AWS Pyspark Development

Below is a detailed explanation of the project structure, full code implementation, test cases, and instructions on how to run or test the application.

Project Structure

```
pyspark_aws_app/
├── src/
│   ├── __init__.py
│   ├── config_reader.py    # Reads configuration from S3
│   ├── athena_loader.py    # Queries Athena tables
│   ├── data_transformer.py # PySpark transformations
│   ├── s3_writer.py        # Writes data to S3
│   └── main.py             # Main application script
├── tests/
│   ├── __init__.py
│   ├── test_config_reader.py
│   ├── test_athena_loader.py
│   ├── test_data_transformer.py
│   ├── test_s3_writer.py
│   └── test_main.py
├── requirements.txt        # Dependencies
└── README.md               # Documentation

```

Code Implementation

`src/config_reader.py`

```
import json
import boto3

def read_config_from_s3(bucket_name, key):
    s3 = boto3.client("s3")
    obj = s3.get_object(Bucket=bucket_name, Key=key)
    return json.loads(obj["Body"].read())

```

`src/athena_loader.py`

```
from pyathena import connect

def query_athena(sql_query, database, work_group):
    conn = connect(database=database, work_group=work_group)
    cursor = conn.cursor()
    cursor.execute(sql_query)
    return cursor.fetchall()

```

`src/data_transformer.py`

```
from pyspark.sql import DataFrame

def transform_data(df: DataFrame) -> DataFrame:
    return df.filter(df["Age"] > 20).select("Name", "Age")

```

`src/s3_writer.py`

```
def write_to_s3(df, bucket_name, key):
    df.write.csv(f"s3a://{bucket_name}/{key}", mode="overwrite", header=True)

```

`src/main.py`

```
from pyspark.sql import SparkSession
from src.config_reader import read_config_from_s3
from src.athena_loader import query_athena
from src.data_transformer import transform_data
from src.s3_writer import write_to_s3

def main():
    spark = SparkSession.builder.appName("PySparkAWSApp").getOrCreate()

    # Read configuration
    config = read_config_from_s3("test-bucket", "config.json")

    # Query Athena
    sql_query = config["athena_query"]
    data = query_athena(sql_query, config["athena_database"], config["athena_work_group"])

    # Load data into PySpark
    df = spark.createDataFrame(data, schema=["Name", "Age"])

    # Transform data
    transformed_df = transform_data(df)

    # Write data back to S3
    write_to_s3(transformed_df, config["output_bucket"], "output/data.csv")

if __name__ == "__main__":
    main()

```

Test Cases

`tests/test_config_reader.py`

```
import boto3
from moto import mock_s3
from src.config_reader import read_config_from_s3

@mock_s3
def test_read_config_from_s3():
    s3 = boto3.client("s3", region_name="us-east-1")
    s3.create_bucket(Bucket="test-bucket")

```

```
    s3.put_object(Bucket="test-bucket", Key="config.json", Body='{"athena_query": "SELECT * FROM test", "output_bucket": "output-bucket"}')

    config = read_config_from_s3("test-bucket", "config.json")
    assert config["athena_query"] == "SELECT * FROM test"

```

`tests/test_athena_loader.py`

```
from src.athena_loader import query_athena

def test_query_athena(mocker):
    mocker.patch("src.athena_loader.connect")
    query_athena("SELECT * FROM test", "test_db", "primary")

```

`tests/test_data_transformer.py`

```
from pyspark.sql import SparkSession
from src.data_transformer import transform_data

def test_transform_data():
    spark = SparkSession.builder.appName("Test").getOrCreate()
    data = [("John", 30), ("Doe", 25), ("Jane", 15)]
    df = spark.createDataFrame(data, ["Name", "Age"])

    transformed_df = transform_data(df)
    assert transformed_df.count() == 2

```

`tests/test_s3_writer.py`

```
from pyspark.sql import SparkSession
from moto import mock_s3
import boto3
from src.s3_writer import write_to_s3

@mock_s3
def test_write_to_s3():
    spark = SparkSession.builder.appName("Test").getOrCreate()
    s3 = boto3.client("s3", region_name="us-east-1")
    s3.create_bucket(Bucket="output-bucket")

    data = [("John", 30), ("Doe", 25)]
    df = spark.createDataFrame(data, ["Name", "Age"])
    write_to_s3(df, "output-bucket", "output/data.csv")

```

How to Run or Test the Application

1. **Install Dependencies:**

```
pip install -r requirements.txt

```

2. **Run Tests:**

```
pytest tests/

```

3. **Run the Application:**

```
python src/main.py

```

Key Notes

- Use the `moto` library to mock AWS services for local testing.
- Use `pytest-mock` for mocking Athena queries.
- Configure `spark-defaults.conf` for local S3 access via `s3a://`.
- For production, switch mock configurations to actual AWS settings.

Let me know if you need further clarifications or enhancements!

---

update the above code, to include reading, trasnformating and writing to redshift

---

To include functionality for reading, transforming, and writing data to Amazon Redshift in your PySpark application, we'll use the `JDBC` connector provided by Spark. Below is the updated project structure, full code, and how to test the application.

Updated Project Structure

```
pyspark_redshift_app/
├── src/
│   ├── __init__.py
│   ├── config_reader.py       # Reads configuration from S3
│   ├── redshift_loader.py     # Reads data from Redshift
│   ├── data_transformer.py    # PySpark transformations
│   ├── redshift_writer.py     # Writes data to Redshift
│   └── main.py                # Main application script
├── tests/
│   ├── __init__.py
│   ├── test_config_reader.py
│   ├── test_redshift_loader.py
│   ├── test_data_transformer.py
│   ├── test_redshift_writer.py
│   └── test_main.py
├── requirements.txt           # Dependencies
└── README.md                  # Documentation

```

Code Implementation

`src/config_reader.py`

```
import json
import boto3
```

```

def read_config_from_s3(bucket_name, key):
    s3 = boto3.client("s3")
    obj = s3.get_object(Bucket=bucket_name, Key=key)
    return json.loads(obj["Body"].read())

```

`src/redshift_loader.py`

```
from pyspark.sql import SparkSession

def read_from_redshift(spark: SparkSession, jdbc_url: str, table: str, user: str, password: str):
    return spark.read.format("jdbc").option("url", jdbc_url) \
        .option("dbtable", table) \
        .option("user", user) \
        .option("password", password) \
        .load()

```

`src/data_transformer.py`

```
from pyspark.sql import DataFrame

def transform_data(df: DataFrame) -> DataFrame:
    return df.filter(df["age"] > 20).select("name", "age")

```

`src/redshift_writer.py`

```
def write_to_redshift(df, jdbc_url, table, user, password):
    df.write.format("jdbc").option("url", jdbc_url) \
        .option("dbtable", table) \
        .option("user", user) \
        .option("password", password) \
        .mode("overwrite") \
        .save()

```

`src/main.py`

```
from pyspark.sql import SparkSession
from src.config_reader import read_config_from_s3
from src.redshift_loader import read_from_redshift
from src.data_transformer import transform_data
from src.redshift_writer import write_to_redshift

def main():
    spark = SparkSession.builder.appName("PySparkRedshiftApp").getOrCreate()

    # Read configuration
    config = read_config_from_s3("test-bucket", "config.json")

    # Load data from Redshift
    df = read_from_redshift(
        spark,
        config["redshift_jdbc_url"],
        config["input_table"],
        config["redshift_user"],
```

```
        config["redshift_password"]
    )

    # Transform data
    transformed_df = transform_data(df)

    # Write transformed data to Redshift
    write_to_redshift(
        transformed_df,
        config["redshift_jdbc_url"],
        config["output_table"],
        config["redshift_user"],
        config["redshift_password"]
    )

if __name__ == "__main__":
    main()

```

Test Cases

`tests/test_redshift_loader.py`

```
from pyspark.sql import SparkSession
from src.redshift_loader import read_from_redshift

def test_read_from_redshift(mocker):
    spark = SparkSession.builder.appName("Test").getOrCreate()
    mocker.patch("pyspark.sql.DataFrameReader.load", return_value=spark.createDataFrame([("John", 30)], ["name", "age"]))
    df = read_from_redshift(spark, "jdbc:redshift://test", "test_table", "user", "password")
    assert df.count() == 1
    assert df.columns == ["name", "age"]

```

`tests/test_redshift_writer.py`

```
from pyspark.sql import SparkSession
from src.redshift_writer import write_to_redshift

def test_write_to_redshift(mocker):
    spark = SparkSession.builder.appName("Test").getOrCreate()
    data = [("John", 30), ("Doe", 25)]
    df = spark.createDataFrame(data, ["name", "age"])

    mocker.patch("pyspark.sql.DataFrameWriter.save")
    write_to_redshift(df, "jdbc:redshift://test", "test_table", "user", "password")

```

`tests/test_data_transformer.py`

```
from pyspark.sql import SparkSession
from src.data_transformer import transform_data

def test_transform_data():
    spark = SparkSession.builder.appName("Test").getOrCreate()
    data = [("John", 30), ("Doe", 25), ("Jane", 15)]
```

```
    df = spark.createDataFrame(data, ["name", "age"])

    transformed_df = transform_data(df)
    assert transformed_df.count() == 2

```

How to Run the Application

1. **Install Dependencies:** Add the following to `requirements.txt`:

   ```
   boto3
   pyspark
   pytest
   pytest-mock

   ```

   Install using:

   ```
   pip install -r requirements.txt

   ```

2. **Set Up Redshift JDBC Driver:**
   - Download the Redshift JDBC driver from AWS.
   - Add the driver to your Spark classpath:

     ```
     export SPARK_CLASSPATH=/path/to/redshift-jdbc-driver.jar

     ```

3. **Run Tests:**

   ```
   pytest tests/

   ```

4. **Run the Application:**

   ```
   python src/main.py

   ```

Key Notes

1. Replace `jdbc:redshift://...` with your actual Redshift JDBC URL in the configuration file.
2. Ensure the IAM role or credentials used have appropriate access to the Redshift cluster.
3. Use `pytest-mock` to mock external dependencies for testing Redshift interactions.

Let me know if you need further assistance!

---

how to package the above code and submit it to the emr cluster

---

Packaging your PySpark application and submitting it to an EMR cluster involves the following steps:

Steps to Package and Submit the Application

1. Organize the Project Structure

Ensure your project is structured properly:

```
pyspark_redshift_app/
├── src/
│   ├── __init__.py
│   ├── config_reader.py
│   ├── redshift_loader.py
│   ├── data_transformer.py
│   ├── redshift_writer.py
│   └── main.py
├── tests/                    # Optional, not required for deployment
├── requirements.txt          # Python dependencies
├── README.md                 # Documentation
├── setup.py                  # For packaging the project
└── config.json               # Configuration file (optional)

```

---

2. Create a `setup.py` File for Packaging

The `setup.py` file allows you to package the code as a Python module.

```
from setuptools import setup, find_packages

setup(
    name="pyspark_redshift_app",
    version="1.0.0",
    packages=find_packages(),
    install_requires=[
        "boto3",
        "pyspark",
        "pyathena"
    ],
    entry_points={
        "console_scripts": [
            "run_app=src.main:main"
        ]
    },
)

```

---

3. Package the Application

1. Create a `.tar.gz` or `.whl` package:

   ```
   python setup.py sdist

   ```

   This will create a `dist/` directory containing the packaged application, e.g., `pyspark_redshift_app-1.0.0.tar.gz`.

2. If you prefer a wheel:

   ```
   python setup.py bdist_wheel

   ```

---

4. Prepare Dependencies

1. List Python dependencies in `requirements.txt`:

```
boto3
pyspark
pyathena

```

2. Ensure the Redshift JDBC driver is available on the cluster. Either:
   - Place the `.jar` file in an S3 bucket.
   - Or directly reference it in your Spark configuration.

---

5. Upload Files to S3

Upload the following files to an S3 bucket accessible by your EMR cluster:

- Application package (`.tar.gz` or `.whl` file from `dist/`)
- `requirements.txt`
- `config.json` (if applicable)
- Redshift JDBC driver `.jar` file (if not already on the cluster).

Example command:

```
aws s3 cp dist/pyspark_redshift_app-1.0.0.tar.gz s3://your-bucket/path/
aws s3 cp requirements.txt s3://your-bucket/path/
aws s3 cp config.json s3://your-bucket/path/  # If used

```

---

6. Submit the Application to EMR

Option 1: Submit Using `spark-submit`

1. SSH into the EMR master node:

```
ssh -i your-key.pem hadoop@<emr-master-node-dns>

```

2. Run `spark-submit`:

```
spark-submit \
  --deploy-mode cluster \
  --master yarn \
  --py-files s3://your-bucket/path/pyspark_redshift_app-1.0.0.tar.gz \
  --jars s3://your-bucket/path/RedshiftJDBC42-no-awssdk-1.2.55.1083.jar \
  s3://your-bucket/path/pyspark_redshift_app-1.0.0.tar.gz/src/main.py

```

Option 2: Submit Using EMR Steps

1. Add a Spark step to your EMR cluster:

```
aws emr add-steps --cluster-id <CLUSTER_ID> \
  --steps Type=Spark,Name="PySparkApp",ActionOnFailure=CONTINUE,\

```

Args=[--deploy-mode,cluster,--master,yarn,--py-files,s3://your-bucket/path/pyspark_redshift_app-1.0.0.tar.gz,--jars,s3://your-bucket/path/RedshiftJDBC42-no-awssdk-1.2.55.1083.jar,s3://your-bucket/path/pyspark_redshift_app-1.0.0.tar.gz/src/main.py]

```

---

### **7. Debugging & Logs**
1. Monitor the EMR console for application status.
2. Review logs in the S3 bucket configured for EMR logs (found under EMR cluster settings).
3. SSH into the master node and check logs:
```bash
yarn logs -applicationId <APPLICATION_ID>

```

Additional Notes

- Ensure the EMR cluster has appropriate IAM roles to access S3, Redshift, and other AWS services.
- Use the `config.json` file to dynamically configure settings like Redshift JDBC URL, user, password, input table, and output table.

Let me know if you need further clarifications!

Explain every file in the above code and what does each file do

---

Here's a detailed explanation of each file in the provided project structure, its role, and how it contributes to the functionality of the PySpark application.

Project Structure Overview

```
pyspark_redshift_app/
├── src/
│   ├── __init__.py
│   ├── config_reader.py
│   ├── redshift_loader.py
│   ├── data_transformer.py
│   ├── redshift_writer.py
│   └── main.py
├── tests/
├── requirements.txt
├── setup.py
├── config.json
└── README.md

```

Detailed Explanation of Each File

1. `src/__init__.py`

- **Purpose**: Marks the `src/` directory as a Python package.
- **Contents**: Typically an empty file, but can optionally include initialization code for the package.
```

- **Usage**: Ensures other scripts can import modules from this directory, e.g., `from src.config_reader import read_config_from_s3`.

---

2. `src/config_reader.py`

- **Purpose**: Handles the reading of configurations stored in an Amazon S3 bucket.
- **Key Functionality**:
  - Connects to S3 using the `boto3` library.
  - Retrieves the configuration file (JSON format) and parses it into a Python dictionary.
- **Why It's Important**:
  - Centralized management of configuration values such as database connection details, table names, and other parameters.
  - Makes the application adaptable to different environments by simply changing the configuration file.

---

3. `src/redshift_loader.py`

- **Purpose**: Reads data from an Amazon Redshift table into a PySpark DataFrame.
- **Key Functionality**:
  - Uses Spark's JDBC connector to fetch data from Redshift.
  - Takes parameters like the JDBC URL, table name, and credentials (user and password).
- **Why It's Important**:
  - Facilitates reading large datasets efficiently from Redshift into Spark for processing.

---

4. `src/data_transformer.py`

- **Purpose**: Contains the data transformation logic for processing the DataFrame.
- **Key Functionality**:
  - Defines functions to filter, clean, and reshape the data based on business requirements.
  - For example, in the provided code, it filters records where `age > 20` and selects only `name` and `age`.
- **Why It's Important**:
  - Ensures the data meets the requirements of downstream systems or applications.
  - Encapsulates the transformation logic to keep the code modular and testable.

---

5. `src/redshift_writer.py`

- **Purpose**: Writes the transformed DataFrame back to an Amazon Redshift table.
- **Key Functionality**:
  - Uses Spark's JDBC connector to write data to Redshift.
  - Supports different write modes (`overwrite`, `append`, etc.).
- **Why It's Important**:
  - Handles writing processed data to the destination database for further use or analysis.
  - Ensures data integrity and proper handling of Redshift table schemas.

---

6. `src/main.py`

- **Purpose**: The entry point for the PySpark application.
- **Key Functionality**:
  - Initializes the Spark session.

- Orchestrates the workflow:
  i. Reads the configuration using `config_reader`.
  ii. Fetches data from Redshift using `redshift_loader`.
  iii. Transforms the data using `data_transformer`.
  iv. Writes the data back to Redshift using `redshift_writer`.
- **Why It's Important**:
  - Provides a high-level overview of the application's workflow.
  - Acts as the glue that integrates all the other components.

---

7. `tests/`

- **Purpose**: Contains unit tests for the application.
- **Key Functionality**:
  - Validates individual components (`config_reader`, `redshift_loader`, etc.) in isolation.
  - Uses mock data to ensure functions work as expected without needing actual connections to S3 or Redshift.
- **Why It's Important**:
  - Promotes test-driven development (TDD).
  - Ensures the application behaves correctly and prevents regressions.

---

8. `requirements.txt`

- **Purpose**: Lists all Python dependencies required for the project.
- **Contents**:
  - `boto3`: AWS SDK for Python, used to interact with S3.
  - `pyspark`: Required for Spark operations.
  - `pyathena`: (Optional) If interacting with Athena.
  - `pytest` and `pytest-mock`: For unit testing.
- **Why It's Important**:
  - Makes it easy to set up the project environment by installing all dependencies with a single command (`pip install -r requirements.txt`).

---

9. `setup.py`

- **Purpose**: Configures the project for packaging and distribution.
- **Key Functionality**:
  - Defines the package name, version, and dependencies.
  - Allows the project to be packaged as a `.tar.gz` or `.whl` file.
- **Why It's Important**:
  - Enables easy deployment of the project to clusters like EMR.
  - Standardizes the project structure for Python applications.

---

10. `config.json`

- **Purpose**: Stores configuration values such as database connection details, table names, and other parameters in JSON format.
- **Example Contents**:

```
1 {
```

```
    "redshift_jdbc_url": "jdbc:redshift://your-redshift-cluster:5439/yourdb",
    "input_table": "source_table",
    "output_table": "target_table",
    "redshift_user": "your_user",
    "redshift_password": "your_password"
}

```

- **Why It's Important**:
  - Provides a simple way to manage environment-specific configurations without hardcoding them in the code.

---

11. `README.md`

- **Purpose**: Provides documentation for the project.
- **Contents**:
  - Overview of the application.
  - How to set up the project.
  - How to run the application.
  - Examples of configuration and testing.
- **Why It's Important**:
  - Helps new developers quickly understand the project.
  - Serves as a reference for maintaining and extending the application.

How These Files Work Together

1. `main.py` drives the workflow by calling functions from the other modules.
2. `config_reader.py` provides the necessary configuration for interacting with Redshift.
3. `redshift_loader.py` retrieves data from Redshift.
4. `data_transformer.py` applies transformations to the retrieved data.
5. `redshift_writer.py` writes the transformed data back to Redshift.
6. **Tests** validate the correctness of each component independently.

This modular design ensures the application is maintainable, testable, and reusable. Let me know if you need further details!