Github Link : https://github.com/eemilycchen/drug_safety_and_recommendation

MediDB

A Drug Safety & Recommendation System

Multi-Database Clinical Decision Support

Sanjana Garimella · Priyanka Nidadavolu · A.J. Olivares · Emily Chen

Project Report · 2026

Table of Contents

1. Executive Summary

This project implements a multi-database clinical decision-support tool that assesses the safety of a proposed medication for a given patient. It integrates four complementary database systems: PostgreSQL, Neo4j, Qdrant, and MongoDB. Each database was chosen for the type of query it answers best. Given a patient ID and a proposed drug, the system retrieves the patient's current medications, checks for drug–drug interactions and polypharmacy risk, finds similar adverse-event cases, and returns a unified safety report with full evidence traceability. Our code is released here for reference and future works: https://github.com/eemilycchen/drug_safety_and_recommendation.

Database	Role	Key Capability	Data Source
`PostgreSQL`	Patient EHR	Current meds, conditions, allergies, profile	Synthea
`Neo4j`	Drug Knowledge Graph	Interactions, side effects, polypharmacy clusters	DrugBank / SIDER
`Qdrant`	Vector Similarity	Similar FAERS cases, safe alternatives	openFDA FAERS
`MongoDB`	Evidence & Audit	Raw FAERS, run logs, full traceability	openFDA FAERS

2. Introduction

2.1 Problem Statement

When clinicians prescribe a new medication, they must simultaneously consider multiple risk dimensions that no single database model handles optimally:

- The patient's current active medications and full clinical profile
- Drug–drug interactions and whether the proposed drug creates or bridges interaction clusters (polypharmacy risk)
- Similar real-world cases from adverse-event databases such as FAERS
- Evidence supporting any warnings and audit trails for reproducibility

While relational databases excel at structured EHR queries, graph databases are ideal for interaction networks. Because adverse-event similarity cannot be expressed as a SQL query or keyword match, a vector database is necessary. It encodes clinical narratives as dense embeddings and retrieves the most semantically similar FAERS cases, surfacing real-world evidence that structured databases would miss entirely. To ensure full traceability, MongoDB stores both the raw openFDA FAERS reports in their original nested form and a complete audit log of every safety-check run, linking each recommendation back to the evidence that produced it. Ultimately, a multi-model architecture is required to capture the full spectrum of patient care.

2.2 Solution Overview

The system orchestrates four specialised databases, each contributing a distinct capability to a unified pipeline.

3. System Architecture

3.1 Data Flow

The diagram below shows the end-to-end data flow: four data sources, four ETL scripts, four databases, and one application that synthesises a unified safety report. The pipeline is triggered by Patient ID and a Proposed Drug, and then it orchestrates four parallel database queries before producing the final output.

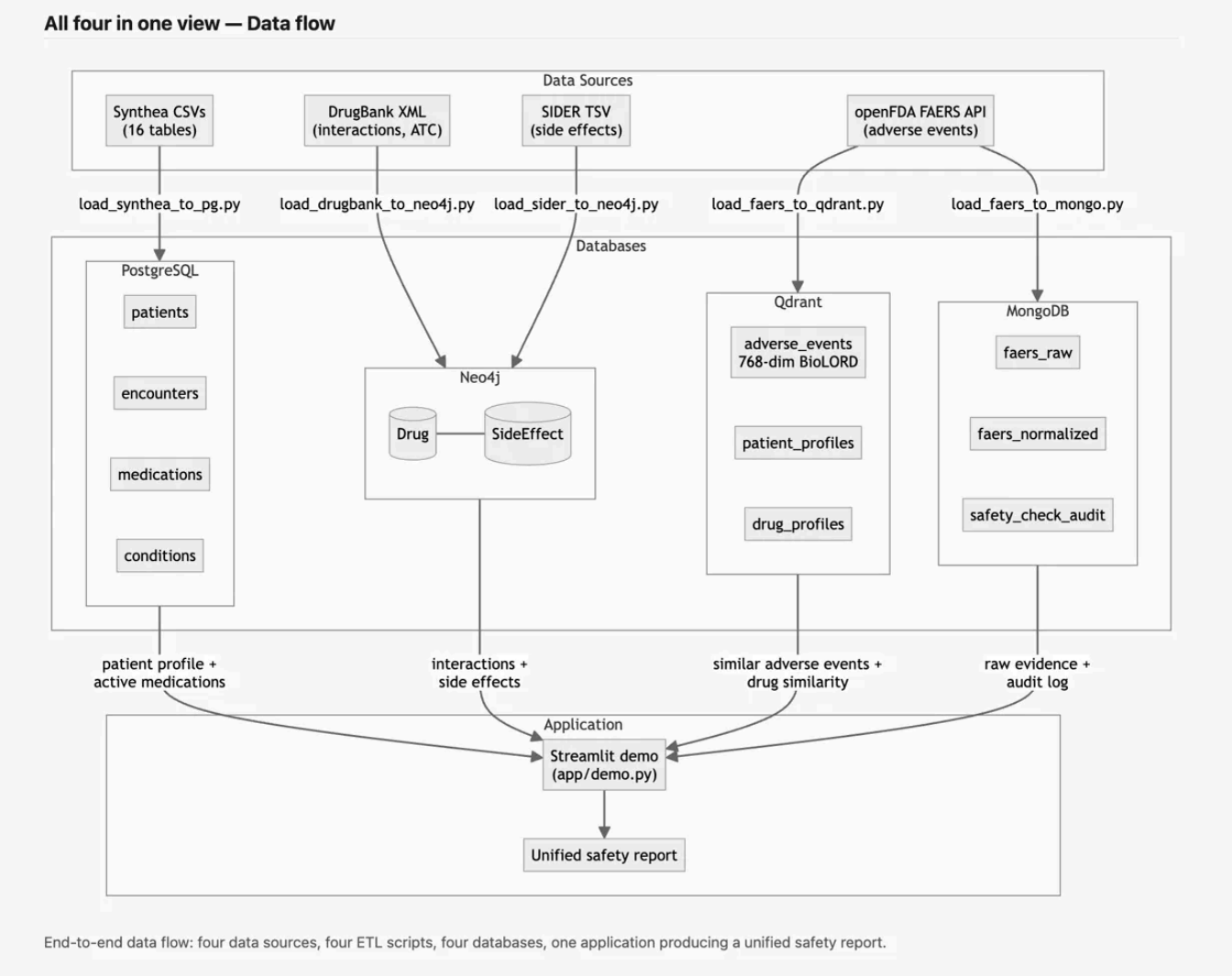


Figure 1 — End-to-end data flow: four data sources, four ETL scripts, four databases, one unified safety report.

The pipeline pseudocode below illustrates how each database is called in parallel:

```
Input: Patient ID + Proposed Drug
  ├── PostgreSQL : get_active_medications(), get_patient_profile()
  ├── Neo4j      : check_interactions(),
detect_polypharmacy_clusters()
  ├── Qdrant     : find_similar_adverse_events_multi_filter()
  └── MongoDB    : get_faers_reports_by_ids(), log_safety_check()
Output: Unified safety report (risk level, interactions,
alternatives, evidence)
```

3.2 Data Sources

Source	Content	Target DB
`Synthea`	Synthetic patient EHR — patients, encounters, medications, conditions, allergies	PostgreSQL
`DrugBank XML`	Drug–drug interaction data (INTERACTS_WITH relationships)	Neo4j
`SIDER TSV`	Drug–side-effect relationships (MedDRA ontology)	Neo4j
`openFDA FAERS`	Real-world adverse event reports	Qdrant + MongoDB

4. Database Design & Implementation

4.1 PostgreSQL — Patient EHR

Why PostgreSQL?

EHR data is highly structured, typed, and relational. Patient records reference encounters. Medications reference patients and encounters, and conditions reference the same hierarchy. PostgreSQL enforces referential integrity through foreign keys, ensures ACID-compliant writes, and enables complex multi-table JOINs that would be cumbersome in document or graph stores. The stop_ts IS NULL predicate enables precise active-medication filtering, which is the first and most critical step in every safety check.

Schema

The schema follows a star design with the patients table at the centre. All clinical tables carry patient_id and encounter_id foreign keys.

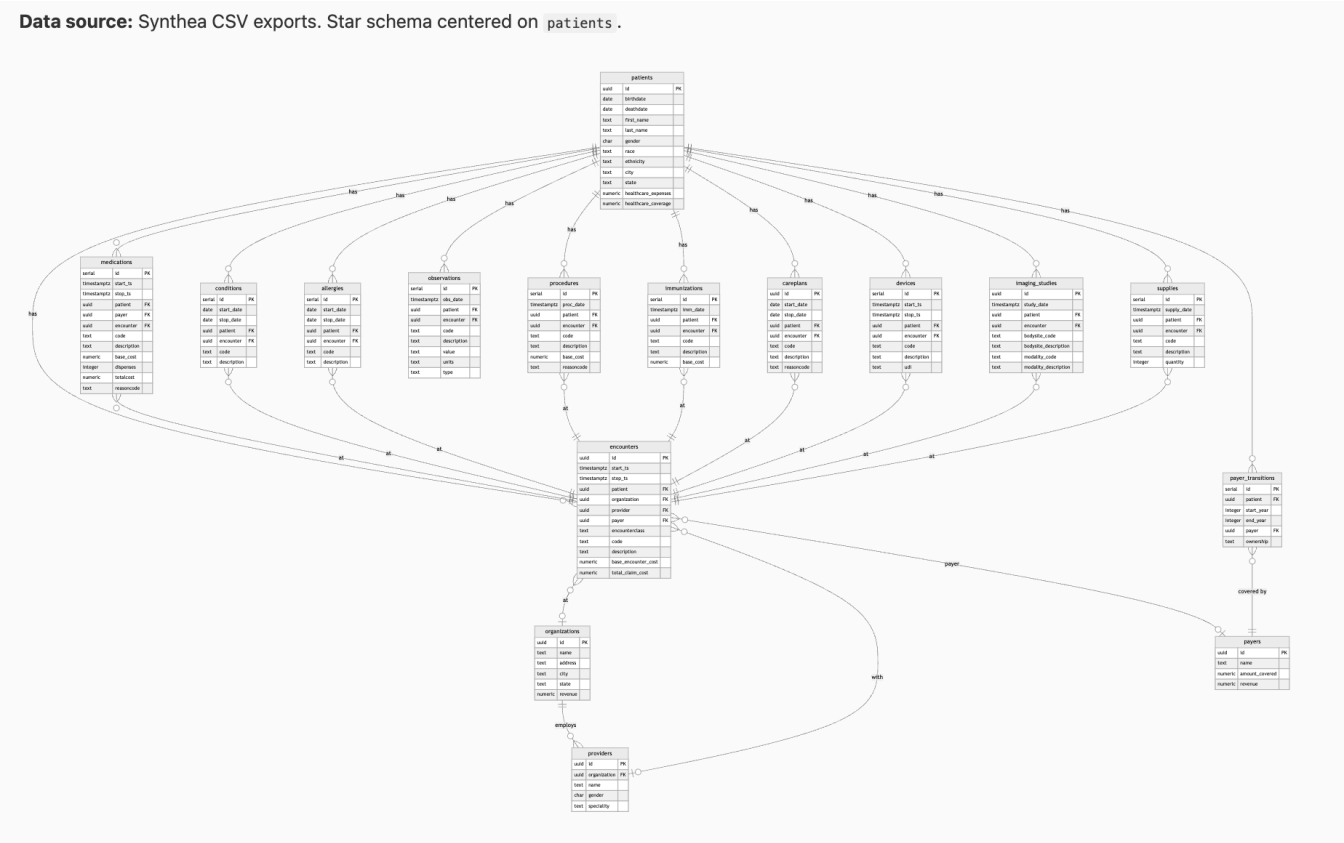


Figure 2 — PostgreSQL entity-relationship diagram. All 16 Synthea tables. Patient-centric indexes on every clinical table for fast per-patient lookups. For our project, we only used a selected amount of tables.

Table	Key Fields
patients	id (PK uuid), birthdate, deathdate, race, ethnicity, gender, city, state, zip, first, last
encounters	id (PK), start_ts, stop_ts, patient_id (FK), provider_id (FK), organization_id (FK), encounter_class, description
medications	id (PK), start_ts, stop_ts, patient_id (FK), encounter_id (FK), description (drug name), reasondescription
conditions	id (PK), start_ts, stop_ts, patient_id (FK), encounter_id (FK), code (SNOMED), description
allergies	id (PK), start_ts, stop_ts, patient_id (FK), encounter_id (FK), description
observations	id (PK), date_ts, patient_id (FK), encounter_id (FK), description, value, units
procedures	id (PK), start_ts, stop_ts, patient_id (FK), encounter_id (FK), description
providers	id (PK), name, organization_id (FK), speciality, gender

Key Interface Functions

- get_active_medications(patient_id) — Returns medications where stop_ts IS NULL
- get_patient_profile(patient_id) — Demographics, active meds, conditions, allergies, recent observations
- get_medication_history(patient_id, …) — Full medication history with optional date filters
- get_patient_timeline(patient_id, …) — Chronological clinical event stream
- list_patients(limit) — Patient list for demo dropdowns
- rolling_total_cost(patient_id) — Running cumulative medication spend per patient (window function)
- rolling_med_count(patient_id) — Running count of prescriptions over time per patient (window function)

Rolling Window Functions

Two PostgreSQL window functions provide rolling analytics over a patient's medication history. Both use the same OVER (PARTITION BY patient_id ORDER BY start_ts) pattern, which computes a running aggregate per patient ordered chronologically, without collapsing rows into a single summary as GROUP BY would.

rolling_total_cost()

```
SELECT
    patient_id,
    description,
    start_ts,
    total_cost,
```

```
    SUM(total_cost) OVER (
        PARTITION BY patient_id
        ORDER BY start_ts
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total_cost
FROM medications
WHERE patient_id = $1
ORDER BY start_ts;
```

rolling_med_count()

```
SELECT
    patient_id,
    description,
    start_ts,
    COUNT(*) OVER (
        PARTITION BY patient_id
        ORDER BY start_ts
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_med_count
FROM medications
WHERE patient_id = $1
ORDER BY start_ts;
```

Critical Query — get_active_medications()

```
SELECT description, start_ts, reasondescription
FROM medications
WHERE patient_id = $1
  AND stop_ts IS NULL
ORDER BY start_ts DESC;
```

4.2 Neo4j — Drug Knowledge Graph

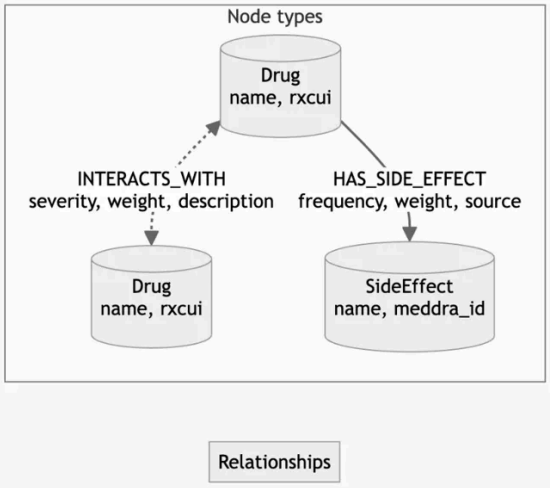
Why Neo4j?

Drug interactions are inherently a graph problem. A patient on N medications requires checking $O(N^2)$ pairwise interactions. A relational approach must perform N^2 SELECT queries or execute a self-join on a large interactions table. In Neo4j, the same check is a single graph traversal, where relationships are first-class citizens stored adjacently on disk, making traversal O(1) per hop regardless of graph size.

Beyond pairwise checks, Neo4j enables polypharmacy cluster detection. It identifies groups of drugs that collectively form a high-risk interaction network. This analysis is computationally impractical in SQL but natural in Cypher.

Graph Schema

Data sources: DrugBank Open XML (`load_drugbank_to_neo4j.py` → Drug nodes + INTERACTS_WITH), SIDER TSV (`load_sider_to_neo4j.py` → SideEffect nodes + HAS_SIDE_EFFECT).



Schema: node labels, properties, and relationship types with their properties.

Figure 3 — Neo4j graph schema: Drug nodes linked by INTERACTS_WITH (severity, weight, description) and HAS_SIDE_EFFECT (frequency, weight, source) to SideEffect nodes.

Component	Type	Properties
`(:Drug)`	Node	name, rxnorm_id, drugbank_id
`(:SideEffect)`	Node	meddra_id, description, frequency
`[:INTERACTS_WITH]`	Relationship	severity ('MAJOR'|'MODERATE'|'MINOR'), source ('DrugBank'), description
`[:HAS_SIDE_EFFECT]`	Relationship	frequency, source ('SIDER')

Example Graph Instance

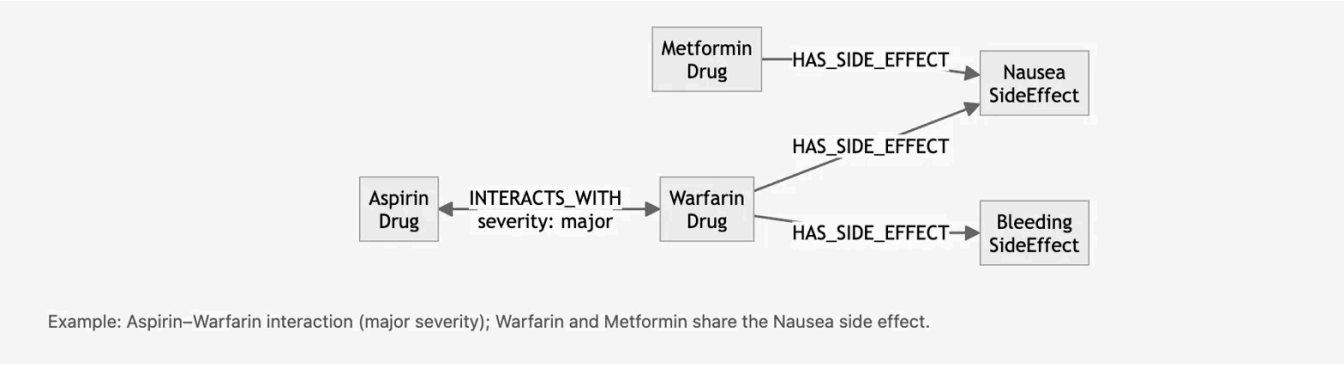


Figure 4 — Example: Aspirin–Warfarin INTERACTS_WITH (major severity); Warfarin and Metformin share the Nausea side effect via HAS_SIDE_EFFECT.

Key Interface Functions

- check_interactions(current_meds, proposed_drug) — Returns all pairwise interaction pairs with severity
- detect_polypharmacy_clusters(current_meds, proposed_drug) — Identifies interaction clusters, bridge drugs, and computes a risk score
- get_side_effects(drug) — Returns known MedDRA side effects for a drug
- find_interacting_group(drug) — Returns all drugs that interact with a given drug

Polypharmacy Example — Why Graph Wins

Consider a patient on four existing medications (Aspirin, Ibuprofen, Lisinopril, Metformin) for whom Warfarin is proposed. Neo4j detects the following in a single traversal:

- Warfarin ↔ Aspirin: MAJOR interaction (bleeding risk)
- Warfarin ↔ Ibuprofen: MAJOR interaction (bleeding / renal risk)
- Warfarin ↔ Lisinopril: MODERATE interaction (hyperkalaemia risk)
- Bridge drug identified: Warfarin connects two pre-existing MAJOR-risk nodes simultaneously
- Cluster risk score: HIGH — 3-node cluster with 2 MAJOR edges triggers polypharmacy alert

```
MATCH (d:Drug)-[r:INTERACTS_WITH]-(n:Drug)
WHERE d.name = $proposed AND n.name IN $current_meds
RETURN d, n, r.severity AS severity, r.description AS description
```

4.3 Qdrant — Vector Similarity Search

Why Qdrant?

Adverse-event similarity cannot be captured by keyword or SQL search. A patient's clinical narrative may reference 'blood thinner' or 'anticoagulant' rather than 'Warfarin'. Their symptoms may partially overlap with cases indexed under different drug names. A vector database encodes the semantic meaning of clinical text into dense embeddings, enabling similarity search across meaning rather than exact string matches.

Qdrant is purpose-built for high-performance approximate nearest-neighbour search with payload filtering, making it ideal for retrieving the top-k most semantically similar FAERS cases filtered simultaneously by drug name, severity, and patient demographics.

Collection Schema & Vector Search Architecture

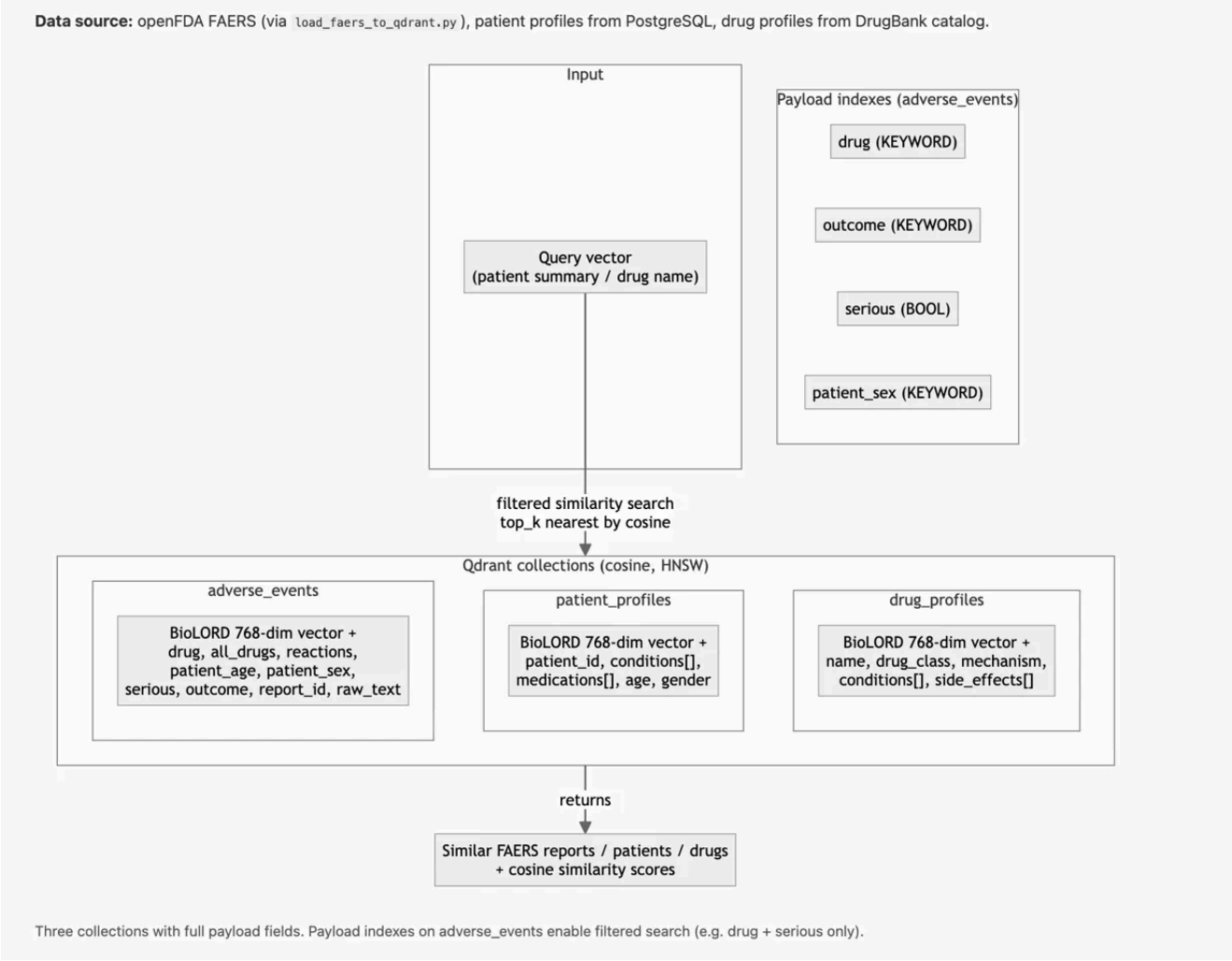


Figure 5 — Qdrant vector search architecture: a BioLORD-encoded query vector performs filtered cosine similarity search across three collections, returning top-k nearest neighbours with cosine similarity scores.

Embedding Model — BioLORD-2023

BioLORD-2023 is a clinical sentence-transformer pre-trained on biomedical ontologies and clinical text corpora. It produces 768-dimensional embeddings optimised for semantic similarity between clinical descriptions, outperforming general-purpose models on medical NLP benchmarks.

Collection Schema — adverse_events

Field	Type / Index	Description
`id`	UUID	FAERS safetyreportid
`vector`	float[768]	BioLORD-2023 embedding of the patient narrative
`drug`	keyword (indexed)	Drug name — payload filter
`outcome`	keyword (indexed)	Hospitalised / Death / Life-threatening / Recovered
`serious`	bool (indexed)	Indexed payload filter
`patient_sex`	keyword (indexed)	Indexed payload filter
`reactions`	text[]	MedDRA reaction terms from the FAERS report

Key Interface Functions

- find_similar_adverse_events_multi_filter(patient_summary, drug_names, top_k) — Multi-filter vector search returning top-k FAERS cases
- analyze_adverse_event_aspects(results) — Aggregates top reactions, severity distribution, and outcome breakdown
- compute_drug_similarity(drug1, drug2) — Cosine similarity between BioLORD embeddings of two drugs
- get_drug_faers_summary(drug, top_k) — FAERS outcome summary for a specific drug
- find_safe_alternatives_candidates(proposed_drug, top_k) — Returns semantically similar drugs as alternative candidates

Multi-Filter Query Example

```
Embedding : BioLORD(patient_summary)
Filter    : drug IN ['Warfarin', 'Aspirin']
          : serious == true
          : patient_sex == 'F'
top_k     : 10 nearest neighbours
```

4.4 MongoDB — Evidence & Audit

Why MongoDB?

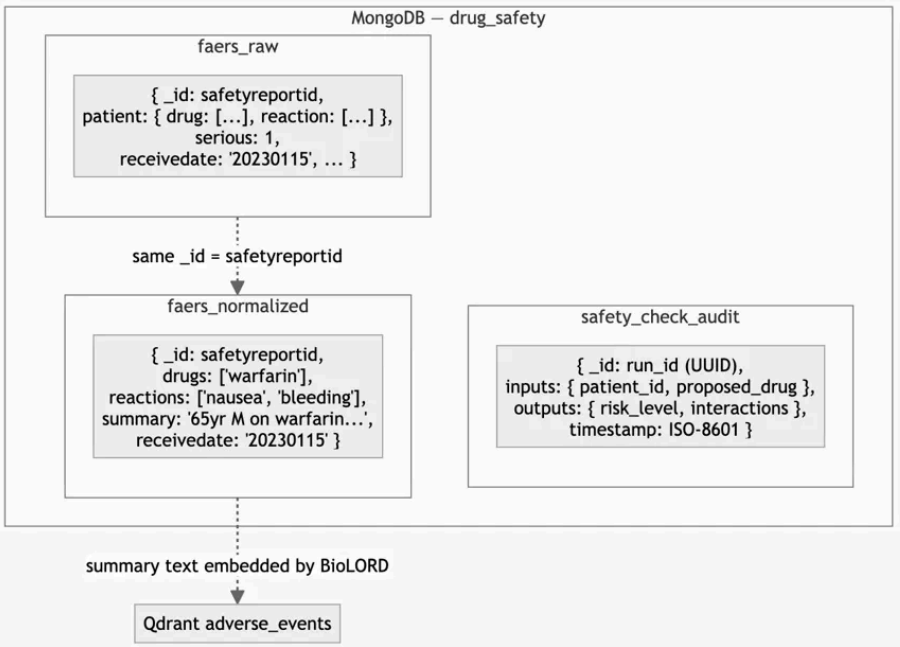
FAERS adverse-event reports are deeply nested JSON documents. A single report may contain a patient object with a variable-length drug array (0–20+ entries), a variable-length reaction array, and a nested sender/receiver structure. Mapping this into a relational model requires multiple tables, JOIN operations on retrieval, and schema migrations whenever the openFDA API format changes.

MongoDB stores each FAERS report as a document, preserving the original nested structure. Retrieval by ID is a direct document fetch with no JOIN overhead. The document model also enables a safety_check_audit collection that stores the full context of each safety-check run as a single document per run.

Collection Schema

Role: Raw FAERS evidence store, normalized summaries for embedding, and audit log of every safety check for traceability.

Database: `drug_safety`. Three main collections.



Collection	Purpose
`faers_raw`	Raw JSON from openFDA Drug Event API; `_id` = safetyreportid. Full traceability — drill down to original FDA report.
`faers_normalized`	Flattened doc (summary text, drugs[], reactions[], dates) used to build Qdrant embeddings and for evidence display. Same `_id` for cross-collection lookup.
`safety_check_audit`	One document per safety-check run: inputs (patient, drug), outputs (risk level, interactions, similar events), timestamp. Audit trail for reproducibility.

Figure 6 — MongoDB drug_safety database: faers_raw stores raw openFDA JSON; faers_normalized stores flattened summaries; safety_check_audit is an append-only log of every safety-check run.

Collection	Purpose
faers_raw	Raw JSON from openFDA Drug Event API; _id = safetyreportid. Full traceability — drill down to original FDA report.
faers_normalized	Flattened doc (summary text, drugs[], reactions[], dates) used to build Qdrant embeddings and for evidence display. Same _id for cross-collection lookup.
safety_check_audit	One document per safety-check run: inputs (patient, drug), outputs (risk level, interactions, similar events), timestamp. Audit trail for reproducibility.

Sample Document Structure (faers_raw)

```
{
  "_id": "20231042",
  "patient": {
    "drug": [{"medicinalproduct": "WARFARIN"},
             {"medicinalproduct": "ASPIRIN"}],
    "reaction": [{"reactionmeddrapt": "BLEEDING"},
                 {"reactionmeddrapt": "BRUISING"}]
  },
  "serious": "1",
  "receivedate": "20231015"
}
```

Key Interface Functions

- get_faers_reports_by_ids(faers_ids, raw=True/False) — Fetches raw or normalised documents by ID list
- log_safety_check(run) — Persists the full safety-check run to the audit collection
- get_safety_check(run_id) — Retrieves a previous run by UUID for reproducibility
- sample_faers_ids(limit) — Returns a random sample of FAERS IDs for demo and testing

5. Application Layer

5.1 Streamlit Demo

The demo application (app/demo.py) provides an interactive interface with five tabs:

- Full Safety Check — Patient and drug dropdowns, unified report with risk level, interaction graph, alternatives, and FAERS evidence
- Patient Data — Patient profile, active medications, medication history, clinical timeline, and analytics
- Drug Knowledge — Polypharmacy cluster analysis, interaction network visualisation, side effects, shared side effects
- FAERS + Alternatives — Qdrant similarity results, DrugBank and NDC alternative candidates
- Evidence & Audit — Fetch FAERS reports by ID, browse the safety-check audit log

5.2 Risk Levels

Level	Display	Trigger Conditions
`Low`	Green banner	No MAJOR or MODERATE interactions detected; low FAERS similarity score
`Moderate`	Yellow banner	MODERATE interactions present; or bridge drug without MAJOR edges
`High`	Red banner	MAJOR interaction detected; bridge drug identified; or high cluster risk score

6. ETL & Data Loading

All data loading is handled by standalone Python ETL scripts. The pipeline must be run in the order shown below:

Script	Source	Target	Notes
`load_synthea_to_pg.py`	Synthea CSVs	PostgreSQL	Loads all EHR tables
`load_demo_patient_to_pg.py`	Demo patient	PostgreSQL	Inserts demo records
`load_sider_to_neo4j.py`	SIDER TSVs	Neo4j	Side-effect edges
`load_drugbank_to_neo4j.py`	DrugBank XML	Neo4j	Drug nodes + INTERACTS_WITH
`load_faers_to_mongo.py`	openFDA FAERS	MongoDB	Raw + normalised docs
`load_faers_to_qdrant.py`	openFDA FAERS	Qdrant	Embeds + upserts
`drugbank_alternatives.py`	DrugBank XML	Cache	Pre-computes alternatives
`openfda_alternatives.py`	NDC / Events	Cache	Fallback alternatives

7. Setup & Run

All four databases are containerised. Start them with Docker Compose, then run the ETL scripts in order, and finally launch the Streamlit demo:

```
# 1. Start all databases
docker compose up -d

# 2. Load EHR data
python etl/load_synthea_to_pg.py
python etl/load_demo_patient_to_pg.py

# 3. Load drug knowledge graph
python etl/load_drugbank_to_neo4j.py
python etl/load_sider_to_neo4j.py

# 4. Load adverse-event data
python etl/load_faers_to_mongo.py --limit 500
python etl/load_faers_to_qdrant.py --limit 500

# 5. Launch demo
streamlit run app/demo.py
```

8. Technologies Used

Technology	Category	Purpose
`Python 3`	Language	Application logic and all ETL scripts
`PostgreSQL`	Relational DB	Patient EHR — structured clinical data
`Neo4j`	Graph DB	Drug interaction network and polypharmacy detection
`Qdrant`	Vector DB	Semantic FAERS similarity search
`MongoDB`	Document DB	Raw FAERS storage and safety-check audit trail
`Streamlit`	UI Framework	Interactive web demo
`BioLORD-2023`	Embedding Model	Clinical sentence-transformer — 768-dim vectors
`Synthea`	Data Generator	Synthetic patient EHR dataset
`openFDA FAERS`	Data Source	Real-world adverse-event reports API
`Docker Compose`	Infrastructure	Containerised database orchestration

9. Conclusion

This project demonstrates that no single database model is sufficient for comprehensive clinical decision support. By combining four specialized systems, each contributing what it does best, the platform delivers a complete, traceable, and evidence-backed drug safety assessment.

- PostgreSQL provides the structured patient state: who the patient is and what they are currently taking
- Neo4j reveals interaction risk at the graph level: which drug combinations collide, and where dangerous clusters form
- Qdrant surfaces real-world evidence: which past FAERS patients experienced similar adverse events
- MongoDB stores the proof and preserves the audit trail: every recommendation is linked to evidence and every run is reproducible

The Streamlit demo provides clinicians and researchers with a practical interface for exploring drug safety, understanding alternatives, and drilling into the underlying evidence, all with full traceability back to the source databases and openFDA FAERS reports.