



Airflow already knows the root cause



Kacper Muda



The only **data engineering agent**, built for Airflow

AVAILABLE IN LABS

- Airflow Specific Capabilities & Skills
- Purpose Built Agent Harness
- Open & Proprietary Knowledge

* See “How We Taught an Agent Airflow: Building Self-Healing Pipelines with Otto” (Airflow Summit 2026)

Investigation

Priority & confidence

How urgent, and how sure

Root-cause narrative

Plain language, not a log dump

Suggested fix

Ready to run, not just a guess

Evidence trail

Every claim links back to a source event

ASTRONOMER

🌟 Otto Investigation

Open in Airflow

MEDIUMP3PERMANENT

🔗🔄📄

dbt not_null data tests on orders model fail (94 NULL amounts) — jaffle_shop DAG failure

Confidence: 95%

The scheduled run of jaffle_shop failed because the dbt test task `dbt_task_group.orders.test` returned a non-zero exit code. dbt ran 10 data tests against the `orders` model and 5 `not_null` tests failed, each reporting "Got 94 results, configured to fail if != 0" — meaning 94 rows in the orders table have NULL values in the amount columns (amount, bank_transfer_amount, coupon_amount, credit_card_amount, gift_card_amount). This is a data-quality failure in the underlying Snowflake data, not an Airflow or code defect; it repeated identically across all 3 attempts and cascaded to mark `generate_dbt_docs` and `on_run_end` as `upstream_failed`.

SUGGESTED FIX

This is a data-quality failure, not an Airflow bug — fix the source data or the test contract. Immediate: query the orders model in Snowflake to find the 94 offending rows, e.g.

```
SELECT order_id, amount, bank_transfer_amount, coupon_amount, credit_card_amount, gift_card_amount FROM DEMO.RE_JAFFLE_SHOP.ORDERS WHERE amount IS NULL OR bank_transfer_amount IS NULL OR coupon_amount IS NULL OR credit_card_amount IS NULL OR gift_card_amount IS NULL;
```

and remediate the upstream load (the payment amount columns should default to 0 rather than NULL — the jaffle_shop orders model normally COALESCEs payment-method amounts to 0, so investigate why those coalesce/null-fill values are now NULL, e.g. a new payment_method value or a broken join in stg_payments/orders). Do NOT simply add retries — the failure is deterministic and retries only waste ~10 min per run. If NULL amounts are legitimately acceptable, relax the not_null tests on those columns in the dbt models/schema.yml for the orders model instead.

> EVIDENCE

DAG-LEVEL CHECKS

✅ DAG-level dagrun_timeout: check whether unfinished tasks were skipped (timeout hallmark) and whether run duration matches a timeout value

No timeout. The downstream tasks generate_dbt_docs and on_run_end are marked upstreamFailed (not skipped), and the DAG failed because dbt_task_group.orders.test failed after exhausting retries. The ~12 min run duration reflects retry delays, not a dagrun_timeout. No 'has timed-out' signal relevant.

❌ Failure scope: is this a single-task failure or a DAG-wide/infra issue

Single root task failure. All seeds, staging models/tests, orders.run and customers run/test succeeded; only orders.test failed on data-quality assertions, cascading to 2 upstreamFailed tasks.

TASK FAILURES

❌ > dbt_task_group.orders.test

❌ > generate_dbt_docs

❌ > on_run_end

> 2 tool calls · 3 reasoning

Impact

Impacted Entities

To assess the full blast radius

Impacted Downstream Assets (11)



s3://compliance-reports/new_york/new_york_city

Type

Airflow Dataset

/

Deployment

Play to Win Databricks DND



/

DAG ID

in_new_york_city



Last updated

a day ago







prepare_compliance_report_ny

Type

Airflow DAG

/

Deployment

Play to Win Databricks DND





Last updated

6 hours ago



Owner

Pranav

databricks

compliance

s3







databricks://PROD/COMPLIANCE

Type

Airflow Dataset

/

Deployment

Play to Win Databricks DND



/

DAG ID

prepare_compliance_report_ny



Last updated

5 days ago







databricks_to_salesforce

Type

Airflow DAG

/

Deployment

Play to Win Databricks DND



Previous 14 runs



Last updated

5 days ago



Owner

Alex

compliance

salesforce

databricks







user_360_bronze_d

Type

Airflow DAG

/

Deployment

Play to Win Databricks DND



Previous 14 runs



Last updated

5 days ago



Owner

airflow







user_360_bronze

Type

Airflow Dataset

/

Deployment

Play to Win Databricks DND



/

DAG ID

user_360_bronze_d



Last updated

5 days ago

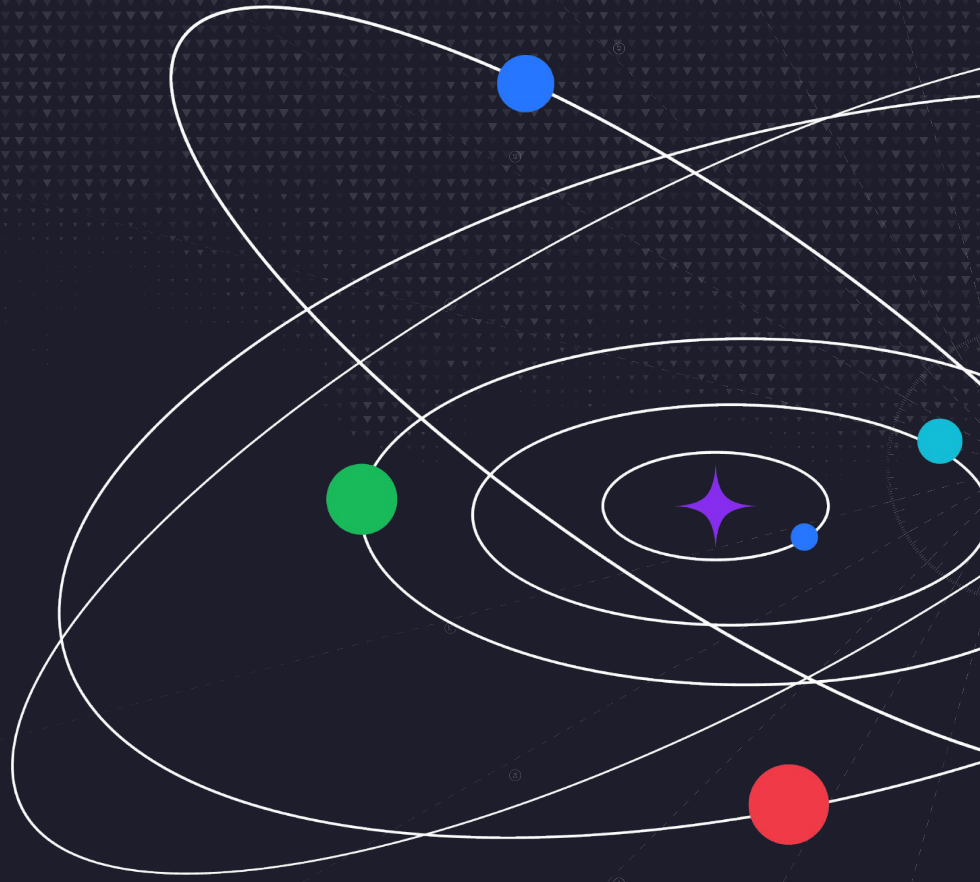




OpenLineage

In 60 seconds

ASTRONOMER



Related talks worth your time

1. OpenLineage in Airflow: A Comprehensive Guide (Airflow Summit 2023)
2. Activating Operational Metadata with Airflow, Atlan, and OpenLineage (Airflow Summit 2024)
3. OpenLineage: From Operators to Hooks (Airflow Summit 2024)

OpenLineage

1. Open standard for metadata collection, no vendor lock-in.
2. Data collected at runtime, not reconstructed after the fact.
3. Easily extensible with your own domain-specific context.
4. Granular control over what gets collected and emitted.
5. Hooks into Airflow through the OpenLineage provider via Airflow's Listener API.
6. Also covers Spark, dbt, Flink, and Hive - auto-linked for some operators when triggered from Airflow.
7. Sends events to HTTP, Kafka, file, and more out of the box - custom transports are easy to add.

Example OpenLineage event

Job

The stable identity - name, namespace, type

Run

Start/running/complete/fail lifecycle, tied to one execution

Dataset

Inputs & outputs - schema, source, location

Facets extend any of the three with more detail.

```
1 {
2   "eventTime": "2026-09-03T11:00:00.764517+00:00",
3   "eventType": "COMPLETE",
4   "job": {
5     "facets": {
6       "jobType": {
7         "integration": "AIRFLOW",
8         "jobType": "TASK",
9         "processingType": "BATCH"
10      }
11    },
12    "name": "report.extract2",
13    "namespace": "test-airflow"
14  },
15  "run": {
16    "facets": {
17      "externalQuery": {
18        "externalQueryId": "query_id_1234",
19        "source": "postgres://192.158.1.38"
20      }
21    },
22    "runId": "0191c13b-a4c8-75cf-a774-37e85f91db02"
23  },
24  "inputs": [
25    {
26      "facets": {
27        "schema": {
28          "fields": [
29            {
30              "name": "uid",
31              "type": "STRING"
32            },
33            {
34              "name": "purchases",
35              "type": "INT"
36            }
37          ]
38        }
39      },
40      "name": "prod.sales.weekly",
41      "namespace": "postgres://192.158.1.38"
42    }
43  ],
44  "outputs": [
45    {
46      "facets": {
47        "documentation": {
48          "description": "Example documentation facet."
49        }
50      },
51      "name": "customer_data.csv",
52      "namespace": "gs://my_bucket"
53    }
54  ],
55  "producer": "https://github.com/apache/airflow/tree/providers-openlineage/2.20.0",
56  "schemaURL": "https://openlineage.io/spec/2-0-2/OpenLineage.json#/$defs/RunEvent"
57 }
```

AN AGENT

Why not just point it at all your logs?

ASTRONOMER



What the agent actually needs

The real questions aren't optional: Which logs? From which tasks? The failing one, or the related ones too - and how do you even know which ones are related? Through a database table? A blob file? An Airflow asset? A trigger / sensor operator? What if logs are bloated? What if logs do not have the information an agent needs?

Up to date

Useful context from the time of the run, not guessed or partially reconstructed afterwards.

Related, not isolated

Upstream and related jobs and data that may have caused the failure.

Minimal but exhaustive

Only what it actually needs, not every log line you have

FINDING THE ROOT CAUSE

Step by step

ASTRONOMER



Seven steps to a root cause

- 1 Know that job failed
- 2 Identify the job
- 3 See the error
- 4 Understand the surrounding
- 5 Trace the blast radius
- 6 Check for anomalies over time
- 7 Add the context only you have

1. Know that job failed

Trivial, but everything starts here.

OpenLineage sends an event at task/DAG start, completion, failure, and skip - plus whenever a run is manually marked failed or succeeded in the UI.

```
1  {  
2    "eventTime": "2026-09-03T11:00:00.764517+00:00",  
3    "eventType": "FAIL",  
4    ***  
5  }
```

2. Identify the job

All the coordinates every other piece of metadata hangs off.

```
1  {
2    "run": {
3      "facets": {
4        "airflow": {
5          "dag": {
6            "dag_id": "etl_hourly_load_snowflake",
7          },
8        },
9        "dagRun": {
10         "run_id": "scheduled__2026-08-24T12:00:00+00:00",
11       },
12       "task": {
13         "mapped": true,
14         "task_id": "load_data_by_market_by_month",
15       },
16       "taskInstance": {
17         "try_number": 2,
18         "map_index": 12,
19         "map_index_label": "market.pl.december",
20       },
21     },
22   },
23 },
24 },
25 },
26 },
27 }
```

3. The error

Error message

The full exception - not just "it failed."

```
1 {
2   "run": {
3     "facets": {
4       "errorMessage": {
5         "message": "The conn_id 'q2s.bigquery_connection' isn't defined",
6         "programmingLanguage": "python",
7         "stackTrace": "Traceback (most recent call last):\n File \"usr/local/lib/pyt
8       }
9     }
10   }
11 }
```

Deadline alerts

Which deadlines were breached, and did that cause the failure.

```
1 {
2   "run": {
3     "facets": {
4       "airflowRun": {
5         "dagRun": {
6           "alerts": {
7             "callback_def": {
8               "class_name": "airflow.callbacks.callback.AsyncCallback",
9               "data": {}
10            },
11            "range": {
12              "slack_webhook_conn_id": "airflow-slack-webhook",
13              "slack_webhook_conn_id": "airflow-slack-webhook",
14              "text": "salute_slack! DAG {{ dag_run.dag_id }} missed its deadline. Volume: {{ dag_run.dag_run_id }}\n",
15              "path": "airflow.providers.slack.notifications.slack_webhook.SlackWebhookNotifier"
16            },
17            "version": 0
18          },
19          "deadline_time": "2020-08-24T20:25:00.474802+00:00",
20          "interval": "2000 s",
21          "raised": false,
22          "reference": {
23            "reference_type": "DagRunDagwebhookDeadline"
24          }
25        }
26      }
27    }
28  }
29 }
30 }
31 }
```

Data quality checks

Which check failed - and is it fatal, or just a warning.

```
1 {
2   "inputs": {
3     "facets": {
4       "dataQualityAssertions": {
5         "assertions": [
6           {
7             "actual": "2644",
8             "assertion": "accepted_range",
9             "column": "id",
10            "content": "SUM(CASE WHEN id IS NULL THEN 1 ELSE 0 END)",
11            "contentType": "sql",
12            "description": "Column-level statistical check against the configured threshold",
13            "expected": "41888",
14            "name": "id.null_check",
15            "params": {
16              "accept_none": true,
17              "less_than": 10000
18            },
19            "severity": "error",
20            "success": true
21          },
22          {
23            "actual": "0",
24            "assertion": "not_null",
25            "column": "event_time",
26            "content": "SUM(CASE WHEN event_time IS NULL THEN 1 ELSE 0 END)",
27            "contentType": "sql",
28            "description": "Column-level statistical check against the configured threshold",
29            "expected": "0",
30            "name": "event_time.null_check",
31            "params": {
32              "accept_none": true,
33              "equal_to": 0
34            },
35            "severity": "error",
36            "success": true
37          }
38        ]
39      }
40    }
41  },
42  "name": "SANDBOX_MY_DATA_ETL_RESULTS",
43  "namespace": "snowflake://q1234-q5678"
44 }
45 }
```

4. Understand the surrounding

Job attributes

Every job carries its own paper trail: an owner to page, and documentation that explains what it's supposed to do, tags used and more.

```

2  ✓
3  ✓
4  ✓
5  ✓
6  ✓
7  ✓
8  ✓
9  ✓
10 ✓
11 ✓
12 ✓
13 ✓
14 ✓
15 ✓
16 ✓
17 ✓
18 ✓
19 ✓
20 ✓
21 ✓
22 ✓
23 ✓
24 ✓
25 ✓
26 ✓
27 ✓
28 ✓
29 ✓
30 ✓
31 ✓
32 ✓
33 ✓
34 ✓
35 ✓
36 ✓

```

```

{
  "job": {
    "facets": {
      "documentation": {
        "contentType": "text/plain",
        "description": "Author web clickstream (DP-304)"
      },
      "ownership": {
        "owners": [
          {
            "name": "dpes-data-eng"
          }
        ]
      },
      "tags": {
        "tags": [
          {
            "key": "clickstream",
            "source": "AIRFLOW",
            "value": "clickstream"
          },
          {
            "key": "non-sla",
            "source": "AIRFLOW",
            "value": "non-sla"
          },
          {
            "key": "snowflake",
            "source": "AIRFLOW",
            "value": "snowflake"
          }
        ]
      }
    }
  }
}

```

Run & trigger info

What parameters were provided, whether a note was left, and who or what triggered the run.

```

1  {
2      "run": {
3          "facets": {
4              "airflow": {
5                  "dagRun": {
6                      "run_type": "manual",
7                      "triggering_user_name": "John Doe",
8                      "note": "Running custom sales report for an important client.",
9                      "conf": {
10                          "client_id": "xfr_123",
11                          "pcode": "075284",
12                          "processed_bucket": "processed-data-87655223",
13                          "raw_flights_bucket": "raw-data-79855099433",
14                          "tile_run_ids": [
15                              "16c7618a-b334-496c-84c2-5427d79de368",
16                          ]
17                      }
18                  }
19              }
20          }
21      }
22  }

```

Human in the loop

Did somebody approve this job to run?
Who? When? What reason did they
provide? A lineage-attached fact - not a
Slack thread nobody can search.

```

1  {
2  |   "run": {
3  |     "facets": {
4  |       "airflow": {
5  |         "task": {
6  |           "hitl_summary": {
7  |             "body": "# Review before delivery **Dataset:**\n- (Browse in S3 console)(https://us-east-1.console.aws.amazon.com/s3/home?region=us-east-1#storage/containers=arn:aws:s3:::hitl-logs-logs-2024-08-24T16:22:21.252159+00:00)",
8  |             "chosen_options": [
9  |               "Deliver"
10 |             ],
11 |             "Multiple": false,
12 |             "options": [
13 |               "Deliver",
14 |               "Reject"
15 |             ],
16 |             "params_input": {},
17 |             "responded_at": "2025-08-24T16:22:21.252159+00:00",
18 |             "responded_by_user": {
19 |               "id": "cmkocoyh2yic@io@on3jdcpe",
20 |               "name": "Arthur Jones"
21 |             },
22 |             "subject": "#Sanity check: {{ dag_run.conf.pcode }} ready for delivery?"
23 |           }
24 |         }
25 |       }
26 |     }
27 |   }
28 | }

```

What actually ran

Source code

What ran, for Python & Bash operators

```
1  {
2  }
3  "job": {
4    "facets": {
5      "sourceCode": {
6        "language": "python",
7        "sourceCode": "def create_table_if_missing():\n \\\n \\\nCreate the
8      }
9    }
10 }
```

SQL query & query ID

The full query that actually executed & the exact query ID, for further debugging in your warehouse.

```
1  {
2  }
3  "job": {
4    "facets": {
5      "sql": {
6        "dialect": "snowflake",
7        "query": "SELECT id, name FROM analytics.public.all_facets_output WHERE id IS NOT NULL"
8      }
9    }
10 }
```

```
1  {
2  }
3  "run": {
4    "facets": {
5      "externalQuery": {
6        "externalQueryId": "some-internal-query-id-001",
7        "source": "snowflake://fy1234-gp5678"
8      }
9    }
10 }
```

5. The blast radius

- **Job relationships** - which jobs feed in, and which consume its output. When something breaks, you need the surrounding job graph immediately.
- **Dataset lineage** - every other job reading or writing the same table.
- **Column-level lineage** - which input columns produced which output columns. Narrows an anomaly to one exact field.
- **Triggering DAG runs** - child runs carry the parent's identifiers automatically.
- **Asset scheduling** - consuming run's lineage event links back to the exact upstream run that updated the asset,s so "what fed this DAG?" has a direct answer instead of a guess.

```
1  {  
2    "run": {  
3      "facets": {  
4        "jobDependencies": {  
5          "upstream": [  
6            {  
7              "job": {  
8                "name": "dag_asset_1_producer.produce_dataset_1",  
9                "namespace": "airflow"  
10             },  
11             "run": {  
12               "runId": "019b6ff1-f2f0-79bf-a797-0bbe6983c753"  
13             },  
14             "airflow": {  
15               "asset_events": [  
16                 {  
17                   "asset_id": 1,  
18                   "asset_uri": "s3://first-bucket/ds1.csv",  
19                   "dag_run_id": "manual_2025-12-30T15:48:06+00:00",  
20                   "asset_event_id": 1  
21                 }  
22               ]  
23             },  
24             "dependency_type": "IMPLICIT_ASSET_DEPENDENCY"  
25           },  
26           {  
27             "job": {  
28               "name": "dag_asset_2_producer.produce_dataset_2",  
29               "namespace": "airflow"  
30             },  
31             "run": {  
32               "runId": "019b6ff4-7f48-7ee5-aacb-a88072516b1e"  
33             },  
34             "airflow": {  
35               "asset_events": [  
36                 {  
37                   "asset_id": 2,  
38                   "asset_uri": "gs://second-bucket/ds2.xlsx",  
39                   "dag_run_id": "manual_2025-12-30T15:50:53+00:00",  
40                   "asset_event_id": 3  
41                 }  
42               ]  
43             },  
44             "dependency_type": "IMPLICIT_ASSET_DEPENDENCY"  
45           }  
46         ],  
47         "downstream": []  
48       }  
49     }  
50   }  
51 }
```

6. Look for anomalies over time

Table schema

Column names & types on every dataset, good for catching drift early

```
1 {
2   "inputs": [
3     {
4       "facets": {
5         "schema": {
6           "fields": [
7             {
8               "description": "Unique row identifier",
9               "fields": [],
10              "name": "id",
11              "ordinal_position": 1,
12              "type": "INTEGER"
13            },
14            {
15              "description": "Full name of the entity",
16              "fields": [],
17              "name": "name",
18              "ordinal_position": 2,
19              "type": "VARCHAR"
20            },
21            {
22              "description": "Nested address record",
23              "fields": [
24                {
25                  "description": "Street address line",
26                  "fields": [],
27                  "name": "street",
28                  "ordinal_position": 1,
29                  "type": "VARCHAR"
30                },
31                {
32                  "description": "City name",
33                  "fields": [],
34                  "name": "city",
35                  "ordinal_position": 2,
36                  "type": "VARCHAR"
37                }
38              ],
39              "name": "address",
40              "ordinal_position": 3,
41              "type": "STRUCT"
42            }
43          ],
44          "name": "input/data.csv",
45          "namespace": "s3://my-company-bucket"
46        }
47      ]
48    }
49  }
50 }
```

State & duration

A slow creep toward a timeout shows up before it becomes a failure

```
1 {
2   "run": {
3     "facets": {
4       "airflowDagRun": {
5         "dagRun": {
6           "airflowState": {
7             "dagRunState": "success",
8             "tasksDuration": {
9               "finish": 0.0,
10              "fivetran_sync_ollie": 41.621537,
11              "load_articles.control_delete_gcs_files": 1.26653,
12              "load_articles.control_gcs": 2.47089,
13              "start": 0.0
14            },
15            "tasksState": {
16              "finish": "success",
17              "fivetran_sync_ollie": "success",
18              "load_articles.control_delete_gcs_files": "success",
19              "load_articles.control_gcs": "success",
20              "start": "success"
21            }
22          }
23        }
24      }
25    }
26  }
27 }
```

7. The context only you have

Whatever you choose to include, with easy, extensible way - either on a global level or per operator.

The schema is extensible: attach anything domain-specific - cost center, PII flags - and it travels with the event, as close to the job as possible.

New ways of retrieving lineage

Custom facets via API

Push lineage facts directly through a documented API.

```
1 from airflow.providers.openlineage.api import emit_dataset_lineage, emit_query_lineage
2
3 @dag(...)
4 def my_custom_dag():
5
6     @task
7     def run_datasets() -> None:
8         emit_dataset_lineage(
9             inputs=[_dataset_with_assertions("s3://example-bucket", "raw/2024/01/01/orders.csv")],
10             outputs=[
11                 _dataset_with_assertions("snowflake://example-acct", "analytics.public.orders_enriched")
12             ],
13             additional_run_facets={"my_custom_run_facet": {"key": "value"}},
14             additional_job_facets={
15                 "sourceCodeLocation": source_code_location_job.SourceCodeLocationJobFacet(
16                     type="git",
17                     url="https://github.com/apache/airflow",
18                     repoUrl="https://github.com/apache/airflow",
19                     path="my/dir/dag.py",
20                     version="main",
21                     branch="main",
22                 ),
23             },
24         )
25
26     @task
27     def run_query() -> None:
28         """Maximal emit_query_lineage ☐ every kwarg set."""
29
30         emit_query_lineage(
31             query_id="qid-max-1",
32             query_source_namespace="snowflake://example-acct",
33             query_text=(
34                 "INSERT INTO analytics.public.user_events_summary "
35                 "SELECT user_id, COUNT(*) FROM analytics.public.user_events GROUP BY user_id"
36             ),
37             is_successful=True,
38             default_database="ANALYTICS",
39             default_schema="public",
40         )
```

ASTRONOMER

Hook-level SQL capture

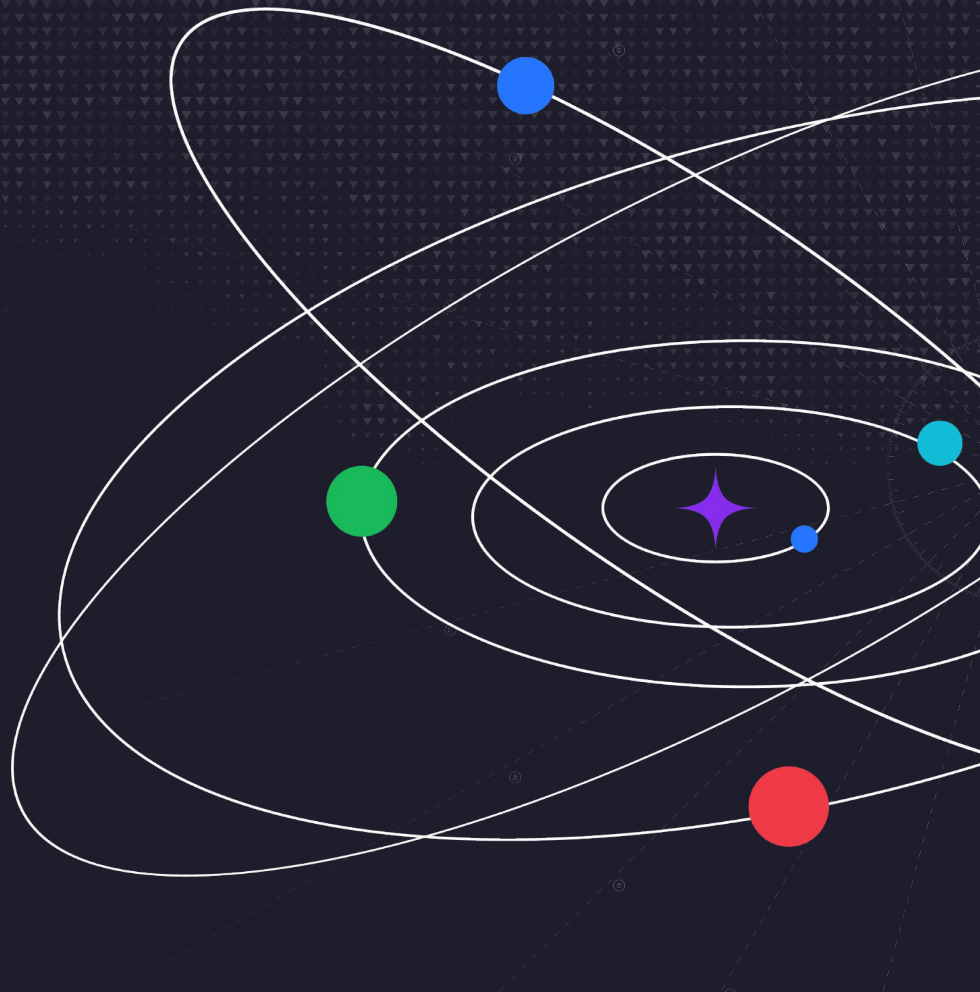
Same lineage depth and details as SQL dedicated operators, from inside Python tasks - without rewriting the code.

```
1 @task
2 def run():
3     table = "SANDBOX.MY_DATA.ETL_RESULTS"
4     sql = (
5         f"CREATE OR REPLACE TABLE SANDBOX.MY_DATA.ETL_RESULTS AS SELECT id, event_time FROM ...;"
6         f"INSERT INTO ... ;"
7     )
8     hook = SnowflakeHook(snowflake_conn_id='snowflake_default')
9     results = hook.run(sql, ...)
10     ...
```

THE GAP

When OpenLineage is not enough

ASTRONOMER



Key Context Pieces



hourly_success_dag failing every run — tasks stuck in queued, no Celery worker executed them

Confidence: 78%

Every scheduled run of `hourly_success_dag` since `~2026-08-25T00:00:00Z` has failed even though its only real task (`always_succeed`) is a no-op that returns 0. The scheduler repeatedly queued the task to `CeleryExecutor`, but no worker ever picked it up; after ~18 minutes of requeue attempts the scheduler marked it failed ("Task requeue attempts exceeded max"). The same pattern hit every DAG in the deployment (`hourly_failure_dag`, `freshness_sla_miss`, the `test_*` DAGs), indicating a deployment-wide Celery worker availability problem rather than any DAG-code issue.

SUGGESTED FIX

Restore Celery worker availability for the deployment — this is a worker-execution outage, not a DAG-code bug. 1) Inspect the worker logs for deployment `cmqfd48xy5f2c01kmqtbhlinj` around 2026-08-25T00:00:00Z onward for crash/OOM/startup errors (Astro UI → Deployment → Logs → Worker, or `Astro_GetDeploymentLogs` with sources=["worker"]); the scheduler window shows no worker activity at all. 2) In the deployment's `default` worker queue settings, confirm the minimum worker count is ≥ 1 and that workers are in a Running/healthy state; if workers are crash-looping or scaled to zero, trigger a redeploy (or bump the min worker count) to bring healthy workers back. 3) After workers recover, clear/re-run the failed runs. Raising `AIRFLOW__SCHEDULER__TASK_QUEUED_TIMEOUT` (default 600s) would only delay the force-fail and will not fix the root cause; do not rely on it. Optionally add `retries` (e.g. `retries=2`) in default queues so transient queue stalls self-heal instead of failing immediately.

EVIDENCE

DAG-LEVEL CHECKS

- 🟢 **DAG-level dagrun_timeout timeout as root cause**
hourly_success_dag defines no dagrun_timeout. The failure signature is not a DAG timeout: the unfinished task 'finish' is upstream_failed (not skipped) and 'always_succeed' is failed. No 'has timed-out' entries appear in scheduler logs for this run. Timeout ruled out.
- 🟢 **Recent code deploy or config change regression**
Most recent deploy was 2026-06-16 (IMAGE_AND_DAG); dags_diff_summary.md reports no DAG files changed between deploys; deployment config changes in the window = 0. No code/config regression.
- 🔴 **Deployment-wide execution health (are other DAGs affected)**
Multiple unrelated DAGs (hourly_failure_dag, freshness_sla_miss, test_dag_*) show the same 'Task stuck in queued/requeue attempts exceeded max' pattern in the same window, indicating a deployment-wide Celery worker availability issue, not a single-DAG problem.

TASK FAILURES

- ```
> always_succeed
> finish
```

- > 8 tool calls · 4 reasoning

Last 14 

Was this summary helpful?  

MEDIUM

P3

PERMANENT



## download\_file task OOM-killed (SIGKILL) on the under-provisioned worker queue in ingest\_files

Confidence: 97%

The `download_file` task in the `ingest_files` DAG runs a Python callable (`consume_memory`) that deliberately allocates ~10MB of memory every 5 seconds in an unbounded loop when the `ENV` environment variable equals `prod`. The task's worker process grew to ~1.68GB and was terminated by the kernel with SIGKILL (signal -9, out-of-memory) on every attempt, exhausting all 3 tries. This is a deterministic, by-design memory-exhaustion failure, not a transient infrastructure problem.

### SUGGESTED FIX

Stop the intentional memory allocation in `consume_memory` in `dags/misc/ingest_files.py` — the loop `memory_hog.append("A" * chunk_size)` (and the trailing `while True:` allocation) will always OOM. If this DAG is meant to do real work, replace the callable with the actual file-download/ingest logic and cap memory usage. If it must remain a demo, either unset/change the `ENV` environment variable so it is not `prod` (the guard `if os.getenv('ENV')=='prod'` gates the allocation), or move the task off the `queue='under-provisioned'` worker queue to one with sufficient memory. To avoid three guaranteed-failing attempts, also set `retries=0` for this task.

### > EVIDENCE

#### DAG-LEVEL CHECKS

- ✅ **DAG-level dagrun\_timeout configured and exceeded**  
No dagrun\_timeout is set in the DAG definition. The run failed because its only task was OOM-killed, not due to a DAG timeout.
- ✅ **Unfinished tasks marked skipped (dagrun\_timeout hallmark)**  
The single task download\_file is marked 'failed' (not skipped), consistent with a task-level OOM rather than a DAG timeout.

#### TASK FAILURES

🔴 > download\_file

> 3 tool calls · 3 reasoning

Last 14  

Was this summary helpful?  

MEDIUM

P2

INTERMITTENT



## Filesystem Isolation Issue: Tasks Cannot Share /tmp Data Across Kubernetes Pods

Confidence: 95%

The disk\_file\_demo DAG fails intermittently because it incorrectly assumes tasks share a local filesystem. The write\_csv task writes to /tmp/airflow\_disk\_demo.csv and read\_csv attempts to read it, but with the Astro (Kubernetes) executor, each task runs in an isolated ephemeral pod. When tasks are scheduled on different pods, read\_csv cannot find the file and fails with FileNotFoundError. The issue became more frequent after adding the 'disk' worker queue with 5 minimum workers on 2026-04-15 at 00:11:42, which increased task distribution across pods. The DAG showed 3 consecutive successes (when tasks happened to land on the same pod) followed by 2 failures (when distributed). Fix by using XCom for data passing or shared cloud storage (Azure Blob/S3) instead of local /tmp.

### DAG-LEVEL CHECKS

#### ✓ DAG timeout configuration check

No dagrun\_timeout configured in DAG definition. DAG run duration (~5 seconds) is normal for this workload.

#### ✓ DAG parsing verification

Scheduler logs show successful DAG parsing from /tmp/airflow/dag\_bundles/astro/main/dags/2026-04-15T00:38:57.4332490Z/dags/disk.py with no import errors.

### TASK FAILURES

#### 🔴 read\_csv

##### ✓ Task execution started

Task logs show execution started at 2026-04-15T00:53:23.984982Z, DAG bundle loaded successfully

##### ✓ Worker pod received and executed task

Task logs show worker PID 65 executed the task, OpenLineage START event emitted

##### 🔴 File exists at expected path /tmp/airflow\_disk\_demo.csv

FileNotFoundError: [Errno 2] No such file or directory: '/tmp/airflow\_disk\_demo.csv' - file written by write\_csv task on different pod is not accessible

##### 🔴 Task runs on same worker pod as write\_csv

With Astro executor and disk worker queue (min=5 workers), tasks scheduled on different ephemeral Kubernetes pods with isolated filesystems

#### ✓ Suggested Fix

Refactor the DAG to use XCom for passing data between tasks instead of local filesystem. For the small CSV in this example, return the CSV content as a string from write\_csv and accept it as a parameter in read\_csv. For larger files, use shared storage like Azure Blob Storage (already configured for logs) or S3 with appropriate Airflow connections.

# WHERE ARE WE HEADING

A look at what's next.

ASTRONOMER



# Otto's future - a ready fix when you start your day

What's better than a good investigation? One that already happened. Auto-triggered the moment a task fails, Otto investigates before anyone's even seen the alert - root cause identified, evidence attached, and a PR with the fix already open. You don't start your morning investigating. You start it reviewing a fix that's already waiting.

# OpenLineage next steps

## More detailed lineage

We're adding a dedicated lineage facet to improve accuracy and detail.

Each output now maps to the exact inputs that produced it, instead of assuming every input feeds every output. Missing or partial lineage can be stated explicitly, dependencies can cross types (e.g. job to dataset), and more.

## More native coverage

Lineage extraction works out of the box, at both the operator and hook level.

We already provide native OpenLineage support for over 80 operators and over 30 hooks across over 30 Airflow providers - from Amazon, Google, and Snowflake to Postgres, Databricks, and more - and we're continuously adding new ones as providers evolve.

## Data Quality provider (in review)

Declarative rules, a dedicated operator, and persisted per-rule history, running on the SQL connections you already have, with hooks to gate downstream tasks against Airflow Assets.

A strong signal the community wants data quality standardized, not reinvented per team.

# Building an Airflow agent? Start with OpenLineage

@kacpermuda  
Muda.Kacper@gmail.com

ASTRONOMER

