**Test Case (helper.py)**

This document provides a detailed breakdown of each test case implemented in the provided code. It includes the purpose of each test case, the approach taken, the expected outcomes, and whether the test case is positive or negative.

**Test Case 1: get\_config\_files**

**Purpose:**

* Fetch JSON configuration files from a directory.
* Validate that all returned file names have a .json extension.

**Approach:**

1. Mock a directory path containing JSON files.
2. Use the helper.get\_config\_files function to retrieve file paths.
3. Validate the output using Pydantic's ConfigFilesModel to ensure all files have a .json extension.

**Expected Outcome:**

* The function should return a list of file paths with .json extensions.
* Pydantic validation should pass without errors.

**Test Type:**

* **Positive Test**: Validates the correct behavior when valid JSON files are present.

**Coverage:**

* File path retrieval.
* Pydantic validation for file extensions.

**Test Case 2: set\_dynamic\_file\_location**

**Purpose:**

* Replace placeholders in a file path dynamically based on environment-specific values.
* Validate the generated file path format.

**Approach:**

1. Define a file path with placeholders (e.g., {catalog\_name}).
2. Use the helper.set\_dynamic\_file\_location function to replace placeholders with environment-specific values.
3. Validate the output using Pydantic's FileLocationModel.

**Expected Outcome:**

* The function should correctly replace placeholders with environment-specific values.
* Pydantic validation should pass for valid environments (e.g., dev, prod).
* The function should raise a ValueError for invalid environments.

**Test Type:**

* **Positive Test**: Validates correct placeholder replacement for valid environments.
* **Negative Test**: Validates error handling for invalid environments.

**Coverage:**

* Dynamic file path generation.
* Error handling for invalid environments.

**Test Case 3: get\_watermark\_value**

**Purpose:**

* Fetch the latest watermark value from a given column in a Spark DataFrame.

**Approach:**

1. Create a Spark DataFrame with a timestamp column (LastModifiedDate).
2. Use the helper.get\_watermark\_value function to fetch the latest timestamp.
3. Compare the fetched value with the expected value.

**Expected Outcome:**

* The function should return the latest timestamp from the LastModifiedDate column.
* The returned value should match the expected value.

**Test Type:**

* **Positive Test**: Validates the correct retrieval of the latest watermark value.

**Coverage:**

* Watermark value extraction from a Spark DataFrame.

**Test Case 4: build\_source\_query**

**Purpose:**

* Construct a SQL query dynamically based on source schema, transformations, and watermark values.
* Validate whether the correct columns are included in the query.

**Approach:**

1. Define test cases with different combinations of source schema, transformations, and watermark values.
2. Use the helper.build\_source\_query function to generate SQL queries.
3. Compare the generated queries with the expected queries.

**Expected Outcome:**

* The function should generate SQL queries that match the expected format.
* The queries should include the correct columns and watermark conditions.

**Test Type:**

* **Positive Test**: Validates the correct generation of SQL queries.

**Coverage:**

* Dynamic SQL query generation.
* Handling of watermark conditions.

**Test Case 5: get\_df\_from\_source**

**Purpose:**

* Fetch data from a source table and validate the schema and data using Pydantic.

**Approach:**

1. Create a Spark DataFrame with valid data and schema.
2. Use the helper.get\_df\_from\_source function to fetch data.
3. Validate the fetched data using Pydantic's SourceDataSchema.

**Expected Outcome:**

* The function should return a DataFrame with valid data.
* Pydantic validation should pass for all rows.

**Test Type:**

* **Positive Test**: Validates the correct retrieval and validation of data.

**Coverage:**

* Data retrieval from a source table.
* Pydantic validation of Spark DataFrame.

**Test Case 6: create\_schema\_from\_transform\_map and get\_spark\_type**

**Purpose:**

* Create a Spark schema from a transformation map and validate the schema using Pydantic.

**Approach:**

1. Define a transformation map with various data types.
2. Use the helper.create\_schema\_from\_transform\_map function to generate a Spark schema.
3. Validate the schema and data using Pydantic's TestSchema.

**Expected Outcome:**

* The function should generate a valid Spark schema.
* Pydantic validation should pass for all data types.

**Test Type:**

* **Positive Test**: Validates the correct generation and validation of Spark schema.

**Coverage:**

* Spark schema generation from a transformation map.
* Pydantic validation of various data types.

**Test Case 7: file\_loader**

**Purpose:**

* (Not implemented in the provided code.)

**Coverage:**

* N/A

**Test Case 8: add\_dynamic\_columns\_with\_type\_check**

**Purpose:**

* Add dynamic columns to a Spark DataFrame and validate the schema using Pydantic.

**Approach:**

1. Define a schema with dynamic columns.
2. Use the helper.add\_dynamic\_columns\_with\_type\_check function to add columns to a DataFrame.
3. Validate the transformed DataFrame using Pydantic's TestSchema.

**Expected Outcome:**

* The function should add dynamic columns to the DataFrame.
* Pydantic validation should pass for all rows.

**Test Type:**

* **Positive Test**: Validates the correct addition and validation of dynamic columns.

**Coverage:**

* Dynamic column addition to a Spark DataFrame.
* Pydantic validation of transformed DataFrame.

**Test Case 9: delta\_table\_exists**

**Purpose:**

* Check if a Delta table exists in a given catalog and schema.

**Approach:**

1. Define test cases with valid and invalid table names.
2. Use the helper.delta\_table\_exists function to check if the table exists.
3. Compare the result with the expected outcome.

**Expected Outcome:**

* The function should return True for existing tables and False for non-existing tables.

**Test Type:**

* **Positive Test**: Validates the correct detection of Delta table existence.

**Coverage:**

* Delta table existence check.

**Test Case 10: add\_etl\_cols**

**Purpose:**

* Add ETL-specific columns (e.g., pac\_batch\_id, pac\_load\_timestamp) to a Spark DataFrame.

**Approach:**

1. Create a sample DataFrame without ETL columns.
2. Use the helper.add\_etl\_cols function to add ETL columns.
3. Validate the presence of the new columns.

**Expected Outcome:**

* The function should add ETL columns to the DataFrame.
* The DataFrame should contain the expected columns.

**Test Type:**

* **Positive Test**: Validates the correct addition of ETL columns.

**Coverage:**

* Addition of ETL-specific columns to a Spark DataFrame.

**Test Case 11: volume\_exists**

**Purpose:**

* Check if a volume exists in a given catalog and schema.

**Approach:**

1. Define test cases with valid and invalid volume names.
2. Use the helper.volume\_exists function to check if the volume exists.
3. Compare the result with the expected outcome.

**Expected Outcome:**

* The function should return True for existing volumes and False for non-existing volumes.

**Test Type:**

* **Positive Test**: Validates the correct detection of volume existence.

**Coverage:**

* Volume existence check.

**Test Case 12: write\_delta\_stream**

**Purpose:**

* (Not implemented in the provided code.)

**Coverage:**

* N/A

**Test Case 13: build\_cdc and union\_by\_name\_dfs**

**Purpose:**

* Detect changes (inserts, updates, deletes) between source and target DataFrames using CDC (Change Data Capture).

**Approach:**

1. Define test cases with different combinations of source and target DataFrames.
2. Use the helper.build\_cdc function to detect changes.
3. Compare the result with the expected DataFrame.

**Expected Outcome:**

* The function should correctly detect inserts, updates, and deletes.
* The result should match the expected DataFrame.

**Test Type:**

* **Positive Test**: Validates the correct detection of changes using CDC.

**Coverage:**

* Change Data Capture (CDC) logic.
* Handling of primary keys and excluded columns.

**Summary of Test Coverage**

| **Test Case** | **Positive Test** | **Negative Test** | **Coverage** |
| --- | --- | --- | --- |
| get\_config\_files | ✅ | ❌ | File path retrieval |
| set\_dynamic\_file\_location | ✅ | ✅ | Dynamic file path generation |
| get\_watermark\_value | ✅ | ❌ | Watermark value extraction |
| build\_source\_query | ✅ | ❌ | Dynamic SQL query generation |
| get\_df\_from\_source | ✅ | ❌ | Data retrieval |
| create\_schema\_from\_transform\_map | ✅ | ❌ | Spark schema generation |
| file\_loader | ❌ | ❌ | N/A |
| add\_dynamic\_columns\_with\_type\_check | ✅ | ❌ | Dynamic column addition |
| delta\_table\_exists | ✅ | ❌ | Delta table existence check |
| add\_etl\_cols | ✅ | ❌ | ETL column addition |
| volume\_exists | ✅ | ❌ | Volume existence check |
| write\_delta\_stream | ❌ | ❌ | N/A |
| build\_cdc | ✅ | ❌ | Change Data Capture (CDC) logic |

|  |  |
| --- | --- |
| **Use Case** | **Approach** |
| Call build\_cdc but track calls | MagicMock(side\_effect=helper.build\_cdc) |
| Mock dependencies (but run build\_cdc) | patch("helper.some\_dependency", return\_value=mock\_data) |
| Profile execution time | time.time() with MagicMock |
| Log function calls | print(helper.build\_cdc.call\_args) |

**Discussion details:**

The first level of description needs to explore the necessity of performing unit testing using Pydantic.

Date: 05/03/25  
 Based on the second discussion with Tyler, we need to integrate unittest.mock with the Pydantic model.

<https://docs.python.org/3/library/unittest.mock.html>