



Task 3.2 2024 Report

**Evolving the CMS experiment software into a
client-service distributed application for HLT**

Author:

Dr. Andrea Bocci, Prof. Fawaz Alazemi, Andrea Valenzuela,

on behalf of the CMS Collaboration

Date:

April 27, 2025



Contents

1	Overview	2
1.1	Introduction	2
1.2	Requirements	3
1.3	Design choices	4
2	Work plan and milestones	6
2.1	Work plan	6
2.1.1	External deliverables	6
2.1.2	Framework deliverables	7
2.1.3	Task 3.2 deliverables	8
	2024 deliverables	8
	2025 deliverables	9
	2026 deliverables	9
	2027 deliverables	10
	2028 deliverables	11
2.1.4	Contractual milestones	11
3	Status	12
3.1	2024 prototype	12
3.2	Conclusions	14

Chapter 1

Overview

1.1 Introduction

The objective of NGT's Task 3.2 is to extend the CMS simulation, reconstruction and online selection software (CMSSW) to a fully distributed application, in order to achieve more flexibility in the design of the CMS HLT farm for Phase-2.

During the Long Shutdown 2 (LS2) period CMSSW was extended to support asynchronous execution and offload to local NVIDIA GPUs using the CUDA platform; part of the High Level Trigger (HLT) algorithms were rewritten to run on GPUs[1]: thanks to continuous developments the fraction of the HLT that could be offloaded to GPUs during Run-3 ranged from 20% in early 2020 to over 40% in 2024. This fraction is expected to increase to 50% during Run-4 and reach 80% during Run-5[2].

During Run-3 the GPU-enabled framework and reconstruction algorithms were rewritten using the `alpaka` *performance portability* library[3, 4, 5]. In this way a single source code implementation can be compiled for different platforms, and the framework can choose the best one at runtime: x86 and ARM CPUs, NVIDIA GPUs based on the CUDA platform, and AMD GPUs based on the ROCm platform; support for Intel GPUs and eventually Altera FPGAs is under development, leveraging the oneAPI platform.

These developments highlight some of the challenges in the design of a heterogeneous HLT farm for Run-4 and beyond:

- **Balancing CPU and GPU resources:** the ratio of GPU to CPU processing power is expected to evolve significantly over the years; while offloading 50% of the workload necessitates a 1:1 GPU-to-CPU processing power ratio, offloading 80% will require a 4:1 ratio.
- **Integrating specialized accelerators:** supporting highly specialized accelerators like FPGAs or TPUs, while offering significant performance gains for specific algorithms, presents unique challenges. Due to their targeted nature, deploying one accelerator per node is likely impractical

and cost-prohibitive.

- **Managing Diverse Accelerator Types:** while technically feasible, combining different accelerator types (*e.g.*, NVIDIA GPUs and Altera FPGAs) within a single node often introduces significant integration complexities. Conflicts can arise at multiple software layers, including the operating system, kernel, drivers, and runtime environments, requiring careful management and robust compatibility solutions.

The research and development work done in this Task aims to address these challenges by decoupling the algorithms running on the different accelerators from those running on the CPU, splitting the High Level Trigger application into separate processes that may run on a single machine or across different nodes. This approach permits *scaling the amount of CPU and GPU processing power* independently, deploying additional dedicated GPU nodes; it supports *different kind of accelerators* with different software setup, and *sharing a small number of special purpose accelerators* among a large number of CPU nodes.

1.2 Requirements

To attain the goals outlined in the previous section, a loose set of requirements has been drafted, taking into account the characteristics of the CMSSW software stack and of the High Level Trigger environment.

The implementation should be **achievable** and maintainable. CMSSW is composed of over 5'000 individual modules and contains almost 4 millions lines of C++ code: updating all the modules or rewriting a significant part of the code base is not feasible¹. The implementation should not require rewriting the algorithms and data structures that may be executed on a remote node, and it should not pose additional burdens on the maintenance of the code base or the physics validation of the application. The developments and updates should be limited to the core framework and a small number of modules, with *minimal impact on existing CMSSW modules*.

The implementation should be **complete**. Over Run-1, Run-2 and Run-3 the High Level Trigger has made use of many of the characteristics of the CMSSW framework (scheduled and unscheduled execution, multi-threading, concurrent event processing and filtering, asynchronous execution and offloading to GPUs, *etc*): the implementation should maintain the maximum flexibility and aim to support all the framework features. In particular, it should *support arbitrary CMSSW configurations*, without restricting the use of any specific features unnecessarily.

The implementation should be **flexible**. For the HLT application to be able to exploit different kinds and numbers of accelerators, the implementation should *support communication across multiple nodes* with different topologies.

¹For example, the migration of CMSSW to multi-threading after Run-1 took years, required the effort of a large number of developers, and was only feasible thanks to its gradual nature that allowed the coexistence of legacy and updated modules.

The implementation should be **efficient**. The distributed application should achieve a performance as close as possible to that of a single application running over local hardware. The implementation should have low latency, support *efficient data transfer to and from accelerators*, leveraging Remote Direct Memory Access (RDMA) and make *efficient use of high performance network* technologies, like Infiniband or RoCE (RDMA over Converged Ethernet).

The implementation should be **robust**. Hardware and software failures should be contained, limited to the smallest possible scope, with minimal impact on the overall system. The implementation should be *fault tolerant*, and foresee the possibility of restarting failed applications.

1.3 Design choices

An extension to the CMSSW framework was designed to achieve the goals and meet the requirements outlined in the previous sections, leveraging past experience with data transfers to remote GPUs[6] and incorporating findings from an initial prototyping phase. This design foresees the introduction of four new modules in CMSSW: two are responsible for setting up the communication channels and coordinate the event processing: the *remote controller* and the *remote*

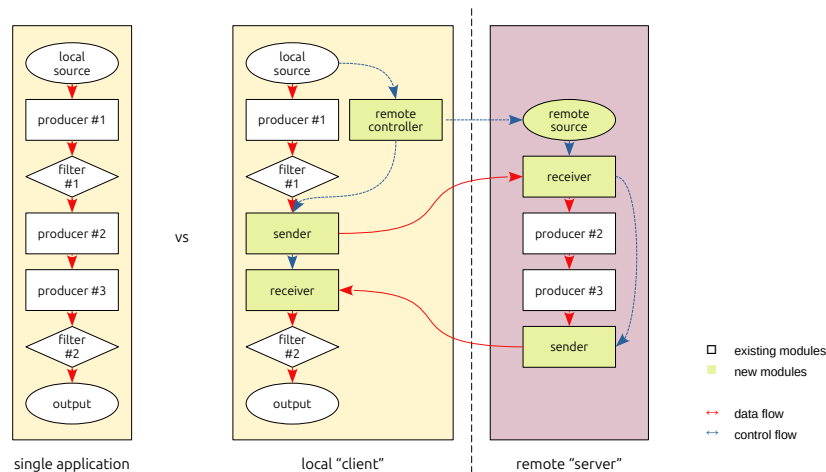


Figure 1.1: Schematic of a distributed CMSSW application, in comparison with a traditional single application. The shapes with a white background indicate the existing modules that compose the traditional application, and are kept unchanged. The shapes with a green background indicate the new modules being developed to make the distributed work possible. The red arrows indicate the data flow between pair of existing or new modules, while the blue arrows indicate the control flow for the distributed application.

source; the other two are responsible for the actual transfer of data products: the *sender* and the *receiver*. This approach leverages the modularity of the CMSSW framework to minimise the impact on the existing code base: since in CMSSW all communication between modules happens through the *event* interface, it is enough to introduce modules that read, synchronise, transfer and replicate the *event* information and data products across processes, leaving all existing modules unchanged.

The initial design adopts Message Passing Interface (MPI) point-to-point operations as the primary communication method. MPI's abstraction layer provides a standardized framework, enabling efficient communication without the need to manage directly the intricacies of diverse network hardware. When paired with high-speed networks and accelerators that support Remote Direct Memory Access (RDMA), high quality MPI implementations can fully leverage the hardware's capabilities to minimize latency and maximize bandwidth. Extensions to the MPI standard, like User Level Failure Mitigation, can be used to provide the necessary robustness for online applications.

Figure 1.1 shows how a single CMSSW application may be split into two separate processes, running on the same or on different machines and communicating over MPI. The original application is composed of a local source (that generates or reads from input files that data to be processed), various sequences of producers (that encapsulate algorithms that process the data) and filters (that apply selection criteria to determine if processing of each event should continue), and an output module (that writes the processed data to some output file).

The distributed application offloads one sequence of producers to a separate process. In this scenario the changes to the local process are minimal: a *remote controller* is added to the local process, in order to set up the MPI communication with the remote process; and the sequence to be offloaded is replaced by a *sender*, to transfer the input data used by that sequence, and a *receiver*, to receive its products, using point-to-point MPI operations. The new remote process uses the *remote source* to set up the MPI communication and synchronise the event processing, a *receiver* receive the input data used by that sequence, the sequence to be offloaded, and a *sender* to send its products back to the local process. All existing modules remain unchanged.

The next chapter describes how this design will be implemented and iterated upon over the duration of the NGTproject.

Chapter 2

Work plan and milestones

2.1 Work plan

To better organise the work and simplify tracking its progress, the overall project has been split into smaller steps and intermediate milestones. Part of these tasks relate to the research and eventual adoption of external libraries and tools, and are grouped as *external deliverables*. Part relate to the CMSSW framework, to identify the existing or missing features required to implement the overall design, and are grouped as *framework deliverables*. Finally, the tasks directly related to the implementation of the HLT client-service distributed application are grouped as *Task 3.2 deliverables*.

The later part of the work plan is expected to evolve as the topics are researched. An up-to-date status and plan of this work is available at <https://cms-ngt-hlt.docs.cern.ch/Task32/workplan/>.

2.1.1 External deliverables

These tasks rely on external tools and libraries, like the ROOT framework or an implementation of the MPI protocol. They indicate the implementation of small prototypes of features required by the overall project, or the evaluation and adoption of specific patterns or libraries.

- **external #1:** serialisation and deserialisation of arbitrary data products. Use ROOT dictionaries¹ to serialise arbitrary data products into a sequence of bytes, copy or transfer them, and deserialise it back into a copy of the original objects.
- **external #2:** implement inter-process communications based on the Message Passing Interface (MPI) standard, for example using the Open

¹ROOT is an open-source data analysis framework: <https://root.cern/>. A ROOT dictionary implements runtime reflection of the C++ types and functions that are available in a library.

MPI open source library². The initial implementation should use the simpler “standard-mode” sends (`MPI_Send`). Open MPI leverages various underlying communication libraries like OpenUCX and NVIDIA GDRCopy to achieve low latency and high bandwidth data transfers across different interconnects.

- **external #3:** investigate alternative MPI send modes: synchronous vs asynchronous, blocking vs non-blocking, *etc.* Alternative modes may lead to a better defined program semantic and achieve better performance.
- **external #4:** evaluate the use of one-sided MPI communications as an alternative to collective operations.
- **external #5:** investigate the use of User Level Fault Mitigation (ULFM) in MPI. ULFM aims to support the continued operation of MPI programs after any type of failures. The key principles are that MPI calls may not block indefinitely after a failure, but must either succeed or raise a non-fatal MPI error. The user program is then responsible for handling the error condition.
- **external #6:** evaluate the use of Remote Procedure Call (RPC)-style communications, for example using the Mercury framework³.
- **external #7:** optionally, evaluate the possibility of using directly a lower-level communication library, like OpenUCX.

2.1.2 Framework deliverables

These tasks rely only on the adoption or development of features in the CMSSW framework, and are independent from the remote communication capabilities being developed in this project. Some of them indicate small prototypes that rely on existing framework features and can later be adopted in the full project, while other indicate missing framework features that are expected to be necessary for the final implementation.

- **framework #1:** generate arbitrary transitions. Implement a framework `Source` that can generate arbitrary run, luminosity section and event transitions, based on its configuration. This is a prerequisite for writing a `Source` that can replicate one process transitions into a separate process.
- **framework #2:** produce collections of arbitrary types. Implement an `EDProducer` that can produce and store into the `Event` collections of arbitrary types, based on its configuration. This is a prerequisite for writing an

²The Open MPI Project is an open source Message Passing Interface implementation developed and maintained by a consortium of academic, research, and industry partners: <https://www.open-mpi.org/>.

³Mercury is a Remote Procedure Call framework specifically designed for use in High-Performance Computing systems with high-performance fabrics: <https://mercury-hpc.github.io/>.

EDProducer that can receive arbitrary collections from a different process and store them into the **Event**.

- **framework #3:** produce collections with arbitrary names. Implement a mechanism for an **EDProducer** to store into the **Event** collections with arbitrary names, instead of using the producer's label. This is a prerequisite for replicating the original name of a collection received from a separate process. A workaround can be implemented in a process configuration using an **EDAlias** or a **SwitchProducer**.
- **framework #4:** implement support for partial event reconstruction and filtering. A key requirement of the High Level Trigger is to stop event reconstruction early and reject events that fail the selection criteria. However, an implementation of an inter-process communication that requires all send and receive operations to be matched may become stuck, if some of these operations are skipped due to the partial event reconstruction on either side of the communication. Additional support may be needed from the CMSSW framework to meet this requirement.
- **framework #5:** implement support for framework reference types (`edm::Ref`, `edm::Ptr`, *etc*). Framework references work within a single process history. A mechanism for letting them work and remain valid when transferred across different processes will be required.
- **framework #6:** handle transfers to and from GPU memory efficiently. Implement support for reading and writing collections directly from and to GPU memory, without copying them into host memory.

2.1.3 Task 3.2 deliverables

These tasks track the actual deliverables of the NGT Task 3.2 project. They have been organised according to an approximate time line, spanning the first three years of the project. During the fourth year the focus will shift from the development of the core features to the performance and robustness of the system. And during the last year the work will concentrate on making the system production-ready for a potential deployment in Run-4.

2024 deliverables

- **step #1:** implement the *remote controller* and *remote source* to provide event synchronisation between a single client and a single server, without data transfers. This builds on top of the functionality demonstrated in the *framework #1* and *external #2* deliverables, and is limited to a single threaded program processing a single event at a time.
- **step #2:** implement a limited version of the *sender* and *receiver* that can transfer collection of types defined at compile-time. This builds on top of the serialisation and deserialisation demonstrated in the *external #1*,

and is limited to a single threaded program processing a single event at a time, with a single sender and a single receiver per process.

- **step #3:** extend the modules to support multi-threaded programs processing multiple events concurrently.
- **step #4:** extend the modules to support multiple senders and receivers per process.
- **2024 prototype:** integrate the previous steps and demonstrate a client-server, multithreaded, distributed test application. This prototype is expected to have some limitations: the types being transferred are defined at compile time, without support for framework references, the configuration does not support event filtering, and the application is limited to a single client and a single server.

2025 deliverables

- **step #5:** extend the *sender* and *receiver* to transfer arbitrary types, defined by the modules' configuration, based on the features demonstrated in *framework #2*. It is based on the use of ROOT dictionaries for serialisation and does not foresee the support of framework references.
- **step #6:** optimise the serialisation and deserialisation of simple C++ collections. In particular scalars, objects of trivially copyable class types, and Structure of Arrays (SoA) types can be copied directly without the need to use ROOT dictionaries.
- **step #7:** extend the *sender* and *receiver* to support event filtering.
- **2025 prototype:** integrate the *steps #5, #6 and #7* on top of the *2024 prototype*. Demonstrate a distributed test application with efficient transfer of collections of arbitrary types, without support for framework references, and limited to a single client and a single server.

2026 deliverables

- **step #8:** read and send data from GPU memory. This builds on top of *framework #6* and the capabilities of MPI to extend the *sender* module to read trivially copyable collections or SoA directly from GPU memory, without first making a copy in CPU memory.
- **step #9:** receive and write data into GPU memory. This builds on top of *framework #6* and the capabilities of MPI to extend the *writer* module to write trivially copyable collections or SoA directly into GPU memory, without first making a copy in CPU memory.
- **first 2026 prototype:** integrate the *steps #8 and #9* on top of the *2025 prototype*. Demonstrate a client-server, multithreaded, distributed

test application with efficient transfer of collections of arbitrary types to and from GPU memory, without support for framework references, and limited to a single client and a single server.

- **step #10:** single client, multiple servers. Extend the *remote controller* to connect to and control multiple remote processes, implementing a distributed application that can offload work to multiple servers.
- **step #11:** multiple clients, single server. Extend the *remote source* to connect to and receive events from multiple remote processes, implementing a server that can be shared by multiple distributed applications.
- **step #12:** distributed configurations. Study how to manage multi-process configurations and ensure their consistency.
- **step #13:** support for framework references. This builds on top on the activity in *framework #5* to implement a mechanism to transfer framework references to a remote process, updating them to remain valid in the new context.
- **second 2026 prototype:** integrate the *steps #10, #11, #12 and #13* on top of the *first 2026 prototype*. This demonstrates a fully distributed application.

2027 deliverables

During 2027 the focus will shift to the optimisation and hardening of the distributed application framework developed during the first three years.

The first topic will be the evaluation of different communication protocols, comparing their performance, robustness, ease of use, and flexibility: synchronous vs asynchronous, blocking vs non-blocking point-to-point MPI communications (*external #3*), one-sided MPI communications (*external #4*), and RPC-style communications (*external #6*). The configuration of the MPI framework and of the underlying protocols, like the UCX library and shared-memory libraries, will be optimised. The direct use of a lower-level library may also be considered, but the added complexity that it would imply must be taken into account. Given the breadth of this topic, these activities are expected to continue into 2028.

The second topic will consider different approaches to improve the resiliency of the distributed application in case of hardware or software issues: the use of redundant servers with automatic fallback, the possibility or automatically restarting failed applications, the introduction of client-side failure mitigation strategies. A potential candidate for MPI-based communications is the User Level Fault Mitigation (ULFM) extension of the MPI standard that will be explored as part of the *external #5* task. These activities will be summarised in a report comparing the approaches to improve the resiliency of the system.



2028 deliverables

During 2028 the evaluation and optimisation of the communication stack is expected to complete, and will be summarised in a report evaluating the performance of different network interconnects and communication protocols.

The final deliverable will be the large scale deployment, evaluation and testing of the whole infrastructure in view of the readiness for the data taking in Run-4.

2.1.4 Contractual milestones

The contractual milestone of **Task 3.2**, to be delivered by the end of 2027, is the “implementation of a client-server, multithreaded, distributed test application, based on the CMS software framework CMSSW, leveraging high-speed host-to-host or shared memory communication.” This will be based on the final prototype developed by the end of 2026, extended based on the findings of the research activity conducted in 2027, and will be the base for the large scale test and deployment foreseen for 2028.

Chapter 3

Status

3.1 2024 prototype

For new CMSSW modules have been developed during 2024 to implement inter-process communication over MPI, following the design described in Section 1.3 and the plan described in Section 2.1: the `MPIController` and `MPISource` to set up the communication channels and synchronise the processing of events between the two processes (see *step #1* in Section 2.1.3); the `MPISender` and `MPIReceiver` to transfer data between the two processes (*step #2*). All modules support multi-threaded operations (*step #3*); the `MPIController` and `MPISource` serialise the event synchronisation (this is an inherent limitation of any `Source` module in CMSSW), while the `MPISender` and `MPIReceiver` can process any number of concurrent events. The distributed application can contain up to 255 pairs of `MPISender` and `MPIReceiver` modules (*step #4*), that can exchange any number of collections defined at compile time. An extension to support transferring arbitrary collection defined by the modules' configuration (*framework #2* and *step #5*) is being reviewed by the CMSSW framework experts.

To demonstrate the capabilities of this design and prove the functionality of the work done so far, a first prototype of a distributed HLT application was developed by the end of 2024. The prototype consists of a full HLT application, based on the configuration used during the 2024 data taking[1, 7], split into two processes: the local process, performing all HLT reconstruction and selection with the exception of the HCAL reconstruction; and the remote process, performing for every event the HCAL local reconstruction and particle flow clustering on GPUs.

The local process reads the raw data from input files, uses an `MPIController` to connect to the `MPISource` in the remote process and propagate the event transitions to it. It uses an `MPISender` to transmit the raw data of every event to the remote process and two `MPIReceiver` modules to collect the HCAL reconstructed hits and particle flow clusters from it. In parallel, the local process

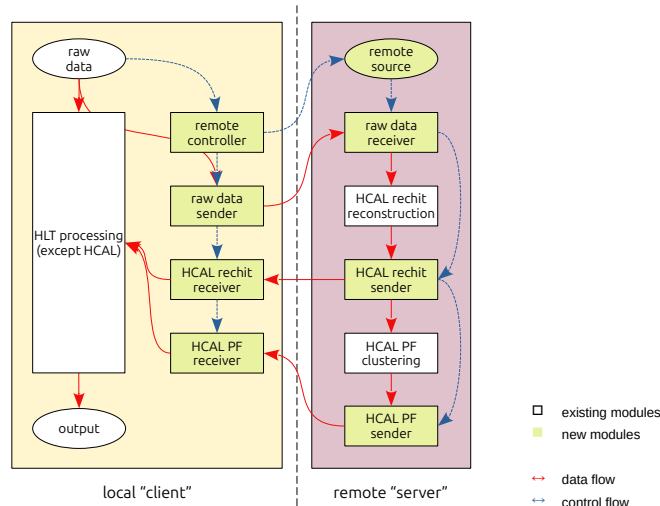


Figure 3.1: Schematic of a distributed HLT application where the HCAL local reconstruction and particle flow clustering is offloaded to a remote process over MPI. The shapes with a white background indicate the modules and sequences that compose the original HLT application. The shapes with a green background indicate the new modules introduced to set up the MPI communications and transfer the data between the two processes. The red arrows indicate the data flow, while the blue arrows indicate the control flow and dependencies for the distributed application.

performs the rest of the HLT reconstruction and event selection.

The remote process uses an `MPISource` to set up the MPI communication and receive the event transitions; it uses an `MPIReceiver` to receive the raw data, it runs the HCAL local reconstruction on GPU producing the HCAL reconstructed hits and particle flow clusters, and it uses two `MPISender` modules to transfer back those collections.

Figure 3.1 shows the diagram of the prototype, split into the local and remote processes. Figure 3.2 shows side by side the logs of the local (left) and remote (right) processes of the prototype HLT application. Three phases of the application are highlighted: (a) the initialisation the processes without (left) and with (right) a GPU, (b) the initialisation of the MPI communication modules, and (c) the event processing. The log of local process shows also the messages from the initialisation of the source reading the raw data and of the FastJet library[8] used in the HLT event processing. Figure 3.3 shows the time spent in the various algorithms that compose the 2024 HLT application menu, and highlights the parts that are offloaded to the remote process, accounting for about 10% of the event processing time.

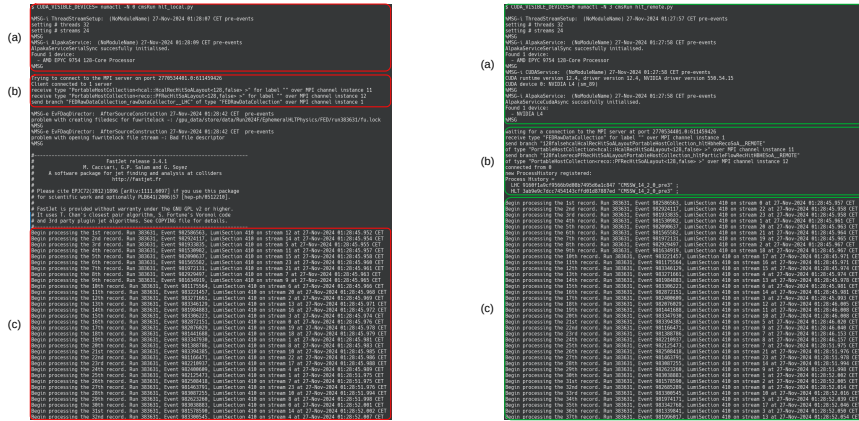


Figure 3.2: Logs of the local (left) and remote (right) processes of the prototype HLT application. Three phases of the application are highlighted: (a) initialisation the processes without (left) and with (right) a GPU; (b) initialisation of the MPI communications; (c) event processing.

3.2 Conclusions

The objective of NGT’s Task 3.2 is to extend the CMS simulation, reconstruction and online selection software (CMSSW) to a fully distributed application, in order achieve more flexibility in the design of the CMS HLT farm for Phase-2.

Chapter 1 describes in detail these goals, the requirements drafted to achieve them, and the initial design choices derived from those requirements.

Chapter 2 presents a detailed work plan for the first three years of the project, with yearly prototypes culminating in the readiness of a fully working distributed HLT application. This is followed by further research activities to improve the application’s performance and robustness during the fourth year, and deployment and commissioning activities during the final year, in view of the readiness for the data taking in Run-4.

Following to the work plan, the first prototype was completed by the end of 2024, as detailed in Section 3.1. This prototype demonstrated a distributed HLT application capable of offloading a portion of the reconstruction algorithms — accounting for approximately 10% of the runtime — to a separate process via MPI.

The work on the 2025 framework tasks has already started, in collaboration with CMSSW experts, and the research activity on the external tasks is expected to begin in earnest in the coming months, with the addition of dedicated person-power to the project. Thus, the activities in Task 3.2 are fully on track to produce the 2025 and 2026 prototypes and meet its contractual milestone by the end of 2027.

Bibliography

- [1] CMS Collaboration. “Development of the CMS detector for the CERN LHC Run 3”. In: *JINST* 19.05 (2024). CERN-EP-2023-136, CMS-PRF-21-001. All figures and tables can be found at <http://cms-results.web.cern.ch/cms-results/public-results/publications/PRF-21-001>, P05064. DOI: [10.1088/1748-0221/19/05/P05064](https://doi.org/10.1088/1748-0221/19/05/P05064). URL: <https://cds.cern.ch/record/2870088>.
- [2] CMS Collaboration. *The Phase-2 Upgrade of the CMS Data Acquisition and High Level Trigger*. Tech. rep. CERN-LHCC-2021-007, CMS-TDR-022. Geneva: CERN, 2021. URL: <https://cds.cern.ch/record/2759072>.
- [3] Benjamin Worpitz. “Investigating performance portability of a highly scalable particle-in-cell simulation code on various multi-core architectures”. Master Thesis. Technische Universität Dresden, Sept. 2015. DOI: [10.5281/zenodo.49768](https://doi.org/10.5281/zenodo.49768). URL: <http://dx.doi.org/10.5281/zenodo.49768>.
- [4] Erik Zenker et al. “Alpaka - An Abstraction Library for Parallel Kernel Acceleration”. In: IEEE Computer Society, May 2016. arXiv: [1602.08477](https://arxiv.org/abs/1602.08477). URL: <http://arxiv.org/abs/1602.08477>.
- [5] Alexander Matthes et al. “Tuning and optimization for a variety of many-core architectures without changing a single line of implementation code using the Alpaka library”. In: June 2017. arXiv: [1706.10086](https://arxiv.org/abs/1706.10086). URL: <https://arxiv.org/abs/1706.10086>.
- [6] Ali Marafi, Andrea Bocci, and CMS Collaboration. “Data transfer to remote GPUs over high performance networks”. In: To be published in the Journal Of Physics: Conference Series. 2025. URL: <https://indico.cern.ch/event/1106990/papers/5011939/files/11762-ACAT2022.pdf>.
- [7] CMS Collaboration. “2024 HLT timing and throughput results”. In: *CMS Detector Performance Summaries* (Oct. 2024). CMS-DP-2024-082. All figures and tables can be found at <https://twiki.cern.ch/twiki/bin/view/CMSPublic/DP2024082>. URL: <https://cds.cern.ch/record/2914421>.
- [8] Matteo Cacciari, Gavin P. Salam, and Gregory Soyez. “FastJet User Manual”. In: *Eur. Phys. J. C* 72 (2012), p. 1896. DOI: [10.1140/epjc/s10052-012-1896-2](https://doi.org/10.1140/epjc/s10052-012-1896-2). arXiv: [1111.6097](https://arxiv.org/abs/1111.6097) [[hep-ph](https://arxiv.org/abs/1111.6097)].