Reliable Context

RELIABLE CONTEXT / RESEARCH & PRACTICAL GUIDANCE

OpenLineage Explained: Runs, Jobs, Datasets and Facets

> Reliable Context is a kernel of truth: a system of record for organizations that fuse private and public information and must defend every value they act on. One canonical record per real thing, a cited source and an observation date behind every value, unknowns declared rather than left blank, and any past state of the record reproducible.

Published by Reliable Context · 2026-10-29 · openlineage · openlineage tutorial · data lineage standard · openlineage facets · open source data lineage

Original article: https://reliablecontext.com/articles/openlineage-explained-runs-jobs-datasets-facets



In this article

1. Why a lineage standard
2. Core model: Run, Job, Dataset
3. Events and the run cycle
4. A worked example: one RunEvent
5. Facets and extensibility
6. Producers and integrations
7. Column-level lineage and its limits
8. Mapping OpenLineage to W3C PROV
9. From job lineage to value provenance
10. Adoption checklist
11. Sources

Why a lineage standard

Data lineage is the record of which processes read which data and wrote which data. Without a shared format, every pairing of tool and catalog needs its own connector. In this article a producer is a tool that emits lineage; a consumer is a system that stores or displays it.

OpenLineage addresses the format problem. Its documentation describes it as "an extensible specification that systems can use to interoperate with lineage metadata", built around datasets, jobs and runs, which can be enriched with extensions called facets. The same page identifies OpenLineage as an LF AI & Data Foundation Graduate project (the foundation's project page dates its graduation to July 2023) and names Marquez as "the reference implementation of the OpenLineage API" (OpenLineage documentation).

The specification is a JSON Schema document. As of 28 October 2026 the core schema declares the identifier `https://openlineage.io/spec/2-0-2/OpenLineage.json`, that is, spec version 2-0-2 (OpenLineage.json on GitHub), while the documentation and client libraries are at release 1.54.0 (OpenLineage 1.54.0 release). The two numbers are independent by design: the spec version changes only when the spec changes, under a MODEL-REVISION-ADDITION scheme in which MODEL marks a breaking change (OpenLineage spec versioning).

Core model: Run, Job, Dataset

The object model has three entities (OpenLineage object model):

- A **Job** is "a process that consumes or produces Datasets": a workflow task, a query, a model or a streaming checkpoint. It is identified by a unique name within a namespace.
- A **Run** is "an instance of a Job that represents one of its occurrences in time". Its `runId` is a UUID that the client generates and keeps constant across every state update for that run; the documentation recommends UUIDv7.

- A **Dataset** is "an abstract representation of data", identified by a namespace derived from its physical location and a name within that namespace.

A namespace scopes names: usually the scheduler instance for jobs, the data store for datasets. The naming conventions fix the format per platform, for example `postgres://{host}:{port}` with name `{database}.{schema}.{table}` for PostgreSQL, and `{dag_id}.{task_id}` for an Airflow task. The same page warns that switching naming formats for an existing store, as with Snowflake's legacy account locator, produces identifiers that no longer match, so lineage nodes stop connecting (OpenLineage naming conventions).

Entity	Identity	Lifetime	Example (fictional)
Job	`namespace` + `name`	Persists across runs; definition may change	`halden-airflow-prod` / `pv_weekly_signal.aggregate_ae_counts`
Run	`runId` (UUID, client-generated)	One execution	`019a1f3c-7b2e-7c41-9d0a-5e2f8b6c4a11`
Dataset	`namespace` + `name`	Persists; contents change	`postgres://pv-db.halden.example:5432` / `safety.pv.case_reports`

Schema 2-0-2 defines three event types. A `RunEvent` reports a run state change and requires `run` and `job`; `JobEvent` and `DatasetEvent` carry design-time metadata and must not contain a run. Every event requires `eventTime`, `producer` (a URI identifying the emitting code, such as a git URL with a tag or SHA) and `schemaURL` (OpenLineage.json on GitHub).

Events and the run cycle

A `RunEvent` is one observation of a run changing state. The `eventType` field takes six values (OpenLineage run cycle):

eventType	Meaning	Terminal
`START`	Beginning of a job run	No
`RUNNING`	Additional information about a run in progress	No
`COMPLETE`	Execution has concluded	Yes
`ABORT`	The job was stopped abnormally	Yes
`FAIL`	The job failed	Yes
`OTHER`	Extra metadata outside the standard cycle; may precede `START`	No

After a terminal event no further events for that run are emitted, and events for one run are accumulative unless stated otherwise: consumers combine them, so inputs can arrive with `START` and outputs with `COMPLETE` (OpenLineage run cycle). The schema's description of `eventType` is stricter: a run must issue one `START` and one of `COMPLETE`, `ABORT` or `FAIL` (OpenLineage.json on GitHub).

That pattern fits batch jobs. For streaming jobs, terminal events occur only on restarts and "should not be relied upon"; events are periodic snapshots of a time window, and services emit `START` followed by `RUNNING` events (OpenLineage run cycle). The `jobType` facet declares a `processingType` of `BATCH`, `STREAMING` or `SERVICE`, and optionally whether events are `ACCUMULATIVE` or `COMPLETE_SNAPSHOT` (JobTypeJobFacet schema 2-0-4). A consumer expecting every run to end in `COMPLETE` will misread streaming jobs.

A worked example: one RunEvent

Halden Therapeutics AG is a fictional pharmaceutical company. Its safety team runs a weekly Airflow DAG, `pv_weekly_signal`, whose task `aggregate_ae_counts` counts serious adverse event case reports per product, event term and week. Below is the `COMPLETE` event for one run. Hosts, identifiers and the producer URL are fictional; the event validates against core schema 2-0-2 and each referenced facet schema.

```
{
  "eventType": "COMPLETE",
  "eventTime": "2026-10-26T06:14:52.318Z",
  "producer": "https://git.halden.example/platform/lineage-emitter/tree/v2.3.0",
  "schemaURL": "https://openlineage.io/spec/2-0-2/OpenLineage.json#/$defs/RunEvent",
  "run": {
    "runId": "019a1f3c-7b2e-7c41-9d0a-5e2f8b6c4a11",
    "facets": {
      "parent": {
        "_producer": "https://git.halden.example/platform/lineage-emitter/tree/v2.3.0",
        "_schemaURL": "https://openlineage.io/spec/facets/1-2-0/ParentRunFacet.json#/$defs/ParentRunFacet",
        "run": { "runId": "019a1f3b-f0c2-7a55-8e31-2c9d07b4e6f2" },
        "job": { "namespace": "halden-airflow-prod", "name": "pv_weekly_signal" }
      }
    }
  },
  "job": {
    "namespace": "halden-airflow-prod",
    "name": "pv_weekly_signal.aggregate_ae_counts",
    "facets": {
      "jobType": {
        "_producer": "https://git.halden.example/platform/lineage-emitter/tree/v2.3.0",
        "_schemaURL": "https://openlineage.io/spec/facets/2-0-
4/JobTypeJobFacet.json#/$defs/JobTypeJobFacet",
        "processingType": "BATCH",
        "integration": "AIRFLOW",
        "jobType": "TASK"
      },
      "sql": {
        "_producer": "https://git.halden.example/platform/lineage-emitter/tree/v2.3.0",
        "_schemaURL": "https://openlineage.io/spec/facets/1-1-0/SQLJobFacet.json#/$defs/SQLJobFacet",
        "query": "INSERT INTO pv.weekly_ae_counts SELECT r.product_code, t.term_name, date_trunc('week',
r.received_on) AS week_start, count(r.case_id) AS case_count FROM pv.case_reports r JOIN ref.ae_terms t ON
r.term_code = t.term_code WHERE r.is_serious GROUP BY 1, 2, 3",
        "dialect": "postgres"
      }
```

```
    }
  },
  "inputs": [
    { "namespace": "postgres://pv-db.halden.example:5432", "name": "safety.pv.case_reports" },
    { "namespace": "postgres://pv-db.halden.example:5432", "name": "safety.ref.ae_terms" }
  ],
  "outputs": [
    {
      "namespace": "postgres://pv-db.halden.example:5432",
      "name": "safety.pv.weekly_ae_counts",
      "facets": {
        "columnLineage": {
          "_producer": "https://git.halden.example/platform/lineage-emitter/tree/v2.3.0",
          "_schemaURL": "https://openlineage.io/spec/facets/1-2-
0/ColumnLineageDatasetFacet.json#/$defs/ColumnLineageDatasetFacet",
          "fields": {
            "term_name": {
              "inputFields": [
                { "namespace": "postgres://pv-db.halden.example:5432", "name": "safety.ref.ae_terms",
"field": "term_name",
                  "transformations": [ { "type": "DIRECT", "subtype": "IDENTITY", "masking": false } ] }
              ]
            },
            "case_count": {
              "inputFields": [
                { "namespace": "postgres://pv-db.halden.example:5432", "name": "safety.pv.case_reports",
"field": "case_id",
                  "transformations": [ { "type": "DIRECT", "subtype": "AGGREGATION", "masking": true } ] }
              ]
            }
          },
          "dataset": [
            { "namespace": "postgres://pv-db.halden.example:5432", "name": "safety.pv.case_reports",
"field": "is_serious",
              "transformations": [ { "type": "INDIRECT", "subtype": "FILTER" } ] },
            { "namespace": "postgres://pv-db.halden.example:5432", "name": "safety.pv.case_reports",
"field": "term_code",
              "transformations": [ { "type": "INDIRECT", "subtype": "JOIN" } ] }
          ]
        }
      },
      "outputFacets": {
        "outputStatistics": {
          "_producer": "https://git.halden.example/platform/lineage-emitter/tree/v2.3.0",
          "_schemaURL": "https://openlineage.io/spec/facets/1-0-
2/OutputStatisticsOutputDatasetFacet.json#/$defs/OutputStatisticsOutputDatasetFacet",
          "rowCount": 1842
        }
      }
    }
```

```
  ]
}
```

Two details:

- `parent` links the task run to the DAG run that spawned it, the use case the object model gives for this facet (OpenLineage object model); version 1-2-0 of the facet can also carry a `root` run and job (ParentRunFacet schema 1-2-0).
- Dataset `facets` hold metadata that is fairly stable from run to run, while `outputFacets` describe what this run wrote, here a row count (OutputStatistics facet schema).
- The column lineage is abbreviated; `product_code`, `week_start` and the `GROUP BY` dependencies are omitted.

Facets and extensibility

A facet is "an atomic piece of metadata identified by its name" (OpenLineage facets). Facets attach to runs, jobs and datasets, with separate input and output facets for what one run read or wrote. Every facet inherits two required fields, `_producer` and `_schemaURL`; the underscore prefix avoids collisions with facet-specific fields. Job and dataset facets may also set `_deleted: true` to delete a facet (OpenLineage.json on GitHub).

Standard facets named in the object model include `sourceCodeLocation` and `sourceCode` for jobs; `nominalTime`, `parent` and `errorMessage` for runs; `schema`, `dataSource`, `lifecycleStateChange` and `version` for datasets; `dataQualityMetrics` and `dataQualityAssertions` for inputs; and `outputStatistics` for outputs (OpenLineage object model).

Custom facets add organization-specific metadata without forking the spec, under three rules (OpenLineage facets):

1. Use a project prefix: `{prefix}{name}{entity}Facet` in PascalCase, attached under the key `{prefix}_{name}` (the page's example is `BigQueryStatisticsJobFacet` under `bigQuery_statistics`).
2. Point `_schemaURL` at an immutable, canonical schema location pinned to a git SHA, not a branch.
3. Expect replacement: emitting a facet with the same name for the same entity replaces the earlier instance entirely.

The replacement rule matters for audits: a consumer that keeps only current facet state loses earlier values unless the events are retained.

Producers and integrations

The documentation lists integrations for Apache Airflow, Apache Flink, Apache Hive, Apache Spark, dbt, Feast, Great Expectations, SQL and Trino (OpenLineage documentation). Four illustrate the range of mechanisms:

Integration	Mechanism per the docs
Airflow	Relocated to the official OpenLineage provider in Airflow for Airflow 2.7 and later; the older integration, last released in 1.41, is unmaintained (Airflow integration)
Spark	`OpenLineageSparkListener`, built on Spark's `SparkListener` interface, inspects RDD and DataFrame dependency graphs (Spark integration)
dbt	`dbt-ol`, a drop-in replacement for `dbt`, reads `manifest.json` and `run_results.json` after a run or parses structured JSON logs during it (dbt integration)
Flink	Flink 1.x needs job code changes and lacks Flink SQL support; the 2.x connector uses native lineage interfaces from FLIP-314 and supports Flink SQL (Flink integration)

None of these mechanisms sees a spreadsheet edited between two jobs.

Column-level lineage and its limits

The `columnLineage` dataset facet maps each output field to the input fields used to compute it (ColumnLineageDatasetFacet schema 1-2-0). Each dependency carries a transformation `type` and `subtype` (column lineage facet):

- `DIRECT`: the output value was derived from the input value. Subtypes: `IDENTITY`, `TRANSFORMATION` (transformed from the input row) and `AGGREGATION` (over several input rows).
- `INDIRECT`: the output is affected by the input but not derived from it. Subtypes: `JOIN`, `GROUP_BY`, `FILTER`, `SORT`, `WINDOW` and `CONDITIONAL`.

A boolean `masking` flag marks whether the input value was obfuscated; the page's examples are `hash` for a transformation and `count` for an aggregation, hence `masking: true` on `case_count` above. The optional `dataset` array records inputs that affect the whole output, such as filter and join columns.

This is schema-level dependency information: it says that `case_count` depends on `case_reports.case_id`, filtered by `is_serious`. It does not say which 37 case reports produced a count of 37 for a given product, term and week, whether any was later corrected, or whether `is_serious` was coded correctly when the job ran. Field mappings are also only as complete as the integration's ability to parse the transformation.

Mapping OpenLineage to W3C PROV

W3C PROV is the general model for provenance, a family of documents published in April 2013 (PROV-Overview). PROV-DM defines an Entity as "a physical, digital, conceptual, or other kind of thing with some fixed aspects", an Activity as "something that occurs over a period of time and acts upon or with entities", and an Agent as bearing "some form of responsibility for an activity taking place"; a Plan is an entity representing the steps agents intend in order to achieve goals (PROV-DM).

OpenLineage does not define a mapping to PROV. The correspondence below is this article's own reading:

OpenLineage	Closest PROV-DM concept	Fit
Run	Activity, with start and end times from `START` and terminal events	Close
Dataset, as read or written by one run	Entity	Close only if a version is identified
Input dataset	`used` (Usage)	Close
Output dataset	`wasGeneratedBy` (Generation)	Close
Job	Plan, referenced through an Association	Partial: a job is a named, evolving process
`producer` URI	Software Agent	Partial: identifies emitting code, not who is responsible
`parent` facet	`wasStartedBy` with a starter activity	Approximate
Column lineage	`wasDerivedFrom` between attribute-level entities	Approximate

The main gap is versioning. A PROV entity has fixed aspects; an OpenLineage dataset names a table whose contents change, unless the producer adds the `version` facet, which captures a version only when the store defines one, such as an Iceberg snapshot ID (OpenLineage object model). Without it, "run R read table T" cannot be resolved to the data R actually read. The W3C PROV standard page covers the PROV side.

From job lineage to value provenance

Lineage events are evidence about processes. They do not establish that a particular value is correct, which source rows it came from, or which document supports it.

Take a Halden output row stating 37 serious cases for one product, term and week. The event shows that run `019a1f3c…` produced it from `case_reports` and `ae_terms` with the recorded SQL, as an aggregation of `case_id` filtered by `is_serious`. To show why the number is 37, a reviewer needs the 37 source rows as they stood at read time and, for each, the source document justifying `is_serious = true`. That is value-level provenance, a different record; query provenance from source rows to a derived value covers the techniques.

Banking supervision shows why the distinction matters. The Basel Committee's principles for risk data aggregation and risk reporting (BCBS 239, January 2013) do not use the word "lineage". Principle 2 expects integrated data taxonomies, including metadata and single identifiers or unified naming conventions (paragraph 33). Principle 3 expects risk data to be reconciled with the bank's sources (paragraph 36(c)) and banks to document and explain all risk data aggregation processes, automated or manual, including manual workarounds (paragraph 39) (BCBS 239). Instrumented lineage can support documentation of automated steps and consistent naming. It does not perform a reconciliation, and it is silent about manual workarounds no integration observes. This article is educational and is not legal or regulatory advice.

Adoption checklist

1. **Fix naming first.** Apply the naming conventions to every store and scheduler and freeze job namespaces; renamed identifiers break the graph silently.
2. **Retain raw events.** Facets are replaced on re-emission, so keep the event stream if lineage will support audits.
3. **Declare processing type** with the `jobType` facet so consumers handle streaming jobs correctly.
4. **Pin versions:** core `schemaURL`, custom facet `_schemaURL` values pinned to a SHA, and `sourceCodeLocation` with a commit.
5. **Capture dataset versions** where the store supports snapshots.
6. **Inventory blind spots:** manual steps, spreadsheets and uninstrumented tools.
7. **Test column lineage on real queries,** including user-defined functions, views and dynamic SQL.
8. **Decide which outputs need value-level provenance.** Job lineage is the index to that evidence, not the evidence itself.

Reliable Context is a system of record built around source-backed values, the layer this checklist ends on.

Sources

1. OpenLineage documentation (home), OpenLineage project, version 1.54.0. https://openlineage.io/docs/
2. Object Model, OpenLineage documentation. https://openlineage.io/docs/spec/object-model
3. The Run Cycle, OpenLineage documentation. https://openlineage.io/docs/spec/run-cycle
4. Facets, OpenLineage documentation. https://openlineage.io/docs/spec/facets/
5. Naming Conventions, OpenLineage documentation. https://openlineage.io/docs/spec/naming
6. Column Level Lineage Dataset Facet, OpenLineage documentation. https://openlineage.io/docs/spec/facets/dataset-facets/column_lineage_facet
7. OpenLineage.json (core spec, $id 2-0-2), OpenLineage GitHub repository. https://github.com/OpenLineage/OpenLineage/blob/main/spec/OpenLineage.json
8. Versioning, OpenLineage GitHub repository. https://github.com/OpenLineage/OpenLineage/blob/main/spec/Versioning.md
9. OpenLineage release 1.54.0, GitHub, 13 October 2026. https://github.com/OpenLineage/OpenLineage/releases/tag/1.54.0
10. JobTypeJobFacet schema 2-0-4, OpenLineage. https://openlineage.io/spec/facets/2-0-4/JobTypeJobFacet.json
11. ColumnLineageDatasetFacet schema 1-2-0, OpenLineage. https://openlineage.io/spec/facets/1-2-0/ColumnLineageDatasetFacet.json
12. OutputStatisticsOutputDatasetFacet schema 1-0-2, OpenLineage. https://openlineage.io/spec/facets/1-0-2/OutputStatisticsOutputDatasetFacet.json
13. ParentRunFacet schema 1-2-0, OpenLineage. https://openlineage.io/spec/facets/1-2-0/ParentRunFacet.json
14. Apache Airflow integration, OpenLineage documentation. https://openlineage.io/docs/integrations/airflow/
15. Apache Spark integration, OpenLineage documentation. https://openlineage.io/docs/integrations/spark/

16. dbt integration, OpenLineage documentation. https://openlineage.io/docs/integrations/dbt
17. Apache Flink integration, OpenLineage documentation. https://openlineage.io/docs/integrations/flink
18. PROV-Overview: An Overview of the PROV Family of Documents, W3C Working Group Note, 30 April 2013. https://www.w3.org/TR/prov-overview/
19. PROV-DM: The PROV Data Model, W3C Recommendation, 30 April 2013. https://www.w3.org/TR/prov-dm/
20. Principles for effective risk data aggregation and risk reporting (BCBS 239), Basel Committee on Banking Supervision, January 2013. https://www.bis.org/publications/201301-guidelines-principles-effective-risk-data-aggregation-and-risk-reporting.pdf
21. OpenLineage project page, LF AI & Data Foundation. https://lfaidata.foundation/projects/openlineage/

Source links

Links cited in this article, in order of first appearance. Full addresses are provided for readers using extracted text.

1. https://lfaidata.foundation/projects/openlineage
2. https://openlineage.io/docs
3. https://github.com/OpenLineage/OpenLineage/blob/main/spec/OpenLineage.json
4. https://github.com/OpenLineage/OpenLineage/releases/tag/1.54.0
5. https://github.com/OpenLineage/OpenLineage/blob/main/spec/Versioning.md
6. https://openlineage.io/docs/spec/object-model
7. https://openlineage.io/docs/spec/naming
8. https://openlineage.io/docs/spec/run-cycle
9. https://openlineage.io/spec/facets/2-0-4/JobTypeJobFacet.json
10. https://openlineage.io/spec/facets/1-2-0/ParentRunFacet.json
11. https://openlineage.io/spec/facets/1-0-2/OutputStatisticsOutputDatasetFacet.json
12. https://openlineage.io/docs/spec/facets
13. https://openlineage.io/docs/integrations/airflow
14. https://openlineage.io/docs/integrations/spark
15. https://openlineage.io/docs/integrations/dbt
16. https://openlineage.io/docs/integrations/flink
17. https://openlineage.io/spec/facets/1-2-0/ColumnLineageDatasetFacet.json
18. https://openlineage.io/docs/spec/facets/dataset-facets/column_lineage_facet
19. https://www.w3.org/TR/prov-overview
20. https://www.w3.org/TR/prov-dm
21. https://www.bis.org/publications/201301-guidelines-principles-effective-risk-data-aggregation-and-risk-reporting.pdf

THE PUBLISHER

About Reliable Context

Reliable Context is a system of record for organizations that fuse private and public information and must defend every value they act on.

The guarantees

One canonical record per real thing: duplicates resolve by rule, ambiguity is shown rather than hidden, and related organizations relate without ever being merged. A verbatim, cited source and an observation date sit behind every value. Unknowns are declared as not found, blocked or not applicable with a reason, never left as an empty cell. Any past state of the record is reproducible, together with what changed since.

Where it runs

It runs inside the customer's perimeter, one database per organization. Permitted sources, model routing, reader clearance and retention are set as the customer's own policies, and every value is exportable with its chain of custody for an auditor or regulator.

What it is not

It is not a vector database or a RAG framework, not a data catalog, MDM or data-quality suite, not a CRM or data-enrichment vendor, and not a chatbot or AI assistant. Where a reader needs one of those, the honest answer is to say so and point them there.

Visit

See Reliable Context.

Reliable Context is in production use in two sectors; more than six thousand of its identity decisions were made by rule and none has been reversed. Articles published here explain general practice, cite their primary and academic sources, and name the jurisdiction of any regulatory rule they describe. They are not legal, regulatory or compliance advice for a specific organization, and requirements change.

Website: https://reliablecontext.com

This article is educational. Verify consequential medical, legal, financial, regulatory and technical decisions with appropriate professionals and the cited primary sources. Company services are described separately from the article's research findings.