@workspace I am starting a new project called "GLP-1 Regulatory Intelligence Platform". I need you to act as the Lead Architect and Senior Python Developer. I will provide the full system design below. Your job is to understand the entire architecture, data flow, and constraints, and then help me implement it file-by-file starting with the data models.

1. Project Overview

We are building a Regulatory Intelligence Dashboard for analyzing and comparing the FDA labels of 10 specific GLP-1 drugs (e.g., Ozempic, Mounjaro, Wegovy).

- **Core Value:** Users can view parsed label text, see AI-extracted entities (Dosage, Side Effects), and use a Semantic Search Chatbot (RAG) to ask questions like "Does Ozempic cause thyroid tumors?".
- **Key Constraint:** Reliability and Deployment-Readiness. We use an "Offline-First" ingestion strategy.

2. System Architecture (The "Bakery" Model)

The system is divided into two phases: **Phase A (Ingestion)** and **Phase B (Application)**.

Phase A: The Data Pipeline (Offline/Background)

- **Source:** We manually download 10 raw XML .zip files from FDA DailyMed and upload them to an AWS S3 Bucket (Cold Storage).
- **Ingestion Logic (builder.py):**
    - **Fetch:** Script pulls the raw zip from S3.
    - **Parse:** Uses lxml and specific LOINC codes (e.g., 34067-9 for Indications) to extract clean, verbatim text.
    - **Enrich (NER):** Uses a BioBERT model to extract structured entities (Strength, Route, Frequency) into JSON.
    - **Vectorize:** Chunks the text and generates embeddings using SentenceTransformer for semantic search.
- **Storage (Polyglot Persistence):**
    - **PostgreSQL:** Stores the clean text, metadata, and JSONB NER data. (Primary Key: SetID + Version).
    - **Vector DB (Pinecone/Chroma):** Stores the embeddings with metadata pointing back to the Postgres SetID.

Phase B: The Application (Online/Runtime)

- **Backend:** FastAPI. It is "Read-Heavy."
  - GET /drugs/{id}: Fetches text from Postgres.
  - POST /chat: Performs RAG (Vector Search -> LLM Synthesis).
- **Frontend:** React (Vite) + Tailwind.
  - **Workspace:** A Rich-Text Reader view (not PDF) with a Split-Pane layout for side-by-side comparison.
  - **Sticky Chat:** A floating RAG widget that is context-aware (filters vector search by the currently viewed Drug ID).

3. Implementation Rules (Strict)

1. **Storage:** Use AWS S3 (boto3) for raw files. Use PostgreSQL (via Docker) for app data.
2. **Versioning:** Use an "Append-Only" strategy. If a drug updates, we insert a new row with Version: 2 and archive Version: 1.
3. **Comparison:** * **Lexical (Red-Lining):** Compares the CLEAN TEXT from Postgres (not raw XML).
   a. **Semantic:** Uses Vector similarity.
4. **Tech Stack:** Python 3.10+, FastAPI, Pydantic v2, SQLAlchemy (Async), React, Docker.

4. Immediate Execution Plan

We are starting with **Phase 1: The Blueprint**. I need you to help me set up the project structure and strictly define the Data Models before we write any logic.

Task 1: Create the directory structure:

```text /glp1_project /backend /core (config) /models (pydantic & sqlalchemy) /services (s3, etl, vector) /data (local temp) docker-compose.yml

**Task 2:** Create backend/models/schemas.py. We need Pydantic models to define the shape of our data.

- DrugSection: title (str), text (str), loinc_code (str).
- DrugMetadata: set_id (UUID), name, manufacturer, version (int), upload_date.
- NEREntity: label (str), text (str), confidence (float).
```

- DrugLabel: The master object containing metadata + list of sections + list of entities.

Task 3: Create docker-compose.yml.

- Service 1: postgres:15-alpine with a named volume for persistence.
- Service 2: pgadmin (optional).

Task 4: Create backend/services/s3_client.py.

- A robust class to handle downloading/uploading to the bucket defined in .env.

References :

This is the **Execution Roadmap**. We are moving from "Planning" to "Doing."

Since we are following the **"Reliability-First"** approach, we will not write a single line of Python until the Infrastructure (S3 + Databases) is ready to receive data.

Here is your broad, step-by-step guide to starting the implementation.

Phase 0: The Tooling Setup (Do this First)

Before handling data, you need the tools to hold it.

1. **AWS Account:** Sign up (Free Tier is fine). Create an **S3 Bucket** (e.g., glp1-project-raw-v1).
    a. *Action:* Go to IAM (Identity & Access Management) and create a User with AmazonS3FullAccess.
    b. *Save:* The ACCESS_KEY and SECRET_KEY. You will need these to let your Python script talk to S3.
2. **PostgreSQL Database:**
    a. *Recommendation:* Since you want this "Deployment Ready," use **Docker** on your local machine. It mirrors how it will run in the cloud.

    b. *Alternative:* Use a managed free tier like **Supabase** or **Neon** if you don't want to manage Docker containers yet.
3. **Vector Database:**
    a. *Recommendation:* Sign up for **Pinecone** (easiest for beginners) or **ChromaDB** (runs locally). Pinecone is better for the "Deployment" goal as it persists online.

Phase 1: The "Raw" Data Acquisition

Goal: Get the "Golden Set" of 10 labels safely into the Cloud Archive.

Step-by-Step Actions:

1. **Manual Download (The Source):**
    a. Go to **DailyMed** (dailymed.nlm.nih.gov).
    b. Search for "Ozempic".
    c. Look for the button/tab: **"Download Mapping / SPL Resources"**.
    d. Download the **XML (zip)** file. Do NOT unzip it.
    e. Repeat for all 10 drugs (Mounjaro, Wegovy, etc.).
    f. *Result:* You have 10 .zip files on your laptop.
2. **The S3 Injection (The Archive):**
    a. *Method A (Easy):* Go to the **AWS Console** website → S3 → Your Bucket. Click **"Upload"** and drag-and-drop the 10 zip files.
    b. *Method B (Pro):* Configure the AWS CLI on your terminal (aws configure) and run aws s3 cp ./local_folder s3://your-bucket-name --recursive.
    c. *Result:* Your raw data is now "Cloud Native." It is safe.

Phase 2: The "Builder" Environment

Goal: Set up the Python logic that will read S3 and write to Postgres.

You are not building a "Web App" yet. You are building a **Data Pipeline Script**.

1. **The Project Folder:** Create a folder named glp1_intelligence. inside, create this structure:

Plaintext

```
/glp1_intelligence
  /backend
    /core
      config.py     # Store AWS Keys / DB URLs here
    /etl           # "Extract Transform Load"
      builder.py    # <--- THE ENGINE. This is your main script.
      parser.py     # The lxml logic
      ner.py       # The AI logic
    schemas.py      # The Pydantic blueprints
  /frontend         # Leave empty for now
```

2. **The Pipeline Logic (How it flows):** You will write the builder.py script to do the following (conceptually):
    a. **Connect:** "Hello S3, give me the list of files."
    b. **Download to Memory:** Pull ozempic.zip into RAM (don't save to disk).
    c. **Shred:** Pass the file to parser.py. It returns clean text.
    d. **Analyze:** Pass the text to ner.py. It returns JSON entities + Vectors.
    e. **Push:**
        i. "Hello Postgres, insert this text."
        ii. "Hello Pinecone, insert these vectors."

Phase 3: The Database Schema (The Destination)

Goal: Prepare Postgres to receive the data.

You need to create the Tables *before* running the script.

1. **The drug_labels Table:**
    a. **Columns:**
        i. set_id (Primary Key, String/UUID) - *The Link*.
        ii. version (Integer) - *For history*.
        iii. name (String) - e.g., "Ozempic".
        iv. manufacturer (String).
        v. content_json (JSONB) - *Stores the sections: { "indications": "...", "dosage": "..." }*.

vi. ner_data (JSONB) - *Stores entities: { "strength": "5mg" }*.
vii. is_active (Boolean) - *True for latest version*.

1. Single Label Analysis (The Workspace View)

When the user clicks "Ozempic," they do **not** see a wall of text.

- **The Layout:** We use a **Split-Pane Layout**.
  - **Left Sidebar (Navigation):** A vertical list of sections derived from the LOINC codes (e.g., *1. Indications*, *2. Dosage*, *3. Warnings*, *4. Adverse Reactions*).
  - **Main Content Area (The Reader):** This displays the **Cleaned Text** (from PostgreSQL).
- **The Experience:**
  - When the user clicks "Dosage" on the left, the Main Content automatically scrolls to that specific section.
  - **NER Highlights:** Inside this text, specific entities (like 0.5 mg) are highlighted with small colored "chips" or background colors to make them pop.
- **Data Source:** This view is purely rendering the **JSONB** content stored in PostgreSQL. It is *not* reading XML.

2. The Comparison View (The "Diff" Tool)

This is where your "Red Lining" and "Color Coding" come into play. The user selects two drugs (e.g., Ozempic vs. Mounjaro) and enters **Comparison Mode**.

We display this as **Two Columns Side-by-Side**.

A. Lexical Comparison (The "Red Lining")

- **What is it?** A standard "Word-for-Word" check (like Microsoft Word "Track Changes").
- **Where does it happen?** On the **Cleaned Text Strings** from PostgreSQL.
- **The Visuals:**
  - **Red Strike-through:** Text that exists in Drug A but *not* in Drug B.

    - **Green Underline:** Text that exists in Drug B but *not* in Drug A.
    - **Black:** Text that is identical.
- **Use Case:** Best for comparing **Version 1 vs. Version 2** of the *same* drug (to see exactly what words the FDA changed).

B. Semantic Comparison (The "Traffic Light" Segregation)

- **What is it?** A "Meaning" check. This uses the **Vectors** to calculate similarity scores between paragraphs.
- **Where does it happen?** It overlays color blocks on top of the **Cleaned Text Paragraphs**.
- **The Visuals:**
    - 🟢 **Green Background:** "High Similarity." (e.g., Ozempic says "Nausea"; Mounjaro says "Nausea". The words match or are synonyms).
    - 🟡 **Yellow Background:** "Partial Match." (e.g., Ozempic says "Thyroid Tumors in Rats"; Mounjaro says "Thyroid Tumors in Mice". The concept is similar but details differ).
    - 🔴 **Red Background:** "Unique / Conflict." (e.g., Ozempic says "Inject Weekly"; Rybelsus says "Take Daily". These are fundamentally different concepts).
- **Use Case:** Best for comparing **Drug A vs. Drug B** (Competitor Analysis).

The Golden Rule of this Architecture

AWS S3 is for "Cold Storage" (The Archive). Databases (Postgres/Vector) are for "Live Access" (The App). The Frontend NEVER talks to S3 directly.

1. AWS S3: The "Raw Archive"

- **What do we store here?**
    - We store the **Raw Zipped XML files** exactly as they came from the FDA.
- **Why?**

    - It is our "Source of Truth." If we mess up our parsing logic later, we can delete our database and re-process everything from S3 without needing to download from the FDA again.
- **Format:** Raw .zip or .xml. **NOT JSON.**
- **Access Pattern:**
    - **Who accesses it?** Only the **Backend Python Script** (builder.py).
    - **When?** Only during the **Ingestion Phase** (Offline).
    - **Does the User see this? NO.** The dashboard never reads from S3.

2. The Processing Logic (The "Bridge")

After the file is in S3, your builder.py script wakes up. It reads the XML from S3 into memory. It does **three things** to that data:

1. **Parses Text:** Extracts clear text (Strings) from the XML tree.
2. **Extracts Entities (NER):** Finds "5mg", "Nausea", etc.
3. **Vectorizes (Embeddings):** Converts text chunks into Numbers ([0.1, 0.5...]).

Now, it sends this processed data to **Two Different Databases**.

3. Database A: PostgreSQL (The "Structured Store")

- **What is stored here?**
    - **Metadata:** Drug Name, SetID (Primary Key), Manufacturer.
    - **The Clean Text:** The actual paragraphs of the label (Indications, Dosage) as long strings.
    - **NER Output:** The structured JSON output from your NER model (e.g., {"strength": "5mg", "side_effects": ["Nausea"]}).
- **Why?**
    - This is what populates the **Dashboard** and the **Workspace**.
- **Access Pattern:**
    - **Who accesses it?** The FastAPI Backend (GET /drugs/{id}).
    - **Scenario:** When the user clicks "Ozempic," the API queries **Postgres** to get the text to display on the screen. **We do NOT display raw XML from S3.** We display the clean text from Postgres.

4. Database B: Vector DB (The "Semantic Brain")

- **What is stored here?**
    - The **Vectors** (Embeddings) derived from the text chunks.
    - **Metadata:** drug_id (Foreign Key) and the text_content of that chunk.
- **Why?**
    - This is EXCLUSIVELY for the **Search Bar** and **RAG Chatbot**.
- **Access Pattern:**
    - **Scenario:** User searches "drugs for stomach pain."
    - The system queries the Vector DB to find mathematically similar concepts.

5. The "Link": Primary Key vs. Foreign Key

You asked how we ensure accuracy. We use the **SetID** (The FDA's unique ID for the drug).

- **In PostgreSQL:** SetID is the **Primary Key**.
    - Row: ID: 123 | Name: Ozempic | Text: "..."
- **In Vector DB:** SetID is stored as **Metadata (Foreign Key)** attached to every single vector.
    - Vector: [0.1, 0.9...] | Metadata: { "drug_id": "123", "chunk_id": 1 }

How RAG uses this:

1. User acts on **Ozempic** (ID: 123).
2. User asks a question.
3. Backend searches Vector DB: *"Find vectors close to this question, BUT...* **FILTER WHERE drug_id == 123**."
4. This guarantees the AI never reads Mounjaro's data while you are looking at Ozempic.

Detailed Scenario Walkthrough

Let's trace the exact flow so you are 100% confident.

Step 1: The User Opens the Dashboard

- **User sees:** A list of 10 drugs (Ozempic, Mounjaro, etc.).
- **Backend Action:** SELECT name, manufacturer, set_id FROM drugs_table; (PostgreSQL).
- **S3 Usage:** None.
- **Vector DB Usage:** None.

Step 2: The User Clicks "Ozempic" (The Workspace)

- **User sees:** The full label text, divided into tabs (Indications, Dosage). They see "0.5mg" highlighted (NER data).
- **Backend Action:** SELECT content, ner_data FROM drugs_table WHERE set_id = '123'; (PostgreSQL).
- **Clarification:** We display the **Clean Text** from Postgres. We do *not* fetch the raw XML from S3. The XML is ugly and hard to read. The Postgres text is already cleaned/stripped of HTML tags during the processing phase.

Step 3: The User Asks a Question (The Chat)

- **User types:** "Does it cause cancer?"
- **Backend Action:**
  - Convert question to Vector.
  - Query **Vector DB**: Find chunks matching "cancer" where drug_id = '123'.
  - Send those chunks to LLM.
  - Return answer.

1. The Revised Storage Strategy (Cloud-Native)

Instead of a data/raw folder on your laptop, we will use **AWS S3 Buckets**.

- **Bucket Name:** glp1-raw-labels
- **Structure:**

Plaintext

```
s3://glp1-raw-labels/
  /ozempic/
    v1_2024.zip
    v2_2025.zip  <-- (Current)
  /mounjaro/
    v3_2025.zip
```

Why this is professional:

1. **Accessible Anywhere:** Your deployed API (running on a server) and your local Dev environment can both access the *exact same files*.
2. **Versioning:** S3 has built-in versioning. You never accidentally lose an old label.

2. The Automated "Watchdog" Pipeline (Deep Dive)

You asked: *How do we replace older versions without user interaction? Is it complicated?*

The Answer: It is **not complicated** if we follow a strict "Check-Compare-Swap" logic. We write a single script (watchdog.py) that runs automatically (e.g., once every 24 hours).

Here is the exact logic flow:

Step A: The Poll (The "Check")

The script wakes up. It holds a list of the 10 drugs we care about (specifically their **SetIDs**).

- **Action:** It loops through the list and hits the FDA DailyMed API history endpoint.
  - *Request:* GET /spls/{set_id}/history
  - *Response:* "The latest version for Ozempic is **Version 5**, published **Jan 20, 2026**."

Step B: The Comparison (The "Detect")

The script looks at your **Postgres Database**.

- *Query:* SELECT version FROM drugs WHERE name = 'Ozempic'
- *Result:* "We currently have **Version 4**."

- **Logic:** Version 5 > Version 4. **UPDATE REQUIRED.**

Step C: The "Silent Swap" (The Execution)

This happens entirely in the background. The user sees nothing until it is finished.

1. **Download:** The script downloads the new Version 5 zip file from the FDA.
2. **Upload to S3:** It saves it to s3://glp1-raw-labels/ozempic/v5.zip.
3. **Process (The Oven):** The script triggers the parsing logic (just like we did manually on Day 1).
    a. Parses XML -> Extracts NER -> Creates Vectors.
4. **Atomic Update (The Switch):** This is the crucial part. We update the database records.
    a. *Postgres:* Update the metadata rows with the new text.
    b. *Vector DB:* **Delete** old vectors for Ozempic and **Insert** the new ones.
    c. *Status:* Update the "Last Updated" timestamp.

Is it complicated? No. It is just a function that runs in a loop.

- **User Interaction:** None.
- **Downtime:** None. The old data remains available until the millisecond the new transaction commits.

3. The Revised Implementation Path

Since we are now using S3, your "Starting Point" shifts slightly. You need to set up the plumbing first.

Step 1: The Setup (Infrastructure)

- **AWS:** Create an S3 bucket (e.g., my-pharma-project-raw). Get your AWS_ACCESS_KEY and AWS_SECRET_KEY.
- **Local Env:** Configure your project to talk to S3 using the boto3 library (standard Python SDK for AWS).

Step 2: Manual Ingestion (Upload to Cloud)

- **Action:** Download the 10 zip files manually (as discussed).
- **Change:** Instead of saving them to a local folder, write a small script (upload_initial.py) to upload them to your S3 bucket.
- *Why?* This mimics the production environment from Day 1.

Step 3: The Blueprint (schemas.py)

- (Same as before) Define your data models.

Step 4: The Processor (builder.py)

- **Change:** Modify the script so it reads **from S3**, not from your hard drive.
  - *Code:* s3_client.download_file(...) → Parse → Push to DB

The User Journey (The "Story")

The User: Jane, a Regulatory Intelligence Analyst. **The Goal:** Compare the safety profiles of **Ozempic** (Novo Nordisk) and **Mounjaro** (Eli Lilly).

Step 1: The Global Dashboard (Instant Access)

- Jane logs in. She sees a clean table listing the 10 monitored GLP-1 drugs.
- **Visuals:** Each row shows the Drug Name, Manufacturer, and a green badge: *“Latest Update: Jan 15, 2026 (Version 3)”*.
- **Action:** She clicks on **"Ozempic"**.

Step 2: The Workspace (Deep Analysis)

- The screen transitions to the **Single Drug View**.
- **Layout:**
  - **Left Sidebar:** Navigation menu (1. Indications, 2. Dosage, 3. Warnings...).

    - **Center:** The cleaned, easy-to-read label text (served from PostgreSQL). It looks like a modern document, not a PDF.
    - **Highlights:** She sees small blue chips floating over text like *"0.25 mg"* or *"Nausea"*. These are the **NER Entities** our AI extracted automatically.
- **Action:** She wants to check for thyroid risks, but doesn't want to read 50 pages.

Step 3: The Sticky Chat (Semantic Search)

- She clicks the **Floating Chat Icon** (bottom-right).
- **Query:** *"Does this drug have a boxed warning for thyroid tumors?"*
- **The System (RAG):**
    - Searches the **Vector Database** specifically for Ozempic chunks.
    - Finds the "Boxed Warning" section.
    - **AI Response:** *"Yes. Ozempic causes thyroid C-cell tumors in rodents. It is contraindicated in patients with a family history of MTC."*
    - **Citation:** The AI provides a clickable link. Jane clicks it, and the center screen auto-scrolls to the exact paragraph in the "Warnings" section.

Step 4: The Comparison (Side-by-Side)

- Jane returns to the Dashboard and selects **"Compare"**. She picks **Ozempic** (Left) and **Mounjaro** (Right).
- **Lexical View (Red-Lining):** She sees the text of both. Words present in Ozempic but missing in Mounjaro are highlighted in **Red**; new words in Mounjaro are **Green**.
- **Semantic View:** She switches to "Concept Mode."
    - The "Adverse Reactions" sections for both align perfectly side-by-side.
    - The system highlights "Nausea" in Green on both sides (indicating a match), even though Ozempic writes "nausea" and Mounjaro writes "nausea and vomiting"

The Tech Stack & Component Guide

This is the inventory of tools you need to install.

1. The Infrastructure (The Foundation)

- **AWS S3 (Boto3): The Raw Archive.**
    - *Usage:* Stores the original .zip files downloaded from FDA.

    - *Why:* Safety. If we break the database, we rebuild from S3.
- **Docker: The Container Engine.**
    - *Usage:* Runs the PostgreSQL database cleanly on your laptop without "installing" it in Windows/Mac.
    - *Why:* Deployment readiness.

2. The Databases (Polyglot Persistence)

- **PostgreSQL (v15+): The Structured Store.**
    - *Usage:* Stores the Clean Text (for the dashboard), Metadata (SetIDs), and NER JSON.
    - *Why:* We need ACID compliance and reliability for the main app data.
- **Pinecone (or ChromaDB): The Vector Store.**
    - *Usage:* Stores the Embeddings (Numbers) for the Chatbot.
    - *Why:* Standard SQL databases cannot perform "Semantic Search" (finding similar meanings).

3. The Processing Core (Python)

- **lxml: The Parser.**
    - *Usage:* Cuts the XML tree into sections using LOINC codes.
    - *Why:* Faster and more accurate than standard Regex or BeautifulSoup for XML.
- **HuggingFace Transformers (BioBERT): The Extractor.**
    - *Usage:* Named Entity Recognition (NER) to find specific medical variables.
- **SentenceTransformers: The Embedder.**
    - *Usage:* Converts text to vectors (model.encode(text)).

4. The Application Layer

- **FastAPI: The Backend API.**
    - *Usage:* Connects the Frontend to the Databases. Handles the "Chat" streaming.
- **React + Tailwind CSS: The Frontend UI.**
    - *Usage:* The Dashboard, Split-Pane view, and Chat Widget.

The Implementation Guide (Step-by-Step)

Follow this sequence strictly to avoid backtracking.

Phase 1: Setup & Infrastructure (Day 1)

1. **Project Init:** Create the folder structure (backend/, frontend/, data/).
2. **AWS S3:** Create a bucket. Manually upload your 10 FDA Zip files.
3. **Docker:** Create docker-compose.yml.

```yaml
YAML
services:
 db:
  image: postgres:15-alpine
  volumes:
   - ./postgres_data:/var/lib/postgresql/data
  ports:
   - "5432:5432"
```

Run docker-compose up -d. You now have a database.

Phase 2: Data Modeling (Day 2)

- **Goal:** Define the "Contract."
- Create backend/schemas.py.
    - Define DrugMetadata (name, set_id, version).
    - Define DrugSection (title, content_text).
    - Define DrugNER (strength, route).

Phase 3: The ETL Pipeline (Day 3-4)

- **Goal:** Move data from S3 → Postgres/Vector DB.
- Create backend/builder.py.
    - **Fetch:** Connect to S3 (boto3) and download ozempic.zip to memory.

    - **Parse:** Use lxml to extract text. *Validation:* Print the text to console to ensure it's clean.
    - **Vectorize:** Generate embeddings for the text.
    - **Load:** Connect to Postgres (using SQLAlchemy) and Pinecone. Insert the data.
    - *Result:* Your database is now populated with real data.

Phase 4: The Backend API (Day 5)

- **Goal:** Expose the data.
- Create backend/main.py.
    - GET /drugs: Select * from Postgres.
    - GET /drugs/{id}: Return the full text for the Workspace.
    - POST /chat: Receive query → Pinecone Search → LLM API → Return Answer.

Phase 5: The Frontend (Day 6-7)

- **Goal:** Visuals.
- Initialize React (npm create vite@latest).
- **Dashboard:** Fetch list from API.
- **Workspace:** Render the HTML text. Use divs for sections.
- **Chat:** Add the floating button component.

Phase 6: The Watchdog (Bonus/Final)

- Create watchdog.py.
- Script checks FDA API for new versions. If found, it triggers builder.py to append the new version to the DB.