Reproducible Postgres load of Tuva seed datasets.*

Ben Bubnick[1,†]

[1] *GrowData Analytics, Ashland, OH 44805, USA*

EXECUTIVE SUMMARY

Purpose. Reproducible Postgres load of Tuva seed datasets with robust validation. This project standardizes how Tuva seeds are created, loaded, and verified in PostgreSQL, ensuring consistent outcomes across environments and teams.

Scope. Core data models (patients, encounters, claims, eligibility, clinical events, locations, practitioners), a comprehensive terminology layer (e.g., ICD, LOINC, SNOMED CT, CVX, HCPCS, MS-DRG), smoke and add-on test suites, automated CI execution on Postgres 16, and SQL linting with pre-commit enforcement.

Outcomes. Reliable schema, quality gates, one-click CI pipeline, developer ergonomics. Specifically:

Approach—

- **Data Modeling:** Modular core tables with deferrable foreign keys where appropriate; terminology tables enable validation and normalization without over-constraining source variability.
- **Validation Strategy:** Baseline smoke tests for structural integrity; domain add-ons for clinical and claims semantics (e.g., LOINC status surfacing, MS-DRG deprecation, immunization series spacing, eligibility continuity).
- **Automation:** GitHub Actions workflow provisions a Postgres 16 service container, runs `make create-db load test`, prints failing checks first, and uploads `test_results.csv`.
- **Developer Experience:** Pre-commit hook integrates sqlfluff via a psql-aware wrapper; optional manual "fix" stage supports consistent formatting without surprise rewrites.

Architecture Overview—

- **Inputs:** `data/*.csv` with headers matching table DDL.
- **Processing:** Optional CSV normalization; `\copy` ingestion; terminology loaded via seeds or external sources where sets are large.
- **Outputs:** Populated core and terminology schemas; `test_results` summary for CI dashboards and audits.

Keywords: Healthcare ETL (352) — CI/CD data pipeline (962) — SQL linting (1489) — Postgres (803) — data quality testing (763) — Tuva seed datasets (4)

1. BACKGROUND & OBJECTIVES

1.1. *Context and Motivation*

Modern healthcare analytics teams frequently start from heterogeneous, partner-specific data extracts that vary in schema, granularity, and code systems. This diversity makes early ingestion fragile and slows downstream analysis. The *Tuva* project provides openly documented, widely adopted seed datasets and conventions designed to shorten time-to-value by standardizing core entities and medical terminologies.[2]

* This package supports dbt version 1.3.x or higher.

† Principal Data Engineer

[2] See the Tuva project website: https://thetuvaproject.com/.

This repository operationalizes those seeds on PostgreSQL as a reproducible, auditable load with built-in quality checks. The design goal is to give teams a portable foundation: a minimal but expressive core schema; terminology tables that enable conformance checks; and a test suite that reveals data issues as early as possible without blocking iteration.

1.2. *Why Tuva Seeds*

- **Shared semantics.** Tuva supplies normalized column names, data types, and widely used clinical and claims vocabularies, reducing one-off mapping logic at each deployment.
- **Fast onboarding.** Seed tables allow engineers to verify plumbing and shape before committing to high-volume production feeds.
- **Transparency.** The seed layer is intentionally simple and open; it is suitable for code review, documentation, and repeatable demos.
- **Extendability.** Teams can snap in richer vocabularies (LOINC, SNOMED CT, ICD, MS-DRG, etc.) and soft-validation tests without changing baseline contracts.

1.3. *Target Audiences*

Data Engineering.—Engineers need deterministic, idempotent DDL and loaders; clean separation of core vs. terminology schemas; and guardrails that catch bad data *before* it complicates orchestration. They also value CI feedback loops and consistent code style.

Analytics/BI.—Analysts benefit from a consistent, queryable core with predictable grain (e.g., one row per encounter, claim line, lab result) and terminology lookups that stabilize dashboards and cohort logic.

Quality Assurance (QA)/Data Governance.—QA needs systematic evidence of data health. A standardized test-results table and repeatable smoke/add-on suites make it easy to spot regressions, triage anomalies, and communicate status to stakeholders.

1.4. *Design Principles*

1. **Reproducibility over convenience.** Every structural decision (one-file-per-table DDL, explicit indexes, light CHECK constraints) prioritizes deterministic outcomes across environments.
2. **Isolation by schema.** All DDL and tests are parameterized with psql variables `:"schema"` and `:"terminology_schema"` so multiple test runs and teams can coexist on the same database without collisions.
3. **Fail fast, but softly.** "Smoke" tests and add-ons prefer *visibility* to *rigidity*: they surface violations and outliers early while keeping ingestion unblocked, unless a constraint is clearly safe to enforce.
4. **Lean dependencies.** Loading uses the psql client's `\copy`, avoiding server-side file access and special extensions.[3]
5. **Operational clarity.** A single *standardized* `test_results` table aggregates checks for parsable CI output and artifact export.
6. **Style as a safety net.** SQL linting (sqlfluff) and pre-commit hooks bake consistency into the workflow and reduce review friction.[4][5]

1.5. *Objectives*

O1. One-file-per-table DDL with portable psql variables.—Each table lives in its own SQL file (`db/tables/*.sql`, `db/terminology/*.sql`). A thin wrapper *sources* these files in a deterministic order. All objects are created under `:"schema"` (core) or `:"terminology_schema"` (terminology), enabling developers to run multiple, isolated stacks on a shared database host.

[3] See the PostgreSQL documentation for `COPY/\copy`: https://www.postgresql.org/docs/current/sql-copy.html.
[4] sqlfluff documentation: https://docs.sqlfluff.com/.
[5] pre-commit framework: https://pre-commit.com/.

O2. Deterministic load via `\copy` (no server FS access).—Ingestion relies on client-side `\copy` from CSVs placed in `data/`. This approach:

- avoids server-side shared directories and superuser privileges;
- yields predictable performance characteristics for reproducible benchmarking;
- keeps deployment friction low across laptops, CI workers, and ephemeral runners.

O3. Early data quality detection (smoke + add-ons).—The test suite encodes practical domain checks:

- *Structure/keys:* primary-key nulls/duplicates; soft uniqueness at natural grains (e.g., claim header + line).
- *Referential presence:* foreign-key existence where applicable (e.g., person, encounter).
- *Temporal logic:* start/end ordering, line-within-header windows, plausible event timing.
- *Plausibility:* nonnegative amounts and quantities; LOINC/CVX/ICD/MS-DRG membership; NDC normalization and RXCUI presence.
- *Consistency:* person/encounter alignment; crosswalk coherence; panel integrity (lab/observation); immunization series spacing; eligibility continuity and gap diagnostics.

All checks emit rows into a uniform shape for rollup and trending.

O4. CI visibility via standardized `test_results`.—Every test query projects a triplet `(test, pass, details)` (and where relevant, supporting counts). CI then:

1. runs `create-db` → `load` → `test` in a Postgres 16 service container;[6]
2. prints a sorted summary (failures first) to logs;
3. exports the table as a CSV artifact for reviewers and dashboards.

This pattern keeps pipelines simple while offering auditors a stable interface.

O5. Enforced SQL style (sqlfluff) and pre-commit hygiene.—A strict but team-friendly sqlfluff config enforces UPPERCASE keywords, *snake_case* identifiers, trailing commas, column qualification in multi-table SELECTs, and CTE-first patterns. A psql-aware wrapper normalizes `:"schema"` and nonstandard types solely for linting, preserving production SQL semantics. Pre-commit runs linters on staged files and offers an opt-in fixer, reducing stylistic noise in pull requests.

1.6. *Non-Goals and Constraints*

- **No vendor-specific server extensions.** The project intentionally avoids features that complicate portability (e.g., superuser-only file access).
- **Right-sized constraints.** Only low-risk `CHECK`s and deferrable FKs are enforced in DDL; nuanced domain rules remain soft tests to respect source variance.
- **Large vocabularies out-of-band.** Extremely large code sets (e.g., full provider registries) are supported via external loading paths to keep seed distribution lean.

1.7. *Measures of Success*

- *Time-to-first-load:* minutes from clone to passing baseline tests on a fresh Postgres instance.
- *Signal quality:* proportion of defects caught at the smoke layer vs. downstream.
- *Operational ergonomics:* median PR review time and lint errors per change trending down.
- *Reproducibility:* identical `test_results` distributions across CI runners for the same input bundle.

[6] GitHub Actions documentation: https://docs.github.com/actions.

1.8. *Related Work and Alignment*

The approach aligns with established data engineering practices: immutable DDL artifacts; client-controlled ingestion; continuous testing; and style automation. It complements Tuva's mission of transparent, standards-aware health data modeling by offering a minimal, reproducible Postgres baseline that teams can extend confidently.[7]

2. SYSTEM OVERVIEW

2.1. *High-Level Architecture*

The system implements a reproducible ingestion and validation pipeline for Tuva seed datasets on PostgreSQL. Figure 1 summarizes the end-to-end flow:

```
+--------------------+
|  Inputs            |
|  data/*.csv        |
+---------+---------+
          |
          v
 +--------+---------+       +--------------------------+
 |  Loader Scripts  |  ==>  |  PostgreSQL 16           |
 |  (Make + psql)   |       |  Schemas:                |
 +--------+---------+       |    : "schema"            |
          |                 |    : "terminology_schema"|
          |                 +------------+-------------+
          |                              |
          |                              v
          |                    +---------+---------+
          |                    |  Tests (SQL)     |
          |                    |  smoke + add-ons |
          |                    +---------+---------+
          |                              |
          |                              v
          |                    +---------+---------+
          |                    | ${schema}.        |
          +------------------> |   test_results    |
                               +---------+---------+
                                         |
                                         v
                               +---------+---------+
                               | CI Artifact (CSV) |
                               | GitHub Actions    |
                               +-------------------+
```

Figure 1. High-level architecture: Inputs → Loader scripts → PostgreSQL schemas; tests emit a standardized `test_results` table consumed by CI.

Rationale.—The design codifies three principles: (i) *reproducibility* across laptops and CI runners; (ii) *early visibility* into data quality via uniform test outputs; and (iii) *portability* through conservative Postgres features and client-side loading. This approach aligns with the Tuva project's goal of standardized, transparent health data modeling.[8]

2.2. *Technology Stack*

- **PostgreSQL 16**. Primary datastore; supports modern SQL features and performant bulk ingest.[9]
- **psql** client. Used for schema creation, bulk ingestion via `\copy`, and test execution.[10]

[7] Background on Tuva seeds and modeling approach: https://thetuvaproject.com/.
[8] The Tuva Project: https://thetuvaproject.com/.
[9] PostgreSQL documentation: https://www.postgresql.org/docs/.
[10] `COPY/\copy`: https://www.postgresql.org/docs/current/sql-copy.html.

- **GNU Make**. Orchestrates deterministic target sequences (`create-db` → `load` → `test`).[11]
- **Python 3.x**. Optional CSV normalization utilities and minor glue scripts.[12]
- **GitHub Actions**. CI runner with a Postgres 16 service container, artifact export of `test_results`.[13]
- **sqlfluff**. SQL linting for consistent style and safer reviews.[14]

2.3. *Core Components*

Inputs.—CSV files under `data/` are the canonical inputs. Headers must match the table DDLs. An optional Python normalizer ensures consistent quoting, delimiter, and encoding for bulk ingest.

Loader Scripts.—Make targets and shell scripts run `psql` in a known order. All DDL is idempotent (`CREATE TABLE IF NOT EXISTS`, `ALTER TABLE ... IF NOT EXISTS`) so re-runs are safe. Bulk ingest uses the client-side `\copy` to avoid server filesystem dependencies or superuser privileges.

PostgreSQL Schemas.—Two schemas separate responsibilities:

- `:"schema"`: core entities (patient, encounter, claims, clinical facts, etc.), plus smoke/add-on tests and the `test_results` aggregator.
- `:"terminology_schema"`: value sets and vocabularies (ICD-9/10, LOINC, SNOMED CT, HCPCS, CVX, place of service, MS-DRG, etc.).

The schema names are parameterized via psql variables, allowing multiple isolated stacks on a shared Postgres instance.

Testing Layer.—Each test is a plain SQL query that emits `(test, pass, details...)`. Suites cover structural checks (PK/dupes), referential presence, temporal logic, plausibility (e.g., nonnegative amounts, geo bounds), terminology membership (e.g., LOINC/ICD/CVX/DRG), and domain-specific add-ons (e.g., immunization series intervals, panel integrity, eligibility continuity). All tests are *soft by default*: they reveal data issues without blocking ingestion unless a constraint is explicitly enforced in DDL.

Results Aggregator.—A unifying query *UNION*s suite outputs into `${schema}.test_results`. CI reads this table to produce logs and a CSV artifact. The consistent shape simplifies dashboards, trend analysis, and audits.

2.4. *Data Flow*

1. Initialization (`create-db`).—

1. Establish schemas (`CREATE SCHEMA IF NOT EXISTS` for `:"schema"` and `:"terminology_schema"`).
2. Source one-file-per-table DDL (core and terminology).
3. Create lightweight `CHECK`s and deferrable FKs appropriate for heterogeneous sources.
4. Create helpful indexes (generally post-load; guarded with `IF NOT EXISTS`).

2. Loading (`load`).—

1. Validate headers and expected columns.
2. Ingest CSVs via `\copy` to their target tables.
3. Optionally run `ANALYZE` or targeted `VACUUM (ANALYZE)` for predictable test performance.

[11] GNU Make manual: https://www.gnu.org/software/make/manual/make.html.
[12] Python: https://www.python.org/doc/.
[13] GitHub Actions: https://docs.github.com/actions.
[14] sqlfluff: https://docs.sqlfluff.com/.

3. Testing (`test`).—

1. Execute smoke and add-on test files in a deterministic order.
2. Populate the standardized `test_results` table.
3. Export a CI-friendly summary and (optionally) detailed CSV.

2.5. *Schema Roles and Boundaries*

Core schema (`:"schema"`): holds operational tables at stable grain (e.g., *one row per claim line*, *one row per lab result*). Constraints are intentionally light to avoid rejecting plausible but imperfect upstream data. Soft tests handle nuanced validation, emitting visibility instead of hard failures.

Terminology schema (`:"terminology_schema"`): provides lookup tables and controlled value sets. Very large vocabularies (e.g., full provider registries) may be supplied externally; tests still check membership where feasible. This decoupling keeps seed distribution lean while enabling robust conformance checks.[15]

2.6. *Continuous Integration (CI)*

The CI job runs in a Linux runner with a Postgres 16 service container. Steps:

1. Bring up the database and wait for health (`pg_isready`).
2. Set environment variables (`PG*`, `schema`, `terminology_schema`).
3. Run `make create-db load test`.
4. Print a sorted summary of `test_results` (failures first) and upload a CSV artifact.

The job is intentionally simple: it treats the database as an ephemeral environment, ensuring parity between local and CI runs.[16]

2.7. *Style and Tooling*

A strict sqlfluff configuration enforces:

- UPPERCASE keywords; *snake_case* identifiers;
- trailing commas in column lists; clear `SELECT`/`UNION` layout;
- column qualification in multi-table `SELECT`s; preference for CTEs over complex `JOIN` subqueries.

A psql-aware wrapper normalizes `:"schema"` and nonstandard types for *linting only*, keeping production SQL intact. Pre-commit hooks run linters before each commit, lowering review friction and maintaining consistency.[17]

2.8. *Operational Characteristics*

Reproducibility.—Idempotent DDL and deterministic Make targets ensure that repeated runs yield stable outcomes. Client-side `\copy` avoids server-specific configuration.

Portability.—The system relies solely on widely available Postgres features and `psql`; no superuser or server filesystem access is required.

Observability.—The `test_results` table provides a single pane of glass for data quality. Because it is ordinary SQL, teams can visualize it in BI tools or export snapshots from CI logs.

Security and Configuration.—Credentials are provided via environment variables or a local `.env` file and are not committed to source control. Schema names are parameterized, enabling namespace isolation per developer, branch, or environment.

[15] Tuva emphasizes standards-aligned modeling and transparent conventions: https://thetuvaproject.com/.
[16] GitHub Actions services: https://docs.github.com/actions/using-containerized-services.
[17] sqlfluff: https://docs.sqlfluff.com/; pre-commit: https://pre-commit.com/.

Performance.—`\copy` provides efficient bulk ingest; indexing after load (or guarded creation) yields predictable test runtimes. Where volumes warrant, parallelization and partitioning are straightforward extensions.

Failure Modes and Recovery.—Most failures manifest in the test phase as non-passing rows; ingestion continues to completion, preserving the evidence necessary for triage. DDL is rerunnable; data may be reloaded table-by-table or wholesale depending on the issue's scope.

2.9. *Extensibility*

The architecture supports incremental additions:

- New core tables follow the one-file-per-table convention and integrate into the test aggregator.
- New terminology sets slot into `:"terminology_schema"` with minimal coupling.
- Additional test suites can be composed and unioned into `test_results` without changing CI.

This modularity preserves the project's original aim: a portable, auditable foundation for standardized health data modeling aligned with Tuva.[18]

3. DATA MODEL (CORE)

3.1. *Overview*

The core schema provides a minimal, consistent foundation for representing people, encounters, claims, orders, observations, and administrative context derived from Tuva seed datasets.[19] Each entity is modeled with a clear grain ("one row per x"), conservative constraints, and soft foreign-key assumptions so teams can ingest heterogeneous sources without blocking iteration. Where strong referential integrity is practical (e.g., `patient.person_id`), we use *deferrable* foreign keys to enable flexible load ordering.[20] Time intervals (e.g., admit/discharge, claim header/line windows) are represented with explicit start/end columns and tested for ordering. Binary state is captured with tri-state flags {0,1,NULL} guarded by light `CHECK` constraints, preserving unknowns while detecting invalid values.

3.2. *Entity–Relationship (ER) Mini-Diagram*

Figure 2 illustrates key relationships at a glance. Solid arrows indicate common, enforced or soft FKs; dotted arrows indicate optional links that may be absent depending on the source.

3.3. *Key Design Patterns*

Natural vs. Surrogate Keys.—The model favors *surrogate* primary keys for rows that amalgamate multiple source identifiers (e.g., `medical_claim_id`, `lab_result_id`), while preserving *natural* identifiers for cross-system matching (e.g., `claim_id`, `accession_number`, `npi`). This enables stable deduplication, change tracking, and late-arriving data handling without over-constraining input feeds.

Soft FK Assumptions.—Referential presence is *checked* via tests and often *enforced* with DEFERRABLE FKs where feasible (e.g., `encounter.person_id` → `patient.person_id`). Where sources are incomplete or multi-tenant, we rely on smoke tests to surface missing links rather than blocking loads.

Date/Interval Semantics.—Start/end columns define closed or half-open intervals depending on source semantics; tests assert ordering (e.g., `end_date` ≥ `start_date`) and soft containment (e.g., line within header window). This yields predictable cohorting and length-of-stay calculations.

Flags with Light `CHECK`s.—Boolean-like columns are {0,1,NULL} with `CHECK` constraints to prevent invalid encodings while preserving unknowns. This supports downstream aggregation that treats NULL as "not recorded."

The following catalog details each table's grain, purpose, core columns, typical relationships, and quality considerations.

[18] The Tuva Project overview: https://thetuvaproject.com/.
[19] The Tuva Project: https://thetuvaproject.com/.
[20] PostgreSQL deferrable constraints: https://www.postgresql.org/docs/current/sql-altertable.html#SQL-ALTERTABLE-SET-CONSTRAINTS.

```
             +--------------+                 +------------------+
             |  practitioner|                 |    location      |
             |(practitioner)|                 | (location_id)    |
             +------+-------+                 +---------+--------+
                    ^                                   ^
                    |                                   |
                    |                                   |
+---------+   +-----+-----+     +----------------+      |    +------------------+
| patient |   | encounter |<----|  medical_claim |-----------| appointment      |
|(person) |   |(encounter)|     | (medical_claim)|      |    | (appointment_id) |
+----+----+   +--+----+---+     +-------+--------+      |    +---------+--------+
     ^           ^    ^                 ^               |              ^
     |           |    |                 |               |              |
     |           |    |                 |               |              |
     |           |    |                 |               |              |
+----+----+   +--+----+----+     +-------+--------+     |     +-------+--------+
|crosswalk|   |  procedure |     | pharmacy_claim |-----------| immunization   |
|(person)  |  |(procedure) |     | (pharmacy_clm) |     |     | (immunization) |
+----+----+   +--+----+----+     +-------+--------+     |     +-------+--------+
     ^           ^    ^                  ^              |              ^
     |           |    |                  |              |              |
     |           |    |                  |              |              |
     |      +----+----+----+      +-----+-----+         |       +-----+-----+
     |      |  observation  |     | lab_result |         |        | medication |
     |      | (observation) |     | (lab_res)  |         |        |(medication)|
     |      +--------+-----+      +-----+------+         |        +-----+------+
     |               ^                 ^                 |              ^
     |               |                 |                 |              |
     |               +-----------------+-----------------+--------------+
     |
+----+------------------+
|     eligibility       |
| (eligibility_id)      |
+-----------------------+
```

```
Legend: Primary keys in parentheses. Arrows generally indicate foreign-key or soft-key references via person_id,
encounter_id, claim_id, location_id, and practitioner_id.
```

Figure 2. Core ER mini-diagram (simplified).

3.4. *patient*

Grain: One row per person. **PK:** `person_id`. **Purpose:** Central identity record with demographics (name parts, sex/gender, race, ethnicity), birth/death data, postal contact fields, and convenience attributes (age, age group). **Relationships:** Referenced by nearly all clinical and claims facts via `person_id`. **Quality:** Soft membership checks for `gender`, `race`, `ethnicity` against terminology; age derived from `birth_date` and run timestamp. **Notes:** Demographics may be sparse or conflicting; keep most fields nullable and prefer soft tests over hard constraints.

3.5. *encounter*

Grain: One row per encounter/visit. **PK:** `encounter_id`. **Purpose:** Episode context (admit/discharge dates, encounter type/group, facility, provider roles, financial tallies). **Relationships:** `person_id` → `patient`; joined from clinical facts (`procedure`, `observation`, `lab_result`, `condition`, `medication`, `immunization`) and claims; optional links to `location`. **Quality:** Ordering of `encounter_end_date` ≥ `encounter_start_date`; admit/discharge code membership; primary diagnosis presence; DRG membership. Flags (`ed_flag`, `lab_flag`, etc.) tri-state with checks.

3.6. *person_id_crosswalk*

Grain: One row per (`person_id`, `patient_id`/`member_id`, payer, plan) mapping. **PK:** Composite or surrogate. **Purpose:** Bridges local identifiers across clinical and claims sources; supports multi-payer and multi-system identity.

Relationships: Drives reconciliation tests ensuring claims rows agree with person mappings. **Quality:** No overlaps on the same member+plan for a given `person_id` unless intentionally versioned.

3.7. *medical_claim*

Grain: One row per claim line (or header where line granularity is absent). **PK:** `medical_claim_id`. **Purpose:** Institutional/professional claims with service dates, place of service, revenue center, HCPCS/CPT+modifiers, provider roles (rendering, billing, facility), and financials (paid/allowed/charge and patient liabilities). **Relationships:** Soft link to `encounter` via `encounter_id`, to `patient` via `person_id`; joins to terminology (place of service, revenue center, DRG, HCPCS, modifiers, bill type, claim type). **Quality:** Header/line date ordering; soft containment (line within header); nonnegative money/units; money monotonicity (`paid` $\leq$ `allowed` $\leq$ `charge`); membership in code sets; soft duplicate detector on (`claim_id`, `claim_line_number`, `data_source`).

3.8. *pharmacy_claim*

Grain: One row per pharmacy claim line/dispense. **PK:** `pharmacy_claim_id`. **Purpose:** Dispenses with NDC, quantities, days supply, refills, and cost fields. **Relationships:** `person_id` $\rightarrow$ `patient`; joins to NDC terminology (digits-only matching) and possibly RxNorm via `rxcui`. **Quality:** Nonnegative quantities/costs; `paid_date` $\geq$ `dispensing_date`; membership of NDC in terminology and presence of `rxcui`; optional crosswalk coherence for `member_id`+payer+plan.

3.9. *eligibility*

Grain: One row per enrollment period for a member. **PK:** `eligibility_id`. **Purpose:** Coverage windows with payer, payer type, plan, subscriber relation, group info, and Medicare-specific attributes (OREC, dual status, Medicare status). **Relationships:** Joins to `patient` via `person_id`; to terminology (payer type, OREC, dual, status). **Quality:** Interval ordering (`end` $\geq$ `start`); soft non-overlap per (`member_id`, `plan`); continuity/gap diagnostics.

3.10. *procedure*

Grain: One row per performed procedure. **PK:** `procedure_id`. **Purpose:** Captures source and normalized codes (ICD/HCPCS/SNOMED), modifiers, performing `practitioner_id`, and links to `encounter/claim`. **Relationships:** Person/encounter consistency checks; terminology membership for code types and modifiers. **Quality:** Date presence/logic; soft duplicate detection (`person_id`+`procedure_date`+`normalized_code`+`data_source`).

3.11. *observation*

Grain: One row per observation (lab or clinical observation), optionally grouped by `panel_id`. **PK:** `observation_id`. **Purpose:** Source/normalized code types (often LOINC), value fields (result, units, reference ranges), and mapping metadata. **Relationships:** `person_id`, optional `encounter_id`, optional `panel_id`. **Quality:** Code-type presence; if numeric, plausibility against reference ranges; panel/date uniqueness soft checks; LOINC status surfacer (Deprecated/Discouraged).

3.12. *lab_result*

Grain: One row per lab result component, linked to an accession and optionally an order. **PK:** `lab_result_id`. **Purpose:** Normalized order and component (often LOINC), timestamps (collection/result), value and units, abnormal flags, specimen, and ordering practitioner. **Relationships:** `person_id`, `encounter_id`, `ordering_practitioner_id`; terminology LOINC and units normalization. **Quality:** Date ordering (collection $\leq$ result); numeric plausibility; add-on surfacers for Deprecated/Discouraged LOINC; soft dupes by (`accession_number`, `normalized_component_code`); panel integrity (consistent order code and collection window per accession).

3.13. *condition*

Grain: One row per condition/diagnosis instance. **PK:** `condition_id`. **Purpose:** Source/normalized diagnosis codes (ICD/SNOMED), rank within claim (principal=1), present-on-admit code, and clinical dates (recorded, onset, resolved). **Relationships:** Links to `encounter` and optionally `claim`; terminology membership (ICD, POA). **Quality:** Date ordering (onset $\leq$ recorded $\leq$ resolved where present); principal diagnosis uniqueness per (`claim_id`, `data_source`) as an optional add-on; person/encounter consistency.

3.14. *medication*

Grain: One row per medication event (prescribed/dispensed). **PK:** `medication_id`. **Purpose:** Source code type, `ndc_code`, `rxnorm_code`, `atc_code`, route/strength/quantity/days supply, and practitioner attribution. **Relationships:** `person_id`, `encounter_id`, `practitioner_id`; joins to terminology (NDC, RxNorm, ATC). **Quality:** Date ordering (prescribing vs. dispensing where present); nonnegative quantities; membership plausibility for each coding system; soft duplicates by (`person_id`, date, `rxnorm/ndc`, `data_source`).

3.15. *immunization*

Grain: One row per immunization record. **PK:** `immunization_id`. **Purpose:** Source and normalized codes (commonly CVX), status and status reason, occurrence date, dose, lot, site/route, optional practitioner/location. **Relationships:** `person_id`, optional `encounter_id`; joins to CVX and immunization status/route/reason terminology. **Quality:** Code membership and date sanity; soft dupe detection by (`person_id`, `occurrence_date`, `normalized_code`, `data_source`); series spacing add-ons (minimum intervals by vaccine family).

3.16. *appointment*

Grain: One row per scheduled appointment. **PK:** `appointment_id`. **Purpose:** Source/normalized appointment type and status, start/end timestamps and duration, location/practitioner, and coded reasons/cancellation reasons. **Relationships:** Optional link to `encounter`; strict FK to `location` when configured; joins to appointment type/status terminology. **Quality:** Duration = `end` − `start` (where provided); status/type membership; soft dupes by (`person_id`, `start_datetime`, `normalized_type`, `data_source`).

3.17. *location*

Grain: One row per facility/location. **PK:** `location_id`. **Purpose:** Facility name/type, address and lat/long, NPI (facility NPI) where applicable, and organization hierarchy. **Relationships:** Referenced by `encounter` and `appointment`. **Quality:** NPI LUHN validation; geo plausibility (latitude/longitude ranges); ZIP/state normalization; soft test surfacing cross-table presence.

3.18. *practitioner*

Grain: One row per practitioner identity. **PK:** `practitioner_id`. **Purpose:** NPI and name/specialty/affiliation fields representing the actor ordering or performing services. **Relationships:** Referenced by `procedure`, `lab_result` (ordering), `medication` (ordering), and optionally `immunization` and `appointment`. **Quality:** NPI LUHN validation; specialty presence where available; soft presence checks across referencing tables.

3.19. *Implementation Notes*

Idempotent DDL and Indexing.—Each table is defined in its own file with `CREATE TABLE IF NOT EXISTS` and defensive `ALTER TABLE ... IF NOT EXISTS`. Helpful indexes target frequent joins and filters (e.g., person, encounter, dates, code columns). Index creation can be deferred until after bulk loads to improve throughput.

Deferrable Integrity.—Where FKs are enabled, they are typically *DEFERRABLE INITIALLY DEFERRED* to support arbitrary load order and late-arriving dimensions, with a final transaction-level validation step.[21]

Testable Semantics.—Rather than embed complex domain rules in constraints, tests provide a repeatable way to surface violations and drifts across sources while producing a standardized `test_results` table for CI and auditing.

Terminology Decoupling.—Large value sets (ICD-10-CM, LOINC, SNOMED CT, MS-DRG, HCPCS, NDC→RxNorm, CVX, etc.) live in a separate terminology schema. Core tables retain lightweight references (code + type) so membership can be validated when the relevant vocabulary is present, without hard runtime coupling.[22]

3.20. *Core Layer*

Tables can be constructed with LaTeX's standard table environment or the AASTeX's deluxetable environment. The deluxetable construct handles long tables better but has a larger overhead due to the greater amount of defined

[21] Constraint timing discussion: https://www.postgresql.org/docs/current/ddl-constraints.html.

[22] This mirrors Tuva's approach of standards-aware modeling with transparent mapping surfaces: https://thetuvaproject.com/.

mark up used to set up and manipulate the table structure. The choice of which to use is up to the author. Examples of both environments are used in this manuscript.

Tables longer than 200 data lines and complex tables should only have a short example table with the full data set available in the machine readable format. The machine readable table will be available in the HTML version of the article with just a short example in the PDF. Authors are required to indicate in the table comments that the data is in machine readable format in the full article. Authors are encouraged to create their own machine readable tables using the online tool at http://authortools.aas.org/MRT/upload.html.

AASTEX v6 introduced five new table features that were designed to make table construction easier and the resulting display better for PASP authors. The items are:

1. Declaring math mode in specific columns,
2. Column decimal alignment,
3. Automatic column header numbering,
4. Hiding columns, and
5. Splitting wide tables into two or three parts.

Full details on how to create each of these special table types are given in the guidelines at http://journals.aas.org/authors/aastex.html.

4. TERMINOLOGY LAYER

4.1. *Rationale and Scope*

Healthcare data arrives with heterogeneous vocabularies: diagnoses and procedures (ICD, SNOMED CT), laboratory concepts (LOINC), drugs (NDC, RxNorm), billing artifacts (HCPCS, MS–DRG, place of service), and demographic/value sets. Without a normalization layer, analytics teams must repeatedly re-implement the same mappings and quality checks for every source feed. The *Terminology Layer* in this project provides:

1. a **one-file-per-table** catalog of value sets and vocabularies,
2. **stable, natural keys** (the code itself) with descriptive attributes,
3. **membership/deprecation surfacing** via lightweight SQL tests, and
4. **join-ready indexes** on code columns for fast conformance checks.

This design aligns with the Tuva project's emphasis on standards-aware, transparent modeling.[23] Core facts (e.g., `medical_claim`, `observation`) retain source codes and optional normalized codes/types; tests then *left join* to terminology to surface unknown or deprecated values instead of enforcing brittle hard FKs at ingest time.

4.2. *Design Principles*

Natural keys & idempotent DDL.—Each terminology table uses the published code as the primary key (e.g., `icd_10_cm`, `loinc`, `ms_drg_code`). DDL is idempotent (`CREATE TABLE IF NOT EXISTS`, guarded `ALTER TABLE`) to support re-runs.

Soft conformance.—Conformance is *observed*, not *enforced*: membership checks occur in test suites that project to the standardized `test_results` table. Where safe, some tables include light constraints (e.g., boolean-like flags {0,1,NULL}).

Versioning and status.—Several vocabularies publish activity or status flags. We store the relevant signals so tests can surface deprecated/discouraged usage (e.g., LOINC `status`, MS–DRG `deprecated`, ICD–10–CM `header_flag`).

Separation of concerns.—Terminology lives in `:"terminology_schema"`; core facts live in `:"schema"`. This supports multi-tenant runs and fast rebuilds without reloading large value sets.

[23] The Tuva Project: https://thetuvaproject.com/.

Normalization helpers.—Where inputs vary (e.g., NDC formatting), helper logic establishes a canonical comparison (digits-only NDC) for reliable membership checks.

4.3. *Implemented Catalog (Grouped)*

CLINICAL

LOINC.—The LOINC table stores `loinc` (code), clinical names (`short_name`, `long_common_name`), the six-part axes (component, property, time aspect, system, scale, method), classification fields (class code/description/type), and a `status` (Active/Trial/Discouraged/Deprecated) with versioning attributes. This structure enables:

- **membership checks** from `observation.normalized_code` or `lab_result.normalized_component_code`,
- **status surfacers** that count discouraged/deprecated codes used in the dataset, and
- **unit hints** via exemplar units (advisory).

LOINC is maintained by Regenstrief; consult the official distribution for license and fields.[24]

SNOMED CT and Transitive Closure.—We include a base `snomed_ct` table (code, description, activity, dates) and a `snomed_closure` table of parent/child edges. A helper view returns all descendants for a given concept, enabling queries such as "all findings under diabetes mellitus." This supports flexible cohorting and normalization for problems/conditions and some procedures.[25]

ICD Families.—We separate diagnosis and procedure code sets:

- **ICD–9–CM** (diagnoses): code + short/long descriptions.[26]
- **ICD–9–PCS** (procedures): code + descriptions (historical).
- **ICD–10–CM** (diagnoses): code + `header_flag` to mark non-billable "header" concepts, short/long descriptions. Tests can surface any header-only codes observed in claims/clinical facts.[27]
- **ICD–10–PCS** (procedures): code + description.[28]

CLAIMS

Place of Service.—A compact code → description table used by `medical_claim`. Tests flag unknown place-of-service codes.

Bill Type.—Stores `bill_type_code`, description, and a deprecation indicator/date. Tests (soft) surface bill types either unknown or marked as deprecated.

MS–DRG & MDC.—The MS–DRG table contains `ms_drg_code`, associated `mdc_code`, a medical/surgical flag, description, and deprecation markers. A companion table holds fiscal-year-specific weights and LOS (`ms_drg_weights_los`), mirroring CMS publications. Tests provide:

- membership checks for DRGs used by `encounter` or `medical_claim`, and
- surfacers for any DRG used that is flagged deprecated.

The `mdc` table (code, description) completes the linkage for DRG groupings.[29]

DEMOGRAPHICS

Gender, Race, Ethnicity, Payer Type.—These small, stable value sets back `patient` and `eligibility`. Tests count distinct unknown codes observed, supporting incremental normalization (e.g., mapping free-text to codes). For cross-ecosystem portability, we preserve the simple `code` + `description` pattern.

[24] LOINC: https://loinc.org/.
[25] SNOMED International: https://www.snomed.org/.
[26] Historical ICD–9–CM references: U.S. CDC/NCHS.
[27] ICD–10–CM: U.S. CDC/NCHS https://www.cdc.gov/nchs/icd/icd10cm.htm.
[28] ICD–10–PCS: U.S. CMS https://www.cms.gov/medicare/icd-10/icd-10-pcs.
[29] CMS MS–DRG references: https://www.cms.gov/.

IMMUNIZATION

CVX.—The CDC-maintained CVX table provides vaccine codes with short/long descriptions. Membership tests link `immunization.normalized_code` to CVX and surface unknowns.[30]

Route, Status, Status Reason.—Three small tables codify route codes (e.g., IM, SQ), status codes (administered, not administered), and a status-reason set (`reason_code` plus `code_type` and description) to support interoperable "why not administered" signals (often SNOMED or ICD-coded).

PROVIDER/LOCATION

HCPCS (Level II) and Modifiers.—The HCPCS L2 table stores the code with short/long descriptions; the modifiers table encodes the two-character modifiers. These are referenced by `medical_claim.hcpcs_code` and modifier columns. Tests surface any HCPCS not present in the vocabulary (ignoring sequence/record-id metadata in source files).[31]

Other Provider Taxonomy.—A mapping of `npi` ↔ taxonomy code (with Medicare specialty and a primary flag) supports analytics on provider specialization. A partial-unique index ensures at most one primary taxonomy per NPI.

NDC → RxNorm/FDA.—A crosswalk table stores digits-only `ndc`, `rxcui`, an RxNorm description, and an FDA description. Soft tests:

- flag NDCs used in `pharmacy_claim` that are *absent* from the crosswalk, and
- surface rows with *missing* `rxcui` for downstream curation.

Reference sources: FDA NDC Directory and NLM RxNorm.[32][33]

ADMINISTRATIVE

Admit Type, Admit Source, Discharge Disposition, Encounter Type.—Compact code sets providing hospital admission sources/types (including newborn-specific description where relevant), discharge dispositions, and a two-column encounter grouping (`encounter_group`, `encounter_type`). Tests report any types present in core data that are missing from these terminologies, and (optionally) summarize by encounter group.

4.4. *Integration with Core Facts*

Join patterns.—Core tables typically carry a `source_code` and optional `normalized_code` with a `normalized_code_type`. Membership tests use *left joins* against the corresponding terminology table, preserving all fact rows and surfacing unknowns by `WHERE t.code IS NULL`. For NDC, a preprocessing step applies digits-only normalization on both sides before joining.

Status/deprecation surfacers.—For vocabularies that encode activity or deprecation, tests emit counts of disallowed or discouraged usage. Examples:

- LOINC `status` ∈ {Deprecated, Discouraged} observed in `observation`/`lab_result`.
- MS–DRG `deprecated`=1 observed in `encounter`/`medical_claim`.
- ICD–10–CM `header_flag`=1 codes observed in `condition` or claims diagnoses.

Hierarchy queries (SNOMED CT).—The transitive closure table enables concept expansion without recursion at query time. A helper view ("all descendants of X") lets analysts define cohorts using high-level SNOMED concepts while retrieving all leaf codes.

Performance considerations.—Each terminology table indexes its code column(s). For very large joins (LOINC, ICD families, SNOMED), we *project* only the needed fields and rely on b-tree indexes for equality joins. Because conformance checks are read-mostly, this design remains fast under repeated test runs.

[30] CDC CVX: https://www2a.cdc.gov/vaccines/iis/iisstandards/vaccines.asp?rpt=cvx.
[31] HCPCS L2: CMS https://www.cms.gov/Medicare/Coding/HCPCSReleaseCodeSets.
[32] FDA NDC: https://www.fda.gov/drugs/drug-approvals-and-databases/national-drug-code-directory.
[33] RxNorm (NLM): https://www.nlm.nih.gov/research/umls/rxnorm/.

4.5. *Large Sets Policy*

Some vocabularies are too large or too frequently updated to ship as small seed files. This project adopts a simple policy:

- **Seeds contain** table *definitions* and test logic.
- **Data loading** for large sets is handled by supported adapters pulling from public cloud storage (or organization-internal mirrors), outside of version control.
- **Graceful degradation:** if a large set is not loaded, tests *skip or report* with clear messaging rather than fail hard, keeping ingest workflows usable.

This balances reproducibility (DDL, tests, and small value sets live with the code) against practicality (large, licensed, or rapidly changing vocabularies are retrieved on demand).

4.6. *Quality and Governance*

Traceability.—Each terminology table includes comments documenting provenance and intended use. Teams are encouraged to record source release versions and load dates for auditability.

Curation loop.—Soft test outputs (`unknown_code_count`, `deprecated_row_count`) form a backlog for data governance: normalize free text, add local synonyms, or update mappings. Because the results roll into a standard `test_results` table, CI can gate changes or at least highlight regressions.

Security/licensing.—Some vocabularies (e.g., SNOMED CT) carry license or distribution constraints; teams should confirm entitlements and loading channels for production. The code in this repo intentionally separates structure from data to facilitate compliant operations.[34]

4.7. *Summary*

The Terminology Layer provides an explicit, testable contract between messy, multi-source inputs and the stable semantics expected by analytics and quality processes. By centering on natural-code primary keys, soft conformance, status-aware surfacers, and scalable loading for large sets, it delivers both *rigor* and *pragmatism*—a combination that mirrors the Tuva project's ethos of standards-aligned, transparent modeling.[35]

5. LOADING PIPELINE

5.1. *Objectives*

The loading pipeline aims to *reproducibly* materialize Tuva-aligned datasets into PostgreSQL using a deterministic, scriptable sequence. It balances three competing needs: (1) portability across laptops/CI, (2) early validation without over-constraining real-world data, and (3) predictable performance on bulk ingest. The pipeline relies only on standard PostgreSQL features and the `psql` client, orchestrated with GNU Make.[36][37]

5.2. *High-Level Flow*

Figure 3 shows the end-to-end sequence. Optional CSV normalization precedes client-side ingestion via `\copy`, followed by tests and result aggregation.

5.3. *Make Targets & Sequence*

The pipeline is encoded as explicit Make targets to guarantee ordering and re-runs. A minimal skeleton:

```
create-db:  ## Build schemas, tables, constraints, views
    psql -v schema=$(schema) -v terminology_schema=$(terminology_schema) \
         -f db/bootstrap.sql
```

[34] SNOMED CT licensing: https://www.snomed.org/get-snomed.
[35] See https://thetuvaproject.com/ for Tuva's broader approach.
[36] PostgreSQL documentation: https://www.postgresql.org/docs/.
[37] GNU Make manual: https://www.gnu.org/software/make/manual/make.html.

```
+-------------------+     +-------------------------+     +-----------------+
|  data/*.csv       | --> | normalize_csvs.py (opt) | --> |  make load      |
+---------+---------+     +------------+------------+     +----+------------+
          |                            |                       |
          v                            v                       v
   Header checks                 Canonical CSV        psql \copy -> core tables
  (names, order)                 (enc, delim, NA)     (idempotent, batched)
          |                                                  |
          v                                                  v
+---------+---------+                                +------+------+
| make create-db  |                                | make test   |
| schemas+DDL+FKs |                                | smoke+add-on|
+---------+---------+                                +------+------+
          |                                                  |
          v                                                  v
     Post-load indexes                                ${schema}.test_results
```

Figure 3. Loading pipeline: initialization → (optional) normalization → ingest → test.

```
normalize:  ## Optional CSV cleanup (encoding, delimiters, headers)
    python scripts/normalize_csvs.py data

load:       ## Bulk ingest via psql \copy (client-side)
    bash scripts/load_to_postgres.sh data

test:       ## Execute smoke/add-on tests and aggregate results
    psql -v schema=$(schema) -f db/tests/run_all.sql
```

Ordering.—The canonical run is:

`make create-db` → `python scripts/normalize_csvs.py` (optional) → `make load` → `make test`.

Make *dependencies* can encode the run list (`test: load`, `load: create-db`) to avoid out-of-order execution.

5.4. *Initialization:* `make create-db`

Schemas.—The pipeline creates two namespaces, parameterized for multi-tenant runs:

- `:"schema"` for core tables,
- `:"terminology_schema"` for vocabularies/value sets.

Using `psql` variables allows teams to run multiple isolated stacks on one database without name collisions.

Tables, Constraints, Views.—All DDL is idempotent using `CREATE TABLE IF NOT EXISTS` and guarded `ALTER TABLE` statements.[38] Where foreign keys are practical, they are declared *DEFERRABLE INITIALLY DEFERRED* to tolerate arbitrary load ordering while still enabling end-of-transaction integrity checks.[39] Lightweight `CHECK` constraints (e.g., tri-state flags {0,1,NULL}) prevent egregious errors without rejecting uncertain values.

5.5. *Optional Normalization:* `python scripts/normalize_csvs.py`

The normalizer standardizes CSVs so bulk ingest is predictable across platforms:

- **Encoding:** ensure UTF-8; convert common encodings (*e.g.*, Windows-1252) where needed.

[38] See PostgreSQL DDL reference: https://www.postgresql.org/docs/current/sql-createtable.html.
[39] Deferrable constraints: https://www.postgresql.org/docs/current/ddl-constraints.html.

- **Delimiter/quoting:** adopt a single delimiter (comma) and quote policy; escape embedded quotes per RFC 4180.[40]
- **Headers:** verify exact header names and order as defined by table DDL (header-driven `\copy`).
- **Null markers:** normalize empty fields vs. explicit `NULL` tokens; prefer empty *cells* to represent SQL NULL at load time.
- **Whitespace:** strip BOMs and trailing spaces that can break header matching.

Normalization is not mandatory if upstream sources already conform, but it reduces cross-environment surprises.

5.6. *Ingestion: `make load` via `psql \copy`*

Client-side COPY.—The pipeline uses `\copy` (the `psql` meta-command) instead of server-side `COPY`. This reads files from the client runner and streams into the database, avoiding server filesystem access or superuser privileges.[41]

Header-driven mapping.—Each `\copy` statement targets an explicit column list aligned with the CSV header, guaranteeing column/value order even if future DDL adds columns with defaults.

Transaction boundaries.—Common practice is “one transaction per table” during ingest:

- It provides atomicity (either the table is fully loaded or not changed),
- keeps memory usage bounded,
- and makes it easy to re-run partial loads.

Batching & large files.—For very large inputs, the loader may split files into chunks (e.g., 250–500 MB) and `\copy` them sequentially. This keeps memory steady and provides progress logging. Batching can be implemented by file splitting or by `\copy` from STDIN in streaming mode.

5.7. *Idempotency & Re-runs*

DDL safety.—All create statements use `IF NOT EXISTS`. Index creation and comments are likewise guarded. Where possible, views are created or replaced (`CREATE OR REPLACE VIEW`) to allow non-destructive updates.

Data re-loads.—The loader is explicit about target tables. Re-runs typically:

1. *truncate-and-reload* a table (fastest),
2. or *append* new partitions/batches with a deduplication key,
3. or *stage-and-swap* (load into `_stg` then `INSERT INTO` the base with de-dup `ON CONFLICT`.

Choice depends on upstream guarantees and whether surrogate keys (e.g., `medical_claim_id`) are stable.

Constraints timing.—Because FKs are deferrable, the final `COMMIT` validates referential integrity; if late-arriving dimensions are expected, tests can surface missing references without failing the load.

5.8. *Performance Considerations*

Index timing.—Creating or refreshing *most* indexes post-load reduces ingest time substantially. Options:

- **Create after load (non-concurrent):** fastest build but takes a short exclusive lock; ideal for ephemeral CI and initial loads.
- **Create Index Concurrently (CIC):** allows concurrent writes but cannot run inside a transaction block and is slower; relevant for shared, long-lived environments.[42]

In CI, non-concurrent builds are simplest and fastest.

[40] RFC 4180: Common Format and MIME Type for CSV Files, https://www.rfc-editor.org/rfc/rfc4180.
[41] `COPY`/`\copy`: https://www.postgresql.org/docs/current/sql-copy.html.
[42] `CREATE INDEX CONCURRENTLY`: https://www.postgresql.org/docs/current/sql-createindex.html.

Work memory for index builds.—When building large indexes, `maintenance_work_mem` can be increased for the session to accelerate sorts; this should be tuned judiciously on shared hosts.

ANALYZE.—After bulk loads, run `ANALYZE` (or rely on autovacuum) so the planner has accurate statistics for test queries. This is especially helpful on wide clinical/claims tables.

COPY settings.—Use `FREEZE` only if you understand the vacuum implications; most pipelines should omit it. Explicitly set `NULL ''` in `\copy` to ensure empty fields map to SQL NULL as intended.

Batching.—For very large tables, chunked ingest plus periodic `ANALYZE` (every N rows) can stabilize runtime, and logging progress (% rows or bytes) helps CI observability.

5.9. *Testing & Aggregation*

After ingestion, the pipeline executes smoke and add-on tests. Each test emits a uniform row shape `(test, pass, details...)`. A final aggregator *UNION*s these into `${schema}.test_results`, which CI exports as a CSV artifact and prints sorted (failures first). This yields a parsable signal for reviewers and a durable audit trail.

5.10. *Operational Concerns*

Configuration.—The pipeline reads connection settings from environment variables (e.g., `PGHOST`, `PGPORT`, `PGUSER`, `PGPASSWORD`, `PGDATABASE`) and `psql` variables `:"schema"` and `:"terminology_schema"`.[43] This keeps the SQL artifacts portable and environment-agnostic.

Security.—Client-side `\copy` avoids mounting server directories or granting superuser privileges. Credentials are provided via environment or a local `.env` that is not committed.

Failure modes.—Typical failures include header mismatches, encoding issues, or referential gaps. The normalization step and soft tests convert many hard failures into informative, fixable findings. Because DDL is idempotent and loads are table-scoped, recovery is usually a targeted re-run.

5.11. *Illustrative Snippets*

Header-locked `\copy`.—

```
\copy :"schema".patient(person_id, first_name, last_name, birth_date, sex, race, ethnicity, data_source)
FROM 'data/patient.csv' WITH (FORMAT csv, HEADER true, NULL '', QUOTE '"', ESCAPE '"');
```

Post-load indexing (guarded).—

```
DO $$
BEGIN
  IF NOT EXISTS (
    SELECT 1 FROM pg_class c JOIN pg_namespace n ON n.oid=c.relnamespace
    WHERE c.relname='patient_person_idx' AND n.nspname = :'schema'
  ) THEN
    EXECUTE format('CREATE INDEX patient_person_idx ON %I.patient(person_id);', :'schema');
  END IF;
END$$;
```

Makefile guard (re-runnable).—

```
.PHONY: create-db load test normalize

test: load
load: create-db
create-db:
    psql -v schema=$(schema) -v terminology_schema=$(terminology_schema) -f db/bootstrap.sql
```

[43] `psql` variables and meta-commands: https://www.postgresql.org/docs/current/app-psql.html.

5.12. *Local vs. CI*

Locally, developers typically point at a running PostgreSQL (Docker or native), set `.env`, and run the sequence. In CI, a containerized Postgres 16 service executes the same targets and publishes `test_results.csv` as an artifact. Because both paths call the same Make targets and SQL, parity is high.[44]

5.13. *Future Enhancements*

- **Incremental loads:** stage-and-merge patterns (`INSERT ... ON CONFLICT`) or partitioned tables by month for claims/observations.
- **Parallel ingest:** run `\copy` on disjoint tables in parallel where I/O and CPU permit.
- **Data lineage:** persist file checksums and row counts per load in a simple audit table.
- **Severity tiers:** classify tests as *error*/*warn*/*info* with CI gates on errors only.

5.14. *Summary*

The pipeline combines idempotent DDL, optional but practical CSV normalization, and client-side bulk ingest to deliver a portable, auditable loading process. Guarded constraints and deferrable FKs preserve flexibility across diverse inputs, while a standardized `test_results` surface provides immediate quality feedback to engineers, analysts, and QA alike.

6. DATA QUALITY & TESTING STRATEGY

6.1. *Purpose and Philosophy*

The testing strategy provides fast, repeatable feedback on the health of Tuva-aligned data once it lands in PostgreSQL.[45] Tests are intentionally *SQL-first*: every check is a self-contained query with a clear name and a boolean pass/fail plus minimal diagnostics. Most checks are *soft* (they *surface* issues without blocking ingestion) because heterogeneous sources frequently contain exceptions that require triage rather than immediate rejection. A small set of lightweight `CHECK` constraints (e.g., tri-state flags {0,1,NULL}) and DEFERRABLE foreign keys backstop egregious errors while preserving portability across environments.[46]

6.2. *Test Taxonomy*

We group tests into complementary layers. Each layer answers a different question, from "is the table *structurally* sane?" to "does this value make *clinical/claims* sense?" The layers are additive; any table can participate in multiple families of tests.

1. **Smoke Tests (Per-table Invariants).** Universal, fast checks that catch structural defects:
    - Primary key not-null and unique; absence of duplicate business keys where applicable.
    - Foreign-key *existence* (soft in tests; hard as DEFERRABLE FKs only when reliable).
    - Date/interval ordering (e.g., `end_date` $\geq$ `start_date`; line within header window).
    - Non-negativity for amounts/quantities; boolean-like flags restricted to {0,1,NULL}.
2. **Add-ons & Domain Checks (Content Validity).** Table-specific logic that encodes external standards and operational common sense:
    - *Membership* in code sets: ICD (CM/PCS), LOINC, CVX, HCPCS, MS–DRG, Present-on-Admission (POA), Place of Service, etc.[47][48][49][50]
    - *Plausibility*: NDC length and "not-all-zeros", lat/long within world bounds, ZIP/state normalization.

[44] GitHub Actions container services: https://docs.github.com/actions/using-containerized-services.
[45] The Tuva Project emphasizes standards-aware, transparent modeling: https://thetuvaproject.com/.
[46] PostgreSQL constraints and deferrable integrity: https://www.postgresql.org/docs/current/ddl-constraints.html.
[47] LOINC: https://loinc.org/.
[48] CDC CVX: https://www2a.cdc.gov/vaccines/iis/iisstandards/vaccines.asp?rpt=cvx.
[49] HCPCS Level II (CMS): https://www.cms.gov/Medicare/Coding/HCPCSReleaseCodeSets.
[50] MS–DRG, MDC (CMS resources): https://www.cms.gov/.

- *Consistency*: person ↔ encounter alignment, crosswalk coherence (member → person), claim line dates within claim header windows.

3. **Soft Duplicate Detectors.** Heuristics that reveal probable duplicates without enforcing a rigid uniqueness key (e.g., procedure by `person_id+date+normalized_code+data_source`; immunization by `person_id+occurrence_date+normalized_code`).

4. **Specialty Add-ons.** Higher-level, domain-specific checks:
   - LOINC status surfacers: counts of Deprecated/Discouraged codes used in `observation/lab_result`.
   - Immunization series spacing: minimal intervals between doses of the same CVX family.
   - Eligibility continuity and gap detection: soft non-overlap and gap flags per (`member_id`, `plan`).
   - Panel integrity: all lab components for an accession share an order code and fall in a coherent collection window; similarly, `observation.panel_id` exhibits consistent code/type and "tight" date window.

6.3. *Implementation Pattern (Per Test)*

Every test query follows a consistent shape for CI consumption:

- **Name**: a stable `test` identifier (`snake_case`) describing the intent (e.g., `medical_claim_money_relationships`).
- **Pass flag**: a boolean `pass` expression.
- **Details**: minimal, numeric/text fields for counts or the number of offending rows.

A canonical skeleton:

```
SELECT 'table_pk_unique' AS test, (COUNT(*) = 0) AS pass, COUNT(*) AS dup_count
FROM (
  SELECT pk FROM :"schema".table GROUP BY pk HAVING COUNT(*) > 1
) d;
```

Each suite emits one or more rows; a final aggregator *UNION*s all outputs into `${schema}.test_results`.

6.4. *Smoke Tests (Illustrative)*

Primary Keys & Duplicates.—For every core table, assert non-null PKs and no repeated PKs. Where natural grains are expected (e.g., claim header + line number), include a *soft* uniqueness test on the composite.

Foreign-key Existence.—Join fact tables to lookup or parent tables, counting missing references when the FK is present in the row. Many real-world sources lack complete keys; the test is informative rather than blocking.

Date Ordering.—Examples include `claim_line_end_date` ≥ `claim_line_start_date`, `discharge_date` ≥ `admission_date`, `result_datetime` ≥ `collection_datetime`. A soft window check validates line dates fit within a claim header window.

Non-negativity & Flags.—Monetary and quantity fields must be ≥ 0 (when not NULL). Flags must be {0,1,NULL}. Lightweight `CHECK` constraints on flags defend the database from invalid encodings while allowing unknowns.

6.5. *Add-ons & Domain Checks*

Membership in Code Sets.—

- *ICD families (CM/PCS)*: verify `condition.normalized_code` (or source code) appears in the respective ICD tables; flag ICD–10–CM `header_flag`=1 used in facts (non-billable "headers").
- *LOINC*: validate `lab_result.normalized_component_code` and `observation.normalized_code` membership.[51]
- *CVX*: confirm immunization normalized codes are in CVX; optional series logic references CVX families.[52]
- *HCPCS & Modifiers*: verify `hcpcs_code` exists in HCPCS L2; modifiers in the allowed modifier list.[53]

[51] LOINC usage and documentation: https://loinc.org/.
[52] CDC CVX tables: https://www2a.cdc.gov/vaccines/iis/iisstandards/vaccines.asp?rpt=cvx.

- *MS–DRG & MDC*: ensure DRGs in encounters/claims exist; surface DRGs flagged deprecated; link to MDC group.[54]

- *POA, Place of Service, Bill Type, Admit/Source/Type, Discharge Disposition*: small code lists that should have near-complete coverage; tests count unknowns by code column.

Plausibility Checks.—

- *NDC*: digits-only normalization; length $\in \{10, 11\}$; not all zeros; membership in NDC $\rightarrow$ RxNorm/FDA crosswalk; surface rows missing `rxcui`.

- *Geo Bounds*: latitude $\in [-90, 90]$; longitude $\in [-180, 180]$; ZIP/state normalization (e.g., state in USPS list, ZIP length 5/9).[55]

- *Money Relationships*: `paid` $\leq$ `allowed` $\leq$ `charge`; patient cost share $\leq$ `allowed`.

Consistency Checks.—

- *Person $\leftrightarrow$ Encounter*: when an encounter is linked, `fact.person_id` matches `encounter.person_id`.

- *Crosswalk coherence*: `member_id`+payer+plan maps to a unique `person_id`; claims rows with a known mapping should carry the same person.

- *Line-in-Header Window*: claim line dates must fall within the claim header window (soft).

6.6. *Soft Duplicate Detectors*

Duplicate detectors identify likely duplicates for manual review. They are *heuristics*, not hard rules. Examples:

- **Procedure**: `person_id`+`procedure_date`+`normalized_code`+`data_source`.

- **Immunization**: `person_id`+`occurrence_date`+`normalized_code`+`data_source`.

- **Observation**: `person_id`+`observation_date`+`normalized_code`+`data_source`.

- **Claims**: (`claim_id`, `claim_line_number`, `data_source`) groups with `COUNT(*)>1`.

The purpose is to flag potential overcounting or feed duplication while allowing legitimate repeats (e.g., two distinct sources).

6.7. *Specialty Add-ons*

LOINC Status Surfacers.—Using the LOINC table's `status` (Active, Trial, Discouraged, Deprecated), tests count discouraged/deprecated usage in `observation` and `lab_result`. This helps prioritize code remediation over time.

Immunization Series Spacing.—For records sharing the same CVX family, enforce minimal intervals between doses as a soft check. The test highlights pairs that violate spacing to support clinical validation.

Eligibility Continuity & Gaps.—Per (`member_id`, `plan`), compute sorted enrollment windows and flag overlaps (soft non-overlap) and gaps >= threshold (e.g., > 1 day) for continuity diagnostics.

Panel Integrity.—For `lab_result`, enforce that components under an accession share a consistent normalized order code and occur within a tight collection window (e.g., within N hours). For `observation`, use `panel_id` similarly to surface inconsistent grouping.

[53] HCPCS Level II (CMS): https://www.cms.gov/Medicare/Coding/HCPCSReleaseCodeSets.
[54] MS–DRG and MDC materials (CMS): https://www.cms.gov/.
[55] USPS two-letter state abbreviations are widely referenced; formal datasets can be joined externally.

6.8. *Aggregation for CI*

All tests output the same columns and are *UNION*ed into `${schema}.test_results`. CI (e.g., GitHub Actions) then:

1. prints a sorted summary (failures first) in logs,
2. exports a CSV artifact for reviewers and dashboards,
3. (optionally) fails the job only on a subset of "error" class tests, keeping "warn/info" tests advisory.

Because `test_results` is an ordinary table, teams can trend pass rates over time or filter by suite.

6.9. *Troubleshooting Views*

To speed root-cause analysis, we add thin "cuts" per suite:

- **Observation suite cut**: last run's counts by test (e.g., numeric plausibility, LOINC membership, panel consistency).
- **Claims suite cut**: money monotonicity failures by payer/plan; unknown HCPCS by service date bucket.
- **Eligibility cut**: coverage overlaps and gap distributions by payer type.

These are simple, parameter-free views that summarize failures without scanning entire fact tables repeatedly.

6.10. *Operational Considerations*

Performance.—Tests are read-only SQL. Use selective projections and indexed join keys (e.g., code columns, person/encounter, dates). Post-load `ANALYZE` improves planner estimates.[56] Very large terminology joins can project to only the *code* and *status* columns to minimize I/O.

Idempotency.—Tests are re-runnable and side-effect free except for populating/refreshing `test_results`. Where needed, truncate and rebuild `test_results` per run.

Graceful Degradation.—If a large terminology set is not loaded (e.g., SNOMED or full LOINC), membership tests either *report "skipped"* (with a distinct test name) or return zero rows, depending on configuration. This keeps ingestion usable while still encouraging full conformance.

Naming and Documentation.—Test names are stable and descriptive; comments explain intent and expected pass conditions. This supports auditor review and change tracking over time.

6.11. *Example Snippets*

Uniform `test_results` shape.—

```
-- Example: nonnegative paid_amount in medical_claim (soft)
SELECT 'medical_claim_paid_nonneg' AS test,
       (COUNT(*) = 0) AS pass,
       COUNT(*) AS fail_count
FROM :"schema".medical_claim
WHERE paid_amount IS NOT NULL AND paid_amount < 0;
```

Aggregator (excerpt).—

```
CREATE TABLE IF NOT EXISTS :"schema".test_results(
  test text, pass boolean, details bigint, run_ts timestamp default now()
);
```

[56] ANALYZE and statistics: https://www.postgresql.org/docs/current/sql-analyze.html.

```
TRUNCATE :"schema".test_results;

INSERT INTO :"schema".test_results(test, pass, details)
SELECT test, pass, COALESCE(row_count, fail_count, unknown_code_count, 0)
FROM (
  -- UNION ALL across suites; each SELECT has (test, pass, <detail alias>)
  SELECT 'patient_pk_unique' AS test, (COUNT(*)=0) AS pass, COUNT(*) AS fail_count
  FROM (SELECT person_id FROM :"schema".patient GROUP BY 1 HAVING COUNT(*)>1) d
  UNION ALL
  SELECT 'observation_loinc_membership', (COUNT(*)=0), COUNT(*)
  FROM :"schema".observation o
  LEFT JOIN :"terminology_schema".loinc t ON t.loinc = o.normalized_code
  WHERE o.normalized_code_type='LOINC' AND t.loinc IS NULL
  -- ... more SELECTs ...
) s;
```

6.12. *Governance and Workflow Integration*

Severity and Policy.—Teams may classify tests as *error/warn/info*. CI gates can fail on *error* while surfacing *warn/info* for visibility. A small allowlist can temporarily mute known issues pending remediation.

Curation Loop.—Membership and plausibility failures become tickets for data governance: map free text to codes, fix source formatting, update terminology loads, or amend business logic. Because outputs are normalized into `test_results`, it is straightforward to trend "unknown code count" down over time.

Documentation.—Each suite and key test has a short description and practical examples in repository docs. Linking failing tests to troubleshooting views creates a fast path from red signals to diagnosis.

6.13. *Limitations*

- **Soft by design.** Some defects require human judgment; soft tests reveal but do not block.
- **Terminology availability.** When large code sets are absent, membership tests cannot validate; the pipeline degrades gracefully but completeness is reduced.
- **Heuristic duplicates.** Duplicate detectors favor recall over precision; review is required to avoid false positives.

6.14. *Summary*

The strategy blends universal smoke tests, standards-based membership checks, pragmatic plausibility and consistency rules, and higher-order domain add-ons. A uniform `test_results` surface enables CI integration, trending, and auditability. The approach is intentionally *pragmatic*: give engineers, analysts, and QA immediate, stable signals without over-constraining ingestion, while preserving a path to stronger enforcement as sources mature.

7. CI/CD

7.1. *Objectives*

The CI/CD pipeline ensures every change to the repository can be built, loaded into PostgreSQL, validated, and reported in a consistent and auditable way. The goals are:

1. **Determinism** across laptops and runners by using the same Make targets and a Postgres 16 service container.
2. **Fast feedback** through smoke and add-on tests aggregated into a standardized `$ {schema}.test_results` table and exported as a CI artifact.
3. **Quality enforcement** via SQL linting (sqlfluff) and pre-commit checks.
4. **Governance** through branch protection rules and required checks that block merge on red tests or lint failures.

The implementation relies on GitHub Actions,[57] PostgreSQL 16,[58] the `psql` client, GNU Make,[59] and sqlfluff.[60]

7.2. *Workflow Architecture*

We use a single *build-and-test* job with a PostgreSQL 16 service container and an optional *lint* job. The build job executes the canonical sequence:

`make create-db` → `make load` → `make test`,

then exports `test_results.csv`. Environment variables wire both the database connection (`PG*`) and schema isolation (`schema`, `terminology_schema`).

Postgres as a Service Container.—The runner starts an isolated Postgres 16 container alongside the job, ensuring a clean database for each run and high parity with local development.[61]

Env Wiring.—The job sets:

- `PGHOST`, `PGPORT`, `PGUSER`, `PGPASSWORD`, `PGDATABASE`,
- `schema` (e.g., `ci_tuva`), `terminology_schema` (e.g., `ci_terminology`).

The `psql` invocations pass `-v schema` and `-v terminology_schema` to parameterize DDL and tests.

7.3. *Canonical Workflow Definition (YAML)*

The following example shows a minimal but production-ready pipeline. It pins versions, waits for Postgres readiness, runs the build sequence, gates on failures, and uploads artifacts.

```yaml
name: ci

on:
  pull_request:
  push:
    branches: [ main ]

concurrency:
  group: ${{ github.workflow }}-${{ github.ref }}
  cancel-in-progress: true

permissions:
  contents: read

jobs:
  lint:
    name: Lint (sqlfluff & pre-commit)
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with: { python-version: '3.11' }
      - name: Install linters
        run: |
          python -m pip install --upgrade pip
          pip install sqlfluff pre-commit
      - name: Run sqlfluff (lint only)
```

[57] GitHub Actions: https://docs.github.com/actions.
[58] PostgreSQL: https://www.postgresql.org/docs/.
[59] GNU Make: https://www.gnu.org/software/make/manual/make.html.
[60] sqlfluff: https://docs.sqlfluff.com/.
[61] Using containerized services in Actions: https://docs.github.com/actions/using-containerized-services.

```yaml
        run: |
          sqlfluff --version
          sqlfluff lint db/ || (echo "::error ::sqlfluff failures" && exit 1)
      - name: Run pre-commit (repo hooks)
        run: |
          pre-commit run --all-files

  test:
    name: Build, Load, Test (Postgres 16)
    runs-on: ubuntu-latest
    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_PASSWORD: postgres
          POSTGRES_USER: postgres
          POSTGRES_DB: gha
        ports: [ "5432:5432" ]
        options: >-
          --health-cmd="pg_isready -U postgres"
          --health-interval=5s
          --health-timeout=5s
          --health-retries=20
    env:
      PGHOST: 127.0.0.1
      PGPORT: 5432
      PGUSER: postgres
      PGPASSWORD: postgres
      PGDATABASE: gha
      schema: ci_tuva
      terminology_schema: ci_terminology
    steps:
      - uses: actions/checkout@v4

      - name: Install psql client and make
        run: |
          sudo apt-get update
          sudo apt-get install -y postgresql-client make

      - name: Wait for Postgres to be ready
        run: |
          for i in {1..60}; do
            pg_isready -h $PGHOST -p $PGPORT -U $PGUSER && break
            echo "waiting for postgres..."
            sleep 2
          done

      - name: Create DB objects (schemas, tables, constraints, views)
        run: |
          make create-db

      - name: Optional CSV normalization
```

```
        if: ${{ false }}  # set to true if normalize is required
        run: |
          python scripts/normalize_csvs.py data

      - name: Load data
        run: |
          make load

      - name: Run tests (smoke + add-ons)
        run: |
          make test

      - name: Export test_results.csv
        run: |
          psql -At -c "\copy :\"schema\".test_results TO 'test_results.csv' CSV HEADER"

      - name: Summarize failing tests (and fail if any red)
        shell: bash
        run: |
          FAILS=$(psql -tAc "SELECT count(*) FROM :\"schema\".test_results WHERE pass = false;")
          echo "Failing tests: $FAILS"
          psql -P pager=off -c "SELECT * FROM :\"schema\".test_results WHERE pass = false ORDER BY test;"
          if [ "$FAILS" -gt 0 ]; then
            echo "::error ::One or more tests failed"
            exit 1
          fi

      - name: Upload artifact
        uses: actions/upload-artifact@v4
        with:
          name: test_results
          path: test_results.csv
```

7.4. *Step-by-Step Commentary*

1) Lint Job.—The `lint` job installs Python 3.11, sqlfluff, and pre-commit. It runs `sqlfluff lint db/` and the repository's pre-commit hooks over all files. Any failure marks the job red, providing early and low-cost feedback before provisioning a database container.

2) Database Service.—The `services.postgres` block launches Postgres 16 with a health check (`pg_isready`). We expose port 5432 to the runner and use simple credentials for the ephemeral CI database. Because each run is isolated, teardown is automatic.

3) Tooling and Readiness.—We install the `psql` client (and `make`) on the runner, then loop on `pg_isready` to avoid racy connections.

4) Build (`make create-db`).—This target creates the `:"schema"` and `:"terminology_schema"` schemas and sources each one-file-per-table DDL. All DDL uses `CREATE IF NOT EXISTS` and guarded `ALTER`s; foreign keys are *DEFERRABLE INITIALLY DEFERRED* where enabled.[62]

5) Load (`make load`).—The loader streams CSVs via client-side `\copy`, ensuring no server filesystem privileges are needed.[63] Header-locked `\copy` mitigates column order drift.

[62] PostgreSQL constraints: https://www.postgresql.org/docs/current/ddl-constraints.html.
[63] `COPY`/`\copy`: https://www.postgresql.org/docs/current/sql-copy.html.

6) Test (`make test`).—Suites execute smoke invariants, membership checks, plausibility, consistency rules, duplicate detectors, and specialty add-ons. Results are inserted (or refreshed) into `$ {schema}.test_results` with a uniform schema.

7) Reporting and Gating.—The job exports `test_results.csv` and prints a sorted list of failing tests. If any row has `pass=false`, the job exits non-zero, blocking the PR.

7.5. *Artifacts and Observability*

The `actions/upload-artifact` step publishes `test_results.csv` for reviewers, dashboards, and archival. Because `test_results` is an ordinary SQL projection, teams can also add a "last run" summary view (e.g., per-suite counts) and export it similarly. For richer logs, the job prints failing tests inline and can emit a Markdown summary using the GHA job summary API.

7.6. *Branch Policy and Required Checks*

We recommend enabling branch protection on `main` (and long-lived release branches) with required status checks:

- **Lint (sqlfluff & pre-commit)** must pass.
- **Build, Load, Test** must pass (red tests fail the job).

Consider enabling:

- *Require pull request reviews* and *dismiss stale approvals.*
- *Require status checks to pass before merging* (select the two checks above).
- *Do not allow bypassing the above settings* except for trusted release admins.

This policy ensures style and data quality gates are enforced consistently.

7.7. *Security and Supply Chain Considerations*

- **Minimal permissions.** The workflow's `permissions` are set to `contents: read`. Escalate only if needed for releases or PR annotations.
- **Pin actions.** Prefer pinning third-party actions to *immutable commit SHAs* for tamper resistance.[64]
- **Secret handling.** Avoid committing credentials. Use ephemeral service credentials for CI (as shown). For production-like runs, store secrets in repository or environment secrets.
- **Isolation.** Schema names (`schema`, `terminology_schema`) isolate parallel runs on a shared database, but in CI we use a fresh container for maximum isolation.

7.8. *Performance and Stability*

- **Concurrency.** The `concurrency` block cancels in-flight runs on the same branch to reduce cost and queue time.
- **Index timing.** Where large tables are loaded, create indexes *after* load (or guard creation with `IF NOT EXISTS`) to speed ingest.
- **ANALYZE.** Post-load `ANALYZE` improves planner estimates for test queries, reducing flakiness due to poor plans.[65]
- **Retry strategy.** Health checks and explicit readiness loops mitigate transient service start-up races.

[64] GitHub security guidance on pinning actions: https://docs.github.com/actions/security-guides/security-hardening-for-github-actions#using-third-party-actions.

[65] ANALYZE: https://www.postgresql.org/docs/current/sql-analyze.html.

7.9. *Local–CI Parity*

Developers run the same Make targets locally against a Dockerized or native Postgres. Because the CI job simply invokes those targets with environment variables, parity is high: failures reproduced locally will fail identically in CI, and vice versa. This alignment shortens diagnosis time.

7.10. *Future Enhancements*

- **Matrix builds** across Postgres versions (e.g., 15/16/17) for forward-compatibility testing.
- **Nightly schedules** to rebuild terminology and detect drifts in large value sets.
- **Severity tiers** in `test_results` (*error*/*warn*/*info*) with CI gating only on errors.
- **Data lineage** artifacts (file checksums, row counts per table) for incremental loads.

7.11. *Summary*

The CI/CD pipeline provides a reliable, auditable path from source changes to validated database state. By standardizing on Postgres 16 service containers, `psql`-driven Make targets, and a uniform `test_results` aggregator, the workflow delivers fast feedback to engineers, analysts, and QA—while branch protection and required checks institutionalize data quality and code hygiene.

8. SQL STYLE & TOOLING

8.1. *Why a Style Guide?*

A consistent SQL style reduces cognitive load, shortens review cycles, and lowers the risk of subtle errors. In a repository that combines DDL (schemas, constraints), DML (bulk ingest), and diagnostics (tests), style is not cosmetics—it drives *maintainability* and *safety*. We formalize our conventions with a strict `sqlfluff` configuration, enforce them via pre-commit hooks, and automate fixes where feasible.[66] The resulting discipline complements PostgreSQL's robust feature set[67] and our CI/CD pipeline.

8.2. *Style Tenets (Authoritative Rules)*

Our rules are intentionally strict to keep diffs small and reviews fast:

1. **UPPERCASE keywords.** All reserved words and functions appear in UPPERCASE (e.g., `SELECT`, `JOIN`, `COALESCE()`).
2. ***snake_case* identifiers.** Schemas, tables, columns, constraints, indexes: lower-case, underscores, no spaces or mixed case. Avoid double-quoted identifiers.
3. **Trailing commas.** In column lists, `SELECT` lists, and multi-line `INSERT` lists, commas trail the element line. This minimizes churn when adding/removing elements and improves blame granularity.
4. **Qualified columns in multi-table queries.** When a query *references more than one table or CTE*, fully qualify columns using a short, meaningful alias (e.g., `p.person_id` not `person_id`).
5. **No subqueries in `JOIN` clauses.** Prefer CTEs (`WITH` clauses) to make join inputs explicit and independently testable; this improves readability and planner visibility.
6. **Explicit aliasing.** Always use `AS` for column aliases; for table aliases, omit `AS` per PostgreSQL idiom or use consistently (we allow either, but be consistent within a file).
7. **Boolean-like flags are tri-state.** Store {0,1,NULL} with light `CHECK` constraints; never use textual booleans for flags.
8. **Stable formatting of DDL.** Place each column on its own line; align types; keep `CONSTRAINT` lines separated; prefer `CREATE TABLE IF NOT EXISTS`.

[66] SQLFluff documentation: https://docs.sqlfluff.com/.
[67] PostgreSQL documentation: https://www.postgresql.org/docs/.

Before/After Examples.—

```
-- Bad (mixed case, leading commas, unqualified):
select p.person_id , firstName, e.encounter_id
from patient p join encounter e on person_id = e.person_id;

-- Good (UPPERCASE, snake_case, trailing commas, qualified):
SELECT
  p.person_id,
  p.first_name,
  e.encounter_id
FROM :"schema".patient     AS p
JOIN :"schema".encounter   AS e
  ON p.person_id = e.person_id;
```

8.3. *CTEs over Subqueries in JOINs*

Subqueries embedded in `JOIN`s obfuscate logic, complicate reuse, and hinder targeted testing. Prefer CTEs:

```
-- Avoid:
SELECT a.person_id, b.cnt
FROM fact a
JOIN (SELECT person_id, COUNT(*) AS cnt FROM events GROUP BY person_id) b
  ON a.person_id = b.person_id;

-- Prefer:
WITH events_by_person AS (
  SELECT person_id, COUNT(*) AS cnt
  FROM events
  GROUP BY person_id
)
SELECT
  a.person_id,
  b.cnt
FROM fact AS a
JOIN events_by_person AS b
  ON a.person_id = b.person_id;
```

This pattern simplifies unit-style testing (`SELECT * FROM events_by_person LIMIT 10;`) and reduces join-nesting complexity.[68]

8.4. *Trailing Commas and Vertical Layout*

We require trailing commas for multi-line lists and place each item on its own line:

```
CREATE TABLE IF NOT EXISTS :"schema".patient (
  person_id         varchar PRIMARY KEY,
  first_name        varchar,
  last_name         varchar,
  birth_date        date,
  sex               varchar,
  -- ...
);
```

[68] PostgreSQL query planning benefits from clearer boundaries; see *The Query Planner* in the docs.

The same pattern applies to `SELECT` and `INSERT` lists. Trailing commas minimize diff noise and enable simple copy/paste line edits.

8.5. *Linting with SQLFluff (Strict Configuration)*

We configure `sqlfluff` to encode the tenets in §8.2. A representative excerpt:

```
# pyproject.toml
[tool.sqlfluff]
dialect = "postgres"
max_line_length = 100
exclude_rules = ["L009"]  # example: if we allow implicit alias AS for tables

[tool.sqlfluff.rules]
# Capitalisation
capitalisation_policy = "upper"

[tool.sqlfluff.rules.L010]  # Capitalisation of keywords
capitalisation_policy = "upper"

[tool.sqlfluff.rules.L014]  # Unquoted identifiers
extended_capitalisation_policy = "lower"

[tool.sqlfluff.rules.L016]  # Line length breaks (enforce vertical lists)
max_line_length = 100

[tool.sqlfluff.rules.L019]  # Commas
comma_style = "trailing"

[tool.sqlfluff.rules.L020]  # Join condition placement
# Ensure JOIN and ON lines are tidy and readable.

[tool.sqlfluff.rules.L021]  # Alias lengths / preferred aliasing
# Enforce short, readable table aliases.

[tool.sqlfluff.rules.L027]  # Consistent aliasing of columns
require_aliases = true

[tool.sqlfluff.rules.L028]  # Qualified references
force_enable = true  # ensure qualification in multi-table SELECTs

[tool.sqlfluff.rules.L031]  # Avoid table aliases in single-table queries
single_table_references = "consistent"

[tool.sqlfluff.rules.L063]  # Prefer CTEs to nested queries in JOIN (custom policy)
# Implemented via a custom plugin or review policy; see wrapper notes.
```

Notes.—Some preferences ("no subqueries in JOINs") are not a first-class rule in all `sqlfluff` versions. We enforce them by policy, reviews, and (optionally) a custom plugin; see below for the pre-commit wrapper that can grep/block specific patterns as a guardrail.

8.6. *Pre-commit Integration*

We use `pre-commit`[69] to run linters locally and in CI.

Goals.—

- Run `sqlfluff lint` before each commit.
- Provide a *psql-aware wrapper* that massages non-standard bits (`:''schema''` variables, `timestamp_ntz`) so linting is accurate but *does not* modify production SQL.
- Offer a manual "fix" hook for opt-in formatting (`sqlfluff fix`).

psql-aware Wrapper (for linting only).—Two recurring issues for linters are (a) psql variables (`:''schema''`, `:''terminology_schema''`), and (b) types not known to the linter (e.g., `timestamp_ntz`). The wrapper:

1. Reads a SQL file,
2. Replaces `:''schema''` and `:''terminology_schema''` with placeholder schema names (e.g., `lint_schema`),
3. Rewrites `timestamp_ntz` to `timestamp without time zone` (or an agreed surrogate),
4. Emits to a temporary file and runs `sqlfluff` against it,
5. Exits with the linter's status.

```
#!/usr/bin/env bash
# scripts/sqlfluff_psql_lint.sh
set -euo pipefail
tmp="$(mktemp).sql"
sed -e 's/:"schema"/lint_schema/g' \
    -e 's/:"terminology_schema"/lint_terminology/g' \
    -e 's/\btimestamp_ntz\b/timestamp without time zone/g' \
    "$1" > "$tmp"
sqlfluff lint "$tmp"
```

Manual "Fix" Hook.—For on-demand formatting, we expose a dedicated hook that runs `sqlfluff fix`. This is *not* required on every commit (to avoid churn), but developers can run it locally when cleaning up a series.

8.7. *Pre-commit Configuration (Repository)*

A representative `.pre-commit-config.yaml` excerpt:

```
repos:
  - repo: local
    hooks:
      - id: sqlfluff-psql-lint
        name: sqlfluff (psql-aware lint)
        entry: bash scripts/sqlfluff_psql_lint.sh
        language: system
        files: \.(sql|ddl|dml)$
      - id: sqlfluff-psql-fix
        name: sqlfluff (manual fix)
        entry: bash -lc 'sqlfluff fix --dialect postgres db/'
        language: system
        pass_filenames: false
        stages: [manual]
```

Why two hooks?—The *lint* hook runs automatically; the *fix* hook is opt-in (manual stage). This separation avoids unexpected, large diffs while enabling standardized formatting with a single command.

[69] Pre-commit framework: pre-commit: CI usage .

8.8. *Developer Workflow*

One-time setup.—

```
pre-commit install
```

Run on the whole tree (e.g., after a rebase).—

```
pre-commit run --all-files
```

On-demand formatting (opt-in).—

```
pre-commit run sqlfluff-psql-fix --all-files
```

Local policy checks.—Combine the wrapper with a simple grep-based guard (advisory) to flag subqueries in JOINs:

```
# scripts/check_no_join_subqueries.sh (advisory)
#!/usr/bin/env bash
set -euo pipefail
if grep -nE 'JOIN\s*\(' "$@" ; then
  echo "Advisory: subquery in JOIN detected. Prefer CTEs." >&2
  exit 1
fi
```

8.9. *File Organization and Conventions*

- **One file per object.** Each table, view, or test lives in its own file. DDL files create objects with `IF NOT EXISTS`, and constraints/indexes use guarded `ALTER` statements.
- **Module headers.** Start each file with a short comment block: purpose, grain, dependencies, and idempotency assumptions.
- **Transaction scope.** DDL files avoid wrapping in transactions; loaders/tests control `BEGIN`/`COMMIT` at orchestration level.
- **Consistent aliasing.** Use short, mnemonic aliases: `p` (`patient`), `e` (`encounter`), `m` (`medical_claim`).

8.10. *Common Pitfalls and How the Tooling Helps*

Unqualified columns.: *Pitfall:* Ambiguous columns in multi-table queries. *Mitigation:* `sqlfluff` rule for qualified references; pre-commit blocks the commit.

Style drift.: *Pitfall:* Mixed capitalization and comma styles across files. *Mitigation:* global rules for capitalization and trailing commas; optional `fix`.

Non-standard types.: *Pitfall:* `timestamp_ntz` not recognized by the linter. *Mitigation:* psql-aware wrapper remaps for linting only.

Opaque JOIN inputs.: *Pitfall:* Subqueries in `JOIN` clauses hide logic. *Mitigation:* CTE policy; advisory guard script and reviewer enforcement.

8.11. *Interplay with CI*

CI runs the same linters with the same configuration, ensuring local–CI parity. The *lint* job fails fast on violations, while the *build/load/test* job operates independently. Because linting uses the psql-aware wrapper, the SQL as committed remains untouched, and the linter still understands our files as valid PostgreSQL.[70]

[70] Using linters in CI with pre-commit: https://pre-commit.com/#usage-in-continuous-integration.

8.12. *Documentation and Rule Waivers*

Inline annotations.—Where a specific rule must be waived (e.g., a vendor DDL file with unconventional identifiers), annotate locally:

```
-- sqlfluff: disable=all
CREATE TABLE vendor."StrangeCase" ("ID" int, "Value" text);
-- sqlfluff: enable=all
```

Prefer narrow, temporary waivers with an explanatory comment. Persistent exceptions should be documented in a per-directory `.sqlfluff` or `pyproject.toml` override.

8.13. *Summary*

A disciplined SQL style backed by automated tooling is an investment that pays dividends in readability, review velocity, and correctness. UPPERCASE keywords, *snake_case* identifiers, trailing commas, qualified references, and CTE-first query design form the backbone of our convention. `sqlfluff` and `pre-commit` operationalize these rules, while a psql-aware wrapper preserves portability and prevents false positives. Together, they make the codebase easier to extend, audit, and trust.

9. SECURITY & GOVERNANCE

9.1. *Purpose and Scope*

This section formalizes the controls that keep a Tuva-aligned PostgreSQL load both *secure* and *governable*. We focus on three pillars: (1) **credential handling** that avoids secret sprawl, (2) **least-privilege database access** with schema isolation for tests, and (3) **supply-chain integrity** across CI runners, actions, and linters. Where appropriate, we reference widely accepted practices (e.g., the Twelve-Factor configuration principle[71], OWASP guidance[72]) and product documentation (PostgreSQL[73], GitHub Actions[74], SQLFluff[75]).

9.2. *Threat Model (Pragmatic)*

We assume: (a) the repository is public or organization-internal, (b) developers work on untrusted laptops or shared CI, (c) the CI database is ephemeral, and (d) terminology datasets may be large and sometimes licensed. Primary risks are credential disclosure (in repo, logs, artifacts), supply-chain tampering, and accidental access escalation (e.g., broad GRANTs in shared instances).

9.3. *Credential Handling*

.env hygiene (no secrets in git).—All connection settings live in a local `.env` file (not tracked). A sanitized template `.env.example` documents required keys. The repo-level `.gitignore` must include `.env` and any machine-local files.

```
# .gitignore (excerpt)
.env
.env.*
*.local
```

Process environment injection.—Tooling (Make, psql, Python) reads from the environment:

```
# .env (local only; never committed)
PGHOST=127.0.0.1
PGPORT=5432
PGUSER=tuva_dev
PGPASSWORD=***redacted***
PGDATABASE=tuva
schema=tuva_core
terminology_schema=tuva_terminology
```

[71] Twelve-Factor App, Config: https://12factor.net/config.
[72] OWASP Secrets Management Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html.
[73] PostgreSQL Documentation: https://www.postgresql.org/docs/.
[74] GitHub Actions Docs: https://docs.github.com/actions.
[75] SQLFluff Docs: https://docs.sqlfluff.com/.

Developers load this into their shell (`set -a; source .env; set +a`). The loader and tests avoid reading files with secrets directly; instead, they rely on `PG*` variables.[76]

CI secrets.—In CI, prefer *ephemeral* database containers with simple credentials. For non-ephemeral targets, store secrets in the platform's encrypted store (e.g., GitHub Actions *repository* or *environment* secrets) and *never* echo them to logs. Masked values should not appear in `set -x` traces.

Secret scanning and pre-commit.—Enable platform secret-scanning and add a local hook (e.g., *detect-secrets* or *gitleaks*) to block commits that contain credential patterns.[77] Treat a red scan as a merge blocker.

9.4. *Least-Privilege Database Roles*

We separate *ownership*, *write*, and *read* concerns, and we isolate CI tests by schema.

Role design.—

- **Owner roles (no login):** own objects, perform DDL; not used by humans.
- **RW roles (login):** DML on core schemas, no DDL.
- **RO roles (login):** SELECT-only on core schemas.
- **CI roles:** limited to CI schemas with destroy-on-finish lifecycle.

Reference DDL (apply per schema).—

```
-- One-time: create schemas and revoke defaults
CREATE SCHEMA IF NOT EXISTS :"schema";
CREATE SCHEMA IF NOT EXISTS :"terminology_schema";

REVOKE ALL ON SCHEMA :"schema" FROM PUBLIC;
REVOKE ALL ON SCHEMA :"terminology_schema" FROM PUBLIC;

-- Roles
CREATE ROLE tuva_owner NOLOGIN;
CREATE ROLE tuva_rw LOGIN PASSWORD '***strong***';
CREATE ROLE tuva_ro LOGIN PASSWORD '***strong***';

-- Schema usage
GRANT USAGE ON SCHEMA :"schema" TO tuva_ro, tuva_rw;
GRANT USAGE ON SCHEMA :"terminology_schema" TO tuva_ro, tuva_rw;

-- Object privileges
GRANT SELECT ON ALL TABLES IN SCHEMA :"schema" TO tuva_ro;
GRANT SELECT ON ALL TABLES IN SCHEMA :"terminology_schema" TO tuva_ro;

GRANT SELECT, INSERT, UPDATE, DELETE
  ON ALL TABLES IN SCHEMA :"schema" TO tuva_rw;

-- Default privileges for future objects
ALTER DEFAULT PRIVILEGES IN SCHEMA :"schema"
  GRANT SELECT ON TABLES TO tuva_ro;
ALTER DEFAULT PRIVILEGES IN SCHEMA :"schema"
  GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO tuva_rw;
```

[76] psql environment variables: https://www.postgresql.org/docs/current/libpq-envars.html.
[77] GitHub secret scanning: https://docs.github.com/code-security/secret-scanning.

```
-- Optional: prohibit DDL for RW/RO
REVOKE CREATE ON SCHEMA :"schema" FROM tuva_ro, tuva_rw;

-- CI user bound to CI schemas only
CREATE ROLE tuva_ci LOGIN PASSWORD '***ephemeral***';
GRANT USAGE ON SCHEMA ci_tuva, ci_terminology TO tuva_ci;
GRANT ALL    ON ALL TABLES IN SCHEMA ci_tuva, ci_terminology TO tuva_ci;
```

This pattern separates DDL (owner) from DML (rw) and protects read-only users from accidental writes.[78] In shared instances, enforce a per-run `search_path` and avoid granting global privileges.

Schema isolation for tests.—Each CI run targets unique schemas (e.g., `ci_tuva_$GITHUB_RUN_ID`). Jobs create, use, and drop those schemas. This eliminates cross-run contamination and minimizes privileges in multi-tenant databases.

9.5. *Logging, Artifacts, and Data Minimization*

Safe logging.—Do not print connection strings or secrets. Avoid `psql -e` in CI. Redact values in examples. Prefer `\copy` progress lines without row-level echoes.

Artifacts.—Export only non-sensitive artifacts (e.g., `test_results.csv`). Treat it as public-by-default: no PHI/PII, only counts and test names. If a future artifact must contain identifiers, store it in restricted environment artifacts with short retention.

Retention.—Set artifact retention windows appropriate to the project (e.g., 7–14 days). In GitHub, use per-workflow *retention-days*. Do not persist database volumes beyond the job.

9.6. *Supply-Chain Integrity*

Pin everything.—

- **Actions:** pin to commit SHAs instead of floating tags.[79]
- **Containers:** pin images by digest (`postgres:16@sha256:...`) rather than `:latest`.
- **Python tools:** pin `sqlfluff` and other dev tools in `pyproject.toml` with an exact version; consider *constraints* files with hashes.
- **Pre-commit hooks:** pin `rev` to immutable versions.

Example (workflow excerpt).—

```
services:
  postgres:
    image: postgres:16@sha256:<digest>
    # ...

steps:
  - uses: actions/checkout@<commit-sha>
  - uses: actions/setup-python@<commit-sha>
    with: { python-version: '3.11' }
```

Automated updates with review.—Use Dependabot or Renovate to propose bumps for pinned SHAs and tool versions; require PR review and CI green before merge. This balances freshness with integrity.

78 GRANT/REVOKE reference: https://www.postgresql.org/docs/current/sql-grant.html.
79 Security hardening for Actions: https://docs.github.com/actions/security-guides/security-hardening-for-github-actions#using-third-party-actions.

Reproducible builds.—CI runs the same Make targets as local; avoid environment-dependent logic. Where a large terminology dataset is pulled from cloud storage, record the source URI and checksum; prefer HTTPS and, when available, signed releases.

9.7. *Governance: Data Classification and Access*

Classification.—Adopt a simple rubric: *Public*, *Internal*, *Restricted*. Seed datasets should be de-identified; treat any accidental PHI/PII as *Restricted*.

Access policy.—

- **Developers:** RO to production-like datasets; RW only in dev namespaces.
- **CI:** RW to ephemeral schemas only; no access to production schemas.
- **Analysts/QA:** RO on curated outputs and terminology.

Access reviews quarterly: enumerate roles, grants, and last login. Remove dormant accounts. Enforce MFA on GitHub and any database bastions.

9.8. *Auditing and Change Control*

DDL ownership and review.—All DDL is committed via PRs. Require review by at least one maintainer. Tag changes with Conventional Commit types (e.g., `feat`, `fix`, `chore`) to provide traceability.

DB auditing (optional).—On shared or long-lived instances, enable:

- **DDL logging:** `log_statement = 'ddl'`.
- **Login auditing:** `log_connections = on`, `log_disconnections = on`.
- **pg_audit** (if available) for granular audit trails.[80]

9.9. *Incident Response and Recovery*

Revocation.—Be ready to rotate credentials rapidly: database roles with expirations or easy password resets; CI secrets replaced and invalidated. Ensure revoking `tuva_rw` does not break read-only operations.

Backups (if stateful).—CI containers are ephemeral; no backups. For shared dev/test instances, schedule periodic logical backups (e.g., `pg_dump`) for schemas that matter, and test restores.

9.10. *Policy Checklist (Copy/Paste)*

```
[ ] .env in .gitignore; .env.example provided; no secrets in repo
[ ] Secret scanning enabled (platform and pre-commit)
[ ] Roles: owner (no login), rw (DML), ro (SELECT); CI role isolated
[ ] Revoke PUBLIC on schemas; grant least privileges; set default privileges
[ ] CI schemas per run; drop on completion
[ ] Actions and images pinned (commit SHAs, digests); sqlfluff pinned
[ ] Artifacts contain no PHI/PII; retention set; logs avoid secrets
[ ] Branch protection requires lint + tests; MFA enforced on VCS
[ ] Periodic access reviews; remove dormant accounts
```

9.11. *Summary*

Security and governance for this project emphasize *prevention by design* (no secrets in git, least-privilege roles, schema isolation) and *integrity of the automation* (pinned actions, pinned tools). CI produces a non-sensitive, standardized `test_results.csv` for visibility, while role and schema boundaries keep the blast radius small. These controls are intentionally lightweight, reproducible, and aligned with community guidance, making them easy to adopt and hard to bypass.

[80] pgaudit project: https://www.pgaudit.org/.

10. PERFORMANCE & SCALABILITY

10.1. *Objectives*

This section provides practical guidance to size, tune, and evolve the Tuva-aligned PostgreSQL deployment for bulk loads and analytical reads. We focus on (1) realistic volume expectations, (2) ingest throughput with `\copy`, (3) index and maintenance strategy (VACUUM/ANALYZE), and (4) optional enhancements such as parallel ingest, partitioning, and UNLOGGED staging. All recommendations target PostgreSQL 16 and rely on standard features.[81]

10.2. *Workload Model & Expected Volumes*

Data volumes vary widely by organization and observation window. Table 1 lists order-of-magnitude ranges to guide storage and time budgeting; adjust per program and retention policy.

Table	**Rows (12–36 mo)**	**Row Width (bytes)**	**Notes**
patient	10^5–10^7	200–400	One row per person.
encounter	10^6–10^8	200–500	Inpatient/ED/OP visits.
person_id_crosswalk	10^5–10^7	100–200	Member/patient map.
medical_claim	10^7–10^9	150–350	Claim *lines* dominate volume.
pharmacy_claim	10^6–10^8	120–250	Dispense lines.
eligibility	10^6–10^8	120–220	Enrollment spans (often multiple per member).
procedure	10^6–10^8	120–250	Normalized code mix.
observation	10^7–10^9	100–220	Clinical observations; skewed by interfaces.
lab_result	10^7–10^9	150–300	Orders/components; can spike.
condition	10^6–10^8	150–300	Diagnoses/problem lists.
medication	10^6–10^8	150–320	Source + RxNorm/ATC joins.
immunization	10^5–10^7	120–220	CVX-based.
appointment	10^6–10^8	150–300	Schedulers can be chatty.
location, practitioner	10^3–10^6	100–200	Dimensions; relatively small.
terminology (per table)	10^2–10^7	varies	Some are large (ICD/LOINC/SNOMED).

Table 1. Placeholder volume ranges for planning. Calibrate to your cohort size and retention.

Storage planning.—A conservative footprint estimate uses:

$$\text{table size} \approx \text{rows} \times \text{avg row width} \times (1.2 \text{ to } 1.5)$$

to account for tuple headers, alignment, and visibility metadata. Indexes often add 20–100% depending on key widths and number of indexes.

10.3. *Ingest Throughput with `\copy`*

Client-side `\copy` (the `psql` meta-command) streams CSV from the runner into the server without requiring superuser access.[82] On modern SSD-backed runners with default WAL settings, single-stream `\copy` typically sustains:

- **Narrow rows (8–12 cols, ~100 B/row)**: *100k–500k rows/s.*
- **Wide rows (30+ cols, ~300 B/row)**: *50k–150k rows/s.*

These are order-of-magnitude guides; network and WAL bandwidth dominate. Heavier indexes and triggers reduce throughput. To avoid surprises:

1. **Load into empty tables** (no secondary indexes yet).

[81] PostgreSQL Documentation: https://www.postgresql.org/docs/.
[82] `COPY` and `\copy`: https://www.postgresql.org/docs/current/sql-copy.html.

2. **Batch large files** (e.g., 250–500 MB chunks) for progress and retry.
3. **Disable synchronous commit *only* in ephemeral CI**:

```
SET LOCAL synchronous_commit = off;  -- CI-only, risk of data loss on crash
```

4. **Avoid FREEZE unless you know the implications**. `COPY ... FREEZE` can reduce vacuum but is appropriate only in controlled scenarios.[83]

Header-locked mapping.—Always specify the column list in `\copy` to decouple CSV order from table evolution:

```
\copy :"schema".medical_claim(
  medical_claim_id, claim_id, claim_line_number, ..., tuva_last_run
) FROM 'data/medical_claim.csv'
  WITH (FORMAT csv, HEADER true, NULL '', QUOTE '"', ESCAPE '"');
```

10.4. *Index Strategy*

Indexes are essential for read performance and for many test joins, but they slow bulk ingest if present during load. The recommended flow is:

1. Load into base tables *without* secondary indexes.
2. Build indexes *after* bulk load.
3. Run `ANALYZE` to refresh statistics.

Build timing.—

- **Non-concurrent** builds are fastest but briefly lock writes; ideal for CI and batch loads into exclusive schemas.
- **CONCURRENTLY** allows reads/writes during build but is slower and cannot run inside a transaction block.[84]

What to index.—Start with joins and common filters:

- Foreign keys / soft-keys: `person_id`, `encounter_id`, `claim_id+claim_line_number`.
- Code lookups: `normalized_code`, `hcpcs_code`, `ndc_code`.
- Time filters: `claim_start_date`, `dispensing_date`, `result_datetime`.
- Compound indexes for frequent predicates (e.g., `member_id, payer, plan`).

Build memory.—Increase session `maintenance_work_mem` for faster index builds (avoid starving the host):

```
SET LOCAL maintenance_work_mem = '1GB';  -- tune to runner capacity
```

10.5. *VACUUM & ANALYZE*

Bulk loads create many new pages; the planner needs statistics and visibility cleanup.

ANALYZE.—Run `ANALYZE` (or rely on autovacuum) after each bulk load so joins and filters choose appropriate plans.[85]

```
ANALYZE :"schema".medical_claim;
ANALYZE :"schema".observation;
-- ... etc.
```

VACUUM.—For append-only loads into empty tables, `VACUUM` need is minimal; normal autovacuum is sufficient. If you perform large `DELETE/TRUNCATE/INSERT` cycles (e.g., reloads), a `VACUUM (ANALYZE)` after the load can be beneficial.

[83] FREEZE option caveats: see *COPY* docs.
[84] `CREATE INDEX CONCURRENTLY`: https://www.postgresql.org/docs/current/sql-createindex.html.
[85] `ANALYZE` Reference: https://www.postgresql.org/docs/current/sql-analyze.html.

Autovacuum knobs (optional).—On busy shared instances with large tables, consider per-table storage parameters to raise the autovacuum threshold (to avoid premature runs) or increase `autovacuum_vacuum_scale_factor` during bulk ingest windows. Use judiciously; defaults are safe.

10.6. *Parallel Ingest (Advanced)*

Multiple streams.—PostgreSQL applies *one* `\copy` per session. To parallelize:

1. **By table:** run separate `\copy` processes for different tables in parallel.
2. **By partition/shard:** split a large table by date or hash (see partitioning below) and load each child in parallel.
3. **By chunk:** split the CSV into N parts and run N parallel `\copy` into a *heap without secondary indexes*, then build indexes after.

Parallelizing *a single indexed heap* can degrade due to WAL and B-Tree contention; favor partitioned parallelism or defer index creation.

WAL considerations.—Heavier parallel ingest increases WAL volume and checkpoint pressure. If permissible, enable WAL compression at the server level to reduce I/O at the cost of CPU.[86]

10.7. *Partitioning (Date- or Key-based)*

Native partitioning improves maintenance and some query patterns by reducing index sizes and enabling pruning.[87]

When to partition.—

- **Very large facts** ($\geq 10^8$ rows) with time predicates: `medical_claim`, `lab_result`, `observation`.
- **Operational maintenance** needs: truncate old months, run maintenance on hot partitions without scanning the world.

Strategy.—

- **Range partitioning by month**: e.g., `claim_start_date` or `result_datetime::date`.
- **Hash partitioning by person_id**: balances skew when time is not selective.

Skeleton.—

```
CREATE TABLE :"schema".lab_result (
  -- columns...
  result_datetime timestamp without time zone
) PARTITION BY RANGE (result_datetime::date);

CREATE TABLE :"schema".lab_result_2025_08
  PARTITION OF :"schema".lab_result
  FOR VALUES FROM ('2025-08-01') TO ('2025-09-01');

-- Index per partition (smaller/faster)
CREATE INDEX ON :"schema".lab_result_2025_08 (person_id);
CREATE INDEX ON :"schema".lab_result_2025_08 (result_datetime);
```

Benefits.—Partition pruning reduces I/O for time-bounded queries/tests; index builds are faster per child; archival is a simple `DETACH`/`DROP`. Costs include more DDL, more indexes to manage, and care when writing queries (ensure partition keys appear plainly in predicates).

[86] WAL compression: https://www.postgresql.org/docs/current/runtime-config-wal.html.
[87] Table Partitioning: https://www.postgresql.org/docs/current/ddl-partitioning.html.

10.8. *UNLOGGED Staging (Selective)*

What it is.—`UNLOGGED` tables skip WAL, speeding writes but losing crash safety and replication.[88]

When useful.—

- **CI and transient ETL staging** where data can be reconstituted.
- **Pre-transform landing** before merging into logged base tables.

Pattern.—

```
CREATE UNLOGGED TABLE :"schema"._stg_medical_claim (LIKE :"schema".medical_claim);

\copy :"schema"._stg_medical_claim (...) FROM 'data/medical_claim.csv' WITH (...);

-- Transform/dedup, then move:
INSERT INTO :"schema".medical_claim (...)
SELECT ... FROM :"schema"._stg_medical_claim;

TRUNCATE :"schema"._stg_medical_claim;
```

This shifts some cost away from WAL during landing. The final `INSERT` into the logged table still produces WAL (as it must), but the approach can reduce contention during parsing/validation or when multiple stages feed the final write.

10.9. *Query Performance Tips*

- **Qualified joins** on indexed keys (`person_id`, `encounter_id`); avoid functions on join columns (precompute if needed).
- **Statistics freshness** via post-load `ANALYZE`. Consider increased `default_statistics_target` on columns with skewed distributions (e.g., code columns).
- **Covering indexes** for frequent filters + sorts (e.g., `(person_id, result_datetime)` for last-result lookups).
- **Avoid wildcards** on very wide tables in ad-hoc queries; project only the needed columns to reduce I/O.

10.10. *Throughput Playbook (Copy/Paste)*

1. Ensure tables are empty and *secondary indexes are absent*.
2. `\copy` in chunked batches; log per-batch rows/s and elapsed time.
3. Build required indexes (optionally bump `maintenance_work_mem`).
4. `ANALYZE` all loaded tables.
5. (Optional) Partition very large facts *before* loading, then parallel-load child tables.
6. (Optional) Use UNLOGGED staging for CI and re-loadable lanes.

10.11. *Risk & Trade-offs*

- **Parallelism vs. contention:** Multiple writers to the same indexed heap can slow down due to WAL and B-Tree page splits; partition-first to parallelize safely.
- **UNLOGGED volatility:** Crashes truncate UNLOGGED tables; never use for durable data.
- **Aggressive knobs:** Disabling `synchronous_commit` or using `COPY FREEZE` without careful scoping can threaten durability or visibility semantics.

[88] UNLOGGED tables: https://www.postgresql.org/docs/current/sql-createtable.html.

10.12. *Summary*

Start simple: single-stream `\copy` into empty tables, build indexes after, and `ANALYZE`. For very large facts, introduce monthly partitions and parallelize by child table. Use UNLOGGED staging and CI-only durability relaxations where data is disposable. These steps typically deliver predictable ingest at six to seven digits of rows per second (aggregate across tables) and responsive analytical joins on well-chosen indexes, while keeping operational complexity modest.

11. OBSERVABILITY

11.1. *Purpose*

Observability in this repository means: (1) making the pipeline's *state* and *outcomes* visible to engineers, analysts, and QA; (2) preserving a durable record of quality signals; and (3) providing fast paths to diagnose regressions. We operationalize this with SQL-first telemetry (tables and views inside PostgreSQL), CI summaries, and an optional BI layer that reads from exported artifacts or directly from the database.[89][90]

11.2. *Signals & Questions We Must Answer*

We focus on a small set of high-signal, low-noise metrics:

1. **Test outcomes**: Which tests failed? How many, by suite and severity? What changed since the last green run?
2. **Time-to-ingest**: How long did `create-db`, `load`, and `test` take? Are we meeting expectations as data grows?
3. **Volumes**: Row counts per table (and optionally table sizes). Did we load the expected number of rows this run? Are there unexpected spikes or drops?

These three pillars—quality, latency, and volume—cover 95% of day-to-day observability needs, while remaining simple to compute and export.

11.3. *Data Model for Observability*

We store observability facts alongside the warehouse under the same parameterized schema conventions (`:"schema"`). The telemetry is narrow, append-friendly, and easy to export.

1) Test results (canonical surface).—The test harness already emits a normalized table:

```
CREATE TABLE IF NOT EXISTS :"schema".test_results (
  run_id     text DEFAULT current_setting('application_name', true),
  test       text,
  suite      text,                        -- e.g., 'patient_smoke', 'observation_addons'
  severity   text DEFAULT 'warn',         -- e.g., 'error'|'warn'|'info'
  pass       boolean,
  details    bigint,                      -- count or diagnostic numeric
  duration_ms integer,                    -- optional: per-test runtime
  notes      text,                        -- optional: short reason or label
  run_ts     timestamp without time zone DEFAULT now()
);
```

Why a `run_id`? It groups rows that belong to the same CI execution and enables comparisons across runs. We default it from `application_name` so CI can set it without schema changes.

2) Run-level metrics (latency & status).—

```
CREATE TABLE IF NOT EXISTS :"schema".run_metrics (
  run_id     text PRIMARY KEY,
  started_at timestamp without time zone,
```

[89] PostgreSQL Documentation: https://www.postgresql.org/docs/.
[90] GitHub Actions: https://docs.github.com/actions.

```
  ended_at   timestamp without time zone,
  step_create_db_ms integer,
  step_load_ms      integer,
  step_test_ms      integer,
  overall_ms        integer,
  status    text CHECK (status IN ('success','failure')),
  run_ts    timestamp without time zone DEFAULT now()
);
```

CI populates this table with one row per run (see §11.4).

3) Row-count snapshots (volumes).—

```
CREATE TABLE IF NOT EXISTS :"schema".table_row_counts (
  run_id     text,
  schema_name text,
  table_name  text,
  row_count   bigint,
  rel_kb      bigint,                   -- optional: relation size in KB
  idx_kb      bigint,                   -- optional: index size in KB
  run_ts      timestamp without time zone DEFAULT now(),
  PRIMARY KEY (run_id, schema_name, table_name)
);
```

A single SQL query can populate this for all user tables each run.

11.4. *Collection & Ingestion in CI*

The GitHub Actions job already runs `make create-db`, `make load`, and `make test`. We extend it minimally to capture timing, set a `run_id`, and export artifacts.

Set a stable `run_id`.—Use the CI-provided run number (or SHA) and inject it as `application_name` so the test harness inserts it automatically:

```
export RUN_ID="gha-${GITHUB_RUN_NUMBER}"
psql -v ON_ERROR_STOP=1 -c "SET application_name = '${RUN_ID}';"
```

Measure step durations.—Wrap each step with timestamps and insert one run-level record:

```
start_run=$(date +%s%3N)

t0=$(date +%s%3N); make create-db; t1=$(date +%s%3N)
t_create=$((t1 - t0))

t0=$(date +%s%3N); make load;      t1=$(date +%s%3N)
t_load=$((t1 - t0))

t0=$(date +%s%3N); make test;      t1=$(date +%s%3N)
t_test=$((t1 - t0))

end_run=$(date +%s%3N)
total=$((end_run - start_run))

psql -v ON_ERROR_STOP=1 <<SQL
INSERT INTO :"schema".run_metrics(run_id, started_at, ended_at,
  step_create_db_ms, step_load_ms, step_test_ms, overall_ms, status)
VALUES (
```

```
  '${RUN_ID}',
  NOW() - INTERVAL '${total} milliseconds',
  NOW(),
  ${t_create}, ${t_load}, ${t_test}, ${total},
  CASE WHEN (SELECT COUNT(*) FROM :"schema".test_results
             WHERE run_id='${RUN_ID}' AND pass = false) > 0
       THEN 'failure' ELSE 'success' END
);
SQL
```

Capture row counts and sizes.—

```
psql -v ON_ERROR_STOP=1 <<'SQL'
WITH rel AS (
  SELECT
    n.nspname AS schema_name,
    c.relname AS table_name,
    pg_table_size(c.oid)  / 1024 AS rel_kb,
    pg_indexes_size(c.oid)/ 1024 AS idx_kb
  FROM pg_class c
  JOIN pg_namespace n ON n.oid = c.relnamespace
  WHERE c.relkind = 'r' AND n.nspname IN (current_schema())
)
INSERT INTO :"schema".table_row_counts(run_id, schema_name, table_name, row_count, rel_kb, idx_kb)
SELECT
  current_setting('application_name', true) AS run_id,
  schemaname, relname, n_live_tup::bigint, rel.rel_kb, rel.idx_kb
FROM pg_stat_user_tables t
JOIN rel ON rel.schema_name = t.schemaname AND rel.table_name = t.relname;
SQL
```

This uses `pg_stat_user_tables` and size helpers to produce one row per table.[91]

Export artifacts for BI use.—

```
psql -At -c "\copy :\"schema\".test_results      TO 'test_results.csv'       CSV HEADER"
psql -At -c "\copy :\"schema\".run_metrics       TO 'run_metrics.csv'        CSV HEADER"
psql -At -c "\copy :\"schema\".table_row_counts  TO 'table_row_counts.csv'   CSV HEADER"
```

Actions can upload these as build artifacts for inspection and dashboard ingestion.[92]

11.5. *Dashboards and Rendered Summaries*

In CI logs (human-first).—Keep a short, high-signal summary in job output:

```
-- Top-line run summary
SELECT run_id, overall_ms, step_create_db_ms, step_load_ms, step_test_ms, status
FROM :"schema".run_metrics
ORDER BY run_ts DESC LIMIT 1;

-- Failing tests (sorted)
SELECT suite, test, severity, details
FROM :"schema".test_results
```

[91] Catalogs and stats views: https://www.postgresql.org/docs/current/monitoring-stats.html.
[92] Uploading artifacts: https://docs.github.com/actions/using-workflows/storing-workflow-data-as-artifacts.

```
WHERE pass = false AND run_id = :'RUN_ID'
ORDER BY severity DESC, suite, test;

-- Failures by suite
SELECT suite, COUNT(*) AS failed
FROM :"schema".test_results
WHERE pass = false AND run_id = :'RUN_ID'
GROUP BY suite ORDER BY failed DESC;
```

Printing only failures keeps logs readable and actionable.

GitHub job summary (rich Markdown).—Optionally, write a Markdown table into the job summary (supported natively by Actions). Include the same three blocks: run summary, failures-by-suite, and the first N failing tests. This gives reviewers a one-screen snapshot without opening artifacts.[93]

Business Intelligence (BI).—Any BI tool can read the exported CSVs (or connect directly to Postgres) and present:

- **Run timeline**: stacked bar of `create-db`, `load`, `test` durations per run.
- **Failures by suite**: clustered columns with drill-down to failing tests.
- **Row counts**: per-table lines over time with thresholds (*expected ranges*) to detect under-/over-load.

Because the tables are normalized, a single *"Observability"* dataset covers all three.

11.6. *Metric Definitions and Queries*

Failures by suite (primary KPI).—

```
SELECT
  run_id,
  suite,
  COUNT(*) FILTER (WHERE pass = false) AS failed,
  COUNT(*) FILTER (WHERE pass = true)  AS passed
FROM :"schema".test_results
WHERE run_id = :'RUN_ID'
GROUP BY run_id, suite
ORDER BY failed DESC;
```

Time-to-ingest (SLO candidate).—We define a soft SLO (e.g., p95 of `overall_ms` under X minutes) and chart it:

```
SELECT date_trunc('day', run_ts) AS day,
       percentile_cont(0.95) WITHIN GROUP (ORDER BY overall_ms) AS p95_overall_ms
FROM :"schema".run_metrics
GROUP BY 1 ORDER BY 1;
```

Row counts per table (volume guard).—Compare the most recent run to a reference (e.g., previous successful run):

```
WITH last_two AS (
  SELECT run_id
  FROM :"schema".run_metrics
  WHERE status='success'
  ORDER BY run_ts DESC
  LIMIT 2
),
curr AS (
```

[93] GitHub job summaries: https://docs.github.com/actions/using-workflows/workflow-commands-for-github-actions.

```sql
  SELECT c.* FROM :"schema".table_row_counts c
  WHERE c.run_id = (SELECT run_id FROM last_two ORDER BY run_id DESC LIMIT 1)
),
prev AS (
  SELECT p.* FROM :"schema".table_row_counts p
  WHERE p.run_id = (SELECT run_id FROM last_two ORDER BY run_id DESC OFFSET 1 LIMIT 1)
)
SELECT curr.schema_name, curr.table_name,
       curr.row_count AS curr_rows,
       prev.row_count AS prev_rows,
       (curr.row_count - prev.row_count) AS delta
FROM curr
LEFT JOIN prev USING (schema_name, table_name)
ORDER BY ABS(curr.row_count - COALESCE(prev.row_count,0)) DESC;
```

Flag large deltas for investigation.

11.7. *Views for Troubleshooting*

Define stable, indexable views so engineers can pivot quickly without re-writing queries:

Per-suite failure counts (last run).—

```sql
CREATE OR REPLACE VIEW :"schema".v_failures_by_suite_last_run AS
SELECT t.run_id, t.suite, COUNT(*) AS failed
FROM :"schema".test_results t
JOIN LATERAL (
  SELECT run_id FROM :"schema".run_metrics ORDER BY run_ts DESC LIMIT 1
) r ON r.run_id = t.run_id
WHERE t.pass = false
GROUP BY t.run_id, t.suite;
```

Run timeline.—

```sql
CREATE OR REPLACE VIEW :"schema".v_run_timeline AS
SELECT run_id, started_at, ended_at,
       step_create_db_ms, step_load_ms, step_test_ms, overall_ms, status
FROM :"schema".run_metrics
ORDER BY run_ts DESC;
```

Row-count heatmap (recent N).—

```sql
CREATE OR REPLACE VIEW :"schema".v_row_counts_recent AS
SELECT *
FROM :"schema".table_row_counts
WHERE run_ts >= now() - interval '14 days';
```

11.8. *Alerting and Thresholds (Lightweight)*

We keep alerting *simple* and CI-native:

- **Red CI job** if any `severity='error'` test fails or if `status='failure'` in `run_metrics`.
- **Advisory warnings** printed when:
  - p95 `overall_ms` over the last week increases by $> 25\%$.
  - Row counts for key facts deviate $> 20\%$ from trailing median.

These checks can be implemented as SQL that returns non-zero rows; CI prints them and optionally fails on breach.

11.9. *Extensibility: More Signals if Needed*

- **Planner introspection**: enable `pg_stat_statements` on dev/test to profile heavy queries; export top-N offenders.[94]
- **Index health**: periodically track bloat and scan types (sequential vs. index) for very large tables.
- **Data drift probes**: add small/fast distribution checks (e.g., distinct code counts by day) and trend them in BI.

Keep these optional and off by default to minimize overhead.

11.10. *Governance and Retention*

Observability tables contain no PHI/PII—only metrics, names, and counts—so we can retain them longer (e.g., 6–12 months) to spot trends. Artifacts in CI should use finite retention (e.g., 7–14 days) and remain non-sensitive: CSVs with metrics, never raw data.

11.11. *Operating the Loop*

The value of observability comes from *use*, not storage. We institutionalize a short loop:

1. Every PR shows a job summary: run timing + failures-by-suite + top failing tests.
2. Daily (or on demand) BI dashboards show trends for latency and volume.
3. On regressions, engineers jump to troubleshooting views (*last run* failure counts, row deltas).
4. Issues are ticketed directly from failing suite names; test identifiers double as search keys.

Because the whole stack is SQL-native and artifact-friendly, it stays portable across environments.

11.12. *Summary*

We implement observability *inside* Postgres with three small tables: `test_results`, `run_metrics`, and `table_row_counts`. CI injects a `run_id`, records durations, exports CSV artifacts, and prints a concise failure summary. BI (optional) reads the same normalized facts to trend failures by suite, time-to-ingest, and per-table volumes. The approach is intentionally minimal, fast to run, and easy to maintain, yet it yields the visibility required to keep quality and performance on track as data scales.

12. DEVELOPER ONBOARDING

12.1. *Audience & Goals*

This guide gets a new contributor from clone to a fully validated local run in under an hour. It is opinionated, reproducible, and mirrors the CI pipeline so that green locally implies green in GitHub Actions. The focus is on a one-page *first day* checklist, common pitfalls (CSV headers, encodings, date formats), and an FAQ of practical recipes. Where helpful, we cite the upstream documentation for PostgreSQL,[95] the `psql` client,[96] GNU Make,[97] Python,[98] sqlfluff,[99] and GitHub Actions.[100]

12.2. *First-Day Checklist (One Page)*

1) Install prerequisites.—

- **PostgreSQL 16** (server or Docker) and **psql** client.
- **GNU Make** (often bundled on Linux/macOS; on Windows use MSYS2 or WSL).
- **Python 3.x** with `pip`; optional: `virtualenv`.
- **sqlfluff** and **pre-commit** (for style/tooling).

[94] `pg_stat_statements`: https://www.postgresql.org/docs/current/pgstatstatements.html.
[95] PostgreSQL Documentation: https://www.postgresql.org/docs/.
[96] psql reference: https://www.postgresql.org/docs/current/app-psql.html.
[97] GNU Make: https://www.gnu.org/software/make/manual/make.html.
[98] Python: https://docs.python.org/3/.
[99] SQLFluff: https://docs.sqlfluff.com/.
[100] GitHub Actions: https://docs.github.com/actions.

2) Clone & bootstrap.—

```
git clone https://github.com/your-org/tuva-postgres.git
cd tuva-postgres
make init
```

3) Configure environment.—

```
cp scripts/setup_env.example .env
# edit .env: PGHOST, PGPORT, PGUSER, PGPASSWORD, PGDATABASE, schema, terminology_schema
# load into shell:
set -a; source .env; set +a
```

We never commit secrets; `.env` is ignored by git.

4) Create objects, load data, run tests.—

```
make create-db     # schemas, tables, constraints, views
python scripts/normalize_csvs.py data   # optional cleanup
make load          # \copy CSVs into Postgres
make test          # smoke + add-ons, aggregates to :schema.test_results
```

5) Verify success.—

```
psql -c "SELECT COUNT(*) FROM :\"schema\".test_results WHERE pass = false;"
psql -c "TABLE :\"schema\".test_results ORDER BY pass, suite, test LIMIT 50;"
```

Zero failing rows indicates a clean run. If failures exist, see §12.5.

12.3. *Repository Tour (Conceptual)*

- `db/tables/`: one file per table (core & terminology), using `CREATE IF NOT EXISTS` and light `CHECK`s.
- `db/tests/`: SQL suites that emit rows into a standardized `:schema.test_results`.
- `scripts/`: *normalize*, loader helpers, and wrappers (e.g., psql-aware sqlfluff).
- `Makefile`: canonical targets (`create-db`, `load`, `test`, plus utility targets).
- `.pre-commit-config.yaml`, `pyproject.toml`: linting & style configuration.

12.4. *Common Pitfalls (and How to Avoid Them)*

CSV headers and column order.—The loader uses explicit column lists in `\copy`. If a CSV header drifts (missing/renamed columns), `\copy` will fail or mis-map.

- Ensure CSV headers exactly match table columns or update the `\copy` column list.
- Use `scripts/normalize_csvs.py` to trim/rename headers before load.

Encodings.—Use UTF-8 without BOM.[101]

- **Symptom:** `ERROR: invalid byte sequence for encoding "UTF8"` during load.
- **Fix:** Re-encode the file (`iconv -f WINDOWS-1252 -t UTF-8`), or cleanse source.

Line endings.—Prefer LF (`\n`). Mixed CRLF/LF can cause row-count mismatches.

- **Fix:** `dos2unix data/*.csv` before load.

[101] On Windows editors, explicitly save as UTF-8 (no BOM).

Date formats.—The schema expects ISO `YYYY-MM-DD` for `date` and ISO timestamps for `timestamp without time zone`.

- **Symptom:** `date/time field value out of range`.
- **Fix:** Normalize upstream; if unavoidable, stage to text then transform via `to_date()` in an interstitial step.

NULL semantics.—CSV empty fields are treated as `NULL` by `\copy` with `NULL ''`. A literal `NULL` string is *not* NULL unless the option is changed.

- Align on the project's `\copy` options; don't change per-file unless you update the loader.

Numeric punctuation.—Thousands separators or locale decimal commas cause parse errors.

- **Fix:** Upstream cleanse to plain digits with `.` decimal.

Quoting/escaping.—The loader uses `QUOTE '"', ESCAPE '"'`; embedded double-quotes must be doubled in CSV per RFC 4180. Prefer **no** embedded newlines in fields unless you control quoting rigorously.

Reserved words & identifiers.—We avoid quoted identifiers. Use *snake_case* and do not rename core columns without updating DDL, tests, and loaders consistently.

psql on Windows.—If using WSL, ensure the Windows path does not shadow WSL `psql`. Run `which psql` and verify version. If connecting to Windows Postgres from WSL, set `PGHOST=127.0.0.1`.

Permissions.—If you use a shared Postgres, confirm you have `CREATE` on the target schemas and `USAGE` on both `:schema` and `:terminology_schema`. See security guidance in the Security & Governance section.

12.5. *Troubleshooting Matrix*

`\copy` fails on column mismatch.: Compare the table DDL to the CSV header; run `head -1 file.csv | tr ',' '\n'` and diff with `\d :"schema".table`. Adjust header or `\copy` list.

Encoding errors.: Re-encode to UTF-8; search the file for bytes `0x92, 0x93, 0x94` (smart quotes) and replace.

Foreign-key test failures.: Load order may be wrong or source missing dimension rows. Confirm the referenced table has the key and re-run `make test`.

Test suite flaked by planner.: Run `ANALYZE` and re-test; consider raising `default_statistics_target` for skewed columns in dev.

CI green but local red.: Ensure your `.env` matches the CI versions (Postgres 16). Drop and recreate schemas: `make recreate-db`.

12.6. *FAQ & Recipes*

How do I (re)load a single table?—

```
# Option A: reload in place (truncate then copy)
psql -c "TRUNCATE :\"schema\".medical_claim;"
\copy :"schema".medical_claim(medical_claim_id, ..., tuva_last_run)
  FROM 'data/medical_claim.csv'
  WITH (FORMAT csv, HEADER true, NULL '', QUOTE '"', ESCAPE '"');

# Option B: stage then insert (safer for transforms)
psql -c "CREATE UNLOGGED TABLE IF NOT EXISTS :\"schema\"._stg_medical_claim
           (LIKE :\"schema\".medical_claim INCLUDING ALL);"
psql -c "TRUNCATE :\"schema\"._stg_medical_claim;"
\copy :"schema"._stg_medical_claim(...) FROM 'data/medical_claim.csv' WITH (...);
psql -c "INSERT INTO :\"schema\".medical_claim(...) SELECT ... FROM :\"schema\"._stg_medical_claim;"
```

How do I run a single test file?—

```
psql -v "schema=$schema" -f db/tests/observation_smoke.sql
```

If the test writes to :schema.test_results, you will see additional rows for that run. To list just failing rows:

```
psql -c "SELECT * FROM :\"schema\".test_results WHERE pass = false ORDER BY suite, test;"
```

How do I add a new table (core or terminology)?—

1. Create db/tables/<name>.sql with CREATE TABLE IF NOT EXISTS.
2. Add minimal indexes and light CHECKs; defer FKs if load order is uncertain.
3. Add a smoke test in db/tests/<name>_smoke.sql.
4. Place a CSV in data/ with matching headers.
5. Run make create-db → make load → make test.

How do I run only linting & style checks?—

```
pre-commit install
pre-commit run --all-files        # lint only
pre-commit run sqlfluff-psql-fix --all-files   # optional auto-fix
```

See the SQL Style & Tooling section for rule details and the psql-aware wrapper.

How do I get row counts and table sizes quickly?—

```
psql -c "SELECT schemaname, relname, n_live_tup
           FROM pg_stat_user_tables WHERE schemaname = :'schema' ORDER BY n_live_tup DESC;"
psql -c "SELECT n.nspname, c.relname,
                pg_table_size(c.oid)/1024 AS kb_table,
                pg_indexes_size(c.oid)/1024 AS kb_index
           FROM pg_class c JOIN pg_namespace n ON n.oid=c.relnamespace
           WHERE n.nspname = :'schema' AND c.relkind='r'
           ORDER BY kb_table DESC LIMIT 20;"
```

How do I investigate a failing test fast?—

1. Grab the failing test name from test_results.
2. Open the SQL file under db/tests/ that defines it.
3. Run the relevant query in isolation with a LIMIT and add diagnostic columns.
4. If it's a membership failure (terminology), inspect the terminology table and sample codes.

How do I mirror CI locally?—Use Docker to run Postgres 16 and match CI env vars:

```
docker run --rm -p 5432:5432 -e POSTGRES_PASSWORD=postgres -e POSTGRES_DB=gha postgres:16
export PGHOST=127.0.0.1 PGPORT=5432 PGUSER=postgres PGPASSWORD=postgres PGDATABASE=gha
export schema=ci_tuva terminology_schema=ci_terminology
make create-db load test
```

12.7. *Local–CI Parity and Versioning*

To prevent "works on my machine":

- **Pin** the major versions you use locally (Postgres 16, Python 3.x, sqlfluff version).
- Use the same Make targets as CI (create-db, load, test).
- Keep .env.example current; new variables must be documented there.

12.8. *Safety & Data Hygiene*

Seed datasets should be de-identified, but treat any real extracts as sensitive. Do not commit data files. Artifacts produced by tests (`test_results.csv`) are non-sensitive and may be uploaded by CI.[102]

12.9. *Example* `.env`

```
# Database
PGHOST=127.0.0.1
PGPORT=5432
PGUSER=postgres
PGPASSWORD=postgres
PGDATABASE=tuva_local

# Schemas
schema=tuva_core
terminology_schema=tuva_terminology

# Optional: tune psql behavior
PGAPPNAME=onboarding
```

12.10. *Make Targets (Quick Reference)*

```
make create-db      # idempotent DDL (schemas, tables, constraints, views)
make load           # \copy all CSVs under data/ (header-mapped)
make test           # run all SQL tests; populate :schema.test_results
make recreate-db    # drop + re-create schemas; re-apply DDL
make lint           # run sqlfluff lint via pre-commit wrapper
```

12.11. *Minimal Etiquette for Contributions*

- Branches use short, descriptive names; commits use Conventional Commits (`feat`, `fix`, `test`, `docs`, `ci`, `chore`).
- Run `pre-commit run --all-files` before pushing; keep diffs focused.
- Include tests when adding tables or rules; aim for a green `test_results`.

12.12. *Summary*

On day one, install the prerequisites, set `.env`, then run `make create-db`, `make load`, and `make test`. The repository is designed for deterministic, schema-parameterized operation; it uses `\copy` for fast client-side loads, aggregates all test outcomes in a standard table, and mirrors CI exactly. If something breaks, start with the pitfalls in §12.4 and the troubleshooting matrix in §12.5. Most issues reduce to header mismatches, encodings, or date formats—all solvable with a small, documented change. The rest of this document provides the depth you need as you begin to extend the model, add tests, and ship improvements.

13. LIMITATIONS & KNOWN GAPS

13.1. *Purpose*

No engineering artifact is without trade-offs. This section documents the limitations we have *intentionally* accepted to keep the Postgres loader reproducible and portable, as well as gaps we expect to address over time. We distinguish between (i) design choices that constrain functionality but improve operability, (ii) external dependencies that may fail or drift, and (iii) data-quality guardrails that are informative but not strictly enforcing. Where appropriate, we provide mitigation guidance and pointers to upstream references (PostgreSQL[103], GitHub Actions[104], and the Tuva Project[105]).

[102] Workflow artifacts: https://docs.github.com/actions/using-workflows/storing-workflow-data-as-artifacts.
[103] PostgreSQL Documentation: https://www.postgresql.org/docs/.
[104] GitHub Actions: https://docs.github.com/actions.
[105] The Tuva Project: https://thetuvaproject.com/.

13.2. *Soft Tests Do Not Block Ingestion*

What this means.—Many of the repository's tests are *soft* or *advisory* by design: they surface anomalies (membership gaps, plausibility issues, potential duplicate clusters) without raising a database error that would abort ingestion. Examples include:

- Membership checks against terminology tables (e.g., LOINC, CVX, HCPCS) reported as counts.
- Window plausibility (e.g., line dates within claim windows) treated as warnings.
- Soft duplicate detectors (e.g., person_id + date + normalized_code) that summarize clusters for review.

Rationale.—Upstream sources are heterogeneous and occasionally inconsistent. Hard-stopping loads for missing or stale reference codes or for rare but explainable orderings (e.g., late file dates) causes fragility and slows triage. A *soft-first* posture keeps the pipeline flowing while making quality signals visible in `:schema.test_results`.

Operational consequence.—Because these tests do not raise SQL exceptions, they *rely on CI policy* to enforce quality bars. If the CI job does not interpret the test results and fail the build on agreed thresholds (e.g., any "error" severity, or more than N "warn" rows in specific suites), poor-quality data can advance. This is an explicit trade-off between velocity and strictness.

Mitigations.—

1. Encode severity in `test_results` and make CI fail on `severity = 'error'` rows; treat `'warn'` as advisory with trend tracking.
2. Publish a project-level SLO: e.g., "no more than 0.1% of observation rows hit LOINC status *Deprecated/Discouraged*" and enforce it in CI.
3. For use-cases that *require* strictness, convert specific tests to hard constraints or pre-insert gating (e.g., staging table → filtered insert).

13.3. *Large Terminology Sets Externalized*

What this means.—Some terminologies are too large or too frequently updated to ship as repository seeds (e.g., ICD-10-CM, ICD-10-PCS, LOINC, SNOMED CT, large provider directories). The project therefore expects consumers to load these from *public cloud storage* or organization-internal mirrors. The core and test SQLs *assume* such tables exist under `:terminology_schema` with agreed shapes.

Risks.—

- **Availability**: network outages or credential issues prevent ingestion on fresh environments.
- **Drift**: reference tables may change upstream (additions, deprecations), impacting test outcomes run-to-run.
- **Version skew**: developers may load differing vintages of the same terminology, making local vs. CI results diverge.
- **Licensing & redistribution**: some vocabularies carry license terms or redistribution limits that preclude bundling.

Mitigations.—

1. **Pin and verify**: record a source URI and checksum for each fetched artifact and verify before load. Archive a copy in an internal, access-controlled bucket with immutability (object lock) when permissible.
2. **Version columns**: store a `version` or `as_of_date` in terminology tables; surface this in CI logs so diffs are explainable.
3. **Cache on first use**: for developer machines, cache artifacts locally and refresh by explicit command.
4. **Fail fast**: if a required terminology is missing, create a targeted test that *fails the pipeline* with a clear message rather than silently degrading other tests.

13.4. *Partial Referential Integrity (FK) Coverage*

What this means.—Not all cross-table relationships are enforced with relational `FOREIGN KEY` constraints. Where sources are sparse or inconsistent (e.g., optional `encounter_id` on claims, late-arriving dimension keys, multiple-member identifiers per person), we use *deferrable FKs* selectively and rely on *presence tests* and *consistency checks* instead.

Rationale.—Strict FKs can be too brittle for multi-source loads: a single missing dimension row stalls the entire ingest. Keeping some relationships soft preserves throughput and allows downstream reconciliation (e.g., via crosswalk tables and late-binding joins).

Consequences.—

- Consumers can accidentally join on keys with low completeness (e.g., sparse `encounter_id` in claims) and infer biased results.
- A typo or mapping drift in `person_id_crosswalk` may not block load but will appear as mismatched person/encounter tests.

Mitigations.—

1. **Surface completeness**: maintain and publish per-key completeness metrics by table (e.g., % of rows with non-null `encounter_id`).
2. **Guide joins**: document preferred join paths per domain (claims → encounters via `person_id + date` when `encounter_id` is absent).
3. **Escalate selectively**: promote specific presence tests to hard FKs once data hygiene is proven for a customer/source family.

13.5. *Temporal Semantics and Time Zones*

Limitations.—The corpus uses `date` for many events and `timestamp without time zone` (*"timestamp_ntz" in prose*) where datetimes are available. This avoids conflating stored values with session time zones but shifts responsibility to consumers to interpret offsets correctly. Cross-system comparisons (e.g., EHR vs. claims) can exhibit off-by-hours behavior around DST transitions if naive comparisons are used.

Mitigations.—

- Treat `timestamp without time zone` as UTC-equivalent when ingest sources are normalized; otherwise, add source-level `time_zone` metadata and convert on read.
- Prefer date-based reasoning for claims periods; reserve time-of-day analytics for explicitly time-stamped clinical feeds.

13.6. *Plausibility Checks Are Narrow by Design*

Scope limits.—Plausibility add-ons (e.g., NDC digit length, geo bounds, LOINC status surfacers, immunization series spacing) are intentionally lightweight. They detect gross errors, not clinical nuance. For instance, numeric lab plausibility does not encode analyte-specific physiological ranges; immunization spacing rules are not brand- or age-tailored without additional rule sets.

Mitigations.—Domain teams can layer richer rule packs (age/brand-aware vaccine intervals, analyte- and unit-aware lab ranges) on top of the core signals. Keeping the core tests lightweight preserves speed and portability.

13.7. *Performance Trade-offs*

Default posture.—Loads favor simplicity over maximal throughput: single-stream `\copy` per table, indexes built post-load but without aggressive session tuning, and partitioning as an *opt-in*. This keeps the repository portable (local laptops, CI containers) at the cost of top-end performance on very large facts.

Consequences.—Extremely large inputs (e.g., 10^9+ claim lines) will benefit from partitioning, parallel ingest, and session-level tuning (e.g., `maintenance_work_mem`) that are not enabled by default.

Mitigations.—Adopt the advanced playbook (partition by month, parallel `\copy` into children, CONCURRENT index builds where needed) when scale demands; document such choices in environment-specific overlays.

13.8. *Security and Data Handling Boundaries*

Out-of-scope items.—The repository does not prescribe encryption at rest or in transit, nor does it ship data masking or row-level security policies. It assumes de-identified seed data and ephemeral CI databases. Productionizing these controls is environment-specific and must be handled by the deploying organization.

Mitigations.—Follow the Security & Governance guidance: least-privilege roles, schema isolation in CI, secret management via environment (no secrets in git), and pinned supply-chain components.[106]

13.9. *Tooling Compatibility*

Linting vs. execution.—The codebase uses psql variables (e.g., `:schemä`) and prose types like *timestamp_ntz* in comments. Linters may not understand these constructs without a wrapper. We provide a sqlfluff wrapper that normalizes the dialect for linting *only*. This introduces a mild risk of drift between what the linter accepts and what Postgres executes.

Mitigations.—Run linting and execution in CI; treat any file that requires a local rule waiver as a candidate for simplification.

13.10. *Externalized Observability*

Scope.—Observability artifacts (`test_results.csv`, `run_metrics.csv`, `table_row_counts.csv`) are intentionally minimal and non-sensitive. They do not include record-level samples, which can slow triage for complex failures.

Mitigations.—Provide reproducible SQL snippets in failing test files to materialize small diagnostic samples locally; add optional sampling tests that write into a short-retention schema for exploratory debugging (never in CI artifacts).

13.11. *Known Functional Gaps (Non-exhaustive)*

- **Terminology deltas**: No built-in diffing of terminology vintages (e.g., which ICD codes changed between releases).
- **Panel semantics**: Observation and lab panel integrity checks are time-windowed but do not model full order-component dependency trees.
- **Provider identity**: NPI plausibility is limited to structural checks; resolving organizations vs. individuals across systems is out of scope.
- **Eligibility overlaps**: "Soft" detection of overlaps and gaps flags issues but does not auto-resolve coverage intervals.

13.12. *Future Work*

1. **Configurable enforcement**: Promote selected soft tests to hard failures via a project config (*"turn warnings into errors"* by suite).
2. **Terminology bundles**: Provide optional, periodically refreshed bundles with checksums to reduce network dependencies.
3. **FK graduation**: Track stability metrics and automatically graduate presence tests to FKs when completeness exceeds thresholds for N consecutive runs.

[106] Security hardening for Actions: https://docs.github.com/actions/security-guides/security-hardening-for-github-actions.

4. **Drift detection**: Add a small *data contract* layer (expected ranges of row counts by table, expected distinct code counts) with CI gating.

5. **Performance profiles**: Ship an advanced profile (partitioning templates, parallel ingest scripts) for very large tenants.

13.13. *Summary*

The project emphasizes portability, speed, and clarity over absolute strictness. Soft tests surface issues without blocking ingestion; large terminologies are externalized to keep the repository light; and referential integrity is applied pragmatically where sources are known to be variable. These choices require discipline in CI policy (to enforce quality bars), careful handling of external dependencies (to prevent drift and outages), and transparency about key completeness. With these guardrails, the system remains reproducible and useful across a wide range of environments while leaving room for organizations to harden and scale where needed.

14. ROADMAP / FUTURE WORK

14.1. *Vision*

The next phase evolves this repository from a high–fidelity, batch-oriented seed loader into a *continuously reliable* ingestion and validation platform. Concretely, we will (1) introduce incremental loads and change data capture (CDC) to reduce runtime and cost; (2) apply physical design patterns (partitioning, targeted indexes, and caching) to sustain interactive query performance at scale; (3) formalize *test severity* and thresholding so that soft findings become policy-driven gates; (4) integrate with catalogs and documentation generators for discoverability and lineage; and (5) provide a synthetic-data generator to light up ephemeral PR environments without exposing sensitive information. Each stream below lists objectives, design options, risks, and success criteria, with references to upstream technology where relevant.[107][108]

14.2. *Incremental Loads & Change Data Capture (CDC)*

Objectives.—Shrink end-to-end wall time and compute by loading only deltas; enable more frequent quality feedback (e.g., intra-day runs) without full reloads.

Minimum viable design.—

1. **Idempotent merge layer**: For each large fact (e.g., `medical_claim`, `lab_result`, `observation`), introduce a *landing* table with metadata columns `source_file`, `landed_at`, and an optional `op` (I/U/D). Ingest new CSVs via `\copy` into landing, then *upsert* into the base using a stable key (e.g., `medical_claim_id`) with `INSERT ... ON CONFLICT ... DO UPDATE`.

2. **Change detection**: If upstream can emit *modified-since* or manifests with checksums, skip files already seen; otherwise compute file digests locally and maintain a manifest table.

3. **Audit and replay**: Record each merge in an `ingest_log(run_id, table_name, rows_ins, rows_upd, rows_del)` table for observability.

Advanced options.—

- **Logical replication / wal2json**: For sources that can publish changes, consume a logical stream and materialize into landing tables.[109]
- **Debezium-style CDC**: Where the system of record is not Postgres, use an external capture pipeline (e.g., Debezium) to produce append-only change topics, then micro-batch into the warehouse.[110]
- **Temporal history**: For a limited set of dimensions (e.g., `practitioner`, `location`), add an effective-dated history table (Type 2 SCD) when downstream consumers must query past states.

[107] PostgreSQL Documentation: https://www.postgresql.org/docs/.
[108] The Tuva Project: https://thetuvaproject.com/.
[109] Logical replication: https://www.postgresql.org/docs/current/logical-replication.html.
[110] Debezium Documentation: https://debezium.io/documentation/.

Risks.—*Key stability* (do upstream identifiers mutate?), *late-arriving* updates that correct prior data, and *delete semantics*. We will treat hard deletes conservatively: first mark as `is_deleted` in base; only physically purge on a retention cycle.

Success criteria.—50–90% runtime reduction on routine runs (dependent on change rate); deterministic idempotence proven by repeated replays; audit log reconciles with test outcomes per run.

14.3. *Partitioning & Query Acceleration*

Objectives.—Maintain sub-second to low–seconds response for common filters and joins as row counts grow to 10^8–10^9+ across facts.

Plan.—

1. **Range partitioning by month** for `observation`, `lab_result`, and `medical_claim` on their canonical date columns (e.g., `result_datetime::date`, `claim_start_date`).[111]
2. **Per-child indexes** for hot predicates (e.g., `(person_id, result_datetime)`, `(member_id, payer, plan)`); ensure planner visibility with `ANALYZE` after loads.
3. **Pruning-aware SQL**: codify patterns that keep partition keys *simple* in WHERE clauses (avoid wrapping in functions).
4. **Materialized accelerators** (optional): month-level aggregates for frequent QA metrics (e.g., distinct code counts, failure tallies) to power dashboards without scanning base facts.[112]

Enhancements.—

- **Parallel load by partition**: micro-batch CSVs per month and run multiple `\copy` sessions in parallel.
- **Covering indexes**: add INCLUDE columns to avoid heap lookups on narrow projections (PostgreSQL 11+).[113]

Success criteria.—Partition pruning verified in `EXPLAIN`; p95 latency improvements for time–bounded queries; index build time contained by per-child scope.

14.4. *Test Severity Levels & Thresholds*

Objectives.—Elevate *soft* tests into policy by annotating each assertion with `severity` (`info`|`warn`|`error`) and enforcing *thresholds* per suite in CI. This reconciles the need for velocity (soft checks) with the need for guardrails (blocking on *error*).

Design.—

1. **Schema**: extend `:schema.test_results` with `severity` and optional `owner` (team label).
2. **Registry**: add a YAML policy file mapping *suite/test* to default severities and thresholds (e.g., max allowed failures).
3. **CI gate**: a small Python step reads the policy, queries `test_results`, and fails the job if thresholds breach. Output a Markdown summary grouped by suite and severity.

Evolution.—Start conservative (block on `error` only); after baselining, promote selected `warn` suites (e.g., ICD/LOINC membership) to `error` for stable sources.

Success criteria.—Deterministic gating in PRs; ability to suppress flapping tests by adjusting thresholds (with audit trail in git).

[111] Table Partitioning: https://www.postgresql.org/docs/current/ddl-partitioning.html.
[112] Materialized Views: https://www.postgresql.org/docs/current/sql-creatematerializedview.html.
[113] Covering indexes via INCLUDE: https://www.postgresql.org/docs/current/sql-createindex.html.

14.5. *Catalog Integration (Docs, Lineage, Tags)*

Objectives.—Make the data model discoverable and traceable across tools. Provide table/column docs, usage metadata, and *where possible* basic lineage from loaders to tables to tests.

Options.—

- **dbt-style artifacts**: emit *manifest*-like JSON for tables/tests (even without dbt) and publish a static site for docs.[114]
- **Open Metadata / Amundsen**: push table/column docs and tags via their APIs for teams with an existing catalog backbone.[115][116]
- **Lightweight SQL comments**: store `COMMENT ON TABLE/COLUMN` in DDL and export via a periodic `psql` job for catalog ingestion.[117]

Tagging.—Use tags to signal semantics and governance (`@terminology`, `@pii:none`, `@core`, `@fact/observation`). Tests can inherit tags from tables to drive dashboards (e.g., failure heatmaps by tag).

Success criteria.—Humans can browse model docs in a single place; tables show owner, freshness (last load), and links to recent `test_results`. PRs that add tables require docs and tags.

14.6. *Synthetic Test Data Generator for PR Environments*

Objectives.—Spin up ephemeral environments for every PR without real data. Support smoke tests, join correctness checks, and performance *shape* tests.

Approach.—

1. **Schema–aware faker**: a Python utility reads table DDL and generates minimally valid rows that satisfy key constraints and CHECKs; includes knobs for skew (e.g., Zipf for code frequencies).
2. **Relational integrity**: derive child rows from parent distributions (e.g., `encounter` from `patient`, `medical_claim` lines per claim header).
3. **Domain packs**: optional packs for realistic codes (LOINC, ICD, HCPCS) using *small* curated subsets to keep the repo light; large vocabularies remain external.[118]
4. **Workload shapes**: ship presets ("tiny", "dev", "scale") with row-count targets per table to exercise parallel loads and partition pruning in CI.

Safeguards.—All generated values are non-sensitive; time-based fields are confined to recent synthetic windows; random seeds are fixed per run type for reproducibility.

Success criteria.—Every PR launches a Postgres container, loads synthetic data in <3 minutes ("dev" shape), and runs the full test suite; failures are attributable to code, not data availability.

14.7. *Cross-Cutting Enhancements*

Observability upgrades.—Add per-step timings to `run_metrics` for CDC vs. full-load comparisons; record partition maintenance durations; track p95/p99 ingest throughput by table to detect regressions (cf. Observability section).

Operational profiles.—Introduce environment overlays:

- **dev**: single-stream `\copy`, no partitions, minimal indexes.
- **staging**: monthly partitions, parallel loads, core indexes.
- **prod-like**: full partitions, extended indexes, materialized accelerators, advanced autovacuum tuning.

Overlays select DDL variants and Make targets via environment variables.

[114] dbt Artifacts (manifest.json): https://docs.getdbt.com/reference/artifacts/manifest-json.
[115] OpenMetadata Docs: https://docs.open-metadata.org/.
[116] Amundsen Project: https://www.amundsen.io/.
[117] COMMENT command: https://www.postgresql.org/docs/current/sql-comment.html.
[118] LOINC and licensing: https://loinc.org/terms-of-use/.

Governance checkpoints.—Require (in CI) that new tables include: (1) table/column comments, (2) at least one smoke test, (3) tags for classification, and (4) an entry in the synthetic-data generator profiles.

14.8. *Risks & Mitigations*

- **Complexity creep**: CDC, partitions, and accelerators can multiply DDL and code paths. *Mitigation*: keep a "simple path" profile; encapsulate CDC in shared SQL templates; aggressively document.
- **Drift between local and CI**: *Mitigation*: pin versions (Postgres 16, Python, sqlfluff), publish a devcontainer or Docker Compose for parity.
- **Flaky gates**: thresholds that oscillate near limits. *Mitigation*: use rolling medians and hysteresis; fail only after N consecutive breaches.

14.9. *Milestones & Measures*

14.10. *Summary*

This roadmap emphasizes *incrementality*, *scale-aware design*, and *operational discipline*. CDC and partitioning reduce time and cost; severity and thresholds transform advisory checks into enforceable policy; catalog integration improves discoverability and trust; and synthetic data makes quality visible on every PR without risking sensitive information. The body of work remains intentionally modular so teams can adopt streams in order of need, while preserving the repository's core tenets: determinism, portability, and clear quality signals.

15. CONCLUSION

This repository delivers a reproducible, validator–first pathway for loading Tuva seed datasets into PostgreSQL. By combining one–file–per–table DDL, deterministic ingestion via `\copy`, a layered terminology model, and a comprehensive suite of smoke and add–on tests, the project turns a traditionally manual, error–prone process into a standardized pipeline with clear quality signals. In practice, it improves reliability, reproducibility, and reviewability for health data engineering and analytics teams depending on the Tuva model for downstream work.[119]

Reliability is increased through several complementary mechanisms applied early in the flow. Core tables enforce light but meaningful integrity guards (date ordering, boolean–like flags, non–negative monetary fields), while smoke tests cover universal invariants such as primary–key uniqueness, foreign–key presence, and line–within–header time windows. Domain add–ons then raise the bar further: membership and plausibility checks against the terminology layer (ICD, LOINC, CVX, HCPCS, MS–DRG, place of service, POA, SNOMED) are joined by healthcare–specific heuristics including panel integrity for laboratory data, eligibility continuity and gap detection, immunization series spacing, and soft duplicate detectors for events that are frequently double–documented. Collectively, these checks surface data drift close to ingestion and translate ambiguous failures into concrete remediation items.

Reproducibility is engineered into the system rather than retrofitted. All schemas are defined in dedicated files (one per table), parameterized by `:''schema''` and `:''terminology_schema''` so the same DDL targets multiple environments without modification. Loads rely on `psql \copy`, avoiding server–side filesystem access and behaving deterministically across developer laptops and CI containers.[120] A short, explicit Make sequence (`create-db` → `load` → `test`) reproduces a known database state on demand. Style and hygiene are standardized with a strict `sqlfluff` configuration, enforced locally via pre–commit hooks and centrally in CI, ensuring that formatting and linting outcomes do not vary by workstation or editor.[121]

Reviewability hinges on consistent commit hygiene and machine–readable test outputs. Conventional commits and a predictable repository layout communicate the intent of each change (schema, loader, terminology, tests) and keep diffs focused. Every check emits a stable `(test, pass, metrics…)` record into a consolidated `$ {schema}.test_results` relation, which CI executes inside a PostgreSQL 16 service container and archives as `test_results.csv`. When critical checks fail, the workflow fails with a clear breadcrumb trail to the offending rows and queries, giving maintainers a single place to review status, compare runs, and attribute regressions to commits.[122]

Some trade–offs are intentional. Large terminology sets are externalized and retrieved through adapter–specific paths rather than shipped as seeds, introducing a dependency on network availability and provenance controls. Foreign–key

[119] The Tuva Project: https://thetuvaproject.com/.
[120] PostgreSQL documentation (including `psql` and `\copy`): https://www.postgresql.org/docs/.
[121] sqlfluff: https://docs.sqlfluff.com/.
[122] GitHub Actions: https://docs.github.com/actions.

Table 2. Roadmap streams with outcome-centric measures.

Stream	Deliverables	Dependencies	Success Metric
Incremental & CDC	MERGE/upsert; `ingest_log`; replay script	Stable keys; manifests/checksums	$\geq$ 50% faster routine runs
Partitioning & Acceleration	Monthly partitions; pruning-safe SQL; per-child indexes; MVs (opt.)	PostgreSQL 16; index storage budget	p95 latency $\downarrow$ 40% (time-bounded)
Severity & Thresholds	`test_results` severity; YAML policy; CI gate step	Policy owners; threshold buy-in	Zero false-positive PR blocks (2 wks)
Catalog Integration	COMMENTS; tags; docs site or catalog push	Catalog platform (optional)	Docs coverage $\geq$ 95%; PRs require docs
Synthetic Data	Schema-aware generator; 3 shapes; PR job wiring	Python toolchain; curated code packs	PR env $<$ 3 min; deterministic results

constraints are applied judiciously (often deferrable) to balance data integrity with heterogenous source feeds. Several domain checks are configured as *soft* until organizations establish severity and thresholds; enforcement is handled by CI policy to avoid blocking ingestion prematurely.

Next steps are clear and map naturally to team ownership. Data Engineering should continue to own DDL evolution, loader scripts, and CI infrastructure, with an eye toward performance features such as incremental loads and partitioning. Analytics Engineering and QA should steward the test catalog, assign severities, and set thresholds that convert advisories into enforceable policies. Security and Platform should maintain credential handling via `.env` (no secrets in the repository), least–privilege roles, and supply–chain pinning for runners and linters. Concretely, the immediate roadmap includes: (i) adding *severity* to `test_results` and gating CI on `error` while reporting `warn`; (ii) introducing idempotent upserts for large facts to reduce wall–time without sacrificing determinism;[123] (iii) applying monthly range partitioning to time–series facts to sustain low–latency reads at scale;[124] (iv) publishing table/column comments and test coverage to a catalog or static documentation site; and (v) bundling a small, schema–aware synthetic data generator to validate pull requests quickly and safely. Make remains the glue for these tasks, providing a transparent, inspectable orchestration layer that plays well in both developer shells and CI runners.[125]

In sum, reliable health analytics pipelines require more than correct schemas; they require repeatability, fast feedback, and shared ownership. This project encodes those principles in code, tests, and CI. The immediate payoff is shorter diagnosis cycles and fewer surprises; the longer–term benefit is trust: stakeholders can see not only that the data loaded, but how well it conforms to the expectations embedded in the model and terminology layer. With severity gating, incremental ingestion, partitioning, and catalog hooks on the near horizon, the repository is positioned to serve as a reference–quality, production–adjacent workflow for teams standardizing on the Tuva model.

ACKNOWLEDGMENTS

I am grateful to the colleagues and collaborators, particularly Mary Armstrong & Barry Wood, who made this work possible through guidance, reviews, and hands–on contributions across engineering, analytics, and domain expertise. Any remaining errors or omissions are my own.

Facilities: Health System Enterprise Analytics (HSEA), HIPAA-compliant cloud research environment (AWS), Health information exchange (HIE) data collaborative

Software: PostgreSQL 16, psql, GNU Make, Bash, GNU coreutils, Python 3.x, Git, pre-commit, sqlfluff, GitHub Actions, Docker (PostgreSQL 16 service container), actions/checkout, actions/setup-python, XeLaTeX (MiKTeX), AASTeX7 class, natbib, hyperref, graphicx, xcolor, url, longtable, array, rotating, ulem, lineno, pifont

[123] PostgreSQL `INSERT …ON CONFLICT` (upsert): https://www.postgresql.org/docs/current/sql-insert.html#SQL-ON-CONFLICT.
[124] PostgreSQL table partitioning: https://www.postgresql.org/docs/current/ddl-partitioning.html.
[125] GNU Make: https://www.gnu.org/software/make/.