Smart Shelves

Real-time Inventory Monitoring & Warehouse Optimization

Big Data Technologies Project 2025

PRESENTED BY

Chiara Belli matr. 258466 chiara.belli@studenti.unitn.it
Camilla Bonomo matr. 255138 camilla.bonomo@studenti.unitn.it
Enrico Maria Guarnuto matr. 255195
enricomaria.guarnuto@studenti.unitn.it

Abstract (overview)

THE PROJECT

The Smart Shelves project presents an end-to-end, event-driven Big Data architecture aimed at monitoring supermarket inventory in real time and coordinating operational decisions between retail stores and warehouses.
The system simulates realistic data sources such as shelf sensors, customer foot traffic, point-of-sale (POS) transactions, and warehouse movements and processes them through a scalable streaming infrastructure.

KEY GOALS

- Design a stateful real-time system capable of tracking stock levels and batch information across store and warehouse.
- Demonstrate the integration of streaming technologies for event ingestion, processing, and state management.
- Implement business-driven alerting and optimized replenishment logic based on inventory conditions and ML model predictions.
- Showcase how modern Big Data architectures can support operational decision-making in retail scenarios.

OUTPUTS

- A fully containerized end-to-end streaming pipeline based on Kafka and Spark Structured Streaming.
- Real-time inventory states for shelves and warehouses.
- Automatically generated alerts and restocking plans optimized by the ML model.
- Persistent storage of raw events, processed states, and operational outputs using Delta Lake and PostgreSQL.

Problem statement and motivation

- **What problems are we solving?**
  - Supermarkets need timely, trustworthy visibility on shelf availability (to avoid stockouts) and batch expirations (to avoid waste and compliance issues).
  - Data arrives from heterogeneous sources (shelf sensors, POS, foot traffic, warehouse operations) and must be fused into a consistent, queryable operational state.

- **Why is this important?**
  - Stockouts drive lost sales and poor customer experience; overstocking increases holding costs and spoilage risk.
  - Expiration-aware replenishment reduces food waste and improves FIFO operations.
  - A streaming architecture enables faster reactions than periodic batch updates.

Data exploration

Since, for privacy reasons, it was not possible to access real supermarket databases, it was necessary to manually generate the inventory datasets, historical data of 1 year of sales, and the sensor simulators. Specifically:

Store inventory table: 1,142 rows, max stock avg 48.13, current stock avg 38.85, visibility range 0.05–0.20 (data/store_inventory_final.parquet).

Warehouse inventory table: 1,142 rows, warehouse capacity avg 306.23 units (data/warehouse_inventory_final.parquet).

Batch-level tables: 1,830 store batches and 2,250 warehouse batches (includes expiry dates) (data/store_batches.parquet, data/warehouse_batches.parquet).

Weekly discounts table: 9,360 rows with discount_pct range 0.0501–0.35 (data/all_discounts.parquet).

Total simulated sales: 2,808,135 units across 2025 (data/sim_out/synthetic_1y_panel.csv).

Shelf Sensors
Pick-up, put-back, weight change events

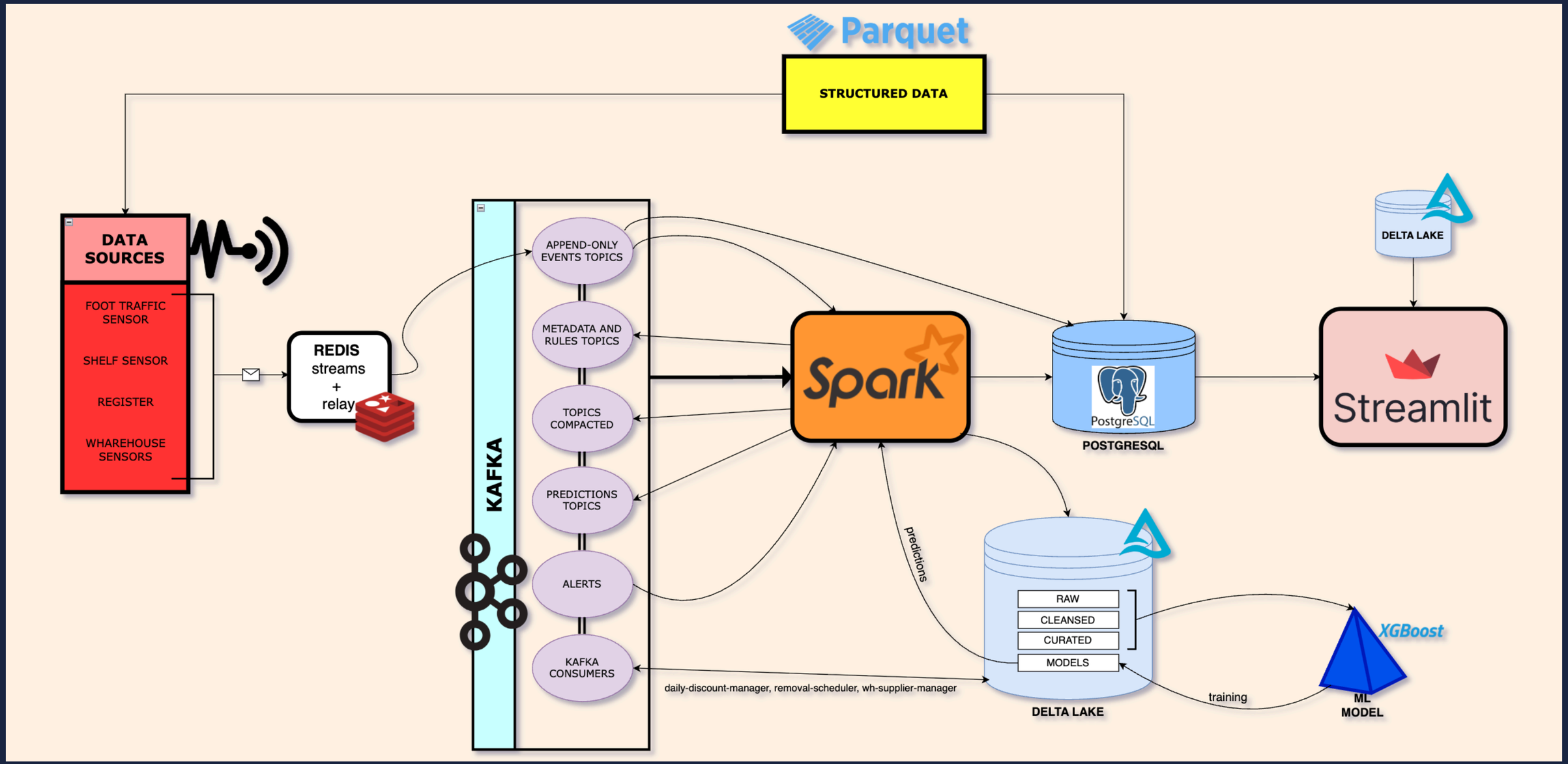
Foot Traffic Sensors
Customer entry, exit, and session duration

POS Transactions
Sales events linked to customers and products

Warehouse Events
Inbound and outbound stock movements

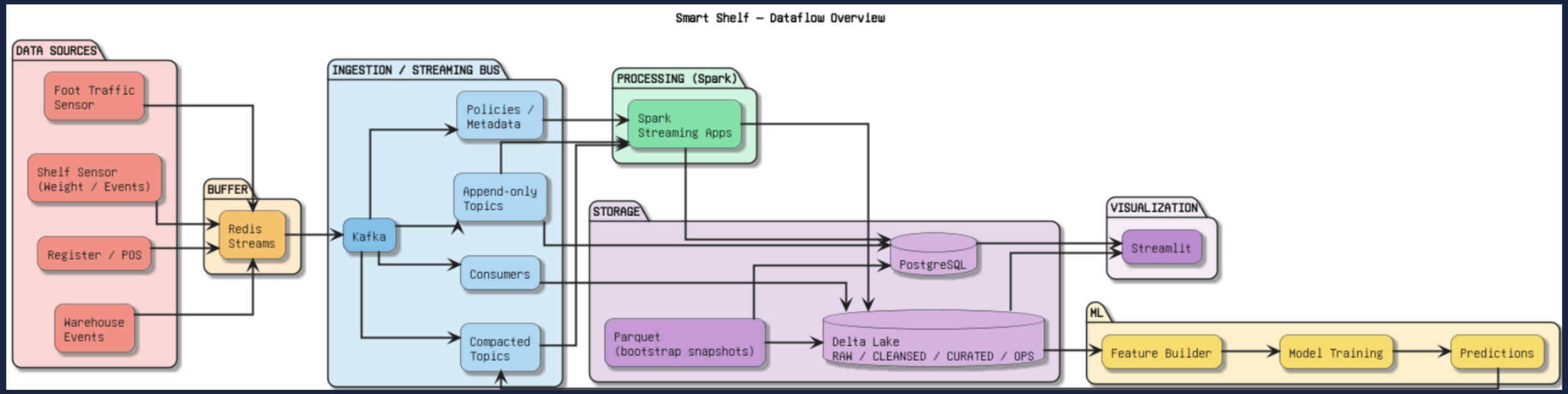
Although synthetic, the data generation process preserves realistic temporal patterns and inter-dependencies between events. Moreover, the simulation logic and datasets broadly follow the information provided to us through a conversation with the Conad of Scandiano regarding the operation and organization of the store.

System Architecture Diagram



Data Flow in the Smart Shelf System

UNDERSTANDING THE JOURNEY OF DATA PROCESSING



Technologies and justification

CORE TECHNOLGIES	
Python — Main programming language	Chosen for its flexibility and rich data ecosystem; used across the project for data generation, Kafka producers, Spark job logic, ML services, and the Streamlit dashboard.
Docker / Docker Compose — Containerization and orchestration	Chosen to ensure reproducible environments, isolate dependencies (Kafka, Postgres, Spark, etc.), and start/stop the entire stack with a single `docker compose` command.

REAL-TIME STREAMING & INGESTION	
Apache Kafka	Chosen as the central event bus to decouple producers and consumers, support high-throughput ingestion, and model both append-only event streams and compacted state/metadata topics.
Apache ZooKeeper — Kafka coordination	Chosen because the Kafka distribution used in this project relies on ZooKeeper for broker metadata/coordination in a lightweight, single-node setup.
Redis Streams	Chosen to absorb bursts and provide a simple backpressure layer for the simulated producers, reducing coupling to Kafka availability and smoothing event publication.

Technologies and justification

STREAM PROCESSING & LAKEHOUSE STORAGE	
Apache Spark (Structured Streaming)	Chosen for scalable micro-batch streaming, stateful aggregations, windowing, and fault-tolerance via checkpoints; used to build shelf/warehouse states, generate alerts, and compute replenishment/supplier plans.
Delta Lake	Chosen to persist raw/cleansed/ops/analytics layers with ACID guarantees, schema evolution support, and reliable streaming sinks/checkpoints on a local filesystem (`./delta`).
Apache Arrow / Parquet (pyarrow)	Parquet is the main file format for bootstrap data, and pyarrow keeps Parquet/Arrow I/O fast and compatible across Spark, Delta, and Python.

DATABASES	
PostgreSQL 16 — Relational database	Chosen for reliability and strong SQL support; used for reference snapshots, configuration/policies, materialized state mirrors, ops tables, and analytics that are easy to query/serve.

VISUALIZATION	
Streamlit — Interactive dashboard	Chosen to quickly build a data-driven UI to inspect pipeline outputs (states/alerts/analytics) without the overhead of a separate frontend stack.

MACHINE LEARNING	
XGBoost — Gradient-boosted trees	Chosen for strong performance on tabular data and efficient training/inference for demand forecasting.
scikit-learn — Baseline ML + preprocessing	Chosen for standardized preprocessing utilities and simple, dependable baseline models used in the demand-forecasting component.

```
.
├── docker-compose.yml                # Full stack (Kafka/Redis/Postgres/Kafka Connect/Spark
├── conf/                             # Spark config (spark-defaults.conf)
├── data/                             # Synthetic datasets + generators + CSVs for Postgres b
│   ├── create_db.py
│   ├── create_simulation_data.py
│   ├── create_discounts.py
│   ├── store_inventory_final.parquet
│   ├── warehouse_inventory_final.parquet
│   ├── store_batches.parquet
│   ├── warehouse_batches.parquet
│   ├── all_discounts.parquet
│   ├── sim_out/                      # Generated simulation outputs
├── delta/                            # Local Delta Lake (raw/cleansed/ops/analytics + checkp
├── dashboard/                        # Streamlit dashboard (map, alerts, state, ML)
│   ├── app.py
│   ├── store_layout.yaml
│   └── assets/
├── images/                           # Logo + diagrams + dashboard screenshots
├── kafka-components/                 # Kafka components (topics, connect, producers)
│   ├── kafka-init/                   # Create topics (append-only + compacted)
│   ├── kafka-connect/                # Kafka Connect + JDBC connector init
│   ├── kafka-producer-foot_traffic/  # Foot traffic producer (session + realistic)
│   ├── kafka-producer-shelf/         # Producer shelf events (pickup/putback/weight_change)
│   ├── kafka-producer-pos/           # POS transactions producer (from foot traffic + shelf
│   ├── shelf-daily-features/         # Kafka-based service for daily ML features
│   ├── daily-discount-manager/       # Kafka-based near-expiry discounts
│   ├── removal-scheduler/            # Kafka-based expired batch removal
│   └── wh-supplier-manager/          # Kafka-based supplier orders/receipts + wh_in
├── ml-model/                         # ML services (training + inference)
│   ├── training-service/             # Train XGBoost, register in Postgres, write artifacts
│   └── inference-service/            # Cutoff-time inference (Sun/Tue/Thu), write pending wh
├── models/                           # Model artifacts + feature columns (mounted to /models
├── postgresql/                       # DDL + init SQL (ref/config/state/ops/analytics + view
│   ├── 01_make_db.sql
│   ├── 02_load_ref_from_csv.sql
│   ├── 03_load_config.sql
│   └── 05_views.sql
├── simulated_time/                    # Shared simulated clock (Redis)
├── spark-apps/                        # Spark Structured Streaming jobs
│   ├── deltalake/                     # Bootstrap empty Delta tables
│   ├── foot-traffic-raw-sink/         # Kafka foot_traffic -> Delta raw
│   ├── shelf-aggregator/              # shelf_events -> raw + shelf_state (Delta + Kafka comp
│   ├── batch-state-updater/           # pos_transactions -> shelf_batch_state (Delta + Kafka
│   ├── shelf-alert-engine/            # shelf_state -> alerts + shelf_restock_plan
│   ├── shelf-restock-manager/         # shelf_restock_plan -> wh_events (FIFO picking)
│   ├── shelf-refill-bridge/           # wh_events (wh_out) -> shelf_events (delay)
│   ├── wh-aggregator/                 # wh_events -> wh_state
│   ├── wh-batch-state-updater/        # wh_events -> wh_batch_state (+ mirror store batches)
│   ├── wh-alert-engine/               # wh_state -> alerts + wh_supplier_plan
│   └── alerts-sink/                   # alerts -> Delta (+ optional Postgres)
```

Implementation and code repository

REPOSITORY LINK:
https://github.com/EnricoMGuarnuto/Big_Data_Project

HOW TO RUN:

- Clone the repo from GitHub: **git clone <https://github.com/EnricoMGuarnuto/Big_Data_Project.git>**

- Once the repository is cloned, it is required to run the two main files that create the simulation data:

  python create_db.py
  python create_discounts.py
  python create_simulation_data.py

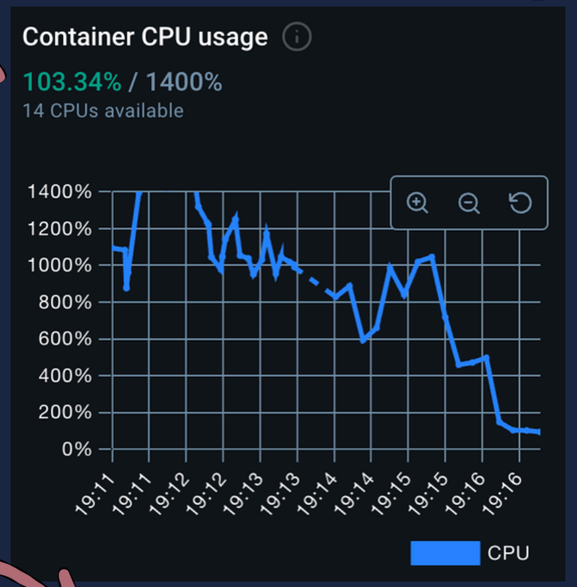
- Start the Stack: **docker compose up -d --build**
- In the case of limited RAM and CPU availability, it is recommended to run **docker compose up -d --build $(docker compose config --services | grep -v '^kafka-producer')** to start the program without the simulators. Wait 5–8 minutes, then run: **docker compose up -d --build $(docker compose config --services | grep '^kafka-producer')** to activate the simulators.

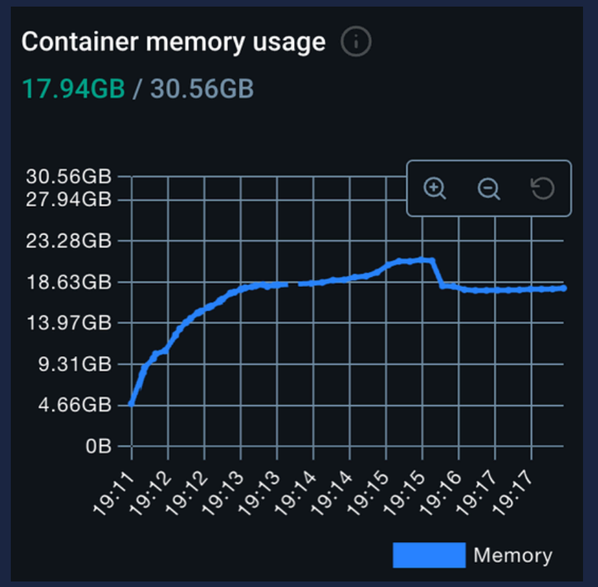
- Check out the streamlit dashboard by connecting to: **http://localhost:8501**

Check out the full README.md document on GitHub to get more info on additional ways you could run the project

Results and performance

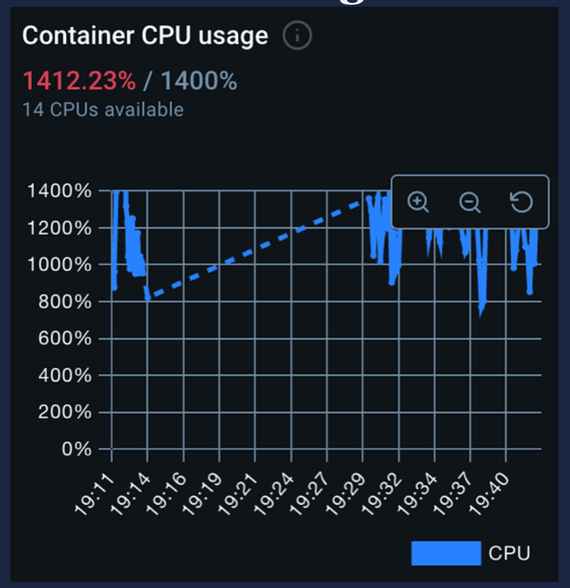
CPU and RAM usage without data simulators

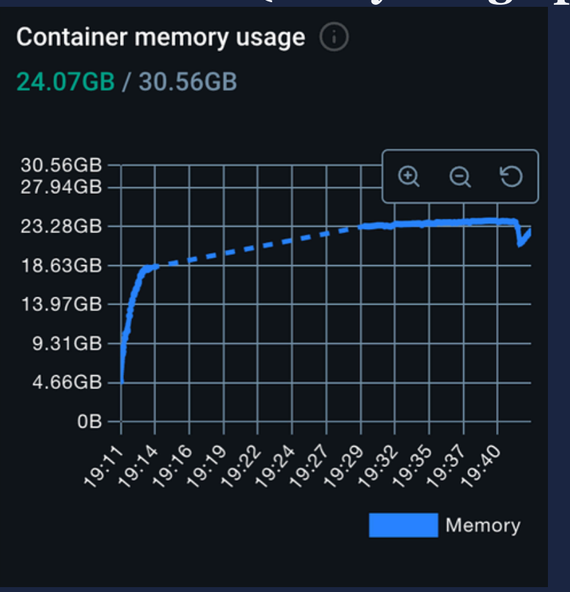




In the CPU and RAM charts without Kafka simulators/producers, you can see a typical warm-up followed by steady-state behavior. The CPU shows a very high initial spike during startup/initialization and the first processing phases, then it settles down and drops to much lower values. RAM usage grows quickly and then stabilizes indicating a fairly stable working set once the job is fully loaded.

CPU and RAM usage with data simulators (everything up)





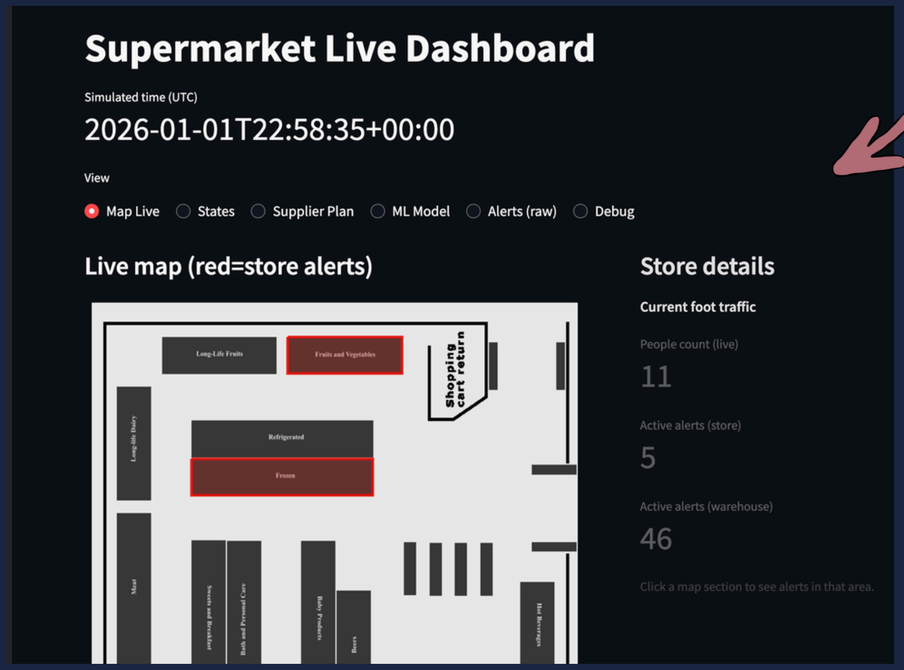
In the CPU and RAM charts for the full stack **with data simulators** enabled, the load is higher and more persistent. The CPU often stays close to saturating all 14 cores showing that, there is constant competition for compute resources. RAM usage also stabilizes at a noticeably higher level, roughly +6 GB, which is consistent with increased buffering/queues (producers and brokers), more in-memory objects on the streaming side, and overall higher system pressure.

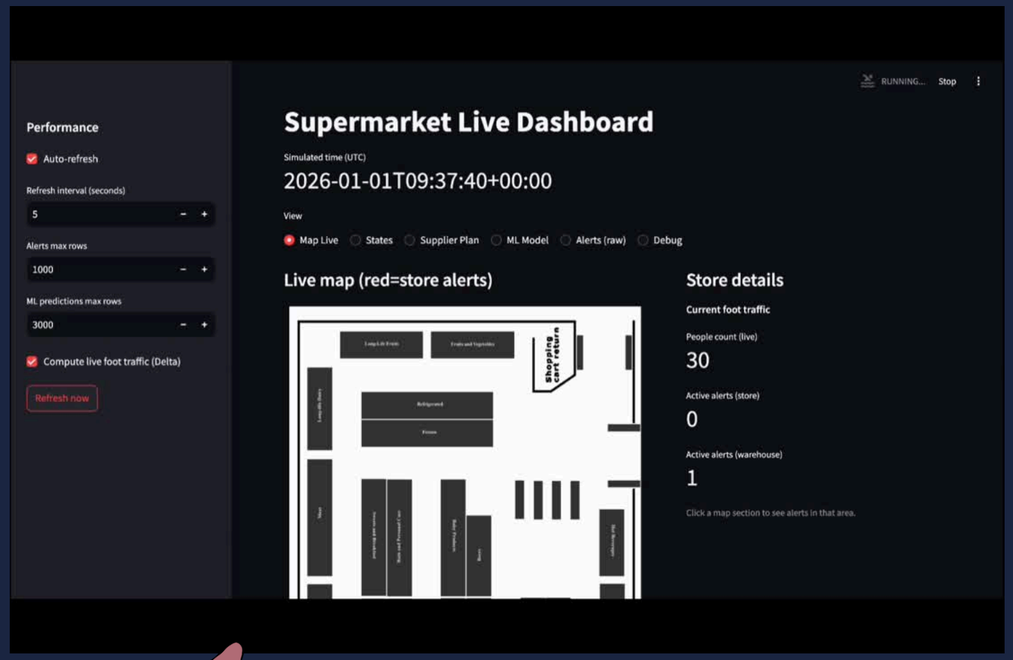
Since no real data source was available, the data stream had to be artificially recreated using Kafka producers. This choice is necessary for end-to-end testing and demos, but it introduces “non-functional” overhead (generating and transporting data instead of just processing it), which puts additional strain on the system and makes **CPU and RAM efficiency look worse than it actually is**.

Results and performance

The end-to-end pipeline stays interactive in the “Supermarket Live Dashboard”, supporting auto-refresh at a 5-second interval while rendering up to 1,000 alert rows and 3,000 ML-prediction rows per view (those settings can be changed).

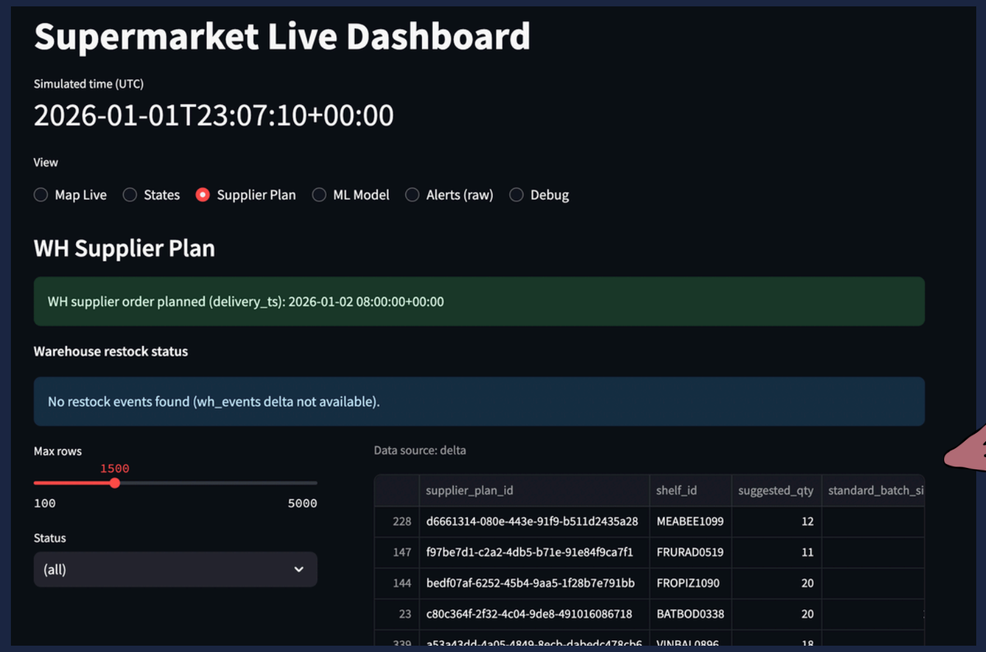
In a representative snapshot, the live map surfaced real-time foot-traffic (e.g., 11 people count live) and alert signals (e.g., 5 store alerts and 46 warehouse alerts) at a given simulated timestamp





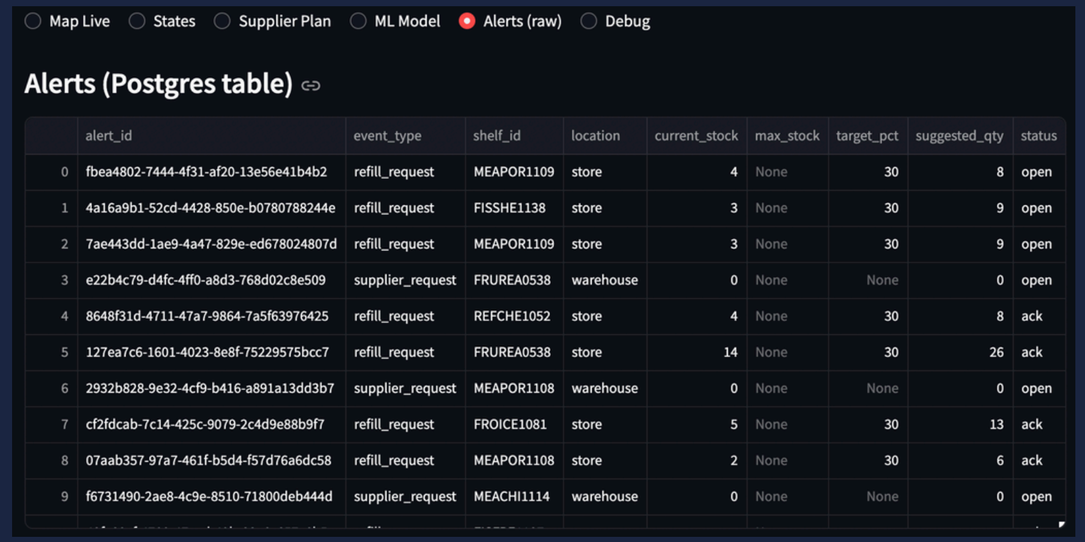
As can be seen, we designed the system so that when one or more shelves require restocking or when items are about to expire (thus generating alerts), the shelves are highlighted in red on the map. If the alert is of the type “near expiry” the items are removed, otherwise they have been purchased. These situations are then addressed and resolved by taking the required items from the warehouse. In the meantime, such store management operations can generate alerts in the warehouse itself. In the next slides, we showed how these warehouse alerts are handled and resolved.

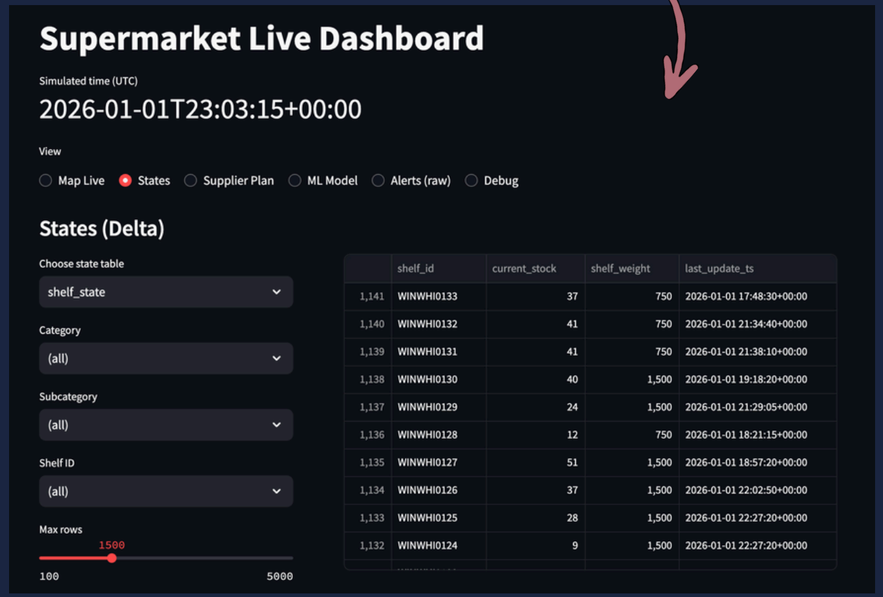
Results and performance



When warehouse shelves need to be replenished (thus generating alerts), the request is added to a warehouse supplier plan, that is, a list of product quantities required to restock the warehouse (*raw_suggested = max(0, reorder_point_qty - wh_current_stock)*). The plan is closed and sent to suppliers at 12:00 on Sundays, Tuesdays, and Thursdays.

Alerts and the states of shelves in both the store and the warehouse, as well as the states of batches in the store and warehouse, are available live in the following tables





Results and performance

The ML component is a supervised regression model that predicts how many supplier batches the warehouse should order, based on daily per-shelf features. It is trained for the first time on historical (simulated) data using an XGBoost regressor with time-aware cross-validation, and the full feature space is saved to ensure consistent inference. Shortly before supplier cutoffs (12:00), the model runs inference on current warehouse conditions, converts predictions into batch quantities with safe rounding and fallback rules, and generates replenishment plans that are logged and made available to downstream systems. The model is then retrainable every week.

Map Live | States | Supplier Plan | ML Model | Alerts (raw) | Debug

ML Model

TRAINED MODEL

Model name	Model version	Trained at
xgb_batches_to_or…	20260101_074940	2026-01-01 07:49:4…

Metrics (latest):


```
{
  "mae" : 0.03479543328285217
  "fold" : 2
  "features" : 335
}
```


Predictions rows	Distinct shelves	Total suggested qty
372	372	18155

INFERENCE

Predictions (filtered):

	model_name	feature_date	shelf_id	predicted_batches	suggested_qty	model_version	created_at
0	xgb_batches_to_order	2025-12-31	BABBAB0477	1	55	20260101_074940	2026-01-01 11:55:35+00:00
1	xgb_batches_to_order	2025-12-31	BABCAR0495	1	80	20260101_074940	2026-01-01 11:55:35+00:00
2	xgb_batches_to_order	2025-12-31	BABCAR0497	2	110	20260101_074940	2026-01-01 11:55:35+00:00
3	xgb_batches_to_order	2025-12-31	BABCOO0471	1	50	20260101_074940	2026-01-01 11:55:35+00:00
4	xgb_batches_to_order	2025-12-31	BABDIA0464	2	30	20260101_074940	2026-01-01 11:55:35+00:00
5	xgb_batches_to_order	2025-12-31	BABDIA0465	1	40	20260101_074940	2026-01-01 11:55:35+00:00
6	xgb_batches_to_order	2025-12-31	BABDIA0466	2	70	20260101_074940	2026-01-01 11:55:35+00:00
7	xgb_batches_to_order	2025-12-31	BABDIA0468	7	70	20260101_074940	2026-01-01 11:55:35+00:00
8	xgb_batches_to_order	2025-12-31	BABDIA0469	1	20	20260101_074940	2026-01-01 11:55:35+00:00
9	xgb_batches_to_order	2025-12-31	BABHOM0482	1	20	20260101_074940	2026-01-01 11:55:35+00:00

At 8:00 a.m. on Mondays, Wednesdays, and Fridays, the warehouse receives the required quantities of goods, optimized also thanks to the model's predictions. The alerts for warehouse are then resolved.

Warehouse restock status

Last restock day	Last restock time	Items restocked (day)
2026-01-02	2026-01-02 08:00:2…	18183

Shelves restocked (day): 374

Restocked products (last restock day):

	shelf_id	item_category	item_subcategory	qty	unit	received_date	timestamp	batch_code	reason
325	BABBAB0477	BABY PRODUCTS	Baby formula	55	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABBAB0477-1	supplier_
339	BABCAR0495	BABY PRODUCTS	Care and hygiene	80	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABCAR0495-1	supplier_
326	BABCAR0497	BABY PRODUCTS	Care and hygiene	110	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABCAR0497-1	supplier_
340	BABCOO0471	BABY PRODUCTS	Cookies	50	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABCOO0471-1	supplier_
327	BABDIA0464	BABY PRODUCTS	Diapers	30	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABDIA0464-1	supplier_
328	BABDIA0465	BABY PRODUCTS	Diapers	40	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABDIA0465-1	supplier_
0	BABDIA0466	BABY PRODUCTS	Diapers	70	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABDIA0466-1	supplier_
341	BABDIA0468	BABY PRODUCTS	Diapers	70	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABDIA0468-1	supplier_
329	BABDIA0469	BABY PRODUCTS	Diapers	20	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABDIA0469-1	supplier_
1	BABHOM0482	BABY PRODUCTS	Homogenized foods	20	ea	2026-01-02	2026-01-02 08:00:25+00:00	SUP-20260102-BABHOM0482-1	supplier_

Lessons learned

MAIN CHALLENGES:

One of the main challenges was designing Kafka topics that guarantee full traceability through append-only logs while still enabling fast access to state via compaction. Ensuring correctness in stateful streaming also required careful attention, particularly in the management of stable keys and rigorous checkpointing. However, due to the very nature of the simulation-based architecture, it is not feasible to avoid resetting checkpoints every time the project is started. Doing so would also require checkpointing the simulated time, an option that was not implemented.
In addition, reliably orchestrating a large number of containers proved to be complex, especially when handling service dependencies such as database initialization, connector startup, and the execution of Spark jobs.

WHAT WORKED WELL:

Using a simulated time source made demos and tests far more predictable, allowing expiry rules, reorder cycles, and cutoffs to be reproduced reliably. The clear separation between raw and derived data sped up debugging and made analytical queries easier to write and reason about.

WHAT DID NOT WORK:

Stateful processing was particularly sensitive to key drift and inconsistent checkpoints, where even small mistakes could cascade into incorrect state.
Due to the need to generate simulated data, the program cannot be supported on machines with very limited CPU and RAM.
Container startup was fragile without explicit dependency sequencing and better operational runbooks.

Limitations and future work

Limitations:

- Data is synthetic/simulated; real deployments need integration with real sensor systems and stronger data quality guarantees.
- Single-node Docker setup (single Kafka broker, local Delta filesystem) doesn't represent production scaling.
- Stateful Spark jobs may become bottlenecked by state size, shuffle-heavy joins, and checkpoint I/O as event rate count increases.
- Checkpoints need to be deleted after every new run.
- CPU and RAM usage are very high.

What would break first (and why):

- Kafka throughput/partitions (insufficient parallelism), Spark micro-batch latency, and Delta checkpoint growth/storage bandwidth.
- PostgreSQL write amplification if too many derived tables are sunk at high frequency.
- Containers go down not for errors but for OOM issues that need to be handled.

Future work:

- Move Delta tables to object storage to separate compute from storage and handle growth reliably.
- Use a schema registry and compatibility rules so producers can change event formats without silently breaking downstream consumers.
- Always compare XGBoost against a simple baseline (like a moving average) to prove the model actually adds value.
- Efficient recovery of the program from checkpoints if it is shut down during processing.
- Optimise the usage of CPU and RAM.

References and acknowledgments

References:

- Drive Research. Grocery store statistics: Where, when, and how much people grocery shop. https://www.driveresearch.com/market-research-company-blog/grocery-store-statistics-where-when-how-much-people-grocery-shop/

- Huang, A., Zhang, Z., Yang, L., & Ou, J. (2024). Design of Intelligent Warehouse Management System. 2024 4th International Conference on Electronic Information Engineering and Computer (EIECT), 534–538. https://doi.org/10.1109/EIECT64462.2024.10866043

Acknowledgments:

- Conad supermarket (Scandiano, Italy) for providing real-world category/inventory structure used to model the synthetic data.

- External contributors/maintainers of the open-source libraries used in the stack.