Chào bạn, đây là một câu hỏi rất hay về tư duy kiến trúc phần mềm (Software Architecture). Với dự án MobileStore sử dụng Flask hiện tại của bạn, câu trả lời ngắn gọn và chuẩn nhất là:

Nên áp dụng mô hình MVC (Model-View-Controller).

Tuy nhiên, để code "chuẩn" và dễ mở rộng (maintainable), bạn không nên dùng MVC sơ khai (dồn hết vào `app.py`) mà nên dùng MVC kết hợp Modular (Blueprints) và Service Layer.

Dưới đây là phân tích chi tiết và lộ trình bạn nên làm để đưa dự án này về chuẩn kiến trúc:

---

1. Tại sao chọn MVC cho Flask?

Flask là một "Micro-framework" được thiết kế sinh ra để hoạt động tốt nhất với mô hình MVC. Cố ép nó theo Layered Architecture thuần túy (như Java Spring Boot) sẽ làm code trở nên cồng kềnh không cần thiết.

- **Model (M): Chính là file `models.py` (SQLAlchemy). Nó đại diện cho dữ liệu.**
- **View (V): Chính là thư mục `templates/` (Jinja2 HTML). Nó đại diện cho giao diện.**
- **Controller (C): Chính là các hàm `@app.route` trong `app.py`. Nó nhận request, xử lý và trả về View.**

Vấn đề hiện tại của bạn: Bạn đang dùng MVC, nhưng là "Fat Controller" (Controller béo phì). File `app.py` đang làm quá nhiều việc: cấu hình app, định nghĩa routes, xử lý logic thanh toán, gọi AI, xử lý session... Điều này khiến file này dài hàng trăm dòng, rất khó sửa lỗi sau này.

---

2. Kiến trúc chuẩn nên áp dụng: Modular MVC + Service Layer

Thay vì để tất cả trong 1 file, hãy tổ chức lại theo cấu trúc này. Đây là chuẩn mực cho các dự án Flask quy mô vừa.

Cấu trúc thư mục đề xuất:

Plaintext

```
MobileStore/
│
├── app/                    # Thư mục chứa source code chính
│   ├── __init__.py         # Khởi tạo Flask app, db, login_manager (App Factory)
│   ├── models.py           # (MODEL) Chỉ chứa Class Database
│   │
│   ├── services/           # (SERVICE LAYER) Chứa logic nghiệp vụ phức tạp
│   │   ├── ai_service.py       # Chứa các hàm gọi Gemini, xử lý JSON từ AI
│   │   ├── cart_service.py     # Logic tính toán giỏ hàng, tổng tiền
│   │   └── product_service.py  # Logic thêm/sửa sản phẩm, xử lý variants
│   │
│   ├── routes/             # (CONTROLLER) Chia nhỏ app.py ra thành các file nhỏ (Blueprints)
│   │   ├── auth.py         # Login, Register, Logout
│   │   ├── main.py         # Home, Detail, Cart, Compare
│   │   └── admin.py        # Dashboard, Add/Edit Product
│   │
│   ├── templates/          # (VIEW) Giữ nguyên
│   └── static/             # CSS, JS, Images
│
├── config.py               # Cấu hình (Secret key, DB URI)
├── run.py                  # File chạy duy nhất (Entry point)
└── requirements.txt
```

---

3. Phân tích lợi ích của mô hình này

A. Tách nhỏ Controller (Sử dụng Flask Blueprints)

Hiện tại `app.py` của bạn đang ôm đồm cả Admin, cả Khách hàng, cả Login.

- **Cách làm: Bạn sẽ tách ra: `routes/admin.py` chỉ chứa các route bắt đầu bằng `/admin`, `routes/auth.py` chỉ chứa route `/login`, `/register`.**
- **Lợi ích: Dễ tìm code. Nếu lỗi phần Admin, bạn biết ngay phải vào file nào.**

B. Áp dụng Service Layer (Tách Logic khỏi Route)

Đây là điểm quan trọng nhất để code "chuyên nghiệp". Controller (Route) chỉ nên làm nhiệm vụ điều phối, không nên chứa logic tính toán.

Ví dụ Code hiện tại (Trong `app.py`):

Python

```
# Controller đang ôm logic xử lý AI và xử lý lỗi

@app.route('/compare', methods=['POST'])

def compare_page():

    # ... Lấy dữ liệu form ...

    # ... Gọi hàm get_comparison_result ...

    # ... Xử lý lỗi ...

    return render_template(...)
```

Ví dụ Code chuẩn (Sử dụng Service):

Python

```
# routes/main.py (Controller chỉ điều phối)

from app.services.ai_service import AIService
```

```
@main.route('/compare', methods=['POST'])
def compare_page():
    p1_id = request.form.get('product1')
    p2_id = request.form.get('product2')

    # Đẩy toàn bộ logic so sánh, cache, xử lý lỗi sang Service
    result = AIService.compare_products(p1_id, p2_id)

    return render_template('compare.html', result=result)
```

C. Model giữ nguyên (Persistence Layer)

File `models.py` của bạn hiện tại đang làm tốt vai trò của nó. SQLAlchemy là một ORM rất mạnh, nó đóng vai trò là lớp Data Access. Bạn không cần tách thêm lớp Repository trừ khi dự án cực lớn.

---

4. Kết luận và Lời khuyên

Với dự án MobileStore hiện tại:

1. **Không nên dùng Layered Architecture thuần túy (Enterprise Java style): Nó sẽ yêu cầu bạn tạo các file Interface, DTO, DAO... rất rườm rà và không Pythonic (không đúng phong cách Python).**
2. **Hãy giữ MVC nhưng tái cấu trúc (Refactor):**
    - **Bước 1 (Dễ): Di chuyển các hàm trong `utils.py` (như gọi AI, parse JSON) thành một class hoặc module trong thư mục `services/ai_service.py`.**
    - **Bước 2 (Trung bình): Sử dụng Flask Blueprints để cắt nhỏ file `app.py`. Tạo thư mục `routes` và chia file `app.py` thành 3 file: `auth.py`, `admin.py`, `public.py`.**

- **Bước 3 (Khó hơn): Tạo `__init__.py` để khởi tạo app (Application Factory Pattern).**

Đánh giá: Code hiện tại của bạn đang chạy tốt ("Working software is the best measure of progress"). Bạn không cần đập đi xây lại ngay lập tức. Nhưng nếu bạn muốn phát triển tiếp hoặc đưa vào CV xin việc, hãy dành thời gian thực hiện Bước 1 và Bước 2 ở trên. Nó sẽ biến dự án từ "Bài tập sinh viên" thành "Dự án chuyên nghiệp".

Bạn có muốn tôi hướng dẫn cách tách file `app.py` ra thành các Blueprints không? Đây là bước hiệu quả nhất để làm code gọn gàng ngay lúc này.