Project Requirements Document

Prepared for Backend Development Batch

1. Objective & Context

Price-savvy shoppers and small resellers spend hours hopping between e-commerce sites to find the right product at the best price. Product information is fragmented, inconsistent, and sometimes hidden behind dynamic pages. Your mission: build a focused Flask backend that feels like a lean startup's MVP; one query in, a clean, comparable set of products out. The service scrapes multiple trusted marketplaces, normalizes prices and ratings, deduplicates near-duplicate listings using fuzzy matching, and exposes simple APIs for search, comparison, and details. The goal is fast and efficient scraping that returns useful insights quickly while respecting source sites' limits.

2. Technical Stack

- Python 3.10+
- Flask (Blueprints or modular structure for routes)
- BeautifulSoup4 for HTML parsing
- Requests or HTTPX for HTTP calls (with timeouts and retries)
- SQLite (via SQLAlchemy or built-in sqlite3) OR MongoDB (PyMongo) for storage
- Concurrency: concurrent.futures ThreadPoolExecutor OR asyncio + aiohttp
- Fuzzy matching: Python difflib.SequenceMatcher (or optional rapidfuzz if preferred)
- In-memory TTL cache (dict or cachetools) for recent queries/responses

3. Core Requirements

1. Expose a Flask REST API with endpoints for search, compare, and product detail, including pagination and sorting.
2. Scrape at least two e-commerce websites using BeautifulSoup; optionally use Selenium for pages relying on client-side rendering.
3. Normalize and deduplicate product data across sources (unified price format, standardized ratings, fuzzy name matching).
4. Implement efficient, concurrent scraping with request deduplication (caching) and basic per-IP rate limiting.
5. Persist normalized products in SQLite or MongoDB with a minimal schema that supports lookup by ID and URL

4. Functional Requirements

- Product Search API: GET /search?q={query}&page;={n}&per;_page={m}&sort;={price|rating} ={asc|desc}. Returns normalized product summaries with pagination metadata.
- Scraping Multiple Websites: For each query, scrape at least two predefined e-commerce sites to collect name, price, rating, product URL, and source. Use BeautifulSoup; if critical data is rendered dynamically, optionally use Selenium as a fallback.
- Data Normalization & Deduplication: Convert price to a numeric format in a single currency/unit (e.g., USD in cents or float). Standardize ratings to a 0–5 scale. Canonicalize titles (lowercase, strip brand/stopwords where reasonable) and merge near-duplicates using fuzzy token similarity with a clear threshold.
- Product Comparison API: GET /compare?ids={comma_separated_ids} OR /compare?q={query}&by;={price|rating} ={k}. Returns aligned fields (name, normalized price, rating, source URLs) and highlights best-by metric.
- Product Detail API: GET /products/{id} OR /products?url={product_url}. Returns full normalized record, source-specific attributes if available, and last_updated timestamp. If stale beyond TTL, trigger refresh in background before returning cached data.
- Pagination & Ranking: Default 20 products per page. Support sorting by price or rating; implement a stable sort so results are deterministic.
- Robust Error Handling: Return structured JSON errors for invalid queries (400), no results (200 with empty list and message), partial scrape failures (200 with per-source status), and rate limit exceeded (429).

5. Non-Functional Requirements

- Performance & Concurrency: Complete a typical two-site search within ~4 seconds on a standard laptop. Use bounded concurrency (e.g., max 5 workers) and per-request timeouts (e.g., 5s). Avoid redundant fetches via TTL query cache (~10 minutes).
- Polite Scraping & Reliability: Respect a per-site request interval and basic rate limiting (e.g., 10 requests/min per IP). Implement retries with exponential backoff for transient failures and fall back to cached results on errors.
- Security & Safety: Validate and sanitize user inputs; restrict outbound requests to a hardcoded allowlist of target domains; avoid executing arbitrary scripts; use a custom User-Agent string.
- API Consistency & UX: Consistent JSON schema with clear field names, currency/rating units documented, and pagination metadata (page, per_page, total, has_next). Deterministic sorting and stable product IDs.
- Observability: Structured logs for each fetch (source, URL, status, duration), scrape errors, cache hits/misses, and rate-limit events. Provide a lightweight /health endpoint.

6. User Flow

1. User calls GET /search with a query and optional sort/pagination parameters.
2. Service checks the TTL cache for that query; for uncached or stale sources, it schedules concurrent fetch tasks across the predefined sites with bounded workers and per-site pacing.
3. Each site scraper fetches the page(s), parses with BeautifulSoup, or uses Selenium if critical content is dynamically rendered; raw fields are extracted (name, price, rating, URL).
4. Pipeline normalizes price and ratings, canonicalizes titles, and clusters near-duplicates via fuzzy similarity; merged records are upserted into the database and cached by query.
5. Service ranks results according to requested sort (price or rating), paginates to 20 per page by default, and returns a JSON payload with results and pagination metadata.
6. User selects items to compare and calls GET /compare with product IDs (or repeats the query with a top-K comparison). API aligns attributes and returns a structured comparison highlighting the best value.
7. User requests a specific product via GET /products/{id}; service returns the stored normalized details and triggers a refresh if the record is stale.
8. If scraping fails or rate limits are hit, the API responds with meaningful messages, partial results when available, and guidance to retry after the indicated window