Data Engineering Interview: Questions & Answers.

Data Engineering roles are in high demand globally, with hundreds of thousands of open positions. Employers consistently seek a mix of strong SQL and programming skills, big data pipeline experience, and the ability to design scalable systems. For example, an analysis of 2025 job postings found SQL mentioned in 61% of listings, Python in 56%, and cloud data warehouse or lake technologies in 78%. Core responsibilities include building reliable **ETL/ELT pipelines**, optimizing data storage (e.g. in **warehouses/lakehouses**), ensuring data quality, and collaborating across teams to deliver data for analytics and machine learning.

Orchestration (Airflow, Dagster, Schedulers)

Focus: Workflow orchestration for data pipelines. Covers how tools like Apache Airflow, Dagster, Prefect manage task scheduling, dependencies, retries, and backfilling. Key points include DAG (Directed Acyclic Graph) structure, scheduling intervals, handling failures (retries, alerting), idempotency of tasks, and best practices (like not putting heavy computation in the orchestrator vs. using it mainly to trigger jobs). Interviewers also probe understanding of how to do catchup/backfill and ensure data integrity across runs.

Why It Matters: Orchestration is glue for complex pipelines. Data engineers are expected to design reliable scheduled workflows, handle failures gracefully, and coordinate tasks. Knowing Airflow (the industry standard) or similar tools is often assumed.

---

32. **(Easy)** What is a DAG in Apache Airflow?

Answer:

- A **DAG (Directed Acyclic Graph)** in Airflow is the fundamental structure for a workflowmedium.com. It's a collection of tasks with explicit relationships (dependencies) that has no cycles (no task is repeated in a loop).
- It defines the **order of execution**: nodes in the DAG are tasks, edges represent dependencies ("Task A must run before Task B").
- You define a DAG in Airflow using Python code – instantiate a DAG object, then define tasks (operators) and set their upstream/downstream relationships via >> or set_upstream/set_downstream calls.
- The DAG is **scheduled** as per its schedule_interval (like daily, hourly, or CRON schedule) and Airflow's scheduler will kick off **DAG Runs** (instances) according to that schedule.
- It being acyclic ensures no infinite loops in dependency graph – you can't have A depends on B and B on A, etc.
- DAG is more of a logical container; the actual work is done by tasks/operators. But conceptually, when people refer to an "Airflow DAG", they usually mean a pipeline definition.
- Example: A DAG might represent "daily ETL pipeline" with tasks to extract, transform, load, each depending on the previous.

What interviewers look for: Basic understanding of Airflow's core abstraction. They might want to ensure you know it's not a flowchart you draw in UI, but code-defined. Also DAG vs DAG Run concept could come up.

Common pitfalls:

- Not stating "Directed Acyclic Graph" meaning – usually an interviewer expects those words.
- Confusing a DAG with a task – DAG is the whole workflow, tasks are steps.
- Not highlighting no cycles – that's key (Airflow will actually detect cycles and fail DAG if present).

Follow-up: What happens if you create a cyclic dependency in an Airflow DAG? (Answer: Airflow will detect the cycle when parsing DAG and throw an error; the DAG won't be schedulable).

33. **(Easy)** How does Airflow schedule a DAG for execution? What is a **schedule interval**?

Answer:

- Airflow has a **scheduler** component that parses DAG definitions and, based on each DAG's **schedule_interval**, determines when to create a new **DAG Run** (execution instance)projectpro.io.
- The **schedule_interval** (which can be a cron expression, a timedelta like @daily, @hourly, etc.) defines how often the DAG should run. For example, schedule_interval='0 3 * * *' means run daily at 3:00 AM.
- At the designated times (plus some Airflow specifics like the concept of execution date which is often the interval end), the scheduler creates a DAG run and then sets tasks as **scheduled**. It respects any start_date (the earliest time it should run) and if catchup is enabled, it might create backfill runs for periods since start_date.
- The scheduler ensures that for a given schedule time, all tasks in the DAG run that are not waiting on upstream are triggered to the executor to run on workers.
- Example: If DAG is daily with start_date Jan1, Airflow will on Jan2 create a run for Jan1 execution date (Airflow uses the end of interval as execution date typically). Then Jan3, it will create run for Jan2, etc. (This often confuses newcomers, but basically schedule "fires" after the period).
- The schedule_interval can be **None** or "@once", meaning it only runs when triggered manually (no automatic scheduling).
- The scheduler checks every something (like every heartbeat) if any new runs should be triggered based on current time vs next run time.
- Summarizing: schedule_interval is how you tell Airflow when to run your DAG, and the scheduler process handles creating runs at those times.

What interviewers look for: Familiarity with scheduling basics in Airflow. They might also see if you know things like default behavior (like catchup which by default is True, meaning if Airflow was down or DAG turned off, when turned on it tries to backfill all missed runs unless you disable catchup).

Common pitfalls:

- Not mentioning that Airflow's scheduling is interval-based and uses execution date that's often one interval behind actual run time (though if not asked specifically, it's fine).
- Confusing schedule with triggering tasks – schedule triggers DAG runs, tasks run within those runs as per dependencies.

Follow-up: What is the difference between **execution date** and the actual run date in Airflow? (This is a common confusion – execution date is a logical timestamp often marking the period the data represents. For daily, the run on Jan2 might have execution_date Jan1, meaning the data of Jan1. It's one of those Airflow gotchas).

34. **(Medium)** In Airflow, how would you **backfill** a DAG for dates in the past that failed or didn't run?

Answer:

- **Backfilling** refers to running a DAG for past execution dates (to "catch up" on missed intervals or to re-run historical dates).
- Airflow provides a CLI command and UI options to backfill. For example: airflow dags backfill <dag_id> -s <start_date> -e <end_date>. This will create DAG runs for each interval between start and end, and run them sequentially (by default) or with -I for independent mode.
- If the DAG's catchup=True (which it is by default unless explicitly False), and the start_date is far in the past, the Airflow scheduler will actually automatically create backfill runs (catch up runs) for all intervals up to the current date when the DAG is turned on. That's effectively automatic backfill.
- If you had failures for certain dates and want to re-run them, you can either:
  - Mark the failed task instances as **failed** in the UI and then clear them (Airflow will retry those specific runs).
  - Or use the backfill command for those execution dates specifically to re-run entire DAGs for those dates.
- Need to ensure any tasks are idempotent if re-run (e.g., not insert duplicates). Backfill essentially just creates runs with those execution_date values.
- Also in newer Airflow, they often encourage using airflow dags trigger with a data interval for manual runs if needed, but traditional backfill covers ranges.
- Keep in mind: If tasks depend on external data by execution date (like partition by date), backfill should produce the same outputs as if it ran on time.
- If using catchup, you often set catchup=False to avoid automatic backfill when not wanted. But to manually backfill, you'd temporarily allow those runs or trigger them manually.
- Summing: To backfill, you either rely on catchup or manually trigger runs for past dates.

What interviewers look for: Knowledge of Airflow's ability to run historical jobs. That you know missing a schedule doesn't mean data is lost – you can catch up.

Common pitfalls:

- Not acknowledging idempotence – e.g., if you rerun a DAG that writes to a table, ensure it overwrites or skip duplicates.
- Confusing backfill with just rerunning failed tasks. Backfill is usually whole DAG runs for past dates.

Follow-up: If your DAG depends on previous runs (like uses yesterday's output), how do you handle backfill without causing collisions or messing up current pipeline? (Possible answer: you might disable triggers for current data or isolate backfill runs by using different output paths; or use Airflow's "depends_on_past" and set it to False for backfill if needed.)

---

35. **(Medium)** What is a **sensor** in Airflow? Give an example of how you'd use one.

Answer:

- A **sensor** in Airflow is a special type of operator that **waits for a certain condition to be met** before moving forwardcloudfoundation.com. It typically pauses the DAG run until some external event or data is available.
- Sensors are often used to synchronize data dependencies between systems. Examples:
  - **FileSensor:** waits for a file to land in a filesystem or storage bucket.
  - **ExternalTaskSensor:** waits for a task (or DAG) in a different workflow to complete.
  - **HdfsSensor, S3KeySensor, etc.:** wait for data in HDFS or S3, respectively.
  - **TimeSensor:** wait until a certain time of day.
- Example usage: DAG B should only start after DAG A's output is ready (say A produces a dataset). In DAG B, you could put an ExternalTaskSensor at the start that waits for DAG A's run for that execution date to finish successfully before continuing.
- Another example: A pipeline stage might start only after a file arrives via FTP. A FileSensor can keep checking a directory for the file existence, and once it's there, the sensor passes and the downstream tasks start processing the file.
- Sensors can be configured with a poke_interval (how often to check) and a timeout (how long to wait before failing). They can run in "poke" mode or "reschedule" mode (reschedule mode frees the worker slot while waiting, which is often recommended to not tie up a worker process).
- Should use sensors judiciously; long-running sensors can occupy resources. The reschedule mode (available by setting mode='reschedule') is better because the sensor task sleeps scheduling itself, rather than holding a slot.

What interviewers look for: That you know sensors are about waiting on external conditions. They might specifically be expecting mention of ExternalTaskSensor (common to coordinate DAGs) or file sensors.

Common pitfalls:

- Not mentioning the potential issue of sensor tying up a worker slot (older Airflow with many sensors in poke mode could exhaust workers).
- Confusing sensor with triggers – sensor is a kind of operator, not a scheduling trigger (though it effectively acts as one inside a DAG).

- Using sensors for short waits vs recommending a better approach if the wait is very long (like maybe triggering downstream instead or sensor in reschedule mode).

Follow-up: How can you ensure sensors don't exhaust all worker slots if you have many waiting? (Answer: Use mode='reschedule' so they don't occupy a worker while sleeping. Or increase worker slots but reschedule mode is primary fix.)

---

36. **(Medium)** What does it mean for a task (or DAG) to be **idempotent** and why is that important in Airflow?

Answer:

- An **idempotent** task is one that can be executed multiple times and produce the same result (or have no additional effect after the first execution)astronomer.io. In other words, if you run it again, it doesn't screw up data or double count – it either does nothing new or cleanly overwrites/upserts.
- In Airflow (or any workflow system), idempotence is important because tasks can be retried upon failure, or DAG runs can be backfilled/re-run. If tasks are not idempotent, a retry might, for example, duplicate data insertion or cause conflicts.
- For example, a task that loads data into a database should ideally be designed such that if it runs twice, it doesn't load duplicate rows. This might be achieved by deleting existing data for that period before insert, or using upsert logic, or writing to a partition keyed by execution date (so second run overwrites the same partition).
- Another aspect: If a DAG partially fails and you clear tasks to rerun them, idempotent tasks ensure the outcome is correct and not cumulative from previous attempt's partial work.
- Idempotence is also a best practice because Airflow by nature might run tasks out of sequence in some recovery scenarios or might have partial runs – if tasks are idempotent, you can safely "reset and rerun".
- Non-idempotent example: A task that sends an email – if retried, user gets duplicate emails (harmless maybe, but example of side effect). Or a task that appends to a file; a retry would append again. To make that idempotent, maybe the task writes to a temp file and renames (so either success or no partial effect), or checks if already done.
- The concept appears in Astronomer docs and Airflow best practices: "make tasks idempotent to handle retries and backfills gracefully"astronomer.io.

What interviewers look for: Understanding and emphasis on safe re-runs. If they've asked this, they want to see you care about reliability and data integrity.

Common pitfalls:

- Saying “idempotent means it runs only once” – not exactly, means multiple runs same effect.
- Not connecting it to Airflow specifics like retries/backfill.
- It's good to maybe mention some technique like using execution_date in output paths or database keys, or using transactions, to achieve idempotence.

Follow-up: How would you design an ETL that pulls yesterday’s records from a source and inserts into a target to be idempotent? (Possible answer: Use a delete-then-insert for that date in target, or track a watermark and ensure each record only loaded once by unique key. Or wrap in transaction and rollback on failure so partial doesn’t persist.)

---

37. **(Hard)** Airflow task failed due to a transient issue (e.g., temporary network glitch). How do you ensure the pipeline still succeeds without manual intervention?

Answer:

- Use Airflow’s built-in **retry mechanism**: Each task can be configured with retries (number of retry attempts) and retry_delay (interval between retries). For transient errors, usually a couple of automatic retries with some delay can resolve the issue without manual intervention.
- Ensure the task is **idempotent** (again) so that if it partially executed before failing, running it again won’t cause duplication or other side effects.
- Optionally implement **alerting** on failure but since question asks without manual intervention, focus is on automation: so primarily automatic retries.
- If the issue might persist for a bit, using an exponential backoff (Airflow allows customizing retry delay or use jitter) could be useful to not hammer the failing system.
- For certain external systems that are flaky, one might even implement logic within the task to catch exceptions and retry some sub-steps. But generally Airflow’s retry at the task level suffices.
- Also consider setting the max_active_runs or concurrency to allow other runs to proceed if one is retrying slowly (so pipeline doesn’t completely stall).
- If even after retries it fails, you could design the DAG such that downstream tasks have conditions (like using TriggerRule.ALL_DONE for an alerting task so main flow fails but an alternate path maybe triggers a compensation or something).
- But typically: number one answer is **retries**. E.g., PythonOperator(..., retries=3, retry_delay=timedelta(minutes=5)).
- Another angle: Use **sensors** to wait out transient issues (like if resource is unavailable, sensor could wait rather than immediate fail). But that might or might not apply.

What interviewers look for: They likely want "set retries in Airflow, so it auto-retries the task"[astronomer.ioreddit.com](). Also verifying you think of non-manual recovery.

Common pitfalls:

- Not mentioning retries at all (just saying monitor and rerun manually – they want automation).
- Overcomplicating: Like suggesting an XCom to signal re-run – no, Airflow already has retry feature.

Follow-up: If a task has to be retried but its failure caused partial data load, how would you guard against reprocessing partial data on retry? (We touched: either idempotent design or cleanup at start of task, e.g., delete partial output if exists.)

---

38. (Hard) Dagster and Prefect are emerging orchestration tools. Name one difference in approach compared to Airflow.

Answer:

- **Code paradigm vs config**: Dagster and Prefect emphasize a more code-centric, modern API approach. For example, Dagster uses software-defined assets and type-aware pipelines, whereas Airflow historically had more static DAG config (though it's Python code too, but more config-like).
- **Orchestration vs Dataflow:** Dagster tries to treat data assets as first-class, tracking lineage between data sets. Airflow is task-focused (doesn't inherently track what data was produced, just tasks).
- **Execution model**: Prefect (v1) didn't use a scheduler like Airflow's cron; it often uses an on-demand or event-driven execution model (Prefect v2 uses Python for orchestration with an API). Airflow uses a central scheduler that triggers jobs at intervals.
- **UI and Monitoring:** Prefect and Dagster come with interactive Pythonic APIs and nice UIs with different metaphors (Prefect calls flows/tasks, Dagster has solids/op definitions). Airflow's UI is more DAG-run centric and a bit older styling (though Airflow has improved).
- **Dynamic/Conditional flows**: Prefect allows easier dynamic control flow (if/else logic in flows) whereas Airflow historically discouraged dynamic task generation at runtime (Airflow 2 allows task mapping now though).
- **Setup**: Airflow requires a bit of infra setup (database, scheduler, etc.), while Prefect Cloud or Dagster Cloud provide more managed experiences, and even locally they can be simpler to start (less components to configure).
- Example difference: Dagster's concept of "ops" with input/output and a type system vs Airflow's simple tasks that just run a script. Dagster encourages functional composition and testing of pipeline logic, whereas Airflow tasks are more black-box.

- They might just expect one concrete difference: e.g., “Dagster treats data assets as first-class with automatic lineage tracking, whereas Airflow you have to manage that yourself” or “Prefect’s model allows for dynamic task mapping without plugins, where Airflow only recently added task mapping in 2.0 and still not as flexible for conditional flows”.

What interviewers look for: Awareness of the ecosystem beyond Airflow, and able to articulate at least one advantage or design difference of new tools. It shows you keep up with trends.

Common pitfalls:

- Not knowing anything about them (try to at least show some awareness).
- Saying one is better without reason. Instead, mention a design philosophy difference or feature.

Follow-up: Do these new orchestrators solve the idempotency or dependency issues inherently? (Expect: No, you still need idempotent tasks and manage failures, but some might have different retry or caching mechanisms etc.)

Sources (Orchestration):

- **Astronomer/Airflow Docs** – Best practices (idempotent DAG, atomic tasks, etc.)[astronomer.io](astronomer.io)
- **Airflow Official** – about sensors[cloudfoundation.com](cloudfoundation.com), schedule interval, retries in default_args etc. (like code examples in Airflow docs)
- **Medium (Ronit Malhotra)** – mentions idempotency and sensors in Q&A form[medium.commedium.com](medium.commedium.com)
- **Dagster vs Airflow blog** (any high-level blog for differences) – not explicitly scraped, but known comparisons (like data assets vs tasks concept)
- **Prefect docs** – possibly differences in architecture (no central scheduler, uses a server, flows)
