ADF SCENARIO BASED Q&A
BY - SHUBHAM WADEKAR

Scenario 1
Your data pipeline fails intermittently due to network issues. You need to implement fault tolerance and automatic retries using Azure Data Factory.

Step 1: Configure Retry Policy

Retry Settings in Activities
- In Copy Data (or any) activity, go to Settings tab
- Set Retry count (e.g., 3)
- Set Retry interval (e.g., 30 seconds)
- Ensures ADF retries the activity if it fails due to transient issues

Step 2: Use Timeout and Timeout Policy

Activity Timeout
- In Settings, set timeout duration (e.g., 00:10:00 for 10 mins)
- Prevents activity from hanging indefinitely

Timeout Policy
- Use onFailure or onTimeout dependencies in control flow
- Redirect flow to alternate path or error handler if timeout occurs

Step 3: Add Error Handling with Control Flow

Use 'If Condition' or 'Switch'
- Route based on activity status (@activity().status == 'Failed')
- Implement conditional logic after failure

Use 'Execute Pipeline' with Dependents
- Chain retry logic via another lightweight retry wrapper pipeline

Step 4: Add Logging and Notifications

Log Failures
- Use stored procedures or Web activity to log failure info to Azure SQL, Blob, or Log Analytics

Send Alerts
- Use Web Activity to call Logic Apps or Azure Functions
- Send email, Teams alert, or push notification on failure

Step 5: Implement Idempotent Design

Ensure Re-runs Are Safe
- Use UPSERTs or deduplication logic in target tables
- Ensure re-processing the same data doesn't cause duplicates

Scenario 2
Your company needs to copy data from an on-premises SQL Server to an Azure SQL Database on a daily schedule using Azure Data Factory.

Step 1: Create Linked Services

On-Prem SQL Server Linked Service
• Go to Manage > Linked Services > New
• Select SQL Server as connector
• Set up Self-hosted Integration Runtime (SHIR) on a local machine
• Enter server, database, and authentication (Windows/SQL)
• Store credentials securely using Azure Key Vault

Azure SQL Database Linked Service
• Select Azure SQL Database as connector
• Enter server name, database name, and auth details
• Use managed identity or Key Vault for secure access

Step 2: Create Datasets

Source Dataset (On-Prem SQL Server)
• Create a dataset using SQL Server connector
• Link to the on-prem SQL linked service
• Select table or provide custom query
• Define or import schema

Sink Dataset (Azure SQL Database)
• Create a dataset using Azure SQL connector
• Link to the Azure SQL linked service
• Specify target table
• Ensure schema matches or map fields appropriately

Step 3: Create and Configure the Pipeline

Copy Activity
• Create a new pipeline and add Copy Data activity
• Set source to on-prem SQL dataset
• Set sink to Azure SQL dataset
• Use Mapping tab to align columns
• Add pre-copy scripts or type conversion if needed

Step 4: Schedule the Pipeline

Trigger Setup
• Go to Triggers > New
• Choose Schedule trigger
• Set recurrence to every 24 hours
• Attach the trigger to the pipeline and publish changes

Scenario 3
You need to transform data from a CSV file in Azure Blob Storage and load the processed output into an Azure SQL Database using Azure Data Factory.

Step 1: Create Linked Services

Azure Blob Storage Linked Service
• Go to Manage > Linked Services > New
• Select Azure Blob Storage as connector
• Use connection string or managed identity
• Store secrets in Azure Key Vault if needed

Azure SQL Database Linked Service
• Select Azure SQL Database as connector
• Enter server name, database name, and auth method
• Prefer managed identity or use Key Vault for credentials

Step 2: Create Datasets

Source Dataset (CSV in Blob Storage)
• Create a dataset using Azure Blob Storage
• Choose DelimitedText as format
• Point to the container and file path
• Define first row as header, delimiter (e.g., comma), and schema

Sink Dataset (Azure SQL Database)
• Create a dataset using Azure SQL connector
• Point to the destination table
• Ensure schema matches transformed structure

Step 3: Create and Configure the Pipeline

Data Flow Activity (for transformation)
• In Author > Data flows > New data flow
• Add Source: Select CSV dataset
• Add transformations as needed:

- Select, Derived Column, Filter, Aggregate, etc.
  • Add Sink: Set Azure SQL as destination
  • Use Mapping tab to map columns to target schema

Pipeline Setup
• Create a pipeline and add a Data Flow activity
• Link the data flow you created
• Set parameters if required for dynamic file paths or tables

Step 4: Schedule the Pipeline

Trigger Setup
• Go to Triggers > New
• Choose Schedule as trigger type
• Set recurrence as needed (e.g., daily)
• Attach the trigger to the pipeline and publish

Scenario 4
You need to copy data from multiple CSV files stored in Azure Data Lake Storage Gen2 (ADLS Gen2) into an Azure SQL Database using Azure Data Factory.

Step 1: Create Linked Services

Azure Data Lake Storage Gen2 Linked Service
- Go to Manage > Linked Services > New
- Select Azure Data Lake Storage Gen2 as connector
- Authenticate using managed identity or account key
- Use Azure Key Vault for secret management if needed

Azure SQL Database Linked Service
- Select Azure SQL Database as connector
- Enter server name, database name, and authentication details
- Prefer managed identity or secure via Key Vault

Step 2: Create Datasets

Source Dataset (ADLS Gen2 - CSV Files)
- Create a dataset using ADLS Gen2 and DelimitedText format
- Point to the folder containing CSV files, not a specific file
- Enable wildcard file path (e.g., *.csv)
- Set first row as header, define delimiter (e.g., comma), and schema

Sink Dataset (Azure SQL Database)
- Create a dataset using Azure SQL Database connector
- Point to the target table in the database
- Define schema to match source or use auto mapping

Step 3: Create and Configure the Pipeline

Copy Data Activity
- Create a pipeline and add Copy Data activity
- Set source to the ADLS Gen2 CSV dataset
- Enable wildcard file processing under source settings
- Set sink to the Azure SQL Database dataset
- Use Mapping tab to map columns from CSV to SQL schema
- Optionally, enable staging, batch size, or pre-copy scripts for performance

Step 4: Schedule the Pipeline

Trigger Setup
- Go to Triggers > New
- Choose Schedule as trigger type
- Set recurrence (e.g., daily or hourly)
- Attach trigger to the pipeline and publish

Scenario 5
You have multiple pipelines in Azure Data Factory and want to ensure one pipeline runs only after another pipeline completes successfully.

Step 1: Use 'Execute Pipeline' Activity

Chaining Pipelines Directly
• In the main (or parent) pipeline, drag and drop the Execute Pipeline activity
• Set the Invoked Pipeline property to the dependent (child) pipeline
• Use success dependency by default (green arrow) to ensure it only runs after successful completion
• Add additional logic/activities after this if needed

Step 2: Use Activity Dependency Conditions

Control Execution Flow
• Use success (green), failure (red), or completion (gray) dependencies between activities
• Right-click on activity > Add dependency > Select 'Success'
• Ensures the next activity only runs if the previous one is successful

Step 3: Trigger Pipeline B from Pipeline A

Logical Grouping in One Orchestrator Pipeline
• Create a master pipeline (Pipeline A)
• Use multiple Execute Pipeline activities
• Link them with success dependencies to form the desired execution order

Step 4: Trigger Chaining Using Event-Based Triggers (Optional)

If Pipelines Are Separate and Not in One Flow
• Use Event Grid or Web Activity in Pipeline A to trigger Pipeline B on success
• Or configure Logic Apps to listen for completion events from Pipeline A
• This is useful when pipelines are independently triggered but still need coordination

Scenario 6
You need to perform a complex transformation in Azure Data Factory that includes filtering, aggregation, and joining data from two different source systems.

Step 1: Create Linked Services

Linked Service for Source A & Source B
• Go to Manage > Linked Services > New
• Create linked services for both source systems (e.g., Azure SQL, Blob Storage, ADLS Gen2, etc.)
• Authenticate using managed identity or Key Vault-secured credentials

Linked Service for Sink (Target System)
• Create a linked service for the destination (e.g., Azure SQL Database, Synapse)
• Use secure authentication methods

Step 2: Create Datasets

Source Datasets (A & B)
• Create datasets for both source systems
• Define connection to tables, views, or files (CSV, Parquet, etc.)
• Import or define schema

Sink Dataset
• Create a dataset for the destination table
• Ensure it can accept the transformed data structure

Step 3: Build Data Flow for Transformation

Create Mapping Data Flow
• Go to Author > Data Flows > New Data Flow
• Add **Source A** and **Source B**
• Apply the following transformations:

- **Filter**: Add Filter transformation on one or both sources to remove unwanted rows
- **Join**: Add Join transformation (Inner, Left, etc.) to combine Source A and B on a common key
- **Aggregate**: Use Aggregate transformation to perform group by, sum, count, avg, etc. on joined data

Add Sink
• Add a Sink transformation
• Map output to the target table in the sink dataset
• Use Mapping tab to match fields

Step 4: Create and Configure Pipeline

Pipeline Setup
• Create a pipeline and add a **Data Flow activity**
• Link the mapping data flow you created
• Set any required parameters for source paths, filters, or dynamic table names

Step 5: Schedule or Trigger Pipeline

Trigger Setup
• Go to Triggers > New
• Choose a Schedule trigger or manual/debug run
• Set recurrence and attach it to the pipeline

Scenario 7
You need to implement incremental loading to move only new or changed data from an on-premises SQL Server to Azure SQL Database using Azure Data Factory (ADF).

Step 1: Create Linked Services

On-Prem SQL Server Linked Service
• Go to Manage > Linked Services > New
• Select SQL Server as connector
• Configure using Self-hosted Integration Runtime (SHIR)
• Provide server name, database, and authentication

Azure SQL Database Linked Service
• Select Azure SQL Database connector
• Provide target server, database, and authentication
• Secure using managed identity or Azure Key Vault

Step 2: Identify the Incremental Column

Choose a Watermark Column
• Use a reliable column such as LastModifiedDate or CreatedDate
• Ensure the column is indexed for performance
• Determine the logic to fetch new/changed records (e.g., WHERE LastModifiedDate > @LastRunTime)

Step 3: Create Parameters and Pipeline Variables

Parameterize the Watermark
• Add a pipeline parameter (e.g., LastRunTime)
• Use Lookup or Stored Procedure to retrieve the last successful value
• Store this value in Azure SQL table or Key Vault for reuse

Step 4: Create Datasets and Source Query

Source Dataset
• Use SQL query with dynamic WHERE clause:

```
SELECT * FROM SourceTable WHERE LastModifiedDate >
'@{pipeline().parameters.LastRunTime}'
```

Sink Dataset
• Point to the Azure SQL target table
• Choose Upsert or Insert based on use case

Step 5: Configure Copy Activity with Mapping

Copy Activity
• Use source query with watermark filter
• Use Mapping tab to map columns
• Optionally use Stored Procedure or Pre-copy script for cleanup/deduplication

Step 6: Update Watermark Post Load

Update LastRunTime
• Add Stored Procedure or Script activity at the end
• Write the new max value of LastModifiedDate back to tracking table

Step 7: Schedule the Pipeline

Trigger Setup
• Go to Triggers > New
• Choose Schedule trigger
• Set desired frequency (e.g., every hour/day)
• Attach the trigger to the pipeline

Scenario 8
You need to integrate data in multiple formats (CSV, JSON, Parquet) from Azure Data Lake Storage Gen2 into a single Azure SQL Database table using Azure Data Factory (ADF).

Step 1: Create Linked Services

Azure Data Lake Storage Gen2 Linked Service
• Go to Manage > Linked Services > New
• Select Azure Data Lake Storage Gen2 connector
• Authenticate using managed identity or Key Vault-secured credentials

Azure SQL Database Linked Service
• Select Azure SQL Database connector
• Provide server name, database, and authentication method
• Use managed identity or Key Vault for secrets

Step 2: Create Datasets for Each Format

CSV Dataset
• Use DelimitedText format
• Point to folder containing CSV files
• Set header, delimiter, and schema

JSON Dataset
- Use JSON format
- Point to JSON file location
- Define structure or use schema projection

Parquet Dataset
- Use Parquet format
- Point to folder with Parquet files
- Schema is auto-detected from the file

Sink Dataset (Azure SQL)
- Use Azure SQL Database format
- Point to the common destination table
- Schema must match the transformed input

Step 3: Create a Mapping Data Flow

Build Unified Transformation Logic
- Create a Mapping Data Flow
- Add three separate Source transformations (one for each format)
- Apply Select transformation to normalize columns to a common schema
- Use Union transformation to merge the three sources into one stream
- Add a Sink to write to the target Azure SQL table

Step 4: Configure the Pipeline

Pipeline Setup
- Create a pipeline and add a Data Flow activity
- Link to the mapping data flow you built
- Set any required parameters (e.g., file paths or formats) for dynamic use

Step 5: Schedule or Trigger the Pipeline

Trigger Setup
- Go to Triggers > New
- Choose Schedule trigger or use an event-based trigger (e.g., when new files arrive)
- Set frequency and attach it to the pipeline

Scenario 9
You need to configure a pipeline in Azure Data Factory (ADF) that dynamically selects different source and sink systems based on input parameters (e.g., file name, table name, environment, etc.).

Step 1: Define Pipeline Parameters

Parameter Creation
• Go to Author > Pipelines > New pipeline
• Click Parameters tab
• Create parameters like:

- SourceType, SinkType, SourcePath, SinkTable, etc.
  • Parameters will drive dynamic behavior

Step 2: Create Parameterized Datasets

Source Dataset (Generic)
• Create one dataset per type (e.g., Blob CSV, SQL, ADLS Parquet)
• Use dynamic expressions in file path/table name:

- Example: @dataset().path = @pipeline().parameters.SourcePath
  • Use **dataset parameters** to allow file/table names to be passed in

Sink Dataset (Generic)
• Similar setup for sink
• Parameterize table name or folder path
• Supports dynamic routing of data

Step 3: Use Expressions in Activities

Configure Copy Data or Data Flow
• In **Copy Activity**, set source and sink dynamically:

- Select appropriate dataset
- Pass pipeline parameters to dataset properties
  • In Data Flow, use parameterized datasets and data flows if needed

Step 4: Add Conditional Logic (Optional)

Use If Condition or Switch Activity
• Use If Condition or Switch to handle multiple source types
• Each branch can have its own logic or datasets
• Helps when source/sink types vary widely (e.g., REST vs SQL vs Blob)

Step 5: Trigger with Parameters

Trigger Setup
• Go to Triggers > New
• When creating a trigger (manual or scheduled), define values for input parameters
• These values will be used at runtime to determine data flow paths

Scenario 10
You need to design a near real-time data processing solution using Azure Data Factory (ADF) to ingest, process, and analyze streaming data (e.g., IoT data, application logs).

Step 1: Understand ADF's Role in Near Real-Time

ADF is primarily batch-oriented, but it can be integrated into near real-time solutions using short-interval triggers or when paired with other services like Event Hubs, Stream Analytics, or Functions.

Step 2: Ingest Streaming Data

Use Azure Event Hubs or IoT Hub
• Stream data from source systems to Event Hub or IoT Hub
• Acts as the entry point for continuous streaming data

Step 3: Stream Processing (Outside ADF)

Use Azure Stream Analytics or Azure Functions
• Use Stream Analytics job to read from Event Hub
• Perform filtering, transformation, aggregation
• Output results to a staging zone like ADLS Gen2, Blob Storage, or Azure SQL

Step 4: Use ADF to Process the Output

ADF Picks Up Processed Data for Further ETL or Analysis
• Use ADF to poll the output sink (e.g., ADLS, SQL) every few minutes
• Create a pipeline with a Copy Data activity
• Transform and load into final storage (Azure SQL, Synapse, etc.)
• Use schedule triggers at short intervals (e.g., every 5 minutes)

Step 5: Build Analytics and Visualization Layer

Analyze with Power BI or Azure Synapse
• Load curated data into Azure SQL Database, Synapse, or Data Lake
• Connect Power BI dashboards to these data stores
• Set auto-refresh intervals in Power BI for near real-time visibility

Step 6: Monitoring and Scalability

Use Monitoring & Alerts
• Monitor Event Hub throughput and ADF pipeline executions
• Use Azure Monitor and Log Analytics for real-time diagnostics
• Configure retry policies in ADF for transient failures

Scenario 11

Your company needs to copy data from a REST API endpoint to an Azure SQL Database every hour. How would you set this up in Azure Data Factory?

Step 1: Create Linked Services

rest API linked service
- Go to Manage > Linked Services > New
- Select REST as the connector
- Provide the base URL (e.g., https://api.company.com/data)
- Configure authentication (API key, OAuth2, or basic)
- Use Azure Key Vault for secrets management

azure sql database linked service
- Select Azure SQL Database as connector
- Provide server name, database, and authentication details
- Use managed identity or Key Vault-secured credentials

Step 2: Create Datasets

source dataset (rest)
- Use REST connector
- Set the relative URL and HTTP method (GET/POST)
- Configure pagination rules if needed
- Set JSON as the data format

sink dataset (azure sql)
- Use Azure SQL Database connector
- Point to the destination table
- Match schema with expected data format

Step 3: Create the Pipeline

add copy data activity
- Add a Copy Data activity in the pipeline
- Configure source and sink using the datasets
- Use Mapping tab to map fields
- Flatten JSON if nested structure is returned

Step 4: Schedule the Pipeline

configure trigger
- Go to Triggers > New
- Choose Schedule trigger
- Set recurrence to every 1 hour
- Attach the trigger to the pipeline

Step 5: Test and Monitor

debug and monitor
• Test the pipeline using Debug mode
• Monitor execution in the Monitor tab
• Set up alerts for failures using Azure Monitor

Scenario 12

You need to perform a lookup operation in Azure Data Factory to fetch a configuration value from an Azure SQL Database table and use it in subsequent activities. Describe how you would do this.

Step 1: Create Linked Service

azure sql database linked service
• Go to Manage > Linked Services > New
• Choose Azure SQL Database connector
• Provide server name, database name, and authentication
• Use managed identity or Azure Key Vault for secure access

Step 2: Create Dataset for Lookup

sql dataset for configuration table
• Create a dataset pointing to the configuration table
• Select the table or use a parameterized query if needed
• This dataset will be used in the Lookup activity

Step 3: Add Lookup Activity to the Pipeline

configure lookup activity
• In the pipeline, add a Lookup activity
• Set the source dataset created above
• In the settings tab, choose query or table
• Example query: SELECT config_value FROM config_table WHERE config_key = 'FilePath'
• Enable "First row only" if expecting a single value

Step 4: Use Output in Subsequent Activities

reference lookup output
• Access the value using dynamic expression:
@activity('LookupActivityName').output.firstRow.config_value
• Use this in parameters of other activities like Copy Data, ForEach, or Script activity
• Can also pass this value into dataset parameters or pipeline variables

Step 5: Test and Monitor

debug and verify
• Run the pipeline in Debug mode to check the lookup value
• Confirm it's being passed correctly to dependent activities
• Monitor execution in the Monitor tab

Scenario 13

Your pipeline must process a large number of files stored in an Azure Data Lake Storage Gen2 account. How would you efficiently process these files using Azure Data Factory?

Step 1: Create Linked Services

azure data lake gen2 linked service
• Go to Manage > Linked Services > New
• Select Azure Data Lake Storage Gen2 connector
• Authenticate using managed identity or Azure Key Vault

Step 2: Create Parameterized Dataset

dataset with dynamic file path
• Create a dataset pointing to the folder containing files
• Use parameters for folder and file name
• Supports dynamic file referencing in pipeline activities

Step 3: Use Get Metadata Activity

list files in folder
• Add a Get Metadata activity to retrieve a list of files
• Set field list to “child items”
• Configure to point to the folder in ADLS Gen2

Step 4: Add ForEach Activity

iterate over file list
• Use the output of Get Metadata as input to ForEach
• Configure ForEach to loop through all file names
• Enable “Batch Count” to control parallel execution

Step 5: Inside ForEach, Add Copy Activity or Data Flow

process each file
• Use a Copy Data activity or Mapping Data Flow inside the ForEach
• Pass the current file name as a parameter to the dataset
• Copy or transform the content as needed to the destination (e.g., Azure SQL, Synapse)

Step 6: Optimize for Performance

enable parallelism
• Set Batch Count in ForEach for concurrent file processing
• Use staging in Copy Activity if moving large datasets
• Use Data Flows with partitioning if complex transformations are needed

Step 7: Monitor and Scale

monitor and troubleshoot
• Monitor performance and failures in the Monitor tab
• Scale integration runtime if needed for performance

Scenario 14

You need to transform and load data from a SQL Server database to a Parquet file in Azure Blob Storage. Describe the steps to achieve this using Azure Data Factory.

Step 1: Create Linked Services

sql server linked service
• Go to Manage > Linked Services > New
• Select SQL Server connector
• Provide server name, database, and authentication details
• Use Self-hosted Integration Runtime if on-premises

azure blob storage linked service
• Select Azure Blob Storage connector
• Authenticate using account key, SAS token, or managed identity

Step 2: Create Datasets

source dataset (sql server)
• Create a dataset using SQL Server connector
• Select the required table or use a custom SQL query

sink dataset (parquet file in blob)
• Create a dataset with Azure Blob Storage and set format to Parquet
• Use dynamic file name if needed (e.g., with timestamp or ID)

Step 3: Create a Data Flow (Optional for Transformation)

create mapping data flow
• Add a new Data Flow
• Source: point to SQL Server dataset
• Add transformations like filter, derive column, join, aggregate, etc.
• Sink: configure Parquet format and output folder in Blob

Step 4: Add Pipeline and Activities

build pipeline
• Add a Data Flow activity if using transformations
• Or use a Copy Data activity if no transformation is required
• Set source and sink datasets
• Enable staging if needed for performance

Step 5: Trigger and Monitor

configure schedule and monitor
• Add a trigger to schedule the pipeline (e.g., daily)
• Monitor execution in the Monitor tab
• Set up alerts for failure handling if necessary

Scenario 15

You need to send an email notification if a pipeline in Azure Data Factory fails. How would you set this up?

Step 1: Create a Logic App for Email Notification

create logic app
• In the Azure portal, create a new Logic App
• Use the "When an HTTP request is received" trigger
• Add an action to send an email (e.g., Outlook, Gmail, or SMTP)
• Customize subject and body with dynamic content
• Save the Logic App to generate the HTTP POST URL

Step 2: Add Web Activity to ADF Pipeline

configure web activity
• In the ADF pipeline, add a Web activity
• Set the URL to the Logic App's HTTP POST endpoint
• Set method to POST and pass the required email content as JSON in the body
• Example: { "subject": "Pipeline Failed", "message": "ADF pipeline 'X' has failed." }

Step 3: Use Activity Dependency for Failure Path

configure on-failure dependency
• Connect the Web activity to the previous activity using a red (Failure) dependency
• This ensures the email is only triggered if the previous activity fails

Step 4: Optional – Send Email on Pipeline Failure (Global Scope)

use pipeline-level failure handling
• Add the Web activity in a separate Failure branch from the main pipeline activities
• Or wrap the main logic in an If Condition or Try-Catch pattern using variables and success flags

Step 5: Test and Monitor

validate notifications
• Trigger the pipeline with an intentional failure to test the email
• Monitor the email delivery and Logic App run history for debugging

Scenario 16

You need to implement a data pipeline that reads data from an Azure Event Hub, processes it in real-time, and writes the results to an Azure SQL Database. Explain how you would achieve this.

Step 1: Set Up Azure Event Hub

create and configure event hub
• In Azure portal, create an Event Hub namespace and Event Hub
• Define consumer group(s) for reading data
• Send test messages to verify the Event Hub is working

Step 2: Use Azure Stream Analytics for Real-Time Processing

create stream analytics job
• Create a Stream Analytics job in Azure
• Set the input to Event Hub using its connection string and consumer group
• Define query to filter, transform, or aggregate incoming data
• Example query:

```
SELECT deviceId, AVG(temperature) AS avgTemp INTO outputSQL FROM inputStream
TIMESTAMP BY eventTime GROUP BY deviceId, TumblingWindow(minute, 1)
```

Step 3: Set Azure SQL Database as Output

configure sql output
• In the Stream Analytics job, add an output of type Azure SQL Database
• Provide the server, database name, and authentication details
• Define the target table schema to match the output of the query

Step 4: Create and Prepare SQL Table

define schema
• In Azure SQL Database, create the table to store processed data
• Ensure columns match the output fields from the Stream Analytics job

Step 5: Start the Pipeline

run stream analytics job
• Start the Stream Analytics job to begin real-time data flow
• Monitor input, output, and any dropped records

Step 6: Monitor and Scale

monitor performance
• Use metrics in Stream Analytics to monitor throughput, latency, and errors
• Scale Event Hub throughput units or Stream Analytics SU (Streaming Units) as needed

Scenario 17

You need to load data from multiple sources (e.g., SQL Server, Oracle, and flat files) into a single data warehouse in Azure Synapse Analytics. Describe your approach using Azure Data Factory.

Step 1: Create Linked Services for Each Source

sql server linked service
• Use the SQL Server connector
• Configure authentication and use Self-hosted Integration Runtime if on-premises

oracle linked service
• Use Oracle connector with proper connection string
• Requires Self-hosted Integration Runtime if on-premises

flat files (csv) linked service
• Use Azure Blob Storage or ADLS Gen2 connector
• Authenticate using account key or managed identity

azure synapse linked service
• Use Azure Synapse Analytics connector
• Provide server, database, and credentials via Key Vault or managed identity

Step 2: Create Source and Sink Datasets

datasets for each source
• Create individual datasets for SQL Server, Oracle, and flat files
• For flat files, configure file path and format (CSV, delimiter, etc.)

sink dataset for synapse
• Create a dataset pointing to the staging or final tables in Synapse

Step 3: Create Pipelines for Each Data Source

build data pipelines
• Use Copy Data activity to move data from each source to Synapse
• Apply pre-copy scripts or transformations if required
• Use staging (e.g., Blob storage) for large datasets to improve performance

Step 4: Use Data Flows for Complex Transformations

configure mapping data flows
• If transformation is required (e.g., joins, filters, conversions), use Mapping Data Flow
• Define transformations and load the result into Synapse

Step 5: Orchestrate with a Master Pipeline

create master pipeline
• Use Execute Pipeline activities to call individual source pipelines
• Handle dependencies using success/failure conditions
• Parameterize source names, file paths, or table names for reusability

Step 6: Schedule and Monitor

add triggers and monitoring
• Add a schedule trigger (e.g., daily load)
• Monitor all pipeline runs in the Monitor tab
• Configure alerts for failure notifications

Scenario 18

Your data pipeline must run under specific conditions, such as when a particular file is available in Azure Blob Storage. How would you configure this trigger in Azure Data Factory?

Step 1: Create Linked Service for Azure Blob Storage

azure blob storage linked service
• In Azure Data Factory, go to Manage > Linked Services > New
• Choose Azure Blob Storage connector
• Authenticate using account key, SAS token, or managed identity

Step 2: Create a Dataset for the File

create blob dataset
• Create a dataset pointing to the container or folder where the file will appear
• Use dynamic file name or wildcards if necessary

Step 3: Configure Event-Based Trigger

create blob event trigger
• Go to Manage > Triggers > New
• Choose Event trigger type
• Select the Azure Blob Storage linked service
• Set the event type to "Blob Created"
• Specify the container and path where the file will be uploaded
• Optionally, use filters for file name patterns (e.g., input/data_*.csv)

Step 4: Associate Trigger with Pipeline

bind trigger to pipeline
• In the pipeline designer, click Add Trigger > New/Edit
• Select the event trigger created above
• Pass file name or path as a parameter if needed in the pipeline

Step 5: Handle File Detection in Pipeline Logic (Optional)

optional file validation
• Use Get Metadata activity to confirm the file exists
• Use If Condition activity to verify file size or last modified time
• Proceed with Copy Data or transformation only if conditions are met

Step 6: Test and Monitor

verify file-based trigger
• Upload a test file to the specified blob location
• Check that the trigger fires and the pipeline runs
• Monitor activity in the ADF Monitor tab and set alerts if needed

Scenario 19

You need to create a pipeline that performs conditional data processing based on the value of a parameter passed at runtime. Explain how you would implement this in Azure Data Factory.

Step 1: Define Pipeline Parameters

add runtime parameters
• In the pipeline, go to the Parameters tab
• Add a parameter (e.g., processType) to control logic at runtime
• This parameter will be set when triggering the pipeline

Step 2: Use If Condition Activity

add conditional logic
• Add an If Condition activity to the pipeline
• In the expression, check the value of the parameter
Example: @equals(pipeline().parameters.processType, 'full')
• The “True” and “False” branches define separate logic flows

Step 3: Add Activities in Each Branch

define conditional execution
• In the True branch (e.g., for full load), add activities like Lookup, Copy Data, or Data Flow
• In the False branch (e.g., for incremental load), define alternative logic
• You can nest further conditions or actions in each branch

Step 4: Trigger Pipeline with Parameter

set parameter at runtime
• When manually triggering or using a scheduled trigger, provide the parameter value
• If triggering via REST API or another pipeline, pass the parameter in the JSON body

Step 5: Test and Validate

verify conditional logic
• Trigger the pipeline with different parameter values
• Confirm that only the appropriate branch executes based on the condition
• Monitor execution in the Monitor tab and review run outputs

Scenario 20

You are required to implement a pipeline that processes daily transactional data and updates a fact table in an Azure SQL Data Warehouse, ensuring no duplicate records. Describe your approach.

Step 1: Create Linked Services

azure data source linked service
• Set up a linked service to the source system (e.g., SQL Server, Blob Storage, etc.) where daily transactional data resides
• Use appropriate authentication and integration runtime

azure synapse analytics linked service
• Create a linked service for Azure SQL Data Warehouse (Synapse Analytics)
• Use managed identity or Azure Key Vault for credentials

Step 2: Create Datasets

source dataset
• Create a dataset pointing to the daily transactional data
• This could be a table or file that gets refreshed daily

sink dataset (fact table)
• Create a dataset for the destination fact table in Synapse Analytics
• Ensure it points to the correct schema and table

Step 3: Build the Pipeline

pipeline setup
• Use a Copy Data activity if no transformation is required
• Use a Data Flow activity for filtering, deduplication, or complex logic
• Add a derived column or filter transformation if needed to tag or limit daily data

Step 4: Implement Deduplication Logic

data flow transformation
• In the Data Flow, use a Join activity between source data and target fact table on primary keys
• Use a conditional split to filter out records that already exist
• Load only new records into the sink

or use stored procedure
• Alternatively, use a Pre-copy stored procedure to remove or flag duplicates in the target table
• Or use MERGE (UPSERT) logic inside a stored procedure

Step 5: Load Data into Fact Table

write to synapse
• Use sink settings in Data Flow or Copy Data activity to load new records
• Set write behavior to "Upsert" or use staging table with post-load merge

Step 6: Schedule and Monitor

schedule pipeline
• Add a schedule trigger to run the pipeline daily
• Monitor pipeline runs in the Monitor tab
• Set up alerts or retry policies in case of failure

Scenario 21
Your organization wants to integrate ADF with an Event Hub to process real-time streaming data.

Question 1: How would you set up ADF to process data from an Event Hub?

Step 1: Understand ADF's Real-Time Capabilities
ADF is primarily designed for batch and scheduled data movement. It does not natively support real-time streaming ingestion directly from Azure Event Hub. However, you can integrate ADF with Azure Stream Analytics or Azure Functions to indirectly support real-time processing.

Step 2: Use Azure Stream Analytics for Real-Time Ingestion
Create a Stream Analytics job to consume data from Azure Event Hub.
Set Event Hub as the input source in the Stream Analytics job.
Configure data serialization (e.g., JSON) and timestamp fields.

Step 3: Process and Output Data to a Staging Store
In Stream Analytics, write the output to a staging area such as Azure Blob Storage, Azure Data Lake Storage Gen2, or directly to Azure SQL Database or Synapse Analytics.
This provides a bridge between real-time data and ADF's batch processing capabilities.

Step 4: Use ADF to Further Transform or Move the Data
Create a pipeline in ADF to pick up processed data (e.g., from Blob Storage or Synapse).
Optionally add data transformations or load into downstream systems.
Trigger the ADF pipeline on a schedule or via event-based triggers when new files arrive in storage.

Question 2: What are the limitations of using ADF for real-time processing?

ADF is not a true real-time stream processing engine, and it has the following limitations:

No native Event Hub integration for streaming inputs – ADF does not directly support streaming input from Event Hubs.
Batch-oriented architecture – ADF works best with scheduled, time-triggered, or event-driven batch pipelines.
Latency introduced by polling and triggers – Even with event-based triggers, there is usually a short delay (often 1–2 minutes or more) before a pipeline runs.
Limited transformation capabilities for real-time use – Complex low-latency event transformations are not possible natively in ADF.
Not suitable for sub-second SLAs – ADF is not designed for use cases requiring near-instant data delivery or processing.

Question 3: How does ADF integrate with other Azure services like Stream Analytics for real-time use cases?

Step 1: Stream Analytics Handles Real-Time Ingestion
Azure Stream Analytics is used to consume streaming data from Event Hubs in real-time.
You can write SQL-like queries in Stream Analytics to filter, aggregate, and transform the streaming data on-the-fly.

Step 2: Send Processed Data to Storage or SQL
Output from Stream Analytics can be sent to Azure SQL Database, Synapse Analytics, Blob Storage, or Data Lake.
These outputs can then act as sources for ADF pipelines.

Step 3: Use ADF for Downstream Processing or Orchestration
ADF can be scheduled to pick up files/data output by Stream Analytics for further processing or archival.
Use ADF to merge real-time data with historical data, run complex aggregations, or load into analytical models.
ADF pipelines can also call stored procedures, notebooks (via Synapse or Databricks), or external APIs to enrich data.

Step 4: End-to-End Integration Flow

1. Event Hub receives streaming data
2. Stream Analytics reads and processes in near real-time
3. Processed data lands in Blob Storage or Synapse
4. ADF picks up and moves/transforms the data as part of larger data workflows

Scenario 22
You have a large dataset stored in Azure Data Lake that needs to be processed by date partitions.

Question 1: How would you design a pipeline to process data in partitions?

Step 1: Store Data in Partitioned Structure
Organize your data in folder structures such as year=2025/month=07/day=16/.
This makes it easier to access and process files for specific dates.

Step 2: Create Pipeline Parameters
Add pipeline parameters like year, month, and day to control which partition to process.

Step 3: Use Parameterized Datasets
In your source dataset, use dynamic expressions in the file path such as:
@concat('datalake/input/year=', pipeline().parameters.year, '/month=', pipeline().parameters.month, '/day=', pipeline().parameters.day)
This allows ADF to read from the correct folder.

Step 4: Add a Copy or Data Flow Activity
Use a Copy Data or Mapping Data Flow activity to read and transform the partitioned data.
Sink can be Azure SQL, Synapse, or another storage layer.

Step 5: Trigger the Pipeline Dynamically
You can run the pipeline with parameters from a parent pipeline, manually, via REST API, or with scheduled triggers.

Question 2: What are the advantages of data partitioning in ADF?

Improved performance – Smaller data volumes per partition reduce processing time and memory usage.
Scalability – You can parallelize execution across multiple partitions for faster throughput.
Maintainability – Easier to track, audit, and reprocess specific date or category segments.
Cost efficiency – Reduces unnecessary scans of entire datasets, lowering compute and read costs.
Flexibility – Allows rerunning failed or missing partitions without reprocessing all data.

Question 3: How would you use dynamic expressions to process each partition dynamically?

Step 1: Use ForEach Activity for Iteration
Define a list of dates or folders to iterate over using a Lookup activity or hardcoded array.
Use a ForEach activity to loop through each partition.

Step 2: Pass Values as Parameters
Inside the ForEach, call an Execute Pipeline or activity that accepts year, month, day as parameters.
Bind the loop item to each parameter using item() function.

Step 3: Parameterize Dataset Paths with Expressions
Use expressions like:
@concat('datalake/input/year=', pipeline().parameters.year, '/month=', pipeline().parameters.month, '/day=', pipeline().parameters.day)
This lets ADF resolve the exact partition at runtime.

Step 4: Enable Parallel Processing
Set the ForEach activity’s Batch Count and enable IsSequential = false to process multiple partitions concurrently.

Scenario 23
You have multiple pipelines, and one pipeline should only start after the successful completion of another.

Question 1: How would you implement dependencies between pipelines in ADF?

Step 1: Use Execute Pipeline Activity
Within a master pipeline, use the Execute Pipeline activity to call and run other pipelines in sequence or parallel.
Configure activity dependencies using success, failure, or completion conditions between each activity.

Step 2: Use Trigger Chaining (Event-Based)
Set up a trigger for the second pipeline that listens for the successful run of the first pipeline.
Go to Manage > Triggers > New and select "Tumbling Window" or "Custom Event Trigger" with a dependency on the previous pipeline run.

Step 3: Monitor Pipeline Execution
Use pipeline monitoring in ADF to verify that dependencies are honored, and that downstream pipelines only run after upstream ones complete successfully.

Question 2: What are the pros and cons of using execute pipeline activity versus trigger chaining?

Execute Pipeline Activity
Pros:

- Easier to manage sequential logic within one pipeline.
- Error handling and retry logic can be defined within the master pipeline.
- Simpler to orchestrate complex dependencies.

Cons:

- All pipelines are tied together; failure in one can block the master pipeline.
- Limits reusability of sub-pipelines in independent scenarios.
- May increase the size and complexity of the master pipeline.

Trigger Chaining
Pros:

- Loose coupling between pipelines for better modularity.
- Pipelines are independently executable and manageable.
- Easier to scale and maintain when pipelines are large and unrelated.

Cons:

- Harder to control execution flow (e.g., conditional logic or error branching).
- Monitoring dependencies is slightly less intuitive.
- Delay between chained trigger executions is possible.

Question 3: How would you handle scenarios where one pipeline fails but others should continue?

Step 1: Use Dependency Conditions
In your master pipeline or data flow, configure the dependency conditions to Completion instead of Success so that the next activity executes regardless of the outcome.

Step 2: Add If Condition or Switch Activities
Use If Condition activity to check the output status of previous steps and decide whether to continue or skip subsequent activities.

Step 3: Implement Retry and Timeout Policies
Add retry settings and timeout properties on critical activities to prevent pipeline failure from transient errors.

Step 4: Use Logging and Alerts
Log errors in a storage or database, and use Azure Monitor or Logic Apps to alert teams about failed pipelines while allowing others to continue.

Scenario 24
You have a master pipeline that orchestrates the execution of multiple child pipelines. Some child pipelines are dependent on the output of others.

Question 1: How would you design the master pipeline to handle dependencies between child pipelines?

Step 1: Use Execute Pipeline Activities
Add multiple Execute Pipeline activities in the master pipeline to call child pipelines.
Each activity can be connected with dependency conditions like success, failure, or completion based on how they relate.

Step 2: Define Activity Dependencies
Set up dependencies using the activity output. For example, connect ChildPipelineA to ChildPipelineB only if A succeeds.
Use the green (success), red (failure), or black (completion) arrows to configure execution flow.

Step 3: Pass Outputs Between Pipelines
Capture outputs from one pipeline and pass them as input parameters to the next.
You can use expressions like activity('ChildPipelineA').output.someOutput in the parameter mapping.

Step 4: Handle Conditional Logic
If certain child pipelines should only run under specific conditions, use an IfCondition activity to evaluate a flag or value returned from a previous activity.

Question 2: What are the differences between the "Wait" and "IfCondition" activities in this context?

Wait Activity

- Used to introduce a fixed delay between steps in the pipeline.
- Useful when you need to allow time for external systems to update or for eventual consistency.
- Does not evaluate any logic or pipeline state.

IfCondition Activity

- Used to execute different branches of logic based on an evaluated expression.
- Enables conditional execution of activities depending on the output of a previous pipeline or a parameter.
- Ideal for controlling flow based on success/failure flags, data volume, or business logic.

Question 3: How would you monitor and troubleshoot issues in a complex pipeline execution?

Step 1: Use the Monitor Tab in ADF
Go to the Monitor section to view activity runs, status, duration, and errors.
Drill down into individual activity logs to inspect inputs, outputs, and error messages.

Step 2: Enable Diagnostic Logging
Send ADF logs to Log Analytics or Azure Storage to keep detailed execution records.
Use Kusto queries in Log Analytics to filter failed runs and track trends.

Step 3: Add Logging Inside Pipelines
Use stored procedures, Web activity, or Append Variable activity to log checkpoints and outputs inside pipelines.
Store custom logs in Blob Storage or a control table for auditing.

Step 4: Implement Alerts with Azure Monitor
Set up alerts to notify on pipeline or activity failures via email, SMS, or integration with ITSM tools.
Configure based on severity or repeated failures.

Step 5: Modularize Pipelines
Break down the master pipeline into manageable child pipelines to isolate and troubleshoot issues more effectively.

Scenario 25
Your pipeline needs to be triggered whenever a file is uploaded to a specific container in Azure Blob Storage.

Question 1: How would you set up an event-based trigger in ADF?

Step 1: Enable Event Grid on the Storage Account
Go to the Azure portal, open the target storage account, and ensure that the Event Grid is enabled under the Events tab.

Step 2: Create an Event-Based Trigger in ADF
In Azure Data Factory, go to Manage > Triggers > New and select Event-based trigger.
Choose the linked storage account, then select the container and file path where file uploads should trigger the pipeline.

Step 3: Configure the Event Filter
Use blob path pattern (e.g., input/foldername/*.csv) to filter which files trigger the pipeline.
Ensure that you define the event type as Blob Created.

Step 4: Set Up Pipeline Parameters (Optional)
Pass dynamic content like file name or path as parameters to the pipeline using @triggerBody().fileName or @triggerBody().folderPath.

Step 5: Publish and Test
Publish the trigger, upload a test file to the configured path, and verify that the pipeline starts automatically.

Question 2: What are the advantages and limitations of using event-based triggers?

Advantages:

- Near real-time response to new data arrival.
- No need for polling or scheduling logic.
- Cost-efficient as it doesn't consume compute when idle.
- Supports metadata injection via dynamic expressions in parameters.

Limitations:

- May miss events if Event Grid is not configured correctly or if the event subscription is deleted.
- Event triggers only work for blob creation events, not for updates or deletions.
- Latency may exist during high-traffic periods.
- Debugging is more complex compared to time-based triggers.

Question 3: How would you handle scenarios where multiple files are uploaded simultaneously?

Step 1: Enable Concurrency in Pipeline
Ensure the pipeline can run multiple instances in parallel by setting concurrency in the trigger and pipeline settings.

Step 2: Design for Idempotency
Make the pipeline idempotent so that processing the same file twice won't cause issues (e.g., using watermarking or file logs).

Step 3: Use Parameters for File Identification
Capture the file name using @triggerBody().fileName to ensure each run processes a specific file independently.

Step 4: Consider a Queue-Based Pattern
For very high volume scenarios, consider using Event Grid to send events to an Azure Function or Logic App, which adds file names to a queue.
ADF can then read from the queue in batches to process files in a controlled way.

Step 5: Implement Logging and Retry Mechanisms
Log every file processed and implement retry logic for failures to avoid missing or duplicating file processing.

Scenario 26
Your pipeline frequently encounters transient network issues when copying data from an on-premises database to Azure SQL Database.

Question 1: How would you implement retry logic to handle transient errors?

Step 1: Use Built-in Retry Settings in Activities
In each activity (e.g., Copy Data), configure the retry policy by setting the Retry count and Retry interval in seconds.
This ensures that if a transient error occurs, the activity will automatically retry before failing.

Step 2: Identify Transient Error Types
Transient errors may include timeouts, dropped connections, or temporary network failures.
Ensure the retry policy is applied to those specific activities where such issues are likely.

Step 3: Use Timeout Settings
Set an appropriate timeout value to avoid premature failures. This gives the activity enough time to recover from short-lived disruptions.

Step 4: Isolate Critical Activities
Place network-sensitive activities in separate pipelines or activities so that only the affected step retries rather than the whole process.

Question 2: What settings in ADF activities allow for retries and delays?

Retry Settings:

- **Retry**: Defines how many times an activity should be retried after failure. Default is 0; you can set up to 10 retries.
- **Retry interval (in seconds)**: Specifies the delay between retry attempts. Default is 30 seconds.

Timeout Settings:

- **Timeout**: Maximum time the activity will run before it is terminated. Set in ISO 8601 duration format (e.g., PT30M for 30 minutes).

These settings are found in the activity's General > Retry and Timeout configuration section.

Question 3: How would you monitor and alert on excessive retries in pipeline executions?

Step 1: Enable Diagnostic Settings
Send ADF metrics and logs to Azure Monitor, Log Analytics, or Storage Account.
Include pipeline run status, activity retry count, and duration in logs.

Step 2: Query Retry Metrics in Log Analytics
Use Kusto queries to identify activities with high retry counts:

```
ADFActivityRun

| where RetryAttempt > 0

| summarize RetryCount = count() by ActivityName, PipelineName
```

Step 3: Set Up Alerts in Azure Monitor
Create alerts based on retry count, duration, or failed runs using metrics like Activity Failed Runs or Activity Retry Count.

Step 4: Notify Teams
Configure email, SMS, or webhook notifications to inform operations or engineering teams when retry thresholds are breached.

Step 5: Review and Tune Retry Settings
Analyze logs to determine if retry logic is effective or if upstream reliability improvements are needed.

Scenario 27
You are required to process only the files uploaded in the last 24 hours from Azure Blob Storage.

Question 1: How would you filter files based on their upload timestamp?

Step 1: Use Get Metadata Activity
Use the Get Metadata activity to retrieve the lastModified timestamp of each file in the blob container.

Step 2: Use ForEach Activity to Loop Through Files
Pass the list of files from Get Metadata into a ForEach activity to iterate over each file individually.

Step 3: Add an IfCondition Activity Inside the ForEach
Evaluate whether the file's lastModified property falls within the last 24 hours using a dynamic expression.
If true, pass the file to the next step (e.g., Copy Data); if not, skip it.

Question 2: What expressions or functions would you use to calculate time-based conditions?

Use the utcnow() function to get the current UTC timestamp.

Compare each file's lastModified value against utcnow() minus one day:

@if(greaterOrEquals(activity('Get Metadata').output.lastModified, addDays(utcnow(), -1)), true, false)

Alternatively, if you're looping files in ForEach, use:

@if(greaterOrEquals(item().lastModified, addDays(utcnow(), -1)), true, false)

This checks if the file was modified within the last 24 hours.

Question 3: How would you handle time zone differences when filtering files?

Step 1: Always Work in UTC
Azure Blob Storage records lastModified timestamps in UTC, and ADF's utcnow() function also uses UTC.
Perform all comparisons in UTC to avoid time zone mismatches.

Step 2: Convert Time Zones If Needed
If you receive input parameters in local time (e.g., from users or external triggers), use convertTimeZone() in Data Flows to standardize timestamps.

Example in Mapping Data Flows:

convertTimeZone('local', 'UTC', fileTimestamp)

Step 3: Use Azure Functions for Complex Time Logic
If your use case requires handling daylight saving or region-specific logic, offload time conversion to an Azure Function and call it from ADF.

Scenario: 28
Your pipeline processes files daily from Azure Blob Storage and loads them into Azure SQL Database. Duplicate files occasionally appear in the storage container.

Question 1: How would you design a pipeline to identify and skip duplicate files?

Step 1: Maintain a File Tracking Table
Create a table in Azure SQL Database to store metadata of processed files (e.g., file name, timestamp, hash).

Step 2: Use Get Metadata and ForEach Activities
Use the Get Metadata activity to list all files in the container and extract file names.
Loop through each file using a ForEach activity.

Step 3: Check Against Tracking Table
Use a Lookup activity inside the loop to query the tracking table to see if the file has already been processed.

Step 4: Conditional Processing
Use an IfCondition activity to process the file only if the Lookup result returns null (i.e., file not previously processed).
After successful processing, insert the file metadata into the tracking table.

Question 2: How would you use metadata (e.g., file names or hashes) to track processed files?

Step 1: Capture Metadata During Ingestion
Capture the file name, last modified timestamp, and optionally, a hash (e.g., MD5) of the file content.

Step 2: Generate Hash (Optional)
Use a custom Azure Function or Data Flow to compute the hash of the file content.
Store this hash in the tracking table to distinguish files with the same name but different content.

Step 3: Use File Name and Hash as Identifiers
Use a combination of file name and hash to determine uniqueness and avoid false positives due to name reuse.

Question 3: How would you recover if a duplicate file causes partial data corruption?

Step 1: Implement Transactional Loading
Use staging tables in Azure SQL Database to first load the data. Only after validating data integrity, move data to final tables.

Step 2: Log File and Load Status
Track the status (success/failure) of each file load in the tracking table.
On failure, record the error and stop further processing.

Step 3: Enable Retry and Rollback Mechanism
If corruption is detected, remove the partially loaded records using the file identifier (e.g., file name or batch ID).
Rerun the pipeline after removing the corrupted entry or fix the file manually.

Step 4: Use Data Validation Checks
Add data validation logic in Data Flow or SQL stored procedures to catch anomalies (e.g., duplicate keys, missing values) before committing the data.

Step 5: Alerting
Set up alerts via Azure Monitor to notify stakeholders when partial loads or duplicate detections occur.

Scenario: 29
Your pipelines process a daily batch of files, and you need to ensure that if a pipeline fails, it can resume from the last successfully processed file.

Question 1: How would you implement state management to track processed files?

Step 1: Create a Control Table for Tracking
Store file processing metadata (file name, processing date, status, and timestamp) in a dedicated table in Azure SQL Database.

Step 2: Lookup Before Processing
For each file in the batch, use a Lookup activity to check the control table for a status of "Processed" or "InProgress".
Skip any file already marked as "Processed".

Step 3: Update Status Throughout Pipeline
Before processing a file, insert or update its status to "InProgress".
Once processing completes, update the status to "Processed".
On failure, mark the status as "Failed" for easy retry.

Question 2: What role do control tables play in this scenario?

Tracking Progress
Control tables record the processing state of each file, allowing the pipeline to resume from where it left off after a failure.

Enabling Idempotency
They ensure that the same file isn't processed multiple times by checking whether it was already completed.

Facilitating Monitoring and Auditing
They provide an auditable trail of processed, failed, and pending files, which is useful for operational visibility and debugging.

Question 3: How would you ensure idempotency in pipeline executions?

Use File-Level Deduplication Checks
Check the control table before processing a file to avoid duplicate loads.

Design Target Tables for Upserts or Deletes
Use stored procedures or Data Flows to implement MERGE, UPSERT, or conditional DELETE + INSERT logic to handle repeated file loads gracefully.

Use Unique Identifiers
Tag each file or batch with a unique identifier (e.g., file name, date, or hash) so the data pipeline knows exactly what has been processed.

Avoid Intermediate State Corruption
Use staging tables or temporary containers to isolate new data until it's fully processed and validated.

Configure Retry and Error Handling
Retry failed files based on the "Failed" status in the control table, instead of re-running the entire batch.

Scenario: 30
You need to process data from multiple Azure regions and consolidate it into a central Azure Data Lake in a cost-efficient manner.

Question 1: How would you design a pipeline to handle geo-distributed data?

Step 1: Use Regional Integration Runtimes
Deploy self-hosted or Azure-hosted Integration Runtimes (IRs) in the same region as each data source to perform extraction locally.

Step 2: Build Region-Specific Pipelines
Create separate pipelines for each region that extract and stage data locally before transferring it to the central lake.

Step 3: Use a Central Orchestration Pipeline
Design a master pipeline that calls region-specific pipelines in parallel using Execute Pipeline activities. After regional pipelines complete, trigger downstream consolidation processes in the central region.

Step 4: Implement File Naming or Partitioning by Region
As you consolidate data into the central Azure Data Lake, prefix or partition the files by region to retain source traceability.

Question 2: What are the cost and performance considerations for cross-region data transfers?

Cost Considerations:

- **Egress Charges:** Data transfer between Azure regions incurs bandwidth charges.
- **Storage Redundancy:** Using geo-redundant storage may increase cost if not needed.

Performance Considerations:

- **Latency:** Higher latency can slow down transfers if data is copied across regions without optimization.
- **Throughput Limits:** Network bandwidth caps in some regions or services may throttle performance.

Mitigation Strategies:

- Compress data before transfer.
- Stage data in region before batching and sending.
- Use asynchronous copying to decouple performance from downstream operations.

Question 3: How can you use regional Integration Runtimes to optimize performance?

Reduced Latency:
Deploying Integration Runtimes in the same region as the data source minimizes the distance data travels during extraction.

Faster Ingestion:
Local IRs allow high-speed reads and writes to regional storage, improving overall data movement performance.

Improved Reliability:
Regional IRs reduce dependency on cross-region network stability and prevent pipeline failures due to transient global network issues.

Design Tip:
Use Azure Data Factory's Auto-Resolve IR for centralized orchestration and assign custom regional IRs for dataset-specific copy activities.

Scenario: 31
You are responsible for ensuring data quality before loading it into the destination system. This includes null checks, duplicate checks, and threshold-based validations.

How would you implement data quality checks in ADF?

Use Mapping Data Flows in Azure Data Factory to define and enforce quality rules before data is loaded to the destination.

- Use the Filter transformation to exclude rows with null values in mandatory fields.
- Apply Derived Column to calculate new validation fields or flags.
- Use Conditional Split to separate valid and invalid records based on business rules.
- Add a sink for valid data (e.g., Azure SQL Database) and a separate sink for invalid data (e.g., error log in Azure Blob Storage).
- Integrate validation logic in a pipeline before loading steps to ensure bad data is never written to the final system.

What role do Data Flow transformations like Filter, Aggregate, and Exists play in these checks?

- Filter: Removes rows that don't meet criteria like null checks or threshold violations.
- Aggregate: Groups data to identify duplicates or perform summary-level validations (e.g., row count thresholds, sum validation).
- Exists: Checks referential integrity by validating whether a value exists in another dataset (e.g., product ID exists in the master product table).
  These transformations form the core of data quality logic and help define rules declaratively within a Data Flow.

How would you handle rows that fail quality checks?

- Use Conditional Split to redirect invalid rows to a separate branch.
- Write invalid data to a quarantine sink (e.g., Azure Data Lake or error table) with failure reasons attached.
- Maintain an error logging mechanism to capture details like row values, timestamp, and error messages.
- Send notifications using web activity or Logic Apps to alert teams of failed records.
- Optionally provide a mechanism to reprocess quarantined rows once corrected by users or automated scripts.

Scenario: 32
Your pipeline processes sensitive financial data that needs to be encrypted during transit and at rest.

How would you ensure end-to-end encryption for sensitive data in ADF?

- Enable HTTPS endpoints for all data movement to ensure encryption in transit.
- Use Azure-managed or customer-managed keys (CMK) to ensure encryption at rest for sources and sinks like Azure SQL, Blob Storage, or Data Lake.
- When using Azure Blob Storage or Data Lake as sources or sinks, ensure Secure Transfer Required is enabled.
- Use Private Endpoints to prevent data from traversing the public internet.
- Ensure SSL encryption is enforced on SQL databases by using encrypted connections in the linked service configuration.

How can you use Azure Key Vault for managing credentials and encryption keys?

- Store sensitive information like connection strings, passwords, API keys, and certificates securely in Azure Key Vault.
- In ADF, configure linked services to reference secrets directly from Key Vault instead of hardcoding them.
- Assign managed identity to ADF and grant it read access to the Key Vault via Access Policies or RBAC.
- For customer-managed encryption, use Key Vault–backed keys to manage encryption of storage accounts, databases, and Synapse workspaces.

What security best practices would you follow to secure data pipelines?

- Use Managed Identities to access Azure resources instead of storing credentials.
- Apply role-based access control (RBAC) to restrict access to pipelines, triggers, and datasets.
- Enable data access auditing and log all activities using Azure Monitor and Azure Activity Logs.
- Avoid using public IPs—prefer VNet integration for your Integration Runtime.
- Ensure data masking or tokenization is applied if handling PII/financial data during transformation.
- Regularly rotate secrets and credentials managed through Key Vault.
- Implement alerting and monitoring for failed activities or suspicious access behaviour.

Scenario: 33
You need to process data stored in partitions (e.g., year/month/day folders), but only for specific time ranges based on runtime parameters.

How would you configure the Copy Activity or Data Flow to read specific partitions dynamically?

- Use parameterized datasets with folderPath like: data/@{formatDateTime(pipeline().parameters.date, 'yyyy/MM/dd')}
- Pass partition values (year, month, day) as pipeline parameters
- Use dynamic content in the source dataset to construct the path
- In Data Flow, use parameterized source paths and source filters
- Optionally, use a Lookup activity to generate a list of required partitions and iterate with ForEach

What functions or expressions would you use to skip unnecessary partitions?

- Use formatDateTime() to format and match folder structure
- Apply conditional expressions like equals(), greaterOrEquals(), lessOrEquals()
- Use Get Metadata + Filter activity to get folder names and filter required dates
- Use expressions in ForEach to include only partitions within a desired date range
- In Data Flow, apply source filters using between() or toDate() on the date column

How can you optimize pipeline performance when dealing with highly partitioned data?

- Avoid wildcard path reads; use direct paths to specific partitions
- Use source partitioning to enable parallel reads (e.g., by folder or file name)
- Disable schema drift if not required
- Enable compression (gzip, snappy) to reduce file size and I/O
- Place Integration Runtime close to data location (region-wise)
- Minimize metadata scanning by targeting specific partitions
- Batch partition processing using ForEach and limit concurrency to balance performance and cost

Scenario: 34
You are tasked with processing unstructured data like log files or free-form text stored in Azure Blob Storage.

How would you handle unstructured data in ADF?

- Use ADF to ingest unstructured files (e.g., .log, .txt) from Azure Blob Storage using a binary dataset.
- Apply Data Flows or external compute (like Databricks) to process or parse the data, since ADF alone has limited parsing capability.
- Use Mapping Data Flows if the structure is partially consistent (e.g., delimited log lines) by reading it as text and applying string functions.
- If logs are JSON formatted or semi-structured, read them using JSON datasets and use schema projection.

What external tools (e.g., Databricks, Cognitive Services) can you integrate with ADF for parsing or extracting insights?

- **Azure Databricks**:
    - Launch a Databricks notebook from ADF using the Databricks activity.
    - Use Python/Scala in the notebook to parse logs using regex, NLP, or Spark transformations.
    - Store structured output (e.g., DataFrame to Parquet or CSV) in Azure Data Lake or Blob Storage.
- **Azure Cognitive Services**:
    - Integrate Text Analytics (sentiment, key phrases) or Language services via REST API.
    - Use Web Activity in ADF to call Cognitive Services and process results with Data Flows or stored procedures.
- **Azure Functions**:
    - Create a custom Azure Function for parsing specific log formats or unstructured content.
    - Call it from ADF via Web Activity or Azure Function activity.

How would you transform this data into a structured format for downstream processing?

- Step 1: **Ingest files** from Blob Storage using ADF Copy Activity or Databricks to raw zone
- Step 2: **Process text/log content**:

  Use Databricks/ADF to split content by line or delimiter

  Apply parsing logic (e.g., regex, split by space, timestamp extraction)
- Step 3: **Create structured schema** in Spark (Databricks) or Data Flow (ADF)

  Convert parsed fields into columns (e.g., timestamp, log level, message)
- Step 4: **Write structured output** to a staging area (e.g., Parquet or CSV in ADLS)
- Step 5: **Load structured data** into Azure SQL Database, Synapse, or another sink using ADF Copy Activity
- Step 6: **Validate and monitor** using ADF pipeline logging, success/failure outputs, or data quality checks

Scenario: 35
You have a pipeline with multiple parallel activities, and one of the activities fails intermittently due to source system issues.

How would you implement exception handling for individual activities in ADF?

- Use the "Continue on Failure" setting in the activity or within the parallel branch to allow execution to proceed even if the activity fails.
- Wrap sensitive activities within an "IfCondition" or "Try-Catch"-like logic using Success, Failure, and Completed dependency conditions.
- Use custom failure paths by connecting additional activities (e.g., logging or alerts) to the failed activity using the Failure condition.
- Optionally, use Execute Pipeline activities to isolate failure-prone logic for easier retry and control.

How can you ensure that the pipeline continues processing unaffected branches?

- Use parallel branches within the pipeline to run multiple tasks simultaneously.
- Set independent branches with no dependency between them to ensure one branch's failure doesn't block the others.
- On each branch, configure failure-prone activities with Continue on Failure = true if downstream logic can proceed regardless.
- Ensure activity dependencies are scoped correctly so failure in one path doesn't cause a cascade stop in other paths.

What strategies would you use to retry or log failed activities?

- Use the Retry policy settings available in most ADF activities:

  Set retry count (e.g., 3 times) and retry interval (e.g., 30 seconds)

- Implement a Failure path that logs errors to a log table, Blob file, or sends alerts (e.g., via Logic App or Azure Function).
- Use Web Activity or Stored Procedure to log the error message and activity metadata.
- For persistent failures, send notifications through Azure Monitor alerts, Log Analytics, or email via Logic App.
- Design the pipeline with modular retries—e.g., use nested pipelines with retries configured on Execute Pipeline activity.
- Maintain a tracking table to log failed records or partitions and use this for later reprocessing.

Scenario: 36
Your pipeline needs to process files dynamically based on folder structure and file patterns in Azure Data Lake.

How would you use wildcard file paths in ADF to process specific files?

- Use wildcard characters (*, ?) in the file path or file name property of the dataset.
- In the Copy Activity or Data Flow, set the folder path and use *.csv, data_*.json, or similar patterns in the file name field.
- Combine with dynamic expressions to parameterize folder or file patterns (e.g., @concat('year=', pipeline().parameters.year, '/month=', pipeline().parameters.month, '/*.csv')).
- Use Get Metadata activity to fetch a list of files matching a pattern and then ForEach loop to process each file individually.

How can you create folders dynamically based on runtime parameters?

- In the sink dataset, configure the output path using dynamic content with parameters (e.g., @concat('archive/', formatDateTime(utcNow(),'yyyy/MM/dd'))).
- Use Copy Activity or Data Flow to write to these dynamically generated folder paths.
- Use pipeline parameters like year, month, or custom variables to build folder structure at runtime.
- ADF will automatically create folders if they don't exist when writing to a dynamic path.

What are the challenges of managing large numbers of folders and files, and how would you address them?

- Challenge: Slow performance in listing or processing thousands of folders/files

  Solution: Use filtering in Get Metadata or partitioned reads to narrow the scope
- Challenge: Handling duplicates or reprocessing

  Solution: Maintain a control table to track processed files by name, timestamp, or checksum
- Challenge: Debugging pipeline failures across many files

  Solution: Use log tables, file-specific error logging, and alerting via Azure Monitor
- Challenge: Cost and performance in reading entire folder hierarchies

  Solution: Design file structures with optimized partitioning and avoid overly deep folder trees
- Challenge: Concurrency when processing many files

  Solution: Use batching in ForEach activity and throttle concurrency using batchCount property

Scenario: 37
Your team needs to collaborate with another team to build pipelines that share dependencies and datasets.

How would you manage shared resources (e.g., Linked Services, Datasets) across teams?

- Create a centralized Azure Data Factory instance or shared integration project where common Linked Services and Datasets are defined.
- Use naming conventions and folder structures to organize shared versus team-specific resources.
- Implement parameterization in Linked Services and Datasets to make them reusable across different pipelines and environments.
- Store and manage secrets like connection strings or credentials using Azure Key Vault for centralized security and access control.
- Set appropriate role-based access control (RBAC) on ADF to allow read/write or view-only access as required for each team.

What strategies would you use to avoid conflicts in pipeline development?

- Use branching strategies in Git (e.g., feature branches) to allow each team to work independently on their components.
- Define clear ownership of pipelines or modules to prevent overlapping edits.
- Implement code review processes and pull requests for changes that impact shared resources.
- Schedule regular integration checkpoints to merge, test, and validate combined work across teams.
- Adopt naming standards and tagging to avoid ambiguity and collisions in resource names.

How can Git integration help streamline collaboration between teams?

- Git integration in ADF enables version control, allowing teams to track changes, roll back, and manage historical versions.
- Teams can work on separate branches and use pull requests for controlled merging.
- Allows for automated CI/CD with tools like Azure DevOps or GitHub Actions to deploy changes across environments.
- Encourages code reuse and modularity by allowing pipeline templates and reusable components to be shared through Git.
- Provides auditability, ensuring changes are documented and traceable across the lifecycle.

Scenario: 38
You need to notify stakeholders immediately when a pipeline or activity fails, including error details.

How would you implement real-time error notifications using Azure Monitor or Logic Apps?

- Enable diagnostic settings in Azure Data Factory to send logs and metrics to Azure Monitor, Log Analytics, or Event Hub.
- Create alerts in Azure Monitor based on specific metrics or activity logs (e.g., pipeline failure, activity duration threshold exceeded).
- Use Azure Logic Apps as an alert action group to send real-time notifications via email, Teams, or SMS.
- Design the Logic App to parse alert payloads, extract error details (pipeline name, activity, error message), and format a clear message for recipients.
- Optionally, log the alert information in a centralized error tracking system for audit and follow-up.

How can you configure email or SMS alerts for pipeline failures?

- Create an Azure Monitor Alert Rule with the condition:
  - *Signal type:* Metric or Log
  - *Condition:* Activity or pipeline run failed
- Add an Action Group with an Email/SMS/Push/Voice action type.
- Provide stakeholders' contact details in the Action Group.
- If more customization is needed (e.g., email body, multi-channel notification), configure the action to trigger a Logic App or Azure Function that formats and sends the message.
- Test the alerts to ensure delivery and accurate content.

What are the key metrics and logs to monitor for proactive issue detection?

- PipelineRun metrics like status (Failed, Succeeded, InProgress), run duration, and concurrency.
- ActivityRun metrics including status, failure type, and error code/message.
- Integration Runtime health, including CPU, memory, and job queue length (for SHIR).
- TriggerRun logs to track scheduled pipeline executions.
- Use Log Analytics queries to analyze trends or detect anomalies such as consistent failures at the same time daily or repeated activity errors.
- Set up dashboard visualizations in Azure Monitor for key performance indicators and health tracking.

Scenario: 39
You are part of a large organization with multiple teams working on separate ADF projects. Central governance is required for Linked Services, triggers, and naming conventions.

How would you implement centralized governance for ADF projects?

- Establish a central architecture or data platform team to define standards, templates, and best practices.
- Maintain shared resources (Linked Services, Datasets, Parameters) in a central ADF instance or in a shared Git repository.
- Use Azure DevOps or GitHub to manage branches, pull requests, and enforce code reviews across ADF solutions.
- Implement CI/CD pipelines to control deployment and approval of changes to shared or production environments.
- Document and enforce standards around naming conventions, folder structure, parameter usage, and access control.
- Use role-based access control (RBAC) to define permissions for different teams, ensuring separation of duties.

How can Azure Policy or Resource Manager templates enforce naming conventions?

- Use Azure Resource Manager (ARM) templates to define and deploy ADF resources with predefined naming patterns.
- Create Azure Policies that include rules for resource naming (e.g., all pipelines must start with pl_, datasets with ds_).
- Apply these policies at the subscription or resource group level to ensure compliance across the organization.
- Integrate naming checks in CI/CD pipelines using custom scripts or linters before deploying to Azure.
- Use tags and metadata in templates to further standardize resource categorization (e.g., environment, owner, business unit).

What strategies would you use to manage shared Linked Services across teams?

- Create and manage shared Linked Services in a central ADF workspace, and provide access via managed identities or Key Vaults.
- Externalize sensitive connection strings and secrets to Azure Key Vault, so they're accessible across ADF instances with proper permissions.
- Define parameterized Linked Services using factory parameters, allowing teams to reuse templates with environment-specific values.
- Use Git repositories to centrally maintain and version shared Linked Services. Teams can pull the latest configurations as needed.
- Ensure clear documentation and ownership of shared resources, and use naming conventions (e.g., ls_sql_prod, ls_blob_dev) to avoid confusion.

Scenario: 40
You need to load data from multiple sources into corresponding tables in a destination, with dynamic schema mapping.

How would you configure dynamic sink mapping in a Copy Activity?

- Use Auto Mapping in the Copy Activity when the source and sink schemas have the same column names and compatible data types.
- If column names differ or require transformation, define a mapping JSON dynamically using data flows or pipeline parameters.
- In source and sink datasets, enable parameterization for table names, file paths, or schema definitions.
- Use Lookup or Metadata activity to retrieve schema metadata at runtime and construct dynamic mapping.
- When using Data Flows, configure Derived Column or Select transformations to align source fields with target schema before loading.

How can parameterization help in automating schema mapping?

- Parameterization allows reusing the same datasets and activities for multiple sources and destinations.
- Define dataset parameters for table names, column lists, file paths, etc., and pass values from the pipeline.
- Use expression language in ADF to build mappings dynamically (e.g., constructing JSON mappings using @concat() or @json() functions).
- Create configuration tables or files that define mappings between source and sink columns, and use Lookup + ForEach to apply them.
- This approach reduces duplication and manual intervention while supporting scalability and maintainability.

What challenges might you encounter when handling mismatched schemas?

- Data type mismatches (e.g., string vs. integer) can cause Copy Activity failures or data truncation.
- Missing columns in the source or sink may lead to runtime errors or incomplete data loads.
- Variations in column order, naming conventions, or case sensitivity can disrupt auto-mapping.
- Maintaining dynamic mappings for highly variable schemas can become complex and error-prone.
- Difficulties in validating schema changes across environments may require additional logging or schema comparison logic.
- Requires robust error handling and logging to track mismatches and data quality issues during execution.

Scenario: 41
You need to implement data retention policies for your pipelines, ensuring that data older than a certain period is deleted or archived.

How would you automate data retention policies in ADF?

- Use a scheduled pipeline (e.g., daily or weekly trigger) to check for data older than the defined retention period.
- Use Get Metadata or Lookup activity to list files or records and extract their creation or modified dates.
- Apply a Filter activity or conditional logic to identify files/records older than the threshold (e.g., 30 days).
- Use ForEach activity to iterate over the filtered list and perform actions such as delete or archive.
- If archiving, copy the data to a separate container, folder, or tier (e.g., cool/archive in Blob Storage) before deletion.

What role does the Delete Activity play in this process?

- The Delete Activity is used to remove datasets such as files or folders in Blob Storage, Data Lake, or other supported storage services.
- It supports filtering by file names, folder paths, and can delete recursively.
- It is typically used inside a ForEach loop or after filtering logic to remove only the qualified (aged) items.
- It includes a log option for previewing which items would be deleted and a wildcard path feature for targeting patterns.

How can you monitor and validate the successful execution of retention policies?

- Enable activity-level logging through Integration Runtime diagnostics or Pipeline monitoring.
- Use custom logging by writing details (e.g., file names, deletion status, timestamps) to a log file or audit table in Azure SQL or Blob.
- Integrate with Azure Monitor, Log Analytics, or Application Insights to set up alerts on failure or missing files.
- Use email notifications via Logic Apps or alerts to notify administrators of the retention pipeline's outcomes.
- Periodically audit storage and logs to ensure compliance with the retention policy and catch anomalies.

Scenario: 42
You need to process semi-structured data (e.g., JSON files with varying schemas) stored in Azure Blob Storage.

How would you handle schema variability while processing semi-structured data in ADF?

- Use Data Flows in Azure Data Factory, which support parsing semi-structured formats like JSON.
- Enable the "Infer drifted column types" option in the source settings to allow flexible schema detection.
- Use schema drift to handle columns that may appear or disappear across different files.
- Apply derived columns and conditional logic to clean or standardize values from optional or inconsistent fields.
- Use parameterized datasets and dynamic column handling to process files with similar structures but slight differences.

What transformations would you use in Mapping Data Flows to parse JSON data?

- Source transformation configured for JSON format and hierarchical structure.
- Use Select transformation to pick relevant fields and optionally rename them.
- Apply Derived Column transformation to modify or standardize extracted fields.
- Use Filter to exclude unnecessary or malformed records.
- When working with arrays, use the Flatten transformation to expand nested objects or arrays into a tabular format.

How can you flatten hierarchical data structures for downstream consumption?

- Use the Flatten transformation in Mapping Data Flows to expand nested arrays or objects.
- After flattening, use Select to rename or reorder the columns for consistency.
- If the JSON contains deeply nested structures, apply multiple Flatten transformations in sequence.
- If processing needs to be repeated, consider building a reusable pattern using parameterized pipelines or reusable data flows.
- For large or deeply nested data, consider pre-processing in Azure Databricks or using a custom transformation before ingesting into ADF.

Scenario: 43
Your pipeline processes files in batches based on their upload time, dynamically creating batches for every 24-hour period.

How would you design a pipeline to identify and process dynamic file batches?

- Use Get Metadata activity to list files in the target container.
- Add a Filter activity to compare the file's lastModified timestamp with the current time minus 24 hours.
- Leverage system variables like @utcNow() and expressions like addHours() to define the 24-hour window dynamically.
- Use ForEach activity to iterate through the filtered list and process each file.
- Maintain a tracking mechanism (e.g., Azure SQL table) to avoid reprocessing already-handled files.

How can you use metadata from Azure Blob Storage to determine batch boundaries?

- Configure Get Metadata activity with the childItems property to retrieve file list.
- For each file, use Get Metadata again with lastModified to capture upload timestamp.
- Store or compare these timestamps using pipeline variables or expressions to group files by 24-hour batch.
- Optionally, log processed file names and timestamps in a control table for validation and auditing.

What challenges might arise in handling late-arriving files, and how would you address them?

- **Challenge:** Files may arrive after the batch window closes, causing them to be skipped.

  Solution: Implement a grace period or buffer (e.g., delay the pipeline execution by a few hours).

- **Challenge:** Late files may be missed permanently.

  Solution: Store last processed timestamp and re-scan unprocessed files in the next batch.

- **Challenge:** Overlapping or duplicate processing.

  Solution: Use a control or audit table to record file names or timestamps and prevent duplicates.

- **Challenge:** Time zone differences may affect timestamp filtering.

  Solution: Normalize all datetime comparisons using UTC and apply consistent timezone handling in logic.

Scenario: 44
Your pipelines need to adapt dynamically based on metadata, such as file names, schema definitions, or transformation rules stored in a database.

How would you design a metadata-driven pipeline in ADF?

- Store metadata in a structured source like Azure SQL Database or a configuration file (e.g., JSON in Blob Storage).
- Use a Lookup activity at the beginning of the pipeline to retrieve metadata.
- Use pipeline parameters and expressions to pass metadata values dynamically to activities (e.g., file path, schema, transformation logic).
- Design the pipeline generically using parameterized datasets, linked services, and Copy/Data Flow activities.
- Use conditional logic (IfCondition, Switch) to apply specific transformations based on metadata values.

How can Lookup and ForEach Activities be used to retrieve and apply metadata?

- Lookup Activity is used to fetch metadata records (single row or array) from a database or configuration file.
- Use ForEach Activity to iterate over each row or item returned by the Lookup.
- Inside ForEach, dynamically set parameters for each child activity (Copy, Data Flow) using metadata values from the current iteration item.
- This allows the pipeline to process multiple files/schemas/rules in a scalable and dynamic manner.

What are the advantages of a metadata-driven approach in large-scale ETL processes?

- Scalability: A single pipeline can process multiple datasets without code duplication.
- Maintainability: Changes in logic or structure can be made in the metadata source without editing pipeline code.
- Flexibility: Easily accommodates new files, formats, or business rules by simply adding new metadata records.
- Consistency: Ensures uniform processing rules across datasets and teams.
- Automation: Enables dynamic processing that can adjust to new inputs or changes automatically.

Scenario: 45
You need to version control your ADF pipelines and collaborate with multiple developers in a shared workspace.

How would you enable Git integration in Azure Data Factory?

- Navigate to Azure Data Factory Studio > Manage > Git Configuration.
- Choose the Git repository type (Azure DevOps Git or GitHub).
- Provide the organization name, repository name, collaboration branch, and root folder.
- Authenticate using OAuth or PAT (Personal Access Token) depending on the repository type.
- Once connected, changes made to the factory UI will sync with the Git repository instead of directly affecting the live (published) environment.

What is the difference between collaboration and publish branches in ADF?

- The collaboration branch is where developers make and commit changes (e.g., main, develop). This branch holds the source version of your pipeline code in JSON format.
- The publish branch is auto-generated when you click "Publish" in ADF Studio. ADF converts pipeline definitions from JSON to ARM templates and stores them here.
- Only the contents of the publish branch are deployed and executed in the live ADF environment.

How do you manage conflicts and changes when multiple developers work on the same pipeline?

- Use feature branches for each developer to work independently and avoid overwriting each other's changes.
- Create pull requests (PRs) to merge feature branches into the collaboration branch after code review.
- Resolve merge conflicts in the PR process using DevOps or GitHub before publishing changes.
- Use clear naming conventions and documentation for pipelines, datasets, and linked services to reduce conflicts.
- Regularly sync and pull updates from the collaboration branch to keep everyone's local changes up to date.

Scenario: 46
You are migrating ETL workloads from SSIS to Azure Data Factory.

How would you lift and shift existing SSIS packages to ADF?

- First, move SSIS packages to Azure SQL Database or Azure SQL Managed Instance.
- Set up an Integration Runtime (IR) of type Azure-SSIS IR within ADF, which supports running SSIS packages in Azure.
- Deploy packages using SSMS (SQL Server Management Studio) or Azure Data Studio into the SSISDB catalog in the cloud.
- Use the Execute SSIS Package activity in a pipeline to invoke the packages.

What are the prerequisites for executing SSIS packages in Azure Data Factory?

- A provisioned Azure-SSIS Integration Runtime in ADF.
- SSIS packages stored in a deployed SSISDB on Azure SQL DB/MI or file system.
- Necessary linked services (for Azure SQL, storage, etc.) and authentication setup.
- Optional: Custom setup for third-party components using custom SSIS IR images or script setup.

How do you manage and monitor SSIS package execution within ADF?

- Monitor package executions via the ADF Monitoring tab, where SSIS activity status, start/end times, and outputs are logged.
- Use SSMS to connect to the SSISDB and review detailed execution reports and logs.
- Configure Azure Monitor or Log Analytics to track pipeline and SSIS activity outcomes.
- Set up alerting mechanisms (e.g., Logic Apps, email) for failed or long-running packages.

Scenario: 47
You are required to process change data capture (CDC) records from a SQL source to a destination.

How would you implement CDC in ADF pipelines to process only changed records?

- Enable Change Data Capture or Change Tracking on the source SQL Server or Azure SQL Database tables.
- Use Mapping Data Flows or Lookup + Filter activities to retrieve and process only the new or changed records based on change metadata (e.g., __$start_lsn, last_modified_date, or version columns).
- Implement watermarking or use a control table to store the last processed change timestamp or LSN (Log Sequence Number).
- In each pipeline run, fetch only records newer than the last stored watermark.

What ADF features or connectors support change tracking or CDC integration?

- Azure SQL Database, SQL Server, and Azure Synapse Analytics support CDC/Change Tracking when enabled at the source.
- Use Copy Activity or Mapping Data Flows to extract and transform incremental data.
- ADF supports Delta Loading patterns via stored procedures or manually constructed queries with filters.
- Integration with Azure Data Explorer, Azure Stream Analytics, or Event Grid can further enhance real-time CDC scenarios.

How would you handle late-arriving or out-of-order changes?

- Maintain a staging layer where all incremental records are landed before final merge.
- Use Merge logic in Mapping Data Flows or stored procedures to upsert data into the target system based on primary keys and timestamps.
- Retain a buffer or lag period (e.g., 24 hours) for processing to allow time for late data to arrive.
- Periodically reconcile the source and target to catch missed changes or reorder incorrectly applied updates.

Scenario: 48
Your data processing involves calling REST APIs to fetch external data before transformation.

How would you use ADF to connect and call REST APIs dynamically?

- Use the Web Activity to initiate REST API calls for triggering or small payload data.
- For larger data loads, use Copy Activity with a REST linked service as the source.
- Parameterize the Base URL, headers, and query parameters in the linked service or dataset for dynamic control.
- Use pipeline parameters and dynamic content expressions to construct URLs dynamically based on inputs like dates or resource identifiers.

How can you handle pagination and authentication when calling REST APIs in ADF?

- Handle pagination using the Pagination rules in the REST dataset, such as AbsoluteUrl, NextPage, or query-based patterns (?page=2).
- For authentication, configure the REST linked service with:
    - OAuth2 (using Azure Active Directory or custom token endpoints)
    - Basic Authentication
    - API Key (passed in headers or query string)
- If the token needs to be refreshed dynamically, use a Web Activity to call the token endpoint and store the access token in a pipeline variable for subsequent API calls.

How would you transform the API response into a structured format for storage?

- Use Mapping Data Flows to parse and flatten nested JSON from the API response.
- Alternatively, write the raw API response to Azure Blob Storage as JSON, and then use Data Flow or Azure Databricks to transform it.
- Use the Flatten transformation in Data Flows to convert hierarchical structures into tabular format.
- Map the flattened data into a sink like Azure SQL Database, Azure Synapse, or a structured file format like Parquet or CSV for downstream use.

Scenario: 49
You need to migrate large datasets from an on-premises data center to Azure in a secure and scalable way.

What options does ADF provide for securely accessing on-prem data?

- Use Self-hosted Integration Runtime (SHIR) to securely connect to on-premises data sources like SQL Server, Oracle, or file systems.
- SHIR acts as a bridge between Azure and your private network without exposing the network to the public internet.
- Data is transferred over an outbound-only HTTPS connection, which eliminates the need to open inbound firewall ports.
- Azure Private Link or VPN Gateway can be used to create a private and encrypted path for data movement.

How do you configure and use a Self-hosted Integration Runtime for large transfers?

- Install the SHIR on a high-performance on-premises server or cluster of machines.
- For large data volumes, scale out SHIR by setting up a SHIR cluster (multiple nodes) to increase throughput and parallelism.
- Use Copy Activity with partitioning options (e.g., source table partitioning, range-based chunking) to split data for parallel execution.
- Enable compression during transfer to reduce payload size and accelerate data movement.

How would you monitor and scale your data movement for optimal performance?

- Monitor SHIR performance using the ADF Monitoring tab and Integration Runtime diagnostics.
- Use Azure Monitor, Log Analytics, and activity run history to track data volumes, duration, and failures.
- Configure concurrency and data integration units (DIUs) in Copy Activity to optimize performance.
- Automate scale-up or fallback mechanisms using Azure Automation or alert-based pipelines if performance thresholds are breached.
- For extremely large migrations, consider combining ADF with Azure Data Box for initial bulk transfers, followed by incremental syncs via SHIR.

Scenario: 50
You are building a reusable pipeline template that other teams can parameterize and deploy.

How would you design a reusable, parameterized pipeline in ADF?

- Create a generic pipeline using parameters for dynamic values like file names, table names, folder paths, and column mappings.
- Use parameterized datasets and linked services to support dynamic data sources and destinations.
- Apply expressions in activity properties to reference parameters (e.g., @pipeline().parameters.SourcePath).
- Build logic using If Condition, Switch, or Lookup + ForEach to enable conditional and flexible processing.
- Use modular pipeline design by separating logic into multiple small pipelines and invoking them with the Execute Pipeline activity.

What are best practices for using pipeline parameters, datasets, and linked service parameters?

- Use clear and consistent naming conventions for parameters to improve readability and reduce confusion.
- Validate parameter inputs using expressions or Lookup activities to prevent unexpected failures.
- Keep default values for common parameters to simplify pipeline usage for consumers.
- Use dataset parameters to define dynamic file paths, table names, or schema values, especially in Copy or Data Flow activities.
- Create parameterized linked services (e.g., SQL Server with dynamic database name or Blob storage with dynamic container) for better reusability across environments.
- Store environment-specific values like credentials or connection strings in Azure Key Vault and reference them in linked services.

How can you publish and share pipeline templates across different teams or environments?

- Use Git integration in ADF to store pipeline templates in a source-controlled repository (e.g., Azure Repos or GitHub).
- Use ARM templates (Azure Resource Manager templates) to export and deploy reusable pipelines across environments.
- Package pipelines, datasets, and linked services into template repositories or shared Git branches for other teams to clone or reference.
- Utilize ADF's template gallery feature for common or organization-approved patterns.
- Document the purpose, parameters, usage instructions, and example configurations for each pipeline to ease adoption by other teams.