ROOT RNTuple and EOS: The Next Generation of Event Data I/O

Jakob Blomer[1,*], *Andreas Joachim* Peters[1,**], *Guilherme* Amadio[1], *Philippe* Canal[2], *Florine* de Geus[1,3], *Jonas* Hahnfeld[1,4], *Matti* Kortelainen[2], *Alaettin Serhan* Mete[5], *Vincenzo Eduardo* Padulano[1], *Danilo* Piparo[1], *Giacomo* Parolini[1], *Andrea* Sciabà[1], and *Markus* Schulz[1]

[1]CERN, European Organization for Nuclear Research, Geneva, Switzerland
[2]Fermi National Accelerator Laboratory (FNAL), Batavia, IL, USA
[3]University of Twente, Enschede, The Netherlands
[4]Goethe University Frankfurt, Frankfurt am Main, Germany
[5]Argonne National Laboratory, Lemont, Illinois, USA

Abstract. For several years, the ROOT team is developing the new RNTuple I/O subsystem in preparation of the next generation of collider experiments. Both HL-LHC and DUNE are expected to start data taking by the end of this decade. They pose unprecedented challenges to event data I/O in terms of data rates, event sizes, and event complexity. At the same time, the I/O landscape is becoming more diverse. HPC cluster file systems and object stores, NVMe disk cache layers in analysis facilities, and S3 storage on cloud resources are mixing with traditional XRootD-managed spinning disk pools.
The ROOT team will finalize a first production version of the RNTuple binary format by the end of 2024. After this point, ROOT will provide backward compatibility for RNTuple data. This contribution provides an overview of the RNTuple feature set, the related R&D activities and the long-term vision for RNTuple. We report on performance, interface design, tooling, robustness, integration with experiment frameworks, and validation results, as well as recent R&D on parallel reading and writing and exploitation of modern hardware and storage systems. We will give an outlook on possible future features after a first production release.
Collaboratively, the IT and EP departments at CERN have launched a formal project within the Research and Computing sector to evaluate the novel data format for physics analysis data utilized in LHC experiments and other fields. This part of the project focuses on validating the scalability of the EOS storage backend during the transition from the over 25 years old TTree production format to the newly developed RNTuple format, using both replicated and erasure-coded storage profiles.

1 Introduction

In High Energy and Nuclear Physics (HENP), the vast majority of event data after reconstruction is stored in ROOT files [1]. Once reconstructed, reduced and calibrated data sets are

*e-mail: jblomer@cern.ch
**e-mail: andreas.joachim.peters@cern.ch

derived, persisted, and analyzed. Together, experiments at the Large Hadron Collider (LHC) collected more than 2 EB from Run 1-3 data [2]. By the end of High-Luminosity LHC (HL-LHC), we expect to have more than 10 EB of ROOT data under management, making storage a major cost driver that consumes roughly half of the total High Energy Physics (HEP) computing budget.

The ROOT RNTuple technology represents a major I/O upgrade towards HL-LHC [3]. Succeeding the ROOT TTree technology, RNTuple is a redesigned columnar I/O system for HENP, built to handle the sharp increase in data rates. Compared to TTree I/O, RNTuple produces more compact files, typically 10% to 50% smaller, while also achieving significantly higher read and write throughput, depending on the scenario. Additionally, it improves robustness through strict checksumming and a modernized, safer API.

RNTuple I/O fully exploits modern parallel storage architectures, such as NVMes and object stores, and is designed with forward-looking constraints in mind, including terabyte-sized events and petabyte-sized files.

The software frameworks of the ATLAS, ALICE, CMS, and LHCb experiments have added initial support for RNTuple, including the ability to read and write their TTree data products using RNTuple. At the same time, performance testing and validation of RNTuple using large-scale shared EOS storage yielded encouraging results.

This contribution summarizes the major milestones reached so far. Section 2 outlines the motivation behind RNTuple's development and highlights its key characteristics. Section 3 provides an overview of single-node and small-scale performance studies. Section 4 presents the results of the large-scale performance testing on shared EOS and CephFS storage.

2 RNTuple Overview

An I/O system for HENP event data has unique requirements and challenges related to data layout, organization, software integration, and scalability.

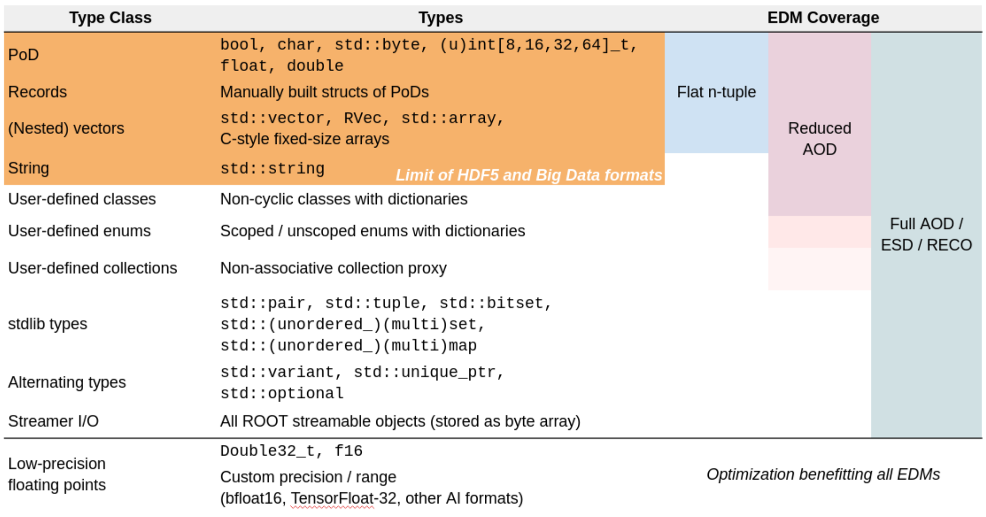
The natural HENP data layout consists of jagged arrays of complex types optimized for a columnar access pattern (see Figure 1). Although Big Data I/O systems such as Apache Parquet [4] provide native support for such data layouts, their type support is too limited (see Figure 2). Similarly, HDF5 [5] I/O, which is commonly used in high performance computing, is not a good fit due to its inherent tensor-based layout, meaning data are organized in multidimensional cubes.

```cpp
struct Hit {
   float x, y, z;
};

struct Particle {
   float E;
   std::vector<Hit> hits;
};

struct Event {
   int eventNo;
   std::vector<Particle> particles;
};
```

Figure 1: Simplified representation of a typical HENP event data model. In practice, more than 1,000 data classes and over 10,000 properties (columns) are common.



Type Class	Types	EDM Coverage		
PoD	`bool, char, std::byte, (u)int[8,16,32,64]_t, float, double`			
Records	Manually built structs of PoDs	Flat n-tuple		
(Nested) vectors	`std::vector, RVec, std::array,` C-style fixed-size arrays		Reduced AOD	
String	`std::string` *Limit of HDF5 and Big Data formats*			
User-defined classes	Non-cyclic classes with dictionaries			
User-defined enums	Scoped / unscoped enums with dictionaries			Full AOD / ESD / RECO
User-defined collections	Non-associative collection proxy			
stdlib types	`std::pair, std::tuple, std::bitset, std::(unordered_)(multi)set, std::(unordered_)(multi)map`			
Alternating types	`std::variant, std::unique_ptr, std::optional`			
Streamer I/O	All ROOT streamable objects (stored as byte array)			
Low-precision floating points	`Double32_t, f16` Custom precision / range (bfloat16, TensorFloat-32, other AI formats)	*Optimization benefitting all EDMs*		

Figure 2: Data types used in LHC experiment data models (EDMs). Analysis Object Data (AOD) is a reduced format with high-level physics objects for efficient analysis. Event Summary Data (ESD) contains detailed reconstruction information for calibration and algorithm development. RECO includes all reconstructed objects and links to raw and simulated data for intermediate processing. All listed types are supported in RNTuple, while other file formats, such as HDF5 and Parquet, can only represent the first few types, which are highlighted with an orange background.

Regarding data organization, HENP data are stored in global federations consisting of sets of files. The I/O layer must enable efficient file access through remote access protocols such as XRootD [6] and HTTP. The rich tooling ensures that files can be merged efficiently, i. e. without recompression and with a complexity comparable to simple concatenation. At runtime, data joining capabilities allow processing of data sets that span multiple files [7].

The HENP I/O libraries must be available for C++ and Python and integrate with experiments' multithreaded and multiprocess frameworks under tight memory constraints. Since experiment data are maintained for decades, strict checksumming, as well as mechanisms for evolving event data models ("schema evolution") are required.

2.1 RNTuple in Practice

To maximize optimization opportunities, RNTuple introduces a new on-disk format and a new API, breaking forward compatibility with TTree. At the same time, RNTuple integrates seamlessly with the established ROOT/HENP ecosystem. Similar to TTree data, RNTuple data is stored in ROOT files; however, unlike TTree, the ROOT file is just one of several possible data containers for RNTuple. Its design allows for alternative storage solutions, such as object stores, to be used as data containers, providing greater flexibility. RNTuple is seamlessly introduced as a new object type within the ROOT file. Analyses written in `RDataFrame` [8] require no code changes and can run equally on TTree and RNTuple data. ROOT tooling, including the browser, `TFileMerger` and the `hadd` command line utility, or `TFile::MakeProject`, applies consistently to RNTuple. Additionally, RNTuple adopts the ROOT I/O schema evolution system [9].

For frameworks and power users, RNTuple provides a modern API for multi-threaded reading and writing, available in both C++ and Python.

The C++ API follows the C++ core guidelines, incorporating features such as smart pointers and signaling runtime errors through exceptions. It was also formally reviewed by an independent panel of experts from the HEP-CCE programne.[1]. The Python API is provided through PyROOT [10], with additional adjustments ("Pythonizations") to better integrate with the Python ecosystem.

The ATLAS, ALICE, CMS, and LHCb experiments have initial integrations of the RNTuple API in their software frameworks.

For languages other than C++ and Python, or special use cases, the RNTuple specification [11] enables 3rd party implementations of the binary format. For instance, initial implementations are already available for the Julia language [12].

3 RNTuple Performance

Based on current and previous measurements, RNTuple consistently demonstrates significant performance benefits in terms of data size, as well as read and write speed, compared to TTree and other I/O formats. Previous studies have already demonstrated the advantages of RNTuple technology over TTree, HDF5, and Apache Parquet [3, 13]. Moreover, RNTuple's implicit parallel I/O and the explicit "parallel writer" provide substantially higher multicore scalability than TTree [14, 15]. RNTuple has been shown to fully leverage modern storage systems, such as NVMe drives with direct I/O and object stores with native data-to-object mappings [16, 17].

The remainder of this section focuses on the latest results in terms of compression and throughput.

3.1 Data Size

Figure 3 compares the average event size of an ATLAS DAOD [18] open data sample, using different compression algorithms. Note that `zlib` is omitted because in all our measurements `zstd` [19] is both faster and produces smaller files. As a result, the default compression in RNTuple changed to `zstd` from the TTree `zlib` default.

RNTuple stores the same content in approximately half the size. Section 4 shows a 39 % size reduction for a `zstd`-compressed CMS nanoAOD [20] sample. Moreover, the relative difference between the various compression algorithms is significantly smaller with RNTuple. When moving existing data production workflows from TTree to RNTuple, it may therefore be beneficial to reconsider the trade-off between throughput and compression ratio in the selection of a compression algorithm, given the substantial differences in compression and decompression throughput.

RNTuple achieves a high compression ratio by structuring data to maximize the effectiveness of standard compression algorithms in identifying and utilizing compressible patterns. One key strategy is the strict separation of offset and data buffers for collections, ensuring that structural metadata and actual values are stored independently, which improves compression efficiency.

Additionally, RNTuple employs type-specific encoding techniques, such as byte-splitting for floating-point numbers and integers. Byte-splitting involves breaking multi-byte values into separate byte streams. This technique enhances compression because the higher-order bytes, which often contain more predictable patterns (such as leading zeros or sign bits),

[1] https://www.anl.gov/hep-cce

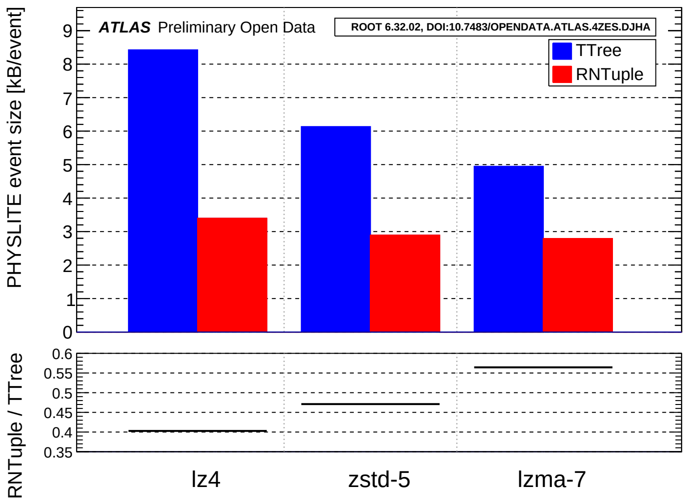


Figure 3: Comparison of the average event size of TTree and RNTuple for an ATLAS DAOD open data sample using different compression algorithms. Credit: Martin Føll

can be compressed more efficiently. For floating-point numbers, byte-splitting can expose common exponent values across multiple entries, further improving compression.

For signed integers, RNTuple uses zigzag encoding to enhance compressibility by mapping both positive and negative values to a closely packed unsigned space. In two's complement, negative numbers have a significantly different binary representation from positive ones, reducing compression efficiency. Zigzag encoding interleaves positive and negative values, ensuring small-magnitude numbers remain compact with minimal bit changes, lowering entropy and improving compression. Furthermore, RNTuple transparently merges identical data buffers ("same-page merging"), which occur, for example, for independent vectors of identical lengths.

3.2 Throughput and Scalability

Figure 4 illustrates the RNTuple read speed-up compared to TTree for various types of final stage ntuples, using different data sources (memory, NVMe drive, HDD, and XRootD remote access). The results confirm the significant performance benefits of RNTuple across a wide range of input types and data access modes, which vary greatly in throughput and latency.

The primary contributors to RNTuple's throughput benefits are asynchronous data prefetching, multi-stream disk access via `io_uring`, and an on-disk data layout optimized for both explicit and implicit parallelization.

These single-core `RDataFrame` benchmarks serve as the baseline for the development of RNTuple. Section 4 presents results for multithreaded and distributed reading.

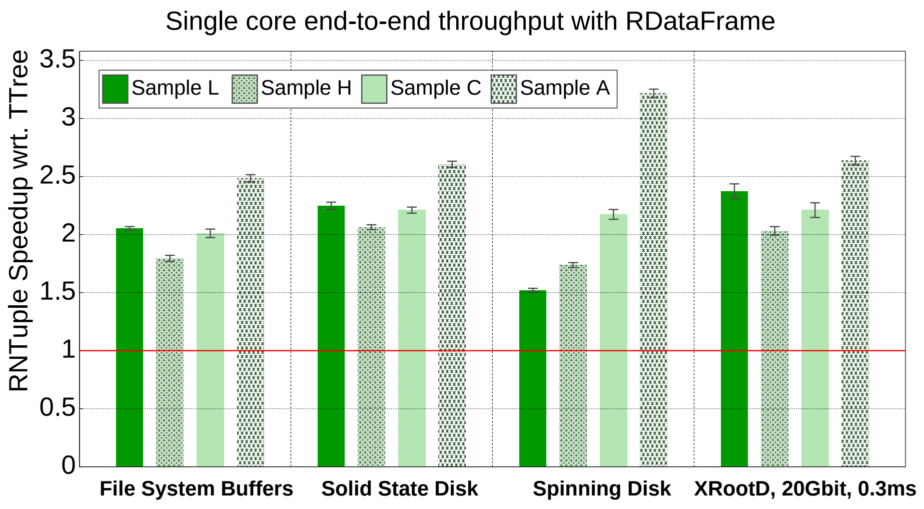


Figure 4: Comparison of read throughput using `zstd`-compressed input files in TTree and RNTuple format for various data sources. Samples 'L', 'H', 'C', 'A' are final-stage ntuples of different shapes (fully flat or with nested vectors) coming from different HEP experiments. The same `RDataFrame` code is used for both TTree and RNTuple inputs.

4 RNTuple and Remote Storage

In the following subsection, we describe the various computing environments that we used to benchmark and optimize RNTuple analysis reading from remote storage. These are based on two open source technologies, EOS [21] and Ceph [22].

CERN EOS Open Storage is a high-performance, scalable, and distributed storage system developed at CERN to meet the data storage and access requirements of large-scale scientific projects, particularly the experiments at the LHC. It is implemented using the XRootD framework.

Ceph is an open source distributed storage platform designed to provide highly scalable, reliable, and flexible storage for modern data-intensive applications. It is widely used in cloud and high-performance computing environments. We use CephFS[2] in our benchmarks.

4.1 Compute and Storage platform

We set up a bare metal computing environment, EO^2C, with 70 compute nodes connected via a 100GbE network for a large-scale validation study. The system was benchmarked using three different storage back-ends. The specifications of the compute nodes are detailed in Table 1.

A shared home directory is available to all compute nodes via a CephFS filesystem, enabling software installation and distributed task execution. The analysis remote access relied on three storage back-ends using file-based erasure coding [23] or object replication:

- EOSPILOT instance with erasure coding RS(10,2)[3] and 20 PB usable space

[2]CephFS is a POSIX-compliant, distributed file system built on top of the Ceph object storage platform.
[3]RS(10,2) employs Reed-Solomon erasure coding with 10 data and 2 parity disks. If not more than two disks are lost from a group of 12, the data remains recoverable and accessible.

Component	**Specifications**
Processor	2x AMD EPYC 7302 and 7313 16-Core Processor
Memory	DDR4 3200 MT/s 16x16 GB - 256 GB
Network Adapter	Intel® Ethernet Network Adapter E810 - 1x100GbE
Filesystems	/ EXT4 2 TB NVME /cvmfs CVMFS filesystem /shared CephFS home directory /jcache CephFS cache directory

Table 1: Compute node system configuration details, including processor specifications, memory, network adapter, and filesystem setup.

- EOSALICEO2 instance with erasure coding RS(10,2) and 150 PB usable space
- CephFS NVME filesystem with 2x object replication and 285 TB usable space

Details of the setup are shown in Figure 5.

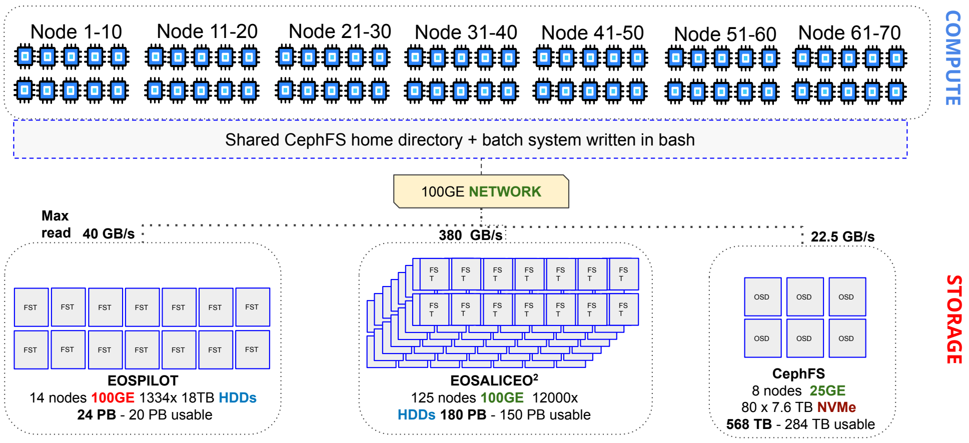


Figure 5: RNTuple benchmarking setup illustrating the compute and storage infrastructure.

4.2 Analysis Grand Challenge Benchmark

The Analysis Grand Challenge (AGC) is a project developed by IRIS-HEP[4] to address challenges in particle physics data analysis. The purpose of this project is to test, benchmark, and improve the scalability, efficiency, and reproducibility of computational tools and workflows required for data analysis in high-energy physics, particularly in preparation for HL-LHC.

Performance studies used the $t\bar{t}$ analysis from the CMS experiment, using a standard analysis Grand Challenge example with `RDataFrame`. The code can be found on GitHub [24]. This analysis represents a worst-case scenario, since only a small fraction of the data is accessed, leading to highly sparse read patterns.

[4]Institute for Research and Innovation in High Energy Physics Software

In all measurements, the data set is ensured to be uncached in both the storage back-end and the client-side cache, wherever applicable. This means that data is not preloaded or stored temporarily in any cache layers, ensuring that the measurements reflect the true performance of data access directly from the back-end storage system. This approach provides a more accurate evaluation of the storage system's capabilities under real-world conditions, where data are usually not cached.

The AGC example can be run in four different modes:

Nodes	*Running Modes*
single	*mt*: multi-threaded
single	*dask-local*: multi-process
multi	*dask-ssh*: multi-node
multi	*dask-remote*: multi-node

Table 2: AGC running modes with corresponding node configurations.

The multiprocess and multinode execution modes utilize `Dask` [25] in combination with `RDataFrame`. The *dask-local* mode launches 64 local worker processes on the 32-core computing node where it is executed. *dask-ssh* mode uses *ssh* to start 64 workers on each server specified on the command line. *dask-remote* mode enables connections to an externally managed `Dask` cluster, such as one deployed on `Kubernetes` [26] infrastructure.

4.2.1 Dataset Sizes

Six datasets were used as inputs for various benchmarks, derived from CMS OpenData $t\bar{t}$, and included three generations of AGC RNTuple files. These are shown in Table 3. Data sets $AGC^{1,2,3}$ reflect the various optimizations performed by modifying the cluster size parameter and the introduction of an adaptive page size algorithm, which led to a further reduction in the size of the data set. The runtime of $t\bar{t}$ AGC is relatively short, making it unsuitable for execution with distributed `Dask` and large-scale parallelism, since processing is shorter than the initialization time[5].

To address this, we generated a $100x$ inflated dataset by duplicating each data file 100 times, resulting in the 104 TB $AGC^{100|200}$ datasets.

4.2.2 Storage Read Pattern

The CMS OpenData analysis $t\bar{t}$ presents a demanding use case for a spinning disk-based infrastructure due to its access pattern: Only 6.4% of the full dataset is read as input, with scattered forward reads that limit HDD performance, as hard drives are optimized for sequential access rather than high IOPS (see Figure 6).

4.2.3 Dataset Size Reduction: TTree to RNTuple

Using `zstd` compression for storing TTree format, instead of the original default `zlib` compression, reduces the size of the dataset by 18%. Switching from the TTree format to the RNTuple format, while using the same `zstd` compression algorithm, results in a 39% reduction in the size of the data set.

[5]This is true for configurations with more than 100 workers.

Name	**Format**	**Comp**	**Size**	**#Files**
AGC^{zlib}	TTree	`zlib` [27]	1.94 TB	787
AGC^{zstd}	TTree	`zstd`	1.59 TB	787
AGC^{1}	RNTuple	`zstd`	1.04 TB	787
AGC^{2}	RNTuple, 2xcondensed Cluster Size 200M	`zstd`	1.04 TB	396
AGC^{3}	RNTuple Cluster Size 100M Adaptive Pagesize	`zstd`	0.965 TB	787
$AGC^{100\|200}$	RNTuple, 100x inflated $AGC^{1\|2}$	`zstd`	104 TB	39600

Table 3: Comparison of different data storage formats and compression techniques for TTree and RNTuple. The table presents file format, compression settings, total data size, and the number of files for each configuration.

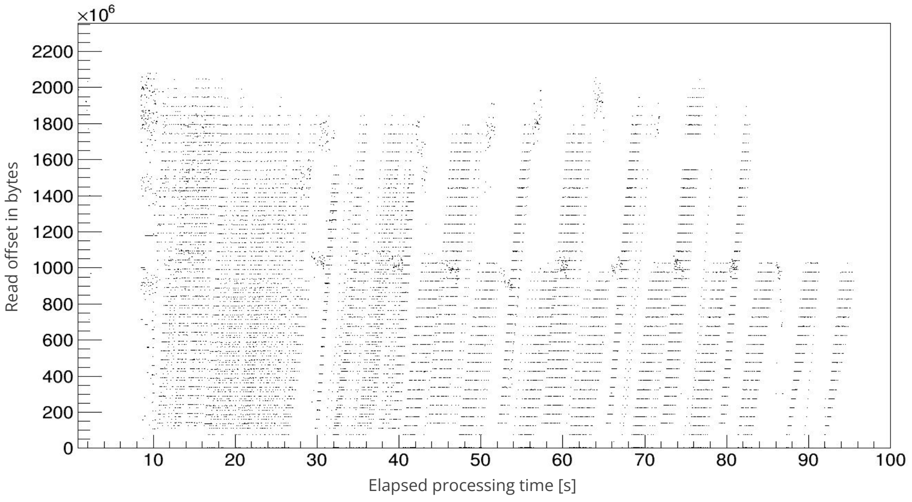


Figure 6: Illustration of the initial AGC^{1} read pattern. The x-axis represents the elapsed processing time (0 to 96 seconds), while the y-axis shows the read offset in bytes within the AGC RNTuple files.

4.2.4 Optimizing Single-Node Multi-threaded AGC Analysis

During the optimization of the multithreaded AGC analysis in a single compute node, we applied five optimizations that reduced the initial run time of the CMS OpenData $t\bar{t}$ analysis from 240 seconds to 70 seconds.

The following optimizations have been subsequently applied:

Vector Read Correction

Building on observations from the previous year and initial attempts in March 2024, we identified a flaw in the vector read implementation when comparing XRootD with HTTP access. This issue caused approximately 60% more data to be read than necessary during each AGC run when using the XRootD protocol. The underlying bug has since been fixed in the RNTuple implementation.

XRootD Connection Demultiplexing

Connection multiplexing allows multiple virtual connections over a single physical link. In XRootD, this boosts resource use, efficiency, and scalability. Each client opens one socket per server, sharing it to access multiple files. In setups like CERN's EOS instances, where servers outnumber client threads, multiplexing has little impact. But it matters for EOSPILOT, which has only 14 servers. Its effect is clear when comparing two runs: the first reads uncached data, while the second is much faster. This may initially seem like just a buffer cache effect. However, when connection demultiplexing is enforced within the clients, the effect nearly disappears, as shown in Table 4:

Mode	**1st Run**	**2nd Run**
Default	142 s	77 s
Demux64	83 s	75 s

Table 4: Runtime comparison of AGC for default multiplexed connections versus 64× demultiplexed connections.

The performance impact of multiplexing originates from the implementation of the XRootD server. The server serializes all requests over a single connection, preventing parallel access to multiple files, even when those files are located on different storage devices. We have added a patch to the AGC code, which creates a thread-exclusive connection from each processing thread to each storage server, as shown in Figure 7. For future production use, de-multiplexing should either be implemented as a feature within the XRootD client or achieved through instance-level configuration by running multiple XRootD daemons per storage server.

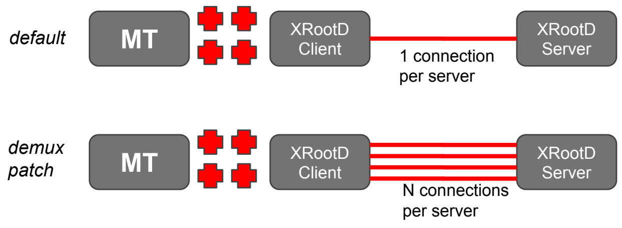


Figure 7: Schematic illustrating the default behavior of XRootD clients, where a single connection is established per storage server (top/default), compared to the patched version, which creates N connections to each storage server (bottom/demux).

Data Format Optimizations

A change in the cluster size and the introduction of an adaptive page size algorithm resulted in a significant reduction in AGC runtime (-50%). The improvement comes from the fact that larger cluster sizes result in larger I/O operations, increasing the average request size from 27 kB to 540 kB. This significantly enhances the bandwidth performance of HDDs.

JCache - a client-side journal cache

As part of this R&D activity, we developed a prototype for a new type of client-side cache JCache [28] that records read and vector read requests as sequential journals in a shared filesystem mounted on computing nodes. Using the NVMe-based CephFS filesystem described in this section, we achieved an additional 7% reduction in runtime, reducing the total runtime from 240 to just 70 seconds, a general improvement of 3.4x.

The contributions of each optimization and the resulting runtime are shown in Figure 8.

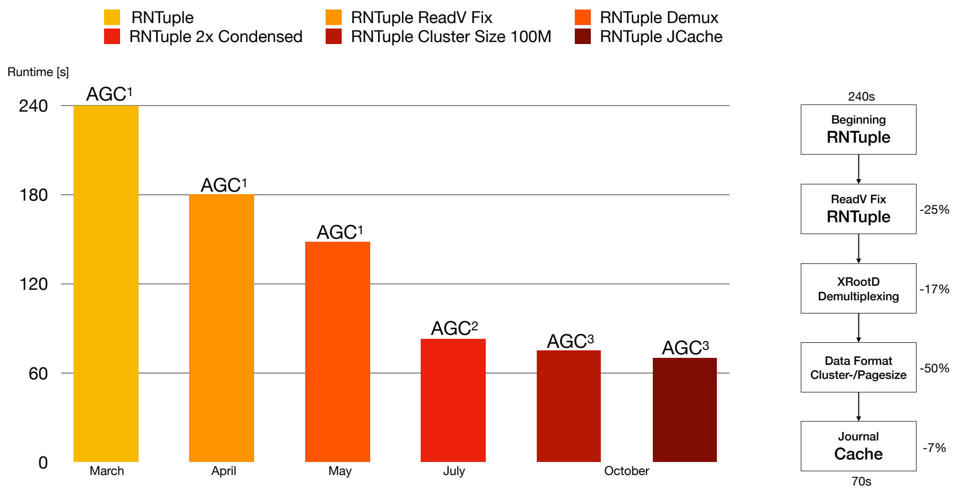


Figure 8: Graph showing the improvement in AGC runtime on a single 32-core computing node over time. The dataset format is labeled at the top of each column, while the right side highlights various optimizations and their contributions to runtime reduction. Data is read from the EOSPILOT instance using the XRootD protocol.

4.2.5 Optimizing Multi-Node `Dask`*-based AGC Analysis*

As previously noted, scaling the 1 TB AGC dataset across multiple nodes is quickly dominated by initialization time, resulting in no significant real-time performance improvements. When scaling AGC^{100} across a few nodes using *dask-ssh*, it successfully saturated instances EOSPILOT and EOSALICEO2, demonstrating that performance was not limited by bandwidth.

Figure 9 illustrates the variation in runtime relative to the number of client nodes. Each client node runs 64 worker processes, e.g. with 30 nodes, we run 1,920 worker processes. Performance effectively scales up to 20 nodes, then scalability diminishes. Considering that EOSALICEO2 is supported by more than 12,000 HDDs, this behavior was not expected. To address the issue based on our previous observations, we created the AGC^{200} RNTuple dataset with a larger cluster size, which significantly reduced the number of read operations

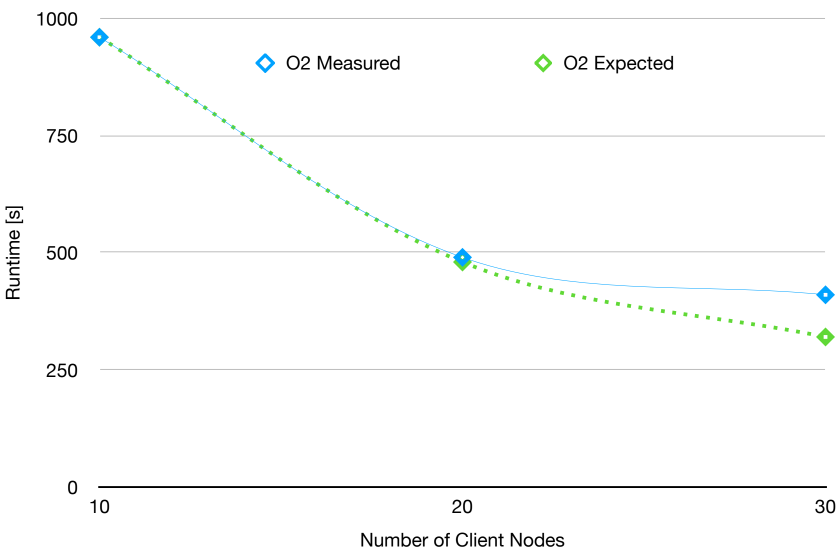


Figure 9: Runtime of the $t\bar{t}$ analysis using the inflated 104 TB AGC^{100} dataset using 10, 20, and 30 client nodes, each running 64 `Dask` worker processes. The green diamond dots represent the expected ideal scaling behavior, while the blue diamond dots depict the measured performance, showing reduced scalability at 30 nodes.

required for the analysis—from approximately two million to just one hundred thousand as explained in

Scalability improvements are shown for the 20 PB EOSPILOT instance in Figure 10 and for the 150 PB EOSALICEO2 in Figure 11. In the last graph, we have isolated the contribution of initialization time to highlight that, at high node counts, it significantly impacts the overall runtime and becomes the primary factor in the reduction of scalability. The first graph demonstrates that the change in the data format has enabled us to achieve a runtime improvement of more than 3x using the same infrastructure. In the EOSALICEO2 measurement, we achieved an average read bandwidth of 43 GB/s during processing.

Figure 12a shows the approximate average and peak bandwidths delivered from the storage system. For 70 nodes we reach 84 GB/s peak and 43 GB/s average. In Figure 12b, we extrapolate the maximum bandwidth required when the data read increases from a sparse 6.4% to a full 100%. This sparsity determines the output bandwidth, which ranges from 43 GB/s to 625 GB/s. Sequential streaming tests on EOSALICEO2 confirm that the instance can achieve an overall streaming throughput of up to 700 GB/s using 8,000 streams.

The comparison of the distributed `Dask` analysis with JCache in CephFS, EOSPILOT, and EOSALICEO2 is shown in Figure 13. The JCache back-end delivers a 18x speedup, EOSPILOT achieves a 28x, and EOSALICEO2 reaches a 40x speedup. The JCache solution is currently limited by the 25GbE network on the storage servers. Upgrading to 100GbE networking could potentially achieve 72x speedup. The key takeaway is that integrating a high-performance NVMe-based cache can significantly accelerate IOPS-limited distributed workloads and is a valid candidate for deployment in an analysis facility.

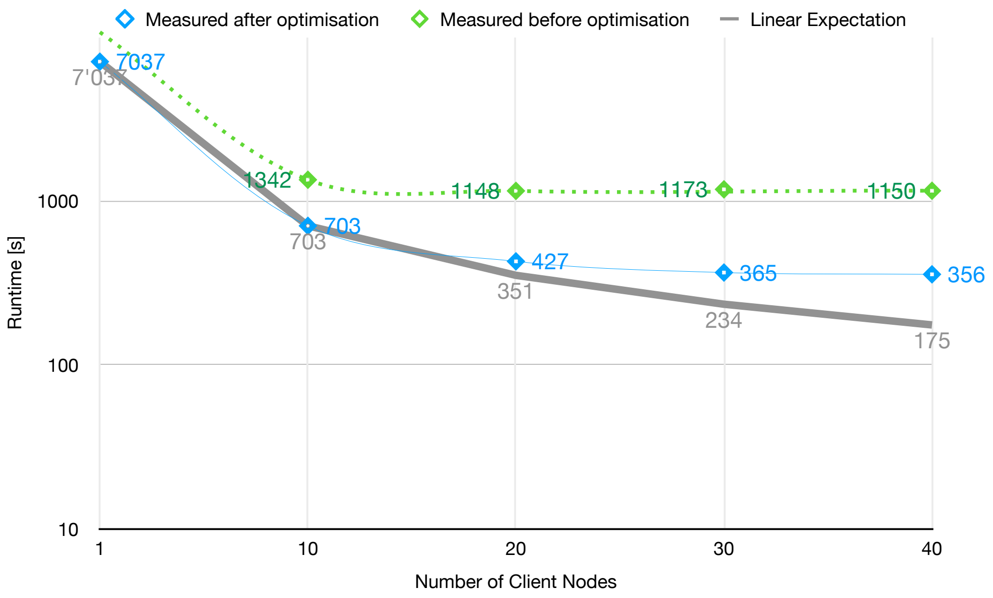


Figure 10: Runtime of the $t\bar{t}$ analysis on the inflated 104 TB $AGC^{100,200}$ datasets using 10 to 40 client nodes, each with 64 `Dask` worker processes, against the EOSPILOT instance. The green line shows results for the non-optimized AGC^{100}, the blue for the optimized AGC^{200}, and the gray line represents ideal scalability. Runtimes plateau beyond 40 clients and are thus omitted.

Summary of Achievements and Insights

RNTuple Improvements Through Benchmarking

Detailed benchmarking has significantly improved the performance of RNTuple for remote access from EOS, as demonstrated with the AGC $t\bar{t}$ example. This effort uncovered several issues that do not manifest in local environments, enabling targeted improvements. Key achievements include a more than threefold reduction in runtime when transitioning from the TTree data format to RNTuple, along with a 39% decrease in data size during this transformation. Furthermore, switching from replication to erasure coding $RS(10+2)$ in EOS resulted in a 40% reduction in data size. Furthermore, a sparse CMS AGC analysis was successfully executed at an impressive processing speed of 43 GB/s using HDD storage.

Implications for Future Analysis Facility Architectures

These results provide valuable information to design future analysis facilities and highlight several key strategies. Combining NVMe and HDDs within shared file systems, along with remote-accessible storage, optimizes both performance and scalability. Adopting a bare metal approach or leveraging platforms like SWAN and HTCondor enables flexible and efficient computational workflows. Additionally, prioritizing the usability and cost-effectiveness of EOS erasure-coded storage enhances large-scale data analysis.

These improvements and recommendations not only improve current workflows, but also lay the foundation for more efficient, scalable, and user-friendly analysis infrastructures.

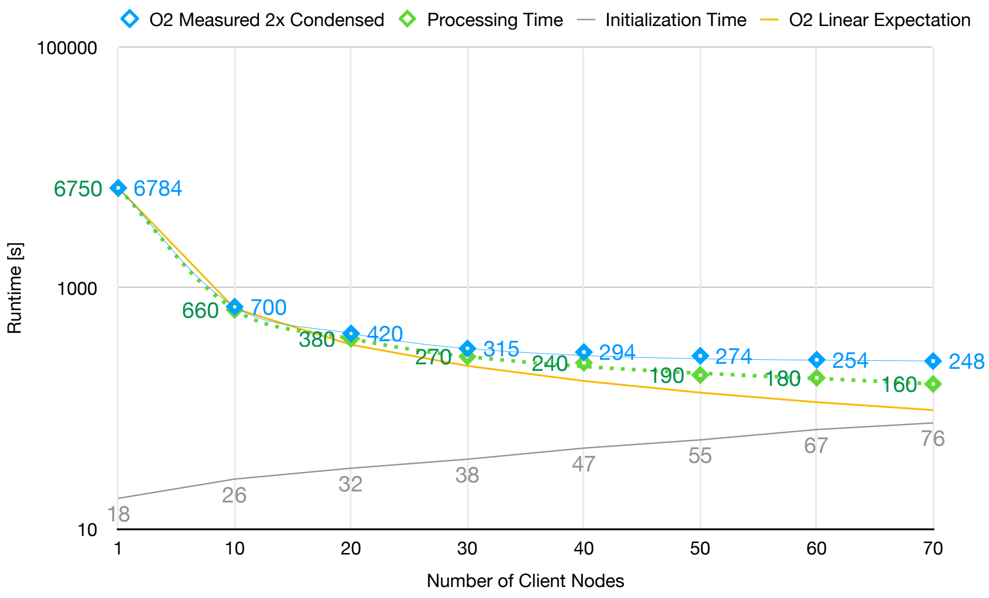
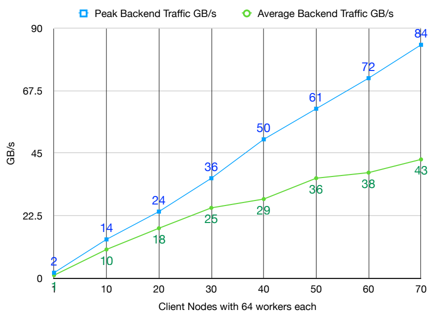
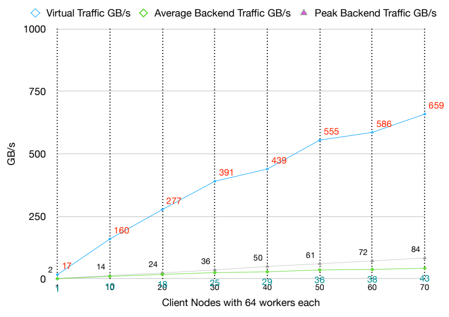


Figure 11: Runtime of the $t\bar{t}$ analysis on the optimized 04 TB AGC^{200} dataset using 10–70 client nodes (64 Dask workers each) against the EOSALICEO2 instance. The gray line shows initialization time, green shows processing time, and blue shows total runtime. The yellow line represents ideal scalability. Runtime plateaus beyond 70 clients.





(a) (b)

Figure 12: (a) Shows the average (green) and peak (blue) read bandwidth provided by EOSALICEO2 during processing. (b) Illustrates the maximum virtual bandwidth (blue line), extrapolated from reading only a 6.4% fraction to 100%.

5 Conclusions

RNTuple is advancing towards full production readiness, establishing a robust foundation for future I/O research and development. Optimizations for both local and remote data access have achieved substantial performance gains, validated through rigorous testing using a challenging worst-case scenario, demonstrating the effectiveness of NVMe and HDD storage back-ends.

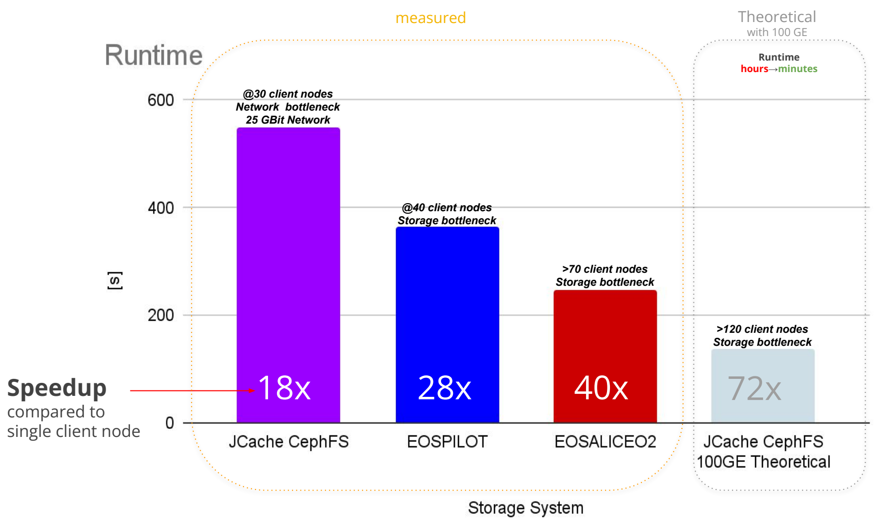


Figure 13: Analysis speedup achieved using different storage back-ends. The first three columns represent measured results, while the fourth column shows a theoretical extrapolation.

In the final phase of this initiative during 2025, we plan to actively collaborate with experiments to refine specific data formats and assess their performance through a large-scale data challenge, ensuring readiness for future challenges of HL-LHC.

References

[1] R. Brun, F. Rademakers, ROOT - an object oriented data analysis framework, Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment A **389**, 81–86 (1997).
[2] LHC Study Group, The large hadron collider: Conceptual design, CERN Yellow Report **CERN-2004-003** (2008).
[3] J. Blomer, P. Canal, A. Naumann, D. Piparo, Evolution of the ROOT Tree I/O, EPJ Web Conf **245**, 02030 (2020).
[4] Apache Software Foundation, Apache parquet (2024), accessed: 2025-05-19, https://parquet.apache.org
[5] The HDF Group, Hierarchical Data Format Version 5 (HDF5) (1997), accessed: 2025-05-19, https://www.hdfgroup.org/solutions/hdf5/
[6] A. Hanushevsky, Xrootd: A scalable data access framework for distributed computing, Journal of Physics: Conference Series **331**, 012017 (2011).
[7] F. de Geus, V.E. Padulano, J. Blomer, P. Canal, A.L. Varbanescu, On-the-fly data set joins and concatenations with ROOT RNTuple, in *Proc. 27th Conf. Computing in High Energy and Nuclear Physics (2024). To appear.* (2024)
[8] G. Amadio, J. Blomer, P. Canal, G. Ganis, E. Guiraud, P.M. Vila, L. Moneta, D. Piparo, E. Tejedor, X.V. Pla, Novel functional and distributed approaches to data analysis available in ROOT, Journal of Physics: Conference Series **1085** (2018).

[9] L. Janyst, R. Brun, P. Canal, ROOT data model evolution (2008), https://root.cern.ch/root/SchemaEvolution.pdf

[10] ROOT Development Team, PyROOT: Python Interface to the ROOT Framework (2025), accessed: 2025-05-19, https://root.cern/manual/python/

[11] The ROOT Team, Tech. Rep. CERN-OPEN-2025-001, CERN, Geneva (2024), https://cds.cern.ch/record/2923186

[12] J. Ling, T. Gal, ROOT RNTuple implementation in Julia programming language, in *Proc. 27th Conf. Computing in High Energy and Nuclear Physics (2024). To appear.* (2024)

[13] J. Lopez-Gomez, J. Blomer, RNTuple performance: Status and outlook, Journal of Physics: Conference Series **2438** (2023).

[14] T.C. collaboration, RNTuple: A CMS Perspective, in *Proc. 27th Conf. Computing in High Energy and Nuclear Physics (2024). To appear.* (2024)

[15] J. Hahnfeld, J. Blomer, T. Kollegger, Parallel Writing of Nested Data in Columnar Formats, in *Euro-Par 2024: Parallel Processing* (2024), p. 18–31

[16] J. Hahnfeld, J. Blomer, P. Canal, T. Kollegger, Direct I/O for RNTuple Columnar Data, in *Proc. 27th Conf. Computing in High Energy and Nuclear Physics (2024). To appear.* (2024)

[17] J. Lopez-Gomez, J. Blomer, Exploring object stores for high-energy physics data storage, EPJ Web Conf **251** (2021).

[18] J. Elmsheuser, C. Anastopoulos, J. Boyd, J. Catmore, H. Gray, A. Krasznahorkay, J. McFayden, C.J. Meyer, A. Sfyrla, J. Strandberg et al., Evolution of the ATLAS analysis model for run-3 and prospects for HL-LHC, EPJ Web Conf **245** (2020).

[19] Y. Collet, Zstandard: A fast real-time compression algorithm, Proceedings of the 2016 Data Compression Conference (DCC) pp. 1–10 (2016). 10.1109/DCC.2016.49

[20] A. Rizzi, G. Petrucciani, M. Peruzzi, A further reduction in CMS event data for analysis: the NANOAOD format, EPJ Web Conf **214** (2019).

[21] A. Peters, L. Janyst, Exabyte scale storage at cern, Journal of Physics: Conference Series **331**, 052015 (2011). 10.1088/1742-6596/331/5/052015

[22] S.A. Weil, S.A. Brandt, E.L. Miller, D.D.E. Long, C. Maltzahn, Ceph: A Scalable, High-Performance Distributed File System, in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)* (USENIX Association, 2006), pp. 307–320, https://www.usenix.org/legacy/event/osdi06/tech/full_papers/weil/weil.pdf

[23] H. Shacham, B. Plank, Erasure codes for storage systems: A survey, ACM Computing Surveys (CSUR) **34**, 309 (2002). 10.1145/507587.507589

[24] IRIS-HEP, Analysis grand challenge (2024), accessed: 2024-03 to 2024-10, https://github.com/iris-hep/analysis-grand-challenge/tree/main/analyses/cms-open-data-ttbar

[25] M. Rocklin, Dask: Parallel computation with blocked algorithms and task scheduling, in *Proceedings of the 14th Python in Science Conference* (2015), pp. 130–136, https://doi.org/10.25080/Majora-7b98e3ed-013

[26] T.K. Authors, Kubernetes: Production-grade container orchestration (2025), accessed: 2025-05-19, https://kubernetes.io

[27] J. loup Gailly, M. Adler, zlib: A massively spiffing library for data compression (1995), accessed: 2022-12-01, https://www.zlib.net/

[28] A.J. Peters, Xrootd jcache plugin (2024), branch: XrdClJCachePlugin, https://github.com/cern-eos/xrootd.git