On-premises to Databricks data migration architecture

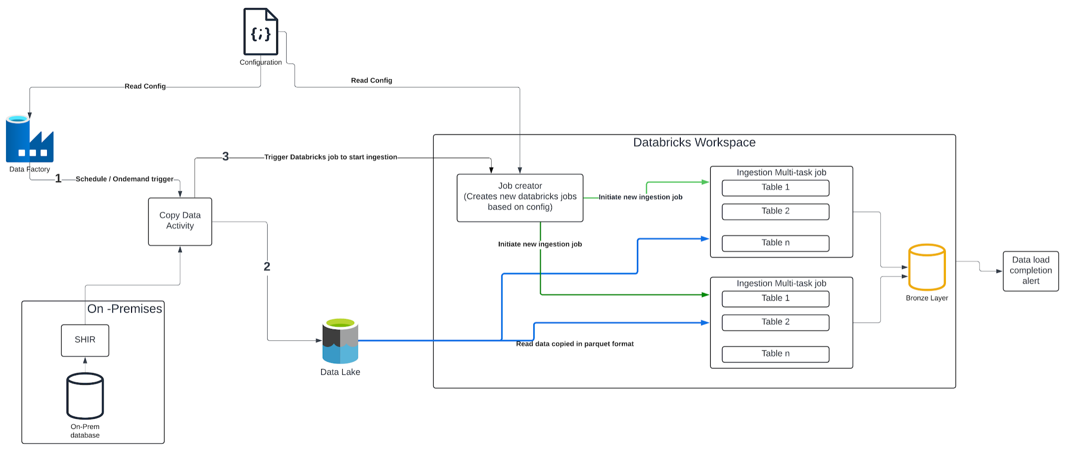
Author - Sandesh Daddi

Overview

On-premises to cloud migrations often involve significant effort in moving historical data. Our client's Oracle to Databricks project required migrating historical data from on-premises Oracle databases to the Databricks platform. This involved over 200+ tables with varying data volumes. Due to data sensitivity, the client mandated against third-party tools and preferred a solution that kept data within their network.

To address these constraints, we developed the following approach

Architecture


Configuration
Read Config
Read Config
Data Factory
1
Schedule / Ondemand trigger
3
Trigger Databricks job to start ingestion
Copy Data Activity
2
On -Premises
SHIR
On-Prem database
Data Lake
Databricks Workspace
Job creator (Creates new databricks jobs based on config)
Initiate new ingestion job
Initiate new ingestion job
Read data copied in parquet format
Ingestion Multi-task job
Table 1
Table 2
Table n
Ingestion Multi-task job
Table 1
Table 2
Table n
Bronze Layer
Data load completion alert

Leveraging the client's existing Azure infrastructure, we utilize Azure Data Factory (ADF) with a Self-Hosted Integration Runtime (SHIR) to bridge on-premises and cloud environments.

Pre-requisites

1. **SHIR deployment**: Install Self hosted integration runtime on a Virtual Machine with connectivity to on-premises data sources.

2. **Storage Account Setup:** Configure an Azure Storage account accessible from both ADF and Databricks.

High level steps

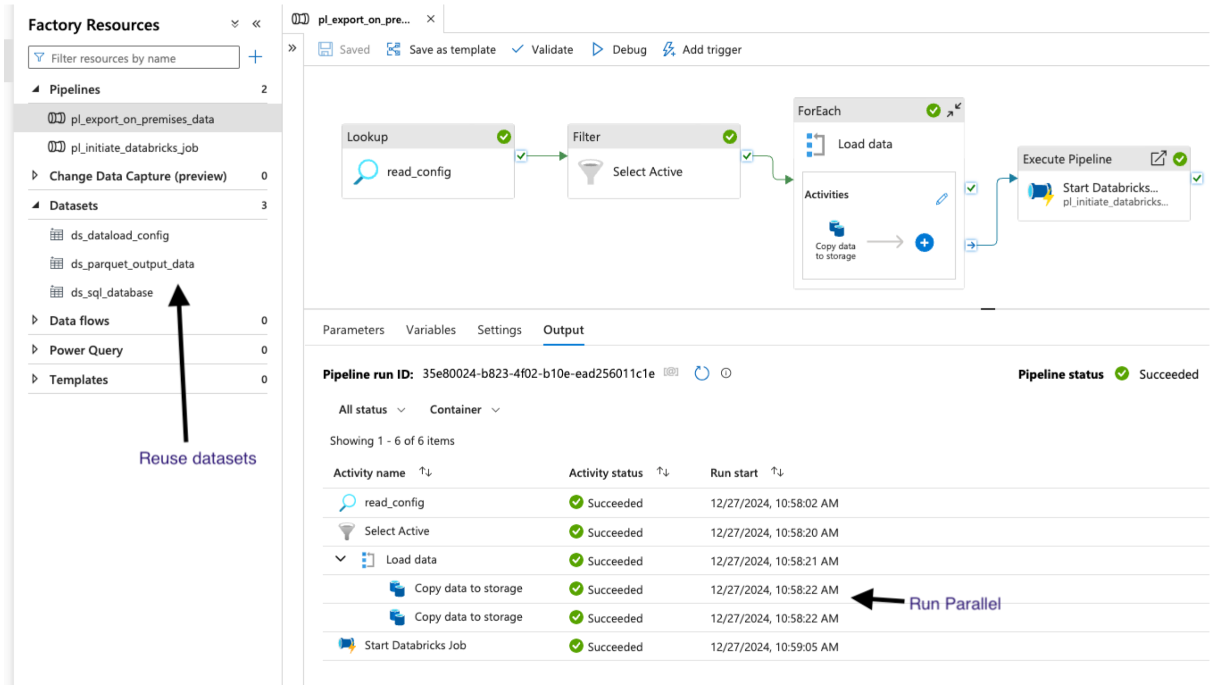
1. Configuration Management:
    a. Create a configuration file with source and destination table details.
    b. Store configuration in a shared storage account which is accessible from ADF as well from databricks.
2. ADF Pipeline Orchestration:
    a. Trigger the ADF pipeline, passing the configuration file path as a parameter.
    b. The pipeline dynamically generates and executes multiple parallel copy data activities.
    c. These activities extract data from on-premises sources and store them in Parquet format within the shared storage account.
    d. Parallelism is controlled to minimize load on both source systems and SHIR.
3. Databricks Job Initiation:
    a. Upon successful data extraction, the ADF pipeline invokes the Databricks API to trigger a "job_creator" job.
4. Databricks Job Creation:
    a. The "job_creator" job reads the configuration file and dynamically creates a new Databricks multi-task job.
    b. This multi-task job parallelizes the loading of exported data from the shared storage account into the Databricks bronze layer.

Sample Configuration File

(can be extended to include multiple other settings)

```
[
  {
    "source_database": "edw",
    "source_table": "customer",
    "source_filter_condition": "",
    "target_catalog": "uson",
    "target_schema": "bz_edw",
    "target_table": "customer",
    "isActive": "true",
    "group_id" : "edw_group_1"
  },
  {
    "source_database": "rpt",
    "source_table": "orders",
    "source_filter_condition": " WHERE TRUNC(created_date) = TRUNC(SYSDATE - 1);",
    "target_catalog": "uson",
    "target_schema": "bz_rpt",
    "target_table": "orders",
    "isActive": "true",
    "group_id" : "edw_group_2"
  }
]
```

ADF Pipeline



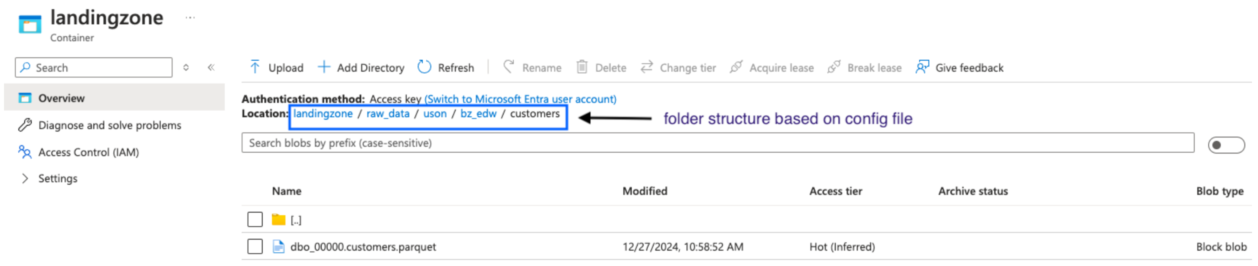
Pipeline run ID: 35e80024-b823-4f02-b10e-ead256011c1e

Pipeline status: Succeeded

Showing 1 - 6 of 6 items

Activity name	Activity status	Run start
read_config	Succeeded	12/27/2024, 10:58:02 AM
Select Active	Succeeded	12/27/2024, 10:58:20 AM
Load data	Succeeded	12/27/2024, 10:58:21 AM
Copy data to storage	Succeeded	12/27/2024, 10:58:22 AM
Copy data to storage	Succeeded	12/27/2024, 10:58:22 AM
Start Databricks Job	Succeeded	12/27/2024, 10:59:05 AM

Output in storage account



Authentication method: Access key (Switch to Microsoft Entra user account)
Location: landingzone / raw_data / uson / bz_edw / customers

Name	Modified	Access tier	Archive status	Blob type
[..]				
dbo_00000.customers.parquet	12/27/2024, 10:58:52 AM	Hot (Inferred)		Block blob

The data reading process can be optimized based on the specific source system. For instance, techniques like physical table partitions or dynamic range partition properties within the copy data activity can be leveraged to enhance performance.

For detailed guidance on executing Databricks jobs from Azure Data Factory, refer to this comprehensive blog post:

https://techcommunity.microsoft.com/blog/analyticsonazure/leverage-azure-databricks-jobs-orchestration-from-azure-data-factory/3123862

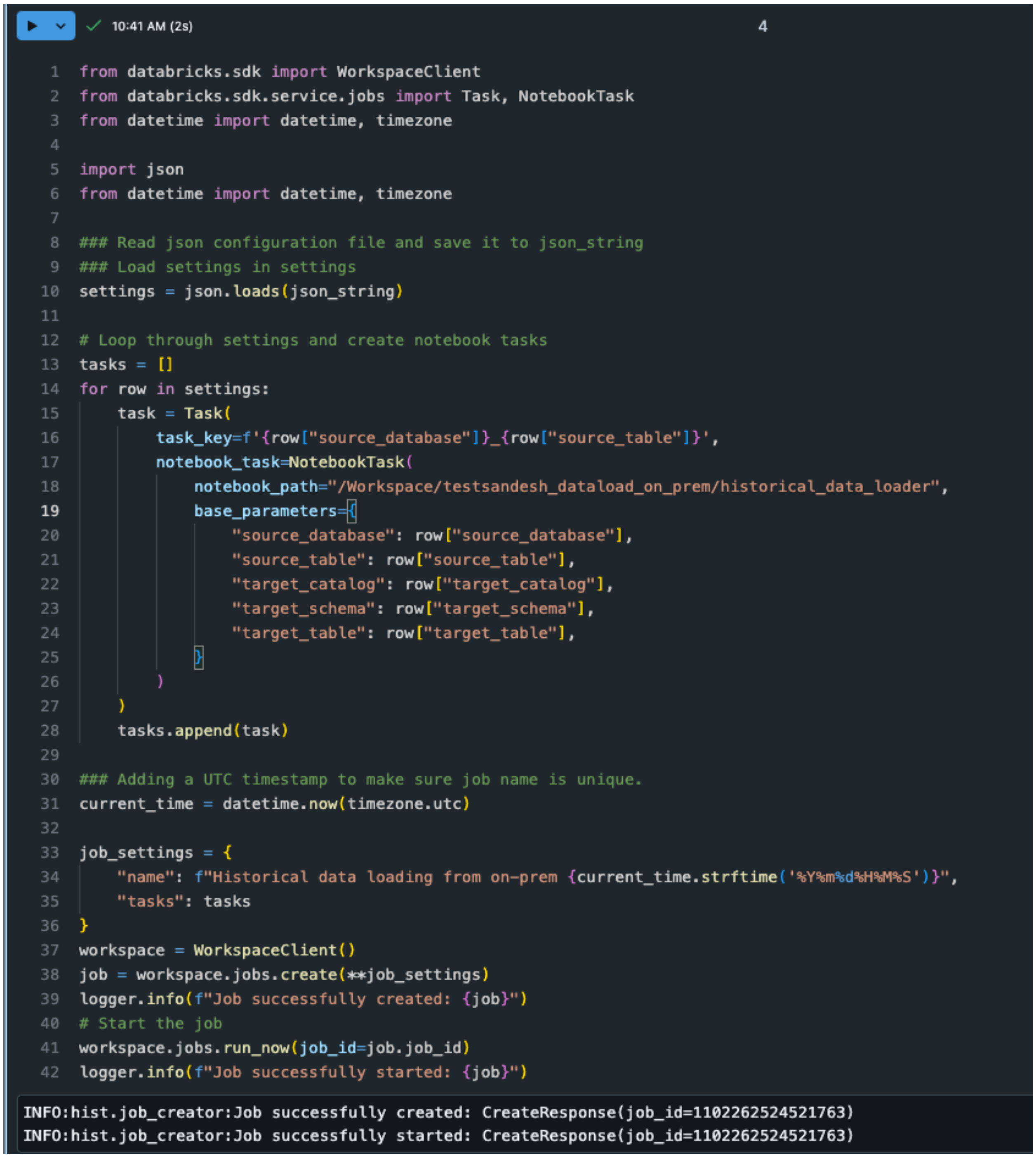
This blog post offers a step-by-step approach to constructing a modular ADF pipeline that can execute any Databricks job using built-in ADF activities and managed identity authentication. It also covers integrating the ADF Managed Identity as a contributor within your Databricks workspace.

Upload to databricks

Once the data is available in the storage account as Parquet files, Databricks is utilized to load it. As depicted in the architecture diagram, two workflows are involved: job_creator and ingestion_multitask_job.

job_creator: This job, triggered by the ADF pipeline via the JOBs API with a JSON configuration file path as input, dynamically creates a new ingestion_multitask_job. This process leverages the Databricks SDK to generate and execute notebook tasks, enabling parallel data loading for multiple tables.

Job creator code:

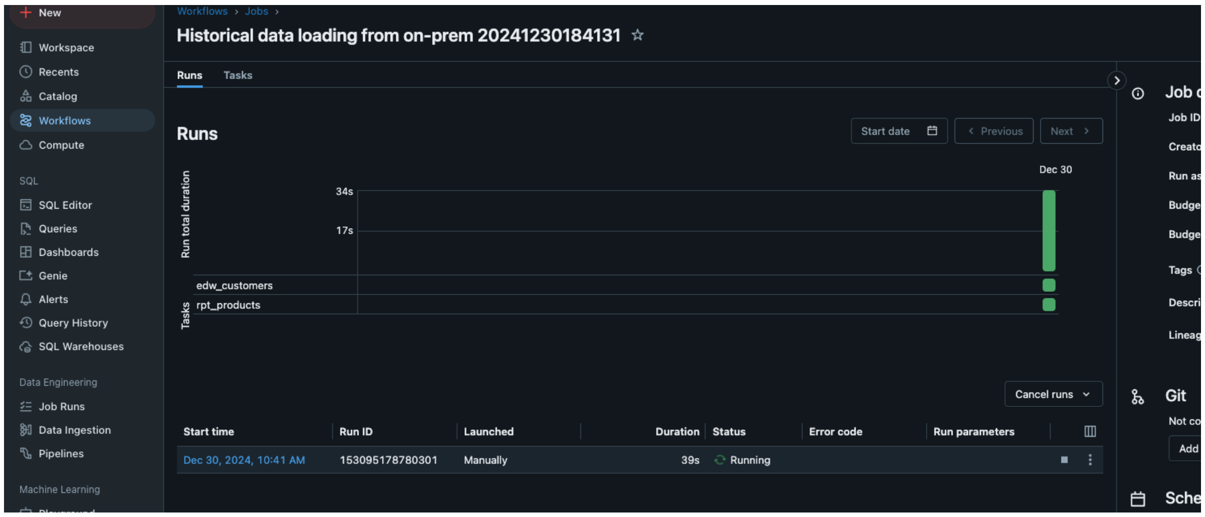


```
10:41 AM (2s)                                   4
```

```python
from databricks.sdk import WorkspaceClient
from databricks.sdk.service.jobs import Task, NotebookTask
from datetime import datetime, timezone

import json
from datetime import datetime, timezone

### Read json configuration file and save it to json_string
### Load settings in settings
settings = json.loads(json_string)

# Loop through settings and create notebook tasks
tasks = []
for row in settings:
    task = Task(
        task_key=f'{row["source_database"]}_{row["source_table"]}',
        notebook_task=NotebookTask(
            notebook_path="/Workspace/testsandesh_dataload_on_prem/historical_data_loader",
            base_parameters={
                "source_database": row["source_database"],
                "source_table": row["source_table"],
                "target_catalog": row["target_catalog"],
                "target_schema": row["target_schema"],
                "target_table": row["target_table"],
            }
        )
    )
    tasks.append(task)

### Adding a UTC timestamp to make sure job name is unique.
current_time = datetime.now(timezone.utc)

job_settings = {
    "name": f"Historical data loading from on-prem {current_time.strftime('%Y%m%d%H%M%S')}",
    "tasks": tasks
}
workspace = WorkspaceClient()
job = workspace.jobs.create(**job_settings)
logger.info(f"Job successfully created: {job}")
# Start the job
workspace.jobs.run_now(job_id=job.job_id)
logger.info(f"Job successfully started: {job}")
```

```
INFO:hist.job_creator:Job successfully created: CreateResponse(job_id=1102262524521763)
INFO:hist.job_creator:Job successfully started: CreateResponse(job_id=1102262524521763)
```

Once ADF executes this job it will create a multi task job as below.



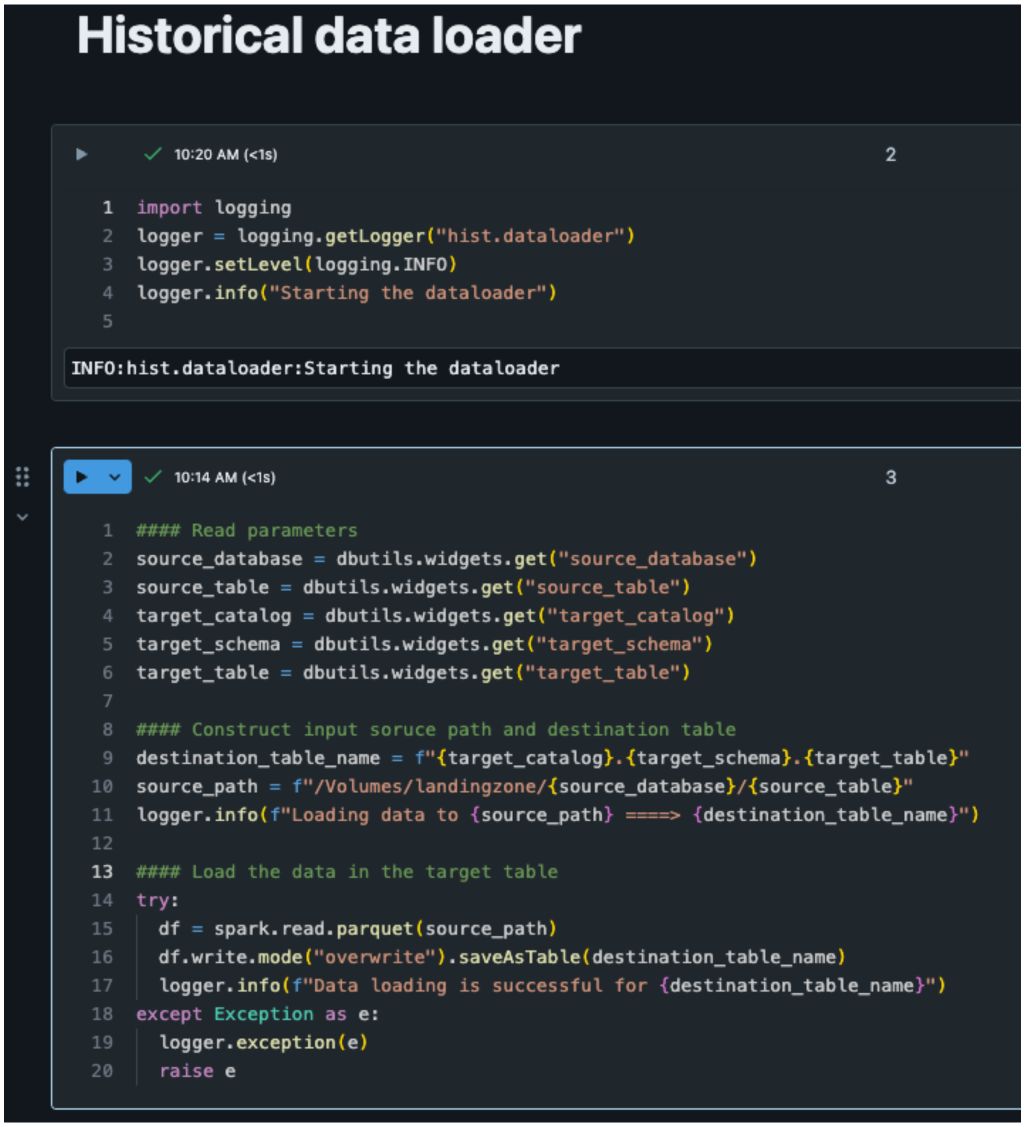
Historical data loading from on-prem 20241230184131

Runs

Start time	Run ID	Launched	Duration	Status	Error code	Run parameters
Dec 30, 2024, 10:41 AM	153095178780301	Manually	39s	Running		

The provided JSON configuration specifies two tables: customers and products. Consequently, the Dataloader notebook job includes two tasks, each responsible for loading data into the corresponding table.

The following code snippet within the Dataloader notebook demonstrates the basic logic for reading Parquet files and loading them into the respective destination tables.



Historical data loader

```
import logging
logger = logging.getLogger("hist.dataloader")
logger.setLevel(logging.INFO)
logger.info("Starting the dataloader")

```

```
INFO:hist.dataloader:Starting the dataloader
```

```
#### Read parameters
source_database = dbutils.widgets.get("source_database")
source_table = dbutils.widgets.get("source_table")
target_catalog = dbutils.widgets.get("target_catalog")
target_schema = dbutils.widgets.get("target_schema")
target_table = dbutils.widgets.get("target_table")

#### Construct input soruce path and destination table
destination_table_name = f"{target_catalog}.{target_schema}.{target_table}"
source_path = f"/Volumes/landingzone/{source_database}/{source_table}"
logger.info(f"Loading data to {source_path} ====> {destination_table_name}")

#### Load the data in the target table
try:
  df = spark.read.parquet(source_path)
  df.write.mode("overwrite").saveAsTable(destination_table_name)
  logger.info(f"Data loading is successful for {destination_table_name}")
except Exception as e:
  logger.exception(e)
  raise e
```

This project successfully demonstrated a robust and secure method for migrating historical data from on-premises Oracle databases to the Databricks platform on Azure. By leveraging Azure Data Factory with a Self-Hosted Integration Runtime, we achieved a controlled and efficient data extraction process while adhering to the client's requirement for on-premises data access and security.

The implementation of a dynamic configuration-driven approach, combined with the parallel execution capabilities of both ADF and Databricks, significantly accelerated the migration process. This solution provides a scalable and flexible framework for future data migrations and ongoing data integration between on-premises and cloud environments