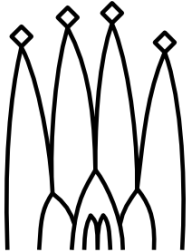
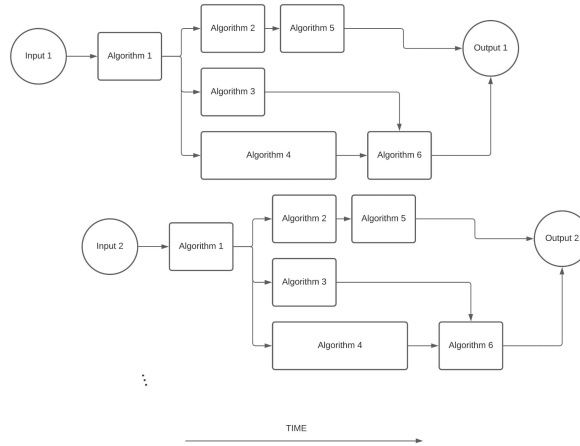
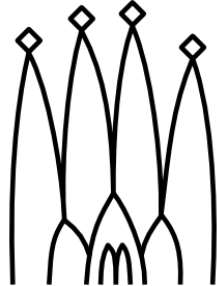


Gaudi as framework for the EIC Detector(s)



GAUDI





What is Gaudi?

Gaudi is a framework software package that is used to build data processing applications for High-Energy Physics experiments. It contains all of the components and interfaces to allow you to build event data processing frameworks for your experiment.

- Started in LHCb, co-developed by LHCb and ATLAS
- Used for many current and future experiments: ATLAS, BESIII, Daya Bay, FCC, Fermi, HARP, LHCb, LBNE, LZ, and MINERvA, core part of the Key4hep stack.
- We used Gaudi for the ATHENA proposal.
- Core principles:
 - Separation between data and algorithms
 - Well defined interfaces
 - Reusable components.
- Well-supported and actively developed
- Recently modernized for multi-threaded computing and heterogeneous computing environments, leveraging modern C++ features.



Components to Gaudi

Application Manager

Top-level application that initializes the framework and starts the processing loop

Scheduler and Loop Manager

Modular components that schedule the algorithm execution tree and actual processing loop

Transient Event Store (TES)

Whiteboard to communicate data between algorithms. Each entry can be written once and read many times.

Job Options

Configuration of *everything* works through a light-weight python script

Services

Provide unique resources to the algorithms. E.g. file storage, geometry service, limited hardware resources (GPU,...), messaging, ...

Algorithms

Functionally pure routines that computes M outputs from N inputs. Called for each data chunk (event/frame/...). Inherently thread-safe.

Tools

Chunks of code shared between algorithms. Typically not preferred, as concurrent execution favors more small algorithms over complicated algorithms with many tools.

Ultimate flexibility: *Everything* is a plugin that can be swapped out at runtime. That includes the TES, scheduler, loop manager, ...

Modern (functional) Gaudi algorithms

Out \ In	0	1 - n	vect
0		Consumer	
1	Producer	Transformer	Merging Transformer
n		MultiTransformer	
vect		SplittingTransformer	
boolean		FilterPredicate	
boolean + 1-n		MultiTransformerFilter	

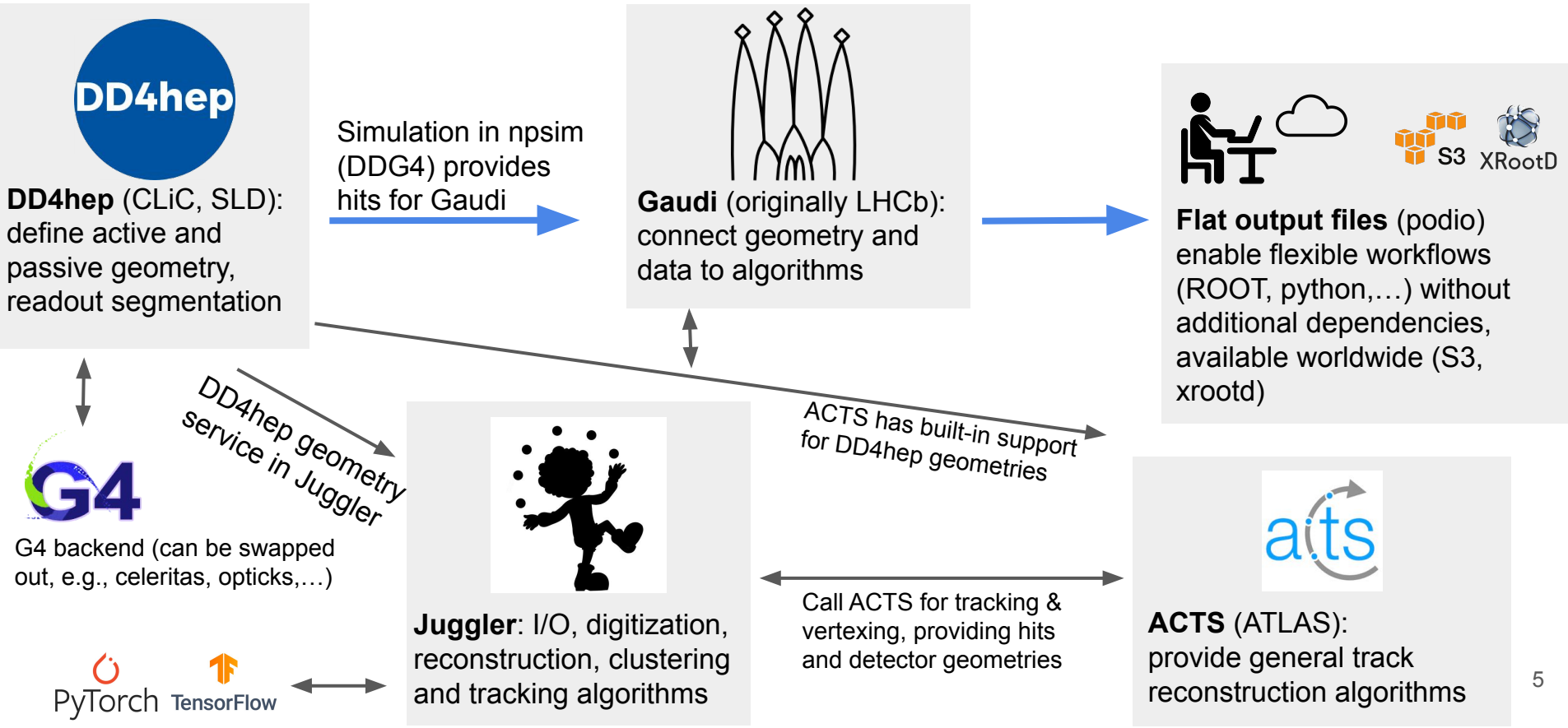
Functional algorithms remove most boilerplate of explicit algorithm writing, and are automatically suited for concurrent processing.

Functional algorithms are explicitly re-entrant.

Can of course also manually write algorithms (or new functional algorithm types), or (rarely) write stateful algorithms with explicit cardinality for concurrent execution.

Example: ATHENA software stack

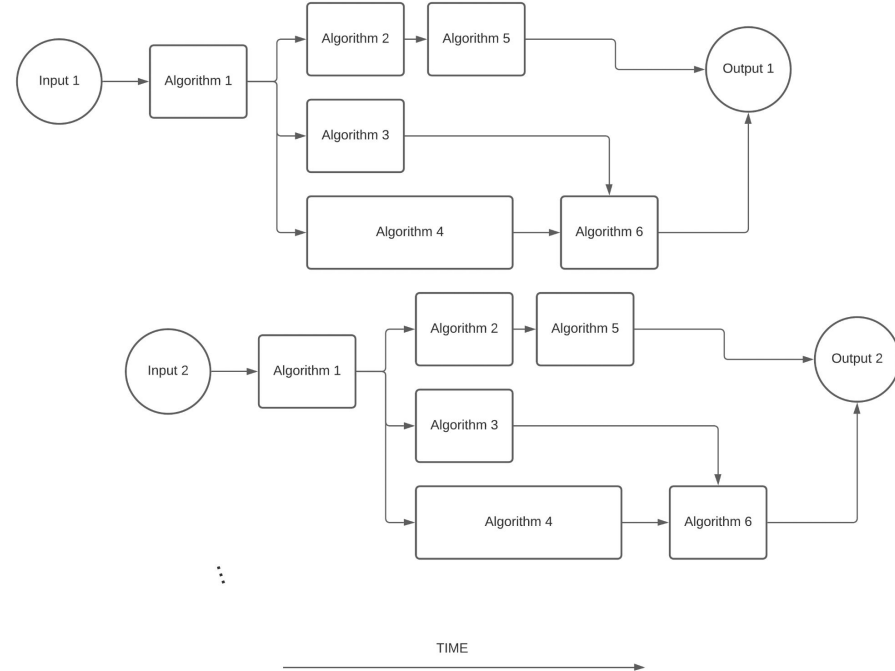
How does Gaudi fit into the EIC software stack?



Gaudi enables highly flexible concurrent workflows



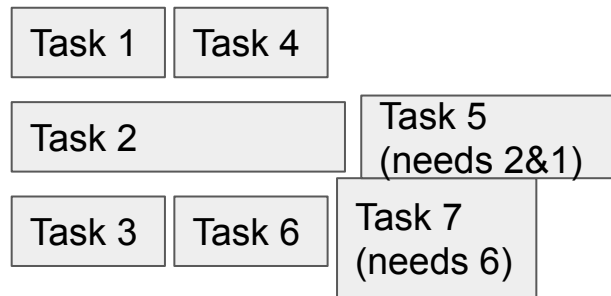
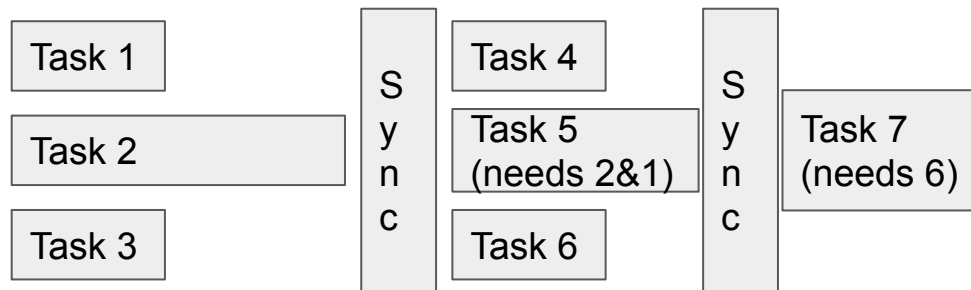
- Juggler is a collection of Gaudi framework components developed for the ATHENA proposal.
- All Juggler algorithms reentrant by design
 - Support concurrent processing, heterogeneous environments from the beginning
 - Highly modular
 - Easy to integrate with external toolkits (ACTS, tensorflow, ...)
 - *Reentrant algorithms easy to write, debug, validate, and compose, even by beginners!*
- Reconstruction: steered through a simple python script → trivial to reconfigure reconstruction for different detector layouts, or for subsystem-only reconstructions
 - Algorithms designed to be as small as possible to allow easy composition/substitution/optimization
 - Even true for the event scheduler! Concurrency in Gaudi enabled by swapping out the event scheduler (can use concurrent, parallel, single-threaded, ...).



Why is concurrent (asynchronous) processing important?

Concurrent versus parallel processing

- Concurrent execution significantly faster than parallel execution for complicated algorithm flows, as we can avoid idle cores.
- Avoids hard limits from **Amdahl's law** (speed ceiling for code limited by the fraction of non-parallel code (e.g. synchronization points)).



$$Speedup = \frac{1}{(1 - p) + p/N}$$

- p : proportion of the code that is parallel
- N : number of cores

How does GAUDI address the framework requirements?

Requirements [[link](#)]



- Must be able to run on **both simulated events and real data**. Algorithms may use truth information, but must run without truth information too.



- Must be able to take advantage of **heterogeneous computing resources** (multiple cores, GPUs, etc).



- Must **encourage modular approaches** to algorithm development, using defined interface layers.



- Algorithms must be implemented using the selected **data model**, and ensure that data are kept separate from the algorithm itself.



- Algorithms must be implemented in the framework **independently from any scheduling strategies** (single/multithreaded, concurrent, online/offline).

Align perfectly with the core design goals for Gaudi!

How does GAUDI address the framework requirements?

Requirements [[link](#)]



- Must be open source, accessible to the entire **community**, and managed by a sustainable core team.



- Must be able to pass (and add) **metadata** and so-called slow control information to the output it produces, so input files are not needed and output files can stand on their own.



- Must be able to run in **streaming readout** mode, that is:
 - with access to only parts of an event (single detector, single sector),
 - with events (or parts of events) appearing out of sequence,
 - individual algorithms must not rely on an algorithm-specific internal state to be able to make sense of disconnected parts of events.

(*) Used for SR and HLT at LHCb, being deployed together with Allen for Run 3

How does it fit within the additional criteria?

- **Amount of ‘boilerplate’ code** that must be written by algorithm developers.
- Ability by the framework to **avoid e.g. memory errors** through interface enforcement mechanisms (e.g. const passing).
- Ability for **shared algorithm development** between the two EIC detector collaborations (and/or outside of the EIC).
- Use of **modern and sustainable coding practices**, including in the code written by algorithm developers and other contributors.
- **Demonstration of performance** in production environments.

Very minimal, especially for functional algorithms (see example)

Baked into the TES (data is read-only). Even further improved through reentrant algorithms.

Perfect for sharing, can use and share algorithms with key4hep and potentially even LHC experiments.

Gaudi modernization project ensured modern C++ and modern design practices (required to support concurrent and heterogeneous computing!)

Demonstrated performance at many ongoing and future experiments (ATLAS, BESIII, Daya Bay, FCC, Fermi, HARP, LHCb, LBNE, LZ, and MINERvA)

An example of using Gaudi

Code Demonstration

Both proposals were asked to address the following (by the second week):

Provide the full code that would be required for a simple algorithm that takes a collection of hits and selects only those hits that are on a particular endcap tracking detector and have a position outside a minimum radial range.

Submissions will be posted on indico by July 6 for community evaluation.

```
1 class RadialHitSelector // The name of our algorithm
2   : public Gaudi::Functional::Transformer< // It's a Transformer
3     std::vector<bool> // It returns a vector of boolean
4     (const edm4hep::TrackerHitCollection&)> // It takes TrackerHits as
5     // input
```

Using modern functional classes. Here we transform a set of hits into a corresponding set of boolean flags.

```
7 private:
8   // We have a configurable radius that we want to use for hit selection
9   Gaudi::Property<double> m_rmin_cut{
10     this,
11     "minRadius", // Name of variable in the options file
12     0.,          // Default value in case it is not set
13     "Minimum radius below which hits will not be accepted" // help
14   };
```

We need a radial cut for our demonstration. Here we make it explicitly configurable (and well-documented) for the options file.

```

15
16 public:
17     RadialHitSelector(const std::string& name, // Algorithm name, set in options
18                     ISvcLocator* pSvcLocator)
19         : // Boilerplate to refer to the service locator
20           Transformer{
21             name, pSvcLocacator,
22             KeyValue{
23                 "inputHits", // Set the input hit collection in the options file
24                 "TrackerEndcapPHits"}, // Default collection name
25             KeyValue{
26                 "outputFlags", // Set the output flag collection in the options
27                 "TrackerEndcapPHitsFlag"}} {} // Default output name
28

```

Let's construct our algorithm. Most of the code here just defines the configuration variables for the options file, and default arguments.

```

29 // The actual algorithm that gets called for every chunk of data
30 // (Event/Frame/...)
31 std::vector<bool> // output will be automatically put on the TES
32 operator()(
33     const edm4hep::TrackerHitCollection& hits // input is explicitly const
34                                             // as we cannot modify objects
35                                             // on the TES.
36 ) const override // The actual algorithm is const, which makes concurrent
37                 // processing trivial (race conditions not possible!)
38 {
39     std::vector<bool> passes_radial_cut;
40     for (const auto& hit : hits) {
41         passes_radial_cut.push_back(
42             std::hypot(hit.getPosition().x, hit.getPosition().y) > m_rmin_cut);
43     }
44     return passes_radial_cut;
45 }

```

And the actual algorithm. The function signature defines what gets read (read-only!) and what gets written into the TES.

Corresponding options file

```
1 from Gaudi.Configuration import *
2
3 from Configurables import ApplicationMgr, EICDataSvc, PodioInput, PodioOutput
4 from Configurables import RadialHitSelector ## our algorithm
5 from GaudiKernel.SystemOfUnits import mm
6
```

Import the stuff we will need:

- Application manager
- Data service of choice - let's use our EIC/Podio service for this example!
- Our example algorithm
- Units, so it is clear what our radial cuts really mean.

```
0
1
2 ## Minimal list of services we need for this example. We only
3 ## need a data service that reads our Podio data structures
4 services = []
5 services.append(EICDataSvc('EventDataSvc', inputs=['input.root'], OutputLevel=WARNING))
6
7 ## in a realistic example we would probably also need a geometry service, conditions
8 ## service, auditor service, ...
9
10
```

Here we only need a data service, which needs to know what input file we are reading from.


```
5 ## input branches to read in the PodioInput
6 input_branches = ['TrackerEndcapNHits'] ## only need the negative endcap tracker hits
7
8 ## list of top-level algorithms
9 algorithms = []
10
11 ## algorithm to fetch our input
12 input_algo = PodioInput('PodioReader', collections=input_branches)
13 algorithms.append(input_algo)
14
15 ## our example algorithm
16 main_algo = RadialHitSelector('ExampleAlgo',
17     inputHits='TrackerEndcapNHits',
18     outputFlags='TrackerEndcapNHitsGood',
19     minRadius=100*mm)
20 algorithms.append(main_algo)
21
22 ## and our output writer
23 output_algo = PodioOutput('PodioWriter', filename='myexample.root')
24 ## All data is available on the TES. Specify what we want to write to the output file
25 output_algo.outputCommands = ['keep *'] ## for this example we can keep everything
26 algorithms.append(output_algo)
```

Three algorithms as we need to do 3 things: read input, execute RadialHitSelector, and write output

```
37
38 ApplicationMgr(
39     TopAlg = algorithms,
40     EvtSel = 'NONE',
41     EvtMax = 100, ## lets process up to 100 chunks
42     ExtSvc = services,
43     OutputLevel = WARNING
44     AuditAlgorithms = False ## optionally audit the algorithm performance,
45                             ## can check for e.g. time spent, or memory usage
46                             ## at the algorithm level
47 )
```

Only thing left is to setup the application, and we are ready for execution!

Can hit the ground running!

Multiple FTEs worth of components available today

Tracking

- ✓ Tracker digitization with noise
- ✓ Realistic tracker hit reconstruction
- ✓ Truth and realistic track seeding (ACTS)
- ✓ Combinatorial Kalman Filter (finding/fitting) from ACTS
- ✓ Far-forward matrix-based track reconstruction (RP/OMD)
- ✓ Initial vertex finder (ACTS)
- ✓ Arbitrary track projection
- ✓ Reconstruction-generation matching

Services

- ✓ (All standard Gaudi services)
- ✓ (All Key4hep services)
- ✓ Custom DD4hep geometry service
- ✓ Custom Podio data service
- ✓ Custom Acts geometry service
- ✓ Particle service

PID

- ✓ RICH reconstruction using the libirt backend
- ✓ Fuzzy-K clustering algorithm
- ✓ Mock truth PID reconstruction



Calorimetry

- ✓ Calorimeter digitization with noise/threshold effects
- ✓ Calorimetry hit merging
- ✓ Realistic calorimeter hit reconstruction
- ✓ Many clustering algorithms: Simple clustering, Island clustering, 2+1D clustering, Topological 3D clustering
- ✓ Hybrid cluster merging
- ✓ AI-based electron-pion-muon separation (leveraging TFLite)
- ✓ Simple track-cluster association

Global

- ✓ Multiple event builders
- ✓ Multiple algorithms for DIS (electron, JB, DA, Sigma)
- ✓ Have momentum: Many more algorithms being actively developed!


Gaudi and the EIC Software Statement of Principles

Used in streaming context, e.g. for HLT at LHCb, and no together with Allen for the DAQ for LHCb run-3

Demonstrated performance in heterogeneous environments at LHC. Concurrent processing model, and flexibility of schedulers (single-process, multi-process, concurrent, parallel, ...) make Gaudi a perfect fit for future changing environments.

Large community already trained in Gaudi, Development paradigm is supremely modular where even schedulers or the data store can be swapped.

EIC SOFTWARE: Statement of Principles

- 
- 1 We aim to develop a diverse workforce, while also cultivating an environment of equity and inclusivity as well as a culture of belonging.**
 - 2 We will have an unprecedented compute-detector integration:**
 - We will have a common software stack for online and offline software, including the processing of streamed data and its time-ordered structure.
 - We aim for autonomous alignment and calibration.
 - We aim for a rapid, near-real-time turnaround of the raw data to online and offline productions.
 - 3 We will leverage heterogeneous computing:**
 - We will enable distributed workflows on the computing resources of the worldwide EIC community, leveraging not only HTC but also HPC systems.
 - EIC software should be able to run on as many systems as possible, while supporting specific system characteristics, e.g., accelerators such as GPUs, where beneficial.
 - We will have a modular software design with structures robust against changes in the computing environment so that changes in underlying code can be handled without an entire overhaul of the structure.
 - 4 We will aim for user-centered design:**
 - We will enable scientists of all levels worldwide to actively participate in the science program of the EIC, keeping the barriers low for smaller teams.
 - EIC software will run on the systems used by the community, easily.
 - We aim for a modular development paradigm for algorithms and tools without the need for users to interface with the entire software environment.

Gaudi and the EIC Software Statement of Principles



5 Our data formats are open, simple and self-descriptive:

- We will favor simple flat data structures and formats to encourage collaboration with computer, data, and other scientists outside of NP and HEP.
- We aim for access to the EIC data to be simple and straightforward.

6 We will have reproducible software:

- Data and analysis preservation will be an integral part of EIC software and the workflows of the community.
- We aim for fully reproducible analyses that are based on reusable software and are amenable to adjustments and new interpretations.

7 We will embrace our community:

- EIC software will be open source with attribution to its contributors.
- We will use publicly available productivity tools.
- EIC software will be accessible by the whole community.
- We will ensure that mission critical software components are not dependent on the expertise of a single developer, but managed and maintained by a core group.
- We will not reinvent the wheel but rather aim to build on and extend existing efforts in the wider scientific community.
- We will support the community with active training and support sessions where experienced software developers and users interact with new users.
- We will support the careers of scientists who dedicate their time and effort towards software development.

8 We will provide a production-ready software stack throughout the development:

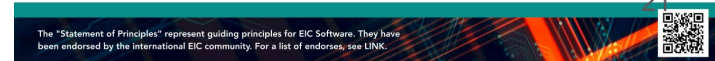
- We will not separate software development from software use and support.
- We are committed to providing a software stack for EIC science that continuously evolves and can be used to achieve all EIC milestones.
- We will deploy metrics to evaluate and improve the quality of our software.
- We aim to continuously evaluate, adapt/develop, validate, and integrate new software, workflow, and computing practices.

Structure of C++ plugin with python configuration leans itself well to reproducibility.

Gaudi is *sustainable*: backed by large development team (not the few developers typical of NP software), and demonstrated reliability (used by many experiments, not all at CERN).

Gaudi supports state-of-the-art modern software and computing paradigms. No need to invest NP resources to achieve this. If any framework comes close to being a community standard, Gaudi is it.

Gaudi has been used for over 20 years for running experiments. We have working digi/reconstruction for EIC in Gaudi *today*.



Gaudi for EIC?

- Large amount of reconstruction algorithms available *today*
- Part of the collaboration already trained in Gaudi, many experts available at member institutions.
- Synergy with Key4hep will allow for common shared development with HEP experiments and first-class support. Could adopt the K4FWCore as basis for our EIC framework for seamless integration with Podio and DD4hep.
- Demonstrated performance for very different experimental needs - used by 10+ high-profile experiments! (and for the EIC detector proposal).
- Idea of algorithm sharing between experiments very attractive.
- **Gaudi hits all the marks for our needs, is well established yet modern, and in my opinion the optimal choice for EIC framework.**

