NotebookLM Local – Development Journal

-by Abhay Singh

Building a Fully Offline AI Research Assistant

The idea behind **NotebookLM Local** was simple: create an AI-powered research assistant that gives users the experience of Google's NotebookLM while keeping every document, embedding, conversation, and model completely on their own computer. Privacy was the primary design principle from the beginning. No files should leave the user's machine, and the application should continue to function even without an internet connection.

While the final application appears straightforward from the user's perspective—upload documents, ask questions, receive intelligent answers—the engineering journey involved solving several architectural and performance challenges. This journal documents the major technical decisions, problems encountered, and the reasoning behind the solutions.

Technology Stack and Design Decisions

Choosing the technology stack was one of the first major architectural decisions because every component had to support a fully local workflow.

Backend – FastAPI (Python)

The backend was built using **FastAPI**. Since almost every modern AI library—including local LLM frameworks, embedding models, OCR

libraries, and vector databases—is built around Python, choosing another language would have created unnecessary complexity.

FastAPI provided several advantages:

- High performance through asynchronous request handling
- Automatic API documentation
- Easy integration with AI and machine learning libraries
- Clean dependency injection
- Simple deployment and testing

Using FastAPI allowed the backend to simultaneously handle document processing, embedding generation, vector searches, SQLite operations, and communication with the frontend without becoming a bottleneck.

Frontend – Vite + Vanilla Web Components

Instead of using React, Angular, or Vue, the frontend was intentionally built using **Vanilla JavaScript with Web Components**.

Initially, React was considered because of its ecosystem, but the application itself does not require complicated state management or thousands of dynamic UI updates. Introducing a large framework would have increased bundle size, startup time, and overall complexity.

Using native Web Components provided:

- Extremely fast startup
- Minimal JavaScript bundle size
- Native browser support
- Better control over rendering
- Easy component isolation

Combined with Vite, development became significantly faster thanks to hot module replacement and near-instant rebuilds.

The result is a lightweight interface with a modern glassmorphism-inspired design that feels similar to native desktop applications.

ChromaDB – Local Vector Database

One of the core features of NotebookLM Local is semantic search.

Instead of relying on cloud-based vector databases such as Pinecone or Weaviate, the project uses **ChromaDB** because it stores embeddings directly on the user's computer.

This approach offers several advantages:

- Complete privacy
- No API costs
- Persistent storage
- Fast local retrieval
- Easy metadata filtering

Every uploaded document is converted into embeddings and stored locally, allowing the application to perform semantic searches even without an internet connection.

SQLite + SQLAlchemy

Although ChromaDB stores embeddings, another database was required for structured application data.

SQLite was selected because it is:

- Lightweight
- Serverless
- Cross-platform
- Reliable
- Easy to distribute inside desktop applications

SQLAlchemy's asynchronous ORM simplified database interactions while keeping the codebase organized.

SQLite stores:

- Chat sessions
- Individual messages

- Uploaded document metadata
- Application settings
- File relationships

Using a relational database made maintaining conversation history significantly easier than storing everything as JSON files.

Electron + PyInstaller

Creating an application that anyone can install was one of the biggest goals.

Initially, Tauri was evaluated because of its smaller application size.

However, packaging a Python backend inside Tauri proved significantly more difficult due to Python dependency management.

Electron ultimately became the better choice because it provided:

- Mature packaging ecosystem
- Excellent cross-platform support
- Easy process management
- Reliable communication with the backend

PyInstaller was used alongside Electron to package the entire Python environment—including large dependencies such as Torch, sentence-transformer models, and ChromaDB—into a standalone executable.

This allows end users to launch the application with a simple double-click instead of manually starting multiple servers.

Engineering Challenges

Challenge 1 – Building an Effective Information Retrieval Pipeline

One of the earliest prototypes followed the simplest possible approach.

Whenever a user uploaded a PDF, the entire extracted text was stored as a single document inside the vector database.

Although this seemed straightforward, the retrieval quality was extremely poor.

Large documents often contain hundreds of pages covering multiple topics. When embedded as one large block, the resulting embedding represented the document as a whole rather than individual concepts.

Consequently, semantic search frequently returned overly broad or partially relevant results.

For example, asking about "binary search trees" inside a 300-page data structures textbook often retrieved an embedding representing the entire textbook rather than the chapter discussing BSTs.

This significantly reduced answer quality.

Solution

The retrieval pipeline was redesigned completely.

Instead of embedding entire documents, every uploaded file goes through several preprocessing stages.

The document is first cleaned to remove unnecessary whitespace and formatting artifacts.

It is then divided into semantically meaningful chunks based on paragraphs and sentence boundaries.

Each chunk is assigned metadata including:

- Source file
- Page number

- Chunk index
- Section title (when available)

Individual embeddings are generated for every chunk and stored separately inside ChromaDB.

During question answering, the user's query is embedded and compared against every stored chunk.

Only the highest-ranking chunks are retrieved.

Additional metadata filtering ensures that retrieval can be restricted to selected files when necessary.

This dramatically improved retrieval precision while reducing irrelevant context being sent to the language model.

Challenge 2 – Managing the LLM Context Window

One of the biggest architectural challenges during development was handling the limited context window of Large Language Models (LLMs). Every model has a maximum number of tokens it can process in a single request, and this limit must accommodate everything sent to the model: the system prompt, retrieved document context, conversation history, and the user's latest question.

During the initial implementation, the prompting strategy was very straightforward. For every user query, the backend assembled a prompt containing the complete conversation history, the relevant uploaded documents, the system prompt, and the user's current message. While this approach worked well for short conversations, it quickly became problematic as chats grew longer or users uploaded large documents.

As more messages accumulated, the prompt size increased continuously. Eventually, the total number of tokens exceeded the model's context window, causing requests to fail with **"context length exceeded"** or **"token limit exceeded"** errors. Even before reaching the hard limit, performance degraded noticeably. The language model spent a significant portion of its context processing old conversation that was no longer relevant to the current question. Response generation became slower, memory consumption increased, and the model occasionally lost track of the discussion, resulting in incomplete or hallucinated answers.

It became clear that sending the entire conversation to the model after every user message was neither scalable nor efficient. The challenge was to preserve long-term conversational memory without continually increasing the prompt size.

Solution – Dynamic Context Injection with Rolling Conversation Summaries

To address this problem, the prompt construction process was completely redesigned. Instead of treating every previous message as equally important, the backend now dynamically selects and compresses the information that is actually useful for answering the current query.

Every prompt sent to the language model is composed of four carefully managed sections:

1. **System Prompt** – A fixed instruction defining the assistant's behavior, response style, and constraints. This ensures the model maintains consistent behavior throughout the conversation.
2. **Retrieved Knowledge (RAG Context)** – Rather than sending entire documents, the backend performs a semantic similarity search on ChromaDB using the user's latest question. Only the top-ranked document chunks (typically the 3–5 most relevant) are injected into the prompt. This ensures the model receives only information directly related to the user's current query while avoiding unnecessary token usage.
3. **Conversation Memory** – Instead of including the complete chat history, the application employs a **rolling summary mechanism** to preserve long-term memory efficiently.
   As conversations grow, older messages are periodically compressed into a concise summary using the language model itself. This summary captures the important facts, decisions, user preferences, previous conclusions, and ongoing discussion topics while discarding repetitive dialogue and conversational filler.
   Once generated, the summary is stored persistently in the SQLite database alongside the chat session. The original messages remain archived for record-keeping, but they are no longer repeatedly sent to the model with every request.
   During subsequent interactions, the backend loads:
   - the latest rolling summary,
   - only the most recent *N* conversation messages (for example, the last 8–12 exchanges),
   - and the current user question.
4. As the conversation continues and newer messages accumulate, another summary is generated that incorporates the previous summary along with the newly completed discussion. This effectively creates a **rolling summary**, where each summary builds upon the previous one. Instead of summarizing the entire conversation from scratch every time, the application continuously updates a compact representation of the chat history.
   This approach allows the assistant to retain context from conversations spanning hundreds of messages while consuming only a small, nearly constant number of tokens.
5. **Current User Query** – Finally, the user's latest question is appended. Since the prompt already contains relevant document context, recent dialogue, and the rolling conversation summary, the language model has sufficient information to generate an

informed response without needing the entire conversation history.

Benefits of the Rolling Summary Approach

The rolling summary strategy offers several important advantages over simply truncating old messages.

Most importantly, it preserves long-term conversational memory. Important facts discussed hours or even days earlier remain available to the model, even though the original messages are no longer included in the prompt.

It also keeps the prompt size relatively constant regardless of how long the conversation becomes. Whether the user has exchanged ten messages or one thousand, the amount of conversational context sent to the model remains manageable.

Another advantage is improved response quality. By removing repetitive dialogue and retaining only meaningful information, the model spends its context window focusing on relevant knowledge rather than processing redundant text. This leads to faster inference, lower memory usage, fewer hallucinations, and more coherent responses over extended conversations.

Overall, combining semantic retrieval from ChromaDB with dynamic context injection and rolling conversation summaries enabled NotebookLM Local to support persistent, long-running research sessions while remaining within the strict context limits imposed by modern language models.

Challenge 3 – Persistent Conversations and State Management

Early versions of the application stored conversations only in memory.

Closing the application meant losing every uploaded document association and every previous conversation.

This made the assistant unsuitable for long-term research.

Researchers often revisit projects over several weeks or months, making persistent conversation history essential.

Solution

SQLite was integrated as the application's persistent storage layer.

A normalized database schema was designed.

The Chats table represents individual conversations.

The Messages table stores user prompts and assistant responses linked through foreign keys.

The Files table records metadata about uploaded documents including filenames, storage paths, and relationships to conversations.

To prevent database logic from spreading throughout the codebase, a Repository pattern was adopted.

Dedicated repository classes encapsulate all database operations, while API routes simply invoke repository methods.

This separation improves maintainability, simplifies testing, and allows future migration to another database with minimal code changes.

Now, reopening the application instantly restores previous conversations, uploaded files, and associated context.

Challenge 4 – Coordinating Multiple System Components

NotebookLM Local consists of several independent components:

- Electron desktop application
- FastAPI backend
- SQLite database
- ChromaDB
- Local LLM
- Frontend interface

Initially, these components required manual startup.

Developers had to open multiple terminal windows and execute separate commands before the application became usable.

This workflow was acceptable during development but completely impractical for end users.

Solution

Electron was configured as the orchestration layer.

When the application launches, Electron automatically starts the packaged Python backend as a background process.

Instead of immediately opening the interface, Electron continuously polls the backend's health endpoint.

Only after the backend reports that all services have initialized successfully does Electron load the frontend.

This startup sequence prevents race conditions where the frontend attempts to communicate with a backend that is still loading models or initializing databases.

The result is a smooth user experience where the application appears ready immediately after launch.

Challenge 5 – Packaging the Entire Application

Packaging turned out to be one of the most difficult engineering challenges.

Unlike traditional desktop software, NotebookLM Local depends on numerous Python libraries, machine learning models, and native binaries.

Simply distributing the source code would require users to install Python, manage virtual environments, download dependencies, and manually configure the application.

This was clearly unacceptable.

Solution

PyInstaller was introduced to freeze the Python backend into a standalone executable.

During the build process, PyInstaller automatically detects Python dependencies and packages them together with the application.

Electron Builder then bundles:

- Electron runtime
- Vite frontend
- Packaged Python executable
- Static assets
- Configuration files

The final output is a platform-specific installer such as a Windows `.exe` or macOS `.dmg`.

From the user's perspective, installation is no different from installing any commercial desktop application.

No Python installation, terminal commands, or manual configuration are required.

Final Architecture

The completed architecture follows a clean offline-first workflow.

1. The user launches the desktop application.
2. Electron starts the packaged FastAPI backend.
3. The backend initializes SQLite, ChromaDB, embedding models, and the local language model.
4. Uploaded documents are parsed, cleaned, chunked, embedded, and stored locally.
5. User questions are converted into embeddings.
6. ChromaDB retrieves the most relevant document chunks.
7. Recent conversation history is loaded from SQLite.
8. A carefully constructed prompt is assembled using the system prompt, retrieved chunks, recent conversation, and current question.
9. The local LLM generates an answer.
10. The response is displayed in the frontend and permanently stored for future conversations.

Throughout this entire process, no document, embedding, conversation, or user query ever leaves the local machine. This architecture fulfills the original objective of creating a private, offline, and production-quality AI research assistant while maintaining fast retrieval, accurate responses, and a seamless desktop user experience.