Preserving analyses: where decomposition into graphs helps

Follow up on Research Objects

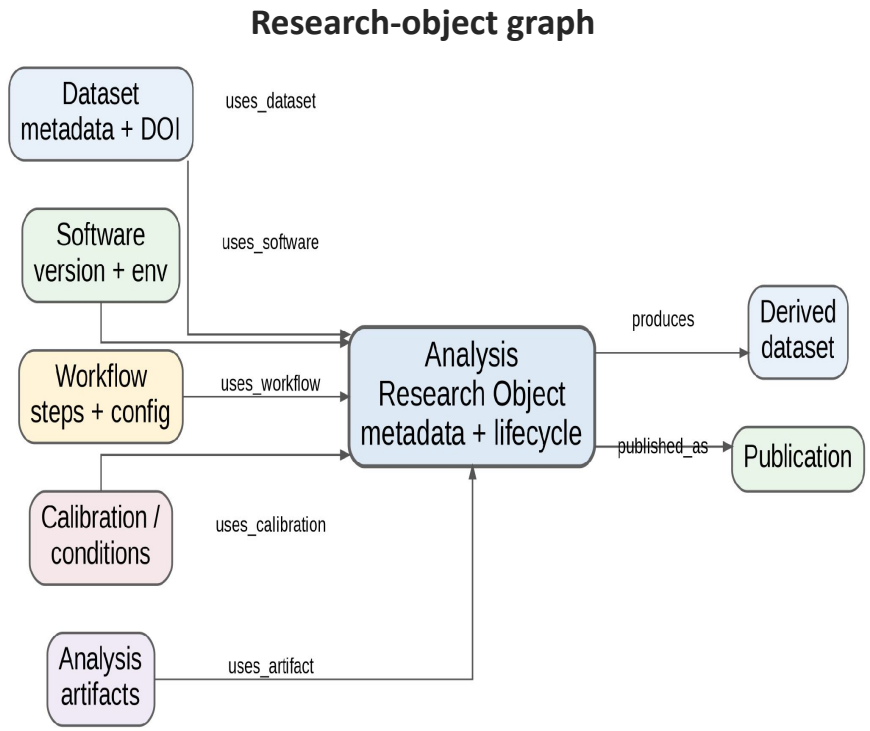
28 August 2026
Eric / Maxim / Jerome /

Several published analyses have been decomposed into graphs, and the consistency of the graphs has been checked

Recall: Why represent an analysis as a graph?

A dataset can be described on its own. An analysis cannot.

- **An analysis is more than a dataset.** Data, software, workflows, configurations, calibrations, executions, and derived results are all involved and interconnected
- **A data card describes a dataset.** It does not fully describe what was done to that dataset, with which code, under which settings, or what the result was.
- **The dependencies are the content of the analysis.** Which run lists a measurement used, which library version produced a figure, and which model turned an observable into the published number are all relations between objects rather than properties of any one of them.



The data card describes the dataset. The graph captures the analysis provenance, making the dataset interpretable and reusable.

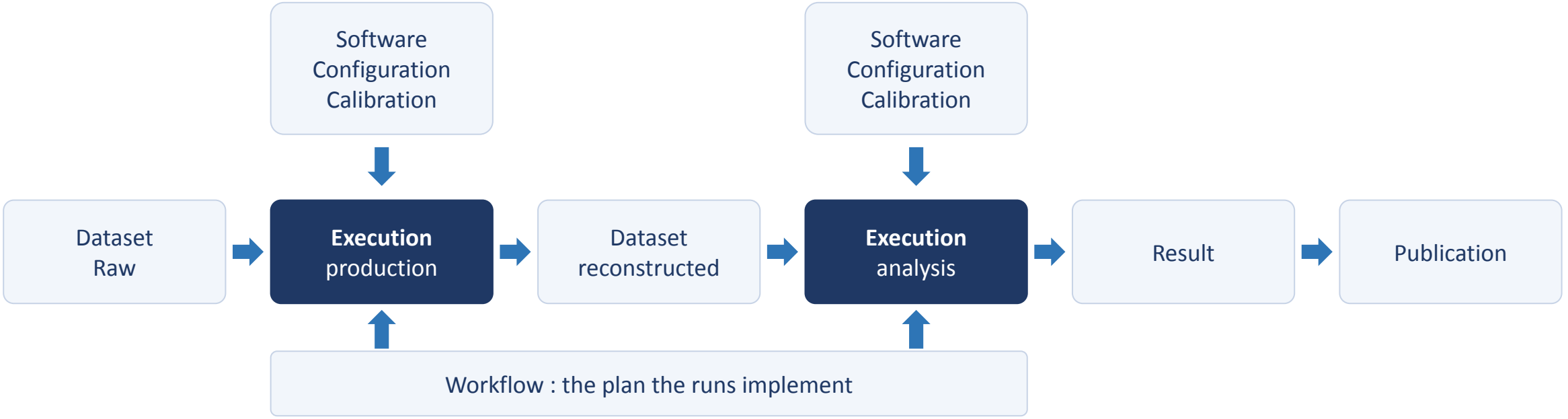
What has been tested

- What was decomposed
  - Sources: the analysis notes, the code repositories, and the references they cite. Decomposition assisted by an LLM.
- What came out of it
  - Nine object classes were sufficient to decompose for all three analyses. No additional class was required.
  - Additional sources exposed intermediate objects and dependencies not visible in the note
  - The graph produced an explicit, itemized list of dependencies and issues requiring investigation.
- Standing of the findings
  - *Every item in this talk was found through inspection of the primary files, and none has yet been confirmed by the analysis authors or by the collaboration.*

The nine classes are a proposal. Three published analyses were used to test them.

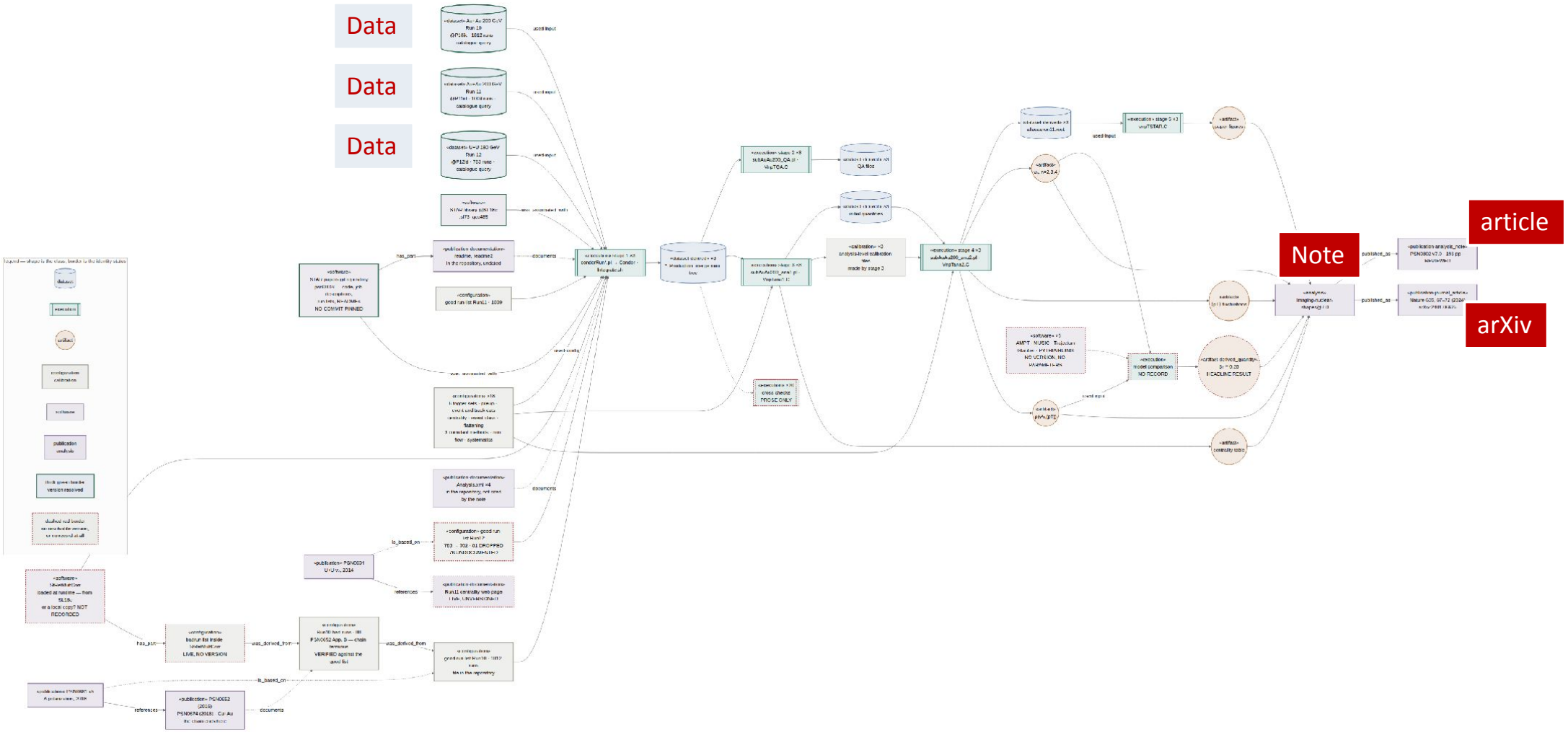
What a typical analysis decomposition looks like

Calibration is itself produced by an execution,



Nine classes (the boxes): *Configuration*, *Execution, Dataset, Software, Workflow, Artifact, Publication, Calibration, Analysis*
And **relations (the arrows)**: *used, was_generated_by, was_associated_with, was_derived_from, is_part_of, published_as*
Defined following some standards

One example of a graph



90 objects, 256 relations.

How a reference is recorded

- Pinned: a commit hash, a checksum, or a frozen file list
- released : a named immutable release or production tag
- Moving: a named location that changes underneath you
  - *a main-branch link, a live web page*
- Runtime: resolved at run time by the loader or the filesystem
  - *gSystem->Load of an absolute path; a relative path into the job's working directory*
- Prose: described in text, with no identifier
- Absent: not referenced at all

The gaps that follow are objects grading moving, runtime, prose or absent. Nothing that grades released is counted as a gap.

Where references do not resolve

Analysis-stage dependencies that do not resolve to the state actually used.

- The environment is not pinned
  - *Ex: the container is referenced by a moving tag, and ROOT and other software are then accessed via an absolute path inside it.*
- The declared release is not the effective one
  - *Ex: jobs name a library release, then symlink a pre-built area that the loader searches. The tag names the release without specifying what loaded.*
- A shared input with no released version
  - *Ex: good-run list.*
- Calibration outside the database mechanism
  - *At production, calibration is timestamp-pinned and reproducible. The analysis-stage case is a calibration produced by one stage of the analysis and consumed by the next, with no record of the version used.*
- Documentation drift
  - *Ex: README instructions that do not match the repository*

All five are in the analysis stage; none is a statement about STAR data production. The run list is recorded as a gap. The rest are items to review with the analysis teams.

Availability is not identity

The package is public, cited, and versioned, but the citation still does not identify the state that produced the result.

Example found by inspection

Analysis note, Table 1

339

Cited code, Param.h

338

Six of nine edges agree; three differ by one unit. No execution was needed to find this.

A citation is not necessarily a pin. A citation says where to look; a pin says which state was used.

Issue:

The citation points to a moving branch, and the package entered the repository after the library release used in the analysis. Because nothing is pinned, it is unclear whether the code changed after the table was created or whether the table was derived differently. To be confirmed with the authors.

One rule for every input

Every object an execution consumed must be resolvable to a fixed state.

- An identifier
  - A stable identifier, not just a file path.
  - *Example: a dataset or catalog identifier rather than /data/...*
- A fixed, recorded reference
  - A commit hash, release, production identifier, manifest or snapshot, never “current”.
  - *The reference must identify the state actually used during execution.*
- A retrievable copy
  - A location that survives, such as a personal directory or build area, is not an archival location: it may disappear with the person or project.
  - *The object must remain available when the analysis is revisited years later.*

- **The important point is not a single repository for everything, but a fixed, resolvable reference for every input.**

Shared inputs need their own authoritative version

An analysis is defined by shared inputs (used by several analyses) and analysis-specific choices. Both must be preserved.

Shared inputs

- Good/bad-run lists
- Calibration sets
- Standard selections
- Shared parameters
- Authoritative released version

Analysis-specific inputs

- Additional cuts
- Binning
- Sample-specific corrections
- Published with the analysis

Example: a shared bad-run list existed only as a live web page. Without a pinned version, an analysis cannot state which list it used.

Avoid private copies of shared inputs

Shared inputs should have one authoritative version that analyses reference.

The note and the code are complementary

Both are required, and neither can be reconstructed from the other.

- The note carries intent
  - Why this choice, how to interpret the result, what was checked, etc…
- The code carries behaviour
  - Exact values, order of operations, run-time inputs, and the effective selection.
- **Each should name the other**
  - The note names the exact code state it used; that state records which analyses it served, one package can serve many notes.

Record what the run actually used

The recorded values should be produced by the execution that produced the result

- **Generated by the run**
  - Cut values and binning
  - Run and event counts
  - Resolved input list, which files the execution actually read, with identifiers and checksums where applicable
  - Effective selections actually applied
  - **Tables** and figures and data points

  Stored as a machine-readable record beside the result, not typed into the note.

- **Written by a person**
  - Why this choice was made
  - What the numbers mean
  - Caveats and open questions
  - The physics argument

Individual cuts → Effective selection actually applied

- `pT > 1 GeV`
- `|eta| < 1`
- `nHits >= 20`
- `DCA < 2 mm`

(pT > 1 GeV)
AND (|eta| < 1)
AND (nHits >= 20)
AND (DCA < 2 mm)
AND good_run
AND trigger_fired

Preservation practices

These can be implemented incrementally...

- Pin every input to a fixed state; a release or production tag counts; “current” does not.
- Give shared inputs a released version (run lists first).
- Generate published numbers from the execution that produced them.
- Preserve the selection applied to the data, including how the individual cuts are combined.
- Review the note against the code state, not only the citations.

The graph is a review instrument.

Its value is not the diagram itself. It is a systematic inventory of an analysis's dependencies and a way to identify preservation issues for review.

Backup

Material cut from the main talk

The schema, on one object

Source of record is YAML; exported as JSON-LD/PROV-O and GraphML.

```
id:     configuration/run12-xx-good-list
class:  Configuration
label:  XX GeV Run 12 good run list
reference:
    locator:  Run12xxgood.list
    grade:    prose        # pinned | released | moving | runtime | prose | absent
provenance:
    used:          [execution/run12-uu-treemaker]
    is_based_on:   [publication/psn1234]
    was_derived_from: [configuration/run12-production-list]
confidence: unverified     # asserted | unverified | hypothesis | disputed
gap:        the selection removing 81 runs is recorded in no source
```

Nine classes: Configuration · Execution · Dataset · Software · Workflow · Artifact · Publication · Calibration · Analysis. Six relations: used · was_generated_by · was_associated_with · was_derived_from · is_part_of · published_as.

Checks you can run on your own analysis

These checks surfaced issues requiring investigation in at least one of the three case studies.

- Does the environment reference use a digest rather than a moving tag?
- Does any script actually produce the artifact that the run script consumes?
- Is the input file list preserved as a file containing identifiers and checksums?
- Do the note's parameter values match the arguments the jobs actually passed?
- Does the documented recipe execute, command by command, against the file tree?
- Are there any files, scripts, or directories the analysis depends on that are not under version control?

None of these checks requires running the analysis. They can be performed by inspection.

What the decomposition surfaced

Six recurring patterns across the three case studies. Each is a check item, not an established defect.

- Citation without pinning
  - A link to a moving branch resolves, but not to the state that was used. From the outside, it looks like good practice.
- The mechanism fails, not the artifact
  - Moving branch pointers, run-time relative paths, and an identifier stored only in a document's link layer.
- History that does not reach back
  - A package's recorded history can begin after the software release under which the analysis ran.
- Hand-typed numbers drift
  - A table transcribed once and code that may have moved since. The document alone cannot tell you which.
- The execution record nobody kept
  - Job input manifests, environment. Without them, everything downstream is inferred.
- Documentation drift
  - READMEs naming a binary, an interface, and a stage order that the repository no longer matches.