Orchestrated columnar-based analysis with *columnflow*

Mathis Frahm[1,*] for the CMS collaboration

[1]Institut für Experimentalphysik, Universität Hamburg

Abstract. *Columnflow* is a tool for columnar-based data analysis. It is written in *Python*, experiment-agnostic in its core, and supports any flat *n*-tuple format, such as *ROOT*-based *TTrees* or *Parquet* files. Leveraging the vast *Python* ecosystem, vectorization and convenient representation of event content can be achieved through *NumPy*, *AwkwardArray* and other libraries. Based on the *Luigi Analysis Workflow* (law) package, *columnflow* provides full analysis automation over arbitrary, distributed computing resources. This approach features persistent, intermediate outputs for purposes of reusing of previously computed results, debugging, and exchange with collaborators. Job submission to various batch systems is natively supported. Remote files can be seamlessly accessed via various protocols using the *Grid File Access Library* (GFAL2). In addition, a sandboxing mechanism can encapsulate the execution of parts of a workflow in dedicated environments, supporting subshells, *Python* virtual environments, and containers. This contribution introduces the key components of *columnflow* and highlights the benefits of a fully automated workflow for complex and large-scale HEP analyses.

1 Introduction

To study and search for increasingly rare physics processes at the Large Hadron Collider (LHC), a staggering amount of data needs to be processed and analyzed with progressively complex methods. Typical analyses performed with the Run 2 data collection campaign (2015-2018) of the Compact Muon Solenoid (CMS) [1] experiment require analyzing tens of billions of recorded and simulated proton-proton collision events. With the LHC scheduled to run until 2041, the amount of data will increase to an unprecedented amount.

The raw detector information is processed into high-level physics objects such as Electrons, Muons, and Jets using multiple reconstruction and identification algorithms. After additional processing steps to filter the per-event information relevant for physics analysis, the final data format still contains around 1-2 kB of data per event, corresponding to a total amount of input data on the order of tens of terabytes for a typical Run 2 CMS analysis using the *NanoAOD* [2] data format.

Each physics analysis utilizes a different subset of this data. The task of the physicists is to define filter and reconstruction algorithms to prepare the data for statistical interpretation. Many analyses also contain machine learning algorithms that are specifically trained on their selected data. At the same time, they need to derive and apply multiple calibrations and corrections. To consider systematic uncertainties, large parts of the analysis need to be re-run multiple times with systematic variations.

*e-mail: mathis.frahm@uni-hamburg.de

Frameworks used for HEP data analysis need to be scalable both in complexity and in the amount of analyzed data. In these proceedings, we present *columnflow*, which is a toolkit for columnar data analysis that achieves scalability in data processing via parallelization and in complexity by providing a fully orchestrated workflow for HEP analyses, encoding all task dependencies from start to finish and allowing users to create highly customizable physics analyses on top.

First, we present the processing paradigm as well as the fully orchestrated workflow that is provided by *columnflow*. We give a short overview of how analyses can be configured and customized when using *columnflow*. Finally, we introduce the conceptual handling of systematic uncertainties.

2 Processing paradigm

The data processing in *columnflow* is performed in a columnar manner, meaning that processing algorithms are defined such that they can be applied on an array of events at once. The scientific Python ecosystem provides many libraries to perform efficient numerical operations on arrays. The *NumPy* [3] package provides the baseline functionality for array-based programming, and it is extended by the *AwkwardArray* [4] package to allow operations on data with jagged (i.e. not rectangular) arrays. The *coffea* [5] *NanoEventsFactory* is utilized to add a physics convenience layer that provides behaviour such as four-vector arithmetic to the arrays.

The main units of processing in *columnflow* are user-defined functions that perform array operations on a chunk of data (100,000 events per default) represented as an *AwkwardArray*, where all relevant columns are *materialized* (i.e. loaded in memory) at once. Each input file chunk is read consecutively, yet in parallel through a multi-threaded I/O interface to optimize CPU utilization while reducing memory overhead. The data processing can be performed either locally or via job submission to widely used batch systems, with native support provided for *HTCondor* [6] and *Slurm* [7]. Job submission to *Worldwide LHC Computing Grid* (WLCG) [8] sites via the *CMS Remote Analysis Builder* (CRAB) tool is also supported.

Intermediate results are stored in the *Parquet* file format using the *PyArrow* library [9]. Files can be stored either locally or remotely. Remote files can be seamlessly accessed via various protocols such as *XRootD*, *WebDAV*, or *GridFTP*, using the *Grid File Access Library* (GFAL2) [10].

3 Workflow orchestration

The defining feature of *columnflow* is a fully orchestrated workflow. The workflow is created using the *Luigi Analysis Workflow* (law) [11] package, which utilizes the *Luigi* [12] workflow engine. Each processing step in *columnflow* is implemented as a *task*. Every task is callable on the command-line with a pre-defined set of task parameters. Dependencies between different tasks are directly encoded in the task itself. When a task is called, all task dependencies are automatically resolved in a directed acyclic graph up to the point where task outputs are already present. This allows users of *columnflow* to run their complete analysis workflow with a single command, ensuring that only the requested results are produced and existing results are reused if possible. All significant task parameters, including a parameter purely for labelling of successive iterations of analysis (*versioning*), are encoded in the output paths of all tasks, allowing users to run their workflow in a highly customizable manner.

The workflow that is implemented in *columnflow* as seen in figure 1 follows the steps that most HEP analyses usually require. First, the data needs to be calibrated and filtered. This

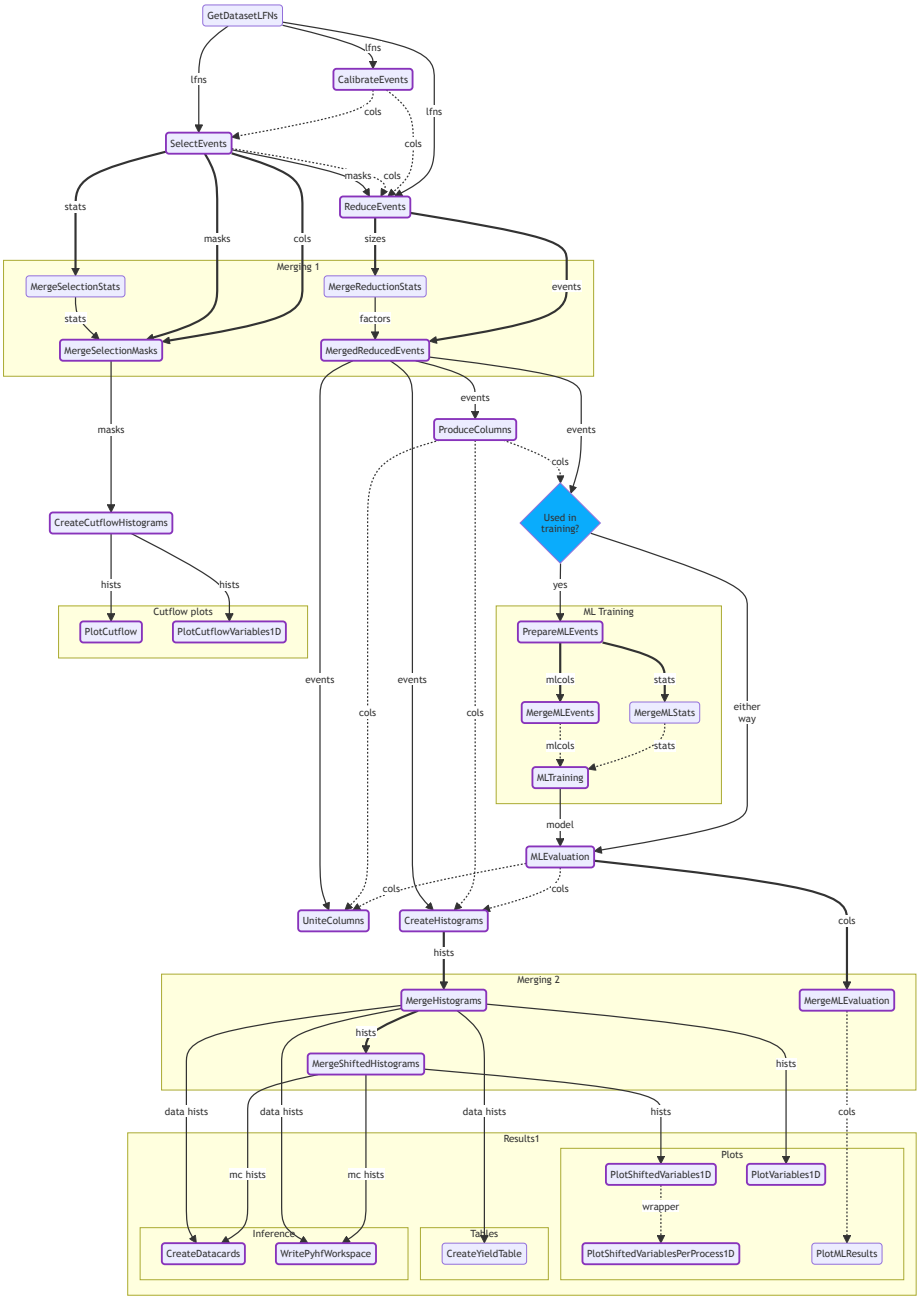

Figure 1. Orchestrated workflow as provided by the *columnflow* framework. Each purple box represents one task that can be triggered on the command-line. Arrows represent dependencies between tasks. When a task is triggered, all upstream dependencies are resolved up to the point where outputs are already present, and tasks with missing outputs are run automatically.

reduced dataset then is potentially passed through different event reconstruction and machine learning algorithms to reconstruct observables of interest. These observables are then stored in histograms, which are subsequently used to create plots or to perform binned likelihood fits to extract physics parameters or to determine compatibility with some physics model.

In *columnflow*, most tasks are executed and parallelized over input datasets consisting of multiple files, systematic variations (*shifts*), and data-taking periods (*configs*). A total number of tasks in the order of 500,000 is not unusual for a full CMS Run 2 analysis. The `GetDatasetLFNs` task fetches and stores the logical file names of all files for a given dataset, config, and shift. The following tasks are parallelized over these files up to a point where a merging step is introduced. The `CalibrateEvents` task is intended to apply calibrations to collections of physics objects such as jets. The `SelectEvents` task is used to define boolean masks, which are applied in the `ReduceEvents` task to filter object collections and to discard events. In `MergeReducedEvents`, files can be partially merged to a requested file size (512 MB per default). These files can be passed through additional reconstruction algorithms in `ProduceColumns` or through a machine learning evaluation in `MLEvaluation` [1]. All produced columns can be used in `CreateHistograms` to create histograms, which are subsequently merged and used to create plots in plotting tasks such as `PlotVariables1D`, or written to various file formats to be used as inputs for statistical inference in `CreateDatacards`.

Each task can define its own software environment via a sandboxing mechanism. *Columnflow* provides a baseline set of sandboxes for columnar analysis, machine learning, and a dedicated *CMS Software Framework* (CMSSW) sandbox. Users can also supply their own software environments to tasks via custom sandboxes. The sandboxing mechanism supports plain shell scripts, *Python* virtual environments, and containers.

While we believe that the default workflow is flexible enough to be used for most physics analyses, it is also possible to customize existing tasks or to extend it with user-defined, analysis-specific tasks. Large parts of the code base are written in a modular way, allowing users to to inherit functionality for their own tasks in a straightforward manner.

4 Workflow customization

While *columnflow* provides the complete workflow orchestration, the central logic of almost every task in this workflow remains fully customizable. This is achieved via so-called *TaskArrayFunctions* that can be passed to the workflow and provide plain *Python* functions that define the implementation of a certain task.

The most notable *TaskArrayFunctions* are the *Calibrator*, *Selector*, and *Producer*, which are called from their task equivalent (`CalibrateEvents`, `SelectEvents`, `ProduceColumns`) to produce their outputs of interest. In the following, we present the functionality of *TaskArrayFunctions* using the *Producer* as an example. A typical implementation of a *Producer* is shown in listing 1. The *@producer* decorator creates a *Producer* subclass, which can be accessed on command-line using the name of the wrapped function. The arguments of this function include the *Producer* instance itself, which contains a variety of task information of metadata such as the currently used dataset and config, and a plain *AwkwardArray* containing the chunk of events to be processed.

The decorator accepts arbitrary keyword arguments that can be utilized to generalize and customize the behaviour of the *Producer*. Most notably, the *uses* and *produces* can be utilized

[1] The default workflow implemented in*columnflow* includes a full pipeline that can be used to train a machine learning algorithm, including *k*-fold cross validation and utilizing data from multiple datasets and configs.

to load only columns that are required and to store columns on disk only when requested [2]. Additionally, a sandbox can be passed to the decorator to customize the software environment for different types of *Producers*.

An example command-line to invoke a workflow is shown in listing 2. The *TaskArrayFunctions* used here are encoded in the output paths of all (downstream) tasks, enabling users to create their analysis in a modular approach, where columns are only produced if requested and existing outputs are automatically reused.

[2]The same functionality could be covered through function tracing with placeholder objects, however, we actively decided for this more verbose option as we believe that the branching of typical analysis code depends on exceedingly dynamic conditions that might be not fully captured by tracing.

```python
@producer(
    uses={"Jet.{pt,eta,phi,mass}"},
    produces={"m_jj"},
)
def dijet_mass(self: Producer, events: ak.Array, **kwargs) -> ak.Array:
    # add four vectors
    dijet = events.Jet[:, 0] + events.Jet[:, 1]

    # store the dijet mass in a new column
    events = set_ak_column(events, "m_jj", dijet.mass, value_type=np.float32)

    return events


@producer(
    uses={dijet_mass, event_weights},
    produces={dijet_mass, event_weights},
)
def features(self: Producer, events: ak.Array, **kwargs) -> ak.Array:
    # combine multiple producers via nesting
    events = self[dijet_mass](events, **kwargs)
    if self.dataset_inst.is_mc:
        events = self[event_weights](events, **kwargs)
    return events
```

Listing 1. Example code snippet of a simple *Producer*. The *@producer* decorator creates a *Producer* subclass that can be accessed on command-line via the name of its wrapped function. The *dijet_mass Producer* requests to load the four-vector components of the `Jet` collection to create a column of the dijet mass, `m_jj`. The *features Producer* utilizes a nesting mechanism to run the *dijet_mass* and *event_weights Producers*. It requests to load and produce all columns that are requested by the *Producers dijet_mass* and *event_weights*.

```
law run cf.PlotVariables1D --version dev1 --workflow htcondor \
    --calibrators jec,jer --selector full --producers features \
    --processes tt,dy --variables "m_jj,jet*_{pt,eta}"
```

Listing 2. Example task call to trigger the workflow to create plots of one-dimensional variables. The *version* parameter is encoded in the output path to allow versioning. The *workflow* argument can be passed to specify whether to run the workload locally or via job submission. The *calibrators*, *selector*, and *producers* arguments define which *TaskArrayFunctions* are used within the workflow. The *processes* parameter is used to decide on which datasets to run the workflow and how to group datasets in the plot. The *variables* argument is used to request the observables of interest, including binning information.

5 Analysis configuration

The main analysis configuration is implemented with the *order* [13] package, which provides containers for different types of metadata such as physics processes, cross sections, and uncertainties. These containers can be modified by users with simple and readable *Python* code to define, for example, datasets of interest, systematic uncertainties, scale factor configurations, or any kind of auxiliary information.

6 Systematic uncertainties

In *columnflow*, the handling of systematic variations (shifts) is seamlessly integrated into the orchestrated workflow and can be configured by the user. The key concept is that tasks themselves define which shift they implement and which shift they depend upon (through upstream tasks). When a shift is requested for a task whose input or own behaviour does not depend on it, its nominal outputs can be reused. When a shift is implemented, the resulting output of this task is changed when a different shift is requested. This is typically achieved by declaring shift dependencies within a given *TaskArrayFunction* and by defining column aliases that encode which input columns to modify when implementing a shift. When a task depends on (but does not implement) a shift, column aliases do not need to be applied, but the varied shift needs to be encoded in the output, since the results have already been modified by an upstream task.

We typically distinguish between three types of shifts: dataset-specific shifts (e.g. generator tune), selection-dependent shifts (e.g. jet energy corrections), and weight-dependent shifts (e.g. pileup reweighting). A dataset-specific shift is implemented in `GetDatasetLFNs` by using a separate list of input files. No outputs can be reused for this type of shift. A selection-dependent shift usually modifies some input observables (e.g. `Jet.pt`) due to variations in some calibration. This will often result in modified selection results in `SelectEvents`, but outputs from the calibration can be reused when producing both the nominal and the varied calibrations at the same time. A weight-specific shift is typically implemented in `CreateHistograms` by modifying the event weight that is passed when filling a histogram.

7 Conclusion

With *columnflow*, we provide a fully orchestrated workflow, which implements the most important aspects of a common high-energy physics analysis. Users can pass custom data-processing code directly to the workflow, allowing for a high level of customizability. A variety of large-scale physics analyses within the CMS experiment are already being performed using *columnflow*, stress-testing the scalability of the framework both in complexity and in the amount of analyzed data. While a stable release is already public, a handful of improvements are still in the pipeline and will soon be published in a major release with a largely frozen API.

References

[1] G.L. Bayatyan, M. Della Negra, Foà, A. Hervé, A. Petrilli (CMS), CMS computing: Technical Design Report, Technical design report. CMS (CERN, Geneva, 2005), submitted on 31 May 2005, https://cds.cern.ch/record/838359

[2] M. Peruzzi, G. Petrucciani, A. Rizzi, for the CMS Collaboration, The NanoAOD event data format in CMS, Journal of Physics: Conference Series **1525**, 012038 (2020). 10.1088/1742-6596/1525/1/012038

[3] C.R. Harris, K.J. Millman, S.J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N.J. Smith et al., Array programming with NumPy, Nature **585**, 357 (2020). 10.1038/s41586-020-2649-2

[4] J. Pivarski, I. Osborne, I. Ifrim, H. Schreiner, A. Hollands, A. Biswas, P. Das, S. Roy Choudhury, N. Smith, M. Goyal, Awkward Array (2018). 10.5281/zenodo.4341376

[5] N. Smith, L. Gray, M. Cremonesi, B. Jayatilaka, O. Gutsche, A. Hall, K. Pedro, M. Acosta, A. Melo, S. Belforte et al., Coffea Columnar Object Framework For Effective Analysis, EPJ Web Conf. **245**, 06012 (2020). 10.1051/epjconf/202024506012

[6] T. Sterling, Condor: A Distributed Job Scheduler, in *Beowulf Cluster Computing with Linux* (MIT Press, 2001), pp. 307–350

[7] A.B. Yoo, M.A. Jette, M. Grondona, SLURM: Simple Linux Utility for Resource Management, in *Job Scheduling Strategies for Parallel Processing* (Springer Berlin Heidelberg, 2003), pp. 44–60, ISBN 978-3-540-39727-4

[8] K. Bos, N. Brook, D. Duellmann, C. Eck, I. Fisk, D. Foster, B. Gibbard, C. Grandi, F. Grey, J. Harvey et al., LHC computing Grid: Technical Design Report. Version 1.06 (20 Jun 2005), Technical design report. LCG (CERN, Geneva, 2005), https://cds.cern.ch/record/840543

[9] N. Richardson, I. Cook, N. Crane, D. Dunnington, R. François, J. Keane, D. Moldovan-Grünfeld, J. Ooms, J. Wujciak-Jens, Apache Arrow, arrow: Integration to 'Apache' 'Arrow' (2025), R package version 19.0.1, https://arrow.apache.org/docs/19.0/r/, https://github.com/apache/arrow/

[10] A.A. Ayllon, M. Arsuaga Rios, G. Bitzes, F. Furano, O. Keeble, A. Manzi, Making the most of cloud storage - a toolkit for exploitation by WLCG experiments, J. Phys.: Conf. Ser. **898**, 062027 (2017). 10.1088/1742-6596/898/6/062027

[11] M. Rieger, End-to-End Analysis Automation over Distributed Resources with Luigi Analysis Workflows, EPJ Web of Conferences **295**, 05012 (2024). 10.1051/epjconf/202429505012

[12] E. Bernhardsson, E. Freider, Luigi, https://github.com/spotify/luigi, last visit: 18.02.2025

[13] M. Rieger, order, https://github.com/riga/order, last visit: 18.02.2025