**DAMG 7245 - Assignment 3 - Team 5**

| **Summary** | The project aims to design, validate, and transform data using Python, Pydantic, and DBT. It involves creating Python classes to represent web pages and PDF files' schemas, enforcing guidelines for data consistency, and generating clean CSV files. |
| --- | --- |
| **URL** | <https://codelabs-preview.appspot.com/?file_id=10kLg_eqlQLwbnRTzetD3K1axOyOcv1lUpjK6VEiu4mg#0> |
| **Category** | Databases |
| **Environment** | Python |
| **Status** | Completed |
| **Github** | <https://github.com/BigDataIA-Spring2024-Sec1-Team5/Assignment-3> |
| **Authors** | Aditya Kanala, Shikhar Patel, Shubh Patel |

**Table of Contents**

[Architecture and Workflow](#_vjfa17joz2h8)

[Overview](#_70p7luabbmg3)

[Workflow](#_hqgc53ofwerd)

[Web Scraping](#_lre49l2hb00x)

Pydantic Classes

[URLClass](#_vt22sd9b3bl7)

[ContentClass](#_rj2sut67dsim)

[MetadataClass](#_wjxojtlo8tbo)

Pytest Classes

[URL PyTest](#_90wj1n5lbyig)

[Content PyTest](#_bfgepsv1mxxm)

[Metadata PyTest](#_sji0mwsjhpva)

[DBT Integration](#_gnumsna3b34s)

[References](#_aiolgsdsbrfh)

**Architecture**

# **Architecture and Workflow**

The workflow depicted in the image outlines a data processing and validation pipeline, with each node representing a service or a step in the process. Here's a brief description of the steps and the functionality of each service:

1. **Previous Architecture (Prev\_Arch):**

Functionality: Acts as the starting point of the workflow, possibly representing the legacy system or the initial data sources.

Steps: Data from the Previous Architecture is exported in two formats:

CSV files, which are web-scraped CSV files.

XML files, which are generated by Grobid, a tool for extracting, parsing, and restructuring scholarly documents.

2. **Web Scraped CSV File (CSV):**

Functionality: Contains structured data scraped from web pages.

Steps: The CSV data is passed to the URL\_Class for schema validation and processing.

3. **Grobid generated XML (XML):**

Functionality: XML files with metadata and content extracted from PDF documents using Grobid.

Steps: The XML data is split into two paths:

One leads to the Content\_Class for content extraction.

The other leads to an S3 bucket for metadata storage.

4. **URL\_Class, Metadata\_Class, Content\_Class (Pydantic Pytest):**

Functionality: These are Python classes that define the schema for the respective data types. They use Pydantic for data validation and Pytest for testing the data against the schema.

Steps: After validation and testing, the URL\_Class and Content\_Class output clean CSV files.

5. **Clean CSV File (CleanCSV):**

Functionality: Represents the validated and cleaned data ready for further processing.

Steps: The Clean CSV files are loaded into Snowflake.

6. **S3 (Amazon S3):**

Functionality: A storage service provided by Amazon Web Services (AWS) for object storage.

Steps: Stores the metadata from the XML files.

7. **Snowflake:**

Functionality: A cloud-based data warehousing service that allows for data storage, processing, and analytic solutions.

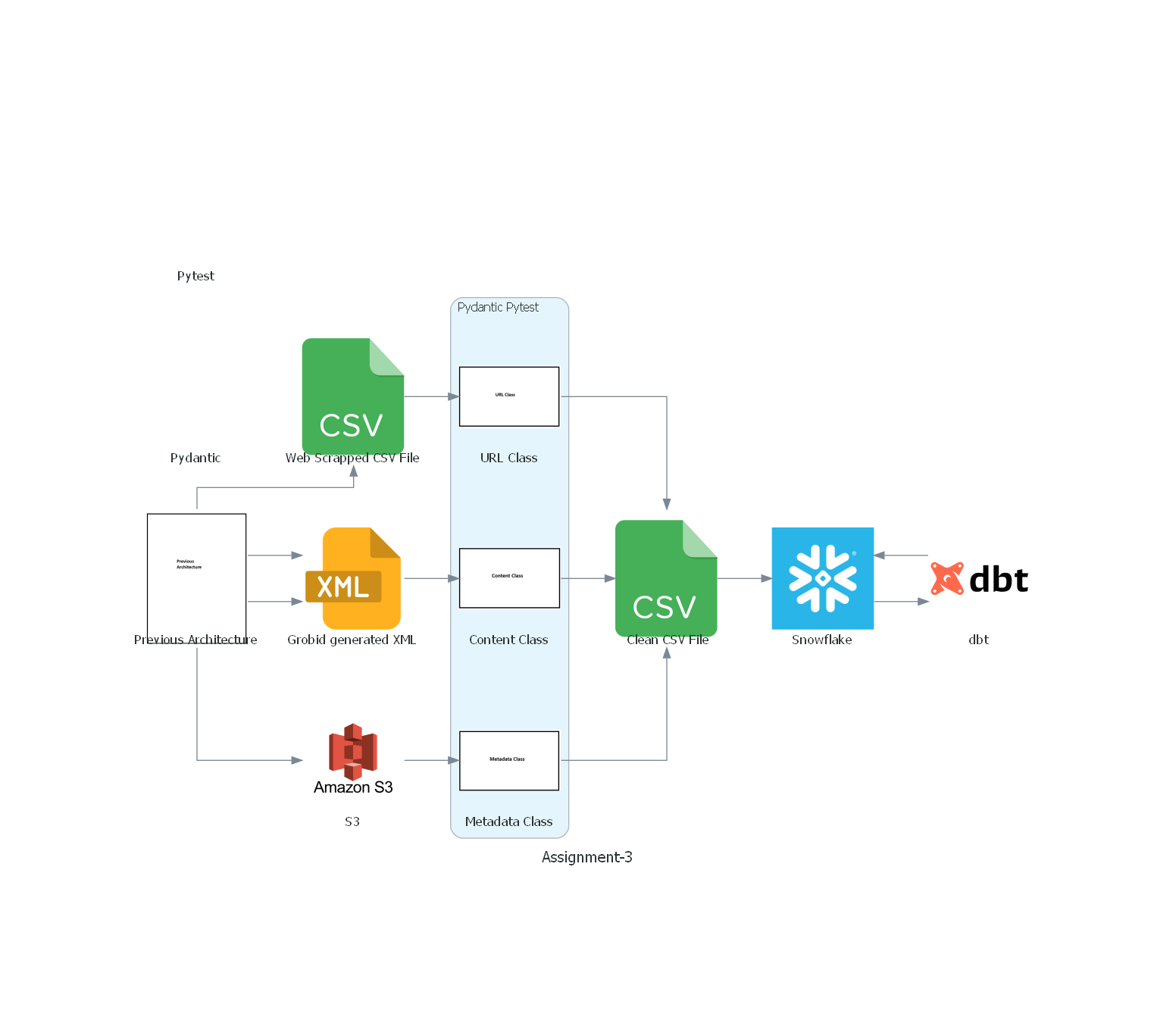
Steps: Receives the clean CSV data for storage and further transformations.

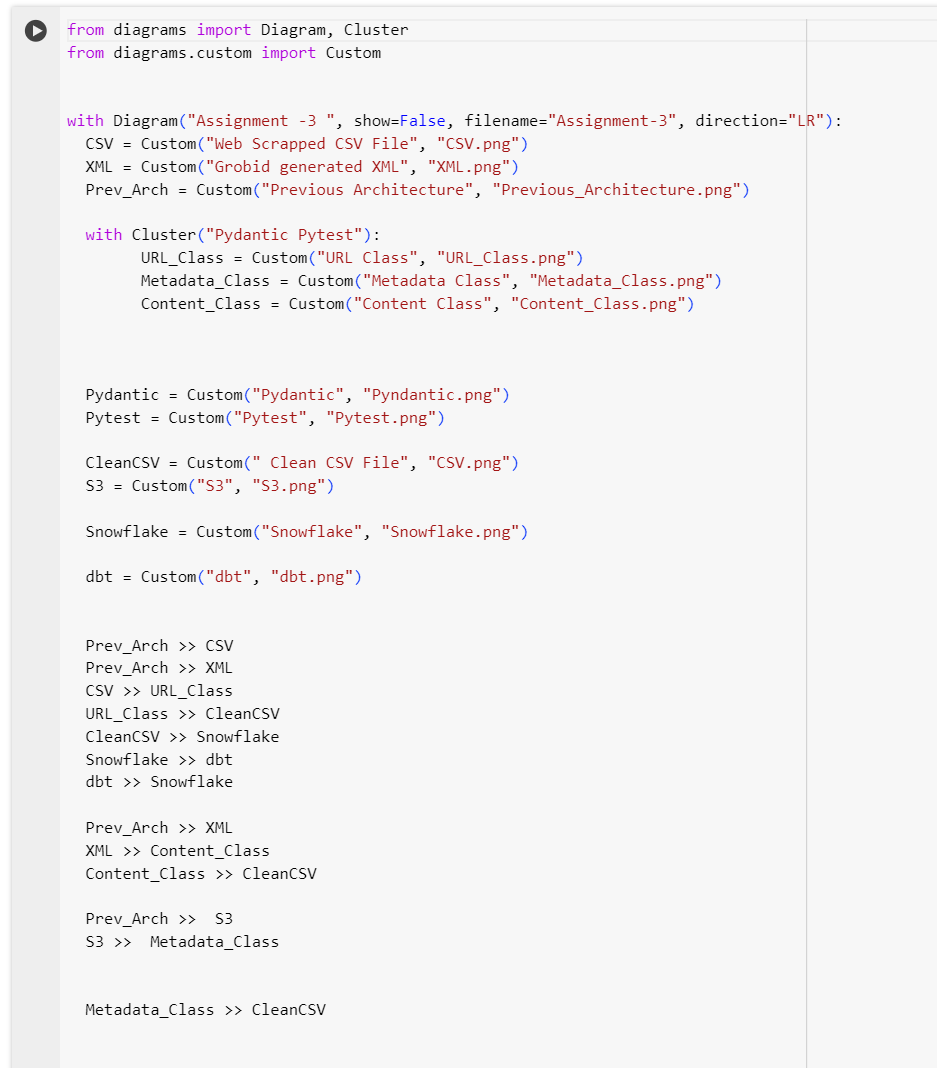
**8. dbt (Data Build Tool):**

Functionality: A command-line tool that enables data analysts and engineers to transform data in their warehouse more effectively.

Steps: dbt is used to transform the data within Snowflake, applying business logic, creating summary tables, and preparing the data for analysis.

#### **Workflow**





#### 

# **Web Scraping**

#### **Overview**:

This script is designed to scrape educational content from a specified set of URLs. It navigates through web pages to collect links to reading materials, extracts relevant information from each reading, and then saves this data into a CSV file. The primary focus is on gathering information such as the topic name, publication year, difficulty level, introduction, learning outcomes, summary, and links to the summary page and PDF files.

#### **Dependencies:**

1. **Selenium:** A web automation tool used to programmatically navigate through web pages and interact with web elements.
2. **WebDriver-Manager:** Helps to manage browser drivers easily, ensuring the correct version of a driver is used.
3. **BeautifulSoup4:** A library for parsing HTML and XML documents, used here to extract and manipulate data from web pages.

#### **Setup:**

1. Install the required packages using pip:

pip install selenium webdriver-manager beautifulsoup4

1. This script uses Chrome and webdriver-manager to automatically handle driver requirements.

#### 

#### **Usage:**

To run the script, simply execute it in a Python environment. The script performs the following steps:

1. Initializes a Selenium WebDriver for Chrome.

2. Iterates through a list of URLs, each pointing to a page with links to reading materials.

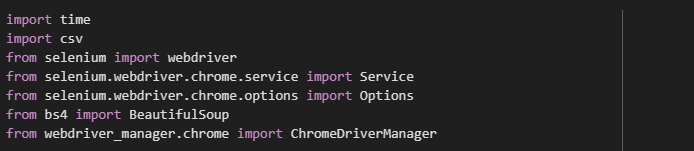
3. Collects all reading links from each page.

4. Visit each reading link to scrape relevant information.

5. Saves the collected data into a CSV file named Team05.csv.

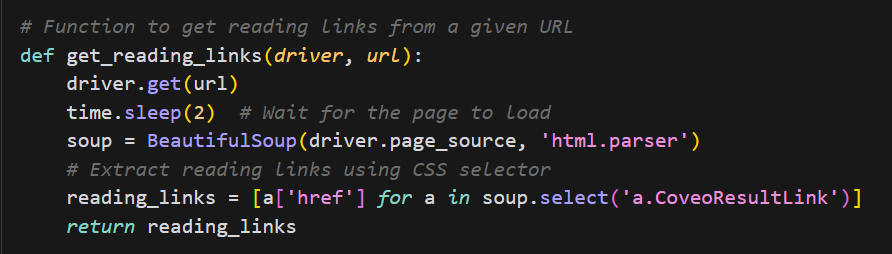
#### **Functions:**

1. ***setup\_driver():***



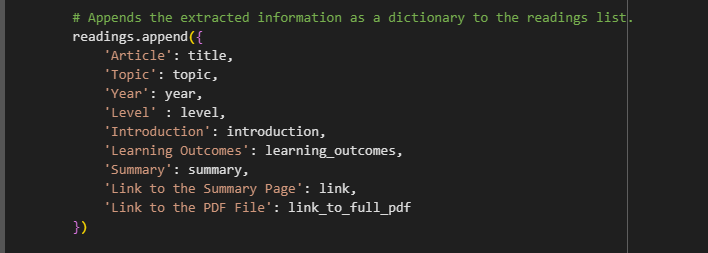
* 1. This function initializes and configures a Chrome WebDriver instance. It sets up the necessary options for the WebDriver and uses ChromeDriverManager().install() to automatically manage the driver binary required for Chrome. This simplifies the process of setting up Selenium by ensuring you always have the correct version of the driver without manually downloading it. The function then returns the configured WebDriver instance for use in navigating and interacting with web pages.

1. ***get\_reading\_links(driver, url):***



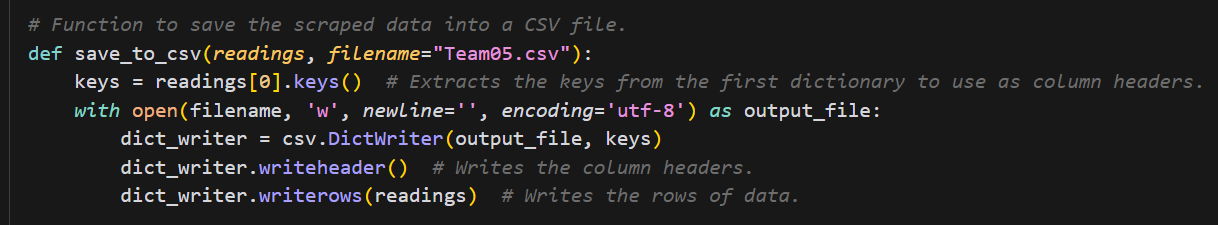
* 1. This function navigates to the specified URL and waits for the page to load. It then uses BeautifulSoup to parse the page's HTML content. The function looks for all anchor tags (<a>) with a class of CoveoResultLink, which are assumed to be the links to the reading materials on the page. It extracts the href attribute (the URL) from each of these anchor tags and returns a list of these URLs. This list represents the reading materials found on the page.

1. ***scrape\_reading\_content(driver, links):***
   1. This function takes a WebDriver instance and a list of URLs (links) to individual reading materials. For each URL in the list, it navigates to the page, waits for it to load, and then parses the page content using BeautifulSoup. It extracts various pieces of information from the page, such as the title, publication year, difficulty level, introduction, learning outcomes, summary, and links to the full PDF file if available. This is achieved through a combination of searching for specific HTML elements and classes (e.g., <h1> for the title, a span with a class of content-utility-curriculum for the year, etc.) and custom functions like extract\_text\_by\_header and extract\_learning\_outcomes that look for content under specific headers. The extracted information is then compiled into a dictionary for each reading and added to a list of readings, which is returned at the end.



### 

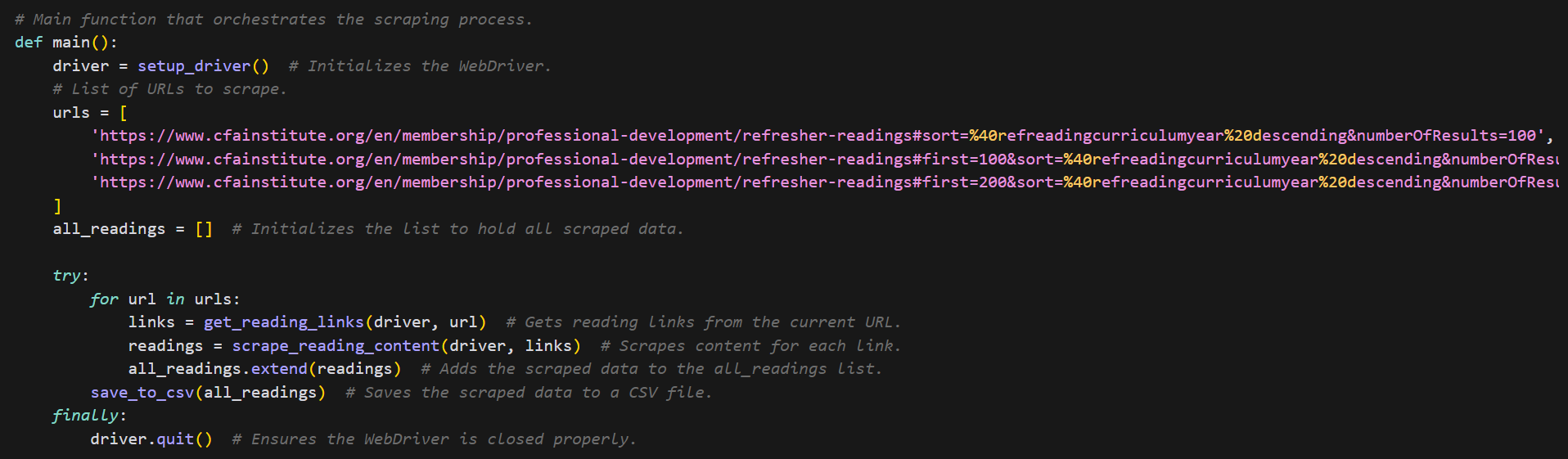
1. ***save\_to\_csv(readings, filename="Team05.csv"):***



* 1. This function takes the list of dictionaries (readings) containing the scraped data for each reading material and a filename for the CSV file. It uses Python's csv module to write this data into a CSV file. The keys from the first dictionary in the list are used as column headers. Each dictionary in the list represents a row in the CSV, with the dictionary values corresponding to the cell values under their respective headers.

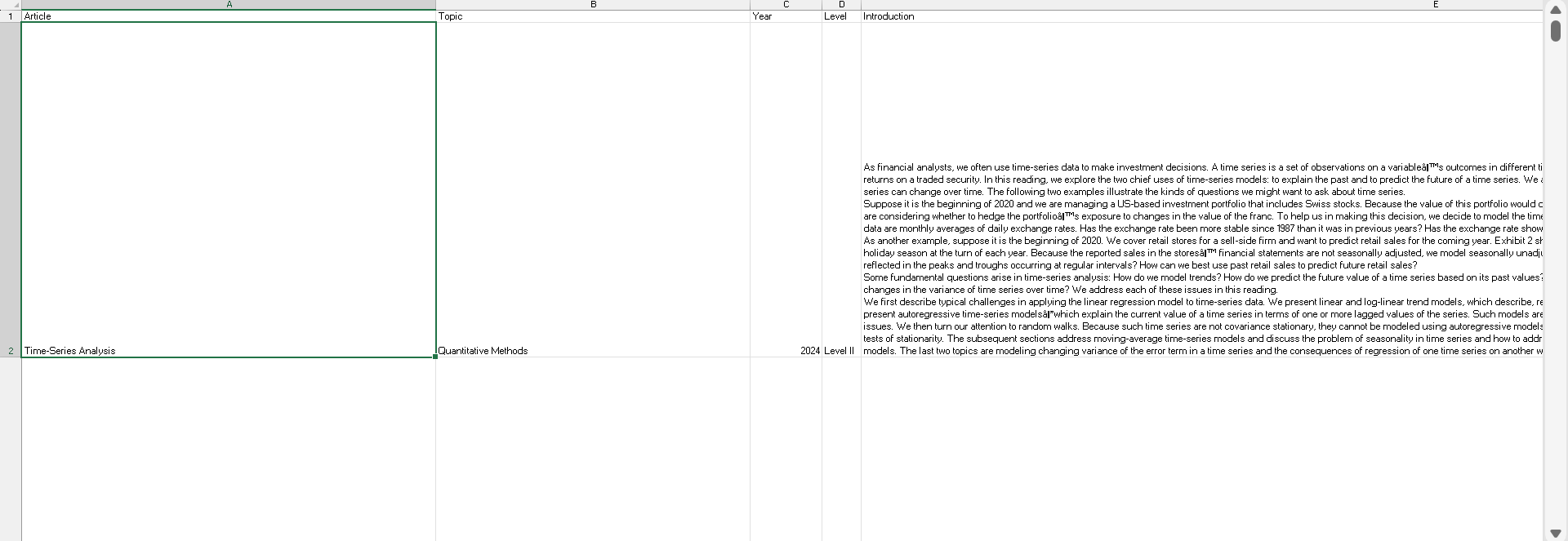
### 

1. ***main():***



* 1. The main function orchestrates the entire scraping process. It initializes the WebDriver, defines the URLs to scrape, and iterates through these URLs to collect links to reading materials. It then scrapes the content from each link and aggregates all the scraped data. Finally, it saves this data to a CSV file and ensures the WebDriver is properly closed, regardless of whether the scraping process completes successfully or encounters an error.

#### **Output:**



The output of the whole scrapping is stored in csv as requested in the assignment. The above image is just an example of how the output is being stored. The columns that have been generated are Name of the topic, Year, Level, Introduction, Learning Outcomes, Summary, Link to the Summary Page and Link to the PDF file.

**Pydantic**

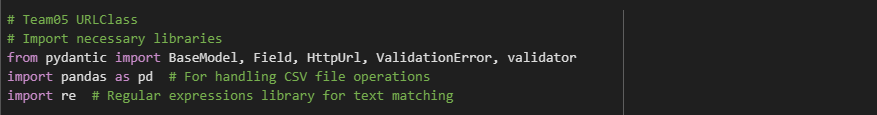
# **URLClass**

**Introduction**

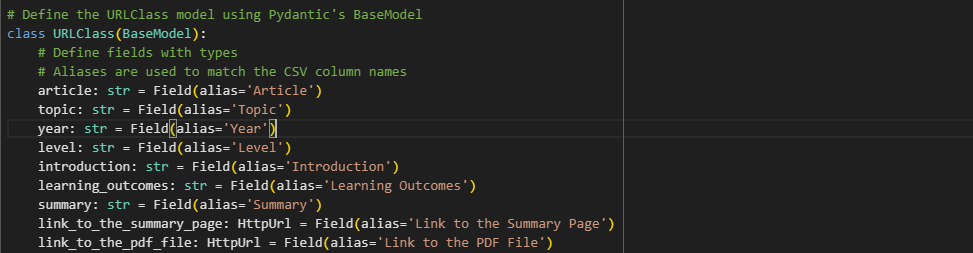
Pydantic is a data validation and settings management library in Python. It is primarily used for data validation, serialization, and deserialization in Python applications. Pydantic provides a way to define data schemas using Python classes, allowing you to enforce data validation rules and ensure the correctness of your data.

## **Installation and Setup**

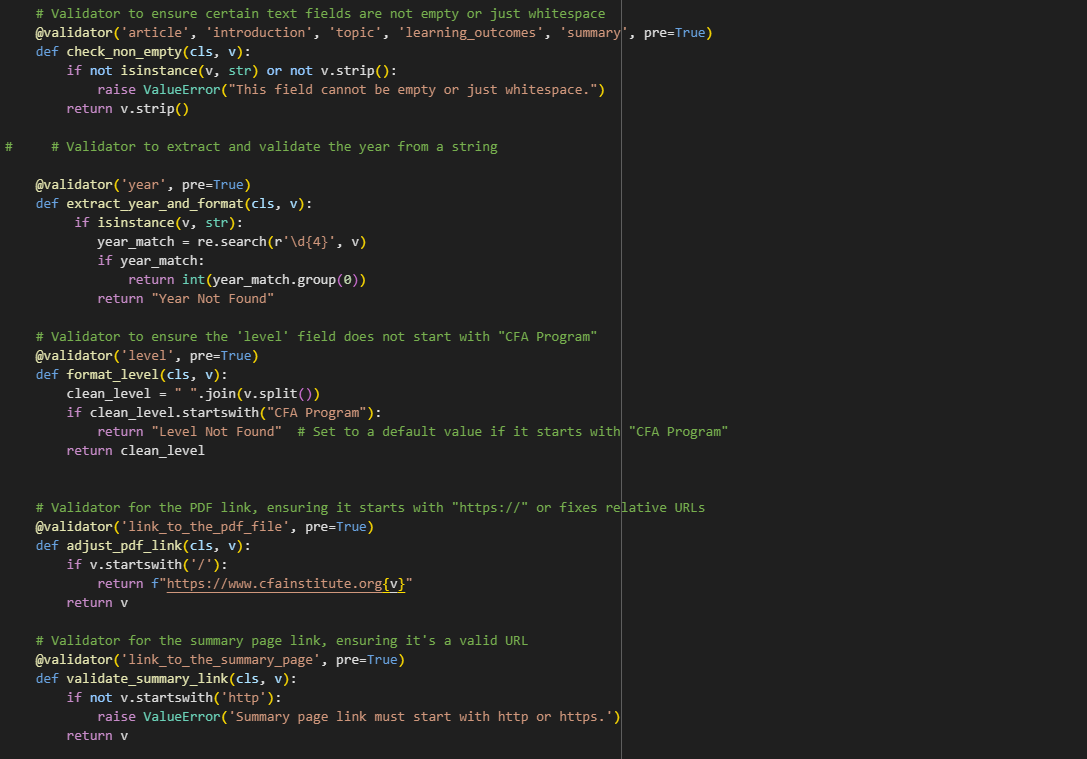
1. Importing Libraries: The code begins by importing necessary libraries such as BaseModel and Field from Pydantic, pandas for handling CSV file operations, and re for regular expressions.



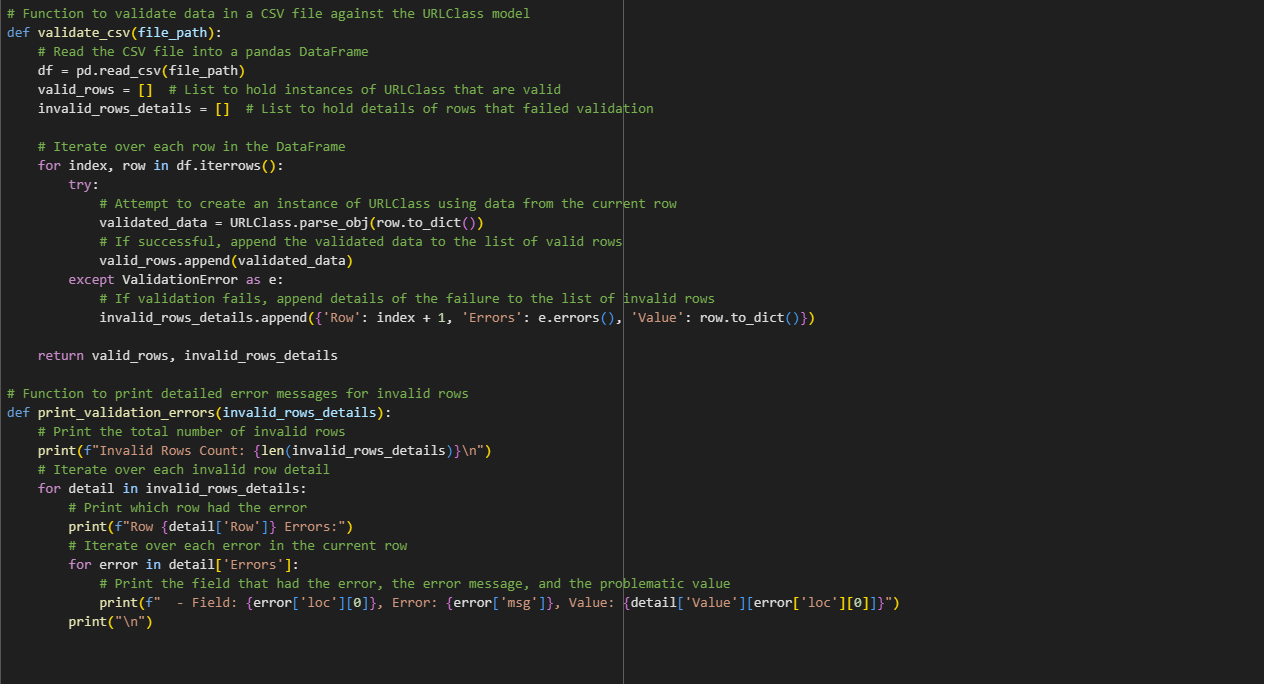
1. Defining the Model: The URLClass is defined as a subclass of BaseModel. It includes fields representing various attributes like article, topic, year, level, etc. Each field is assigned an alias to match the corresponding column names in a CSV file.



1. Validators: Validators are methods within the URLClass class that enforce specific validation rules on the fields. These validators are used to ensure that certain fields are not empty, extract and format the year, format the level, adjust PDF links, and validate summary page links.



1. CSV Validation Functions: Two functions are defined for validating CSV data against the URLClass model. validate\_csv reads the CSV file into a pandas DataFrame, iterates over each row, attempts to create an instance of URLClass using the data from the row, and captures both valid and invalid rows. print\_validation\_errors is a helper function to print detailed error messages for invalid rows.



1. Defining transform\_and\_validate\_data Function:

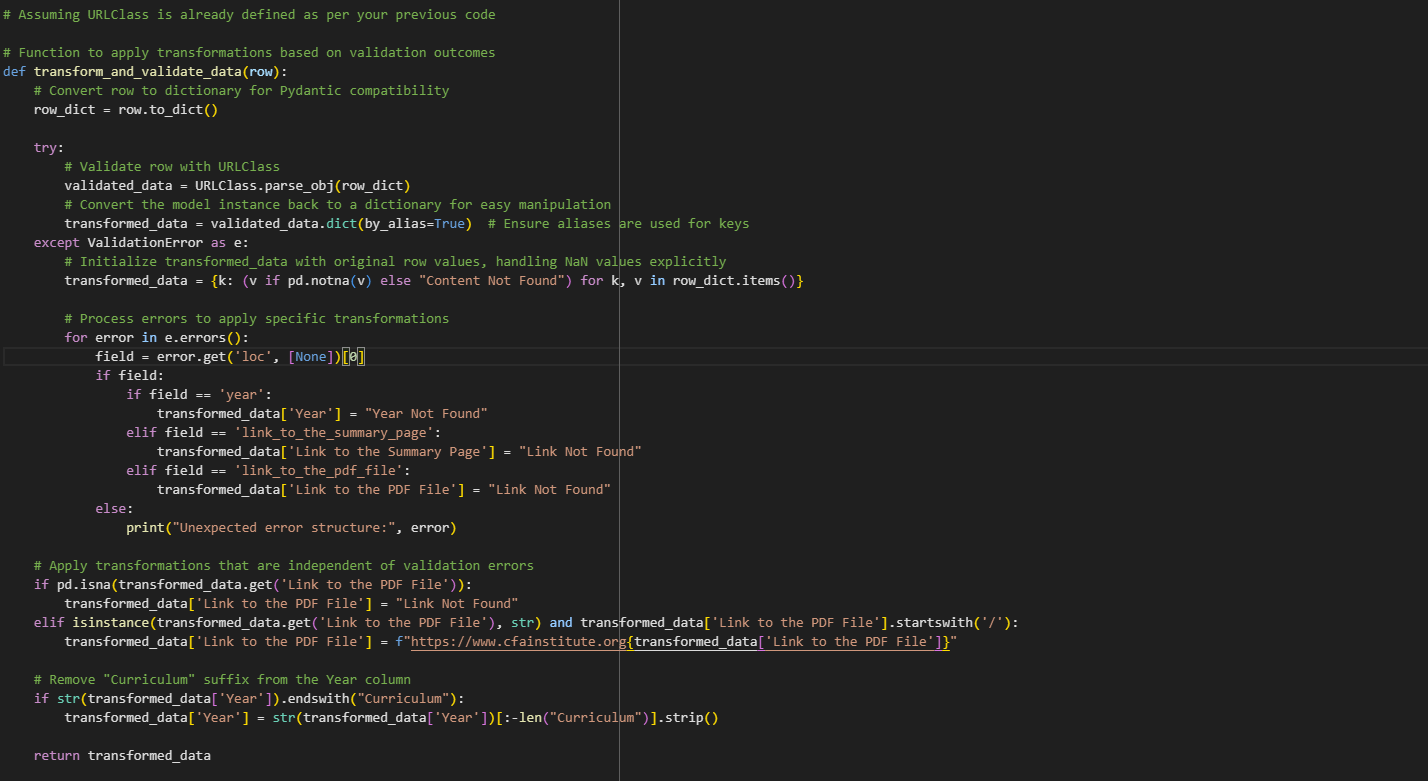
This function takes a row from the original CSV file as input and performs transformations and validations on it.

It converts the row to a dictionary (row\_dict) to make it compatible with Pydantic's parse\_obj method.

Inside a try-except block, it attempts to validate the row data using the URLClass model.

If validation succeeds, the validated data is converted back to a dictionary (transformed\_data) using dict(by\_alias=True) to ensure that aliases defined in the URLClass are used as keys.

Finally, it removes the "Curriculum" suffix from the 'Year' column, if present.



1. Loading the Original CSV File:

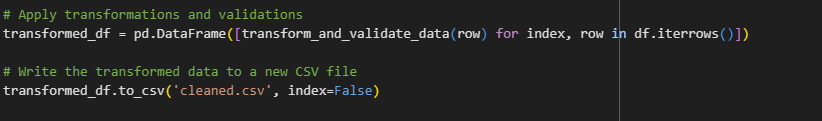
The code reads the original CSV file (Team05.csv) using pd.read\_csv() and stores it in a pandas DataFrame (df).



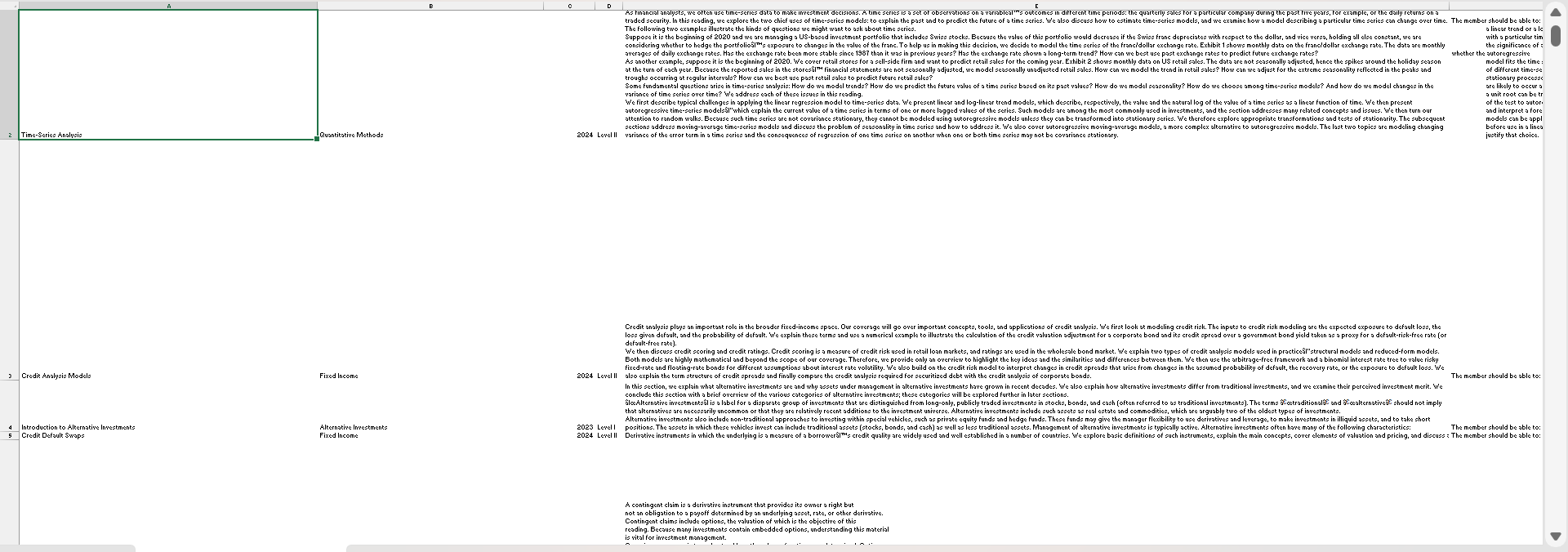
1. Applying Transformations and Validations:

It applies the transform\_and\_validate\_data function to each row of the DataFrame using a list comprehension. The result is a list of transformed dictionaries.

This list of dictionaries is then used to create a new pandas DataFrame (transformed\_df), where each row represents a cleaned and transformed record.



Output CSV File after Validations:



# **ContentClass**

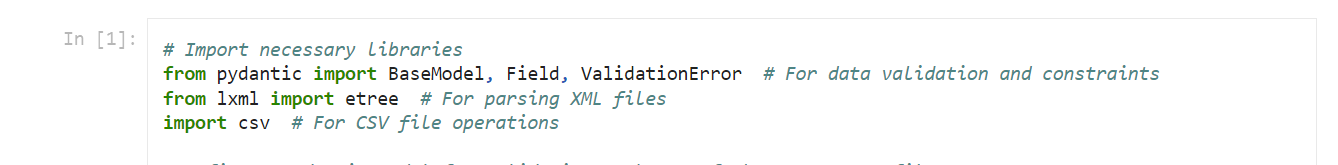
## **Introduction**:

ContentPDFClass is a crucial component designed to capture and organize extracted content from PDF files, serving as a foundational element in data processing pipelines. With its capabilities, this class enables the extraction and structuring of textual information from PDF documents, facilitating subsequent analysis, indexing, and manipulation. By encapsulating functionalities for parsing and storing PDF content, ContentPDFClass empowers developers to efficiently manage and utilize textual data extracted from diverse PDF sources. Its robust architecture ensures reliability and scalability in handling PDF data, making it an indispensable tool for applications requiring seamless integration of PDF content into broader workflows and systems.

## **Installation and Setup**

1. Pull the Libraries Import:

* pydantic: Used for data validation and defining constraints for the data model.
* lxml.etree: Utilized for parsing XML files using XPath expressions.
* csv: Required for performing operations on CSV files.



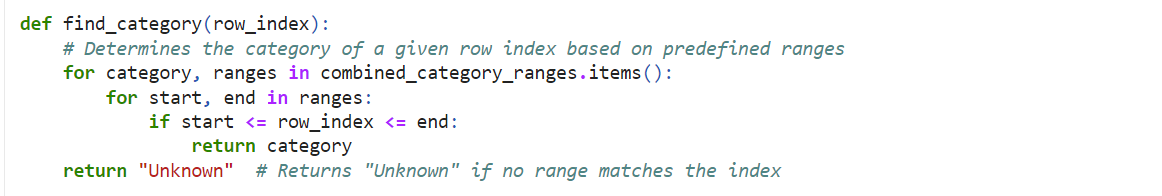
1. Definition of RowModel:

This Pydantic model defines the structure and validation rules for each row of the output CSV file. It enforces constraints such as the Level being an integer between 1 and 3, and requires fields like Category, Topic, and Learning Outcomes.



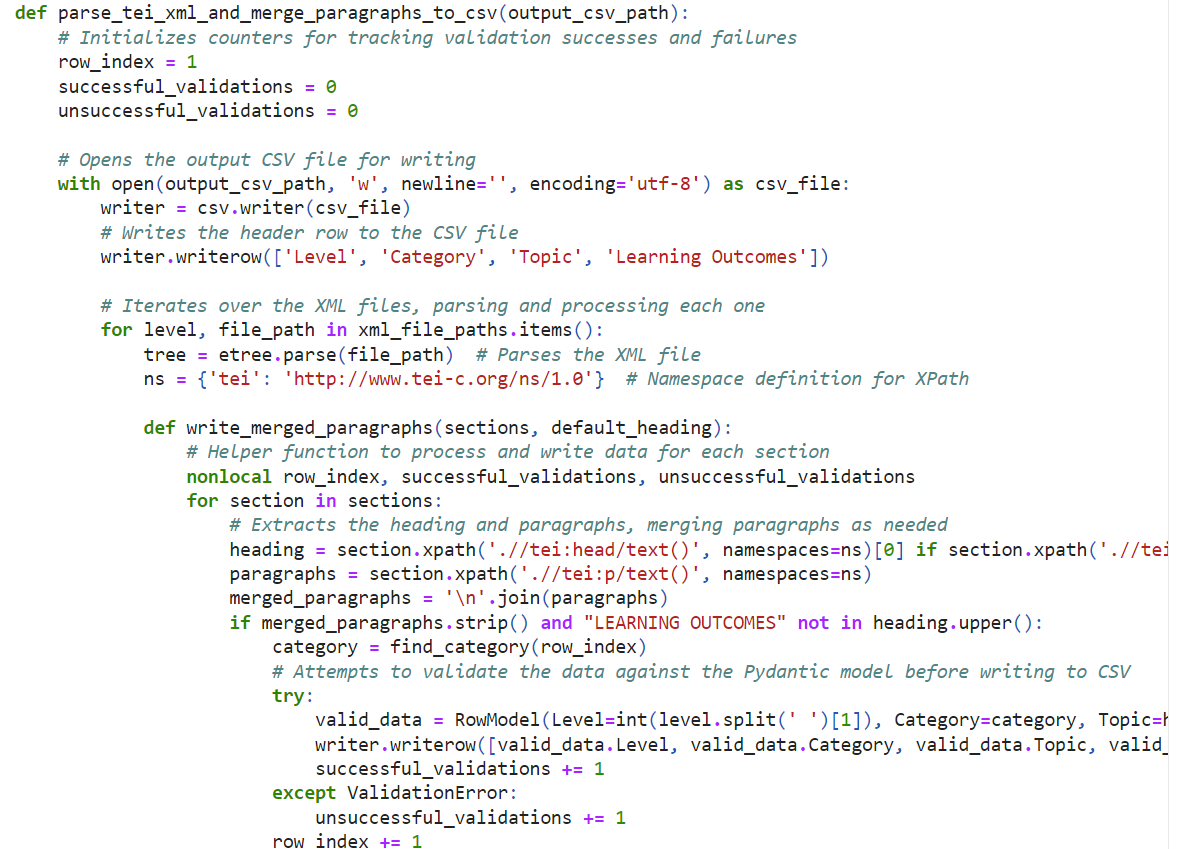
1. Category Ranges Mapping:

Maps categories to row index ranges for categorization purposes. Each category is associated with one or more tuples of start and end row indices.



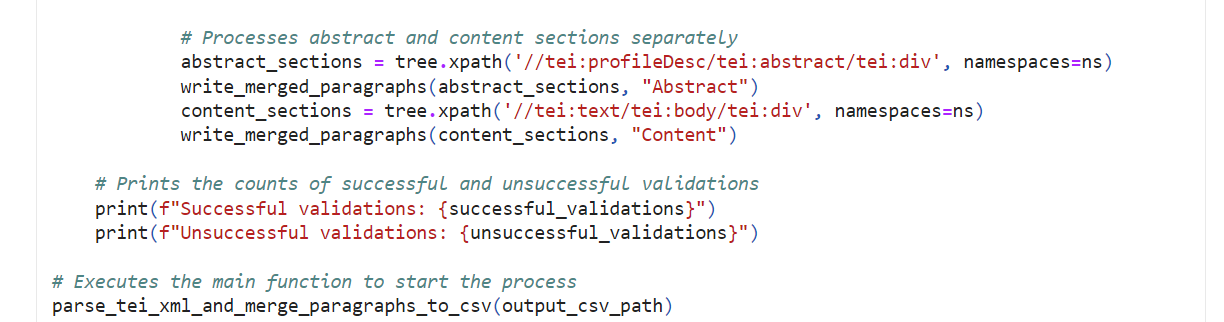
1. parse\_tei\_xml\_and\_merge\_paragraphs\_to\_csv Function:

* This function performs the main processing:
* Opens the output CSV file for writing.
* Iterates over each XML file, parsing and processing its contents.
* Extracts sections such as abstracts and content.
* Merges paragraphs within each section.
* Validates the merged data against the RowModel.
* Writes the valid data to the CSV file.
* Tracks successful and unsuccessful validations.

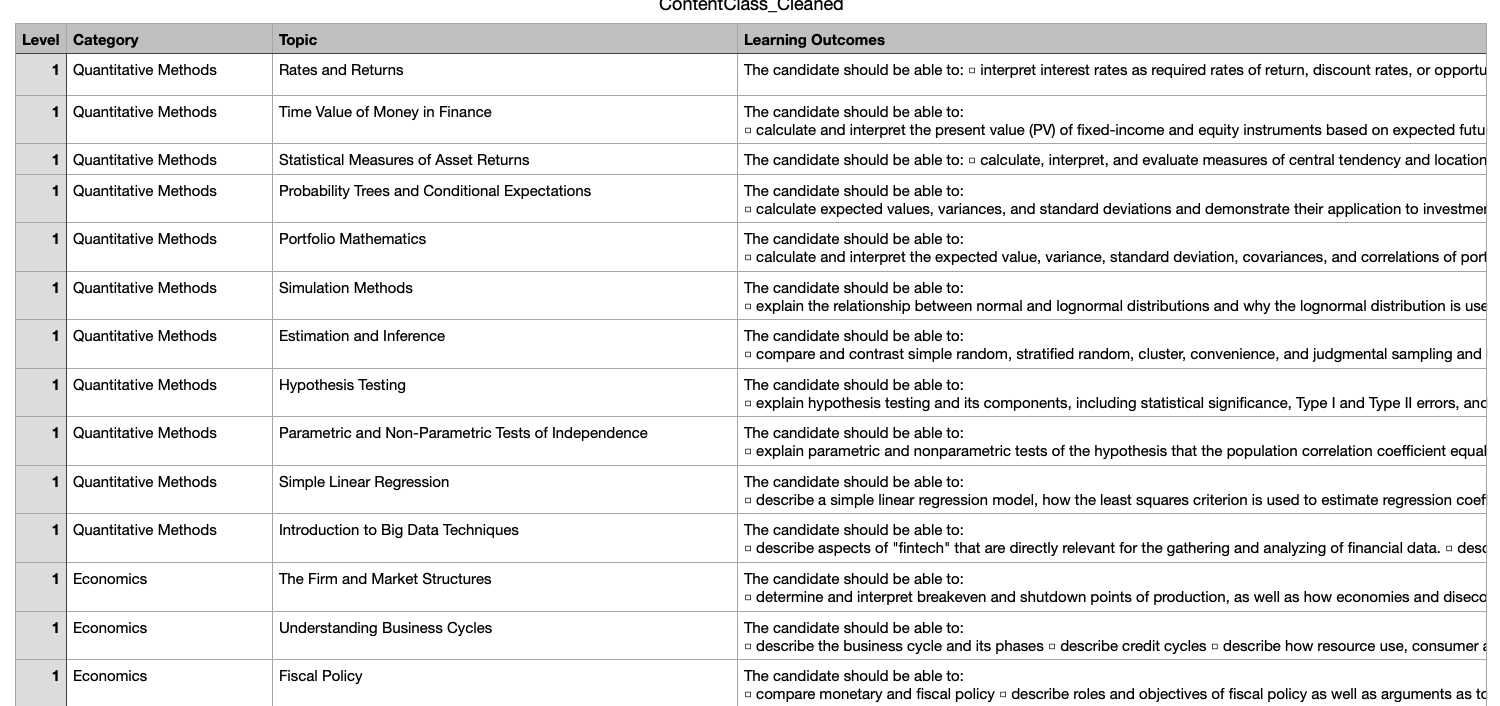


5. Execution:

* Calls the parse\_tei\_xml\_and\_merge\_paragraphs\_to\_csv function to initiate the processing.
* Outputs the counts of successful and unsuccessful validations.



### Cleaned CSV Output Snippet - ContentClass



# **MetaDataClass**

Steps:

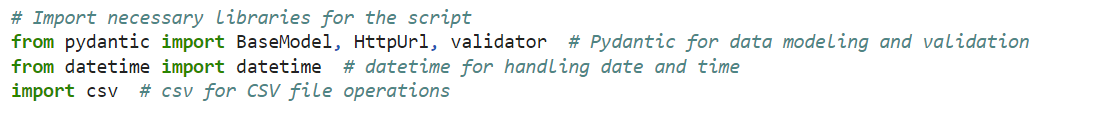
1. Import Necessary Libraries:

The script begins by importing the required libraries:

- `BaseModel`, `HttpUrl`, and `validator` from Pydantic for data modeling and validation.

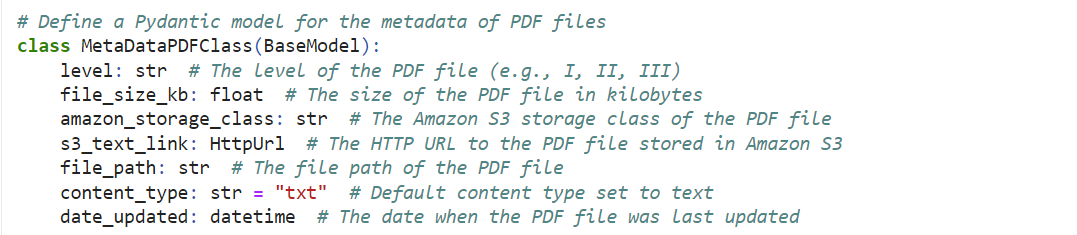
- `datetime` from the datetime module for handling date and time.

- `csv` for CSV file operations.



2. Define Pydantic Model:

The `MetaDataPDFClass` is defined as a Pydantic model using the `BaseModel` class. It specifies attributes representing metadata for PDF files such as `level`, `file\_size\_kb`, `amazon\_storage\_class`, `s3\_text\_link`, `file\_path`, `content\_type`, and `date\_updated`. Some attributes have default values, and validators are defined using the `@validator` decorator to ensure data integrity.



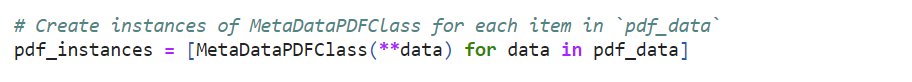
3. Define Test Data:

Metadata for multiple PDF files is defined as a list of dictionaries named `pdf\_data`. Each dictionary contains metadata attributes for a single PDF file.



4. Create Instances of the Pydantic Model:

Using list comprehension, instances of the `MetaDataPDFClass` are created for each item in `pdf\_data`. This ensures that each PDF's metadata adheres to the defined model's structure and constraints.



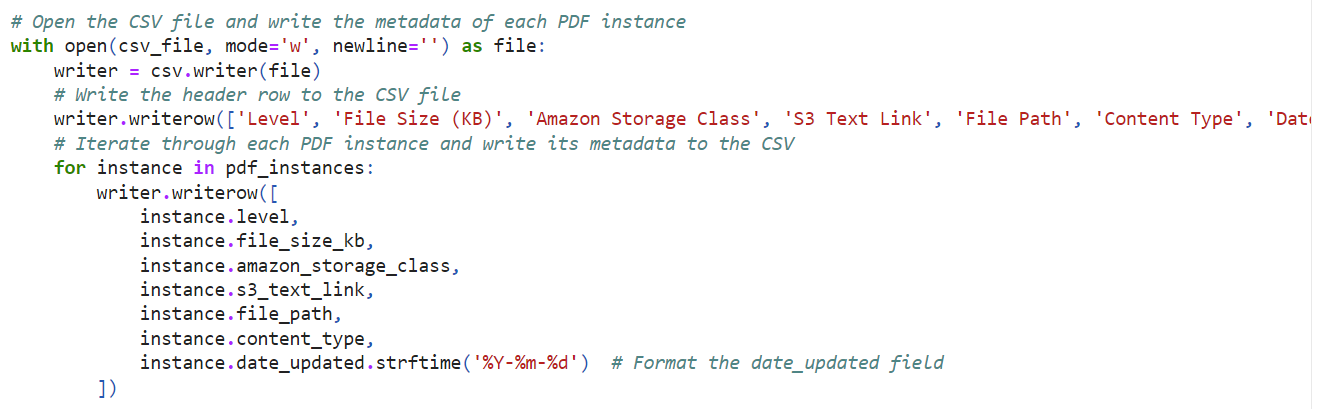
5. Specify Output CSV File:

The output CSV file's path is specified using a variable named `csv\_file`.



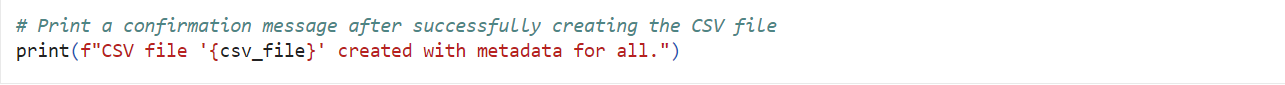
6. Write Metadata to CSV File:

The script opens the CSV file in write mode and writes the metadata of each PDF instance to the CSV file row by row. It first writes the header row containing column names and then iterates through each instance, writing its metadata values to the CSV file.



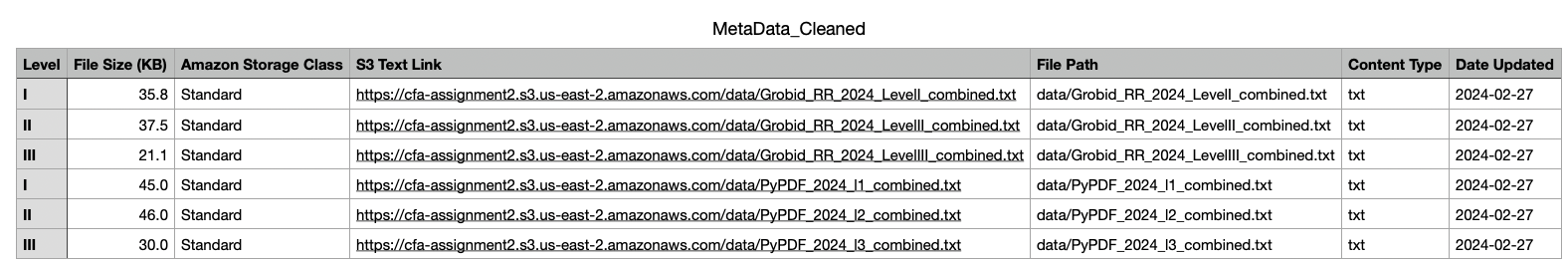
7. Confirmation Message:

Finally, a confirmation message is printed indicating that the CSV file has been successfully created with metadata for all PDF files.



### 

### Cleaned CSV Output Snippet - MetadataClass



**PyTest**

# 

# **PyTest: URLClass**

**Overview**

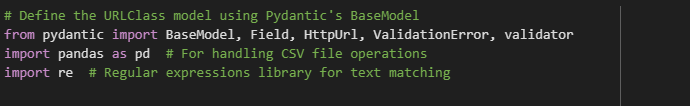
Pytest is a testing framework for Python that allows developers to write simple and scalable test cases. It provides a wide range of features for writing and organizing tests efficiently. Pytest is known for its simplicity, ease of use, and compatibility with other testing tools and libraries.

**Steps:**

Here's how you can write and execute test cases for the URLClass using Pytest:

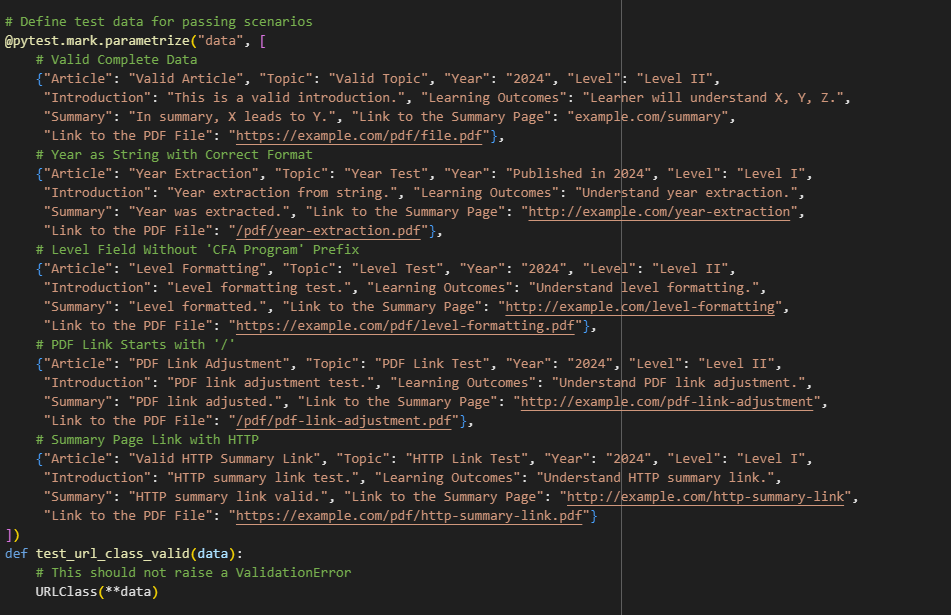
1. Define the URLClass Model:

Define the data model (URLClass) using Pydantic's BaseModel class. Specify the fields along with their types, aliases, and validation rules.



1. Write Test Functions:

Write test functions using Pytest to validate the behavior of the URLClass model under different scenarios. Separate test functions can be created for both passing and failing scenarios.



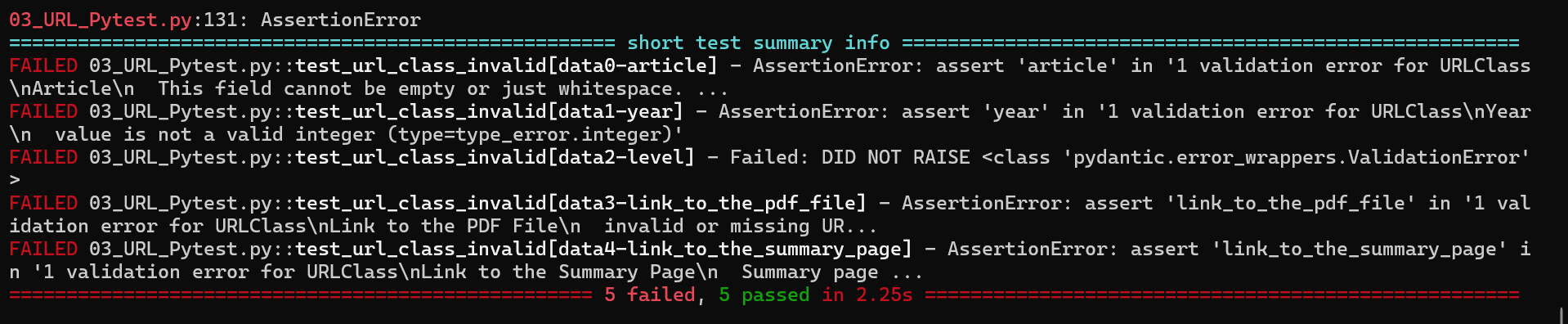
1. Parametrize Test Functions:

Use Pytest's @pytest.mark.parametrize decorator to parametrize test functions with different sets of input data. This allows you to run the same test function with multiple inputs and expected outcomes.



1. Run Pytest:

Execute the test suite using Pytest. Pytest will automatically discover and run the test functions, providing detailed information about the test results, including pass/fail status, error messages, and coverage.



# **PyTest\_ContentClass**

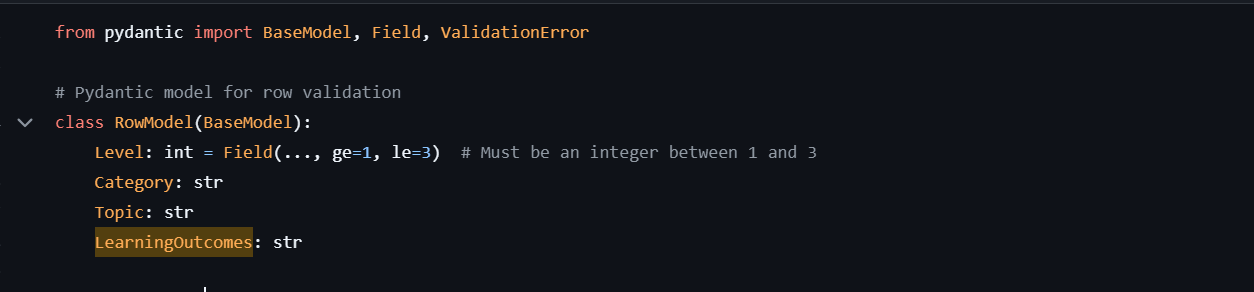
**Introduction:**

The presented code snippet illustrates the utilization of Pydantic, a Python library for data validation, in conjunction with pytest, a testing framework, to ensure the accuracy and integrity of data models. The `RowModel` class defines a Pydantic model with specific constraints, such as integer levels between 1 and 3, ensuring that data adheres to predefined rules. Subsequent test cases employ pytest to verify the behavior of the `RowModel`, encompassing a range of scenarios from valid data inputs to erroneous ones, including missing fields and invalid data types. Through this approach, developers can systematically validate data structures, fostering confidence in the reliability and consistency of data processing within Python applications.

**Steps:**

1. Import Statements:

From pydantic import BaseModel, Field, ValidationError: Imports necessary modules from Pydantic for defining models and fields, and handling validation errors.



1. Definition of RowModel:

This Pydantic model represents a row of data and specifies validation constraints for each field.

Level: An integer field constrained to be greater than or equal to 1 and less than or equal to 3.

Category: A string field representing the category of the topic.

Topic: A string field representing the topic of the row.

LearningOutcomes: A string field representing the learning outcomes associated with the topic.



1. Test Cases:

The code includes 10 test cases to validate instances of the RowModel.

Five test cases (test\_valid\_row\_model\_level\_1 to test\_valid\_row\_model\_max\_level) test valid instances of the RowModel with different levels and data values.

Five test cases (test\_invalid\_row\_model\_level\_below\_min to test\_invalid\_row\_model\_non\_integer\_level) test invalid instances of the RowModel by intentionally violating the validation constraints.

Each test case utilizes the pytest.raises context manager to check if a ValidationError is raised when attempting to create an invalid instance of the RowModel.



# **PyTest: MetadataClass**

### **Valid Test Cases**

The valid test cases are designed to confirm that the MetaDataPDFClass correctly accepts valid data. Each test case is a dictionary of parameters that represent a valid state of the MetaDataPDFClass fields. When the MetaDataPDFClass is instantiated with these parameters (\*\*data), it should not raise any exceptions, indicating that the data conforms to the model's constraints.



### **Invalid Test Cases**

The invalid test cases are designed to ensure that the MetaDataPDFClass raises ValidationError when instantiated with invalid data. Each test case in this set contains at least one field that violates the model's constraints. The with pytest.raises(ValidationError) context manager is used to expect a ValidationError when the model is instantiated with invalid data.

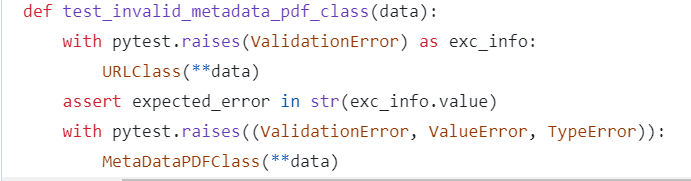


### **Testing Validators**

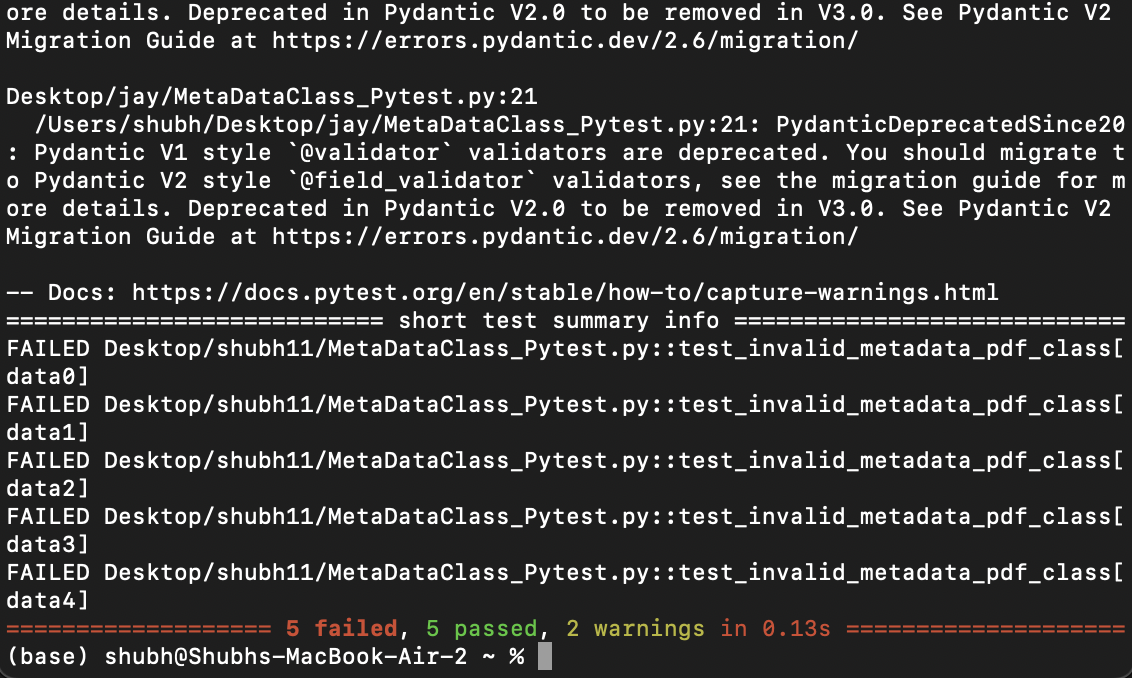
The validators within the MetaDataPDFClass are also implicitly tested here. For example, the test case with a negative file size is expected to fail because the file\_size\_must\_be\_positive validator should raise a ValueError when a non-positive number is provided.

### **Asserting Exceptions**

The assert statement is used within the context manager to ensure that the expected exception is raised. If the exception is not raised, or if a different exception is raised, the test will fail.



Output:



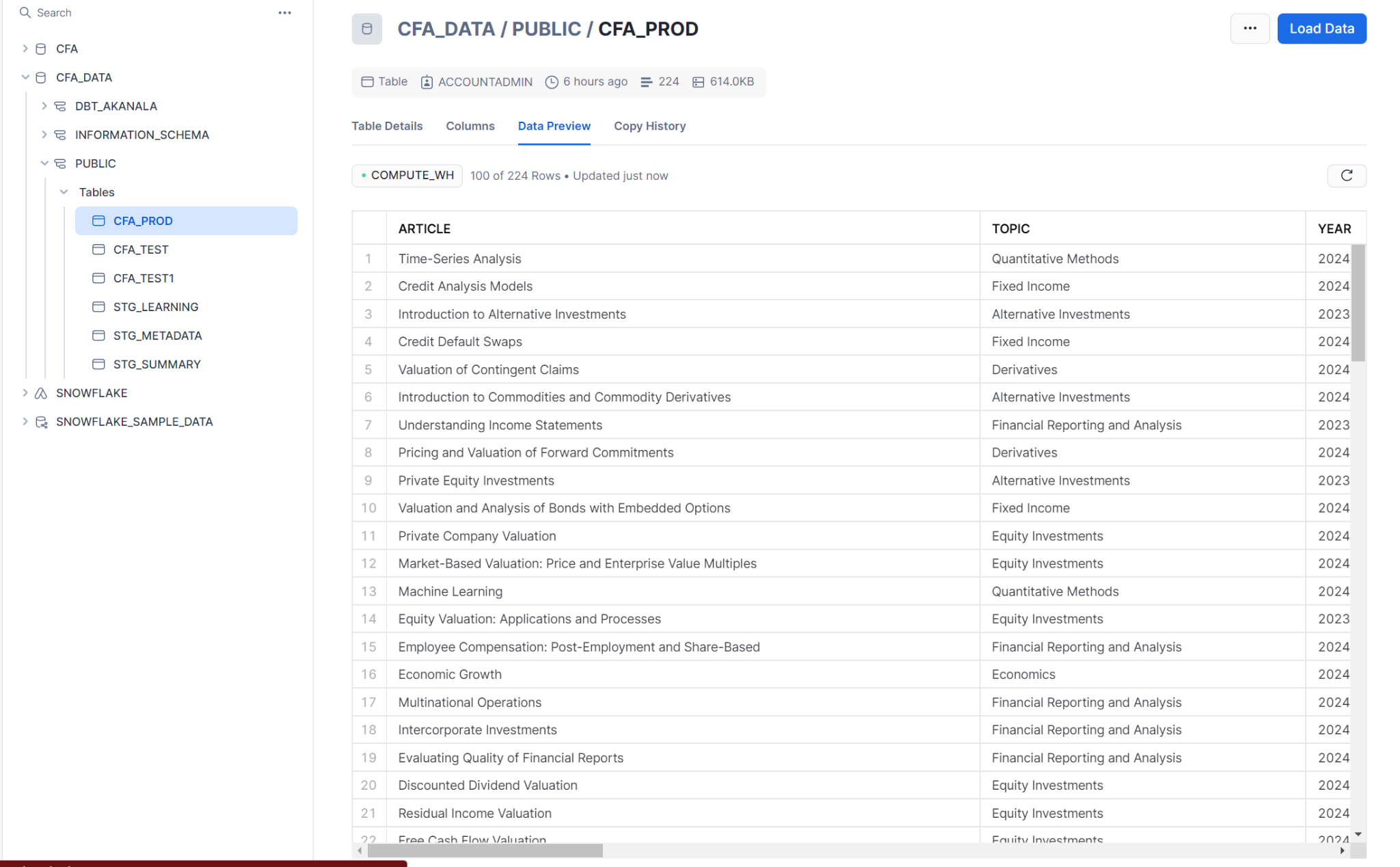
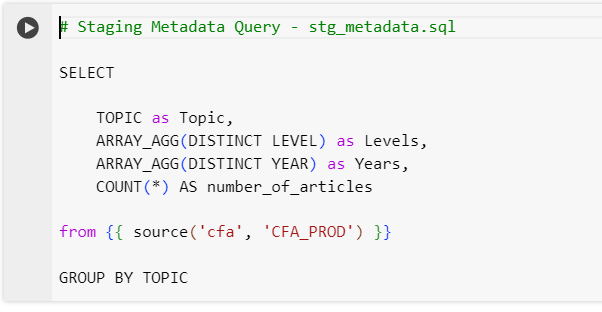
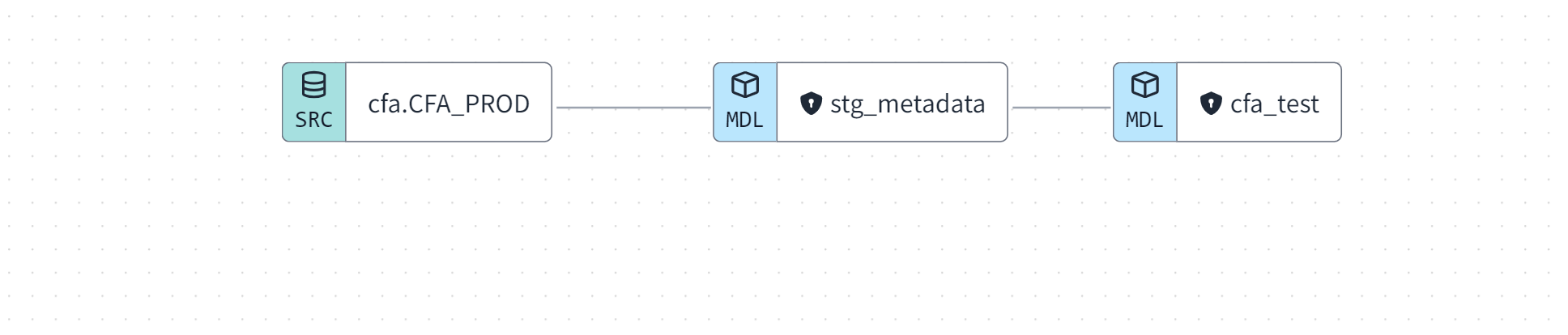
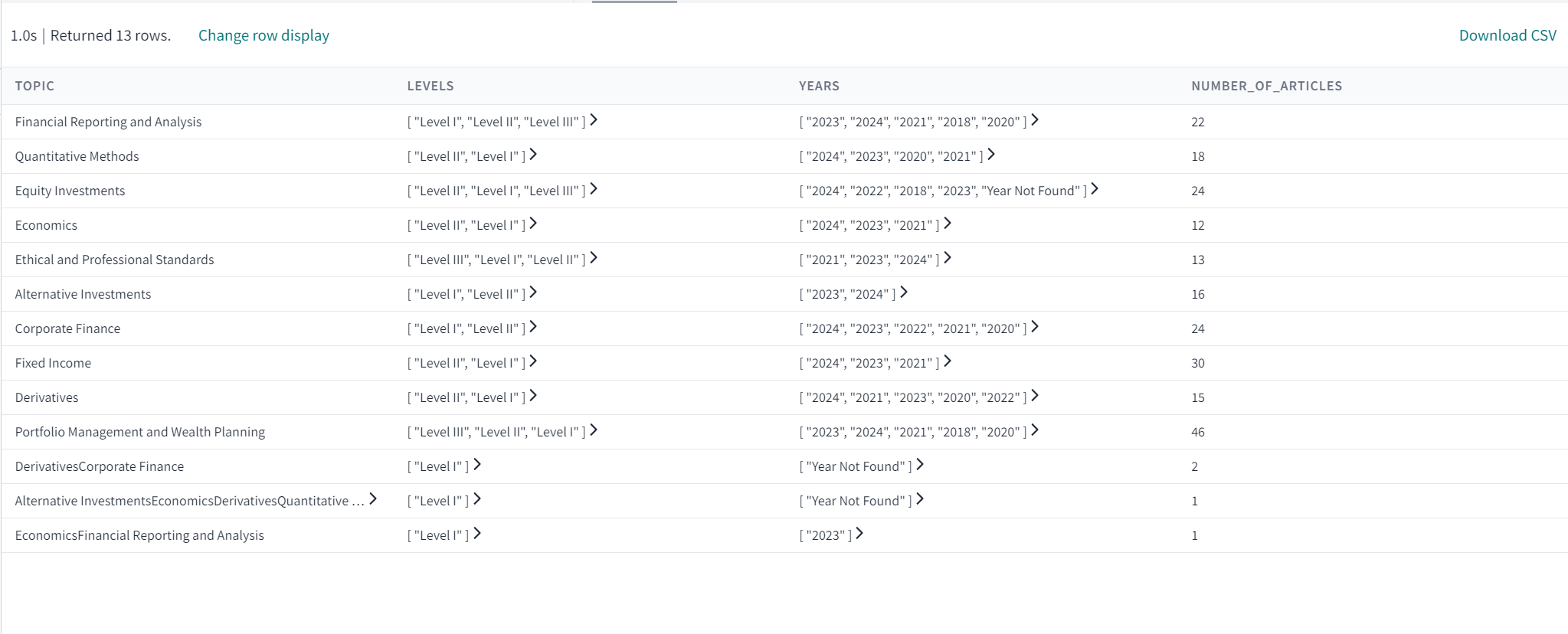
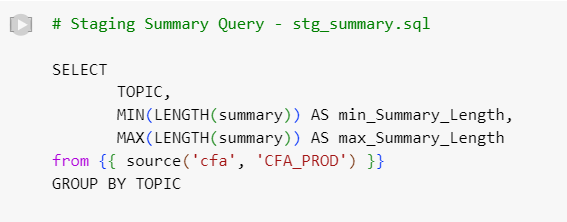
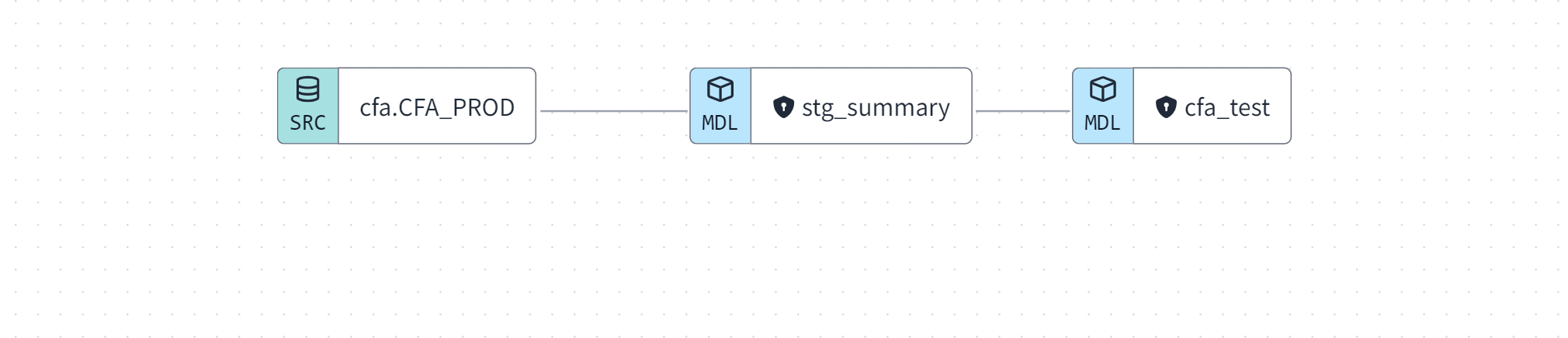
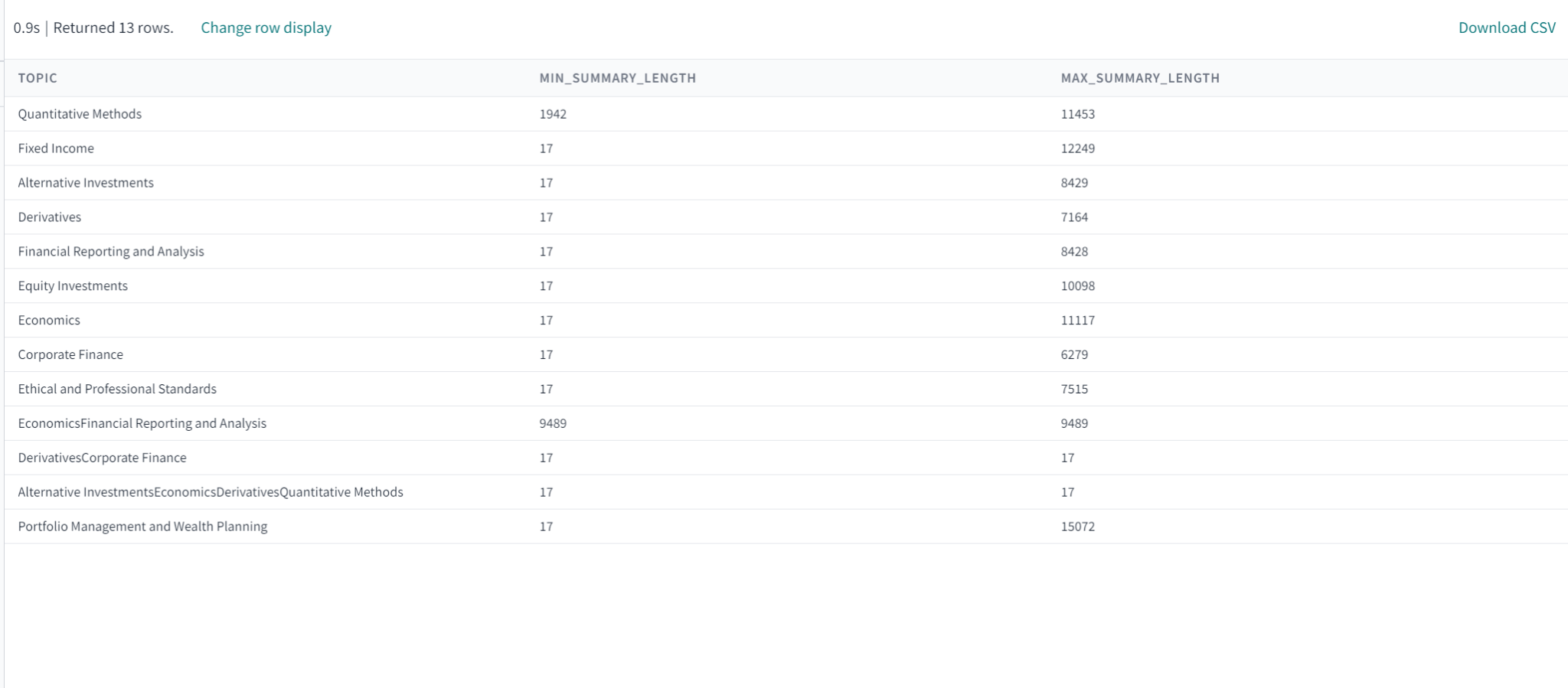
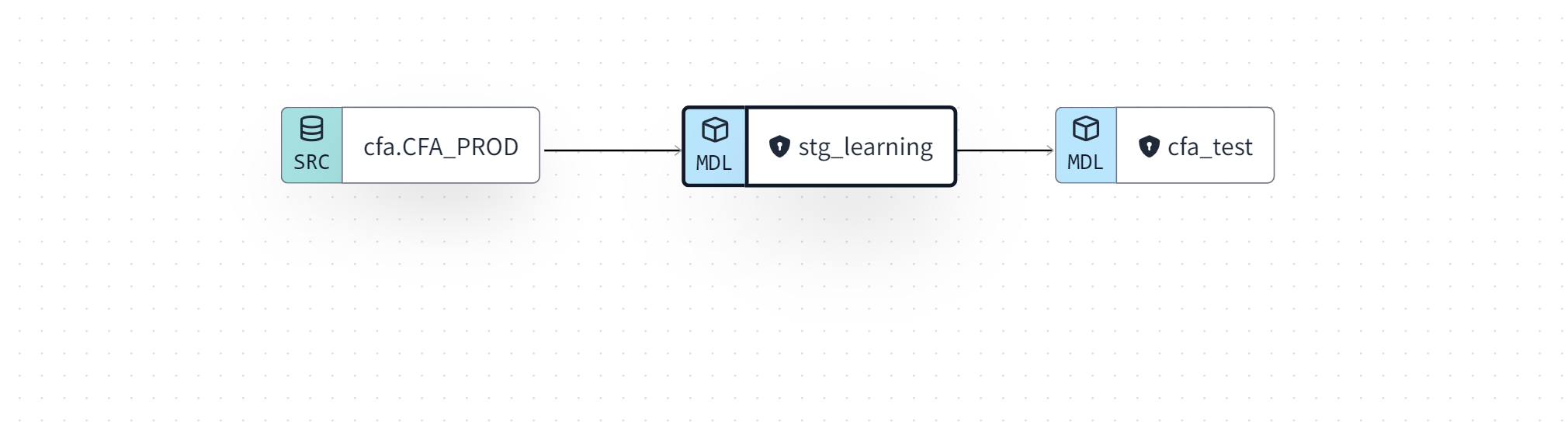
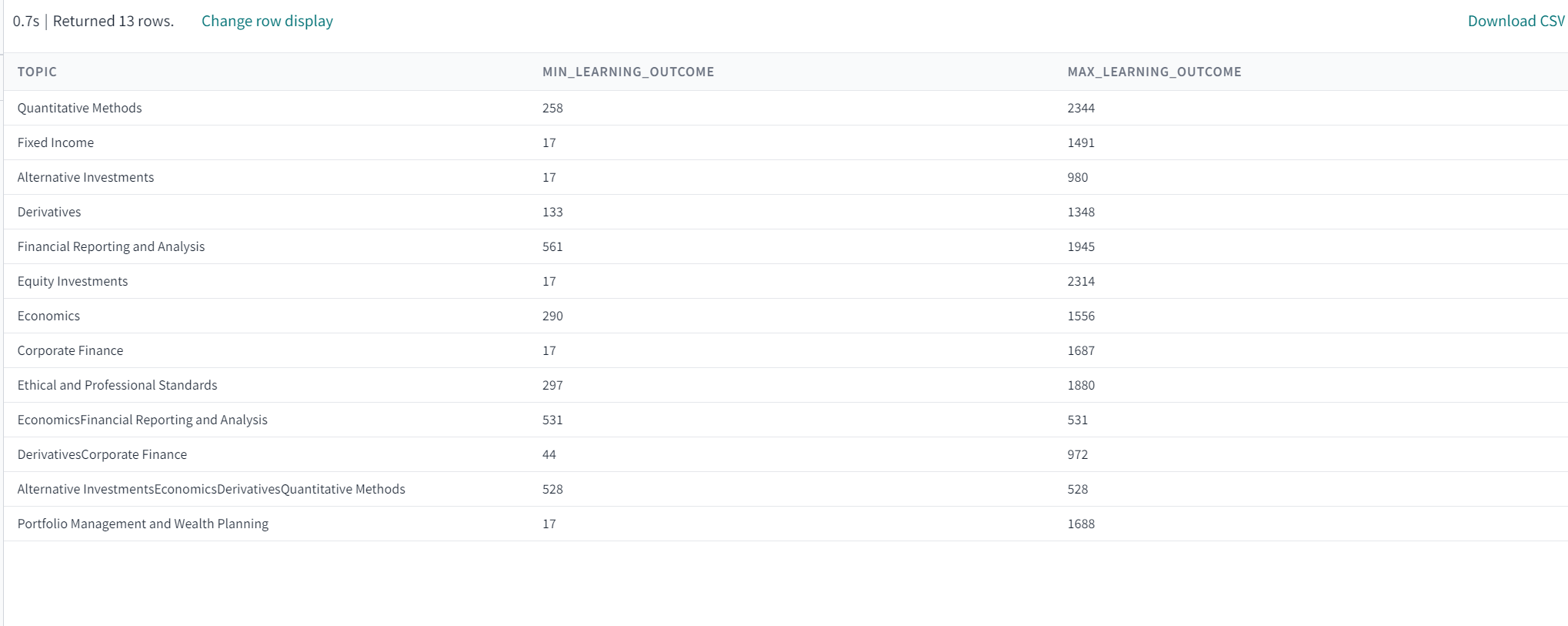
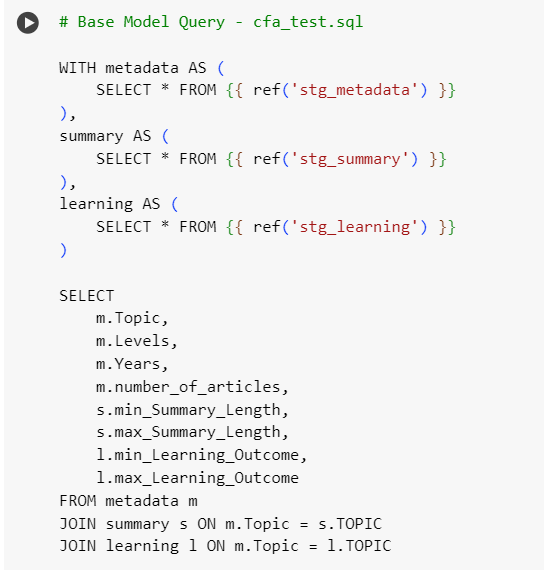
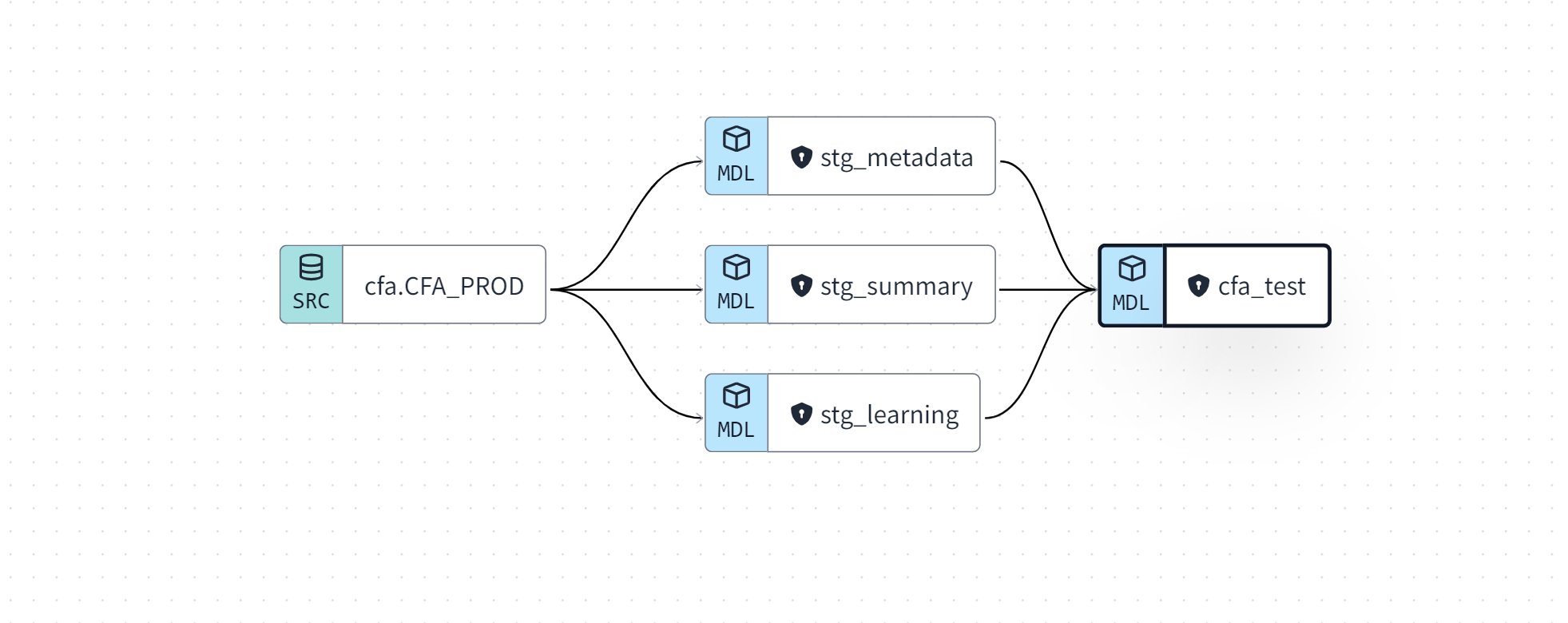
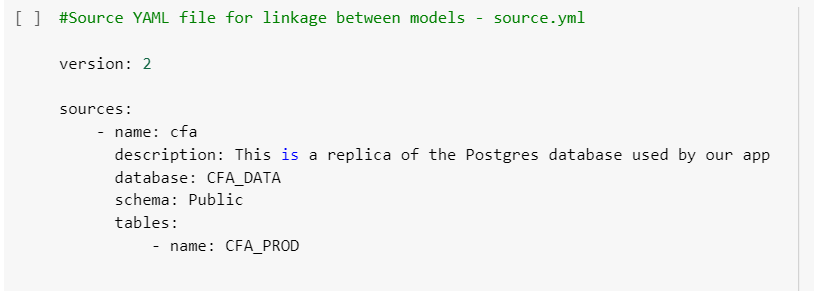
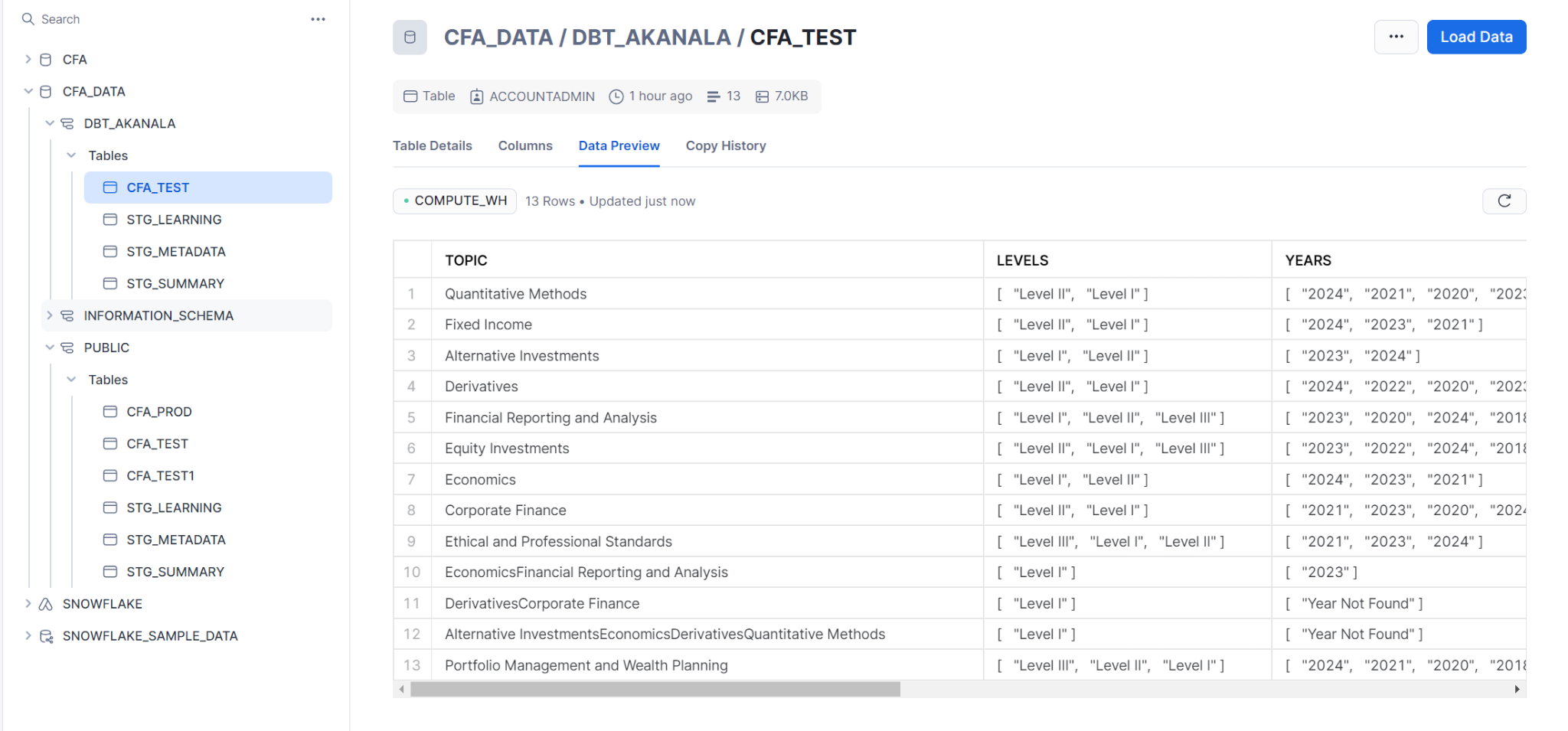
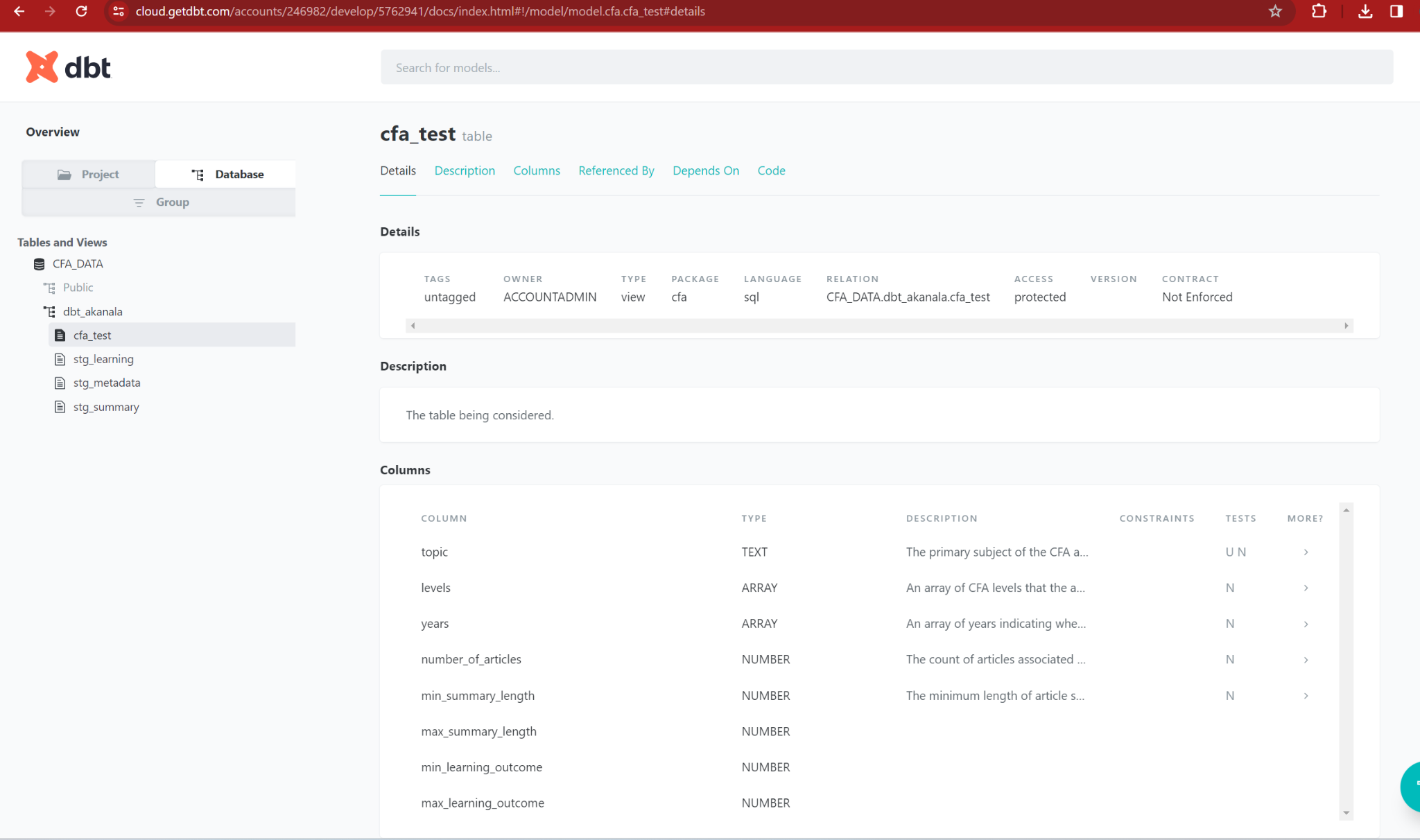
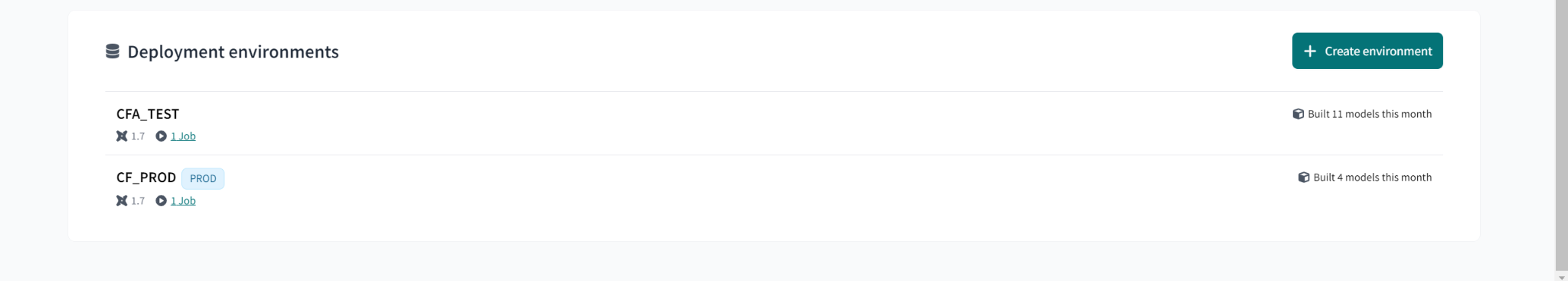
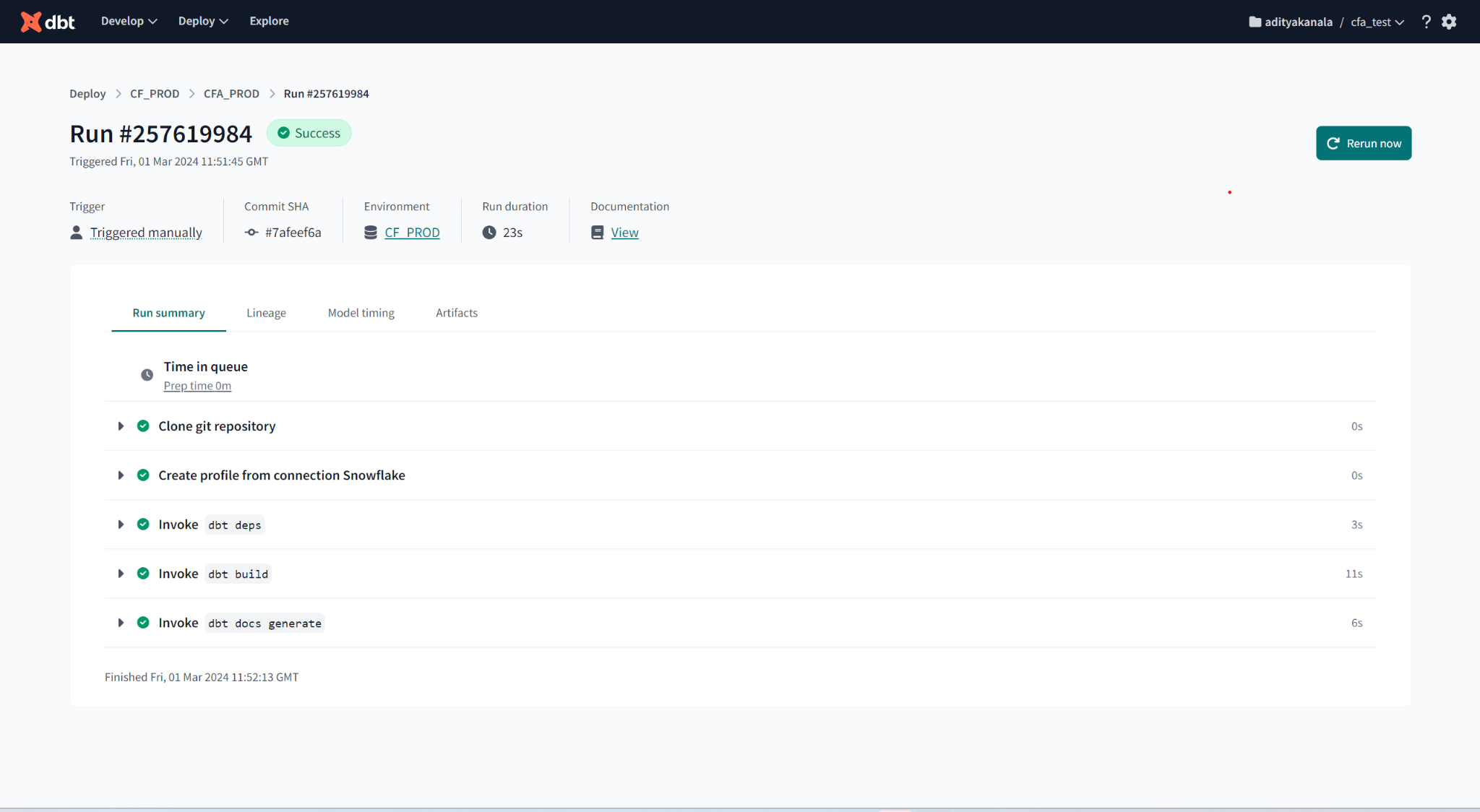
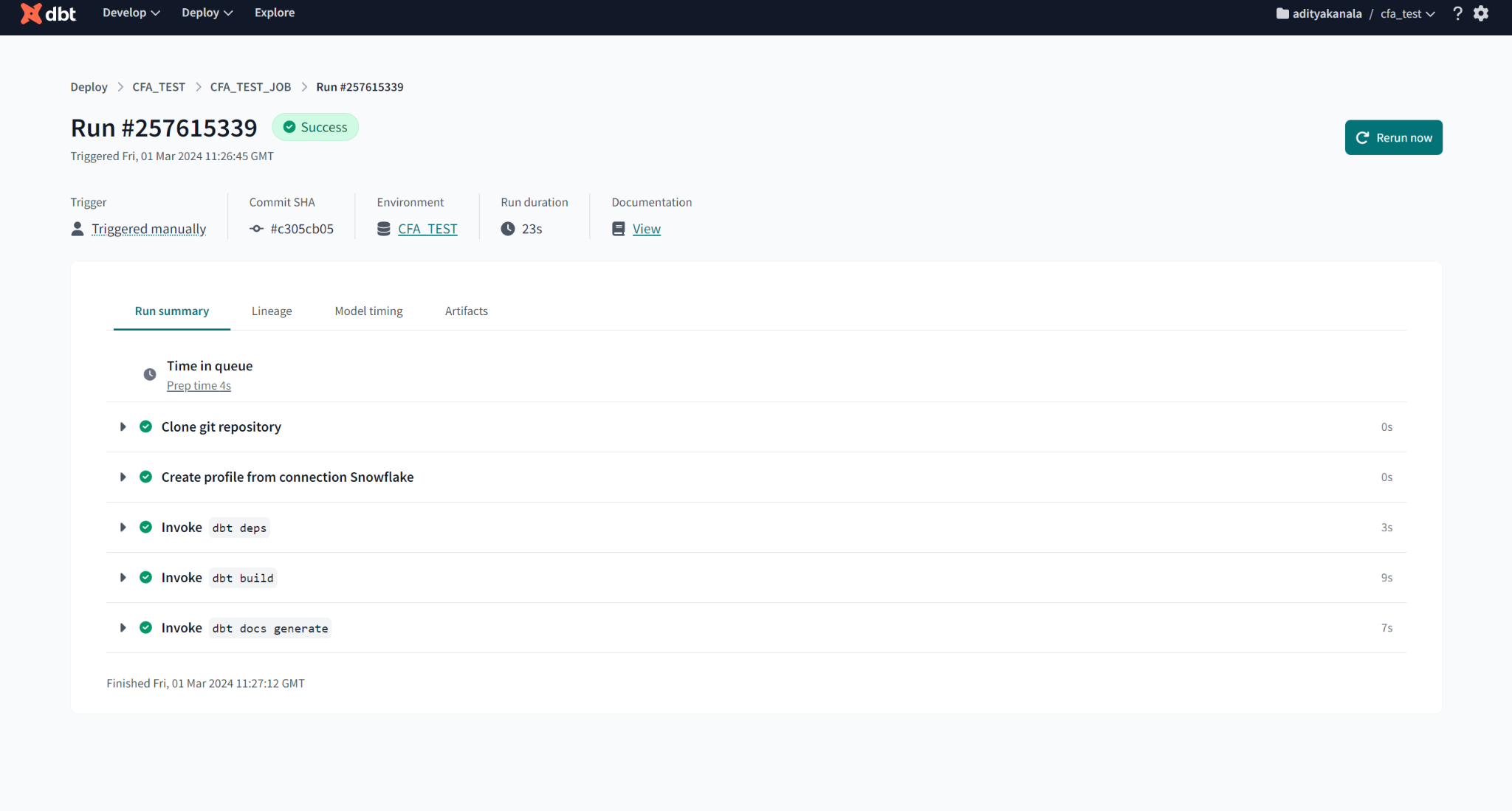
# **DBT Integration**

**Overview:**

In this section, we discuss the SQL scripts designed to upload snowflake data DBT Cloud. This integration facilitates efficient cloud storage of data, enabling easy access for further analysis.

**SQL Script Usage:**

The whole process has been divided into a few steps.

1. Loading the clean CSV files into snowflake by creating 2 environments, Production and Testing.
   1. 
   2. 
2. Writing multiple stage tables in dbt for the required schema..
   1. Metadata Staging - Consists of the fields which depict the high level details of the data.
      1. 
      2. 
      3. 
   2. Summary Staging - Consists of the maximum and minimum length of the Summary for each Topic.
      1. 
      2. 
      3. 
   3. Learning Outcome Staging - Consists of the maximum and minimum length of the Learning Outcomes for each topic.
      1. 
      2. 
      3. 
3. Building Models one above the other so that they are properly referred and related.
   * 1. 
     2. 
     3. 
4. Yaml files for building the models.
   1. 
5. Deploying the final build model in Snowflake. The dbt table named ‘DBT\_AKANALA’ is created in the snowflake which consists of the models that have been deployed.
   1. 
6. Generating the documents for the DBT code by executing the command ‘ dbt get docs’.
   1. 
7. Create 2 Environments in DBT Cloud named, test and prod as per the requirement..
   1. 
8. Create a job in each environment to run all the scripts to deploy models and generate docs automatically..
   1. 
   2. 

**Considerations for Test and Production Environment:**

While working with 2 different environments, our team had the following considerations for performing the model building using the data from Snowflake.

**Separation**: We kept Test and Production data separate to avoid any mix-ups. This usually means separate databases or tables or schemas in Snowflake.

**Access Control**: Limit access to production data to authorized personnel only, and ensure test data doesn't contain sensitive information.

**Version Control**: Use Git branches to manage code changes, with clear paths for merging into the main branch for production deployment.

**Data Validation**: Write dbt tests to check the integrity and accuracy of your data models.

**Monitoring**: Keep an eye on the performance of your Snowflake environments, and try to set up alerts for any issues.

**Documentation**: Keep your model documentation up to date so that everyone knows how the data models work and what they contain.

# **References**

<https://www.getdbt.com/>

<https://docs.getdbt.com/guides/snowflake?step=1>

<https://docs.snowflake.com/en/developer-guide/python-connector/sqlalchemy>

<https://docs.pytest.org/en/8.0.x/>

<https://docs.pydantic.dev/latest/>