ChatGPT

Development Approach

- **Microservices Architecture:** Design the system as loosely-coupled services. For example, separate FastAPI "sub-agents" handle distinct tasks (retrieval, ranking, comparison) while a central FastAPI-based conversational service orchestrates them [1] [2]. FastAPI is chosen for its native **async/await** support, which lets the server pause on long-running LLM API calls without blocking other requests [3]. An API gateway (or the chief agent) routes user queries to the appropriate microservices (e.g. the retrieval agent). Each service can scale independently and maintain its own data (e.g. MongoDB collections) for its bounded context [4] [2].

- **Data Management:** Use MongoDB to store product catalogs and user data. For each product domain (e.g. electronics, books, fashion), define collections with fields like name, category, price, attributes, etc. When APIs are available, call them (using Python requests or SDKs) to fetch product data; otherwise implement web-scraping (e.g. BeautifulSoup, Scrapy) to crawl sites and populate MongoDB. Also build pipelines to compute and store text embeddings for product descriptions (using either OpenAI embeddings or open-source embedder) so the retrieval agent can perform similarity search [5] [6]. Optionally use a specialized vector store (Milvus, Qdrant) if scale requires it; otherwise embed documents in Mongo or Redis.

- **Retrieval Agent:** Implement a FastAPI service that takes user constraints (keywords, budget, features) and queries the data store. It can use MongoDB text search or vector similarity: e.g. embed the query and perform a k-NN search over the product embeddings [6]. Hybrid methods improve precision – combine vector search with keyword filters or a lightweight reranking model [7]. The agent returns a shortlist of relevant products.

- **Ranking Agent:** Build a FastAPI service to sort or score the retrieved items. This can use custom logic (e.g. matching user preferences or boosting certain attributes) or even a small ML model. In practice, systems often **filter and rank** results by criteria like similarity score, price proximity, or popularity [8]. For example, you might refine the ordering by calling an LLM (prompting it to rank given items by relevance) or use a learned model. The ranking agent returns a sorted list with confidence or score fields.

- **Comparison Agent:** Implement a service that takes two or more selected products and generates a human-readable comparison. For example, this agent can call the LLM (OpenAI or open-source) to produce bullet-point differences or a short explanation comparing key features. This helps fulfill the "explainable" aspect: users see *why* one item is recommended over another. The agent's API might be called by the conversational agent after the ranking step.

- **Conversational Chief Agent:** The main FastAPI service handles the user dialogue. It uses Streamlit UI to present a chat interface. Behind the scenes, it uses an LLM (e.g. OpenAI GPT-4 or an open-source model like LLaMA/Mistral via Hugging Face) to understand user input and maintain context. This agent collects constraints (budget, use-case, preferences) through dialog, then invokes the sub-agents in sequence. For example, after gathering needs it calls the retrieval agent, then ranking, then comparison, and finally uses the LLM to **compose a response**. This LLM-driven reply includes the recommended products along with an explanation (e.g. "I recommend X because …") and any comparison insights. Research shows that conversational interfaces (built on NLP) enable human-like interactions [9] and that personalizing the dialogue

improves user satisfaction [10]. The agent can also clarify ambiguities by asking follow-up questions, much like a shopping assistant.

- **LLM Integration:** Use the OpenAI API for high-quality language and retrieval tasks. For reproducibility or cost saving, also integrate an open-source model (e.g. Vicuna, Mistral) using a library like `transformers`. You might switch between them or ensemble their outputs. FastAPI's async support is crucial here: GPT calls are I/O-bound, so async endpoints prevent server lock-ups [3]. Cache frequent queries (prompt hashing) to reduce API cost [11]. Design prompts carefully (prompt engineering) to guide the LLM in each sub-agent's task (retrieval summaries, ranking justifications, comparison blurb).

- **Explainability:** Incorporate "explainable AI" by having the LLM articulate reasons. Research suggests effective XAI should be *conversational*, mirroring natural Q&A between humans [12]. For each recommendation, have the agent generate a brief justification (e.g. citing relevant features or constraints) so the user understands the decision. Visual cues (like highlighting matching features) or text explanations should accompany each suggestion. The multi-agent setup facilitates this: after ranking, the chief agent uses the comparison agent's output and LLM text to present a clear, user-friendly explanation of *why* each product was chosen.

- **UI/UX:** Use Streamlit to build a responsive front end. Provide a chat window (for text conversation) and a results panel showing ranked product cards. Each card lists the product name, price, image, and key specs, plus the LLM-generated explanation. Streamlit easily connects to FastAPI endpoints via requests or websockets. The interface should allow free-text queries and also structured inputs (dropdown filters, sliders for budget). Per the reference design, a conversational UI that adapts to context and personalizes results leads to better engagement [9] [10]. Ensure the UI can display explanations inline (e.g. expandable "Why this?" sections) to keep the process transparent.

Development Steps

1. **Setup Environment:** Initialize a Python 3.9+ project. Install FastAPI, Uvicorn, Motor (async MongoDB driver), Pydantic, `openai` SDK, `transformers`, and Streamlit. Securely store API keys (OpenAI key, any scraping/API credentials) in environment variables. Spin up a MongoDB database (locally or Atlas) and test connectivity (e.g. using Motor). Choose deployment tools early (e.g. Docker for containers).

2. **Define Data Schema:** Design MongoDB collections for each product domain. For example, a `products` collection with fields like `name`, `category`, `price`, `features`, and a `tags`/`description` field for search. Ensure flexibility for multi-domain use (e.g. allow different attributes per category). Write scripts or use Mongo's import tools to load sample data.

3. **Data Ingestion Pipelines:**

4. *With APIs:* Implement Python client modules to fetch product data from available APIs (e.g. Amazon Product API, eBay API). Parse the JSON responses and upsert into MongoDB.
5. *Without APIs:* Develop web-scraping modules (with Requests/BeautifulSoup or Scrapy) for target sites. Extract product details, clean them, and store in MongoDB. (Be mindful of legal/ethical scraping limits.)

6. Schedule regular runs or triggers to keep data fresh (e.g. new arrivals, price changes).

7. **Embedding Generation:** For each product, generate a text embedding (using OpenAI's `text-embedding-3` or an open-source model). For example, concatenate name+description+category and call `openai.Embedding`. Store the vector alongside the product (either in Mongo as an array or in a dedicated vector store). This enables semantic search. Use [9] as guidance: chunk data if needed, and create a vector index to speed up k-NN retrieval.

8. **Build Retrieval Microservice:** Create a new FastAPI app for retrieval (e.g. `retrieval_service`). Define an endpoint (e.g. `/retrieve`) that accepts a user query or constraints. In the handler, convert the query into an embedding and query the vector store or perform a MongoDB text search. Return the top-N product IDs and basic details. Implement caching of frequent queries. Use asynchronous DB calls.

9. **Build Ranking Microservice:** Implement a separate FastAPI app (`ranking_service`). Expose an endpoint (e.g. `/rank`) that receives a list of product IDs and user context. This service can fetch product details from Mongo, score each (e.g. based on similarity to user preferences or other heuristics), and return a sorted list. Optionally, call an LLM: e.g. send a prompt like "Rank these products [list] for a user who wants X features under $Y." Use the LLM response or a simple rule-based sort. Ensure this runs quickly (async or lightweight model).

10. **Build Comparison Microservice:** Create `comparison_service` in FastAPI. Its endpoint (e.g. `/compare`) takes two or more product IDs. It fetches their specifications from Mongo and constructs a prompt for the LLM: e.g. "Compare these products and highlight differences." The LLM reply (perhaps bullet points) is returned. This service enhances explainability by giving human-friendly comparisons.

11. **Implement Conversational Agent:** In the main FastAPI app (`chat_service`), implement a chat endpoint (e.g. `/chat`). When the Streamlit UI sends a user message, do the following:
    a. Use the LLM to interpret it (handle context) and extract needed info (budget, product type, preferences). You may keep a conversation state or use system prompts.
    b. Once constraints are gathered, call the retrieval agent (`/retrieve`) with that query.
    c. Take its results and call the ranking agent (`/rank`) to refine the list.
    d. Optionally, select top 2–3 and call the comparison agent (`/compare`).
    e. Finally, compose a full response: Use the LLM to generate the final message combining the top recommendations and explanation. For example, feed it a prompt like: "Given these products [with key features], which do you recommend and why?"
    f. Return the LLM's response and product cards to the frontend.

This chain follows a RAG-like pattern: the LLM remains the orchestrator but relies on retrieved data [6]. Throughout, ensure each LLM prompt includes context ("Answer only based on the product data provided") to avoid hallucination.

1. **Streamlit Frontend:** Develop a Streamlit app that connects to the `chat_service`. Use a chat widget or text input for user queries. On submission, POST to `/chat` and display the returned conversation turn and product results. Show product names, images (if you stored image URLs), prices, and the explanation text. Allow filters (e.g. budget slider) as part of the chat or sidebar. Streamlit simplifies deployment and instant UI updates.

2. **Testing and Iteration:** Write unit tests for each microservice. Simulate sample dialogues and check if constraints are correctly parsed and recommendations seem reasonable. Use metrics:

e.g. hit-rate of relevant products, or A/B test LLM vs. open-source model. Pay attention to response speed and try to cache or pre-compute where possible (e.g. product embeddings). Conduct user testing to ensure explanations are understandable – good explanations should feel conversational [12] [9].

3. **Deployment:** Containerize each service (FastAPI apps and Streamlit) using Docker. Use Kubernetes or Docker Compose to orchestrate them. Host MongoDB on a managed cloud service (e.g. MongoDB Atlas). Deploy on cloud (AWS/GCP/Azure). Ensure secure communication between services (HTTPS, API keys). Monitor performance (logging, health checks).

Throughout development, apply agile refinement: collect logs of failed queries or hallucinations and improve prompts or data coverage. Leverage feedback loops to update the database or tweak ranking logic. The resulting system will combine traditional recommendation logic (retrieval+ranking) with generative AI, yielding **personalized, conversational recommendations** supported by transparent explanations [5] [9].

Sources: Architecting recommendations via microservices [5] [2]; FastAPI async patterns for LLMs [3]; retrieval/reranking techniques for RAG systems [6] [7]; conversational UX and personalization best practices [9] [10]; conversational XAI principles [12].

---

[1] [5] [8] Real-time AI Recommender System: Building a Production-Ready Architecture | by Akashpaul | Medium
https://medium.com/@akashpaul2030/real-time-ai-recommender-system-building-a-production-ready-architecture-3dc35ac31a33

[2] [4] Microservices Architecture Style - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[3] [11] Architecting Scalable AI Backends with FastAPI and OpenAI | by Afolabi Ifeoluwa James | Dec, 2025 | Medium
https://medium.com/@afolabiifeoluwa06/architecting-scalable-ai-backends-with-fastapi-and-openai-2671cbcb24bb

[6] [7] Why I Use Retrieval-Augmented Generation (RAG) in Real AI Systems — And How You Should Too | by Priya Singh | Medium
https://medium.com/@PriyaSingh325/why-i-use-retrieval-augmented-generation-rag-in-real-ai-systems-and-how-you-should-too-6761ca7ceac5

[9] [10] The GenAI Reference Architecture | by Ali Arsanjani | Medium
https://dr-arsanjani.medium.com/the-genai-reference-architecture-605929ab6b5a

[12] From Black Boxes to Conversations: Incorporating XAI in a Conversational Agent
https://arxiv.org/html/2209.02552v3