AES Fabric ETL User Guide

Version 1.0

Contact: support@assuranceeservices.com
Website: www.assuranceeservices.com

Table of Contents

1. Overview
2. Architecture
3. Prerequisites
4. Installation & Setup
5. Core Components
6. Configuration
7. Usage Guide
8. Command Types
9. Advanced Features
10. Monitoring & Logging
11. Troubleshooting
12. Best Practices
13. API Reference

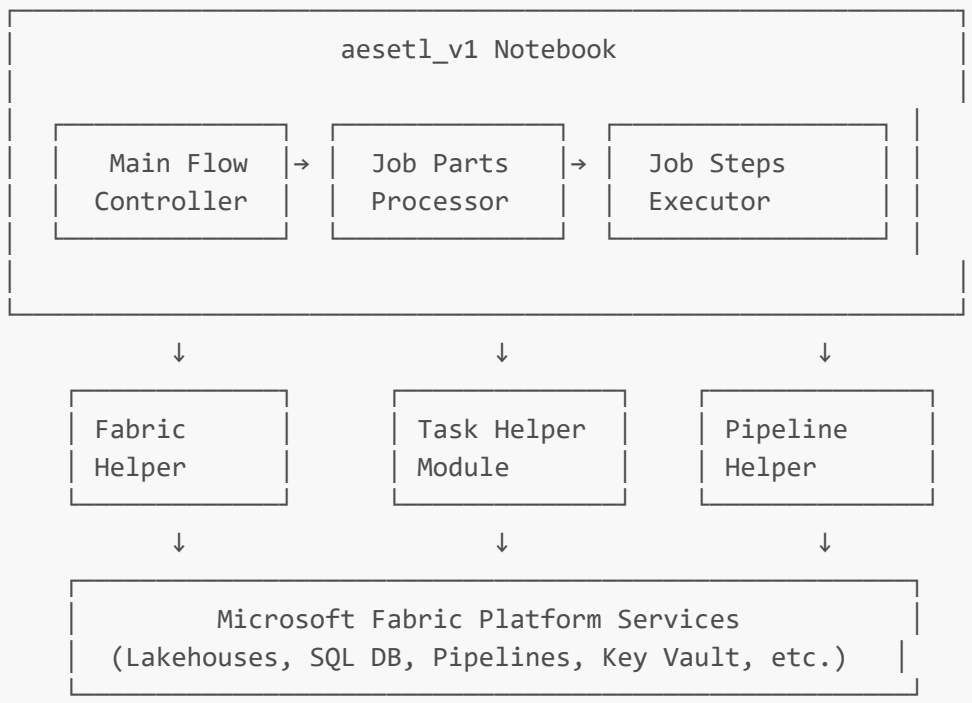
Overview

AES Fabric ETL (`aesetl_v1`) is a metadata-driven ETL orchestration framework designed for Microsoft Fabric. It provides enterprise-grade capabilities for managing complex data pipelines with features including:

- **Metadata-driven execution**: Configure jobs via SQL database tables
- **Parallel processing**: Execute multiple tasks concurrently using ThreadPoolExecutor
- **Incremental loading**: Track and manage data extraction with key value ranges
- **Multiple command types**: Support for MSSQL, Spark SQL, Python, data files, and pipelines
- **Variable substitution**: Dynamic parameter replacement with Azure Key Vault integration
- **Comprehensive logging**: Thread-safe lakehouse-based logging with job auditing
- **Error handling**: Robust error management with configurable failure policies
- **Resume capability**: Prevent duplicate execution of running jobs

Architecture

High-Level Design



Component Structure

```
aesetl_v1/
├── aesetl_v1.ipynb           # Main orchestration notebook
├── builtin/
│   └── scripts/
│       ├── fabric_helper.py      # Fabric platform utilities
│       ├── lakehouse_helper.py   # Lakehouse operations
│       ├── pipeline_helper.py    # Pipeline management
│       └── task_helper.py        # Task execution functions
├── dependencies/
│   ├── Environment/
│   └── Lakehouses/
└── env/
```

Prerequisites

Platform Requirements

- **Microsoft Fabric Workspace** with appropriate capacity
- **Fabric SQL Database** (DWInstrumentation) for instrumentation/metadata
- **Fabric Lakehouse** for data storage and logging

- **Fabric Environment** with `pymssql` library (required for SQL Server connectivity)

Important: The `pymssql` Python library is not available by default in Microsoft Fabric Runtime. You must create a Fabric Environment and add the library from external repositories (PyPI.org) in Quick mode. See Step 2: Create and Configure Environment for details.

Required Permissions

- Read/Write access to target Lakehouse(s)
- Execute, Read and Write permissions on (DWInstrumentation) SQL Database Tables and stored procedures
- Access to create/run Fabric pipelines
- Azure Key Vault access (if using secret management)

Database Schema

The instrumentation database must contain the following tables and stored procedures:

Core Tables:

- `dbo.ETLJobs` - Job definitions and status
- `dbo.ETLJobSteps` - Individual task configurations
- `dbo.ETLJobVariables` - Job and global variables
- `dbo.ETLCommandTypes` - Command type templates
- `dbo.ETLDataFileLoadDetails` - Data file load tracking (DATAFILELOAD & DATAFILEPOSTPROCESS)

Audit Tables:

- `dbo.ETLJobAudit` - Job execution audit trail (historical records of job runs)
- `dbo.ETLJobStepAudit` - Job step execution audit trail (historical records of step runs)

Stored Procedures:

- `dbo.ETLInitiateJob` - Initialize job execution
- `dbo.ETLGetJobParts` - Retrieve job parts (step groupings)
- `dbo.ETLGetJobStepsProcessNo` - Get process numbers for parallel execution
- `dbo.ETLGetJobStepsToProcess` - Retrieve steps to execute
- `dbo.ETLGetJobStepCommandToProcess` - Retrieve command XML for a specific step
- `dbo.ETLUpdateJobStepStatus` - Update step execution status
- `dbo.ETLUpdateJobStatus` - Update job execution status
- `dbo.ETLUpdateJobStepNextKeyValue` - Update incremental load key values
- `dbo.ETLGetJobVariables` - Retrieve job and global variables
- `dbo.ETLFinalizeJob` - Finalize job execution and update status
- `dbo.ETLInitiateFileAudit` - Create audit record for DATAFILELOAD operations
- `dbo.ETLGetDataFileLoadDetails` - Retrieve file load details for a step
- `dbo.ETLGetDataFilePostProcessDetails` - Retrieve file post-process details
- `dbo.ETLUpdateDataFileLoadStatus` - Update file load status
- `dbo.ETLUpdateDataFilePostProcessStatus` - Update file post-process status

Note: All commands are stored in XML format within the ETLJobSteps table.

UI Stored Procedures (Fabric ETL Control Panel):

These stored procedures are used by the Fabric ETL Control Panel application for job configuration and management:

Job Management:

- `dbo.UIAddJob` - Create a new job
- `dbo.UIUpdateJob` - Update job properties
- `dbo.UICopyJob` - Copy an existing job configuration
- `dbo.UIGetJobs` - Retrieve list of jobs
- `dbo.UIGetJobAuditDetails` - Get job execution history

Job Step Management:

- `dbo.UIAddJobStep` - Create a new job step
- `dbo.UIUpdateJobStep` - Update job step properties
- `dbo.UICopyJobStep` - Copy an existing job step
- `dbo.UIGetJobSteps` - Retrieve job steps for a job
- `dbo.UIGetJobPartName` - Get distinct job part names
- `dbo.UIGetJobStepMessage` - Get job step status message

Command Configuration:

- `dbo.UIGetCommand` - Retrieve command XML for a step
- `dbo.UIUpdateCommand` - Update command XML for a step
- `dbo.UIGetCommandTypes` - Get list of available command types
- `dbo.UIGetCommandTemplate` - Get command template XML

Next Key Command Configuration:

- `dbo.UIGetNextKeyCommand` - Retrieve next key command XML
- `dbo.UIUpdateNextKeyCommand` - Update next key command XML
- `dbo.UIGetNextKeyCommandTypes` - Get list of next key command types
- `dbo.UIGetNextKeyCommandTemplate` - Get next key command template XML

Variable Management:

- `dbo.UIAddJobVariables` - Create a new job variable
- `dbo.UIUpdateJobVariables` - Update job variable
- `dbo.UIGetJobVariables` - Retrieve job variables

Data File Tracking:

- `dbo.UIGetDataFileLoadDetails` - View file load audit details
- `dbo.UIGetDataFilePostProcessDetails` - View file post-process audit details

System:

- `dbo.UITestConn` - Test database connectivity

Installation & Setup

Step 1: Create Instrumentation Database

1. Create a Fabric SQL Database in your workspace. Recommended to name as DWInstrumentation
2. Execute the database schema scripts to create required tables and stored procedures
3. Configure job metadata (Jobs, Job Steps, Job Variables)
4. **Important**: All command configurations use XML format with CDATA sections for queries and scripts

Key Tables to Create:

- `ETLJobs` - Job definitions
- `ETLJobSteps` - Task configurations (stores XML commands)
- `ETLJobVariables` - Variables and secrets
- `ETLCommandTypes` - Reusable command templates (optional)
- `ETLJobAudit` - Job execution history
- `ETLJobStepAudit` - Job step execution history
- `ETLDataFileLoadDetails` - Data file load tracking

Step 2: Create and Configure Environment

Required: The `pymssql` library is not available by default in Fabric Runtime and must be installed via a custom Environment.

1. **Create a new Fabric Environment**:
    - Navigate to your Fabric workspace
    - Click **+ New** → **Environment**
    - Name it (e.g., "AESETLEnvironment")
2. **Add pymssql library**:
    - Open the Environment settings
    - Go to **Public Libraries** tab
    - Click **Add from PyPI**
    - Select **Quick mode**
    - Search for and add: `pymssql`
    - Click **Save** and wait for the environment to publish
3. **Attach Environment to Notebook**:
    - Open `aesetl_v1.ipynb`
    - In the notebook toolbar, select the Environment you created
    - The notebook will now have access to the pymssql library

Note: Any additional custom libraries required by your ETL jobs should also be added to this Environment.

Step 3: Deploy Notebook

1. Import `aesetl_v1.ipynb` into your Fabric workspace
2. Copy the `builtin/scripts/` folder to the notebook's resource directory

3. Attach the notebook to the Environment created in Step 2

Step 4: Configure Lakehouse

1. Create or identify the target Lakehouse for data storage
2. Create a `logs/` folder structure in the Lakehouse Files area
3. Add the Lakehouse as a dependency to the notebook

Step 5: Set Parameters

Configure the notebook parameters (Cell 2):

```
strjobname = "YourJobName"              # Job name from instrumentation DB
instrumentationdbname = "DWInstrumentation"  # SQL DB name
lakehousetolog = "lhdw"                 # Lakehouse for logging
strsessionid = "manualrun"              # Session identifier
pipelineid = "manualnotebookrun"        # Pipeline run ID
```

Core Components

1. Main Notebook (`aesetl_v1.ipynb`)

The orchestration engine that:

- Initializes job execution
- Retrieves job configuration from instrumentation database
- Manages parallel processing of job parts
- Handles error conditions and logging
- Finalizes job status

2. Helper Modules

- **Fabric Helper** (`fabric_helper.py`): Provides core Fabric platform functions
- **Lakehouse Helper** (`lakehouse_helper.py`): Manages Lakehouse operations and SCD merges
- **Task Helper** (`task_helper.py`): Executes specific command types
- **Pipeline Helper** (`pipeline_helper.py`): Manages Fabric pipeline orchestration

Configuration

Job Configuration Structure

Jobs are configured in the instrumentation database with the following hierarchy:

```
ETLJobs (Job Definition)
  ↓
ETLJobSteps (Individual tasks with logical grouping via strJobStepPartName)
```

```
  ↓
ETLJobVariables (Dynamic parameters - Global or Job-specific)
```

Execution Flow:

1. Job steps are grouped by `strJobStepPartName` Alias `Job Task Group` (e.g., "01 Copy Tables", "02 Shadow Deletes")
2. Parts execute sequentially based on their naming order
3. Within each part, **all ProcessNo values (0, 1, 2, etc.) run in parallel**
4. Within each ProcessNo, multiple tasks are executed in order based on `intProcessOrder`
5. Each step can have incremental loading via `strCurrentKeyValue` tracking

Job Metadata Tables

ETLJobs Table

```
- intJobID (PK)
- strJobName (Unique job identifier)
- strJobDescription
- strJobStatus (Current status: P=Processing, C=Complete, F=Failed, I=Inactive)
- strJobDefaultProcessType (P = Delta, F = Full)
- strJobCurrentProcessType (P = Delta, F = Full)
- dtmStartDate (Job start timestamp)
- dtmEndDate (Job end timestamp)
- strSessionID (Session/pipeline run ID)
- strLogFileName (Log file path)
- dtmUpdateDate (Last update timestamp)
- strUpdateUser (User who last updated)
```

ETLJobSteps Table

Note: Job Steps are organized by `strJobStepPartName` (executed sequentially) and `intProcessNo` (executed in parallel within each part).

Key Fields:

```
- intJobStepID (PK)
- strJobName (Job identifier)
- strJobStepName (Task name/description)
- strJobStepPartName (Logical grouping/Job Task Group - parts run sequentially)
- strJobStepStatus (Step status: P=Processing, C=Complete, F=Failed, I=Inactive)
- intProcessNo (Parallel execution group within a part - all ProcessNo values run
in parallel)
- intProcessOrder (Execution order within the same ProcessNo - lower values
execute first)
- strStartKeyValue (Initial key value for incremental loading)
- strEndKeyValue (End key value for incremental loading)
- strCommandType (Task type - see Command Types section)
```

```
- strCommand (Command XML to execute)
- strNextKeyCommand (Query to get next key value - XML format)
- strNextKeyCommandType (Query type: MSSQL, SPARKSQL, PYTHON,
INSTRUMENTATIONDBSQL)
- strCurrentKeyValue (Last successfully processed value)
- strNextKeyValue (Next key value calculated by strNextKeyCommand)
- strFailPackageonFailure (Y/N - stop job on error)
- dtmStartDate (Step start timestamp)
- dtmEndDate (Step end timestamp)
- strMessage (Status or error message)
- dtmUpdateDate (Last update timestamp)
- strUpdateUser (User who last updated)
```

ETLJobVariables Table

```
- intJobVariableID (PK)
- strJobName (Job name or 'Global' for global variables)
- strJobVariableName (Variable name)
- strJobVariableValue (Variable value)
- bitDeleted (Soft delete flag)
- dtmUpdateDate (Last update timestamp)
- strUpdateUser (User who last updated)
```

Note: Variables with `strJobName = 'Global'` are available to all jobs.

ETLJobAudit Table

This table maintains a historical record of all job executions, providing an audit trail for job runs.

```
- intJobAuditID (PK) - Unique audit record identifier
- intJobID - Foreign key to ETLJobs table
- strJobName - Job name
- strJobStatus - Job completion status (P=Processing, C=Complete, F=Failed)
- strJobDefaultProcessType - Default process type (P=Delta, F=Full)
- strJobCurrentProcessType - Process type used for this run
- dtmStartDate - Job start timestamp
- dtmEndDate - Job completion timestamp
- strSessionID - Session/pipeline run identifier
- strLogFileName - Path to log file for this run
- strMessage - Status or error message
- dtmUpdateDate - Audit record creation timestamp
- strUpdateUser - User who initiated the job
```

Purpose: Provides historical tracking of job executions for compliance, troubleshooting, and performance analysis.

ETLJobStepAudit Table

This table maintains a historical record of all job step executions, providing detailed audit trail for each task.

```
- intJobStepAuditID (PK) - Unique audit record identifier
- intJobStepID - Foreign key to ETLJobSteps table
- intJobAuditID - Foreign key to ETLJobAudit table (links to parent job run)
- strJobName - Job name
- strJobStepName - Step name/description
- strJobStepPartName - Logical grouping/Job Task Group
- strJobStepStatus - Step completion status (P=Processing, C=Complete, F=Failed,
I=Inactive)
- intProcessNo - Parallel execution group number
- intProcessOrder - Execution order within ProcessNo
- strStartKeyValue - Key value at start of execution
- strEndKeyValue - Key value at end of execution
- strCommandType - Command type executed
- strCommand - Full command XML executed
- strNextKeyCommand - Next key command XML (if applicable)
- strNextKeyCommandType - Next key command type
- strCurrentKeyValue - Last successfully processed value before this run
- strNextKeyValue - Next key value calculated during this run
- strFailPackageonFailure - Whether failure stops entire job (Y/N)
- dtmStartDate - Step start timestamp
- dtmEndDate - Step completion timestamp
- strMessage - Status or error message
- dtmUpdateDate - Audit record creation timestamp
- strUpdateUser - User who initiated the job
```

Purpose: Provides detailed execution history for troubleshooting, performance analysis, and identifying patterns in step failures or execution times.

ETLDataFileLoadDetails Table

This table tracks individual file loads for DATAFILELOAD command types, maintaining a record of all files processed.

```
- intFileLoadID (PK) - Unique file load record identifier
- intJobStepID - Foreign key to ETLJobSteps table
- intJobStepAuditID - Foreign key to ETLJobStepAudit table (links to specific
execution)
- strFileFolder - Folder path where file was stored
- strFileName - Name of the file loaded
- dtmFileDate - Date the file data represents
- strSessionID - Session/pipeline run identifier
- strStatus - File load status (P=Processing, C=Complete, F=Failed)
- strMessage - Status or error message
- dtmLoadStartDate - File load start timestamp
- dtmLoadEndDate - File load completion timestamp
- dtmCreateDate - Record creation timestamp
- strCreateUser - User who initiated the load
```

Purpose: Provides file-level audit trail for DATAFILELOAD operations, enabling tracking of which files have been processed, when, and their status. Useful for identifying missing files, reprocessing scenarios, and data quality investigations.

Variable Substitution

Variables are replaced in commands using the syntax: `$$VariableName$$`

Global Variables: Variables with `strJobName = 'Global'` are available to all jobs.

Example:

```
-- Define Global Variable:
INSERT INTO dbo.ETLJobVariables
    (strJobName, strJobVariableName, strJobVariableValue)
VALUES
    ('Global', 'CurrencyLayerAPIKey', '223978105a8e1e10d737984b69a1665e');

-- Use in command:
<datafileloadpythonscript><![CDATA[
API_KEY = "$$CurrencyLayerAPIKey$$"
BASE_URL = "http://api.currencylayer.com/historical"
]]></datafileloadpythonscript>
```

Azure Key Vault Integration

Store sensitive values in Azure Key Vault and reference them using:

```
$azkv$KeyVaultName.secret.SecretName$azkv$
```

Example:

```
-- Define variable with Key Vault reference:
INSERT INTO dbo.ETLJobVariables
    (strJobName, strJobVariableName, strJobVariableValue)
VALUES
    ('Global', 'azkv_sourcedata_connstring',
     '$azkv$kvaesetlsandbox.secret.sqlserver1-connectionstring$azkv$');

-- Use in command:
<copymssql2lakehouse>
  <sqlconnectionstringtype>connectstring</sqlconnectionstringtype>
  <sqlconnectstring><![CDATA[$$azkv_sourcedata_connstring$$]]></sqlconnectstring>
  ...
</copymssql2lakehouse>
```

Supported Secret Types:

- `secret` - Key Vault Secret
- Additional types may be configured as needed

Usage Guide

Running a Job Manually

1. **Open the notebook**: `aesetl_v1.ipynb`
2. **Set parameters** (Cell 2):

```
strjobname = "DWCurrencyRates"
instrumentationdbname = "DWInstrumentation"
lakehousetolog = "lhdw"
strsessionid = "manualrun"
pipelineid = "manualnotebookrun"
```

3. **Run all cells** or use "Run All" from the notebook menu
4. **Monitor execution**:
   - View real-time output in notebook
   - Check log files in Lakehouse: `Files/logs/{JobName}/{Timestamp}_{SessionID}.log`
   - Query instrumentation database for job status

Running from a Pipeline

1. **Create a Fabric Pipeline**
2. **Add Notebook Activity**:
   - Select `aesetl_v1` notebook
   - Configure parameters:

```
{
  "strjobname": "YourJobName",
  "instrumentationdbname": "DWInstrumentation",
  "lakehousetolog": "lhdw",
  "strsessionid": "@{pipeline().RunId}",
  "pipelineid": "@{pipeline().RunId}"
}
```

3. **Run the pipeline**

Command Types

The framework supports multiple command types, each using **XML format** for configuration. Commands are stored in ETLJobSteps with XML structure wrapped in CDATA sections for SQL queries and Python scripts.

Command Templates

Command templates can be stored in the `ETLCommandTypes` table for reusability:

```
-- ETLCommandTypes Table Structure
- strCommandType (PK) - Command type identifier
- strCommandTemplate - XML template for the command
- strNextKeyCommandTemplate - XML template for next key calculation
- bitActive - Enable/disable command type
- dtmUpdateDate - Last update timestamp
- strUpdateUser - User who last updated
```

XML Format Guidelines

- All commands use XML structure
- SQL queries and Python scripts are wrapped in `<![CDATA[...]]>` sections
- Variable substitution uses `$$VariableName$$` syntax
- Key value placeholders use `@strStartKeyValue` and `@strEndKeyValue`
- Connection strings can reference gateways or use direct connections

---

1. COPYMSSQL2LAKEHOUSE

Purpose: Copy data from SQL Server to Lakehouse with Type 1 or Type 2 SCD merge

Command Format (XML):

```
<copymssql2lakehouse>
  <sqlconnectionstringtype>gateway</sqlconnectionstringtype>
  <sqlconnectstring>sqlserver1</sqlconnectstring>
  <sqlquery><![CDATA[
    SELECT *, ModifiedDate AS LastUpdateDate
    FROM [AdventureWorks].[SalesLT].[Product]
    WHERE ModifiedDate >= '@strStartKeyValue'
    AND ModifiedDate <= '@strEndKeyValue'
  ]]></sqlquery>
  <stagingtablename>lhdw.dwstg.adventureworks_saleslt_product</stagingtablename>

<destinationtablename>lhdw.dw.adventureworks_saleslt_product</destinationtablename
>
  <destinationtablemergetype>type2</destinationtablemergetype>
  <destinationtablemergetype2datepart>SECOND</destinationtablemergetype2datepart>
  <sourcetableprimarykeycolumns>productid</sourcetableprimarykeycolumns>
  <desttablealternatekeycolumns>productid</desttablealternatekeycolumns>
  <desttablecolumnsignorechanges></desttablecolumnsignorechanges>
  <convertschematolowercase>true</convertschematolowercase>
  <batchsize>100000</batchsize>
```

```
  <refreshendpoint>true</refreshendpoint>
</copymssql2lakehouse>
```

**Connection String Types (sqlconnectionstringtype) **:

- **gateway**: References a gateway name (e.g., "sqlserver1")
- **connectstring**: Direct connection string (supports variable substitution)

Destination Table Merge Type (destinationtablemergetype):

- **type1**: Performs Type 1 SCD (overwrites existing records)
- **type2**: Performs Type 2 SCD (maintains full history)
- **type2datepart**: Performs Type 1 until the record update date passes the specified datepart threshold, then performs Type 2
  - For `destinationtablemergetype = type2datepart`, the parameter **destinationtablemergetype2datepart** must be configured
  - Possible values: `SECOND`, `MINUTE`, `HOUR`, `DAY`, `MONTH`, `YEAR`

Example with Azure Key Vault Connection String:

```
<copymssql2lakehouse>
  <sqlconnectionstringtype>connectstring</sqlconnectionstringtype>
  <sqlconnectstring><![CDATA[$$azkv_sourcedata_connstring$$]]></sqlconnectstring>
  ...
</copymssql2lakehouse>
```

NextKeyCommand Example (XML):

```
<mssql>
  <sqlconnectionstringtype>gateway</sqlconnectionstringtype>
  <sqlconnectstring>sqlserver1</sqlconnectstring>
  <sqlquery><![CDATA[
    SELECT MAX(CONVERT(VARCHAR(23), ModifiedDate, 121)) AS strNextKeyValue
    FROM [AdventureWorks].[SalesLT].[Product]
  ]]></sqlquery>
  <lakehouse>lhdw</lakehouse>
</mssql>
```

2. PYTHON

Purpose: Execute custom Python code

Command Format (XML):

```
<python>
  <pythonscript><![CDATA[
from datetime import datetime, timedelta, timezone
```

```
try:
    # Available values to use in the context (alogfile_abfs_path, spark)
    # Your code goes below here

    # Sample code to return yesterday's date as strNextKeyValue
    yesterday_utc = datetime.now(timezone.utc) - timedelta(days=1)
    formatted_utc = yesterday_utc.strftime("%Y-%m-%d")

    # Your transformation logic here
    df = spark.table("dwstg.SourceTable")
    df_transformed = df.withColumn("NewColumn", current_timestamp())
    df_transformed.write.mode("overwrite").saveAsTable("dw.TargetTable")
except Exception as e:
    raise
  ]]></pythonscript>
</python>
```

NextKeyCommand Format (XML):

```
<python>
  <pythonscript><![CDATA[
from datetime import datetime, timedelta, timezone
try:
    yesterday_utc = datetime.now(timezone.utc) - timedelta(days=1)
    formatted_utc = yesterday_utc.strftime("%Y-%m-%d")
    scriptresult = {"status":"success", "data":{"strNextKeyValue":formatted_utc}}
except Exception as e:
    scriptresult = {"status":"error", "error":str(e)}
  ]]></pythonscript>
</python>
```

3. SPARKSQL

Purpose: Execute Spark SQL statements (supports parallel execution)

Command Format (XML):

```
<sparksql>
  <sqlcommand><![CDATA[
    OPTIMIZE lhdw.dw.adventureworks_saleslt_product ZORDER BY (_iscurrent,
productid);
    OPTIMIZE lhdw.dw.adventureworks_saleslt_address ZORDER BY (_iscurrent,
addressid);
    OPTIMIZE lhdw.dw.adventureworks_saleslt_productcategory ZORDER BY (_iscurrent,
productcategoryid);
    OPTIMIZE lhdw.dw.adventureworks_saleslt_customer ZORDER BY (_iscurrent,
customerid);
  ]]></sqlcommand>
  <executeparallel>true</executeparallel>
```

```
  <maxparallelism>4</maxparallelism>
</sparksql>
```

Single Statement Example:

```
<sparksql>
  <sqlcommand><![CDATA[
    CREATE TABLE IF NOT EXISTS dw.FactSales
    USING DELTA
    AS
    SELECT
        s.SalesID,
        s.ProductID,
        s.CustomerID,
        s.SalesAmount,
        s.SalesDate
    FROM dwstg.StagingSales s
    WHERE s.ModifiedDate BETWEEN '@strStartKeyValue' AND '@strEndKeyValue'
  ]]></sqlcommand>
</sparksql>
```

NextKeyCommand Example (XML):

```
<sparksql>
  <sqlquery><![CDATA[
    SELECT date_format(to_utc_timestamp(current_timestamp(), current_timezone()),
'yyyy-MM-dd HH:mm:ss') as strNextKeyValue;
  ]]></sqlquery>
</sparksql>
```

3a. MSSQL

Purpose: Query SQL Server database (primarily used in NextKeyCommand for incremental load key calculations)

NextKeyCommand Format (XML):

```
<mssql>
  <sqlconnectionstringtype>gateway</sqlconnectionstringtype>
  <sqlconnectstring>sqlserver1</sqlconnectstring>
  <sqlquery><![CDATA[
    SELECT MAX(CONVERT(VARCHAR(23), ModifiedDate, 121)) AS strNextKeyValue
    FROM [AdventureWorks].[SalesLT].[Product]
  ]]></sqlquery>
  <lakehouse>lhdw</lakehouse>
</mssql>
```

Alternative with Direct Connection String:

```
<mssql>
  <sqlconnectionstringtype>connectstring</sqlconnectionstringtype>
  <sqlconnectstring><![CDATA[$$azkv_sourcedata_connstring$$]]></sqlconnectstring>
  <sqlquery><![CDATA[
    SELECT MAX(CONVERT(VARCHAR(23), ModifiedDate, 121)) AS strNextKeyValue
    FROM [AdventureWorks].[SalesLT].[Address]
  ]]></sqlquery>
  <lakehouse>lhdw</lakehouse>
</mssql>
```

Note: This command type is typically used for calculating the next key value from source systems rather than as a main command for data movement.

---

4. DATAFILELOAD

Purpose: Load data files from external APIs or generate files with custom Python scripts

Command Format (XML):

```
<datafileload>
  <datafileworkspacename>aesfabricetl</datafileworkspacename>
  <datafilelakehousename>lhdw</datafilelakehousename>
  <datafilefolderpattern>exchangerates/yyyy/mm/dd</datafilefolderpattern>
  <datafilenameprefixpattern>yyyymmdd</datafilenameprefixpattern>
  <datafilename>_currencyexchangerates</datafilename>
  <datafiletype>json</datafiletype>
  <datafrequencytype>DAY</datafrequencytype>
  <datafrequency>1</datafrequency>
  <datafileloadpythonscript><![CDATA[
# Available values to use in the context (datadate, datafile, datafrequencytype,
datafrequency, spark)
# Your code goes below here

# Get currency rates from API
import requests
import time
time.sleep(1)
API_KEY = "$$CurrencyLayerAPIKey$$"
BASE_URL = "http://api.currencylayer.com/historical"

lparams = {
    "access_key": API_KEY,
    "date":  datadate[:10],
    "source": "USD",
    "format": 1
}
```

```
try:
    response = requests.get(BASE_URL, params=lparams)
    ldata = response.json()
    if ldata["success"]:
        scriptresult = {"status": "success", "data": ldata}
    else:
        scriptresult = {"status": "error", "error": ldata["error"]["info"]}
except Exception as e:
    scriptresult = {"status": "error", "error": str(e)}
  ]]></datafileloadpythonscript>
</datafileload>
```

Date Pattern Tokens:

- `yyyy` - 4-digit year
- `mm` - 2-digit month
- `dd` - 2-digit day

5. DATAFILEPOSTPROCESS

Purpose: Post-process data files loaded by DATAFILELOAD tasks

Command Format (XML):

```
<datafilepostprocess>
  <datafileloadjobtaskid>5</datafileloadjobtaskid>

<datafilepostprocessworkspacename>aesfabricetl</datafilepostprocessworkspacename>
  <datafilepostprocesslakehousename>lhdw</datafilepostprocesslakehousename>
  <datafilepostprocesspythonscript><![CDATA[
# Available values to use in the context:
# (datafilepostprocessworkspace_id, datafilepostprocesslakehouse_id, datadate,
datafile, spark)

import json
from datetime import datetime, timezone
from pyspark.sql.types import StructType, StructField, StringType, DoubleType
from delta.tables import DeltaTable
from notebookutils import mssparkutils

# Construct full ABFS path to the table
lschema_name = "dw"
ltable_name = "currency_exchange_rates"
table_abfs_path =
f"abfss://{datafilepostprocessworkspace_id}@onelake.dfs.fabric.microsoft.com/{data
filepostprocesslakehouse_id}/Tables/{lschema_name}/{ltable_name}"

scriptresult = {}
try:
    lraw = mssparkutils.fs.head(datafile, 50 * 1024 * 1024)
    lpayload = json.loads(lraw)
```

```
    if not lpayload.get("success", False):
        raise ValueError(lpayload.get("error", {}).get("info", "API payload
indicates failure."))

    lquotes = lpayload.get("quotes", {})
    ldate = lpayload.get("date")
    lingested_at = datetime.now(timezone.utc).strftime("%Y-%m-%d %H:%M:%S")

    lrows = [
        {
            "rate_date": ldate,
            "currencycode": k,
            "currencyrate": float(v),
            "_lastupdatedate": lingested_at
        }
        for k, v in lquotes.items()
    ]

    schema = StructType([
        StructField("rate_date", StringType(), True),
        StructField("currencycode", StringType(), True),
        StructField("currencyrate", DoubleType(), True),
        StructField("_lastupdatedate", StringType(), True)
    ])

    ldf_rates = spark.createDataFrame(lrows, schema=schema)

    if DeltaTable.isDeltaTable(spark, table_abfs_path):
        deltaTable = DeltaTable.forPath(spark, table_abfs_path)
        deltaTable.alias("t").merge(
            ldf_rates.alias("s"),
            "t.rate_date = s.rate_date AND t.currencycode = s.currencycode"
        ).whenMatchedUpdate(set = {
            "currencyrate": "s.currencyrate",
            "_lastupdatedate": "s._lastupdatedate"
        }).whenNotMatchedInsertAll().execute()
    else:
        ldf_rates.write.format("delta").save(table_abfs_path)

    scriptresult = {
        "status": "success",
        "data": {
            "table": table_abfs_path,
            "rows_processed": len(lrows),
            "rate_date": ldate
        }
    }
except Exception as e:
    scriptresult = {"status": "error", "error": str(e)}
  ]]></datafilepostprocesspythonscript>
</datafilepostprocess>
```

6. RUNPIPELINE

Purpose: Trigger another Fabric pipeline

Command Format (XML):

```
<runpipeline>
  <pipelinename>DMAdventureWorks</pipelinename>
  <ignoreifrunning>true</ignoreifrunning>
  <waitforcompletion>false</waitforcompletion>
  <failtaskonfailure>false</failtaskonfailure>
</runpipeline>
```

Configuration Options:

- **pipelinename**: Name of the pipeline to execute
- **ignoreifrunning**: Skip if pipeline is already running
- **waitforcompletion**: Wait for pipeline to complete before proceeding
- **failtaskonfailure**: Fail this task if pipeline fails

7. SHADOWDELETEMSSQL2LH

Purpose: Detect and propagate deletions from source to lakehouse (soft delete for Current Record)

Command Format (XML):

```
<shadowdeletemssql2lakehouse>
  <sqlgatewayname>sqlserver1</sqlgatewayname>
  <sqlquery><![CDATA[
    SELECT customerid
    FROM [AdventureWorks].[SalesLT].[Customer]
    ORDER BY customerid
  ]]></sqlquery>

<stagingtablename>lhdw.dwstg.pk__adventureworks_saleslt_customer</stagingtablename
>

<destinationtablename>lhdw.dw.adventureworks_saleslt_customer</destinationtablenam
e>
  <sourcetableprimarykeycolumns>customerid</sourcetableprimarykeycolumns>
  <desttablealternatekeycolumns>customerid</desttablealternatekeycolumns>
  <convertschematolowercase>true</convertschematolowercase>
  <batchsize>100000</batchsize>
  <refreshendpoint>true</refreshendpoint>
</shadowdeletemssql2lakehouse>
```

How it Works:

1. Retrieves current primary keys from source system

2. Compares with lakehouse table to identify deleted records
3. Marks deleted records by setting `_deleteddate` = current timestamp

Note: This is different from Type 2 SCD which uses `_enddate` and `_iscurrent` columns. Shadow delete specifically tracks deletion timestamps.

8. INSTRUMENTATIONDBSQL

Purpose: Query the instrumentation database itself

NextKeyCommand Example (XML):

```
<instrumentationdbsql>
  <sqlquery><![CDATA[
    SELECT strCurrentKeyValue AS strNextKeyValue
    FROM dbo.ETLJobSteps AS S
    WHERE intJobStepId = 5
  ]]></sqlquery>
</instrumentationdbsql>
```

Note: This command type is typically used in NextKeyCommand to reference values from other job steps or configuration tables within the instrumentation database. one usage is trigger cascade of next task.

Advanced Features

Parallel Processing

All `intProcessNo` values within a `strJobStepPartName` execute in parallel using Python's ThreadPoolExecutor. Within each ProcessNo, multiple tasks are executed sequentially based on `intProcessOrder`.

```
Job: "DWAdventureWorks"
  ├─ Part: "01 Copy Tables" (runs first)
  │   ├─ Process No 0 (runs in parallel with Process No 1 and 2)
  │   │   └─ Load Product Table
  │   ├─ Process No 1 (runs in parallel with Process No 0 and 2)
  │   │   ├─ Load Address Table (ProcessOrder 1 - runs first)
  │   │   └─ Load Customer Table (ProcessOrder 2 - runs after Address)
  │   └─ Process No 2 (runs in parallel with Process No 0 and 1)
  │       └─ Load Category Table
  └─ Part: "02 Shadow Deletes" (runs after "01 Copy Tables" completes)
      └─ Process No 0
          └─ Shadow Delete Customer
```

Key Points:

- Parts (strJobStepPartName) run sequentially based on their naming order

- **All ProcessNo values (0, 1, 2, etc.) within a part run in parallel simultaneously**
- Within each ProcessNo, multiple tasks execute sequentially based on `intProcessOrder`
- Lower `intProcessOrder` values execute before higher values within the same ProcessNo

Incremental Loading

The framework supports incremental data extraction using key value tracking:

1. **Initial Load**: `strCurrentKeyValue` is empty or default
2. **Get Next Key**: Execute `strNextKeyCommand` to determine load range
3. **Load Data**: Extract data between `@strStartKeyValue` and `@strEndKeyValue`
4. **Update Key**: Store `strNextKeyValue` as new `strCurrentKeyValue`
5. **Next Run**: Start from last successful key value

Example:

```
-- First run: Load 2024-01-01 to 2024-01-31
-- strCurrentKeyValue: '2024-01-01'
-- strNextKeyValue: '2024-01-31'

-- Second run: Load 2024-02-01 to 2024-02-28
-- strCurrentKeyValue: '2024-01-31' (from previous run)
-- strNextKeyValue: '2024-02-28'
```

Slowly Changing Dimensions (SCD)

Type 1 SCD (Overwrite)

```
# Updates existing records with new values
lakehouse_etl_merge_data_type1(
    source_table="dwstg.DimProduct",
    target_table="dw.DimProduct",
    merge_keys=["ProductID"]
)
```

Type 2 SCD (Historical Tracking)

```
# Maintains history with effective dates and current flag
lakehouse_etl_merge_data_type2(
    source_table="dwstg.DimCustomer",
    target_table="dw.DimCustomer",
    merge_keys=["CustomerID"],
    effective_date_column="EffectiveDate",
    end_date_column="EndDate",
    current_flag_column="IsCurrent"
)
```

Concurrent Execution Prevention

The framework prevents duplicate job execution:

1. Job initiation checks current job status
2. If job is already running, verifies pipeline status
3. If pipeline is truly running, exits gracefully with status "U" (Unchanged)
4. If pipeline is not running but status shows running, marks as failed for recovery

Thread-Safe Logging

All logging operations use thread locks to prevent conflicts during parallel execution:

```
_log_file_lock = threading.Lock()

with _log_file_lock:
    mssparkutils.fs.append(log_file_path, log_text, True)
```

Log File Structure

Logs are stored in the Lakehouse with the following structure:

```
Files/
└── logs/
    └── {JobName}/
        └── {Timestamp}_{SessionID}.log
```

Example: `Files/logs/DWCurrencyRates/20260427120000_pipeline_run_123.log`

Log Entry Format

```
[2026-04-27 12:00:00] [Job Task Id: 00001] Job Step Process Initiated
[2026-04-27 12:00:05] [Job Task Id: 00001] 2024-01-31 Next Key Value Generated.
[2026-04-27 12:00:10] [Job Task Id: 00001] Executing Command Type for Job Step:
```

```
COPYMSSQL2LAKEHOUSE
[2026-04-27 12:05:30] [Job Task Id: 00001] Job Step Completed Successfully
```

Copyright Notice in Logs

Each log file includes copyright and licensing information:

```
--------------------------------------------------------------------------
Copyright (c) 2024-2030 Assurance eServices Inc. All rights reserved.

This software is licensed under the Business Source License 1.1 (BSL).
- Free for community use (up to 1,000 Task Runs/month).
- Commercial License required for usage exceeding 1,000 Task Runs/month.

Change Date: 2030-06-01
After the Change Date, this software converts to the Apache License 2.0.

For commercial licensing, contact: support@assuranceeservices.com
Website: www.assuranceeservices.com
--------------------------------------------------------------------------
```

Job Status Values

- **P** - Processing: Job is currently running
- **C** - Complete: Job finished successfully
- **F** - Failed: Job encountered errors
- **U** - Unchanged: Job already running (duplicate execution prevented)
- **I** - Inactive: Job is disabled/inactive

Troubleshooting

Common Issues

1. Job Won't Start - "Already Running"

Symptom: Job exits gracefully with message in log "Job is already running"

Cause: Another instance of the job is executing

Solution:

- Check if pipeline is truly running
- Wait for current execution to complete
- If stuck, manually update job status in database:

```
UPDATE dbo.ETLJobs
SET strJobStatus = 'F', strMessage = 'Manually reset'
```

```
WHERE strJobName = 'YourJobName'
```

2. Connection Errors to SQL Database

Symptom: "Error connecting to database" or timeout errors

Cause: Network issues, incorrect connection string, or permissions

Solution:

- Verify SQL Database is accessible from Fabric workspace
- Check connection string format
- Ensure notebook has permissions to SQL Database
- Test connection using `get_mssql_connection()` in a test cell

3. Missing pymssql Library

Symptom: `ModuleNotFoundError: No module named 'pymssql'` or import errors

Cause: The `pymssql` library is not available in the default Fabric Runtime

Solution:

1. Create a Fabric Environment in your workspace
2. Add `pymssql` from PyPI in Quick mode:
    - Environment \u2192 Public Libraries \u2192 Add from PyPI
    - Search for: `pymssql`
    - Click Save and wait for environment to publish
3. Attach the Environment to your notebook
4. Restart the notebook kernel

Note: This is a required step for the framework to function correctly.

4. Lakehouse File Not Found

Symptom: "File not found" or "Path does not exist"

Cause: Incorrect file path

Solution:

- Verify Lakehouse path is correct.
- Check file path format: `Files/folder/file.csv` (not `./Files/...`)
- Use `mssparkutils.fs.ls("Files/")` to verify file existence

5. KeyVault Access Denied

Symptom: "Access denied" when retrieving secrets

Cause: Missing permissions or incorrect KeyVault name

Solution:

- Grant "Key Vault Secrets User" role to Fabric workspace identity
- Verify KeyVault name in variable value
- Test access: `get_azure_keyvault_object_value()`

6. Parallel Processing Deadlocks

Symptom: Job hangs or steps don't complete

Cause: Resource contention or circular dependencies

Solution:

- Review `intProcessNo` assignments - ensure no circular dependencies
- Reduce parallel threads by consolidating steps
- Check for table locks in Spark SQL commands

7. Incremental Load Reprocessing Same Data

Symptom: Same data loaded multiple times

Cause: `strCurrentKeyValue` not updating correctly

Solution:

- Verify `strNextKeyCommand` returns valid value
- Check `ETLUpdateJobStepNextKeyValue` stored procedure
- Manually inspect `strCurrentKeyValue` in database

8. Python Import Errors

Symptom: "ModuleNotFoundError" for helper modules

Cause: Helper scripts not in correct location or path issue

Solution:

- Verify files exist in `builtin/scripts/` folder
- Check `sys.path` includes scripts folder
- Restart notebook kernel to clear module cache

Note: This is different from issue #3 (pymssql library). This issue relates to the framework's helper modules (fabric_helper.py, lakehouse_helper.py, etc.).

Best Practices

1. Job Design

- **Logical Grouping**: Use `strJobStepPartName` (Job Task Groups) to group related operations

- **Concurrency Management**: Use `intProcessNo` to organize parallel tasks - all ProcessNo values run simultaneously
- **Sequential Execution**: Use `intProcessOrder` to control execution sequence within the same `intProcessNo`
- **Granular Steps**: Keep individual steps focused on single tasks
- **Idempotency**: Design steps to be safely re-runnable

2. Error Handling

- **Selective Failure**: Use `strFailPackageonFailure = 'N'` for non-critical steps
- **Error Messages**: Include detailed context in error messages
- **Retry Logic**: Implement retry mechanisms for transient failures
- **Validation Steps**: Add validation steps after critical operations

3. Performance Optimization

- **Parallel Execution**: Maximize parallelism where dependencies allow
- **Partitioning**: Use partition columns for large datasets
- **Incremental Loading**: Always prefer incremental over full loads
- **Resource Sizing**: Ensure adequate Spark cluster resources

4. Security

- **Key Vault**: Store all credentials and sensitive data in Azure Key Vault
- **Least Privilege**: Grant minimal required permissions
- **Audit Trail**: Maintain comprehensive job execution history
- **Data Masking**: Avoid logging sensitive data values

5. Maintenance

- **Regular Cleanup**: Archive old log files and audit records
- **Version Control**: Track changes to job configurations
- **Testing**: Test job changes in non-production environment first
- **Documentation**: Document custom command implementations

6. Monitoring

- **Fabric ETL Control Panel**: Set up alerts for failed jobs

---

API Reference

Main Functions

`writeto_lakehouselogfile(alogfile_abfs_path, astrJobStepID, atext)`

Thread-safe logging function.

Parameters:

- `alogfile_abfs_path` (str): Full ABFS path to log file

- `astrJobStepID` (str): Job step identifier for log entry
- `atext` (str): Text to append to log

Returns: dict with status and message

`run_job_step_thread(alogfile_abfs_path, aworkspaceid, aworkspacename, ainstrumentationdb_connectstring, astrjobname, astrjobpartname, aintprocessno, astrsessionid, astrlogfilename)`

Execute job steps for a specific process number in parallel.

Parameters:

- `alogfile_abfs_path` (str): Log file path
- `aworkspaceid` (str): Fabric workspace ID
- `aworkspacename` (str): Fabric workspace name
- `ainstrumentationdb_connectstring` (str): SQL database connection string
- `astrjobname` (str): Job name
- `astrjobpartname` (str): Job part name
- `aintprocessno` (int): Process number for parallel execution group
- `astrsessionid` (str): Session identifier
- `astrlogfilename` (str): Log file name

Returns: tuple (process_number, error or None)

Helper Module Functions

Refer to individual helper modules for detailed API documentation:

- **fabric_helper.py**: Fabric platform interactions
- **lakehouse_helper.py**: Lakehouse data operations
- **pipeline_helper.py**: Pipeline management
- **task_helper.py**: Task execution functions

Appendix A: Sample Job Configuration

Example 1: DWCurrencyRates - Daily Currency Exchange Rates Load

This job demonstrates API data loading with file generation and post-processing.

ETLJobs Table

```
INSERT INTO dbo.ETLJobs
    (strJobName, strJobStatus, strJobDefaultProcessType, strJobCurrentProcessType)
VALUES
    ('DWCurrencyRates', 'C', 'P', 'P');
```

ETLJobSteps Table

Step 1: Load JSON Files from API

```
INSERT INTO dbo.ETLJobSteps
  (strJobName, strJobStepName, strJobStepPartName, strJobStepStatus,
   intProcessNo, intProcessOrder, strStartKeyValue, strEndKeyValue,
   strCommandType, strCommand, strNextKeyCommand, strNextKeyCommandType,
   strFailPackageonFailure)
VALUES
  ('DWCurrencyRates', 'currencyrates_api', '01 Get JSON Files', 'I',
   0, 1, '2026-04-15', '2049-12-31',
   'DATAFILELOAD',
   '<datafileload>
     <datafileworkspacename>aesfabricetl</datafileworkspacename>
     <datafilelakehousename>lhdw</datafilelakehousename>
     <datafilefolderpattern>exchangerates/yyyy/mm/dd</datafilefolderpattern>
     <datafilenameprefixpattern>yyyymmdd</datafilenameprefixpattern>
     <datafilename>_currencyexchangerates</datafilename>
     <datafiletype>json</datafiletype>
     <datafrequencytype>DAY</datafrequencytype>
     <datafrequency>1</datafrequency>
     <datafileloadpythonscript><![CDATA[
import requests
import time
time.sleep(1)
API_KEY = "$$CurrencyLayerAPIKey$$"
BASE_URL = "http://api.currencylayer.com/historical"

lparams = {
    "access_key": API_KEY,
    "date":  datadate[:10],
    "source": "USD",
    "format": 1
}

try:
    response = requests.get(BASE_URL, params=lparams)
    ldata = response.json()
    if ldata["success"]:
        scriptresult = {"status": "success", "data": ldata}
    else:
        scriptresult = {"status": "error", "error": ldata["error"]["info"]}
except Exception as e:
    scriptresult = {"status": "error", "error": str(e)}
     ]]></datafileloadpythonscript>
   </datafileload>',
   '<instrumentationdbsql>
     <sqlquery><![CDATA[
       SELECT strCurrentKeyValue AS strNextKeyValue
       FROM dbo.ETLJobSteps
       WHERE intJobStepId = 5
     ]]></sqlquery>
```

```
    </instrumentationdbsql>',
    'INSTRUMENTATIONDBSQL', 'Y');
```

Step 2: Post-Process JSON Files to Delta Table

```
INSERT INTO dbo.ETLJobSteps
  (strJobName, strJobStepName, strJobStepPartName, strJobStepStatus,
   intProcessNo, intProcessOrder, strStartKeyValue, strEndKeyValue,
   strCommandType, strCommand, strFailPackageonFailure)
VALUES
  ('DWCurrencyRates', 'currencyrates_postprocess', '02 Post Process', 'I',
   0, 1, '2026-04-15', '2049-12-31',
   'DATAFILEPOSTPROCESS',
   '<datafilepostprocess>
     <datafileloadjobtaskid>5</datafileloadjobtaskid>

<datafilepostprocessworkspacename>aesfabricetl</datafilepostprocessworkspacename>
     <datafilepostprocesslakehousename>lhdw</datafilepostprocesslakehousename>
     <datafilepostprocesspythonscript><![CDATA[
import json
from datetime import datetime, timezone
from pyspark.sql.types import StructType, StructField, StringType, DoubleType
from delta.tables import DeltaTable
from notebookutils import mssparkutils

lschema_name = "dw"
ltable_name = "currency_exchange_rates"
table_abfs_path =
f"abfss://{datafilepostprocessworkspace_id}@onelake.dfs.fabric.microsoft.com/{data
filepostprocesslakehouse_id}/Tables/{lschema_name}/{ltable_name}"

try:
    lraw = mssparkutils.fs.head(datafile, 50 * 1024 * 1024)
    lpayload = json.loads(lraw)
    lquotes = lpayload.get("quotes", {})
    ldate = lpayload.get("date")
    lingested_at = datetime.now(timezone.utc).strftime("%Y-%m-%d %H:%M:%S")

    lrows = [{"rate_date": ldate, "currencycode": k, "currencyrate": float(v),
"_lastupdatedate": lingested_at} for k, v in lquotes.items()]

    schema = StructType([StructField("rate_date", StringType(), True),
StructField("currencycode", StringType(), True),
                         StructField("currencyrate", DoubleType(), True),
StructField("_lastupdatedate", StringType(), True)])
    ldf_rates = spark.createDataFrame(lrows, schema=schema)

    if DeltaTable.isDeltaTable(spark, table_abfs_path):
        deltaTable = DeltaTable.forPath(spark, table_abfs_path)
        deltaTable.alias("t").merge(ldf_rates.alias("s"), "t.rate_date =
s.rate_date AND t.currencycode = s.currencycode"
        ).whenMatchedUpdate(set = {"currencyrate": "s.currencyrate",
```

```
"_lastupdatedate": "s._lastupdatedate"
        }).whenNotMatchedInsertAll().execute()
    else:
        ldf_rates.write.format("delta").save(table_abfs_path)

    scriptresult = {"status": "success", "data": {"table": table_abfs_path,
"rows_processed": len(lrows), "rate_date": ldate}}
except Exception as e:
    scriptresult = {"status": "error", "error": str(e)}
     ]]></datafilepostprocesspythonscript>
   </datafilepostprocess>', 'Y');
```

Example 2: DWAdventureWorks - Copy Tables from SQL Server

This job demonstrates parallel loading of multiple tables with Type 2 SCD.

ETLJobs Table

```
INSERT INTO dbo.ETLJobs
    (strJobName, strJobStatus, strJobDefaultProcessType, strJobCurrentProcessType)
VALUES
    ('DWAdventureWorks', 'C', 'P', 'P');
```

ETLJobSteps Table

Step 1: Load Product Table (Process No 0)

```
INSERT INTO dbo.ETLJobSteps
  (strJobName, strJobStepName, strJobStepPartName, intProcessNo, intProcessOrder,
   strStartKeyValue, strEndKeyValue, strCommandType, strCommand,
   strNextKeyCommand, strNextKeyCommandType, strFailPackageonFailure)
VALUES
  ('DWAdventureWorks', 'dw.adventureworks_saleslt_product', '01 Copy Tables',
   0, 1, '1900-01-01', '2049-12-31',
   'COPYMSSQL2LAKEHOUSE',
   '<copymssql2lakehouse>
     <sqlconnectionstringtype>gateway</sqlconnectionstringtype>
     <sqlconnectstring>sqlserver1</sqlconnectstring>
     <sqlquery><![CDATA[
       SELECT *, ModifiedDate AS LastUpdateDate
       FROM [AdventureWorks].[SalesLT].[Product]
       WHERE ModifiedDate >= ''@strStartKeyValue''
       AND ModifiedDate <= ''@strEndKeyValue''
     ]]></sqlquery>

<stagingtablename>lhdw.dwstg.adventureworks_saleslt_product</stagingtablename>

<destinationtablename>lhdw.dw.adventureworks_saleslt_product</destinationtablename
```

```
>
     <destinationtablemergetype>type2</destinationtablemergetype>

<destinationtablemergetype2datepart>SECOND</destinationtablemergetype2datepart>
     <sourcetableprimarykeycolumns>productid</sourcetableprimarykeycolumns>
     <desttablealternatekeycolumns>productid</desttablealternatekeycolumns>
     <convertschematolowercase>true</convertschematolowercase>
     <batchsize>100000</batchsize>
     <refreshendpoint>true</refreshendpoint>
   </copymssql2lakehouse>',
   '<mssql>
     <sqlconnectionstringtype>gateway</sqlconnectionstringtype>
     <sqlconnectstring>sqlserver1</sqlconnectstring>
     <sqlquery><![CDATA[
       SELECT MAX(CONVERT(VARCHAR(23), ModifiedDate, 121)) AS strNextKeyValue
       FROM [AdventureWorks].[SalesLT].[Product]
     ]]></sqlquery>
     <lakehouse>lhdw</lakehouse>
   </mssql>',
   'MSSQL', 'N');
```

Step 2: Load Address Table (Process No 1 - runs in parallel with other Process No 1 steps)

```
INSERT INTO dbo.ETLJobSteps
  (strJobName, strJobStepName, strJobStepPartName, intProcessNo, intProcessOrder,
   strStartKeyValue, strEndKeyValue, strCommandType, strCommand,
   strNextKeyCommand, strNextKeyCommandType, strFailPackageonFailure)
VALUES
  ('DWAdventureWorks', 'dw.adventureworks_saleslt_address', '01 Copy Tables',
   1, 1, '1900-01-01', '2049-12-31',
   'COPYMSSQL2LAKEHOUSE',
   '<copymssql2lakehouse>
     <sqlconnectionstringtype>connectstring</sqlconnectionstringtype>
     <sqlconnectstring><![CDATA[$$azkv_sourcedata_connstring$$]]>
</sqlconnectstring>
     <sqlquery><![CDATA[
       SELECT *, ModifiedDate AS LastUpdateDate
       FROM [AdventureWorks].[SalesLT].[Address]
       WHERE ModifiedDate >= ''@strStartKeyValue''
       AND ModifiedDate <= ''@strEndKeyValue''
     ]]></sqlquery>

<stagingtablename>lhdw.dwstg.adventureworks_saleslt_address</stagingtablename>

<destinationtablename>lhdw.dw.adventureworks_saleslt_address</destinationtablename
>
     <destinationtablemergetype>type2</destinationtablemergetype>

<destinationtablemergetype2datepart>SECOND</destinationtablemergetype2datepart>
     <sourcetableprimarykeycolumns>addressid</sourcetableprimarykeycolumns>
     <desttablealternatekeycolumns>addressid</desttablealternatekeycolumns>
     <convertschematolowercase>true</convertschematolowercase>
```

```
      <batchsize>100000</batchsize>
      <refreshendpoint>true</refreshendpoint>
    </copymssql2lakehouse>',
    '<mssql>
      <sqlconnectionstringtype>connectstring</sqlconnectionstringtype>
      <sqlconnectstring><![CDATA[$$azkv_sourcedata_connstring$$]]>
</sqlconnectstring>
      <sqlquery><![CDATA[
        SELECT MAX(CONVERT(VARCHAR(23), ModifiedDate, 121)) AS strNextKeyValue
        FROM [AdventureWorks].[SalesLT].[Address]
      ]]></sqlquery>
      <lakehouse>lhdw</lakehouse>
    </mssql>',
    'MSSQL', 'N');
```

Step 3: Shadow Delete for Customer Table

```
INSERT INTO dbo.ETLJobSteps
  (strJobName, strJobStepName, strJobStepPartName, intProcessNo, intProcessOrder,
   strStartKeyValue, strEndKeyValue, strCommandType, strCommand,
strFailPackageonFailure)
VALUES
  ('DWAdventureWorks', 'shadow_delete_customer', '02 Shadow Deletes',
   0, 1, '1900-01-01', '2049-12-31',
   'SHADOWDELETEMSSQL2LH',
   '<shadowdeletemssql2lakehouse>
     <sqlgatewayname>sqlserver1</sqlgatewayname>
     <sqlquery><![CDATA[
       SELECT customerid
       FROM [AdventureWorks].[SalesLT].[Customer]
       ORDER BY customerid
     ]]></sqlquery>

<stagingtablename>lhdw.dwstg.pk__adventureworks_saleslt_customer</stagingtablename
>

<destinationtablename>lhdw.dw.adventureworks_saleslt_customer</destinationtablenam
e>
     <sourcetableprimarykeycolumns>customerid</sourcetableprimarykeycolumns>
     <desttablealternatekeycolumns>customerid</desttablealternatekeycolumns>
     <convertschematolowercase>true</convertschematolowercase>
     <batchsize>100000</batchsize>
     <refreshendpoint>true</refreshendpoint>
   </shadowdeletemssql2lakehouse>', 'N');
```

Step 4: Optimize Tables with Spark SQL

```
INSERT INTO dbo.ETLJobSteps
  (strJobName, strJobStepName, strJobStepPartName, intProcessNo, intProcessOrder,
```

```
    strStartKeyValue, strEndKeyValue, strCommandType, strCommand,
strFailPackageonFailure)
VALUES
  ('DWAdventureWorks', 'optimize_tables', '03 Optimize',
   0, 1, '1900-01-01', '2049-12-31',
   'SPARKSQL',
   '<sparksql>
     <sqlcommand><![CDATA[
       OPTIMIZE lhdw.dw.adventureworks_saleslt_product ZORDER BY (_iscurrent,
productid);
       OPTIMIZE lhdw.dw.adventureworks_saleslt_address ZORDER BY (_iscurrent,
addressid);
       OPTIMIZE lhdw.dw.adventureworks_saleslt_customer ZORDER BY (_iscurrent,
customerid);
     ]]></sqlcommand>
     <executeparallel>true</executeparallel>
     <maxparallelism>3</maxparallelism>
   </sparksql>', 'N');
```

Global Variables Configuration

ETLJobVariables Table

```
-- API Key for currency layer service
INSERT INTO dbo.ETLJobVariables
    (strJobName, strJobVariableName, strJobVariableValue)
VALUES
    ('Global', 'CurrencyLayerAPIKey', '223978105a8e1e10d737984b69a1665e');

-- Azure Key Vault reference for SQL Server connection string
INSERT INTO dbo.ETLJobVariables
    (strJobName, strJobVariableName, strJobVariableValue)
VALUES
    ('Global', 'azkv_sourcedata_connstring',
     '$azkv$kvaesetlsandbox.secret.sqlserver1-connectionstring$azkv$');

-- Workspace name for data operations
INSERT INTO dbo.ETLJobVariables
    (strJobName, strJobVariableName, strJobVariableValue)
VALUES
    ('Global', 'DWWorkspaceName', 'AESFabricETL');
```

Appendix B: Database Schema Scripts

The following tables are defined in the instrumentation database (DWInstrumentation):

Core Metadata Tables

ETLCommandTypes

Stores command type templates (optional - used by UI applications for command generation).

```
CREATE TABLE [dbo].[ETLCommandTypes](
    [strCommandType] [varchar](20) NOT NULL PRIMARY KEY,
    [strCommandTemplate] [nvarchar](max) NULL,
    [dtmUpdateDate] [datetime2](7) NULL DEFAULT (getdate()),
    [strUpdateUser] [nvarchar](100) NULL DEFAULT (suser_sname())
)
```

ETLJobs

Main job definitions and current execution status.

```
CREATE TABLE [dbo].[ETLJobs](
    [intJobID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
    [strJobName] [nvarchar](300) NOT NULL UNIQUE,
    [strJobDescription] [nvarchar](max) NULL,
    [strJobStatus] [char](1) NULL,
    [strJobDefaultProcessType] [char](1) NULL,
    [strJobCurrentProcessType] [char](1) NULL,
    [dtmStartDate] [datetime2](7) NULL,
    [dtmEndDate] [datetime2](7) NULL,
    [strSessionID] [varchar](100) NULL,
    [strLogFileName] [varchar](4000) NULL,
    [strMessage] [nvarchar](max) NULL,
    [dtmUpdateDate] [datetime2](7) NULL DEFAULT (getdate()),
    [strUpdateUser] [nvarchar](100) NULL DEFAULT (suser_sname())
)
```

ETLJobSteps

Individual task configurations with command XML and execution tracking.

```
CREATE TABLE [dbo].[ETLJobSteps](
    [intJobStepID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
    [strJobName] [nvarchar](300) NOT NULL,
    [strJobStepName] [nvarchar](300) NOT NULL,
    [strJobStepPartName] [nvarchar](300) NULL,
    [strJobStepStatus] [char](1) NULL,
    [intProcessNo] [int] NULL,
    [intProcessOrder] [int] NULL,
    [strStartKeyValue] [nvarchar](300) NULL,
    [strEndKeyValue] [nvarchar](300) NULL,
    [strCommandType] [varchar](20) NULL,
    [strCommand] [nvarchar](max) NULL,
```

```
    [strNextKeyCommand] [nvarchar](max) NULL,
    [strNextKeyCommandType] [varchar](20) NULL,
    [strCurrentKeyValue] [nvarchar](300) NULL,
    [strNextKeyValue] [nvarchar](300) NULL,
    [strFailPackageonFailure] [char](1) NULL,
    [dtmStartDate] [datetime2](7) NULL,
    [dtmEndDate] [datetime2](7) NULL,
    [strMessage] [nvarchar](max) NULL,
    [dtmUpdateDate] [datetime2](7) NULL DEFAULT (getdate()),
    [strUpdateUser] [nvarchar](100) NULL DEFAULT (suser_sname())
)
```

ETLJobVariables

Global and job-specific variables for parameterization.

```
CREATE TABLE [dbo].[ETLJobVariables](
    [intJobVariableID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
    [strJobName] [nvarchar](300) NULL,
    [strJobVariableName] [nvarchar](1000) NULL,
    [strJobVariableValue] [nvarchar](max) NULL,
    [bitDeleted] [bit] NULL DEFAULT ((0)),
    [dtmUpdateDate] [datetime2](7) NULL DEFAULT (getdate()),
    [strUpdateUser] [nvarchar](100) NULL DEFAULT (suser_sname())
)
```

Audit and Tracking Tables

ETLJobAudit

Historical record of all job executions.

```
CREATE TABLE [dbo].[ETLJobAudit](
    [intJobAuditID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
    [intJobID] [int] NOT NULL,
    [strJobName] [nvarchar](300) NOT NULL,
    [strJobStatus] [char](1) NULL,
    [strJobDefaultProcessType] [char](1) NULL,
    [strJobCurrentProcessType] [char](1) NULL,
    [dtmStartDate] [datetime2](7) NULL,
    [dtmEndDate] [datetime2](7) NULL,
    [strSessionID] [varchar](100) NULL,
    [strLogFileName] [varchar](4000) NULL,
    [strMessage] [nvarchar](max) NULL,
    [dtmUpdateDate] [datetime2](7) NULL DEFAULT (getdate()),
    [strUpdateUser] [nvarchar](100) NULL DEFAULT (suser_sname())
)
```

ETLJobStepAudit

Historical record of all job step executions.

```
CREATE TABLE [dbo].[ETLJobStepAudit](
    [intJobStepAuditID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
    [intJobStepID] [int] NOT NULL,
    [intJobAuditID] [int] NOT NULL,
    [strJobName] [nvarchar](300) NOT NULL,
    [strJobStepName] [nvarchar](300) NOT NULL,
    [strJobStepPartName] [nvarchar](300) NULL,
    [strJobStepStatus] [char](1) NULL,
    [intProcessNo] [int] NULL,
    [intProcessOrder] [int] NULL,
    [strStartKeyValue] [nvarchar](300) NULL,
    [strEndKeyValue] [nvarchar](300) NULL,
    [strCommandType] [varchar](20) NULL,
    [strCommand] [nvarchar](max) NULL,
    [strNextKeyCommand] [nvarchar](max) NULL,
    [strNextKeyCommandType] [varchar](20) NULL,
    [strCurrentKeyValue] [nvarchar](300) NULL,
    [strNextKeyValue] [nvarchar](300) NULL,
    [strFailPackageonFailure] [char](1) NULL,
    [dtmStartDate] [datetime2](7) NULL,
    [dtmEndDate] [datetime2](7) NULL,
    [strMessage] [nvarchar](max) NULL,
    [dtmUpdateDate] [datetime2](7) NULL DEFAULT (getdate()),
    [strUpdateUser] [nvarchar](100) NULL DEFAULT (suser_sname())
)
```

ETLDataFileLoadDetails

Tracks individual file loads for DATAFILELOAD operations.

```
CREATE TABLE [dbo].[ETLDataFileLoadDetails](
    [intFileLoadID] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY,
    [intJobStepID] [int] NOT NULL,
    [intJobStepAuditID] [int] NOT NULL,
    [strFileFolder] [nvarchar](1000) NULL,
    [strFileName] [nvarchar](1000) NULL,
    [dtmFileDate] [datetime2](7) NULL,
    [strSessionID] [varchar](38) NULL,
    [strStatus] [char](1) NULL,
    [strMessage] [nvarchar](max) NULL,
    [dtmLoadStartDate] [datetime2](7) NULL,
    [dtmLoadEndDate] [datetime2](7) NULL,
    [dtmCreateDate] [datetime2](7) NULL DEFAULT (getdate()),
    [strCreateUser] [nvarchar](100) NULL DEFAULT (suser_sname())
)
```

Note: For complete stored procedure definitions, refer to the database schema file: `DWInstrumentation-{guid}.Database.sql`

Appendix C: AESFabricControlPanel

A user-friendly interface is available in Microsoft Fabric Apps for configuring, maintaining, monitoring, and gaining insights into your ETL jobs.

Support & Contact

For technical support, licensing inquiries, or feature requests:

- **Email**: support@assuranceeservices.com
- **Website**: www.assuranceeservices.com
- **Documentation**: Check for updates and additional resources on our website

---

License

This software is licensed under the Business Source License 1.1 (BSL).

- **Community Use**: Free for up to 1,000 Task Runs per month
- **Commercial Use**: License required for usage exceeding 1,000 Task Runs per month
- **Change Date**: 2030-06-01
- **Post-Change License**: Apache License 2.0

For full license text, see LICENSE.md file.

Document Version: 1.0
Last Updated: April 27, 2026