Reconstruction in Key4hep using Gaudi

Juan Miguel Carceller[1*], *Wenxing* Fang[2], *Mateusz* Fila[1], *Brieuc* Francois[1], *Frank-Dieter* Gaede[3], *Gerardo* Ganis[1], *Benedikt* Hegner[1], *Xingtao* Huang[4], *Sang Hyun* Ko[5], *Weidong* Li[2], *Teng* Li[4], *Tao* Lin[2], *Thomas* Madlener[3], *Leonhard* Reichenbach[1,6], *André* Sailer[1], *Swathi* Sasikumar[1], *Juraj* Smiesko[1], *Graeme A* Stewart[1], *Álvaro* Tolosa-Delgado[1], *Xiaomei* Zhang[2], and *Jiaheng* Zou[2]

[1]CERN, Geneva, Switzerland
[2]IHEP Beijing, China
[3]Deutsches Elektronen-Synchrotron DESY, Germany
[4]Shandong University, China
[5]Seoul National University, Korea
[6]University of Bonn, Germany

Abstract.
Key4hep, a software framework and stack for future accelerators, integrates all the steps in the typical offline pipeline: generation, simulation, reconstruction and analysis. The different components of Key4hep use a common event data model, called EDM4hep. For reconstruction, Key4hep leverages Gaudi, a proven framework already in use by several experiments at the LHC, to orchestrate configuration and execution of reconstruction algorithms.
In this contribution, a brief overview of Gaudi is given. The specific developments built to make Gaudi work seamlessly with EDM4hep (and therefore in Key4hep) are explained, as well as other improvements requested by the Key4hep community. The list of developments includes a new I/O service to run algorithms that read or write EDM4hep files in multithreading in a thread-safe way and a possibility to easily switch the EDM4hep I/O to the new ROOT RNTuple format for reading or writing. We show that both native (algorithms that use EDM4hep as input and output) and non-native algorithms from the ILC community can run together in Key4hep, picking up on knowledge and software developed over many years. A few examples of algorithms that have been created or ported to Key4hep recently are given, featuring the usage of Key4hep-specific features.

1 Introduction

Event processing frameworks form the backbone of modern high-energy physics software, orchestrating the complex chain of algorithms that transform raw detector data into physics results. For future collider studies, the Key4hep project provides a common software framework that enables sharing and reusing development efforts across different experiments. This framework has evolved significantly in recent years, particularly in its event processing capabilities.

*e-mail: j.m.carcell@cern.ch

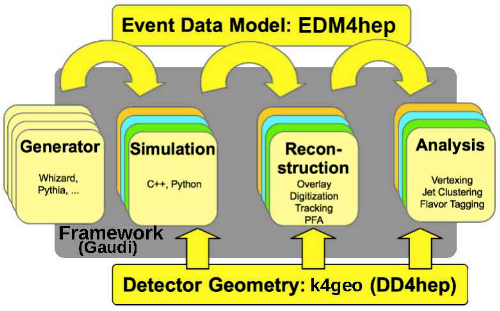


Figure 1. Main ingredients for the Key4hep project: geometry information, event data information, and an event-processing framework.

This paper focuses on the event processing implementation in Key4hep, specifically the integration and extension of the Gaudi framework to meet the needs of future collider experiments. We describe how Key4hep leverages Gaudi's capabilities to support modern multithreaded processing through functional algorithms while maintaining compatibility with code developed by the Linear Collider community. Recent developments enable efficient parallel processing without sacrificing interoperability with existing software.

The foundation of Key4hep rests on three main components, illustrated in Figure 1: Gaudi [1] for event processing, DD4hep [2, 3] for geometry description and detector simulation, and EDM4hep [4–7] implemented using podio [8, 9] for the event data model and I/O operations. A consistent stack is provided on CVMFS, built with Spack [10]. Several experiments are actively participating in Key4hep development, including CEPC, CLIC, EIC [11], ILC, FCC, and the Muon Collider project. The source code is openly available on GitHub[1], with regular open meetings to discuss ongoing developments.

2 The Key4hep Framework

The Key4hep framework integrates various components to provide a complete event processing solution for future collider experiments. At its core, the framework employs Gaudi for event orchestration, with significant adaptations to integrate it within Key4hep and support compatibility with existing software ecosystems. This section describes Gaudi, the different types of existing algorithms and other related components in Key4hep.

2.1 Gaudi

Gaudi is an event-processing framework used by LHCb, ATLAS, and other high-energy physics experiments. It provides tools and services for running and monitoring algorithms. The objects produced by these algorithms are stored in Event Stores, which can be single or multiple instances in multithreaded environments. In a simplified view, Event Stores function as maps that associate object names with their corresponding data objects and algorithms access data through these Event Stores, see Figure 2. As a comprehensive framework, Gaudi offers additional capabilities including multithreading support, result auditing, and histogramming functionality, among others.

While an Event Store can handle any object type, integrating it with EDM4hep presents several challenges. A key complexity is that EDM4hep collections must always interface

[1]https://github.com/key4hep

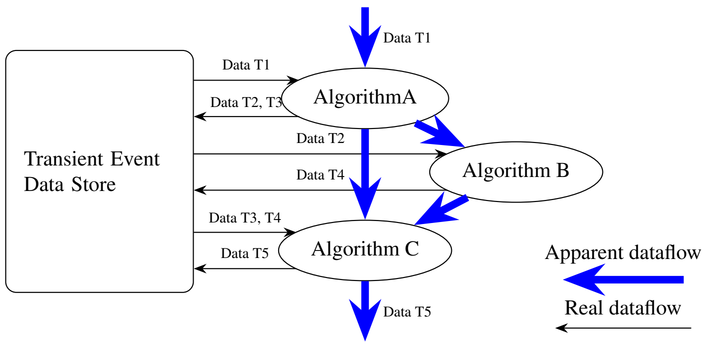


Figure 2. Diagram showing the data flow between algorithms and the Transient Event Data Store. The thick blue arrows show the apparent data flow, while the thin black arrows show the real data flow.

through a `podio::Frame` for input and output, which serves as a collection container and maintains ownership of its collections. Initially, the `PodioDataSvc` was developed to facilitate reading from and writing to `podio::Frame`. In this article, we will refer to algorithms utilizing this service as handle-based algorithms. However, the `PodioDataSvc` lacks multithreading support, which led to the development of a new service called `IOSvc`. This newer service is specifically designed to work with functional algorithms (stateless algorithms that can operate in a multithreaded environment) enabling efficient reading and writing of EDM4hep objects in parallel processing contexts.

While Gaudi provides the fundamental framework for event processing, its integration with EDM4hep required specific implementation strategies. The following sections describe two approaches to this integration: handle-based algorithms, which were the initial solution but are now deprecated, and functional algorithms, which were developed to enable multithreaded processing.

2.2 Handle-based algorithms

The initial integration of EDM4hep and Gaudi relied on the `PodioDataSvc` service, along with read and write algorithms (`PodioInput` and `PodioOutput`) and custom handle and wrapper classes. A typical algorithm chain operates as follows:

1. The `PodioDataSvc` either creates an empty `podio::Frame` or, when there is an input file, the frame is populated with data read from the file.

2. If there is an input file, `PodioInput` runs first, instructing the `PodioDataSvc` to place either all collections or a specified subset into the Event Store.

3. Subsequent algorithms execute, reading their inputs from the store. Any output they generate is written to the `podio::Frame` managed by the `PodioDataSvc` and a wrapper of the data is put in the Event Store.

4. When writing to a file, `PodioOutput` runs last, writing either all collections or a configurable subset from the `podio::Frame` to the output file.

In summary, the key feature of this workflow is that `PodioDataSvc` maintains a `podio::Frame` that retains ownership of all collections, whether they originate from file

```cpp
struct ExampleProducer final :
  k4FWCore::Producer<edm4hep::MCParticleCollection()> {
  ExampleProducer(const std::string& name, ISvcLocator* svcLoc)
      : Producer(name, svcLoc, {},
                 KeyValues("OutputCollection", {"MCParticles"})) {}

  edm4hep::MCParticleCollection operator()() const override {
    auto coll = edm4hep::MCParticleCollection();
    return coll;
  }
};
```

```python
producer = ExampleProducer("Producer",
           OutputCollection=["MCParticles"])
ApplicationMgr(
    TopAlg=[producer],
    EvtSel="NONE",
    EvtMax=10,
    ExtSvc=[EventDataSvc("EventDataSvc")],
    OutputLevel=INFO,
)
```

Figure 3. Example of a functional algorithm in C++ (left) and the corresponding configuration in Python (right). The algorithm creates an empty collection of `edm4hep::MCParticle` and puts it in the Event Store. With additional configuration it could be saved to a file, for example.

input or algorithm output. This functionality is achieved by implementing a custom wrapper (called `DataWrapper`), different from Gaudi's default wrapper, which does not assume collection ownership. When Gaudi deletes all wrappers in the Event Store after each event, any data that is not owned by a `podio::Frame` is also destroyed. Algorithm developers interact with this system through custom `DataHandle`s, which internally manage the custom `DataWrapper`s, keeping the implementation details hidden.

For metadata management, `PodioDataSvc` maintains another dedicated `podio::Frame`. Algorithms can interact with this frame to add or retrieve parameters. Retrieving a parameter is always possible, but adding is only allowed before or after the main event loop, not during its execution.

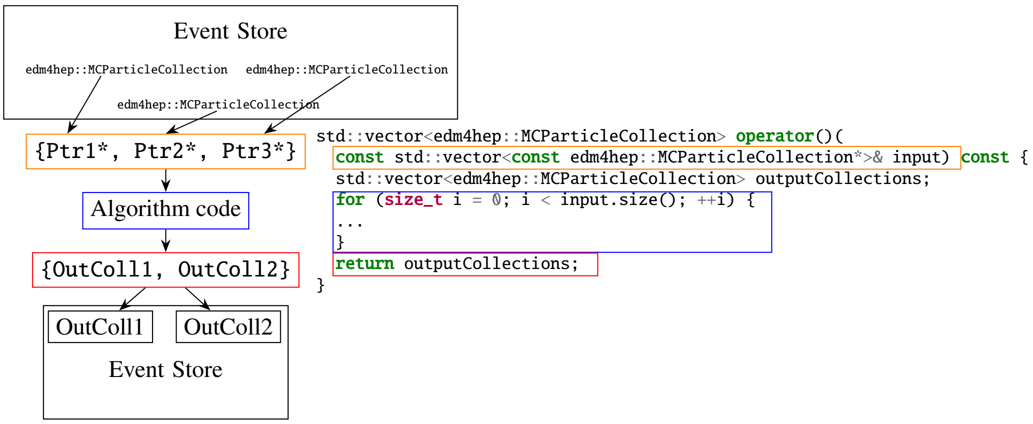
The handle-based algorithms were not designed with multithreading in mind. Since the `PodioDataSvc` does not implement the required interface in Gaudi for multithreading tools, a new data service was developed specifically to work with functional algorithms, enabling parallel processing capabilities.

2.3 Functional algorithms

Functional algorithms originate from Gaudi, which provides a set of base classes for algorithm development. Algorithm developers choose one of these classes and implement their algorithm by overriding the `operator()` method, see Figure 3. One feature of the algorithms is their stateless nature: each algorithm is implemented as a class where the main event loop, `operator()`, is declared `const`. This design enables algorithms to run efficiently in multithreaded environments, taking full advantage of Gaudi's parallel processing capabilities that were not accessible with traditional handle-based algorithms.

The implementation of functional algorithms in Key4hep required extensions beyond the original Gaudi design to accommodate specific framework needs. A key requirement in Key4hep is supporting generic algorithms that can work across different detectors, where aspects like the number of subdetectors remain unknown until runtime. A notable extension is the ability to handle an arbitrary number of collections, as some algorithms must process varying numbers of collections depending on the detector configuration (see Figure 4). To support this functionality, `k4FWCore` provides reimplemented versions of two fundamental algorithm types: transformers, which process existing collections and produce new ones, and consumers, which read and process existing collections. These implementations cover most use cases, with the flexibility to add more as needed. Algorithm developers utilize these enhanced classes instead of the original Gaudi implementations.

To enable file I/O operations with functional algorithms, a new service called `IOSvc` was developed, along with two specialized algorithms: a `Reader` that distributes collections



```
std::vector<edm4hep::MCParticleCollection> operator()(
  const std::vector<const edm4hep::MCParticleCollection*>& input) const {
  std::vector<edm4hep::MCParticleCollection> outputCollections;
  for (size_t i = 0; i < input.size(); ++i) {
  ...
  }
  return outputCollections;
}
```

Figure 4. Diagram showing the data flow between algorithms and the Event Store in a functional algorithm that takes as input and output an arbitrary number of collections. Pointers to three `edm4hep::MCParticleCollection` are taken from the store and put in a `std::vector` that the algorithm an process. Two new `edm4hep::MCParticleCollection` are produced, which are stored as a `std::vector` and put back in the store.

loaded by `IOSvc`, and a `Writer` that prepares collections in a `podio::Frame` for file storage. These components serve as functional equivalents to the `PodioInput` and `PodioOutput` used in data-handle based algorithms. The `IOSvc` leverages capabilities from podio to enable seamless switching of the output format to the new ROOT RNTuple format [12]. For input operations, `IOSvc` employs podio's generic reader, which automatically detects the correct format for the input file.

For metadata, a new service with the same behavior to the previous one was implemented to be able to read or write metadata easily with functional algorithms.

Compatibility between functional and handle-based algorithms was a crucial consideration during development. Through comprehensive testing in Key4hep, it has been demonstrated that both algorithm types can coexist within the same processing chain when using the new `IOSvc`, particularly when reading and writing EDM4hep collections to files. This compatibility ensures a smooth transition path for existing handle-based algorithm chains while enabling the adoption of functional algorithms for new developments.

2.3.1 *Examples of functional algorithms*

One example of a functional algorithm that is being used is the overlay algorithm. It has been ported from the original `OverlayTiming` algorithm from the Linear Collider software [13]. The Overlay algorithm takes as inputs signal and background files and merges the collections in them according to certain rules. For example, `edm4hep::MCParticles` are always merged for background and signal files, but `edm4hep::SimTrackerHit` will only be in the final result if they are within a configurable time window. There are multiple detector concepts in development in Key4hep, so this algorithm has to be able to work for any of them, meaning that it has to work for any number of subdetectors. The implementation uses the functionality to take as input or output any number of collections described above.

Another example is tracking, where the number of tracking subdetectors is not known until a detector is chosen at the time of running but combining all the hits in all the subdetectors is necessary to produce accurate tracks.

2.4 k4FWCore

`k4FWCore` is the repository where the services and interfaces between Gaudi and EDM4hep are implemented. The main components of `k4FWCore` are the `PodioDataSvc` and the `IOSvc`, together with accompanying algorithms for reading and writing collections.

2.4.1 k4FWCore components

In addition to the mandatory services for I/O and algorithm definition, `k4FWCore` also includes a set of other components that can be used by any algorithm:

- `UniqueIDGenSvc` is a service that produces a seed for random number generators to guarantee reproducibility. This is done by using quantities intrinsic to each event, such as a combination of the run number, event number, and also the name given to the algorithm in the steering file. These quantities are combined and hashed together, ensuring that the random number generator will produce the same values for the same event when running through the same algorithm, independent of factors like event ordering, thereby ensuring reproducibility.
- `EventHeaderCreator` is an algorithm that creates event headers and fills them with run numbers and event numbers.
- `CollectionMerger` is an algorithm that combines several EDM4hep collections into a single one.
- Interfaces: a common set of interfaces that algorithm developers can implement downstream for use across several repositories.

2.4.2 Running algorithms

`k4FWCore` provides a script for users to run algorithms: `k4run`. When given a *steering file*, `k4run` will show the different options that have been provided (or their default values) and will call Gaudi to run algorithms as specified in the steering file. In addition, `k4run` allows users to change parameters passed to algorithms and services from the command line, without having to edit any file.

2.4.3 Testing

`k4FWCore` has an extensive set of tests that make use of functional algorithms. The different algorithm types are tested, both in memory and reading and writing to files. There are also tests that process an arbitrary number of input or output collections, tests that run multithreaded and tests that combine handle-based and functional algorithms in the same chain. In addition, there are tests that check that the script that users need for running algorithms, `k4run`, behaves as specified.

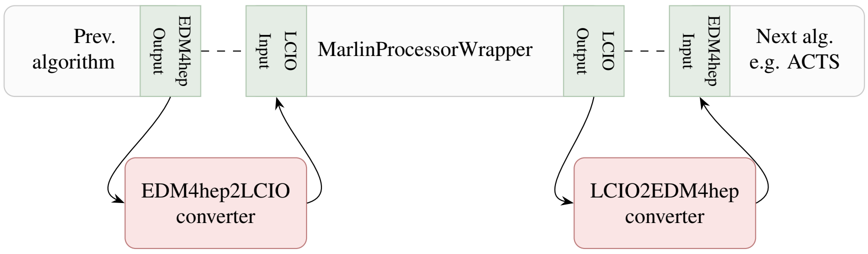


Figure 5. Diagram showing the data flow between algorithms when using the Marlin wrapper. The Marlin processor wrapper converts the data from EDM4hep to LCIO and back and vice versa.

2.5 The Marlin wrapper

One of the key components of Key4hep is the Marlin wrapper, available in the `k4MarlinWrapper` repository. This wrapper enables Marlin [14] processors (the event-processing framework in the Linear Collider software) to run within the Gaudi framework by integrating them as Gaudi algorithms. During processing, events are converted between `EDM4hep` and `LCIO` [15] (the event data model used by the Linear Collider community). These conversions occur dynamically based on whether the subsequent processing step uses a Gaudi algorithm or a Marlin processor (see Figure 5). However, this approach introduces performance overhead through both conversion time and increased memory usage, as events must be duplicated during the process.

Recent developments in the Marlin wrapper have expanded its capabilities to support functional algorithms and integration with `IOSvc`. However, the wrapper maintains its single-threaded execution model, as neither the wrapper nor the original Marlin processors were designed with multithreading capabilities.

3 Summary

Gaudi is the event-processing framework used by Key4hep to orchestrate running algorithms. Recently, Key4hep has developed a new service, `IOSvc`, to enable the use of functional algorithms that can run in a multithreaded environment, and at the same time be compatible with existing algorithms. Additional features, like being able to easily change to the new ROOT RNTuple format and enabling algorithms to take an arbitrary number of collections have been developed to fulfill the needs of Key4hep.

Future work will focus on encouraging a gradual migration to use `IOSvc` and functional algorithms for improved performance while deprecating legacy algorithms and components like the `PodioDataSvc`, and expanding the algorithm library with commonly needed reconstruction components.

Acknowledgements

This work benefited from support by the CERN Strategic R&D Programme on Technologies for Future Experiments (CERN-OPEN-2018-006). This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 101004761. This project has received funding from the European Union's Horizon 2020 Research and Innovation programme under Grant Agreement no. 871072. This work has been sponsored by the Wolfgang Gentner Programme of the German Federal Ministry of Education and Research (grant no. 13E18CHA).

References

[1] G. Barrand et al., GAUDI - A software architecture and framework for building HEP data processing applications, Comput. Phys. Commun. **140**, 45 (2001). 10.1016/S0010-4655(01)00254-5

[2] M. Frank, F. Gaede, M. Petric, A. Sailer, AIDASoft/DD4hep, https://doi.org/10.5281/zenodo.592244

[3] M. Frank, F. Gaede, C. Grefe, P. Mato, DD4hep: A Detector Description Toolkit for High Energy Physics Experiments, J. Phys. Conf. Ser. **513**, 022010 (2013). 10.1088/1742-6596/513/2/022010

[4] V. Volkl, T. Madlener, F. Gaede, A. Sailer, C. Helsens, P.F. Declara, G.A. Stewart, W. Deconinck, J. Smiesko, L. Forthomme et al., key4hep/EDM4hep, https://doi.org/10.5281/zenodo.4785062

[5] F. Gaede, G. Ganis, B. Hegner, C. Helsens, T. Madlener, A. Sailer, G.A. Stewart, V. Volkl, J. Wang, EDM4hep and podio - The event data model of the Key4hep project and its implementation, EPJ Web Conf. **251**, 03026 (2021). 10.1051/epjconf/202125103026

[6] F. Gaede, T. Madlener, P. Fernandez Declara, G. Ganis, B. Hegner, C. Helsens, A. Sailer, G. A. Stewart, V. Voelkl, EDM4hep - a common event data model for HEP experiments, PoS **ICHEP2022**, 1237 (2022). 10.22323/1.414.1237

[7] T. Madlener, EDM4hep - The common event data model for the Key4hep project, in *CHEP 2024* (Krakow, Poland, 2024), https://indico.cern.ch/event/1338689/contributions/6015945

[8] F. Gaede, B. Hegner, P. Mato, PODIO: An Event-Data-Model Toolkit for High Energy Physics Experiments, J. Phys. Conf. Ser. **898**, 072039 (2017). 10.1088/1742-6596/898/7/072039

[9] F. Gaede, B. Hegner, G.A. Stewart, PODIO: recent developments in the Plain Old Data EDM toolkit, EPJ Web Conf. **245**, 05024 (2020). 10.1051/epjconf/202024505024

[10] J. Carceller, Building the Key4hep Software Stack with Spack, in *CHEP 2024* (Krakow, Poland, 2024), https://indico.cern.ch/event/1338689/contributions/6010679

[11] D. Lawrence, EIC Software Overview, in *CHEP 2023* (Norfolk, Virginia, USA, 2023), https://indico.jlab.org/event/459/contributions/11457/

[12] J. Blomer, ROOT RNTuple: Next Generation Event Data I/O for HENP, in *CHEP 2024* (Krakow, Poland, 2024), https://indico.cern.ch/event/1338689/contributions/6005282/

[13] P. Schade, A. Lucaci-Timoce, Description of the signal and background event mixing as implemented in the Marlin processor OverlayTiming (2011), https://cds.cern.ch/record/1443537

[14] F. Gaede, Marlin and LCCD: Software tools for the ILC, Nucl. Inst. & Meth. **A559**, 177 (2006). 10.1016/j.nima.2005.11.138

[15] F. Gaede, T. Behnke, N. Graf, T. Johnson, LCIO: A Persistency framework for linear collider simulation studies, eConf **C0303241**, TUKT001 (2003), physics/0306114. 10.48550/arXiv.physics/0306114