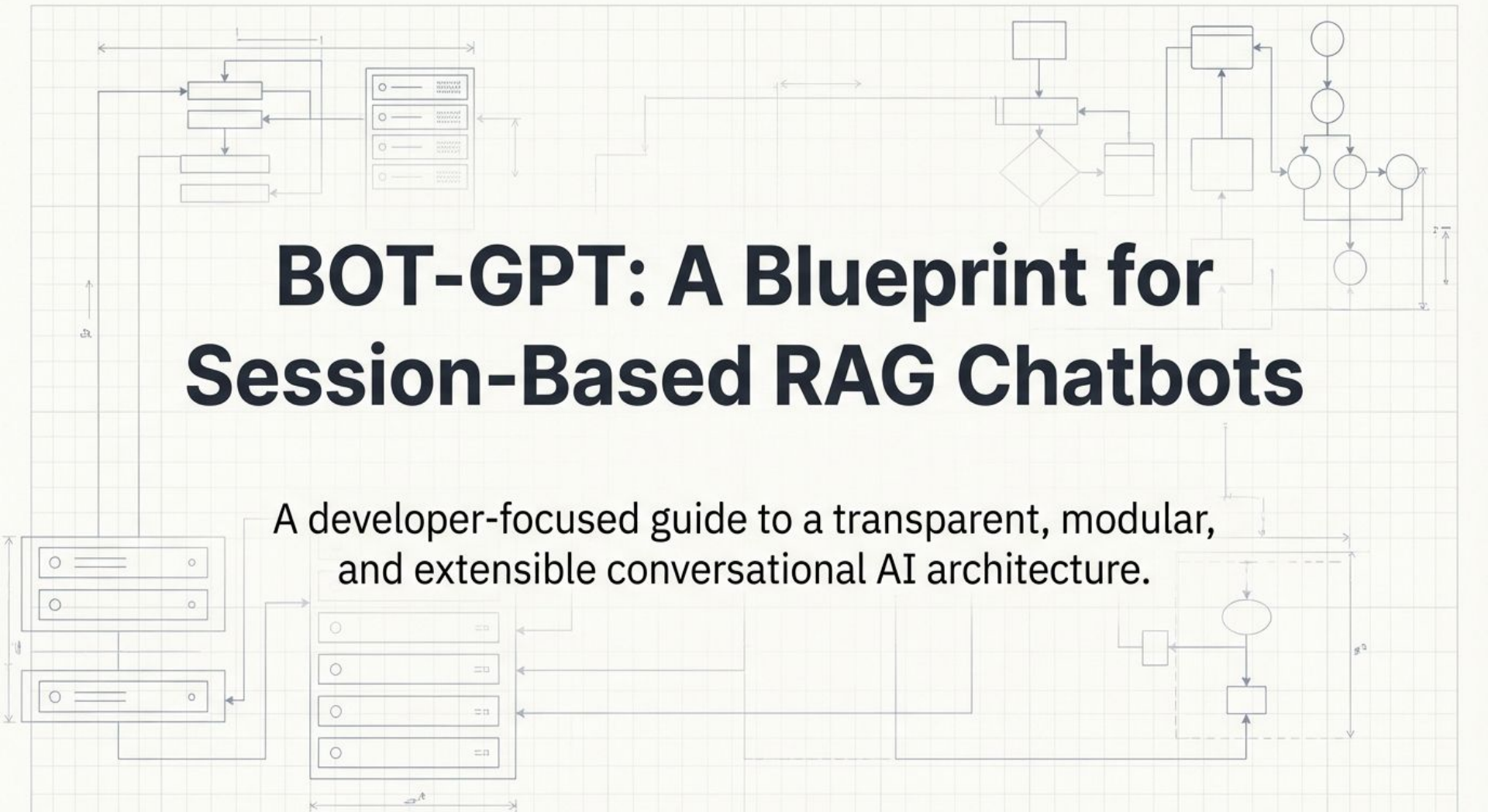


BOT-GPT: A Blueprint for Session-Based RAG Chatbots

A developer-focused guide to a transparent, modular, and extensible conversational AI architecture.

A Simple, Modular, and Transparent AI Bot Architecture

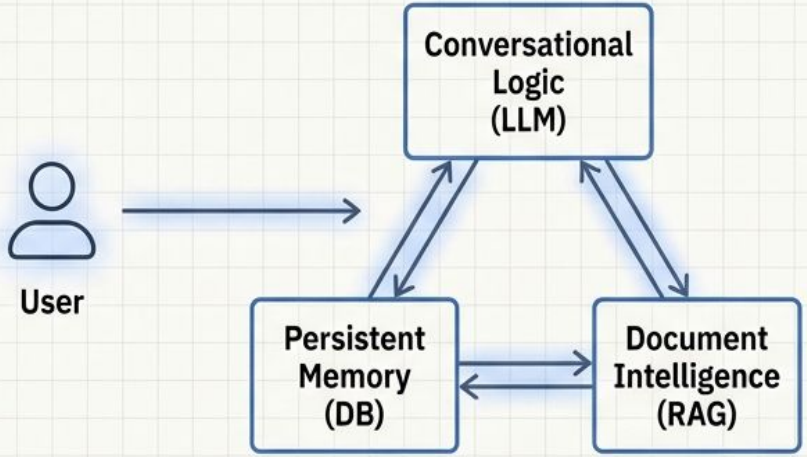
BOT-GPT is a lightweight, developer-friendly prototype of a complete conversational AI system. Its primary goal is to serve as a clean, extensible blueprint for building session-aware RAG chatbots with minimal infrastructure and maximum clarity.

Key Principles

Simple: Easy to monitor with minimum moving parts.

Transparent: Every component and data flow is designed to be easily inspected and understood.

Hackable: Built to be modified—swap out LLMs, databases, or UI components with ease.



Core System Capabilities

Multi-Turn Conversational Chat

- Powered by Google Gemini.
- Maintains full conversation history on a per-session basis.
- Memory persists across reloads.

Persistent Session Management

- Create, view, and delete entire conversation sessions.
- All interactions are stored reliably in a relational database (MySQL).

Document-Grounded RAG

- Upload PDF documents within a specific session.
- Generates OpenAI embeddings and stores them in a local vector database (Chroma).
- Enables session-specific, document-aware answers.

Editable Conversation Timelines

- Unique 'Cut from this message' feature.
- Prunes a conversation from any point, removing the history from both the database and the LLM's context.

The Technology Stack: Proven Tools for a Prototype

Backend & Orchestration

Python 3

Flask
Lightweight API and UI server.

LangChain
Orchestration for memory, prompts, and chains.

AI & Embeddings

Google Gemini (`gemini-2.5-flash-lite`)
Core generative model.

OpenAI (`text-embedding-3-small`)
High-quality document embeddings.

Data Persistence

MySQL
Relational storage for sessions and messages.

Pinecone
Managed vector database for RAG.

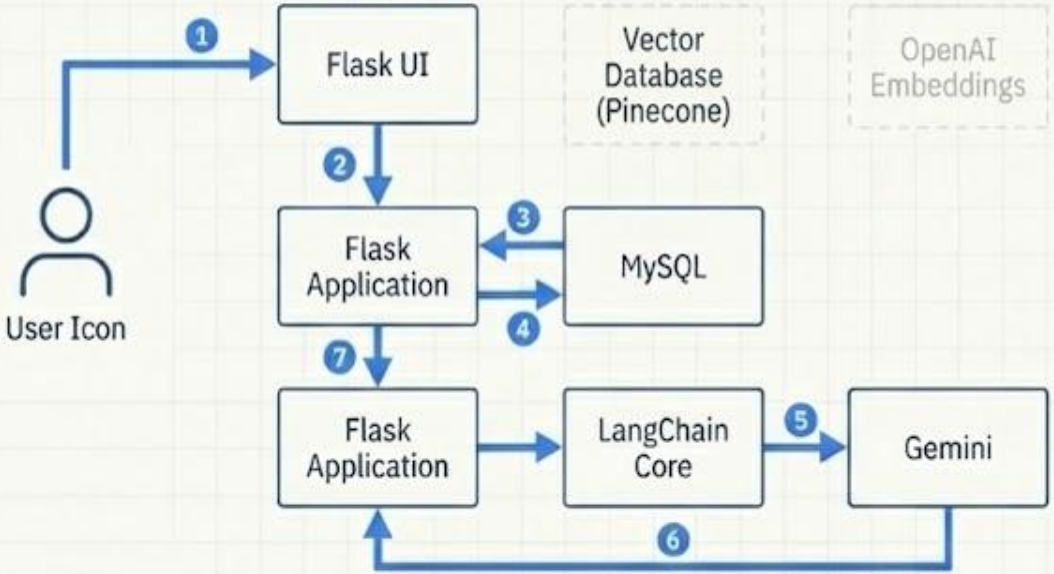
Utilities & Frontend

PyPDF
PDF text extraction.

Jinja2 Templates
Basic HTML rendering.

Google Sans Flex
UI Font.

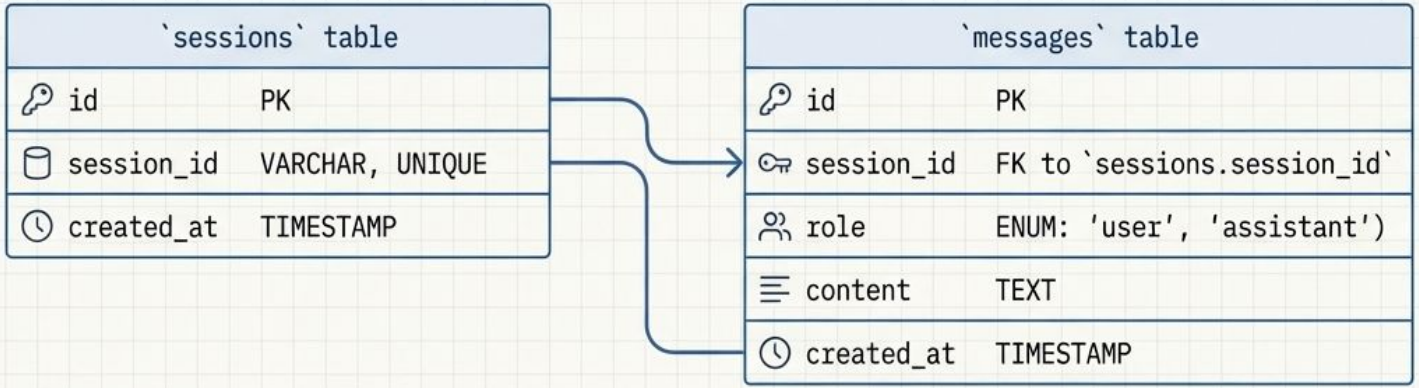
Data Flow 1: The Lifecycle of a Standard Chat Message



1. **User sends message** via the Flask UI for a given `session_id`.
2. **Flask backend** receives the request.
3. **Sync Memory:** The system reads all prior messages for the `session_id` from **MySQL** to construct the current conversation history. The DB is the single source of truth.
4. **Persist User Message:** The new user message is saved to **MySQL** with `role="user"`.
5. **Invoke LLM:** The LangChain runnable is invoked with the full history and the new message, calling the **Gemini API**.
6. **Persist Assistant Message**: Gemini's reply is saved to **MySQL** with `role="assistant"`.
7. **UI Updates**: The session page is reloaded, displaying the complete, updated conversation.

MySQL: The Single Source of Truth for Conversation History

To ensure consistency, LangChain's in-memory history is stateless. Before every LLM call, the history is rebuilt from the database, which is the canonical record.



This simple, normalized schema allows for efficient retrieval of ordered messages for any session, forming the backbone of the bot's memory.

Data Flow 2:
The RAG Ingestion Pipeline

User Uploads PDF

Extract Text
pypdf

Split into Chunks
RecursiveCharacterTextSplitter

Generate Embeddings

Store in Vector DB

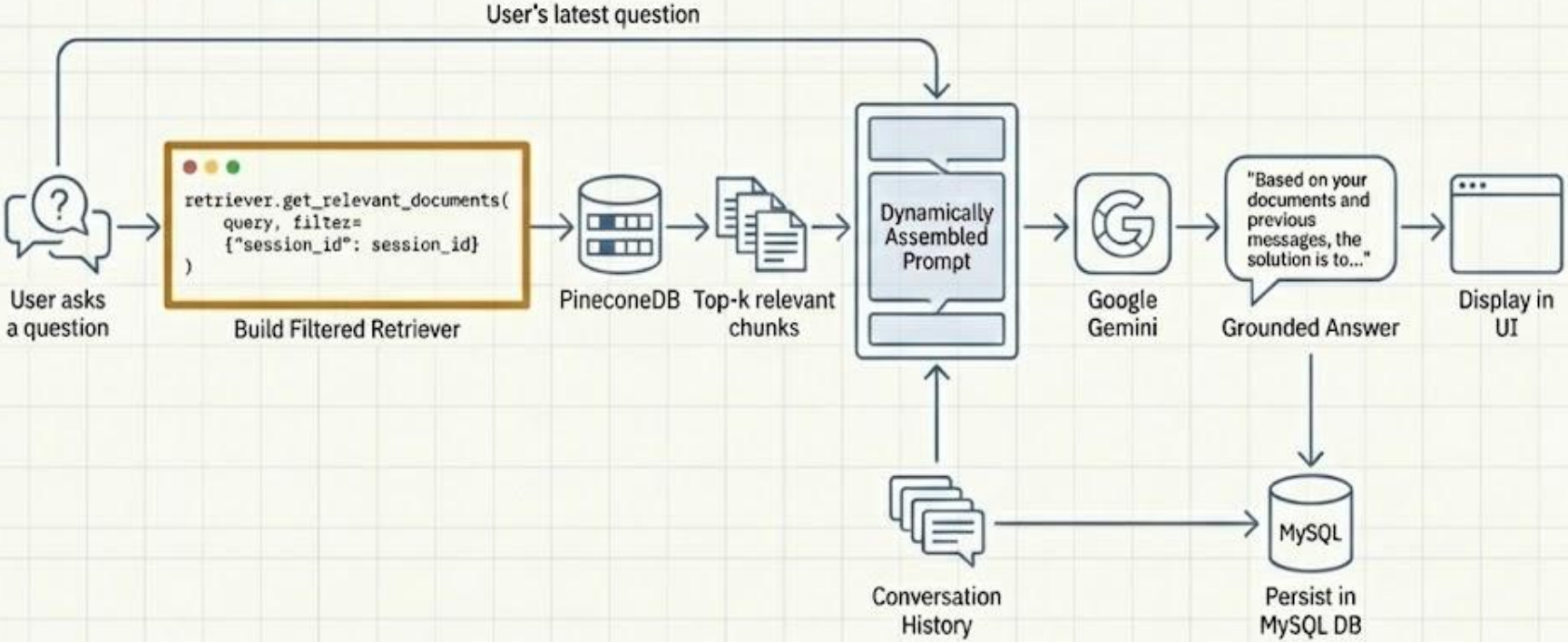
Pinecone Metadata

```
{
  "session_id": "ab12-cd34-ef56",
  "chunk_index": 5
}
```

Storing `session_id` as metadata is the key to enabling retrieval that is sandboxed to a single conversation.

NOTE: A full, more complex RAG system can be added in the future.

RAG in Action: Retrieval and Grounded Generation



The Build Process: An Iterative Development Timeline

1. **LangChain Experiment:** Start with a basic LLM chat in a notebook.
2. **Flask Scaffolding:** Create the core API structure and endpoints.
3. **MySQL Integration:** Add relational database for sessions and messages.
4. **Session Tooling:** Implement core logic for delete and cut operations.
5. **UI Implementation:** Build basic Flask/Jinja2 templates for interaction.
6. **PDF & Vector DB:** Integrate the document ingestion pipeline (PDF -> Chunks -> Embeddings -> Pinecone)
7. **RAG Integration:** Wire the retriever into the LLM prompt for grounded responses.

Takeaway: The system was built incrementally, ensuring each component was functional and tested before adding the next layer of complexity. This iterative process is reflected in the modularity of the final design.