Simpler, faster and bigger: HEP analysis in the LHC Run 3 era

Enrico Guiraud
CERN, Meyrin, Switzerland
E-mail: enrico.guiraud@cern.ch

Abstract. The production, validation and revision of data analysis applications is an iterative process that occupies a large fraction of a researcher's time-to-publication. Providing interfaces that are simpler to use correctly and more performant out-of-the-box not only reduces the community's average time-to-insight but it also unlocks completely novel approaches that were previously impractically slow or complex. All of the above becomes especially true at the unprecedented integrated luminosity that will be achieved during LHC Run 3 and beyond, which further motivates the fast-paced evolution that has been taking place in the HEP analysis software ecosystem in recent years. This talk analyzes the trends and challenges that characterize this evolution. In particular we focus on the emerging pattern of strongly decoupling end-user analysis logic from low-level I/O and work scheduling by interposing high-level interfaces that gather semantic information on the particular analysis application. We show how this pattern brings benefits to analysis ergonomics and reproducibility, as well as opportunities for performance optimizations. We highlight potential issues in terms of extensibility and debugging experience, together with possible mitigations. Finally, we explore the consequences of this convergent evolution towards smart, HEP-aware "middle-man analysis software" in the context of future analysis facilities and data formats: both will have to support a bazaar of high-level solutions while optimizing for typical low-level data structures and access patterns. Our goal is to provide novel insights useful to boost the ever-ongoing, stimulating conversation that, since always, characterizes the HEP software community.

1. Introduction

The Large Hadron Collider (LHC) at CERN produces vast amounts of proton-proton and heavy ion collision data; thousands of physicists from scientific institutes from all over the world write data processing software applications to sift through this data in order to answer fundamental questions about the nature of particle interactions. In July 2022, LHC inaugurated its third data production period, Run 3, which will provide more than double the integrated luminosity of the previous run. In turn, that is roughly only one tenth of the integrated luminosity expected at the end of Run 4, or high luminosity LHC [1]. High-energy physics (HEP) is therefore facing a scenario in which analysis tools and applications will soon have to deal with unprecedented data volumes, with a corresponding increase in the complexity of auxiliary tasks such as data ingestion and storage, caching of intermediate artifacts, job scheduling, scale out and so forth.

Indeed, the HEP software ecosystem is renovating at a fast pace in order to guarantee that analysts can continue to work effectively in the LHC Run 3 era and beyond [2], taking advantage of the latest technological advancements: these include, to name a few, GPU acceleration of parts of the analysis pipeline that can express the appropriate level of parallelism; the exploration of

object store technology for scalable, high-throughput I/O (e.g. [3]); simplified job submission and scheduling with the help of state-of-the-art tools such as Dask (e.g. [4], [5]); simplified setup and maintenance of computing clusters via container orchestration (e.g. [6]).

In this context, analyses with particularly inclusive phase spaces (which involve the processing of billions of events already today) can operate as "canaries" for the kind of challenges we can expect to face [7], and realistic analysis benchmarks such as the Analysis Grand Challenge [8] provide generic test beds for the evaluation of novel technological improvements.

After highlighting the trends and challenges that characterize this heterogeneous, quickly evolving software environment, we will show how the introduction of modern "middleman analysis software" in the analysis pipeline can benefit ergonomics, reproducibility, performance and sustainability of HEP data processing applications; finally, we will investigate what consequences this pattern might have on the design of future analysis facilities and data formats.

2. The HEP analysis pipeline

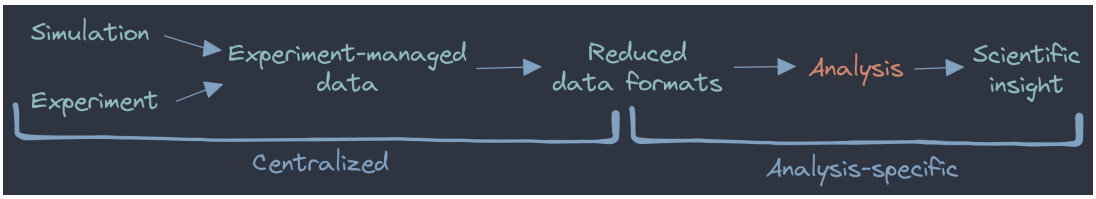

Figure 1. A schematic representation of a typical analysis pipeline. The first half usually corresponds to centralized data processing, while the second half features more freedom and variety of implementations on the part of the single analysis group or even a single analyst.

For the purposes of this work, we can represent a typical LHC experiment analysis pipeline as in Fig. 1. Raw data is produced by an experiment or by simulation software and, at first, it is processed and stored centrally, using experiment-wide tools. At this level, code bases are large and evolve slowly; borrowing the "cathedral and bazaar" terminology from [9], this is a situation in which it is natural to develop software as a cathedral: changes are planned carefully by code owners that are usually experienced programmers, in order to ensure that the stringent performance and feature requirements of the experiment are satisfied.

Later in the pipeline, data is transformed into the so-called "reduced data formats", which have a structure that is more immediately useful to analyzers; this can still happen centrally (as it is the case for CMS's NanoAOD format [10] or ATLAS's XAOD_PHYSLITE [11]) or it can be a pre-processing step performed by a specific analysis group. At this step, the original data is augmented with interesting derived quantities, calibrations are applied to physics objects and uninteresting events are filtered out in order to simplify and speed up further processing.

At this point the actual physics analysis work begins. This step features a large variety of (possibly quickly changing) requirements, a high level of innovation, and it is largely an iterative process in which physicists refine the code together with their understanding of the data, over time. It is made of a myriad of sub-tasks that range from metadata handling to the book-keeping of systematic variations, to machine learning training and inference. In the following we will mostly focus on data skimming, transformation and aggregation and assume that machine learning training as well as statistical modeling can be factored out and handled separately, as it is often the case. At the actual analysis step, then, the software development strategy cannot but resemble a bazaar: in contrast to the centralized development of the previous half of the pipeline, here it is common to see many actors involved, with several moving parts working together and code bases that change very quickly. It is important to highlight the role that

aggregators and standardizers such as the ROOT framework [12] and Scikit-HEP [13] play in such an ecosystem: they offer a platform to popularize best practices or commonly accepted solutions as well as robust implementations of difficult, core features such as I/O which do not require frequent reinvention. In order to avoid fragmentation, it is also crucial to develop common agreed-upon protocols to share analysis results and other artifacts; for example, the ROOT data format (ideally coupled with a simple event data model) is a language that most if not all HEP tools understand. Similarly, there are ongoing efforts towards standardizing the representation of statistical and machine learning models; maybe interestingly, in the case of complex N-dimensional histograms, instead, the quick growth of Python tools made it so that there is no single protocol that analysts can use to share these results across the ecosystem. This might be an open avenue for future useful developments.

2.1. Optimizing for the physicist's iteration time

Analysis, as defined by Fig. 1, occupies only a small fraction of the CPU time of HEP computing clusters. At the same time, however, it occupies the largest fraction of the working time of HEP physicists. That is what we want to optimize for: we want to reduce the time it takes for the analyst to iterate over ideas and produce the desired scientific insight.

In the ideal case in which the performance and ergonomics of analysis tools are improved to the point that this iteration time lowers by one or more orders of magnitude, completely new data explorations are unlocked that might have previously been prohibitively expensive. Nevertheless, even smaller performance or quality-of-life improvements can provide large benefits: a factor two speed-up in a physicist's time-to-plot might reduce the waiting time in job queues by a similar factor (or, conversely, allow to serve two times more users with the same computing power); a simpler job submission system with smarter scheduling could similarly provide better resource utilization. It is natural, then, to ask what is required of HEP analysis software in order to provide such improvements.

2.2. Pillars of high quality software

Much like for other software, there are three main ingredients to optimizing the physicist's iteration time across the full lifetime of an analysis:

- Ergonomics: this includes onboarding, clear documentation, simplicity of debugging correctness as well as performance bottlenecks, extensibility, prototyping; in general, making simple things simple and complex things possible
- Performance: providing the best possible throughput and hardware utilization with the least possible effort on the part of the user
- Sustainability: result validation, stability across releases, user support and prompt bug fixes when required, over the lifetime of the experiment (years if not decades)

In our experience, when it comes to end users having to pick which tool to use, ergonomics are often the dominating factor. Service managers and experiment coordination typically value performance and sustainability as well. Note that the choice of programming language impacts all three aspects: it is certainly not a coincidence that the current trend is to employ Python interfaces (for ergonomics) with C or C++ implementations (for performance), and there is a high cost to switch to other languages or introduce more languages into the mix in terms of long-term sustainability.

3. Modern middleman analysis software

Middleman analysis software is made of those libraries and tools that sit between the analyst on one side and the computing resources and data on the other. As depicted in Fig. 2, the user

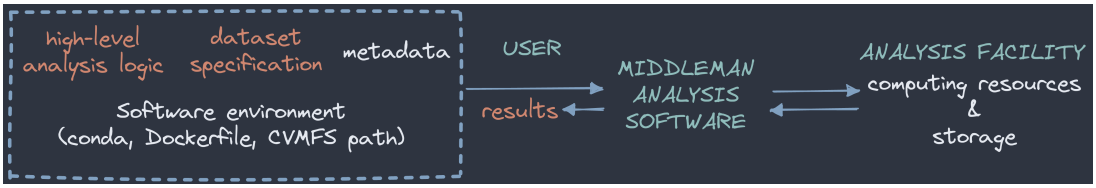

Figure 2. A diagram representing the relations between user, middleman analysis software and analysis facility: the user provides a high-level description of the analysis logic and other relevant information to the middleman; the middleman converts it to appropriate, concrete logic adapted to the specific hardware and storage configuration of the analysis facility (or laptop/workstation/server); results (e.g. histograms) are finally propagated back to the user.

provides the middleman with high-level information about the task they want to accomplish (what kind of operations should be performed on what data); the middleman then "digests" that input, integrates it with information about the software environment as well as available computing resources and data-delivery services, and it comes up with an execution plan that provides the desired results to the user in the most efficient way possible. The crucial element here is the *complete decoupling of analysis logic from low-level execution details*.

This is of course not a completely novel concept: in HEP, classic interfaces such as `TTree::Draw` or tools such as PROOF have been playing this role; outside of HEP, declarative interfaces and query languages that completely decouple user logic (the "what") from execution strategy (the "how") are widespread. However, the middleman analysis software pattern is now more relevant than ever: firstly, current technologies (Python, Dask or Intel TBB schedulers, reproducible software environments, containerization) supercharge what we can do with it, as we hope to illustrate here; secondly, we need middlemen more than ever as analysis pipelines become more complicated, hardware possibly more so (e.g. due to the presence of accelerators such as GPUs or non-trivial many-core CPU architectures with NUMA effects) and performance becomes more critical.

It is important to point out that modern middleman software has interest in collecting as much *semantic* information as possible regarding the workload it will execute: for example which parts of the analysis correspond to the nominal case and which to systematic variations, which derived quantities are the result of machine learning inference rather than other types of calculations, which quantities only depend on metadata that is fixed across the processing of a certain sample and which ones instead depend on quickly-varying inputs. It is indeed challenging to design interfaces that are easy to use while being expressive enough to convey this kind of information; however, we contend that tackling this challenge is crucial to allow the middleman layer to transparently perform the kind of optimizations discussed in this section.

As an example, in Listing 1 ServiceX [14] and FuncADL [15] collect information on what object selection the user wants to perform on what dataset, and automatically find the right files via a lookup to the appropriate ATLAS catalog, spin up a container with the software environment required to actually read the data and run the query, and finally return the selected jet momenta as a Python array, caching the query result on local storage to speed up future iterations.

Listing 2 reports another example in which RDataFrame [16] collects information on systematics via the `Vary` method and automatically propagates the variations through the event selection (the `Filter` call), the calculation of derived quantities and the filling of histograms.

Other examples of modern middleman HEP analysis software include Coffea [5], Bamboo [17], CutLang [18] and O2 [19].

```
dataset_xaod = "mc15_13TeV:mc15_13TeV.361106.PowhegPyhia8EvtGen"
ds = ServiceXSourceXAOD(dataset_xaod)
data = (
 ds
  .SelectMany('lambda e: (e.Jets("AntiKt4EMTopoJets"))')
  .Where('lambda j: (j.pt()/1000)>30')
  .Select('lambda j: j.pt()')
  .AsAwkwardArray(['JetPt'])
  .value()
)
```

Listing 1: A FuncADL/ServiceX query that, for each event in the input dataset, extracts the momentum of jets that have momentum greater than 30 GeV.

```
ROOT.EnableImplicitMT() # enable multi-threading
h_nominal = (
    RDataFrame('Events', 'root://eos.server/data/*.root')
    .Vary('Muon_pt', 'RVec<RVecF>{0.9*Muon_pt, 1.1*Muon_pt}', ['down', 'up'])
    .Filter('nMuon == 2 && Muon_charge[0] != Muon_charge[1]')
    .Define('mass', 'InvariantMass(Muon_pt, Muon_eta, Muon_phi, Muon_mass)')
    .Histo1D('mass')
)
# dictionary with keys 'nominal', 'Muon_pt:down', 'Muon_pt:up'
h_dict = ROOT.RDF.VariationsFor(h_nominal)
```

Listing 2: In this snippet the user informs RDataFrame that the `Muon_pt` column has two varied values attached to it for the "down" and "up" variations. RDataFrame then makes use of this information to produce three histograms instead of one from the same multi-thread event loop, only recomputing what is required for the different variations.

3.1. What the middleman can do for the user

Referring back to the three pillars of high-quality software mentioned above, the advantages of a middleman layer in terms of ergonomics should be immediately obvious: users can interface with APIs that "speak their language" and easily allow performing common HEP analysis tasks such as event selection, object calibrations, definition of derived quantities and finally produce hundreds of histograms with weighted events. At the same time, users do not have to take care of implementation details that are not strictly related to the analysis logic such as parallelization, I/O, error recovery, job scheduling, result merging, dataset splitting and so forth; the middleman is in charge of these details, which makes it simpler to obtain good performance out of the box, transparently adapting the execution plan to the available hardware resources. In terms of sustainability, analysis code that is not entangled with low-level implementation details (and therefore is more decoupled from the particularities of the computing resources it will run on) is more maintainable in the long term, less sensible to changes in the underlying libraries, and benefits from enhanced reproducibility.

This separation of high-level and low-level concerns also greatly simplifies the introduction of other improvements such as transparent caching of pre-processed data into object stores [20], GPU offloading (e.g. for machine learning inference), transparent caching of ML inference results and automation of analysis preservation, among others.

For all the advantages the middleman analysis software brings to the table, this layer must

be carefully designed not to degrade the user's debugging experience (where "debugging" here is intended as the investigation of both logical and performance issues within the application). The challenge is to let physicists debug their analysis logic without having to "look behind the curtain", that is having to understand and step through the logic of the middleman layer. We propose two ways in which this issue can be mitigated. Firstly, given that, fundamentally, user logic performs transformations on event data, analysis tools should make it possible for analysts to step through those transformations for the desired events without having to understand how the event loop is internally handled. Using native debugging tools for this purpose is especially challenging when the middleman layer runs an event-by-event loop or when the event loop is dispatched to a sub-process, suggesting that dedicated interfaces might have to be developed for this purpose. Secondly, error and performance reporting need to be explicitly designed with end users in mind (as opposed to framework developers); in fact, two levels of reporting (one for end users and one for developers of the middleman layer itself) are desirable. Because of the many abstraction layers between user logic and what actually gets executed (which often mix many complex frameworks if not different programming languages), again here mechanisms built into programming languages and frameworks are usually not up to the task, or at least not without careful utilization: top-notch, clear error and performance reports in middleman software can only be an effect of an explicit design decision.

4. Recurring implementation details in HEP analysis tools

4.1. Event-wise vs bulk-wise logic

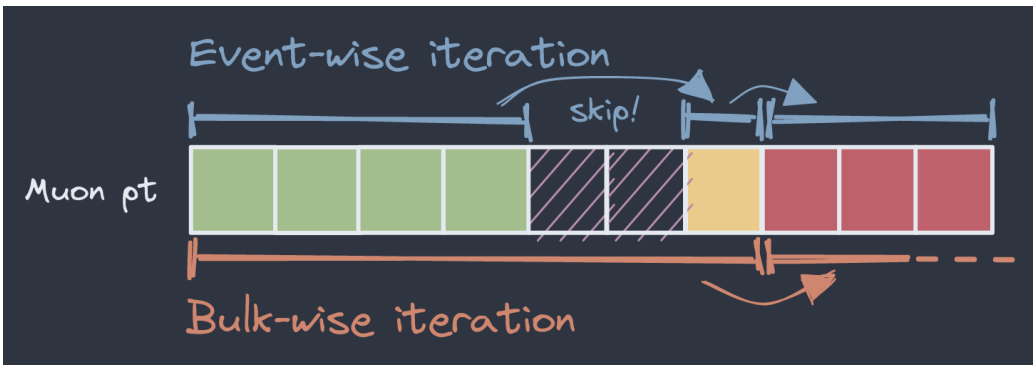

Figure 3. A schematic comparison of event-wise (above) and bulk-wise (below) data processing.

Developers of modern HEP analysis tools face the choice of offering event-wise or bulk-wise interfaces to their users. Event-wise data processing loads and operates on one event at a time: with rare exceptions, due to collision event independence, the physics ideas behind data transformations (e.g. object selections) are most often event-wise, so this paradigm is a natural fit to express analysis logic. In contrast, the bulk-wise approach expresses and performs operations on bulks (or chunks) of many events at a time via array programming: this is common in pure Python interfaces in order to amortize the runtime cost of computational overheads (e.g. Python expression evaluation) across the bulk; in general it is certainly desirable to reduce the amount of CPU instructions that are executed per event. However, for complex operations, writing array-oriented logic can prove cumbersome for users [21]. It is also worth noting that, contrary to widespread belief, bulk-wise processing does not guarantee CPU vectorization of the data processing operations because of the frequent presence of event or object masks which introduce branching at the level of CPU instruction execution. See Fig. 3 for a visual comparison of the two paradigms.

Fortunately, the separation of user-facing logic and actual execution that the middleman paradigm provides can go a long way in resolving this dilemma, by separating which approach is used by user-facing APIs as opposed to the actual data crunching. Raw disk I/O as well as large parts of ROOT I/O are already bulk-wise; much of the internal middleman software logic can be implemented in a bulk-wise fashion; at the same time, users can express their custom logic event-wise, either in C++ or as Python functions that are compiled to fast native code by Numba [22]. Exposing bulks of events to users is then left as an expert feature that can be useful to implement efficient kernels for specialized tasks such as machine learning inference.

4.2. Efficient object collections

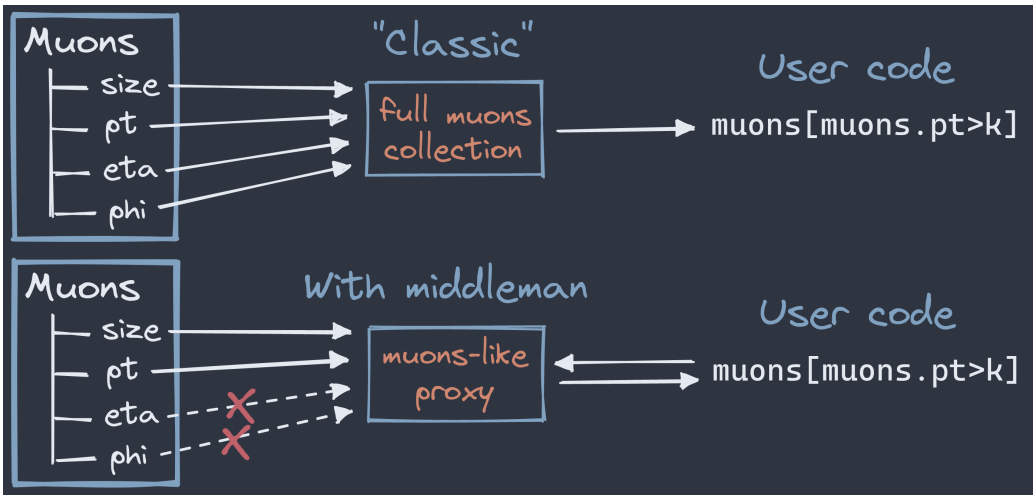

Figure 4. The amount of data actually read from storage can be reduced by introducing proxy objects that mimic the interface of a collection of physical objects (e.g. an array of muon objects) but do not require ahead-of-time reads of all the muon properties. Rather, these proxy objects can record which properties are accessed and only request those from the I/O layer.

When it comes to event data models, the need to optimize I/O performance and reduce storage usage (which implies simple, streamlined event data models that do not involve complex deserialization steps) is often at odds with the desire to expose full physics objects to end users (e.g. a collection of electrons rather than many separate collections representing different electron properties) for increased ergonomics.

This design tension can be resolved through the middleman pattern by exposing to users proxy objects that act like full physics objects but do not actually require full deserialization of the objects: the middleman layer than translates the operations performed on these proxy objects into lower-level operations on a simpler schema. This way the I/O layer does not need to deserialize full electron or muon objects, but physicists can still reason in these terms.

For example, Bamboo [17] and Coffea [5] both add a user-level representation of full physics objects while, under the hood, performing operations on the "flat" NanoAOD schema.

5. Building analysis facilities for the bazaar

Analysis facilities (AFs) are a dedicated platform to perform typical HEP analysis tasks. They are the HEP analysis analogous of a makerspace, or a gym: a place (digital, in this case) that offers specialized hardware and support for a specific kind of task (HEP analysis in this case).

To an extent, existing computing clusters such as LXBATCH are AFs, but the user experience can be greatly improved by leveraging modern tools and the middleman pattern.

Common building blocks of modern AFs are (see also [2]):

- containerization (via Docker/Singularity, with Kubernetes for orchestration)
- Dask as a scheduler, often in tandem with HTCondor/SLURM
- JupyterLab as front-end (SSH access also allowed)
- high-bandwidth connection to data storage

Some examples of typical analysis sub-tasks that AFs can simplify are: user authentication to experiment-specific services; data access authorization; smart scheduling to guarantee better usage of all levels of data caches; job monitoring, both to give feedback to users when e.g. their data throughput is worryingly low and to give feedback to analysis tool developers (e.g. about high average latencies for particular hardware configurations).

We propose two main principles for AF developers to follow in order to build platforms that are well integrated in the bazaar-like HEP analysis software ecosystem. Firstly, AFs should optimize for common patterns and behaviors typical of HEP analyses rather than specialize for a given software stack: patterns change slowly, software stacks evolve at a fast pace. One example of such a pattern is the typical distinction between quick exploration (characterized by a need for low latency, ideally interactivity and small datasets that can benefit from small, fast data caches) and full analysis jobs (high throughput on very large datasets where higher latency is tolerated if not expected). In the case of quick interactive explorations, an analysis facility could even let users partially share some of the computing cores, as a user might only require them at full capacity during a fraction of the lease time. For the full analysis job, smart scheduling that maximizes data reuse could provide beneficial; analysis facilities in use by ALICE have this idea built into them and run analysis jobs in "trains" that "stop at" the same datasets all together [23]: with enough user pressure, caches are always cold, but smart scheduling can provide long enough cache coherence time to improve overall throughput.

The second principle is to build monitoring in such a way that users and admins can compare solutions. Current computing centers run a large variety of jobs and it is often hard for administrators to isolate outliers or problematic patterns in I/O or core utilization. By providing monitoring specialized for typical analysis workloads on platforms that exclusively run such workloads, administrators and developers can obtain invaluable insights.

6. Conclusions

The HEP analysis software ecosystem is healthy and evolving at a fast pace with a good mix of R&D and production-grade developments. The LHC Run 3 era will benefit from a new generation of analysis tools that focus on gathering semantic information about the analysis (input datasets, software environment, custom data transformations, ...) and HEP-specific concepts (systematics, physics objects, ...) in order to a) speak the physicist's language and hide low-level technical details and b) automatically execute what users need on the available hardware in the most efficient way possible.

Current challenges include but are not limited to: letting users debug logical and performance issues in the "tip of the iceberg" without having to understand the full software stack; smart use of data caches, a necessity to meet desired performance goals that requires collaboration between analysis code, schedulers and analysis facilities; simplifying heterogeneous computing by transparently offloading appropriate computations to accelerators such as GPUs.

We are curious to see how further developments in the HEP analysis software ecosystem will tackle such challenges and hope to have provided insights useful to direct such efforts.

References

[1] Apollinari G, Béjar Alonso I, Brüning O, Fessia P, Lamont M, Rossi L and Tavian L 2017 URL https://www.osti.gov/biblio/1767028

[2] Stewart G A, Elmer P, Eulisse G, Gouskos L, Hageboeck S, Hall A R, Heinrich L, Held A, Jouvin M, Khoo T J, Laycock P, Mato Vila P, Pivarski J, Rembser J, Rodrigues E, Schaarschmidt J, Sexton-Kennedy E, Shadura O, Simpson N, Skidmore N, Sokoloff M, Watts G, Mohamed A, Burzynski J, Cardwell B, Craik D C, Dado T, Delgado Peris A, Doglioni C, Eren E, Eriksen M B, Eschle J, Fitzpatrick C, Flix Molina J, Gasiorowski S, Goel A, Guiraud E, Gupta K, Hernández Villanueva M, Hernández J M, Hrivnac J, Lieret K, Kreczko L, Krumnack N, Kuhr T, Kundu B, Lancon E, Lange J, Manganelli N J, Novak A, Perez-Calero Yzquierdo A, Proffitt M, Rybkin G, Schreiner H F, Schulz M, Sciabà A, Sekmen S, van Daalen T, Simko T, Singh J, Smith N, Strong G C, Unel G, Vassilev V, Waterlaat M, Yazgan E, Das A and Galewsky B 2022 Hsf iris-hep second analysis ecosystem workshop report URL https://zenodo.org/record/7418818

[3] López-Gómez, Javier and Blomer, Jakob 2021 *EPJ Web Conf.* **251** 02066 URL https://doi.org/10.1051/epjconf/202125102066

[4] Padulano V E, Kabadzhov I D, Tejedor Saavedra E, Guiraud E and Alonso-Jordá P 2023 *Journal of Grid Computing* **21** 9 ISSN 1572-9184 URL https://doi.org/10.1007/s10723-023-09645-2

[5] Gray L, Smith N and on behalf of the CMS Collaboration 2023 *Journal of Physics: Conference Series* **2438** 012033 URL https://dx.doi.org/10.1088/1742-6596/2438/1/012033

[6] Adamec M, Attebury G, Bloom K, Bockelman B, Lundstedt C, Shadura O and Thiltges J 2021 Coffea-casa: an analysis facility prototype *EPJ Web of Conferences* vol 251 (EDP Sciences) p 02061

[7] Bendavid J 2023 *Journal of Physics: Conference Series* **2438** 012002 URL https://dx.doi.org/10.1088/1742-6596/2438/1/012002

[8] Held A and Shadura O 2022 *PoS* **ICHEP2022** 235

[9] Raymond E 1999 *Knowledge, Technology & Policy* **12** 23–49

[10] Rizzi A, Petrucciani G and Peruzzi M 2019 A further reduction in cms event data for analysis: the nanoaod format *EPJ Web of Conferences* vol 214 (EDP Sciences) p 06021

[11] Elmsheuser J, Anastopoulos C, Boyd J, Catmore J, Gray H, Krasznahorkay A, McFayden J, Meyer C J, Sfyrla A, Suruliz K *et al.* 2020 Evolution of the atlas analysis model for run-3 and prospects for hl-lhc *EPJ Web of Conferences* vol 245 (EDP Sciences) p 06014

[12] Brun R and Rademakers F 1997 *Nuclear instruments and methods in physics research section A: accelerators, spectrometers, detectors and associated equipment* **389** 81–86

[13] Rodrigues E *et al.* 2020 *EPJ Web Conf.* **245** 06028 (*Preprint* 2007.03577)

[14] Galewsky, B, Gardner, R, Gray, L, Neubauer, M, Pivarski, J, Proffitt, M, Vukotic, I, Watts, G and Weinberg, M 2020 *EPJ Web Conf.* **245** 04043 URL https://doi.org/10.1051/epjconf/202024504043

[15] Proffitt, Mason and Watts, Gordon 2021 *EPJ Web Conf.* **251** 03068 URL https://doi.org/10.1051/epjconf/202125103068

[16] Piparo D, Canal P, Guiraud E, Valls Pla X, Ganis G, Amadio G, Naumann A and Tejedor Saavedra E 2019 *EPJ Web Conf.* **214** 06029

[17] David P 2021 *EPJ Web Conf.* **251** 03052 (*Preprint* 2103.01889)

[18] Ünel G and Sekmen S 2018 *Computer Physics Communications* **233** 215–236 ISSN 0010-4655 URL https://www.sciencedirect.com/science/article/pii/S0010465518302315

[19] Eulisse G, Konopka P, Krzewicki M, Richter M, Rohr D and Wenzel S 2019 Evolution of the alice software framework for run 3 *EPJ Web of Conferences* vol 214 (EDP Sciences) p 05010

[20] Padulano V E, Tejedor Saavedra E, Alonso-Jordá P, López Gómez J and Blomer J 2022 *Cluster Computing* 1–16

[21] Smith, Nicholas, Gray, Lindsey, Cremonesi, Matteo, Jayatilaka, Bo, Gutsche, Oliver, Hall, Allison, Pedro, Kevin, Acosta, Maria, Melo, Andrew, Belforte, Stefano and Pivarski, Jim 2020 *EPJ Web Conf.* **245** 06012 URL https://doi.org/10.1051/epjconf/202024506012

[22] Lam S K, Pitrou A and Seibert S 2015 Numba: A llvm-based python jit compiler *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* pp 1–6

[23] Schwarz K, Fleischer S, Grosso R, Knedlik J, Kollegger T and Kramp P 2019 The alice analysis facility prototype at gsi *EPJ Web of Conferences* vol 214 (EDP Sciences) p 08027

[24] Pivarski J, Osborne I, Ifrim I, Schreiner H, Hollands A, Biswas A, Das P, Roy Choudhury S, Smith N and Goyal M 2018 Awkward Array

[25] Blomer J, Canal P, Naumann A and Piparo D 2020 Evolution of the root tree i/o *EPJ Web of Conferences* vol 245 (EDP Sciences) p 02030

[26] Lopez-Gomez J and Blomer J 2023 Rntuple performance: status and outlook *Journal of Physics: Conference Series* vol 2438 (IOP Publishing) p 012118

[27] Graur D, Müller I, Proffitt M, Fourny G, Watts G T and Alonso G 2023 *Journal of Physics: Conference*

Series **2438** 012034 URL https://dx.doi.org/10.1088/1742-6596/2438/1/012034
[28] Köster J and Rahmann S 2012 *Bioinformatics* **28** 2520–2522
[29] Rieger M 2020 Design pattern for analysis automation on distributed resources using luigi analysis workflows *EPJ Web of Conferences* vol 245 (EDP Sciences) p 05025

Appendix

Appendix A. Performance targets for analysis applications

Given 1 PB of compressed data, of which 100 TB are actually read by the analysis, we expect most analyses based on modern tools to be able to run in a) 10 minutes on a cluster of 64 nodes, or b) 4 hours on a single many-core machine with 128 cores. The throughput required is respectively a) ∼3 GB/s/node or b) ∼100 MB/s/core, for I/O plus processing.

Analysis facilities need a hardware setup that can sustain such throughput. At the same time, applications cannot afford to read more bytes than what's strictly needed and must make good use of the available hierarchy of storage options: remote storage, a large shared cache at the level of the computing facility (e.g. xcache or high-bandwidth object stores) and small user-level caches.

In order to complement these abstract calculations, here are some examples of what is possible already today:

- "turnaround of a few hours for [...] thousands of histograms of the CMS Run 2 data on a batch system" [17]
- 3.2B events, O(1000) systematics, 70 5-dim histograms in 45 minutes (SSD storage, 128 threads) [7]

RNTuple and other advancements should provide another factor N speed-up in the medium term, with $1 < N < 10$.

Appendix B. Making our own tools still makes sense

One could ask whether, today, there is still a need to develop HEP-specific analysis tools rather than adopting products that are successful in industry. This is an excellent question, because trying to answer it is a good way to figure out what problems we are facing actually need dedicated solutions and which do not.

Specializing for our use case should not mean reinventing the wheel but weirder. In fact, a first common reason to develop HEP-specific software is to adapt existing tools and concepts to specific HEP needs. Relevant examples are the need to efficiently manipulating large datasets with a hierarchical data model (events contain objects such as reconstructed electrons, which in turn are collections of properties such as the momentum of each electron) and nested, jagged collections (e.g. every event might contain a variable number of electrons); the book-keeping and runtime implications of introducing systematic variations; or the prevalence of histograms (with attached error bars) as the most common data aggregation and as a common starting point for statistical fits. Awkward arrays [24] are a good example of a library that solves a HEP-specific need (handling jagged arrays efficiently) adapting tools that work well in industry (NumPy arrays and array programming).

Another valid motivation behind the implementation of ad-hoc tools is performance optimization by leveraging HEP-specific features. For example, RNTuple [25] (ROOT's proposed TTree successor as the de-facto standard HEP data format) shows superior I/O performance with respect to industry solutions for typical HEP schemas [26]. Similarly, a comparison of the HEP-specific analysis interface RDataFrame with state-of-the-art cloud query systems shows that RDataFrame can yield better performance as well as more concise queries [27].

Appendix C. Two levels of computation graphs

Middleman software requires a structured representation of data transformations and their dependencies in order to reason about what the user requested and deliver it as efficiently as possible. It is natural then that, in many implementations of the middleman concept, user logic is stored as a computation graph: this representation can then be leveraged for workflow optimization, caching of intermediate steps, potentially even auto-differentiation. Maybe interestingly, in the case of HEP analyses, *two* separate levels of computation graphs are usually employed: one is concerned with actual data transformations (data in input, results as output, and what needs to be done with the input to get to the results; this is the level RDataFrame is concerned with, for example); the other computation graph deals with the more general analysis workflow (fetching input datasets from remote sites, retrieving information from condition databases, scheduling auxiliary tasks as required such as machine learning training; Snakemake [28] or law [29] are examples of tools that can be used to build computation graphs for this level of abstractions).