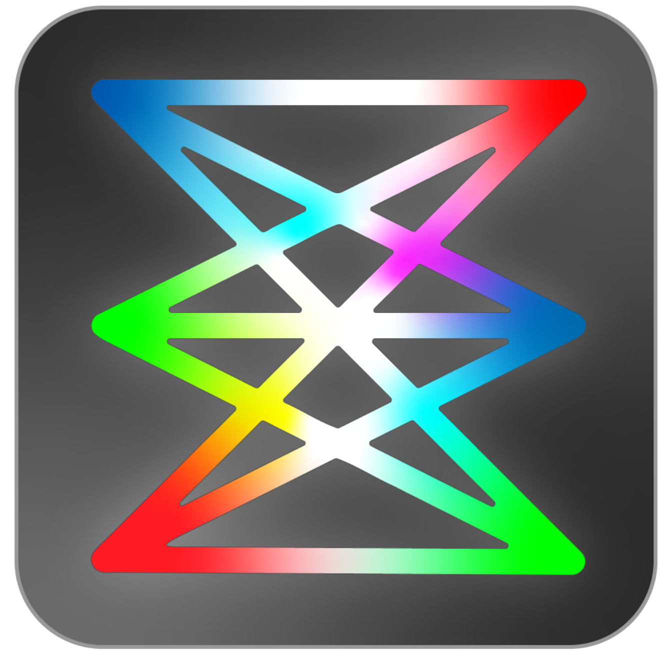
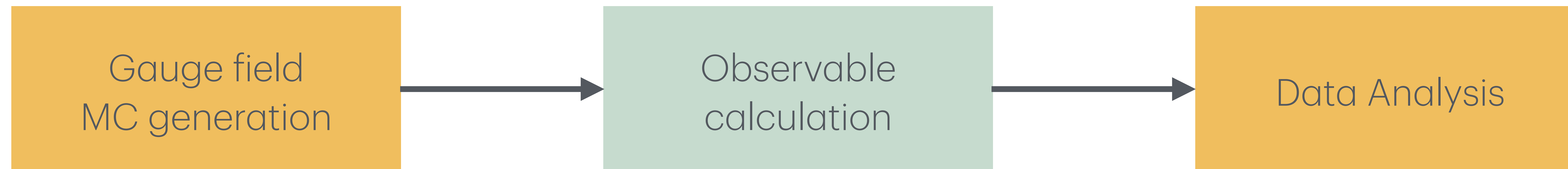


Hadrons

General introduction

Antonin Portelli (U of Edinburgh) — Hadrons User Workshop — 14 April 2026, Edinburgh, UK

High-level lattice QCD workflow



- Gauge field generation: MCMC sampler, on supercomputer, using HPC lattice library
- Observable calculation: process gauge fields into observable samples, typically QFT correlation functions; on supercomputer
- Data analysis: statistical data analysis of observables, on laptop/workstation
- **Hadrons aims at solving the observable calculation step**

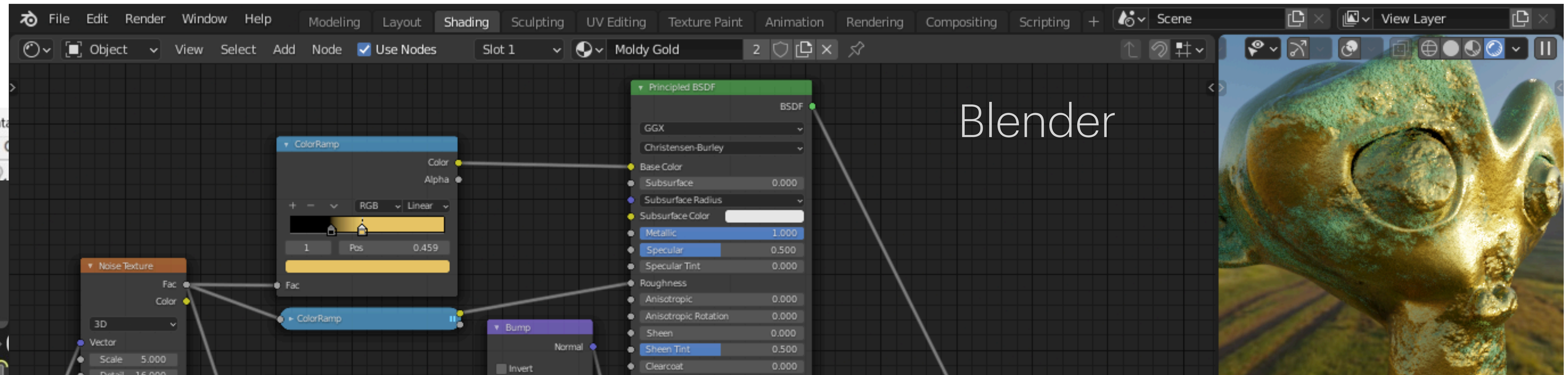
Main problems to solve

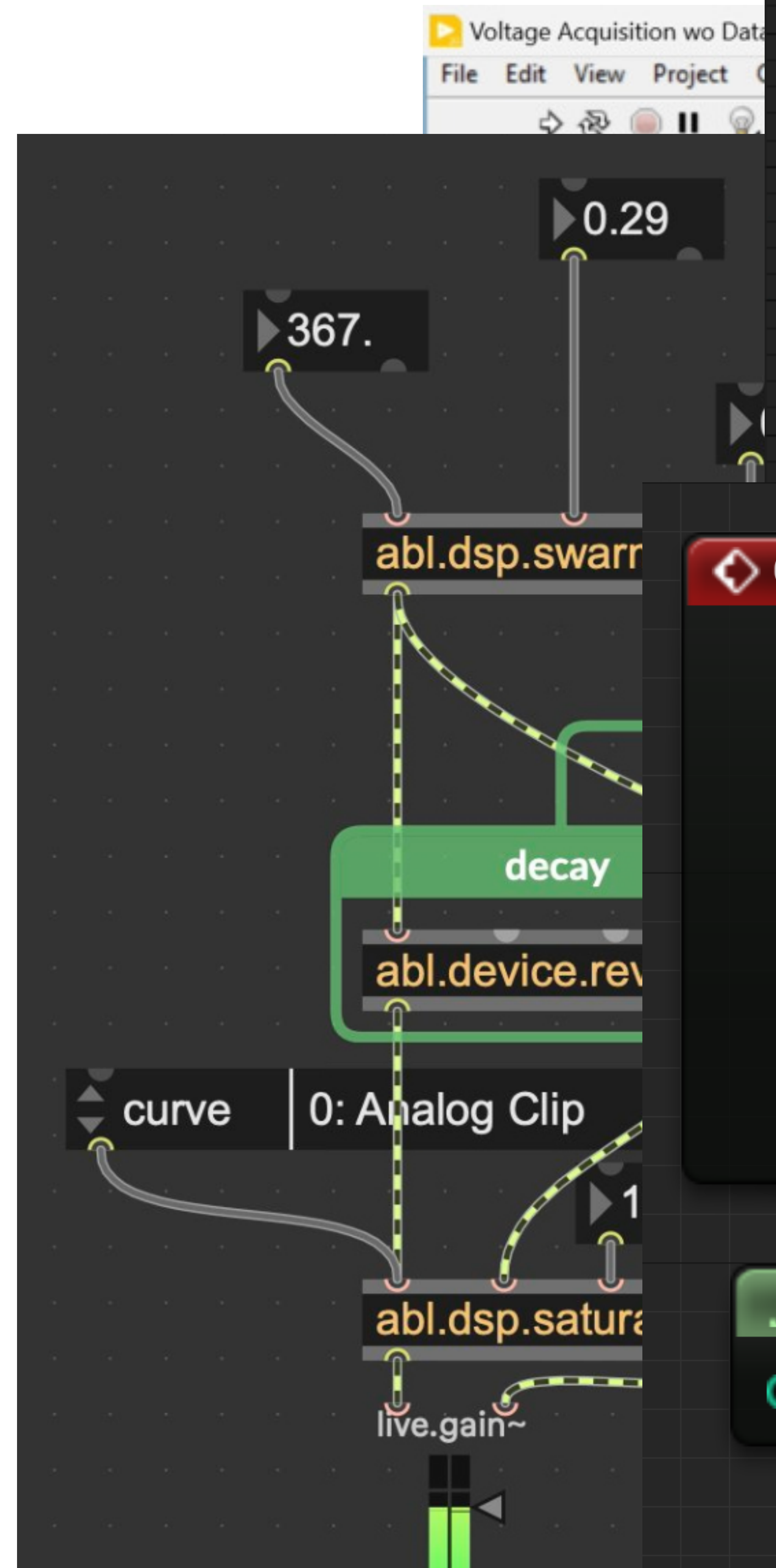
- MCMC sampler is quite homogeneous in terms of workflows
- Observable calculation varies very much across the field
- However: **elementary steps are similar**
  (e.g. quark propagator calculations, Feynman diagram contractions, etc...)
- Similar low-level steps, heterogeneous high-level structure

→ ***Good application for dataflow programming paradigm***

Dataflow programming

- A calculation is a graph off elementary steps (*modules* or *nodes*)
- A modules take inputs and generate outputs
- Inputs are generated by other modules etc...
- Initial modules have no inputs, final modules have no outputs
- *Data flows* between computation steps
- *Module design is independent from complexity of high-level calculation*





On Component Begin Overlap (Box)
Overlapped Component
Other Actor
Other Comp
Other Body Index
From Sweep
Sweep Result

Branch
True
Condition
False

==

Launch Character
Target is Character
Target
Launch Velocity
X 0.0 Y 0.0 Z 0.0
XYOverride
ZOverride

Get Player Pawn
Player Index 0
Return Value

Cast To Character
Object
Cast Failed
As Character

Epic Games Unreal Engine

General design of Hadrons

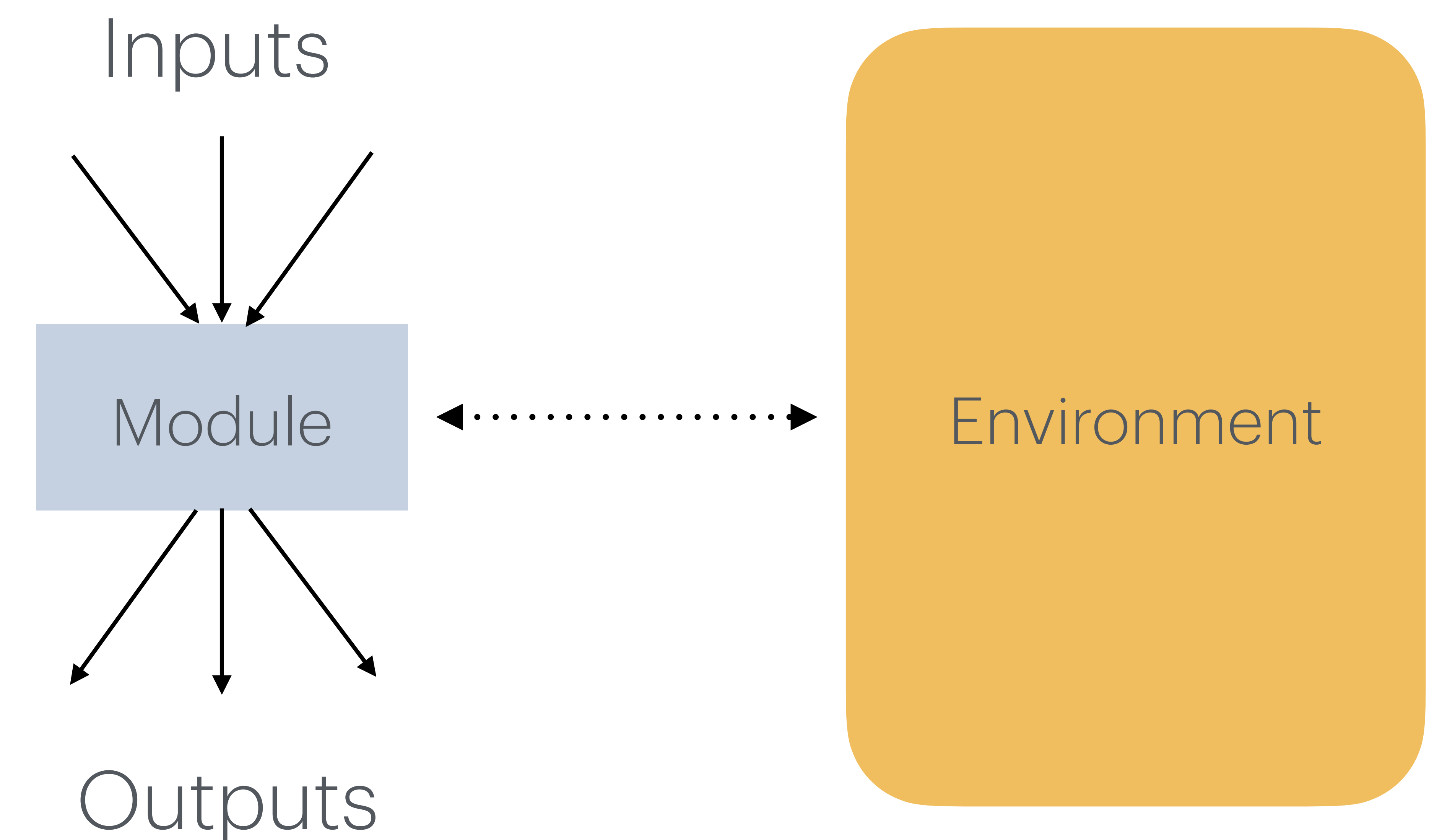
Language and dependencies

- Hadrons is a GPL v2 open-source C++14 library
  🔗 https://github.com/aportelli/hadrons
- It is built as a high-level extension of the **Grid library**
  🔗 https://github.com/paboyle/Grid
- Grid is a **multi-platform HPC lattice library** supporting x86, ARM, AMD/NVIDIA/Intel GPUs
- Hadrons use Grid API for low-level operations
- Documentation https://aportelli.github.io/Hadrons-doc

General design of Hadrons

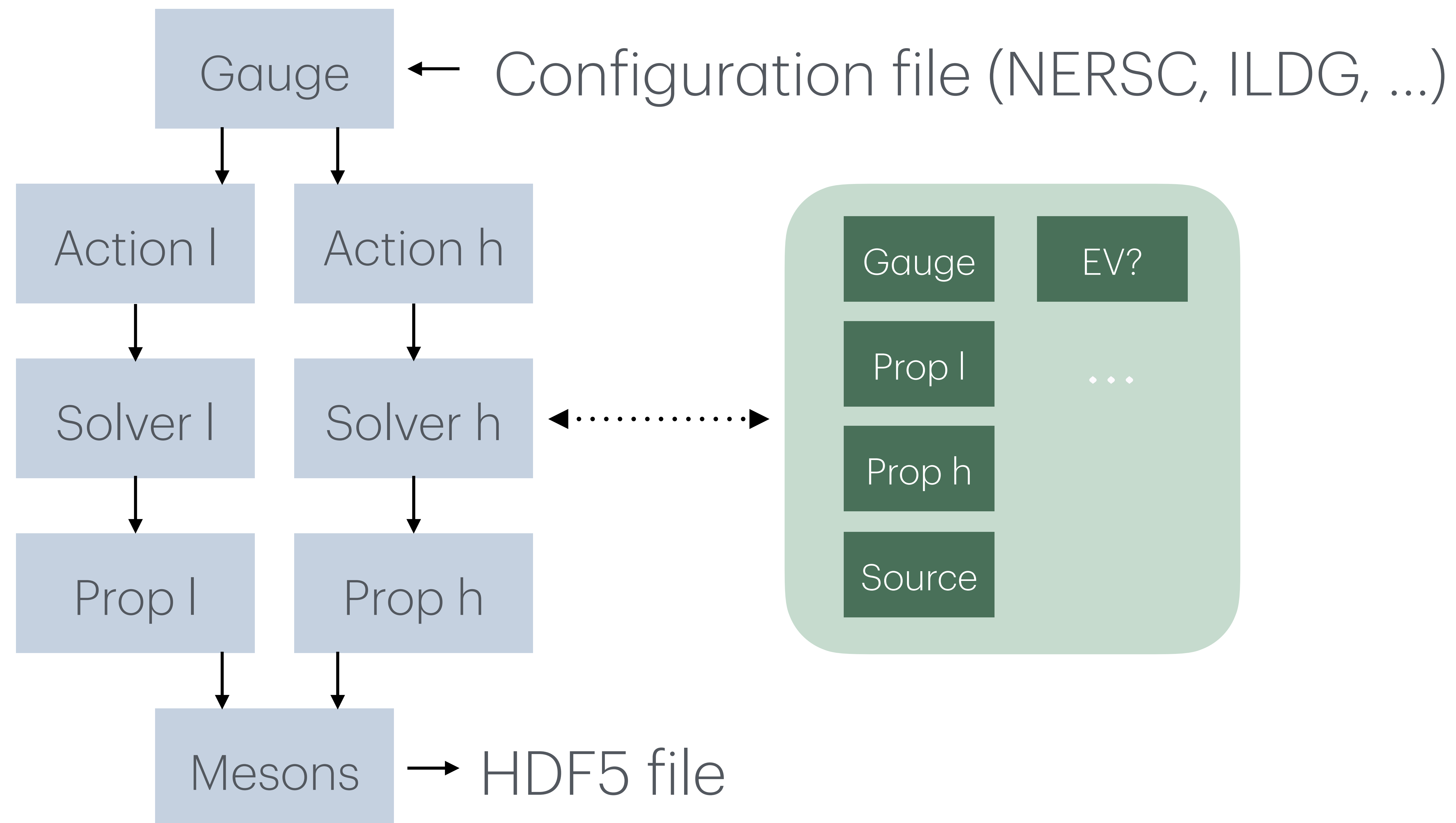
Modules

- The *Environment* is a global object container
- Objects are identified by strings and can have any C++ type
- *Modules* are arbitrary Grid routines that
  1. (maybe) read inputs from the environment
  2. Do some processing
  3. (maybe) store outputs in the environment



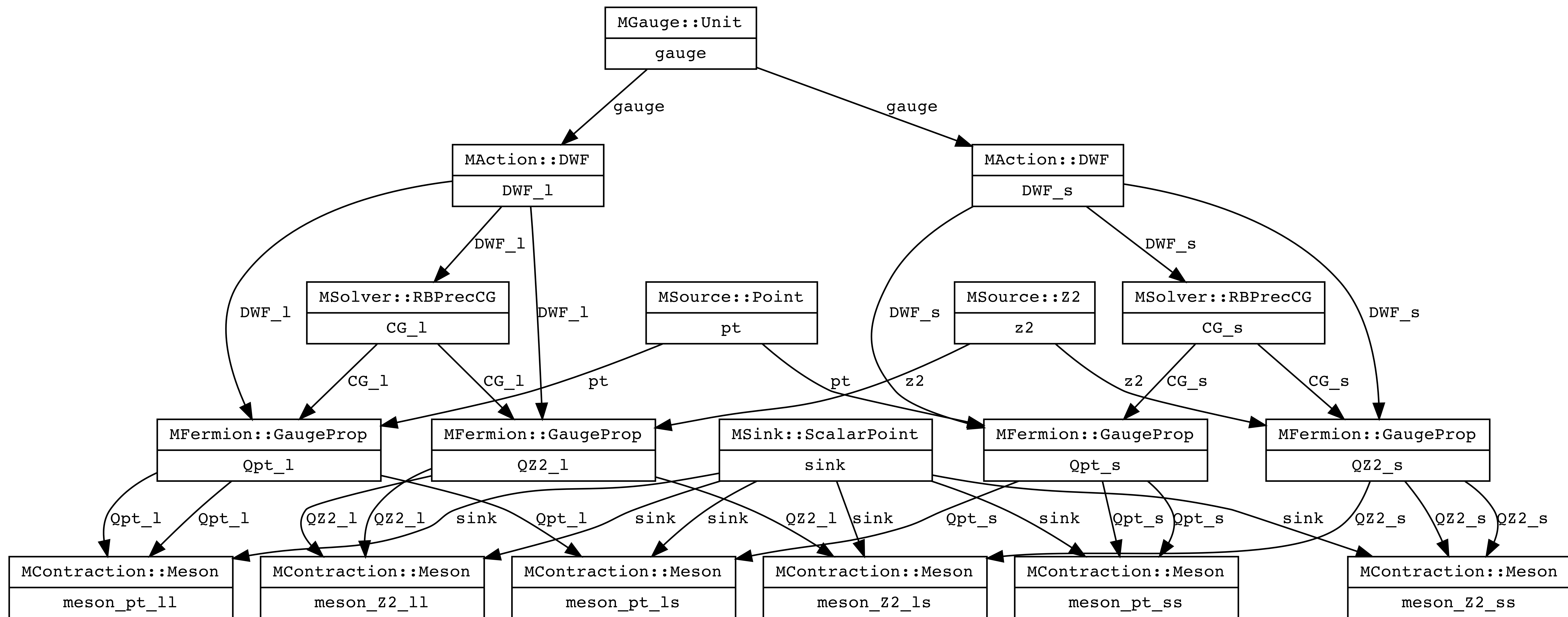
General design of Hadrons

Module graph overview



General design of Hadrons

Real module graph



General design of Hadrons

Global containers: `Environment`

- The environment class is implemented using a singleton pattern
- It can be accessed anywhere with `Environment::getInstance()`
- It can be accessed within a module with `env()`
- It stores grids and objects, they can be created using the functions above, or the convenience macros from `Module.hpp` (look for all `#define env…`)
- Memory management is automatic (I'll come to this in a few slides)
- Object can have standard/temporary/cache scope

General design of Hadrons

Global containers: `VirtualMachine`

- Also a singleton (access with `VirtualMachine::getInstance()` and `vm()`)
- Stores the modules themselves
- Stores dependency graphs for modules and objects
- Can execute a module graph
- At a high-level, *it is unlikely you need to directly interact with it*

General design of Hadrons

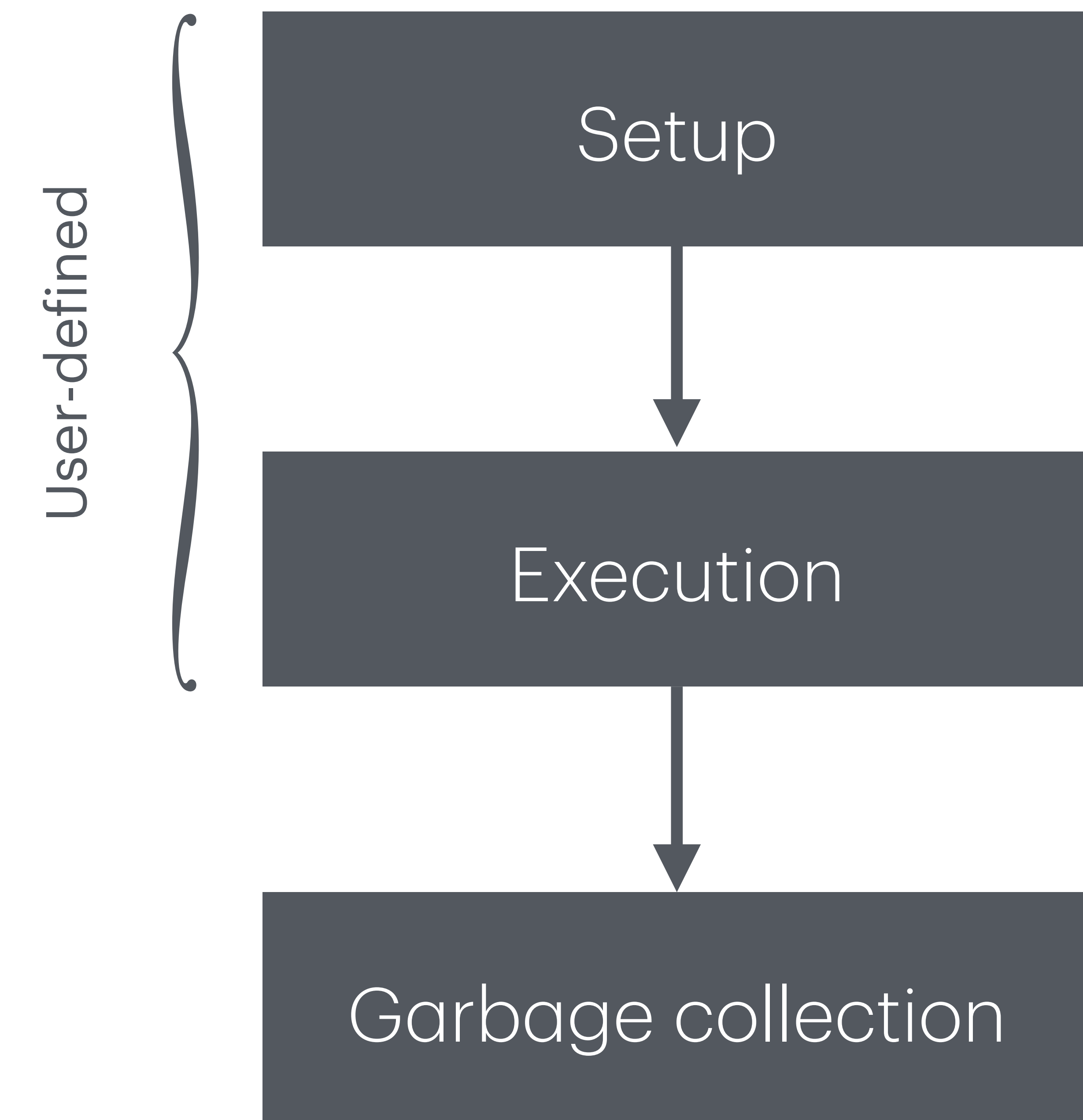
Application class

- The **`Application`** class is a high-level API to create a workflow with modules
- Handles interactions with the VM, including:
  - Inserting modules in the graph
  - Running the application
- Applications created using an XML or C++ API
- C++ API *strongly recommended* for production

```
Application              application;

// global parameters
Application::GlobalPar globalPar;
globalPar.trajCounter.start                  = 1500;
globalPar.trajCounter.end                    = 1520;
globalPar.trajCounter.step                   = 20;
application.setPar(globalPar);

// gauge field
application.createModule<MGauge::Unit>(“gauge");

// sources
MSource::Z2::Par z2Par;
z2Par.tA = 0;
z2Par.tB = 0;
application.createModule<MSource::Z2>("z2", z2Par);
MSource::Point::Par ptPar;
ptPar.position = "0 0 0 0";
application.createModule<MSource::Point>("pt", ptPar);

// sink
MSink::Point::Par sinkPar;
sinkPar.mom = "0 0 0";
application.createModule<MSink::ScalarPoint>("sink", sinkPar);
application.createModule<MSink::SMatPoint>("sinkMat", sinkPar);

// ...

// execution
application.saveParameterFile("spectrum.xml");
application.run();
```

Module execution sequence

- All modules derive from `ModuleBase`
- The `setup` virtual function deals with memory allocations and small parameter parsing
- The `execute` virtual function do the actual physics calculation
- `execute` is always ran after `setup`
- After the execution step, garbage collection is done (this is done automatically, see next slide)



Module dependencies

- Modules can overload `getInput()` and `getOutput()`
- Both functions just return a vector of strings with the inputs/outputs names
- More advanced: `getObjectDependencies()` setup dependencies between objects (e.g. if an output own a reference on an input)
- When the graph is complete, all inputs must be the outputs of modules
- Outputs do not need to be inputs
- The dependencies ignore types, a type error would potentially happen in `execute`

Module parameters

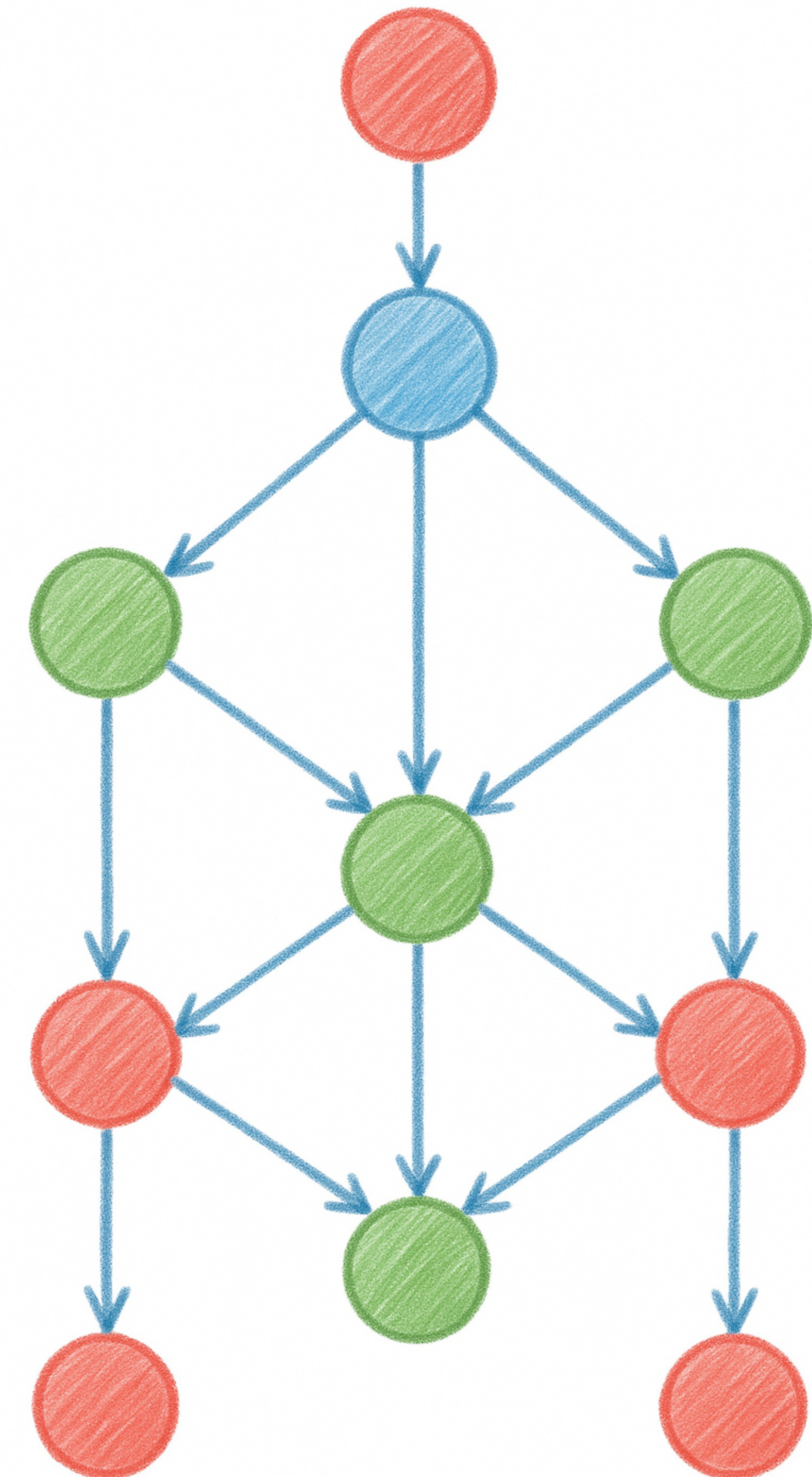
- Module parameters are provided as a Grid serialisable class
- This is a template argument of Module
- Anywhere in the module, runtime parameters can be accessed through `par()`
- Parameters are provided through the C++/XML API when the module is created

```
class PointPar: Serializable
{
public:
    GRID_SERIALIZABLE_CLASS_MEMBERS(PointPar,
                                    std::string, position);
};

template <typename FImpl>
class TPoint: public Module<PointPar>
{
    // ...
};

// in the module code one can use par().position
// to get the string above
```

Scheduling

Execution scheduling

- Assuming the graph is acyclic, scheduling is done through a topological sort
- This is standard, however in general many schedules are possible (factorial growth)
- Two modes
  - Naive scheduler: modules scheduled in the order they were inserted in the application
  - Genetic scheduler: a genetic algorithm try to find a schedule minimising the maximum memory occupancy



Scheduling

Garbage collection scheduling

- After each module execution

  1. Any object with temporary scope is destroyed

  2. Any object which is
     a) not the input of a future module
     b) not the dependency of any required object

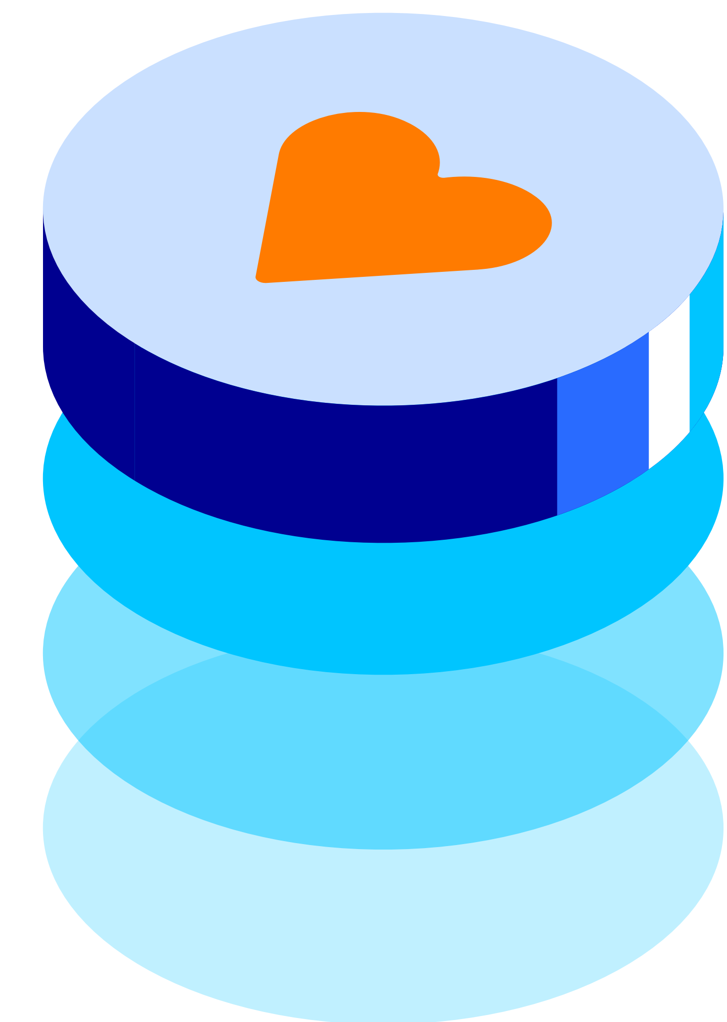
     is destroyed

- Things can get complicated, the log says explicitly what is destroyed

- The debug log will dump the whole garbage collection schedule

Database support

- Hadrons supports cataloguing informations in SQLite databases
- Three databases
    - Application DB: module graph and parameters
    - Stat DB: realtime run statistics
    - Result DB: result file catalogue (user-defined)
- See Test_hadrons_spectrum for an usage example of the result DB
- *Data analysis crawling through a file tree is fragile, use databases!*



Integrated profiling

- At the of a run the application will automatically return an ordered list of the heaviest modules by names and types
- Steps in a module can also be profiled using the `startTimer`/`stopTimer` functions
- This is zero-overhead ad-hoc profiling
- *In most cases it will get you 90% of the way to optimal code, think about it before using complicated third-party profilers*

```
............... Module type breakdown
                        Grid::Hadrons::MFermion::GaugeProp: 5.788e+04 s (58.4%)
                        Grid::Hadrons::MFermion::ZGaugeProp: 3.196e+04 s (32.2%)
                          Grid::Hadrons::MFermion::EMLepton: 3327 s (3.4%)
                  Grid::Hadrons::MContraction::WardIdentity: 1774 s (1.8%)
         Grid::Hadrons::MIO::LoadCoarseFermionEigenPack250F: 1652 s (1.7%)
             Grid::Hadrons::MContraction::WeakMesonDecayKl2: 1517 s (1.5%)
                         Grid::Hadrons::MContraction::Meson: 536.9 s (0.5%)
                                Grid::Hadrons::MSink::Smear: 323.8 s (0.3%)
                          Grid::Hadrons::MSource::SeqAslash: 80.95 s (0.1%)
                          Grid::Hadrons::MAction::ScaledDWF: 34.77 s (0.0%)
```

module type profile in log

```
for (unsigned int step = 1; step <= par().steps; step++) {
    flowt += evolve.epsilon;
    LOG(Message) << "Step " << step << " (tau = "<< flowt << ")" << std::endl;
    startTimer("evolution");
    evolve.evolve_gauge(Uwf);
    stopTimer("evolution");
    if (step % par().meas_interval == 0) {
        LOG(Message) << "Compute observables" << std::endl;
        startTimer("observables");
        evolve.gauge_status(Uwf,result,flowt);
        stopTimer("observables");
    }
}
```

custom timers for profiling within a module

Showcase

PHYSICAL REVIEW LETT

Light and Strange Vector Resonances from

Peter Boyle,[1,2] Felix Erben,[3,2] Vera Gülpers,[2] Maxwe
Nelson Pitanga Lachini,[4,2,*]
[1]Physics Department, Brookhaven National L
[2]School of Physics and Astronomy, University of Ed
[3]CERN, Theoretical Physics Dep
[4]DAMTP, University of Cambridge, (
Wilberforce Road, Cambridge,
[5]RIKEN Center for Computational

(Received 15 July 2024; revised 17 October 2024;

We present the first *ab initio* calculation at physical c
lightest pseudoscalar mesons interacting via the strong
chromodynamics, we postdict the defining parameter
$K^*(892)$, which manifest as complex energy poles i
The calculation proceeds by first computing the finite-
and then determining the amplitudes from the energi
includes a data-driven systematic error, obtained by sc
the spectrum from Euclidean correlators, as well as t
results, obtained by analytically continuing multiple p
$M_\rho = 796(5)(50)$ MeV, $\Gamma_\rho = 192(10)(31)$ MeV, M_{K^*}
where the subscript indicates the resonance and M an
where the first bracket indicates the statistical and th

DOI: 10.1103/PhysRevLett.134.111901

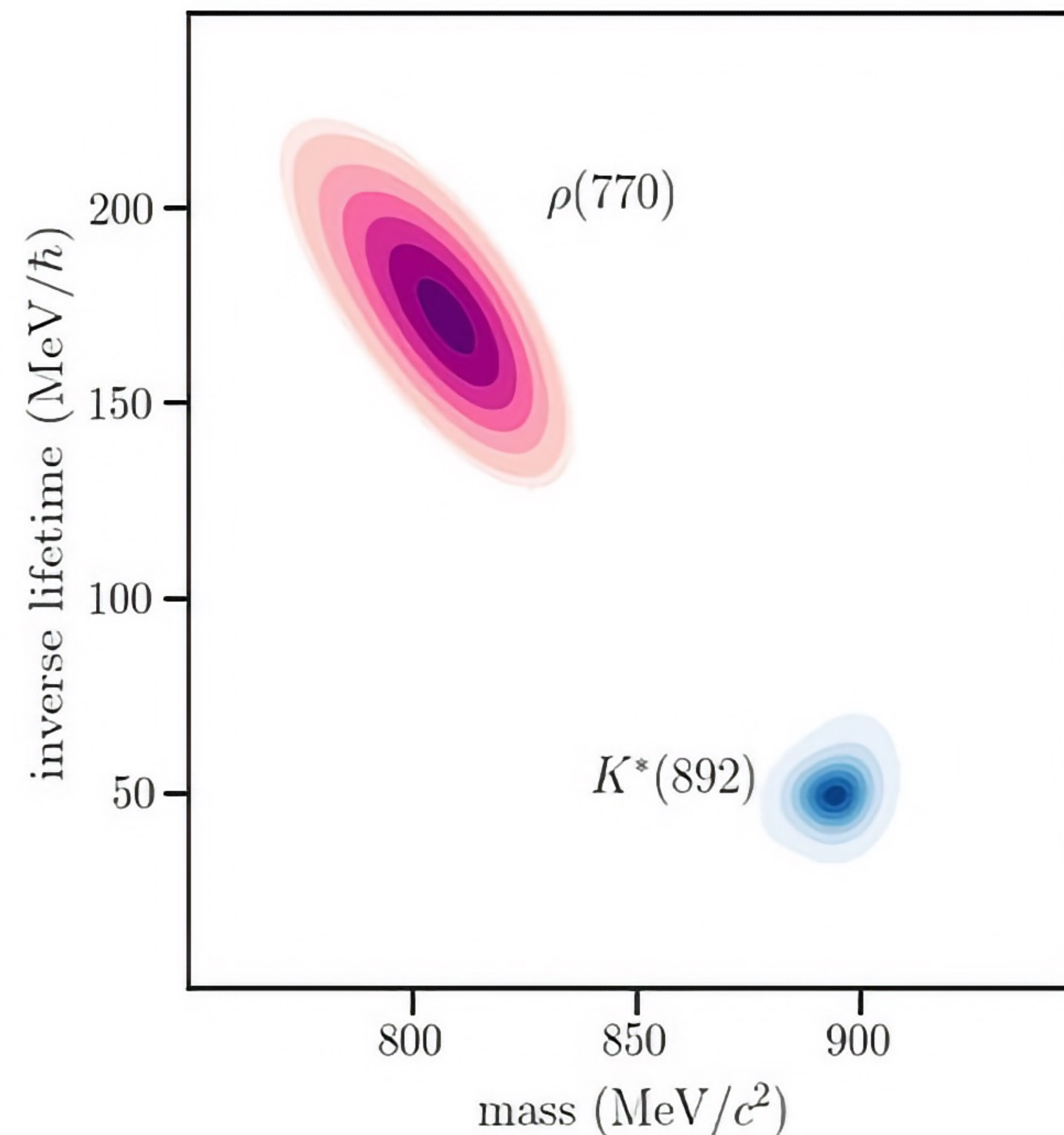


Image credit: Nelson Lachini (DiRAC Science Highlight)

ear projects

ny thousands of modules

)s of TB of intermediate products

ı the same application in parallel
both a CPU and a GPU system

How to get started

- Module documentation: https://aportelli.github.io/Hadrons-doc
- Read simple module code (e.g. sources or actions)
- Read examples in the `tests` folder
- A C++ application template is available in the `application-template` folder

Future features

- The Grid serialisation is rigid and bound to C++ code
  - Overhaul of the serialisation in Hadrons as a separate library: Scribe
  - Will support schemas, HDF5 & JSON, Python binding, etc…
- The genetic scheduler does not currently scale for large workflows
  - The main issue is the memory counting
  - Will be overhauled with a parallel algorithm based on freeing priority
- Jet GUI: cf. Ryan’s talk now