PyHEP.dev 2024 Workshop Summary Report, August 26–30 2024, Aachen, Germany

Azzah Alshehri[1] **Jan Bürger**[2] **Saransh Chopra**[3] **Niclas Eich**[4] **Jonas Eppelt**[5] **Martin Erdmann**[6] **Jonas Eschle**[7] **Peter Fackeldey**[2,4] **Maté Farkas**[4] **Matthew Feickert**[8] **Tristan Fillinger**[9] **Benjamin Fischer**[4] **Stefan Fröse**[2,10] **Lino Oscar Gerlach**[11] **Nikolai Hartmann**[12] **Alexander Heidelbach**[5] **Alexander Held**[8] **Marian I Ivanov**[13] **Josué Molina**[14] **Yaroslav Nikitenko**[4] **Ianna Osborne**[11] **Vincenzo Eduardo Padulano**[15] **Jim Pivarski**[11] **Cyrille Praz**[9] **Marcel Rieger**[16] **Eduardo Rodrigues**[17] **Oksana Shadura**[18] **Juraj Smieško**[15] **Giordon Holtsberg Stark**[19] **Judith Steinfeld**[2] **Angela Warkentin**[2]

[1] *University of Hafr AlBatin*
[2] *ErUM-Data-Hub*
[3] *University College London (UCL)*
[4] *RWTH Aachen University*
[5] *Karlsruher Insititute of Technology (KIT)*
[6] *Rheinisch Westfaelische Tech. Hoch.*
[7] *Syracuse University*
[8] *University of Wisconsin-Madison*
[9] *High Energy Accelerator Research Organization (KEK)*
[10] *TU Dortmund University*
[11] *Princeton University*
[12] *Ludwig Maximilians Universitat*
[13] *GSI - Helmholtzzentrum fur Schwerionenforschung GmbH*
[14] *EAP Zamorano*
[15] *CERN*
[16] *Hamburg University*
[17] *Oliver Lodge Laboratory, University of Liverpool*
[18] *University of Nebraska Lincoln*
[19] *SCIPP, University of California Santa Cruz*

Abstract: The second PyHEP.dev workshop, part of the "Python in HEP Developers" series organized by the HEP Software Foundation (HSF), took place in Aachen, Germany, from August 26 to 30, 2024. This gathering brought together nearly 30 Python package developers, maintainers, and power users to engage in informal discussions about current trends in Python, with a primary focus on analysis tools and techniques in High Energy Physics (HEP).

The workshop agenda encompassed a range of topics, such as defining the scope of HEP data analysis, exploring the Analysis Grand Challenge project, evaluating statistical models and serialization methods, assessing workflow management systems, examining histogramming practices, and investigating distributed processing tools like `RDataFrame`, `Coffea`, and `Dask`. Additionally, the workshop dedicated time to brainstorming the organization of future PyHEP.dev events, upholding the tradition of alternating between Europe and the United States as host locations.

This document, prepared by the session conveners in the weeks following the workshop, serves as a summary of the key discussions, salient points, and conclusions that emerged.

Contents

1 Introduction

The PyHEP.dev workshop [1], the second in the "Python in HEP developers" workshop series of the HEP Software Foundation (HSF) [2], took place in Aachen, Germany, on August 26–30, 2024. Thirty Python package developers, maintainers, and power users, all of whom are authors of this report, gathered together in an informal setting to discuss relevant and timely trends in Python, largely targeting end-user analysis. The following topics were central points of discussion: how PyHEP does (or does not) relate to physicists' needs; the Analysis Grand Challenges that run analyses at scale at analysis facilities; leveraging key packages from the ecosystem; the development of statistical packages, models, interfaces, and serialization; workflow management systems; histogramming; and key distributed processing tools like `RDataFrame` [3], `Coffea` [4], and `Dask` [5]. Finally, we brainstormed the organization of future PyHEP.dev workshops and how to connect PyHEP activities with the larger scientific software developer community.

Participation was in-person, but a Zoom connection with an *Owl* camera also broadcast all sessions to participants who could not attend. About four to five people joined remotely, mostly listening, though one talk was presented from Japan.

In the week following the workshop, the organizers curated this document, which is meant to be a summary reflecting the main discussions, the key points raised, and our collective conclusions. The document was written collaboratively by nearly all participants, refined into a coherent whole, and reviewed by all participants in its final form before it was made public.

2 Scope of the workshop

Our first discussion on Monday aimed to define the scope of the workshop, ensuring that all participants shared a common understanding of what "HEP data analysis" entails, both in terms of software and infrastructure. For the purpose of the workshop, we focused on the following understanding of the term:

> Data analysis transforms raw measurements into human-interpretable formats, such as tables and visualizations, for the extraction of physics quantities of interest. The process includes summary statistics, statistical inference, and machine learning. The act of analyzing data is highly iterative, changing strategy in response to partial results.

We interpreted High Energy Physics (HEP) broadly to include any experimental physics of fundamental particles. Many — but not all — of the PyHEP.dev workshop participants collaborate on LHC (Large Hadron Collider [6]) and flavor physics experiments, primarily ATLAS [7], CMS [8], LHCb [9], and Belle II [10]. In particular, the tools developed by PyHEP.dev participants target the phase of work after central production (e.g., after triggers and event reconstruction) through final publication.

During the discussion, we recognized that participants had different views about whether high-level physics/statistics should be strictly separated from low-level computing details. Some 5–10 years ago, there was a much stronger consensus that data analysts should be completely isolated from computing details, possibly through a new domain-specific language specialized for HEP. In this model, everything concerning where and how data are loaded and computations are performed would be handled automatically. We noted that this is no longer a goal shared by all developers; some physicist-users will need to be aware of certain computational details. Specifically, it needs to be possible to break through the high-level abstraction to control low-level details in some situations, particularly to optimize for best performance and to enable debugging. This was crucial for the

"200 Gbps Analysis Challenge" run by the Institute for Research and Innovation in Software for High Energy Physics (IRIS-HEP) in Spring 2024 [11], which required significant debugging and coordination between the analyzers and the system facilitators (acknowledging that it was the first test of its kind). We did agree, though, that HEP analysis tools should initially or primarily present a high-level view, at least for first-time users, in which low-level computing details are only encountered by choice.

It was mentioned that visualizations do not need to be limited to two-dimensional plots. It was also proposed to improve physics visualizations by using graphics formats native to final publications, such as PGF/TikZ [12] for LaTeX [13] articles (publications and Beamer [14] slides).

Another aspect of HEP analysis is its long-term nature. Data safety — that is, synchronization and backups — was also mentioned in a dedicated presentation as an essential condition for analysis preservation.

Due to the collaborative nature of physics data analysis, data exchange formats were discussed at length. Agreement on common formats, and easy conversions between them, are key to interoperability. In particular, we talked about the HS3 HEP Statistics Serialization Standard [15] in `JSON`/`YAML`, `Dask` graphs for delayed computations, and `Awkward Array`s [16] for Python-Julia integration. Although some users and developers use Python's pickle format, especially for temporary storage or transmitting data over a network, pickle is not stable across software version updates. It is not interchangeable with C++ and other languages, and it is fundamentally unsafe, since loading a pickle file can execute arbitrary code. A similar complaint was raised about `YAML`, which can run arbitrary code, but most `YAML` deserializers have a `safe_load` function, which refuses to run code embedded in the `YAML` document. Developers should use such functions exclusively. (The equivalent option does not exist for pickle.)

We also discussed the transition of software from development stages to maintenance stages of its life cycle. Some Python projects that began in an exploratory way had to stabilize to remain useful to those who adopted them, either direct users or downstream dependencies; interface changes would disrupt these users. Maintaining stable software is a different activity from developing new, prospective software, requiring a different class of developers, such as postdoctoral researchers and Research Software Engineers (RSEs), rather than undergraduate or graduate students. In many cases, it requires different funding, since most funding sources target research and development. The problem is not unique to HEP; it also affects Astronomy and the Astropy project [17] in particular. While there are some organizations that sponsor or fund open-source scientific Python projects — notably NumFOCUS [18], Quansight [19], Anaconda [20], and QuantStack [21] — none of these are known to fund software developers outside of their organizations. NumFOCUS is a non-profit organization that helps with financial administration and legal services, while Quansight, Anaconda, and QuantStack are commercial companies that only employ a few open-source developers; most of their revenue comes from consulting. At the end of this discussion, we did not identify, unfortunately, any funding sources that could support postdoctoral researchers and RSEs to maintain foundational Python software for analysis.

Finally, we discussed the impact that large computations have on climate change [22]. We cannot ignore the environmental impact of our research, especially the significant CO_2 emissions and energy consumption associated with the powerful computing resources we rely on. Recognizing that our actions have consequences, we need to act decisively and quickly to mitigate the damage we are causing. We need to raise awareness, share knowledge, and optimize our data and code production in the short term. Looking further ahead, we must adapt our computing practices to align with the availability of renewable energy sources. We concluded that the time to act is now, and we must work together to ensure a sustainable future for both our research and our planet.

3 Analysis Grand Challenge

HEP analyses, as described in the previous section, vary widely depending on the target experiment, the university group, or specific user requirements. This makes it difficult to quantify the properties of "a typical HEP analysis", though some kind of benchmark is needed to guide discussion about what is easy or hard for physicists to do, what has acceptable performance, and what needs improvement. In this regard, the Analysis Grand Challenge (AGC) project [23] defines a sample analysis task that captures aspects of relevant physics analysis problems, including the treatment of systematic variations, statistical modelling and inference, and machine learning. A variety of implementations have been developed for this task, allowing us to probe user experience and interoperability, and helping to center community discussions around a common benchmark. The first implementation makes use of many tools in the Python HEP ecosystem and, in particular, a stack of Scikit-HEP libraries [24, 25]. Although the workflow shown in the AGC is mostly based around ATLAS and CMS physics analyses, it still serves as a tutorial that shows how to build an end-to-end HEP analysis pipeline more generally, since it stresses all of the computational features of a HEP analysis task [26].

Both the AGC presentation and the following demo brought up many questions from the PyHEP.dev participants, some of whom were introduced to `Dask` for the first time. The demo of the AGC implementation using `dask-awkward` [27] included simple examples of that library [28]. It highlighted how `Uproot` [29], `Awkward Array`, `Dask`, and `Coffea` can be used together for data processing and visualization. Upon seeing the `Dask` visualizations, some participants were concerned that users would have to become experts in a whole new set of monitoring tools, but we concluded that `Dask` visualizations are for workflow debugging and optimization, not for physics. Key points were raised about the transition from vectors implemented in `Coffea` to the `Vector` [30] library and the need for static type-checking in `Awkward Array`'s `behaviors`.

The discussion about distributed computing and the approach used for the IRIS-HEP Data Analysis Pipeline (IDAP) 200 Gbps Challenge [31] spanned a broad range of topics: caching strategies, handling the memory footprint, and the Aachen team's approach of using high-memory `Dask` workers for giant histograms.

The AGC project revealed a number of topics to consider when running analysis pipelines in this format across a variety of facilities, including the following:

- **memory footprint**, which relates to choosing a suitable number of partitions for each dataset;
- **integrating heterogeneous computing resources**, such as GPUs;
- **scaling strategies**, such as automatic scaling provided by `Dask`.

The best configuration may vary by facility and use case. Interactions with companies like Coiled.io [32], the core `Dask` team or the broader `Dask` community may also yield useful insights for good patterns to adopt.

A discussion followed on differences between `Dask`'s collections (`dask.array`, `dask.frame`, `dask.bag`, and `dask_awkward`) and how `Dask` handles partitions. We discussed performance profiling, focusing on bottlenecks such as decompression and array concatenation. The scalability of the `Dask` network, especially with `Kubernetes`, was also examined. Concerns about `Dask`'s memory limits for workers and the potential for heterogeneity were raised, with suggestions to get support from `Dask` experts beyond HEP. The session emphasized the importance of addressing these challenges and keeping track of them for future collaboration and problem-solving.

The PyHEP.dev participants were interested in having more transparent environment handling for easier debugging and development at `coffea-casa` [33] analysis facilities.

4 Statistical Models, Interfaces and Serialization

The HEP community uses many different statistical tools. At PyHEP.dev we discussed some of the most popular packages and how they interact with each other. In the following, we present how these tools can fit together to build statistical models and perform experiment-specific inference. In addition, we also discuss how models may be serialized for out-of-core computations and later retrieval.

First, we made a distinction between "open-world" probability tools and specialized ones. Specialized tools target a specific statistical problem, and the limited scope allows them to reduce boilerplate code or even be configured declaratively, which eases the path to serialization. Open-world tools are intended for users to construct their own specialized models, and that open-ended scope makes it harder to encapsulate all the things the tool can do, especially if it allows users to write their own functions. `RooFit` [34] and `zfit` [35] are examples of open-world tools, as they allow users to build a likelihood using any probability distribution, both using pre-defined components in the library as well as arbitrary functions. Thus, they allow both binned and unbinned statistical inference. However, they necessarily have a steeper learning curve, which can be mitigated by higher-level interface layers such as `xRooFit` [36]. New cross-cutting features are harder to add to open-world tools since they must ensure support across the breadth of the library.

This led to the development of specialized tools that intentionally constrain their scope for specific purposes, such as `pyhf` [37], `cabinetry` [38], `evermore` [39], `Gammapy` [40], `CMS Combine` [41], `HistFitter` [42], `Combiner` [43], `RooUnfold` [44], `HAverager` [45, 46], and `BLUE` [47]. The scopes of these tools partially overlap. For example, while a specialized tool like `pyhf` is flexible enough to allow data-driven estimation of fake events via the ABCD method [48], the interface is challenging for a typical user and a wrapper, `abcd_pyhf` [49], was written to smooth that experience.

Next, we considered function minimizers and statistical inference tools. There is a natural boundary between model-building, function minimizing, and statistical inference: building models is one well-defined scope, minimizing negative log-likelihoods is another, and interpreting the minimum as a statistical inference is yet another. Software can be better scoped to focus on each of these topics individually, but only if we can reach an agreement on protocols for common formats for likelihoods and their minima. By way of analogy, a similar factorization was possible in Scikit-HEP's histogram packages, which focus on fast histogram filling, plotting, slicing, projecting, and rebinning, thanks to agreements on the Unified Histogram Interface (UHI) [50]. An exchange format for likelihoods would have to standardize the call signature, parameter configurations (bounds, initial values, fixed parameters), and the information returned about the minimum of the likelihood. Statistical inference packages need to have enough information in this output to be specialized for particular experiments.

Although protocols only standardize Python-callable interfaces, serialization formats specify byte-formats for analysis steps that can be saved for later processing, transmitted through a network, or passed between programs written in different languages. We discussed the ongoing effort to serialize statistical models using HS3 in `JSON`/`YAML`, but also considered the possibility that users would want to serialize hand-written functions. Formats such as StableHLO IR and LLVM IR encode functions that can be compiled and used in any language, by feeding them into industry-standard compiler chains — XLA [51] and LLVM [52], respectively — without any dependence on big libraries such as `JAX` [53, 54] or `TensorFlow` [55]. The `dill` library [56] can save executable Python functions, but only as black boxes that can't be inspected or generalized. Also, whereas Intermediate Representations (IRs) can be restricted to mathematical computations only, functions saved with `dill` can run arbitrary code. HS3 without arbitrary functions is easier to inspect and identify the physics-motivated parts than any serialization format for arbitrary functions, and providing both necessarily introduces two ways

of doing the same thing. StableHLO IR guarantees backward compatibility.

The HS3 format is still under development and needs further consolidation. As of yet, serializing arbitrary IR functions is not part of the HS3 scope.

5 Workflow Management Systems

High-level physics analyses usually consist of a considerable amount of logically separated workloads. In general, the interface between these workloads does not rely on an event-by-event (chunk) data flow such as utilized by, for example, experiment software or parallel computing libraries like `Dask`. The workloads to perform a specific analysis often form a loose collection of inhomogeneous procedures, encoded in executable files such as Shell and Python scripts, and are often executed manually. Hereby, their execution order is dictated by the dependencies between them in terms of the existence of results of one or more previous procedures. Beyond a certain scale and complexity, the manual steering of analysis workflows can be time-consuming, prone to errors, and potentially leads to undocumented relations between workloads.

A Workflow Management System (WMS)[1] is software to describe, manage, and execute a sequence of arbitrary workloads. It is agnostic to the concrete type of operations and the data these workloads are performed on. In the HEP context, a WMS allows one to describe all necessary steps of an analysis bridging Monte Carlo simulation (MC) production, reconstruction, and statistical analysis. Common implementations also provide tools to split execution logic from computing infrastructure.

`Dask` can be understood as a specialized WMS with more focus on the parallelization of loops and array-based calculation as well as the scalability of these processes. Consequently, general WMS engines and `Dask` can be used complementary in HEP analysis workflows. This allows for both bridging heterogeneous workflows and profiting from `Dask`'s parallelization mechanisms.

Currently, several different WMS engines are used in HEP. At PyHEP.dev, `Luigi` [57] was most thoroughly discussed (not counting `Dask`), followed by `Snakemake` [58, 59], and Apache `Airflow` [60] was only briefly mentioned. Users and maintainers of `Luigi` extensions were present and talked about their work.

WMS engines are increasingly common in HEP, both for end-user analysis and for services such as calibration. We identified the following cases:

- **ATLAS**
  - `Yadage` [61] workflows used in `RECAST` [62, 63] full-analysis pipelines, which most ATLAS searches for physics beyond the Standard Model implement as part of the analysis preservation process prior to publication.
- **CMS**
  - `LAW` [64] workflows in end-user analysis, such as `columnflow` [65] and `PocketCoffea` [66].
- **LHCb**
  - `Snakemake` workflows in end-user analysis. In common cases, `Snakemake` runs in GitLab CI, and one physics working group went as far as having a series of workflows implemented for many of the working group analyses, providing group-level maintenance of templates.
- **Belle II**

[1] Commonly referred to as "workflow engines" or "workflow languages".

- `b2luigi` [67, 68] in automated, collaboration-wide processes, such as systematic uncertainty calculations and data-MC validation.
- `b2luigi` and `Snakemake` in end-user analyses.
- Apache `Airflow` steers the calibration framework.

`Luigi`-based workflows are primarily limited by the built-in scheduler's capacity to scale, up to about 10,000 dependencies. These workflows can't be further granularized, which puts an upper limit on the scope of debuggable code. Since no alternate schedulers exist for `Luigi`-based workflows, scaling beyond this limit means rewriting the workflow. An example case where this comes up is per-dataset debugging. However, there are simple ways to overcome this limitation in practical cases such as step-wise execution of parts of exceedingly large dependency trees.

At this workshop, two WMS extensions were presented in detail: `b2luigi` and `LAW`.

`b2luigi` extends `Luigi` to provide specialized solutions for situations that arise in HEP. `b2luigi` integrates with `HTCondor` [69] and `LSF` [70], which are often found in HEP computing clusters, and it, therefore, allows users to write a workflow for both of these batch systems. Furthermore, the package provides mechanisms for task-specific and global settings, such as I/O, software environment, and batch system user configurations and quality-of-life improvements for analysts. `b2luigi` provides specialized interfaces for Belle II workflows, allowing particularly smooth interaction with the Belle II software framework. Workflows such as the automated validation framework, the systematic uncertainty framework, and many Belle II physics analyses have been automated using `b2luigi`. Maintenance of `b2luigi` is now in the hands of the Belle II collaboration and will be followed up by its software group.

`LAW` stands for `Luigi` Analysis Workflow, and it is also based on `Luigi`. `LAW` establishes a generic design pattern for analyses of arbitrary scale and complexity by providing building blocks to integrate with remote resources in a generic, interchangeable way so that it does not depend on a particular cluster infrastructure. In particular, it completely separates analysis algorithms from run locations, storage locations, and software environments. To cope with sophisticated demands of end-to-end HEP analyses, `LAW` supports job execution on `WLCG` infrastructure (`ARC`, `gLite`, `CMS-CRAB`) [71] as well as local computing clusters (`HTCondor`, `Slurm` [72], `LSF`), remote file access via various protocols using the Grid File Access Library (GFAL2) [73], and an environment sandboxing mechanism with support for sub-shells and virtual environments, as well as Docker [74] and Singularity [75] containers. `LAW` ultimately aims to provide analysis preservation out-of-the-box. `LAW` is developed in an open-source, experiment-agnostic way, and its user base has steadily increased over the past eight years.

`b2luigi` and `LAW` implement similar functionalities differently, with `LAW` presenting a more extensive feature set. While `b2luigi` intends to be a drop-in replacement for `Luigi`, `LAW` is more focused on extending the overall library with user-targeted features. What they have in common is extensive support for typical HEP batch systems, easier and more automated handling of `Luigi` targets and steering of user environments.

We discussed a possible merge of these packages, though it would be quite challenging at this stage. Beyond the technological differences and differences in project goals, both packages already have many users who depend on them as foundational libraries.

We also discussed the future of these WMS extensions: parameter simplification, developments related to Python's Global Interpreter Lock (GIL), and upstreaming features to `Luigi`.

On the topic of parameter simplification, the fact that large dependency trees generate a large number of parameters to configure complicates user code and makes workflows less flexible. We discussed ways to implicitly control the branching of Directed Acyclic Graphs (DAGs), expand the ex-

isting `Parameters` object in `Luigi`, and describe complex dependency trees with aggregation schemes. We identified options to implement these ideas and planned to follow up on them together.

With ongoing developments in Python to make the GIL optional—such as PEP 703 [76]—there is potential to improve the performance of core elements of `Luigi`, such as the parallel determination of the dependency tree that constitutes the workflow. Currently, this computation is being carried out in a single thread, since the GIL would prevent performance advantages from multiple threads. (Multiprocessing is not a viable option because of the need to share data between threads.) Our community should identify whether the option to remove the GIL and increase parallel processing can improve the performance of `Luigi`'s DAG-building process.

`b2luigi` and `LAW` extend `Luigi` with features that, arguably, should be in `Luigi` itself. We identified a handful of features that could be upstreamed. However, if the community decides to use `Luigi` to a wider extent in mission-critical projects, we'll need to have a larger discussion with future stakeholders to unify requests for features for the HEP community with the rest of the world. Ideally, members of the HEP community would have to become core developers and maintainers of the `Luigi` project.

6 Histogramming

Histogramming concerns three major topics: the production, presentation, and serialization of histograms. Additionally, the accumulation — that is, merging — of histograms from concurrent histogram–filling processes, where each process fills a separate copy of the histogram, into a single summed histogram set can be considered separately. All of these topics were raised during the presentations and the discussions that followed.

The production of histograms had once been a contentious topic in Python, with dozens of libraries providing mutually-incompatible histogram objects. This changed with `Boost::Histogram` and its Python interfaces, `boost-histogram` [77] and `Hist` [78]. In addition, the Unified Histogram Interface (UHI) defined protocols to share boost-style histogram interfaces between libraries, so libraries that adhere to these protocols can be compatible without sharing memory layouts. For example, `dask-histogram` implements delayed histograms and `cuda-histogram` implements histograms on GPUs, which are not `Boost::Histogram` objects, but can be readily converted. Currently, `Boost`-style histograms are still incompatible with `ROOT`'s [79] histograms (`Uproot`'s conversion from `Boost` to `ROOT` is lossy), but `ROOT` will be adopting UHI in the near future. [80]

`Boost-histogram` is now widely used for filling histograms in Python on CPUs. Histogram-filling on GPUs with `cuda-histogram` is in development. Some cases, such as dynamically growing axes based on incoming data, are not currently possible in the Python interfaces to `boost-histogram` because a different memory model would be needed.

Until now, histograms have been stored in memory as a rectilinear N-dimensional array of values. Such an array cannot grow dynamically, which prevents growing axes and sparse representations of very large histograms. We discussed new memory layouts, such as `scipy.sparse arrays`, `Blosc2`-compressed arrays, in-memory databases, and file-based formats like `HDF5` (`h5py`) [81] and `Zarr` [82]. Some of these have internal search trees, some have internal chunking, and some are simply compressed in a random-accessible way to allow data to grow. These formats are particularly attractive when bin access is not uniform but concentrated in a small region of the hypercube. Disk formats like `HDF5` and `Zarr` have the added advantage that serializing and saving the histogram are not a separate step. With all of the data safely on disk, the process can address histograms that would be too large to fully load into available RAM.

Before Dask and WMS engines (see above), event-processing jobs had to be followed by an accumulation job, which could be very time-consuming if the number of parallel tasks n (and therefore histogram copies) was large. Dask and WMS engines are both capable of tree-reduction, which adds histograms in $\mathcal{O}(\log n)$ steps instead of $\mathcal{O}(n)$ steps. However, accumulation is still an issue for very large histograms (or very large histogram sets) because enough memory may not be available for n copies of the histogram data before accumulation. In fact, the boost-histogram model of histograms with arbitrarily many axes encourages this: the number of histogram bins grows exponentially with the number of axes. For these cases, we must consider a single copy of the histogram with atomic operations to increment bin contents.

In addition to ROOT's histogram presentation (plotting), many Python packages provide visualization routines. Nearly all of these are based on Matplotlib [83], and two are widely used in HEP: mplhep [84] and plothist [85]. These two libraries were developed because Matplotlib, by itself, has poor support for plotting pre-binned histograms, which are surprisingly uncommon outside of HEP. Although they have substantial overlap, they also have different areas of focus: mplhep provides style templates to exactly match the fonts, line widths, and branding of the LHC experiments, while plothist provides more ways to compare data, such as measurements versus stacked MC simulations. Users will likely want features of both, so we discussed in depth the possibilities of merging the packages or keeping them separate but linking them with new protocols. In the end, the developers of both libraries decided to combine efforts in a single package.

As noted above, ROOT will soon adhere to the UHI protocol, which will greatly enhance interoperability between Python/Matplotlib and ROOT histogram filling and visualization.

Finally, we discussed histogram serialization. An upcoming release of boost-histogram (the Python package) will support serialization using a JSON-based schema and bin contents in HDF5, with the intention to port this format to other containers, such as ROOT and Zarr. Currently, many HEP users and developers serialize histograms by pickling them, which has performance and security issues mentioned above. We also agreed that histograms should carry arbitrary (JSON-like) metadata, allowing new use cases to be developed without requiring all histogram libraries to support them as built-in features.

7 RDataFrame, Coffea, and Dask

Distributed computing has always played a crucial role in HEP, from batch systems with manual job submission to automated WLCG gridware and orchestration systems like Dask [86, 87], Apache Spark [88], and Stellar HPX [89, 90], as well as WMSs like Luigi and Snakemake. Recently, orchestration systems have changed how we think about distributed computing by removing much of the boilerplate and setup required to parallelize applications. In particular, Dask separates the process of describing computation DAGs from the process of executing them, which allows for alternative schedulers, such as Ray [91] and TaskVine [92].

Dask has gradually increased in prominence in the Python software stack, evolving from one choice of backend (the other usually being Spark) to a system that users address directly. Some tools have started to rely on it, as well as computing infrastructures [93, 33, 94] that feature integration between a Jupyter front-end and Dask diagnostics.

One question that we discussed at PyHEP.dev, and that deserves discussion beyond the workshop, is whether our community is best served by the default Dask scheduler or whether we should investigate different scheduling systems like Ray and TaskVine. On the one hand, the Dask scheduler has the clear advantage of being able to connect to many types of resource managers, most notably those

batch systems that are already widely employed in our field, such as `HTCondor` and `Slurm`. This is handled through concrete implementations of cluster connection classes that all refer back to a single user-facing entry point, the `dask.distributed.Client`. This allows the entire set-up configuration to be abstracted away from users by providing cluster configuration options upfront by the computing facility, in a way that can be reused in any Python script. On the other hand, the default `Dask` scheduler orchestrates all tasks through a single-threaded process and it typically isn't used for HEP-scale, multi-user, persistent `Dask` clusters; more often, scientists launch a single-user `Dask` cluster for a specific project, then shut down the entire cluster. In addition to implementing a distributed scheduler, `TaskVine` is being developed by computer scientists at the University of Notre Dame, with whom the HEP community is actively engaging.

The architecture of `Dask` clearly separates DAG building from execution. In fact, these are two Python packages that can be installed separately. The `Dask` graph is a format that is widely used in the sciences, and can be considered a serialization format for DAGs that can be inspected and optimized by other libraries. In our discussions, however, we concluded that we do not know if the `Dask` DAG format is sufficiently generic for all future scheduling systems: as a data structure, it is by nature static, which could prevent a future analysis tool form dynamically adapting to specific requirements of analyses (e.g., treating different data samples) or optimizing the application (e.g., with a dynamic tree-reduction step that picks up already finished tasks instead of waiting for pre-planned tasks to finish). These problems could be solved by leveraging the `Dask` client and injecting custom workflows in the nodes of the graph, a technique already used by `RDataFrame`, although this goes beyond the boundaries of what is directly offered by the tool.

Moreover, user experiences with `Dask` in Python HEP analyses encounter performance limitations in very large graphs that are slow to build or optimize, as well as memory issues that cause `Dask` workers to fail and restart. HEP use-cases are pushing the complexity limits of `Dask` DAGs, particularly when all systematic variations are included in the computation. `Dask` DAGs are, after all, Python dictionaries — not a packed binary format. One solution that we discussed was to add an opt-in function to tell `Dask` that a particular function can be computed on all partitions independently, similar to the `Coffea Processor` before `dask-awkward`, so that `Dask` sees a complex function with many operations as a single DAG node.

User-friendly analysis frameworks like `PocketCoffea` simplify the set-up of HEP workflows by expressing them in a declarative configuration. These frameworks build on top of `Coffea` and `RDataFrame` to provide further abstractions for the most common workflows. Finding a good balance between user-friendly abstractions and flexibility is crucial to avoid restrictive setups.

8 Future of PyHEP.dev

Next year (2025), PyHEP.dev will be in the United States. In our final session, we discussed whether it should return to Princeton as in 2023, be co-located with the international SciPy Conference [95] (in Tacoma, Washington, USA for 2025), or whether we should consider another venue. There were two arguments for co-locating with SciPy: it encourages PyHEP developers to interact with the larger scientific Python community, and an increasing number of high-energy physicists have been attending SciPy in recent years. Additionally, stringent travel budgets and environmental considerations related to air travel mean that combining PyHEP and SciPy into a single trip is advantageous.

An argument against co-locating with SciPy is that the combined conference would span more than a week, which may be longer than some participants can justify. (Notably, no one was interested in making PyHEP.dev merely a Birds-of-a-Feather session within SciPy, reducing a multi-day workshop

to a one hour session.) Historically, the third general PyHEP workshop was planned to be co-located with SciPy in 2020, but this was disrupted by the COVID-19 pandemic. In a sense, co-locating PyHEP.dev with SciPy in 2025 revisits that idea.

Looking further ahead, we considered future PyHEP.dev workshops in Europe, such as 2026. Connecting with ErUM-Data Hub [96], our host and co-organizer this year, was incredibly fruitful, but we acknowledge that the ErUM-Data Hub team's generosity in hosting the 2024 workshop was a one-time event, as ErUM-Data Hub must diversify across scientific domains. We also considered the possibility of co-locating with EuroSciPy [97], though this remains very provisional.

Another idea raised was to consider locations beyond the United States and Europe, to enable broader participation from collaborators. With the success of HSF-India [98] and strong interest from the Indian community in computation in HEP, we discussed the possibility of a future PyHEP.dev in India. However, this must be balanced against the travel constraints, budgets, and CO_2 emissions associated with the core PyHEP developers traveling longer distances.

9 Summary

The workshop was a great success. The format of short morning talks, longer afternoon discussions, and hackathon free time led to many issues being addressed, and some even resolved through code.

10 Acknowledgments

We heartily thank ErUM-Data Hub for their gracious assistance in locally organizing and financially supporting the workshop. It couldn't have been done without all of your help!

Also, we thank the Python Software Foundation (PSF) and the National Science Foundation (grants OAC-2103945 and PHY-2323298) for their financial support.

References

[1] *PyHEP.dev 2024 - "Python in HEP" Developer's Workshop*. https://indico.cern.ch/e/PyHEP2024.dev.

[2] *HEP Software Foundation (HSF)*. https://hepsoftwarefoundation.org/.

[3] Danilo Piparo et al. "RDataFrame: Easy Parallel ROOT Analysis at 100 Threads". In: *EPJ Web Conf.* 214 (2019), p. 06029. DOI: 10.1051/epjconf/201921406029.

[4] Lindsey Gray et al. *coffea*. DOI: 10.5281/zenodo.3266454. URL: https://github.com/CoffeaTeam/coffea.

[5] *Dask*. https://www.dask.org/.

[6] K. M. Potter. "The Large Hadron Collider (LHC) project of CERN". In: *28th International Conference on High-energy Physics*. Aug. 1996, pp. 1760–1764.

[7] G Aad et al. "The ATLAS Experiment at the CERN Large Hadron Collider". In: *JINST* 3 (2008). Also published by CERN Geneva in 2010, S08003. DOI: 10.1088/1748-0221/3/08/S08003. URL: https://cds.cern.ch/record/1129811.

[8] S Chatrchyan et al. "The CMS experiment at the CERN LHC. The Compact Muon Solenoid experiment". In: *JINST* 3 (2008). Also published by CERN Geneva in 2010, S08004. DOI: 10.1088/1748-0221/3/08/S08004. URL: https://cds.cern.ch/record/1129810.

[9] A. A. Alves Jr. et al. "The LHCb detector at the LHC". In: *JINST* 3 (2008), S08005. DOI: 10.1088/1748-0221/3/08/S08005.

[10] J. Kahn. "The Belle II Experiment". In: *CERN-BINP Workshop for Young Scientists in e+e- Colliders*. 2017, pp. 45–54. DOI: 10.23727/CERN-Proceedings-2017-001.45.

[11] B. P. Bockelman. *Demonstrator Analysis 200 Gb/s*. https://indico.cern.ch/event/1369601/timetable/?view=standard#72-demonstrator-analysis-200-g.

[12] Till Tantau. *The TikZ and PGF Packages. Manual for version 3.0.0*. Dec. 20, 2013. URL: http://sourceforge.net/projects/pgf/.

[13] *LaTeX – A document preparation system*. https://latex-project.org/.

[14] *The beamer class*. https://tug.ctan.org/macros/latex/contrib/beamer/doc/beameruserguide.pdf/.

[15] *HS3 - HEP Statistics Serialization Standard*. https://github.com/hep-statistics-serialization-standard/hep-statistics-serialization-standard.

[16] *Awkward Array*. https://awkward-array.org/.

[17] *The Astropy Project*. https://www.astropy.org/.

[18] *NumFOCUS - Non-profit organization promoting accessible and reproducible scientific and technical computing*. https://numfocus.org/.

[19] *Quansight*. https://quansight.com/.

[20] *Anaconda*. https://www.anaconda.com/.

[21] *QuantStack*. https://quantstack.net/.

[22] Marilyn Cruces et al. "Resource-aware Research on Universe and Matter: Call-to-Action in Digital Transformation". In: *arXiv* 2311.01169v1 (2023). DOI: https://doi.org/10.48550/arXiv.2311.01169.

[23] *The IRIS-HEP Analysis Grand Challenge*. URL: https://agc.readthedocs.io/.

[24] *Scikit-HEP*. https://scikit-hep.org/.

[25] Eduardo Rodrigues et al. "The Scikit HEP Project – overview and prospects". In: *EPJ Web Conf.* 245 (2020). Ed. by C. Doglioni et al., p. 06028. DOI: 10.1051/epjconf/202024506028. arXiv: 2007.03577 [physics.comp-ph].

[26] *AGC Wishlist*. https://github.com/iris-hep/analysis-grand-challenge/issues/224.

[27] *dask-awkward*. https://dask-awkward.readthedocs.io/.

[28] *IDAP 200Gbps ATLAS*. https://gist.github.com/alexander-held/aebf2b726176e7f99e58ca4a855a0494.

[29] *Uproot*. https://uproot.readthedocs.io/.

[30] *Vector*. https://vector.readthedocs.io/.

[31] https://github.com/iris-hep/idap-200gbps-atlas/blob/main/materialize_branches.ipynb.

[32] *Coiled. A Dask Company*. https://www.coiled.io/.

[33] *Coffea-Casa Project*. https://coffea-casa.readthedocs.io/.

[34] Wouter Verkerke and David P. Kirkby. "The RooFit toolkit for data modeling". In: *eConf* C0303241 (2003). Ed. by L. Lyons and Muge Karagoz, MOLT007. arXiv: physics/0306116.

[35] Jonas Eschle et al. "zfit: scalable pythonic fitting". In: *SoftwareX* 11 (2019), p. 100508. DOI: 10.1016/j.softx.2020.100508. arXiv: 1910.13429 [physics.data-an].

[36] *xRooFit. Extra tools for RooFit projects*. https://gitlab.cern.ch/will/xroofit.

[37] Lukas Heinrich et al. "pyhf: pure-Python implementation of HistFactory statistical models". In: *Journal of Open Source Software* 6.58 (2021), p. 2823. DOI: 10.21105/joss.02823. URL: https://doi.org/10.21105/joss.02823.

[38] Alexander Held et al. *cabinetry*. DOI: https://doi.org/10.5281/zenodo.4742752. URL: https://doi.org/10.5281/zenodo.4742752.

[39] *evermore*. https://pypi.org/project/evermore/.

[40] *Gammapy. A Python package for gamma-ray astronomy*. https://gammapy.org/.

[41] Aram Hayrapetyan et al. "The CMS statistical analysis and combination tool: COMBINE". Submitted to *Comput. Softw. Big Sci.* 2024. arXiv: 2404.06614 [physics.data-an].

[42] *HistxFitter A software framework for statistical data analysis*. https://histfitter.github.io.

[43] Tomas Dado. *Combiner: 0.0.2*. Version 0.0.2. June 2024. DOI: 10.5281/zenodo.12007778. URL: https://doi.org/10.5281/zenodo.12007778.

[44] Lydia Brenner et al. "Comparison of unfolding methods using RooFitUnfold". In: *Int. J. Mod. Phys. A* 35.24 (2020), p. 2050145. DOI: 10.1142/S0217751X20501456. arXiv: 1910.14654 [physics.data-an].

[45] *HAverager at xFitter external meeting in Dubna*. https://indico.cern.ch/event/458944/contributions/1972242/.

[46] A. Glazov. "Averaging of DIS Cross Section Data". In: *AIP Conference Proceedings* 792.1 (Oct. 2005), pp. 237–240. ISSN: 0094-243X. DOI: 10.1063/1.2122026. eprint: https://pubs.aip.org/aip/acp/article-pdf/792/1/237/11400433/237_1_online.pdf. URL: https://doi.org/10.1063/1.2122026.

[47] *BLUE (Best Linear Unbiased Estimator)*. https://agiamman.web.cern.ch/blue/.

[48] *Background estimation with the ABCD method*. https://cms-opendata-workshop.github.io/workshop-lesson-abcd-method/01-introduction/index.html.

[49] *abcd-pyhf*. https://iris-hep.org/projects/abcd-pyhf.html.

[50] *UHI*. https://uhi.readthedocs.io/.

[51] *XLA (Accelerated Linear Algebra)*. https://openxla.org/xla.

[52] *The LLVM Compiler Infrastructure*. https://llvm.org.

[53] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs*. URL: http://github.com/google/jax.

[54] *JAX: Compatibility guarantees for custom calls*. https://jax.readthedocs.io/en/latest/export/export.html#compatibility-guarantees-for-custom-cal

[55] *TensorFlow)*. https://www.tensorflow.org.

[56] *dill*. https://dill.readthedocs.io/.

[57] *Luigi*. https://github.com/spotify/luigi.

[58] *Snakemake*. https://Snakemake.readthedocs.io/.

[59] F Mölder et al. "Sustainable data analysis with Snakemake". In: *F1000Research* 10.33 (2021). DOI: 10.12688/f1000research.29032.1.

[60] *Apache Airflow*. https://airflow.apache.org.

[61] Kyle Cranmer and Lukas Heinrich. "Yadage and Packtivity - analysis preservation using parametrized workflows". In: *J. Phys. Conf. Ser.* 898.10 (2017). Ed. by Richard Mount and Craig Tull, p. 102019. DOI: 10.1088/1742-6596/898/10/102019. arXiv: 1706.01878 [physics.data-an].

[62] Kyle Cranmer and Itay Yavin. "RECAST: Extending the Impact of Existing Analyses". In: *JHEP* 04 (2011), p. 038. DOI: 10.1007/JHEP04(2011)038. arXiv: 1010.2506 [hep-ex].

[63] Lukas Heinrich and Matthew Feickert. *recast-atlas: v0.4.2*. Version 0.4.2. DOI: 10.5281/zenodo.5854896. URL: https://github.com/recast-hep/recast-atlas/releases/tag/v0.4.2.

[64] *LAW: Luigi Analysis Workflow*. https://law.readthedocs.io/.

[65] *columnflow*. https://columnflow.readthedocs.io/.

[66] *PocketCoffea*. https://pocketcoffea.readthedocs.io/.

[67] *b2luigi*. https://b2luigi.readthedocs.io/.

[68] Michael Eliachevitch et al. *belle2/b2luigi: v1.0.2*. Version v1.0.2. July 2024. DOI: 10.5281/zenodo.12760796. URL: https://doi.org/10.5281/zenodo.12760796.

[69] *HTCondor Software Suite (HTCSS)*. https://htcondor.org.

[70] *IBM Spectrum LSF*. https://cloud.ibm.com/catalog/content/ibm-spectrum-lsf.

[71] *The Worldwide LHC Computing Grid (WLCG)*. https://home.cern/science/computing/grid.

[72] *Slurm Workload Manager*. https://slurm.schedmd.com/overview.html.

[73] *GFAL2*. https://dmc-docs.web.cern.ch/dmc-docs/gfal2/gfal2.html.

[74] *Docker*. https://docs.docker.com.

[75] *Introduction to Singularity*. https://docs.sylabs.io/guides/3.5/user-guide/introduction.html.

[76] *PEP 703 – Making the Global Interpreter Lock Optional in CPython*. https://peps.python.org/pep-0703/.

[77] *boost-histogram*. https://boost-histogram.readthedocs.io/.

[78] *Hist*. https://hist.readthedocs.io/.

[79] R. Brun and F. Rademakers. "ROOT: An object oriented data analysis framework". In: *Nucl. Instrum. Meth. A* 389 (1997). Ed. by M. Werlen and D. Perret-Gallix, pp. 81–86. DOI: 10.1016/S0168-9002(97)00048-X.

[80] ROOT Team. *ROOT Plans for 2024*. https://indico.cern.ch/event/1366207/contributions/5747214/. Jan. 2024.

[81] *HDF5 for Python*. https://docs.h5py.org/.

[82] *Zarr-Python*. https://zarr.readthedocs.io/.

[83] *Matplotlib: Visualization with Python*. https://matplotlib.org/.

[84] Andrzej Novak, Henry Schreiner, and Matthew Feickert. *mplhep*. Apr. 2020. DOI: 10.5281/zenodo.6807166. URL: https://github.com/scikit-hep/mplhep.

[85] Cyrille Praz and Tristan Fillinger. *plothist*. DOI: 10.5281/zenodo.10995667. URL: https://github.com/cyrraz/plothist/.

[86] L. Gray. *Fine-Grained HEP Analysis Task Graph Optimization with Coffea and Dask*. https://indico.jlab.org/event/459/contributions/11533/.

[87] Tommaso Tedeschi et al. *Prototyping a ROOT-based distributed analysis workflow for HL-LHC: The CMS use case*. https://doi.org/10.1016/j.cpc.2023.108965.

[88] Diogo Castro et al. *Apache Spark usage and deployment models for scientific computing*. https://www.epj-conferences.org/articles/epjconf/abs/2019/19/epjconf_chep2018_07020/epjconf_chep

[89] Beojan Stanislaus. *Next generation task scheduler for ATLAS software framework*. https://indico.cern.ch/event/1106990/contributions/4991224/.

[90] Paolo Calafiura et al. *Towards a distributed heterogeneous task scheduler for the ATLAS offline software framework*. https://www.epj-conferences.org/articles/epjconf/abs/2024/05/epjconf_chep2024_03041/epjconf_chep

[91] *Ray*. https://docs.ray.io/en/latest/ray-core/scheduling/index.html.

[92] *TaskVine*. https://cctools.readthedocs.io/en/latest/taskvine/.

[93] *The Swan Service*. https://swan.web.cern.ch/swan/.

[94] *INFN CMS Analysis Facility*. https://infn-cms-analysisfacility.readthedocs.io/.

[95] *SciPy Conferences*. https://conference.scipy.org/.

[96] *ErUM-Data-Hub*. https://erumdatahub.de/en/.

[97] *EuroSciPy Conferences*. https://euroscipy.org/.

[98] *AccelNet-Implementation: HSF-India - Research Software Networks in Physics*. https://research-software-collaborations.org/assets/pdf/hsf-india-2pager.pdf.