Architectural Foundations and Orchestration Dynamics: A Comprehensive Analysis of Apache Airflow for Enterprise Data Engineering

The evolution of Apache Airflow from a centralized task scheduler into a distributed, data-aware orchestration platform represents a critical advancement in the management of complex data lifecycles. Originally developed to solve the problem of fragmented cron jobs, Airflow has matured through its 2.x and 3.x iterations to provide a framework where workflows are treated as Directed Acyclic Graphs (DAGs). This architectural paradigm allows for a rigorous separation of concerns between workflow definition, scheduling logic, and task execution.[1] The current state of Airflow orchestration emphasizes security, scalability, and modularity, particularly with the transition toward the Airflow 3.0 architecture, which decouples user-defined code from internal system resources through a centralized API server.[3]

Section 1: Core Architecture and Component Interaction

The Airflow architecture consists of several specialized components that operate asynchronously to ensure the reliable execution of workflows. Understanding the precise flow of information between these components is essential for both certification and production system design. In legacy architectures, nearly every component maintained a direct connection to the metadata database, creating significant overhead and security risks. The modern architecture, specifically within Airflow 3.0, introduces a mediator in the form of the API Server to mitigate these issues.[5]

The Functional Core: Scheduler, API Server, and Metadata Database

The Scheduler serves as the primary intelligence of the Airflow environment. It continuously monitors all tasks and DAGs, determining which task instances are eligible for execution based on their defined dependencies and scheduling rules.[1] Behind the scenes, the scheduler initiates a persistent loop that identifies active DAGs, creates DagRun objects, and selects schedulable TaskInstance objects while respecting pool limits and concurrency constraints.[8]

In the Airflow 3.0 paradigm, the API Server (often implemented as a FastAPI service) serves as the gatekeeper for the Metadata Database. This database, typically PostgreSQL or MySQL, stores the definitive state of the environment, including DAG runs, task instances, XComs, and connection metadata.[1] By forcing Workers and the UI to communicate through the API Server, Airflow achieves a level of component isolation that prevents user-defined code from executing direct, potentially malicious queries against the database.[3]

The Execution Tier: Executors and Workers

The Executor is a configuration property of the scheduler that dictates the strategy for running tasks. It does not perform the work itself but manages the allocation of tasks to workers.[1] The choice of executor significantly impacts the scalability of the environment.

Executor Type	**Execution Mechanism**	**Ideal Environment**	**Scalability**
SequentialExecutor	Single-process, single-threaded execution.	Local development and testing.	None (Serialized).
LocalExecutor	Parallel processes on the same machine.	Small to medium workloads on single nodes.	Vertical.
CeleryExecutor	Distributed execution across a pool of workers via Redis/RabbitMQ.	High-volume production pipelines.	Horizontal.
KubernetesExecutor	Each task runs in a dedicated, transient pod.	Cloud-native environments with fluctuating load.	Dynamic (Elastic).

The Workers are the actual compute units where task code is executed. In a distributed setup, workers poll a queue (managed by the executor) to retrieve tasks and execute the associated Python or Bash code.[1] Upon completion, the worker reports the result back to the API Server, which updates the state in the metadata database.[9]

The Deferral Layer: The Triggerer

The Triggerer is an optional but highly efficient component introduced to handle long-running, idle tasks such as sensors or external process waiters.[1] Standard operators occupy a full worker slot while waiting for a condition to be met, which leads to resource starvation in high-concurrency environments. Deferrable operators, however, yield their execution and state to the Triggerer, which uses an asynchronous event loop (asyncio) to monitor thousands of conditions simultaneously with minimal resource footprint.[11]

Data Flow Scenario: The Journey of a Task Instance

To illustrate the interaction of these components, consider a standard scheduled workflow. The process begins when the DAG Processor parses the Python files in the DAG directory and serializes the DAG structure into the metadata database.[5]

1. **Parsing and Serialization**: The DAG Processor converts Python code into a serialized format, allowing the Scheduler to read DAG definitions without executing the author's Python script repeatedly.[5]
2. **Scheduling Decision**: The Scheduler identifies that a DAG is due for execution. it creates a DagRun in the metadata database.[8]
3. **Task Readiness**: The Scheduler identifies a task within the DagRun whose upstream dependencies are met. It transitions the task state from None to Scheduled and then Queued.[14]
4. **Executor Submission**: The Executor places the task on a message broker or queue (e.g., Redis for Celery).[5]
5. **Worker Execution**: A Worker picks up the task from the queue. It requests necessary resources (like Airflow Connections) from the API Server and begins execution.[5]
6. **State Reporting**: Upon completion, the Worker sends a status update to the API Server. The API Server records the Success or Failure in the database, triggering the Scheduler to evaluate the next downstream tasks.[9]

Section 2: DAG Parsing and Execution Model

A Directed Acyclic Graph (DAG) defines the logical structure of a workflow. However, the performance of the entire Airflow environment is often tied to how these files are parsed and the quality of the code contained within them.[10]

The Parsing Mechanism and Performance Bottlenecks

The DAG Processor scans the configured DAG bundles at a regular interval (defaulting to 30 seconds).[17] During this scan, the processor executes the Python file to extract DAG objects. A critical distinction must be made between "parsing time" and "task execution time." Code written at the top level of a DAG file—code that is not encapsulated within a task—is executed every single time the file is parsed.[18]

If a developer includes heavy computations, API requests, or database queries at the top level of a DAG file, it introduces significant latency. For instance, a Variable.get() call at the module level forces a database connection every 30 seconds per DAG file, potentially overwhelming the metadata database as the number of DAGs grows.[19]

Correcting Patterns for Efficient Parsing

To maintain a responsive environment, developers should utilize the TaskFlow API and delay heavy operations until the task actually runs on a worker.

Inefficient Pattern (Heavy Top-Level Code):

Python

```python
from airflow.sdk import DAG
from airflow.providers.standard.operators.python import PythonOperator
import requests

# BAD: This API call runs every 30 seconds during parsing
response = requests.get("https://api.example.com/config")
config_data = response.json()

with DAG(dag_id="bad_parsing_dag", schedule="@daily") as dag:
    t1 = PythonOperator(
        task_id="process_task",
        python_callable=lambda: print(f"Processing with {config_data}")
    )
```

Optimized Pattern (Delayed Execution):

Python

```python
from airflow.sdk import dag, task
from airflow.providers.standard.operators.python import PythonOperator

@dag(dag_id="good_parsing_dag", schedule="@daily")
def optimized_dag():

    @task
    def fetch_config():
        # GOOD: This only runs when the task is executed on a worker
        import requests
        return requests.get("https://api.example.com/config").json()

    @task
    def process_task(config):
        print(f"Processing with {config}")

    process_task(fetch_config())

optimized_dag()
```

By moving imports and network calls into the tasks, the parsing time remains under one second, ensuring the scheduler can loop through the entire DAG directory without being blocked by network I/O or CPU-intensive logic.[18]

Import-Time Execution and Serialization

The Scheduler does not read the .py files directly during its scheduling loop; instead, it relies on serialized versions of the DAGs stored in the database.[5] This serialization process separates the "Authoring" environment from the "Scheduling" environment. If the DAG Processor fails to parse a file due to a syntax error or a timeout (governed by dagbag_import_timeout), the DAG will disappear from the UI, and its scheduled runs will stall.[20]

Section 3: Scheduling, Intervals, and Timetables

Scheduling is perhaps the most nuanced aspect of Airflow, governed by the interaction of start_date, schedule, and catchup.[1] Airflow operates on an interval-based scheduling system, which is fundamentally different from a point-in-time trigger like a standard cron job.[23]

The Data Interval and Logical Date

Each DAG run is associated with a specific "data interval," representing the period of data the run is responsible for processing.[23] For a daily DAG, the data interval might span from midnight of Day 1 to midnight of Day 2. The key insight is that the DAG run for Day 1 will not trigger until the clock reaches the start of Day 2, ensuring that all Day 1 data is complete before processing begins.[8]

The "Logical Date" (formerly execution_date) marks the beginning of this interval.[15] In modern Airflow versions, the UI displays several relevant timestamps to provide clarity on the timeline:

Timestamp	**Definition**
Logical Date	The reference point for the interval (start of

	the period).
Run After	The earliest time the DAG run can actually be triggered (end of the period).
Start Date (Run)	The actual physical timestamp when the task began running.
End Date (Run)	The physical timestamp when the task completed.

Cron Scheduling Field Breakdown

Airflow supports standard cron expressions, which are parsed as minute hour day_of_month month day_of_week.[1]

Example 1: 0 0 * * *

- Triggers at midnight every day.
- Data interval: 2025-01-01 00:00 to 2025-01-02 00:00.
- Trigger time: 2025-01-02 00:00.

Example 2: */10 * * * 1-5

- Triggers every 10 minutes, but only on weekdays (Monday-Friday).[1]
- Field 1: */10 (every 10 minutes).
- Field 5: 1-5 (Monday to Friday).

Timetable Comparisons: CronDataInterval vs. CronTrigger

With the introduction of the Timetable API, Airflow formalized the difference between interval-based and trigger-based scheduling.[22]

1. **CronDataIntervalTimetable**: This is the default behavior for string-based cron expressions in legacy Airflow. It creates continuous, non-overlapping data intervals. It is "backfill-friendly" because if a system goes down, it will attempt to trigger runs for every missed interval to ensure no data is lost.[22]
2. **CronTriggerTimetable**: This behaves more like a traditional cron utility. It represents a specific point in time and does not inherently carry the concept of a "data range." If the system is down during a trigger window, that run is simply missed, and it will not automatically catch up unless manually intervened.[22]

Timeline Example: enablement at 3:00 PM on January 31st for a @daily DAG.

- **CronDataInterval**: Immediately triggers a run for January 30th (the interval that just ended).[25]
- **CronTrigger**: Waits until midnight on February 1st to trigger the first run.[25]

Specialized Timetables: Delta and Events

For workflows that do not fit a calendar-based cron, Airflow provides DeltaTriggerTimetable and EventsTimetable.[25]

- **DeltaTriggerTimetable**: Used for fixed deltas from the previous run. If a DAG is scheduled with timedelta(hours=3), it will run exactly every 3 hours. This is distinct from a cron 0 */3 * * *, which triggers at 3, 6, 9, and 12 on the clock regardless of when the previous run actually finished.[1]
- **EventsTimetable**: Used for irregular events, such as a list of corporate holidays or product launch dates. The scheduler will only create DAG runs on the specific datetime objects provided in the list.[25]

Catchup vs. Backfill Scenarios

Catchup is an automatic feature. If a DAG has a start_date of one month ago and is enabled today with catchup=True, the scheduler will immediately queue 30 DAG runs to fill the historical gap.[1] Backfill is a manual process initiated via the CLI (e.g., airflow dags backfill -s 2025-01-01 -e 2025-01-05) or by clearing historical tasks in the UI.[1]

Section 4: Tasks, Communication, and Lifecycle

Tasks are the atomic units of work. The interaction between tasks, their states, and how they share data is fundamental to the orchestration model.[2]

Operators and the TaskFlow API

Traditional operators like the BashOperator or PythonOperator are defined as standalone objects. The TaskFlow API (using the @task decorator) streamlines this by treating Python functions as tasks.[2]

Classic BashOperator Example:

Python

```python
from airflow.providers.standard.operators.bash import BashOperator
```

```
run_bash = BashOperator(
    task_id="classic_bash",
    bash_command="echo 'Hello from classic operator'"
)
```

Modern TaskFlow @task.bash Example:

Python

```
from airflow.sdk import task

@task.bash
def run_bash_task():
    # The return string is what gets executed
    return "echo 'Hello from TaskFlow'"

run_bash_task()
```

XComs: Mechanism and Limitations

XComs (Cross-Communications) facilitate data sharing between tasks. In TaskFlow, the return value of a function is automatically pushed to XCom under the key return_value.[4]

Explicit XCom Example:

Python

```
@task
def sender_task(ti=None):
    # Manual push
    ti.xcom_push(key="metadata", value={"id": 123, "status": "active"})

@task
def receiver_task(ti=None):
    # Manual pull
    meta = ti.xcom_pull(task_ids="sender_task", key="metadata")
    print(f"Processing ID: {meta['id']}")
```

A critical limitation of XComs is data volume. Because XCom values are serialized (typically as JSON) and stored in the metadata database, they are restricted by the database's column limits.[1] Large dataframes or files should never be passed via XCom. Instead, a task should write the file to an external store (S3/GCS) and pass only the URI as a reference.[1]

Task Instance Lifecycle and States

Every TaskInstance undergoes a state machine transition. Monitoring these colors and states in the Grid View is essential for debugging.[14]

1. **None**: Created in the database but dependencies not yet checked.
2. **Scheduled**: Dependencies met; the scheduler has decided it should run.[14]
3. **Queued**: Assigned to an executor; waiting for a worker slot.[14]
4. **Running**: Currently being executed by a worker.[14]
5. **Success / Failed**: Terminal states.
6. **Upstream_failed**: A state applied when a parent task fails, preventing execution of children.[14]

Trigger Rules and Branching

Trigger rules determine when a task should run based on its parents' states.

- **all_success (Default)**: All parents must succeed.[30]
- **all_done**: Runs as soon as parents are finished, regardless of success or failure. This is often used for "Cleanup" tasks.[30]
- **one_success**: Runs as soon as at least one parent succeeds; does not wait for others to finish.[30]
- **none_failed**: Runs if all parents either succeeded or were skipped. This is necessary when using a BranchPythonOperator.[30]

Branching Confusion Example: If Task A branches to Task B or Task C, and Task D is downstream of both B and C, Task D must have a trigger rule of none_failed or one_success. If Task D is left at the default all_success, it will stay in a Skipped state forever because either Task B or Task C will always be skipped.[31]

Section 5: Orchestration and Data-Aware Scheduling

Orchestration across multiple DAGs has evolved from direct triggering to data-aware dependencies.[1]

Dataset/Asset-Based Scheduling

In Airflow 3.0, "Datasets" have been renamed to "Assets".[7] This feature allows for loosely coupled, event-driven pipelines.

The Producer DAG:

Python

```
from airflow.sdk import Asset, dag
from airflow.providers.standard.operators.bash import BashOperator

# Define a logical data asset
s3_file = Asset("s3://bucket/processed_data.csv")

@dag(schedule="@daily", start_date=datetime(2025, 1, 1))
def producer_dag():
    BashOperator(
        task_id="create_file",
        bash_command="echo 'data' > /tmp/data.csv",
        outlets=[s3_file] # Updates the asset on success
    )
```

The Consumer DAG:

Python

```
@dag(schedule=[s3_file], start_date=datetime(2025, 1, 1))
def consumer_dag():
    # This DAG triggers ONLY when the s3_file asset is updated
    ...
```

Comparison with TriggerDagRunOperator

The TriggerDagRunOperator is an explicit, synchronous method of DAG orchestration. The parent DAG must know the exact dag_id of the child.[1]

Feature	**TriggerDagRunOperator**	**Asset-Based Scheduling**
Coupling	Tight (Parent calls Child).	Loose (Child waits for Data).

Parameters	Can pass custom conf JSON.	Primarily depends on data updates.
Visibility	Hard to track in a global graph.	High visibility in the "Assets" tab.
Concurrency	Can wait for completion.	Entirely asynchronous.

Airflow Fundamentals Exam Traps You Must Understand

To succeed in the certification, one must distinguish between intuitive assumptions and the actual mechanics of the Airflow engine.

Trap 1: The First Run Trigger

Example: A DAG has a start_date of 2025-01-01 and a schedule of @daily. The user expects it to run on Jan 1st. **Correction**: The first run will trigger on **Jan 2nd**. A DAG run for an interval only starts once that interval is complete.[8]

Trap 2: The Role of the Executor

Example: Question asks "Which component executes the actual Python code of a task?" and lists Executor as an option. **Correction**: The **Worker** executes the code. The **Executor** merely decides how the task is queued and which worker gets it.[1]

Trap 3: Catchup and Paused DAGs

Example: A DAG is paused for 3 days. It has catchup=True and a @daily schedule. What happens when unpaused? **Correction**: Airflow will immediately trigger **3 DAG runs** to fill the missed daily intervals.[1]

Trap 4: Variables and Database Connections

Example: A student uses Variable.get("my_key") at the top level of their script to define a task's parameter. **Correction**: This is an anti-pattern that creates a **new DB connection every 30 seconds** per DAG during parsing. Use Jinja templates like {{ var.value.my_key }} instead.[18]

Trap 5: Logical Date in a Backfill

Example: A user runs a backfill today for 2024-01-01. They expect {{ ds }} to be today's date. **Correction**: {{ ds }} will be **2024-01-01**. The logical date is pinned to the interval being processed, regardless of the actual physical time of execution.[15]

Trap 6: Trigger Rule all_success with Skips

Example: A BranchOperator skips Task B. Task C is downstream of Task B and has the default trigger rule. **Correction**: Task C will be **Skipped**. The all_success rule requires every upstream task to have a state of Success. Skipped is not Success.[15]

Trap 7: XCom Size Limits

Example: A task returns a 2GB log file as a string to pass to the next task. **Correction**: The task will likely **fail or crash the metadata database**. XComs are for small metadata, not large data payloads.[1]

Trap 8: Environment Variable Precedence

Example: A variable API_KEY is set in the UI as '123' and as an environment variable AIRFLOW_VAR_API_KEY as '456'. **Correction**: The task will receive **'456'**. Environment variables take precedence over the UI/Metadata database.[40]

Trap 9: Sensor Default Timeout

Example: A sensor is waiting for a file that arrives once a week. The user sets no timeout. **Correction**: The sensor will fail after **7 days**. This is the default timeout for all sensors.[1]

Trap 10: start_date and now()

Example: Developer sets start_date=datetime.now(). **Correction**: This makes the start_date dynamic. Every time the DAG parses, the "first interval" moves forward, and the DAG **may never run**.[20]

Trap 11: Task Instance State Transition

Example: Question asks for the state between Scheduled and Running. **Correction**: The state is **Queued**. The executor has acknowledged the task and is placing it into the worker queue.[1]

Trap 12: Connection UI dropdown

Example: Snowflake is missing from the connection dropdown in a fresh Airflow install. **Correction**: You must **install the Snowflake provider package** separately. Providers are no longer bundled with Airflow core.[1]

Trap 13: depends_on_past vs Upstream

Example: Task B depends on Task A. Task B also has depends_on_past=True. **Correction**: Task B will only run if Task A (in the *current* run) succeeds **AND** Task B (in the *previous* run) succeeded.[15]

Trap 14: XCom Pickling Security

Example: In Airflow 3.0, a user tries to pass a custom Python class instance through XCom. **Correction**: This will **fail** because pickling is disabled by default for security. Users must use JSON-serializable objects (dicts, lists, strings).[42]

Trap 15: DAG ID Duplication

Example: Two different teams name their DAG process_data in different files. **Correction**: The scheduler will struggle to reconcile which DAG is "correct," leading to **unpredictable task execution**.[1] DAG IDs must be globally unique.

Works cited

1. Certification Exam airflow.pdf
2. Architecture Overview — Airflow 3.1.7 Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/overview.html
3. Best practices for migrating from Apache Airflow 2.x to Apache Airflow 3.x on Amazon MWAA | AWS Big Data Blog, accessed February 24, 2026, https://aws.amazon.com/blogs/big-data/best-practices-for-migrating-from-apache-airflow-2-x-to-apache-airflow-3-x-on-amazon-mwaa/
4. Upgrading to Airflow 3 — Airflow 3.1.7 Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/installation/upgrading_to_airflow3.html
5. Apache Airflow® components | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/airflow-components
6. Apache Airflow 3.0 Release: Architectural Innovation drives a Data-Centric Paradigm (Part 1) - Medium, accessed February 24, 2026, https://medium.com/compile-compose/apache-airflow-3-0-release-architectural-innovation-drives-a-data-centric-paradigm-part-1-68a916786ea2
7. Introducing Apache Airflow 3 on Amazon MWAA: New features and capabilities - AWS, accessed February 24, 2026, https://aws.amazon.com/blogs/big-data/introducing-apache-airflow-3-on-amazon-mwaa-new-features-and-capabilities/
8. Scheduler — Airflow 3.1.7 Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/scheduler.html
9. Apache Airflow 3.0 — A General to Deep Picture — Part 1 | by Khanh Nguyen - Medium, accessed February 24, 2026, https://medium.com/learning-data/apache-airflow-3-0-a-general-to-deep-pictur

e-part-1-d09cee921cd7
10. Apache Airflow for Beginners: Simple Guide, Concepts, and Real-World Workflow Map, accessed February 24, 2026, https://new2026.medium.com/mastering-apache-airflow-mental-models-ecosystem-a-practical-mind-map-d142ce2b6064
11. Deferrable Operators & Triggers - Apache Airflow, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/authoring-and-scheduling/deferring.html
12. Deferrable operators | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/deferrable-operators
13. Deferrable Operators & Triggers - Apache Airflow, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/2.2.5/concepts/deferring.html
14. Tasks — Airflow 3.1.7 Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/tasks.html
15. Concepts — Airflow Documentation - Apache Airflow, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/1.10.9/concepts.html
16. Dags — Airflow 3.1.7 Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dags.html
17. Scaling Airflow to optimize performance | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/airflow-scaling-workers
18. Apache Airflow Best Practices. 23 Tips to Boost Performance &... | by Pedro Pagano | Indicium Engineering | Medium, accessed February 24, 2026, https://medium.com/indiciumtech/apache-airflow-best-practices-bc0dd0f65e3f
19. Stop Creating Bad DAGs - Optimize Your Airflow Environment By Improving Your Python Code | Towards Data Science, accessed February 24, 2026, https://towardsdatascience.com/stop-creating-bad-dags-optimize-your-airflow-environment-by-improving-your-python-code-146fcf4d27f7/
20. FAQ — Airflow 3.1.7 Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/faq.html
21. Use Airflow variables | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/airflow-variables
22. Schedule DAGs in Apache Airflow® | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/scheduling-in-airflow
23. Airflow Data Intervals: A Deep Dive | Towards Data Science, accessed February 24, 2026, https://towardsdatascience.com/airflow-data-intervals-a-deep-dive-15d0ccfb0661/
24. Dag Run Status - Apache Airflow, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/core-concepts/dag-run.html
25. Timetables — Airflow Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/2.5.3/authoring-and-scheduling/timetable.html
26. airflow.timetables.trigger — Airflow 3.1.7 Documentation - Apache Airflow, accessed February 24, 2026,

https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/timetables/trigger/index.html
27. How to Schedule DAGs in Airflow: Timetables and Datasets - Damavis Blog, accessed February 24, 2026, https://blog.damavis.com/en/types-of-schedules-in-apache-airflow-timetables-and-datasets/
28. BashOperator — apache-airflow-providers-standard Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow-providers-standard/stable/operators/bash.html
29. Concepts — Airflow Documentation - Apache Airflow, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/2.0.0/concepts.html
30. Understanding Airflow Trigger Rules: A Comprehensive Visual Guide - Astronomer, accessed February 24, 2026, https://www.astronomer.io/blog/understanding-airflow-trigger-rules-comprehensive-visual-guide/
31. Airflow trigger rules | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/airflow-trigger-rules
32. Understanding trigger rules in Apache Airflow - Damavis Blog, accessed February 24, 2026, https://blog.damavis.com/en/trigger-rules-in-apache-airflow/
33. What is the difference between airflow trigger rule "all_done" and "all_success"?, accessed February 24, 2026, https://stackoverflow.com/questions/41670256/what-is-the-difference-between-airflow-trigger-rule-all-done-and-all-success
34. What are Trigger Rules in Airflow? | by Mihir Samant - Medium, accessed February 24, 2026, https://medium.com/@mihirs202/what-are-trigger-rules-in-airflow-37143c90f528
35. Assets and data-aware scheduling in Airflow | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/airflow-datasets
36. Apache Airflow: Use TriggerDagRunOperator the right way | by Lars Herwegh - Medium, accessed February 24, 2026, https://lshw.medium.com/apache-airflow-use-triggerdagrunoperator-the-right-way-df5db079b95c
37. Cross-DAG dependencies | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/cross-dag-dependencies
38. Webinars - Astronomer, accessed February 24, 2026, https://www.astronomer.io/events/webinars/
39. execution_date in airflow: need to access as a variable - Stack Overflow, accessed February 24, 2026, https://stackoverflow.com/questions/36730714/execution-date-in-airflow-need-to-access-as-a-variable
40. Listeners — Airflow 3.1.7 Documentation, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/administration-and-deployment/listeners.html

41. Managing Variables — Airflow 3.1.7 Documentation - Apache Airflow, accessed February 24, 2026, https://airflow.apache.org/docs/apache-airflow/stable/howto/variable.html
42. Upgrade from Apache Airflow® 2 to 3 | Astronomer Documentation, accessed February 24, 2026, https://www.astronomer.io/docs/learn/airflow-upgrade-2-3
43. Upgrading Airflow 2 to Airflow 3 - A Checklist for 2026 - Astronomer, accessed February 24, 2026, https://www.astronomer.io/blog/upgrading-airflow-2-to-airflow-3-a-checklist-for-2026/