Database Design & ETL Automation — Project Documentation

Students: Oltean Simion, Ihos Iarina, Cuibus Dorin, Andreican Rares, Nora Girda

October 24, 2025

1 Project Overview

Goal. This project aims to design and implement a fully automated Extract–Transform–Load (ETL) pipeline using modern relational database techniques. A real-world CSV dataset is continuously ingested into a dedicated *staging* schema, where new or updated records are detected intelligently. From there, validated and structured information is incrementally propagated into a highly normalized *production* schema. All operations are idempotent, ensuring data consistency even when jobs run repeatedly.

Motivation. In practical data engineering workflows, raw input frequently changes over time. Automating the ingestion and transformation stages eliminates repetitive manual work, improves data freshness, and ensures traceability of updates and errors. This project demonstrates how an enterprise-style ETL pipeline can be implemented efficiently using SQL-centric technologies.

Scope. The pipeline includes:

- automated ingestion of a CSV dataset into staging as JSONB
- change detection based on file modification timestamps
- schema-enforced transformation into a production database model
- execution logging with success/failure/error transparency
- strict domain validation via constraints and referential integrity

Automation. Using the `pg_cron` extension, all ETL processes are scheduled to run periodically without user intervention. This creates a "hands-off" workflow where data correctness and freshness are continually maintained.

Stack. PostgreSQL 16 with `pg_cron` for scheduling, deployed automatically through Docker to ensure full reproducibility and a consistent execution environment across systems.

2 Dataset Summary

The dataset (`dataset.csv`) contains real-world inspired observations describing a student's path from education to early career development. Each record represents one student, uniquely identified by a stable natural key (`student_id`). The dataset provides a rich feature set spanning demographics, academic performance, skills development, and employment outcomes, making it ideal for demonstrating multi–table normalization and progressive transformation through a data pipeline.

The dataset includes 19 attributes categorized as follows:

- **Demographics:** static personal information relevant for grouping and filtering
- **Academic Performance:** indicators of pre-career academic achievement
- **Skills & Extracurricular Activities:** practical learning and professional preparation metrics

- **Career Outcomes:** success indicators after entering the workforce

`student_id`	Natural key for a student (string, unique).
`age`	Student's age (integer).
`gender`	Gender label (string).
`high_school_gpa`	HS GPA in [2.00, 4.00].
`sat_score`	SAT score in [900, 1600].
`university_gpa`	University GPA in [2.00, 4.00].
`field_of_study`	Bachelor's major/field of study.
`internships_completed`	Number of completed internships (0–4).
`projects_completed`	Number of completed academic/practical projects (0–9).
`certifications`	Count of professional certifications earned (0–5).
`soft_skills_score`	Rating of communication/teamwork skills (1–10).
`networking_score`	Professional networking ability score (1–10).
`job_offers`	Number of job offers received upon graduation (0–5).
`starting_salary`	Initial salary (25 000–1 000 000).
`career_satisfaction`	Job satisfaction rating (1–10).
`years_to_promotion`	Years until first promotion (1–5).
`current_job_level`	Career stage: Entry/Mid/Senior/Executive.
`work_life_balance`	Balance score (1–10).
`entrepreneurship`	Whether the student chose a start-up path (Yes/No).

Each column has a well-defined domain, which is later enforced in the *production* schema through `CHECK` constraints. The current structure makes the dataset suitable for:

- rigorous normalization across multiple relational tables,
- incremental updates via `student_id` during ETL operations,
- further analytical use cases such as correlating skills to hiring success.

Since the CSV may evolve over time, the *staging* layer stores rows as JSONB to retain flexibility during ingestion while ensuring the *production* schema stores only validated and well-typed values.

3 Architecture & Workflow

The implemented pipeline follows a two–stage ETL workflow that supports incremental data ingestion, schema evolution, and strict integrity validation. Automation ensures continuous data refresh with zero manual intervention. Figure **??** illustrates the processing flow.

High-Level Flow

1. **File-based Ingestion to Staging.** A scheduled task periodically checks the modification timestamp of `dataset.csv`. If the file is new or updated, rows are bulk-loaded into a temporary table and then **upserted** into `staging.events` in JSONB format. This provides schema flexibility and preserves raw input for traceability.
2. **Transformation and Load to Production.** A second scheduled job extracts well-typed attributes from JSONB and populates a fully normalized *production* schema via `INSERT ...ON CONFLICT DO UPDATE`. This allows repeated execution without duplication, ensuring **idempotent synchronization** between staging and production.

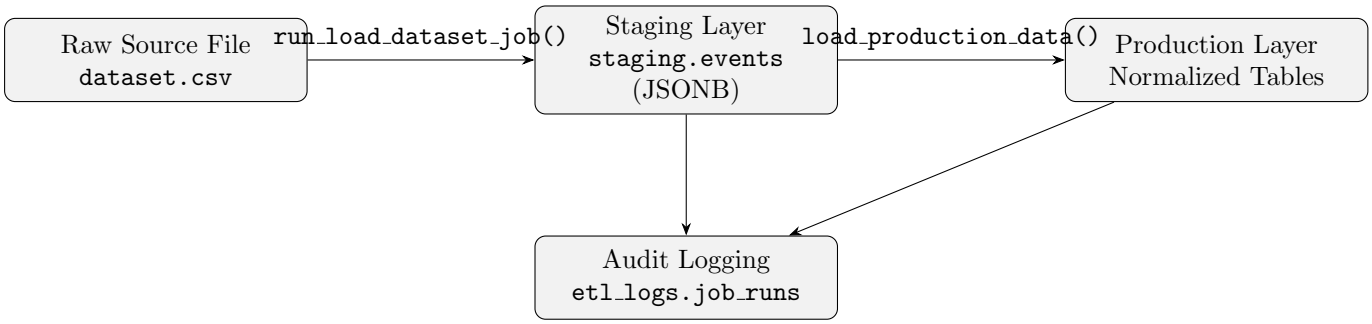
3. **Operational Monitoring & Auditing.** Each job execution writes a lifecycle entry to `etl_logs.job_runs`, including timestamps, processed record counts, and status states: *running*, *completed*, *skipped*, or *failed*. This supports real-world observability requirements.

Separation of Concerns

The staging layer absorbs raw variability, while the production schema enforces:

- data validation through `CHECK` constraints,
- referential integrity via foreign keys,
- subject-based segregation of attributes (3NF).

This layered approach reduces error propagation and improves recovery robustness.



figureAutomated ETL Architecture integrating staging, normalized production, and execution logging.

4 Schemas

The database is structured in two layers to separate raw ingestion from validated, relational data storage. This supports reliability, maintainability, and independent evolution of each part of the ETL pipeline.

Staging Schema

`staging.events`(`stagEventId SERIAL PK`, `content JSONB`, `load_date TIMESTAMP`)
The staging layer acts as a *landing zone* that accepts raw CSV input with minimal assumptions. All fields are ingested as JSONB to provide:

- **schema flexibility** when the source file evolves
- **traceability** of original data before transformation
- **efficient upsert checks** based on `student_id` comparison
- **auditable** timestamp of ingestion (`load_date`)

Only minimal uniqueness logic is applied here; validation and constraints are deferred to the production layer.

ETL Logging Schema

`etl_logs.job_runs(logId PK, jobname, status, error_message, start_date, end_date, records_count)`
This table stores execution metadata for `pg_cron` jobs. Helper functions such as:

`return_max_date()`, `update_job_run()`, `update_job_skipped()`, `update_job_failed()`

standardize ETL lifecycle state transitions to *running*, *completed*, *skipped*, or *failed*. This provides real-world **observability** needed in automated data pipelines.

Production Schema (Normalized)

Production tables enforce a clean, 3NF structure with strongly typed attributes. The data model is structured around the natural key `student_id`, forming one-to-one relationships:

- `production.student(student_id PK, age, gender)`
  Demographics: attributes that describe a person directly.
- `academic_performance(student_id PK, FK, GPA, SAT score, field_of_study)`
  Education achievements and knowledge foundations.
- `skills_extracurriculars(student_id PK, FK, internships, projects, certifications, soft skills, networking score)`
  Practical skill development and career readiness.
- `career_outcomes(student_id PK, FK, job offers, salary, satisfaction, promotion timeline, job level, work-life balance, entrepreneurship)`
  Early career success indicators after graduation.

To ensure relational integrity:

- **All foreign keys cascade on delete** — removing a student removes dependent facts.
- **CHECK constraints** enforce valid value ranges and categorical membership.
- **Idempotent** upserts ensure reprocessing does not create duplicates.

This schema structure enables efficient querying across dimensions while preserving data correctness and consistency across incremental loads.

5 Normalization (1NF, 2NF, 3NF)

The transformation stage restructures raw JSONB into a clean relational model that satisfies classical normalization rules. Since the dataset contains distinct thematic categories (demographics, academics, skills, career outcomes), the final schema avoids redundancy and ensures strong data integrity guarantees.

1NF (First Normal Form).

- All values in production tables are atomic and strongly typed (e.g., integers, numeric ranges, constrained enums).
- No repeating groups or multi-valued fields are present.
- JSONB appears only in the staging layer where schema flexibility is necessary; once validated, data moves into scalar columns.

2NF (Second Normal Form).

- Every table has a single-column primary key: `student_id`.
- All non-key attributes depend solely on the entire key, eliminating partial dependencies.

3NF (Third Normal Form).

- Non-key attributes do not depend on other non-key attributes.
- Tables reflect single-subject domains: demographics, academic performance, skills development, and career outcomes are separated.
- CHECK constraints enforce valid domains, preventing semantic anomalies.

This ensures a high-quality analytical data model: consistent updates, reduced duplication, and improved query performance.

6 ETL Implementation Details

The ETL pipeline operates in two independent but coordinated stages: **staging ingestion** and **production loading**. Both stages are **idempotent**, allowing safe re-execution without creating duplicate or inconsistent data.

Staging Loader (Change-Aware Upsert)

Function: `run_load_dataset_job()`

This function implements **incremental ingestion** by detecting whether the source CSV has changed:

1. Check file modification timestamp using:
   `pg_stat_file('/app/dataset.csv').modification`
2. Compare with `MAX(end_date)` from previous successful executions of job *dataset-load*.
3. If the file is updated or first ingestion occurs:
   - Bulk-load into a temporary table via `COPY`
   - Convert each row to JSONB
   - **UPDATE** existing rows only if content changed
   - **INSERT** new rows if `student_id` not present
4. If no change is detected, the job status is set to `skipped` to avoid redundant writes.

Benefits:

- Reduced I/O load (**only changed data is written**)
- Safe repeated execution without corrupting data
- Full traceability because raw data stays in staging

Execution status, error messages, and the number of affected records are automatically written to the audit table `etl_logs.job_runs`, which improves debugging and operational monitoring in a real production environment.

Production Loader (Idempotent Upsert)

Function: `load_production_data()`

This function transforms validated data from `staging.events` into a fully normalized production schema. JSONB fields are cast into strongly typed columns, while `INSERT ...ON CONFLICT DO UPDATE` ensures that loading is **idempotent**: data can be refreshed repeatedly without creating duplicates or losing referential integrity.

Processing Steps:

1. Extract typed values from JSONB using `content->>'field'`
2. **Insert or update** into:
   - `production.student` — demographics
   - `production.academic_performance` — education metrics
   - `production.skills_extracurriculars` — practical skill indicators
   - `production.career_outcomes` — workforce success measures
3. Domain rules are applied automatically through:
   - **CHECK constraints** (e.g., GPA/score ranges)
   - **foreign key enforcement** with cascading deletes

Benefits:

- Guarantees synchronized data between staging and production
- Maintains consistency even under repeated execution or partial failures
- Enforces relational correctness before acceptance into the analytical layer

This stage completes the ETL pipeline by transforming semi-structured raw input into reliable, query-optimized relational data.

7 Automation (Scheduling with `pg_cron`)

The entire pipeline is automated using the PostgreSQL extension `pg_cron`, enabling scheduled SQL tasks to run inside the database without external tools.

Configuration:

- Extension installed in Docker image: `postgresql-16-cron`
- PostgreSQL configuration updated:

```
shared_preload_libraries = 'pg_cron'
cron.database_name = 'test_db'
```

- Scheduler becomes active as soon as the container initializes

Recurring Jobs Deployed in This Project:

- `load-dataset` job — runs **every 1 minute**
  `SELECT run_load_dataset_job();`
  - detects changes in the source file
  - ingests and upserts raw data into staging

- `etl-job` — runs **every 2 minutes**
  `SELECT load_production_data();`
  - normalizes staged data into production tables
  - enforces integrity and validation rules

  Both tasks are created automatically during initialization using:

  `SELECT cron.schedule(...);`

  Operational Impact:

  - Data freshness is continuously preserved
  - Historical job runs are fully auditable via `etl_logs.job_runs`
  - No human intervention required after deployment

Thus, the system behaves like a real production ETL pipeline, combining automation, monitoring, and resilience in a fully containerized environment.

8 Deployment & Reproducibility

A major requirement of this project was to ensure a fully automated and reproducible deployment process. To achieve this, the entire ETL stack (PostgreSQL 16, pg_cron scheduler, staging + production schemas, and jobs) is packaged inside a Docker image. Upon container startup, the entire system self-configures, eliminating any need for manual SQL execution.

Docker Image and Initialization

Dockerfile Highlights:

- Based on the official `postgres:16` image, ensuring reliability and security.
- Installs the **pg_cron** extension, activating in-database automation.
- Configures:

```
shared_preload_libraries = 'pg_cron'
cron.database_name = 'test_db'
```

- Automatically copies initialization SQL scripts and the dataset:
  - schema creation (staging, production, logging)
  - ETL functions (`run_load_dataset_job()`, `load_production_data()`)
  - pg_cron job scheduling
- Zero manual configuration is required — the pipeline becomes operational at first boot.

This setup ensures consistency across environments and supports rapid redeployment in case of failures.

Build & Run Instructions

Running the ETL system locally or on any server requires just two terminal commands:

1. **Build the image:**

```
docker build -t dbd-pipeline .
```

2. **Start a container:**

```
docker run --name dbd-pg -p 5432:5432 \
    -e POSTGRES_PASSWORD=postgres dbd-pipeline
```

Once launched, PostgreSQL automatically:

- creates schemas and tables,
- loads and normalizes data,
- registers two scheduled ETL jobs with pg_cron,
- begins continuous ingestion and transformation.

Users can immediately connect with:

```
psql -h localhost -U postgres -d test_db
```

Reproducibility is guaranteed: the same results are produced regardless of the machine running the container, fulfilling modern DevOps and data engineering deployment standards.

9 Logging & Monitoring

`etl_logs.job_runs` captures: job name, status, timestamps, error message, and processed record counts. Helper functions consistently transition statuses from *running* to terminal states. Use:

```
SELECT * FROM etl_logs.job_runs ORDER BY logId DESC LIMIT 20;
```

10 Analysis: Issues & Improvements (Requirement #7)

The implemented solution successfully demonstrates a fully automated ETL pipeline, but also reveals several practical challenges and opportunities for enhancement.

Strengths and Successful Design Choices

- **Idempotent ETL:** UPSERT logic ensures that data synchronization remains correct even under repeated executions or system restarts.
- **Change-aware ingestion:** Incremental loading based on file modification timestamps minimizes I/O and avoids unnecessary churn in the staging layer.
- **Strict normalization:** Moving from JSONB to strongly typed 3NF tables reduces redundancy, isolates update responsibilities, and improves query quality.

- **Full automation:** The pipeline operates continuously without human intervention, automatically triggering staging and production loads at scheduled intervals.
- **Operational observability:** Detailed run metadata (status, timestamps, error causes, row counts) enables reliable monitoring and debugging.

Identified Limitations

- **Dataset coupling to container:** Since `dataset.csv` is bundled into the Docker image, updating data requires a full rebuild instead of hot reloading.
- **Staging schema has minimal validation:** Malformed or missing fields in the input may only be detected later during production load.
- **Split ETL stages may cause temporary inconsistency:** If staging succeeds but production fails, data may be in a partially refreshed state until the next scheduled run.
- **Performance scalability:** JSONB queries and full-table scans may become costly as dataset size grows beyond a few thousand records.

Planned Improvements

- **External dataset volume + checksum-based change detection:** Mounting the input file via Docker volume and checking SHA256 fingerprints would enable dynamic updates and stronger change validation.
- **Transactional production load:** Wrapping the full `load_production_data()` execution inside a single transaction ensures all-or-nothing consistency.
- **Early validation rules in staging:** Introducing JSON schema or `CHECK` constraints (e.g., numeric casting) at ingestion time would reduce rejected rows during later processing.
- **Indexing strategy:** Adding dedicated indexes on `(content->>'student_id')` and foreign-key columns improves lookup performance during upserts.
- **Extended monitoring:** Tracking execution duration and error categories would provide more actionable operational insights in production environments.

Overall Assessment

The current system fully satisfies project requirements by integrating automation, normalization, and incremental data refresh. The listed improvements would further enhance maintainability, fault tolerance, and scalability, making the pipeline suitable for real-world workloads with continuously evolving datasets.

Appendix A: File Map

`01_database_setup.sql`	Create DB, schemas (`staging, production, etl_logs`), enable extensions.
`02_staging_schema.sql`	Staging table, job log table, staging loader & helper functions, cron job for ingestion.
`03_production_schema.sql`	Production tables with constraints, production loader function, cron job for ETL.

`Dockerfile`	PostgreSQL 16 + pg_cron; copies dataset and init scripts; sets preload libraries.
`install.sh`	Convenience script to install Docker and Compose on Ubuntu.

Compilation Notes

This document uses standard LaTeX packages. Compile with `latexmk -pdf DBD_Documentation.tex`. If you see a "rerun" notice, compile a second time to refresh bookmarks/outlines.