Brain

Privacy-First Personal AI Assistant

Architecture & Implementation Guide

Generated: January 12, 2026

Executive Summary

Brain is a privacy-first, MacOS-native personal AI assistant designed for complete data sovereignty and end-to-end encrypted communication. The system uses Obsidian as the canonical knowledge base, Signal for secure messaging, and a pluggable LLM architecture supporting both cloud APIs (Claude) and future local models (via Ollama).

The architecture prioritizes modularity and progressive enhancement, with four distinct implementation phases progressing from text-based interaction to full voice and phone capabilities. All components run locally on a MacBook Pro with the exception of LLM API calls, ensuring complete control over personal data.

Feature	Implementation
Knowledge Base	Obsidian (local markdown files)
Vector Search	Qdrant + Ollama embeddings
Messaging	Signal (E2EE via signal-cli)
LLM Backend	LiteLLM (Claude → future Ollama)
Agent Framework	Pydantic AI
Operational State	PostgreSQL
System Integration	Hammerspoon + PyXA

System Architecture

Core Principles

1. Obsidian as Canonical Source: All knowledge, conversations, and extracted facts live in Obsidian markdown files. This provides human-readable, version-controllable, tool-agnostic storage.

2. Privacy-First Design: Signal provides end-to-end encryption for all text communications. Local processing via Ollama eliminates dependency on cloud providers for embeddings.

3. Pluggable LLM Architecture: LiteLLM provides a unified API allowing seamless switching between Claude (cloud), GPT-4, Gemini, or local models via Ollama (Llama, Mistral, etc.).

4. Separation of Concerns: Canonical data (Obsidian), operational state (PostgreSQL), and derived data (Qdrant embeddings) are cleanly separated.

Data Flow Architecture

The system operates in three distinct layers:

Layer	Components	Purpose	Persistence
Canonical	Obsidian vault (markdown files)	Knowledge base, conversations, extracted facts	Permanent (Git/cloud backup)
Operational	PostgreSQL	Pending tasks, action logs, queue state	Short-term (archived to Obsidian)
Derived	Qdrant vector DB	Embeddings for semantic search	Regeneratable (from Obsidian)

Core Components

Obsidian + Smart Connections

Obsidian serves as the primary knowledge management system with the Smart Connections plugin providing in-UI semantic search. The Local REST API plugin exposes programmatic access for the agent to query and modify notes.

Key Configuration: Smart Connections operates independently with its own embeddings (bge-micro-v2) stored in .smart-env/. This is separate from the agent's Qdrant-based system, allowing UI convenience without coupling to the agent architecture.

Vector Search: Qdrant + Ollama

The agent's semantic search infrastructure uses Qdrant as the vector database and Ollama for local embedding generation. This combination provides fast, privacy-preserving similarity search across the entire knowledge base.

Embedding Model: nomic-embed-text (768 dimensions) via Ollama provides high-quality embeddings with Apache 2.0 licensing. The model runs locally using GPU acceleration on Apple Silicon.

Indexing Strategy: Markdown files are chunked at heading boundaries (## and ###) with 500-token chunks and 50-token overlap. YAML frontmatter and #tags are extracted as filterable metadata.

LLM Abstraction: LiteLLM

LiteLLM provides a unified API across 100+ models in OpenAI format. This enables seamless switching between cloud providers (Claude, GPT-4, Gemini) and local models (Ollama) by simply changing the model parameter.

Example usage: `completion(model='claude-sonnet-4-20250514', messages=...)` or `completion(model='ollama/llama3.1', messages=...)`

Benefits: Cost tracking, request caching, automatic retry/fallback logic, and observability at ~8ms P95 latency even under load.

Agent Framework: Pydantic AI

Pydantic AI provides type-safe agent orchestration with FastAPI-style ergonomics. The framework supports structured outputs via Pydantic models, human-in-the-loop approvals for sensitive actions, and durable execution for long-running workflows.

Alternative Considered: LangGraph provides more sophisticated state machine capabilities for complex multi-step workflows, but Pydantic AI's simpler model is better suited for personal assistant use cases.

Secure Messaging: Signal via signal-cli

Signal provides end-to-end encrypted messaging with the assistant accessible via text from anywhere. The signal-cli tool (wrapped in signal-cli-rest-api Docker container) enables programmatic access while maintaining Signal's E2EE guarantees.

Architecture: signal-cli runs as a linked device (like Signal Desktop) on the MacBook. Messages sent from the iPhone Signal app are received by the agent via webhook or polling, processed, and responses sent back through Signal.

Security Note: Unlike iMessage (where Apple manages keys), Signal's E2EE is verifiable and provides stronger security guarantees for sensitive commands.

Implementation Roadmap

Phase 1: Text Interaction & PKM

Status: Current phase - building foundational text-based assistant

Component	Purpose	Status
Obsidian + Smart Connections + REST API	Knowledge base with UI semantic search	Configured
Qdrant (Docker)	Vector database for agent queries	Ready to deploy
Ollama (native)	Local embedding generation	Installed
LiteLLM	LLM abstraction layer	Ready to integrate
Pydantic AI	Agent orchestration	Implementation pending
PostgreSQL (Docker)	Operational state storage	Ready to deploy
signal-cli-rest-api (Docker)	E2EE messaging interface	Ready to deploy
Hammerspoon	MacOS system automation	To be configured

Deliverables: Agent responds to Signal messages, searches knowledge base via Qdrant, creates/queries Obsidian notes, integrates with Calendar and Reminders, executes background tasks, and logs all actions to PostgreSQL.

Phase 2: Local Voice Interface

Timeline: After Phase 1 completion (est. 4-6 weeks)

Component	Purpose
whisper.cpp (Metal-optimized)	Local speech-to-text with Apple Silicon GPU acceleration
Piper	Fast local text-to-speech (sub-50ms latency)
openWakeWord	Custom wake word detection ("Hey Brain")
Optional: Pipecat	Real-time conversational voice (if natural dialogue needed)

Use Case: At-desk voice interaction with microphone. Wake word triggers listening, whisper.cpp transcribes speech, agent processes, and Piper speaks the response.

Phase 3: POTS Phone Support

Timeline: After Phase 2 (est. 8-12 weeks from start)

Components: Twilio Voice with Media Streams for phone number and real-time audio. WebSocket server handles bidirectional audio streaming. whisper.cpp and Piper integrated into call flow.

Cost: Approximately $1-2/month for phone number rental, $0.0085/min incoming, $0.013/min outgoing.

Use Case: Call the assistant from any phone, anywhere. Full conversational AI over traditional phone lines.

Phase 4: SMS Fallback

Timeline: After Phase 3 or in parallel

Components: Google Voice integration for non-E2EE SMS. Token-based authentication for untrusted channel.

Use Case: Convenience backup when Signal is unavailable or messaging from someone else's phone. Lower security but higher accessibility.

Technical Implementation Details

Docker Services Configuration

All services except Ollama (GPU acceleration required) and native MacOS integrations run in Docker containers orchestrated via docker-compose:

Service	Port	Purpose
qdrant	6333	Vector database
redis	6379	Task queue and caching
postgres	5432	Operational state storage
signal-api	8080	Signal messaging interface
brain-agent	-	Main agent daemon

Project Structure

The project follows a modular architecture with clear separation between configuration, core agent logic, and tool implementations:

Directory/File	Purpose
docker-compose.yml	Service orchestration
Dockerfile	Agent container definition
pyproject.toml	Python dependencies (Poetry)
.env	Configuration and API keys
src/agent.py	Main agent daemon
src/indexer.py	Obsidian vault indexer
src/config.py	Settings management
src/models.py	Database and API models
src/tools/obsidian.py	Obsidian REST API client
src/tools/calendar.py	Calendar integration
src/tools/reminders.py	Reminders integration
data/	Docker volumes (gitignored)
logs/	Application logs

Memory Management Strategy

Current Approach (Phase 1)

Simple conversation history management with last N messages in context plus RAG retrieval from Qdrant. Conversations are written to Obsidian markdown files for permanent storage.

Future Enhancement: MemGPT/Letta

If context management becomes unwieldy, the MemGPT/Letta framework can be integrated. Letta provides agent-controlled memory management with two tiers:

Core Memory (~2KB): Always in context. Contains essential facts about the user (preferences, key info) and agent persona. The agent updates via core_memory_append() and core_memory_replace() tool calls.

Archival Memory (unlimited): Stored in Qdrant. Full conversation history and retrieved facts. The agent explicitly searches via archival_memory_search(query) and inserts via archival_memory_insert(content).

Key Innovation: Instead of the developer deciding what context to inject, the LLM agent actively manages its own memory, deciding what to remember, forget, and retrieve.

Integration Note: If Letta is adopted, it would replace Pydantic AI as the agent runtime. Letta and Pydantic AI cannot be used simultaneously—they are alternative orchestration approaches.

Security & Privacy Considerations

Data Sovereignty

All canonical data (knowledge base, conversations, preferences) resides locally in Obsidian markdown files. Operational state (pending tasks, action logs) lives in a local PostgreSQL instance. Vector embeddings are derived and regeneratable.

External Dependencies: The only data leaving the MacBook is LLM API requests to Anthropic (Claude) or future cloud providers. These contain conversation context and query text but no persistent personal data.

End-to-End Encryption

Signal provides cryptographic E2EE for all text communications. Unlike iMessage (where Apple manages keys), Signal's protocol is open source and independently verifiable. The signal-cli implementation maintains these security guarantees when running as a linked device.

API Key Management

API keys and sensitive configuration live in .env file (excluded from version control via .gitignore). For production deployment, consider using MacOS Keychain via the Python keyring library for additional protection.

Future Considerations

Local LLM Migration: When local models (Llama 4, Mistral, etc.) reach acceptable quality, the system can eliminate cloud LLM dependencies entirely by routing through Ollama. LiteLLM makes this a simple configuration change.

Database Encryption: PostgreSQL data-at-rest encryption could be added for additional operational state protection, though this data is short-lived and logged to Obsidian.

Development Workflow

Quick Start Commands

Start all services:

```
cd ~/brain && docker-compose up -d
```

View logs:

```
docker-compose logs -f agent
```

Rebuild after code changes:

```
docker-compose up -d --build agent
```

Re-index Obsidian vault:

```
poetry run python src/indexer.py --full-reindex
```

Access Qdrant dashboard:

```
open http://localhost:6333/dashboard
```

Backup Strategy

Critical (daily): Obsidian vault (already backed up via iCloud/Git)

Important (weekly): PostgreSQL data directory (~/brain/data/postgres)

Optional: Qdrant and Redis (regeneratable from Obsidian)

Testing Strategy

Pydantic AI includes built-in test mode for agent actions. This allows dry-run testing of tool calls without executing actual system modifications:

```
poetry run python src/agent.py --test "Create a reminder for tomorrow"
```

Immediate Next Steps

Week 1-2: Foundation

- ✓ Obsidian + Smart Connections + Local REST API configured
- ✓ Ollama installed with nomic-embed-text model
- ■ Extract bootstrap.zip to ~/brain
- ■ Configure .env with API keys and paths
- ■ Start Docker services (docker-compose up -d)
- ■ Link Signal device (QR code scan)
- ■ Run initial vault indexing

Week 3-4: Core Agent

- ■ Implement Pydantic AI agent skeleton
- ■ Connect Qdrant client for semantic search
- ■ Implement Obsidian tool (search, create, append)
- ■ Build Signal message handler (webhook/polling)
- ■ Add basic conversation logging to Obsidian
- ■ Implement PostgreSQL task/reminder storage
- ■ Test end-to-end: Signal → agent → Obsidian → response

Week 5-6: System Integration

- ■ Add Calendar integration (PyXA)
- ■ Add Reminders integration (PyXA)
- ■ Configure Hammerspoon hotkeys
- ■ Implement background task execution
- ■ Set up launchd for auto-start
- ■ Add LangSmith/Langfuse for observability

Phase 1 Complete → Phase 2 Planning

Once Phase 1 is stable with reliable text interaction, schedule Phase 2 kickoff to add voice capabilities. Estimated 4-6 weeks after Phase 1 start.

References & Resources

Key GitHub Repositories

Project	Repository	Purpose
Obsidian Local REST API	coddingtonbear/obsidian-local-rest-api	Programmatic vault access
Smart Connections	brianpetro/obsidian-smart-connections	Local RAG in Obsidian
signal-cli	AsamK/signal-cli	Command-line Signal client
signal-cli-rest-api	bbernhard/signal-cli-rest-api	REST wrapper for signal-cli
LiteLLM	BerriAI/litellm	LLM abstraction layer
Pydantic AI	pydantic/pydantic-ai	Type-safe agent framework
Khoj	khoj-ai/khoj	Reference second brain impl
whisper.cpp	ggml-org/whisper.cpp	Local STT (Phase 2)
Letta (MemGPT)	letta-ai/letta	Agent memory management

Documentation

• Anthropic API: https://docs.anthropic.com

• Qdrant Docs: https://qdrant.tech/documentation

• Ollama Documentation: https://ollama.com/docs

• Signal Protocol: https://signal.org/docs

• PyXA (MacOS Automation): https://github.com/SKaplanOfficial/PyXA

• Hammerspoon: https://www.hammerspoon.org/docs

Community Resources

• r/ObsidianMD - Obsidian community

• r/LocalLLaMA - Local LLM discussion

• Anthropic Discord - Claude API support

• Qdrant Discord - Vector DB help

Appendix: Design Decisions

Why Not Use Khoj?

While Khoj (31.8k GitHub stars) provides a comprehensive second brain solution, it doesn't align with the project's architectural requirements:

• Khoj is opinionated and monolithic, making component swapping difficult

• No Signal integration; uses its own chat interfaces

• Different memory model incompatible with potential MemGPT integration

• Includes UI/web app overhead not needed for headless assistant

The custom architecture provides precise control over each component, enabling the specific privacy, modularity, and extensibility requirements.

Signal vs iMessage

While iMessage automation is technically possible (accessing chat.db + AppleScript), Signal was chosen for:

• Verifiable end-to-end encryption (Signal protocol is open source)

• User-controlled key management (Apple manages iMessage keys)

• Cleaner API access via signal-cli without database hacking

• Better separation between personal messages and assistant commands

Pydantic AI vs LangGraph vs Letta

Pydantic AI (chosen): Clean Python API, type safety, simpler for personal projects. Best for straightforward agent workflows.

LangGraph: More complex state machine model. Better for multi-step workflows with branching logic. Overkill for Phase 1 but valuable reference.

Letta: Specialized in memory management. May replace Pydantic AI if context handling becomes complex. Cannot be used simultaneously with Pydantic AI.

PostgreSQL vs SQLite

PostgreSQL chosen over SQLite for operational state despite single-user context:

• Better concurrency handling if future multi-agent scenarios arise

• Richer feature set for complex queries on action logs

• Easy Docker deployment and backup strategies

• Consistent production-grade approach across all components

The containerized PostgreSQL instance has negligible overhead for personal use.