The Top 21 Airflow Interview Questions and How to Answer Them
Part - 1

Created by - Shubham Wadekar

Airflow Interview Basics and Core Concepts

1. What is Apache Airflow? How is it most commonly used?

Answer: Apache Airflow is an open-source data orchestration tool that allows data practitioners to define data pipelines programmatically with the help of Python. Airflow is most commonly used by data engineering teams to integrate their data ecosystem and extract, transform, and load data.

Tell me more: Airflow is maintained under the Apache software license (hence, the prepended “Apache”).

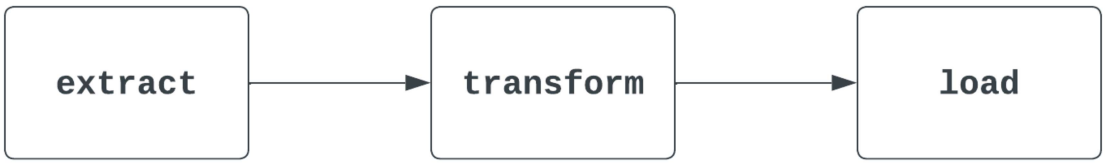
A data orchestration tool provides functionality to allow for multiple sources and services to be integrated into a single pipeline.

What sets Airflow apart as a data orchestration tool is its use of Python to define data pipelines, which provides a level of extensibility and control that other data orchestration tools fail to offer. Airflow boasts a number of built-in and provider-supported tools to integrate any team’s data stack, as well as the ability to design your own.

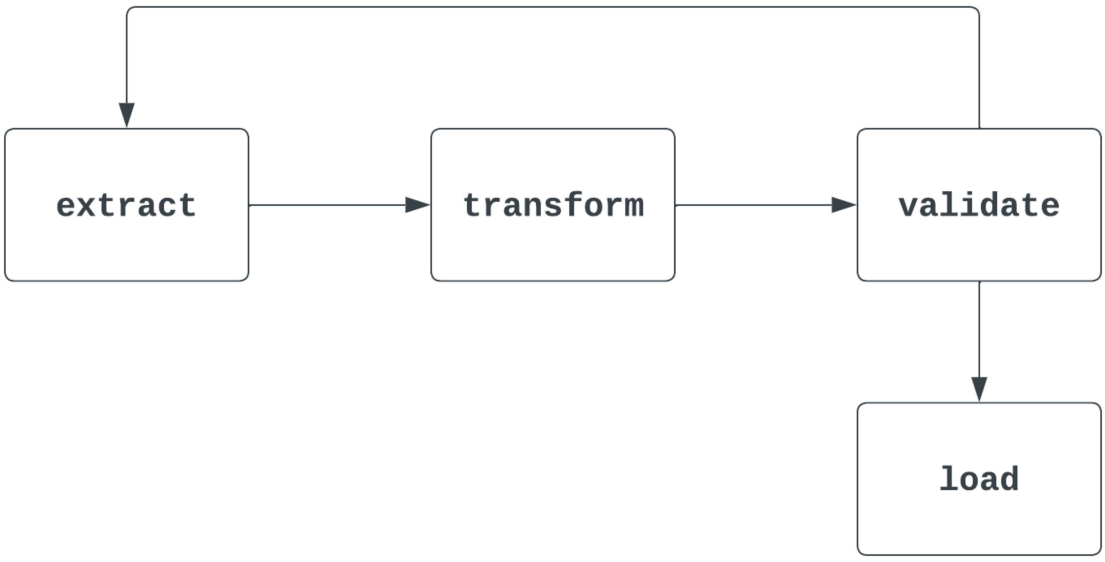
2. What is a DAG?

Answer: A DAG, or a directed-acyclic graph, is a collection of tasks and relationships between those tasks. A DAG has a clear start and end and does not have any “cycles” between these tasks. When using Airflow, the term “DAG” is commonly used, and can typically be thought of as a data pipeline.

Tell me more: This is a tricky question. When an interviewer asks this question, it’s important to address both the formal “mathy” definition of a DAG and how it’s used in Airflow. When thinking about DAGs, it helps to take a look at a visual. The first image below is, in fact, a DAG. It has a clear start and end and no cycles between tasks.



The second process shown below is NOT a DAG. While there is a clear start task, there is a cycle between the **extract** and **validate** tasks, which makes it unclear when the **load** task may be triggered.



3. What are the three parameters needed to define a DAG?

Answer: To define a DAG, an **ID**, **start date**, and **schedule interval** must be provided.

Tell me more: The ID uniquely identifies the DAG and is typically a short string, such as "sample_da." The start date is the date and time of the first interval at which a DAG will be triggered.

This is a timestamp, meaning that an exact year, month, day, hour, and minute are specified. The schedule interval is how frequently the DAG should be executed. This can be every week, every day, every hour, or something more custom.

In the example here, the DAG has been defined using a dag_id of "sample_dag". The datetime function from the datetime library is used to set a start_date of January 1, 2024, at 9:00 AM. This DAG will run daily (at 9:00 AM), as designated by the @daily scheduled interval. More custom schedule intervals can be set using cron expressions or the timedelta function from the datetime library.

```
with DAG(
     dag_id="sample_dag",
    start_date=datetime(year=2024, month=1, day=1, hour=9, minute=0),
    schedule="@daily",
) as dag:
    …
```

4. What is an Airflow task? Provide three examples of Airflow tasks.

Answer: Airflow tasks are the smallest unit of execution in the Airflow framework. A task typically encapsulates a single operation in a data pipeline (DAG). Tasks are the building blocks for DAGs, and tasks within a DAG have relationships between them that determine in what order they are executed. Three examples of tasks are:

- Extracting data from a source system, like an API or flat-file
- Transforming data to a desired model or format
- Loading data into a data storage tool, such as a database or data warehouse

In an ETL pipeline, the relationships would be:

- The “transform” task is downstream of the “extract” task, meaning that the “extract” logic executes first
- The “load” task is downstream of the “transform” task. Similar to above, the “load” task will run after the “transform” task

Tell me more: Tasks can be very generic or quite custom. Airflow provides two ways to define these tasks: traditional operators and the TaskFlow API (more on that later).

One of the benefits of open-source is contribution from the wider community, which is made upof not only individual contributors but also players such as AWS, Databricks, Snowflake, and a whole lot more.

Chances are, an Airflow operator has already been built for the task you’d like to define. If not, it’s easy to create your own. A few examples of Airflow operators are the SFTPToS3Operator, S3ToSnowflakeOperator, and DatabricksRunNowOperator.

5. What are the core components of Airflow's architecture?

Answer: There are four core components of Airflow's architecture: the scheduler, executor, metadata database, and the webserver.

Tell me more: The scheduler checks both the DAG directory every minute and monitors DAGs and tasks to identify any tasks that can be triggered. An executor is where tasks are run. Tasks can be executed locally (within the scheduler) or remotely (outside of the scheduler).

The executor is where the computational "work" that each task requires takes place. The metadata database contains all information about the DAGs and tasks related to the Airflow project you are running. This includes information such as historic execution details, connections, variables, and a host of other information.

The webserver is what allows for the Airflow UI to be rendered and interacted with when developing, interacting with, and maintaining DAGs.

Airflow DAGs Interview Questions

You've made it clear you know the basics of the Airflow framework and its architecture. Now, it's time to test your DAG-authoring knowledge.

6. What is the PythonOperator? What are the requirements to use this operator? What is an example of when you'd want to use the PythonOperator?

Answer: The PythonOperator is a function that allows for a Python function to be executed as an Airflow task. To use this operator, a Python function must be passed to the python_callable parameter. One example where you'd want to use a Python operator is when hitting an API to extract data.

Tell me more: The PythonOperator is one of the most powerful operators provided by Airflow. Not only does it allow for custom code to be executed within a DAG, but the results can be written to XComs to be used by downstream tasks.

By passing a dictionary to the op_kwargs parameter, keyword arguments can be passed to the Python callable, allowing for even more customization at run time. In addition to op_kwargs, there are a number of additional parameters that help to extend the functionality of the PythonOperator.

Below is a sample call of the PythonOperator. Typically, the Python function passed to python_callable is defined outside of the file containing the DAG definition. However, it was included here for verbosity.

```
def some_callable(name):
print("Hello ", name)

...

some_task = PythonOperator(
task_id="some_task",
    python_callable=some_callable,
    op_kwargs={"name": "Charles"}
)
```

7. If you have three tasks, and you'd like them to execute sequentially, how would you set the dependencies between each of them? What syntax would you use?

Answer: There are quite a few ways to do this. One of the most common is to use the >> bit-shift operator. Another is to use the .set_downstream() method to set a task downstream of another. The chain function is another useful tool for setting sequential dependencies between tasks. Here are three examples of doing this:

```
# task_1, task_2, task_3 instantiated above

# Using bit-shift operators
task_1 >> task_2 >> task_3

# Using .set_downstream()
task_1.set_downstream(task_2)
task_2.set_downstream(task_3)

# Using chain
chain(task_1, task_2, task_3)
```

Tell me more: Setting dependencies can be simple, while others may get quite complex! For sequential execution, it's common to use the bit-shift operators to make this more verbose. When using the TaskFlow API, setting dependencies between tasks can look a little different.

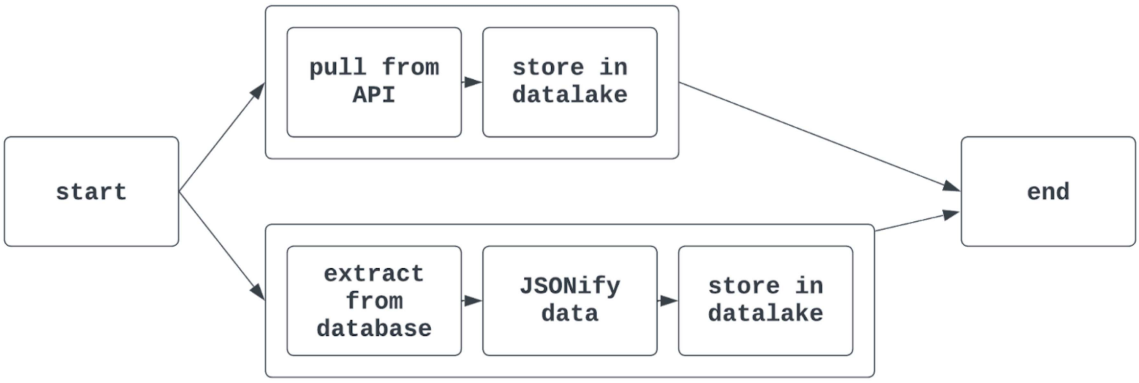
If there are two dependent tasks, this can be denoted by passing a function call to another function, rather than using the techniques mentioned above.

8. What are Task Groups? How are they used within DAGs?

Answer: Task groups are used to organize tasks together within a DAG. This makes it easier to denote similar tasks together in the Airflow UI. It may be useful to use task groups when extracting, transforming, and loading data that belong to different teams in the same DAG.

Task groups are also commonly used when doing things such as training multiple ML models or interacting with multiple (but similar) source systems in a single DAG.

When using task groups in Airflow, the resulting graph view may look something like this:



Tell me more: Using traditional Airflow syntax, the TaskGroup function is used to create task groups. Task groups can be explicitly or dynamically generated nbut must have unique IDs (similar to DAGs).

However, different task groups in the same DAG can have tasks with the same task_id. The task will then be uniquely identified by the combination of the task group ID and the task ID. When leveraging the TaskFlow API, the @task_group decorator can also be used to create a task group.

9. How can you dynamically generate multiple DAGs without having to copy and paste the same code? What are some things to keep in mind?

Answer: Dynamically generating DAGs is a handy technique to create multiple DAGs using a single “chunk” of code. Pulling data from multiple locations is one example of how dynamically creating DAGs is quite useful. If you need to extract, transform, and load data from three airports using the same logic, dynamically generating DAGs helps to streamline this process.

There are a number of ways to do this. One of the easiest is to use a list of metadata that can be looped over. Then, within the loop, a DAG can be instantiated. It’s important to remember that each DAG should have a unique DAG ID. The code to do this might look a little something like this:

```
airport_codes = ["atl", "lax", "jfk"]

for code in airport_codes:
with DAG(
dag_id=f"{code}_daily_etl"
        start_date=datetime(2024, 1, 1, 9, 0),
        schedule="@daily"
) as dag:
# Rest of the DAG definition
...
```

This code would spawn three DAGs, with DAG IDs atl_daily_etl, lax_daily_etl, and jfk_daily_etl. Downstream, the tasks could be parameterized by using the same airport code to ensure each DAG executed as expected.

Tell me more: Dynamically generating DAGs is a technique commonly used in an enterprise setting. When deciding between programmatically creating task groups in a single DAG, or dynamically generating DAGs, it’s important to think of the relationships between operations.

In our example above, if a single airport is causing an exception to be thrown, using task groups would cause the entire DAG to fail. But, if DAGs are instead dynamically generated, this single point of failure would not cause the other two DAGs to fail.

Looping over a Python iterable is not the only way to dynamically generate DAGs - defining a create_dag function, using variables/connections to spawn DAGs, or leveraging a JSON configuration file are common options used to achieve the same goal. Third-party tools such as gusty and dag-factory provide additional configuration-based approaches to dynamically generate DAGs.

The Top 21 Airflow Interview Questions and How to Answer Them

Part - 2

Created by - Shubham Wadekar

Advanced Airflow Interview Questions

10. Given a data_interval_start and data_interval_end, when is a DAG executed?

Answer: As the name suggests, data_interval_start and data_interval_end are the temporal boundaries for the DAG run. If a DAG is being backfilled, and the time the DAG is being executed is greater than the data_interval_end, then the DAG will immediately be queued to run.

However, for “normal” execution, **a DAG will not run until the time it is being executed is greater the **data_interval_end**.**

Tell me more: This is a difficult concept, especially with how DAG runs are labeled. Here’s a good way to think about it. You want to pull all data for March 17, 2024, from an API.

If the schedule interval is daily, the data_interval_start for this run is 2024-03-17, 00:00:00 UTC, and the data_interval_end is 2024-03-18, 00:00:00 UTC. It wouldn’t make sense to run this DAG at 2024-03-17, 00:00:00 UTC, as none of the data would be present for March 17. Instead, the DAG is executed at the data_interval_end of 2024-03-18, 00:00:00 UTC.

11. What is the catchup parameter, and how does it impact the execution of an Airflow DAG?

Answer: The catchup parameter is defined when a DAG is instantiated. catchup takes value True or False, defaulting to True when not specified. If True, **all DAG runs between the start date and the time the DAG's status was first set changed to active will be run.**

Say that a DAG's start date is set to January 1, 2024, with a schedule interval of daily and catchup=True. If the current date is April 15, 2024, when this DAG is first set to active, the DAG run with data_interval_start of January 1, 2024, will be executed, followed by the DAG run for January 2, 2024 (and so on).

This will continue until the DAG is "caught up" when it will then resume normal behavior. This is known as "backfilling". Backfilling might happen quite quickly. If your DAG run only takes a few minutes, a few months of historic DAG runs can be executed in just a few hours.

If False, no historic DAG runs will be executed, and the first run will begin at the end of the interval during which the DAG status was set to run.

Tell me more: Being able to backfill DAG runs without significant changes to code or manual effort is one of Airflow's most powerful features. Let's say you're working on an integration to pull all transactions from an API for the past year.

Once you've built your DAG, all you need to do is set your desired start date and catchup=True, and it's easy to retrieve this historical data.

If you don't want to backfill your DAG when first setting it to active, don't worry! There are a number of other ways to systematically trigger backfills. This can be done with the Airflow API and the Airflow (and Astro CLI).

12. What are XComs, and how are they typically used?

Answer: XComs (which stands for cross-communications) are a more nuanced feature of Airflow that allows for messages to be stored and retrieved between tasks.

XComs are stored in key-value pairs, and can be read and written in a number of ways. When using the PythonOperator, the .xcom_push() and .xcom_pull() methods can be used within the callable to “push” and “pull” data from XComs. XComs are used to store small amounts of data, such as file names or a boolean flag.

Tell me more: In addition to using the .xcom_push() and .xcom_pull(), there are a number of other ways to write and read data from XComs. When using the PythonOperator, passing True to the do_xcom_push parameters writes the value returned by the callable to XComs.

This is not just limited to the PythonOperator; any operator that returns a value can have that value written to XComs with the help of the do_xcom_push parameter. Behind the scenes, the TaskFlow API also uses XComs to share data between tasks.

13. Tell me about the TaskFlow API, and how it differs from using traditional operators.

Answer: The TaskFlow API offers a new way to write DAGs in a more intuitive, “Pythonic” manner. Rather than using traditional operators, Python functions are decorated with the @task decorator, and can infer dependencies between tasks without explicitly defining them.

A task written using the TaskFlow API may look something like this:

```
import random

...

@task
def get_temperature():
# Pull a temperature, return the value
    temperature = random.randint(0, 100)
    return temperature
…
```

With the TaskFlow API, it’s easy to share data between tasks. Rather than directly using XComs, the return value of one task (function) can be passed directly into another task as an argument. Throughout this process, XComs are still used behind the scenes, meaning that large amounts of data cannot be shared between tasks even when using the TaskFlow API.

Tell me more: The TaskFlow API is part of Airflow’s push to make DAG writing easier, helping the framework appeal to a wider audience of data scientists and analysts. While the TaskFlow API doesn’t meet the needs of Data Engineering teams looking to integrate a cloud data ecosystem, it’s especially useful (and intuitive) for basic ETL tasks.

14. What is idempotency? Why is this important to keep in mind when building Airflow DAGs?

Answer: Idempotency is a property of a process/operation that allows for that process to be performed multiple times without changing the initial result. More simply put, if you run a DAG once, or if you run it ten times, the results should be identical.

One common workflow where this is not the case is when inserting data into structured (SQL) databases. If data is inserted without a primary key enforcement and a DAG is run multiple times, this DAG will cause duplicates in the resulting table. Using patterns such as delete-insert or “upsert” helps to implement idempotence in data pipelines.

Tell me more: This one isn’t quite Airflow-specific, but it is essential to keep in mind when designing and building data pipelines. Luckily, Airflow provides several tools to help make implementing idempotency easy. However, most of this logic will need to be designed, developed and tested by the practitioner leveraging Airflow to implement their data pipeline.

Interview Questions on Managing and Monitoring Production Airflow Workflows

15. After writing a DAG, how can you test that DAG? Walk through the process from start to finish.

Answer: There are a few ways to test a DAG after it's been written. The most common is by executing a DAG to ensure that it runs successfully. This can be done by spinning up a local Airflow environment and using the Airflow UI to trigger the DAG.

Once the DAG has been triggered, it can be monitored to validate its performance (both success/failure of the DAG and individual tasks, as well as the time and resources it took to run).

In addition to manually testing the DAG via execution, DAGs can be unit-tested. Airflow provides tools via the CLI to execute tests, or a standard test runner can be used. These unit tests can be written against both DAG configuration and execution as well as against other components of an Airflow project, like callables and plugins.

Tell me more: Testing Airflow DAGs is one of the most important things a Data Engineer will do. If a DAG hasn't been tested, it's not ready to support production workflows.

Testing data pipelines is especially tricky; there are edge and corner cases that aren't typically found in other development scenarios. During a more technical interview (especially for a Lead/Senior Engineer), make sure to communicate the importance of testing a DAG end-to-end in tandem with writing unit tests and documenting the results of each.

16. How do you handle DAG failures? What do you do to triage and fix the issue?

Answer: No one likes DAG failures, but handling them with grace can set you apart as a Data Engineer. Luckily, Airflow offers a plethora of tools to capture, alert up, and remedy DAG failures. First, a DAG's failure is captured in the Airflow UI. The state of the DAG will change to "failed," and the grid view will show a red square/rectangle for this run. Then, the logs for this task can be manually parsed in the UI.

Typically, these logs will provide the exception that caused the failure and provide a Data Engineer with information to further triage.

Once the issue is identified, the DAG's underlying code/config can be updated, and the DAG can be re-run. This can be done by clearing the state of the DAG and setting it to "active."

If a DAG fails regularly but works when retried, it may be helpful to use Airflow's `retries` and `retry_delay` functionality. These two parameters can be used to retry a task upon failure a specified number of times after waiting for a certain period of time. This may be useful in scenarios like trying to pull a file from an SFTP site that may be late in landing.

Tell me more: For a DAG to fail, a specific task must fail. It's important to triage this task rather than the entire DAG. In addition to the functionality built into the UI, there are tons of other tools to monitor and manage DAG performance.

Callbacks offer Data Engineers basically unlimited customization when handling DAG successes and failures. With callbacks, a function of a DAG author's choosing can be executed when a DAG succeeds or fails using the `on_success_callback` and `on_failure_callback` parameters of an operator. This function can send a message to a tool like PagerDuty or write the result to a database to be later altered upon. This helps to improve visibility and jumpstart the triage process when a failure does occur.

17. To manage credentials for connecting to tools such as databases, APIs, and SFTP sites, what functionality does Airflow provide?

Answer: One of Airflow's most handy tools is "connections." Connections allow for a DAG author to store and access connection information (such as a host, username, password, etc.) without having to hardcode these values into code.

There are a few ways to store connections; the most common is using Airflow's UI. Once a connection has been created, it can be accessed directly in code using a "hook." However, most traditional operators requiring interaction with a source system have a conn_id (or very similarly named) field that takes a string and creates a connection to the desired source.

Airflow connections help to keep sensitive information secure and make storing and retrieving this information a breeze.

Tell me more: In addition to using the Airflow UI, the CLI can be used to store and retrieve connections. In an enterprise setting, it's more common to use a custom "secrets backend" to manage connection information. Airflow provides a number of support secrets backends to manage connections. A company using AWS can easily integrate Secrets Manager with Airflow to store and retrieve connection and sensitive information. If needed, connections can also be defined in a project's environment variables.

18. How would you deploy an Airflow project to be run in a production-grade environment?

Answer: Deploying an Airflow environment to be run in a production-grade environment can be difficult. Cloud tools such as Azure and AWS provide managed services to deploy and manage an Airflow deployment. However, these tools require a cloud account and may be somewhat expensive.

A common alternative is to use Kubernetes to deploy and run a production Airflow environment. This allows for complete control of the underlying resources but comes with the additional responsibility of managing that infrastructure.

Looking outside of cloud-native and homegrown Kubernetes deployments, Astronomer is the most popular managed service provider of Airflow in the space.

They provide a number of open-source tooling (CLI, SDK, loads of documentation) in addition to their “Astro” PaaS offering to make Airflow development and deployment as smooth as possible. With Astro, resource allocation, platform access control, and in-place Airflow upgrades are natively supported, putting the focus back on pipeline development.

Scenario-Based Airflow Interview Questions

19. Your team is currently supporting a legacy data pipeline that leverages homegrown tooling. You're tasked with migrating this pipeline to Airflow. How would you approach this?

Answer: This is a fun one! With a question like this, the world is at your fingertips. The most important thing is this - when walking through the process, **make sure to pick tools and processes in the legacy data pipeline that you are familiar with.** This shows your expertise and will make your answer to your question more informed.

This is the perfect opportunity to also show off your project management and leadership skills. Mention how you would structure the project, interact with stakeholders and other engineers, and document/communicate processes. This shows an emphasis on providing value and making your team's life easier.

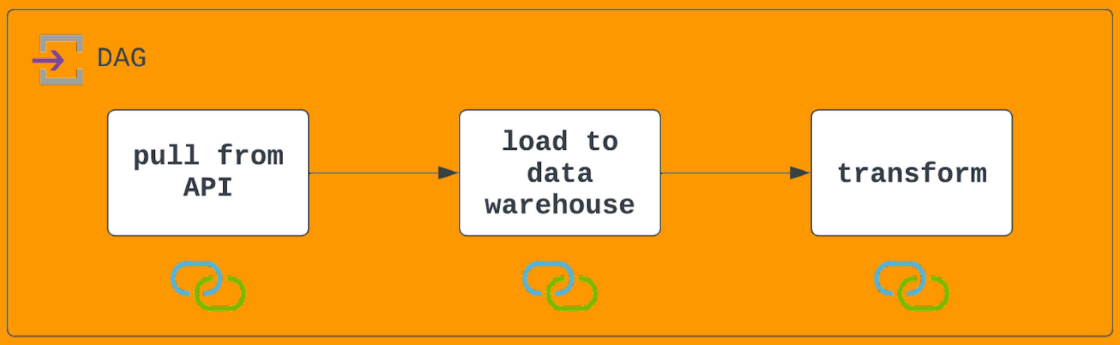
If a company has a certain tool in their stack (let's say, Google BigQuery), it might make sense to talk about how you can refactor this process to move off of something like Postgres and onto BigQuery. This helps to show awareness and knowledge not only of Airflow but also of other components of a company's infrastructure.

20. Design a DAG that pulls data from an API and persists the response in a flat-file, then loads the data into a cloud data warehouse before transforming it. What are some important things to keep in mind?

Answer: To design this DAG, you'll first need to hit the API to extract data. This can be done using the PythonOperator and a custom-built callable. Within that callable, you'll also want to persist the data to a cloud storage location, such as AWS S3.

Once this data has been persisted, you can leverage a prebuilt operator, such as the S3ToSnowflakeOperator, to load data from S3 into a Snowflake data warehouse. Finally, a DBT job can be executed to transform this data using the DbtCloudRunJobOperator.

You'll want to schedule this DAG to run at the desired interval and configure it to handle failures gracefully (but with visibility). Check out the diagram below!



It's important to keep in mind that Airflow is interacting with both a cloud storage file system, as well as a data warehouse. For this DAG to execute successfully, these resources will need to exist, and connections should be defined and used. These are denoted by the icons below each task in the architecture diagram above.

Tell me more: These are some of the most common Airflow interview questions, focusing more on high-level DAG design and implementation, rather than minute, technical details. With these questions, it's important to keep a few things in mind:

- Make sure to break the DAG down into clear, distinct tasks. In this case, the interviewer will be looking for a DAG that has three tasks.
- Mention specific tools that you might use to build this DAG, but don't go into too much detail. For example, if you'd like to use the DbtCloudRunJobOperator, mention this tool, but don't feel the need to elaborate much more than that.
- Remember to mention a few potential snags or things to keep in mind. This shows the interviewer that you have the awareness and experience to address edge and corner cases when building data pipelines.

21. Outside of traditional Data Engineering workflows, what are other ways that Apache Airflow is being used by data teams?

Answer: Airflow's extensibility and growing popularity in the data community has made it a go-to tool for more than just Data Engineers. Data Scientists and Machine Learning Engineers use Airflow to train (and re-train) their models, as well as perform a complete suite of MLOps. AI Engineers are even starting to use Airflow to manage and scale their generative AI models, with new integrations for tools such as OpenAI, OpenSearch, and Pinecone.

Tell me more: Thriving outside of traditional data engineering pipelines may not have been something that the initial creators of the Airflow envisioned. However, by leveraging Python and open-source philosophies, Airflow has grown to meet the needs of a rapidly evolving data/AI space. When programmatic tasks need to be scheduled and executed, Airflow just may be the best tool for the job!