# Future Trends in Nuclear Physics Computing 2020

## Workshop Summary

Alexander Kiselev (BNL), Amber Boehnlein (Jefferson Lab), Graham Heyes (Jefferson Lab), Mark Ito (Jefferson Lab), Markus Diefenthaler (Jefferson Lab), Ofer Rind (BNL), Paul Laycock (BNL), Torre Wenaus (BNL)

# Common Scientific Software

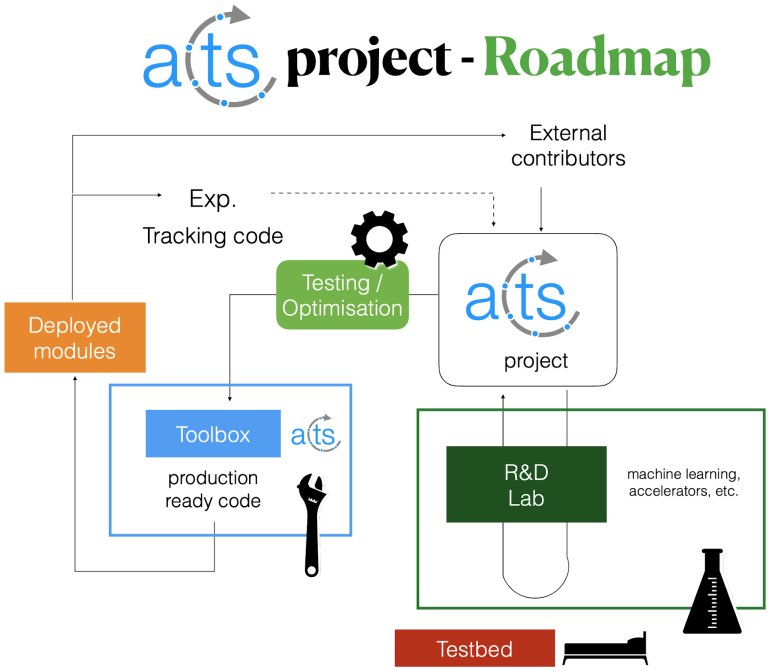
The opening session of the workshop addressed common scientific software, exploring how successful common software projects have taken shape and have led to strong common efforts in the community. We had talks from leaders of two prominent common software projects that have emerged in recent years, from different domains of HEP software, on their experiences and findings: Andreas Salzburger (CERN), an originator of the ACTS tracking software project, and Mario Lassnig (CERN), an originator of the Rucio scientific data management software project. Complementing those talks, Daniel S. Katz (NCSA/U Illinois), a longtime leader in cultivating and developing common research software, spoke about building enduring, sustainable common research software, an essential element of which is providing proper credit, recognition and quality career paths for the developers vital to building and sustaining our software.

## Developing common software: ACTS

In his ACTS talk, Andreas recalled some track reconstruction history: the deep history of the LEP era when many still-current algorithms were developed, most often with a FORTRAN code base, followed in the early 2000s by the era of C++ object oriented (or over-object oriented) design, yielding codes (OO C++ but showing FORTRAN roots where performance most mattered) that became the starting point for developing the first production LHC codes. The first generation LHC codes showed the value of trial by fire: the code became extremely well ironed-out through processing and validating many tens of billions of events. Algorithms were tuned to very high performance, approaching 100% technical track reconstruction efficiency.

This first generation LHC software constituted a great starting point towards solving the challenges of the HL-LHC with much greater particle multiplicities (and thus combinatorics), but the software was showing its scars in departed developers and dated technology, leading to the creation of the ACTS project. ACTS began with community driven common tracking software as the objective, and a clear mission statement: preserve and advance the state of the art (reviewing existing code and taking what's good); develop and deploy production ready software for HL-LHC and beyond; establish R&D testbeds for new technologies (algorithms, machine learning (ML), GPU, detectors); and work & educate in state of the art technology/workflows.

The project sought interest and buy-in from others from the beginning. There were discussions with others at Okinawa CHEP (2015), and the abstraction layer started with ATLAS in mind was seen as a starting point to see if others could use it.




Modernize, simplify, and streamline were watchwords. A technical paradigm shift: flatter data structures (for data store independence, simpler memory layout, CPU and GPU performance), fixed size matrix multiplications for high performance, polymorphism moved to compile time. Code sizes were shrunk dramatically, C++ FORTRAN was replaced with performance-engineered C++ using modern libraries. Building a production ready toolbox followed the CERN recommendation to use an outside provider (not CERN GitLab) for common projects; github was chosen with a carefully structured repository organizing core, extension, test, plugin and example codes. Dependencies were kept to the essential minimum, coding standards were defined, an open source license (Mozilla) was chosen early. A plugin architecture supported the experiment agnostic design. Parallelism was designed in. The guideline for experiment integration was “keep it simple”, but not overly simple: practicalities like interfacing to experiment framework log messaging need to work. A general guideline was “fire brigade design” of function calls: you call us, we never call you.

ACTS aims to provide a research testbed for algorithm R&D, with testbed detectors and algorithm examples. The standalone repository eases R&D, simplifying extensions and tests, with testbeds allowing for rapid feedback on development. The R&D testbed has been used to create strong community links in itself, for example a fast simulation extension used for kaggle and codalab ML tracking challenges that attracted great community interest and innovation, and established a reference data set still in use for much tracking R&D. ML is a prominent R&D direction, e.g. track seeding. ACTS has a dedicated community subgroup for the important R&D

topic of parallelization for accelerators, evolving the code to run on different heterogeneous architectures, producing valuable lessons and feedback to the core project and its data model and algorithm design. The aim is that successful R&D finds its way to the production code base, eventually. R&D extends also to learning how to integrate external non-HEP software.

The ACTS project recognizes the necessity of staying attractive to the best technical students: deploying modern development workflows, being open to new technologies and standards, and providing pathways to publication. Elements of the ACTS workflow ecosystem include GitHub based Travis continuous integration (CI), docker, codecov, clang code cleaning tools, a human review/merge process, readthedocs, doxygen and Mattermost. A high level of unit testing, CI coverage and code review are given emphasis, with the direct benefit of delivering highly tested code to experiments and curious developers looking at the project. Modern, rigorous development workflows and practices also contribute to building valuable experience within the developer team, producing developers valuable not only for tracking software expertise but modern coding practices as well. In ATLAS for example, the migration of ATLAS software to git/CI was pioneered by ACTS developers. Importantly, the modern technology/workflow expertise accrued by developers facilitates careers outside our field as well.

The ACTS project's objective of developing community driven common tracking software is showing impressive success. ATLAS has adopted it for partial use in Run-3 and full migration of tracking to ACTS for Run-4 (HL-LHC). There are currently 42 acts-project forks from individuals and experiments. sPHENIX recently made ACTS its default tracking software. EIC is studying it with interest. The 10-15 active developers on the project include sPHENIX, Belle II and other non-ATLAS contributors. Version v01.00.00 was released in Sep 2020, after 4 years of maturation. Encapsulating the tracking as standalone software independent of the experiment software stack came with a price tag for ATLAS, but it is paying off in its benefits for the community, including ATLAS: ACTS has enabled R&D in many areas from which ATLAS as well as others benefit.

A lesson from experience ("if you could do one thing differently…") has been the importance of defining clear objectives and using them to plan and drive the development. In ATLAS, ambiguity over whether the project was aimed at Run-3 (2021) or Run-4 (2027) complicated the development objectives and the expectations from the experiment; clarifying that only limited objectives are targeted for Run-3, with the project primarily directed at Run-4, was important to establishing a clear roadmap for the near and long term.

The ACTS project sees itself as one project in a new ecosystem of community driven software, under the umbrella of the HSF, working and learning together with other community software projects. An example is ACTS transmitting and sharing Eigen experience with other projects via HSF. Projects should be designed and developed to play together, with encouragement to put modules together and build systems. And finally, we should work together!

## Developing common software: Rucio

In his talk on Rucio, Mario set out to answer the question we would all like to know the answer to: “What’s the secret sauce?”. Rucio, like ACTS, is something of a poster child for common software, starting life on ATLAS and gradually being adopted by a growing number of experiments both in and outside of the HEP community. While Rucio’s precursor, Don Quixote 2 (DQ2) was designed specifically for ATLAS, Rucio as its replacement was foreseen from the beginning to be a more generic system, albeit with a laser focus on the requirements of ATLAS, able to scale ultimately to HL-LHC, and capable one day of use beyond ATLAS. Based on painful Run-1 operations experience with DQ2 and the realization that adapting that solution simply wouldn’t work, the obvious conclusion was that a completely new design was needed. An excited group of software engineers and physicists set out determined that this time they would do it right. A crucial factor in being able to do it right from the beginning as open source software was support from both the experiment and the developer’s line management. For management, it was part of the mission from the start to look beyond ATLAS.

Rucio’s precursor, DQ2, had both technical and operational limitations. It had been designed at a time when the MONARC model dominated thinking about data management. In the MONARC model, data placement is essentially static, meaning data management software has relatively modest requirements. However, during the early years of the LHC it became apparent that network bandwidth far exceeded MONARC predictions and the experiments wanted to capitalize on this and use a much more dynamic data placement model, shipping data to where it was needed and increasing the number of copies of popular datasets. The ATLAS data management team learned the hard way exactly what DQ2 could not do, and this operations experience fed into the design of Rucio.

At its heart, Rucio provides a horizontally scalable catalog for files, datasets and collections, together with their metadata. It is not a distributed file system, it provides seamless federation of academic and commercial storage, together with their networks, at distributed locations and facilitates transfers of data between those locations. Monitoring, data analytics and many other advanced features can all be selectively enabled. Instead of the traditional, imperative approach to data management, “copy file X from location A to location B”, Rucio uses an expressive, declarative approach which simplifies operations “I need two copies of file X”. This concept proved to be the key to solving the scaling issues seen in DQ2.

The ingredients of Rucio’s secret sauce started with an open-source mindset, and then Mario added a few other software engineering best practices. Code versioning used git from the start, with mandatory unit tests, continuous integration and code reviews together with a validated master release process. Early in the life of Rucio the team were still engaged in active development on DQ2, allowing them to set a realistic timeline for Rucio (3-4 years) which also gave them the space to explore new ideas and paradigms from outside of the HEP community. All the while there were open discussions with the data management teams of the other LHC experiments, and integration of interested people into the development team.

Despite these advantages, Mario was candid about the first steps with Rucio, not everything worked as planned, some beloved features of DQ2 were missing, but nevertheless the core platform was ready. Over the next 4 years, and with a fraction of the effort required in the initial design period, continuous improvements were worked on, made easier by Rucio's modular design. The Rucio team remains a very open group, strong in communication, welcoming R&D from PhD students and actively engaging industry as well as the scientific community. An important adaptation in light of experience that the project made early on was to integrate development and operations in the same group: no dev/ops split in responsibilities of team members. This ensured that developers were directly aware of real operational issues. The developers get an intrinsic feeling of the system if they're using it themselves, and they can reengineer it in an informed way. This is particularly important for data management software, sitting in front of very expensive storage facilities and important data. A product and team that gives operations central importance builds confidence.

The growing number of experiments adopting Rucio is testament to the open-mindedness and communication skills of the team, tied with a growing awareness that, perhaps, a common data management solution might be a good idea. In research where we already share infrastructure, particularly network and often storage at data centers, having a common solution to tackle shared challenges makes sense. Given the growing scale of scientific data storage needs and its requisite cost, funding agencies are keen to see the kind of detailed accounting plots that Rucio can provide. As time goes by this will surely become an expectation. The potential for improving throughput by better use of networks and maybe even storage across multiple experiments, even up to orchestrating data flows across experiments, is also something that funding agencies are likely to be interested in.

Meanwhile software development for Rucio, now supporting multiple experiments, has successfully moved to community-driven development. Guided by the core team and discussed in public meetings, it is members of the community that take responsibility for implementing and testing a desired feature. As the community members grow in experience, they in turn help newcomers who also benefit from more automation and containerisation of the software. Experience on Belle II attests to the fact that newcomers feel very well supported.

While Rucio benefited from the legacy of DQ2, and indeed the operations experience was key to Rucio's success, it also suffers a little from that history. Although the underlying design of Rucio broke away from DQ2, the APIs were left mostly intact to make life easy for client software. This has led to some overly complex APIs lingering around which, like any software that gets used, are difficult to clean up. Nevertheless, with good documentation and a very active Slack community, such issues tend to be bumps rather than blockers. Rucio was designed after all to get a very particular job done and done well. It just so happens that the team behind it are an open-minded, communicative bunch with a penchant for generic design and a desire to make life in operations as simple as possible. Mario's impression and experience is that working on Rucio has been a positive thing in terms of career path for its developers. Don Quixote would be proud.

Looking to the future, scaling to next-generation data intensive experiments like HL-LHC, the team sees system scalability as in good shape, with its support for horizontal scaling. Larger issues are in deployment and operations. The current site installation approach is complex and constrained, it isn't possible to just go in and add a new feature, which is prohibitive to effective R&D. Also new compact analysis data formats may bring new challenges, and 'virtual data' (data generation on demand in real time), which is receiving attention for its economization of storage, may require new interactions with the data management system. Asked what might be the 'next Rucio', a promising open source project emerging in the community, Mario suggested Harvester, a modular component interfacing between a facility and distributed workflow and data systems.

## Software sustainability

In his talk on research software sustainability, Dan began with the importance of software in the research community. Across the past decade, about 20% of NSF projects representing $10B of research funding discuss software in their abstracts. In the DOE Exascale Computing Project (ECP), ⅔ of the activity areas are research software, representing about $4B. Software intensive projects represent a majority of current publications, and papers on methods and software are among the most cited publications. Almost all researchers use research software, and a majority would not be able to do their research without it. About half of researchers develop software themselves as part of their research, and according to one survey, most spend more of their time developing software than they did a decade ago. A strong contributor to the growth has been the rise of open source software over the past 20 years, for example from there being 200k Sourceforge users in 2001 to 37M GitHub users in 2019.

Accordingly, there are a number of research software organizations that gather, discuss and disseminate experience, techniques, and tools to improve developer productivity and software sustainability, and to coordinate collaborative research software internationally. Some like the Software Sustainability Institute in the UK are long established, others like the NSF's US Research Software Sustainability Institute (URSSI) are just taking shape, responding to the growth.

Software in the research cycle can be divided into two categories, research software and infrastructure software, the former tied to particular research activities, the latter inherently shared and more the target for community software projects, sometimes receiving funding explicitly as a software project. The sharing and recognition of infrastructure software promotes appreciation and reward in the community. We in the community can make conscious choices to favor shared infrastructure software, to build software that is useful to others. Doing so carries costs as well as advantages, such as private code not being an option, giving up exclusive IP, a broadened scope to encompass the requirements of others, and broadened governance to support collaborating with others. Advantages include leveraging the intellectual contributions of others; shared algorithms and methods can be redeveloped and improved by others.

Building open community software proceeds through many stages, from a lone developer to an internal team to a community team, and ultimately to a self-governing open software project supporting a broad user community. Each stage represents a decision point to think about methods, goals and consequences; resources and skills needed vs. those available; incentives and rewards; metrics for success; and mechanisms for sustained support. As a project progresses to include outside contributors, it's important to recognize that the relationship with outside people is different: stress 'friendly', be very open and helpful.

In the absence of active maintenance, software collapse may ensue. This applies across the layers making up a software stack, from project-specific to discipline-specific to the scientific and non-scientific infrastructure software layers below. Software depends on the software layered below it; any change below may cause collapse.

Dan defined software sustainability as the capacity of the software to endure in the future, on new platforms, meeting new needs, fixing bugs when found, following best practices. This takes human effort, from open source contributions or explicit support/funding. Most important to 'doing things right' to build successful, sustainable common software projects is the people, not the software. It's important to set up an organization that establishes the incentives and rewards to create and maintain the software. Open source software success may give rise to explicit support over time when funders see the value; OpenSSL is one particularly interesting example. A factor imperiling sustainability is the 'bus factor'; is project expertise shared across developers, or could a wayward bus collapse the project. Estimates for many open source community projects put the bus factor at 1 or 2. A related key problem for sustainability is that researchers are not rewarded for software development and maintenance in academia. Sustainable software requires sustainable software careers.

Sustaining our developers with recognition and careers requires incentives, citation/credit models, and metrics such as citation counts, user counts (when measurable) and download counts (where possible). Software citation is a pressing issue today: how to adapt a mechanism created for papers and books to software. Software and other digital resources currently appear in publications in very inconsistent ways, such as by citing a publication, a user manual, website, or simply a name. Citing data and facilities have similar issues.

The Journal of Open Source Software (JOSS), a developer friendly, open access journal for research software is designed to help. "If you've already licensed your code and have good documentation then we expect that it should take less than an hour to prepare and submit your paper." JOSS papers are archived, have DOIs, and are increasingly indexed. Zenodo is also an easy and popular way to make the software itself citable with a persistent DOI. It's not enough to have the mechanisms available to cite software, software authors must use them to make their software citable.

The Software Citation Working Group from 2015 to 2017 reviewed community practices, developed use cases and published software and data citation principles. The key principles are importance, credit and attribution, unique identification, persistence, accessibility, and specificity.

An implementation working group took over in 2017, currently working to implement software citation together with institutions, publishers, funders, researchers and others. Metadata standards are being aligned with schema.org, citation checklists for authors/developers have been published, best practices for registries and repositories are in development. A journals task force started this year has developed and published a guidance document for journals and conferences to use.

Essentially all the citation and recognition issues that exist for software also exist for the data sets used by the software. Data deserve the same attention and care as the software. There has in general been much more work on data citation than software citation.

Scientific software developers need better career paths. Most universities do not offer clear career paths. Labs do, and with good financial rewards and promotion opportunities, but availability is limited. University centers such as NCSA, SDSC and TACC offer good examples, giving programmers a home and career path. The Moore/Sloan Data Science Program has contributed to strengthening university opportunities.

Career path and recognition opportunities would be improved by better recognition of Research Software Engineer (RSE) as a profession, particularly outside the labs, as part of the academic community. While not being independent researchers, they have deep engagement with the scientific activities of research groups. They provide continuity, sustainability, stability and maintenance to research software for the research group, the institution and for shared software, the community. But in most academic institutions there is no such career path.

The UK has been a strong driver for the RSE, with RSE fellowships awarded since 2016 and a series of workshops building the community. The international Society of Research Software Engineering was formed in the UK in 2019. In the US, as of Sep 2020 there were 491 US RSE association members. In Canada, licensing restrictions to the use of ‘engineer’ lead to use of the title “research software developers”.

Given a career path, evaluation and promotion determine its trajectory. Appropriate guidelines for both are important. Influencing guidelines and processes demands engagement with them from researchers, particularly senior ones, supportive of software career paths. Templates and guidelines are available for recognizing software contributions and encouraging respected organizations to adopt them.

Our community has begun to specifically address software sustainability. IRIS-HEP and HSF held an initial Sustainable Software in HEP workshop in July 2020, with a report now published. Recommendations include scaling up training and professionalizing training through visiting fellowships, investing in tools that enable and support software sustainability, and creating awards for software work.Common scientific software: Common discussion
Does NP need a NP Software Foundation, analogue to the HSF? Or better that NP leverages and participates in HSF? The balance of the discussion was that participating in the HSF in a more full way would be better than reinventing it. NP’s priorities don’t get focused on in the HSF

today; the HSF is dominated by LHC. An important factor in this is that the HSF operates as a do-ocracy: the influence and direction is in the control of those who participate. HSF's participants make it what it is. The way to ensure that NP priorities get attention in HSF is through NP participation. The HSF would welcome enlarging beyond HEP to NP. The two communities are very close and have much overlap, as well as significant differences, in their needs and priorities. One initial means of creating a closer connection could be to have HSF join JLab and BNL as a sponsor of the S&C Round Table. Round table discussions are a good venue to bring people together around common needs and efforts, leading to combined projects.

Sylvester Joosten offered thoughts on common software from a users perspective, leading to a discussion on common software ecosystems. Independent, modular components that minimize lock-in are preferable even though they take more development time. Orthogonality of tools is important; the interplay between software projects should be well defined and overlapping functionality avoided. Integrated software stacks have a value, in establishing a consistent well-defined ensemble of packages that work together, but it should be a collection of modular components, not a monolith. CERN's EP-SFT group has a project Key4HEP underway to provide such a turnkey stack, similar to tooling ANL is developing. Current, high quality documentation is always a challenge. The openness to a general audience of new projects like ACTS (doc in public and not behind login walls!) is a big benefit and incentive for adoption.

Support for permanent staff to work on software and computing is frequently not part of DOE NP/HEP projects, which means development in those areas often rely on the contributed labor of postdocs and students. It's not uncommon for the main (or sole) developer of key research software to graduate or leave for a new job on a time-scale that is short compared to the full project lifetime. Even if you establish a rigorous process for accepting research software into a project, you can't prevent people from leaving, which results in an increasing body of unmaintained, orphaned code. Are there strategies that can help reduce this problem? Answers to this posed question included

- Career paths! A point made in all the talks and discussed in more detail in Dan's
- Discouraging single-developer software and encouraging teams
- Professional best practices in software development. Unit testing is key, it ensures at least the maintainability of static code. Documentation is key.
- As discussed in the talks, the use of modern tools and best practices is also important to properly training young people. Modern tools often match what they've learned prior to arriving, and sending them off to their next career step with training and experience in modern tools gives them a valuable asset.
- Common software is itself a means of making the software more sustainable. Project developers with their deep knowledge of a key piece of software become a valuable commodity in the field, they get jobs, and expertise on the software is sustained. We see this effect with projects new and old: Rucio, ACTS, Geant4, ROOT.

Common software has benefits but also costs relative to addressing a need (of tractable scale) internally. Incentives are important. "If I only get credit for my internal contributions, why should I

do extra work to help outsiders." The incentives can come at all levels, the research group, the institution, the experiment, and funding agencies. All these levels can recognize the value of, promote and reward common software efforts. Funding agencies in the US and Europe pretty much require that supported software efforts be cross-experiment shared common efforts. But this is an area where NP is distinctly different from HEP. NP has a lot of small groups; group level software is a reality that won't go away. The software problems to be solved often are tractable at the group level. Changing this culture in favor of more broadly scoped common software, requiring more effort and more 'professional' development, is not likely to happen. It is notable that in the astronomy community, also made up of small groups, the Astropy software has been a successful collective common software project widely used in the community.

A key path to success for a common software project in experimental NP/HEP is for the software to emerge from the experiments. This ensures the software is addressing a real need effectively, and gives credibility when others consider it for collaboration and adoption. This is true for ACTS, Rucio, Geant4 and many others.

# The Role of Data Centers in Scientific Discovery

## Data Centers In A Decade

Eric began his presentation by noting the challenge, and perhaps folly, of making technological forecasts a decade in advance. The talk was organized into three sections looking at the scientific landscape, the computing hardware landscape, and culminating in the challenges future data centers would need to meet going forward.

The BNL experience reflects a number of the important ways in which changes in the scientific landscape are affecting computing efforts. The era of Big Data in science is upon us and is no longer the sole purview of High-Energy Physics. The data challenges of the HL-LHC era are well known, but other fields are catching up. NSLS-II and its photon science program at BNL is a case in point, with numerous beamlines and detectors coming on line over the next decade generating GB/s data rates. Nanoscale science, such as the research being done at the BNL Center for Functional Nanomaterials, is generating increasingly large amounts of data from increasingly sophisticated devices, that need to be processed and analyzed quasi-online, requiring modern AI/ML techniques and powerful coprocessors. With the LHC and RHIC entering the exascale era at the same time that the NSLS-II and other instruments begin generating their own large datasets, the challenge for the BNL SDCC and other data centers like it is clear. Data centers will need to adapt to the requirements of these new experimental efforts. A perfect example is the rising use of machine learning methods to meet the growing demand for rapid analysis of increasingly large and complex datasets. Data centers are being asked to provide the specialized coprocessors, fast storage and high-performance networks that these methods need, along with the infrastructure for training on large data samples and, in some cases, tight integration with and feedback to the experimental facilities.

Turning to the hardware landscape, it has become evident that the “golden age” of easy computing is over. We have seen the flattening of the conventional CPU performance growth curve. Future improvements in computing hardware will rely on multiple new approaches, be they novel semiconductor technologies, domain specific accelerators, or even quantum computing, to name a few. New software algorithms, including code that is customized to match application specific accelerators, will also be needed. On the storage side, disk will continue to dominate over the next decade, with both SSD and tape remaining a relatively small but growing fraction. SSDs are cheaper to operate than disk (in terms of power and cooling) and faster to access, but still come in at approximately ten times the cost of traditional HDD. Even with cheaper, lower performance SSDs on the horizon, they will not take over the role of HDDs in data centers for some time. The market for HDD is being driven by hyperscalers (Amazon, Apple, Google, etc.) who consume about half of the market, so we can expect such drives to be around for the foreseeable future. These same hyperscalers are the main consumers of tape systems as well, so that technology will also remain available. The biggest concern is the consolidation of suppliers. IBM, for example, is now the sole remaining company manufacturing tape drives. Still, tape will remain the most competitive means of storing large amounts of infrequently accessed data. The cost of secure and reliable archival storage is substantially lower than that of local disk or cloud when all cost factors (power, cooling, retrieval) are considered. Tape will be part of all storage solutions as they evolve towards improved hierarchical architectures. Finally, it is clear that any efficient solution for data intensive computing requires a high performance, reliable network, the quality of which typically determines the speed of science delivery. Multi-Tbps LAN will be needed, for data streaming from scientific instruments as well as for data processing and analysis. The WAN will grow to the multi-hundreds of Gbps level needed for data exchange between centers.

In light of the changes coming from both the experiment side and the hardware landscape, how will the data center adapt over the next decade? It will no longer be able to provide a one-size-fits-all solution, but at the same time, it must maintain a certain scale to operate cost effectively. It must, therefore, evolve into a composable system that can be tailored to stakeholder needs, even as those change. The foundation will remain the network, long term data repositories and centralized storage, with the other components evolving over time. Cloud-based services may also be incorporated as needed. There will be a need to provide scalable heterogeneous resources in support of new computing paradigms. The agility to adapt to changing user requirements will be critical. Strong engagement with the science will also be required in order to anticipate and plan for evolving needs and to propose/develop optimized solutions.

The data center will no longer be able to function as an isolated entity, but as more of an embedded system in close relationship with both stakeholders and computer scientists. Ideally, computer scientists will belong to the same organization as the data centers, and data center staff will also be embedded within major stakeholder projects. Success will come from collaboration among these three groups. There will be increased integration between the data center and the projects it supports. These projects will benefit from the better hardware

(reliability, scalability, robustness, low latency, etc.), expertise, and cost advantages that the data center can provide. As long as high-performance WAN costs remain sufficiently low, instrument data storage can be geographically distributed and, depending on the data model, network caching can provide fast access, though these potential gains are not obvious and need to be assessed.

The natural conceptual extension of all these changes is the Superfacility Model, developed at LBNL. It offers a blueprint for seamlessly integrating experimental, computational and networking resources to support reproducible science. It is similar to the current LHC distributed computing model but more fully integrated. An example is the Belle II remote RAW data center, which will operate key experiment services outside the host laboratory. The Superfacility Model offers the advantages of shared data center infrastructure, shared expertise and solutions, and easy access for new programs without deploying their own infrastructure. The data center, in essence, becomes a user facility, with easy access to resources for thousands of users from a variety of origins. This access will evolve to be primarily web based. Authentication will require federated identity (no more SSH keys) with authorization management requiring significant development of new tools. The data center will need to provide greater user support, including support for projects (digital repositories, collaborative tools) and communication/feedback.

In summary, the data center of the future will:

- Be an energy-efficient multi-program facility;
- Need to provide high level services as an added value beyond hardware operation, including software solutions to match the needs of users;
- Be more tightly integrated with instruments;
- Be more of a diverse user facility;
- Be a hub for development in close collaboration with computer scientists and researchers;
- Collaborate with other data centers to provide complementary services, share knowledge, and further development.

Following Eric's presentation, the discussion touched upon the future of storage and network technologies and the role of Leadership Class Facilities (LCF) in the Nuclear Physics community, as well as support for the nuclear data repositories that provide the basis for particle transport codes and detector R&D. On the latter point, the BNL SDCC does provide infrastructure and support for the National Nuclear Data Center - an example of the type of scientific engagement promoted in the talk. On the future of storage, Fujifilm's projection of 400 TB tape cartridges will keep that technology competitive. When comparing with the cost of archival cloud storage, like Amazon Glacier, one needs to account for transfer costs. Even so, a hierarchical storage capability that is adaptable to various use cases (even potentially including cloud storage) is important; there is no single solution for all. Strategies for efficient use of tape, such as writing with retrieval order in mind and data carousel, are also important (notably, LTO-8 streaming speeds are already comparable to SATA SSD). We can expect increased usage of object stores, both with and without an additional posix layer, but this will require further software development. When it comes to network technology, our needs should be

well-covered by ESNET for the next 5 to 10 years. The equipment, however, is a significant cost and there is concern that updates like ESNET6 will bring capabilities that lead to new cost models based on quality of service. On the plus side, more advanced network services, such as network caching, do have the potential to be game-changing, depending on data models. Finally, on the question of whether there is a push within NP to increase usage of HPC and LCFs, it is important to make a distinction between the two. Capabilities of HPC technology, such as co-processors, will continue to grow, driven by the market for AI-related research and the community should be working to adapt the code needed to exploit them. The push for LCF usage, however, is being driven by the perception of a shortfall in computing resources needed for HL-LHC with current budget projections and, even now, getting allocations on these systems is becoming difficult.

## Data and Analysis Preservation and Open Data: Experience in PHENIX

Maxim presented the PHENIX data and analysis preservation (DAP) experience in order to benefit the NP community from the shared knowledge and to identify potential areas for collaboration. PHENIX, one of the two large RHIC experiments, is a complex, general purpose detector that completed data taking in 2016 with ~24 PB of accumulated raw data. Active analysis is ongoing however, with about 10 published articles per year.

When considering data and analysis preservation, one must take a holistic approach that includes not only the data, but also the software, along with the build and verification systems and necessary conditions data. The problem is that these elements are typically stored separately and evolve along independent timescales that are much shorter than the desired preservation period. DAP has been recognized as an important effort in HEP and NP, even as a standard item in agency reviews. Unfortunately, the acknowledged “cost justification” does not always translate into the needed funding.

DAP implementation requires two elements: facility infrastructure and knowledge management. The facility must have the continuity and expertise to provide not just mass storage, but also software build and provisioning capabilities, databases and much more. The BNL SDCC fills this role for PHENIX. The NPPS group at BNL provides additional cross-experiment software support, including for DAP.

Knowledge management is experiment specific and includes software provenance, dependencies, APIs and configuration, along with conditions and calibration data that may be analysis specific. There are a lot of moving parts to capture and document. The legacy method for analysis preservation in PHENIX was based on the use of analysis notes which, despite providing a descriptive template along with a tarball of analysis code, were not really designed for reusability. Core software documentation, over the years, has become a problem as well. Knowledge of detector subsystems over time has become dispersed, often relying on PhD theses and other references that are not always easily accessible.

Regardless of the DAP tools being employed, if they are not built into the standard experimental frameworks from the beginning, it is nearly impossible to retrofit them. This challenge has been evident in PHENIX. Diminishing manpower has worsened the situation, along with the pressures on students and postdocs that leave no space and offer no meaningful reward for the extra effort needed to provide comprehensive records of analyses. As pointed out by Daniel Katz above, this type of work, despite its importance, requires a culture change driven by recognition and incentive.

Faced with the challenge above, PHENIX has taken a practical three-pronged approach to DAP consisting of analysis capture, web-based documentation, and a full-featured research material repository. A more focused effort for analysis capture began last year and is limited to annotation and documentation of a few use cases. It is too late at this point to move toward containerized analyses, so it relies on CVS, AFS and scripts for accessing conditions databases and related data files. A new PHENIX website has recently been built and organized in order to serve as a long-term hub for all DAP information. It is a fast and secure static build that relies on external services for hosting content (e.g. GitHub, Zenodo) and DOIs for reference integrity, where possible. Leveraging CERN tools such as InspireHEP, Zenodo and HEPData has provided rich functionality while avoiding the need to develop custom solutions. Indeed, these tools have worked well enough to allow the PHENIX IB to institute a recent policy change requiring a HEPData package as a prerequisite for publication. PHENIX is also making forays into providing “Level 2” open data in the form of annotated ntuples hosted on Zenodo or the CERN Open Data Portal.

A summary of the lessons learned from the PHENIX experience are as follows:

- Data and analysis preservation needs to start early and, preferably, as part of someone’s job description
- Leverage the rich set of available tools
- Website content needs to be curated to avoid content rot
- A flexible and easy-to-use Conditions DB is key
- Prioritize the list of analyses to preserve
- Effort invested in setting up the needed frameworks will pay off in reduction of maintenance costs

The community should consider collaborating in the development of common analysis workflow templates conducive to analysis capture within frameworks such as REANA. Upcoming workshops will provide an opportunity for this type of collaboration as well as further exchange of experience.

The discussion following Maxim’s presentation touched on the difficulties of documenting software over the long term (up to 50 years, as in NNDC efforts); the potential for using a tool like REANA (too early to say as investigations are just beginning); and the expected longevity of containers in the face of changing hardware architectures (in most cases, hoping for the best - may require preservation of old hardware). Other groups chimed in on the state of DAP in their collaborations. JLAB programs have been struggling with this for some time, but have long

made the effort to preserve all information, such as log books, in electronic format. STAR has begun an effort this year to transfer old analyses into HEPData. In Belle-II, the conditions db explicitly includes analysis workflows and could be a useful starting point.

## User Perspective and Requirements

In his presentation Graham addressed the tension between, on one hand, user expectations and needs and, on the other hand, the response of the data center. The focus of the majority of facility users is on science, with computing as a means to an end rather than an end in itself. The genre of user requirements were summarized as:

- ❏ Access to resources.
- ❏ Knowing how to use them.
- ❏ Knowing where and how to get help.
- ❏ Getting the work done.

Each of these was discussed in more detail during the talk. Access to computational resources has three aspects, availability, allocation and priority. Computational facility users are frequently unaware of what resources are available and how to request access or allocation. In the absence of concrete examples and adequate tools and metrics users find it difficult to estimate need and will frequently over or underestimate. Having identified an available resource and receiving an allocation users frequently find the hoops that they have to jump through to gain authorization to use the allocation to be frustrating. Finally having got to the point where work can finally be done, performance often does not meet expectations as the user competes with other workflows that are larger and higher priority than their own. The responsibility for resolving these issues lies with the data center which must provide adequate support in the form of documentation, tools, and clearly identified support staff.

At a lab there are many experts. Knowing who they are and how to contact them is a problem that users frequently face. If that task is made too difficult the user will resort to “folk lore” and copying workflows that worked for others without necessarily understanding what they are doing. In the long term this can multiply problems as configurations that work well in one case may lead to poor performance in others. Providing clear instruction on how to get help, and encouraging it, is important. People often leave asking for help until all other avenues are exhausted which is a significant waste of time. Leaving a user feeling that nobody cares and that they must work it out themselves can lead to more problems. For example, if it is too hard to work out how to submit jobs to a batch system the user may consider using other resources, like powerful desktop computers. A frequent complaint is that it is difficult to obtain adequate, accurate, documentation in an accessible way. Much documentation and institutional knowledge is now provided in an electronic format in an ad-hoc mix of formats that make it hard to quickly find answers. In this respect a searchable knowledge base is very important. It is fair to say that out of date information is trouble for both users and service providers.

Even for experienced users, who have well defined allocations and access to resources using them efficiently can be challenging. A frequent concern is fairness, real or perceived, where a

user believes that they are not able to obtain their allocation. In many cases this is perception or a “feeling” and it is important to the users that they  can clearly understand why they get the performance they do and how to improve. Users frequently believe that system outages are more frequent than they are. It is very easy to remember the few times when things went badly and forget how often things go well (particularly when you are unlucky enough to have been impacted). If left unaddressed this can impact the reputation of a facility.

The goal of the data center is to support the science mission by addressing the needs of the user. The areas to be addressed fall into two broad groupings, tangible and intangible. The tangible things include the computing infrastructure and operation, the compute resources (large clusters or leadership class machines) themselves, storage systems and networking. The intangible include the computing environment, batch systems, software frameworks and libraries, tools for management and movement of data, as well as user support services.

Addressing the tangibles requires and understanding of the needs of the user community:

- ❏ What do the users need?
- ❏ When do they need it?
- ❏ How is it funded?
- ❏ How does it fit into the constantly changing computing landscape?

To achieve this a requirements-based approach is followed. Metrics are developed that users and data centers can use to measure scope of need. There must be a process for gathering and reviewing these metrics and converting needs into a plan to deliver resources. The review phase is particularly important because most science users of computational resources are not in a position to accurately estimate needs. The review process provides the ability to compare new requests with existing or past projects. NP computing workflows are complex and there was general agreement in the discussion after the talk that it is difficult to find a general-purpose benchmark that is a reliable proxy for running user code. It was suggested that maybe some effort can be put into developing a common benchmark that can be shared by facilities. One aspect of facility use patterns, particularly at JLab where there are a small number of high throughput users, is that the use tends to be bursty in nature. There are periods where the systems are heavily loaded and others where they are relatively idle. The reasons for this are complex but two factors weigh heavily, use is data driven and, high precision measurements lead to cautious researchers. Typically, after a data taking run there will be a pause while researchers validate that their detector is well calibrated and understood. After that a sample of data is processed and validated that the processing makes sense before large batches of data are processed, with periods of time between batches when the validation is repeated. In general, once data processing starts there is a requirement to complete as quickly as possible. At JLab the dataset size and processing time for events can vary from experiment to experiment which adds a further level of complexity.

Given the bust nature of the computing need, how should this be addressed? In an ideal world the data center would provision hardware to meet the peak need even if the resource is, on average, underutilized. Unfortunately, at the scale of a national lab, the cost is prohibitive. It is also an impractical approach. The timing of resource need is driven by experiment scheduling

which is, to a large extent, outside the control of the user. So, although the average need can be calculated, with overlapping experiments, the peak need is almost impossible to predict. One strategy is the delayed allocation model. In this model the cluster is sized to be slightly larger than what would be needed to meet the sum of user needs over a twelve month period. Each user is given a fixed sized allocation and a priority. Allocations expire at some rate if not used and the expired allocation distributed amongst active users. In this way, anyone who makes use of their allocated resource is guaranteed to get that resource, or more, during the course of the year. This is problematic when applied to an experimental nuclear physics program. There is an expectation that a dataset will be processed in a timely manner, spreading the allocation over a year means that at worst data processing could take up to a year. Allocations can't expire, detectors generate new data and the rate of processing has to match or exceed the rate new data is generated. In the end computing is not the driver, science is. A group of researchers needs to be sure that processing their data produces accurate results, we cannot tell an experiment that they had their chance to process data and missed it. The solution to this problem is to size the local compute resources for the average annual load, supplement with offsite resources to cover the tall peaks in load, and pay back (if required) by making onsite resources available to offsite users during the dips in local load.

Several types of offsite computing resources were discussed. By far the most convenient, as well as cheapest, is to partner with institutions that collaborate on experiments at the lab. Computing resources become a contribution from the institution to the collaboration. Both GLUEX and CLAS12 at JLab take advantage of this. Another source is the Open Science Grid. Currently both GLUEX and CLAS12 at JLab use OSG resources to supplement the local cluster at JLab. With OSG there is the complication that JLab should contribute in kind by allowing OSG jobs to run on the JLab cluster. The complication is ensuring that the contribution from OSG to JLab is of the same scale as the contribution by JLab to OSG. The Office of Science also has leadership facilities at ORNL and NERSC that can be used in experiment workflows. At the moment this is not a preferred route since these facilities are geared to workflows that are not a good fit, basically they tend to follow the delayed allocation model. Finally there is commercial cloud, this is currently viewed as a last resort except in certain niche areas where the costs can be tightly controlled. For example, AI/ML training, where access to a large number of high end GPUs for a short time may be useful.

To run effectively in a distributed mode it is preferable that as much of the underlying mechanisms as possible be hidden from the user. The talk began with the user's perspective which was very much that simple and easy to understand is the preferred approach. Ideally the user should be presented with one mechanism for job submission and the underlying system takes care of the choice of running locally or remotely.

The talk concludes by asking the question what is the role of the local datacenter in a distributed computing environment? The answer is, to provide the support that the user needs to get their work done. This includes the intangible things such as understanding and interaction with the community and the tangible things like operating local resources and providing access to

remote facilities. In the end, the data center and user community are in a partnership to get the best science out of the facility that is possible.

## The Role of Data Centers in Scientific Discovery: Common Discussion

There is a tension between HPC and HTC when it comes to computing resources. NP traditionally requires HTC, while groups looking at ML require HPC capabilities. Does it make sense for the data center to try to provide both? Or should HPC application be left to the supercomputing facilities? Eric Lancon points out that new hardware will all include coprocessors alongside traditional multicore CPUs, which is not the same thing as HPC, and notes that LCFs are not necessarily the best places for ML training, due to the large data samples that may be needed. Either way, the data centers will need to adapt to address evolving use cases and there is a concern, at both BNL and JLAB, that current AI/ML analysis is being done by individuals who are unaware of the resources the data centers can already provide.

Part of the issue is educating users about how the data center is not like their laptops. For example, understanding the trade-offs between latency and throughput leads to more efficient use of both tape and computing (see PHENIX analysis train or JLAB Swif), but it requires education of and buy-in from the users. This brings up the question: is there enough communication and collaboration between facilities and users at this point? JLAB has moved physicists into the scientific computing staff, which is important, but CS professionals are certainly needed as well, including embedded within the experiments themselves. The data center cannot simply be a place to house and operate an experiment's computing resources. It needs to be integrated in order to function optimally. We need more outreach to computer scientists who can help us incorporate the latest software and computing technology, tools, and processes. Facility ops people and CS research people often do not speak the same language, so we need to actively foster this communication via small projects, workshops and institutional proximity. JLAB and BNL are already doing these things. Physicists in "bridge" positions are important and it would also help to hire CS postdocs and recruit graduate students doing thesis research on relevant topics. The results from such activities will not be immediate, but we will reap the benefit over time.

From the facility point of view, what is the best way to manage stakeholder expectations? The first thing is to be honest and realistic; don't overpromise. The second is to be proactive and do the work to monitor the landscape and anticipate their needs. By the same token, users need to realize that large asks require funding and effort. Large facility support for the smaller NP projects poses particular problems: limited funding, lack of clarity about how to leverage facility resources, and reluctance to cede control over their own S&C infrastructure. From the facility perspective, this support can be managed through early interaction, robust communication, and meaningful integration into the planning process.

The topic of software training in the context of careers was brought up again. There is a strong need for scientists who can program. We can not expect universities to provide this, so our community needs to offer the training, resources, and, most importantly, the recognition and incentives needed in order to create successful career scientists with these skills. We need to keep discussing these new operational and career models in future forums, such as HSF.

Finally, on the DAP topic, there was a caution against limiting ourselves by not preserving raw data. Future technologies (up to 50 years hence) could make analysis/storage of large raw datasets increasingly trivial. For this reason it should be considered, but it must be very well documented.

# Unique Software Challenges for Nuclear Physics

## Unique Challenges: Experience from the 12 GeV Science Program

David Lawrence gave an overview of software developed and used for the 12 GeV program at Jefferson Lab for the experimental nuclear physics effort. The 12 GeV Upgrade was granted Critical Decision Zero status in March 2004, and its component projects form the basis for the principal scientific activity of Jefferson Lab today. Over these 16 years, there have been, and continues to be, a huge number of software activities of different scope and scale to address a variety of functions in obtaining and analyzing nuclear physics data produced at the CEBAF accelerator/detector complex. He focused on the highlights and emphasized lessons learned.

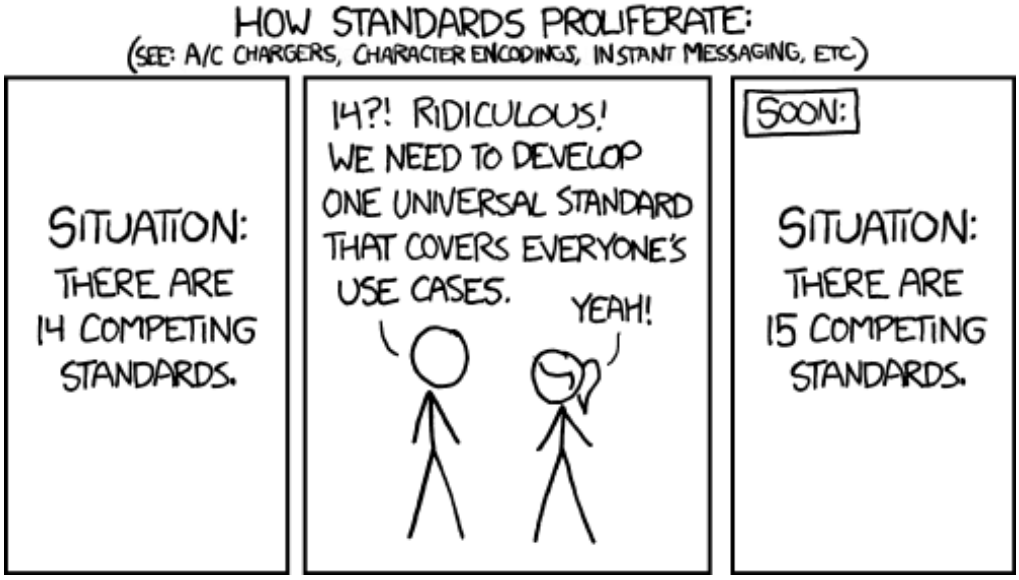
In the realm of data acquisition, the CEBAF Online Data Acquisition system (CODA) was deployed successfully in all of the experimental halls, as was the case during the 6 GeV era at JLab. This is probably the best example of a single code base serving multiple experimental groups at JLab. It required a dedicated group to support it and to implement the unique customizations needed by each experimental hall. Future plans for CODA feature a large role for streaming readout and thus are aimed at a wide user base within the nuclear physics community.

The need to process a large number of calibration streams online is important, not only in simplifying or reducing offline tasks but also by significantly speeding up the time from data taking to usable reconstructed data. A scheme for accomplishing online calibration was implemented in Hall D.

Data processing workflows vary vastly among the experimental halls creating a challenge in writing software common to multiple efforts. This will likely remain a feature of the landscape in

future nuclear physics experiments where a one-size-fits all approach to software will not have success.

Data quality monitoring systems were different for all of the experimental halls despite the surface similarity of the problem in each hall. Here a machine learning approach was rolled out in Hall D. This technique is well suited to the problem; it is a classic classification problem: good data? Bad data?



xkcd.com

Developing software in a collaborative environment always has tension between, on the one hand, creating solutions specific to the problem at hand, and on the other hand, of constructing a software system that is useful outside of the context of the current crisis. Related to this is the choice between developing software within a local group or using a package developed by an outside group. One would like to avoid dependence on large code libraries that over-provide features and functionality while avoiding having to code functions that already have excellent implementations in the wider world. Code written in-house will be well understood, with local expertise, but will likely have to be maintained in-house for the lifetime of the project and perhaps beyond the involvement of the expert. Pride of ownership and personal preference can play large roles in the decision making process, for good and ill. In this area, the importance of communication within the group cannot be overemphasized.

In its planning the 12 GeV program gave emphasis to rapid turnaround from datataking to papers, including software and computing. Here the importance of online calibrations has already been mentioned. Certain aspects of software development are important to keep in mind in this context. Flattening the learning curve for data analyzers is one key area. This includes intuitive interface design and non-expert level documentation with lots of examples. Tasks that are commonly performed should get special attention as candidates for common software. In addition, one of the standard causes of delay in getting to publishable results is “understanding the detector”: reliable efficiency/acceptance calculations, systematic effects, flux

normalization, and the like. One thing to consider is how the software can support obtaining this detector knowledge with internal consistency checks and redundant measurements of relevant quantities.

## Unique Challenges: Software Challenges in Streaming Readout

Jan Bernauer from Stony Brook University gave an inspirational overview of the current status and the prospects of the Streaming Readout (SRO) concept, as compared to the conventional triggered DAQ systems. The two inter-connected distinguishing SRO features are 1) absence of a non-local fixed latency trigger signal, which would control the sub-detector frontend electronics, 2) sub-detector data association by the time stamps rather than by a trigger number.

A traditional DAQ system typically implies a level one (L1) hardware trigger, composed out of a logical combination of  the prompt signals from several separate detector components. Such a signal serves as an indication of a particular physics event type occurrence. The trigger gets distributed to the frontends and prompts them to either perform a conversion of the analog information to a quantity like time difference (TDC) or accumulated charge (QDC) in a fixed length time window, or - in a more modern system - to select a particular continuous range of the stored analog (or digital) samples in a respective circular buffer. The event can be discarded via a fast clear signal by the higher level trigger logic.

In the SRO-based data acquisition system the primary decisions on how to handle the incoming information, perform pre-selection, digitization, and push a subset of the incoming detector data further downstream are made by the (pre-configured) frontends themselves. There is no event builder in a conventional sense in such a system. An SRO frontend would almost always perform a per-channel zero suppression on its own, possibly a per-detector noise suppression and low-level feature extraction, but eventually it will stream the time-stamped fraction of the “essential” portion of the data, based on its own built-in pre-configured logic, without any external validation like L1 or a high level trigger, or a fast clear signal, involved. Jan also made a comment that data selection based on the extracted features would be equivalent to a local trigger decision.

Building a SRO-based system is a challenging task, but - as Jan pointed out - it has several advantages. First, one can eliminate the trigger logic in the frontends. Second, one can relax the timing constraints (no artificial delay lines and long memory buffers). Third, there is no need in the real time event builders, which is always a challenge. And last but not least, a bulk of complexity in the overall system is moved from hardware to software where more people can contribute.

In a short exchange with the audience it was also agreed that the real time event building in case of a pileup is hardly possible anyway, since unambiguous hit-to-event association can not be performed. One can at best talk only about the *event groups* rather than the *individual events*.

An interesting discussion occurred around the question of raw vs post-processed data saving on tape. Indeed e.g. in case of the continuous digitizer hardware one can either store the full waveforms above noise level or only the TOT (time over threshold) / TOA (time of arrival) level integral quantities. It is anticipated that the best strategy can be experiment-specific, and depend on the available resources and the acceptable level of data loss risk, but in general the SRO-based scheme should allow one to store “more information per byte” compared to the traditional triggered DAQ.

Quite some attention was paid in the talk and the discussion to the software challenges associated with the streaming readout paradigm. At the very least one needs to build a robust scalable framework, which can orchestrate the data taking process, monitor data integrity, and incorporate some elements of the near-online event processing on a fraction of data in order to perform the real time data quality checks. The real time data analysis can be problematic to perform for several reasons (lack of the CPU resources at a time of the data taking, but also the realization that a proper calibration of at least some of the sub-detectors may require iterations, and the respective calibration data may simply be not available in the quality required for the “final” data processing pass on the anticipated time scale).

It was emphasized in the discussion that the communication protocols like frontend communication with the concentrators should better be standardized as much as possible in order to simplify the incorporation of the new subsystems.

The data formats, association of data coming from different sub-detector systems, algorithms of data reduction should be designed with the high level processing in mind, efficient use of HPC resources in the offline analysis in particular. The common voice in the discussion was that while it must be beneficial to write individual output files on the per-component basis, one has to pay attention to the efficiency of the subsequent data stream merging in the process of the event building, where different requirements may apply for calibration data extraction and the actual physics analyses.

An interesting discussion happened around the MC generators in the SRO-based data acquisition environment. Indeed in order to have a valid apple-to-apple comparison of the real and the simulated data one needs to either convert MC events produced by the conventional generators into a continuous data stream, or perhaps create a different class of MC generators. The common voice is that the existing “event-based” MC generators should suffice, and it must be the task of the after-burner and digitizer level software to convert hits from individual events into a continuous data stream which would mimic a SRO data acquisition environment.

Several challenges associated with the SRO but also of a more generic nature, were identified in the talk and in the discussion. Contrary to HEP, the NP software effort is often underfunded and understaffed. The NP experiments are typically smaller, therefore more often the same people do both the software development and data analysis. The software landscape is changing, but it looks like the core codes will be mostly written in C/C++ (rather than eg. Python)

for the foreseeable future, ROOT is not going to retire, and the simulations will be done using GEANT 4. A question occurred: what should be the desirable common NP (HEP) code complexity level (indeed e.g. ACTS codes heavily based on the more modern C++17 standard are much harder to read than ROOT and GEANT). The audience tended to agree that it is a matter of continuous self-education, but also teaching the young scientists the modern standards from the beginning of their careers. It also turned out that there exist standard methodology tools to actually estimate the quality (and complexity) of a particular software suite, and a respective GEANT4-focused publication will become available soon.

It is not obvious that the SRO concept automatically leads to the experiment data volume increase. After all, the problem is somewhat equivalent to the (usually hard) decisions on the trigger budget, which is at the end determined by the available resources. A SRO concept however allows a more efficient use of the allowed maximum data rate, as data selections can be adjusted on the fly.

It was clear from the discussion that there is no “one size fits all” SRO solution, but the proponents expressed confidence that even small experiments can benefit from the concepts, protocols and software developed for the large ones. Discussion on the shared software went far beyond the SRO context. The common voice was that the community would benefit from the well packaged libraries with the minimal external dependencies rather than from the framework-level bundles.

A substantial part of the discussion focused on documenting the software. Writing documentation is a time-consuming process, and - as was pointed out in the talk - “the best documentation can only express what the author thinks the code does, not what the code [actually] does”. Good documentation can not improve a poor quality code. It was mentioned by several people in the audience that teaching young scientists the basics of software philosophy, and helping them to write well-structured code is very important for the future of the field. One way to do this is to organize dedicated software tutorials, or perhaps summer schools for undergrads. It was also suggested that writing documentation in parallel with the software development actually helps one to generate a higher quality code. It was acknowledged that the high level tools like UML, which allow one to produce better structured codes, and visualize the connections between the building blocks, are not widely spread in the community. Learning curve is one reason, but physicists tend to ignore IDEs and debuggers as well.

## Unique Challenges: Survey from Nuclear Physics Students and Young Postdocs

The majority of the discussion was on how the software problems and solutions relevant for nuclear physics differ from those of other fields, in particular high energy physics. Two themes

emerged: differences in the scientific problem space itself, and differences in the size of software teams.

## Scientific Problem Space

Nuclear physics experiments may need specialized event generators for Monte Carlo simulations distinct from those publicly available and responsibility for their development will fall on teams within the experimental collaboration. Examples include generators for spin-dependent measurements, relatively recent QCD phenomena, such as GPDs and TMDs, and in general a focus on non-perturbative QCD phenomena versus those amenable to perturbative analysis. Some experiments take on the complexity of multi-dimensional, strongly correlated relationships among data as opposed to focusing on rare events with novel topologies. Others may aim at high precision results which require complex analyses to control systematic errors. These types of features involve analyses that have to consider large numbers of signal events simultaneously, as opposed to separating a few events from a large number of background events. All of these experiments require unique software and computing strategies. Also, the relatively smaller size of experiments, discussed below, is correlated with relatively shorter experimental life cycles and faster changes in scientific goals.

This inhomogeneous landscape in the problem space provides a challenge for application of global solutions and big-picture planning for software and computing.

## Technical Problem Space

Discuss how the streaming readout paradigm challenges NP / gives new opportunities

## Small Group Size

Although some are very large, the average size of a collaboration in nuclear physics is smaller than those in high energy physics. There is a tendency for everyone to “do their own thing.” Indeed, if the task is modest, it may be desirable for individuals to solve problems for the group. And even on the larger experiments, individual analyses can be numerous and quite different from one another, with a small team on each topic.

On the surface, this appears to argue against development of common software solutions within an experiment, let alone for use in the field as a whole. On the other hand, there is general agreement that this non-unified approach has inhibited progress in the field in the past. The field has been in the process of transitioning to experiments with larger data sets and more complex analyses where software projects pursued by individuals working in isolation are losing their value. And these trends are more pronounced now than ever. The old culture, even with its many virtues, cannot effectively address the problems presented by the scale of experiments we see going forward. That said, with relatively smaller groups and fewer team members on a task,

there is likely a premium on careful planning and design of the software effort, with a mix of in-house development, adoption of outside packages, and the choice of appropriate scale throughout. Finding the right balance is a challenge.

# Conclusions

## Common scientific software: Conclusions and take-aways

- Building successful scientific software projects
    - The most important consideration from the beginning in building a successful and sustainable software project is the people: setting up an organization to create the right incentives and rewards to create and sustain the software.
    - Review existing code and take what's good. Engage and consult other experiments early even if the project isn't cross-experiment at the beginning. Cultivate reuse and collaboration rather than building from scratch.
    - Be clear on project objectives and timeline. Short term goals are important, and the project should be grounded in real world deliverables from the beginning, but they need to be aligned with long term objectives.
    - Do not separate development and operations. Developers are most effective when they are aware of their software in use, facing real operational issues. You get an intrinsic feeling for the system if you're using it yourself. A product and team that gives operations central importance builds confidence.
    - Modern, rigorous, professional development workflows and practices contribute to building valuable experience within the developer team, producing developers valuable not only for their software expertise but modern coding practices as well, facilitating careers within and beyond our field. It is essential that projects stay attractive to the best technical students by deploying modern development workflows, being open to new technologies and standards, and providing pathways to publication. Allowing young people to leverage what they've already learned when they arrive, and/or are hacking in the evenings, like the ML software stack and other modern and quickly evolving tools, benefits them and us.
    - Watchwords in successful projects include: modernize, simplify, streamline. Plugin architectures support experiment agnostic designs. Simpler flatter data structures provide data store independence, simpler memory layout, and better CPU and GPU performance. Successful practices include code versioning using git, mandatory unit tests, continuous integration and code reviews together with a validated master release process.
    - Common R&D testbeds help build strong community links and cultivate external contributions.
    - Successful (experimental HENP) software projects most often emerge from experiments. Software can be generic by design from the beginning, but a focus on real experiment use cases gives credibility and trust.

- Building collaborative common software projects
    - We can make conscious choices to favor shared software, to build software that is useful to others. Doing so carries costs as well as advantages, such as no option for private code, giving up exclusive IP, a broadened scope to encompass the requirements of others, and broadened governance to support collaborating with others. Advantages include leveraging the intellectual contributions of others; shared algorithms and methods can be redeveloped and improved by others.
    - Management support is important for successful common software projects. It requires acceptance of objectives wider than just those of the originating experiment, at least over time. Management support means management recognizing the value of the wider investment.
    - Support from funding agencies is important also, and we are seeing this, e.g. in DOE HEP, a software project has to be cross-experiment to be supported. It is widely the case that shared common efforts are seen positively.
    - Developers need the time and space to develop something new, rather than something just a little better. Wider, more generic objectives make for more flexible, capable software from which the originating experiment benefits also. Software is designed with well thought out abstractions making it customizable. But beware of too many abstraction layers for the sake of being generic.
    - Effectively engaging outside collaborators should follow the same processes as for inside collaborators, but the relationship is different with outside people: it's important to be very friendly, open and helpful.
    - Cross-experiment goals should not just (over)burden the core team; spread the effort. Communities within a project should work towards their own priorities, contributing code accordingly. This makes for broadly pursued R&D directions and a richer feature set.
    - Our community has begun to specifically address software sustainability. IRIS-HEP and HSF held an initial Sustainable Software in HEP workshop in July 2020, with a report recently completed. Recommendations include scaling up training and professionalizing training through visiting fellowships, investing in tools that enable and support software sustainability, and creating awards for software work.
- Scientific software career support
    - An emphasis throughout the workshop was the importance of scientific software career support. Not only is this the responsible thing to do to support our developers, it is an important means of ensuring continuity of development and support amid comings and goings of students, postdocs and other software contributors. Sustainable software requires sustainable software careers.
    - A key problem for sustainability is that researchers are generally not rewarded for software development and maintenance in academia. Career path and recognition opportunities would be improved by better recognition of Research Software Engineer (RSE) as a profession, particularly outside the labs, as part of the academic community. While not being independent researchers, they have

deep engagement with the scientific activities of research groups. They provide continuity, sustainability, stability and maintenance to research software for the research group, the institution and for shared software, the community. But in most academic institutions there is no such career path.
    - Encouragement and reward are important. Make our software jobs sticky because they are attractive, they teach and employ valuable skills, and offer a career path.
    - Common software projects create a pool of highly valuable, valued developers who can carry expertise on a key tool to other experiments and communities. By emphasizing common software we contribute to building a pool of experts valuable across the field, contributing to sustainability of the software.
    - Software citations are an important means of supporting developers and their careers. Zenodo is an easy way to make software citable. The Journal of Open Source Software (JOSS), a developer friendly, open access journal for research software is designed to help. Software authors must use such mechanisms to make their software citable. Work is needed to convince the academic environment to count software DOIs as publications in career evaluation.
    - Essentially all the citation and recognition issues that exist for software, also exist for the data sets used by the software. Data deserve the same attention and care as the software.
    - Software publications are the fullest solution to recognizing and recording software work, both research and products.
- NP software
    - Should NP participate in the HSF or build an organization of its own? There are pros and cons, but on balance, opinions favored participating in and leveraging the HSF, and the HSF warmly welcomes NP's participation. The HSF acts as an umbrella for a new ecosystem of community driven software. HSF being a do-ocracy -- its participants make it what it is -- participating is the way that NP can ensure that the interests and priorities of NP are well represented in the HSF.
    - NP has a lot of small groups, the software being written by physicists, not trained programmers. The scale often makes it tractable to meet software needs by writing it in-house. We need to work with this reality.

## Community Forum

There was consensus on starting a new community forum (e.g., a new google group) to discuss issues in software and computing in nuclear physics. It could be seeded with the participants of this workshop.