BONUS QUESTIONS-Task-3

1.Why is storing cleaned data in Azure Blob Storage important for real-time pipelines?

Storing cleaned data in Azure Blob Storage provides a central, scalable, and reliable data lake that downstream systems can access in real time. Raw data often contains duplicates, missing values, and inconsistencies, which can slow down analytics and lead to inaccurate insights. By cleaning the data first and storing it in Blob Storage, you create a "single source of truth" that can be consumed by multiple real-time systems such as Azure Databricks, Azure Synapse, or machine learning models. Blob Storage also supports high availability, durability, and geo-redundancy, which makes the cleaned dataset immediately accessible to different services and applications across regions. In short, it reduces data latency, improves trust in analytics, and avoids repeating the same cleaning steps in every downstream pipeline.

2.What's the difference between pipeline artifacts and Blob Storage uploads?

Although both are storage mechanisms, they serve different purposes:

- Pipeline Artifacts (Azure DevOps):
    - Used mainly for short-term, internal storage within DevOps pipelines.
    - They capture the outputs of a job or stage (e.g., generated CSVs, logs, build outputs).
    - Artifacts are designed to be shared between jobs in the same pipeline or downloaded by developers for debugging.
    - They are temporary and not meant for large-scale production data sharing.
- Azure Blob Storage:
    - A cloud-native storage solution for long-term, scalable data persistence.
    - Accessible globally via APIs, SDKs, or connectors by multiple systems (not just Azure DevOps).

- Ideal for production data pipelines, machine learning models, analytics, and business reporting.
- Provides features like access tiers (Hot, Cool, Archive), lifecycle policies, and integration with event-driven systems (e.g., Event Grid).

3. How would you handle failures in file uploads in a production setup?

In production, file uploads must be resilient, fault-tolerant, and observable. Several strategies can be applied:

1. Retry Policies with Backoff
    - Implement automatic retries (e.g., 3–5 attempts) with exponential backoff in case of network glitches or temporary Azure service outages.
2. Idempotent Uploads
    - Use overwrite=True (like in your script) to make uploads safe even if retried, ensuring no duplicates.
3. Exception Handling & Logging
    - Capture errors (e.g., authentication issues, missing files) and log them to monitoring systems like Azure Monitor, Application Insights, or a logging framework.
4. Alerts & Monitoring
    - Set up alerts for failed uploads, so the DevOps or data engineering team is notified immediately.
5. Fallback or Dead-letter Strategy
    - In case repeated failures occur, move problematic files to a “dead-letter” container for manual investigation without blocking the pipeline.

6. Data Validation Before Upload
    - Verify file size, schema, and completeness before uploading, to prevent bad data from reaching production.