**Assignment-2**

**Q1) What is the role of DAGs in monitoring and auditing pipelines?**

**Answer (Deep Dive):**  
In Airflow, a **DAG (Directed Acyclic Graph)** is the contract that turns monitoring/auditing from ad-hoc scripts into a **repeatable, observable control plane** for data quality.

**What DAGs give you for auditing**

* **Deterministic order & dependencies:** You explicitly wire pull\_raw\_data → run\_checks → summarize\_results → alert. Airflow guarantees checks don’t run before data arrives and results aren’t posted until checks finish.
* **Time-based governance:** Schedules (hourly/daily) ensure audits always happen—no manual triggers needed—and **backfills** let you re-run audits for past dates if you need historical assurance.
* **Automatic resiliency:** Retries with exponential backoff, SLAs, and failure callbacks mean audits recover from transient issues and **escalate** when problems persist.
* **Traceability & audit trail:** Task logs, XComs (for small results), and persisted artifacts (e.g., /tmp/audit\_result.json or a warehouse table) form a complete trail for compliance teams.
* **Branching & policy enforcement:** With BranchPythonOperator, you can **gate** downstream tasks on check outcomes (e.g., stop publishing dashboards if quality fails).

**Typical audit checks**

* **Freshness:** “Latest partition < 2 hours old?”
* **Volume thresholds:** “Did we receive ≥ N rows?”
* **Schema/contract:** “All required columns present? Types unchanged?”
* **Business rules:** “Refund rate < 3%?”, “Total debits = total credits?”

**Operator patterns**

* **SQL checks:** SQLCheckOperator, BigQueryCheckOperator, etc.
* **Great Expectations:** GreatExpectationsOperator to run expectations suites.
* **Custom checks:** PythonOperator to compute rates/anomalies, raise ValueError on failure.

**Example – stop the line if checks fail**

from airflow.operators.python import BranchPythonOperator

def route\_on\_quality(\*\*ctx):

passed = ctx['ti'].xcom\_pull(task\_ids='run\_quality')['passed']

return 'publish\_success' if passed else 'raise\_incident'

decide = BranchPythonOperator(task\_id='decide', python\_callable=route\_on\_quality)

run\_quality >> decide >> [publish\_success, raise\_incident]

**Q2) How can Airflow be adapted for event-driven workflows (reacting to external changes)?**

**Answer (Deep Dive):**  
Airflow is traditionally schedule-driven, but you can make it **data-/event-driven** using several mechanisms:

**1) Sensors (wait for an event)**

* **File/Data arrival:** S3KeySensor, FileSensor, GCSObjectExistenceSensor wait for a file like orders\_2025-08-19.csv.
* **Service availability:** HttpSensor/SqlSensor to wait for an API/DB condition.
* **External DAG completion:** ExternalTaskSensor waits for a producer DAG.

Use **deferrable sensors** (e.g., S3KeySensor(deferrable=True)) so they don’t block worker slots. The **Triggerer** handles the async wait efficiently.

**2) Dataset-aware scheduling (Airflow 2.4+)**  
Model data as **datasets**. Producer DAGs **publish** updates to a dataset; consumer DAGs **subscribe** and run **when the dataset changes**—no cron needed.

from airflow import DAG

from airflow.datasets import Dataset

orders\_ds = Dataset("s3://bronze/orders/")

# Producer: writes to orders\_ds

# Consumer: schedule=[orders\_ds] → runs when producer updates it

consumer\_dag = DAG(dag\_id="consume\_orders", schedule=[orders\_ds], start\_date=...)

**3) External triggers (API/webhooks)**

* Trigger DAG runs via the **REST API** or a webhook from upstream systems (e.g., your ingestion app calls Airflow when it finishes).
* Use TriggerDagRunOperator inside orchestrator DAGs to cascade triggers.

**4) Event buses + lightweight glue**

* A Lambda/Cloud Function subscribes to S3/Kafka events and **POSTs** to Airflow’s API to kick off the DAG immediately with the relevant run config.

**Example – deferrable S3 wait + validation**

from airflow.providers.amazon.aws.sensors.s3 import S3KeySensor

wait\_for\_orders = S3KeySensor(

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

bucket\_key="orders/{{ ds }}/orders.csv",

bucket\_name="my-bucket",

deferrable=True, # async, efficient

poke\_interval=60

)

**Q3) Compare Airflow with cron-based scripting, and also with Prefect/Luigi.**

**Answer (Deep Dive):**

**Airflow vs. Cron**

* **Dependencies & orchestration:** Cron knows time, not **order**. Airflow’s DAGs encode **A → B → C** and prevent load if transform failed.
* **Observability:** Airflow has a **UI**, task logs, run history, retries, SLAs, and alerts. Cron leaves you tailing syslog.
* **Backfills & catchup:** Re-run missed days cleanly in Airflow; cron requires custom loops.
* **Templating & params:** Jinja + macros ({{ ds }}, {{ ti.xcom\_pull }}) vs brittle shell scripting.
* **Scalability:** Executors (Celery, Kubernetes) let you **parallelize** safely; cron tends to overload single hosts.

**Airflow vs. Luigi**

* **Luigi:** Great for Pythonic dependency graphs, lightweight; historically weaker UI/scheduling ergonomics; fewer provider integrations.
* **Airflow:** Mature scheduler, rich provider ecosystem (AWS/GCP/Snowflake/Databricks/Slack/etc.), strong ops features.

**Airflow vs. Prefect**

* **Prefect:** Very Pythonic developer UX (imperative flows), **task mapping**/dynamicity is ergonomic, and **managed Prefect Cloud** makes setup easy.
* **Airflow:** Enterprise-grade, massive community/operators, native **dataset scheduling**, RBAC, and multiple executors.
* **Rule of thumb:** Prefer Airflow when you need **broad integrations, mature ops, and existing team familiarity**; choose Prefect for **fast developer velocity** and a managed cloud-first experience.

**Concrete scenario**

* **Simple nightly script:** Cron is fine.
* **Warehouse ELT with dependencies, QA gates, alerts, dashboard refresh:** Airflow.
* **Research team prototyping dynamic Python flows with quick cloud hosting:** Prefect.
* **Small Python-only batch with minimal ops overhead:** Luigi.

**Q4) How can Airflow be integrated with external logging/alerting systems?**

**Answer (Deep Dive):**  
Airflow has hooks, callbacks, and logging backends that plug into your org’s **observability stack**.

**Logging options**

* **Remote log storage:** Push task logs to S3/GCS/Azure Blob or **Elasticsearch** for centralized search.
* **Structured logging:** Emit JSON with correlation IDs (dag\_id, run\_id, task\_id) so tools like **Splunk/Datadog** can parse and alert.
* **Metrics:** Expose StatsD/Prometheus metrics for **Grafana** dashboards (task duration, failures, queue times).

**Alerting patterns**

* **Built-ins:** email\_on\_failure, email\_on\_retry, and email in default\_args.
* **Callbacks:** on\_failure\_callback, on\_retry\_callback, on\_success\_callback to call Slack, PagerDuty, Opsgenie, Microsoft Teams, etc.
* **Providers:** Use ready-made hooks like SlackWebhookHook, PagerdutyEventsHook.

**Examples**

*Slack on failure (callback)*

from airflow.providers.slack.hooks.slack\_webhook import SlackWebhookHook

def slack\_fail\_alert(context):

dag = context['dag'].dag\_id

task = context['task\_instance'].task\_id

run = context['dag\_run'].run\_id

msg = f":red\_circle: {dag}.{task} failed on {run}"

SlackWebhookHook(slack\_webhook\_conn\_id="slack\_alerts").send(text=msg)

default\_args = {

"owner": "data",

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

"email\_on\_failure": True,

"on\_failure\_callback": slack\_fail\_alert,

}

*SLA miss notifications*

dag = DAG(

dag\_id="dq\_audit",

sla\_miss\_callback=lambda \*\*ctx: print("SLA missed!", ctx),

# ...

)

*Ship audit artifacts for compliance*

* Write JSON/CSV results to a “**controls**” table or s3://audit-bucket/…
* Add a Bash/Python task that **POSTs** summaries to a governance API or indexes them into **Elasticsearch**.

**Best practices**

* **Don’t log secrets/PII.** Mask with Airflow’s connection/secret backends.
* **Make alerts actionable:** include dag\_id, task\_id, run\_id, link to UI, and a short “next step.”
* **Set severity tiers:** warnings for transient delays, pages for persistent data contract breaks.