Milla Brain Functional Specification

Milla Project Docs

December 2, 2025

1 Executive Summary

Milla Brain is a Clojure-based wrapper around a local Ollama LLM instance that:

- Provides a simple Lisp-like DSL `(fact ...)`, `(desire ...)`, `(opinion ...)` to store persistent knowledge about the user and environment.
- Logs chat history (user prompts and model replies) to a SQLite database.
- Exposes command-line and REPL/Emacs entrypoints for interactive use.
- Supports multi-node "kittens": smaller Milla instances (e.g., a car/laptop node) that can synchronize their local brain with the home Milla via database synchronization.
- Supports creation of a backlog of tasks and ideas via a dedicated tag so that Milla or her kittens can mark things to revisit later.
- Is designed to be operated from Emacs (locally or remotely) so that the user can use it as a cognitive co-processor.

In later iterations, Milla Brain may be extended to propose and apply self-modifying changes to its own Clojure code and configuration, under human supervision and with safety checks.

2 Goals and Non-Goals

2.1 Goals

G1. **Persistent Memory**:
- Store structured "facts," "desires," and "opinions" expressed via Clojure functions.
- Store full chat transcripts (user and assistant) with timestamps.
- Store backlog items, tagged explicitly, so that the system can track tasks to revisit later.

G2. **Local LLM Integration (Ollama)**:
- Send prompts to Ollama via HTTP.
- Allow specifying `model`, `prompt`, and optional `system` prompt.
- Persist both sides of the interaction.
- Preferred runtime: long-lived HTTP daemon (`bin/milla-serve`); CLI forwards to the daemon if running.

G3. **Developer-Friendly Interface**:
- Provide a simple CLI for quick calls.
- Provide a REPL-first workflow (CIDER + Emacs).

G4. **Multi-node "Kitten" Sync**:
- Allow one-way and two-way synchronization of SQLite databases between:
  - Home Milla (canonical or primary node), and
  - Remote Milla kittens (e.g., mobile or experimental nodes).
- Use simple mechanisms such as `rsync` or attach/merge via SQLite.

G5. **Remote Emacs Operation**:
- Allow users to SSH into Milla, run Emacs locally or remotely, and control Milla Brain's REPL from Emacs (CIDER, TRAMP, ssh).

G6. **Self-Reflection and Proposed Self-Modification (Experimental)**:
- Allow Milla and her kittens to *propose* changes to their own code and configuration (e.g., refactors, new helper functions) and store these proposals as backlog items.
- Support controlled, human-reviewed application of such changes to avoid unsafe or destructive behavior.

2.2 Non-Goals

- Full-featured web UI (may be added later).
- Complex automated conflict resolution for database synchronization; the initial version may treat:
  - The home node as canonical, or
  - Use a simple union-by-ID/timestamp strategy with manual conflict resolution.
- Authentication and authorization for the local CLI/Ollama API (assume trusted LAN for now).

3 System Overview

Milla Brain is both:

- A local service (Clojure code + SQLite DB + Ollama) and
- A library that can be used from:
  - Command line (`bin/milla`, `clj -M -m milla.core ...`),
  - Clojure REPL (CIDER/Emacs),
  - Future: HTTP API or other UIs.

High-level flow:

1. User issues a prompt (via CLI or REPL) or calls one of the DSL forms (`fact`, `desire`, `opinion`, `backlog`).
2. Clojure code writes structured records to SQLite.
3. For chat:
   3.1. Construct a request to Ollama's `/api/chat` (normalized even if config points at `/api/generate`).
   3.2. Send the request via `clj-http`.
   3.3. Receive the model's reply and log it to the `chat` table.
4. For statements (facts/desires/opinions/backlog):
   4.1. Insert rows into `statements` with appropriate kinds/tags.
5. For kittens:

5.1. Periodically or on demand, a synchronization script uses `rsync` or a merge procedure to synchronize SQLite databases between nodes.

4 Repository Structure

Proposed top-level layout:

```
milla-clj/
|-- README.md                 ; project overview, quickstart
|-- deps.edn                  ; Clojure deps
|-- .gitignore
|-- milla_memory.db           ; SQLite database (ignored by git)
|-- src/
|   `-- milla/
|       |-- core.clj          ; main API: ask!, fact/desire/opinion, init!
|       |-- memory.clj        ; recall logic, system-prompt builder (planned)
|       `-- sync.clj          ; helpers for DB sync (planned)
|-- bin/
|   |-- milla                 ; CLI wrapper script
|   |-- milla-init-db         ; script to ensure DB schema exists
|   `-- milla-sync-db         ; script to sync kitten <-> home DB
`-- doc/
    `-- milla-brain-spec.tex  ; this functional spec
```

Additional files (e.g., `memory.clj`, `sync.clj`) can be introduced as functionality grows.

5 Key Files

5.1 README.md (Outline)

The repository's `README.md` should include at least:

R1. Project name and purpose: "Milla Brain – Clojure/Ollama memory wrapper".

R2. Feature summary:
- Local LLM (Ollama) integration.
- Persistent memory: fact/desire/opinion/backlog DSL.
- Chat logging to SQLite.
- Multi-node kittens with DB synchronization.
- Emacs/CIDER integration.

R3. Requirements:
- Java (OpenJDK),
- Clojure CLI (`clj`),
- Ollama running on `localhost:11434`,
- SQLite (through `org.xerial/sqlite-jdbc`).

R4. Installation and setup:
- Clone the repository,

- Run `bin/milla-init-db` or `clj -M -m milla.core` once to initialize the DB.

R5. Usage (CLI and REPL examples).
R6. Database synchronization examples (`bin/milla-sync-db` + `rsync`).
R7. Emacs integration notes (CIDER, TRAMP, remote workflows).

6 Functional Requirements

6.1 Core Memory DSL

FR-1: Fact Storage The system shall provide a function:

```
(fact & strings)
```

Each string shall be stored as a separate row in `statements` with:

- `kind = "fact"`,
- `text = <string>`,
- `created_at = current timestamp`,
- `source_node = <this node ID>`.

FR-2: Desire Storage The system shall provide:

```
(desire & strings)
```

Each string shall be stored as `kind = "desire"` with the same metadata fields as facts.

FR-3: Opinion Storage The system shall provide:

```
(opinion & strings)
```

Each string shall be stored as `kind = "opinion"` with the same metadata fields as facts.

FR-4: Backlog Storage The system shall provide a way to create backlog items. This may be either:

- A separate function:

  ```
  (backlog & strings)
  ```

- Or a tagged variant of `fact`/`desire` that sets a `backlog` flag.

Each backlog string shall be stored with:

- Either `kind = "backlog"` *or* a combination of `kind` and `tag`,
- `text = <string>`,
- `created_at = current timestamp`,
- `source_node = <this node ID>`.

The backlog shall be queryable (e.g., by a function `(backlog-items)`), so that Milla or a human can periodically review tasks and ideas "to deal with later."

FR-5: Initialization The function `(init!)` shall ensure all required tables exist, creating them if they do not.

6.2 Chat Interface

FR-6: Ask the Model The system shall provide:

```
(ask! {:keys [model prompt system session]})
```

- Required keys: `model`, `prompt`.
- Optional keys:
  - `system`: string containing additional instructions and possibly facts/desires/opinions/backlog context.
  - `session`: string to group related messages (`"default"` if omitted).

Behavior:

1. Ensure the database schema exists via `init!`.
2. Insert a user message into the `chat` table with:
   - `role = "user"`,
   - `model`, `content = prompt`,
   - `session`,
   - `created_at`, `source_node`.
3. Call Ollama's `/api/chat` endpoint with:
   - `model`,
   - `messages` (chat format; includes system prompt plus history/summaries),
   - `stream = false`,
   - `keep_alive` (default `10m`, configurable),
   - `options.num_ctx` (prompt context size; default 2000, configurable).
4. Insert an assistant message into `chat` with:
   - `role = "assistant"`,
   - `content = response text`,
   - `model`, `session`, `created_at`, `source_node`.
5. Return the response text.

6.3 Recall and System Prompt

FR-7: Recall Statements The system shall provide:

```
(all-statements)
(facts)
(desires)
(opinions)
(backlog-items)
```

- `all-statements` returns all rows from `statements`, sorted by `created_at`.
- `facts`, `desires`, `opinions` filter by `kind`.
- `backlog-items` returns statements marked as backlog tasks.

FR-8: Build System Prompt The system shall provide:

```
(build-system-prompt)
```

This function shall:

- Collect facts, desires, opinions, and optionally a summary of backlog items.
- Include node id/location (from config) for context.
- Include recent chat history up to the configured window.
- Include rolling summaries: when unsummarized history exceeds \$1.5 $\times$\$$theconfiguredwindow, theoldest$\$N$\$

- Serialize them into a structured system message, for example:
- Optional RAG: when enabled, relevant facts/summaries from SQLite embeddings (via an Ollama embedding model) are pulled into a RAG CONTEXT block in the system prompt (top-K, score threshold configurable).

```
You are Milla, the user's local assistant.

FACTS:
- ...

DESIRES:
- ...

OPINIONS:
- ...

BACKLOG (for reference, do not execute actions):
- ...
```

By default, `ask!` or a wrapper may use `build-system-prompt` whenever a caller does not specify an explicit `system` prompt.

6.4 Multi-node Synchronization (Kittens)

FR-9: Node Identification Each node (home Milla or kitten) shall have a configurable `node_id` string, such as:

- `"home-milla"`,
- `"car-kitten"`,
- `"laptop-kitten"`.

The `node_id` shall be stored with each row in both `statements` and `chat` as `source_node`.

FR-10: One-way Sync Script The script `bin/milla-sync-db` shall support:

```
./bin/milla-sync-db pull user@host:/path/to/milla_memory.db
./bin/milla-sync-db push user@host:/path/to/milla_memory.db
```

MVP behavior:

- Use `rsync` over SSH to copy the remote DB to a local temporary file (for pull), or copy the local DB to the remote path (for push).
- After a pull, invoke a Clojure-based merge function to incorporate remote rows into the local DB.

FR-11: Merge Strategy The merge function (e.g., in `milla.sync/merge`) shall:

1. Attach the remote DB as a secondary database in SQLite.
2. For each relevant table (`statements`, `chat`):
   - Insert rows from the remote table into the local table where they do not already exist, based on:
     - ID, or
     - a generated UUID, or
     - a uniqueness heuristic on `(source_node, created_at, kind, text)`.
3. Conflicts (e.g., same ID but different content) shall be logged and skipped in the first version.

6.5 Remote Emacs Operation

FR-12: Support for Remote REPL The system shall be usable with at least the following workflows:

1. **SSH + Tmux + Emacs on Milla**:
   - User SSHs into Milla.
   - Starts `tmux`.
   - Runs Emacs on Milla.
   - Uses `M-x cider-jack-in-clj` inside the `milla-clj` project.
2. **Local Emacs + TRAMP**:
   - Open remote files via TRAMP (e.g., `/ssh:dan@milla:/home/dan/src/milla-clj/...`).
   - Start a remote CIDER REPL via TRAMP-aware jack-in or `cider-connect` to a remote nREPL port.

The project shall be a standard Clojure CLI project (`deps.edn`) and shall avoid hardcoded machine-specific paths outside of its own directory tree.

6.6 Self-Modification and Backlog-Driven Refactoring

FR-13: Proposal of Code Changes Milla and her kittens may *propose* changes to their own code and configuration, but those changes shall be captured as backlog items rather than applied blindly. For example:

```
(backlog "Refactor milla.core/ask! to support streaming responses.")
(backlog "Add milla.memory namespace with smarter recall ranking.")
```

Each such item shall be a record in `statements` (or a dedicated backlog table) and reviewed by a human before implementation.

FR-14: Human-in-the-Loop Application Any automated or semi-automated code modification mechanism (e.g., self-rewriting functions) shall:

- Operate on a separate working copy (e.g., a branch or temporary file),
- Be subject to human review and explicit approval (e.g., `git diff`, tests),
- Avoid directly mutating the live, running system without restart or explicit confirmation.

FR-15: Safety and Testing A future self-modification workflow should ideally:

1. Generate candidate changes (patches, new functions, configuration tweaks).
2. Store them as backlog items with attached code snippets.
3. Allow running tests or checks (e.g., `clj -M:test`) before application.
4. Apply changes only after manual inspection.

The initial version of Milla Brain does *not* require fully autonomous self-modifying behavior; the specification only requires that proposals and backlog items be representable and traceable.

7 Database Design

7.1 Tables

7.1.1 `statements`

Column	Type	Description
`id`	INTEGER	Primary key (auto-increment).
`kind`	TEXT	`"fact"`, `"desire"`, `"opinion"`, `"backlog"`, etc.
`text`	TEXT	The statement body.
`created_at`	TEXT	ISO-8601 timestamp.
`source_node`	TEXT	Node ID where this was created (e.g., `"home-milla"`).

7.1.2 `chat`

Column	Type	Description
`id`	INTEGER	Primary key (auto-increment).
`role`	TEXT	`"user"` or `"assistant"`.
`model`	TEXT	Model name, e.g., `"llama3.2"`.
`content`	TEXT	Message text.
`session`	TEXT	Session identifier (nullable or default).
`created_at`	TEXT	ISO-8601 timestamp.
`source_node`	TEXT	Node ID (e.g., `"home-milla"`, `"car-kitten"`).

7.2 Logical Diagram (ASCII ERD)

```
| kind        (T)   |           | role        (T)   |
| text        (T)   |           | model       (T)   |
| created_at  (T)   |           | content     (T)   |
| source_node (T)   |           | session     (T?)  |
+-------------------+           | created_at  (T)   |
                                | source_node (T)   |
                                +-------------------+
```

- statements: global knowledge (facts/desires/opinions/backlog)
- chat: timeline of interactions with the model
- source_node: where the row came from (for sync + auditing)
- session: group of chat messages for a conversation

8 Setup and Scripts

8.1 bin/milla-init-db

Purpose: ensure the SQLite schema is present without needing to call `ask!`.

Example behavior:

```
#!/bin/sh
cd "\$(dirname "\$0")/.."
clj -M -e "(require 'milla.core)
           (milla.core/init!)
           (System/exit 0)"
```

8.2 bin/milla (CLI wrapper)

Purpose: friendly CLI for `ask!`.

Example:

```
#!/bin/sh
cd "\$(dirname "\$0")/.."
clj -M -m milla.core "\$@"
```

Usage:

```
./bin/milla llama3.2 "Say hello from Milla and mention Gabriel."
```

8.3 bin/milla-sync-db

Purpose: synchronize Milla Brain DB between home and a kitten using `rsync` and merge.

- Pull: `bin/milla-sync-db pull user@host:/path/to/milla_memory.db` (rsyncs to `milla_memory.remote` then merges into local via `milla.sync/merge`).

- Push: `bin/milla-sync-db push user@host:/path/to/milla_memory.db` (rsyncs local to remote).

9 Non-Functional Requirements

9.1 Configuration

Runtime configuration is stored in a YAML file (by default `milla-config.yaml`; `config/milla.yaml` and `MILLA_CONFIG` overrides are also supported). A sample configuration:

```
db:
  path: milla_memory.db

node:
  id: home-milla
  location: unknown

ollama:
  url: http://localhost:11434/api/chat
  default_model: llama3.2
  keep_alive: 10m

chat:
  default_session: default
  history_limit: 50

prompt:
  max_tokens: 2000

thermal:
  enabled: false
  max_c: 85
  cooldown_ms: 120000
  sensor_path: /sys/class/thermal/thermal_zone0/temp

server:
  port: 17863
  pid_file: milla.pid
  heartbeat_ms: 5000
  # server refuses to start if heartbeat is fresh; restart via bin/milla-restart-server
```

Merging: use `bin/milla-merge /path/to/output.db /path/to/db1 /path/to/db2` to union and dedupe statements/chat/chat_summaries, then re-run summarization per session. Global summary: merge also writes a bullet summary of all chat into a `global_summary` table. Global summary: merge also writes a bullet summary of all chat into a global$_summarytable$.

First-run setup All helper scripts support -h/–help for usage info.

Script `bin/milla-setup` prompts the user for node location, facts, desires, opinions, and backlog items and stores them via the Clojure API. It is optional but recommended on first launch.

Conversation history strategy: keep the last 5 messages verbatim; summarize older messages into a rolling summary per session (included in the system prompt). Recent messages are trimmed to fit `prompt.max_tokens` if needed. Token-aware trimming keeps the recent set within prompt.max$_tokensbeforesending$

Environment variables can override specific keys (`MILLA_DB_PATH`, `MILLA_NODE_ID`, `MILLA_NODE_LOCATION`, `OLLAMA_URL`, `OLLAMA_MODEL`, `OLLAMA_KEEP_ALIVE`, `CHAT_HISTORY_LIMIT`, `MAX_PROMPT_TOKENS`, `MILLA_DEFAULT_SE` or the path via `MILLA_CONFIG`). Logging payloads can be disabled via `:log { :request_bodies? false }`, and the log is capped to 5MB.

- **Portability**: runs on Linux and should be portable to macOS/BSD, assuming Java, Clojure CLI, SQLite, and Ollama are available.
- **Simplicity**: favor clear, small scripts and Clojure namespaces that are easy to inspect and debug.
- **Transparency**: the SQLite DB shall be easy to inspect with `sqlite3` for debugging/backups.
- **Durability**: DB sync shall never silently destroy data; conflicts shall be logged and require manual resolution.
- **Safety of Self-Modification**: any self-modifying behavior shall be human-in-the-loop and test-gated, with proposals captured via backlog items rather than applied silently.