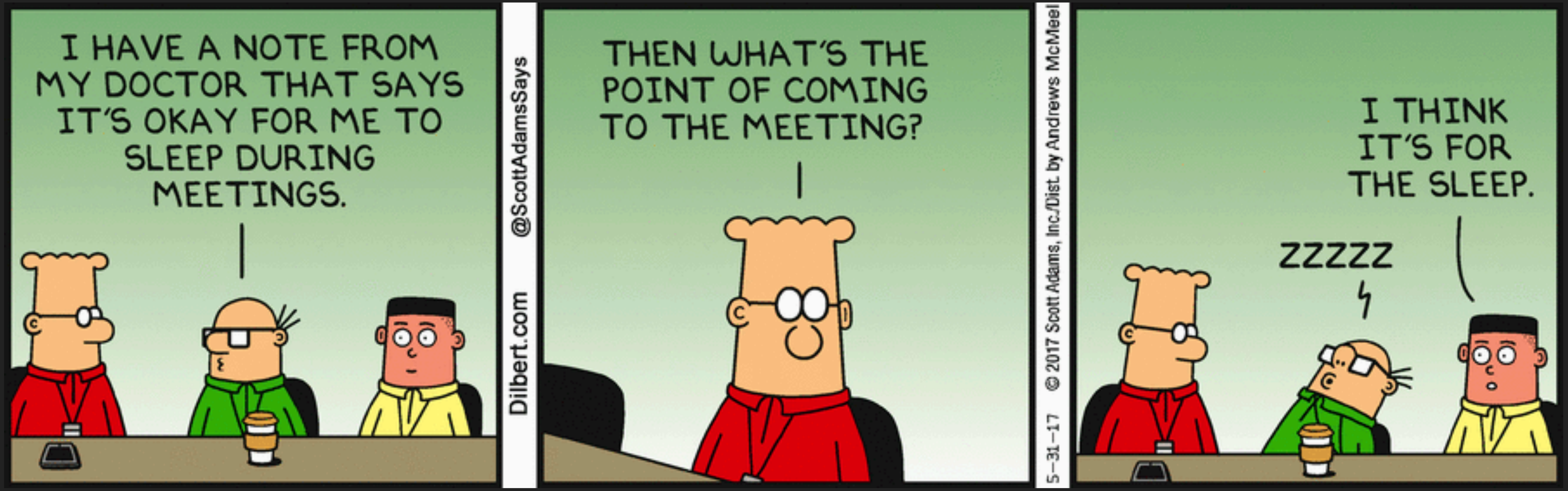
V. FORMATO - CHEP 2018 - 12/07/18, SOFIA

# A PLUGIN-BASED APPROACH TO DATA ANALYSIS FOR THE AMS EXPERIMENT ON THE ISS

## DISCLAIMER


I HAVE A NOTE FROM MY DOCTOR THAT SAYS IT'S OKAY FOR ME TO SLEEP DURING MEETINGS.
@ScottAdamsSays
Dilbert.com
THEN WHAT'S THE POINT OF COMING TO THE MEETING?
© 2017 Scott Adams, Inc./Dist. by Andrews McMeel
5-31-17
ZZZZZ
I THINK IT'S FOR THE SLEEP.

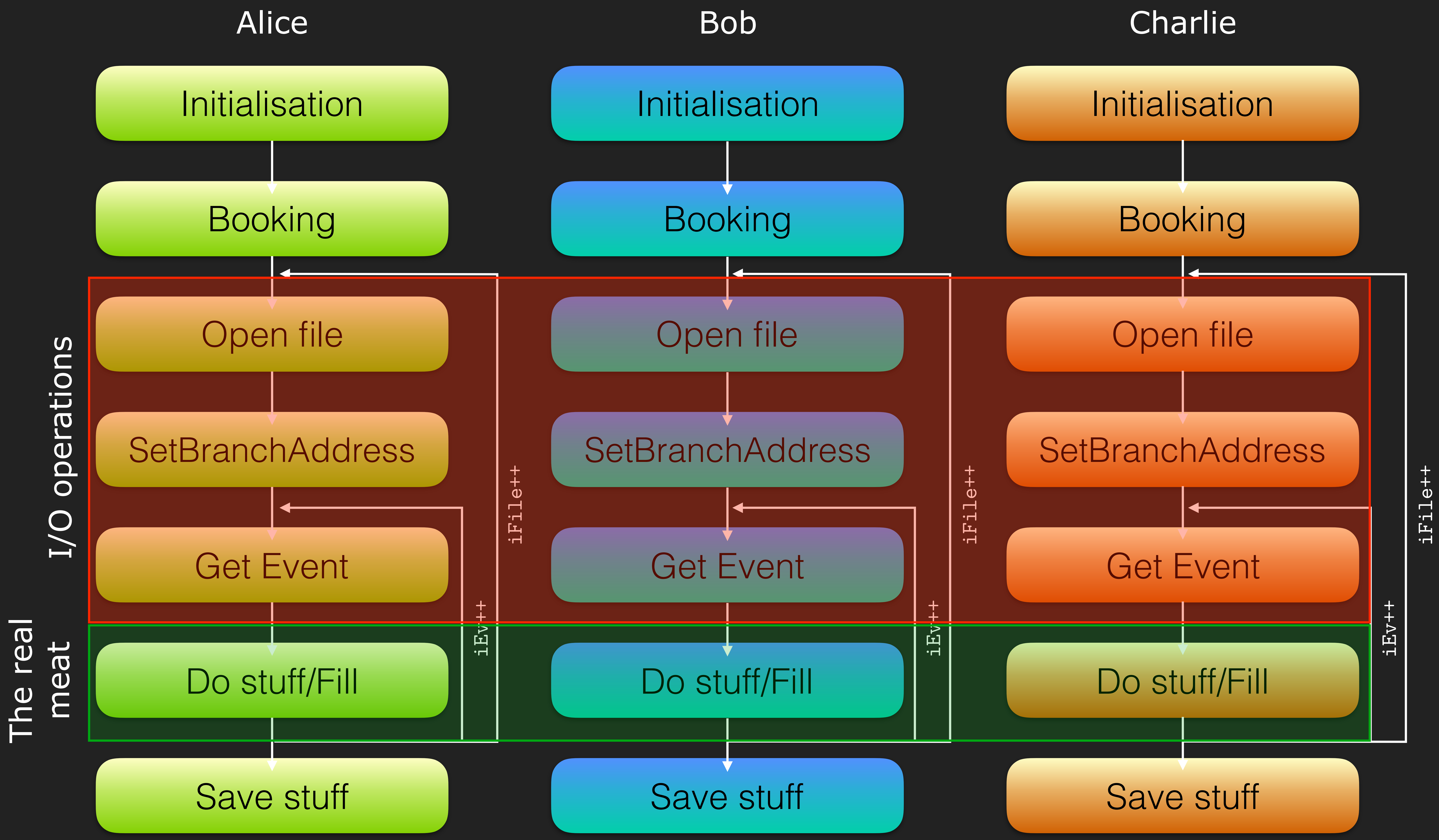
If any of what will follow sound familiar and it's boring, feel free to sleep through this talk 😅

## HOW "WE" DO ANALYSIS

The analysis group: **8+** people from different institutions, **4+** of them working 100% on the analysis.

**Typical** workflow:

- Shared "reduced" dataset
- Each user studies one particular aspect of the analysis
- Work **replicated** between users

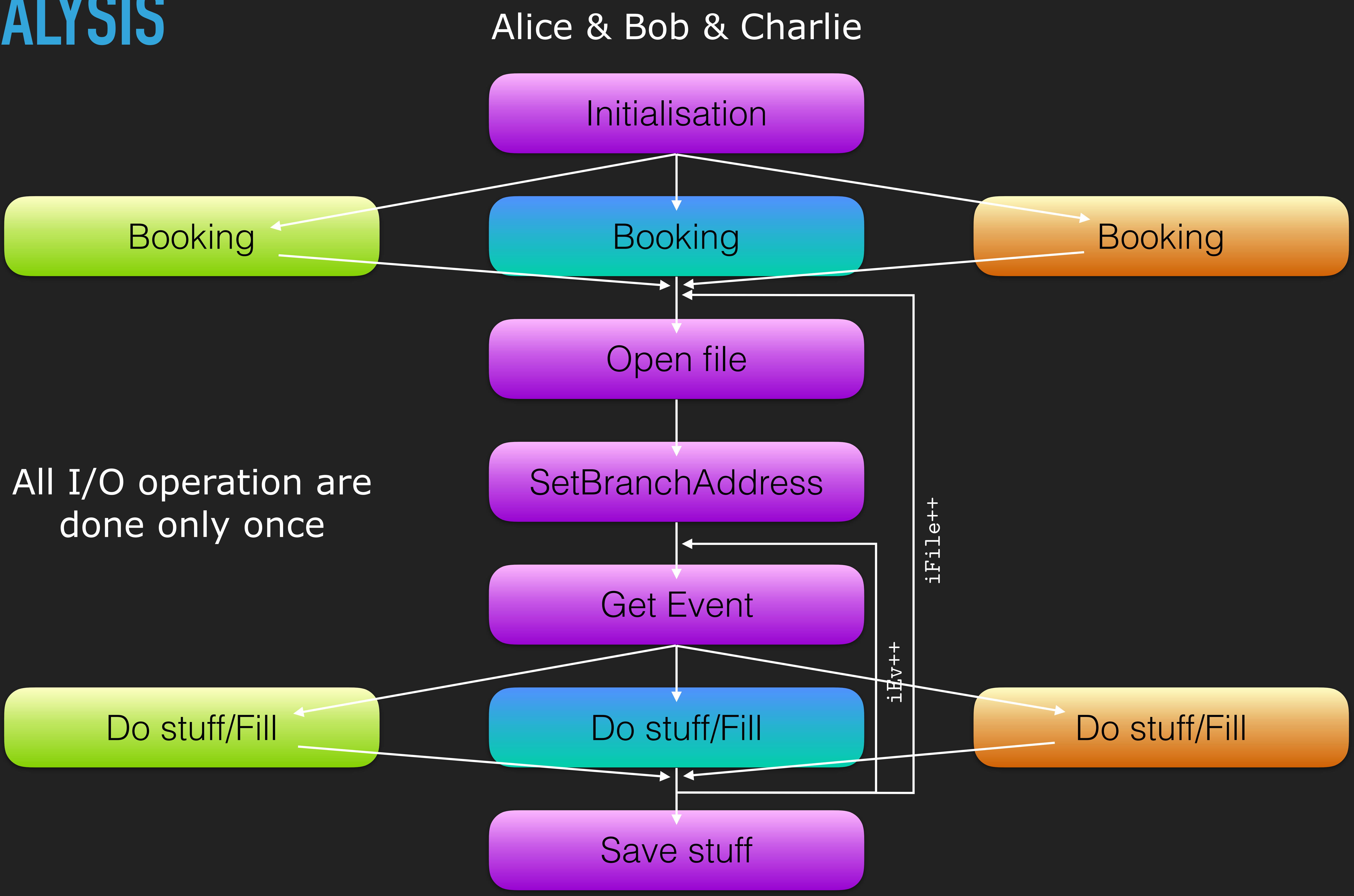


## HOW "WE" WANT TO DO ANALYSIS

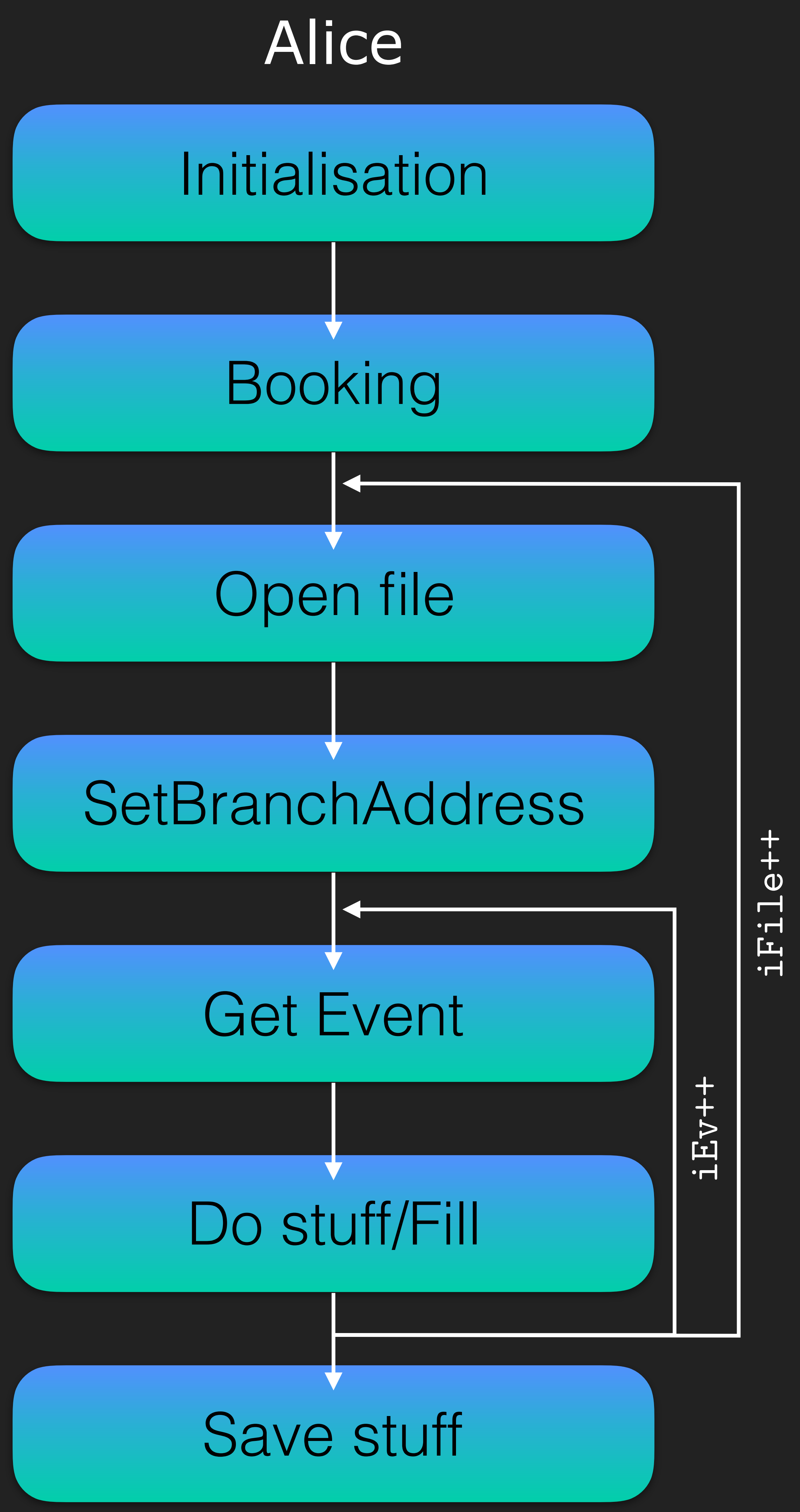
The analysis group: **8+** people from different institutions, **4+** of them working 100% on the analysis.

**Desired** workflow:

- Shared "reduced" dataset
- Each user only writes "what he wants to do with the data"
- All common operations done only **once**, code specialisation only **where needed**
- Code is encapsulated in defined analysis stages



## DON'T REINVENT THE WHEEL

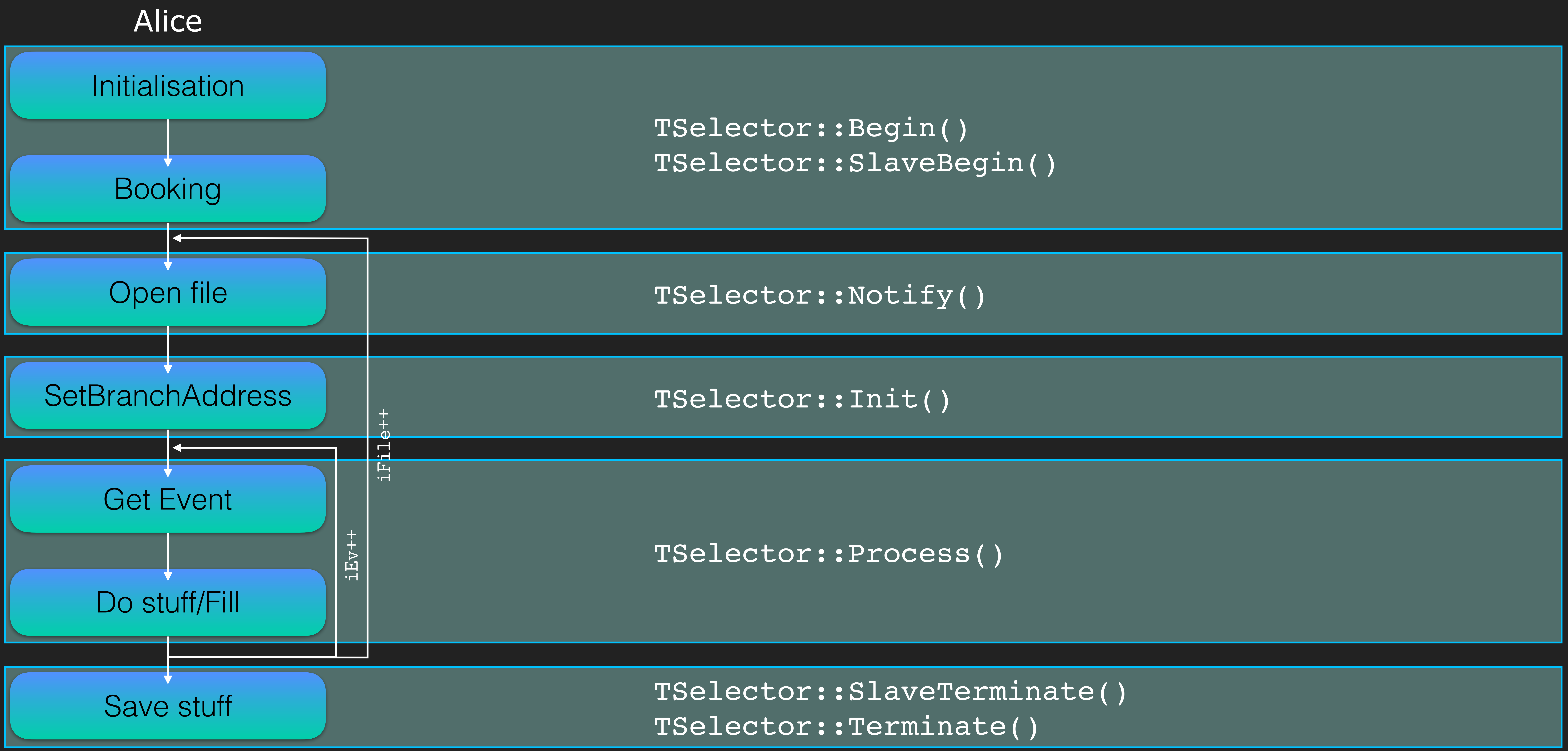


The **TSelector** class in ROOT already offers this kind of **modular** approach to data analysis.

It is still overlooked since it is not really flexible and it adds a non-negligible code overhead to the user, and offers few benefits at the single-user/single-task level.

However it offers some functionality that make it a good candidate for a **shared** approach (either **multi-user/multi-task** or just **single-user/multi-task**) since it provides us with a clear **point of insertion** for user code.

## DON'T REINVENT THE WHEEL

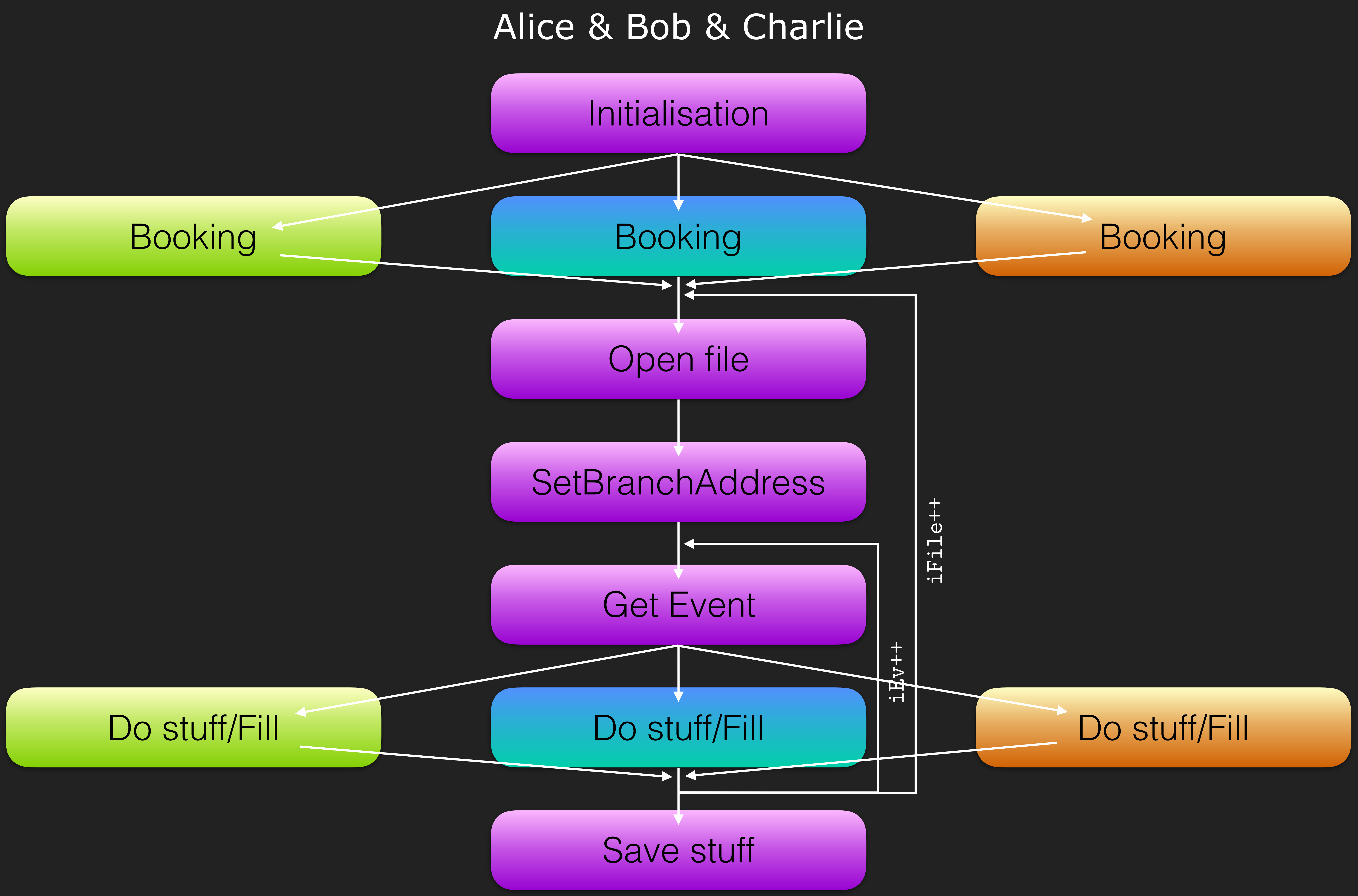


## CREATING A MANAGER

The **TSAnaPluginManager** class is the one performing all the tasks in purple

Users will write their analysis code in classes inheriting from a base class **TSAnaPlugin**

The manager will load all the required plugins and in turn call each plugin **booking** and **process** methods



## IMPLEMENTING A PLUGIN

Both base and manager class inherit from **TSelector** and from our **Event** class

```
class TSAnaPlugin : public TSelector, public Event, public TNamed {
public:
  //...
  TSAnaPlugin(TTree * /*tree*/ = 0)
      : fChain(0), rat(1), _pManager(nullptr), _debug(false), _idir(-1){};
  virtual ~TSAnaPlugin(){};
  virtual Int_t Version() const { return 2; };
  virtual void Begin(TTree *tree);
  virtual void SlaveBegin(TTree *tree);
  virtual void Init(TTree *tree);
  virtual Bool_t Notify();
  virtual Bool_t Process(Long64_t entry);
  virtual Int_t GetEntry(Long64_t entry, Int_t getall = 0) {
    return fChain ? fChain->GetTree()->GetEntry(entry, getall) : 0;
  }
  virtual void SetOption(const char *option) { fOption = option; }
  virtual void SetObject(TObject *obj) { fObject = obj; }
  virtual void SetInputList(TList *input) { fInput = input; }
  virtual TList *GetOutputList() const { return fOutput; }
  virtual void SlaveTerminate();
  virtual void SetRatio(float rr) { rat = rr; }
  virtual void Terminate();
  virtual TTree *GetChain();
```

## MANAGING A PLUGIN

The approach is to have each plugin compiled into a **shared library**, which is then loaded by the plugin manager and enlisted for execution.

This is to avoid having everything in a single program that won't compile if a user forgot a semicolon… 😅

No checks are performed at runtime. A possible solution to avoid bad code execution would be to have a *dry run* before running on the full dataset to validate each plugin, soon to be implemented.

```cpp
bool TSAnaPluginManager::AddPluginFromLibrary(const char *library_path) {

  // load the plugin library
  void *pluginlib = dlopen(library_path, RTLD_LAZY);
  if (!pluginlib) {
    cerr << "Cannot load library: " << dlerror() << '\n';
    return false;
  }

  // reset errors
  char *dlsym_error = dlerror();

  // load the symbols
  create_t *create_plugin = (create_t *)dlsym(pluginlib, "create");
  dlsym_error = dlerror();
  if (dlsym_error) {
    cerr << "Cannot load symbol create: " << dlsym_error << '\n';
    return false;
  }

  _constructors.push_back(create_plugin);

  destroy_t *destroy_plugin = (destroy_t *)dlsym(pluginlib, "destroy");
  dlsym_error = dlerror();
  if (dlsym_error) {
    cerr << "Cannot load symbol destroy: " << dlsym_error << '\n';
    return false;
  }

  _destructors.push_back(destroy_plugin);

  _plugins.emplace_back(create_plugin()); // call plugin constructor

  return true;
};
```

## MANAGING A PLUGIN

A plugin can be run in **managed mode** (if loaded from a manager) or in standalone mode (if the user wants to run his analysis by himself…)

In managed mode all the plugin data structures are pointing to the manager ones, and the manager is responsible for loading and filling the data.

```
void TSAnaPlugin::Init(TTree *tree){
  if( _pManager ){
   if (_debug)
      printf("[TSAnaPlugin::Init] - Working in plugin mode, getting branches "
             "from plugin manager \n");

    fChain = _pManager->fChain;
    if( _pManager->SHeader )  SHeader  = _pManager->SHeader;
    if( _pManager->Header )   Header   = _pManager->Header;
    if( _pManager->MCHeader ) MCHeader = _pManager->MCHeader;
    if( _pManager->Trd )      Trd      = _pManager->Trd;
    if( _pManager->Tof )      Tof      = _pManager->Tof;
    if( _pManager->Tracker )  Tracker  = _pManager->Tracker;
    if( _pManager->Rich )     Rich     = _pManager->Rich;
    if( _pManager->Ecal )     Ecal     = _pManager->Ecal;
    if( _pManager->Anti )     Anti     = _pManager->Anti;
    if( _pManager->SA )       SA       = _pManager->SA;
  } else {
   if (_debug)
      printf("[TSAnaPlugin::Init] - Working in standalone mode, getting "
             "branches from file\n");

    if( !tree ) return;
    fChain = tree;
    if( fChain->GetBranch("SHeader") )  fChain->SetBranchAddress( "SHeader",  &(SHead
    if( fChain->GetBranch("Header") )   fChain->SetBranchAddress( "Header",   &(Heade
    if( fChain->GetBranch("MCHeader") ) fChain->SetBranchAddress( "MCHeader", &(MCHea
    if( fChain->GetBranch("Trd") )      fChain->SetBranchAddress( "Trd",      &(Trd)
    if( fChain->GetBranch("Tof") )      fChain->SetBranchAddress( "Tof",      &(Tof)
    if( fChain->GetBranch("Tracker") )  fChain->SetBranchAddress( "Tracker",  &(Track
    if( fChain->GetBranch("Rich") )     fChain->SetBranchAddress( "Rich",     &(Rich)
    if( fChain->GetBranch("Ecal") )     fChain->SetBranchAddress( "Ecal",     &(Ecal)
    if( fChain->GetBranch("Anti") )     fChain->SetBranchAddress( "Anti",     &(Anti)
    if( fChain->GetBranch("SA") )       fChain->SetBranchAddress( "SA",       &(SA) )
    rti.clear();
  }
```

## MANAGING A PLUGIN

The manager then loads the data for current event and then calls each plugin Process() before moving to the next event.

A set of stopwatches measures runtime of a few key calls, especially:

- TSAnaPluginManager::GetEntry
- TSAnaPlugin::Process

```
Bool_t TSAnaPluginManager::Process(Long64_t entry) {
  _bench->Start("PluginManager::GetEntry");
  GetEntry(entry);
  _bench->Stop("PluginManager::GetEntry");

  rti_info = nullptr;
  unsigned int utime = SHeader->utime;
  if (rti.find(utime) != rti.end()) {
    rti_info = rti[utime];
  } else {
    if (_debug)
      printf("[TSAnaPluginManager::Process] - No RTI info for second %i \n",
             utime);
  }

  filemc_info = nullptr;
  unsigned int run = SHeader->run;
  if (filemc.find(run) != filemc.end()) {
    filemc_info = filemc[run];
  } else {
    if (_debug)
      printf("[TSAnaPluginManager::Process] - No FileMCInfo info for run %i \n",
             run);
  }

  Bool_t retValue = kTRUE, tempret;

  for (auto &pl : _plugins) {
    _bench->Start((std::string)((TNamed *)pl.get())->GetName() + "::Process");
    tempret = pl->Process(entry);
    retValue = retValue && tempret;
    _bench->Stop((std::string)((TNamed *)pl.get())->GetName() + "::Process");
  }

  return retValue;
}
```

## SAVING RESULTS

Users declare histograms and trees they want to save to the plugin manager, which then **manages creation and fill** of such objects.

TTrees are created as disk-resident, while memory usage checks will be introduced to avoid users allocating many huge histograms (on one of the first tests a user managed to book 7Gb worth of histograms... 😅)

Histograms and trees are **accessed by name**. To avoid name collision between different users, the plugin name is combined with the histogram name to make it unique.

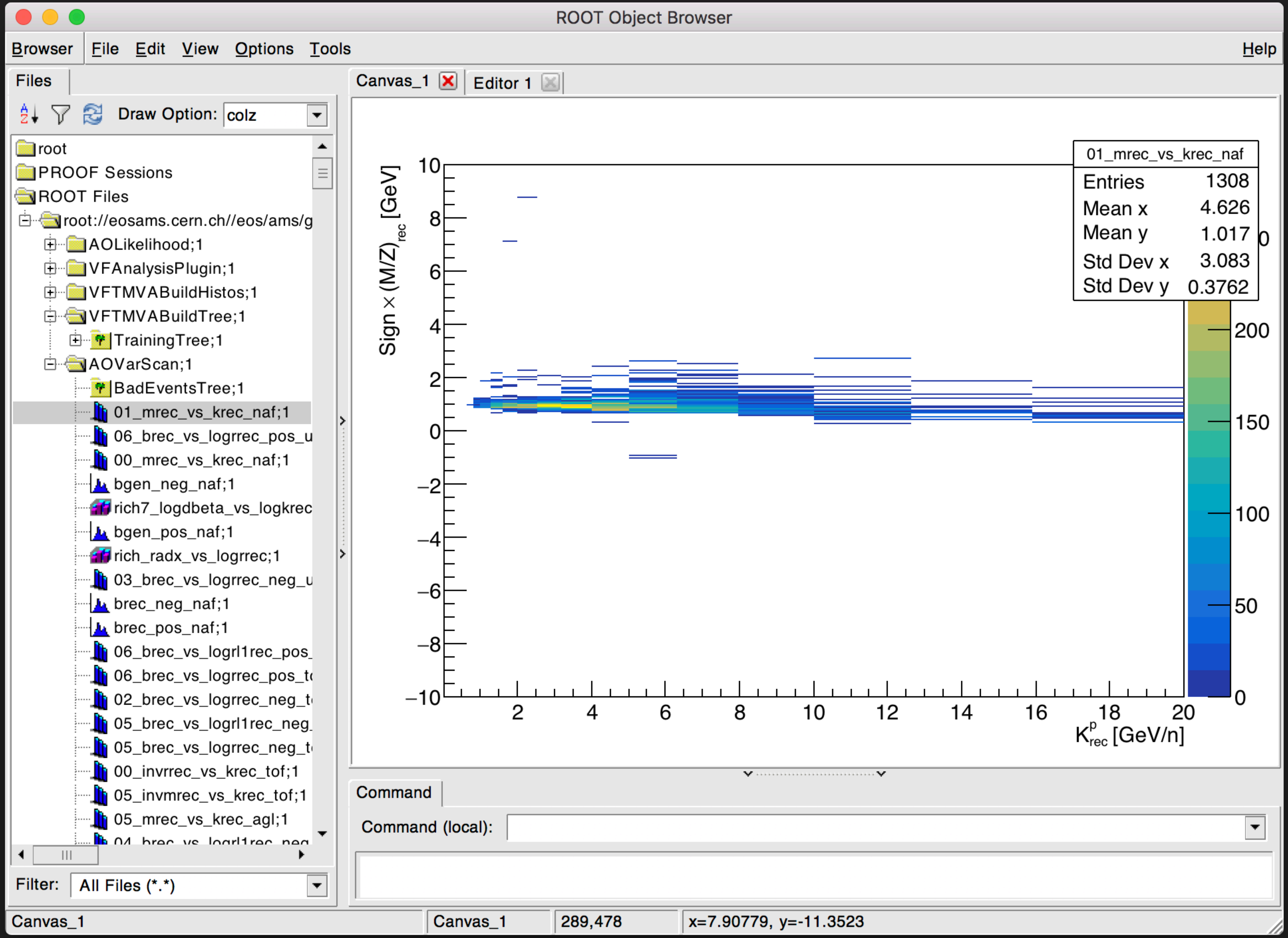
```
void TSAnaPlugin::AddHisto(TH1* hist){
  if(_debug) printf("[TSAnaPlugin::AddHisto] - Enter \n");

  hist->SetName( TranslateObjectName(hist).c_str() );
  hman->Add( hist );
};

void TemplatePlugin::BookHistos(){
  AddHisto( new TH1D("LiveTimeF", ";lf;Counts", 100, 0., 1.) );
  AddHisto( new TH1D("nParticle", ";nP;Counts", 6, -0.5, 5.5) );
  AddHisto( new TH1D("nTrTrack" , ";nTrack;Counts", 10, -0.5, 9.5) );
}

template <typename T, typename... Vals>
void TSAnaPlugin::Fill(T&& hname, Vals... vals){
  hman->Fill(TranslateObjectName(std::forward<T>(hname)), vals...);
};

Bool_t TemplatePlugin::Process(Long64_t entry){
  TSAnaPlugin::Process( entry ); //preparation

  //===========================================================
  //Do your Process stuff here, this will be your analysis code
  // example:
  //Histogram fill example
  Fill("nParticle", Header->nparticle);
  Fill("nTrTrack" , Header->ntrtrack);
  //===========================================================

  return kTRUE;
};
```

## SAVING RESULTS

The plugin output is saved in a **dedicated TDirectoryFile** inside the output TFile object.

User data directories will then be separated in different files after the production is finished.

This allows **easier file management in batch jobs** (no need to know how many plugins are running/which files to transfer).



Draw Option: colz

```
root
PROOF Sessions
ROOT Files
  root://eosams.cern.ch//eos/ams/g
    AOLikelihood;1
    VFAnalysisPlugin;1
    VFTMVABuildHistos;1
    VFTMVABuildTree;1
      TrainingTree;1
    AOVarScan;1
      BadEventsTree;1
      01_mrec_vs_krec_naf;1
      06_brec_vs_logrrec_pos_u
      00_mrec_vs_krec_naf;1
      bgen_neg_naf;1
      rich7_logdbeta_vs_logkrec
      bgen_pos_naf;1
      rich_radx_vs_logrrec;1
      03_brec_vs_logrrec_neg_u
      brec_neg_naf;1
      brec_pos_naf;1
      06_brec_vs_logrl1rec_pos_
      06_brec_vs_logrrec_pos_t
      02_brec_vs_logrrec_neg_t
      05_brec_vs_logrl1rec_neg_
      05_brec_vs_logrrec_neg_t
      00_invrrec_vs_krec_tof;1
      05_invmrec_vs_krec_tof;1
      05_mrec_vs_krec_agl;1
```

Filter: All Files (*.*)

```
01_mrec_vs_krec_naf
Entries      1308
Mean x      4.626
Mean y      1.017
Std Dev x   3.083
Std Dev y  0.3762
```

Command (local):

Canvas_1 | Canvas_1 | 289,478 | x=7.90779, y=-11.3523

## SHARING MORE WORK

A set of **most common parametric selections** used in the analysis is already defined in a dedicated library.

To allow flexibility cuts are defined as **named lambda functions** with a default set of parameters. This allow **full-customisation from users**, who can also create their own selections on the fly.

The first plugin that asks for the result of a selection triggers the condition check, the result for the current event and the required parameters values is then cached.

```
class CutAction {

public:
  CutAction( //full constructor
    std::string name,                                      // cut name
    std::vector<double> defParams,                         // default parameters
    std::function<bool(Event*, std::vector<double>)> cut // lambda function
  ) : _name(name), _defParams(defParams), _cut(cut) {}

  virtual ~CutAction(){};

  std::string GetName(){ return _name; }
  std::vector<double> GetDefParams() { return _defParams; }

  bool Check( Event* Ev, std::vector<double> params ){ return _cut(Ev, params);

private:
  std::string _name;
  std::vector<double> _defParams;
  std::function<bool(Event*, std::vector<double>)> _cut;
};
```

## SHARING MORE WORK

A set of most common parametric selections used in the analysis is already defined in a dedicated library.

The main interface to cut actions are cut instances. Instances are a **many to one** map to actions, you could have two instances using the same action (each with different parameters) in your selections.

Instances come with a proxy which **handles the caching** of the action result.

```
class CutInstance {
public:
  CutInstance(std::string name, std::string cutname);
  virtual ~CutInstance(){};
 bool Check(Event *Ev) { return _cutProxy.Check(Ev, _cutAction); };
  void SetParameters(const std::vector<double> params) {
    _cutProxy.SetParameters(params);
  };
  void SetParameter(const unsigned int nParam, double value) {
    _cutProxy.SetParameter(nParam, value);
  };

private:
  std::string _name;
  CutProxy _cutProxy;
  std::shared_ptr<CutAction> _cutAction;
};

bool CutProxy::Check(Event* Ev, std::shared_ptr<CutAction> cutAction){
  if( Ev->SHeader->run != lastRun || Ev->SHeader->event != lastEv ){
    lastRun = Ev->SHeader->run;
    lastEv  = Ev->SHeader->event;
    cachedResults.clear();
  }
  if( cachedResults.find( _params ) != cachedResults.end() ) {
    lastResult = cachedResults[_params];
  } else {
    lastResult = cutAction->Check(Ev, _params);
    cachedResults.emplace( _params, lastResult );
  }
  return lastResult;
};
```

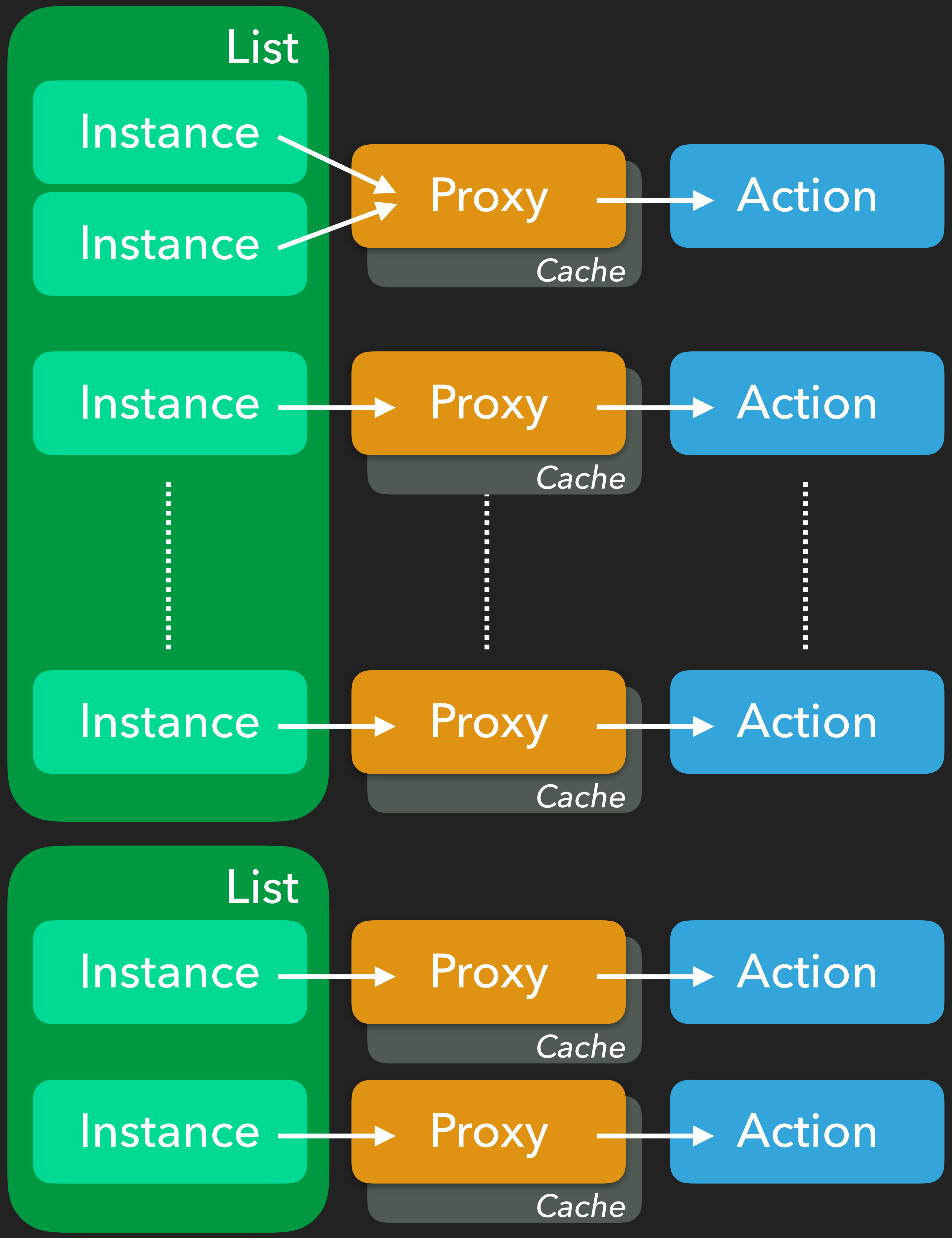
## SHARING MORE WORK

A set of most common parametric selections used in the analysis is already defined in a dedicated library.

The main interface to cut actions are cut instances. Instances are a **many to one** map to actions, you could have two instances using the same action (each with different parameters) in your selections.

Instances come with a proxy which **handles the caching** of the action result.

**Cut lists** can be used to **chain instances** with other instances or lists.



## SHARING MORE WORK

A set of most common parametric selections used in the analysis is already defined in a dedicated library.

Each plugin holds a CutMgr object that handles the logic behind the list/instance check.

Users can exclude on the fly any number of instances when checking a particular list.

```cpp
template <typename T>
bool CutMgr::CheckSelection(Event *Ev, T &&selname,
                                             std::vector<std::string> excluded) {
  static std::string debugSelname;
  if (_debug)
    debugSelname = selname;

  bool result = true;

  if (std::find(excluded.begin(), excluded.end(), selname) != excluded.end()) {
    return true;
  }

  ResPair res = Find(std::forward<std::string>(selname));

  if (res.first && res.second) {
    std::cerr << "[CutMgr::CheckSelection] Fatal Error. Skipping" << std::endl;
    return true;
  }

  if (res.first) {
    for (auto sel : res.first->GetList()) {
      result &= CheckSelection(Ev, std::move(sel), excluded);
    }
  } else if (res.second) {
    result &= res.second->Check(Ev);
  }

  return result;
}
```

## SHARING MORE WORK

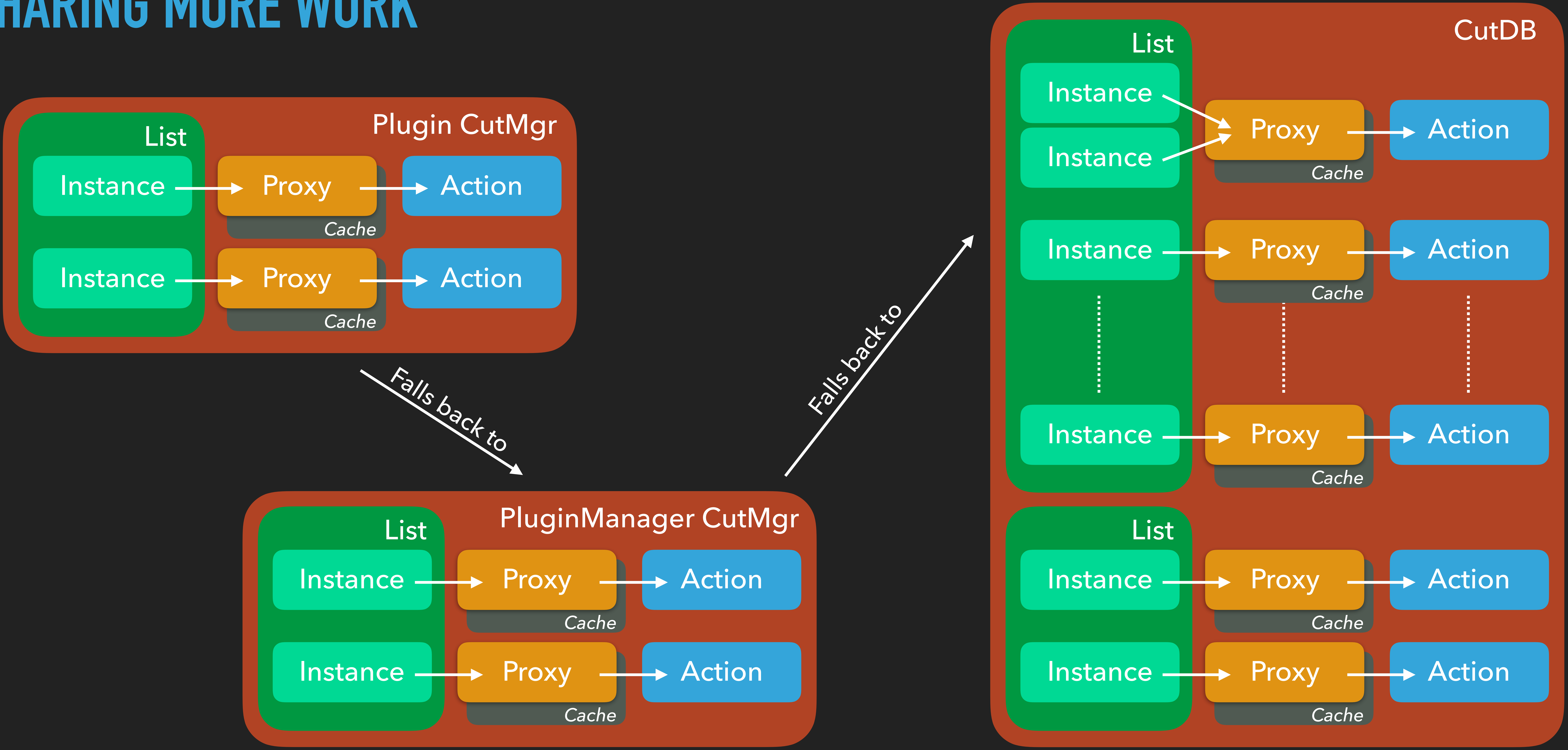
A set of most common parametric selections used in the analysis is already defined in a dedicated library.

Each plugin holds a CutMgr object that handles the logic behind the list/instance check.

The “local” CutMgr holds all **user-created cuts**, it falls back to the PluginManager CutMgr if the cut is not found locally, and finally it falls back to a **preloaded set of cuts** (CutDB) which holds the **most common selections used for the analysis**.

```
ResPair CutMgr::Find(std::string selname) {
  auto selRes = _selStore.find(selname);
  if (selRes != _selStore.end()) {
    return std::make_pair(selRes->second, nullptr);
  }

  auto instRes = _instStore.find(selname);
  if (instRes != _instStore.end()) {
    return std::make_pair(nullptr, instRes->second);
  }

  if (_fallbacks.size()) {
    for (auto mgr : _fallbacks) {
      auto res = mgr->Find(selname);
      if (res.first || res.second)
        return res;
      else {
        std::cerr << "[CutMgr::Find] Error - Selection " << selname
                  << " not present in the main DB." << std::endl;
      }
    }
  } else {
    auto res = CutDB::Head().Find(selname);
    if (res.first || res.second)
      return res;
    else {
      std::cerr << "[CutMgr::Find] Error - Selection " << selname
                << " not present in the main DB." << std::endl;
    }
  }

  return std::make_pair(nullptr, nullptr);
}
```

## SHARING MORE WORK



## CONCLUSIONS

- A framework is being developed within the AMS dbar analysis group to **encapsulate code from each user** and **share all the I/O operations**.
- The framework leverages the **TSelector** structure to encapsulate code into **dynamically linked libraries** that can be **loaded at runtime** by a plugin manager that loads the data and runs each plugin on said data.
- Users can **easily define and fill histograms and trees** which are then managed and saved by the plugin manager.
- Users can **apply parametric selections to each event**, and for each unique selection **the result is calculated only once and then cached** for following calls.
- The aim of this work is to setup **nightly analysis trains** to allow for day-scale iteration on the analysis and **optimise user time** (you come to the office, look at the latest results, debug all day, finally push your new plugin to the repo and go home, rinse and repeat)
- More feature can and will be implemented (**performance checks, web dashboard, and so on…**)