algorithms

The case for framework- and experiment-independent algorithms at ~~EPIC~~ ePIC

Sylvester Joosten
on behalf of CompSW & SimQA

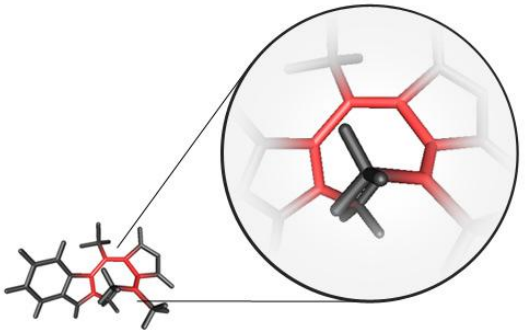
ePIC Collaboration Meeting
January 9, 2023

Let’s take a second to appreciate how far we’ve come!

- Finished Software Choice process by early August
- Went through rigorous Software Review in September
- Migrated the software from proposal period into our common Software Stack in only a few months
    - Including the implementation of fully new reconstruction framework code (EICRecon)
- This presentation will propose a path forward, but first want to fully acknowledge where we are right now!

How do we go from here? Zoom in on our weak points

- Had to “cut some corners” to get where we are in the time we had
  - Framework code rigid and tending towards the monolithic
  - Overly reliant on C++ for all aspects of the reconstruction
  - Overly centralized decision process
  - Unsustainable responsibility load for core developers requiring involvement levels (high risk of burn-out)
  - Design (out of necessity) by a small group under extreme time pressure
    - No time for holistic design inside the software stack
    - No time to engage Users in the design process
    - No time to focus on forward sustainability, …
- Not a knock on the software choices or the work performed! I view what we accomplished in the last few months as a great success!
- Consider the current EICRecon as a Software Prototyping stage
  - Strong proof of concept - it *can* be done with the current technologies!
  - Learned valuable lessons early to map our path forward for the decades to come



The case for “Hardcore” modularism

Starting from the EIC Software Statement of Principles

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.

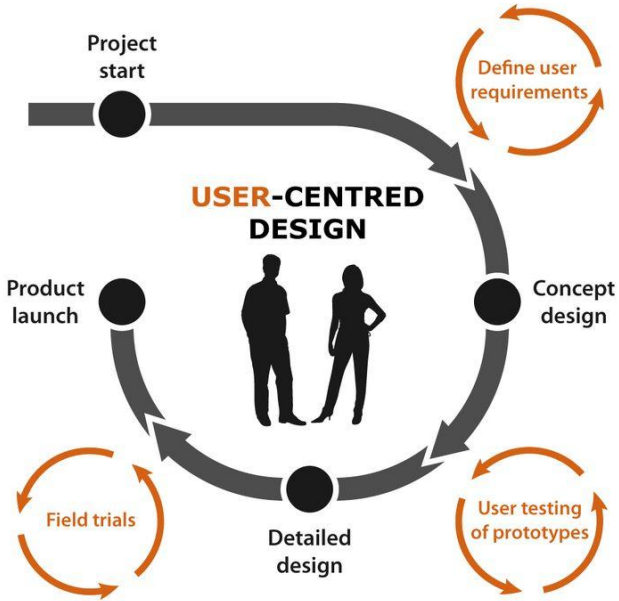
4 We will aim for user-centered design:

- 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.

- Need more rigorous separation of different domains:
  - Framework
  - Algorithms
  - Configuration
  - Resources
  - User workflow
  - …
- This will enhance user experience, improve maintainability, increase flexibility against future changes, reduce scope of developer responsibility (everyone is the ruler of their own realm)

User centered design

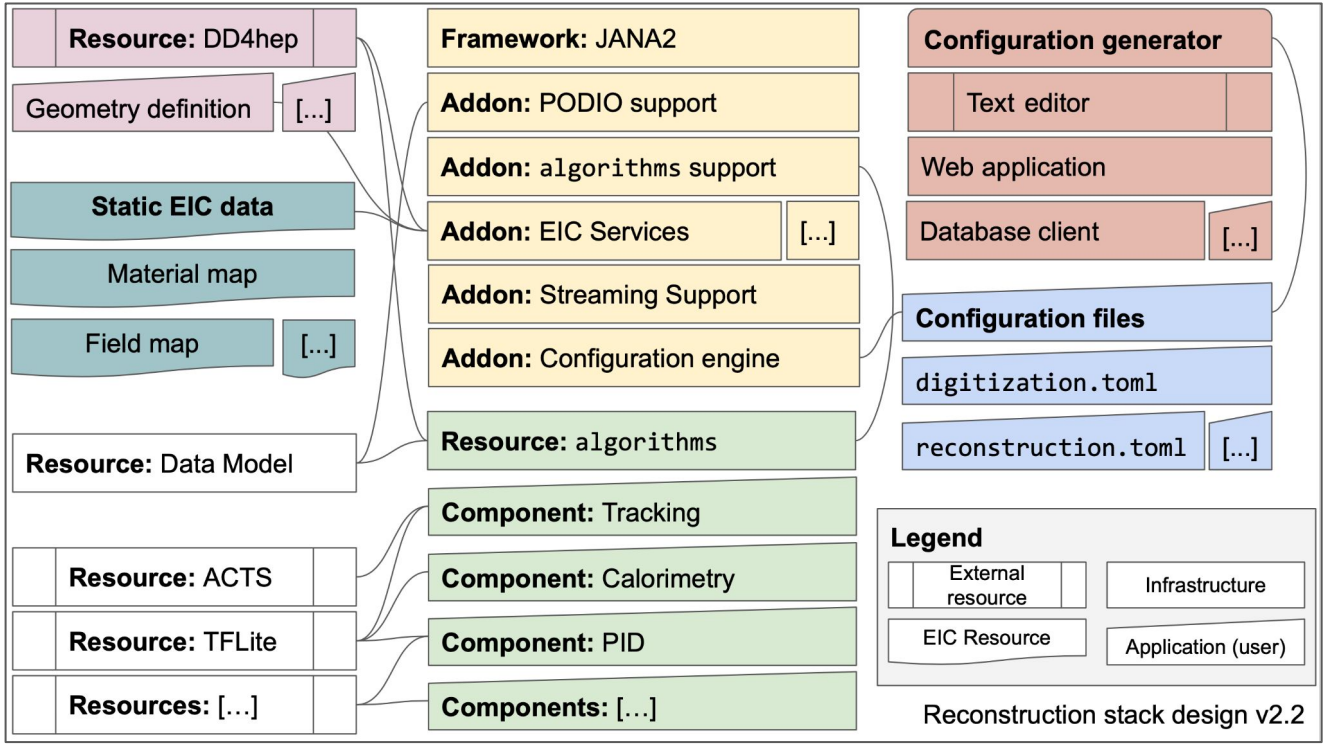
- Need to support workflows actually needed by the Users
  - Create, test, and run a new reconstruction algorithm with minimal work, support new stand-alone plugins with minimal friction
  - Evaluate changes in geometry by changing only the geometry definition and relevant configuration file (no need to change/recompile *everything*) - again, minimize friction
  - Get reproducible (and easily altered) reconstruction configurations without needing to do any additional work (zero-friction reproducibility)
  - Provide domains of responsibility where Users of all experience levels can make meaningful contributions
  - Distinct domains of responsibility also make clear *who* to talk to, no more single persons supporting everything at once.



Bottom-line: need to revisit design choices based on user requirements and real-world experience.

Evolving of the EPIC reconstruction stack design

- Strictly modular approach reduces scope of each component
- Easier to onboard new users in any singular piece of the stack
- Every user can find their place based on experience and needs
- Better maintainability and more resilient against changing software needs
- Baked-in reproducibility by enforcing configuration files in every workflow



Strawman approach ticks quite some boxes

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.

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.

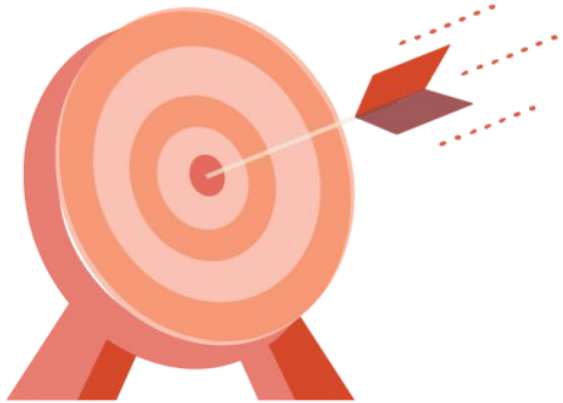
6. **We will have reproducible software:**
   - Data and analysis preservation will be an integral part of EIC software and the workflows of the community.
   - We aim for fully reproducible analyses that are based on reusable software and are amenable to adjustments and new interpretations.

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.

Why generic algorithms? What are the design goals?

- Enable algorithm sharing across experiments and even communities.
  - ACTS illustrates that this can be highly successful
- Framework-agnostic algorithms reduce scope and requirements of what Users (algorithm writers) need to deal with - lower barrier of entry
- Software stack already has the required interfaces to facilitate this - EDM4hep/EDM4eic data model and DD4hep geometries
- Can minimize the boilerplate by taking out explicit framework responsibilities - reduced friction for the Users



That sounds nice in theory, but is this even possible?

- **… Yes!** As a matter of fact, we have had a working prototype for `algorithms` for months!
  - Standalone prototype library: https://github.com/eic/algorithms (documentation coming once API design complete)
  - Has been part of the Juggler reconstruction flow for almost half a year
  - JANA2 integration coming soon (February 2 CompSW+SimQA meeting)
    - API design considered complete once successfully integrated with two frameworks

What does an `algorithms` algorithm look like?

```cpp
using ClusteringAlgorithm = Algorithm<
    Input<edm4eic::ProtoClusterCollection, std::optional<edm4hep::SimCalorimeterHitCollection>>,
    Output<edm4eic::ClusterCollection,
           std::optional<edm4eic::MCRecoClusterParticleAssociationCollection>>>;

/** Clustering with center of gravity method.
 *
 *  Reconstruct the cluster with Center of Gravity method
 *  Logarithmic weighting is used for mimicking energy deposit in transverse direction
 *
 * \ingroup reco
 */
class ClusterRecoCoG : public ClusteringAlgorithm {
public:
  using WeightFunc = std::function<double(double, double, double)>;

  // TODO: get rid of "Collection" in names
  ClusterRecoCoG(std::string_view name)
      : ClusteringAlgorithm{name,
                            {"inputProtoClusterCollection", "mcHits"},
                            {"outputClusterCollection", "outputAssociations"},
                            "Reconstruct a cluster with the Center of Gravity method. For "
                            "simulation results it optionally creates a Cluster <-> MCParticle "
                            "association provided both optional arguments are provided."} {}

  void init() final;
  void process(const Input&, const Output&) const final;

private:
  edm4eic::MutableCluster reconstruct(const edm4eic::ProtoCluster&) const;

  // TODO FIXME does the sampling fraction belong here or in the hit reconstruction?
  Property<double> m_sampFrac{this, "samplingFraction", 1.0, "Sampling fraction"};
  Property<double> m_logWeightBase{this, "logWeightBase", 3.6, "Weight base for log weighting"};
  Property<std::string> m_energyWeight{this, "energyWeight", "log", "Default hit weight method"};
  Property<std::string> m_moduleDimZName{this, "moduleDimZName", "", "z-dim name of the module"};
  // Constrain the cluster position eta to be within
  // the eta of the contributing hits. This is useful to avoid edge effects
  // for endcaps.
  Property<bool> m_enableEtaBounds{this, "enableEtaBounds", true, "Constrain cluster to hit eta?"};

  WeightFunc m_weightFunc;

  const GeoSvc& m_geo = GeoSvc::instance();
};
```

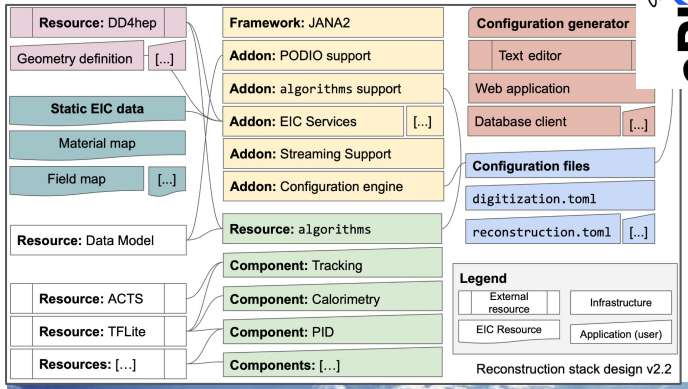
- Define limited user functions (init and process)
- Fully declarative in nature:
  - Algorithms signature defined once, automatically drives data store interactions at framework side
  - Properties defined once with as one-liners, drives the configuration setup at the framework side.
  - Documentation fields required in all cases

Bottom line - No repetition:

User defines everything *only once*

Proposed path forward?

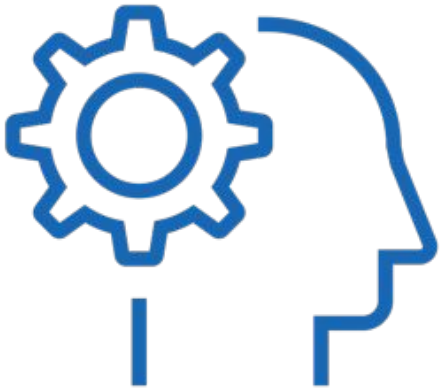
- Ensure a continued stable software stack, need to support EICRecon while we prepare an alternative route
- Prepare a full prototype of `algorithms` by February 2 software meeting (with full JANA2 integration)
- Crystalize the different realms of our reconstruction stack, identify key persons to manage each realm.
  - In particular, identify technological solutions to be implemented
- Start seamless migration (cannot impact operations) starting February 2023



Thank you!

What are the challenges for truly generic algorithms?

- Providing framework functionality while being a thin layer on top of multiple frameworks non-trivial
  - What to do with services? Context? Data store interactions? Properties and configuration?
  - Need to avoid duplication of definitions
  - How to minimize boilerplate (zero-line algorithm integration)?
  - Need showcase in multiple frameworks (JANA2 and Gaudi)
- But… it doesn't make sense to separate all algorithms, what about fine-tuned capabilities for e.g. DAQ
  - Correct, not everything should be a generic algorithm.
  - But most code (80-90%) could be, and I argue that the Users will greatly benefit from this.



What does an `algorithms` algorithm look like (2/2)?

```cpp
using ClusteringAlgorithm = Algorithm<
    Input<edm4eic::ProtoClusterCollection, std::optional<edm4hep::SimCalorimeterHitCollection>>,
    Output<edm4eic::ClusterCollection,
           std::optional<edm4eic::MCRecoClusterParticleAssociationCollection>>>;

/** Clustering with center of gravity method
 *
 *  Reconstruct the cluster with Center of Gravity method
 *  Logarithmic weighting is used for mimicking energy deposit in transverse direction
 *
 * \ingroup reco
 */
class ClusterRecoCoG : public ClusteringAlgorithm {
```

Supports definition of *standard* (required) Collections, *optional* Collections, and *vectors* of Collections (e.g. hits from different detectors)

```cpp
void ClusterRecoCoG::process(const ClusterRecoCoG::Input& input,
                             const ClusterRecoCoG::Output& output) const {
  const auto [proto, opt_simhits] = input;
  auto [clusters, opt_assoc]      = output;

  for (const auto& pcl : *proto) {
    auto cl = reconstruct(pcl);

    if (aboveDebugThreshold()) {
      debug() << cl.getNhits() << " hits: " << cl.getEnergy() / dd4hep::GeV << " GeV, ("
              << cl.getPosition().x / dd4hep::mm << ", " << cl.getPosition().y / dd4hep::mm << ", "
              << cl.getPosition().z / dd4hep::mm << ")" << endmsg;
    }
    clusters->push_back(cl);
```

No explicit data store interactions, we get pointers to data collections managed by the framework that are *guaranteed* to be valid.

How do we integrate services?

```cpp
// Thread-safe lazy-evaluated minimal service system
// CRTP base class to add the instance method
// This could have been part of DEFINE_SERVICE macro, but I think it is better
// to keep the macro magic to a minimum to maximize transparency
template <class SvcType> class Service : public PropertyMixin, public NameMixin {
public:
  static SvcType& instance() {
    // This is guaranteed to be thread-safe from C++11 onwards.
    static SvcType svc;
    return svc;
  }
  // constructor for the service base class registers the service, except
  // for the ServiceSvc which is its own thing (avoid circularity)
  Service(std::string_view name) : NameMixin{name} { ServiceSvc::instance().add(name, this); }
};

} // namespace algorithms
```

```cpp
// Note: the log action is responsible for dealing with concurrent calls
//       the default LogAction is a thread-safe example
class LogSvc : public Service<LogSvc> {
public:
  using LogAction = std::function<void(LogLevel, std::string_view, std::string_view)>;
  void defaultLevel(const LogLevel l) { m_level.set(l); }
  LogLevel defaultLevel() const { return m_level; }
  void action(LogAction a) { m_action = a; }
  void report(const LogLevel l, std::string_view caller, std::string_view msg) const {
    m_action(l, caller, msg);
  }

private:
  Property<LogLevel> m_level{this, "defaultLevel", LogLevel::kInfo};
  LogAction m_action = [](const LogLevel l, std::string_view caller, std::string_view msg) {
    static std::mutex m;
    std::lock_guard<std::mutex> lock(m);
    fmt::print("{} [{}] {}\n", logLevelName(l), caller, msg);
  };
  ALGORITHMS_DEFINE_SERVICE(LogSvc)
};
```

- Services as lazy-evaluated singletons
- Support standalone minimal interface
  - Interface has usable defaults for standalone operation
  - Standalone defaults are meant to be overridden by the framework by defining callbacks
- Prototype currently implements LogSvc, GeoSvc, and RandomSvc
- Special ServiceSvc provides framework with all required services, so it can handle the bindings

What about Properties?

```
private:
  edm4eic::MutableCluster reconstruct(const edm4eic::ProtoCluster&) cor

  Property<double> m_sampFrac{this, "samplingFraction", 1.0};
  Property<double> m_logWeightBase{this, "logWeightBase", 3.6};
  Property<std::string> m_energyWeight{this, "energyWeight", "log"};
  Property<std::string> m_moduleDimZName{this, "moduleDimZName", ""};
  Property<bool> m_enableEtaBounds{this, "enableEtaBounds", true};

  WeightFunc m_weightFunc;

  const GeoSvc& m_geo = GeoSvc::instance();
};
} // namespace algorithms::calorimetry
```

- Need a way to define properties for algorithms
- Ideally they should provide for a programatic way to deal with automatic initialization at the framework end (non-trivial)
- Currently choose a Gaudi-like Property<T> class that has run-time performance of a bare T, while providing an avenue for the framework to set the property
- Automatic handling possible through a visitor pattern (framework side works automagically!)

Open challenges

- Self-announcing algorithms (so the framework can query on plugin load what algorithms are available)
- Rigorous context management (API already defined)
- JANA2 integration and re-evalutation of service API to properly serve both JANA2 and Gaudi
- Finish porting the rest of the Juggler algorithms (`algorithms` shares a history with Juggler so retains full git history!)

This is a short list - can have this in the next few weeks!