ChatGPT

Basics, Options and Gaps in Apache Airflow

Overview

Apache Airflow is an open-source platform that orchestrates data workflows. It emerged at Airbnb and has since become the de facto standard for managing complex, scheduled data pipelines. Airflow represents workflows as **directed acyclic graphs (DAGs)**, where each node is a discrete **task** and edges define dependencies. Instead of running ad-hoc scripts with cron, Airflow lets data engineers build reproducible pipelines with scheduling, dependency management, retries and monitoring. Dataquest notes that engineers often begin with separate scripts scheduled via cron; as volumes grow these scripts multiply, dependencies tangle and failures cascade [1]. Orchestration solves these problems by coordinating interdependent tasks in the correct order and conditions [2].

Core Concepts and Architecture

- **DAGs (Directed Acyclic Graphs)** – A DAG is the blueprint of a workflow, connecting tasks in a specific order. Airflow treats each DAG as a Python file. DAGs can be triggered on schedules or in response to events [3].

- **Tasks and Operators** – A task is the smallest unit of work; operators encapsulate specific types of tasks (e.g., BashOperator, PythonOperator, PostgresOperator). In Airflow 2.x the *TaskFlow API* allows any Python function decorated with `@task` to become a task and automatically infers dependencies when functions call each other [4].

- **Scheduler** – The scheduler parses DAG files and decides when tasks can run. It checks DAG definitions and triggers tasks as their dependencies complete. In Airflow 2.0 the scheduler was rewritten for high availability and scalability [5].

- **Executor and Workers** – The executor determines how tasks run. A *LocalExecutor* runs tasks in separate processes on the scheduler machine. Remote executors distribute work across multiple machines; options include **CeleryExecutor** (requires RabbitMQ or Redis) and **KubernetesExecutor**, which launches each task in its own container [6]. Airflow 2.10+ supports multiple executors concurrently [7].

- **Metadata Database** – Stores information about DAGs, task instances and users. Development setups use SQLite, but production environments should use PostgreSQL or MySQL [8].

- **Webserver** – Provides a UI to view DAGs, trigger runs, inspect logs, analyse task durations and manage configurations [9].

Supported Options and Capabilities

Airflow has evolved beyond simple cron-like scheduling. Key capabilities include:

Scheduling Options

- **Cron and interval schedules** – DAGs can run on regular intervals using cron syntax (`@daily`, `0 12 * * *`, etc.).
- **Sensors** – Special operators that wait for external conditions (e.g., file sensors, API sensors) before continuing. Dataquest emphasises deferrable sensors and the *triggerer* for efficiency [10].
- **Datasets and assets** – Airflow 2.4 introduced *datasets* and Airflow 3 added *assets*, allowing DAGs to trigger based on upstream data availability rather than time [11]. Datasets model the readiness of a dataset; assets extend this concept and make it easier to declare data dependencies.
- **Event-driven scheduling** – Airflow 3.0+ can trigger DAGs on external events (e.g., messages from SQS or Kafka) [12]. This removes the need for periodic polling and enables near-real-time reactions.
- **API triggers** – Airflow's REST API lets external systems launch DAGs on demand.

Task Definitions

- **TaskFlow API** – Use `@task` and `@dag` decorators to define tasks as Python functions and automatically manage dependencies. This API simplifies DAG authoring [4].
- **Dynamic task mapping** – Creates tasks at runtime based on input values. Useful for parallelizing work over lists or dictionaries (e.g., one task per file or per model).
- **Custom operators** – Extend `BaseOperator` to implement bespoke behavior. Airflow's modular design encourages building custom operators, sensors or hooks.
- **Deferrable operators** – Release worker resources while waiting for external events (e.g., asynchronous file arrival) to improve scalability and reduce costs.

Executions and Scalability

- **Executor choices** – Users can select a **LocalExecutor** for simplicity, **CeleryExecutor** for distributed processing with a message queue, or **KubernetesExecutor** for containerized tasks and autoscaling [6]. Airflow 2.10 introduced the ability to configure multiple executors simultaneously and assign specific tasks to specific executors [7].
- **Parallelism and concurrency** – Airflow allows hundreds of tasks to run concurrently within a DAG if there are no dependencies, enabling high throughput [13].
- **Deferrable sensors and idempotent backfills** – Airflow frees worker resources while waiting on sensors [10] and supports backfill operations to rerun past dates safely.

Integration and Extensibility

- **Hooks and providers** – Airflow ships with many hooks for databases (PostgreSQL, MySQL, Oracle), cloud services (AWS, GCP, Azure), messaging platforms (Slack, Kafka), BI tools (Tableau) and more [14]. These simplify connecting to external systems.
- **REST API** – Airflow 2.0 introduced a full REST API for programmatic control of DAGs and tasks [15].
- **Variables, connections and secrets** – Centralize configuration and credentials.

- **Assets for ML/AI and hybrid workflows** – Airflow 3 supports event-driven scheduling and dynamic mapping; features like dynamic task mapping and custom XCom backends make Airflow more capable for ML and AI pipelines [16].

Limitations and Missing Features

Despite its strengths, Airflow is not suitable for every use case and has gaps:

1. **Not a processing engine or streaming system** – Airflow orchestrates tasks but delegates heavy compute to external tools. It is designed for batch or micro-batch workflows and is not a real-time streaming engine [17]. Streaming workloads should use systems like Kafka or Flink, with Airflow orchestrating infrastructure and downstream processing [18].

2. **Requires programming skills** – Airflow follows a “workflow as code” philosophy, meaning users must write Python. This makes it unsuitable for non-developers [19].

3. **No built-in data quality or lineage features** – Airflow doesn’t include native data quality checks or lineage tracking. The Decube article notes that it lacks built-in data quality control and lineage, requiring additional tools to detect errors and inconsistencies [20].

4. **No DAG versioning** – Airflow does not record DAG version history. When tasks are removed from a DAG, they disappear from the UI along with their metadata; there’s no way to roll back to previous pipeline versions [21]. Users must implement their own version control via Git or maintain separate DAG IDs.

5. **Steep learning curve and insufficient documentation** – Many users find Airflow’s documentation abridged and lacking step-by-step guides. Over 36 percent of surveyed users want improved documentation [22], and newcomers often struggle with the initial setup and concepts [23].

6. **Complex production setup** – Deploying Airflow at scale requires managing multiple components (scheduler, workers, database, message brokers) and configuring executors like Celery or Kubernetes. This complexity can lead organisations to hire consultants or use managed services [24].

7. **Limited UI support for multi-team collaboration** – Airflow’s built-in UI is powerful for monitoring but lacks advanced collaboration features such as DAG-level ownership assignments or interactive approvals (although AIP 90 is under development to add human-in-the-loop capabilities [25]).

8. **Data quality and lineage limitations** – As highlighted above, Airflow does not manage data quality or lineage. The lack of built-in lineage information makes it difficult to trace dependencies and understand the impact of changes [26].

What’s Needed or In Development

To address these gaps, the Airflow community and ecosystem are working on several improvements:

1. **Human-in-the-loop interactions** – Airflow Improvement Proposal 90 (AIP 90) aims to allow users to pause workflows and provide inputs or approvals mid-execution, beneficial for ML and AI tasks [25].

2. **Enhanced event-driven orchestration** – Airflow 3 introduces event-driven scheduling that can trigger DAGs upon receiving messages from systems like SQS or Kafka [12]. Further enhancements will expand the range of supported event sources.

3. **Improved documentation and onboarding** – The community is actively expanding tutorials, best practices and how-to guides to lower the learning curve [22].

4. **Native lineage and observability** – Tools in the ecosystem (e.g., OpenLineage) and features like Airflow Assets aim to capture lineage information. Future releases may integrate lineage more deeply into the core platform.

5. **DAG versioning** – Although not yet available, there is ongoing discussion about adding DAG versioning. Until then, users should store DAG definitions in version control and maintain backward-compatible DAG IDs.

6. **Simplified deployment** – Managed services like Astronomer Astro and Google Cloud Composer abstract away infrastructure management, provide autoscaling and high availability, and integrate observability tools. Continued work on easier local development environments (e.g., Airflow CLI, Docker images) aims to streamline setup.

7. **Better integrations for data quality** – While Airflow itself doesn’t handle data quality, the ecosystem offers tools like **dbt tests**, **Great Expectations** and **Marquez**. Integrating these more tightly with Airflow could provide built-in testing and lineage.

Summary

Airflow is a powerful orchestrator for batch and micro-batch workflows. It provides flexible scheduling (time-based, sensor-based, dataset/asset-based and event-driven), modular tasks via the TaskFlow API, distributed execution through different executors, and extensive hooks to integrate with diverse systems. Its open source community drives innovation and features such as dynamic task mapping and event-driven DAGs. However, Airflow is not a streaming engine and lacks built-in data quality, lineage and versioning features. Running it in production requires significant infrastructure and Python expertise. Ongoing initiatives like AIP 90, event-driven enhancements, improved documentation and integrations with data quality and lineage tools aim to address these gaps and broaden Airflow’s applicability.

---

[1] [2] [3] [4] [10] Introduction to Apache Airflow – Dataquest
https://www.dataquest.io/blog/introduction-to-apache-airflow/

[5] Debunking myths about Airflow’s architecture and performance
https://www.astronomer.io/blog/debunking-myths-about-airflows-architecture-and-performance/

[6] [8] [9] [13] [14] [15] [19] [21] [22] [23] [24] The Good and the Bad of Apache Airflow Platform
https://www.altexsoft.com/blog/apache-airflow-pros-cons/

[7] Executor — Airflow 3.1.5 Documentation
https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/executor/index.html

[11] [12] [16] [17] [18] [25] Debunking myths about Airflow’s use cases
https://www.astronomer.io/blog/debunking-myths-about-airflows-use-cases/

[20] [26] Why Apache Airflow is not the best tool for data quality checks? | Decube
https://www.decube.io/post/why-apache-airflow-is-not-the-best-tool-for-data-quality-checks