AIRFLOW_IQ

1. **What is Airflow? -**
   Airflow is an open-source tool for programmatically authoring, scheduling, and monitoring data pipelines. 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.
2. **What issues does Airflow resolve?**
   Crons are an old technique of task scheduling. Scalable Cron requires external assistance to log, track, and manage tasks. The Airflow UI is used to track and monitor the workflow's execution. Creating and maintaining a relationship between tasks in cron is a challenge, whereas it is as simple as writing Python code in Airflow. Cron jobs are not reproducible until they are configured externally. Airflow maintains an audit trail of all tasks completed.
3. **Explain how workflow is designed in Airflow?**
   A directed acyclic graph (DAG) is used to design an Airflow workflow. That is to say, when creating a workflow, consider how it can be divided into tasks that can be completed independently. The tasks can then be combined into a graph to form a logical whole. The overall logic of your workflow is based on the shape of the graph. An Airflow DAG can have multiple branches, and you can choose which ones to follow and which to skip during workflow execution. Airflow Pipeline DAG Airflow could be completely stopped, and able to run workflows would then resume through restarting the last unfinished task. It is important to remember that airflow operators can be run more than once when designing airflow operators. Each task should be idempotent, or capable of being performed multiple times without causing unintended consequences.
4. **Explain Airflow Architecture and its components?**
   Airflow has six main components:
   The web server for serving and updating the Airflow user interface.
   The metadata database for storing all metadata (e.g., users, tasks) related to your Airflow instance.
   The scheduler for monitoring and scheduling your pipelines.
   The executor for defining how and on which system tasks are executed.
   The queue for holding tasks that are ready to be executed.
   The worker(s) for executing instructions defined in a task.
   Airflow runs DAGs in six different steps: The scheduler constantly scans the DAGs directory for new files. The default time is every 5 minutes.After the scheduler detects a new DAG, the DAG is processed and serialized into the metadata database.The scheduler scans for DAGs that are ready to run in the metadata database. The default time is every 5

seconds.Once a DAG is ready to run, its tasks are put into the executor's queue.Once a worker is available, it will retrieve a task to execute from the queue.The worker will then execute the task.

5. **What are the types of Executors in Airflow?**
   The executors are the components that actually execute the tasks, while the Scheduler orchestrates them. Airflow has different types of executors, including SequentialExecutor, LocalExecutor, CeleryExecutor and KubernetesExecutor. People generally choose the executor which is best for their use case. Types of Executor
   SequentialExecutor - Only one task is executed at a time by SequentialExecutor. The scheduler and the workers both use the same machine.
   LocalExecutor - LocalExecutor is the same as the Sequential Executor, except it can run multiple tasks at a time.
   CeleryExecutor - Celery is a Python framework for running distributed asynchronous tasks. As a result, CeleryExecutor has long been a part of Airflow, even before Kubernetes. CeleryExecutors has a fixed number of workers on standby to take on tasks when they become available.
   KubernetesExecutor- Each task is run by KubernetesExecutor in its own Kubernetes pod. It, unlike Celery, spins up worker pods on demand, allowing for the most efficient use of resources.
6. **What are the pros and cons of SequentialExecutor?**
   It's simple and straightforward to set up. It's a good way to test DAGs while they're being developed. Pros: It isn't scalable. It is not possible to perform many tasks at the same time. Unsuitable for use in production
7. **What are the pros and cons of LocalExecutor?**
   Able to perform multiple tasks. Can be used to run DAGs during development. The product isn't scalable. There is only one point of failure. Unsuitable for use in production.
8. **What are the pros and cons of CeleryExecutor?**
   It allows for scalability. Celery is responsible for managing the workers. Celery creates a new one in the case of a failure. Cons: Celery requires RabbitMQ/Redis for task queuing, which is redundant with what Airflow already supports. The setup is also complicated due to the above-mentioned dependencies.
9. **What are the pros and cons of KubernetesExecutor?**
   It combines the benefits of CeleryExecutor and LocalExecutor in terms of scalability and simplicity. Fine-grained control over task-allocation resources. At the task level, the amount of CPU/memory needed can be configured. Cons: Airflow is newer to Kubernetes, and the documentation is complicated.
10. **How to define a workflow in Airflow?**
    Python files are used to define workflows. DAG (Directed Acyclic Graph) The DAG Python class in Airflow allows you to generate a Directed Acyclic Graph, which is a representation of the workflow. from Airflow.models import DAG from airflow.utils.dates import days*ago*

args = { 'start_date': days_ago(0), }dag = DAG(dag_id='bash_operator_example', default_args=args, schedule_interval=' * * * *',)* You can use the start date to launch a task on a specific date. The schedule interval specifies how often each workflow is scheduled to run. '_ * * * *' indicates that the tasks must run every minute.

10. **How do you make the module available to airflow if you're using Docker Compose?**
    If we are using Docker Compose, then we will need to use a custom image with our own additional dependencies in order to make the module available to Airflow. Refer to the following Airflow Documentation for reasons why we need it and how to do it.
11. **How to schedule DAG in Airflow?**
    DAGs could be scheduled by passing a timedelta or a cron expression (or one of the @ presets), which works well enough for DAGs that need to run on a regular basis, but there are many more use cases that are presently difficult to express "natively" in Airflow, or that require some complicated workarounds.
12. **What is XComs In Airflow?**
    XCom (short for cross-communication) are messages that allow data to be sent between tasks. The key, value, timestamp, and task/DAG id are all defined.
13. **What is xcom_pull in XCom Airflow?**
    The xcom push and xcom pull methods on Task Instances are used to explicitly "push" and "pull" XComs to and from their storage. Whereas if do xcom push parameter is set to True (as it is by default), many operators and @task functions will auto-push their results into an XCom key named return value. If no key is supplied to xcom pull, it will use this key by default, allowing you to write code like this: Pulls the return_value XCOM from "pushing_task" value = task_instance.xcom_pull(task_ids='pushing_task')
14. **What is Jinja templates?**
    Jinja is a templating engine that is quick, expressive, and extendable. The template has special placeholders that allow you to write code that looks like Python syntax. After that, data is passed to the template in order to render the final document.
15. **How to use Airflow XComs in Jinja templates?**
    We can use XComs in Jinja templates as given below:
    SELECT * FROM {{ task_instance.xcom_pull(task_ids='foo', key='table_name') }}
16. **How does Apache Airflow act as a Solution?**
    Failures: This tool assists in retrying in case there is a failure.
    Monitoring: It helps in checking if the status has been succeeded or failed.
    Dependency: There are two different types of dependencies, such as:
    Data Dependencies that assist in upstreaming the data
    Execution Dependencies that assist in deploying all the new changes
    Scalability: It helps centralize the scheduler
    Deployment: It is useful in deploying changes with ease
    Processing Historical Data: It is effective in backfilling historical data

17. **How would you design an Airflow DAG to process a large dataset?**
    When designing an Airflow DAG to process a large dataset, there are several key considerations to keep in mind.
    First, the DAG should be designed to be modular and scalable. This means that the DAG should be broken down into smaller tasks that can be run in parallel, allowing for efficient processing of the data. Additionally, the DAG should be designed to be able to scale up or down depending on the size of the dataset.
    Second, the DAG should be designed to be fault-tolerant. This means that the DAG should be designed to handle errors gracefully and be able to recover from them. This can be done by using Airflow's retry and catchup features, as well as by using Airflow's XCom feature to pass data between tasks.
    Third, the DAG should be designed to be efficient. This means that the DAG should be designed to minimize the amount of data that needs to be processed and to minimize the amount of time it takes to process the data. This can be done by using Airflow's features such as branching, pooling, and scheduling.
    Finally, the DAG should be designed to be secure. This means that the DAG should be designed to protect the data from unauthorized access and to ensure that only authorized users can access the data. This can be done by using Airflow's authentication and authorization features.
    By following these guidelines, an Airflow DAG can be designed to efficiently and securely process a large dataset.
18. **What strategies have you used to optimize Airflow performance?**
    When optimizing Airflow performance, I typically focus on three main areas:
    Utilizing the right hardware: Airflow is a distributed system, so it's important to ensure that the hardware you're using is up to the task. This means having enough memory, CPU, and disk space to handle the workload. Additionally, I make sure to use the latest version of Airflow, as this can help improve performance.
    Optimizing the DAGs: I make sure to optimize the DAGs by using the best practices for Airflow. This includes using the right operators, setting the right concurrency levels, and using the right execution dates. Additionally, I make sure to use the right parameters for the tasks, such as setting the right retry limits and timeouts.
    Utilizing the right tools: I make sure to use the right tools to monitor and analyze the performance of Airflow. This includes using the Airflow UI, the Airflow CLI, and the Airflow Profiler. Additionally, I make sure to use the right metrics to measure performance, such as task duration, task throughput, and task latency.
    By focusing on these three areas, I am able to optimize Airflow performance and ensure that the system is running as efficiently as possible.
19. **How do you debug an Airflow DAG when it fails?**
    When debugging an Airflow DAG that has failed, the first step is to check the Airflow UI for the failed task. The UI will provide information about the task, such as the start and end

time, the duration of the task, and the error message. This information can help to identify the cause of the failure.
The next step is to check the Airflow logs for the failed task. The logs will provide more detailed information about the task, such as the exact command that was executed, the environment variables, and the stack trace. This information can help to pinpoint the exact cause of the failure.
The third step is to check the code for the failed task. This can help to identify any errors in the code that may have caused the failure.
Finally, if the cause of the failure is still not clear, it may be necessary to set up a debugging environment to step through the code and identify the exact cause of the failure. This can be done by setting up a local Airflow instance and running the DAG in debug mode. This will allow the developer to step through the code and identify the exact cause of the failure

20. **What is the difference between a Directed Acyclic Graph (DAG) and a workflow in Airflow?**
    A Directed Acyclic Graph (DAG) is a graph structure that consists of nodes and edges, where the edges represent the direction of the flow of data between the nodes. A DAG is acyclic, meaning that there are no loops or cycles in the graph. A DAG is used to represent the flow of data between tasks in a workflow.
    Airflow is a platform for programmatically authoring, scheduling, and monitoring workflows. Airflow uses DAGs to define workflows as a collection of tasks. A workflow in Airflow is a DAG that is composed of tasks that are organized in a way that reflects their relationships and dependencies. The tasks in a workflow are connected by edges that represent the flow of data between them.
    The main difference between a DAG and a workflow in Airflow is that a DAG is a graph structure that is used to represent the flow of data between tasks, while a workflow in Airflow is a DAG that is composed of tasks that are organized in a way that reflects their relationships and dependencies.
21. **How do you handle data dependencies in Airflow?**
    Data dependencies in Airflow are managed using the concept of Operators. Operators are the building blocks of an Airflow workflow and are used to define tasks that need to be executed. Each Operator is responsible for a specific task and can be configured to handle data dependencies.
    For example, the PythonOperator can be used to define a task that runs a Python script. This script can be configured to read data from a source, process it, and write the results to a destination. The PythonOperator can also be configured to wait for a certain set of data to be available before executing the task.
    The TriggerRule parameter of an Operator can also be used to define data dependencies. This parameter can be used to specify the conditions that must be met before the task is executed. For example, a task can be configured to run only when a certain file is present

in a certain directory.
Finally, the ExternalTaskSensor Operator can be used to wait for the completion of a task in another DAG before executing a task. This is useful when a task in one DAG depends on the completion of a task in another DAG.
In summary, Airflow provides a variety of Operators and parameters that can be used to manage data dependencies. By configuring these Operators and parameters correctly, data dependencies can be managed effectively in an Airflow workflow.

22. **What is the best way to handle errors in an Airflow DAG?**
The best way to handle errors in an Airflow DAG is to use Airflow's built-in error handling features. Airflow provides a number of ways to handle errors, including retries, email alerts, and logging.
Retries: Airflow allows you to set a maximum number of retries for a task, which will cause the task to be re-run if it fails. This can be useful for tasks that may fail due to transient errors, such as network issues.
Email Alerts: Airflow can be configured to send an email alert when a task fails. This can be useful for quickly identifying and addressing errors.
Logging: Airflow provides a logging system that can be used to track errors and other events. This can be useful for debugging and troubleshooting errors.
In addition to these built-in features, it is also important to ensure that your DAGs are well-structured and that tasks are properly configured. This will help to minimize the number of errors that occur in the first place.
23. **How do you ensure data integrity when using Airflow?**
Data integrity is an important consideration when using Airflow. To ensure data integrity when using Airflow, I would recommend the following best practices:
Use Airflow's built-in logging and monitoring features to track data changes and detect any anomalies. This will help you identify any potential issues with data integrity.
Use Airflow's built-in data validation features to ensure that data is accurate and complete. This will help you ensure that data is consistent and reliable.
Use Airflow's built-in scheduling and task management features to ensure that data is processed in a timely manner. This will help you ensure that data is up-to-date and accurate.
Use Airflow's built-in security features to protect data from unauthorized access. This will help you ensure that data is secure and protected.
Use Airflow's built-in data backup and recovery features to ensure that data is recoverable in the event of a system failure. This will help you ensure that data is not lost in the event of a system failure.
By following these best practices, you can ensure that data integrity is maintained when using Airflow.
24. **How do you handle data security when using Airflow?**
When using Airflow, data security is of utmost importance. To ensure data security, I take

the following steps: I use secure authentication methods such as OAuth2 and Kerberos to authenticate users and restrict access to the Airflow environment. I use encryption for data in transit and at rest. This includes encrypting data stored in databases, files, and other storage systems. I use secure protocols such as HTTPS and SFTP to transfer data between systems. I use role-based access control (RBAC) to restrict access to sensitive data and resources. I use logging and monitoring tools to detect and respond to security incidents. I use vulnerability scanning tools to identify and address potential security issues. I use secure coding practices to ensure that the code is secure and free from vulnerabilities. I use secure configuration management to ensure that the Airflow environment is configured securely. I use secure deployment processes to ensure that the Airflow environment is deployed securely. I use secure backup and disaster recovery processes to ensure that data is backed up and can be recovered in the event of a disaster.

25. **How do you ensure scalability when using Airflow?**
    When using Airflow, scalability can be achieved by following a few best practices.
    First, it is important to ensure that the Airflow DAGs are designed in a way that allows them to be easily scaled up or down. This can be done by using modular components that can be reused and scaled independently. Additionally, it is important to use Airflow's built-in features such as the ability to set up multiple workers and the ability to set up multiple DAGs. This allows for the DAGs to be scaled up or down as needed.
    Second, it is important to use Airflow's built-in features to ensure that the DAGs are running efficiently. This includes using Airflow's scheduling capabilities to ensure that tasks are running at the right time and using Airflow's logging capabilities to ensure that tasks are running correctly. Additionally, it is important to use Airflow's built-in features to ensure that tasks are running in the most efficient way possible. This includes using Airflow's task retry capabilities to ensure that tasks are retried if they fail and using Airflow's task concurrency capabilities to ensure that tasks are running in parallel.
    Finally, it is important to use Airflow's built-in features to ensure that the DAGs are running securely. This includes using Airflow's authentication and authorization capabilities to ensure that only authorized users can access the DAGs and using Airflow's encryption capabilities to ensure that the data is secure.
    By following these best practices, scalability can be achieved when using Airflow.
26. **What are Variables (Variable Class) in Apache Airflow?**
    Variables are a general way to store and retrieve content or settings as a simple key-value pair within Airflow. Variables in Airflow can be listed, created, updated, and deleted from the UI. Technically, Variables are Airflow's runtime configuration concept
27. **Why don't we use Variables instead of Airflow XComs, and how are they different?**
    An XCom is identified by a "key," "dag id," and the "task id" it had been called from. These work just like variables but are alive for a short time while the communication is being done within a DAG. In contrast, the variables are global and can be used throughout the

execution for configurations or value sharing.
There might be multiple instances when multiple tasks have multiple task dependencies; defining a variable for each instance and deleting them at quick successions would not be suitable for any process's time and space complexity.

28. **What are the states a Task can be in? Define an ideal task flow.**
Just like the state of a DAG (directed acyclic graph) being running is called a "DAG run", the tasks within that dag can have several tasks instances. they can be:

- none: the task is defined, but the dependencies are not met.
- scheduled: the task dependencies are met, has got assigned a scheduled interval, and are ready for a run.
- queued: the task is assigned to an executor, waiting to be picked up by a worker.
- running: the task is running on a worker.
- success: the task has finished running, and got no errors.
- shutdown: the task got interrupted externally to shut down while it was running.
- restarting: the task got interrupted externally to restart while it was running.
- failed: the task encountered an error.
- skipped: the task got skipped during a dag run due to branching (another topic for airflow interview, will cover branching some reads later)
- upstream_failed: An upstream task failed (the task on which this task had dependencies).
- up_for_retry: the task had failed but is ongoing retry attempts.
- up_for_reschedule: the task is waiting for its dependencies to be met (It is called the "Sensor" mode).
- deferred: the task has been postponed.
- removed: the task has been taken out from the DAG while it was running.
Ideally, the expected order of tasks should be : none -> scheduled -> queued -> running -> success.

30. **What is the role of Airflow Operators?**
There are three main types of operators:

- Action: Perform a specific action such as running code or a bash command.
- Transfer: Perform transfer operations that move data between two systems.
- Sensor: Wait for a specific condition to be met (e.g., waiting for a file to be present) before running the next task

31. **What is Branching in Directed Acyclic Graphs (DAGs)?**
Branching tells the DAG to run all dependent tasks, but you can choose which Task to move onto based on a condition. A task_id (or list of task_ids) is given to the "BranchPythonOperator", the task_ids are followed, and all other paths are skipped. It can

also be "None" to ignore all downstream tasks.
Even if tasks "branch_a" and "join" both are directly downstream to the branching operator, "join" will be executed for sure if "branch_a" will get executed, even if "join" is ruled out of the branching condition.

32. **What are ways to Control Airflow Workflow?**
By default, a DAG will only run an airflow task when all its Task dependencies are finished and successful. However, there are several ways to modify this:

- **Branching (BranchPythonOperator):** We can apply multiple branches or conditional limits to what path the flow should go after this task.
- **Latest Only (LatestOnlyOperator):** This task will only run if the date the DAG is running is on the current data. It will help in cases when you have a few tasks which you don't want to run while backfilling historical data.
- **Depends on Past (depends_on_past = true; arg):** Will only run if this task run succeeded in the previous DAG run.
- **Trigger rules ("trigger_rule"; arg):** By default, a DAG will only run an airflow task when all of its previous tasks have succeeded, but trigger rules can help us alter those conditions. Like "trigger_rule = always" to run it anyways, irrespective of if the previous tasks succeeded or not, OR "trigger_rule = all_success" to run it only when all of its previous jobs succeed.

33. **Explain the External task Sensor?**
An External task Sensor is used to sense the completion status of a DAG_A from DAG_B or vice-versa. If two tasks are in the same Airflow DAG we can simply add the line of dependencies between the two tasks. But Since these two are completely different DAGs, we cannot do this.
We can Define an ExternalTaskSensor in DAG_B if we want DAG_B to wait for the completion of DAG_A for a specific execution date.
There are six parameters to an External Task Sensor:

- external_dag_id: The DAG Id of the DAG, which contains the task which needs to be sensed.
- external_task_id: The Task Id of the task to be monitored. If set to default(None), the external task sensor waits for the entire DAG to complete.
- allowed_states: The task state at which it needs to be sensed. The default is "success."
- execution_delta: Time difference with the previous execution, which is needed to be sensed; the default is the same execution_date as the current DAG.
- execution_date_fn: It's a callback function that returns the desired execution dates to the query.

34. **What is TaskFlow API? and how is it helpful?**
    We have read about Airflow XComs (cross-communication) and how it helps to transfer data/messages between tasks and fulfill data dependencies. There are two basic commands of XComs which are "xcom*pull" used to pull a list of return values from one or multiple tasks and "xcom_push" used for pushing a value to the Airflow XComs.*
    Now, Imagine you have ten tasks, and all of them have 5-6 data dependencies on other tasks; writing an xcom_pull and x_push for passing values between tasks can get tedious. So TaskFlow API is an abstraction of the whole process of maintaining task relations and helps in making it easier to author DAGs without extra code, So you get a natural flow to define tasks and dependencies.
    _Note: TaskFlow API was introduced in the later version of Airflow, i.e., Airflow 2.0. So can be of minor concern in airflow interview questions.
35. **How are Connections used in Apache Airflow?**
    Apache Airflow is often used to pull and push data into other APIs or systems via hooks that are responsible for the connection. But since hooks are the intermediate part of the communication between the external system and our dag task, we can not use them to contain any personal information like authorization credentials, etc. Now let us assume the external system here is referred to as a MySQL database. We do need credentials to access MySQL, right? So where does the "Hook" get the credentials from?
    That's the role of "Connection" in Airflow.
    Airflow has a Connection concept for storing credentials that are used to talk to external systems. A Connection is a set of parameters - such as login username, password, and hostname - along with the system type it connects to and a unique id called the "conn_id".
    If the connections are stored in the metadata database, metadata database airflow supports the use of "Fernet" (an encryption technique) to encrypt the password and other sensitive data.
    Connections can be created in multiple ways:

    - Creating them directly from the airflow UI.
    - Using Environment Variables.
    - Using Airflow's REST API.
    - Setting it up in the airflows configuration file itself "airflow.cfg".
    - Using airflow CLI (Command Line Interface)

36. **Explain Dynamic DAGs.**
    Dynamic-directed acyclic graphs are nothing but a way to create multiple DAGs without defining each of them explicitly. This is one of the major qualities of apache airflow, which makes it a supreme "workflow orchestration tool".
    Let us say you have ten different tables to modify every day in your MySQL database, so you create ten DAG's to upload the respective data to their respective databases. Now

think if the table names change, would you go to each dag and change the table names? Or make new dags for them? Certainly not, because sometimes there can be hundreds of tables.

37. **How to control the parallelism or concurrency of tasks in Apache Airflow configuration?**
    Concurrency is the number of tasks allowed to run simultaneously. This can be set directly in the airflow configurations for all dags in the Airflow, or it can be set per DAG level. Below are a few ways to handle it:
    In config :

- parallelism: maximum number of tasks that can run concurrently per scheduler across all dags.
- max_active_tasks_per_dag: maximum number of tasks that can be scheduled at once.
- max_active_runs_per_dag: . the maximum number of running tasks at once.
  DAG level (as an argument to an Individual DAG) :
- concurrency: maximum number of tasks that can run concurrently in this dag.
- max_active_runs: maximum number of active runs for this DAG. The scheduler will not create new DAG runs once the limit hits.

38. **What are Macros in Airflow?**
    Macros are functions used as variables. In Airflow, you can access macros via the "macros" library. There are pre-defined macros in Airflow that can help in calculating the time difference between two dates or more! But we can also define macros by ourselves to be used by other macros as well, like we can use a macro to dynamically generate the file path for a file. Some of the examples of pre-defined and most-used macros are:

- Airflow.macros.datetime*diff_for_humans(dt, _since=None*): Returns difference between two datetimes, or one and now. *(Since = None refers to "now"*)**
- airflow.macros.ds*add(_ds*, number*of__days*) : Add or subtract n number of days from a YYYY-MM-DD(ds), will subtract if number_of_days is negative.

39. **List the types of Trigger rules.**

- all_success: the task gets triggered when all upstream tasks have succeeded.
- all_failed: the task gets triggered if all of its parent tasks have failed.
- all_done: the task gets triggered once all upstream tasks are done with their execution irrespective of their state, success, or failure.
- one_failed: the task gets triggered if any one of the upstream tasks gets failed.
- one_success: the task gets triggered if any one of the upstream tasks gets succeeds.
- none_failed: the task gets triggered if all upstream tasks have finished successfully or

been skipped.

- none_skipped: the task gets triggered if no upstream tasks are skipped, irrespective of if they succeeded or failed.

40. **What are SLAs?**
    SLA stands for Service Level Agreement; this is a time by which a task or a DAG should have succeeded. If an SLA is missed, an email alert is sent out as per the system configuration, and a note is made in the log. To view the SLA misses, we can access it in the web UI.
    It can be set at a task level using the "timedelta" object as an argument to the Operator, as sla = *timedelta(seconds=30)*.
41. **What is Data Lineage?**
    Many times, we may encounter an error while processing data. To determine the root cause of this error, we may need to track the path of the data transformation and find where the error occurred. If we have a complex data system then it would be challenging to investigate its root. Lineage allows us to track the origins of data, what happened to it, and how did it move over time, such as in S3, HDFS, MySQL or Hive, etc. It becomes very useful when we have multiple data tasks reading and writing into storage. We need to define the input and the output data sources for each task, and a graph is created in Apache Atlas, which depicts the relationships between various data sources.
42. **What if your Apache Airflow DAG failed for the last ten days, and now you want to backfill those last ten days' data, but you don't need to run all the tasks of the dag to backfill the data?**
    We can use the Latest Only (LatestOnlyOperator) for such a case. While defining a task, we can set the latest_only to True for those tasks, which we do not need to use for backfilling the previous ten days' data.
43. **What will happen if you set 'catchup=False' in the dag and 'latest_only = True' for some of the dag tasks?**
    Since in the dag definition, we have set catchup to False, the dag will only run for the current date, irrespective of whether latest_only is set to True or False in any one or all the tasks of the dag. 'catchup = False' will just ensure you do not need to set latest_only to True for all the tasks.
44. **How would you handle a task which has no dependencies on any other tasks?**
    We can set "trigger_rules = 'always'" in a task, which will make sure the task will run irrespective of if the previous tasks have succeeded or not.
45. **How can you use a set or a subset of parameters in some of the dags tasks without explicitly defining them in each task?**
    We can use the "params" argument. It is a dictionary of DAG-level parameters that are made accessible in jinja templates. These "params" can be used at the task level. We can pass "params" as a parameter to our dag as a dictionary of parameters such as {"param1":

"value1", "param2": "value2"}. And these can be used as "echo {{params.param1}}" in a bash operator.

46. **What Executor will you use to test multiple jobs at a low scale?**
    Local Executor is ideal for testing multiple jobs in parallel for performing tasks for a small-scale production environment. The Local Executor runs the tasks on the same node as the scheduler but on different processors. There are other executors as well who use this style while distributing the work. Like, Kubernetes Executor would also use Local Executor within each pod to run the task.
47. **If we want to exchange large amounts of data, what is the solution to the limitation of XComs?**
    Since Airflow is an orchestrator tool and not a data processing framework, if we want to process large gigabytes of data with Airflow, we use Spark (which is an open-source distributed system for large-scale data processing) along with the Airflow DAGs because of all the optimizations that It brings to the table.
48. **What would you do if you wanted to create multiple dags with similar functionalities but with different arguments?**
    We can use the concept of Dynamic DAGs generation. We can define a create_dag method which can take a fixed number of arguments, but the arguments will be dynamic. The dynamic arguments can be passed to the create_dag method through Variables, Connections, Config Files, or just passing a hard-coded value to the method.
49. **Is there any way to restrict the number of variables to be used in your directed acyclic graph, and why would we need to do that?**
    Airflow Variables are stored in the Metadata Database, so any call to a variable would mean a connection to the database. Since our DAG files are parsed every X seconds, using a large number of variables in our DAG might end up saturating the number of allowed connections to our database. To tackle that, we can just use a single Airflow variable as a JSON, as an Airflow variable can contain JSON values such as {"var1": "value1", "var2": "value2"}.
50. **How can you use a set or a subset of parameters in some of the dags tasks without explicitly defining them in each task?**
    We can use the "params" argument. It is a dictionary of DAG-level parameters that are made accessible in jinja templates. These "params" can be used at the task level. We can pass "params" as a parameter to our dag as a dictionary of parameters such as {"param1": "value1", "param2": "value2"}. And these can be used as "echo {{params.param1}}" in a bash operator.