**Problem 5 — Airflow orchestration & DAG design (Production-ready, interview Q&A + code)**

Nice — this is the orchestration piece that ties your Bronze→Silver→Gold pipeline together.  
Below: short problem restatement, a baseline DAG (PyPI-style Airflow 2+), extensions for Databricks/Delta use, best practices, follow-up interviewer questions with **expected answers**, and a one-page cheat-sheet you can memorize.

**Interview Q&A (short questions + expected deep answers)**

**Q1 — *How do you prevent overlapping DAG runs from corrupting a Delta table?***

**Expected answer:**  
Set max\_active\_runs=1 on the DAG and/or use a task-level lock (Semaphore via Airflow Pool). Ensure the transformation notebooks are idempotent (MERGE with updated\_at checks) so retries/overlap are safe. Use optimistic concurrency of Delta; for critical sections, serialize writes using a control table / lease row.

**Q2 — *How would you implement retries and avoid duplicate downstream side effects (e.g., duplicate audit rows)?***

**Expected answer:**  
Make downstream operations idempotent (MERGE upserts, deterministic audit\_key). For non-idempotent side effects, use transactional patterns: write status markers first, perform operation, then write completion marker; if side-effect fails after commit, use reconciliation job. Prefer deriving audit from Delta CDF after successful commit.

**Q3 — *How do you pass a dynamic list of files discovered to a downstream Databricks job?***

**Expected answer:**  
Write the list of discovered file paths into a control/staging table or store as a small JSON in blob store and pass the path/manifest as a Databricks job parameter. Avoid big XComs. The Databricks notebook reads the manifest and processes files deterministically.

**Q4 — *What if ingestion/merge takes longer and misses downstream SLAs?***

**Expected answer:**  
Use alerting for SLA misses and auto-escalation. For long-running workloads, consider splitting pipeline into smaller partitions (per date/partition) and parallelizing non-conflicting parts — or move heavy merges to off-peak windows. Also tune cluster size and use autoscaling.

**Q5 — *How to handle downstream dependent tasks when a task fails and is retried?***

**Expected answer:**  
Use proper trigger\_rule defaults (e.g., downstream should wait for success). If downstream tasks can be run independently (idempotent), allow them to proceed. Record run states in a control table and let downstream tasks check run\_id status to decide actions (skip/continue).

**Q6 — *How to support ad-hoc runs and backfills safely?***

**Expected answer:**  
Provide DAG run parameters (run\_date, backfill=true) and ensure notebooks take these as input to process specific date ranges. For backfills, disable automatic downstream notifications and use a separate backfill orchestration with smaller partitions to avoid interfering with production.

**Q7 — *Explain how you’d do end-to-end testing for this pipeline in CI.***

**Expected answer:**

* Unit test Python transformations.
* Integration test notebooks on a small ephemeral cluster using representative sample data (use pytest + Databricks test harness).
* Use an isolated test environment & synthetic S3/ADLS.
* Run the DAG via airflow test or CI runner to validate orchestration logic. Validate outcomes (Delta table rows, audit entries).

**One-page cheat sheet (memorize these lines)**

* **DAG basics:** max\_active\_runs=1, catchup=False, retries=2-3, sensible timeouts.
* **Sensors:** prefer event/manifest-driven or reschedule-mode sensors (no tight loops).
* **Idempotency:** always design tasks (notebooks) to be idempotent — MERGE + audit\_key.
* **Atomicity:** rely on Delta transactions; use CDF to generate audit post-commit.
* **Concurrency:** use pools / max\_active\_runs / serialized control table for critical writes.
* **Backfills:** support run\_date params, separate backfill orchestration.
* **Secrets & configs:** Airflow Connections, Variables, Secret Backends.
* **Monitoring:** emit row counts, errors, and expose metrics; configure SLA emails.
* **Manifest pattern:** discover files → write manifest → pass manifest path to compute job.
* **Avoid large XComs** — use small pointers only.

**Problem (brief)**

Design an Airflow DAG to run the pipeline:

1. Ingest files → Bronze (Auto Loader / batch).
2. Run incremental dedupe & MERGE → Silver.
3. Transform/aggregate → Gold.  
   Requirements: idempotency, retries, observability, backfill support, SLA alerts, and safe concurrency (no overlapping runs that conflict).

**Baseline DAG (Airflow 2.x — Python)**

# dags/pipeline\_prod.py

from datetime import datetime, timedelta

from airflow import DAG

from airflow.operators.python import PythonOperator

from airflow.providers.databricks.operators.databricks import DatabricksSubmitRunOperator

from airflow.operators.empty import EmptyOperator

from airflow.sensors.filesystem import FileSensor

default\_args = {

"owner": "you",

"depends\_on\_past": False,

"retries": 2,

"retry\_delay": timedelta(minutes=5),

"email\_on\_failure": True,

"email": ["oncall@example.com"],

}

with DAG(

dag\_id="bronze\_silver\_gold\_pipeline",

start\_date=datetime(2025, 1, 1),

schedule\_interval="0 2 \* \* \*", # daily at 02:00

default\_args=default\_args,

catchup=False,

max\_active\_runs=1, # avoid overlapping runs

tags=["data-engineering", "prod"],

) as dag:

start = EmptyOperator(task\_id="start")

# Optional: wait for file landing (safety)

wait\_for\_files = FileSensor(

task\_id="wait\_for\_files",

fs\_conn\_id="my\_raw\_fs", # configure connection

filepath="/mnt/raw/vendor/\_READY", # marker file or folder

poke\_interval=300,

timeout=60 \* 60 \* 6

)

# Submit Databricks job to ingest into Bronze (Auto Loader or batch)

databricks\_ingest = DatabricksSubmitRunOperator(

task\_id="databricks\_ingest\_bronze",

databricks\_conn\_id="databricks\_default",

json={

"existing\_cluster\_id": "{{ var.value.db\_cluster\_id }}",

"notebook\_task": {"notebook\_path": "/Repos/ingest/bronze\_ingest"},

"timeout\_seconds": 3600

},

)

# Submit Databricks job for dedupe + MERGE to Silver

databricks\_merge = DatabricksSubmitRunOperator(

task\_id="databricks\_merge\_silver",

databricks\_conn\_id="databricks\_default",

json={

"existing\_cluster\_id": "{{ var.value.db\_cluster\_id }}",

"notebook\_task": {"notebook\_path": "/Repos/transform/merge\_silver"},

"timeout\_seconds": 3600

},

)

# Transform to Gold

databricks\_transform = DatabricksSubmitRunOperator(

task\_id="databricks\_transform\_gold",

databricks\_conn\_id="databricks\_default",

json={

"existing\_cluster\_id": "{{ var.value.db\_cluster\_id }}",

"notebook\_task": {"notebook\_path": "/Repos/transform/gold\_aggregates"},

"timeout\_seconds": 3600

},

)

end = EmptyOperator(task\_id="end", trigger\_rule="all\_done")

# DAG ordering

start >> wait\_for\_files >> databricks\_ingest >> databricks\_merge >> databricks\_transform >> end

**Notes**

* max\_active\_runs=1 plus depends\_on\_past=False prevents overlapping runs while allowing retries.
* Using Databricks operators keeps heavy compute outside Airflow. You can also use SparkSubmitOperator or custom PythonOperator for local runs.
* Use Airflow Variables and Connections for secrets and cluster IDs (no hard-coded creds).

**Production best practices & patterns**

* **Idempotency**: Ensure downstream tasks (Databricks notebooks) are idempotent (MERGE + audit keys). Airflow job retries should be safe.
* **Atomicity**: Do state-changing updates in Delta with atomic MERGE. Generate audit after successful merge (or use Delta CDF).
* **Task Timeouts & Retries**: sensible retries (2–3) with exponential backoff; task-level timeouts to avoid zombie jobs.
* **Max active runs / Pools**: limit concurrency by setting max\_active\_runs and use pools for resource control.
* **Sensors**: prefer lightweight sensors (reschedule mode) or use event-driven triggers (object created notifications) instead of tight poking.
* **Backfill & Catchup**: set catchup=False for daily scheduled jobs and provide a separate backfill mode that takes file date ranges.
* **SLAs & Alerts**: set sla on tasks or use monitoring hooks to notify on missed runtimes.
* **XCom usage**: keep small metadata (file lists, run IDs) in XCom; avoid large payloads.
* **Task idempotency markers**: write per-run marker rows to a control table (run\_id, file\_list, status) to help reconciliation.
* **Observability**: emit metrics (rows processed, fail counts) to Prometheus/Datadog; log structured JSON from notebook steps.
* **Secrets**: use secrets backend (Airflow Secrets, Databricks secret scopes) — never inline credentials.
* **Testing**: unit test DAG logic (pytest + Airflow test fixtures) and integration tests that run notebooks on ephemeral clusters.

**Scenario:**  
You must orchestrate the full pipeline:

* Ingest raw → Bronze (Problem 3)
* Incremental processing → Silver (Problem 4)
* CDC upsert + audit → Gold (Problem 1)
* Heavy joins with skew handling (Problem 2)

The pipeline should be **automated, monitored, and recoverable**.

**Baseline approach (expected in interviews):**

* Define a DAG with tasks for each stage.
* Add retries, SLA alerts, task dependencies.
* Store configs (paths, tables) in Airflow Variables.
* Example DAG snippet:

from airflow import DAG

from airflow.providers.databricks.operators.databricks import DatabricksRunNowOperator

from datetime import datetime, timedelta

default\_args = {

"owner": "data\_eng",

"depends\_on\_past": False,

"email\_on\_failure": True,

"email": ["alerts@company.com"],

"retries": 2,

"retry\_delay": timedelta(minutes=5)

}

with DAG(

dag\_id="customer\_pipeline",

start\_date=datetime(2025, 1, 1),

schedule\_interval="@daily",

default\_args=default\_args,

catchup=False

) as dag:

bronze\_task = DatabricksRunNowOperator(

task\_id="ingest\_bronze",

databricks\_conn\_id="databricks\_default",

job\_id="123-bronze-job"

)

silver\_task = DatabricksRunNowOperator(

task\_id="process\_silver",

databricks\_conn\_id="databricks\_default",

job\_id="456-silver-job"

)

gold\_task = DatabricksRunNowOperator(

task\_id="merge\_gold",

databricks\_conn\_id="databricks\_default",

job\_id="789-gold-job"

)

bronze\_task >> silver\_task >> gold\_task

**Advanced considerations:**

* **Dynamic DAGs:** auto-generate tasks for each partition/date.
* **Backfills:** support reprocessing past dates with catchup=True.
* **SLAs & monitoring:** use SLAs, sensors (file sensors, external task sensors).
* **Idempotency:** jobs should be safe to re-run (use MERGE instead of overwrite).
* **Lineage tracking:** integrate with tools like OpenLineage, Databricks Unity Catalog.
* **CI/CD:** DAGs stored in Git → deployed via CI/CD pipeline.

**Follow-up questions:**

* How do you handle failures in middle of DAG?
  + Use Airflow retries, mark downstream tasks as failed, rerun only failed ones.
* How do you handle dependencies between pipelines?
  + Use ExternalTaskSensor or event-based triggering.
* How do you reprocess one partition/day?
  + Parametrize DAG (e.g., execution\_date) and re-run with backfill.
* How to ensure monitoring/alerting?
  + Airflow email alerts + Datadog/Prometheus integration.

✅ With Problem 5, your **story arc is now end-to-end**:

“We ingest raw data into Bronze with validation (Problem 3), process it incrementally into Silver (Problem 4), apply CDC merges with audit into Gold (Problem 1), handle skewed joins for transformations (Problem 2), and orchestrate the entire pipeline with Airflow DAGs (Problem 5) including monitoring, retries, and backfills.”

This is the **“hero story”** that interviewers love because it shows you can design **real production-ready systems**.

If you want, next I can:

* Generate a **sample Airflow DAG** that implements a manifest pattern (File discovery → write manifest to Bronze manifest table → Databricks job reads manifest), or
* Roleplay an interview with 8–10 rapid Qs on Airflow where you answer and I give feedback.