□ Configurations (`dags/configs/`)

`__init__.py`

A module marker—allows Python to treat `dags/configs` as a package.

`dags_config.py`

Defines global **file templates** and the target **database backend**:

```
FILE_TYPES = {
    "yellow": "yellow_tripdata_{year}-{month:02d}.parquet",
    "green":  "green_tripdata_{year}-{month:02d}.parquet",
    "fhv":    "fhv_tripdata_{year}-{month:02d}.parquet",
}
DATABASE_TO_RUN = 'SNOWFLAKE'  # or 'POSTGRES'
```

- **FILE_TYPES**: Maps service names to TLC-parquet naming patterns.
- **DATABASE_TO_RUN**: Toggles ETL between Postgres and Snowflake .

`db_config.py`

Loads Postgres connection parameters from environment using `python-dotenv`:

```
class PostgresConfig:
    def __init__(self):
        self.host     = os.getenv("POSTGRES_HOST", "testdb_postgres")
        self.port     = int(os.getenv("POSTGRES_PORT", 5432))
        self.database = os.getenv("POSTGRES_DB", "test_data")
        self.user     = os.getenv("POSTGRES_USER", "user")
        self.password = os.getenv("POSTGRES_PASSWORD", "password123")
```

```
def as_dict(self):
    return {
        &amp;quot;host&amp;quot;:     self.host,
        &amp;quot;port&amp;quot;:     self.port,
        &amp;quot;database&amp;quot;: self.database,
        &amp;quot;user&amp;quot;:     self.user,
        &amp;quot;password&amp;quot;: self.password,
    }

```

Use `PostgresConfig().as_dict()` when instantiating DB connections .

`s3_config.py`

Wraps S3/LocalStack settings from `.env`:

```
class S3Config:
    def __init__(self):
        self.endpoint_url          = os.getenv("S3_ENDPOINT_URL", "")
        self.aws_access_key_id     = os.getenv("AWS_ACCESS_KEY_ID", "test")
        self.aws_secret_access_key = os.getenv("AWS_SECRET_ACCESS_KEY", "test")
        self.region_name           = os.getenv("AWS_REGION", "us-east-1")
        self.bucket_name           = os.getenv("S3_BUCKET_NAME", "cityride-raw")
```

```
def as_dict(self):
    return {
        &amp;quot;endpoint_url&amp;quot;:          self.endpoint_url,
        &amp;quot;aws_access_key_id&amp;quot;:     self.aws_access_key_id,
        &amp;quot;aws_secret_access_key&amp;quot;: self.aws_secret_access_key,
        &amp;quot;region_name&amp;quot;:           self.region_name,
        &amp;quot;bucket_name&amp;quot;:           self.bucket_name,
    }

```

Facilitates boto3 client configuration .

□ Connections (`dags/connections/`)

`__init__.py`

Marks the connections package.

`postgres_conn.py`

Provides simple wrappers over **psycopg2**:

- `run_query(sql)`: Executes a SQL statement, returns fetched rows for `SELECT`, else commits.
- `insert_records(sql, records)`: Bulk inserts lists of tuples.

It sources credentials via `PostgresConfig` .

`s3_conn.py`

`S3Connection` class using **boto3** (with optional LocalStack endpoint):

- **Initialization** logs credential masks and creates an S3 client.
- `upload_fileobj(file_obj, s3_key)`: Simple upload.
- `list_objects(prefix)`: Lists keys under a prefix.
- `upload_file_with_progress_bar(...)`: Upload with tqdm progress bar for large files.

Example usage embedded in docstring .

□ Transformations (`dags/transformations/`)

`__init__.py`

Package marker.

`common_transforms.py`

Pandas-based cleanup & mapping:

- `parse_timestamps(df, timestamp_cols)`: Coerces columns to UTC datetimes.
- `correct_numeric_types(df, numeric_cols)`: Converts specified columns to numeric, coercing errors.
- `filter_invalid_trips(df)`: Drops rows with negative fares, zero distances/passengers.
- `map_columns_to_table(df, file_type)`: Renames & reorders raw columns to match target schema, injecting missing columns as `None`.

Great for chunk-wise ETL in the **processing DAG** .

□ Utilities (`dags/utils/`)

`helpers.py`

Central `run_query(query, params=None)` that:

- Reads `DATABASE_TO_RUN` from Airflow **Variable**.
- Retrieves Airflow connection via `BaseHook`:
  - **Postgres**: uses psycopg2.
  - **Snowflake**: uses `snowflake.connector`.
- Executes SQL, returns results, and commits.

Abstracts differences between backends .

□ DAG Definitions

1. Ingestion DAG (`nyc_taxi_ingestion`)

File: `dags/ingestion_dag.py`
Schedule: `0 6 2 * *` (2nd of each month)
Tasks:

1. **init_metadata**: Inserts `MISSING` entries into `cityride_metadata.file_metadata`.
2. **process_files**:
   - Queries `file_metadata` statuses.
   - HEAD requests to TLC API → marks `AVAILABLE`.
   - Downloads available files into S3 raw prefix.
   - Updates `UPLOADED` status.
3. **trigger_processing_dag**: Fires `nyc_taxi_processing` with `{year, month}` conf.

```
flowchart TD
    A[init_metadata] --> B[process_files]
    B --> C[trigger_processing_dag]
```

See ETL logic sourcing `FILE_TYPES`, S3Hook, and `run_query` .

---

2. Processing DAG (`nyc_taxi_processing`)

File: `dags/processing_dag.py`
Trigger: Manual (via ingestion)
Tasks:

- **process_month_files**:
  - For **SNOWFLAKE**: Issues `COPY INTO` + INSERT statements.
  - For **POSTGRES**: Streams Parquet from S3 via PyArrow, applies `common_transforms`, bulk-inserts into `trip_data`, and logs into `processed_dag_metadata`.
- **trigger_analytics_dag**: Fires `nyc_trip_analytics_etl` with the same conf.

```
flowchart TD
    P[nyc_taxi_processing] --> Q[trigger_analytics_dag]
```

Key helpers: `insert_records`, `get_snowflake_copy_command`, `get_trip_data_insert_query` .

---

3. Analytics DAG (`nyc_trip_analytics_etl`)

File: `dags/analytics_dag.py`
Manual trigger; expects `{year, month}`.
Single PythonOperator:

- `load_incremental_fact_tables`:
  - Iterates fact tables: `fact_trips`, `fact_trips_daily_agg`, `fact_trips_hourly_agg`.
  - Logs “started” & “loaded” statuses in `etl_analytics_log`.
  - Executes INSERT … SELECT statements into star-schema fact tables.

```
flowchart LR
    X[load_incremental_fact_tables]
```

Orchestrates incremental loads; uses `run_query`.

□ Database Schema (Postgres & Snowflake)

```
erDiagram
    dim_date {
        DATE date_id PK
        INT year
        INT month
        INT day
        INT day_of_week
        VARCHAR day_name
        VARCHAR month_name
        INT quarter
        BOOL is_weekend
    }
    dim_location {
        INT location_id PK
        VARCHAR location_type
        VARCHAR borough
        VARCHAR zone
    }
    trip_data {
        BIGINT trip_id PK
        VARCHAR file_type
        INT year
        INT month
        TIMESTAMP pickup_datetime
        TIMESTAMP dropoff_datetime
        FLOAT passenger_count
        FLOAT trip_distance
        NUMERIC payment_type
        FLOAT fare_amount
        …other fare cols…
    }
    fact_trip_summary {
        INT year PK
        INT month PK
        VARCHAR file_type PK
```

```
    NUMERIC total_trips
    NUMERIC total_passengers
    NUMERIC avg_trip_distance
    NUMERIC avg_trip_time
    NUMERIC total_fare
    NUMERIC total_amount
}
fact_fare_summary {
    INT year PK
    INT month PK
    VARCHAR file_type PK
    NUMERIC total_fare
    NUMERIC total_tips
    NUMERIC total_tolls
    NUMERIC total_surcharges
    NUMERIC total_amount
}
fact_trip_locations {
    INT year PK
    INT month PK
    VARCHAR file_type PK
    INT pickup_location_id PK
    INT dropoff_location_id PK
    NUMERIC total_trips
    NUMERIC avg_trip_distance
    NUMERIC avg_trip_time
    NUMERIC total_amount
}
file_metadata {
    INT id PK
    INT year
    INT month
    VARCHAR file_type
    VARCHAR status
    TIMESTAMP last_checked
    TIMESTAMP uploaded_at
    INT retry_count
}
processed_dag_metadata {
    BIGINT id PK
    VARCHAR file_type
    INT year
    INT month
    VARCHAR status
    BIGINT rows_in_file
    BIGINT rows_loaded
}
trip_data ||--|| dim_date        : pickup_date_key
trip_data ||--|| dim_date        : dropoff_date_key
trip_data }|--|| dim_location    : pickup_location_id
trip_data }|--|| dim_location    : dropoff_location_id
```

- **Dimensions**: `dim_date`, `dim_location`.
- **Facts**: `trip_data`, `fact_trip_summary`, `fact_fare_summary`, `fact_trip_locations`.
- **Metadata**: `file_metadata` & `processed_dag_metadata`.

□ Deployment & Local Development

`airflow.Dockerfile`

基于 `apache/airflow:2.7.1`，切换至 `airflow` 用户并安装：

- `tqdm` (进度条)
- `snowflake-connector-python`
- Airflow provider 包 (Postgres, Amazon, Snowflake).

```
FROM apache/airflow:2.7.1
USER airflow
RUN pip install tqdm snowflake-connector-python \
    && pip install --no-cache-dir \
       apache-airflow-providers-postgres \
       apache-airflow-providers-amazon \
       apache-airflow-providers-snowflake
```

`docker-compose.yaml`

Orchestrates a **LocalExecutor** Airflow stack + dependencies:

- **testdb_postgres** (Postgres 14)
- **localstack** (S3 mock)
- **airflow-webserver** & **airflow-scheduler** (built with above Dockerfile)
- Volume mounts for `dags`, `logs`, `plugins`.
- Environment variables wiring Airflow → Postgres connection. .

□ Visualization (`visualization/`)

All apps built with **Streamlit** + **Plotly**:

- `snowflake_connector.py`: `load_data_from_source(sql)` → fetches data from Snowflake.
- `main.py`:
  - Configures page (`wide`, □ icon).
  - Sidebar navigation to:
    - `show_executive_dashboard()`

- `show_service_comparison()`
- `show_demand_heatmap()`
- `show_top_routes()`
- `show_monthly_trends()`
- `show_airport_traffic()`

Each module (e.g., **`demand_heatmap.py`**, **`airport_traffic.py`**, **`top_routes.py`**, **`monthly_trends.py`**, **`service_comparison.py`**, **`executive_dashboard.py`**) follows a pattern:

1. **Custom CSS** injection for styling.
2. **Data loading** via `load_data_from_source()`.
3. **Metrics & Charts** with Plotly Express / GraphObjects.
4. **Layout** using Streamlit columns, tabs, metrics, and expanders for raw data.

E.g., `show_demand_heatmap()` displays a 7×24 heatmap of hourly demand intensity and quick insights (peak hour/day, average intensity) .

□ System Context (C4)

```
C4Context
    title CityRide Analytics Platform
```

```
Person(user, &quot;Data Engineer&quot;, &quot;Designs &amp; triggers ETL workflows&quot;)
System(airflow, &quot;Airflow&quot;, &quot;Orchestrates ingestion, processing, analytics DAGs&quot;)
SystemDb(postgres, &quot;Postgres&quot;, &quot;Holds raw metadata &amp; processed trip_data&quot;)
SystemDb(snowflake, &quot;Snowflake&quot;, &quot;Data warehouse for star-schema analytics&quot;)
System(s3, &quot;S3 / LocalStack&quot;, &quot;Stores raw TLC Parquet files&quot;)
System(streamlit, &quot;Streamlit App&quot;, &quot;Interactive dashboards&quot;)

Rel(user,        airflow,    &quot;uses&quot;)
Rel(airflow,     s3,         &quot;reads/writes raw trip files&quot;)
Rel(airflow,     postgres,   &quot;writes metadata &amp; fact tables&quot;)
Rel(airflow,     snowflake,  &quot;loads analytics tables&quot;)
Rel(streamlit,   snowflake,  &quot;reads analytics data&quot;)

```

This documentation captures each component's role, interfaces, and inter-dependencies, providing a comprehensive guide to the **CityRide Analytics** codebase.

Focused appendix — DAGs, Tables, and Metadata

- What each DAG does

- Ingestion DAG (nyc_taxi_ingestion)
  - Initializes file metadata for the target period with status MISSING in cityride_metadata.file_metadata.
  - Probes TLC URLs; when a file is reachable, updates status to AVAILABLE and last_checked.
  - Streams the file to S3 raw/ and updates status to UPLOADED and uploaded_at.
  - Triggers the Processing DAG with year and month.
- Processing DAG (nyc_taxi_processing)
  - Snowflake path:
    - COPY INTO CITYRIDE_METADATA.{yellow_raw|green_raw|fhv_raw} from the external stage.
    - Inserts normalized rows into CITYRIDE_ANALYTICS.trip_data.
    - Upserts CITYRIDE_METADATA.etl_processing_log with rows_in_file, rows_loaded, status (loaded or error), and timestamps.
  - Postgres path:
    - Reads Parquet from S3 in row-group chunks, applies transforms, bulk-inserts into cityride_analytics.trip_data.
    - Upserts cityride_analytics.processed_dag_metadata with rows_in_file, rows_loaded, status=loaded.
  - Triggers the Analytics DAG.
- Analytics DAG (nyc_trip_analytics_etl)
  - For each fact table (fact_trips, fact_trips_daily_agg, fact_trips_hourly_agg):
    - Writes a started record to CITYRIDE_METADATA.etl_analytics_log.
    - Runs INSERT … SELECT to load new data for the {year, month}.
    - Updates CITYRIDE_METADATA.etl_analytics_log to status loaded (or failed on error).

- Key tables created/used by the DAGs

  - Raw/landing (Snowflake)
    - CITYRIDE_METADATA.yellow_raw, green_raw, fhv_raw: direct COPY targets holding raw TLC columns for the month.
  - Core staging
    - CITYRIDE_ANALYTICS.trip_data (Snowflake) / cityride_analytics.trip_data (Postgres): canonical row-level trips with harmonized columns (timestamps, fares, locations, ids).
  - Aggregated facts (Postgres examples in repo; analogous Snowflake facts in scripts)
    - cityride_analytics.fact_trip_summary: total_trips, total_passengers, avg distance/time, fare/tips/amount per {year, month, file_type}.
    - cityride_analytics.fact_fare_summary: totals of fare, tips, tolls, surcharges, total_amount per {year, month, file_type}.
    - cityride_analytics.fact_trip_locations: route-level metrics per pickup/dropoff location pairs per month.
  - Dimensions (Postgres examples)

        - cityride_analytics.dim_date: calendar enrichment for time-based analysis.
        - cityride_analytics.dim_location: distinct pickup/dropoff locations with optional attributes.
    - Metadata and operational logging
        - cityride_metadata.file_metadata (Postgres) / CITYRIDE_METADATA.file_metadata (Snowflake):
            - Tracks per-month file presence and pipeline status. Status transitions: MISSING → AVAILABLE → UPLOADED.
            - Columns include last_checked, uploaded_at, retry_count, and timestamps.
        - cityride_analytics.processed_dag_metadata (Postgres):
            - Per file_type/year/month progress for the Processing DAG. Fields: status (e.g., loaded), rows_in_file, rows_loaded.
        - CITYRIDE_METADATA.etl_processing_log (Snowflake):
            - Equivalent of processed_dag_metadata with richer timestamps: processing_start_time, processing_end_time, status, rows_in_file, rows_loaded.
        - CITYRIDE_METADATA.etl_analytics_log (Snowflake):
            - Analytics DAG lifecycle per fact table with status, record_count, started_at, completed_at, and error_message.

- Metadata-driven state transitions

    - Ingestion lifecycle
        - MISSING: initialized by Ingestion DAG.
        - AVAILABLE: set when TLC HEAD check returns 200.
        - UPLOADED: set after successful download to S3.
    - Processing lifecycle
        - processed_dag_metadata (Postgres) / etl_processing_log (Snowflake) records:
            - status: loaded or error, with row counts and timing.
    - Analytics lifecycle
        - etl_analytics_log records:
            - started → loaded (or failed) for each fact table increment.

```
stateDiagram-v2
    title File and ETL Metadata Lifecycle
    [*] --> MISSING
    MISSING --> AVAILABLE: TLC HEAD 200
    AVAILABLE --> UPLOADED: S3 upload success
    UPLOADED --> PROCESSED: Processing DAG loads trip_data\n(processed_dag_metadata / etl_proce
    PROCESSED --> ANALYZED: Analytics DAG loads facts\n(etl_analytics_log = loaded)
```

- How DAGs interact with tables (high level)

```
sequenceDiagram
    participant ING as Ingestion DAG
    participant S3 as S3 Raw
```

```
participant DBM as file_metadata
participant PRC as Processing DAG
participant TRIP as trip_data
participant PLOG as processed_dag_metadata / etl_processing_log
participant ANL as Analytics DAG
participant FACT as fact_* tables
participant ALOG as etl_analytics_log
```

```
ING-&gt;&gt;DBM: INSERT (year, month, file_type, status=MISSING)
ING-&gt;&gt;DBM: UPDATE status=AVAILABLE (if TLC HEAD=200)
ING-&gt;&gt;S3: PUT raw/{file}.parquet
ING-&gt;&gt;DBM: UPDATE status=UPLOADED, uploaded_at
ING--&gt;&gt;PRC: Trigger with {year, month}

PRC-&gt;&gt;TRIP: INSERT normalized trips (COPY+INSERT or chunked insert)
PRC-&gt;&gt;PLOG: UPSERT status=loaded, rows_in_file, rows_loaded

PRC--&gt;&gt;ANL: Trigger with {year, month}

ANL-&gt;&gt;ALOG: INSERT started (fact_trips, daily_agg, hourly_agg)
ANL-&gt;&gt;FACT: INSERT … SELECT for {year, month}
ANL-&gt;&gt;ALOG: UPDATE status=loaded (or failed)

```