ETL Pipeline Project Structure Explained

This document provides a detailed explanation of the ETL pipeline project structure. It outlines every component of the project, what it does, and how everything fits together to create a robust, flexible data processing pipeline.

Table of Contents

---

1. Overview

The project is organized to separate concerns clearly, from pipeline orchestration to SQL templating and data quality checks. This modularity makes it easier to manage, test, and extend the pipeline. The project supports multiple layers (bronze, silver, gold) for data ingestion, transformation, and reporting, following best practices such as Data Vault modeling and star schema for analytics.

---

2. Project Structure

2.1. `pipelines/` Directory

This folder contains the core orchestration scripts and utilities that manage the ETL workflow.

- **`pipeline.py`**
  Primary orchestration script:
  - Traverses the YAML configuration (stages → groups → steps).
  - Manages both parallel and sequential execution.
  - Implements retry logic and logs each step's performance.
  - Calls data quality (DQ) checks as needed.
  - Supports multiple pre-step views with a list (`pre_views`) to register temporary tables, useful for lookups or joins in SQL.
- **`run_pipeline.py`**
  Entry-point / CLI script:
  - Boots up Spark (if using PySpark).
  - Loads the `pipelines_config.yaml`.
  - Instantiates and runs the pipeline.
  - Saves final execution logs to JSON.
  - Optionally uploads logs or writes metadata to Delta via utility modules.
- **`flows/main_flow.yaml`**
  YAML-based pipeline definition:
  - Describes the complete DAG, including stages, groups, step dependencies, parallelism, and retries.
  - Acts as the single source of truth for pipeline orchestration.
  - Allows flexible reconfiguration without touching the code.
  - Supports reusable SQL templates, DQ checks, and conditional step logic.
  - Can be extended to include notifications, checkpoints, secrets, and more.
- **`pipelines/flows/visualize_pipeline_dag.py`**
  This Python script helps you visualize the pipeline's Directed Acyclic Graph (DAG) based on your YAML configuration. It leverages:
  - **PyYAML** for parsing YAML.
  - **Graphviz** to build and render the DAG.
  - The script processes the pipeline definition and generates both PDF and PNG visualizations, making it easier to understand step dependencies and flow structure.

  ```bash ## Usage & Requirements

  ## 1. Install Required Libraries pip install pyyaml graphviz

  ## 2. On Linux (wsl), ensure Graphviz is installed: sudo apt update && sudo apt install graphviz xdg-utils

  ## 3. Run the script directly: pip install pyyaml graphviz ```
- **`logs/pipeline_log.json`**
  *Central logging output (JSON):*
  - Captures each step's success/failure, timing, retry counts, etc.
  - Useful for post-run analysis, troubleshooting, or performance metrics.
- **`utils/` Directory**
  Contains helper modules to support SQL operations, data quality checks, metadata logging, and datasource management.
- **`sql_loader.py`**
  *Jinja2-based SQL rendering:*
  - Loads `.sql` files from disk and renders them with dynamic parameters.
  - Injects custom contexts such as step name, batch ID, and temporary views.
  - Promotes clean separation of SQL logic from Python orchestration code.
  - Supports templating for table/view names in SQL scripts.
- **`sql_runner.py`**
  *Spark SQL execution module:*
  - Executes rendered SQL strings in Spark.
  - Optionally writes results to a destination (Delta, Parquet, etc.).
  - Returns row count or success status for logging and validation.
  - Supports output as a Spark temporary view for downstream steps.
  - Pre-validates SQL syntax using Spark's parser to catch errors early.
  - Optionally outputs logical/physical plans for debugging.
  - Supports dynamic write options specified in YAML.
- **`dq_checks.py`**
  *Data Quality (DQ) checks module:*
  - Runs SQL queries from the DQ folder (e.g., `row_count.sql`) to validate data quality.
  - Raises exceptions if quality thresholds aren't met.
- **`metadata_logger.py`**
  *Pipeline metadata logging:*
  - Stores pipeline run logs (the same JSON logs) into Delta tables if needed.
- **`datasource.py`**
  *DataSource management:*
  - Centralizes connection configurations (connection string, username, password, etc.).
  - Facilitates integration with secrets managers or environment variable injection.
  ```

- Generates Spark-compatible configurations for JDBC or other data sources.

- **`__init__.py`**
  - Marks the `utils/` folder as an importable Python package.

2.2. `sql/` Directory

Houses all SQL scripts, organized by layer:

- **Bronze Layer (`sql/bronze/`):**
  Contains raw ingestion SQL scripts.
- **`students.sql`** and **`schools.sql`**:
  - Pull data directly from source systems (like JDBC databases).
  - Apply lightweight transformations (e.g., renaming columns, filtering records).
  - Assumes data is registered as temporary views for further processing.
- **Silver Layer (`sql/silver/`):**
  Contains SQL scripts for Data Vault modeling.
- **Data Vault Scripts (e.g., `hub_student.sql`, `link_enrollment.sql`, `sat_student_demographics.sql`):**
  - Deduplicate keys, establish relationships, and build satellites.
  - Use techniques like `MERGE INTO` or `INSERT INTO SELECT DISTINCT` to maintain data integrity.
- **Gold Layer (`sql/gold/`):**
  Contains SQL scripts for reporting.
- **Star Schema Scripts (e.g., `dim_student.sql`, `dim_school.sql`, `fact_enrollment.sql`):**
  - Transform and flatten Data Vault models into reporting-friendly structures.
  - Include surrogate key generation, aggregations, and business logic.

2.3. `dq/` Directory

Organized by layer, this directory contains SQL files for data quality checks:

- **Bronze DQ:**
  - Example: `students/row_count.sql` and `students/not_null_studentid.sql` check basic conditions like row counts and non-null constraints.
- **Silver DQ:**
  - Example: `hub_student/unique_keys.sql` ensures the uniqueness of business keys.
- **Gold DQ:**
  - Example: `dim_student/not_null_dim_id.sql` validates critical fields needed for downstream joins.
  - Additional checks like `fact_enrollment/valid_dates.sql` ensure logical consistency (e.g., start date should be before end date).

2.4. `ingestion/` Directory

Contains scripts for data extraction and transformation across layers:

- **Bronze:**
- **`extract_students_jdbc.py`**:
  - Extracts raw data (e.g., Ed-Fi `students` table) via JDBC.
  - Writes data to Delta format in the raw data layer.
- **Silver:**
- **`transform_hub_student.py`**:
  - Reads raw data from Bronze.
  - Applies Data Vault hub logic to extract and deduplicate business keys.
  - Prepares data for merging into the hub table.
- **Gold:**
- **`build_dim_student_profile.py`**:
  - Joins silver layer outputs to create a dimension for reporting.
  - Flattens data and computes derived fields (e.g., full name, status).
  - Suitable for BI tools like Power BI or Looker.

2.5. `tests/` Directory

A suite of unit tests to ensure every component works as expected.

- **`test_dq.py`**
  - Tests the DQ checks by simulating Spark sessions.
  - Verifies behavior for both passing and failing quality checks.
- **`test_retry.py`**
  - Mocks step functions to simulate random failures.
  - Ensures retry logic and logging are functioning correctly.
- **`test_sql_loader.py`**
  - Validates Jinja2 SQL rendering.
  - Checks proper substitution of variables and error handling for missing or malformed templates.
- **`test_checkpoints.py`**
  - Tests checkpointing functionality.
  - Verifies marker file creation and behavior when steps are skipped due to checkpoint presence.
- **`test_batch_id.py`**
  - Validates batch ID generation.
  - Checks consistency across pipeline runs and proper fallback to timestamp-based IDs.
- **`test_pipeline_structure.py`**
  - Ensures the YAML pipeline definition adheres to the required schema.
  - Catches common configuration errors before deployment.
- **`__init__.py`**
  - Marks the `tests/` folder as an importable test package.
  - Enables discovery of tests using pytest or similar frameworks.

---

3. Files Structure

This section offers a bird's-eye view of the entire project file system, organized like a treasure map leading you to all the essential components of your ETL pipeline. You'll see clearly defined directories for your pipeline orchestration scripts, SQL logic (spread across Bronze, Silver, and Gold layers), data quality checks, ingestion scripts, and tests. Each folder is purpose-built to keep your code modular, maintainable, and ready to scale. It's like having a backstage pass to the inner workings of your data pipeline—now you know exactly where every crucial piece of your project lives.

```
.
├── pipelines/
│   ├── pipeline.py
│   │   ├─ (1) Primary orchestration script:
│   │   │       - Defines how to traverse the YAML config (stages → groups → steps).
│   │   │       - Manages parallel vs. sequential execution.
│   │   │       - Implements retry logic & logging for each step.
│   │   │       - Calls data quality checks when needed.
│   │   │       - Supports multiple pre-step views using a list (`pre_views`) to register multiple temp tables before executing SQL.
│   │   │             - This is useful for lookups or joins, for example when building facts.
│   ├── run_pipeline.py
│   │   ├─ (2) Entry-point / CLI script:
│   │   │       - Boots up Spark (if you're using PySpark).
│   │   │       - Loads `pipelines_config.yaml`.
│   │   │       - Instantiates and runs the pipeline.
│   │   │       - Saves final execution logs to JSON.
│   │   │       - Optionally uploads logs or writes metadata to Delta via utility modules.
│   ├── flows/
│   │   └── main_flow.yaml
│   │       ├─ (3) YAML-based pipeline definition:
│   │       │       - Describes the complete DAG: stages, groups, step dependencies, parallelism, retries, etc.
│   │       │       - Acts as the single source of truth for pipeline orchestration.
│   │       │       - Allows flexible reconfiguration without touching code.
│   │       │       - Supports reusable SQL templates, DQ checks, and conditional step logic.
│   │       │       - Can be extended to include notifications, checkpoints, secrets, and more.
│   ├── logs/
│   │   └── pipeline_log.json
│   │       ├─ (4) Central logging output (JSON):
│   │       │       - Captures each step's success/failure, timing, number of retries, etc.
│   │       │       - Useful for post-run analysis, troubleshooting, or metrics.
│   └── utils/
│       ├── sql_loader.py
│       │   ├─ (5) Jinja2-based SQL rendering:
│       │   │       - Loads `.sql` files from disk using Jinja2 templating.
│       │   │       - Supports custom context injection: step name, batch ID, temp views, etc.
│       │   │       - Promotes clean separation of SQL logic from Python orchestration code.
│       │   │       - Used by each step to dynamically prepare the Spark SQL query to run.
│       │   │       - Supports table/view templating like: `SELECT * FROM {{ pre_view_name }}`
│       ├── sql_runner.py
│       │   ├─ (6) Spark SQL execution:
│       │   │       - Accepts a fully rendered SQL string and runs it in Spark.
│       │   │       - Optionally writes results to a destination path (Delta, Parquet, etc.).
│       │   │       - Returns row count or success status for logging and validation.
│       │   │       - Supports output as a Spark temporary view for chained downstream steps.
│       │   │       - Pre-validates SQL syntax using Spark's parser before execution to catch issues early.
│       │   │             - Use spark.sessionState.sqlParser.parsePlan(sql_text) to validate SQL before runtime.
│       │   │             - Helpful for debugging complex SQLs or validating template rendering.
│       │   │       - Optionally calls `.explain()` on the DataFrame to output the logical/physical plan.
│       │   │             - Useful for understanding performance or Spark optimizations.
│       │   │             - Could be controlled with a flag: debug: true in the YAML step.
│       │   │       - Supports dynamic write options (format, path, mode) from YAML.
│       │   │             - Could be controlled in the YAML with parameters like format (e.g., delta, parquet, json)
│       │   │                   - format (e.g., delta, parquet, json)
│       │   │                   - path (e.g., /mnt/output# Where to write the result)
│       │   │                   - mode (e.g., Spark write mode (overwrite, append, etc.))
│       │   │             - Default is (e.g., 'delta' + 'overwrite'), but customizable per step.
│       ├── dq_checks.py
│       │   ├─ (7) Data Quality checks:
│       │   │       - Runs queries from the DQ folder (e.g. `row_count.sql`) to validate data.
│       │   │       - Can raise exceptions if checks fail (like a row_count < threshold).
│       ├── metadata_logger.py
│       │   ├─ (8) Pipeline metadata logging :
│       │   │       - If you want to store pipeline run logs (those same JSON logs) into Delta tables.
│       ├── datasource.py
│       │   ├─ (9) DataSource class to manage all your JDBC (or other source) connection:
│       │   │       - Centralizes connection config: connection string, username, password, etc.
│       │   │       - Makes it easier to inject credentials from a secrets manager or .env
│       │   │       - Can generate Spark-compatible .option() dictionaries or full configs
│       └── __init__.py
│           ├─ (10) Makes `utils/` an importable Python package
│               so modules can be imported like `from pipeline.utils import sql_loader`.

├── sql/
│   ├── bronze/
│   │   ├── students.sql
│   │   │   ├─ (11) Raw ingestion SQLs:
│   │   │   │   - These scripts pull data directly from source systems (like JDBC databases).
│   │   │   │   - Used in Bronze steps to land raw-but-structured data into your Lakehouse.
│   │   │   │   - They usually do lightweight transformations like renaming columns or filtering bad records.
│   │   │   │   - SQLs here assume your pipeline has already registered the raw input as a temp view (e.g., students).
│   │   │   │   - Example: `SELECT * FROM students`
│   │   └── schools.sql
│   │       ├─ (12) Another raw load example for a different entity (schools).
│   │       │       - Might use a different view or join source tables.
│   │       │       - Helps standardize data before applying modeling logic in Silver.
│
│   ├── silver/
│   │   ├── hub_student.sql
│   │   ├── hub_school.sql
│   │   ├── link_enrollment.sql
│   │   └── sat_student_demographics.sql
│   │       ├─ (13) Data Vault SQL scripts:
│   │       │   - These model your data into **Hubs, Links, and Satellites** using the Data Vault pattern.
│   │       │   - Typically work off the outputs of Bronze and turn them into deduplicated keys, relationships, and attributes.
│   │       │   - Often use SQL constructs like `MERGE INTO` or `INSERT INTO SELECT DISTINCT`.
│   │       │   - Assumes upstream views (like `students`) are pre-registered by the pipeline.
│   │       │   - Example: `INSERT INTO hub_student SELECT DISTINCT student_id, load_date FROM students`
│
│   └── gold/
│       ├── dim_student.sql
│       ├── dim_school.sql
│       └── fact_enrollment.sql
│           ├─ (14) Star Schema SQLs for reporting:
│           │   - These scripts build **dimension** and **fact** tables for analytics and dashboards.
│           │   - They join and reshape the normalized Data Vault structures into flattened business entities.
│           │   - They're typically SELECT-heavy and may include surrogate key logic, aggregations, and business rules.
│           │   - Example: `SELECT ROW_NUMBER() OVER (...) AS dim_student_id, ... FROM hub_student LEFT JOIN sat_student_demographics`

├── dq/
│   ├── bronze/
│   │   ├── students/
│   │   │   ├── row_count.sql
│   │   │   └── not_null_studentid.sql
│   │   │   ├─ (15) Data Quality checks for Bronze `students`:
│   │   │   │       - Small SQL queries that validate data after it's written to its destination.
│   │   │   │       - These checks run **after** the step finishes writing (via `dq_checks` in YAML).
│   │   │   │       - Example: `row_count.sql` might contain `SELECT COUNT(*) FROM students WHERE some_col IS NOT NULL`
│   │   │   │       - Example: `not_null_studentid.sql` checks for nulls in key fields (`student_id IS NOT NULL`)
│   │   │   │       - You can configure multiple checks per step in your YAML config.
│   │   └── schools/
│   │       └── row_count.sql
│   │       ├─ (16) Simple row count validation for `schools` ingestion.
│   │       │       - These are often used to guard against empty data loads or corrupted files.
│
│   ├── silver/
│   │   ├── hub_student/
│   │   │   └── unique_keys.sql
│   │   │   ├─ (17) Silver-level DQ check:
│   │   │   │       - Verifies uniqueness of primary keys or business keys in Data Vault Hubs.
```

```
│   │   │   │       - Example: `SELECT student_id, COUNT(*) FROM hub_student GROUP BY student_id HAVING COUNT(*) > 1`
│   │   ├── hub_school/
│   │   │   └── (possible checks)
│   │   ├── link_enrollment/
│   │   │   └── (possible checks)
│   │   └── sat_student_demographics/
│   │       └── (possible checks)
│   │       ├─ (18) Other optional Silver DQ:
│   │       │       - Common examples: checking referential integrity, ensuring all fields have valid types, etc.
│
│   └── gold/
│       ├── dim_student/
│       │   └── not_null_dim_id.sql
│       │   ├─ (19) Gold-level DQ:
│       │   │       - Ensures all records in `dim_student` have a valid `dim_id`, which is crucial for joins in facts.
│       │   │       - Example: `SELECT COUNT(*) FROM dim_student WHERE dim_student_id IS NULL`
│       ├── dim_school/
│       │   └── (possible checks)
│       └── fact_enrollment/
│           └── valid_dates.sql
│           ├─ (20) Fact-level DQ:
│           │       - Checks for data consistency and logic (e.g. `start_date < end_date`)
│           │       - Example: `SELECT COUNT(*) FROM fact_enrollment WHERE enrollment_date > graduation_date`

├── ingestion/
│   ├── bronze/
│   │   └── extract_students_jdbc.py
│   │       ├─ (21) Bronze layer extractor for Ed-Fi `students` table:
│   │       │       - Connects to the source SQL Server using JDBC.
│   │       │       - Extracts the `Students` table from the Ed-Fi ODS.
│   │       │       - Writes to Delta format at a raw layer (e.g., `/mnt/bronze/students`).
│   │       │       - Can be reused across multiple pipelines or stages.
│
│   ├── silver/
│   │   └── transform_hub_student.py
│   │       ├─ (22) Silver layer transformer for `hub_student`:
│   │       │       - Reads raw `students` data from Bronze.
│   │       │       - Applies Data Vault hub logic to extract business keys.
│   │       │       - Deduplicates keys and prepares a DataFrame for merging into the hub table.
│   │       │       - Good candidate for unit testing key integrity and structure.
│
│   └── gold/
│       └── build_dim_student_profile.py
│           ├─ (23) Gold layer builder for `dim_student_profile`:
│           │       - Joins multiple Silver layer outputs (e.g., `hub_student`, `sat_student_demographics`).
│           │       - Flattens the structure into a dimension ready for reporting (e.g., Power BI or Looker).
│           │       - Generates surrogate keys and computes derived fields like full name, status, etc.

└── tests/
    ├── test_dq.py
    │   ├─ (24) Unit tests for `dq_checks.py`:
    │   │       - Mocks a Spark session to simulate running DQ SQL checks.
    │   │       - Validates behavior when queries return valid vs. invalid results.
    │   │       - Tests that exceptions are raised for threshold violations.
    │   │       - Can be extended to test multiple SQL files with parametrized cases (e.g., pytest.mark.parametrize).
    │
    ├── test_retry.py
    │   ├─ (25) Unit tests for retry logic in `pipeline.py`:
    │   │       - Mocks a step function that randomly fails to simulate retry scenarios.
    │   │       - Ensures the pipeline attempts the correct number of retries before giving up.
    │   │       - Verifies that retry-related logs are created and accurate.
    │   │       - Can use time mocking or patching to avoid actual sleep delays.
    │
    ├── test_sql_loader.py
    │   ├─ (26) Unit tests for `sql_loader.py`:
    │   │       - Tests the Jinja2 rendering with various context dictionaries.
    │   │       - Validates correct substitution of variables into SQL templates.
    │   │       - Handles edge cases like missing variables, malformed syntax, etc.
    │   │       - Uses sample `.sql` files or in-memory strings for isolated tests.
    │
    ├── test_checkpoints.py
    │   ├─ (27) Tests for checkpointing logic in `pipeline.py`:
    │   │       - Verifies `is_step_checkpointed()` behavior with/without marker files.
    │   │       - Mocks filesystem interactions (e.g., `os.path.exists`) to simulate file states.
    │   │       - Ensures `save_step_checkpoint()` writes the expected marker file correctly.
    │   │       - Confirms that checkpointed steps are skipped during pipeline run.
    │
    ├── test_batch_id.py
    │   ├─ (28) Tests for batch ID generation:
    │   │       - Validates `batch_id` is pulled from config if present.
    │   │       - Verifies fallback logic generates a timestamp-based ID when missing.
    │   │       - Ensures batch_id is consistent throughout a pipeline run.
    │
    ├── test_pipeline_structure.py
    │   ├─ (29) Validation of pipeline YAML structure:
    │   │       - Verifies required keys exist in each step (e.g., `name`, `script`, `source`).
    │   │       - Catches common YAML schema errors early (missing paths, bad formats).
    │   │       - Can be used as part of CI/CD to lint the config before deployment.
    │
    └── __init__.py
        ├─ (30) Marks the `tests/` folder as a Python test package:
        │       - Enables pytest and other test runners to discover all test modules.
        │       - May define global test fixtures or mocks shared across modules.

└── run_pipeline.sh
    ├── A bash script located in the root directory that serves as a quick and convenient way to execute your ETL pipeline. It defaults to running the 'main_flow' pipeline if no argument is provided. The
        - Stops execution on error to prevent cascading failures.
        - Determines the pipeline name and corresponding YAML configuration.
        - Announces the pipeline execution and the configuration file being used.
        - Invokes the Python CLI (`pipelines/flows/main_flow.py`) with the specified config.
```

---

4. Example: `main_flow.yaml`

Below is an example of the `main_flow.yaml` file used to define your pipeline. This YAML file is the single source of truth for your ETL orchestration—laying out pipeline name, retry logic, notifications, checkpoints, stages, groups, and steps exactly as specified:

```yaml
pipeline:
  name: edfi_etl_pipeline  # Unique name for this entire pipeline run
  description: "End-to-end ETL pipeline using Medallion Architecture with Data Vault and Star Schema"  # Quick summary of what this pipeline does

  default_retry:
    attempts: 3            # Retry failed steps up to 3 times
    delay_seconds: 10      # Wait 10 seconds between retries

  notifications:
    on_failure: Teams://#data-alerts    # Where to scream if something fails
    on_success: Teams://#data-success   # Where to celebrate success (optional, but cute)

  checkpoint:
    enabled: true
    path: "{{base_path}}/_checkpoints/{{pipeline.name}}"  # Where to store state/progress so restarts can resume intelligently

  stages:  # Top-level ETL layers (Bronze, Silver, Gold)
    - name: bronze
      description: "Raw ingestion from SQL Server to Delta Lake"
      parallel: true             # Steps in this stage can run in parallel
      depends_on: []             # This is the first stage—no upstream dependencies

      groups:
        - name: ingestion        # Logical group within the bronze stage
          parallel: true         # These steps can also run in parallel
```

```yaml
        depends_on: []         # No dependencies—it's the start of the flow

        steps:
          - name: students
            depends_on: []     # Doesn't depend on any other step
            source:
              datasource: sqlserver_edfi
              table: dbo.Students
              query: "sql/{{stage.name}}/{{name}}.sql"   # External SQL file (templated for reusability)
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"  # Where to write the data in the lake
            dq_checks:  # Data quality validations to run post-write
              - "dq/{{stage.name}}/{{name}}/row_count.sql"
              - "dq/{{stage.name}}/{{name}}/not_null_studentid.sql"

          - name: schools
            depends_on: []     # Also independent
            source:
              datasource: sqlserver_edfi
              table: dbo.Schools
              query: "sql/{{stage.name}}/{{name}}.sql"
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"
            dq_checks:
              - "dq/{{stage.name}}/{{name}}/row_count.sql"

  - name: silver
    description: "Refined Data Vault modeling"
    parallel: false            # We'll run these groups sequentially
    depends_on: [bronze]       # Wait until bronze finishes

    groups:
      - name: hubs
        parallel: true         # Hubs can be built in parallel
        depends_on: [bronze]   # Depends on raw data being ingested

        steps:
          - name: hub_student
            depends_on: [students]  # Needs student data ingested
            source:
              type: delta
              query: "sql/{{stage.name}}/{{name}}.sql"
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"
            dq_checks:
              - "dq/{{stage.name}}/{{name}}/unique_keys.sql"

          - name: hub_school
            depends_on: [schools]
            source:
              type: delta
              query: "sql/{{stage.name}}/{{name}}.sql"
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"

      - name: links
        parallel: true
        depends_on: [hubs]     # Wait until all hubs are built

        steps:
          - name: link_enrollment
            depends_on: [hub_student, hub_school]  # Join step needs both hubs
            source:
              type: delta
              query: "sql/{{stage.name}}/{{name}}.sql"
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"

      - name: satellites
        parallel: true
        depends_on: [links]    # Satellites extend the links, so wait for them

        steps:
          - name: sat_student_demographics
            depends_on: [link_enrollment]
            source:
              type: delta
              query: "sql/{{stage.name}}/{{name}}.sql"
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"

  - name: gold
    description: "Star Schema for reporting"
    parallel: false
    depends_on: [silver]       # Wait until silver layer is fully built

    groups:
      - name: dimensions
        parallel: true
        depends_on: [satellites]  # Dimensions are built from refined satellite data

        steps:
          - name: dim_student
            depends_on: [sat_student_demographics]  # This fact depends on enriched student data
            source:
              type: delta
              query: "sql/{{stage.name}}/{{name}}.sql"
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"
            dq_checks:
              - "dq/{{stage.name}}/{{name}}/not_null_dim_id.sql"

          - name: dim_school
            depends_on: [hub_school]  # Directly built from school hub
            source:
              type: delta
              query: "sql/{{stage.name}}/{{name}}.sql"
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"

      - name: facts
        parallel: true
        depends_on: [dimensions]  # Facts are last—they depend on dimensions

        steps:
          - name: fact_enrollment
            depends_on: [dim_student, dim_school]  # Classic star schema join
            source:
              type: delta
              query: "sql/{{stage.name}}/{{name}}.sql"
            destination:
              format: delta
              path: "{{base_path}}/{{stage.name}}/{{name}}"
            dq_checks:
              - "dq/{{stage.name}}/{{name}}/valid_dates.sql"
datasources:
```

```yaml
  sqlserver_edfi:
    type: jdbc
    url: jdbc:sqlserver://your-sql-server:1433;databaseName=EdFi_Ods
    user: your_user
    password: your_password
    driver: com.microsoft.sqlserver.jdbc.SQLServerDriver
```

---

5. Roadmap

Roadmap for Mid-to-Long Term Improvements

This roadmap outlines strategic initiatives to enhance the data pipelines. It is designed to provide a robust, scalable solution that ensures reliable data ingestion, processing, governance, and operational excellence.

1. **Environment & Configuration Management**
   - Establish robust development, testing, and production environments.
   - Manage configurations and secrets securely (e.g., via environment variables or Vault).
2. **CI/CD Pipeline & Automation**
   - Define a deployment process with clear branching strategies and merge workflows.
   - Implement automated testing (unit, integration, data validation) and versioning for both code and YAML configurations.
3. **Data Governance & Lineage**
   - Implement a data catalog to track metadata, lineage, and business definitions.
   - Ensure traceability from raw ingestion through transformation stages (Bronze → Silver → Gold).
4. **Integration Contracts & Data Vault Adaptation**
   - Define clear integration contracts for Ed-Fi ODS and other educational data sources.
   - Adapt Data Vault models for new SQL database platforms, including schema modifications and performance tuning.
5. **Security & Compliance by Design**
   - Establish data classification standards, audit trails, and encryption protocols for sensitive data.
   - Ensure compliance with educational data standards and regulations.
6. **Monitoring & Alerting**
   - Set up real-time alerts (e.g., via Slack or email) for failures, slow queries, or anomalous behavior.
   - Integrate operational metrics and dashboards (using tools such as Grafana or Datadog) for proactive monitoring.
7. **Performance Tuning & Resource Usage**
   - Optimize Spark configurations (e.g., executor memory, partitions, shuffle settings) for efficiency.
   - Apply effective caching strategies and partitioning techniques to manage large data sets.
8. **Big Data Performance Testing**
   - Generate synthetic datasets to simulate real-world data volumes for load and stress testing.
   - Benchmark job execution times, memory usage, and resource utilization; integrate these tests into the CI/CD pipeline to detect regressions.
9. **Advanced Scheduling & Orchestration**
   - Document triggers for scheduled or event-based pipeline runs (e.g., new file arrivals in blob storage).
   - Update dependency graphs and refine workflow management practices as the pipeline evolves.
10. **Backup & Disaster Recovery**
    - Establish robust checkpointing and recovery procedures to handle partial failures or data corruption.
    - Develop an immutable storage strategy (e.g., using Delta tables) to ensure reliable data recovery.
11. **Cloud Migration Testing**
    - Formulate strategies to test the pipeline in cloud environments, ensuring performance parity and data integrity during migration.
    - Validate resource allocation, network configurations, and data transfer consistency, with fallback procedures for migration issues.
12. **Operational Playbook**
    - Create detailed on-call procedures and incident management guidelines for rapid response.
    - Outline performance testing protocols and escalation processes for system outages or failures.
13. **FAQ / Troubleshooting Section**
    - Compile common pitfalls (e.g., Spark memory issues, missing checkpoint files) along with quick fixes.
    - Provide clear support channels and documentation for fast issue resolution.
14. **Training & Onboarding**
    - Develop quick-start guides, video walkthroughs, and a repository of reusable code snippets and templates.
    - Facilitate ongoing knowledge sharing to ensure new team members can quickly get up to speed.
15. **Glossary & Documentation Enhancements**
    - Build a comprehensive glossary of key terms (e.g., Data Vault concepts, pipeline components) for easy reference.
    - Maintain living documentation to ensure clarity and consistency across the project.
16. **Enhancements Wishlist**
    - Keep an evolving list of future features such as streaming ingestion, machine-learning-driven data quality checks, and additional data source integrations.

---

6. Conclusion

This project structure is designed to provide clarity, flexibility, and robustness to your ETL pipeline. By separating concerns across different directories and layers, it enables:

- **Modular development:** Each component is isolated for easier maintenance and testing.
- **Scalability:** The architecture supports adding new data sources, transformations, and quality checks with minimal disruption.
- **Transparency:** Detailed logging and metadata capture make it easier to troubleshoot issues and analyze performance.

Feel free to tweak or extend any component to suit your specific use case, and remember: a clean structure today saves you from headaches tomorrow!

Enjoy building your pipeline and may your data always be as clean as your code!