ChatGPT

Technical Design Document: Auto-Analyst

This document details the architecture, components and data flow of the **Auto-Analyst** system. It emphasises the use of free models, open-source libraries and evaluation methodologies.

System Architecture

Auto-Analyst follows a modular, agent-orchestrated pipeline. Each component is responsible for a specific stage of the research process. The workflow is managed using **LangGraph**, a stateful orchestration framework that supports persistent memory, cyclical workflows and streaming [1].

High-Level Components

Component	Description
Planner Agent	Decomposes the user's question into sub-queries and defines a plan for retrieval.
Search Tool	Uses free APIs (DuckDuckGo, Wikipedia, SearxNG) to find relevant web pages. Respects `robots.txt` and rate limits.
Fetcher & Parser	Downloads web pages and PDFs, then extracts text using BeautifulSoup and pdfplumber. Splits long documents into overlapping chunks.
Embedder & Vector Store	Converts text chunks into embeddings using a free sentence-transformers model. Stores embeddings in a local vector database (ChromaDB or FAISS).
Retriever	Performs similarity search on the vector store to find the most relevant chunks for a given query.
Answer Generator	Feeds the query and retrieved context to a local LLM (Mistral/Llama/Phi-3) to produce a draft answer with inline citations.
Verifier Agent	Re-asks the model to critique the draft answer and removes unsupported claims. Ensures each statement is grounded in the retrieved context.
Streamlit UI	Presents the interface for entering questions, adjusting the number of sources and viewing results. Manages session state.

Data Flow

1. **Query Input:** A user submits a research question via Streamlit.
2. **Planning:** The planner agent breaks the question into search tasks (e.g., summarise key impacts, locate statistics). It outputs one or more search queries.
3. **Retrieval:** The search tool queries free sources and returns a list of URLs. The fetcher downloads these pages, extracts and cleans the text, and splits it into chunks.
4. **Embedding:** Each chunk is embedded into a high-dimensional vector using sentence-transformers. Chunks, along with metadata (URL, title, index), are upserted into ChromaDB/FAISS.

5. **Context Selection:** Given the user query, the retriever computes embeddings and selects the top-K relevant chunks via cosine similarity.
6. **Answer Generation:** The generator receives the query and context. It constructs a prompt that includes instructions to cite sources. The model outputs a draft answer with numbered citations corresponding to the retrieved chunks.
7. **Verification:** A separate verifier agent critiques the draft answer, checking for unsupported claims. It modifies or removes statements that lack corresponding context. The final answer and citation list are returned.
8. **User Presentation:** The Streamlit app displays the final answer with inline citations, along with an expandable list of sources (URL, title, snippet).

Use of LangGraph

LangGraph's graph-based architecture gives the system fine-grained control over workflow. Nodes represent functions (plan, search, fetch, embed, retrieve, generate, verify), while edges determine how state flows between nodes. LangGraph's persistent state allows the system to remember previous interactions, enabling session-level memory and retry logic. It supports loops, enabling agents to re-evaluate results and correct themselves [1]. Native streaming provides token-by-token updates, improving responsiveness [2].

Technology Choices

- **Models:** Use open-source LLMs (e.g., Mistral-7B Instruct, Llama-3.1 8B, Phi-3) and sentence-transformers embedding models.
- **Search:** DuckDuckGo, Wikipedia API, SearxNG (self-hosted). All are free and do not require API keys.
- **Vector Store:** ChromaDB for local development; FAISS as an alternative.
- **Orchestration:** LangGraph orchestrates the pipeline; it offers persistence, streaming and human-in-the-loop features [1] [2].
- **UI:** Streamlit is used to build a simple web interface with session memory.

Evaluation Strategy

Evaluating RAG systems requires assessing both the **retriever** and **generator**. Five key metrics are widely used [3]:

1. **Context relevance:** Measures how relevant the retrieved context is to the user query [3].
2. **Context sufficiency:** Assesses whether the retrieved context contains enough information to answer the question [3].
3. **Answer relevance:** Evaluates how well the generated answer addresses the user's question [3].
4. **Answer correctness:** Rates the factual accuracy of the answer [3].
5. **Answer hallucination:** Detects unsupported or fabricated statements in the answer [3].

Additional metrics (e.g., BLEU, ROUGE) may be referenced but are generally less suitable for RAG due to their reliance on exact word matches [4]. Instead, embedding-based similarity and LLM-as-a-judge approaches are recommended [5].

Security & Ethical Considerations

- **Robots.txt compliance:** The fetcher respects site restrictions. Only freely accessible and non-sensitive data is retrieved.

- **Data privacy:** No user data is stored beyond the session. The vector store contains only public information.
- **Responsible AI:** The verification agent ensures that only supported statements appear in the final answer. Human-in-the-loop review can be added for high-stakes domains.

---

1 2 A Developer's Guide to LangGraph for LLM Applications | MetaCTO
https://www.metacto.com/blogs/a-developer-s-guide-to-langgraph-building-stateful-controllable-llm-applications

3 4 5 RAG Evaluation Metrics: Best Practices for Evaluating RAG Systems
https://www.patronus.ai/llm-testing/rag-evaluation-metrics