Did we accidentally build a compiler for HEP workflows?

FAST-HEP Flow

from declarative workflows to executable plans

Luke Kreczko

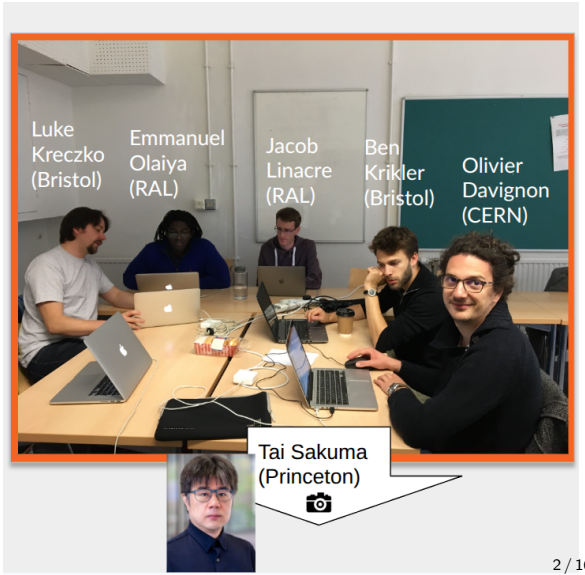
University of Bristol

PyHEP.dev 2026

F.A.S.T

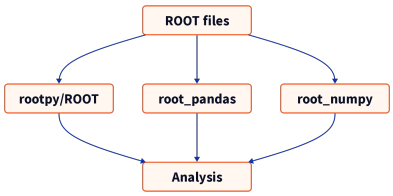
Faster Analysis Software Taskforce

- Started around 2017 - The dark ages before `uproot`
- `root_numpy` and `root_pandas` were tempting us
- Short "hack-shops" to test ideas quickly



What were we trying to solve?

- New Python tools made array-based analysis increasingly attractive
- Adopting them required analysts to learn unfamiliar libraries, data models, and programming styles
- Most physicists wanted to analyse data - not rebuild their software stack every few years



Could we separate the analysis from the technology?

The first idea

```
stages:
  - BasicVars: fast_carpenter.define.Define
  - DiMuons: cms_hep_tutorial.DiObjectMass
  - EventSelection: fast_carpenter.selection.CutFlow
  - DiMuonMass: fast_carpenter.summary.BinnedDataframe

BasicVars:
  variables:
    - Muon_Pt: "sqrt(Muon_Px ** 2 + Muon_Py ** 2)"
    - IsoMuon_Idx: "(Muon_Iso / Muon_Pt) < 0.10"

DiMuons:
  mask: IsoMuon_Idx

EventSelection:
  selection:
    All:
      - NIsoMuon >= 2
      - triggerIsoMu24 == 1
```

Describe what to compute

- Analysis stages were declared in YAML
- Parameters described selections and derived quantities
- Python modules supplied the implementation
- Analysts did not need to rewrite the array-processing machinery

What we thought we had built

A declarative analysis language

It worked - until it did not

What worked

- Analyses became shorter and easier to read
- Scientists wrote less framework-specific code
- The same approach worked across several experiments
- New users could start from an analysis description

What did not

- Libraries and data representations evolved
- Configuration spread across many files
- Custom Python had no clear contracts
- Missing products failed late during execution
- “Replaceable” components remained tightly coupled
- Contributors and original authors moved on

Declarative syntax had moved the complexity - not removed it.

The next question

Scientific workflow

???

Execution

What should happen here?

- validate the workflow
- resolve dependencies
- determine required datasets and fields
- bundle the complete workflow into one representation
- prepare execution

Keep the part that worked

- Authors still describe the analysis declaratively
- YAML remains concise and approachable
- Scientific intent stays separate from implementation

But

The workflow description is no longer executed directly.

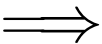
```yaml
analysis:
  stages:
    - id: BasicVars
      op: hep.define
      params:
        variables:
          - name: Muon_Pt
            expr: "sqrt(Muon_Px ** 2 + Muon_Py ** 2)"
    - id: DiMuons
      op: hep.di_object_mass
      params:
        collection: Muon
```

Concise for authors; explicit before execution.

First: assemble the complete workflow

Workflow components

- scientific workflow
- datasets
- styles
- output definitions
- profiles and registries

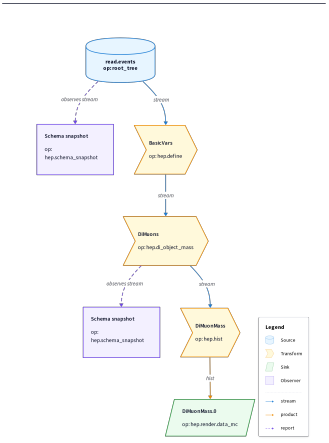


Normalised workflow

- includes assembled
- defaults materialised
- shorthand expanded
- implementations recorded
- one inspectable description

No more: “Which file supplied that correction?”

Then: make the data flow visible



- sources
- transforms
- sinks
- observers
- typed products

Why?

Everything gains a clear place in the workflow.

The graph made experimentation cheap

Instead of hard-coding...

- ROOT as the only input
- one event stream
- one histogram implementation
- one output format
- built-in profiling

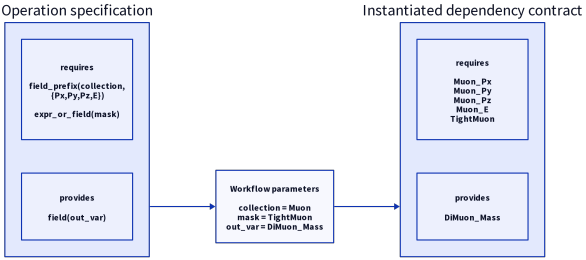
...everything became replaceable

- register another source
- add joins
- register another histogram product
- select another sink
- attach observers

Flow orchestrates capabilities.

Implementations become plugins.

Operations tell Flow what they need



Specifications declare

- parameters and defaults
- required symbols
- provided symbols

Flow can then

- infer source columns
- validate schemas
- fail before execution

Runtime surprise becomes compile-time feedback.

Do not bake product semantics into the framework

Before

- histogram data stored in multi-index pandas objects
- merging assumed by the framework
- slow for large analyses
- replacement required widespread changes

Now

- each product type supplies its merge behaviour
- the runtime invokes the registered contract
- no internal histogram representation is assumed
- try `hep.myFastHist` without changing Flow

Flexibility should make experimentation cheaper.

Finally: produce an execution plan

The plan records

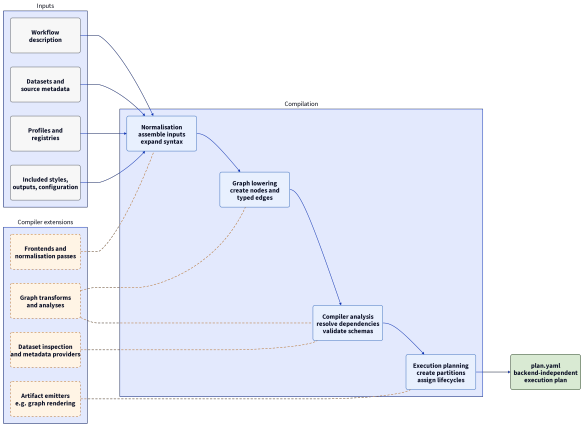
- graph nodes and ports
- selected implementations
- dataset partitions
- lifecycle scopes
- merge and materialisation policies

The runtime no longer decides

- what fields are required
- which stages apply
- how products compose
- when final sinks should run

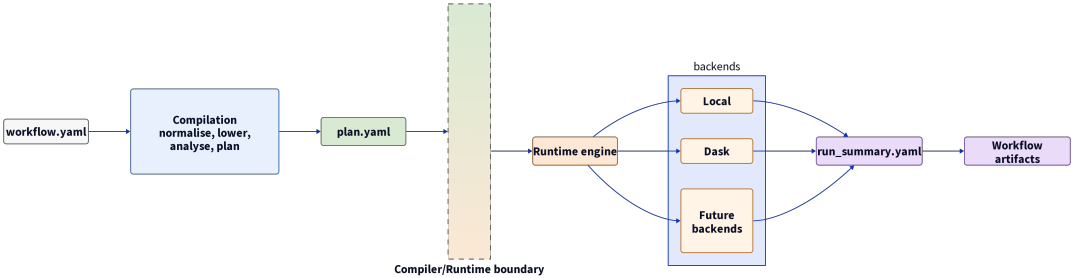
`workflow.yaml` $\Longrightarrow$ **`plan.yaml`**

Flow today



Looking back, this is much closer to our original vision.

The plan separates semantics from infrastructure



The same plan can run locally or through distributed backends.

Is this useful beyond FAST-HEP?

What it has given us

- earlier validation
- inspectable representations
- genuinely replaceable components
- simpler external packages
- traceable artifacts

Where I would go next

- distributed provenance
- additional execution backends
- caching and reuse
- further compiler analyses

Do these problems and architectural ideas
resonate with your projects?

Find out more at `https://fast-hep.github.io`
Paper (draft): `https://arxiv.org/abs/2608.18745`

Backup slides

Backup: multiple input sources

Backup: declarative did not mean unambiguous

A real analysis accumulated

- datasets
- corrections
- systematics
- styles and plots
- custom Python
- profiles
- submission scripts

Questions newcomers still had

- What is actually executed?
- Which file is the production entry point?
- Which defaults and corrections are active?
- Which implementation is selected?
- How do I inspect or debug the assembled workflow?

Compilation consolidates the inputs into one inspectable plan.

Backup: observers attach non-transforming analyses

Dataset inspection

- files
- entry counts
- available schemas
- planning metadata

Used during compilation.

Schema snapshots

- fields
- types
- partition context
- graph boundary

Used for inspection and validation.

Performance

- execution time
- memory
- node-level metrics
- partition behaviour

Used for profiling and diagnostics.

Observers inspect streams, products, or artifacts without changing them.

Backup: execution modifiers

Observers

- inspect execution
- emit reports or metadata
- do not change the value

Execution modifiers

- participate in execution
- wrap nodes or boundaries
- may change runtime behaviour

Examples

- preload data or models onto a GPU
- JIT-compile an operation before repeated execution
- introduce device-specific preparation or caching
- alter the execution environment without changing the workflow stage

Similar attachment model; deliberately different semantics.

Backup: dependency contracts

```
requires:
  symbols:
    - from: params.collection
      kind: field_prefix
      suffixes: [Px, Py, Pz, E]
    - from: params.mask
      kind: expr_or_field

provides:
  symbols:
    - from: params.out_var
      kind: field_list
```

With:

- `collection=Muon`
- `mask=TightMuon`
- `out_var=DiMuon_Mass`

Flow derives:

- `Muon_Px/Py/Pz/E`
- `TightMuon`
- provides `DiMuon_Mass`

Backup: execution lifecycle

$$\text{Partition} \Longrightarrow \text{Dataset} \Longrightarrow \text{Workflow}$$

Partition	Read and process one concrete unit of input data.
Dataset	Merge partition products and execute dataset-level work.
Workflow	Execute final rendering, reports, and publication.

Lifecycle decisions are encoded in the plan, not rediscovered by the runtime.

Backup: provenance structure

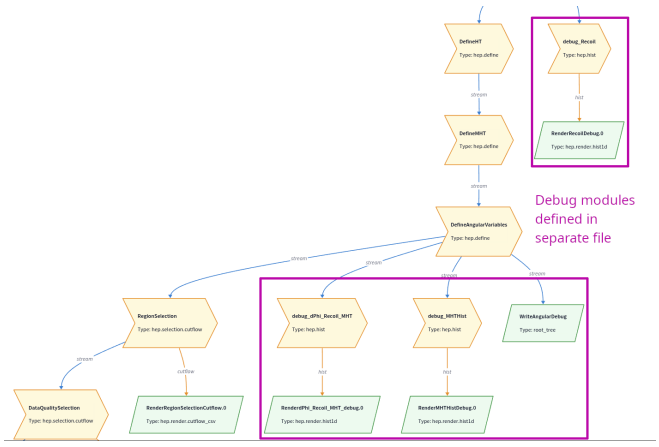
Execution	Manifest	Artifact record
workflow artifacts	artifact paths	producer execution
software versions	producer nodes	partition
runtime environment	record locations	immediate inputs
partitions	record hashes	artifact kind

From an artifact back to the workflow, data, software, and runtime conditions that produced it.

Backup: No more print statements—debugging made easy

Attach a diagnostic sink

```
- id: WriteAngularDebug
  role: sink
  op: root_tree
  needs: [DefineAngularVariables]
  params:
    path: angular_debug.root
    tree: events
    keep:
      - MHT_pt
      - MHT_px
      - MHT_py
      - MHT_phi
      - Recoil_pt
      - Recoil_phi
      - MHT_over_Recoil
      - TrkMET_pt
      - TrkMET_phi
      - dPhi_Recoil_MHT
      - dPhi_Recoil_TrkMET
  when: partition_end
```



Flow makes it easy to attach diagnostic workflows without modifying the existing analysis code.

Backup: Provenance makes implicit choices explicit

Recorded provenance


```
"cms.jetid.2024.winter24.ak4puppi.tight": {
  "kind": "correctionlib",
  "requested_era": "2024_Winter24",
  "selected": {
    "configured_path": ".../JME/2024_Winter24/jetid.json.gz",
    "resolved_path": ".../JME/2022_Summer22/jetid.json.gz",
    "symlink_target": "../2022_Summer22/jetid.json.gz",
    "correction": "AK4PUPPI_Tight",
    "correction_version": 2,
    "sha256": "2ae07e7b7c67332fa4ccc374c7b3cfe6..."
  }
}
```


What actually happened

- Requested jetID: `2024_Winter24`
- The configured 2024 file is a `symbolic link`.
- It resolves to the `2022 file`.
- The exact content is identified by its `SHA-256 checksum`.

The workflow records what was actually used - not merely what the analysis requested.