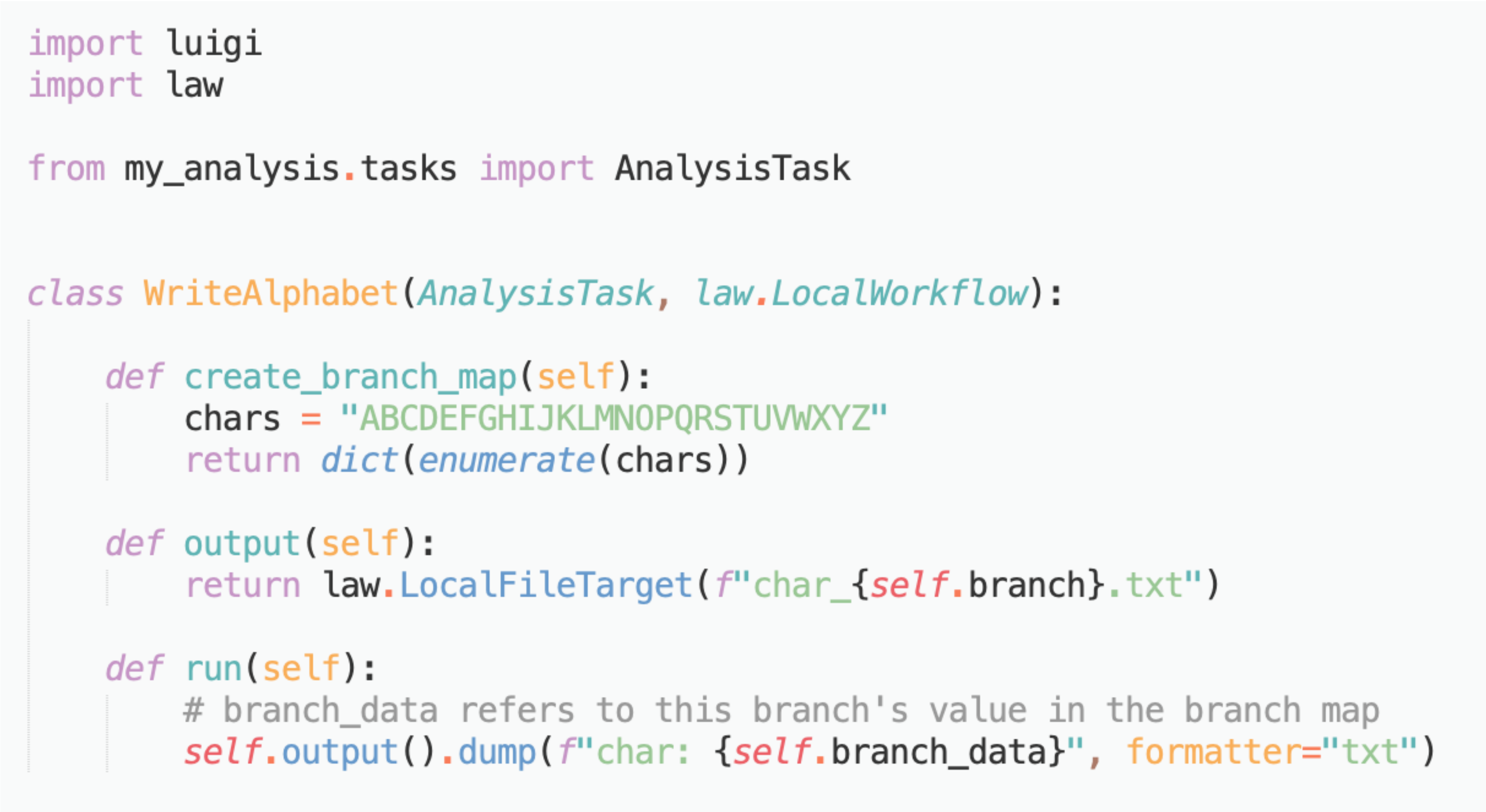
Implemented by task

```python
    def create_branch_map(self):
        """ Maps branch numbers to arbitrary payloads, e.g.
            ``return {0: "file_A.txt", 1: "file_C.txt", 2: ...}``
            To be implemented by inheriting tasks.
        """
        raise NotImplementedError

    def requires(self):
        """ usual requirement definition """

    def output(self):
        """ usual output definition """

    def run(self):
        """ usual run implementation """
```

When "`is_workflow`", seen by luigi as `requires()`, `output()` and `run()`

Workflows: example usage

- Tasks that each write a single character into a text file
- Character assigned to them though the branch map as their "branch data"



```python
import luigi
import law

from my_analysis.tasks import AnalysisTask


class WriteAlphabet(AnalysisTask, law.LocalWorkflow):

    def create_branch_map(self):
        chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
        return dict(enumerate(chars))

    def output(self):
        return law.LocalFileTarget(f"char_{self.branch}.txt")

    def run(self):
        # branch_data refers to this branch's value in the branch map
        self.output().dump(f"char: {self.branch_data}", formatter="txt")
```

Workflows: remote workflows & jobs

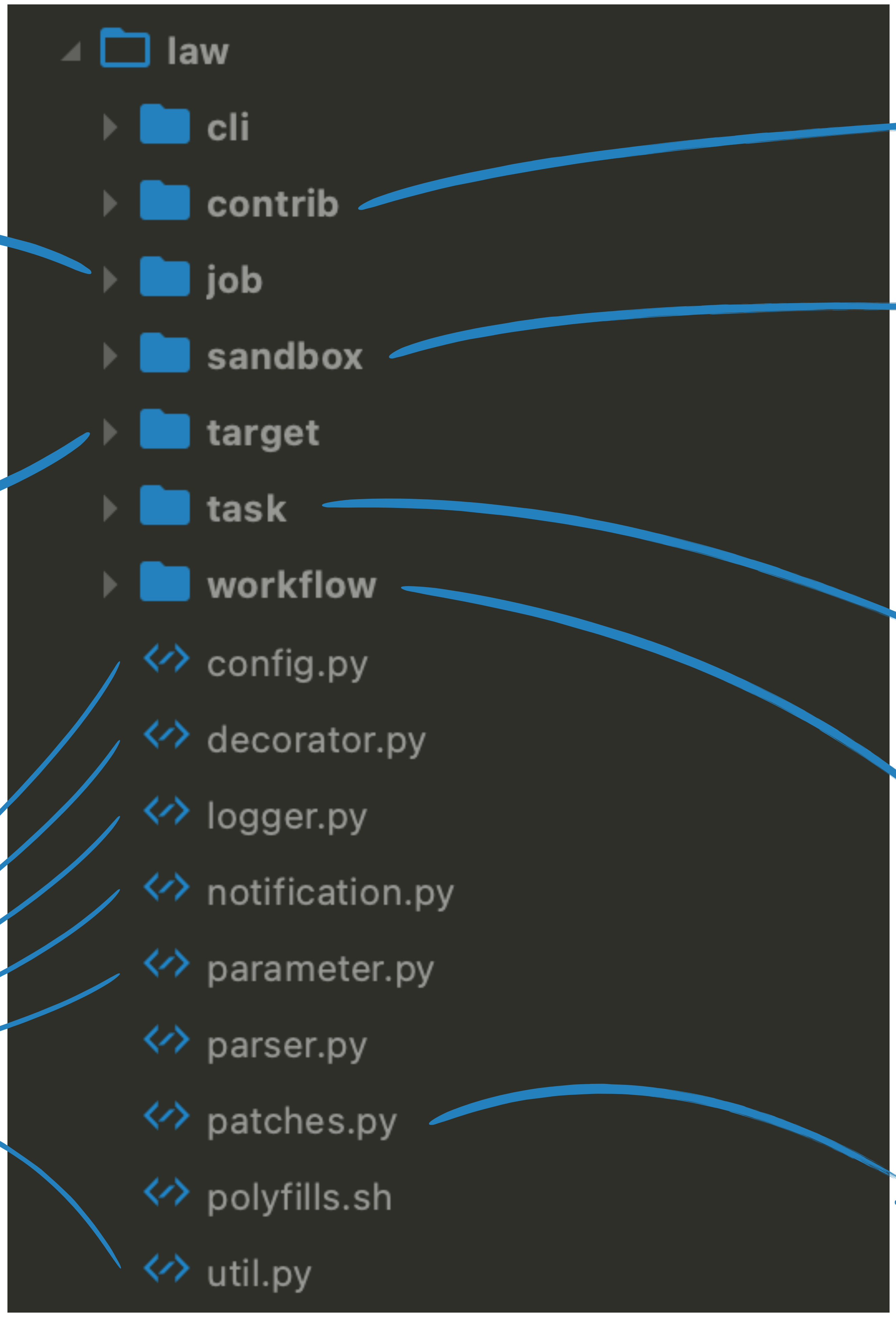
- 6 **remote workflow implementations come with law**
  - htcondor, glite, lsf, arc, slurm, cms-crab (in PR#150)
  - Based on generic "job manager" implementations in contrib packages

- **Job managers fully decoupled from most law functionality**
  - Simple extensibility
  - No "auto-magic" in submission files, rather minimal and configurable through tasks
  - Usable also without law

- **Most important features**
  - Job submission functionality "declared" via task class inheritance
  - Provision of software and job-specific requirements through `workflow_requires()`
  - Control over remote jobs through parameters:
    - `--branch` `--branches` : granular control of which tasks to process
    - `--acceptance` `--tolerance` : defines when a workflow is complete / failed
    - `--poll-interval` `--walltime` : controls the job status polling interval and runtime
    - `--tasks-per-job` `--parallel-jobs` : control of resource usage at batch systems

Miscellaneous

Package structure



```
law
  cli
  contrib
  job
  sandbox
  target
  task
  workflow
  config.py
  decorator.py
  logger.py
  notification.py
  parameter.py
  parser.py
  patches.py
  polyfills.sh
  util.py
```

Command-line interface

Job interface
- Job file factory interface
- Job manager interface
- Generic remote job script

Target definitions
- Generic + file interace
- Local target impl.
- Remote target interfaces

Config parsing & tools
Task decorators
Custom loggers
Notification tools (for e.g. slack/telegram)
Custom parameters
Utilities & helpers

3rd party tools

Sandboxing mechanism
- Sandbox task
- Sandbox interface
- Bash sandbox impl.

Base task definitions

Base workflow definition
- Local workflow impl.
- Remote workflow interface

Lightweight patches of luigi, e.g.:
- Disable dep. checks in sandboxes
- Colorize logs

→ Could be added directly to luigi

Package structure

Command-line interface

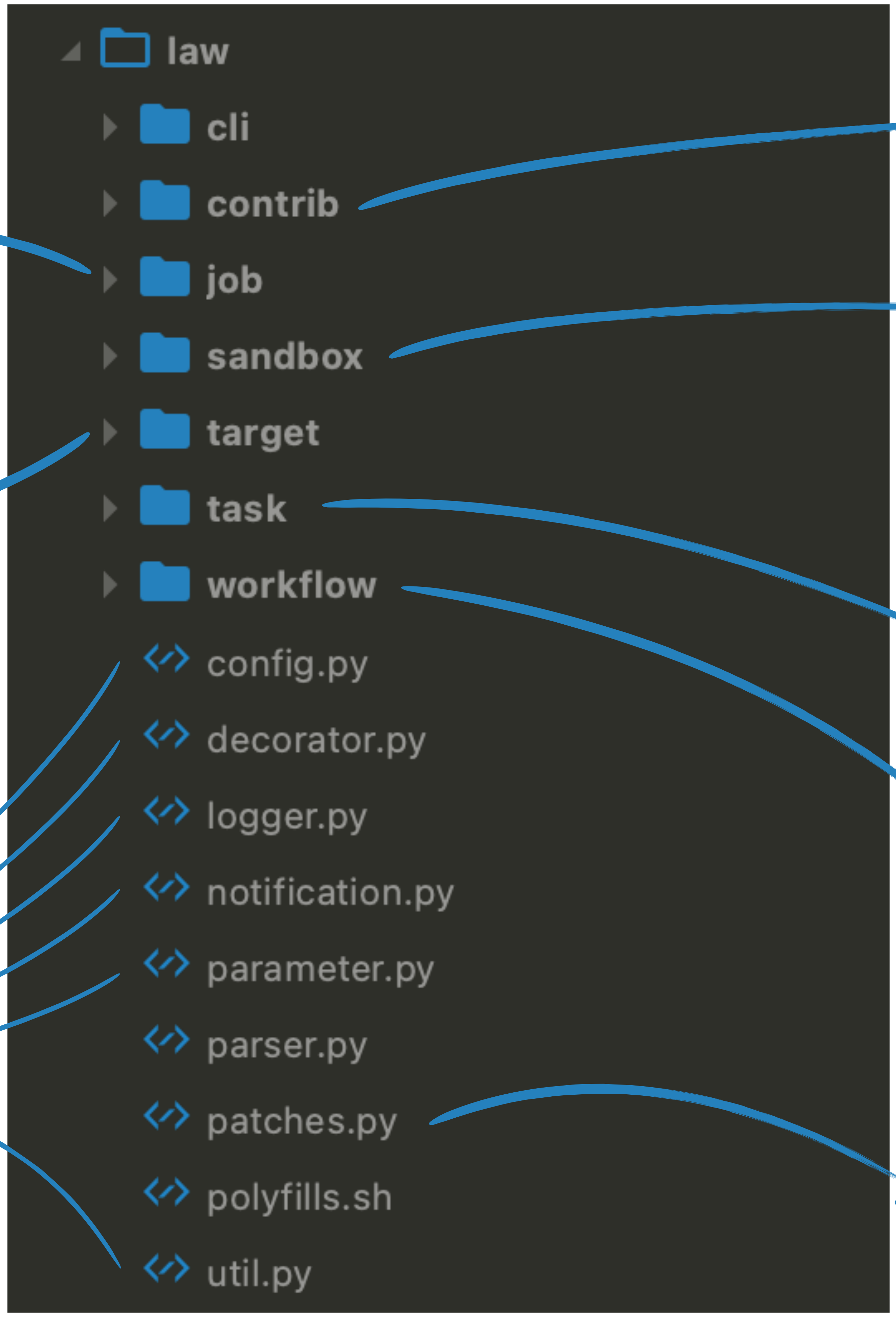
Job interface

- Job file factory interface
- Job manager interface
- Generic remote job script

Target definitions

- Generic + file interace
- Local target impl.
- Remote target interfaces

Config parsing & tools
Task decorators
Custom loggers
Notification tools (for e.g. slack/telegram)
Custom parameters
Utilities & helpers



```
law
  cli
  contrib
  job
  sandbox
  target
  task
  workflow
  config.py
  decorator.py
  logger.py
  notification.py
  parameter.py
  parser.py
  patches.py
  polyfills.sh
  util.py
```

S

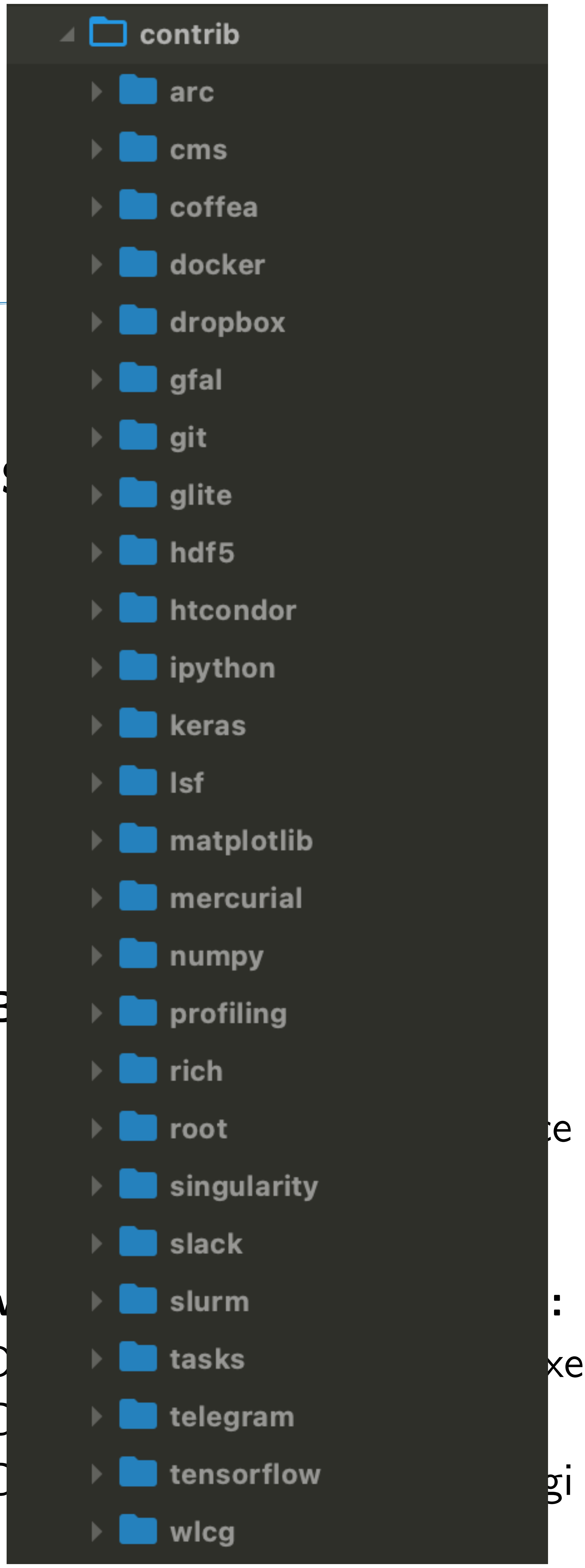
E

te

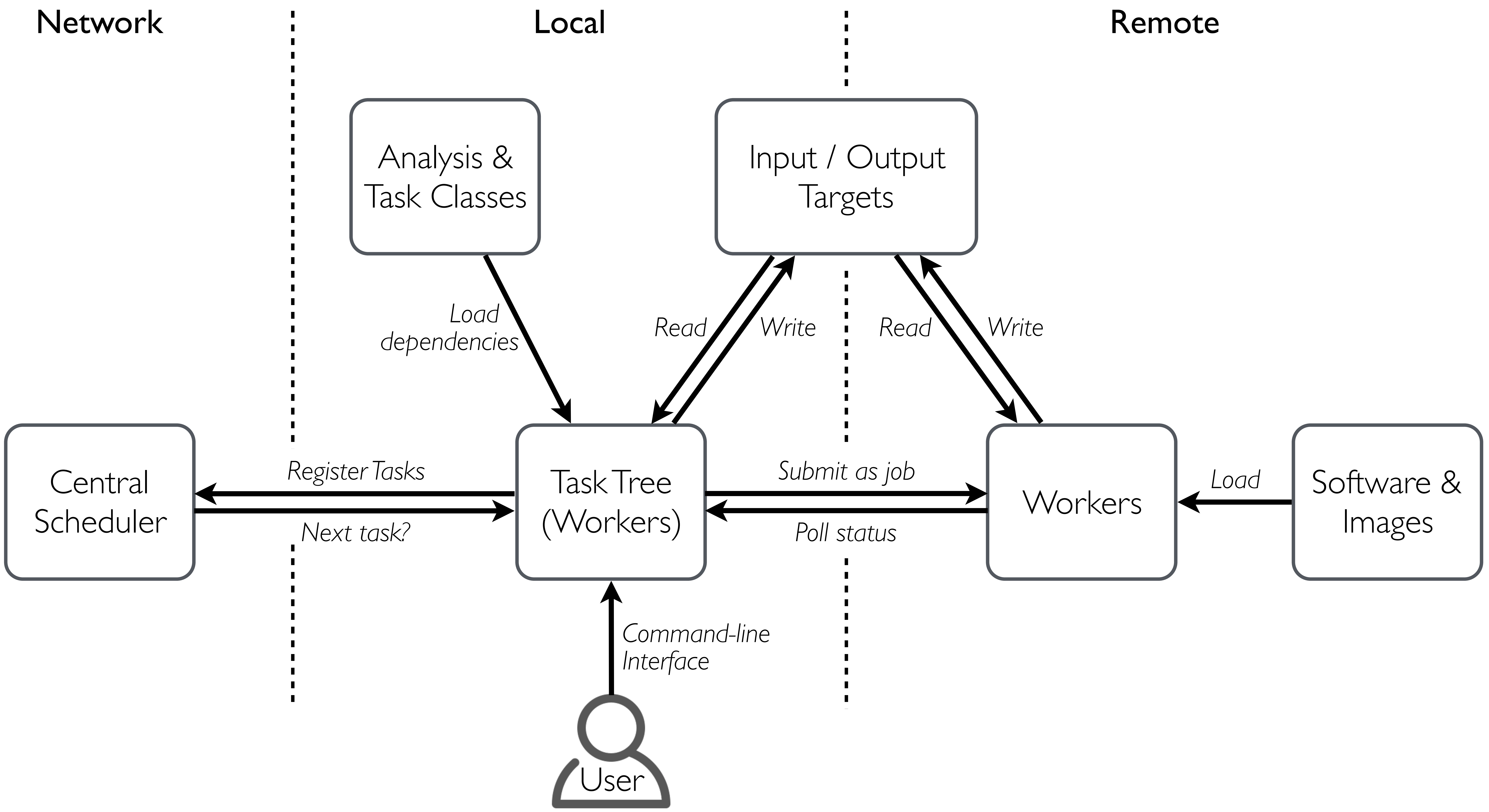
Lightw :

- D xes
- C

→ C gi



```
contrib
  arc
  cms
  coffea
  docker
  dropbox
  gfal
  git
  glite
  hdf5
  htcondor
  ipython
  keras
  lsf
  matplotlib
  mercurial
  numpy
  profiling
  rich
  root
  singularity
  slack
  slurm
  tasks
  telegram
  tensorflow
  wlcg
```



Network

Local

Remote

Analysis & Task Classes

Input / Output Targets

Load dependencies

Read

Write

Read

Write

Central Scheduler

Register Tasks

Next task?

Task Tree (Workers)

Submit as job

Poll status

Workers

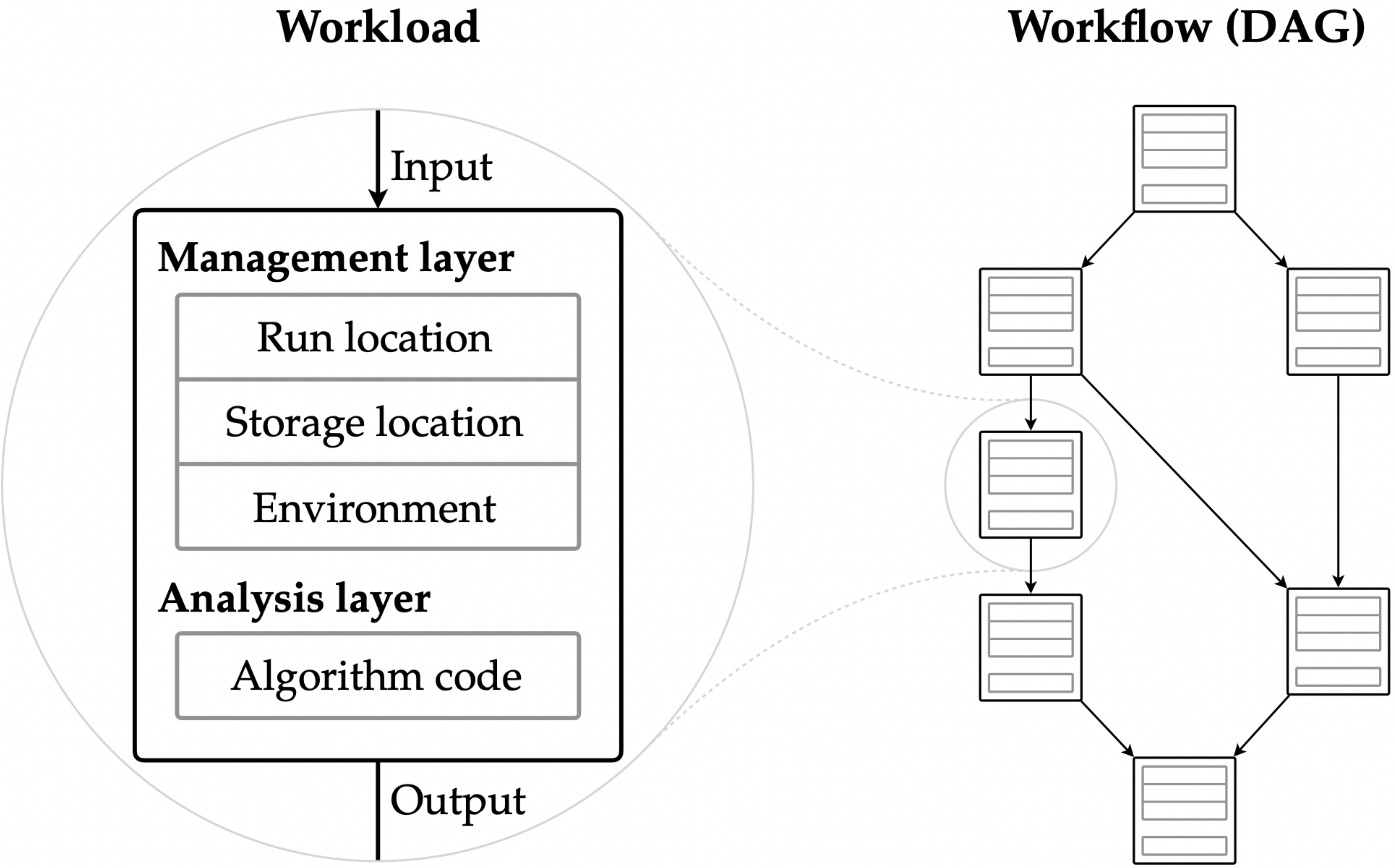
Load

Software & Images

Command-line Interface

User

DAG abstraction



Links

- *law* - *luigi* analysis workflow

Repository	☞ github.com/riga/law
Paper	☞ arXiv:1706.00955 (CHEP16 proceedings)
Documentation	☞ law.readthedocs.io (in preparation)
Minimal example	☞ github.com/riga/law/tree/master/examples/loremipsum
HTCondor example	☞ github.com/riga/law/tree/master/examples/htcondor_at_cern
Contact	☞ Marcel Rieger

- *luigi* - Powerful Python pipelining package (by Spotify)

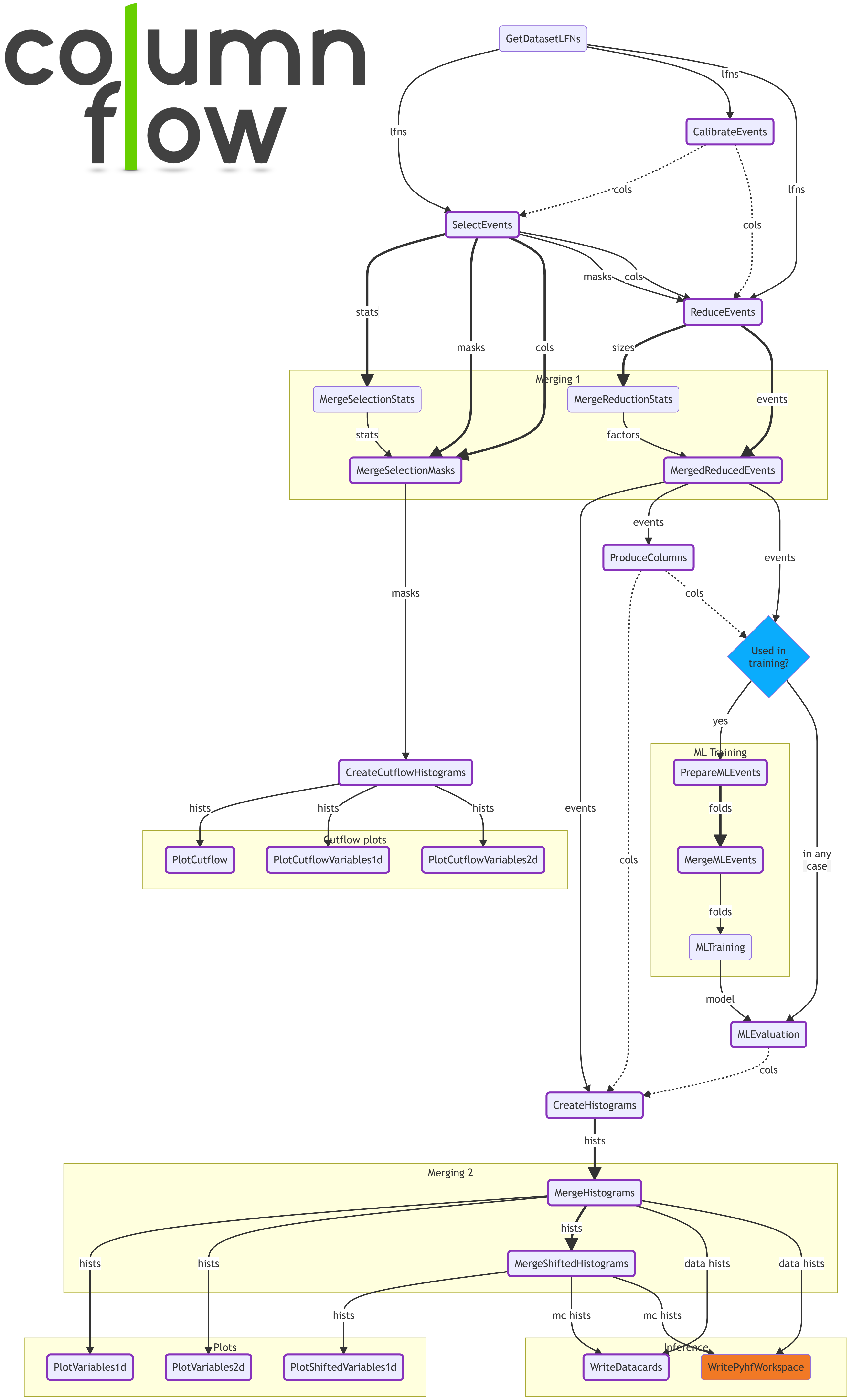
Repository	☞ github.com/spotify/luigi
Documentation	☞ luigi.readthedocs.io
“Hello world!”	☞ github.com/spotify/luigi/blob/master/examples/hello_world.py

- Technologies

GFAL2	☞ dmc.web.cern.ch/projects/gfal-2/home
Docker	☞ docker.com
Singularity	☞ singularity.lbl.gov

columnflow

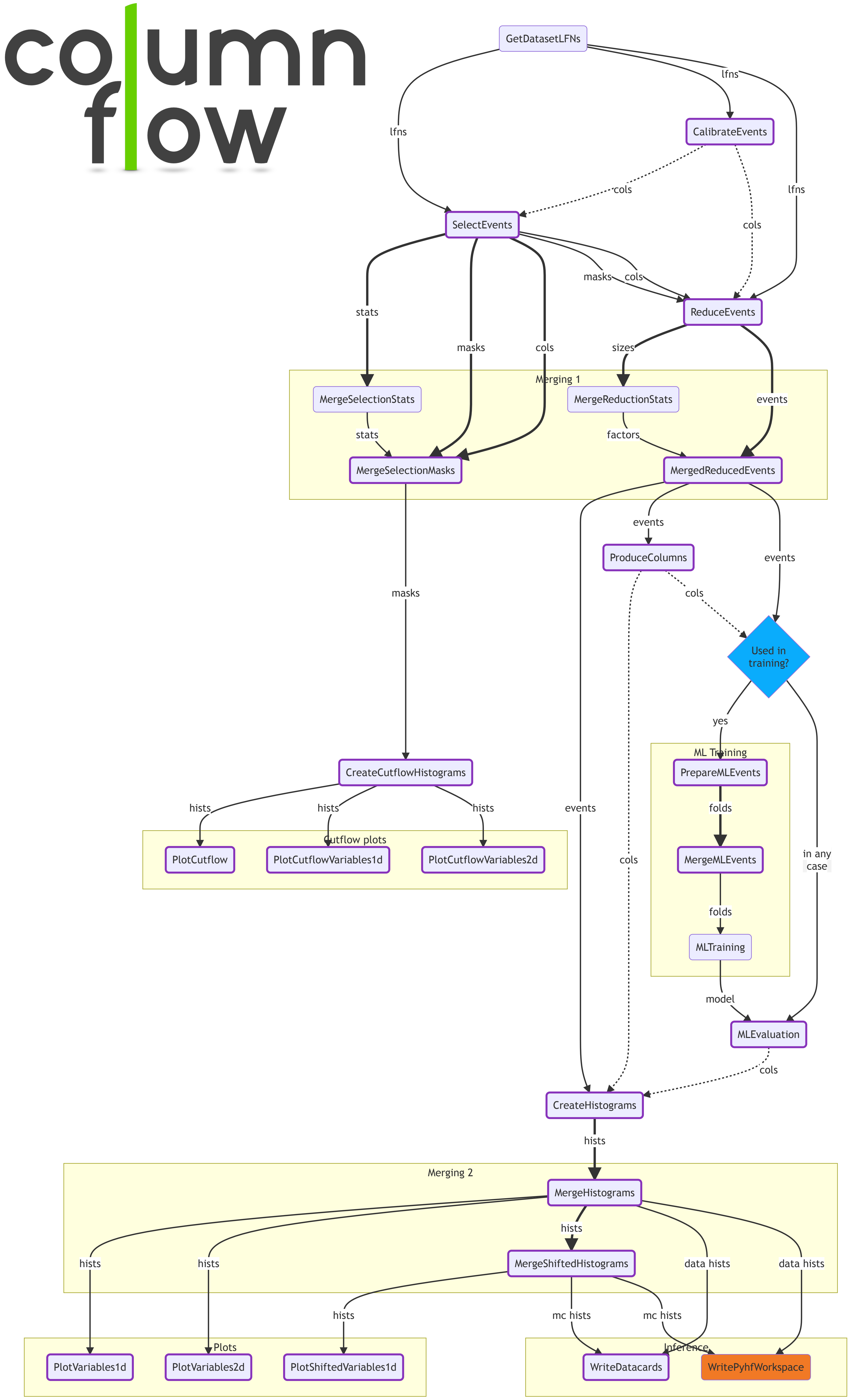
Example: Resolution of systematic uncertainties



Initial tasks

Final results

Example: Resolution of systematic uncertainties



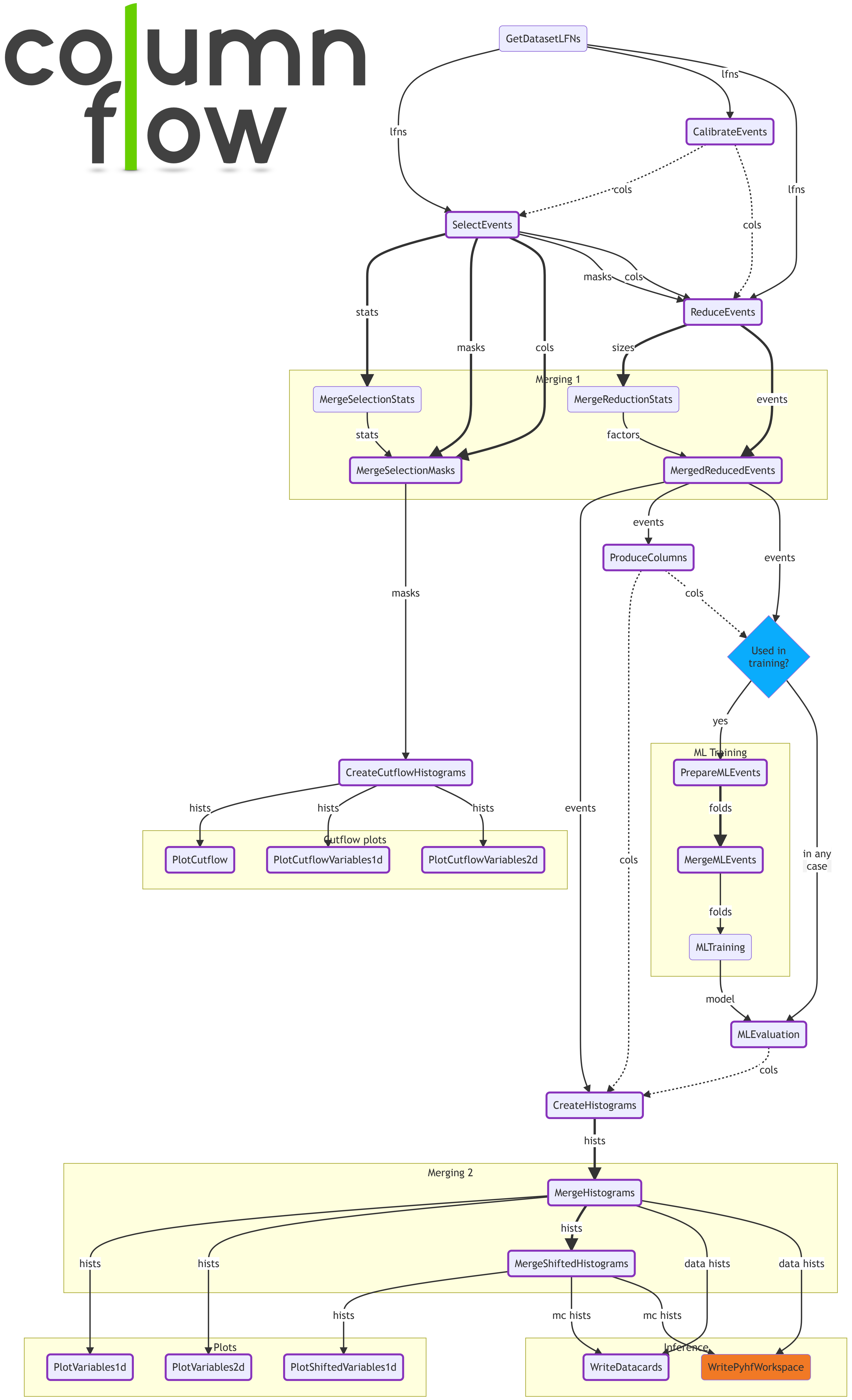
Initial tasks

Final results

"nominal"

Example: Resolution of systematic uncertainties

column flow

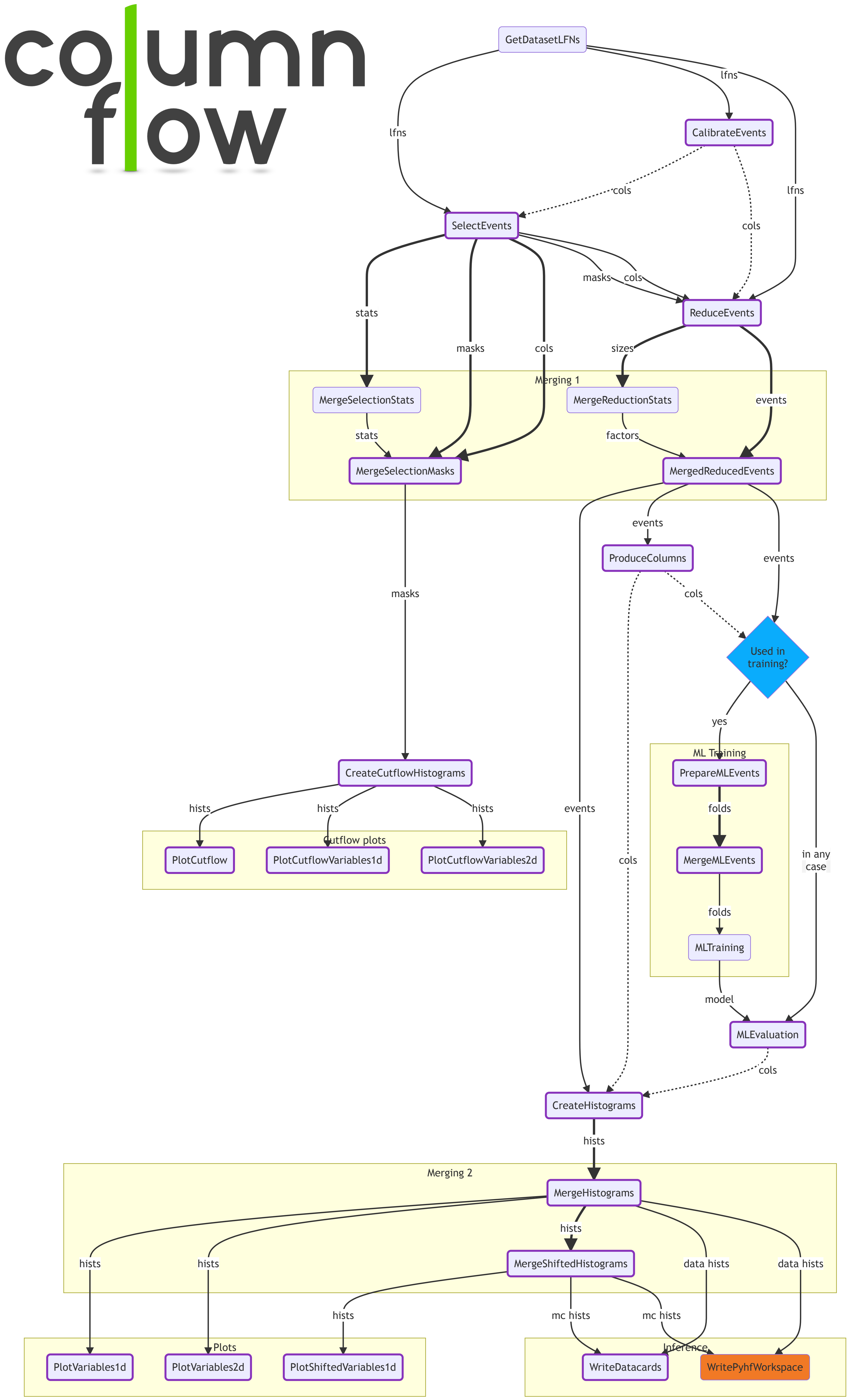


Initial tasks

Final results

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

Example: Resolution of systematic uncertainties



Initial tasks

Final results

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

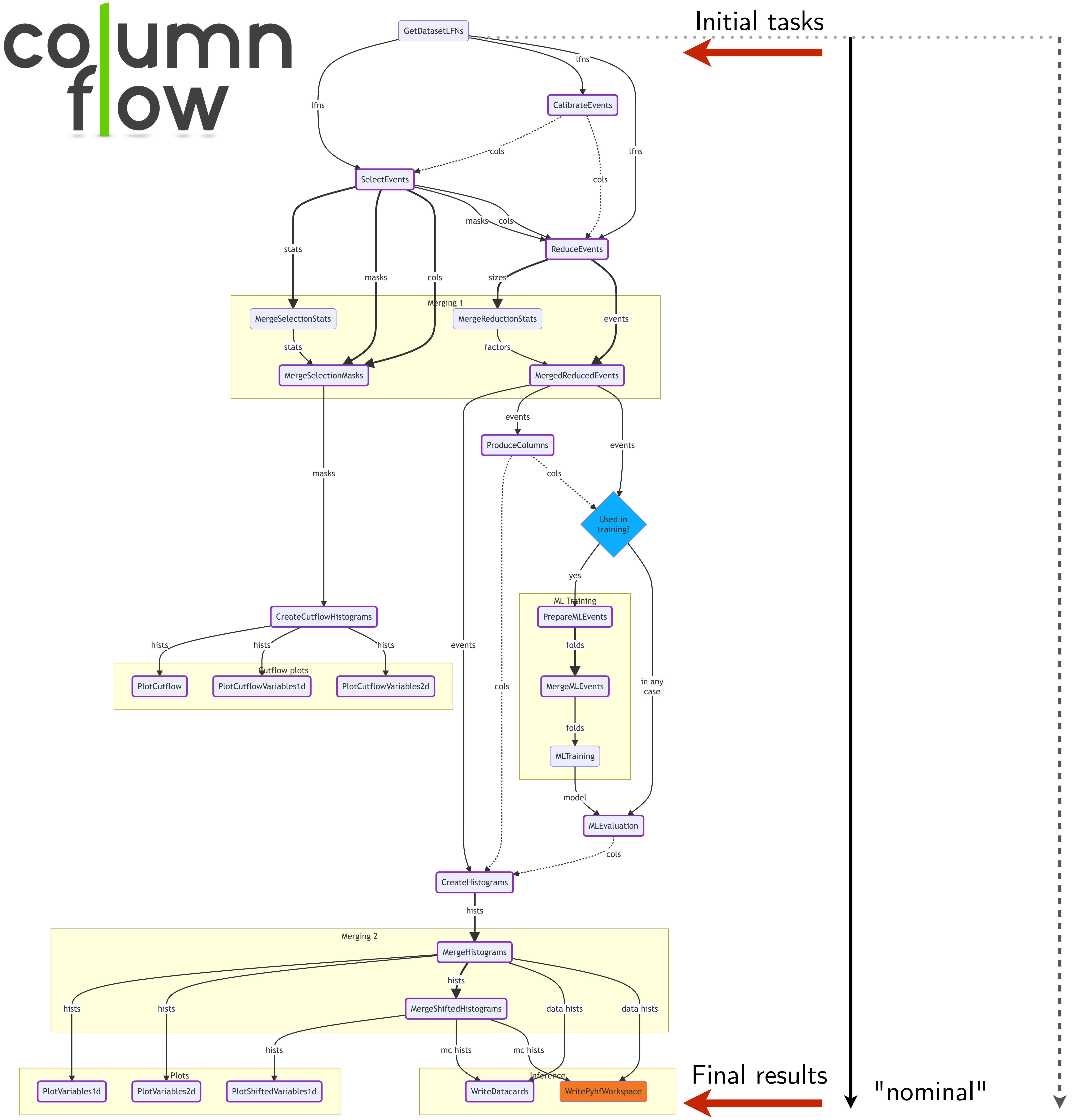
Key idea

Tasks *know* which uncertainties

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

Example: Resolution of systematic uncertainties

columnflow



"tune(up|down)" "jec(up|down)" "pileup(up|down)" ...

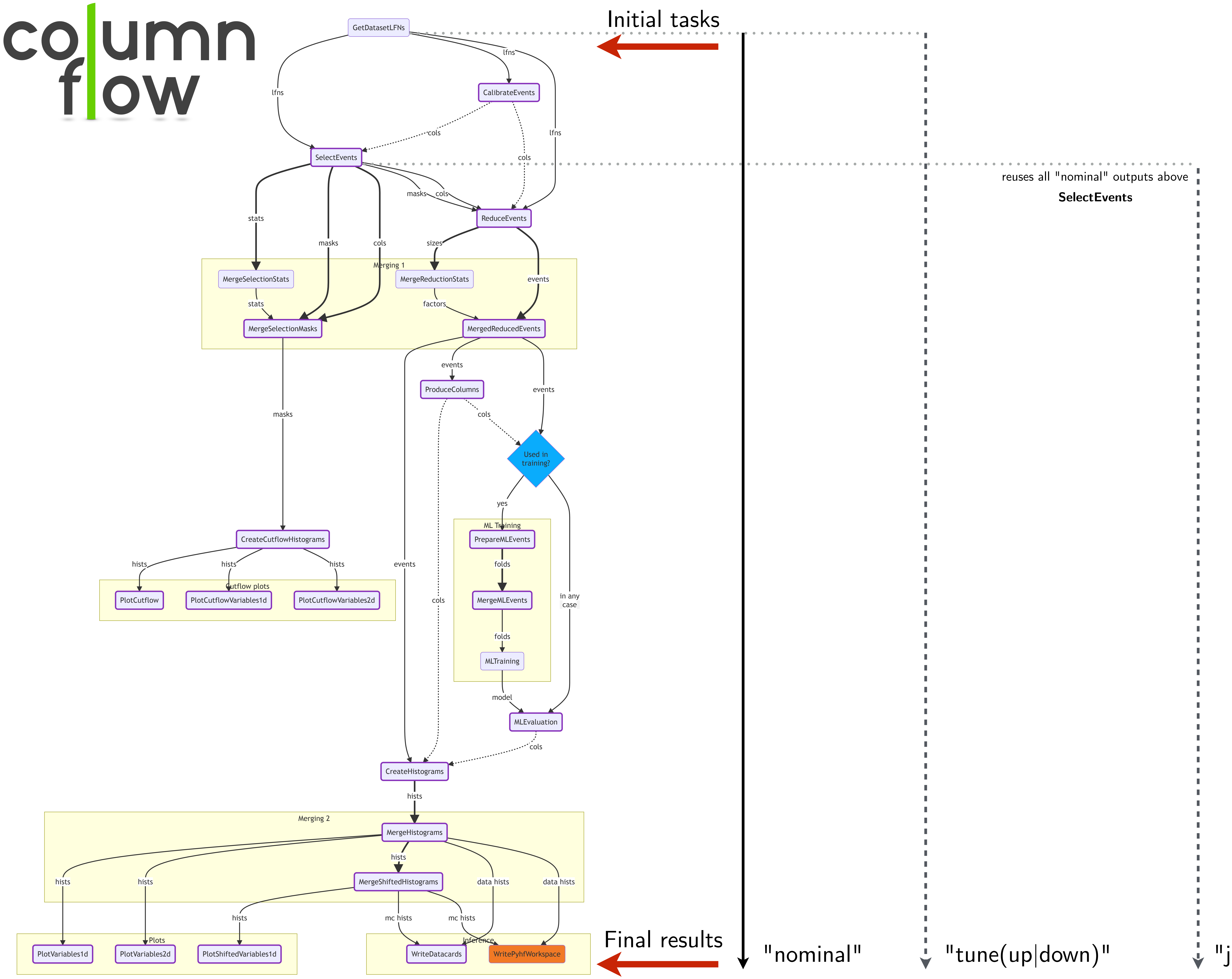
Key idea

Tasks *know* which uncertainties

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

Example: Resolution of systematic uncertainties

columnflow

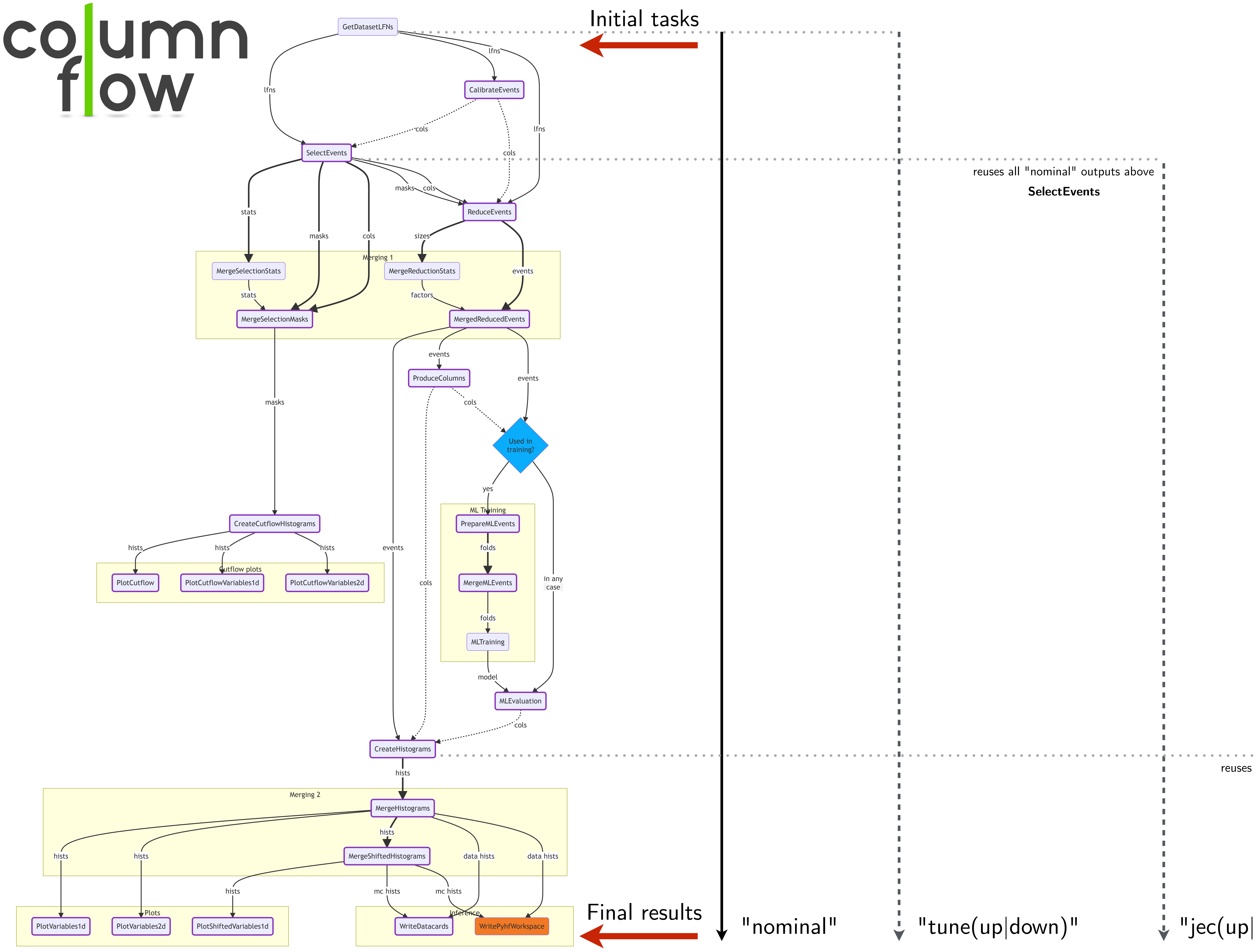


Key idea

Tasks *know* which uncertainties

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

Example: Resolution of systematic uncertainties



Key idea

Tasks *know* which uncertainties

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