# ARCHITECTURE

The system follows a **modular microservice-inspired architecture** where each component has a clear responsibility. This separation makes the system **scalable, maintainable, and testable**.

## Components

1. **FastAPI Backend (FastAPI.py)**
   * Acts as the **orchestrator service**.
   * Accepts API requests from the frontend (/v1/items/analyze).
   * Validates input (ensures exactly 4 valid, publicly accessible image URLs).
   * Calls three different model services in **parallel threads**:
     + **Gemini Vision (gemini-2.5-flash)** → extracts semantic attributes such as category, brand, material, style, fit.
     + **Google Cloud Vision (Heuristic Model)** → specialized for low-level tasks, here used for color classification.
     + **Meta LLaMA** **(LLM Call)** → handles structured clothing attributes like sleeve length, neckline, closure type.
   * Aggregates responses from all models, logs timing for each, and prepares a unified JSON response.
2. **PostgreSQL Database**
   * Stores inference results in a table called inference\_results.
   * Uses **JSONB columns** for attributes, model\_info, and processing metadata.
   * This makes it efficient to query attributes (e.g., “find all Polo shirts with long sleeves”) and easy to extend if more attributes are added later.
   * Provides persistence so results are not lost between API calls.
3. **Streamlit Frontend (frontend.py)**
   * Lightweight UI for end-users.
   * Allows users to paste or upload 4 image URLs.
   * Sends a request to the FastAPI backend and displays the results (category, brand, color, style, etc.) in a clean dashboard format.
   * Useful for demos, validation, and quick iteration.

## Data Flow

1. **User Interaction**
   * The user enters 4 image URLs into the Streamlit UI.
2. **Request Validation**
   * The frontend sends a POST request to the FastAPI backend.
   * Backend ensures images are valid, publicly accessible, and ≤10MB.
   * If validation fails, an error JSON is returned.
3. **Model Orchestration**
   * FastAPI launches 3 worker threads (Gemini, Cloud Vision, LLaMA).
   * Each thread processes its assigned attributes.
   * Failures in any single model are logged, and default "unknown" values are returned to maintain schema consistency.
4. **Response Assembly**
   * Backend merges all model outputs into a single attributes JSON object.
   * Attaches model\_info (model names, attributes handled, latency) and processing (total latency, per-model timings, status).
5. **Persistence**
   * Results are inserted into Postgres (inference\_results table).
   * This enables historical queries, analytics, and auditing.
6. **Frontend Display**
   * Backend returns final JSON to Streamlit.
   * Streamlit renders the structured attributes in a human-readable UI for the user.

**HOW TO RUN**

1. Setup environment

Make sure you have Python 3.10+ installed.

Create and activate a virtual environment:

python -m venv env

source env/bin/activate # On Linux / macOS

env\Scripts\activate # On Windows

Install dependencies:

pip install -r requirements.txt

2. Configure environment variables

Create a .env file in the backend folder and add:

POSTGRES\_HOST=localhost

POSTGRES\_PORT=5432

POSTGRES\_DB=yourdbname

POSTGRES\_USER=yourdbuser

POSTGRES\_PASSWORD=yourdbpassword

GEMINI\_API=your\_gemini\_api\_key

GROQ\_API=your\_groq\_api\_key

3. Run FastAPI backend

Run the backend using uvicorn:

uvicorn FastAPI:app --reload

The backend will be live at http://127.0.0.1:8000

Endpoints:

- POST /v1/items/analyze → analyze 4 images

- GET /v1/status → check API health

Frontend (Streamlit)

1. Install dependencies

Inside the same virtual environment, install Streamlit if not already installed:

pip install streamlit

2. Run frontend

Run the frontend:

streamlit run frontend.py

## Diagram

A simplified diagram

A screenshot of a computer screen

AI-generated content may be incorrect.