Reflective Memory Kernel: Deep Dive Architecture

1. System Overview

The Reflective Memory Kernel (RMK) is a distributed system designed to provide long-term, evolvable memory for AI agents. It decouples “Computation” (LLM Inference) from “Memory” (Knowledge Graph), allowing the agent to maintain a persistent identity and knowledge base across sessions.

1.1 Core Components

- **Front-End Agent (Go/Gin)**: The user-facing gateway. It manages WebSocket/REST connections, auth, and fast conversational loops. It acts as the “Consciousness,” holding short-term context.
- **Memory Kernel (Go)**: The “Hippocampus.” It handles ingestion, entity extraction, graph operations, and retrieval (consultation).
- **Reflection Engine (Go)**: The “Default Mode Network.” It runs asynchronously to consolidate memory, find patterns, and generate insights.
- **Knowledge Graph (DGraph)**: The persistent storage layer. A distributed graph database storing all facts, entities, and relationships.
- **Message Bus (NATS JetStream)**: The nervous system. Handles high-throughput, persistent messaging between the Agent, Kernel, and Reflection Engine.
- **AI Service (External)**: An abstract inference layer (likely Python/FastAPI) that interfaces with LLMs (e.g., GPT-4, Claude 3.5 Sonnet) to generate text and extract entities.

---

2. Minute-Level Data Flow: Group Chat V2

The Group Chat feature introduces “Strict Data Isolation” into this architecture. Here is the step-by-step lifecycle of a message.

2.1 The Request (User to Agent)

1. **Input**: User sends `POST /api/chat` with body `{"message": "Hello team", "context_type": "group", "context_id": "group_123"}`.
2. **Auth**: `JWT Middleware` validates the `Bearer` token.
3. **Context Resolution** (`handleChat`):
   - The Agent inspects `context_type`.
   - **CRITICAL**: It determines the `namespace`.
     - If `group`: `namespace = "group_123"`
     - If `user`: `namespace = "user_<UserID>"`
4. **State Lookup**: The Agent retrieves the active `Conversation` struct from memory or creates one.

2.2 The Consultation (Retrieval)

Before answering, the Agent asks the Kernel: “What do we know about this?”

1. **RPC Call**: Agent calls `MKClient.Consult()`.
2. **Payload**: `ConsultationRequest` struct is sent:

   ```
   {
       UserID: "user_A",
       Namespace: "group_123", // TARGETS SHARED MEMORY
       Query: "Hello team"
   }
   ```

3. **Kernel Execution**:
   - **Retrieval**: `ConsultationHandler` executes a DGraph query using the `namespace` filter: `func: type(Node) @filter(eq(namespace, "group_123") AND ...)`
   - **Isolation**: This query is physically incapable of seeing nodes with `namespace="user_A"`.
4. **Synthesis**: Kernel summarizes found facts into a `SynthesizedBrief`.
5. **Response**: Kernel returns `ConsultationResponse` with the brief.

2.3 The Processing (LLM Inference)

1. **Construction**: Agent constructs a prompt for the AI Service:
   - “You are a helpful assistant...”
   - “Context: [SynthesizedBrief from Group Memory]”
   - “User: Hello team”
2. **Inference**: `AIClient` sends this to the external AI Service.
3. **Generation**: The LLM generates a response (“Hello! How can I help the team?”).

2.4 The Ingestion (Memory Formation)

Now the conversation must be stored.

1. **Event Creation**: Agent creates a `TranscriptEvent`:

   ```
   {
       Namespace: "group_123", // PERSISTS TO SHARED MEMORY
       UserQuery: "Hello team",
       AIResponse: "Hello!..."
   }
   ```

2. **Async Publish**: Agent publishes this event to NATS topic `transcripts.user_A` (or a group-specific topic).
3. **Kernel Ingestion**:
   - `IngestionPipeline` receives the event.
   - **Extraction**: It calls the AI Service to extract entities (`Type: Group`, `Name: Team`).
   - **Graph Write**: It writes these nodes to DGraph, explicitly setting `namespace = "group_123"` on every new Node and Edge.

3. Data Models (The Schema)

The DGraph schema is the source of truth.

3.1 The User & Group Nodes

- **User (`type User`)**: Represents an identity.
- **Group (`type Group`)**:
  - `uid` : Unique ID.
  - `name` : "Engineering Team".
  - `created_by` : Edge to a `User` node (Admin).
  - `group_members` : List of Edges to `User` nodes.
  - **NO MEMORY**: Validates membership only. Memory is stored in *other* nodes with the group's namespace.

3.2 The Knowledge Nodes (`type Node`)

Everything else (Facts, Entities, Events) is a `Node` .

- **`namespace` (string, indexed)**: The security boundary. "user_123" or "group_456".
- **`activation` (float)**: Priority score (0.0-1.0) for retrieval. Decays over time (Reflection Engine).
- **`edges`**: Relationships (`likes` , `knows` , `works_on`). All edges are directed.

4. Frontend Architecture (The App Shell)

The frontend is a lightweight Single Page Application (SPA).

- **Global State**: `currentContext` (`{type: 'group', id: '...'}`).
- **Navigation**:
  - Clicking "My Memory" sets `currentContext = {type: 'private'}` .
  - Clicking "Group X" sets `currentContext = {type: 'group', id: 'X'}` .
- **API Layer**: `sendMessage` reads `currentContext` and injects it into the JSON payload of every request.

5. Security Model (Subuser & Admin)

- **Authentication**: JWT (Stateless). User identity is confirmed per request.
- **Admin Check**:
  - `DELETE /groups/{id}` checks if `RequesterID == Group.created_by`.
- **Subuser Creation**:
  - Admin sends `POST /subusers`.
  - System creates a new `User` node (detached from any real email/auth provider for now).
  - System links `Group -> group_members -> NewUser`.
  - NewUser allows an AI agent (or team member) to access the `group_123` namespace.

---

6. Technical Implementation Specifications

6.1 API Endpoints (Agent Layer)

Chat Interaction

- `POST /api/chat`
  - **Payload**: `{"message": string, "context_type": "user"|"group", "context_id": string}`
  - **Handler**: `handleChat` (in `server.go`)
  - **Logic**: Resolves `namespace` -> Calls `MKClient.Consult` -> Calls `AIClient.Generate` -> Streams to NATS.

Group Management

- `GET /api/groups`
  - **Returns**: List of groups where `RequesterID` is in `group_members`.
- `POST /api/groups`
  - **Payload**: `{"name": string}`
  - **Logic**: Creates `Group` node, links `created_by` to Requester, adds Requester to `group_members`.
- `DELETE /api/groups/{id}`
  - **Auth**: Requires `RequesterID == Group.created_by`.
  - **Effect**: Deletes Group node. *Note: Does not currently cascade delete memory nodes for safety.*

Member Management

- `POST /api/groups/{id}/subusers`
  - **Payload**: `{"username": string, "password": string}`
  - **Logic**: Registers new User -> Adds to Group.
- `DELETE /api/groups/{id}/members/{username}`
  - **Logic**: Removes edge `Group -> group_members -> TargetUser`.

6.2 Key Code Modifications

`internal/agent/server.go`

- **Updated `ChatRequest` struct**: Added `ContextType` and `ContextID`.
- **Updated `handleChat`**: Logic to switch `namespace` variable based on context.

`internal/kernel/ingestion.go`

- **Updated `IngestionPipeline.Process`**: Now accepts `namespace` argument.
- **Updated `processBatchedEntities`**: Uses `namespace` to construct the DGraph `namespace` predicate for every new node.

  ```
  node.Namespace = event.Namespace // e.g. "group_123"
  ```

`internal/kernel/consultation.go`

- **Updated `ConsultationHandler.Handle`**: Extracts `Namespace` from request.
- **Updated `getUserKnowledge`**:
  - **Old**: `func: has(name)` (implied user scope)
  - **New**: `func: eq(namespace, $namespace)` (Explicit scope)

6.3 DGraph Schema Changes (`internal/graph/schema.go`)

- **Added**: `namespace: string @index(exact) .`
- **Updated**: `TranscriptEvent` and `ConsultationRequest` structs in Go now map to this schema field.