

A Local-First macOS AI Workspace for Private, Context-Aware Assistance

Akshaya Lohia

Computer Systems Lab | Thomas Jefferson High School for Science and Technology | Dr.
Gabor

May, 2026

Abstract

This project presents a native macOS AI workspace designed to deliver private, local-first, and deeply context-aware assistance for everyday knowledge work. The macOS application integrates local and cloud large language models (LLMs) with a hybrid retrieval system over a user's own data, including files, mail, calendar, and iMessages, while preserving strong privacy guarantees by default. The system combines a streaming chat interface, an agentic tool layer, and a unified "Brain" search contract backed by a Qdrant vector database, enabling grounded, source-cited answers and multi-step tool use within a single workspace. Over the course of the year, the project evolved from an initial idea for a generic personal context system into a production-grade macOS app with iMessage ingestion, Google data integration, local model support via Ollama, and a lazy-loaded skill architecture for token-efficient agents.

Introduction

Motivation

Modern LLM-based assistants are powerful but often cloud-only, opaque in how they use personal data, and weakly grounded in a user's actual digital life. Users who care about privacy and control must choose between convenience and security, and even advanced users lack tools that can seamlessly reason over their own files, mail, and messages while remaining local-first. This project aims to close that gap by building an AI workspace that is:

- **Local-first:** core workflows run offline on Apple Silicon, with cloud models strictly opt-in.
- **Private by design:** no telemetry, no silent data exfiltration; sensitive data stays on the machine.
- **Context-aware:** retrieval and memory are explicitly grounded in the user's documents, mail, calendar, and messages through a unified search layer.

Problem Statement

The central problem addressed is how to design and implement a macOS-native AI system that:

- Offers a unified chat and research experience instead of being a thin API wrapper.
 - Provides grounded, source-cited responses over heterogeneous personal data.
 - Balances local and remote inference, enabling offline use without sacrificing access to state-of-the-art models.
 - Preserves privacy and user control while still being practical and performant on consumer hardware.
-

Related Work

Existing LLM Memory and Context Systems

Commercial systems such as ChatGPT, Perplexity, and other consumer chatbots support basic memory and history, but typically store user data in the cloud and provide limited transparency into retrieval mechanisms or long-term storage.

Reverse-engineering work on ChatGPT memory suggests a layered context structure (system instructions, user memory, conversation summary, current messages) without explicit vector-based RAG over user archives, trading some grounding for simplicity and efficiency.

Several products, such as “all-in-one” AI assistants, attempt to build persistent context by syncing with user emails, calendars, and notes, but are mostly cloud-based and proprietary, offering limited control or inspectability.

Retrieval-Augmented Generation (RAG)

RAG has emerged as a standard pattern to ground LLMs in external data by embedding documents and queries into a shared vector space and retrieving semantically similar chunks.

Industry guides and cloud provider documentation emphasize the combination of dense vector search with metadata filtering and chunking strategies to balance recall, precision, and cost.

This project adopts and extends these ideas by pairing a Qdrant vector database with BM25-style sparse retrieval and explicit collection selection, tailored to personal knowledge bases.

Local Models and Agent Frameworks

Open-source frameworks such as llama.cpp, LangChain, and emerging agent libraries show how to orchestrate tools and retrieval with LLMs, but many solutions assume server-class hardware or cloud environments and do not fully address macOS permissions (TCC) or filesystem integration. Existing macOS-native chat applications provide strong UIs but rarely expose a full, open-source, local-first, agentic architecture with hybrid retrieval and life-data indexing.

System Design

High-Level Architecture

This project is implemented as a native macOS application using Swift and SwiftUI, targeting Apple Silicon and macOS 14. The UI follows an MVVM architecture with observable view models, and all UI updates are confined to the main actor for stability. Under the hood, the system is composed of four major layers:

1. **Chat and Workspace UI:** streaming chat, branching conversations, multimodal input, LaTeX/Markdown rendering, and dedicated surfaces for research and creation.
2. **Agent Layer:** an LLM-driven tool calling loop that can search the web, query the local Brain, load skills, and call OS-level capabilities.
3. **Retrieval Layer (“Brain” and Experts):** a unified hybrid search system backed by Qdrant, with explicit collections and “Expert” personas scoped to project-specific corpora.
4. **Model Layer:** a unified transport that routes requests transparently to local llama.cpp models or remote OpenAI-compatible cloud APIs.

Local-First and Privacy Guarantees

The system is architected around the principle that all sensitive data should remain on the user’s machine by default. Core workflows, including chat over local models and retrieval over personal data, run fully offline using llama.cpp-based inference on Apple Silicon and a local Qdrant instance for vector search.

No telemetry or tracking is implemented; the only time data leaves the machine is when the user explicitly opts into using a cloud LLM provider.

Unified Brain and Experts

The application’s retrieval model is organized into two complementary modes:

- **Experts:** scoped, persona-like contexts that operate inside a curated corpus, such as a specific project folder, set of PDFs, or research archive.

The agent uses these to perform project-focused reasoning and GraphRAG-style relationship-aware retrieval.

- **Brain:** a global, unified search index over “life data,” including files, iMessages, email, documents, notes, and calendar events.

The Brain is backed by Qdrant with hybrid dense (embeddings) and sparse (BM25-style) retrieval, plus heuristic reranking and context expansion around winning chunks.

The Brain search tool requires explicit query and collections parameters; there is no opaque “search everything” behavior. This design makes the retrieval scope visible and controllable, and retrieval decisions are exposed in an Agent Trace view that shows which collections were queried and which results were injected.

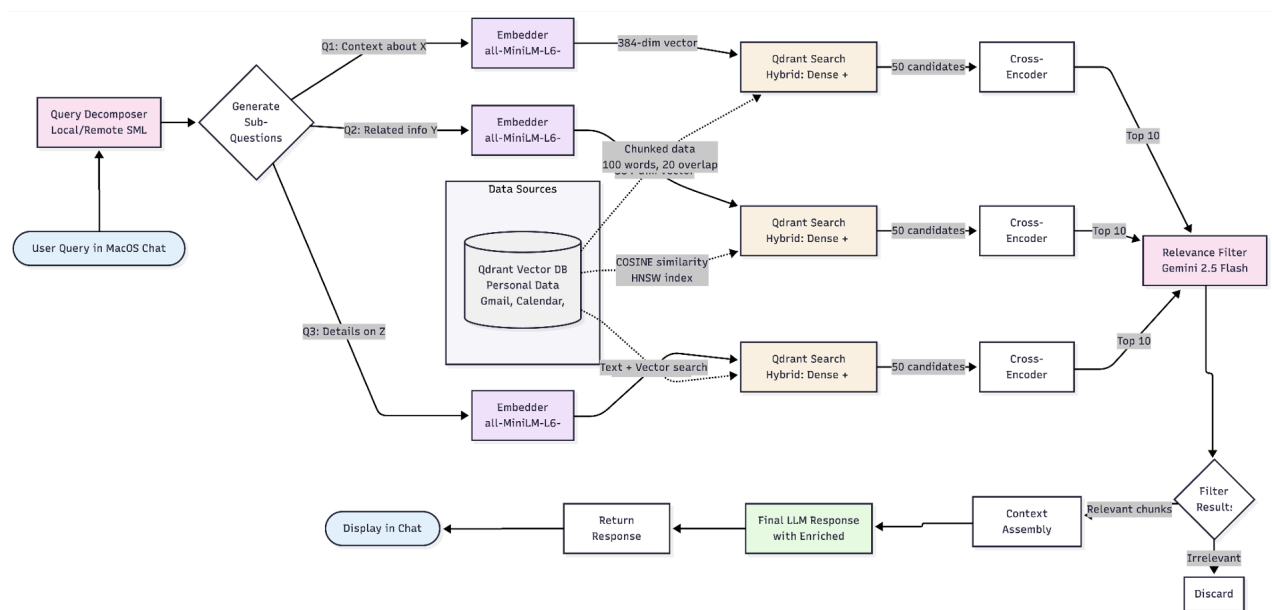


Figure 1: Brain Tool Workflow Diagram

Methods

Data Ingestion and Retrieval

Google Mail and Calendar

The project includes a Python-based ingestion pipeline for Gmail and Google Calendar via OAuth2, using carefully scoped permissions to download a user's messages and events over a chosen time window.

To avoid rate limits and API bans, the pipeline batches requests and introduces deliberate delays while fetching messages in bulk. Initial versions stored raw text in files; later iterations migrated to a Qdrant-backed vector store with a local embedding model (all-MiniLM-L6-V2) running on the same machine.

iMessage Integration

On macOS, iMessage history is stored in a local SQLite database at `Library/Messages/chat.db`. The project designs and partially implements a read-only ingestion pipeline that:

- Reads message threads and metadata through SQL queries.
- Normalizes and groups messages into conversational windows of roughly 10–20 messages.
- Attaches sender and timestamp metadata for each chunk.
- Pipes those chunks into the existing KnowledgeBrain ingestion path as message documents.

To keep development safe, the early implementation targets a mock database rather than the live store, and the app requires explicit Full Disk Access configuration from the user before accessing real messages.

A dedicated settings UI lets users control which conversations and time ranges are indexed, and supports excluding entire threads or contacts from the Brain.

Hybrid Retrieval

Retrieval over a user's life data uses a hybrid approach:

- Dense semantic embeddings (e.g., all-MiniLM-based encoders) stored in Qdrant.
- Sparse BM25-style keyword signals for lexical matching.
- Heuristic reranking and expanding to additional user context to include surrounding messages or document sections.

The agent must explicitly specify which collections (e.g., messages, email, documents, calendar) to search, and optional filters include date range, participants, sender, and context flags.

This contract is designed to keep retrieval explainable and avoid accidentally searching “everything” without user awareness.

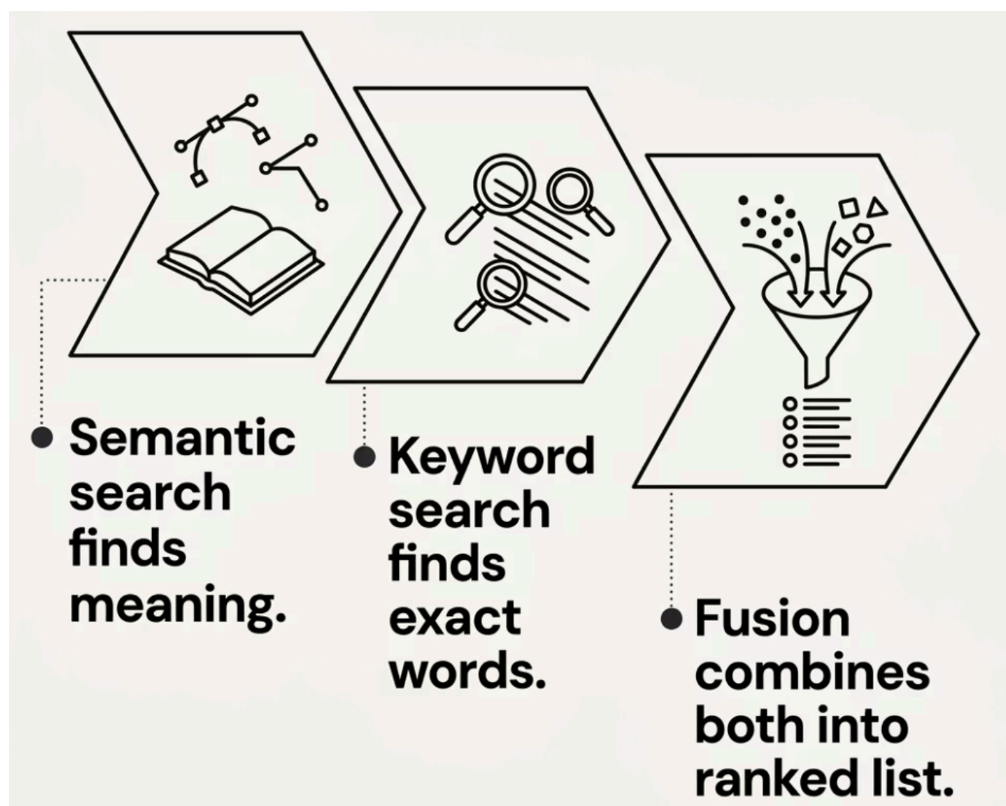


Figure 2: Hybrid Search

```
Swift
public func search(
    query: String,
    vectorIndex: SimilarityIndex,
    bm25Index: BM25Index,
    topK: Int = 20
) async -> [HybridResult] {
```

```

// Run both searches (vector search is async)
let vectorResults = await vectorIndex.search(
    query: query,
    maxResults: topK * 2,
    threshold: vectorThreshold
)

let bm25Results = bm25Index.search(query: query, topK: topK * 2)

Self.logger.debug("Vector search returned \(vectorResults.count), BM25 returned
\((bm25Results.count) results")

return fuseResults(
    vectorResults: vectorResults,
    bm25Results: bm25Results,
    topK: topK
)
}

/// Fuse vector and BM25 results using Reciprocal Rank Fusion
private func fuseResults(
    vectorResults: [SearchResult],
    bm25Results: [BM25Index.SearchResult],
    topK: Int
) -> [HybridResult] {
    // Build maps from document identifier to rank and score
    // For vector results, we use the source URL + chunk index as identifier
    var vectorRanks: [String: (rank: Int, score: Float, result: SearchResult)] = [:]
    for (index, result) in vectorResults.enumerated() {
        let key = makeKey(source: result.sourceUrlText ?? "", chunkIndex:
result.itemIndex)
        vectorRanks[key] = (index + 1, result.score, result) // 1-indexed rank
    }

    var bm25Ranks: [String: (rank: Int, score: Double, result:
BM25Index.SearchResult)] = [:]
    for (index, result) in bm25Results.enumerated() {
        let key = makeKey(source: result.document.source, chunkIndex:
result.document.chunkIndex)
        bm25Ranks[key] = (index + 1, result.score, result)
    }

    // Collect all unique document keys
    var allKeys = Set(vectorRanks.keys)
    allKeys.formUnion(bm25Ranks.keys)

    // Calculate RRF scores and build results
    var hybridResults: [HybridResult] = []

    for key in allKeys {
        let vectorData = vectorRanks[key]

```

```

let bm25Data = bm25Ranks[key]

// RRF score = sum of 1/(k + rank) for each ranking
var rrfScore: Double = 0

if let vr = vectorData {
    rrfScore += 1.0 / (rrfK + Double(vr.rank))
}
if let br = bm25Data {
    rrfScore += 1.0 / (rrfK + Double(br.rank))
}

```

Figure 3: Hybrid Retrieval Implementation within Swift

Model Orchestration

Local Models via Ollama and llama.cpp

The project integrates local LLMs using Ollama as a local HTTP endpoint that wraps llama.cpp-based inference. Over the course of the project, multiple models were tested, including Qwen 3B/4B, Phi-4 Mini, and Gemma 3 1B, with a focus on maximizing tokens per second on a MacBook Air while staying within modest RAM limits.

Empirical testing showed that smaller models (around 1–1.7B parameters) can handle short, tool-free chats reasonably well but struggle with long system prompts and complex tool-calling schemas; larger 3B–4B models improve capability at the cost of higher latency and resource usage. The final system supports running any Ollama-served model, but practical offline tool-calling remains constrained by current hardware and model capabilities.

Cloud Models and Unified Transport

The macOS client can route requests to remote LLM providers over OpenAI-compatible or Gemini-style APIs if model reasoning is the user’s priority over privacy. A single code layer works for both local and remote paths, so the UI and agent orchestration code do not need to change when the user switches models. This allows users to mix and match local inference for private workflows with cloud models for heavy tasks, while keeping routing explicit.

Agent and Skill System

Agent Loop

The agent uses an iterative loop where the LLM:

1. Reads the conversation, system instructions, and trace of previous actions.
2. Decides whether to issue tool calls (e.g., web search, Brain search, file operations).
3. Receives results from an executor that runs tools.
4. Compresses large tool outputs using a worker model before continuing, to preserve context window space.
5. Repeats until the task is complete or a bounded iteration limit is reached.

All tool calls and results are visible in an Agent Trace dashboard that records timing, selected tools, and compression steps, helping users inspect and trust the system's behavior.

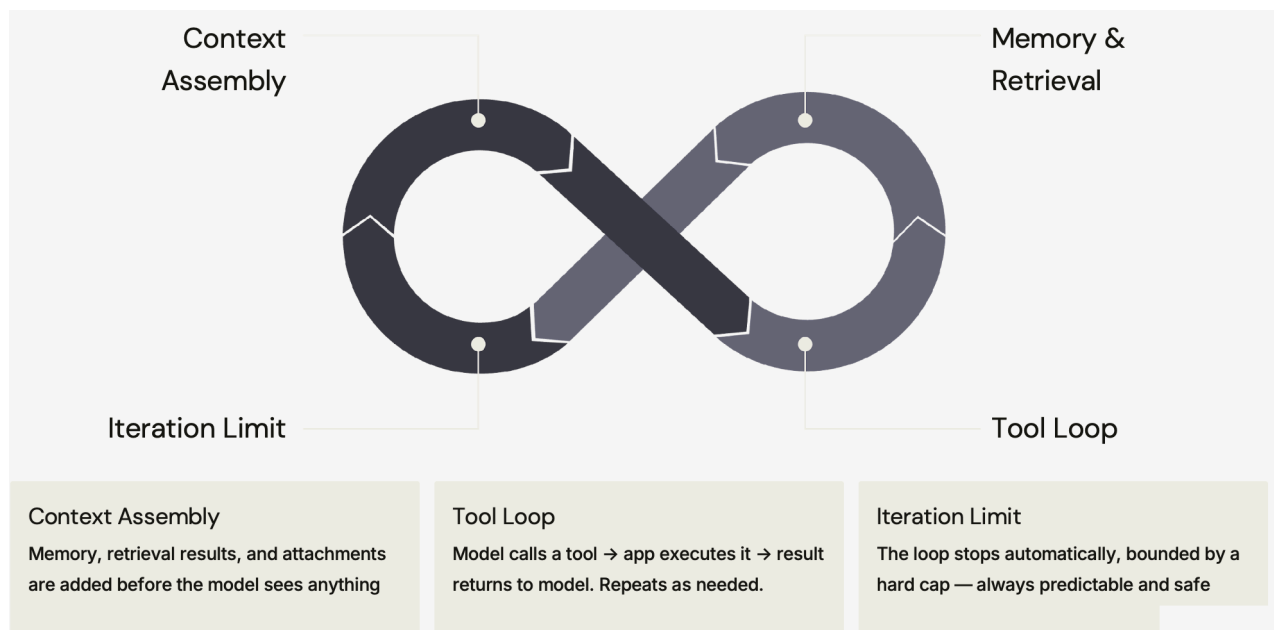


Figure 4: Agentic tool loop

Swift

```
// Tell the LLM to use functions when enabled and server does not support native  
// tool calling  
// Use enabled functions from FunctionSelectionManager if no custom functions  
// provided  
let enabledFunctions: [any AnyFunctionBox]
```

```

if let customFunctions = functions {
    enabledFunctions = customFunctions
} else {
    // Get core always-on functions + any dynamically loaded skills
    let coreFunctions = await MainActor.run {
        FunctionSelectionManager.shared.getCoreEnabledFunctions() }
    let activeSkillFunctions = await MainActor.run {
        SkillManager.shared.getActiveSkillFunctions() }
    enabledFunctions = coreFunctions + activeSkillFunctions
}
// Check if we should encourage using query_database function
if let expert = expert,
    !expert.isDefault,
    expert.resources.resources.count > 0,
    Settings.useFunctions && useFunctions {
    // Check if query_database function is enabled
    let hasQueryDatabaseFunction = enabledFunctions.contains { function in
        return function.name == "query_database"
    }
    if hasQueryDatabaseFunction {
        let expertDatabasePrompt: String = ""
        The `\"(expert.name)` is currently active. Use `query_database` to query the
        `\"(expert.name)` database whenever it might help inform your response.
        ""
        fullSystemPromptComponents.append(expertDatabasePrompt)
    }
}
if Settings.useFunctions && useFunctions && !enabledFunctions.isEmpty {
    fullSystemPromptComponents.append(InferenceSettings.useFunctionsPrompt)
    // Inject function schema if no native tool calling
    if !InferenceSettings.hasNativeToolCalling {
        fullSystemPromptComponents.append(InferenceSettings.functionsSchemaPrompt)
        let functions: [any AnyFunctionBox] = enabledFunctions
        for function in functions {
            fullSystemPromptComponents.append(function.getJsonSchema())
        }
    }
}
}

```

Figure 5: Context and Prompt Assembly Code Snippet

Lazy-Loaded Skills

A key innovation in the project is the **lazy-loaded skill system**, designed to avoid burning tokens by loading every tool schema into the prompt on every turn.

Instead, the system exposes:

- A compact **skill index** listing names and short descriptions of available skills.

- A small set of **core tools** (web search, Brain search, load_skills) that are always available.

When the LLM decides it needs a specific skill, it calls `load_skills`, which returns the full JSON schema for that skill only. Loaded skills persist for several turns before expiring if unused, which keeps prompts small and token usage efficient while still enabling rich tool coverage.

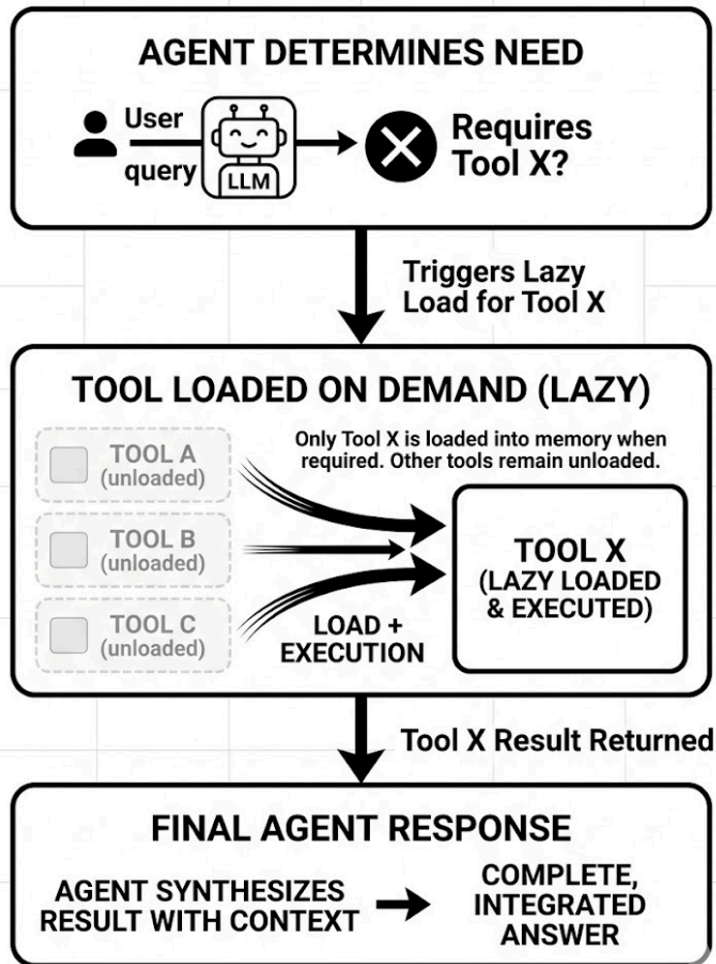


Figure 6: Lazy Loading Explained

Swift

```

/// Load specific skills by path (e.g., "calendar/add_event" or "calendar/*")
/// - Parameter paths: Array of skill paths to load
/// - Returns: Array of loaded AnyFunctionBox implementations
func loadSkills(_ paths: [String]) throws -> [AnyFunctionBox] {
    var loaded: [AnyFunctionBox] = []
    var loadedNames: Set<String> = []

    for path in paths {

```

```

        if path.hasSuffix("/*") {
/// Called at start of each user turn: increment counter, evict expired skills
func onNewUserTurn() {
    currentTurn += 1

    // Evict skills not used in the last N turns
    let expiredSkills = activeSkills.filter { _, lastUsed in
        currentTurn - lastUsed > skillExpiryTurns
    }.keys

    for skillName in expiredSkills {
        activeSkills.removeValue(forKey: skillName)
    }
}
}

```

Figure 7: Lazy Loading Code Snippet

MacOS Integration and Permissions

Because the project runs as a macOS app, it must respect Apple’s Transparency, Consent, and Control (TCC) system. Permissions are required and passed to the user when accessing calendar, contacts, reminders, and the filesystem; tools that depend on sensitive data are gated behind these OS-level checks.

The agent’s file navigation capabilities were refactored to support relative paths, shortcuts to common locations (e.g., Desktop, Documents), and clearer error messages when files are missing or moved, making everyday workflows smoother.

Results

Functional Capabilities

By the end of the development period, the project implements the following core user-facing capabilities:

- A streaming chat UI with branching message trees, multimodal input, and rich rendering for Markdown, LaTeX, and citations.

- A “Deep Research” surface that guides the agent through multi-step workflows such as clarify → plan → web research → draft → refine.
- Creation-focused surfaces, including Canvas-style freeform workspaces, diagramming with Mermaid, and a slide studio for presentations.
- A unified Brain search that can retrieve and cite information from files, email, calendar, and (in-progress) iMessages based on the user’s own archives.
- Optional integration with cloud LLMs through OpenAI-compatible APIs and Gemini routes, alongside local models via Ollama.
- A settings UI and ingestion pipeline that allow users to control which iMessage threads and time ranges are indexed, and to exclude conversations entirely.

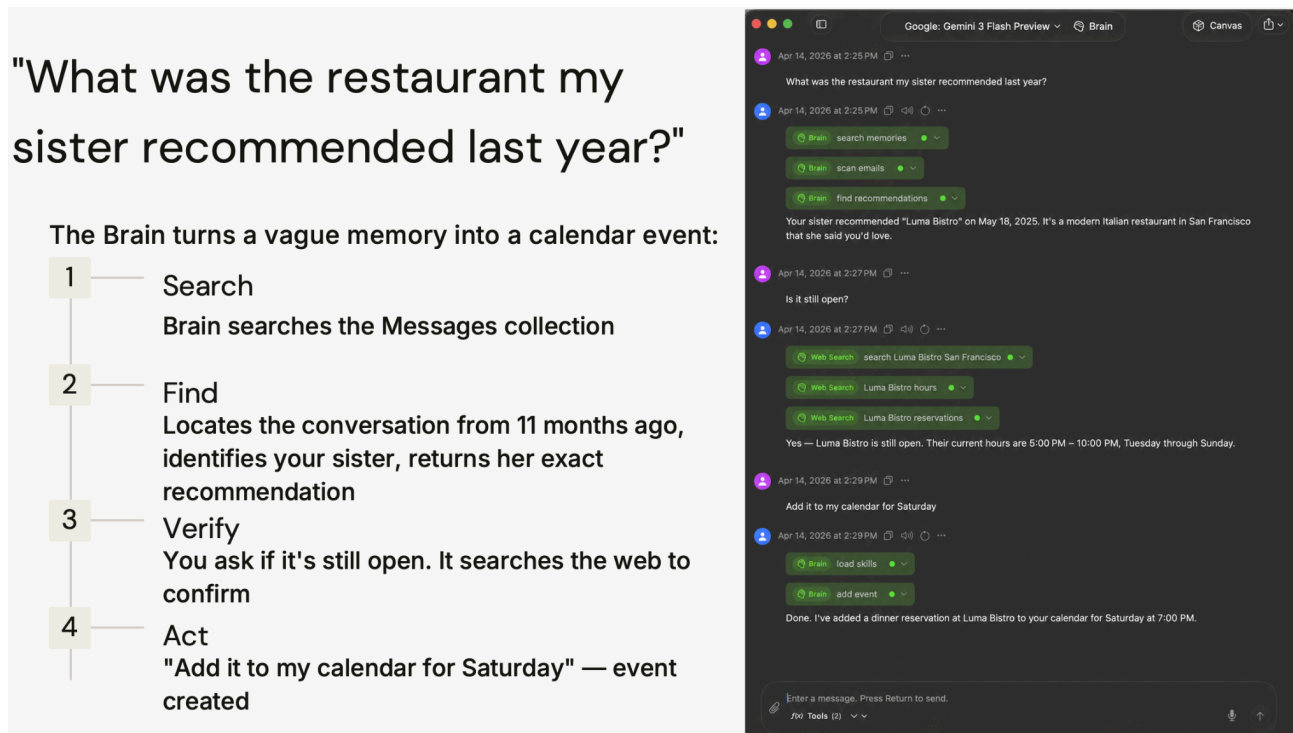


Figure 8: Example Agentic Tool Loop

Engineering Iterations and Bug Fixes

Over the course of the project, several non-trivial engineering issues were identified and resolved:

- **Dropped messages:** asynchronous request handling occasionally caused messages to be queued but never dispatched; this was fixed by introducing a queueing mechanism and ensuring each new request waits for previous generations to complete.
- **UI freezes:** an automatic chat-titling feature was blocking the main thread while waiting for an LLM response; this was ultimately removed to restore stability.
- **Prompt bloat:** early designs appended tool instructions even when tools were disabled, inflating prompts and degrading local model performance; the system was revised so tool instructions are only added when tools are active.
- **File access errors:** file-reading tools originally returned generic errors when paths were invalid; error messages were improved to include attempted paths and guidance for users.

These iterations materially improved reliability and user experience, moving the project from experimental prototype toward a robust daily driver.

Discussion

Lessons on Local Models

Empirical testing showed that very small local models (around 1B parameters) can provide reasonable conversational quality for short, simple prompts but currently struggle under the demands of tool-rich, long-context agent frameworks.

The combination of lengthy system prompts, tool schemas, and multi-step reasoning pushes these models beyond their effective capacity, leading to incoherent outputs and slow generations.

This suggests that, at present, there is a practical lower bound on model size for robust local agents with tools on consumer laptops; either prompts must be aggressively distilled, or users must accept higher resource usage and latency for larger models.

At the same time, the rapid improvement of 3B–4B-class models observed during the project indicates that this constraint is softening over time; future iterations may be able to rely more heavily on offline agents as model quality improves.

Design Trade-Offs

Several important trade-offs shaped the final design:

- **UI vs. platform:** the project shifted from building a custom chat UI to integrating with and then effectively re-implementing a more focused macOS workspace, recognizing that the unique value lies in the context system and agent architecture rather than yet another frontend.
- **MCP vs. in-app tools:** early exploration of MCP servers led to a later focus on a single integrated app where tools have first-class access to macOS APIs and TCC permissions.
- **Automatic features vs. stability:** some non-essential features (such as auto-titling chats) were removed when they caused instability, reflecting a priority for reliability over minor convenience.

Limitations

Despite substantial progress, several limitations remain:

- iMessage ingestion is still under active development; bugs in ingestion and chunking need to be resolved before it can be considered production-ready.
- Local tool-calling agents remain constrained by model size and laptop hardware; many complex workflows still benefit from or require cloud models.
- The system currently targets macOS, with no Windows or Linux support, limiting the audience to Apple Silicon users.

Addressing these limitations would be a natural future work.

Conclusion

This project demonstrates that it is feasible to build a local-first, privacy-preserving AI workspace on macOS that combines chat, retrieval, memory, and tools into a coherent agentic system.

Through an iterative, year-long development process, the project produced a native, open source

macOS application that can ground LLMs in a user's life data via a unified Brain, orchestrate tools through a lazy-loaded skill system, and flexibly route between local and cloud models.

The work highlights both the promise and current constraints of local models, the importance of explicit retrieval contracts for trust and privacy, and the practical engineering challenges of building a robust agent on consumer hardware.

As local models and hardware continue to improve, systems like this one can play a key role in making powerful, context-aware AI accessible to users who care deeply about privacy and control over their own data.

References

- Amazon. (2024). *What is RAG? - Retrieval-Augmented Generation Explained - AWS*. Amazon Web Services, Inc. <https://aws.amazon.com/what-is/retrieval-augmented-generation/>
- Divyansh. (2023, July 4). *Text Vectorization: Transforming Words into Meaningful Representations*. Medium.
<https://medium.com/@divyansh3021/text-vectorization-transforming-words-into-meaningful-representations-e03b31bb3cb3>
- Gmail API Overview*. (2025). Google for Developers.
<https://developers.google.com/workspace/gmail/api/guides>
- Hugging Face. (n.d.). *sentence-transformers/all-MiniLM-L6-v2* · Hugging Face. Huggingface.co. Retrieved January 1, 2026, from
<https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2>
- Introduction to OAuth 2.0 | Apigee*. (n.d.). Google Cloud. Retrieved January 6, 2026, from
<https://cloud.google.com/apigee/docs/api-platform/security/oauth/oauth-introduction>
- Jha, A. (2021, July 1). *Vectorization Techniques in NLP [Guide]*. Neptune.ai.
<https://neptune.ai/blog/vectorization-techniques-in-nlp-guide>
- LangChain Accounts. (2023, March 13). *LLMs and SQL*. Langchain.com; LangChain Blog.
<https://blog.langchain.com/llms-and-sql/>
- Mei, B. (2024, February 7). *billmei/every-chatgpt-gui*. GitHub.
<https://github.com/billmei/every-chatgpt-gui>
- OpenAI. (2025). *Memory FAQ | OpenAI Help Center*. Openai.com.
<https://help.openai.com/en/articles/8590148-memory-faq>
- Recommended hardware for running LLMs locally*. (2023, December 16). Hugging Face Forums.
<https://discuss.huggingface.co/t/recommended-hardware-for-running-llms-locally/66029>

Schwaber-Cohen, R. (2023, May 3). *What is a Vector Database?* Pinecone.

<https://www.pinecone.io/learn/vector-database/>