**1. Overview and Rationale**

**Design Goal:**  
Create a memory module that provides efficient, safe, and secure persistent storage and in-memory caching for user data. It must expose a clear API so that other components need not know any implementation details.

**Recommended Approach:**  
An embedded SQL database.

**2. Architecture Overview**

**Storage Layer**

* **Persistent Storage via Embedded Database:**  
  Use SQLite (or a similar lightweight SQL engine) as the primary data store because of its ACID properties, built-in file locking, support for atomic transactions, and cross-platform compatibility.

**Memory API Abstraction**

* **Modular API Design:**  
  Develop a clean Memory API that hides storage implementation details. This API should provide methods to add, update, retrieve, and securely delete memory entries. The API will serve as the entry point for all memory operations, ensuring consistency across components.
* **Data Format Definitions:**  
  Define clear and documented data formats (e.g., JSON-like structures) for input and output from the API. This allows for easy extension or modification in the future.

**3. Detailed Design Considerations**

**A. Stability and Speed**

* **File Locking & Concurrency:**  
  Leverage the concurrency features of the embedded database to handle simultaneous read/write operations. Ensure that additional file-system–level locks are used where necessary (e.g., when managing cache persistence or dealing with log files).
* **Atomic File Operations:**  
  Ensure that all write operations to disk are atomic. In the case of a failure, the system should be able to recover without corrupting the database. Databases like SQLite already support transactional writes that can be paired with careful file-level operations for any supplemental JSON configuration or cache persistence files.
* **User Data:**  
  When Jupiter recognises a user, user data should be extracted and stored in the user prompt for use.
* **Error Recovery:**  
  Implement monitoring and recovery strategies such that in the event of data corruption (identified through checksums or integrity scans), the system can either restore from a backup or attempt to repair the data where possible.

**B. Security**

* **Input Sanitization:**  
  All user inputs that are stored must be thoroughly sanitized. This includes stripping out malicious code or formatting issues before the data is processed or stored.
* **Data Structure Validation:**  
  Introduce robust validation routines that inspect data formats both at entry (write operations) and exit (read operations) to ensure that only valid and expected data is processed.
* **Secure Deletion:**  
  Include secure deletion methods so that user data can be permanently and securely erased from storage. This might involve overwriting data in memory and on disk to prevent recovery.

**C. Modularity**

* **Clean API Abstraction:**  
  Provide a well-documented Memory API that abstracts away the details of file operations, caching, and encryption. Other components should interact exclusively with this API.
* **Separation of Concerns:**  
  Split functionality into two main areas:
  + **Storage Logic:** Handles all direct interactions with the underlying embedded database and file system.
  + **Extraction Logic:** Manages the interpretation, formatting, and preparation of data for presentation to other modules.
* **Clear Data Formats:**  
  Define and document explicit data contracts for how memory entries are stored and retrieved. This ensures that any changes to storage formats are isolated from application logic.

**D. Maintainability**

* **Cross-Platform Identity Simplification:**  
  Standardize identity systems (e.g., using GUIDs or another consistent identifier) across all platforms. This simplifies data tracking regardless of the operating system.
* **Comprehensive Documentation:**  
  Maintain in-code and external documentation regarding the structure of data, the meaning of various fields, and the expected behavior of API calls. This documentation will be crucial for future developers.
* **Consistent Error Handling Patterns:**  
  Utilize a unified error handling strategy across read/write operations. This includes providing clear error messages, status codes, and logging mechanisms.
* **Data Validation on Read/Write:**  
  Integrate validation routines that verify data integrity on every read and write operation. These routines ensure that corrupted or malformed data is detected immediately.
* **Configuration-Driven Memory Categories:**  
  Consider a configuration-driven approach where memory categories (e.g., user preferences, session history, temporary caches) are defined in external files. This allows the system to be reconfigured easily without altering core code.
* **Incorporate configuration files** or environment settings that allow easy adjustment of memory categories and storage parameters without code changes.

**Jupiter Memory System - Database Schema Design**

**Core Schema**

**1. Users Table**

CREATE TABLE Users (

user\_id TEXT PRIMARY KEY, -- UUID as text

name TEXT NOT NULL,

created\_at INTEGER NOT NULL, -- Unix timestamp

last\_seen INTEGER NOT NULL, -- Unix timestamp

metadata TEXT -- JSON for extensibility

);

**2. Platforms Table**

CREATE TABLE Platforms (

platform\_id INTEGER PRIMARY KEY AUTOINCREMENT,

platform\_name TEXT UNIQUE NOT NULL -- 'gui', 'discord', 'terminal', etc.

);

**3. UserPlatforms Table**

CREATE TABLE UserPlatforms (

mapping\_id INTEGER PRIMARY KEY AUTOINCREMENT,

user\_id TEXT NOT NULL,

platform\_id INTEGER NOT NULL,

platform\_username TEXT NOT NULL,

platform\_specific\_id TEXT, -- e.g., Discord ID

last\_active INTEGER NOT NULL, -- Unix timestamp

FOREIGN KEY (user\_id) REFERENCES Users(user\_id),

FOREIGN KEY (platform\_id) REFERENCES Platforms(platform\_id),

UNIQUE (platform\_id, platform\_username),

UNIQUE (platform\_id, platform\_specific\_id)

);

**4. KnowledgeCategories Table**

CREATE TABLE KnowledgeCategories (

category\_id INTEGER PRIMARY KEY AUTOINCREMENT,

category\_name TEXT UNIQUE NOT NULL, -- 'location', 'likes', etc.

data\_type TEXT NOT NULL, -- 'string', 'list', 'date', etc.

is\_personal BOOLEAN NOT NULL DEFAULT 0

);

**5. UserKnowledge Table**

CREATE TABLE UserKnowledge (

knowledge\_id INTEGER PRIMARY KEY AUTOINCREMENT,

user\_id TEXT NOT NULL,

category\_id INTEGER NOT NULL,

value TEXT NOT NULL, -- Actual data (JSON for lists)

confidence REAL NOT NULL DEFAULT 1.0, -- 0.0 to 1.0

source TEXT NOT NULL, -- 'user\_stated', 'extracted', 'edited'

created\_at INTEGER NOT NULL,

updated\_at INTEGER NOT NULL,

FOREIGN KEY (user\_id) REFERENCES Users(user\_id),

FOREIGN KEY (category\_id) REFERENCES KnowledgeCategories(category\_id)

);

**Conversation Logging Integration**

**6. Conversations Table**

CREATE TABLE Conversations (

conversation\_id TEXT PRIMARY KEY, -- UUID

user\_id TEXT NOT NULL,

platform\_id INTEGER NOT NULL,

started\_at INTEGER NOT NULL,

ended\_at INTEGER, -- NULL if ongoing

extracted BOOLEAN NOT NULL DEFAULT 0, -- Flag if processed

FOREIGN KEY (user\_id) REFERENCES Users(user\_id),

FOREIGN KEY (platform\_id) REFERENCES Platforms(platform\_id)

);

**7. Messages Table**

CREATE TABLE Messages (

message\_id INTEGER PRIMARY KEY AUTOINCREMENT,

conversation\_id TEXT NOT NULL,

is\_from\_user BOOLEAN NOT NULL, -- TRUE if from user, FALSE if from Jupiter

content TEXT NOT NULL,

timestamp INTEGER NOT NULL, -- Unix timestamp

FOREIGN KEY (conversation\_id) REFERENCES Conversations(conversation\_id)

);

**8. ExtractionJobs Table**

CREATE TABLE ExtractionJobs (

job\_id INTEGER PRIMARY KEY AUTOINCREMENT,

conversation\_id TEXT NOT NULL,

status TEXT NOT NULL, -- 'pending', 'processing', 'completed', 'failed'

created\_at INTEGER NOT NULL,

completed\_at INTEGER, -- NULL if not completed

error TEXT, -- NULL if no error

FOREIGN KEY (conversation\_id) REFERENCES Conversations(conversation\_id)

);

**Indexes for Performance**

CREATE INDEX idx\_userplatforms\_user\_id ON UserPlatforms(user\_id);

CREATE INDEX idx\_userknowledge\_user\_id ON UserKnowledge(user\_id);

CREATE INDEX idx\_userknowledge\_category ON UserKnowledge(category\_id);

CREATE INDEX idx\_messages\_conversation ON Messages(conversation\_id);

CREATE INDEX idx\_conversations\_user\_id ON Conversations(user\_id);

CREATE INDEX idx\_conversations\_extracted ON Conversations(extracted);

**Conversation Logging Integration Approach**

The conversation logging integration offers several benefits:

1. **Complete Audit Trail**: Every interaction is stored with timestamps and context
2. **Asynchronous Extraction**: The LLM extraction process can happen:
   * At the end of a conversation
   * During idle times
   * In a separate thread/process
3. **Source Tracking**: Each piece of knowledge is linked to its source message, allowing verification
4. **Confidence Management**: Knowledge can be stored with confidence levels (higher for direct statements, lower for inferences)
5. **Privacy Control**: The is\_personal flag can help manage sensitive information

**Implementation Notes**

1. **Initial Data**: Populate the KnowledgeCategories table with predefined categories at setup
2. **Extraction Process**:
   * Create an ExtractionJob when a conversation ends
   * Process messages through the LLM
   * Compare new information with existing knowledge
   * Apply updates with appropriate confidence levels
3. **API Interface**: The Memory API would provide methods to:
   * Retrieve a user profile (all knowledge categorised)
   * Log conversation messages
   * Schedule extraction jobs
   * Get/update specific knowledge items

This schema provides a robust foundation for Jupiter's memory system while enabling the conversation logging capabilities you requested. It separates the storage concerns from application logic while maintaining data integrity and relationships.