ATLAS Job Transforms: A data driven workflow engine

Graeme Stewart
for the ATLAS collaboration

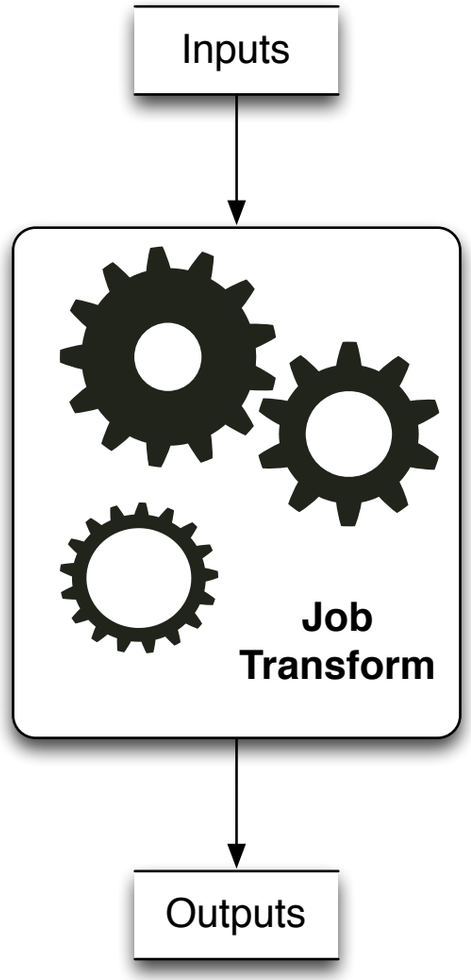
University of Glasgow | Department of Physics & Astronomy

Overview

- What are Job Transforms (and why?)
- A history of the framework
- A New Start
  - Design Principles
  - Multi-step workflows
  - Flexibility and Enhancements
- Next steps

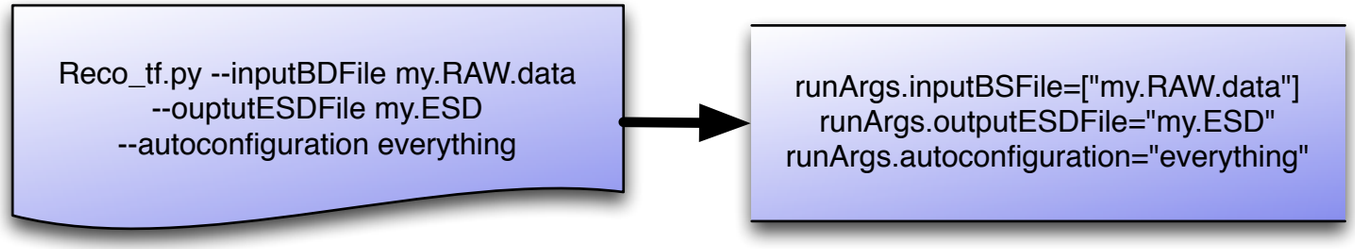
Job Transforms

- A job transform is a script that 'transforms' one type of data into another
  - e.g., RAW data into AOD
- Takes care of configuring the offline software appropriately
- Manages input and output files
- Provides a robust reporting framework for job outcomes



Software Configuration

- ATLAS offline software, Athena, uses the Gaudi framework
- Jobs are configured using a python layer, which is elegant, but can become very complex
- Job transforms provide a way to convert command line into job options:



- They also simplify job configuration using a job options *template*, which is tuned to a particular transform
  - e.g. skeleton.RAWtoESD_tf.py

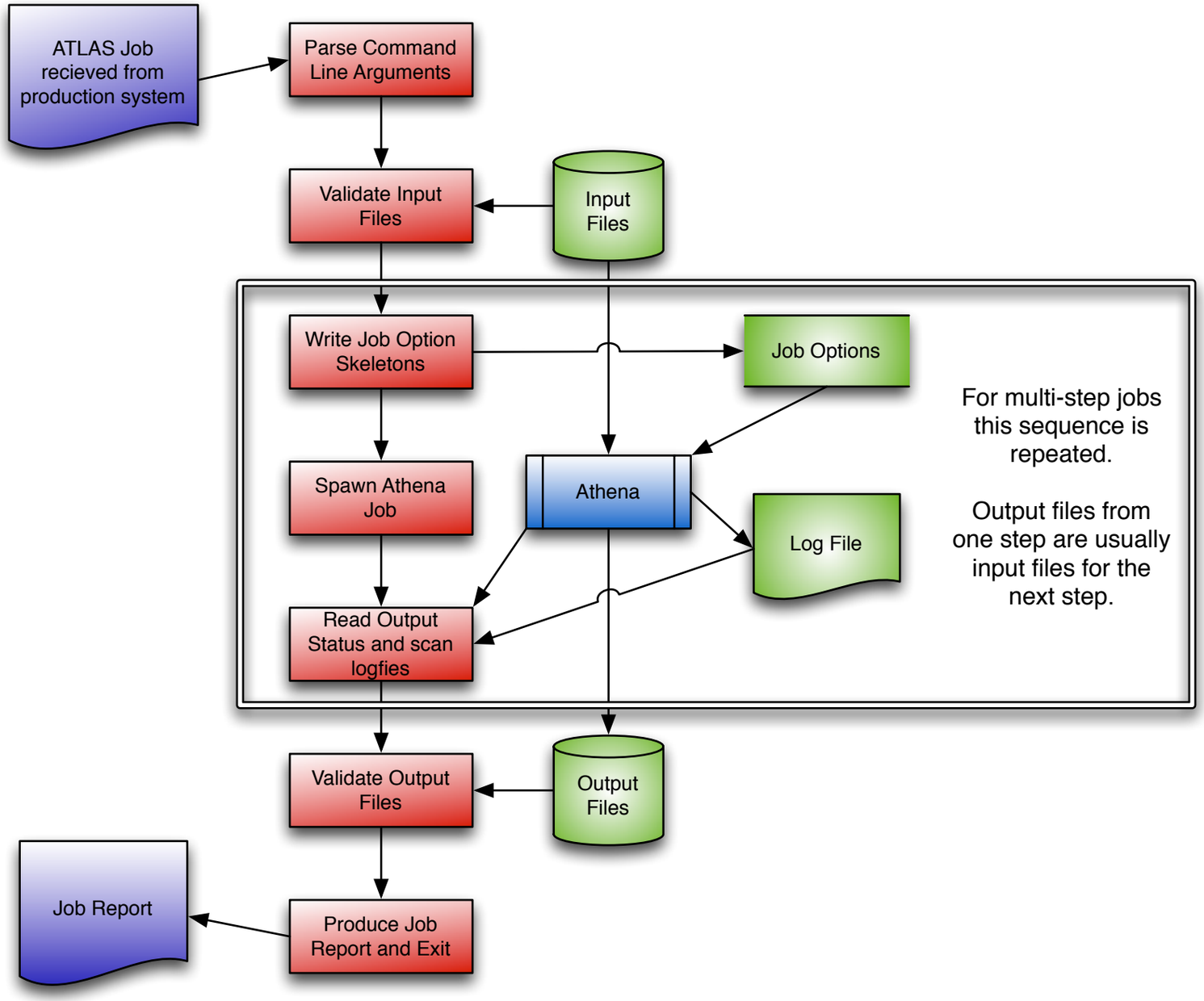
File Validation

- Ensuring that input and output files are valid in the transform provides robustness and better diagnostics
  - For input files fast checks are made, e.g., event counting
  - For output files deeper checks are made, unpacking ROOT baskets
    - This helps find rarer corruption after write problems

Metadata

- Book-keeping is an essential part of any production system, so information must flow upstream describing:
- Job level Information
  - Such as job configuration (including all substeps)
  - A clear exit status and error message
- File level Information
  - What types of files were produced
  - The number of events in each file
  - Other pertinent data, such as geometry and conditions tags

Workflow


ATLAS Job recieved from production system
Parse Command Line Arguments
Validate Input Files
Input Files
Write Job Option Skeletons
Job Options
Spawn Athena Job
Athena
Log File
Read Output Status and scan logfies
For multi-step jobs this sequence is repeated.
Output files from one step are usually input files for the next step.
Validate Output Files
Output Files
Job Report
Produce Job Report and Exit

A brief history of... transforms

- Original shell script version
  - Rather limited in scope, but proved utility
- Second better designed version in python
  - Lots of features
  - Rather elaborate implementation
  - Ultimately very successful for ATLAS
    - Millions of jobs run at Tier-0 and on the grid

Issues in the Old Framework

- Though the second, python, framework was successful the original developers moved on
  - Lots of problems were then addressed by ad-hoc extensions
    - Which led to tangled and poorly understood code
      - Too many implicit steps
      - Anachronistic code segments
      - No built-in support for multi-step transforms (and this is the critical use case!)

Review Process

- ATLAS reviewed the transform framework in 2012
- We decided to keep it - it's useful
  - Especially multi-step transforms
- But we decided to embark on a new implementation
  - Ameliorate the risk of making major changes to the old framework during data taking
  - Provide better support for the multi-step use case
  - Develop the new transforms in parallel, then gradually deploy them

New Transforms: First Principles

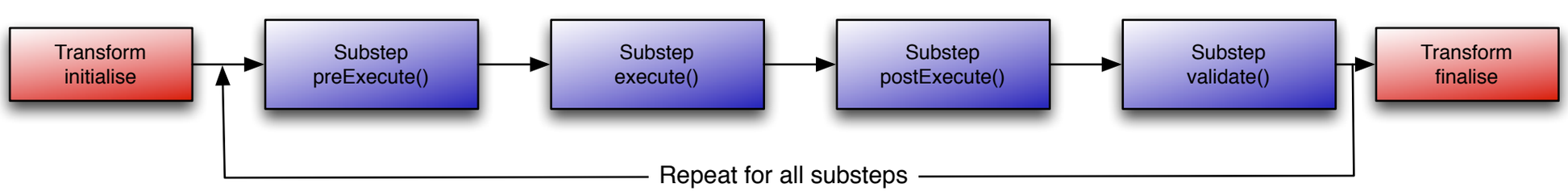
- Make the workflow *explict*
  - Clearer code, easier to understand and maintain
- Solve actual problems, not possible ones
- Use standard python modules where ever possible (like `logging`)
  - And extend on top if necessary (`argparse`)
- Write lots of unit and end to end tests

Errors and Failures

- Execution can fail in many different ways, for many different reasons
- We have to plan for this and program expecting it
- Use exceptions to transmit errors through the framework
- Transform exceptions have two attributes:
  - exitCode - becomes the return code if this exception is unrecoverable
  - exitMessage - detailed message with all useful information
- Failing gracefully and transmitting information upstream is very important!

Workflow

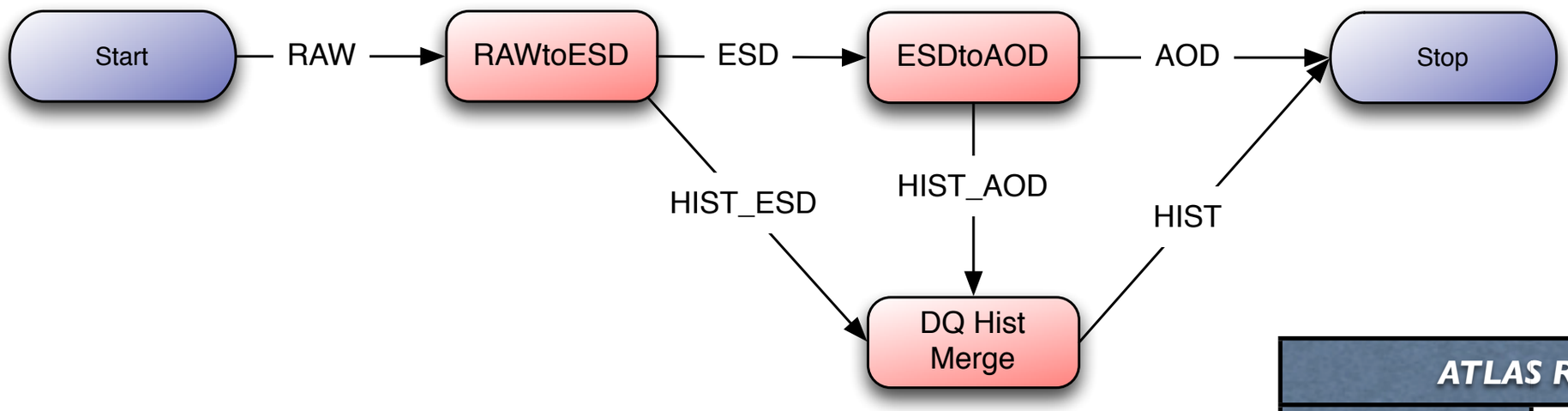
- Take an analogous workflow to event processing
  - initialise(), eventLoop(), finalise()
- But instead of events we process substeps
- And each substep has it's own workflow:
  - preExecute(), execute(), postExecute(), validate()



Multi-step Jobs

- In these jobs data typically is produced by one substep, then used as input to the next
- This gives rise to a natural representation of a job as a graph
  - Nodes are substeps
  - Data types are vertexes
- Such graphs are directed and acyclic (DAGs)
- Input data comes from a special 'start' node, output data flows to a special 'stop' node

Basic Reconstruction Job



ATLAS Reconstruction	
RAW	Detector data
ESD	Complete Reconstruction Data
AOD	Physics Analysis Input Data
HIST	Data Quality Histograms

```
$ Reco_tf.py --inputRAWFile foo
     --outputAODFile bar
     --outputHISTFile baz
```

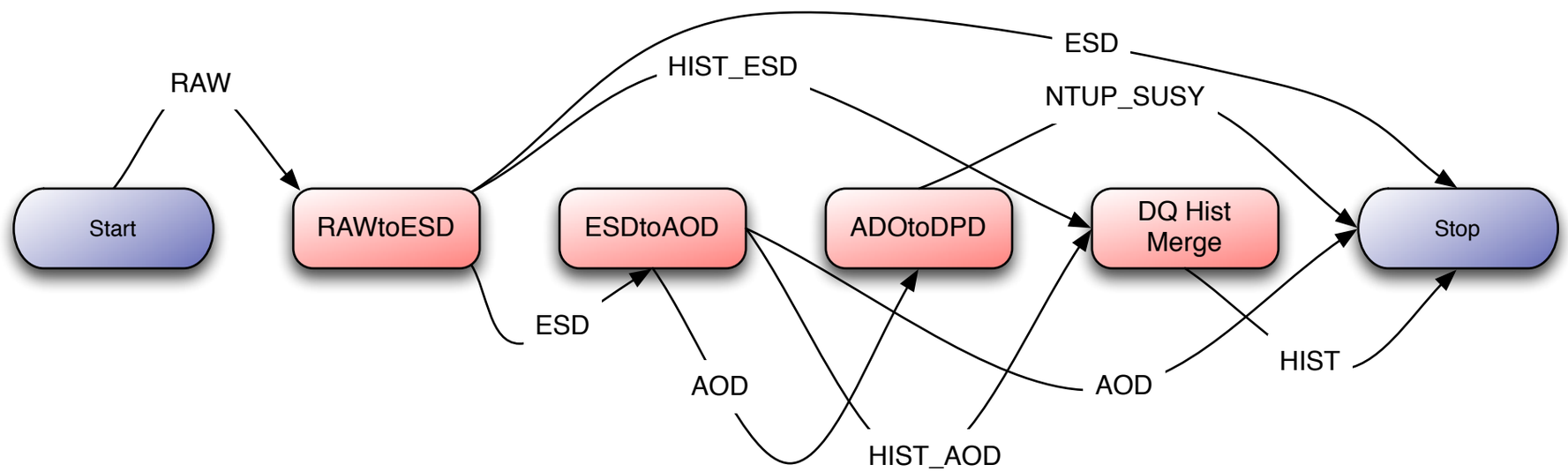
- Transform makes AOD and HIST as requested outputs
- ESD, HIST_ESD, HIST_AOD are ephemeral, but made automatically to derive the necessary outputs

How it works

- Note this is not a traditional graph path problem:
- The transform is not tracing one single path and looking for the lowest cost
- Instead we trace a path for every data type we require (final and ephemeral)
  - Assume that the cost of executing a substep is independent of how many data types it produces
  - So any already activated substep is then 'free' for additional data types
  - We need to minimise the sum cost of these paths

Solution I

- First do a topological ordering of the graph
  - So that all edges point left to right



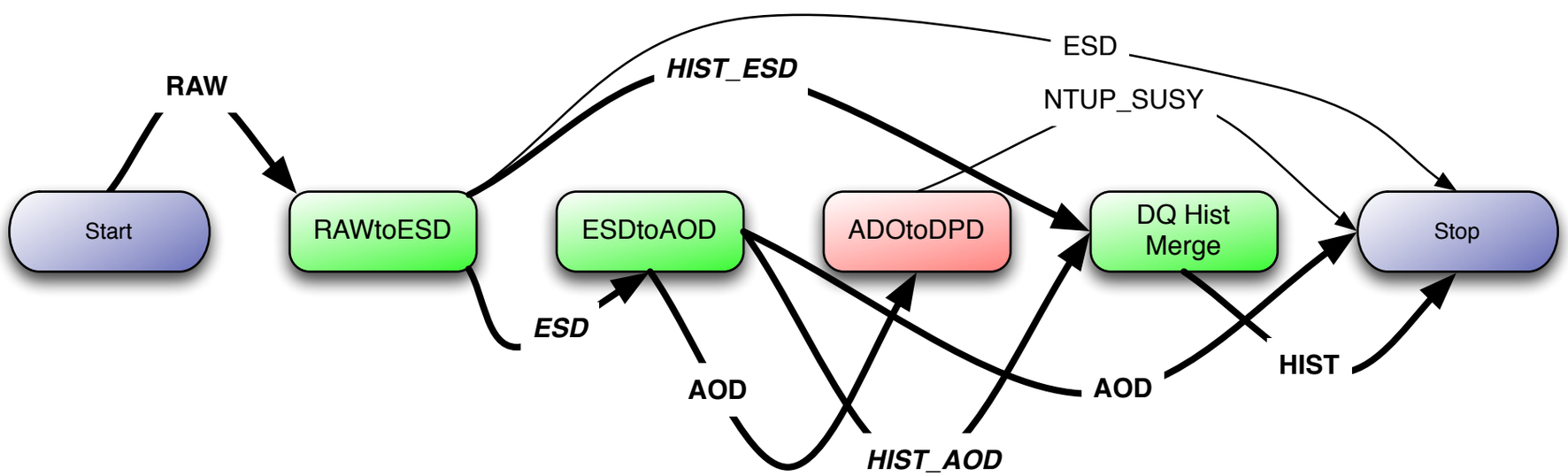
- This also orders data types topologically, according to their origin

Solution II

- Add input data to the *available* list
- Add output data to the *required* list
- Order the required list, considering the data that can be produced earliest first
  - For each required piece of data trace a path to some available data, picking up new data types as necessary
  - Activate all substeps passed through, if not already done
  - Add this data type to the *available* list
  - Add any new data types to the *required* list

Solved

- Once all data is *available* the algorithm can stop
- Now all activated substeps are known
  - And all data a substep needs to produce is known



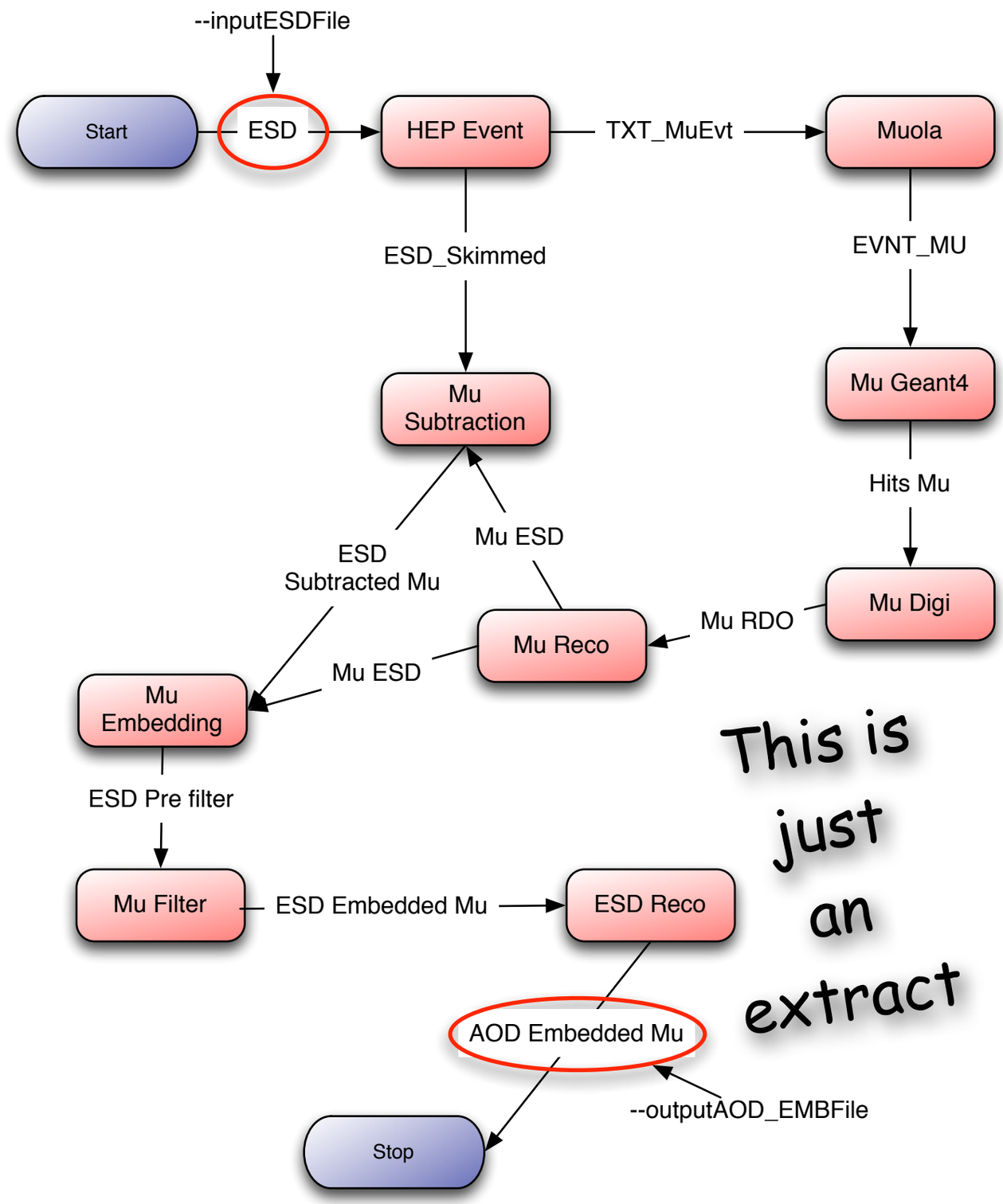
Developer's View

- One definite goal was to have the execution graphs easy to use
- All a developer needs to specify is the input and output data types for a substep
- Then the transform infrastructure will do all the rest

```
executorSet.add(athenaExecutor(name = 'RAWtoESD', skeletonFile = 'RecJobTransforms/skeleton.RAWtoESD_tf.py',
                               substep = 'r2e', inData = ['BS', 'RDO'], outData = ['ESD', 'HIST_ESD'],
                               perfMonFile = 'ntuple_RAWtoESD.pmon.gz'))
executorSet.add(athenaExecutor(name = 'ESDtoAOD', skeletonFile = 'RecJobTransforms/skeleton.ESDtoAOD_tf.py',
                               substep = 'e2a', inData = ['ESD'], outData = ['AOD', 'HIST_AOD'],
                               perfMonFile = 'ntuple_ESDtoAOD.pmon.gz'))
executorSet.add(DQMergeExecutor(name = 'DQHistogramMerge', inData = [('HIST_ESD', 'HIST_AOD')], outData = ['HIST']))
executorSet.add(athenaExecutor(name = 'ESDtoDPD', skeletonFile = 'PATJobTransforms/skeleton.ESDtoDPD_tf.py',
                               substep = 'e2d', inData = ['ESD'], outData = ['NTUP'],
                               perfMonFile = 'ntuple_ESDtoDPD.pmon.gz'))
executorSet.add(athenaExecutor(name = 'AODtoDPD', skeletonFile = 'PATJobTransforms/skeleton.AODtoDPD_tf.py',
                               substep = 'a2d', inData = ['AOD', 'EVNT'], outData = ['NTUP'],
                               perfMonFile = 'ntuple_AODtoDPD.pmon.gz'))
executorSet.add(athenaExecutor(name = 'AODtoTAG', skeletonFile = 'RecJobTransforms/skeleton.AODtoTAG_tf.py',
                               inData = ['AOD'], outData = ['TAG'],))
```

Scaling

- The most complicated transform we wrote to date was to embed simulated $H \rightarrow \tau\tau$ events into real data di-muon events
- This transform has multiple flavours of AOD outputs and 60 different substeps
- The graph tracer runs in less than 200ms, c.f., a job runtime of ~20 hours



Special Workflows and Odd Tricks

- The use of substep executor classes with separate preExecute(), execute(), postExecute(), validate() steps is very powerful
- New substep executors can be written with specialist methods for any of these steps
- Cleanly separate special cases from core behaviour, e.g.
  - Setup an old release within a job to rerun an older version of the trigger
  - Special 'Goldilocks' merging options when Athena runs in multi-process mode, producing multiple output files

Summary

- Job Transforms are a vital part of ATLAS production at Tier-0 and on the grid
- A new transform infrastructure has been written that is
  - Simplified and clearer, with better reporting
  - Has powerful support for multi-step transforms
  - Provides an easy way to customise jobs and workflows
- Many new transforms are already in production
  - The rest will come during the LHC's long shutdown 1