PharmaPulse-ET: Building an End-to-End Analytical Platform for Telegram Medical Data

Author: Miheret Girmachew

Date: July 15, 2025

GitHub Repository: https://github.com/Miheret-Girmachew/PharmaPulse-ET

1. Introduction: The Business Need

In Ethiopia's rapidly growing digital landscape, public Telegram channels have become vital hubs for commerce, including the exchange of information about medical products and pharmaceuticals. For a data science company like Kara Solutions, this unstructured data represents a rich, untapped resource. The challenge, however, is that this data is noisy, unstructured, and distributed across numerous channels.

This project addresses this challenge by building a robust, end-to-end data platform. The goal is to ingest raw data from public Telegram channels, process it through a modern ELT pipeline, enrich it with computer vision, and expose the final, cleaned insights through an analytical API. This platform will empower analysts to answer critical business questions, such as identifying top-selling products, tracking market trends, and understanding channel activity.

2. Final Data Architecture

The entire system was designed using a modern, layered data architecture, ensuring scalability, reliability, and reproducibility. The data flows through several distinct stages, from raw ingestion to a final, query-optimized format.

Diagram Description:

- **Extract & Load:** A Python script using the Telethon library scrapes messages and images from specified Telegram channels, loading them into a local file system that acts as our Data Lake. This preserves the raw data as our source of truth.
- **Load to Warehouse:** A second Python script reads the raw JSON files from the Data Lake and loads them into a raw schema in a PostgreSQL database, our data warehouse.
- **Transform & Model:** dbt (Data Build Tool) connects to the warehouse. It transforms the raw data into clean staging models and then builds a final, analytics-ready star schema in a marts schema.
- **Enrich:** A Python script leveraging a pre-trained YOLOv8 model scans for new images, performs object detection, and writes the results (detected classes and confidence scores) to an enrichments table in the database.
- **Serve:** A FastAPI application queries the final marts tables to provide high-performance, documented analytical endpoints for end-users or other services.
- **Orchestrate:** Dagster manages the entire workflow, defining the dependencies between each step, and providing a UI to schedule, execute, and monitor the pipeline from start to finish.

3. Data Model: The Star Schema

To optimize for analytical queries, we modeled our data warehouse using a classic star schema. This design separates our data into central "fact" tables containing key events (messages, image detections) and surrounding "dimension" tables containing descriptive attributes (channels, dates).

- **fct_messages (Fact Table):** The core of our model. Each row represents a single message posted to a channel. It contains foreign keys to the dimension tables and key metrics like view_count and message_length.
- **dim_channels (Dimension Table):** Contains information about each unique Telegram channel, such as its name and a unique surrogate key (channel_key), generated using dbt_utils.
- **dim_dates (Dimension Table):** A standard date dimension table, pre-populated with dates and attributes like year, month, and day of the week, allowing for easy time-based analysis.
- **fct_image_detections (Fact Table):** A secondary fact table linked to fct_messages that stores the results from the YOLO object detection, connecting visual content directly to the original message.

4. Technical Implementation & Choices

Technical Choice (Docker)

We containerized the entire application (Python, PostgreSQL, all dependencies) using Docker and Docker Compose. This choice was critical for guaranteeing a consistent, reproducible environment. It eliminated "it works on my machine" problems and streamlined the complex dependency management required by libraries like PyTorch and OpenCV.

Implementation: We created a Dockerfile for our Python environment, a docker-compose.yml to define and link our application and database services, and a .env file (kept out of Git via .gitignore) to manage secrets like API keys and database passwords securely.

Technical Choice (ELT over ETL, dbt)

We chose a modern ELT (Extract, Load, Transform) approach. Raw data is loaded into the warehouse first, and all transformations happen inside the database using dbt. This is more scalable than traditional ETL because it leverages the power of the data warehouse for transformations. dbt was chosen for its ability to build modular, testable, and well-documented SQL-based data models.

Implementation: A Telethon script scraped the data into a data lake. A Python script loaded this into a raw.telegram_messages table. Our dbt project then executed a series of models:

- **stg_telegram_messages.sql:** Cleaned and cast data types from the raw JSONB.
- **dim_channels.sql & dim_dates.sql:** Created our dimension tables.
- **fct_messages.sql & fct_image_detections.sql:** Joined the staging models to build our final fact tables.

Testing: We implemented built-in (unique, not_null) and custom SQL tests to validate our data's integrity. For example, a custom test ensured view_count is always positive, and built-in tests validated the referential integrity between fact and dimension tables.

Technical Choice (YOLOv8)

To enrich our dataset with insights from images, we used YOLOv8, a state-of-the-art, pre-trained object detection model. It was chosen for its high accuracy and performance, allowing us to quickly integrate computer vision without needing to train a model from scratch.

Implementation: A Python script scans the image data lake for unprocessed images. For each new image, it runs the YOLOv8 model, extracts detected objects (e.g., "bottle", "person") and their confidence scores, and inserts these findings into a new enrichments.image_detections table, linking them back to the original message ID. This step required careful dependency management within the Dockerfile to install system libraries like libgl1-mesa-glx and libglib2.0-0 needed by OpenCV.

Technical Choice (FastAPI)

FastAPI was chosen to build our analytical API due to its high performance, automatic data validation with Pydantic, and automatic generation of interactive API documentation (via Swagger UI). This allows for rapid development and easy consumption by other services or users.

Implementation: We created several endpoints that directly query our dbt-built marts tables to answer the key business questions, using SQLAlchemy for database communication.

Technical Choice (Dagster)

A collection of scripts is not a production pipeline. We chose Dagster to orchestrate our workflow because of its excellent local development experience, its focus on data-awareness, and its clear UI for monitoring and scheduling.

Implementation: We defined each step of our pipeline (scrape, load, dbt, enrich) as a Dagster "op". These ops were combined into a "job" (end_to_end_telegram_pipeline), which defines the dependencies and execution order. We also configured a daily schedule to run the entire pipeline automatically, making the system a true, automated data product.

5. API Endpoints Demonstration

The final data product is exposed through a set of analytical endpoints, demonstrated below via curl.

Endpoint: `GET /api/channels/{channel_name}/activity`

This endpoint queries the data warehouse to return key activity metrics for a specified channel.

```
# Request
curl -X GET "http://localhost:8000/api/channels/CheMed123/activity"
```

```
// Response
{
  "channel_name": "CheMed123",
  "message_count": 472,
  "first_message_date": "2025-07-23T10:00:00Z",
  "last_message_date": "2025-07-23T14:30:00Z"
}
```

6. Difficulties & Key Takeaways (Reflection)

This project was a comprehensive journey through the modern data stack, and several challenges and learnings emerged:

Biggest Challenge: The initial environment setup with Docker was the most significant hurdle. It required resolving a cascade of issues, from low-level Docker engine connection problems on Windows to port conflicts (port is already allocated) and complex dependency management. Specifically, resolving the torch download hash mismatches and the "dependency whack-a-mole" for OpenCV (the missing libGL.so.1 and libgthread-2.0.so.0 libraries) required a deep dive into Docker image building and system dependencies. Overcoming this provided a profound appreciation for the power and complexity of containerization.

Key Takeaway 1 (The Power of Layering): The layered architecture (Raw -> Staging -> Marts) was incredibly valuable. When I discovered a SQL syntax error in my dim_dates model (dayofweek vs. dow), I only had to modify that single dbt model. The raw data remained untouched and the rest of the pipeline was unaffected. This modularity makes the system robust and easy to maintain.

Key Takeaway 2 (Testing is Non-Negotiable): Implementing dbt tests immediately caught data quality issues and validated my logic. Seeing the relationships test pass gave me confidence that my star schema joins were correct. It proved that a data pipeline without automated tests is fundamentally untrustworthy.

Key Takeaway 3 (Models are only as good as their training data): A fascinating insight came from the YOLOv8 enrichment results. The pre-trained model identified objects like "clock" and "hot dog" in images of medical products. This was a powerful, practical lesson that a general-purpose model is not sufficient for a specialized domain. It highlights the importance of fine-tuning models on domain-specific data for production use cases.

If I Had More Time:

- **Improve Data Enrichment:** I would implement a more sophisticated Natural Language Processing (NLP) model to perform Named Entity Recognition (NER). This would allow the pipeline to accurately extract specific drug names (e.g., "Paracetamol 500mg"), dosages, and company names from message text, providing far more valuable insights than simple keyword counts.
- **Deploy to the Cloud:** I would deploy the entire stack to a cloud provider like AWS. This would involve using S3 for the data lake, RDS for the PostgreSQL warehouse, and ECS/Fargate for running the Docker containers (Dagster, FastAPI). This would transform the project from a local prototype into a scalable, production-ready system.

7. Conclusion

The PharmaPulse-ET project successfully demonstrates the construction of a complete, end-to-end data platform. By leveraging tools like Docker, Telethon, PostgreSQL, dbt, YOLO, FastAPI, and Dagster, we were able to transform chaotic, raw Telegram data into a structured, enriched, and accessible data product. This platform provides Kara Solutions with a powerful, automated foundation for generating actionable insights into the Ethiopian medical market, proving that modern data engineering practices can unlock immense value from unconventional data sources.