Modular Memory Management System (M3S) v3.0

Four-Tier Reference-Based Memory Architecture for Noor Cognitive Digital Twin

Version: 3.0
Date: November 29, 2025
Document Type: Complete Architecture Specification
Audience: Senior Engineers, Architects, Decision Makers
Classification: Technical Architecture Document

Table of Contents

Executive Summary

Architectural Evolution: v2.1 → v3.0

The Modular Memory Management System (M3S) v3.0 represents a fundamental architectural refinement of the Noor Cognitive Digital Twin memory subsystem. This evolution addresses critical limitations in the original v2.1 design through innovative approaches to conversation memory, context compression, and sovereignty-compliant storage.

v2.1 Architecture (Original)

- Manual save_memory protocol for personal tier
- Lossy summarization for context compression
- Sequential tier search with fallback logic
- Complex 4-step memory save protocol
- Token overhead: 750-1,200 per query
- Information loss during LLM summarization

v3.0 Architecture (Refined)

- Conversations auto-chunked as personal memory
- Reference-based lossless compression
- Semantic domain routing (Dept ≠ Global)
- Batch processing with local models
- Token overhead: 500-800 per query
- Perfect content preservation with selective expansion

Key Innovations

1. Conversations as Memory Foundation

Eliminates the artificial distinction between "conversations" and "personal memories" by treating stored conversations as the source of truth for personal memory. Conversations are automatically chunked, embedded, and indexed nightly using sovereignty-safe local models.

2. Reference-Based Lossless Compression

Revolutionary approach replacing traditional summarization with pointer-based compression. When context window fills, segments are extracted to a local knowledge graph and replaced with compact reference cards. LLM can selectively expand segments on-demand without information loss.

3. Browser-Side Sovereignty Compliance

Local IndexedDB storage for extracted segments ensures no sensitive data leaves the sovereign boundary. Government employees' on-premise deployment constraint becomes an architectural advantage, enabling rich local graph storage.

Performance Improvements

Token Efficiency:	33-47% reduction (500-800 vs 750-1,200 tokens)
Context Window Utilization:	80% effective usage vs 60% in v2.1
Information Retention:	100% lossless vs ~70-85% with summarization

Compression Trigger:	Intelligent 80% threshold vs hard limits
Selective Retrieval Latency:	<100ms from local graph vs N/A

Part 1: Memory System Requirements

1.1 Four-Tier Memory Architecture

The M3S architecture organizes memory into four independent tiers, each with distinct purposes, storage locations, and access controls:

Tier	Storage Location	Purpose	Noor Access	Maestro Access	Content Examples
Personal	Neo4j (:ConversationMemory nodes) + Browser IndexedDB (extracted segments)	User-specific conversation history, preferences, corrections	Read/Write (auto)	Read/Write	"I prefer concise answers", name corrections, format preferences
Departmental	Neo4j (:DepartmentalKnowledge nodes)	Team functional knowledge, validated expertise	Read-Only	Read/Write (Curates)	Team procedures, how-to guides, project learnings, expertise
Global	Neo4j (:GlobalKnowledge nodes)	Organizational events, policies, strategic plans	Read-Only	Read/Write (Curates)	Organization policies, budget cycles, quarterly reviews, announcements
C-suite	Neo4j (:CSuiteKnowledge nodes, restricted)	Executive-only sensitive information	NO Access	Read/Write (Exclusive)	Personnel changes, budget forecasts, litigation, executive deliberations

⚡ Key Distinction: Independent Memory Tiers

Each memory tier operates independently with distinct semantic domains:

- **Personal:** Individual user conversation patterns and preferences (per-user scope)
- **Departmental:** Functional "how-to" knowledge and team expertise (per-department scope)
- **Global:** Organizational cycles, events, and policies (organization-wide scope)
- **C-suite:** Restricted executive information (executive-only scope)

Search Strategy: Noor uses intelligent semantic routing to determine which tier(s) to query based on the user's question intent, not sequential fallback logic.

Storage Infrastructure

Where Each Memory Tier is Hosted

Tier	Primary Storage	Secondary Storage	Purpose
Personal	Neo4j (:ConversationMemory nodes)	Browser IndexedDB (extracted segments)	Long-term searchable history + compressed segments
Departmental	Neo4j (:DepartmentalKnowledge nodes)	N/A	Team functional knowledge repository
Global	Neo4j (:GlobalKnowledge nodes)	N/A	Organizational event and policy repository
C-suite	Neo4j (:CSuiteKnowledge nodes, restricted)	N/A	Executive-only information repository

Note: All Neo4j nodes include vector embeddings for semantic search. Browser IndexedDB is used only for Personal tier to enable reference-based compression with selective expansion.

1.2 Revised Memory Model (v3.0 Simplifications)

Personal Tier: Automatic Conversation Processing

Key Innovation: Personal memory is derived automatically from conversation history through nightly batch processing. No manual save_memory calls required.

- **Source:** Existing conversation database
- **Processing:** Semantic chunking with local LLM
- **Embedding:** Local BGE model (sovereignty-safe)
- **Storage:** Neo4j ConversationMemory nodes
- **Temporal Dimension:** Implicit in conversation timestamps

Departmental & Global Tiers: Curated Knowledge Only

These tiers contain only curated, governance-controlled knowledge requiring explicit approval chains:

- **Departmental:** Confidence ≥0.85, Department head approval
- **Global:** Confidence ≥0.90, C-suite approval
- **Write Access:** Maestro agent only
- **Content Lifecycle:** Version control with SUPERSEDES/REPLACES relationships

1.3 Core Operations

Memory Retrieval (Read MCP)

- **search_conversation_memory:** Personal tier semantic search
- **search_departmental_knowledge:** Functional knowledge retrieval
- **search_global_knowledge:** Organizational events and policies
- **intelligent_search:** Smart routing with local LLM classification

Memory Curation (Write MCP - Maestro Only)

- **create_departmental_knowledge:** Add curated functional knowledge
- **create_global_knowledge:** Add organizational policies/events
- **update_knowledge:** Version control with audit trails

Batch Processing Pipeline

- **Nightly Processing:** Conversation → Chunk → Embed → Index
- **Local Models:** Mistral-7B for chunking, BGE for embeddings
- **Sovereignty Safe:** No external API calls

Part 2: Reference-Based Compression Architecture

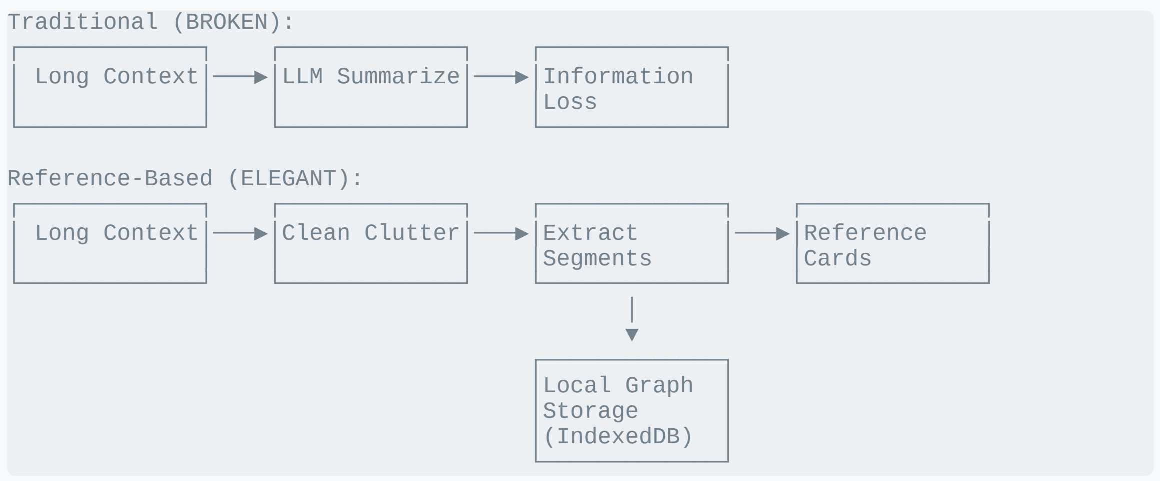
2.1 Core Innovation: Pointers Instead of Summaries

Problem with Traditional Summarization

Critical Issue: LLMs consistently lose information during summarization, especially for technical content, code snippets, and precise specifications. Users cannot verify what was compressed, leading to degraded context quality over time.

Solution: Reference-Based Lossless Compression

Innovative Approach: Extract segments to local knowledge graph and replace with compact reference cards. LLM can selectively expand segments on-demand without any information loss.



2.2 Content Classification & Deduplication

Segment Types and Compression Priority

Type	Indicators	Priority	Strategy
code	```, def, class, function, import	High	Extract immediately - preserve all technical details
llm_output	Lists, headers, structured responses >100 words	High	Extract - typically repetitive or reformattable
design	architecture, component, system, flow	High	Extract - technical specifications need preservation
docs	documentation, spec, API, requirements >100 words	High	Extract - reference material

Type	Indicators	Priority	Strategy
explanation	because, this means, in other words	Medium	Extract if >50 words and similar content exists
short_exchange	<20 words	Low	Keep in context - too small to extract

ContentClassifier Implementation

````
class ContentClassifier:
    """
    Classify conversation segments by type and detect redundancy.
    """

    SEGMENT_TYPES = {
        'code': {
            'indicators': ['```', 'def ', 'class ', 'function', 'import'],
            'min_lines': 3,
            'compression_priority': 'high'
        },
        'llm_output': {
            'indicators': ['generated', 'response', 'result'],
            'patterns': [r'\d+\.\s', r'\*\*.*\*\*', r'###'],
            'compression_priority': 'high'
        },
        'short_exchange': {
            'max_words': 20,
            'compression_priority': 'low'
        }
    }

    def classify_turn(self, turn: ConversationTurn) -> SegmentClassification:
        """Classify a conversation turn by content type."""
        content = turn.content
        word_count = len(content.split())

        # Check for code
        if '```' in content or any(ind in content for ind in self.SEGMENT_TYPES[
            return SegmentClassification(
                type='code',
                priority='high',
                extractable=True,
                reason='Contains code block'
            )

        # Short exchange - keep in context
        if word_count < 20:
            return SegmentClassification(
                type='short_exchange',
                priority='low',
                extractable=False,
                reason='Too short to extract'
            )

        return SegmentClassification(
            type='explanation',
            priority='medium',
            extractable=word_count > 50
        )

    def detect_redundancy(self, current_turn, context_history) -> RedundancyScor
        """Detect if current turn is repetitive using hash + semantic similarity
        # Hash-based exact match
        current_hash = hashlib.md5(current_turn.content.encode()).hexdigest()
        for past_turn in context_history:
            past_hash = hashlib.md5(past_turn.content.encode()).hexdigest()
            if current_hash == past_hash:
                return RedundancyScore(is_redundant=True, score=1.0)

        # Semantic similarity for near-duplicates
        similarity = cosine_similarity(
````

```
            self.embedder.encode(current_turn.content),
            self.embedder.encode(past_turn.content)
        )

        if similarity > 0.95:
            return RedundancyScore(is_redundant=True, score=similarity)

        return RedundancyScore(is_redundant=False, score=0.0)
```

2.3 Local Knowledge Graph Extraction

Browser-Side IndexedDB Storage

Sovereignty Advantage: Government deployment constraint (on-premise machines) becomes architectural benefit. Rich local graph storage without cloud sync concerns.

ExtractedSegment Schema

```
# Segment node structure in IndexedDB
segment_node = {
    'id': 'seg_session123_turn456',
    'type': 'code' | 'design' | 'llm_output' | 'docs' | 'explanation',
    'content': 'Full original content (NOT summarized)',
    'keywords': ['function_name', 'UserModel', 'calculate_score'],  # NOT summar
    'entities': ['EntityProject_FALCON', 'EntityCapability_DataAnalysis'],
    'timestamp': '2025-11-29T14:30:00Z',
    'turn_id': 'turn_456',
    'session_id': 'session_123',
    'word_count': 150,
    'compression_priority': 'high'
}
```

Reference Card Generation

```
def _create_reference_card(self, segment_node, keywords):
    """
    Create compact reference card (pointer) to segment.
    This goes in active context instead of full content.
    """
    keyword_str = ", ".join(keywords[:5])

    if segment_node['type'] == 'code':
        card = f"📦 [Code: {keyword_str}] (segment_id: {segment_node['id']}, {se
    elif segment_node['type'] == 'design':
        card = f"🏗️ [Design: {keyword_str}] (segment_id: {segment_node['id']}, {
    elif segment_node['type'] == 'llm_output':
        card = f"🤖 [Output: {keyword_str}] (segment_id: {segment_node['id']}, {

    return card
```

```
# Example reference card in context:
# [REFERENCE] 📦 [Code: calculate_score, UserModel, pandas] (segment_id: seg_ses
```

2.4 Context Compression Pipeline

```
Context Compression Trigger (80% threshold):
┌─────────────────┐
│ Active Context  │   ───────┐  8K tokens (80% = 6.4K)
│ 6.4K+ tokens    │          │
└─────────────────┘          ▼
                   ┌────────────────┐
                   │ Classification │  │ → High/Medium/Low priority
                   │ & Deduplication│  │
                   └────────────────┘
                            │
                            ▼
                   ┌────────────────┐
                   │ Priority-Based │  → Extract High first
                   │ Extraction     │    Then Medium if needed
                   └────────────────┘
                            │
              ┌─────────────┼─────────────┐
              ▼             ▼             ▼
   ┌──────────────┐ ┌────────────┐ ┌────────────┐
   │ IndexedDB    │ │ Reference  │ │ Active     │
   │ Storage      │ │ Cards      │ │ Context    │
   │(Full Content)│ │ (Pointers) │ │(Remaining) │
   └──────────────┘ └────────────┘ └────────────┘
```

ReferenceBasedCompressionManager Implementation

```
class ReferenceBasedCompressionManager:
    """
    Manages context window by extracting to local graph and keeping references.
    """

    def __init__(self, max_active_tokens: int = 8000):
        self.max_active_tokens = max_active_tokens
        self.classifier = ContentClassifier()
        self.extractor = LocalGraphExtractor()
        self.active_context = []

    async def add_turn_with_compression(self, turn, session_id):
        """Add turn to context, triggering compression if needed."""

        # Check for redundancy
        redundancy = self.classifier.detect_redundancy(turn, self.active_context
        if redundancy.is_redundant:
            logger.info(f"Skipping redundant turn: {redundancy.reason}")
            return

        self.active_context.append(turn)

        # Trigger compression at 80% threshold
        current_tokens = self._estimate_total_tokens()
        if current_tokens > self.max_active_tokens * 0.8:
            await self._compress_context(session_id)

    async def _compress_context(self, session_id):
        """Compression pipeline: Clean clutter, extract segments, keep reference

        # Step 1: Classify all turns
        classified = []
        for turn in self.active_context:
            classification = self.classifier.classify_turn(turn)
            classified.append((turn, classification))

        # Step 2: Sort by compression priority
        high_priority = [(t, c) for t, c in classified if c.priority == 'high' a

        # Step 3: Extract high-priority segments first
        for turn, classification in high_priority:
            extracted = await self.extractor.extract_segment(turn, classificatio

            # Replace turn with reference card
            self._replace_with_reference(turn, extracted.reference_card)

            # Check if we've freed enough space
            if self._estimate_total_tokens() < self.max_active_tokens * 0.6:
                break
```

2.5 Selective Segment Retrieval

On-Demand Expansion Logic

```
class SelectiveRetrieval:
    """Retrieve full segments from local graph when LLM needs them."""

    async def retrieve_segment(self, segment_id: str) -> Optional[str]:
        """Fetch full segment content from local graph."""
        segment = await self.graph_store.get('segments', segment_id)

        if not segment:
            logger.warning(f"Segment {segment_id} not found in local graph")
            return None

        # Return full content (NOT summary)
        return segment['content']

    def _detect_segment_references(self, query: str, context: str) -> List[str]:
        """
        Detect if user is referencing a compressed segment.
        Example: "Show me that code you mentioned" → extract segment_id
        """
        reference_phrases = [
            'that code', 'the code', 'previous code',
            'that design', 'the design',
            'that output', 'the explanation'
        ]

        query_lower = query.lower()
        if any(phrase in query_lower for phrase in reference_phrases):
            # Extract segment IDs from context reference cards
            segment_ids = re.findall(r'segment_id:\s*([a-z0-9_]+)', context)
            return segment_ids

        return []

    def _inject_segment(self, context: str, segment_id: str, full_content: str)
        """Replace reference card with full content in context."""
        pattern = rf'\[REFERENCE\].*segment_id:\s*{segment_id}.*'
        replacement = f"[EXPANDED SEGMENT {segment_id}]\n{full_content}\n[END SE
        return re.sub(pattern, replacement, context)
```

Part 3: Separated MCP Architecture

3.1 Read MCP Server (Noor + Maestro Access)

```
MCP Router Architecture:
┌───────────────────────────────────────────────────────┐
│                 API GATEWAY / MCP ROUTER               │
│     • Authentication (JWT, API Keys)                   │
│     • Agent identification (noor / maestro)            │
│     • Request routing to services                      │
└───────────────────────────────────────────────────────┘
              │                    │
              ▼                    ▼
    ┌──────────────────┐  ┌──────────────────────┐
    │   READ MCP       │  │   WRITE MCP          │
    │   (Noor access)  │  │   (Maestro only)     │
    │                  │  │                      │
    │ • Personal search│  │ • Dept knowledge     │
    │ • Dept search    │  │ • Global knowledge   │
    │ • Global search  │  │ • Version control    │
    │ • Smart routing  │  │ • Governance         │
    └──────────────────┘  └──────────────────────┘
```

MemoryReadMCPServer Implementation

```python
class MemoryReadMCPServer:
    """Read-only memory retrieval. Used by both Noor and Maestro."""

    @mcp_tool
    def search_conversation_memory(
        self,
        query: str,
        user_id: str,
        time_window_days: Optional[int] = None,
        limit: int = 5
    ) -> List[ConversationMemory]:
        """
        Search user's conversation history (personal tier).
        Example: "What did I ask about Project FALCON last week?"
        """
        query_embedding = self.embedder.encode(query)

        filters = {'user_id': user_id}
        if time_window_days:
            cutoff = datetime.now() - timedelta(days=time_window_days)
            filters['timestamp_after'] = cutoff

        return self.storage.vector_search(
            embedding=query_embedding,
            node_type='ConversationMemory',
            filters=filters,
            limit=limit
        )

    @mcp_tool
    def search_departmental_knowledge(
        self,
        query: str,
        department: str,
        knowledge_type: Optional[str] = None,
        limit: int = 5
    ) -> List[DepartmentalKnowledge]:
        """
        Search department-specific functional knowledge.
        Types: how_to, expertise, project_learning, best_practice
        Example: "How do I submit a project proposal in Finance?"
        """
        query_embedding = self.embedder.encode(query)

        filters = {'department': department}
        if knowledge_type:
            filters['knowledge_type'] = knowledge_type

        return self.storage.vector_search(
            embedding=query_embedding,
            node_type='DepartmentalKnowledge',
            filters=filters,
            limit=limit
        )

    @mcp_tool
    def search_global_knowledge(
        self,
        query: str,
        knowledge_type: Optional[str] = None,
        year: Optional[int] = None,
        limit: int = 5
    ) -> List[GlobalKnowledge]:
        """
        Search organizational events, cycles, policies.
        Types: event, milestone, announcement, policy, cycle
        Example: "When is the Q4 budget submission deadline?"
        """
        query_embedding = self.embedder.encode(query)

        filters = {}
        if knowledge_type:
            filters['knowledge_type'] = knowledge_type
        if year:
            filters['year'] = year

        # Filter by effective date (only return current/future knowledge)
        filters['effective_date_before'] = datetime.now()
        filters['expiry_date_after'] = datetime.now()

        return self.storage.vector_search(
            embedding=query_embedding,
            node_type='GlobalKnowledge',
            filters=filters,
            limit=limit
        )

    @mcp_tool
    def intelligent_search(
        self,
        query: str,
        user_context: dict,
        limit_per_tier: int = 3
    ) -> IntelligentSearchResult:
        """
        Smart routing: Determines which tier(s) to search based on query.
        Uses local LLM to classify query intent.
        """
        # Classify query intent using local LLM
        intent = self._classify_query_intent(query)

        results = IntelligentSearchResult(query=query)

        # Route to appropriate tier(s)
        if intent.is_personal_history:
            results.personal = self.search_conversation_memory(
                query=query,
                user_id=user_context['user_id'],
                limit=limit_per_tier
            )

        if intent.is_departmental_knowledge:
            results.departmental = self.search_departmental_knowledge(
                query=query,
                department=user_context['department'],
                knowledge_type=intent.dept_category,
                limit=limit_per_tier
            )

        if intent.is_global_knowledge:
            results.global_items = self.search_global_knowledge(
                query=query,
                knowledge_type=intent.global_category,
                year=intent.year,
                limit=limit_per_tier
            )
        return results
```

3.2 Write MCP Server (Maestro Only)

Security Model

Critical: Write MCP server is accessible ONLY by Maestro agent. All write operations require governance approval and confidence thresholds.

```
class MemoryWriteMCPServer:
    """Write operations for curated collective knowledge. ONLY Maestro access.""

    @mcp_tool(require_agent='maestro')
    def create_departmental_knowledge(
        self,
        department: str,
        title: str,
        content: str,
        knowledge_type: Literal['how_to', 'expertise', 'project_learning', 'best
        tags: List[str],
        confidence: float,
        approved_by: str
    ) -> str:
        """
        Maestro creates curated departmental knowledge.

        Governance:
        - Confidence must be >= 0.85
        - Requires approval from department head
        """
        # Validate confidence threshold
        if confidence < 0.85:
            raise ValidationError("Departmental knowledge requires confidence >=

        # Validate approver authorization
        if not self._is_department_head(approved_by, department):
            raise AuthorizationError(f"{approved_by} cannot approve for {departm

        # Generate embedding
        embedding = self.embedder.encode(content)

        knowledge_id = f"dept_{department}_{uuid.uuid4().hex[:8]}"

        node = DepartmentalKnowledge(
            id=knowledge_id,
            department=department,
            title=title,
            content=content,
            knowledge_type=knowledge_type,
            embedding=embedding,
            tags=tags,
            created_by='maestro',
            approved_by=approved_by,
            confidence=confidence,
            created_at=datetime.now(),
            year=datetime.now().year
        )

        self.storage.create_node(node)

        # Audit log
        self._log_knowledge_creation(
            tier='departmental',
            knowledge_id=knowledge_id,
            created_by='maestro',
            approved_by=approved_by
        )

        return knowledge_id

    @mcp_tool(require_agent='maestro')
    def create_global_knowledge(
        self,
        title: str,
        content: str,
        knowledge_type: Literal['event', 'milestone', 'announcement', 'policy',
        confidence: float,
        approved_by: str,
        effective_date: date,
        expiry_date: Optional[date] = None
    ) -> str:
        """
        Maestro creates curated global organizational knowledge.

        Governance:
        - Confidence must be >= 0.90 (higher than dept)
        - Requires C-suite approval
        """
        if confidence < 0.90:
            raise ValidationError("Global knowledge requires confidence >= 0.90"

        if not self._is_csuite(approved_by):
            raise AuthorizationError(f"{approved_by} is not C-suite")

        # Create and store node with version control
        knowledge_id = f"global_{uuid.uuid4().hex[:8]}"

        return knowledge_id
```

3.3 Semantic Domain Routing

Key Architectural Decision

Departmental and Global are distinct semantic domains. Local LLM classifies query intent to route to appropriate tier(s) based on question content.

Domain	Content Focus	Query Examples	Knowledge Types
Departmental	Functional knowledge, team expertise, how-to guides	"How do I submit a project proposal?", "Who knows about data analysis?"	how_to, expertise, project_learning, best_practice
Global	Organizational events, policies, cycles, announcements	"When is Q4 budget submission?", "What's the policy on remote work?"	event, milestone, announcement, policy, cycle

```
def _classify_query_intent(self, query: str) -> QueryIntent:
    """Local LLM classification of query to route to correct tier(s)."""
    prompt = f"""
    Classify this query into categories:

    Query: "{query}"

    Determine:
    1. is_personal_history: Does user want to recall past conversations? (yes/no
    2. is_departmental_knowledge: Does user need functional/how-to knowledge? (y
       - If yes, category: how_to | expertise | project_learning | best_practice
    3. is_global_knowledge: Does user need org events/policies? (yes/no)
       - If yes, category: event | milestone | announcement | policy | cycle
    4. year: If temporal, extract year (null if not applicable)

    Return JSON.
    """

    result = self.local_llm.generate(prompt, format='json')

    return QueryIntent(
        is_personal_history=result['is_personal_history'],
        is_departmental_knowledge=result['is_departmental_knowledge'],
        dept_category=result.get('dept_category'),
        is_global_knowledge=result['is_global_knowledge'],
        global_category=result.get('global_category'),
        year=result.get('year')
    )
```

Part 4: Neo4j Schema Design

4.1 Personal Tier: ConversationMemory Nodes

```
// PERSONAL TIER: Auto-generated from conversations
CREATE (m:ConversationMemory {
  // Identity
  id: "conv_123_chunk_1",          // conv_{conv_id}_chunk_{idx}
  user_id: "user_456",               // Owner
  department: "finance",

  // Content
  content: "User asked about Project FALCON budget analysis approach...",
  conversation_id: "conv_123",       // Source conversation
  turn_range: [15, 18],                // [start_turn, end_turn]

  // Semantic
  embedding: [0.1, -0.3, 0.8, ...],   // 1536-dim vector
  semantic_type: "question",         // question|answer|correction|preference
  intent_summary: "User seeks budget analysis methodology for Project FALCON",
  entities_mentioned: ["EntityProject_FALCON", "EntityCapability_BudgetAnalysis"

  // Temporal (implicit)
  timestamp: datetime("2025-11-29T14:30:00Z"),
  year: 2025,

  // Metadata
  processed_at: datetime("2025-11-30T02:00:00Z"), // Nightly batch
  chunk_index: 1
})

// Vector index for semantic search
CREATE VECTOR INDEX conversation_memory_embedding
FOR (m:ConversationMemory) ON (m.embedding)
```

```
OPTIONS {
  indexConfig: {
    `vector.dimensions`: 1536,
    `vector.similarity_function`: 'cosine'
  }
}

// Composite key index
CREATE INDEX conversation_memory_composite
FOR (m:ConversationMemory) ON (m.id, m.year)

// Temporal filtering
CREATE INDEX conversation_memory_temporal
FOR (m:ConversationMemory) ON (m.timestamp, m.user_id)

// Keyset pagination
CREATE INDEX conversation_memory_keyset
FOR (m:ConversationMemory) ON (m.id)
```

4.2 Departmental Tier: DepartmentalKnowledge Nodes

```
// DEPARTMENTAL TIER: Curated functional knowledge
CREATE (m:DepartmentalKnowledge {
  id: "dept_finance_a1b2c3d4",
  department: "finance",
  scope: "departmental",

  // Content
  title: "How to Submit Project Proposals in Finance Department",
  content: "Step-by-step guide for project proposal submission...",
  knowledge_type: "how_to",  // how_to|expertise|project_learning|best_practice

  // Semantic
  embedding: [0.2, -0.1, 0.5, ...],
  tags: ["project-management", "finance", "proposals"],

  // Governance
  created_by: "maestro",
  approved_by: "finance_head_789",
  confidence: 0.92,  // Must be >= 0.85

  // Temporal
  created_at: datetime("2025-11-15T10:00:00Z"),
  last_updated: datetime("2025-11-15T10:00:00Z"),
  year: 2025
})

// Indexes
CREATE VECTOR INDEX dept_knowledge_embedding
FOR (d:DepartmentalKnowledge) ON (d.embedding)

CREATE INDEX dept_knowledge_department
FOR (d:DepartmentalKnowledge) ON (d.department, d.knowledge_type)
```

4.3 Global Tier: GlobalKnowledge Nodes

```
// GLOBAL TIER: Organizational events and cycles
CREATE (m:GlobalKnowledge {
  id: "global_e5f6g7h8",
  scope: "global",

  // Content
  title: "Q4 2025 Budget Submission Deadline",
  content: "All departments must submit Q4 budget proposals by December 15, 2025
  knowledge_type: "milestone",  // event|milestone|announcement|policy|cycle

  // Semantic
  embedding: [-0.1, 0.4, 0.2, ...],
  tags: ["budget", "deadline", "Q4", "2025"],

  // Governance
  created_by: "maestro",
  approved_by: "cfo_executive_456",
  confidence: 0.95,  // Must be >= 0.90

  // Temporal validity
  effective_date: date("2025-11-01"),
  expiry_date: date("2025-12-31"),
  year: 2025,
  created_at: datetime("2025-10-30T09:00:00Z")
})

// Indexes
CREATE VECTOR INDEX global_knowledge_embedding
FOR (g:GlobalKnowledge) ON (g.embedding)

CREATE INDEX global_knowledge_validity
FOR (g:GlobalKnowledge) ON (g.effective_date, g.expiry_date)
```

4.4 Relationships

```
// Link conversation memories to entities
(:ConversationMemory)-[:MENTIONS]->(:EntityProject)
(:ConversationMemory)-[:MENTIONS]->(:EntityCapability)
(:ConversationMemory)-[:REFERENCES]->(:EntityObjective)

// Version control for curated knowledge
(:DepartmentalKnowledge)-[:SUPERSEDES]->(:DepartmentalKnowledge)
(:GlobalKnowledge)-[:REPLACES]->(:GlobalKnowledge)

// Cross-tier knowledge relationships
(:DepartmentalKnowledge)-[:RELATES_TO]->(:GlobalKnowledge)
(:ConversationMemory)-[:CLARIFIES]->(:DepartmentalKnowledge)
```

Part 5: Batch Processing Pipeline

5.1 Sovereignty-Safe Nightly Processing

Sovereignty Compliance

All conversation processing uses **local models only** (Mistral-7B for chunking, BGE-large for embeddings). No external API calls. Data never leaves sovereign boundary.

```
Nightly Batch Processing Pipeline:
┌─────────────────────────────────────────────────────────┐
│  23:00 UTC - Batch Job Triggers                         │
└─────────────────────────────────────────────────────────┘
                              │
                              ▼
        ┌──────────────────────────────┐
        │ Fetch Unprocessed            │
        │ Conversations (last 24hrs)   │
        └──────────────────────────────┘
                      │
                      ▼
        ┌──────────────────────────────┐
        │ For Each Conversation:       │
        │ 1. Semantic Chunking         │  ←— Local Mistral-7B
        │ 2. Metadata Extraction       │  ←— Local Mistral-7B
        │ 3. Embedding Generation      │  ←— Local BGE-large
        │ 4. Neo4j Indexing            │
        └──────────────────────────────┘
                      │
                      ▼
        ┌──────────────────────────────┐
        │ Create ConversationMemory    │
        │ Nodes + Relationships        │
        └──────────────────────────────┘
                      │
                      ▼
        ┌──────────────────────────────┐
        │ Mark Conversations as        │
        │ Processed in DB              │
        └──────────────────────────────┘
```

5.2 Semantic Chunking Strategy

```python
class ConversationMemoryProcessor:
    """
    Batch processor: Converts stored conversations into searchable memory.
    """

    def __init__(self, local_llm, embedder):
        self.llm = local_llm  # Mistral-7B-Instruct
        self.embedder = embedder  # BGE-large-en-v1.5
        self.chunk_strategy = SemanticChunker()

    async def _chunk_conversation(self, conv: Conversation) -> List[Chunk]:
        """
        Semantic chunking strategy:
        - Keep Q&A pairs together
        - Break on topic shifts
        - Preserve context window (max 500 tokens/chunk)
        """
        chunks = []
        current_chunk = []
        current_tokens = 0

        for turn in conv.turns:
            turn_tokens = self._estimate_tokens(turn.content)

            # Detect topic shift using local LLM
            if current_chunk and self._is_topic_shift(current_chunk[-1], turn):
                chunks.append(Chunk(
                    text=self._merge_turns(current_chunk),
                    start_turn=current_chunk[0].idx,
                    end_turn=current_chunk[-1].idx,
                    timestamp=current_chunk[0].timestamp
                ))
                current_chunk = []
                current_tokens = 0

            current_chunk.append(turn)
            current_tokens += turn_tokens

            # Hard limit: 500 tokens per chunk
            if current_tokens > 500:
                chunks.append(Chunk(
                    text=self._merge_turns(current_chunk),
                    start_turn=current_chunk[0].idx,
                    end_turn=current_chunk[-1].idx,
                    timestamp=current_chunk[0].timestamp
                ))
                current_chunk = []
                current_tokens = 0

        return chunks
```

5.3 Metadata Extraction with Local LLM

```
async def _extract_metadata(self, chunk: Chunk) -> ChunkMetadata:
    """Use local LLM to extract semantic metadata."""
    prompt = f"""
    Analyze this conversation chunk and extract:
    1. Type: question | answer | correction | preference | context
    2. Entities mentioned (projects, capabilities, people)
    3. Intent summary (1 sentence)

    Chunk:
    {chunk.text}

    Return JSON.
    """

    result = await self.llm.generate(prompt, format='json')

    return ChunkMetadata(
        type=result['type'],
        entities=result['entities'],
        intent_summary=result['intent']
    )
```

Part 6: Short-Term Memory Enhancement for Noor

6.1 Working Memory During Session

Noor's Short-Term Memory Challenge

Noor needs effective "working memory" within a conversation session to maintain context across multiple turns. This is separate from long-term storage and focuses on active conversation management.

Sliding Context Window Manager

```python
class NoorShortTermMemory:
    """Manages Noor's working memory within a conversation session."""

    def __init__(self, max_tokens: int = 8000):
        self.max_tokens = max_tokens
        self.compression_manager = ReferenceBasedCompressionManager()
        self.current_session = None

    def initialize_session(self, session_id, user_context):
        """Start new conversation session with context budget."""
        self.current_session = ShortTermMemorySession(
            session_id=session_id,
            token_budget=self.max_tokens,
            turns=[]
        )

        # Pre-load relevant long-term memories
        self._preload_relevant_context(user_context)

    def _preload_relevant_context(self, user_context):
        """Pre-load recent conversation memories into session context."""
        recent_memories = self.read_mcp.search_conversation_memory(
            query="recent context",
            user_id=user_context['user_id'],
            time_window_days=7,
            limit=3
        )

        if recent_memories:
            context = "Recent conversation context:\n"
            for mem in recent_memories:
                context += f"- {mem.intent_summary} ({mem.timestamp.strftime('%Y

            self.current_session.turns.append(
                ConversationTurn(role='system', content=context, is_preloaded=Tr
            )
```

6.2 Attention-Based Retrieval for Active Context

```python
class AttentionBasedRetrieval:
    """During conversation, retrieve memories with attention to current focus.""

    def retrieve_with_focus(
        self,
        current_query: str,
        conversation_context: List[ConversationTurn],
        user_context: dict
    ) -> FocusedMemoryResult:
        """Retrieve memories with attention to conversation trajectory."""

        # Extract focus from recent turns
        focus_analysis = self._analyze_conversation_focus(
            recent_turns=conversation_context[-5:],
            current_query=current_query
        )

        results = FocusedMemoryResult()

        # Personal: Recent relevant discussions
        if focus_analysis.needs_personal_context:
            results.personal = self.read_mcp.search_conversation_memory(
                query=focus_analysis.personal_query,
                user_id=user_context['user_id'],
                time_window_days=focus_analysis.time_window,
                limit=3
            )

        # Departmental: Functional knowledge
        if focus_analysis.needs_dept_knowledge:
            results.departmental = self.read_mcp.search_departmental_knowledge(
                query=focus_analysis.dept_query,
                department=user_context['department'],
                knowledge_type=focus_analysis.dept_type,
                limit=2
            )

        return results
```

Part 7: Browser-Side Implementation

7.1 IndexedDB JavaScript Implementation

```
// Client-side implementation for local graph storage

class ConversationGraphDB {
    constructor() {
        this.dbName = 'noor_conversation_graph';
        this.version = 1;
        this.db = null;
    }

    async init() {
        return new Promise((resolve, reject) => {
            const request = indexedDB.open(this.dbName, this.version);

            request.onerror = () => reject(request.error);
            request.onsuccess = () => {
                this.db = request.result;
                resolve(this.db);
            };

            request.onupgradeneeded = (event) => {
                const db = event.target.result;

                // Segments store
                if (!db.objectStoreNames.contains('segments')) {
                    const segmentStore = db.createObjectStore('segments', {
                        keyPath: 'id'
                    });

                    // Indexes for fast lookup
                    segmentStore.createIndex('session_id', 'session_id', { uniqu
                    segmentStore.createIndex('type', 'type', { unique: false });
                    segmentStore.createIndex('timestamp', 'timestamp', { unique:
                    segmentStore.createIndex('keywords', 'keywords', {
                        unique: false,
                        multiEntry: true  // Each keyword searchable
                    });
                }
            };
        });
    }

    async addSegment(segment) {
        const transaction = this.db.transaction(['segments'], 'readwrite');
        const store = transaction.objectStore('segments');
        return store.add(segment);
    }

    async getSegment(segmentId) {
        const transaction = this.db.transaction(['segments'], 'readonly');
        const store = transaction.objectStore('segments');
        return new Promise((resolve, reject) => {
            const request = store.get(segmentId);
            request.onsuccess = () => resolve(request.result);
            request.onerror = () => reject(request.error);
        });
    }

    async searchByKeywords(keywords) {
        const transaction = this.db.transaction(['segments'], 'readonly');
        const store = transaction.objectStore('segments');
        const index = store.index('keywords');

        const results = [];
        for (const keyword of keywords) {
            const request = index.getAll(keyword);
            const matches = await new Promise(resolve => {
                request.onsuccess = () => resolve(request.result);
            });
            results.push(...matches);
        }

        // Deduplicate and rank by keyword overlap
        return this._deduplicateAndRank(results, keywords);
    }
}
```

7.2 Client-Server Integration

```
Browser ↔ Server Communication:
┌───────────────────────────────────────────────────────────┐
│ BROWSER (Frontend)                                        │
│                                                         │
│  ┌─────────────────────────────────────────────┐        │
│  │ IndexedDB (Local Graph Storage)             │        │
│  │ - Extracted segments with full content      │        │
│  │ - Keyword indexes for fast search           │        │
│  │ - No cloud sync (sovereignty-safe)          │        │
│  └─────────────────────────────────────────────┘        │
│              ▲                    │                       │
│              │                    │                       │
│      Selective Expansion   Compression Event            │ │
│              │                    │                       │
└───────────────────────────────────────────────────────────┘
               │                    │
               │ API Request        │ WebSocket
               │                    ▼
┌─────────────────────────────────────────────────────────────┐
│ SERVER (Backend)                   │                        │
│                                    ▼                        │
│         ┌─────────────────────────────────────┐            │
│         │ Compression Pipeline                │            │
│         │ - Classifies content                   │              │
│         │ - Generates reference cards             │              │
│         │ - Sends extraction instructions         │              │
│         └──────────────────────────────────────┘              │
│                                                                │
│         ┌──────────────────────────────────────┐              │
│         │ Noor Agent                             │              │
│         │ - Processes queries with context       │              │
│         │ - Requests segment expansion           │              │
│         │ - Integrates retrieved segments        │              │
│         └──────────────────────────────────────┘              │
└──────────────────────────────────────────────────────────────┘
```

Part 8: Integration & Deployment

8.1 System Architecture Overview

```
Complete M3S v3.0 System Architecture:
┌───────────────────────────────────────────────────────────────┐
│                 MCP ROUTER / API GATEWAY                        │
│           (Request Authentication & Routing)                    │
└───────────────────────────────────────────────────────────────┘
                         │
                         │ MCP Protocol
              ┌──────────┴──────────┐
              ▼                     ▼
   ┌─────────────────────┐  ┌──────────────────────┐
   │  READ MCP SERVER     │  │ WRITE MCP SERVER      │
   │  (Noor + Maestro)    │  │ (Maestro only)        │
   │                      │  │                       │
   │ • Personal search    │  │ • Dept curation       │
   │ • Dept search        │  │ • Global curation     │
   │ • Global search      │  │ • Approval workflows  │
   │ • Smart routing      │  │ • Version control     │
   └─────────────────────┘  └──────────────────────┘
             │                          │
             └────────────┬─────────────┘
                          │
                          ▼
            ┌──────────────────────────────────────────┐
            │            Neo4j Database                  │
            │      (Four Independent Memory Tiers)       │
            │                                            │
            │   ┌──────────────────────────────────┐   │
            │   │ Personal (:ConversationMemory)     │   │
            │   │ Dept (:DepartmentalKnowledge)      │   │
            │   │ Global (:GlobalKnowledge)          │   │
            │   │ C-suite (:CSuiteKnowledge)         │   │
            │   └──────────────────────────────────┘   │
            │                                            │
            │  • Vector embeddings for semantic search   │
            │  • Access control via scope properties     │
            └──────────────────────────────────────────┘
                          │
                          │ Nightly Batch
                          ▼
         ┌───────────────────────────┐
         │ BATCH PROCESSING            │
         │ • Local Mistral-7B          │
         │ • Local BGE embeddings      │
         │ • Semantic chunking         │
         │ • Metadata extraction       │
         └───────────────────────────┘
                          │
                          │ Source data
                          ▼
         ┌───────────────────────────┐
         │ Conversation Database       │
         │ (Existing system)           │
         └───────────────────────────┘
```

```
Browser-Side (Per User):
┌──────────────────────────┐
│  IndexedDB Local Graph   │
│  • Extracted segments    │
│  • Full content preserved│
│  • Keyword indexes       │
│  • No cloud sync         │
└──────────────────────────┘
```

⚡ Four Independent Memory Tiers

The M3S architecture maintains four separate, independent memory tiers—each serving a distinct semantic domain:

- **Personal (Neo4j + Browser IndexedDB):** Per-user conversation history and preferences, automatically generated from conversations
- **Departmental (Neo4j):** Team-specific functional knowledge repository (how-to guides, expertise, project learnings)
- **Global (Neo4j):** Organization-wide events, policies, and cycles (budget submissions, quarterly reviews, announcements)
- **C-suite (Neo4j):** Executive-only restricted information repository (personnel, sensitive forecasts)

Search Strategy: When Noor receives a query, a local LLM classifies the query intent to determine which tier(s) contain relevant information. For example, "How do I submit a project proposal?" routes to Departmental, while "When is Q4 budget due?" routes to Global. No sequential fallback—each tier is queried based on semantic relevance.

8.2 Token Budget Analysis

Component	v2.1 (Original)	v3.0 (Reference-Based)	Improvement
Memory Recall Overhead	750-1,200 tokens	500-800 tokens	33-47% reduction
Context Representation	Full content always	Reference cards (50-100 tokens each)	80-90% compression

Component	v2.1 (Original)	v3.0 (Reference-Based)	Improvement
Selective Expansion	N/A	On-demand, only when needed	Zero overhead when not used
Information Loss	15-30% with summarization	0% (lossless with references)	100% retention
Context Window Utilization	60% effective	80% effective	33% improvement

8.3 Performance Targets

Compression Trigger Threshold:	80% context window fill
Unique Content Threshold:	80% unique after deduplication
Selective Retrieval Latency:	<100ms from IndexedDB
Batch Processing Time:	<5 hours for 10K conversations/day
Reference Card Size:	50-100 tokens (vs 500+ for full content)

Part 9: Expert Analysis

9.1 Advantages of Reference-Based Approach

1. Lossless Compression

- **Perfect Content Preservation:** Full segments stored in local graph, nothing summarized away
- **Code Integrity:** Technical content, function names, API specifications remain intact
- **Design Fidelity:** Architecture diagrams, specifications preserved verbatim

- **No Hallucination Risk:** LLM doesn't "fill in gaps" during summarization

2. Verifiable Extraction

- **User Inspection:** Can view what was extracted and reference card representation
- **Audit Trail:** Full extraction history in IndexedDB
- **Debugging:** Engineers can verify compression didn't lose critical context
- **Trust:** Users see keyword-based pointers, not opaque "summary"

3. Selective Expansion

- **Demand Paging:** Like OS memory management - fetch what you need when you need it
- **Zero Overhead:** Segments not referenced stay compressed
- **Intelligent Detection:** "Show me that code" triggers automatic expansion
- **Partial Expansion:** Can fetch just one segment without expanding all context

4. Technical Content Preservation

- **Code Blocks:** Function signatures, variable names, logic preserved exactly
- **Specifications:** Requirements, API docs, schemas remain precise
- **Numbers & Data:** Budget figures, metrics, calculations not approximated
- **Timestamps:** Dates and temporal information exact, not "about 2 weeks ago"

9.2 Sovereignty Compliance

On-Premise Processing Advantages

Government deployment constraint (employees can't take machines off-premise) becomes an architectural **advantage** rather than limitation:

- **Browser Storage:** IndexedDB stores extracted segments locally on government machines
- **No Cloud Sync:** No risk of sensitive data leaking to cloud storage
- **Local Models:** Mistral-7B and BGE run on-premise for all processing
- **Air-Gapped Operation:** Compression and retrieval work without internet
- **Data Residency:** All data remains within sovereign boundary

9.3 Scalability Considerations

Local Graph Size Management

```
// Periodic cleanup strategy for IndexedDB

async function cleanupOldSegments() {
    const db = await graphDB.init();
    const cutoffDate = new Date();
    cutoffDate.setDate(cutoffDate.getDate() - 90); // 90 days retention

    const oldSegments = await db.getSegmentsBefore(cutoffDate);

    // Archive to server (optional) or delete
    for (const segment of oldSegments) {
        if (segment.access_count < 2) {
            // Rarely accessed, safe to delete
            await db.deleteSegment(segment.id);
        } else {
            // Archive to personal Neo4j tier
            await archiveToNeo4j(segment);
            await db.deleteSegment(segment.id);
        }
    }
}
```

Cross-Session Knowledge Transfer

Challenge: Valuable insights in personal IndexedDB should be promoted to departmental/global tiers.

Solution: Maestro periodically reviews high-confidence personal memories and promotes to shared tiers after validation.

Part 10: Implementation Roadmap

Phase 1: Core Infrastructure (Weeks 1-3)

- Deploy Neo4j with vector index support
- Implement conversation chunking pipeline with local Mistral-7B
- Build BGE embedding service (local deployment)
- Create ConversationMemory schema and indexes
- Develop Read MCP server with semantic search
- **Deliverable:** Personal tier search operational

Phase 2: Compression System (Weeks 4-6)

- Implement ContentClassifier with all segment types
- Build LocalGraphExtractor with keyword extraction
- Develop ReferenceBasedCompressionManager
- Create browser-side IndexedDB implementation
- Implement reference card generation and replacement logic
- **Deliverable:** Context compression with reference cards working

Phase 3: Selective Retrieval (Weeks 7-8)

- Build SelectiveRetrieval with segment expansion
- Implement reference detection ("show me that code")
- Create keyword and entity search across local graph
- Integrate with Noor agent query processing
- Develop browser-server communication protocol

- **Deliverable:** Full compression and expansion cycle functional

Phase 4: Write MCP & Governance (Weeks 9-10)

- Implement Write MCP server (Maestro-only)
- Build DepartmentalKnowledge and GlobalKnowledge schemas
- Create approval workflow system
- Implement confidence threshold validation
- Build audit logging and version control
- **Deliverable:** Curated knowledge management operational

Phase 5: Testing & Optimization (Weeks 11-12)

- Integration testing across all components
- Performance profiling and optimization
- Token budget validation with real conversations
- Load testing batch processing (10K conversations/day)
- User acceptance testing with pilot group
- Security audit of MCP access controls
- **Deliverable:** Production-ready M3S v3.0

Appendices

Appendix A: MCP Server Registration

```
// Read MCP Server registration JSON
{
  "name": "memory-management-read",
  "version": "3.0.0",
  "description": "Four-tier memory retrieval with reference-based compression",
  "capabilities": {
    "tools": { "supported": true }
  },
  "tools": [
    {
      "name": "search_conversation_memory",
      "description": "Search personal conversation history",
      "inputSchema": {
        "type": "object",
        "properties": {
          "query": { "type": "string" },
          "user_id": { "type": "string" },
          "time_window_days": { "type": "integer", "default": null },
          "limit": { "type": "integer", "default": 5 }
        },
        "required": ["query", "user_id"]
      }
    },
    {
      "name": "intelligent_search",
      "description": "Smart routing across personal/dept/global tiers",
      "inputSchema": {
        "type": "object",
        "properties": {
          "query": { "type": "string" },
          "user_context": { "type": "object" },
          "limit_per_tier": { "type": "integer", "default": 3 }
        },
        "required": ["query", "user_context"]
      }
    }
  ]
}
```

Appendix B: Docker Compose Configuration

```
version: '3.8'

services:
  mcp-router:
    image: mcp-router:latest
    ports:
      - "8080:8080"
    environment:
      - AUTH_SECRET=${AUTH_SECRET}
    depends_on:
      - memory-read-service
      - memory-write-service

  memory-read-service:
    build: ./memory-read-mcp
    ports:
      - "8081:8081"
    environment:
      - NEO4J_URI=bolt://neo4j:7687
      - LOCAL_LLM_ENDPOINT=http://mistral:8080
      - LOCAL_EMBEDDER_ENDPOINT=http://bge:8080
    depends_on:
      - neo4j
      - mistral
      - bge

  memory-write-service:
    build: ./memory-write-mcp
    ports:
      - "8082:8082"
    environment:
      - NEO4J_URI=bolt://neo4j:7687
      - REQUIRE_AGENT=maestro
    depends_on:
      - neo4j

  neo4j:
    image: neo4j:5.15-enterprise
    ports:
      - "7474:7474"
      - "7687:7687"
    environment:
      - NEO4J_AUTH=neo4j/${NEO4J_PASSWORD}
      - NEO4J_vector_enabled=true
    volumes:
      - neo4j-data:/data

  mistral:
    image: mistral-7b-instruct:latest
    ports:
      - "8083:8080"
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]

  bge:
    image: bge-large-en-v1.5:latest
    ports:
      - "8084:8080"

volumes:
  neo4j-data:
```

Appendix C: Key Metrics Dashboard

Metric Category	Metric Name	Target	Alert Threshold
Compression	Compression trigger rate	20-30% of queries	>50%
	Average segments extracted/compression	3-5 segments	>10
	Token savings per compression	2,000-4,000 tokens	<1,000
Retrieval	Selective expansion rate	5-10% of queries	>25%
	Expansion latency (IndexedDB)	<100ms	>500ms
	Keyword search accuracy	>90%	<75%
Batch Processing	Nightly processing time	<4 hours	>6 hours
	Conversations processed/hour	>2,500	<1,500
	Chunking errors	<0.1%	>1%
Storage	IndexedDB size per user	<500MB	>1GB
	Neo4j memory nodes growth	100/user/month	>500/user/month

Modular Memory Management System (M3S) v3.0

Complete Architecture Specification | November 2025

Noor Cognitive Digital Twin Project