NearBuy

FRONTEND

E-Commerce Workshop

TECHNOLOGIES USED:

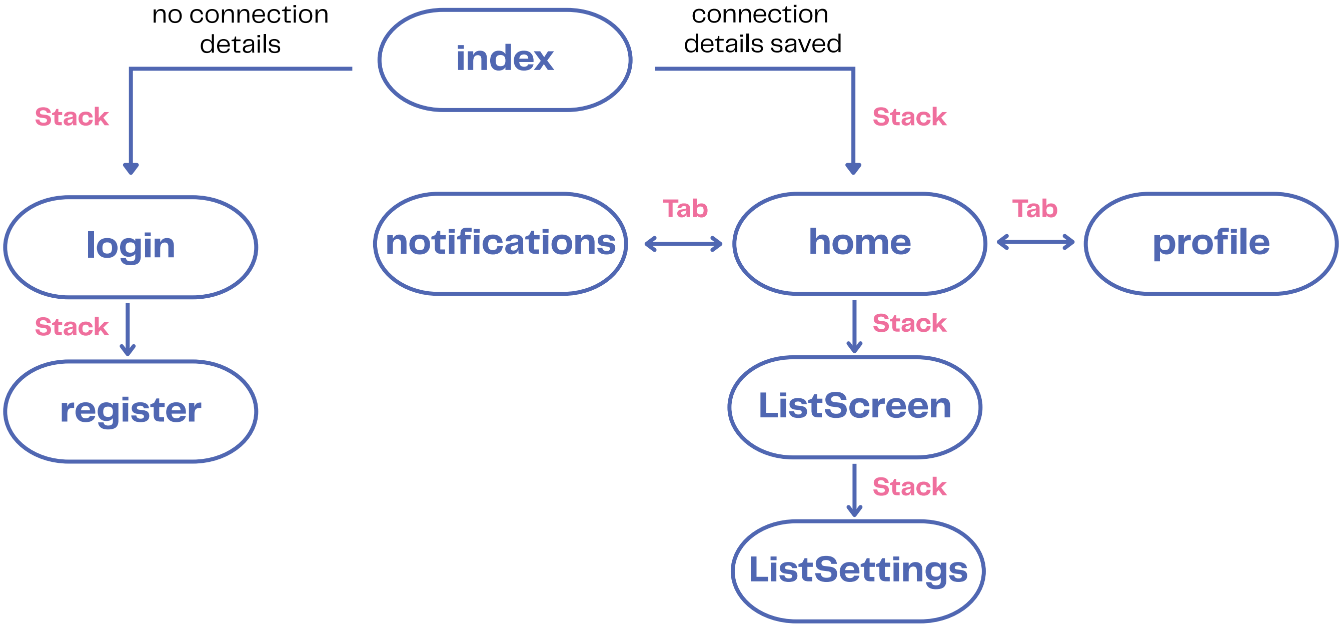
- **React Native** – Framework for building native mobile apps using JavaScript and React, for both Android and iOS.
- **TypeScript** – Adds type safety and better tooling to the codebase.
- **Expo** – A set of tools and services built around React Native for fast development, testing, and deployment.
- **Axios** – HTTP client for making API requests to the backend.

COSTUMED COMPONENTS:

- CheckList
- Item
- ListCard
- NotificationCard
- RecommendationItem

Screens & Navigation

- **Stack Navigation** – used for hierarchical flow where one screen leads to another, allowing users to move forward and back through a "stack" of screens.
- **Tab Navigation** – provides quick access to main sections from the bottom of the app, allowing users to switch between them directly and in no particular order.



NearBuy
BACKEND
E-Commerce Workshop

TECHNOLOGIES USED:

- **FastAPI:** A modern, fast (high-performance) web framework for building APIs with Python 3.7+ based on standard Python type hints.
- **Supabass:** Providing Postgres database, authentication, and real-time subscriptions.
- **SQLAlchemy:** An SQL toolkit and Object Relational Mapper (ORM) that gives developers the full power of SQL.
- **Requests:** An elegant and simple HTTP library for Python, used for making API calls to the ML component.
- **Pydantic:** A data validation and settings management library using Python type hints, used for defining data models.
- **Firebase Cloud Messaging (via Expo):** Used to send push notifications for alerts to Android devices. Integrated through Expo's notifications module.

NOTIFICATIONS

The BE implements two types of alerts: location-based and deadline-based.

LOCATION-BASED ALERTS:

The location_update endpoint is triggered by client-side location updates. Its purpose is to notify users when they are near a store that is likely to have items from their active shopping lists that have geo_alert enabled.

- **Input:** The endpoint receives the user's current latitude and longitude.
- **Item Filtering:** The system fetches all active, geo-alert-enabled items whose deadlines have not passed.
- **Proximity Check:** We calculate the user's distance from each store using the Haversine formula, and flag stores within 500 meters.
- **Availability Check:** If the user stays within 500 meters of a store for 2 minutes, it creates a proximity record. It then checks for existing item availability predictions in the database. If missing, it queries the ML component and stores the result to avoid repeated calls.
- **Notification Trigger:** If items are likely available, the system sends a push notification to the user's registered devices and logs the alert.
- **Exiting Proximity:** The proximity record is deleted once the user leaves the store's area.

DEADLINE-BASED ALERTS:

The check_deadlines_and_notify function is designed to run periodically as a scheduled job. It identifies lists and items with approaching deadlines and sends push notifications to users.

- **Scan for Due Lists:** The system finds active lists with a deadline set, not yet notified, and due within 24 hours.
- **Notify Lists:** We send a push notification for each due list and set its deadline_notified flag to prevent duplicates.
- **Scan for Due Items:** Similarly, the system identifies individual items with upcoming deadlines that haven't been notified yet.
- **Notify Items:** If the item's approaching deadline is different from the parent list's, a separate notification is sent, and the item's deadline_notified flag is updated.

RECOMMENDATIONS

The backend integrates with the ML component to provide item recommendations, triggered when a new list is created and updated twice daily for all lists.

- **Product-Based Recommendations:** For each item, the backend requests similar products using the ML component, based on embeddings and category classification.
- **List-Based Recommendations:** Additional suggestions are generated based on the list's name using community-driven similarity.
- **Accepting/Rejecting Suggestions:** Users can add or dismiss suggestions. Accepted ones are marked as used, dismissed ones are marked as rejected.
- **Filtering and Storage:** Suggestions already in the list, rejected, or used are skipped. Old ones are cleared. New filtered ones are saved.

NearBuy
ML COMPONENT
E-Commerce Workshop

Models

RECOMMENDATION SYSTEM ARCHITECTURE

GOAL

Our system is designed to recommend relevant products based on:

- A given shopping list name (e.g., "birthday party", "new apartment")
- A specific product name (e.g., "iPhone cable", "pillow")

It achieves this through a hybrid model combining community-driven list similarity, product embedding search, and a category classification model trained on curated data.

ARCHITECTURE BREAKDOWN

List-Based Recommendation Flow

- **Input:** GET /recommend_by_list_name?list_name=...
- The Similar Lists Unit searches for lists similar to the given list_name.
- Using the all-mpnet-base-v2 SentenceTransformer, the list name is embedded and matched against a FAISS index containing all existing embedded lists in the database.
- The matched lists are aggregated in the Producer Unit, which extracts product names.
- The Products Recommender Unit uses the aggregated product names to recommend relevant items.
- **Output:** a curated list of recommended products.

DESIGN REASONING & VALIDATION

- By leveraging existing community data, we treat shopping lists as implicit signals of intent.
- This lets us generalize recommendations for new users or new lists using patterns in existing list data.
- Embedding-based FAISS search ensures recommendations are semantically aligned and typo-resilient.
- High-quality dataset, tagged manually by human and 95% training accuracy.

ARCHITECTURE BREAKDOWN

Product-Based Recommendation Flow

- **Input**: GET /recommend_similar_products?product_name=...
- The Embedder Unit encodes the product name.
- The Top Categories Predictor (MLPClassifier - NN) predicts the most relevant product categories using features: TF-IDF vectors and product embeddings.
- This classifier was trained on a manually curated dataset of product names and their corresponding categories.
- Based on the predicted categories, the system narrows down the search domain to only relevant products.
- A Similarity Search Unit uses FAISS to find the closest products within those categories.
- **Output**: a list of similar products.

DESIGN REASONING & VALIDATION

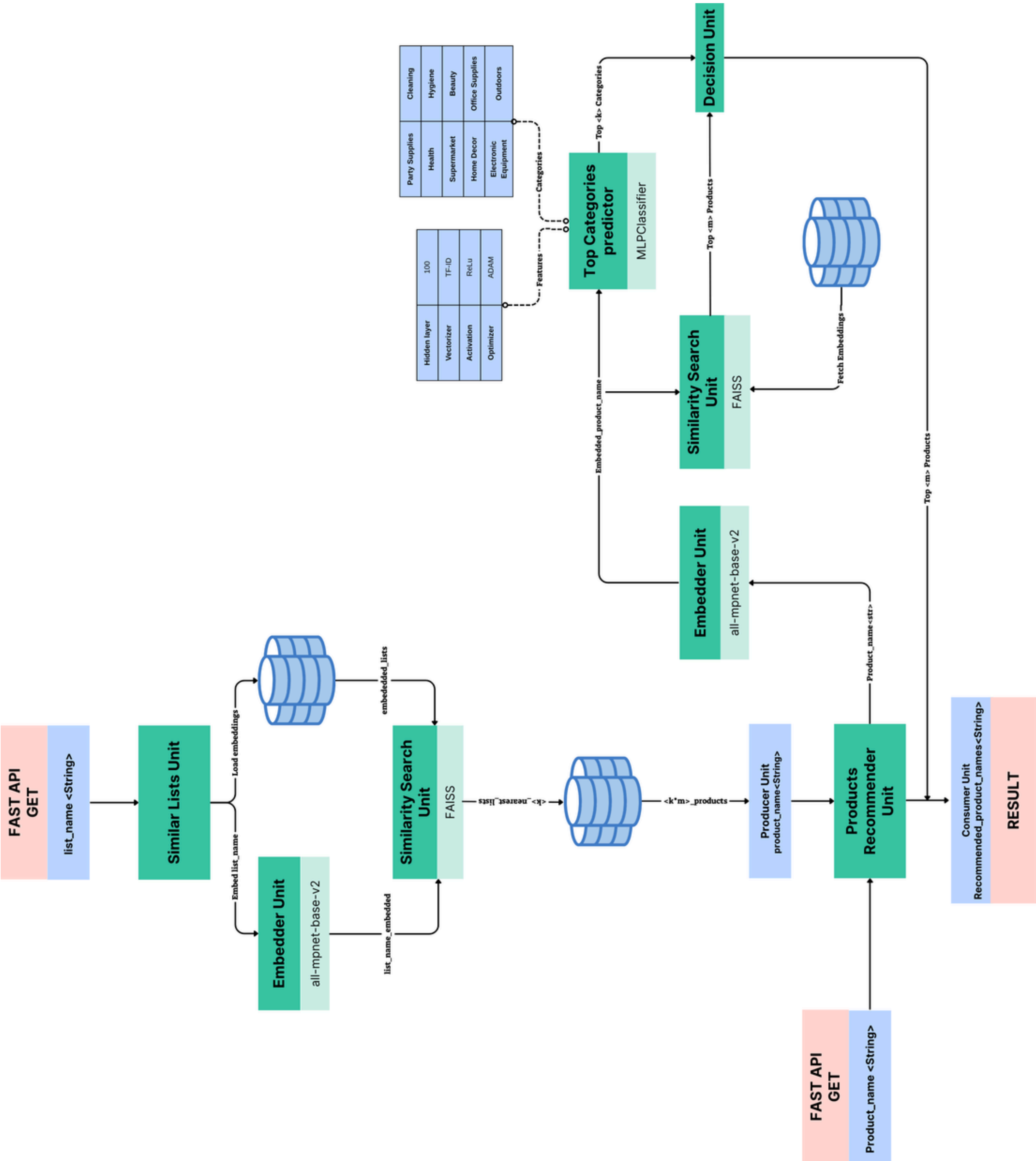
- Combining classification with embedding search reduces irrelevant matches and boosts relevance.
- The MLPClassifier allows the system to inject domain knowledge from labeled data into an otherwise unsupervised setup.
- Embedding-based retrieval ensures flexible handling of phrasing differences or misspellings.
- The MLPClassifier was trained on manually collected and labeled product-category pairs.
- TF-IDF vectorization captures name-specific keyword signals.
- A hidden layer with 100 neurons and ReLU activation and Adam optimizer.
- The classifier was evaluated using cross-validation and showed high precision in top-3 category prediction.
- We evaluated both pipelines using: manual inspection for semantic quality and real user-created shopping list to verify product coherence.
- The system performs well even with misspelled or incomplete input, thanks to the robustness of the embedding models and FAISS similarity search.

THE POWER OF COMMUNITY

One of the strongest aspects of this design is the community-based insight baked into the list recommender pipeline:

- Every new shopping list added by users contributes to a growing pool of intent-labeled data.
- The system learns from trends across users (e.g., what products are commonly grouped for “summer camping”).
- This enables cold-start handling and socially informed recommendations, which traditional content-based systems often lack.

HIGH LEVEL DESIGN



AI AGENT - NOTIFICATIONS BASED ON PRODUCT AVAILABILITY

GOAL

The system is designed to determine whether a specific **product** is likely **available** in a given **store**, based on real-time web information and intelligent automation. It receives input via a FastAPI endpoint, orchestrates multiple tools using a modular AI agent, and returns a structured response including confidence, reasoning, and even pricing if available.

ARCHITECTURE BREAKDOWN

Entry Point – FastAPI Endpoint

- Inputs: product_name <String>, store_name <String>
- Exposes a RESTful GET endpoint that acts as the trigger for the entire process.

Store-Product Recommender Agent

- Core coordinator of the pipeline.
- Receives input and orchestrates actions using agent tools and automation logic.
- Uses both retrieval-based logic (via FAISS caching) and LLM-based reasoning.

Agent Automation Unit

- Implements the high-level reasoning and decision-making flow using LangGraph.
- LangGraph enables conditional, step-by-step execution paths similar to workflow graphs but optimized for LLM orchestration.
- Ensures the agent explores external pages only when needed and stops early when high confidence is achieved.

Agent Tools (Pluggable Modules)

- <u>Find Store Website Tool:</u> Locates the official website of the store based on its name.
- <u>Extract Relevant URLs Tool:</u> Parses internal pages like catalog, product listings, or search results.
- <u>Summarize Page Tool:</u> Analyzes each relevant page to determine if the product is present.
- These tools combine web automation (Playwright), HTML parsing (BeautifulSoup), and OpenAI-powered LLM reasoning.

Local Semantic Caching Unit

- A FAISS-based local cache using all-MiniLM-L6-v2 sentence embeddings.
- Used to avoid re-computation for previously seen (store, product) pairs.
- Supports fast fuzzy matching and drastically reduces latency.

Result Format

Returns JSON structured output including: Product and Store names, Recommendation (True/False), Confidence score (Float), Reason (LLM-generated explanation) and Price (if found).

DESIGN CHOICES

Modular Agent Design

- The use of individual tools allows the system to be extensible and debuggable.
- Tools are independently testable and reusable.
- Future enhancements (e.g., discount detector, stock checker) can be added without rewriting the core logic.

LangGraph for LLM Orchestration

- LangGraph allows defining stateful, conditional flows using language models.
- It enables early stopping, looping through pages, and tool-triggered decision-making — far beyond simple function chaining.
- This makes the agent intelligent, adaptive, and traceable.

Live Web Context vs. Static Data

- Traditional recommendation systems often rely on historical or user-generated data. Our system augments this with real-time web context — including:
  - Up-to-date product listings
  - Discounts and availability
  - Store-specific catalog structure
- This enhances precision and user trust in recommendations.

VALIDATION & EVALUATION

We validated the model through:

- **Manual evaluation:** Comparing recommendations to actual store websites.
- **Tool confidence scoring:** LLM outputs include structured confidence and reasoning for debug, and prompt/ design tuning.
- **Caching accuracy:** Verified that FAISS retrieval aligns with previous full-run results.

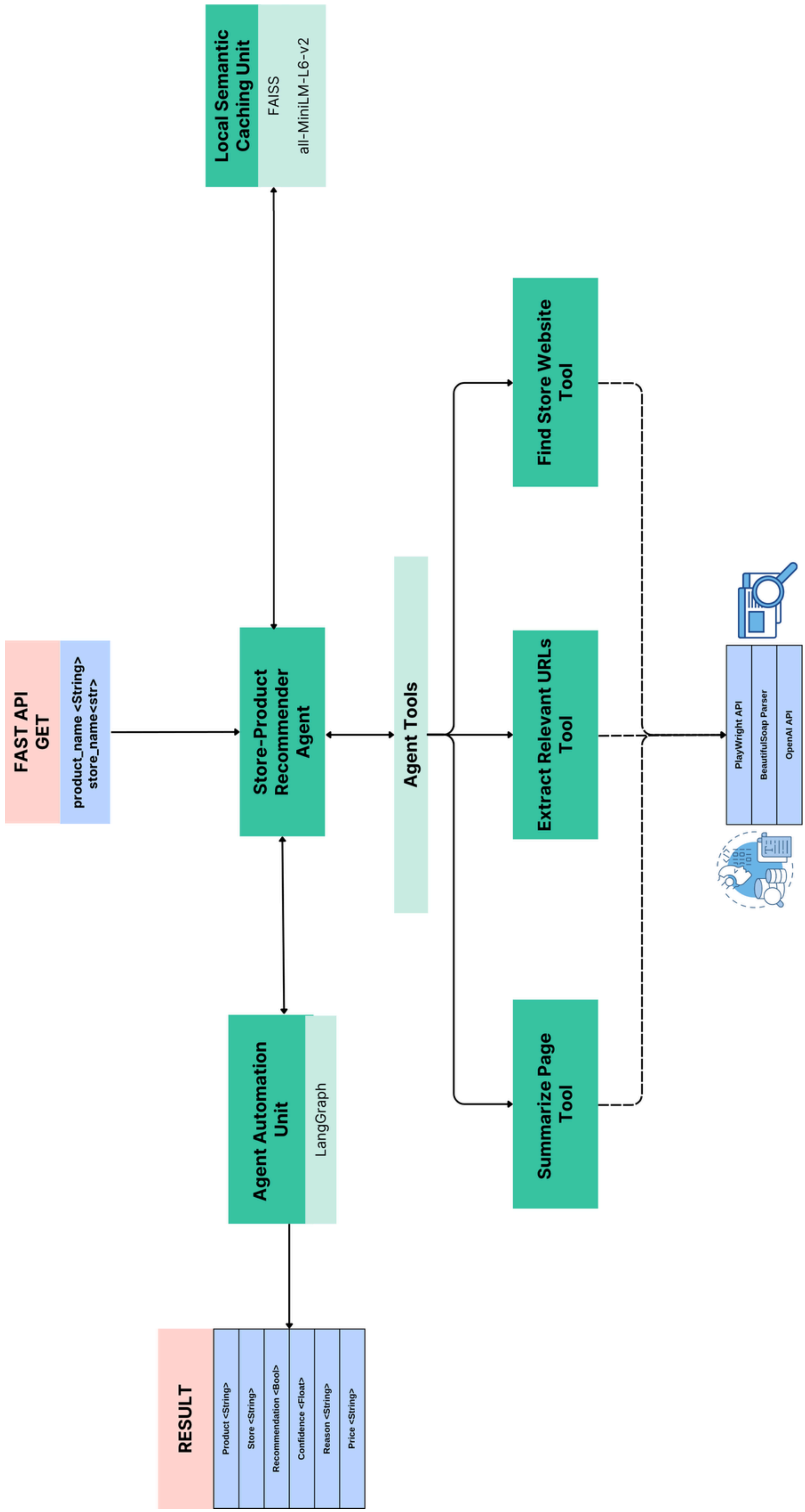
Planned future validations:

- A/B testing against baseline models (e.g., collaborative filtering).
- User feedback loop integration (e.g., thumbs up/down on predictions).
- Monitoring false positives with discount-related edge cases.

ADVANTAGES

- **Real-time intelligence:** Product detection is based on the live state of the web.
- **Flexible and scalable:** Tools can be reused across use cases.
- **Modern orchestration:** Using LangGraph brings cutting-edge LLM control flow into production

HIGH LEVEL DESIGN



NearBuy
DATABASE
E-Commerce Workshop

DATABASE OVERVIEW

Our system uses **Supabase** to manage structured, relational data via a scalable Postgres database. To support our ML features, we manually generated a realistic dataset of users, lists, and stores, ensuring both coverage and credibility for training and evaluation.

Core Data

Table	Purpose	Data Source
auth.users	App login info	Created during signup flow
user_profiles	User settings & preferences	Set at signup or via user settings
lists	User-created shopping lists	Created by users
lists_items	Items inside a shopping list	Created by users
stores	Store names & locations	OpenStreetMap & manual additions

ML Data

Table	Purpose	Data Source
items_suggestions	Extra items recommended for a list	AI-based product engine
store_item_availability	Predicts if a store has a given item	AI model prediction
alerts	Reminders for deadlines or store nearby	Triggered by backend logic
user_store_proximity	Tracks if user is near a store and notified	Triggered by backend logic