Pandas in the Cloud

Simplifying AWS Data Workflows with AWS SDK for Pandas

Jim O'Neil – Senior Solutions Architect / Python Developer

LinkedIn: www.linkedin.com/in/jimoneil38

About Me

- Former CTO at a large e-commerce company called HobbyLink Japan
- Led architecture and backend for a just-finished 2-year AWS + Pandas project
- 20+ years in software and cloud
  (yeah… been doing this: a while)
- Mentor junior engineers and early-career devs
- First time back on stage in a while
- Now exploring new opportunities

“Views expressed are my own and not those of any employer—past, present, or future.”

What You'll Learn

- How to eliminate boilerplate code with AWS SDK for Pandas
- Ways to read/write to cloud services like S3 and Athena
  without touching boto3
- Why Wrangler makes your ETL pipelines more maintainable and Pythonic
- Where it shines - and where you **shouldn't** use it (yes, there are limits!)
- Example code using real-world patterns:
  CSVs, partitions, Excel, and DynamoDB
- Lessons from running Wrangler in production at scale (for 2 years)

Why Pandas?

- The go-to tool for data cleaning, exploration, and quick ETL
- Loved for its flexibility, hated for its lack of cloud awareness
- Native tools (like `to_csv()` or `read_excel()`)
  don't scale well in cloud pipelines
- When you try to combine Pandas with S3, Glue, or Athena…
  suddenly you're writing **a lot of boto3 and retry loops**
- Schema evolution? Glue Catalog? Table partitions?
  Not Pandas' problem — until it is

"Pandas is great… until you want to use it like a grown-up in the cloud."

Why AWS SDK for Pandas?

- Built for Python devs, not cloud engineers
- Replaces **dozens of lines of boto3 boilerplate** with a single `.to_parquet()`
- Auto-registers tables in the **Glue Catalog**
- Makes Athena queries feel like calling `.read_sql_query()`
- Works seamlessly with **pandas**, **pyarrow**, and even **Excel**
- Supports **S3 partitioning**, **DynamoDB I/O**, **Redshift COPY/UNLOAD** all in familiar syntax
- Keeps your ETL pipeline *Pythonic* and testable
- Community-driven, AWS-maintained with real-world adoption

What It Is

- A high-level wrapper around `boto3`, `pyarrow`, and `pandas`
- A simple way to move structured data in/out of:
  - S3 (CSV, Parquet, JSON, Excel)
  - Glue Catalog
  - Athena (read/write queries)
  - Redshift, DynamoDB, Oracle, MySQL, DeltaLake…
- A productivity boost for Python developers working with AWS data
- A great fit for **medium-scale** data tasks (GBs, not PBs)
- A bridge between your local DataFrame and the AWS ecosystem

What It Isn’t

❌ Not a distributed compute engine
(use **Spark**, **Dask**, or **Athena CTAS** for that)

❌ Not optimized for ML pipelines or real-time streaming

❌ Doesn’t handle **multi-node scaling or orchestration**

❌ Not a silver bullet
- you still need to manage IAM, schema evolution(?), and formats

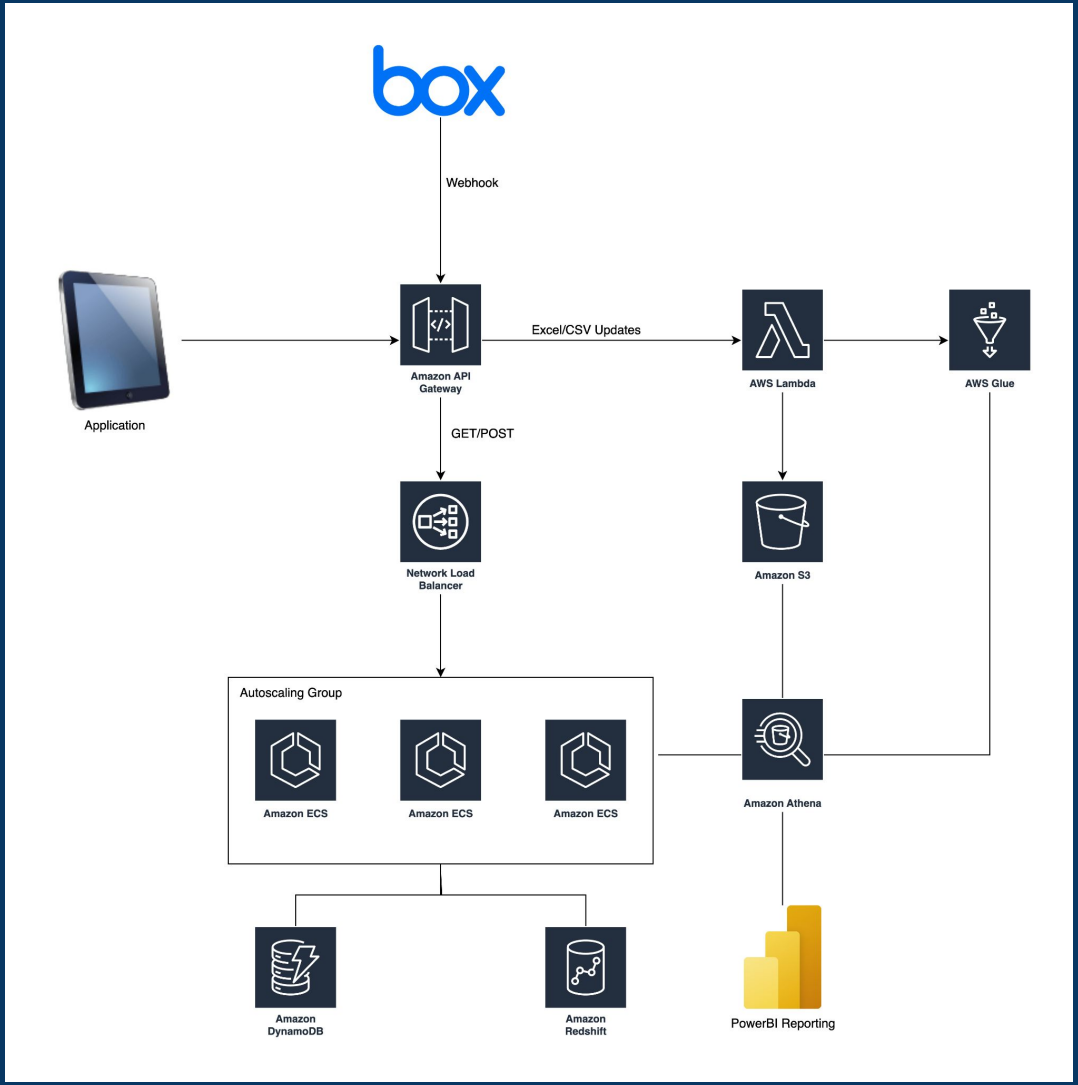
✅ But: it’s awesome for 80% of ETL, reporting, and batch jobs

Quick Overview: AWS Glue

- AWS Glue is a **data catalog + schema registry** for structured data
- Wrangler uses the **Glue Data Catalog** to register S3 tables (via `to_parquet()`)
- Makes your data **queryable in services like:**
  Athena
  Redshift Spectrum
  Lake Formation
  S3 (Storage of flat files, parquet...)
- You don’t need to run Glue Crawlers as **Wrangler handles schema creation!**
- Think of it as the **“metadata glue”** that connects your DataFrames to AWS analytics tools

You’re not writing Glue jobs here. Wrangler uses Glue for what it does best - tracking schemas and partitions so Athena/RedShift doesn’t throw errors at 2 a.m.

Very High-Level



CSV to Parquet (Boto3 vs. Wrangler)

```
# BEFORE: Pandas + boto3 to write Parquet and register with Glue
import pandas as pd
import boto3
import logging
from datetime import datetime
import os

# ENVIRONMENT VARIABLES
S3_BUCKET_NAME = os.environ.get('S3_BUCKET_NAME', 'demo-bucket-changeme')
GLUE_DATABASE_NAME = os.environ.get('GLUE_DATABASE_NAME', 'demo-glue-catalog-changeme')

# Configure logging for BEFORE section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Initialize AWS clients
s3 = boto3.client("s3")
glue = boto3.client("glue")

# Read CSV from S3
df = pd.read_csv(f"s3://{S3_BUCKET_NAME}/movies.csv")

# Extract clean title and year, detect remakes
df[['title', 'release_year']] = df['title'].str.extract(r'^(.*?)\s*\((\d{4})\)')

# Handle invalid years gracefully - best practice for production code
df['release_year'] = pd.to_numeric(df['release_year'], errors='coerce')

# Convert pipe-separated genres to list for better searchability
# Athena can query arrays with contains() function: WHERE contains(genres, 'Action')
# Note: contains() is case-sensitive - MovieLens uses proper case (Action, Comedy, Sci-Fi)
df['genres'] = df['genres'].str.split('|')

# Manually partition data by release_year and upload to S3
for release_year, group in df.groupby('release_year'):
    # Save each year's data to local parquet file
    parquet_file = f"/tmp/movies_year_{release_year}.parquet"
    group.to_parquet(parquet_file)
    # Upload to S3 with partition structure
    with open(parquet_file, "rb") as f:
        s3.upload_fileobj(f, S3_BUCKET_NAME, f"movies/release_year={release_year}/movies.parquet")

# Manually register partitioned table with Glue Data Catalog
glue.create_table(
    DatabaseName=GLUE_DATABASE_NAME,
    TableInput={
        "Name": "movies",
        "StorageDescriptor": {
            # Define schema for all columns except partition column
            "Columns": [{"Name": col, "Type": "string"} for col in df.columns if col != "release_year"],
            "Location": f"s3://{S3_BUCKET_NAME}/movies/",
            # Specify Parquet input/output formats
            "InputFormat": "org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat",
            "OutputFormat": "org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat",
            "SerdeInfo": {"SerializationLibrary": "org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe"}
        },
        # Define partition column
        "PartitionKeys": [{"Name": "release_year", "Type": "int"}]
    }
)
```

```
# AFTER: AWS SDK for Pandas (wrangler) simplifies the entire flow
import awswrangler as wr
import logging
from datetime import datetime

# Configure logging for AFTER section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# ENVIRONMENT VARIABLES
S3_BUCKET_NAME = os.environ.get('S3_BUCKET_NAME', 'demo-bucket-changeme')
GLUE_DATABASE_NAME = os.environ.get('GLUE_DATABASE_NAME', 'demo-glue-catalog-changeme')

# Read CSV from S3
df = wr.s3.read_csv(f"s3://{S3_BUCKET_NAME}/movies.csv")

# Extract clean title and release year
df[['title', 'release_year']] = df['title'].str.extract(r'^(.*?)\s*\((\d{4})\)')

# Handle invalid years gracefully - best practice for production code
df['release_year'] = pd.to_numeric(df['release_year'], errors='coerce')

# Convert pipe-separated genres to list for better searchability
# Athena can query arrays with contains() function: WHERE contains(genres, 'Action')
# Note: contains() is case-sensitive - MovieLens uses proper case (Action, Comedy, Sci-Fi)
df['genres'] = df['genres'].str.split('|')

# Write partitioned parquet dataset and auto-register with Glue in one step
# This single function call handles partitioning by year, uploading to S3,
# and registering the table schema with Glue Data Catalog automatically
# Store release year as integer, not string
wr.s3.to_parquet(
    df=df,
    path=f"s3://{S3_BUCKET_NAME}/movies/",
    dataset=True,
    database=GLUE_DATABASE_NAME,
    table="movies",
    partition_cols=["release_year"],
    dtype={'release_year': 'int64'}
)
```

Athena Queries

```
# BEFORE: Athena query with boto3
import boto3
import time
import pandas as pd
import logging
import os

# Configure logging for BEFORE section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment variables
GLUE_DATABASE_NAME = os.environ.get('GLUE_DATABASE_NAME', 'movielens')
ATHENA_RESULT_LOCATION = os.environ.get('ATHENA_RESULT_LOCATION')

# Initialize Athena client and start query execution
athena = boto3.client("athena")
response = athena.start_query_execution(
   QueryString="SELECT title, genre FROM movies WHERE release_year = 1995",
   QueryExecutionContext={"Database": GLUE_DATABASE_NAME},
   ResultConfiguration={"OutputLocation": ATHENA_RESULT_LOCATION}
)

# Poll for query completion status manually
# Must continuously check until query finishes (succeeded, failed, or cancelled)
query_id = response["QueryExecutionId"]
while True:
   execution_details = athena.get_query_execution(QueryExecutionId=query_id)
   state = execution_details["QueryExecution"]["Status"]["State"]
   if state in ["SUCCEEDED", "FAILED", "CANCELLED"]:
       break
   time.sleep(1)

# Load results from S3 after query completes
if state == "SUCCEEDED":
   df = pd.read_csv(f"{ATHENA_RESULT_LOCATION.rstrip('/')}/{query_id}.csv")
else:
   logger.info(f"Query failed with state: {state}")

print(df.head(10))
```

```
# AFTER: Query Athena with awswrangler
import awswrangler as wr
import logging
from datetime import datetime

# Configure logging for AFTER section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment variables
GLUE_DATABASE_NAME = os.environ.get('GLUE_DATABASE_NAME', 'movielens')
ATHENA_RESULT_LOCATION = os.environ.get('ATHENA_RESULT_LOCATION')

# Matches the BEFORE example for direct comparison
df = wr.athena.read_sql_query(
   sql="SELECT title, genres FROM movies WHERE release_year = 1995",
   database=GLUE_DATABASE_NAME
)

print(df.head(10))
```

Excel to Parquet

Kana Normalization

```
# BEFORE: Read messy Excel, clean with jaconv, write manually
import pandas as pd
import jaconv
import boto3
import logging
import os

# Configure logging for BEFORE section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment variables
S3_BUCKET_NAME = os.environ.get('S3_BUCKET_NAME', 'demo-bucket-changeme')

# Read Excel file with Japanese text data
df = pd.read_excel("employees.xlsx")

# Standardize Japanese text in title and department columns
# jaconv.z2h() converts full-width characters to half-width (    ａ→a,  １→1)
# jaconv.hira2kata() converts hiragana to katakana (    ひらがな→カタカナ )
df[["title", "department"]] = df[["title", "department"]].applymap(lambda x:
jaconv.hira2kata(jaconv.z2h(x)))

# Manually partition data by department and write to S3
# Each department gets its own partition folder
for department, group in df.groupby('department'):

group.to_parquet(f"s3://{S3_BUCKET_NAME}/employees/department={department}/employees.parquet")

# Manually register partitioned table with Glue Data Catalog
# Define schema, partition keys, and Parquet format specifications
glue = boto3.client("glue")
glue.create_table(
   DatabaseName="employees",
   TableInput={
       "Name": "employees",
       "StorageDescriptor": {
           "Columns": [{"Name": "id", "Type": "string"}, {"Name": "title", "Type": "string"}],
           "Location": f"s3://{S3_BUCKET_NAME}/employees/",
           "InputFormat": "org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat",
           "OutputFormat": "org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat",
           "SerdeInfo": {"SerializationLibrary":
"org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe"}
       },
       "PartitionKeys": [{"Name": "department", "Type": "string"}]
   }
)
```

```
# AFTER: AWS SDK for Pandas + jaconv to Excel to Glue table
import pandas as pd
import jaconv
import awswrangler as wr
import logging
import os

# Configure logging for AFTER section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment variables
S3_BUCKET_NAME = os.environ.get('S3_BUCKET_NAME', 'demo-bucket-changeme')
GLUE_DATABASE_NAME = os.environ.get('GLUE_DATABASE_NAME', 'company')

# Read Excel file with Japanese text data
df = pd.read_excel("employees.xlsx")

# Standardize Japanese text in title and department columns
# jaconv.z2h() converts full-width characters to half-width (    ａ→a,  １→1)
# jaconv.hira2kata() converts hiragana to katakana (    ひらがな→カタカナ )
df[["title", "department"]] = df[["title", "department"]].applymap(
          lambda x: jaconv.hira2kata(jaconv.z2h(x))
          )

# Write partitioned parquet dataset and auto-register with Glue in one step
# Automatically handles partitioning, S3 upload, and Glue table registration
wr.s3.to_parquet(
   df=df,
   path=f"s3://{S3_BUCKET_NAME}/employees_clean/",
   dataset=True,
   database=GLUE_DATABASE_NAME,
   table="employees",
   partition_cols=["department"]
)

print(df.head(10))
```

CSV to DynamoDb

```
# BEFORE: Manual individual DynamoDB writes with complex type conversion


import pandas as pd
import boto3
import time
import logging
import os
from botocore.exceptions import ClientError

# Configure logging for BEFORE section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment variables
S3_BUCKET_NAME = os.environ.get('S3_BUCKET_NAME', 'demo-bucket-changeme')
DYNAMODB_TABLE_NAME = os.environ.get('DYNAMODB_TABLE_NAME', 'movies')

# Initialize DynamoDB client (not resource) for manual type handling
dynamodb = boto3.client("dynamodb")

# Read movies CSV from S3 and limit to first 100 rows for demo
df = pd.read_csv(f"s3://{S3_BUCKET_NAME}/movies.csv").head(100)

# Extract year and clean title
df['release_year'] = df['title'].str.extract(r'\((\d{4})\)')
df['title'] = df['title'].str.replace(r'\s*\(\d{4}\).*$', '', regex=True)

# Convert pipe-separated genres to list for better searchability
df['genres'] = df['genres'].str.split('|')

# Manual individual writes with type conversion and error handling
successful_writes = 0
failed_writes = 0

for _, row in df.iterrows():
    try:
        # Manual DynamoDB type conversion - required for each field
        # DynamoDB requires explicit type annotations: S=String, N=Number, SS=StringSet
        item = {
            'movieId': {'S': str(row['movieId'])},           # String type
            'title': {'S': str(row['title'])},               # String type
            'release_year': {'N': str(row['release_year'])}, # Number type (as string)
            'genres': {'SS': row['genres']}                  # String Set type
        }

        # Individual put_item call - no batching optimization
        dynamodb.put_item(
            TableName=DYNAMODB_TABLE_NAME,
            Item=item
        )
        successful_writes += 1

        # Rate limiting to avoid throttling - manual delay between writes
        time.sleep(0.01)  # 10ms delay between writes

    except ClientError as e:
        failed_writes += 1
        if e.response['Error']['Code'] == 'ProvisionedThroughputExceededException':
            time.sleep(1)  # Back off on throttling
        logger.error(f"Failed to write item {row['movieId']}: {e}")
    except Exception as e:
        failed_writes += 1
        logger.error(f"Unexpected error for item {row['movieId']}: {e}")

logger.info(f"Completed: {successful_writes} successful, {failed_writes} failed")
```

```
# AFTER: Read CSV and write to DynamoDB with awswrangler
import awswrangler as wr
import logging

# Configure logging for AFTER section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment variables
S3_BUCKET_NAME = os.environ.get('S3_BUCKET_NAME', 'demo-bucket-changeme')
DYNAMODB_TABLE_NAME = os.environ.get('DYNAMODB_TABLE_NAME', 'movies')

# Read movies CSV from S3 and limit to first 1000 rows
df = wr.s3.read_csv(f"s3://{S3_BUCKET_NAME}/movies.csv").head(1000)

# Extract year and clean title
df['release_year'] = df['title'].str.extract(r'\((\d{4})\)')
df['title'] = df['title'].str.replace(r'\s*\(\d{4}\).*$', '', regex=True)

# Convert pipe-separated genres to list for better searchability
df['genres'] = df['genres'].str.split('|')

# Write entire dataframe to DynamoDB in one operation
# Automatically handles all the complexity:
# - Batching into 25-item chunks
# - Error handling and retries
# - Data type conversions
# - Rate limiting
wr.dynamodb.put_df(
    df=df,
    table_name=DYNAMODB_TABLE_NAME
)
```

ETL to DynamoDb

```
# BEFORE: Manual Athena query execution + DynamoDB writes
# Requires: Query execution, polling, result retrieval, data transformation
import boto3
import pandas as pd
import time
import logging
import os

# Configure logging for BEFORE section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment variables - AWS resource identifiers
S3_BUCKET_NAME = os.environ.get('S3_BUCKET_NAME', 'demo-bucket-changeme')
GLUE_DATABASE_NAME = os.environ.get('GLUE_DATABASE_NAME', 'movielens')
ATHENA_RESULT_LOCATION = os.environ.get('ATHENA_RESULT_LOCATION')
DYNAMODB_TABLE_NAME = os.environ.get('DYNAMODB_TABLE_NAME', 'top-movies')

# Initialize AWS clients - separate clients for each service
athena_client = boto3.client('athena')  # For query execution
dynamodb_resource = boto3.resource('dynamodb')  # For batch writes

# Step 1: Execute analytical query in Athena
# Find movies by popular genres for fast operational lookups
query = """
SELECT
  movieId,
  title,
  release_year,
  genres,
  CASE
      WHEN release_year >= 2000 THEN 'Modern'
      ELSE 'Classic'
  END as era
FROM movies
WHERE contains(genres, 'Action')
  OR contains(genres, 'Comedy')
  OR contains(genres, 'Drama')
LIMIT 1000
"""

# Execute query and wait for completion - manual async handling
response = athena_client.start_query_execution(
  QueryString=query,
  QueryExecutionContext={"Database": GLUE_DATABASE_NAME},
  ResultConfiguration={"OutputLocation": ATHENA_RESULT_LOCATION}
)

query_id = response["QueryExecutionId"]

# Poll for completion - manual status checking with sleep intervals
while True:
  execution_details = athena_client.get_query_execution(QueryExecutionId=query_id)
  state = execution_details["QueryExecution"]["Status"]["State"]
  if state in ["SUCCEEDED", "FAILED", "CANCELLED"]:
      break
  time.sleep(2)  # Wait 2 seconds between status checks

if state != "SUCCEEDED":
  raise Exception(f"Query failed with state: {state}")

# Load results from S3 - manual CSV retrieval
# Athena stores results as CSV files in S3
result_df = pd.read_csv(f"{ATHENA_RESULT_LOCATION.rstrip('/')}/{query_id}.csv")

# Write to DynamoDB with manual batch operations
# Requires manual type conversion and batch management
table = dynamodb_resource.Table(DYNAMODB_TABLE_NAME)

try:
  with table.batch_writer() as batch:  # Handles batching automatically
      for _, row in result_df.iterrows():
          # Convert to DynamoDB item format - manual type conversion required
          item = {
              'pk': row['pk'],  # Partition key for query efficiency
              'sk': row['movie_key'],  # Sort key for range queries
              'title': row['title'],
              'release_year': int(row['release_year']),  # Must convert to int
              'genres': row['genres'],
              'era': row['era']
          }
          batch.put_item(Item=item)

  logger.info(f"Successfully loaded {len(result_df)} movies to DynamoDB")

except Exception as e:
  logger.error(f"Error writing to DynamoDB: {e}")
```

```
# AFTER: Streamlined ETL with awswrangler
# Simplifies: Query execution, result handling, and DynamoDB writes
import awswrangler as wr
import logging
import os

# Configure logging for AFTER section
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

# Environment variables - same resources, simpler usage
GLUE_DATABASE_NAME = os.environ.get('GLUE_DATABASE_NAME', 'movielens')
DYNAMODB_TABLE_NAME = os.environ.get('DYNAMODB_TABLE_NAME', 'top-movies')

# Step 1: Execute analytical query and get results directly
# Same SQL query as BEFORE section (with comma fix)
query = """
SELECT
   movieId,
   title,
   release_year,
   genres,
   CASE
       WHEN release_year >= 2000 THEN 'Modern'
       ELSE 'Classic'
   END as era
FROM movies
WHERE contains(genres, 'Action')
  OR contains(genres, 'Comedy')
  OR contains(genres, 'Drama')
LIMIT 1000
"""

# Single function call handles query execution, polling, and result retrieval
# Automatically manages: async execution, status polling, S3 result retrieval
result_df = wr.athena.read_sql_query(
   sql=query,
   database=GLUE_DATABASE_NAME
)

# Step 2: Write to DynamoDB in one operation
# Automatically handles: batching, retries, type conversions, error handling
wr.dynamodb.put_df(
   df=result_df,
   table_name=DYNAMODB_TABLE_NAME
)

logger.info(f"ETL completed: {len(result_df)} movies transferred from Athena to DynamoDB")
```

Lessons Learned

The best code is the code you don’t have to write

- **Use what's already built**
  - Wrangler saved time by giving us simple functions for complex AWS operations
- **Less glue code means fewer bugs**
  - Skipping boto3 boilerplate reduced logic we’d otherwise need to maintain
- **Avoided writing our own Athena polling, pagination, and retries**
  - Wrangler handled all of that for us
- **Most data workflows were one-liners**
  - Reading/writing to S3, querying Athena, even working with DynamoDB and RedShift
- **New team members understood the code quickly!**
  - Nothing exotic, just familiar Pandas patterns
- **Falling back to boto3 was rare**
  - Wrangler covered nearly everything we needed, cleanly
- **Consistent patterns led to better collaboration**
  - We reused high-level abstractions instead of writing custom code
- **Less code means less to test**
  - Fewer edge cases, less CI overhead, and more confidence in changes

Best Practices: Code and Data Patterns

- **Start with the SDK, not with boto3**
  - Begin with what Wrangler offers before writing low-level AWS client code
- **Partition early and thoughtfully**
  - Use `partition_cols` when writing to S3 for better performance and organization
- **Avoid hardcoded paths or schema definitions**
  - Dynamic and schema-aware code is more resilient and reusable
- **Prefer Parquet over CSV**
  - More compact and query-efficient, especially in Athena workflows
- **Use clear and descriptive column names**
  - Makes joins and transformations easier to understand and maintain

Best Practices: Workflow and Process

- **Register tables with the Glue Data Catalog**
  - Enables reliable access across multiple AWS services
- **Batch operations to reduce round trips**
  - Fewer calls to AWS means faster and more cost-effective processing
- **Wrap common I/O tasks in reusable functions**
  - Promotes consistency and simplifies team collaboration
- **Clean up temporary data**
  - Manage Athena results and S3 staging folders with cleanup logic or lifecycle rules
- **Let Wrangler do the work whenever possible**
  - Reduces custom code, simplifies testing, and accelerates development

Key Takeaways

- **Wrangler bridges the gap** between Pandas and AWS services
  - You can work with cloud-scale data using familiar Python tools
- **Less code means fewer bugs**
  - Let the SDK handle the AWS details so you can focus on your data
- **Standard patterns scale better than custom glue**
  - Reuse over reinvention makes pipelines easier to maintain and share
- **Wrangler isn't a silver bullet**
  - It's a productivity tool, not a distributed compute engine
- **Stay in the Pandas mindset**
  - You don’t need to become a cloud engineer to work with cloud data

Thanks for coming!

I'll be around after the session if you'd like to chat.

Happy to talk Python, cloud architecture, mentoring, or anything else on your mind.

Looking forward to connecting!

GitHub repository!