ChatGPT

Restoration and Enhancement Plan for Self-Hosted Search Engine

Overview: This plan outlines a step-by-step strategy to restore all core features of the self-hosted search engine and introduce new enhancements. We break down the tasks into modular components, covering crawling, indexing, search UX improvements, LLM integration, data policy, single-user scope, and technology choices. An architecture diagram is provided for clarity on how the pieces fit together. Each section below details the changes and justifications, with clear subcomponents and recommended tools.

Architecture Overview

Figure: Proposed architecture with data flow between modules. The user's actions (browsing or querying) flow through crawling and indexing pipelines into local databases, which the search and LLM modules use to retrieve and generate results.

At a high level, the system consists of the following modules working in tandem:

- **Crawler & Scraper:** Handles automatic web crawling of visited or seeded sites (using Scrapy and headless browser as needed) [1].
- **Content Normalizer:** Cleans and extracts text from HTML (e.g. via Trafilatura/BeautifulSoup).
- **Indexing Engine:** Stores extracted content into a **Whoosh** BM25 text index and a **vector store** (for embeddings) [2], along with metadata in a local database.
- **Local Knowledge Base:** The collection of stored pages, summaries, and metadata (in local files/DB) that the search engine and LLM can draw upon.
- **Search Query Handler:** Combines keyword search (Whoosh) with semantic search (vector similarity) [2], merges results, and provides explanations for result rankings.
- **LLM Integration:** Local models (Gemma-3, GPT-OSS for chat; EmbeddingGemma for embeddings) run via Ollama to assist in discovery (finding new pages), summarization, and answer generation [3] [4].
- **Control Center & Policy Module:** User interface to toggle features (e.g. shadow crawling, LLM usage) and enforce permissions (e.g. whether to auto-save content, handle large files).

With this context, we proceed to detailed steps for each set of features.

1. Restoration of Existing Features

The first phase is to **audit and re-enable all core modules** that may be broken or disabled – namely crawling, scraping, indexing, local storage, and basic search. We ensure the data flows from page capture to search results are working end-to-end:

- **Crawler (Scrapy) Reactivation:** Re-enable or fix the Scrapy-based crawler so it can run automatically in the background. The crawler should start whenever new sites are encountered (through user browsing or a search trigger) and politely crawl pages:
- **Automatic Triggers:** Ensure that when the user visits a new domain or issues a search query on an empty index, the crawler is invoked. The app already supports a "cold-start" focused crawl on first query [5] ; verify this is working and repair if needed.
- **Continuous Crawling:** Implement a background job or scheduler so that any URL added to the frontier (via visiting or manual input) gets crawled and stored. Use Scrapy's robust scheduling to manage the crawl queue efficiently [1] . Confirm that `make crawl` and the focused crawl pipeline (`bin/crawl_focused.py`) operate without errors.

- **Coverage and Depth:** Configure the crawler to fetch not just the exact visited page but also follow relevant internal links (to a certain depth or within the same domain) to enrich the index. For example, if the user visits a documentation page, the crawler could fetch related pages (subsections, "next" links, etc.) within that site, staying respectful of `robots.txt`.

- **Scraping & Content Extraction:** Verify the HTML content of fetched pages is being properly captured and parsed:

- **Shadow Browser Capture:** The integrated researcher browser's shadow mode should send each visited page to the backend for capture [6] . Check that `/api/shadow/snapshot` is functioning: it should receive the page URL or HTML and store a snapshot (HTML, text, metadata) locally.
- **Normalization Pipeline:** Ensure that after crawling or shadow capture, each page's HTML is normalized to clean text. The project uses Trafilatura + Playwright for this [6] – confirm these are working. If a page requires a real browser (dynamic content), the Playwright-based fetch should handle it; otherwise, a faster HTML fetch can be used. All extracted text is then ready for indexing.

- **Metadata & Stats:** Extend the parsing step to record metadata like title, author, publish date (if available), and content stats (e.g. word count, language). Determine each page's "function" or type if possible (perhaps via simple rules or ML – e.g., identify if it's a blog post, documentation, forum, etc., based on URL or content). This metadata will be stored in the index for later filtering and for explaining results.

- **Indexing into Local DB:** Once content is extracted, it needs to be indexed and stored:

- **Whoosh BM25 Index:** Rebuild or repair the Whoosh index for full-text search. All newly normalized pages should be added to the Whoosh on-disk index [7] . If Whoosh segments got corrupted or stale, consider clearing and re-indexing everything (there are CLI tools like `make reindex-incremental` [8]). Going forward, use the incremental indexing pipeline (DuckDB metadata helps track what's new [9]) to add only new or updated pages.

- **Vector Index (Chroma or alternative):** Restore the vector index functionality so that each page's content is embedded by the LLM and stored. Currently the system uses a local Chroma DB for embeddings [7]. Check that the `VectorIndexService` in the backend is running: it should chunk pages into passages, call the embedding model, and upsert vectors [10] [11]. Fix any issues with model availability (e.g. ensure `embeddinggemma` model is downloaded and loaded via Ollama).
- **Local Storage:** Verify that the content and indices are indeed stored locally: e.g. `data/crawl/` for raw HTML, `data/normalized/normalized.jsonl` for text, `data/whoosh/` for Whoosh index, `data/chroma/` for vector store, and metadata in DuckDB/SQLite [12]. If any of these directories are missing or modules not writing to them, update the paths or code so that all data is persisted. This ensures the search engine has a knowledge base to draw from on subsequent queries.

- **Search Functionality Testing:** Finally, test the basic search pipeline end-to-end and address any breakages:

- **API Endpoints:** The backend's search endpoint (`/api/search`) should accept a query and return combined results from Whoosh and the vector index [13]. Make sure this endpoint is enabled and not returning errors. If it was disabled for any reason, re-enable it in the Flask app.
- **CLI and UI Search:** Use `make search Q="example query"` to verify CLI search works [14]. Also test through the Next.js UI search bar. If queries return no results despite content existing, debug whether the indexing didn't happen or the search code (Whoosh query parser or vector query) is malfunctioning.
- **Ranking Blend:** Ensure that the default ranking merges keyword and semantic hits. The engine was intended to use Whoosh BM25 for initial candidates and then a vector reranker [2]. Confirm this reranking is in place. If not yet implemented, plan to implement a simple rerank: e.g., take top N Whoosh results and re-sort them by a weighted combination of BM25 score and embedding similarity (or use LLM to score them). This ensures relevant results aren't missed even if wording differs from the query.

By the end of this phase, the system should once again be fully functional in its basic capabilities: any site you visit or crawl gets stored locally, and search queries will retrieve those pages. We next enhance the search experience and performance.

2. Search Engine Behavior Enhancements

With core search up and running, we focus on improving *user experience and result quality* for the search feature. This includes making results more transparent (so the user knows *why* a result was shown), improving ranking with embedding-based scoring, and giving the user control over what gets added to their knowledge base.

- **Result Transparency (Why is this result shown?):** Augment the search UI to display context about each result's relevance:
- **Highlighting Matches:** For keyword (BM25) hits, show the snippet with **highlighted query terms**. This immediately shows which words matched between the query and the page content or metadata.

- **Semantic Relevance Explanation:** For results surfaced via vector similarity, provide a short note like "*semantic match*" or a similarity score. You can even show which snippet of the document had high embedding similarity (perhaps by storing the top-scoring sentence from the embedding search).
- **Metadata in Results:** Display stored metadata such as the page's source (domain), date, or category. For example, label a result as "[Doc] Example Site – *How to install X* (2023)". This metadata gives clues why the page was retrieved (e.g., it's a documentation page).

- **"About this result" Panel:** Consider an expandable section for each result with detailed info: which query terms were found, the BM25 score and vector score, and any tags (like "from your library" vs "newly fetched"). This caters to power-users who want full transparency.

- **Advanced Ranking & Retrieval:** Upgrade the search ranking algorithm to use the best of both keyword and vector retrieval, improving speed and accuracy:

- **Hybrid Search Pipeline:** Continue using **Whoosh** for instant keyword filtering and BM25 scoring, but integrate a high-performance vector similarity search for re-ranking. The current pipeline already blends BM25 with a vector reranker [2] ; we should finetune this:
    - Determine an optimal cutoff (e.g., take top 20 BM25 results, then use the embedding model to compute similarity between the query and each result's content embedding, then reorder).
    - For queries with few or no keyword hits, fall back to pure vector search over the whole index. The system's "smart search" already triggers a focused crawl if results are under a threshold [15] – ensure this threshold logic still works, and also use the vector store to find any semantic matches even if keywords differ.
- **Performance Optimizations:** If Whoosh search becomes a bottleneck with growing data, consider switching to an in-memory full-text search (e.g., using SQLite FTS5 for smaller scale, or a more optimized search library). However, Whoosh should be sufficient for moderate data sizes and keeps the stack simple.

- **Result Score Combination:** Develop a clear strategy to combine scores. For transparency, you can even show a breakdown (e.g., *"score: BM25 12.5 + semantic 0.82"*). A simple linear combination or learning-to-rank approach can be used. The goal is to ensure highly relevant pages (by meaning) rank higher, even if exact keywords differ (thanks to embeddings), while still respecting literal matches.

- **Local Knowledge Toggle (Light vs. Library Mode):** Provide a user control to decide if a browsed page or a search result should be kept in the permanent index:

- **"Incognito" Browsing Option:** In the UI, allow a toggle (e.g., a "Record Browsing" switch). When off, the shadow capture can still fetch the page for the AI to use *temporarily* (for answering a question), but *will not* add it to the long-term index. This addresses use-cases where the user wants to look something up without cluttering their personal knowledge base.
- **Result Inclusion Toggle:** Similarly, in search results or chat answers, if new pages were fetched on the fly (via the focused crawler or LLM agent), present an option like "☆ Save to library" for each such page. If the user clicks it, then keep it in the index; if not, allow it to expire (perhaps still cached for the session, but not stored in `data/normalized` or Whoosh index permanently).
- **Implementation:** Back-end wise, this could be handled by tagging content with a flag (e.g., `metadata: {"temp": true}` for ephemeral content). The indexing pipeline can check this flag – if an item is temporary and the user hasn't saved it, exclude it from Whoosh/Chroma when

rebuilding or set it to auto-delete after session. The front-end should make it clear with an icon or color which results are from the permanent library vs. just fetched.
- **User Prompt for Large Additions:** If a new crawl is about to add a large number of pages (say the user triggers a crawl of an entire site), consider prompting "Index 50 new pages from example.com?" with proceed/cancel options. This keeps the user in control of the knowledge base size.

By implementing these UX improvements, the search engine becomes more user-friendly and efficient: Users will **understand why** results are appearing and can fine-tune what knowledge is retained. The next step is ensuring our AI models are well-integrated to enrich these capabilities.

3. Model Integration (Gemm3, GPT-OSS, EmbeddingGemma)

This phase integrates local LLMs to enhance both the indexing and the querying process. The models **Gemma-3** and **GPT-OSS** will be used for chat/planning, while **EmbeddingGemma** will handle text embedding for the vector index. We ensure that model usage is transparent and that any web content the models consume gets stored for future reuse.

- **Embedding Model (EmbeddingGemma) Integration:** Confirm the embedding pipeline uses the latest EmbeddingGemma model for vectorization:
- **Model Setup:** EmbeddingGemma is a 308M param model optimized for on-device semantic embeddings, yielding state-of-the-art representations for search [16]. Make sure the model is downloaded (via Ollama's autopull or manual) and the backend is loading it as the default embedder [17] [18]. In `EngineConfig`, the `models.embed` should point to `embeddinggemma` by default [19].
- **Index Embedding Process:** When new text is normalized, the vector index service should call the embedder to generate embeddings [11]. If EmbeddingGemma wasn't previously used, minor adjustments might be needed (e.g., ensuring the embedding dimensionality is consistent and that multiple chunks are handled). Test this by indexing a page and then using the vector search API (`POST /api/index/search`) to see if it returns that page for a semantically similar query [20].

- **Quality and Performance:** Leverage EmbeddingGemma's quality – its high MTEB benchmark performance ensures relevant semantic matches [16] [4]. Because it's efficient, embedding generation and search should be fast even on CPU. We should still batch process texts for embedding to speed up indexing. If embedding latency is an issue, consider quantizing the model (EmbeddingGemma supports running in <200MB RAM with quantization [21] [22]). This keeps indexing snappy.

- **Chat/Planner Model (Gemma-3 and GPT-OSS) Integration:** These larger LLMs are used for chat responses and for planning which pages to crawl when needed:

- **Ollama Model Management:** The first-run setup should download Gemma-3 (the primary model) and GPT-OSS (fallback) [23] [24]. Verify this still works. Gemma-3 is presumably a powerful (possibly ~27B) model by DeepMind/Google for general QA, while GPT-OSS is an open-source GPT variant (20B/120B) provided via OpenAI/Ollama [25]. They are large, so ensure the system checks model availability and provides clear status (as the wizard does [26]).

- **Integration Points:** Identify where the LLM is invoked in the app:
  - *Chatbot:* When the user asks a question in the chat UI, Gemma-3/GPT-OSS should generate answers, potentially using retrieved context. Ensure the `/api/chat` endpoint uses the vector index to pull relevant documents (RAG) and passes them along with the question to the LLM. The UI LLM Assist panel should list these models and allow selection [27].
  - *Focused Crawl Planning:* The LLM can help generate additional search URLs (via `seed_guesser.py`) when a query has few results [28] [29]. Make sure this is enabled behind the "Use LLM for discovery" toggle [30]. Test that when toggled on, the planner model (Gemma-3 or fallback) is actually called and returns reasonable URLs, which the crawler then fetches.
  - *Result Re-ranking:* Optionally, the LLM (Gemma-3) could be used to re-rank or summarize top results for very complex queries. This isn't a current feature, but could be added: e.g., take the top 5 results and have the LLM output which one seems most likely to contain the answer (and even generate an answer). This would be similar to ChatGPT's behavior when it cites sources. Such a step should be optional (perhaps on-demand when the user clicks "Ask the AI about these results").
- **Processed Content Storage:** A crucial policy is that **whenever the LLM fetches or uses external content, that content is saved locally**:
  - For example, if the user asks a question and the agent decides to scrape a new webpage for the answer, the fetched page must go through the normal normalize-and-index pipeline so it's added to the knowledge base. The roadmap already indicates that every touched domain is recorded in seeds and every fetched page goes to the index [31] [29]. We should double-check this: the `DiscoveryEngine.discover()` and focused crawl should mark new pages for indexing.
  - In chat flows, if the user provides a URL or the assistant browses (when explicitly allowed), that page's content should be indexed too. We can implement a hook in the `/api/extract` or agent tool execution: after extracting text, call the same `index/upsert` logic as if it were a normal crawl.
  - The benefit is **no content is "seen" by the AI without being retrievable later by the user**. This transparency helps trust and reusability – the next time a similar question is asked, the answer might be direct from the local index without hitting the web again.

- **Transparency in LLM Usage:** Log and perhaps display when the LLM is used for these tasks. For instance, if the LLM added some URLs or re-ranked, note "(LLM assisted)" in the UI. Also handle failures gracefully – e.g., if Gemma-3 isn't loaded, the system should fall back to GPT-OSS or just skip the LLM step, and notify the user that advanced semantic discovery is offline [32] [33].

- **Vector Store Efficiency (FAISS/Qdrant):** To support the above, ensure the vector storage is both **transparent** (understandable how data is stored) and **efficient** in retrieval:

- The current Chroma + DuckDB setup works locally, but we can improve or offer alternatives. **FAISS** is a good choice for an in-process high-performance similarity search library (very fast vector indexing/search in C++ with Python bindings). It would avoid network overhead and can handle large numbers of embeddings quickly. We could integrate FAISS by replacing or augmenting the `VectorStore` class to use a FAISS index (Flat or IVFFlat for larger scale) for similarity search.
- **Qdrant** is another option if we foresee scaling up or needing a standalone vector DB. Qdrant is an open-source, high-performance vector database written in Rust, designed for massive-scale

applications [34]. It could run as a separate service. We might not need that overhead for a single-user environment unless the dataset grows huge (millions of vectors). Still, designing our vector store interface to be modular will let advanced users plug in Qdrant or others. For example, define an abstraction `VectorIndex` with implementations for Chroma, FAISS, Qdrant, etc., and make it configurable.
- **Persistence and Size:** With either FAISS or Qdrant, ensure the vector index persists to disk (FAISS indexes can be saved to file, Qdrant keeps data in its storage). Document how to rebuild the vector index from stored text if needed (since text is the source of truth). This adds transparency — users know that vectors come from their data and can be regenerated if models update.

By integrating these models properly, the search engine gains powerful **semantic understanding**. EmbeddingGemma will ensure relevant semantic matches (it's top in class for multilingual embeddings under 500M [16]), and Gemma-3/GPT-OSS will provide robust conversational and planning abilities. All of this remains *self-hosted*, with no external API calls, aligning with the project's goals [35].

4. Security and Data Policy

Maintaining user trust is paramount, so we strengthen the system's data handling policies. The user must have **full control and knowledge** of what data is stored, especially for potentially sensitive or large content. We implement explicit consent for capturing data and sensible defaults to protect privacy and storage.

- **Explicit Permission for Large Media:** For any large or non-text content, require user confirmation before storing:
- **Detection:** When the crawler or shadow capture encounters large files (e.g. videos, audio, very large PDFs) or pages with big media embeds, the system should pause before downloading or indexing them. For example, if a video file > 100 MB is linked, instead of auto-downloading, record its URL and size in metadata and flag it.
- **User Prompt:** Surface a prompt in the UI (possibly in the Control Center or as a notification ribbon) saying "Large content detected: 200MB video on example.com. Save to your library?" with *Yes/No* options. Only proceed to download/store if user approves. This could also apply to high-resolution images or any payload that would consume significant disk space.
- **Policy Settings:** Add configuration for what "large" means (default threshold, e.g., 50 MB) and allow the user to adjust it. Also allow blanket rules (e.g., "Never auto-save videos" or "Always save PDFs"). These can be part of the global/per-domain policy JSON (similar to shadow policies).

- **Background Handling:** If user consents, perform the download in the background with progress indication, and then index the content (e.g. extract audio transcript for video, OCR for images/PDFs if needed, etc., which are future enhancements). If user declines, the system should skip it and possibly remember that decision (so it doesn't prompt again for the same item).

- **Auto-Capture User Consent (Privacy Toggle):** Ensure that capturing of browsing content (shadow mode) and any AI-initiated web access is **opt-in and clearly indicated**:

- The application already has **global and per-domain privacy policies** that the user can toggle [36]. Build on this by making the controls very visible (e.g., a main toggle in the UI header for "Capture my

browsing for knowledge base"). If this is off, the shadow browser should not call `/api/shadow/snapshot` at all – effectively browsing becomes truly private.
- On first launch or first use of the researcher browser, explicitly ask the user if they want auto-capture enabled. Explain that with it on, visited pages' content will be saved locally for search (and that nothing is sent to third-party servers, for reassurance [35]). The user can then proceed knowing what's happening.
- For the LLM agent, have a separate toggle "Allow AI to auto-search and scrape content" (the "Local Discovery" or agent toggle in Control Center). When off, the AI should not fetch new pages on its own – it will be limited to what's already indexed or will ask the user for permission. When on, it can autonomously browse but still must respect the large content rules above.

- **Audit & Logs:** Provide an easy way for the user to see what has been captured and indexed at a glance. For instance, a log of recent pages indexed (with timestamps and source) available in the UI. This goes hand-in-hand with transparency: the user can review and remove anything they don't want kept. The Control Center could list recently added documents and allow deletion or editing of their metadata.

- **Robots.txt and Ethical Crawling:** Continue to respect `robots.txt` by default during crawling/snapshot unless the user explicitly overrides for a domain [6]. This shows good data hygiene and respect for target websites. In the policy settings, the user can have an override per domain if they need to crawl a site that disallows scraping (with a caution that they assume responsibility).

- **Data Isolation:** Since this is a single-user system (see next section), data is inherently private to that user. We must ensure that if in the future multi-user or sharing features are added, a user's data isn't exposed without consent. For now, clarifying in documentation that all data stays on the user's machine and under their control reinforces trust.

By enforcing these policies, we make sure the search engine behaves like a personal, privacy-respecting assistant. The user is never surprised by large downloads or unwitting data hoarding, and they have *complete control* over what information is stored locally.

5. Single-User Scope

This project is intentionally scoped for a **single user** (or a single-tenant environment). We should confirm and maintain this focus, which simplifies design and avoids unnecessary complexity:

- **No User Accounts or Auth:** The system doesn't require login or multi-account management – it runs locally for one user. We will not add any multi-user login or role-based differences. All features (crawler, index, etc.) operate under the assumption that there's a single trusted operator. This keeps the interface straightforward and avoids having to partition indexes per user.
- **Localhost Services Only:** The backend API (Flask) and front-end are accessible to the single user (often on `localhost`). Ensure it stays binding to local interface by default (for security). If remote access is needed, it should be a conscious decision by the user to expose it. In code/config, the default Flask host can remain `127.0.0.1` [37] and same for Next.js client calls.
- **Simplified Concurrency:** Since only one user (and likely one primary session) is using the system, we don't need to design complex concurrency or isolation layers. The SQLite and file storage can be accessed without per-user schemas. Background tasks (crawls, indexing) will be triggered by that

user's actions only. We just need to ensure thread-safety for those background tasks (which the current code likely handles via locks in the index service [38] [39]).

- **Profile Management (if any):** In case the user wants separate profiles (e.g., work vs personal indexes), an easy way would be to allow running multiple instances with different data directories. We don't introduce multi-tenant profiles within one running instance, as that adds overhead. Document how to start the app with an alternate data path if needed instead (perhaps via an environment variable or config to point to a different `data/` folder).
- **Testing Under Single User:** We can simplify certain checks knowing there's no untrusted input from other users. For example, any admin interfaces or diagnostics can be accessible without auth (the user runs it on their machine). This simplifies development and testing – focus on functionality, not on user management.

By explicitly keeping the scope to one user, we reduce potential bugs and security issues. The design can assume full trust of the operator, which is appropriate for a self-hosted tool. All improvements in other sections are thus made with this single-user assumption (no need for multi-user data segregation, etc.).

6. Libraries and Stack Considerations

Lastly, we choose **minimal, fast, and reliable tools** for each component, in line with the project's philosophy of avoiding heavy external dependencies [35]. We will stick mostly to the current stack, making swaps only if they markedly improve performance or simplicity:

- **Crawling & Scraping:** Continue with **Scrapy** for web crawling, as it's a proven framework for large-scale crawling and scraping [1]. Scrapy's efficiency in scheduling and throttling will help as the user's crawl frontier grows. For single-page captures and dynamic content, use **Playwright** (already in use for shadow browsing) in headless mode. This combination (Scrapy for breadth, Playwright for tricky pages) covers most needs. Both are Python-based and run locally.
- **Content Parsing: Trafilatura** is currently used for HTML to text extraction, which is lightweight and effective for readability. Keep using it for normalization. Optionally, integrate *BeautifulSoup4* for parsing specific elements if needed (BS4 is also lightweight). For PDFs or other formats, Python libraries like PDFMiner or PyMuPDF can be added when needed – but only on demand to avoid bloat.
- **Text Index (Whoosh):** Whoosh, being pure Python, is easy to embed and requires no external service. It is sufficient for personal use scale (hundreds to low-thousands of documents). We will keep Whoosh for its simplicity and because it's already integrated into the CLI and code. In the future, if performance becomes an issue, an alternative is an SQLite FTS5 index (which could leverage SQLite already in use). However, sticking to Whoosh for now avoids rewriting query logic.
- **Vector Store:** Chroma with DuckDB is already set up, but as discussed, we will abstract this. For a minimal stack with high speed, **FAISS** is a strong candidate – it's a single library (C++ core) and can be used in-memory. Installing FAISS (especially CPU-only) is not heavy, and it avoids running a separate service. **Qdrant** is another powerful option but would add a Rust service running alongside; we might suggest it as an advanced configuration (perhaps for power users who want to scale up) [34]. By default, we can continue with the embedded DuckDB/Chroma or switch to FAISS for speed. Both options keep data local and query latency low.
- **LLM Runtime:** Rely on **Ollama** as the serving backend for Gemma-3, GPT-OSS, and EmbeddingGemma. This removes the need to integrate each model's runtime in our code – we just make HTTP calls to Ollama on `localhost` which is straightforward. Ollama is a minimal dependency from the user perspective (just a binary to run these models). Make sure to document

that Ollama is required for LLM features and provide instructions to get the models (the first-run wizard already helps with this [26]).

- **Backend Framework:** Flask is being used and is lightweight for an API-only backend. We continue with Flask as it's sufficient for single-user request loads. Since Flask is synchronous, ensure that long-running tasks (crawling, indexing) are run in background threads or subprocesses to not block the main thread (the current design likely does this via CLI tools and monitoring logs). This is fine given Python GIL because heavy tasks (IO-bound crawling, or releasing GIL in embedding calls) can still run concurrently. No need for a heavier async framework unless we encounter specific scaling issues.
- **Frontend:** The Next.js frontend (with Tailwind and shadcn/UI) is already in place and provides a rich UI. We keep this stack as is. It's decoupled from backend except via the REST API. One improvement is to ensure the UI remains **responsive** even during heavy background activity: use web workers or the existing status polling to inform the user of progress (which the Status ribbon already does [40] [41]).
- **Dependencies Audit:** Go through `requirements.txt` or equivalent and confirm each dependency is needed. Remove anything unused to slim the install. The target is a lean installation. All chosen libraries (Scrapy, Whoosh, etc.) are reasonably lightweight and open-source. Avoid adding any closed-source or cloud-tied library (consistent with being self-hosted). For instance, do not integrate something like Pinecone (cloud vector DB) – we stick to local solutions.

In summary, the stack will remain **Python-based and self-contained**: Scrapy, Flask, Whoosh, DuckDB/FAISS, and local models via Ollama. Each component is chosen for being resource-efficient for personal use. This minimalistic approach ensures the system runs smoothly on a typical laptop or desktop without requiring powerful servers, aligning with the on-device philosophy [21].

By executing this plan in phases, we will restore the search engine's full functionality and enhance it with modern capabilities. The end result will be a **powerful personal search assistant** that automatically learns from everything the user reads, provides intelligent search results with explanations, and leverages local AI models – all while keeping the user in control of their data. Each module is designed to be modular and maintainable, setting a strong foundation for future features on this self-hosted platform.

Sources:

- Project README and docs for current design and features [42] [15] [6]
- Google Developers Blog on EmbeddingGemma for embedding model capabilities [16] [4]
- Qdrant documentation for vector DB performance claims [34]
- Scrapy documentation for crawling framework details [1]

---

[1] Scrapy 2.13 documentation — Scrapy 2.13.4 documentation
https://docs.scrapy.org/en/latest/

[2] [3] [5] [6] [7] [8] [9] [12] [13] [14] [15] [20] [23] [24] [26] [27] [28] [29] [30] [31] [32] [33] [35] [36] [37] [40] [41] [42]
README.md
https://github.com/sulaimonao/self_hosted_search_engine/blob/0944644e8ce162bd136d1bb6c3af9332fc0ce611/README.md

4 16 21 22 Introducing EmbeddingGemma: The Best-in-Class Open Model for On-Device Embeddings - Google Developers Blog
https://developers.googleblog.com/en/introducing-embeddinggemma/

10 11 17 18 19 38 39 vector_index.py
https://github.com/sulaimonao/self_hosted_search_engine/blob/0944644e8ce162bd136d1bb6c3af9332fc0ce611/backend/app/services/vector_index.py

25 gpt-oss - Ollama
https://ollama.com/library/gpt-oss

34 Qdrant - Vector Database - Qdrant
https://qdrant.tech/