

# ***Workshop on Computing, Software, Simulation and Offline Code & Physics***

**August 2014 - Fermilab**

# The plan for this workshop

---

**Tuesday: Introduction to Art**

**Wednesday: Introduction to the Simulation**

**Thursday: More advanced Art and Offline code  
and how other experiments use Art**

**Friday: Physics questions and moving forward**

# **We'll scratch the surface**

---

**There is a lot to cover and you won't be an expert when this workshop is done. But,**

- o You'll know the basics**
- o You'll know where to look for information**
- o You'll be able to ask questions**

# An architecture

---

**I'm slowly but surely writing an architecture and implementation document for Muon g-2 software and computing.**

**An architecture describes computing resources, components and interconnections. The implementation describes how the architecture is used to manage, generate, and process scientific data to obtain physics results.**

# Principles of the architecture

---

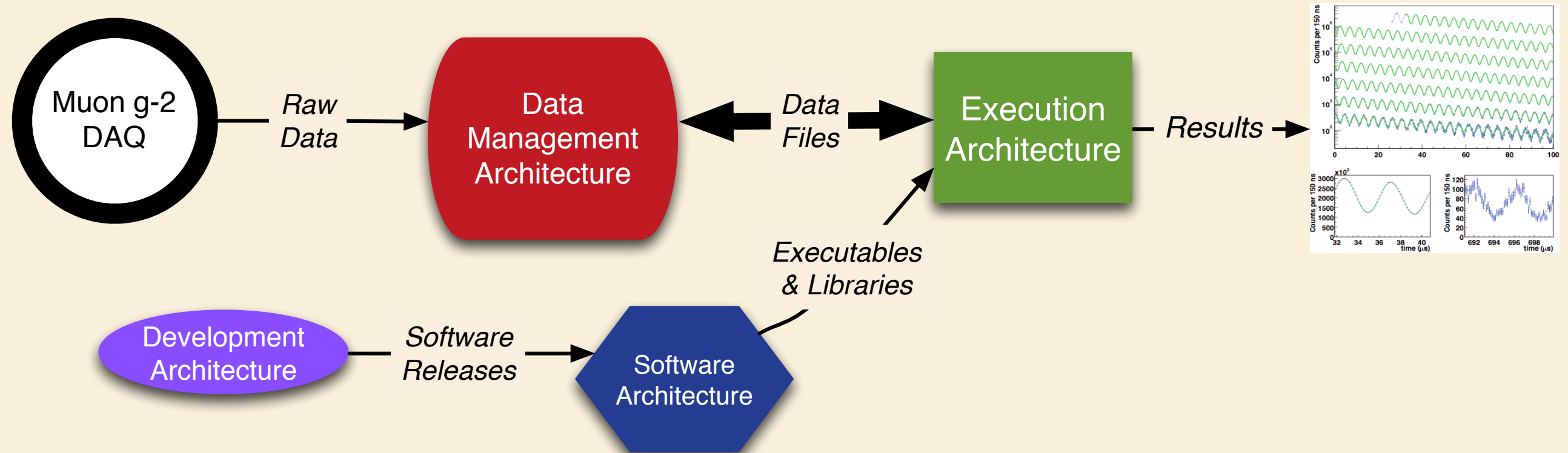
**Reproduce Results**

**Enable Collaboration**

**Emphasize Physics, not Computing**

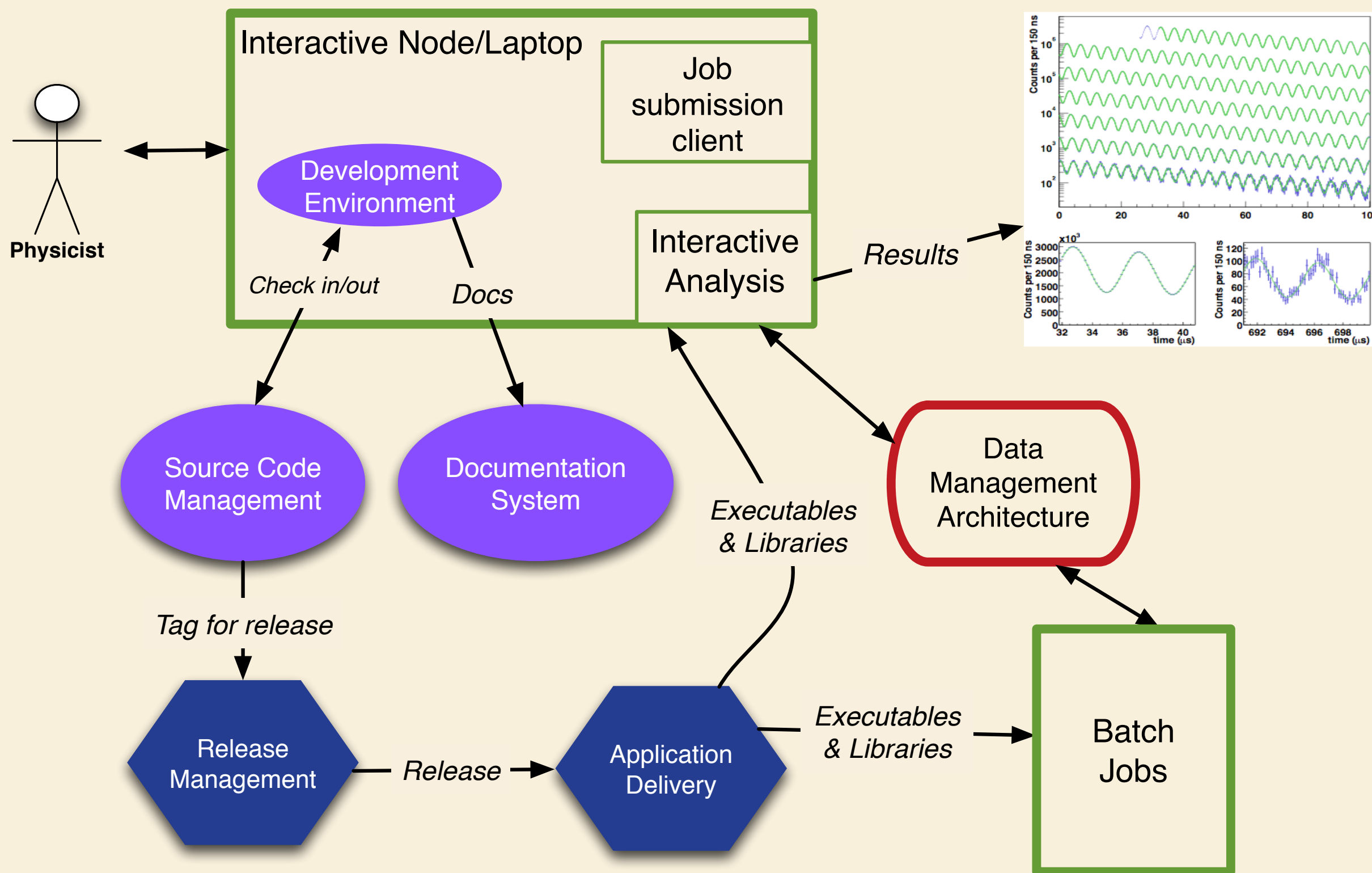
**Enable Participation from Everyone**

# The overall architecture

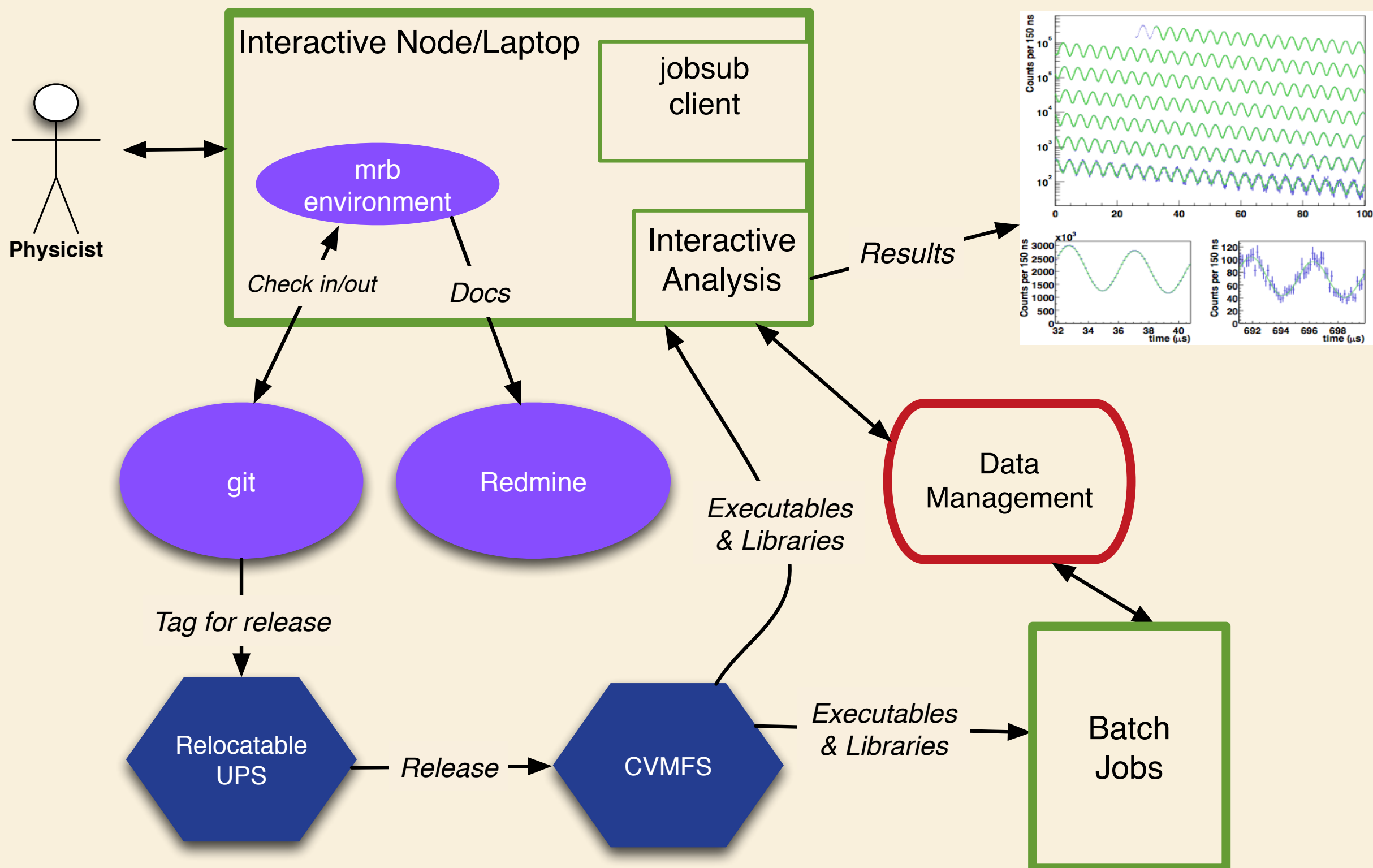




# Software & Development Arch

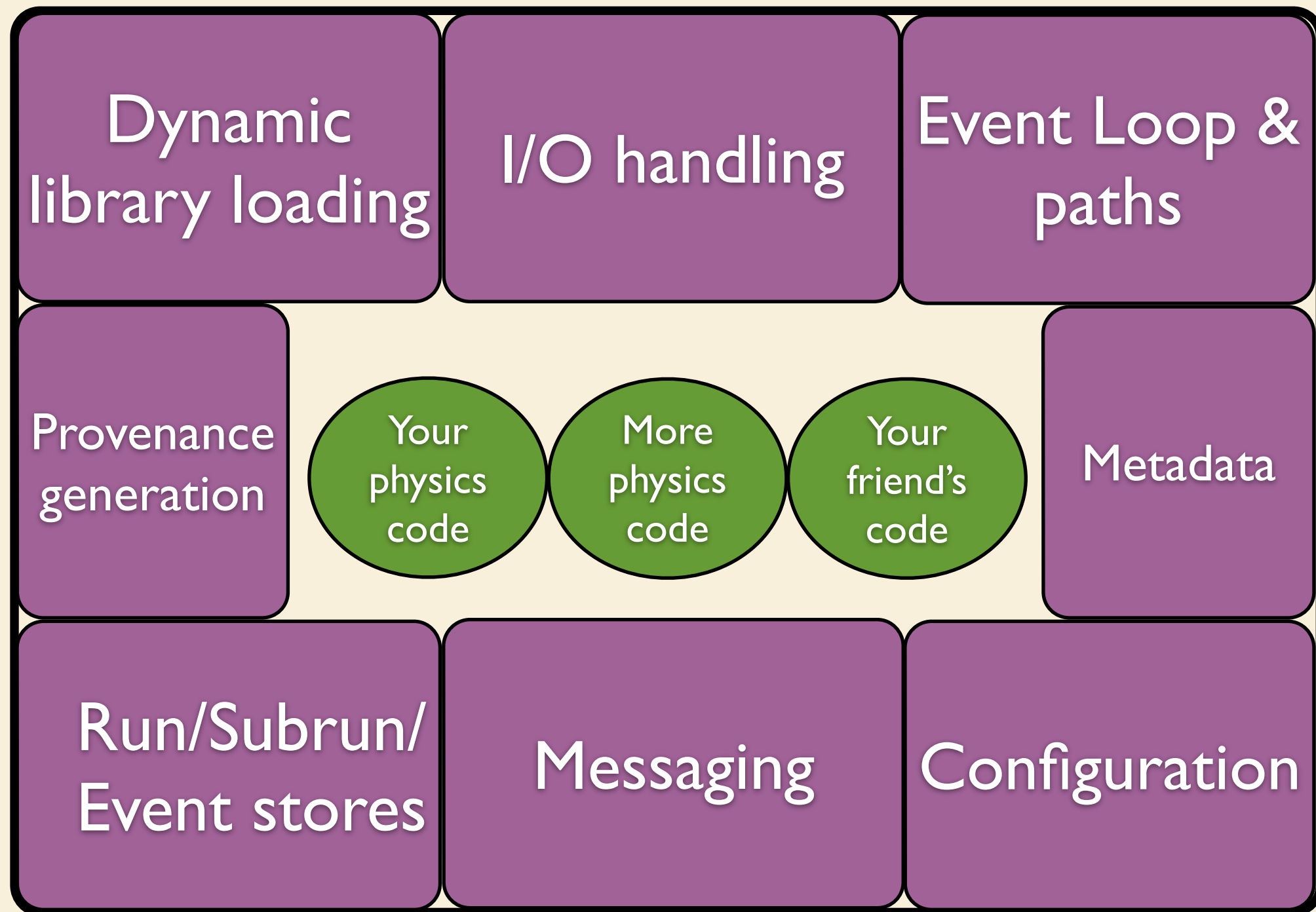



# Software & Development Impl





# And there's the Art framework



 **Code you write**

 **Code you use from the framework**

# What a framework gives you

---

**Allows you to write your physics code without worrying about the infrastructure. Makes it easy to work with others.**

**But not for free – you have to learn it!**

**Some people find such a system constraining:**

**Infrastructure is hidden behind the scenes from you**

**Your ideas may not be included**

**You have to trust a system you didn't write**

**You miss out on the fun of writing super-cool complicated C++ code**

**Some people find such a system liberating:**

**You can concentrate on physics code**

**Your C++ is pretty easy (you are *using* a complicated system, not *writing* it)**

**You get to miss out having to maintain the complicated C++ code (yay!)**

**You can use code from others and share yours with others**

**You can get services for free (e.g. data handling)**

# Why is this important?

---

**The story**

# In g2migtrace/src/primaryConstruction.cc

```
// constructionMaterials is essentially a "materials library" class.
// Passing to to construction functions allows access to all materials

/**** BEGIN CONSTRUCTION PROCESS ****/

// Construct the world volume
labPTR = lab -> ConstructLab();
// Construct the "holders" of the actual physical objects
#ifdef TESTBEAM
    Arch.push_back(labPTR);
#else
    Arch = arc->ConstructArcs(labPTR);
#endif
// Build the calorimeters
// cal -> ConstructCalorimeters(Arch);
// station->ConstructStations(Arch);
#ifdef TESTBEAM
// Build the physical vacuum chambers and the vacuum itself
Vach = vC -> ConstructVacChamber(Arch);
```

# In g2migtrace/src/primaryConstruction.cc

```
// constructionMaterials is essentially a "materials library" class.
// Passing to to construction functions allows access to all materials

/**** BEGIN CONSTRUCTION PROCESS ****/

// Construct the world volume
labPTR = lab -> ConstructLab();
// Construct the "holders" of the actual physical objects
#ifdef TESTBEAM
    Arch.push_back(labPTR);
#else
    Arch = arc->ConstructArcs(labPTR);
#endif
// Build the calorimeters
// cal -> ConstructCalorimeters(Arch);
// station->ConstructStations(Arch);
#ifdef TESTBEAM
// Build the physical vacuum chambers and the vacuum itself
Vach = vC -> ConstructVacChamber(Arch);
```

**I don't think we can't simultaneously  
maintain this code and our sanity**

# In g2migtrace/src/primaryConstruction.cc

```
// constructionMaterials is essentially a "materials library" class.  
// Passing to to construction functions allows access to all materials
```

```
/**** BEGIN CONSTRUCTION PROCESS *****/
```

What if we have a different test beam?

```
// Construct the world volume
```

```
labPTR = lab -> ConstructLab();
```

```
// Construct the "holders" of the actual physical objects
```

```
#ifdef TESTBEAM
```

```
Arch.push_back(labPTR);
```

```
#else
```

```
Arch = arc->ConstructArcs(labPTR);
```

```
#endif
```

```
// Build the calorimeters
```

```
// cal -> ConstructCalorimeters(Arch);
```

```
station->ConstructStations(Arch);
```

What if I want a different detector configuration?

```
#ifndef TESTBEAM
```

```
// Build the physical vacuum chambers and the vacuum itself
```

```
Vach = vC -> ConstructVacChamber(Arch);
```

this kind of code is hard to excise later

**I don't think we can't simultaneously maintain this code and our sanity**



# **Maintaining sanity is hard**

**It's hard to blame the person who did this**

**He just wanted results!**

**We don't have a system that tries to make this easy**

**It's not the system's fault - it wasn't written for that**

**Writing such a system is hard (need experts)**

**Learning such a system is non-trivial too**

# Use a system that makes this easy

---

**Want a system that makes it easy to work together**

**ART**

**Modular (you write modules that piece together)**

**Built in Root i/o**

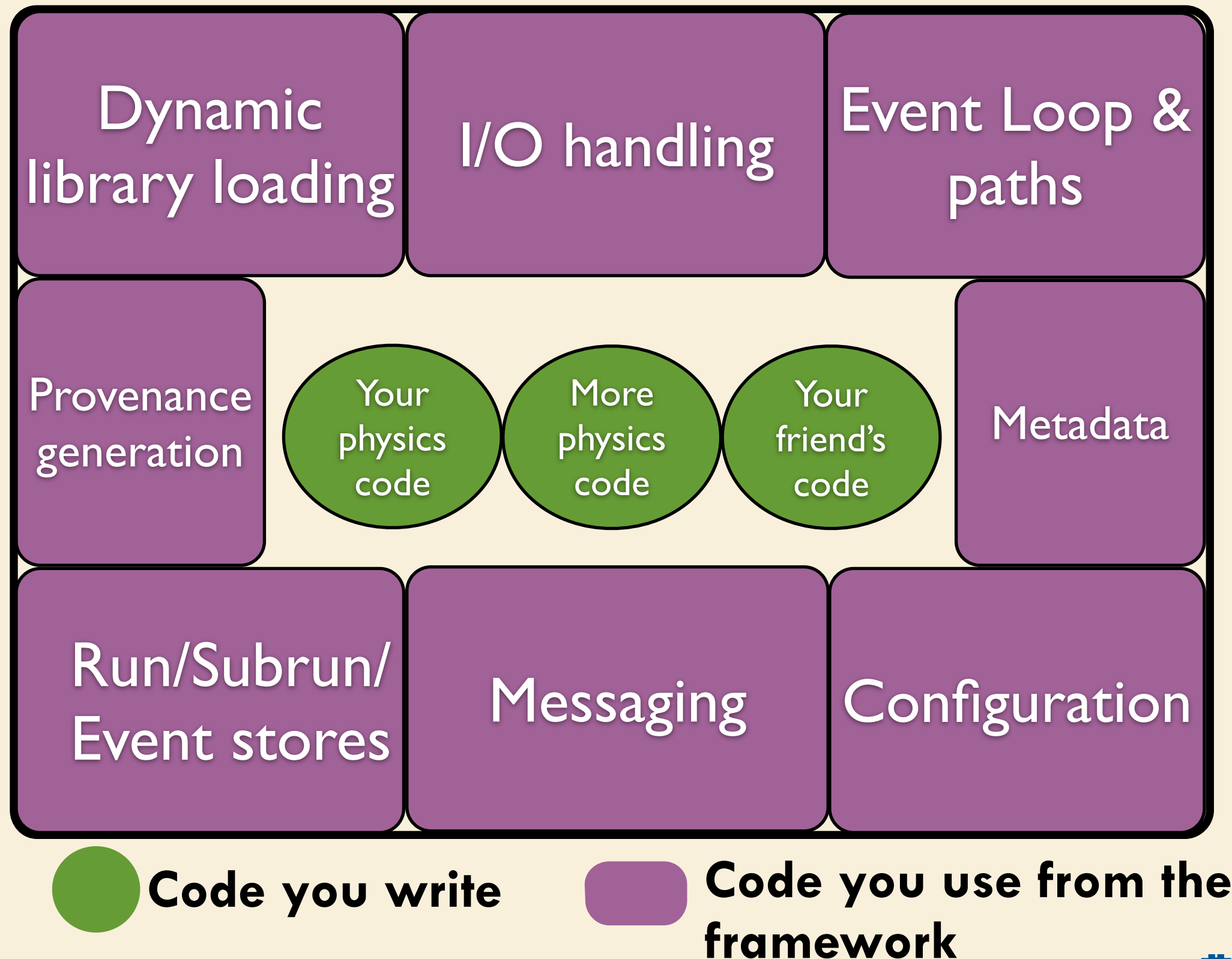
**Built in Configuration System**

**The idea:**

**Using ART, build a modular Geant4 system where the configuration file defines the simulation**

**Here's a little bit about ART (not a full tutorial)...**

# What does a framework do?



# What do you write?

**You write modules that can access data and do things at certain times**

## Types of MODULES:

(All modules can read data from the event)

### o Input source:

A source for data. E.g. a ROOT file or Empty for start of simulated data

### o Producers:

Create new event data from scratch or by running algorithms on existing data

### o Filters:

Like producers, but can stop running of downstream modules

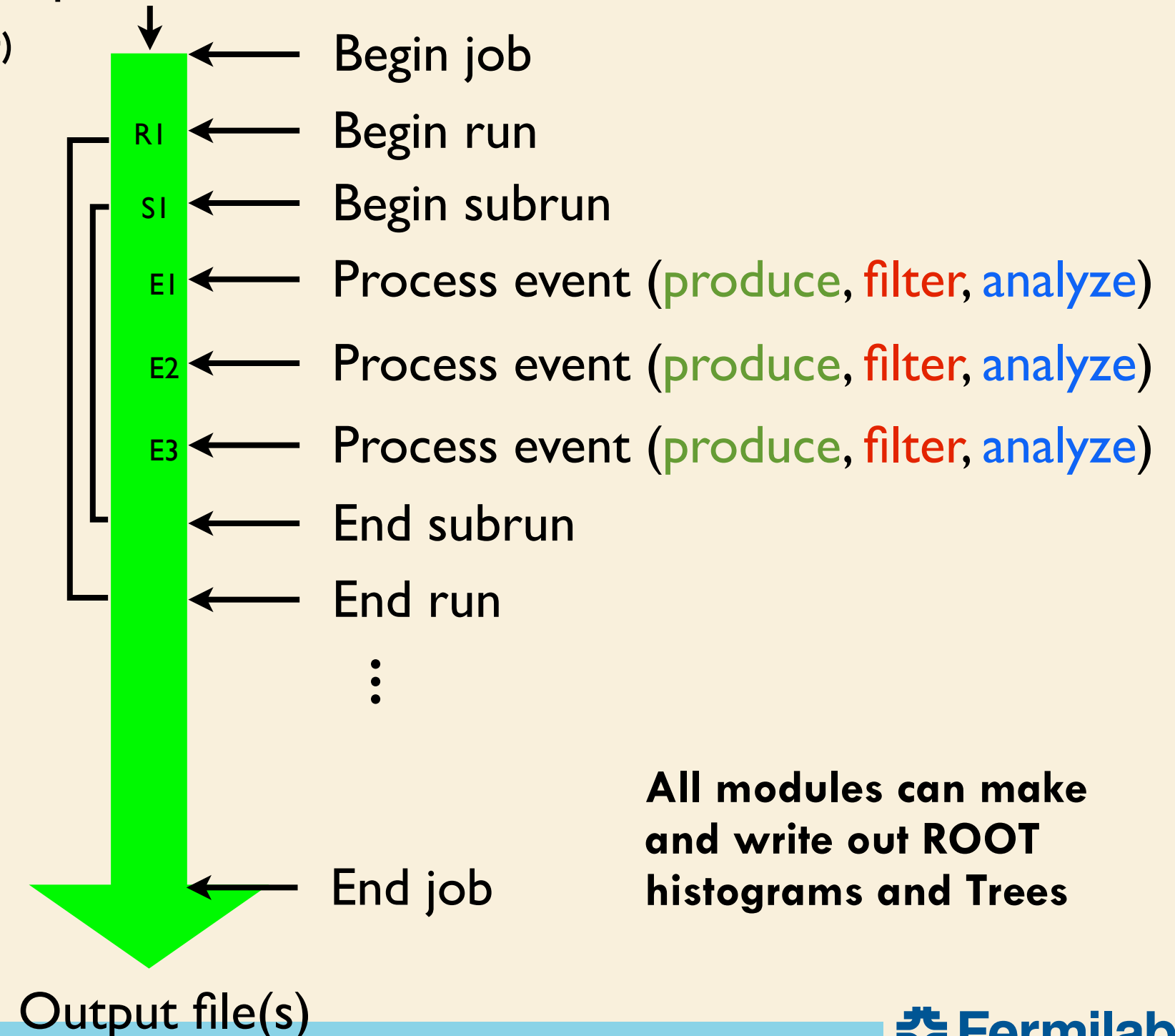
### o Analyzers:

Cannot save to event. For, e.g. diagnostics plots

### o Output module:

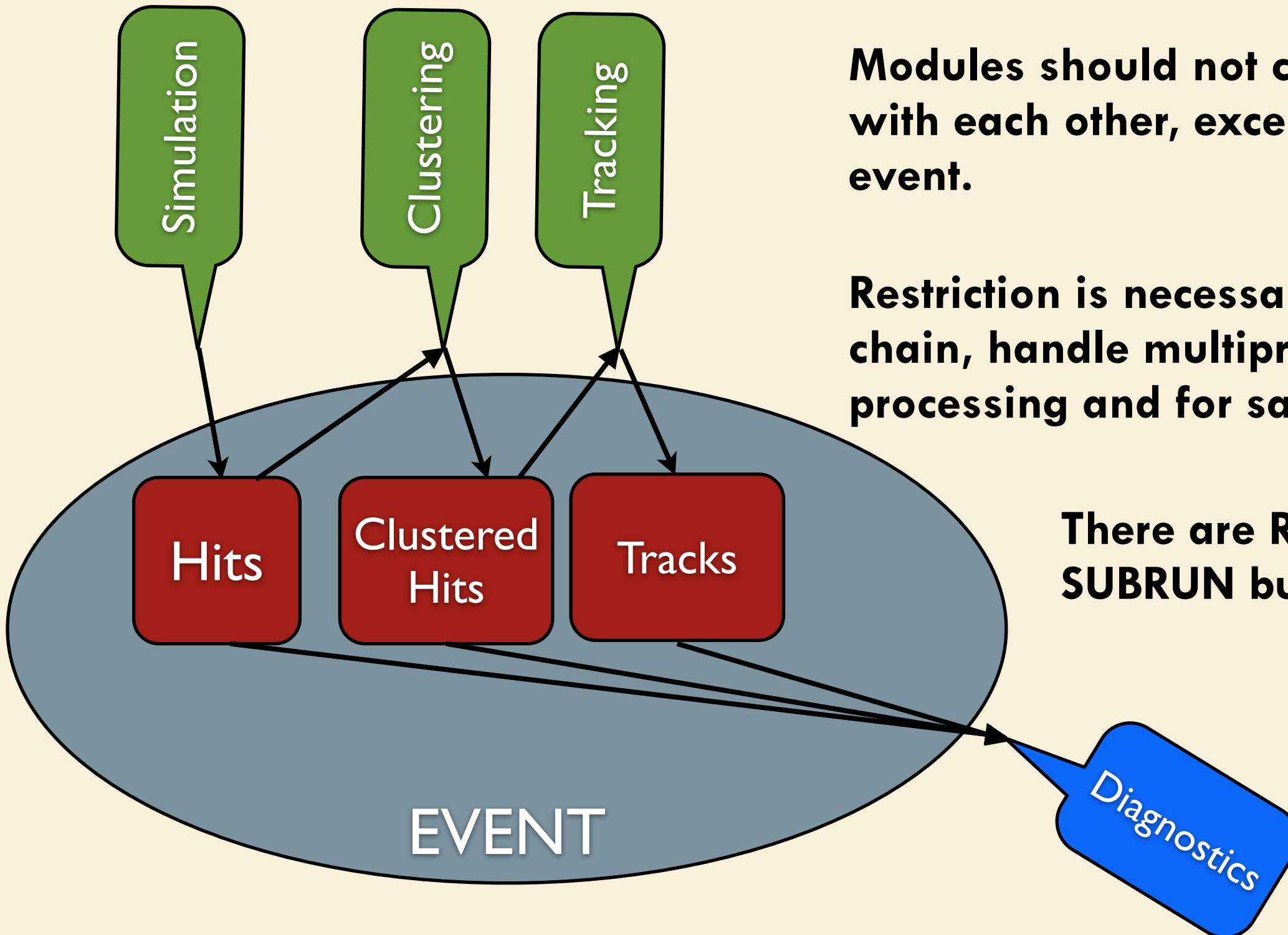
Writes data to output file (ROOT). Can specify conditions and have many files

Input source



# Chain modules - but an important golden rule

**Modules must only pass data to each other via the EVENT**



**Modules should not communicate with each other, except through the event.**

**Restriction is necessary to break chain, handle multiprocessor processing and for sanity.**

**There are RUN and SUBRUN buckets too**

# An example config (FHICL) file

**Note empty source**

**analyzers**

**module label and module\_type**

**Run this with**

```
[lyon@gm2gpvm01 ~]$ gm2 -c hello1.fcl
```

```
1  #include "minimalMessageService.fcl"
2  services.message: @local::default_message
3
4  process_name: helloWorld1
5
6  source: {
7      module_type: EmptyEvent
8      maxEvents: 2
9  }
10
11 physics: {
12
13     analyzers: {
14
15         hello: {
16             module_type: HelloWorld1
17         }
18     }
19
20     path1: [ hello ]
21     end_paths: [ path1 ]
22
23 }
24
```



# An example “Hello world” module

HelloWorld1\_module.cc

Note no header

override is helpful

Must have  
DEFINE at bottom

The “artmod”  
scripts writes this  
skeleton for you

This gets built  
into its own  
shared object

```
1  #include "art/Framework/Core/EDAnalyzer.h"
2  #include "art/Framework/Core/ModuleMacros.h"
3  #include "art/Framework/Principal/Event.h"
4
5  namespace artex {
6
7
8      class HelloWorld1 : public art::EDAnalyzer {
9
10     public:
11
12         explicit HelloWorld1(fhicl::ParameterSet const& pset);
13
14         void analyze(const art::Event& event ) override;
15
16         virtual ~HelloWorld1();
17
18     };
19
20     HelloWorld1::HelloWorld1(fhicl::ParameterSet const& ){}
21
22     HelloWorld1::~~HelloWorld1() {}
23
24     void HelloWorld1::analyze(const art::Event& event){
25
26         mf::LogVerbatim("test") << "Hello, world. From analyze. " << event.id();
27
28     }
29
30 }
31
32 using artex::HelloWorld1;
33 DEFINE_ART_MODULE(HelloWorld1)
```

# Services – an extremely useful feature

Globally accessible objects can be managed by ART as Services

Provide functionality to many modules (same object is accessible to all modules)

Examples:

Message facility, timers, memory checkers, Random numbers, Geometry information

Since a service is an ordinary C++ object, it can hold data and state

**BUT - Remember the golden rule!** Event information goes into the EVENT, not a service

Easy to create:

Your class .cc file simply needs

```
1 #include "art/Framework/Services/Registry/ServiceMacros.h"
2
3 // Ordinary class implementation goes here
4
5 using artg4example::PhysicsListService;
6 DEFINE_ART_SERVICE(PhysicsListService)
```

Easy to use:

The handle acts  
just like a pointer to  
the object

```
1 #include "artg4/services/PhysicsListHolder_service.hh"
2
3 // ...
4
5 void artg4::artg4Main::beginRun(art::Run & r)
6 {
7     // Get the physics list and pass it to Geant and initialize the list if necessary
8     art::ServiceHandle<PhysicsListHolderService> physicsListHolder;
9     runManager_>SetUserInitialization( physicsListHolder->makePhysicsList() );
10    physicsListHolder->initializePhysicsList();
11
12    // ...
```

# Services must be in your FHICL

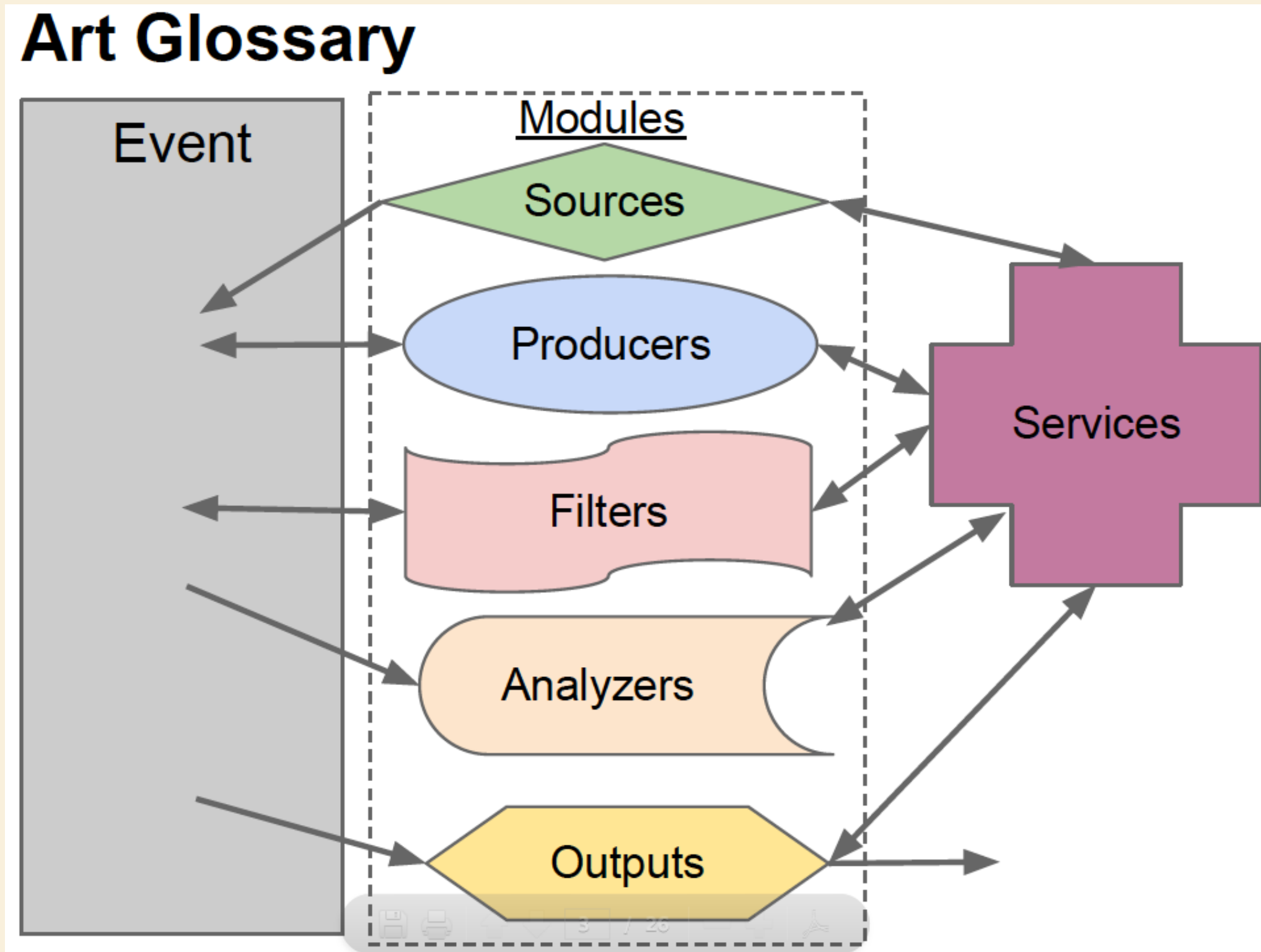
e.g. **Gm2PhysicsList\_service.cc**

**Build system creates  
artg4example\_Gm2PhysicsList\_service.so**

**Specifying Gm2PhysicsList in FCL will  
find it in your LD\_LIBRARY\_PATH**

```
1  services: {
2
3      message : {
4          debugModules : ["*"]
5          suppressInfo : []
6
7          destinations : {
8              LogToConsole : {
9                  type : "cout"
10                 threshold : "DEBUG"
11             }
12             LogToFile : {
13                 type : "file"
14                 filename : "gm2ringsim.log"
15                 append : false
16                 threshold : "DEBUG"
17             }
18         }
19     }
20
21     user : {
22
23         // Mandatory ArtG4 services
24         DetectorHolder: {}
25         ActionHolder: {}
26         PhysicsListHolder: {}
27         RandomNumberGenerator: {}
28
29         Gm2PhysicsList: {}
30     }
31 }
32
33 }
34
35 # ...
```

# Summary



# Where to look for documentation

---

**g-2 Redmine wiki (fast computing index)**

**Art Workbook**

**Art Tutorial**

**Other experiments' documentation**

**C++ Class and Wiki**

**Previous workshops**

# Note

---

**Note that we're using a new version of art and g-2 software (gm2 v5\_0\_0)**

**This is an intermediate version!**



# Demo

---

**[Flip to architecture implementation]**

**Log into gm2gpvm (ssh). Explain gm2gpvm machines.  
[try Mac and virtual machine too]**

**Explain CVMFS, setup environment**

**Show UPS. \$PRODUCTS. Ups list. Ups active.**

**Show gm2 v5\_0\_0**

**Make development area (mrb newDev). Show structure.**

**Check out gm2artexamples (mrb g). Branches. git flow**

# Demo continued

---

**CMakeLists.txt, product\_deps**

**Setup environment (. mrbs) FHICL\_FILE\_PATH**

**Build (mrbs)**

**Try gm2 -c hello1.fcl**

**Go through annotated example. Show libraries**

**Show how modules and data are discovered. Try missing symbol.  
Dependencies follow headers, not libraries [otool -L; ldd; c++filt]**

**Data best practices**

**Art best practices**

# Demo continued

---