Engineer

Pipeline

a run of steps carrying data from source to use

What it is

A pipeline, in full a data pipeline, is a set of jobs that run in a fixed order, each taking what the job before it produced, so that raw records from the systems that create them end up cleaned, combined and stored where reports, dashboards or other programs can read them. The name comes from the Unix shell, where `|` feeds one command's output into the next. Most pipelines run unattended, on a schedule or whenever a new file arrives.

Few pipelines are a straight line. A job that loads the day's sales may need both the currency rates and the customer list first, while those two can run side by side. The jobs and the "must finish first" arrows between them form a directed acyclic graph: every arrow points one way, and no job can end up waiting on itself. Apache Airflow, an open-source platform for scheduling and monitoring such workflows, calls each graph a Dag, written that way rather than DAG because its meaning in Airflow has grown past the bare mathematical structure.

A batch pipeline wakes on a schedule or an event, processes a bounded slice of data, such as one day of meter readings, and stops. A streaming pipeline keeps running and passes each event on as it arrives. Airflow describes itself as a platform for orchestrating batch workflows, best suited to work with a clear start and end that runs on a schedule.

The simplest pipeline is a crontab line that starts a script at two in the morning. That holds until several scripts must run in order, one of them breaks, and someone has to work out which ran and what to start again. An orchestrator takes over that bookkeeping: it holds the graph, starts each job once the jobs upstream of it have succeeded, retries the ones that break, keeps a log per job and per run, and records which periods of data each run covered. In Airflow every

scheduled run covers a data interval. A daily run for 1 January starts just after midnight on the 2nd, once the day is over, and its jobs can read bounds such as `data_interval_start` to know exactly which slice of data is theirs.

Use the concept in a decision

Put “What is a pipeline?” to work in a real decision. We can help relate the concept to the system you are building or buying. Numbers arriving where they are needed, on a schedule, without anyone copying a spreadsheet at the end of the month.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. One infrastructure plan for 14 departments — 14 research departments on one infrastructure plan

One infrastructure plan for 14 departments Get in touch →

Why it matters

A pipeline turns a procedure into code. The order of the jobs, the checks between them, what happens when one breaks and how often everything runs move out of one person’s memory into files that can be reviewed, versioned and tested like any other software. Airflow defines its workflows entirely in Python, an approach it calls workflows as code, and names version control and testing among the benefits.

A fault also gets an address. When the whole process is one long script, a crash at minute forty means starting again from minute zero. Split into jobs with recorded state, a broken run shows which job stopped and why, and only that job and the ones after it need to run again. Airflow lets an operator rerun only the failed tasks, or backfill runs for an earlier period, which is how a fix made on Wednesday reaches Monday’s data.

Repetition becomes safe only when each job is written for it. Airflow’s best-practice guide asks each task to behave like a database transaction that never leaves partial results behind and to “produce the same outcome on every re-run”. A job built that way can be retried by the scheduler at three in the morning, rerun after a fix or replayed for last month, and nobody has to inspect the target first.

The price is software to own. An orchestrator is one more service with its own database, upgrades and permissions, and every pipeline in it is code that needs review and tests. For two scripts on one schedule, cron and a log file can be the

honest answer. The case for an orchestrator grows with the number of jobs that depend on each other, the number of people who need to see what ran, and the cost of a morning report built on a load that stopped halfway.

What it looks like in practice

Running many pipelines the same way was the problem on one platform, where twenty GIS data pipelines and the application-data ETL were delivered and watched by hand, with a standing risk that one would stop quietly and leave the data downstream wrong before anyone noticed. All of them came under automated delivery, with scheduling, logging and failure handling standardised so that every pipeline behaved alike and a broken one showed, and were folded into the existing infrastructure automation and reporting. All twenty then ran automatically and predictably, with current data and nobody walking it through by hand.

A pipeline can also build something whole in one command. At one documented stage, a database-driven CV and portfolio generator drew every document from one SQLite database of 23 tables, assembled by a 19-step pipeline that ran from an empty file to a complete database. The steps were ordered and each was written to be safe to repeat, so the pipeline could be run against an existing database without duplicating a row. Forty-one Python modules, 10,580 lines in all, then rendered six output formats from that one source, with only two runtime dependencies. These counts describe the implementation at that documented stage.

Inside a single job, speed matters as much as order. An hourly electricity aggregation pipeline takes JSONL data reported every hour, every 15 minutes or at irregular times, aligns it to strict hourly intervals, sums the production mix, adds imports and subtracts exports, and produces one row per hour per zone. It was built three ways, in Python, in JQ and in SQL on PostgreSQL, and each version was benchmarked on one day and thirty days of data. The Python version was the fastest, at 34 ms for one day and 180 ms for thirty, and its code was written so that it could be extended to more zones or integrated into an ETL pipeline.

Where it goes wrong

A re-run gives a different answer. A daily job appends its totals with `INSERT`, breaks halfway, and the scheduler retries it, so the rows written before the break are now there twice. A job that selects everything since the current time minus a day reads a different slice on the retry than on the first attempt, and a job that reads the latest version of an input picks up whatever changed in between. Airflow's best-practice guide names all three: replace `INSERT` with an upsert, read and write a specific partition keyed on the run's `data_interval_start`, and keep `now()` out of any critical computation. For an hourly summary table in PostgreSQL, one plain way is to replace the whole interval inside a single transaction.

```
BEGIN;
DELETE FROM hourly_load
WHERE hour_start >= :'start' AND hour_start < :'end';
INSERT INTO hourly_load (zone, hour_start, mwh)
SELECT zone, date_trunc('hour', read_at), sum(mwh)
FROM reading
WHERE read_at >= :'start' AND read_at < :'end'
GROUP BY zone, date_trunc('hour', read_at);
COMMIT;
```

Run as `psql -v ON_ERROR_STOP=1 -v start=... -v end=... -f hourly_load.sql`, with the start and end of the run's data interval passed in by the orchestrator, the job deletes and rewrites exactly that interval's rows, so a second run leaves the table as the first one did. The `:'start'` form is psql's own syntax for inserting a variable quoted as an SQL literal.

A job finishes green and the data is wrong. An orchestrator sees whether a job's process exited cleanly, and that is all it sees. psql is a sharp example: by default it carries on after an error in a script and returns 0 to the shell, so a step that runs `psql -f` reports success even when a statement inside it broke. With `ON_ERROR_STOP` set, psql stops at the first error and exits with code 3, so the step fails where it should. A clean exit still only proves the program ran, which is why Airflow's guide suggests self-checks: a downstream task that confirms the partition was written and that the data looks right before anything reads it. For a run that hangs rather than breaks, Deadline Alerts, added in Airflow 3.1 to replace the SLA feature removed in 3.0, trigger a callback, such as an email, once a run passes a time threshold.

Switching a pipeline back on starts a flood of runs. With `catchup=True`, the scheduler creates a run for every data interval it skipped since the start date, and the Airflow documentation adds that catchup is also triggered when a Dag is turned off for a while and then re-enabled. A daily pipeline paused for a month can come back with thirty runs queued against the same source. The default, `catchup_by_default=False`, makes the scheduler create a run only for the latest interval, and the documentation says a Dag whose jobs read the current time instead of their interval should keep catchup off. History is better filled on purpose with `airflow backfill create`, which takes a date range and a limit on parallel runs through `--max-active-runs`.

Heavy code at the top of a Dag file slows the scheduler. Airflow parses every Dag file again and again, with at least `min_file_process_interval` seconds between parses, so code at the top level of the file, outside any task, runs far more often than the pipeline itself. A database query, a network call or a slow import there costs time on every parse, and the best-practice guide shows an example in which one such call makes a Dag take an additional 1000 seconds to parse. Keep the top level to declaring tasks and their order, and move expensive imports and calls inside the task functions that need them.

Further reading

- Apache Airflow — the project's own overview: workflows as Python code, batch orchestration, backfills and reruns of only the failed tasks.
- Best Practices — tasks treated as transactions, the same outcome on every re-run, top-level code and self-checks.
- Dags — how a Dag declares its schedule, tasks, dependencies and default arguments such as retries, and where Deadline Alerts fit.
- Dag Runs — data intervals, catchup, backfill, and clearing tasks so they run again.
- Tasks — dependencies between tasks, timeouts, retry policies and the move from SLAs to Deadline Alerts.
- psql — variables and their quoting, `ON_ERROR_STOP`, and the exit codes an orchestrator reads.



Glossary

Choose for your actual constraints

Bring a representative dataset, the questions it must answer and the current bottleneck. We can agree how to measure quality, throughput and recovery.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a pipeline?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/pipeline/