Session-as-RAG: Teaching Your Local AI to Actually Remember

February 13, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** Session-as-RAG treats your conversation history as a searchable document corpus. Each exchange gets embedded and stored in ChromaDB. When you start a new conversation, the system searches your past sessions for relevant context and injects it into the prompt — so your local AI actually remembers what you discussed last week. The core stack is Ollama (embeddings via nomic-embed-text) + ChromaDB (vector storage) + Python (glue code). Topic-aware chunking splits meandering conversations into coherent segments for better retrieval. [S] and [D] markers in the prompt tell the model which context came from past sessions versus reference documents. The full pipeline is about 80 lines of Python.

📚 **More on this topic:** Why Your Chatbot Forgets · Local RAG Guide · Embedding Models for RAG · Context Length Explained · Planning Tool

The previous article in this series explained the six reasons your AI assistant forgets everything between sessions. No persistent storage, no semantic search over history, no cross-session retrieval. Every major chatbot has these problems.

This article fixes them. Session-as-RAG is the approach: treat your conversation history as a document corpus, embed it in a vector database, and retrieve relevant past exchanges whenever you start a new conversation. Your local AI goes from goldfish memory to something that actually knows what you discussed last month.

The implementation is straightforward Python. If you've set up a basic RAG pipeline before, you already know 80% of the pattern.

Conversations Are Documents — Treat Them Like It

Standard RAG indexes PDFs, codebases, and notes. You chunk the documents, embed the chunks, store the vectors, and retrieve relevant chunks at query time. Session-as-RAG applies the exact same pattern to your conversations.

Each conversation exchange — your message plus the model's response — becomes a "document" in the vector store. When you ask a new question, the system embeds your query,

searches for semantically similar past exchanges, and injects the best matches into the current prompt as additional context.

The model reads your past conversation and responds with that context available. It's not remembering. It's reading its own notes.

```
You (today): "What was that ChromaDB error I was getting last week?"

System retrieves: [Session: 2026-02-06]
  User: "ChromaDB keeps throwing 'collection not found' after restart"
  Assistant: "Your client is using EphemeralClient() instead of
  PersistentClient(). Switch to chromadb.PersistentClient(path='./chroma_db')
  and the collection survives restarts."

Model responds with that context — no hallucination, no guessing.
```

The concept is simple. The quality of the implementation depends entirely on two things: how you chunk conversations and which embedding model you use.

Chunking Conversations

Document chunking has natural boundaries — paragraphs, headings, page breaks. Conversations don't. A 90-minute debugging session is one continuous stream of messages with no section headers. How you slice it determines whether retrieval finds the right context or pulls in noise.

Fixed Token Windows — The Naive Approach

The simplest method: split the conversation into chunks of N tokens (say, 512). This is what you'd do with a PDF.

It doesn't work well for conversations. A 512-token window will split mid-thought, separating a question from its answer. You'll retrieve half of an exchange — the part where you described the error but not the part where the model explained the fix. Useless.

Exchange-Based Chunking — The Practical Default

One user message + one assistant response = one chunk. This preserves the full context of each exchange: what you asked, what you got back.

```
def chunk_by_exchange(messages):
    """Group messages into user+assistant pairs."""
    chunks = []
    i = 0
    while i < len(messages) - 1:
        if messages[i]["role"] == "user" and messages[i + 1]["role"] == "assistant":
            chunks.append({
                "text": f"User: {messages[i]['content']}\nAssistant: {messages[i + 1]['content']
                "timestamp": messages[i].get("timestamp"),
                "session_id": messages[i].get("session_id"),
            })
            i += 2
        else:
            i += 1
    return chunks
```

This is the right default for most setups. Each chunk is self-contained: a complete question and a complete answer. Retrieval pulls in coherent context.

The downside: some exchanges are very short (“Thanks!” / “You’re welcome!”) and some are very long (a detailed code review spanning 3,000 tokens). Short chunks add noise to the vector store. Long chunks dilute the embedding. The vector tries to represent too many ideas at once, so it becomes less similar to any specific query.

For long exchanges, split at logical breaks (code blocks, numbered steps) while keeping the user’s original question attached to each sub-chunk. For short exchanges, you can merge 2-3 consecutive short exchanges into a single chunk.

Topic-Aware Chunking — The Best Results

Real conversations wander. You start debugging a Python import error, drift into discussing your project structure, then end up talking about deployment. If the entire session is one chunk (or even chunked by exchange), a query about deployment retrieves the Python debugging context too.

Topic-aware chunking detects when the conversation shifts topics and inserts boundaries. Each topic segment becomes its own chunk, tagged with a topic index.

The detection method: compute the embedding of each exchange, then measure cosine similarity between consecutive exchanges. When similarity drops below a threshold, the topic probably changed.

```
import numpy as np

def detect_topic_boundaries(chunks, embeddings, threshold=0.7):
    """Find indices where the conversation topic shifts."""
    boundaries = [0]
    for i in range(1, len(embeddings)):
        sim = cosine_similarity(embeddings[i - 1], embeddings[i])
        if sim < threshold:
            boundaries.append(i)
    return boundaries

def cosine_similarity(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))
```

A threshold of 0.65-0.75 works for most conversations. Lower catches more subtle shifts but creates more segments. Higher only splits on dramatic topic changes. Start at 0.7 and adjust based on your retrieval quality.

After detecting boundaries, group consecutive exchanges between boundaries into topic segments:

```
def split_by_topic(chunks, boundaries):
    """Group exchanges into topic segments."""
    segments = []
    for i in range(len(boundaries)):
        start = boundaries[i]
        end = boundaries[i + 1] if i + 1 < len(boundaries) else len(chunks)
        segment_text = "\n\n".join(c["text"] for c in chunks[start:end])
        segments.append({
            "text": segment_text,
            "session_id": chunks[start]["session_id"],
            "topic_index": i,
            "timestamp": chunks[start]["timestamp"],
        })
    return segments
```

Each segment gets its own embedding and its own entry in ChromaDB. A query about deployment now retrieves only the deployment discussion, not the Python debugging that happened in the same session.

Embedding and Storing Sessions

You need two things: an embedding model and a vector database. For local setups, the practical stack is Ollama + ChromaDB.

Picking an Embedding Model

You don't need a large embedding model for conversations. Conversation chunks are shorter than typical document chunks (a few hundred tokens vs. multi-page sections), so even smaller models produce good vectors.

Model	Size	Context	Quality (MTEB)	Best For
nomic-embed-text	274 MB	8K tokens	62.3	Default choice — fast, accurate enough, runs on CPU
Qwen3-Embedding 0.6B	1.2 GB	32K tokens	70.7	Higher retrieval quality, still runs on CPU
all-minilm	46 MB	512 tokens	56.3	Minimum hardware, small conversation logs
bge-m3	1.2 GB	8K tokens	68.4	Multilingual conversations

nomic-embed-text is the right default. It's 274 MB, runs on CPU alongside any chat model, handles 8K tokens per chunk (more than any reasonable conversation exchange), and produces vectors that retrieve accurately. Install it via Ollama:

```
ollama pull nomic-embed-text
```

If you want better retrieval quality and can spare 1.2 GB, **Qwen3-Embedding 0.6B** scores significantly higher on benchmarks. For a deeper comparison, see our embedding models guide.

ChromaDB Setup

ChromaDB is the simplest local vector store. It's a Python package, stores everything on disk, and needs zero configuration.

```
pip install chromadb
```

Create a persistent collection for your conversation history:

```
import chromadb
from chromadb.utils.embedding_functions import OllamaEmbeddingFunction

# Connect to ChromaDB with persistent storage
client = chromadb.PersistentClient(path="./session_memory")

# Use Ollama's nomic-embed-text for embeddings
embedding_fn = OllamaEmbeddingFunction(
    model_name="nomic-embed-text",
    url="http://localhost:11434/api/embeddings",
)

# Create (or get) the sessions collection
collection = client.get_or_create_collection(
    name="conversation_history",
    embedding_function=embedding_fn,
)
```

Storing a Conversation Chunk

Each chunk goes into ChromaDB with metadata — session ID, timestamp, and topic index if you're doing topic-aware chunking:

```
import uuid
from datetime import datetime

def store_exchange(collection, text, session_id, topic_index=0):
    """Store a conversation chunk in ChromaDB."""
    collection.add(
        ids=[str(uuid.uuid4())],
        documents=[text],
        metadatas=[{
            "session_id": session_id,
            "topic_index": topic_index,
            "timestamp": datetime.now().isoformat(),
            "source_type": "session",
```

```
        }],
    )
```

ChromaDB handles the embedding automatically using the `OllamaEmbeddingFunction` you configured on the collection. You pass in raw text, it embeds and stores.

Conversation history generates far fewer vectors than a document corpus. A year of daily conversations might produce 5,000-10,000 chunks. ChromaDB handles this without breaking a sweat — you won't hit performance issues until you're well into the hundreds of thousands of entries.

Retrieval at Query Time

When the user sends a new message, search the vector store for relevant past context before passing anything to the chat model.

```
def retrieve_context(collection, query, top_k=5):
    """Find the most relevant past exchanges for a query."""
    results = collection.query(
        query_texts=[query],
        n_results=top_k,
    )

    retrieved = []
    for doc, meta in zip(results["documents"][0], results["metadatas"][0]):
        retrieved.append({
            "text": doc,
            "timestamp": meta["timestamp"],
            "session_id": meta["session_id"],
            "source_type": meta["source_type"],
        })
    return retrieved
```

Top-k of 3-5 works for most cases. More than that and you start crowding out the current conversation in the context window. Each retrieved chunk might be 200-500 tokens, so 5 chunks costs you 1,000-2,500 tokens of context budget.

[S] vs [D] Markers — Knowing Where Context Came From

If your system retrieves context from both past sessions and reference documents (a combined RAG setup), the model needs to distinguish between them. Use markers:

- **[S: 2026-02-06]** — this came from a past session on that date
- **[D: project-readme.md]** — this came from an uploaded document

This matters for you too, not just the model. When the model references something from a past conversation, you want to know which session it's pulling from so you can verify it's not citing stale information.

Formatting the Prompt

Inject retrieved context into a dedicated section of the system prompt:

```
def format_prompt(current_messages, retrieved_context):
    """Build a prompt with retrieved session context."""
    if not retrieved_context:
        return current_messages

    memory_section = "## Relevant Past Sessions\n\n"
    for ctx in retrieved_context:
        date = ctx["timestamp"][:10]  # YYYY-MM-DD
        marker = f"[S: {date}]"
        memory_section += f"{marker}\n{ctx['text']}\n\n"

    memory_section += (
        "---\n"
        "Use the above past sessions for context when relevant. "
        "Cite sessions as [S: date] when referencing them. "
        "If past context conflicts with the current conversation, "
        "prefer the current conversation.\n"
    )

    # Prepend to system message or insert as a separate system block
    system_msg = {
        "role": "system",
        "content": memory_section,
    }
    return [system_msg] + current_messages
```

The instruction to prefer current conversation over past context is important. Old sessions may contain outdated information — you might have switched from PostgreSQL to SQLite since that conversation in January.

Topic Splitting — Why One Session Isn't One Topic

Here's the full topic splitting pipeline, combining the pieces from earlier into a working flow:

```
import ollama

def embed_text(text):
    """Get embedding from Ollama."""
    response = ollama.embed(model="nomic-embed-text", input=text)
    return response["embeddings"][0]

def process_session(messages, session_id, collection, threshold=0.7):
    """Chunk, split by topic, and store a full session."""
    # Step 1: Chunk by exchange
    chunks = chunk_by_exchange(messages)
    if not chunks:
        return

    # Step 2: Embed each chunk
    embeddings = [embed_text(c["text"]) for c in chunks]

    # Step 3: Detect topic boundaries
    boundaries = detect_topic_boundaries(chunks, embeddings, threshold)

    # Step 4: Group into topic segments
    segments = split_by_topic(chunks, boundaries)

    # Step 5: Store each segment
    for segment in segments:
        store_exchange(
            collection,
            text=segment["text"],
            session_id=session_id,
            topic_index=segment["topic_index"],
        )
```

Call `process_session()` at the end of each conversation (or periodically during long ones). The function chunks the conversation into exchanges, embeds each exchange, detects where topics shift, groups exchanges into topic segments, and stores each segment in ChromaDB.

A typical 30-minute conversation with 15-20 exchanges might split into 3-5 topic segments. Each segment gets its own vector, so retrieval is precise: a query about your database schema pulls in only the database discussion, not the unrelated CSS debugging from the same session.

Tuning the Threshold

The cosine similarity threshold controls how aggressively you split topics:

Threshold	Behavior	Good For
0.60	Aggressive splitting, many small segments	Conversations that jump between topics frequently
0.70	Balanced — catches clear topic changes	General-purpose default
0.80	Conservative — only splits on dramatic shifts	Focused conversations that stay on-topic

Start at 0.70. If retrieval keeps pulling in irrelevant context from the same session, lower it. If your segments are too fragmented (single exchanges instead of coherent discussions), raise it.

The Core Loop — Full Pipeline

Here's the complete flow, combining storage and retrieval into a conversation loop:

```
import chromadb
from chromadb.utils.embedding_functions import OllamaEmbeddingFunction
import ollama
import uuid
from datetime import datetime

# --- Setup ---
client = chromadb.PersistentClient(path="./session_memory")
embedding_fn = OllamaEmbeddingFunction(
    model_name="nomic-embed-text",
    url="http://localhost:11434/api/embeddings",
)
collection = client.get_or_create_collection(
```

```
    name="conversation_history",
    embedding_function=embedding_fn,
)

def chat_with_memory(user_message, conversation, session_id):
    """Send a message with relevant past context retrieved."""

    # 1. Retrieve relevant past sessions
    retrieved = retrieve_context(collection, user_message, top_k=5)

    # 2. Build prompt with memory
    conversation.append({"role": "user", "content": user_message})
    augmented = format_prompt(conversation, retrieved)

    # 3. Get response from the chat model
    response = ollama.chat(model="llama3.2", messages=augmented)
    assistant_msg = response["message"]["content"]

    # 4. Add response to conversation
    conversation.append({"role": "assistant", "content": assistant_msg})

    # 5. Store this exchange for future retrieval
    exchange_text = f"User: {user_message}\nAssistant: {assistant_msg}"
    store_exchange(collection, exchange_text, session_id)

    return assistant_msg
```

That's the core loop. Every message you send gets augmented with relevant past context. Every response gets stored for future retrieval. The vector store grows over time, and retrieval quality improves as the system accumulates more of your conversation history.

The chat model can be anything you run via Ollama — Llama 3, Qwen 2.5, Mistral, whatever fits your hardware.

---

mycoSwarm — A Working Reference

The sky-memory-system (used by mycoSwarm) implements a version of this pattern with a multi-layer architecture:

Layer	What It Stores	How It's Accessed
NOW.md	Current session state, active tasks	Loaded into every prompt

Layer	What It Stores	How It's Accessed
MEMORY.md	Long-term facts, preferences, project context	Loaded into every prompt
ChromaDB	Semantic embeddings of past knowledge	Searched at query time
SQLite	Structured relationships between concepts	Queried for linked context

The file-based layers (NOW.md, MEMORY.md) handle the "always-available" context — your name, your current project, your preferences. These get loaded into every prompt unconditionally.

The vector layer (ChromaDB) handles the "sometimes-relevant" context — past debugging sessions, old architectural decisions, specific conversations. These get retrieved only when the current query is semantically related.

The relational layer (SQLite) handles structured connections — "project X uses technology Y" or "bug A was caused by decision B." This is harder to implement but enables graph-style traversal that pure vector search can't do.

You don't need all four layers. Start with exchange-based chunking into ChromaDB (the core of this article). Add file-based memory if you want persistent preferences. Add SQLite if you need structured relationships.

Gotchas and Practical Limits

Embedding Model Lock-In

If you switch embedding models, your existing vectors become incompatible. The new model produces vectors in a different space — cosine similarity between old and new embeddings is meaningless. You'd need to re-embed every stored conversation.

Pick a model and stick with it. nomic-embed-text is a safe long-term choice: it's open-source, well-maintained, and performant enough that you're unlikely to need an upgrade. See our embedding models guide for the full comparison.

Stale Context

Your conversation from three months ago might reference outdated information. You've switched frameworks, changed databases, refactored your codebase. The model doesn't know the retrieved context is stale — it treats it as current fact.

Options:

- **Decay weighting:** Prefer recent sessions over old ones by adding a time-based penalty to similarity scores
- **Manual pruning:** Periodically review and delete obsolete entries from ChromaDB
- **Prompt instructions:** Tell the model to prefer current conversation over retrieved context (already in the prompt template above)

Context Budget

Retrieved chunks compete with the current conversation for context window space. Five retrieved chunks at 400 tokens each costs 2,000 tokens. On a model with 8K context, that's 25% of your budget gone before the conversation starts.

Keep top-k low (3-5), keep chunks concise, and monitor whether retrieved context is actually helping. If retrieval quality is low (the retrieved chunks aren't relevant), reduce top-k or tighten your topic splitting.

Chunk Size Tuning

Chunk Size	Retrieval Quality	Noise Level	Storage Cost
Single exchange	High precision	Low	More vectors, small each
3-5 exchanges	Moderate	Medium	Fewer vectors, broader each
Full topic segment	Broad recall	Higher	Fewest vectors, large each

Start with single exchanges. If retrieval returns too many similar-but-slightly-different chunks from the same conversation, switch to topic segments.

Privacy

All your conversations are stored on disk in ChromaDB. That's the point — local storage means local control. But be aware of what accumulates. If you discuss sensitive information (credentials, personal data, internal business details), it lives in `./session_memory/` until you delete it.

Encrypt the ChromaDB directory if you're storing sensitive conversations. Or maintain separate collections for sensitive and non-sensitive topics.

Where to Go From Here

Session-as-RAG turns a stateless text generator into something that builds knowledge over time. The implementation is ~80 lines of Python on top of tools you may already be running.

The stack: Ollama for embeddings and chat, ChromaDB for vector storage, Python for the glue. If you haven't set up basic RAG yet, start with our local RAG guide to get the fundamentals down, then come back here to add conversation memory on top.

If you want this without writing code, AnythingLLM and Letta (formerly MemGPT) offer persistent memory workspaces out of the box. They implement variations of this pattern under the hood. See our best LLMs for RAG guide for model recommendations that pair well with these tools.

The cloud platforms will eventually build this. But right now, this is something you can only get by running local.

Get notified when we publish new guides.

Subscribe — free, no spam

Source: https://insiderllm.com/guides/session-as-rag-local-ai-memory/

Free guides for running AI locally