Architecture Brief

Overview

This microservice is designed to asynchronously crawl product pages from a public e-commerce site, persist product data reliably in a relational database, and expose AI-powered insights through a well-structured REST API. Key AI functionalities include lightweight product embeddings for similarity search and concise LLM-generated marketing summaries. The service is architected with modularity, scalability, and maintainability in mind, balancing production readiness with time constraints.

Key Architectural Trade-offs and Technology Choices

1. Library and Framework Selection

- **FastAPI:** Chosen for its asynchronous nature, type safety via Pydantic models, and automatic interactive API documentation. FastAPI's async support aligns well with the async crawler and scalable service needs.
- **SQLAlchemy ORM + Alembic:** Utilized for database abstraction and schema versioning.
  Enables easy migration between SQLite (local/dev) and PostgreSQL (production-ready).
  Alembic handles incremental schema changes, including indexes and constraints, improving maintainability.
- **aiohttp / asyncio-based crawler:** Provides high concurrency for efficient web scraping while being lightweight and flexible compared to heavier frameworks like Scrapy.
- **sentence-transformers (`paraphrase-MiniLM-L6-v2`)**
  Selected for embeddings because:
  - Lightweight and fast, suitable for local development and modest dataset sizes (~500 products).
  - Balances acceptable semantic quality with practical inference cost and latency.
  - Runs locally without dependency on cloud, aligning with limited infrastructure availability.
- **OpenAI GPT-3.5 Turbo:** Chosen for marketing summary generation, providing high-quality, human-like text output. Calls are cached persistently in the DB to control API usage and reduce latency/cost.

2. Database, Schema, and Indexing Choices

- **SQLite** is used for local development due to its zero setup and lightweight nature. Easily replaceable with **PostgreSQL** for production via SQLAlchemy.
- **Schema Design**
  - Normalized design with tables for products and embeddings.
  - Used indexes on `name`, `price`, `rating`, and a unique constraint on `URL` for efficient filtering and duplicate avoidance.
  - Removed index on `category` due to SQLite limitations, causing repeated deployment/test failures. With more time, this index should be added and managed via migrations in production Postgres.

3. Caching Strategy:

Choose **persistent caching in the database** for LLM-generated marketing summaries instead of in-memory or external caches.

Reasons: Limited time and no dedicated cloud caching infrastructure available (e.g., Redis), Database caching is simple, reliable, and persists over restarts, ensuring cost-effective usage of OpenAI API, Simplifies architecture by avoiding additional caching components, Easily extendable: with more time and infrastructure, can migrate to dedicated caching layers to improve performance.

Horizontal Scalability and Performance Optimizations

1. Horizontal Scaling

- The stateless FastAPI microservice can be scaled horizontally behind a load balancer in multiple cloud regions.
- Event-driven or queue-based scaling of the crawler and AI workload is possible using message brokers (Kafka, RabbitMQ) to handle crawl or AI processing jobs asynchronously and reliably.
  Database scalability is planned by migrating to PostgreSQL with replication and partitioning.

2. AI Workflow Optimization: **Embedding Search and Marketing Summary Generation**

- Currently uses in-memory cosine similarity suitable for moderate datasets.
- At scale, replace with Approximate Nearest Neighbor (ANN) libraries such as **FAISS** or **Annoy** to reduce latency and memory footprint.
- Batch or asynchronous generation and caching to avoid synchronous API latency on requests.
- Optionally run LLM locally if the infrastructure supports, to reduce dependency and costs.

3. Async IO Throughout

- Async APIs and crawlers maximize concurrency and throughput.
- Non-blocking I/O enables handling many simultaneous requests with low resource use.

Next Steps and Improvements with More Time

- **Complete migration to PostgreSQL:** Leverage advanced indexing, full-text search, and robust scaling with replicas.
- **Introduce Feature Store & Streaming Pipelines:** To better handle real-time analytics, incremental embedding updates, and more diverse AI feature management.
- **Use Managed or Custom Caching Service:** Integrate Redis or Memcached for AI summaries and recommendation caching, improving performance and scalability.
- **Build Queue/Task Orchestration:** Use Celery or other task queues for embedding and summary generation, decoupling heavy AI work from request paths.
- **Implement rate limiting and throttling:** For API endpoints to maintain quality of service at scale.
- **Comprehensive Monitoring and Logging:** Integrate distributed tracing, log aggregation, and alerting for operational robustness.

Summary: Every architectural choice balances:

- Simplicity and rapid iteration with Pythonic idioms and familiar libraries.
- Production readiness via modular design, containerization, and migrations.
- Scalability potential through async design and decoupled AI processing.
- Cost and resource efficiency by caching and local embedding models.

With more time and cloud infrastructure access, the system can evolve into a highly performant, resilient, and scalable recommendation and analytics platform.