Reflective Memory Kernel (RMK) - Complete Product Feature Documentation

Product Overview

Reflective Memory Kernel (RMK) is an enterprise AI agent platform that transforms reactive RAG (Retrieval-Augmented Generation) into proactive Agent-Augmented Generation (AAG). It provides AI agents with persistent, evolving memory capabilities using a biological-inspired knowledge graph architecture.

Table of Contents

---

1. Executive Summary

Core Value Proposition

Capability	Business Value
Persistent Memory	AI agents that remember, learn, and evolve over time
Proactive Intelligence	Moves from question-answer to anticipatory assistance
Biological Memory Dynamics	Implements activation decay, reinforcement, and associative learning
Enterprise-Ready	Multi-tenant architecture with strict namespace isolation

Key Differentiators

1. **Hybrid Storage Architecture**: Knowledge Graph (DGraph) + Vector Search (Qdrant) = 100% recall
2. **Three-Phase Memory**: Ingestion → Reflection → Consultation
3. **Pre-Cortex Cognitive Firewall**: Reduces LLM costs by up to 90%
4. **Vector-Native Document Processing**: Mathematical compression vs expensive LLM processing
5. **Multi-LLM Routing**: OpenAI, Anthropic, Ollama, NVIDIA NIM, GLM support

Problem Solved

Traditional RAG systems are **reactive** - they wait for questions and retrieve documents. RMK is **proactive** - it learns from every interaction, synthesizes insights, and anticipates user needs.

Traditional RAG	RMK (AAG)
Question → Retrieve Document	Context → Proactive Brief
One-shot queries	Persistent memory across sessions
Static knowledge	Evolving understanding
No learning	Biological memory dynamics

2. Core Architecture

Dual-Agent Cognitive Model

```
┌─────────────────────────────────────────────────────────────────────┐
│                              USER LAYER                             │
│  ┌───────────────────────────────────────────────────────────────┐  │
│  │                       Chat UI (Port 8080)                     │  │
│  └───────────────────────────────────────────────────────────────┘  │
└─────────────────────────────────────────────────────────────────────┘
                                  │ HTTP/WebSocket
                                  ▼
┌─────────────────────────────────────────────────────────────────────┐
│                   FRONT-END AGENT ("The Consciousness")             │
│  Port 3000 (standalone) or 8080 (monolith)                          │
│                                                                     │
│  • WebSocket gateway for real-time chat                             │
│  • Session management & JWT authentication                          │
│  • Pre-Cortex cognitive firewall                                    │
│  • Consults Memory Kernel for context                               │
└─────────────────────────────────────────────────────────────────────┘
                                  │
                                  │
                   ┌──────────────┴──────────────┐
                   │                             │
                   ▼                             ▼
┌──────────────────────────────────┐  ┌─────────────────────────────────┐
│      MEMORY KERNEL               │  │         PRE-CORTEX              │
│    ("The Subconscious")          │  │      Cognitive Firewall         │
│  Port 9000 (standalone)          │  │                                 │
│                                  │  │  1. Semantic Cache (90%+ hit)   │
│  • Knowledge graph management    │  │  2. Intent Classifier           │
│  • Memory algorithms             │  │   3. DGraph Reflex Engine       │
│  • Activation decay/boost        │  │   → Reduces LLM calls by 90%    │
└──────────────────────────────────┘  └─────────────────────────────────┘
                   │
                   ▼
┌─────────────────────────────────────────────────────────────────────┐
│                        STORAGE & INFRASTRUCTURE                     │
│                                                                     │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐ │
│  │   DGraph    │  │    NATS     │  │   Redis     │  │   Qdrant    │ │
│  │  (Graph)    │  │  (Stream)   │  │  (Cache)    │  │  (Vectors)  │ │
│  │   Port 9080 │  │  Port 4222  │  │  Port 6379  │  │  Port 6333  │ │
│  └─────────────┘  └─────────────┘  └─────────────┘  └─────────────┘ │
└─────────────────────────────────────────────────────────────────────┘
                   │
                   ▼
┌─────────────────────────────────────────────────────────────────────┐
│                   AI SERVICES ("The Cerebral Cortex")               │
│  Port 8000                                                          │
│                                                                     │
│  • LLM orchestration (OpenAI, Anthropic, Ollama, NIM, GLM)          │
│  • Entity extraction (SLMs)                                         │
│  • Document ingestion (tiered extraction)                           │
│  • Embedding generation (768-dim vectors)                           │
└─────────────────────────────────────────────────────────────────────┘
```

Component Reference

Component	Port	Purpose	Technology
Front-End Agent	3000/8080	WebSocket gateway, session management, real-time chat	Go + Gorilla WS
Memory Kernel	9000	Knowledge graph management, memory algorithms	Go + gRPC
AI Services	8000	LLM orchestration, entity extraction	Python + FastAPI
DGraph Alpha	9080	Knowledge graph storage	DGraph
DGraph Zero	5080	Cluster management	DGraph
NATS	4222	Event streaming	NATS JetStream
Redis	6379	Hot path cache	Redis
Qdrant	6333	Vector embeddings	Qdrant
Ollama	11434	Local LLM	Ollama

Storage Infrastructure

System	Purpose	Data Type
DGraph	Knowledge Graph storage	Entities, relationships, insights, patterns
Qdrant	Vector embeddings	Semantic similarity search (768-dim vectors)
Redis	Hot path cache	Recent messages (50 msg ring buffer), activation scores
NATS JetStream	Event streaming	Async transcript processing

3. Feature Set

3.1 Memory-Augmented Conversations

Description

Every conversation is remembered, indexed, and made available for future context. Unlike traditional chatbots that start fresh each session, RMK maintains a persistent memory of all interactions.

Capabilities:

- Persistent memory across sessions
- Context-aware responses using retrieved history
- Namespace-based isolation (private vs shared workspaces)
- Real-time streaming via WebSocket

How It Works:

```
User Message → Front-End Agent → Memory Kernel Consultation
                                            |
                                            ▼
                                    [Relevant Facts]
                                    [Insights]
                                    [Patterns]
                                    [Proactive Alerts]
                                            |
                                            ▼
                    AI Services + Context → Enhanced Response
                                            |
                                            ▼
                                    NATS (Async Transcript)
                                            |
                                            ▼
                                    Memory Kernel (Ingestion)
```

API Endpoints:

Endpoint	Method	Purpose
`/api/chat`	POST	Send message with memory context
`/api/conversations`	GET	Retrieve conversation history
`/ws/chat`	WS	Real-time WebSocket chat

Request Example:

```
POST /api/chat
{
  "user_id": "user_abc123",
  "conversation_id": "conv_xyz789",
  "message": "My partner Alex loves Thai food"
}
```

Response Example:

```
{
  "conversation_id": "conv_xyz789",
  "response": "I've noted that Alex loves Thai food. I'll remember this for future conversations.",
  "latency_ms": 234,
  "context_used": {
    "facts_retrieved": 3,
    "insights_applied": 1
  }
}
```

3.2 Three-Phase Memory Architecture

The Memory Kernel operates on a continuous three-phase loop that transforms raw conversations into actionable intelligence.

Phase 1: Ingestion (Real-time)

Receives transcripts via NATS JetStream and processes them immediately.

```
NATS Transcript → Entity Extraction → DGraph Write
                       │
                       ▼
              [Entities Extracted]
              [Relations Identified]
              [Facts Stored]
```

Process:

1. Receives transcript from Front-End Agent
2. Sends to AI Services for entity extraction
3. Writes structured knowledge to DGraph
4. Updates activation scores

Key Characteristics:

- **Latency**: < 100ms for most transcripts
- **Throughput**: Handles 100+ concurrent sessions
- **Reliability**: NATS JetStream ensures no data loss

Phase 2: Reflection (Async Rumination)

The background “digital rumination” process that transforms raw facts into insights.

Four Sub-Phases:

Sub-Phase	Function	Example Output
Active Synthesis	Discovers emergent insights from disconnected facts	“Thai food + peanut allergy = warning”
Predictive Anticipation	Learns behavioral patterns from temporal data	“Monday mornings = Project Alpha review”
Self-Curation	Resolves contradictions in stored knowledge	“Manager Bob (Jan) → Manager Alice (Jun)”
Dynamic Prioritization	Applies activation boost/decay to memories	Frequently accessed facts stay top

Reflection Pipeline:

```
┌─────────────────────────────────────────────────────────────────┐
│                    REFLECTION ENGINE                            │
│  Triggered every 5 minutes or on 100 new facts                  │
└─────────────────────────────────────────────────────────────────┘
                          │
        ┌─────────────────┼─────────────────┐
        ▼                 ▼                 ▼
┌──────────────┐  ┌──────────────┐  ┌──────────────┐
│  Synthesis   │  │  Curation    │  │ Prioritization│
│              │  │              │  │              │
│  • Cross     │  │  • Detect    │  │  • Decay     │
│    reference │  │    conflicts │  │    old       │
│  • Generate  │  │  • Resolve   │  │  • Boost     │
│    insights  │  │    winner    │  │    accessed  │
└──────────────┘  └──────────────┘  └──────────────┘
        │                 │                 │
        └─────────────────┼─────────────────┘
                          ▼
                  DGraph Updates
```

API Endpoint:

- `POST /api/reflect` - Manually trigger reflection cycle (for testing)

Phase 3: Consultation

Retrieves and synthesizes pre-computed insights for context-aware responses.

Consultation Flow:

```
Query → Memory Kernel
            │
            ▼
    [Relevant Facts by Activation]
            │
            ▼
    [Pre-computed Insights]
            │
            ▼
    [Detected Patterns]
            │
            ▼
    [Proactive Alerts]
            │
            ▼
    Synthesized Brief
```

Request Example:

```
POST /api/consult
{
  "user_id": "user_abc123",
  "query": "What should we have for dinner?",
  "max_results": 10,
  "include_insights": true,
  "topic_filters": ["food", "preference"]
}
```

Response Example:

```
{
  "request_id": "req_xyz789",
  "synthesized_brief": "Based on your history, Alex loves Thai food but you have a peanut allergy. You might consider suggesting Thai cu
  "relevant_facts": [
    {
      "uid": "0x1",
      "name": "Alex",
      "description": "User's partner",
      "activation": 0.85
    },
    {
      "uid": "0x2",
      "name": "Thai Food",
      "description": "Cuisine preference",
      "activation": 0.72
    }
  ],
  "insights": [
    {
      "insight_type": "warning",
      "summary": "Thai food commonly contains peanuts",
      "action_suggestion": "Mention allergy when Thai food is discussed"
    }
  ],
  "patterns": [
    {
      "pattern_type": "preference",
      "confidence": 0.91
    }
  ],
  "proactive_alerts": [
    "If Thai food is mentioned, remind about peanut allergy"
  ],
  "confidence": 0.87
}
```

3.3 Biological Memory Dynamics

RMK implements biological-inspired memory dynamics that mimic how human memory works.

Activation Decay

Memories fade over time when not accessed, preventing information overload.

Algorithm:

```
daysSinceAccess := (now - lastAccessed) / 24 hours
if daysSinceAccess > 1 {
    decayFactor = (1 - decayRate) ^ daysSinceAccess
    newActivation = max(activation * decayFactor, minActivation)
}
```

Configuration:

Parameter	Default	Description
`DECAY_RATE`	0.005	0.5% decay per day
`MIN_ACTIVATION`	0.01	1% minimum (pruning candidate)
`MAX_ACTIVATION`	1.0	100% maximum

Example:

- A fact not accessed for 30 days: `1.0 * (0.995)^30 ≈ 0.86` activation
- A fact not accessed for 365 days: `1.0 * (0.995)^365 ≈ 0.16` activation

Reinforcement (Boost)

Accessed memories receive immediate activation boost, creating a “heat map” of important topics.

Algorithm:

```
On Access:
    newActivation = min(activation + boostPerAccess, maxActivation)
    accessCount++
    lastAccessed = now
    mentionedEntities.forEach(secondaryBoost)
```

Boost Types:

Type	Amount	Trigger
Primary Boost	+0.1	Direct access to node
Secondary Boost	+0.05	Mentioned in conversation
Co-activation	+0.03	Related node accessed

Dynamic Reordering

Retrieval queries automatically sort by activation score, surfacing the most relevant information first.

```
# DGraph query with activation ordering
{
  memories(func: ge(activation, 0.3), orderdesc: activation, first: 10) {
    uid
    name
    description
    activation
    last_accessed
  }
}
```

Result: High-activation facts surface first, core identity traits remain accessible.

3.4 Knowledge Graph System

RMK uses DGraph as its knowledge graph backbone, providing a flexible schema for storing entities, relationships, and insights.

Node Types

Type	Description	Example
User	User profiles with auth and preferences	Central node for all user knowledge
Entity	People, organizations, locations, concepts	“Alex”, “Project Alpha”, “Thai Food”
Event	Temporal occurrences with timestamps	“Project Alpha review on Monday”
Insight	Synthesized insights from reflection	“Thai food may contain peanuts”
Pattern	Detected behavioral patterns	“Requests brief before meetings”
Preference	User preferences and attributes	“Prefers email over calls”
Fact	Verified information	“User is vegan”
Rule	Logical rules and constraints	“Prepare brief on Monday mornings”
Group	Shared workspaces for collaboration	“Engineering Team”
Conversation	Chat session history	“Conversation about dinner plans”

Relationship Types (Edges)

Personal Relationships:

```
PARTNER_IS      - Romantic partner (max 1 current)
FAMILY_MEMBER   - Family relationships
FRIEND_OF       - Friendship connections
```

Professional Relationships:

```
HAS_MANAGER     - Manager relationship (max 1 current)
WORKS_ON        - Project involvement
WORKS_AT        - Employment (max 1 current)
COLLEAGUE       - Work colleague connections
```

Preferences:

```
LIKES           - Positive preferences
DISLIKES        - Negative preferences
IS_ALLERGIC_TO  - Allergy information
PREFERS         - General preferences
HAS_INTEREST    - Interest areas
```

Causal & Logical:

```
CAUSED_BY       - Causal relationship
BLOCKED_BY      - Blocking dependency
RESULTS_IN      - Consequence relationship
CONTRADICTS     - Contradiction marker
```

Meta Relationships:

```
DERIVED_FROM        - Source tracking
SYNTHESIZED_FROM    - Insight sources
SUPERSEDES          - Replacement relationship
MEMBER_OF_COMMUNITY - GraphRAG clustering
```

Edge Facets (Metadata)

Every edge can carry metadata:

Facet	Type	Range	Description
`activation`	float	0.0-1.0	Memory accessibility
`confidence`	float	0.0-1.0	Truthfulness score
`created_at`	datetime	-	When the edge was created
`updated_at`	datetime	-	When the edge was last modified
`status`	string	current/archived	Current state

Self-Curation Example

```
January: User says "My manager is Bob"
  → Creates: (User) -[HAS_MANAGER {status: current}]-> (Bob)

June: User says "My new manager is Alice"
  → Detects functional constraint violation
  → Archives: (User) -[HAS_MANAGER {status: archived}]-> (Bob)
  → Creates: (User) -[HAS_MANAGER {status: current}]-> (Alice)
  → Creates: (Alice) -[SUPERSEDES]-> (Bob)
```

3.5 Pre-Cortex Cognitive Firewall

A “cognitive firewall” that intercepts requests before reaching external LLMs, reducing costs by up to 90%.

Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                        PRE-CORTEX LAYER                         │
│                                                                 │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │ Step 1: SEMANTIC CACHE                                    │  │
│  │ • Exact match lookup (fastest path)                       │  │
│  │ • Vector similarity search (0.92+ threshold)              │  │
│  │ → HIT? Return cached response immediately                 │  │
│  └───────────────────────────────────────────────────────────┘  │
│                            │ MISS                               │
│                            ▼                                    │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │ Step 2: INTENT CLASSIFICATION                             │  │
│  │ • Greeting → Handle with deterministic response           │  │
│  │ • Navigation → Return UI action                           │  │
│  │ • Fact Retrieval → Try DGraph Reflex                      │  │
│  │ • Complex → Pass to LLM                                   │  │
│  └───────────────────────────────────────────────────────────┘  │
│                            │ COMPLEX                            │
│                            ▼                                    │
│  ┌───────────────────────────────────────────────────────────┐  │
│  │ Step 3: DGRAPH REFLEX ENGINE                              │  │
│  │ • Pattern-based fact extraction                           │  │
│  │ • Direct DGraph queries                                   │  │
│  │ • Template-based response generation                      │  │
│  └───────────────────────────────────────────────────────────┘  │
│                            │ NO MATCH                           │
│                            ▼                                    │
└─────────────────────────────────────────────────────────────────┘
                               │ Only ~10% reach LLM
                               ▼
                        EXTERNAL LLM
```

Three-Layer Defense

Layer 1: Semantic Cache

Type	Method	Speed
Exact Match	Normalized string lookup	< 1ms
Vector Search	Cosine similarity (0.92+ threshold)	< 50ms

Layer 2: Intent Classification

Intent	Pattern	Handler
Greeting	“Hi”, “Hello”, “Good morning”	Deterministic response
Navigation	“Go to settings”, “Open dashboard”	UI action
Fact Retrieval	“What is my email?”, “List my groups”	DGraph Reflex
Complex	Everything else	LLM

Layer 3: DGraph Reflex Engine

Direct graph queries for simple fact retrieval:

Query Pattern	Example	Response
`user_email`	“What is my email?”	Returns email from User node
`user_name`	“What is my name?”	Returns name from User node
`user_groups`	“List my groups”	Returns groups user is member of
`user_preferences`	“What do I like?”	Returns stored preferences

Cost Impact

Metric	Without Pre-Cortex	With Pre-Cortex
LLM Calls	100% of requests	~10% of requests
Average Latency	2-5 seconds	<100ms cached
Cost per 1K requests	~$10-50	~$1-5

Configuration

```
# Pre-Cortex Settings
ENABLE_SEMANTIC_CACHE=true
ENABLE_INTENT_ROUTER=true
ENABLE_DGRAPH_REFLEX=true
CACHE_SIMILARITY=0.92    # Minimum similarity for cache hit (0.0-1.0)
CACHE_TTL=600           # Cache TTL in seconds (default: 10 minutes)
```

3.6 Hybrid Memory System (Cognee + GraphRAG)

RMK implements a dual-track memory system combining atomic fact storage with holistic insight generation.

Track Comparison

Feature	Track A: Cognee	Track B: GraphRAG
Input Data	SQL Dumps, JSON, APIs	PDFs, Logs, Long-form Text
Goal	Exact Fact Reconstruction	Thematic Understanding
Key Unit	Entity (Node)	Insight (Cluster Summary)
Technique	1:1 Mapping, Relation Extraction	Community Detection (Leiden)
Query Type	“What is X’s email?”	“What are common complaints?”

Architecture

```
┌─────────────────────────────────────────────────────────────────┐
│                    HYBRID MEMORY ARCHITECTURE                    │
│                                                                  │
│                         ┌──────────────┐                         │
│                         │  Data Router │                         │
│                         └──────┬───────┘                         │
│                    ┌───────────┴───────────┐                     │
│                    ▼                       ▼                     │
│          ┌─────────────────┐     ┌─────────────────┐             │
│          │   TRACK A       │     │   TRACK B       │             │
│          │   (Cognee)      │     │   (GraphRAG)    │             │
│          │                 │     │                 │             │
│          │  📊 Structured  │     │  📄 Unstructured │             │
│          │  SQL, JSON, APIs│     │  PDFs, Logs, Docs│            │
│          │                 │     │                 │             │
│          │  🎯 Atomic Facts │     │  🔮 Holistic     │             │
│          │  Exact Recall   │     │  Pattern Insight│             │
│          └────────┬────────┘     └────────┬────────┘             │
│                   │                       │                      │
│                   └───────────┬───────────┘                      │
│                               ▼                                  │
│                     ┌─────────────────┐                          │
│                     │     DGraph      │                          │
│                     │ Knowledge Graph │                          │
│                     └─────────────────┘                          │
└─────────────────────────────────────────────────────────────────┘
```

GraphRAG Pipeline

Step 1: Entity Extraction

```
# Extract entities and relations from unstructured text
chunks = chunk_text(request.content, size=512)
entities = []
relations = []

for chunk in chunks:
    result = await llm.extract_entities_and_relations(chunk)
    entities.extend(result.entities)
    relations.extend(result.relations)
```

Step 2: Community Detection (Leiden)

```
# Run Leiden algorithm for community detection
partition = leidenalg.find_partition(
    g,
    leidenalg.ModularityVertexPartition,
    resolution_parameter=1.0
)
```

Step 3: Community Summarization

```
// Generate Insight nodes from community members
func (k *Kernel) SummarizeCommunity(ctx context.Context, communityID int) error {
    nodes, _ := k.graphClient.GetCommunityMembers(ctx, communityID)
    context := buildContext(nodes)
    summary, _ := k.aiClient.SummarizeCommunity(context)

    insight := &graph.Insight{
        Summary:     summary,
        CommunityID: communityID,
        Level:       0,
        Confidence:  0.85,
    }

    return k.graphClient.CreateInsightWithLinks(ctx, insight, nodes)
}
```

Configuration

```
# Hybrid Architecture Settings
HYBRID_MODE=true
COGNEE_ENABLED=true
GRAPHRAG_ENABLED=true

# Clustering Settings
LEIDEN_RESOLUTION=1.0
CLUSTER_INTERVAL=10m          # Re-cluster every 10 minutes
CLUSTER_MIN_NODES=100         # Minimum nodes before clustering

# Community Summarization
SUMMARIZE_TOP_N=50            # Max nodes per community summary
INSIGHT_MIN_CONFIDENCE=0.7
```

3.7 Document Ingestion & Processing

RMK supports tiered document processing optimized for cost and accuracy.

Tiered Extraction

Tier	Method	Cost	Use Case
1	Rule-based (regex, spaCy NER)	FREE	Basic extraction
2	Smart chunking + clustering	CHEAP	Document organization
3	LLM extraction	EXPENSIVE	Complex entities
4	Vision LLM	EXPENSIVE	Charts/diagrams

Vector-Native Architecture

Instead of expensive LLM chunking, RMK uses hierarchical vector trees for semantic compression.

```
Document → Chunks (512 tokens) → Embeddings → K-Means Clustering
                                             │
                                             ▼
                                    Parent Nodes (Centroids)
                                             │
                                             ▼
                                    Recursive Clustering
                                             │
                                             ▼
                                Hierarchical Vector Tree
```

Benefits:

- 10-100x cheaper than LLM summarization
- Preserves semantic relationships
- Enables efficient similarity search
- Mathematically reproducible

API Endpoints

Endpoint	Method	Purpose
`/api/upload`	POST	Upload documents
`/ingest-document`	POST	Tiered ingestion
`/ingest-vector-tree`	POST	Vector-native ingestion
`/extract-vision`	POST	Vision-based extraction

Request Example:

```
POST /api/upload
{
  "namespace": "user_abc123",
  "files": ["document.pdf", "image.png"],
  "extraction_tier": "llm"
}
```

3.8 Multi-LLM Routing

RMK supports multiple LLM providers with intelligent dispatch based on availability and task complexity.

Supported Providers

Provider	Models	Priority	Cost
OpenAI	GPT-4, GPT-3.5-turbo	1 (if API key set)	$$$
Anthropic	Claude 3 Haiku/Opus	2 (if API key set)	$$$
Ollama	Llama 3, Mistral (local)	3 (always available)	FREE
NVIDIA NIM	High-performance inference	Optional	$$
GLM (Zhipu AI)	Cost-effective option	Optional	$

Intelligent Dispatch

```python
def select_llm(task_complexity):
    # Priority 1: OpenAI (if key set)
    if os.getenv("OPENAI_API_KEY") and task_complexity == "high":
        return OpenAIProvider()

    # Priority 2: Anthropic (if key set)
    if os.getenv("ANTHROPIC_API_KEY") and task_complexity == "medium":
        return AnthropicProvider()

    # Priority 3: Always available Ollama
    return OllamaProvider()
```

Configuration

```
# LLM Providers
OPENAI_API_KEY=sk-...
ANTHROPIC_API_KEY=sk-...
OLLAMA_HOST=http://ollama:11434
NIM_API_KEY=...
GLM_API_KEY=...

# Routing
LLM_PREFERENCE=openai,anthropic,ollama
FALLBACK_TO_LOCAL=true
```

3.9 Workspace Collaboration (Google Docs-like)

RMK supports shared memory spaces for team collaboration.

Features

- **Shared Memory Spaces**: Teams collaborate on AI knowledge
- **Invitation System**: Invite existing users by username
- **Shareable Links**: Generate tokens with expiration/usage limits
- **Role-Based Access**: Admin vs Subuser permissions
- **Namespace Isolation**: Strict data separation

Role Permissions

Action	Admin	Subuser	Non-Member
Read/Write memories	✅	✅	❌
Invite users	✅	❌	❌
Create share links	✅	❌	❌
Remove members	✅	❌	❌
Delete workspace	✅	❌	❌

API Endpoints

Endpoint	Method	Purpose
`/api/workspaces/{id}/invite`	POST	Invite user
`/api/workspaces/{id}/share-link`	POST	Create share link
`/api/join/{token}`	POST	Join via share link
`/api/invitations`	GET	List pending invites
`/api/invitations/{id}/accept`	POST	Accept invitation
`/api/workspaces/{id}/members`	GET	List members

Share Link Security

```go
// Generate secure token (256 bits)
func GenerateToken() string {
    bytes := make([]byte, 32) // 256 bits
    rand.Read(bytes)
    return base64.URLEncoding.EncodeToString(bytes)
}
```

Validation:

- Token exists
- Link is active (not revoked)
- Link not expired
- Usage limit not reached
- User not already a member

3.10 Search & Retrieval

RMK provides comprehensive search capabilities across multiple data types.

Search Types

Type	Method	Use Case
Semantic Search	Vector similarity over Qdrant	"Similar to…" queries
Graph Traversal	DGraph relationship queries	"Connected to…" queries
Hybrid Search	Combined semantic + graph	Best results (100% recall)
Temporal Query	Time-based searches	"Last 7 days" queries

Consultation API

Comprehensive context retrieval with:

- Relevant facts by activation
- Pre-computed insights
- Detected patterns
- Proactive alerts

Request:

```
POST /api/consult
{
  "user_id": "user_abc123",
  "query": "Tell me about my projects",
  "max_results": 10,
  "include_insights": true,
  "topic_filters": ["project", "work"]
}
```

Response:

```
{
  "synthesized_brief": "You're working on Project Alpha with a deadline approaching. You also have Project Beta in early planning.",
  "relevant_facts": [...],
  "insights": [...],
  "patterns": [...],
  "proactive_alerts": ["Project Alpha deadline: Friday"],
  "confidence": 0.87
}
```

Search Endpoints

Endpoint	Method	Purpose
`/api/search`	POST	General search
`/api/search/temporal`	POST	Time-based search
`/api/semantic-search`	POST	Vector similarity search

3.11 Security & Multi-Tenancy

Namespace Isolation

RMK enforces strict namespace separation for multi-tenancy.

Namespace Types:

Type	Format	Description
User Namespace	`user_<uuid>`	Private memory space
Group Namespace	`group_<uuid>`	Shared workspace

Query Enforcement:

```
# All queries enforce namespace filter
query GetFacts($namespace: string) {
    facts(func: type(Fact)) @filter(eq(namespace, $namespace)) {
        name
        description
        # ...
    }
}
```

Guarantees:

- User A's workspace cannot see memories from User B's workspace
- Cross-tenant protection enforced at database level
- No data leakage between namespaces

Authentication

- JWT-based authentication
- CSRF protection
- Secure share link tokens (256-bit entropy)

```
// Middleware validates JWT on every request
func (m *JWTMiddleware) Middleware(next http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        authHeader := r.Header.Get("Authorization")
        if authHeader == "" || !strings.HasPrefix(authHeader, "Bearer ") {
            http.Error(w, "Unauthorized", http.StatusUnauthorized)
            return
        }
        // ... validate token ...
    })
}
```

Authorization

- Admin vs Subuser roles
- Workspace-level permissions
- Audit logging

3.12 Admin & Management

RMK includes comprehensive admin capabilities for managing users and the system.

User Management

- Create/delete users
- Manage subscriptions
- View user statistics

Admin APIs

Endpoint	Method	Purpose
`/api/admin/users`	GET	List users
`/api/admin/users`	POST	Create user
`/api/admin/users/{id}`	DELETE	Delete user
`/api/admin/affiliates`	GET	Affiliate system
`/api/admin/finance`	GET	Finance management
`/api/admin/support`	GET	Support tickets

Dashboard Analytics

Memory Statistics:

- Entity count
- Fact count
- Insight count
- Pattern count

Activation Metrics:

- High-activation nodes
- Recent insights
- Active patterns

Ingestion Metrics:

- Documents processed
- Entities extracted
- Processing time

Dashboard Endpoints:

Endpoint	Method	Purpose
`/api/dashboard/stats`	GET	Overview metrics
`/api/dashboard/graph`	GET	Visual graph representation
`/api/dashboard/ingestion`	GET	Ingestion stats

3.13 SDK & Developer Tools

RMK provides SDKs for multiple programming languages.

Available SDKs

SDK	Language	Location
Go SDK	Go	`sdks/go/`
Python SDK	Python	`sdks/python/`
TypeScript SDK	TypeScript	`sdks/ts/`

MCP (Model Context Protocol) Server

RMK includes an MCP server for integrating with AI agent tools.

```
# Start MCP server
go run ./cmd/mcp
```

Location: `cmd/mcp/main.go`

Deployment Options

Option	Description	Use Case
Docker Compose	Full stack deployment	Local development
Monolith	Single container	Simplified deployment
Kubernetes	K8s manifests	Production scaling

---

4. Configuration

Environment Variables

```
# ============================================
# LLM Providers
# ============================================
OPENAI_API_KEY=sk-...
ANTHROPIC_API_KEY=sk-...
OLLAMA_HOST=http://ollama:11434
NIM_API_KEY=...
GLM_API_KEY=...

# ============================================
# Infrastructure
# ============================================
DGRAPH_URL=dgraph-alpha:9080
NATS_URL=nats://nats:4222
REDIS_URL=redis:6379
QDRANT_URL=http://qdrant:6333
AI_SERVICES_URL=http://ai-services:8000

# ============================================
# Memory Settings
# ============================================
DECAY_RATE=0.005              # 0.5% per day
ACTIVATION_BOOST=1.0
MIN_ACTIVATION=0.01
MAX_ACTIVATION=1.0

# ============================================
# Pre-Cortex
# ============================================
ENABLE_SEMANTIC_CACHE=true
CACHE_SIMILARITY=0.92
CACHE_TTL=600

# ============================================
# GraphRAG
# ============================================
LEIDEN_RESOLUTION=1.0
CLUSTER_INTERVAL=10m
CLUSTER_MIN_NODES=100

# ============================================
# Security
# ============================================
JWT_SECRET=your-secret-key-min-32-chars
```

Configuration File

RMK also supports configuration via YAML file:

```yaml
# config/rmk.yaml
server:
  port: 8080
  frontend_only: false

memory:
  decay_rate: 0.005
  activation_boost: 1.0
  min_activation: 0.01
  max_activation: 1.0

precortex:
  enable_semantic_cache: true
  enable_intent_router: true
  enable_dgraph_reflex: true
  cache_similarity: 0.92
  cache_ttl: 600

graphrag:
  enabled: true
  leiden_resolution: 1.0
  cluster_interval: 10m
  cluster_min_nodes: 100

llm:
  providers:
    - openai
    - anthropic
    - ollama
  fallback_to_local: true
```

5. API Reference Summary

Front-End Agent (Port 3000/8080)

Endpoint	Method	Purpose
`/api/chat`	POST	Send message
`/api/search`	POST	Search memory
`/api/search/temporal`	POST	Time-based search
`/api/conversations`	GET	Chat history
`/api/upload`	POST	Upload documents
`/api/workspaces/{id}/invite`	POST	Invite user
`/api/workspaces/{id}/share-link`	POST	Create share link
`/api/join/{token}`	POST	Join via link
`/api/invitations`	GET	List invites
`/api/dashboard/*`	GET	Analytics
`/ws/chat`	WS	Real-time chat

Memory Kernel (Port 9000)

Endpoint	Method	Purpose
`/api/consult`	POST	Query memory
`/api/stats`	GET	Memory stats
`/api/reflect`	POST	Trigger reflection
`/api/ensure-user`	POST	Create user node

AI Services (Port 8000)

Endpoint	Method	Purpose
`/extract`	POST	Entity extraction
`/curate`	POST	Resolve contradictions
`/synthesize`	POST	Create brief
`/generate`	POST	Generate response
`/embed`	POST	Generate embedding
`/semantic-search`	POST	Vector search
`/ingest-document`	POST	Document ingestion
`/extract-vision`	POST	Vision extraction

6. Data Flow Diagrams

Conversation Flow

```
User → FEA → Pre-Cortex (cache check)
           ↓ (miss)
       Memory Kernel (consultation)
           ↓
       AI Services (generate response)
           ↓
       User + NATS (async transcript)
```

Memory Flow

```
NATS (transcript) → MK Ingestion
                         ↓
                   Entity Extraction
                         ↓
                   DGraph (write nodes/edges)
```

Reflection Flow

```
Timer (5 min) → Reflection Engine
                      ↓
        ┌─────────────┼─────────────┐
        ▼             ▼             ▼
    Synthesis     Curation      Prioritization
        │             │             │
        └─────────────┼─────────────┘
                      ↓
                DGraph Updates
```

Document Ingestion Flow

```
Document Upload → Tier Selection
                      │
        ┌─────────────┼─────────────┐
        ▼             ▼             ▼
    Rule-based   Chunking      LLM/Vision
    (FREE)       (CHEAP)       (EXPENSIVE)
        │             │             │
        └─────────────┼─────────────┘
                      ↓
                Entity Extraction
                      ↓
                DGraph + Qdrant
```

---

7. Technology Stack Summary

Layer	Technology	Purpose
Frontend	HTML/JS, React, Vite, Tailwind CSS	Chat interface
API Gateway	Go + Gorilla WebSocket	Low-latency conversation
Memory Engine	Go + gRPC	Persistent background agent
AI Orchestration	Python + FastAPI	LLM orchestration
Graph Database	DGraph	Knowledge Graph storage
Vector Database	Qdrant	Semantic similarity search
Cache	Redis	Hot path caching
Message Queue	NATS JetStream	Async transcript streaming
Local LLM	Ollama	Local LLM/embedding service
Containerization	Docker + Docker Compose	Service isolation

---

8. Deployment

Quick Start

```
# Clone repository
git clone https://github.com/your-org/rmk.git
cd rmk

# Start all services
docker-compose up -d

# Check status
docker-compose ps

# View logs
docker-compose logs -f monolith
```

Access Points

Service	URL	Credentials
Chat UI	http://localhost:8080	Create account
Memory Kernel API	http://localhost:9000	N/A
AI Services API	http://localhost:8000	N/A
DGraph UI	http://localhost:8080	N/A
NATS Monitoring	http://localhost:8222	N/A

Health Checks

```
# Check all services
curl http://localhost:8080/health
curl http://localhost:9000/health
curl http://localhost:8000/health
```

Production Deployment

```
# Use production configuration
docker-compose -f docker-compose.prod.yml up -d

# Or deploy to Kubernetes
kubectl apply -f k8s/
```

9. Verification

Feature Checklist

- ✅ Persistent memory across sessions
- ✅ Three-phase memory architecture (Ingestion, Reflection, Consultation)
- ✅ Biological memory dynamics (activation decay/boost)
- ✅ Knowledge graph with 10+ node types
- ✅ Pre-Cortex cognitive firewall
- ✅ Hybrid memory system (Cognee + GraphRAG)
- ✅ Tiered document ingestion
- ✅ Multi-LLM routing (5+ providers)
- ✅ Workspace collaboration
- ✅ Semantic + graph search
- ✅ Multi-tenant namespace isolation
- ✅ Admin dashboard
- ✅ SDKs (Go, Python, TypeScript)
- ✅ MCP server integration
- ✅ Docker deployment

API Verification

```
# Test chat endpoint
curl -X POST http://localhost:8080/api/chat \
  -H "Content-Type: application/json" \
  -d '{"user_id": "test", "message": "Hello"}'

# Test consultation
curl -X POST http://localhost:9000/api/consult \
  -H "Content-Type: application/json" \
  -d '{"user_id": "test", "query": "What do you know?"}'

# Test reflection
curl -X POST http://localhost:9000/api/reflect
```

Performance Benchmarks

Metric	Target	Actual
Chat latency (cached)	< 100ms	~50ms
Chat latency (LLM)	< 5s	~2s
Memory ingestion	< 100ms	~50ms
Reflection cycle	< 30s	~10s
Semantic search	< 100ms	~50ms

Appendix

Related Documentation

- Architecture Overview - System architecture details
- API Reference - Complete API documentation
- AI Services - LLM orchestration details
- Pre-Cortex - Cognitive firewall deep dive
- Hybrid Memory - Cognee + GraphRAG details
- Knowledge Graph - DGraph schema details
- Workspace Collaboration - Collaboration features

Glossary

Term	Definition
AAG	Agent-Augmented Generation - proactive AI assistance
RAG	Retrieval-Augmented Generation - reactive document retrieval
SLM	Small Language Model - specialized AI for specific tasks
Cognee	Track A memory system for atomic fact storage
GraphRAG	Track B memory system for holistic insights
Pre-Cortex	Cognitive firewall layer before LLM
Activation	0.0-1.0 score indicating memory relevance
Namespace	Isolation boundary for multi-tenancy

Last Updated: January 2026