Refactoring Obsidian Mind into a Category/Tag Based Claude Memory Architecture

Architecture Proposal

***Abstract*—This paper proposes a compact refactor of Obsidian Mind's Claude memory approach. The refactor retains Obsidian Mind's strongest implementation detail: QMD-backed semantic retrieval exposed through agent tools. The goal is not to prove that one system can represent more workflows than the other. Both can represent decisions, implementation notes, bugs, standups, incidents, and review evidence. The difference is operational. Obsidian Mind combines semantic search with workflow-heavy folder routing; the proposed model keeps semantic search but moves note identity into one `category` field, with retrieval context represented by `tags`. This reduces routing branches for the agent and lowers memory-management overhead.**

***Index Terms*—Claude memory, Obsidian, agent memory, knowledge architecture, retrieval, tags, categories.**

I. Problem

OBSIDIAN Mind's useful contribution is not its exact folder tree, but its agent-memory mechanism: session-start injection, prompt classification, retrieval before action, write validation, and wrap-up checks. Those mechanisms solve a real problem for coding agents, which otherwise repeatedly lose project context, prior decisions, and debugging history between sessions.

The part that should be refactored is the ontology. In Obsidian Mind, folders such as `work/`, `org/`, `perf/`, `brain/`, and `reference/` are not merely storage locations. They also encode workflow meaning. That forces the agent to classify along multiple axes at once: what the note is, which workflow owns it, which folder receives it, and which side-effect indexes need to be updated.

The proposed model separates those responsibilities. Folders describe storage role, categories describe note identity, tags describe retrieval context, links describe semantic relationships, and Bases expose category/tag views. QMD remains the semantic retrieval engine in both approaches. The comparison is therefore not "semantic search versus no semantic search," but routing complexity around the same retrieval layer: for the same task, how many decisions must the agent make, and where can it fail?

II. Architecture

The architectural rule is simple: folders must not duplicate categories. Folders such as `Debugging/`, `Patterns/`, `Snippets/`, or `Architecture/` would reintroduce semantic folders. They should instead be values of the `category` property.

The resulting folder architecture is intentionally small:

TABLE I
SEPARATION OF RESPONSIBILITIES IN THE PROPOSED VAULT MODEL.

Layer	Responsibility
Folders	Storage role and source pipeline.
Categories	Single primary note identity.
Tags	Cross-cutting retrieval context.
Links	Semantic relationships and evidence.
Bases	Category/tag retrieval entry points.
Claude memory	Injected rules, active tags, recent context, and retrieval affordances.
QMD/RAG	Semantic candidate generation and note lookup.

```
Categories/ category and tag views
Clipping/ source-derived analysis
Attachments/ raw files, logs, screenshots
Inbox/ unprocessed dumps
Templates/ category templates
Cards/ optional review cards
Root level/ durable owned notes
.claude/ .codex/ .gemini/ agent config
```

A note in `Clipping/` can still be a lecture, an integration note, a debugging note, or review evidence. The folder only says that the note came from interpreting an external source. The category says what the note is.

III. Categories and Tags

For the coding context, the relevant categories are coding-memory roles rather than general second-brain types:

```
Project, Decision, Architecture, Implementation,
Debugging, Integration, Pattern, Snippet,
Environment, Incident, Meeting, Review, Inbox
```

These are identities, not folders. A bug note should be `category: Debugging`; it should not require a `Debugging/` folder. A reusable shell command should be `category: Snippet`; it should not require a `Snippets/` folder.

Tags should not duplicate categories. Good tags describe project, layer, domain, technology, or concern: `OfficeOS`, `Openclaw`, `Backend`, `MCP`, `Sandbox`, `Retrieval`, `TypeScript`, `Docker`, `Permissions`, or `DataModel`. Bad tags are note types such as `Decision`, `Debugging`, `Meeting`, or `Project`, because those belong in `category`.

IV. Retrieval and Workflow

At session start, the agent should receive a compact operating contract: folder rules, category registry, known project tags, common technical tags, Base entry points, recent changed note names, and the current Inbox/TODO count. It should

TABLE II
CAPTURE WORKFLOW COMPARISON FOR COMMON CODING MEMORY.

Task	Proposed model	Obsidian Mind friction
Technical decision	Root note; `category: Decision`; project and context tags.	Choose between active work, key decisions, reference, project note, or decision record.
External API insight	`Clipping/`; `category: Integration`; API/domain tags.	Could be reference, pattern, project update, or skill note.
Bug investigation	Root note; `category: Debugging`; project and concern tags.	Could be incident, gotcha, active work, project note, pattern, or review evidence.
Standup	`category: Meeting`; project/context tags; link to decisions if needed.	Dedicated workflow may update active work, people, tasks, and indexes.
Review evidence	`category: Review`; links to evidence notes.	Choose between brag doc, evidence folder, competency notes, and review cycle.

not load the whole vault. Obsidian Mind already implements the right retrieval primitive here: QMD semantic search, including embeddings, query/update/embed commands, and MCP tools such as `mcp__qmd__query`, `mcp__qmd__get`, and `mcp__qmd__multi_get`. The refactor should keep that mechanism.

Retrieval should become metadata-scoped RAG. To load context before implementing OfficeOS integrations, the agent first scopes candidates through category Bases filtered by the `OfficeOS` tag: `Project`, `Decision`, `Architecture`, `Implementation`, `Debugging`, `Integration`, and relevant `Pattern` notes. QMD then ranks or expands within that candidate set. The improvement is not that the refactor has RAG and Obsidian Mind does not; the improvement is that QMD is guided by stable category/tag metadata rather than by several workflow folders whose relevance depends on previous filing choices.

The same idea also explains the personal-vault case. A lecture note can be stored as `Clipping/ + category: Vorlesung + tag: Softwareentwurf`. Retrieval starts from `Categories/Vorlesung.base`. The coding equivalent is `root + category: Debugging + tags: [Openclaw, Permissions]`, retrieved through `Categories/Debugging.base`.

V. TOKEN ESTIMATE

The token estimate concerns memory-management overhead only, not the entire coding session. It also assumes both systems can use QMD. Actual implementation work, file reads, tool outputs, and debugging discussion may dominate total token use. The expected gain comes from reducing injected ontology, prompt-side routing, and exploratory follow-up after semantic search, not from replacing QMD.

Using midpoint estimates, the Obsidian Mind style costs roughly 9,750 memory-management tokens per normal coding session, while the refactored category/tag style costs roughly 4,975. The full estimated ranges are 6,300–13,200 tokens versus 3,150–6,800 tokens. This suggests roughly 35–55% less memory-management overhead. The estimate should be read as an upper-bound planning estimate: QMD already reduces raw retrieval cost in Obsidian Mind, so the remaining savings

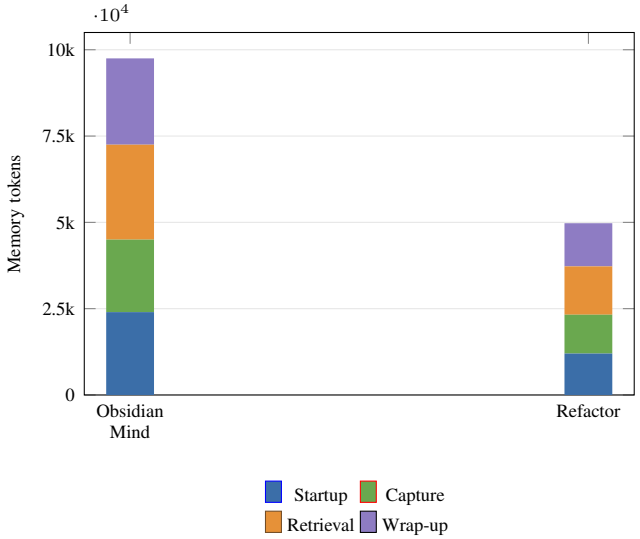

Fig. 1. Midpoint estimate of memory-management token overhead in a normal coding session with one startup, three captures, one retrieval workflow, and one wrap-up.

come from fewer ontology branches and cleaner post-retrieval routing.

VI. CONCLUSION

The refactor should keep Obsidian Mind's Claude memory mechanism, including QMD semantic search, but replace its folder ontology. Both architectures can store the same information and both can use RAG-style retrieval. The proposed model is preferable because it separates storage role from semantic identity: folders are architecture, categories are identity, tags are retrieval context, Bases provide metadata views, and QMD provides semantic ranking. That separation reduces routing branches, reduces duplicate filing, improves retrieval precision, and makes the agent's behavior easier to validate at wrap-up.