ChatGPT

Enhancing Local Deep Researcher with MCP-Based Retrieval & Evaluation

Multi-Strategy Retrieval as a Quick Win

One immediate improvement is to incorporate **multiple retrieval strategies** side-by-side, as demonstrated in the advanced RAG projects. The `adv-rag` repository implements *six* different retriever methods – e.g. naive vector search, BM25 keyword search, LLM-based reranking (contextual compression), multi-query expansion, ensemble (rank fusion), and semantic chunking [1]. Similarly, the *LangChain Retrieval Methods* notebooks explore **seven** strategies (adding a "parent document" retriever for hierarchical chunking) [2]. By adopting these **plug-and-play retrievers**, the local researcher app can let users or agents choose the best approach per query. For example, simple factoid questions might use the fast BM25 or naive vector retriever, whereas complex queries benefit from multi-query expansion or ensemble results [3]. This multi-strategy support can be layered in incrementally: start by adding a BM25 retriever (e.g. using a lightweight IR library or Lucene) alongside the existing vector search, then introduce more advanced retrievers (multi-query, rerankers) as needed. Each retriever can follow the **factory pattern** as in `adv-rag` – a central retriever module instantiates the chosen strategy without altering the core pipeline [4] [5]. The key is that all retrievers share the same interface (LangChain's `Retriever`), so swapping or comparing them is straightforward. This enables **benchmarking of retrieval quality** and helps identify "quick win" improvements (for instance, adding an LLM reranker drastically improved context precision in tests [6]). Exposing this in the UI could be as simple as a dropdown of retrieval modes or a toggle to run **all strategies in parallel** for a side-by-side answer comparison [7] [8]. Early on, logging each retriever's top results and timing will build intuition on which strategy excels for the user's data.

Dual Interface via MCP (Command vs. Query Pattern)

To integrate these retrieval tools into intelligent agents, we can mirror the **dual-interface architecture** from `adv-rag`. This design uses **CQRS (Command Query Responsibility Segregation)** to separate "full answer" operations from raw data fetches [9]. In practice, this means running two modes: a **Command mode** (MCP Tools) that executes the complete RAG pipeline (retrieve relevant docs, then have the LLM synthesize a final answer with citations), and a **Query mode** (MCP Resources) that performs a lightweight retrieval-only call returning documents or snippets [9] [10]. The same underlying code serves both, avoiding duplication, by automatically converting standard API endpoints into MCP-compatible agent tools [11] [5]. For the local researcher, this could be a **"research" tool** vs. **"lookup" resource**: e.g. an agent using the app can either ask *"Answer this question with full context"* (which triggers the chain and yields a nicely formatted answer) or do *"Find relevant documents for X"* (which quickly returns raw text chunks for further analysis). Implementing this dual mode is feasible as an incremental step: one can wrap existing LangChain QA chain in a FastAPI endpoint (if not already) and then use a library like **FastMCP** to expose it as a tool automatically [12]. In parallel, a leaner endpoint that skips the LLM step (just calls the vector store or search index) can be exposed as a read-only resource URI (e.g. `retriever://bm25/{query}`) [13]. The benefit, confirmed by the advanced RAG system, is significant performance optimization – raw retrieval calls return ~**3-5× faster** (a few seconds) whereas full generation might take 20+ seconds [10]. This MCP-based pattern means the local app can act as an **agent toolbox**, enabling external AI agents or a chain-of-thought to invoke domain-specific retrieval when needed. For UI integration, the application could offer a **“fast mode”** for just retrieving sources (for power users who want to read the references themselves) versus a **“complete answer mode”** for normal Q&A usage. Under the hood it’s the same pipeline, just skipping or including the final LLM answer step as appropriate [14] [15]. Deploying this doesn’t require separate codebases – just run separate entry points or servers for each interface. The `adv-rag` example shows how a single codebase can spawn both a REST API and an MCP agent interface with zero duplication [16] [17].

Orchestration Patterns: CQRS, Chunking, and Ingestion Workflows

Adopting some proven orchestration patterns will make the system more robust and scalable. **CQRS** was already touched on above – beyond splitting the interface, it encourages thinking in terms of **commands (write/full compute)** vs **queries (read-only)**. The advanced RAG project strictly keeps the retrieval logic in a core module and treats the MCP/FastAPI layers as an interface-only layer [18] [19]. We should enforce a similar separation: the LangChain chains (vector search, rerank, etc.) stay in one place, and the UI/agent calls simply trigger them. This makes it easier to add new strategies or swap vector databases without changing how the UI or agent interacts. Another pattern is optimizing **document chunking and indexing.** The second repository and related guides emphasize that chunk size and strategy critically affect retrieval effectiveness [20]. A quick enhancement is to implement **semantic chunking** or a hierarchical chunking approach. For example, one could index documents in overlapping chunks or by paragraphs but also store a “parent document” pointer to allow retrieving the broader context [21]. This *parent document retriever* pattern (used in the John Wick evaluation) ensures that if a small chunk is retrieved, the system can easily pull in its neighbors or original document for additional context [1]. Similarly, **context-aware splitting** (avoiding splitting mid-topic or sentence) can be achieved using LangChain’s `RecursiveTextSplitter` or even ML-based segmentation [20]. Incrementally, the app can start with its existing splitting and later integrate these advanced chunking strategies (perhaps configurable per dataset). It’s also wise to formalize the **ingestion workflow**. In `adv-rag`, an ingestion script reads a CSV of movie reviews and indexes them into Qdrant (vector DB) and sets up any BM25 index [22]. We can generalize this: create a pipeline (could be a script or notebook) where new data sources (PDFs, Markdown, etc.) are loaded, chunked, and inserted into the chosen stores. This pipeline can track metadata like collection names, index sizes, and embedding model used. By making ingestion repeatable, one can update the knowledge base or rebuild indexes without manual steps. For example, an **“Ingest Data”** button in the UI might run this pipeline (with progress feedback) to load new documents. On the orchestration side, consider adding **caching layers** and **error handling** too. The advanced RAG system includes Redis caching to speed up repeated queries [23] – integrating a cache for embeddings or LLM outputs could be a quick win for performance if the same questions or documents are accessed frequently. Proper error trapping (e.g., if one retriever fails or API key is missing) with fallback to simpler methods will improve reliability, as those repositories followed **production-ready patterns with error handling and monitoring** [24].

Evaluation Framework Integration (RAGAS & Phoenix)

To ensure these retrieval enhancements actually add value, integrating an **evaluation framework** is crucial. Both referenced projects highlight using **RAGAS** (Retrieval Augmented Generation Assessment) to quantitatively evaluate QA performance. RAGAS provides LLM-based metrics that judge different aspects of the QA pipeline [25] [6]. We should set up a small **“golden test set”** of question-answer pairs (the John Wick example used a synthetic test set generated by GPT-4 [26]). Then, after the system answers these questions, we run RAGAS to compute metrics like **Answer Relevancy** (does the answer actually address the question?),

Context Precision (proportion of retrieved content that was relevant), **Context Recall** (did we retrieve all the relevant info?), and **Faithfulness** (is the answer supported by the context, i.e. no hallucinations) [6]. These metrics – often output as scores per query – are valuable data artifacts to expose via a dashboard. For instance, we could present an aggregate RAGAS score or each metric for every test query in a React UI table or chart. This would allow the developer (or even end-users, if appropriate) to **see the impact of switching retrieval methods** or changing prompts. An example result from the advanced retriever comparison: the multi-query retriever had the highest context recall (catching more relevant info), while an ensemble method led in overall answer relevancy [27]. Such insights can drive iterative improvements. Technically, incorporating RAGAS can be done in steps: initially, run evaluations offline in a notebook or script, then later automate it as a periodic job or on-demand through the app. The integration with **LangSmith** (LangChain's experiment tracker) in the retrieval methods repo is another angle – it allowed logging each run's metrics and visualizing them on a hosted dashboard [26] [28]. If the local researcher app can connect to LangSmith (or a similar service), it could push evaluation results there for a richer UI. Alternatively, for a self-hosted approach, we might use **Arize Phoenix** – an open-source LLM telemetry tool – as in `adv-rag`. Phoenix was run alongside the app (via Docker) to capture traces and metrics of each query in real time [29] [30]. With Phoenix, one can inspect each step of an agent's reasoning or each retriever's output in a web UI (at `localhost:6006` in their setup [31]). We could integrate Phoenix by emitting MCP telemetry events or using their Python client. For example, after running the retrieval comparison script, `adv-rag` logged all results to Phoenix and confirmed through its UI that all strategies were performing as expected (100% correctness on the test questions, latencies under target) [32]. Incorporating Phoenix incrementally might start with using it in a development environment for debugging and **"AI agent observability"**, as the README calls it [33]. Eventually, the app's front-end could even call a backend endpoint to fetch summary stats from Phoenix or RAGAS runs – for instance, "show me the last evaluation run metrics". An early UI idea is a simple **metrics dashboard page**: after running an eval, the page displays the scores for each metric per retriever, maybe with green/red highlighting, and perhaps a link to detailed logs (if using Phoenix or LangSmith) [34]. This gives a **React/Vercel-style** slick view into how well the system is answering questions. Even exposing the raw JSON of RAGAS results via an API route (for power users) would be useful. The main point is to treat evaluation data as first-class output of the system – not buried in a console – so improvements can be driven by evidence.

UI Integration & Deployment Feasibility

All these enhancements should be layered thoughtfully into the UI and deployment pipeline. On the **frontend**, we can start with minimal changes: e.g. add a selector for retrieval strategy when asking a question (default to "Auto" or the current method for simplicity). This allows easy A/B comparison by the user. As more MCP tools are added, a user (or an AI agent) could have a list of tools like "Search (fast)" versus "Answer (detailed)" – in a chat interface, this might appear as different actions the assistant can take. A React-based UI could also incorporate an **evaluation view** as described: perhaps a tab showing "Evaluation Results" with a table of metrics or even charts (a bar chart of each retriever's RAGAS score, etc.). Many chart libraries could consume the JSON output from a RAGAS run and display interactive graphs. If a Vercel-style deployment is intended (serverless frontend with an API backend), we must ensure the backend can handle the heavy lifting (LLM calls, vector DB queries, etc.). It might be impractical to run something like Phoenix in a serverless environment, so one approach is to keep Phoenix and vector databases as optional local Docker services for development and self-hosted scenarios [35] [36]. In production, using managed services (like a hosted Qdrant or Pinecone for vectors, Redis Cloud for caching, and possibly LangSmith cloud for eval tracking) could replace those components. The **deployment feasibility** of each feature varies: adding pure retrieval strategies and even the FastAPI/MCP dual interface is just code, which deploys wherever the app runs. Meanwhile, RAGAS evaluation requires access to an LLM (e.g. GPT-4) to score outputs, which means API costs and latency – this might be run on-demand or in batch rather than for every user query. We should document these requirements in the README (e.g. needing OpenAI API keys and possibly Cohere for reranker as in adv-rag [37] [38]). For containerized deployment, we can extend the `docker-compose.yml` to include services like Qdrant, Redis, and Phoenix, similar to the reference setup [39]. Each addition can be toggled based on environment (for example, spin up Phoenix only in a debug mode). **Incremental rollout** is key: we might first deploy the multi-retriever version (ensuring the new indexes like BM25 are built and working), then add the MCP agent endpoints in the next version (allowing power users to connect AI agents like Claude or GPT-4 to the tool interfaces [40]), and later integrate the evaluation dashboard once enough data is gathered. By layering features one by one – and using the metrics to validate each improvement – the LangChain local-deep-researcher will steadily evolve into a more **powerful, observable RAG system**. It will feature flexible retrieval plug-ins, an agent-friendly MCP interface, robust evaluation feedback loops, and a UI that not only answers questions but also gives insight into *how* those answers are produced and how well the system is performing. This aligns with the best practices shown in these repositories: **start simple, instrument everything, and iterate with evidence-based tweaks** [41] [42]. With these reusable components and workflows, the application can achieve production-grade retrieval augmentation and continuously improve via measured outcomes.

Sources:

- Donbr, *Advanced RAG with Dual MCP Interface Architecture* (GitHub README) [11] [9] [10]
- Donbr, *LangChain Retrieval Methods* (Notebooks & README) [2] [6]
- *TheDataGuy – Evaluating Advanced RAG Retrievers: A Practical Comparison* (2025) [43] [26]
- *LangChain Blog – Evaluating RAG pipelines with RAGAS + LangSmith* (2023) [25] (context on RAGAS metrics)

---

[1] [3] [6] [20] [21] [26] [27] [28] [34] [41] [42] [43] Evaluating Advanced RAG Retrievers: A Practical Comparison | TheDataGuy
https://thedataguy.pro/blog/2025/05/evaluating-advanced-rag-retrievers/

[2] [35] [36] retrieval_method_comparison_langsmith.ipynb
https://github.com/donbr/langchain-retrieval-methods/blob/8a90000af52777171e74603a840bde776f183543/notebooks/retrieval_method_comparison/retrieval_method_comparison_langsmith.ipynb

[4] [5] [12] [16] [17] [18] [19] CLAUDE.md
https://github.com/donbr/adv-rag/blob/7d29defec71bf30e0a1242b3ee085d25efef8a3e/CLAUDE.md

[7] [8] [9] [10] [11] [13] [14] [15] [23] [24] [29] [31] [33] [37] [38] [40] README.md
https://github.com/donbr/adv-rag/blob/7d29defec71bf30e0a1242b3ee085d25efef8a3e/README.md

[22] [30] [32] [39] PHOENIX_MCP_VALIDATION_ANALYSIS.md
https://github.com/donbr/adv-rag/blob/7d29defec71bf30e0a1242b3ee085d25efef8a3e/docs/PHOENIX_MCP_VALIDATION_ANALYSIS.md

[25] Evaluating RAG pipelines with Ragas + LangSmith
https://blog.langchain.com/evaluating-rag-pipelines-with-ragas-langsmith/