Contents

Cortex Memory System - Engineering Guide

Version: 4.20.0
Last Updated: January 2026
Audience: Backend Engineers, Platform Engineers, AI/ML Engineers

---

Table of Contents

---

1. System Architecture

1.0 The "Retrieval Dance" - Runtime Query Logic

Before diving into components, understand how the system answers a question:

```
// The four-step "Retrieval Dance"
async function retrievalDance(query: string, tenantId: string, userId: string): Promise<Respons
  // Step 1: INTENT PARSING (Hot Tier)
  const hotContext = await hotTier.getSessionContext(tenantId, userId);
  const ghostVector = await hotTier.getGhostVector(tenantId, userId);
  const entities = await nlp.extractEntities(query); // "Pump 302", "Q4 Report"

  // Step 2: GRAPH TRAVERSAL (Warm Tier)
  const graphResults = await warmTier.traverseGraph(tenantId, entities, {
    hops: 3,
    checkGoldenRules: true  //  CRITICAL: Check for overrides
  });

  // If Golden Rule exists, it takes priority
  if (graphResults.hasGoldenOverride) {
    return graphResults.goldenAnswer; // Skip further retrieval
  }

  // Step 3: DEEP FETCH (Cold Tier) - Only if needed
  let coldContent = null;
  if (graphResults.requiresColdFetch) {
    // Fetch ONLY the specific page/section needed, not entire documents
    coldContent = await coldTier.fetchViaStubNode(
      graphResults.stubNodeId,
      graphResults.specificRange // e.g., "pages 47-48 of 500-page PDF"
    );
  }

  // Step 4: SYNTHESIS (Foundation Model)
  const response = await llm.generate({
    query,
    context: hotContext,
    graphLogic: graphResults.paths,
    coldContent,
    chainOfCustody: buildAuditTrail(graphResults) // "Bob verified this on Jan 23"
  });

  return response;
}
```

1.1 Component Overview

```
                            CORTEX MEMORY SYSTEM



      HOT TIER                WARM TIER              COLD TIER



       Redis                   Neptune             S3 Iceberg
```

```
     Cluster              Graph DB             Tables


    DynamoDB             pgvector             Athena
    Overflow             Embeddings           Query





                       TIER COORDINATOR

                     • Data Flow Control
                     • TTL Enforcement
                     • Auto-Promotion
                     • GDPR Erasure
```

1.2 Technology Stack

Component	Technology	Purpose
Hot Cache	Redis 7.x (Cluster Mode)	Sub-10ms key-value storage
Hot Overflow	DynamoDB	Large value storage (>400KB)
Graph DB	Amazon Neptune	Relationship traversal
Vector Store	Aurora PostgreSQL + pgvector	Semantic similarity search
Archive Store	S3 + Apache Iceberg	Historical data warehouse
Query Engine	Amazon Athena	SQL over Iceberg tables
Orchestration	TierCoordinator Lambda	Data movement automation

1.3 File Structure

```
packages/
  shared/src/types/
     cortex-memory.types.ts       # Type definitions

  infrastructure/
     migrations/
        V2026_01_23_002__cortex_memory_system.sql

     lambda/
        shared/services/cortex/
```

```
             tier-coordinator.service.ts
             hot-tier.service.ts
             warm-tier.service.ts
             cold-tier.service.ts

         admin/
             cortex.ts              # Admin API handler

      lib/stacks/
          cortex-stack.ts           # CDK infrastructure

apps/admin-dashboard/
  app/(dashboard)/cortex/
      page.tsx                      # Overview dashboard
      graph/page.tsx                # Graph explorer
      conflicts/page.tsx            # Conflict resolution
      gdpr/page.tsx                 # GDPR erasure
```

---

2. Hot Tier Implementation

2.1 Redis Key Design

All keys follow the tenant-isolated pattern:

```
interface HotTierKeySchema {
  // Pattern: {tenant_id}:{type}:{identifier}

  sessionContext: `${tenantId}:session:${userId}:context`;
  ghostVector: `${tenantId}:ghost:${userId}`;
  telemetryFeed: `${tenantId}:telemetry:${streamId}`;
  prefetchCache: `${tenantId}:prefetch:${documentId}`;
}
```

2.2 Session Context Structure

```
interface SessionContext {
  userId: string;
  tenantId: string;
  conversationId: string;
  messages: ContextMessage[];
  activeTools: string[];
  tokenCount: number;
  createdAt: Date;
  expiresAt: Date;
}

interface ContextMessage {
  role: 'system' | 'user' | 'assistant';
  content: string;
  timestamp: Date;
  tokenCount?: number;
}
```

2.3 Ghost Vector Storage

Ghost Vectors are 4096-dimensional personality embeddings:

```
interface CortexGhostVector {
  userId: string;
  tenantId: string;
  vector: number[];  // 4096 dimensions
  personality: PersonalityTraits;
  lastUpdated: Date;
  interactionCount: number;
  version: number;
}

interface PersonalityTraits {
  formality: number;      // 0-1
  verbosity: number;      // 0-1
  technicalLevel: number; // 0-1
  humor: number;          // 0-1
  empathy: number;        // 0-1
}
```

2.4 Hot Tier Service Implementation

```
// hot-tier.service.ts
import { Redis } from 'ioredis';

class HotTierService {
  private redis: Redis;

  async getSessionContext(tenantId: string, userId: string): Promise<SessionContext | null> {
    const key = `${tenantId}:session:${userId}:context`;
    const data = await this.redis.get(key);
    return data ? JSON.parse(data) : null;
  }

  async setSessionContext(
    tenantId: string,
    userId: string,
    context: SessionContext,
    ttlSeconds: number = 14400
  ): Promise<void> {
    const key = `${tenantId}:session:${userId}:context`;
    await this.redis.setex(key, ttlSeconds, JSON.stringify(context));
  }

  async getGhostVector(tenantId: string, userId: string): Promise<CortexGhostVector | null> {
    const key = `${tenantId}:ghost:${userId}`;
    const data = await this.redis.get(key);
    return data ? JSON.parse(data) : null;
  }

  async updateGhostVector(
    tenantId: string,
    userId: string,
    vector: CortexGhostVector
  ): Promise<void> {
    const key = `${tenantId}:ghost:${userId}`;
    await this.redis.setex(key, 86400, JSON.stringify(vector)); // 24h TTL
  }

  async deleteAllForUser(tenantId: string, userId: string): Promise<number> {
    const pattern = `${tenantId}:*:${userId}:*`;
    const keys = await this.redis.keys(pattern);
    if (keys.length > 0) {
      return await this.redis.del(...keys);
    }
    return 0;
  }
}
```

2.5 DynamoDB Overflow

For values exceeding Redis limits (>400KB):

```
interface DynamoDBOverflowItem {
  pk: string;           // {tenant_id}#{type}
  sk: string;           // {identifier}
  data: string;         // Gzipped JSON
  ttl: number;          // Unix timestamp
  sizeBytes: number;
  createdAt: string;
}

async function storeWithOverflow(
  redis: Redis,
  dynamo: DynamoDB,
  key: string,
  value: object,
  ttlSeconds: number
): Promise<void> {
  const json = JSON.stringify(value);
  if (json.length < 400000) {
    await redis.setex(key, ttlSeconds, json);
  } else {
    // Store pointer in Redis, data in DynamoDB
    const [tenantId, type, ...rest] = key.split(':');
    const identifier = rest.join(':');

    await dynamo.putItem({
      TableName: 'cortex-hot-overflow',
      Item: {
        pk: { S: `${tenantId}#${type}` },
        sk: { S: identifier },
        data: { S: gzip(json) },
        ttl: { N: String(Math.floor(Date.now() / 1000) + ttlSeconds) },
        sizeBytes: { N: String(json.length) },
        createdAt: { S: new Date().toISOString() }
      }
    });

    await redis.setex(key, ttlSeconds, JSON.stringify({
      overflow: true,
      dynamoKey: `${tenantId}#${type}:${identifier}`
    }));
  }
}
```

---

3. Warm Tier Implementation

3.1 Graph-RAG Architecture

The Warm tier implements hybrid Graph-RAG search:

```
Query → Vector Search (40%) + Graph Traversal (60%) → Merged Results
```

3.2 Neptune Graph Schema

Golden Rules & Override System

```
// Golden Rule types - highest priority overrides
interface GoldenRule {
  id: string;
  tenantId: string;
  entityId: string;
  ruleType: 'force_override' | 'ignore_source' | 'prefer_source' | 'deprecate';
  condition: string;          // Query pattern that triggers this rule
  override: string;           // The verified answer to use
  reason: string;             // Why this override exists
  verifiedBy: string;          // Email of verifier
  verifiedAt: Date;
  signature: string;           // SHA-256 for audit trail
  expiresAt?: Date;            // Optional expiration
}

// Chain of Custody - audit trail for every fact
interface ChainOfCustody {
  factId: string;
  source: string;              // Original document/source
  extractedAt: Date;
  verifiedBy?: string;         // "Chief Engineer Bob"
  verifiedAt?: Date;           // "Jan 23, 2026"
  signature?: string;          // Digital signature
  supersedes?: string[];       // IDs of facts this replaces
}
```

Node Properties

```
// Document node
g.addV('document')
  .property('id', uuid)
  .property('tenantId', tenantId)
  .property('label', 'API Documentation v2.0')
  .property('source', 'confluence://page/12345')
  .property('hash', sha256)
  .property('confidence', 0.95)

// Entity node
g.addV('entity')
  .property('id', uuid)
  .property('tenantId', tenantId)
  .property('label', 'UserAuthenticationService')
  .property('entityType', 'class')
  .property('confidence', 0.88)

// Procedure node (evergreen)
g.addV('procedure')
  .property('id', uuid)
  .property('tenantId', tenantId)
  .property('label', 'Password Reset Flow')
  .property('isEvergreen', true)
  .property('confidence', 0.92)
```

Edge Relationships

```
// Document mentions entity
g.V(docId).addE('mentions').to(g.V(entityId))
```

```
  .property('weight', 0.8)
  .property('confidence', 0.95)

// Causal relationship
g.V(causeId).addE('causes').to(g.V(effectId))
  .property('weight', 0.7)

// Dependency
g.V(dependentId).addE('depends_on').to(g.V(dependencyId))
  .property('weight', 0.9)

// Version supersession
g.V(newVersionId).addE('supersedes').to(g.V(oldVersionId))
  .property('weight', 1.0)
```

3.3 Hybrid Search Implementation

```
// warm-tier.service.ts

interface HybridSearchResult {
  nodeId: string;
  label: string;
  nodeType: string;
  hybridScore: number;
  graphScore: number;
  vectorScore: number;
  path?: string[];
}

class WarmTierService {
  async hybridSearch(
    tenantId: string,
    query: string,
    queryVector: number[],
    options: {
      graphWeight?: number;
      vectorWeight?: number;
      limit?: number;
      nodeTypes?: string[];
    } = {}
  ): Promise<HybridSearchResult[]> {
    const {
      graphWeight = 0.6,
      vectorWeight = 0.4,
      limit = 10,
      nodeTypes
    } = options;
    // 1. Vector search via pgvector
    const vectorResults = await this.vectorSearch(tenantId, queryVector, limit * 2);

    // 2. Graph traversal from vector results
    const graphResults = await this.expandWithGraph(tenantId, vectorResults, nodeTypes);

    // 3. Merge and score
    const merged = this.mergeResults(vectorResults, graphResults, graphWeight, vectorWeight);

    return merged.slice(0, limit);
  }

  private async vectorSearch(
    tenantId: string,
    queryVector: number[],
    limit: number
  ): Promise<Array<{ nodeId: string; score: number }>> {
    const result = await executeStatement(`
      SELECT id, label, node_type,
             1 - (embedding <=> $2::vector) as similarity
      FROM cortex_graph_nodes
      WHERE tenant_id = $1 AND status = 'active'
      ORDER BY embedding <=> $2::vector
      LIMIT $3
    `, [tenantId, `[${queryVector.join(',')}]`, limit]);

    return result.rows.map(row => ({
      nodeId: row.id,
      score: row.similarity
    }));
  }

  private async expandWithGraph(
    tenantId: string,
    vectorResults: Array<{ nodeId: string; score: number }>,
    nodeTypes?: string[]
  ): Promise<Map<string, { score: number; path: string[] }>> {
    const nodeIds = vectorResults.map(r => r.nodeId);

    // Query Neptune for connected nodes
    const gremlinQuery = `
      g.V().has('tenantId', '${tenantId}')
        .hasId(within(${nodeIds.map(id => `'${id}'`).join(',')}))
        .repeat(both().simplePath())
        .times(2)
        .path()
        .by('id')
    `;
    const paths = await this.neptuneClient.query(gremlinQuery);

    const scores = new Map<string, { score: number; path: string[] }>();

    for (const path of paths) {
      const startScore = vectorResults.find(r => r.nodeId === path[0])?.score || 0;
      const decay = 0.7; // Score decays along path

      path.forEach((nodeId: string, index: number) => {
        const pathScore = startScore * Math.pow(decay, index);
        const existing = scores.get(nodeId);

        if (!existing || pathScore > existing.score) {
          scores.set(nodeId, { score: pathScore, path });
        }
      });
    }

    return scores;
  }

  private mergeResults(
    vectorResults: Array<{ nodeId: string; score: number }>,
    graphResults: Map<string, { score: number; path: string[] }>,
    graphWeight: number,
    vectorWeight: number
  ): HybridSearchResult[] {
    const allNodeIds = new Set([
      ...vectorResults.map(r => r.nodeId),
      ...graphResults.keys()
    ]);

    const merged: HybridSearchResult[] = [];

    for (const nodeId of allNodeIds) {
      const vectorScore = vectorResults.find(r => r.nodeId === nodeId)?.score || 0;
      const graphData = graphResults.get(nodeId) || { score: 0, path: [] };

      const hybridScore = (vectorScore * vectorWeight) + (graphData.score * graphWeight);

      merged.push({
        nodeId,
        label: '', // Fetch from DB
        nodeType: '',
        hybridScore,
        graphScore: graphData.score,
        vectorScore,
        path: graphData.path
      });
    }

    return merged.sort((a, b) => b.hybridScore - a.hybridScore);
  }
}
```

3.4 Deduplication Logic

```
async runDeduplication(tenantId: string): Promise<{ merged: number; errors: number }> {
  // Find duplicate nodes by normalized label
  const duplicates = await executeStatement(`
    SELECT LOWER(TRIM(label)) as label_norm,
           COUNT(*) as count,
           array_agg(id ORDER BY confidence DESC) as ids
    FROM cortex_graph_nodes
    WHERE tenant_id = $1 AND status = 'active'
    GROUP BY LOWER(TRIM(label))
    HAVING COUNT(*) > 1
    LIMIT 100
  `, [tenantId]);

  let merged = 0;
  let errors = 0;

  for (const dup of duplicates.rows) {
    const [keepId, ...mergeIds] = dup.ids;

    try {
      // Merge source documents
      await executeStatement(`
        UPDATE cortex_graph_nodes
        SET source_document_ids = (
          SELECT array_agg(DISTINCT doc_id)
          FROM cortex_graph_nodes, unnest(source_document_ids) doc_id
          WHERE id = ANY($1)
        )
        WHERE id = $2
      `, [[keepId, ...mergeIds], keepId]);

      // Redirect edges
      await executeStatement(`
        UPDATE cortex_graph_edges
        SET source_node_id = $1
        WHERE source_node_id = ANY($2)
      `, [keepId, mergeIds]);
      await executeStatement(`
        UPDATE cortex_graph_edges
        SET target_node_id = $1
        WHERE target_node_id = ANY($2)
      `, [keepId, mergeIds]);

      // Mark duplicates as deleted
      await executeStatement(`
        UPDATE cortex_graph_nodes
        SET status = 'deleted'
        WHERE id = ANY($1)
      `, [mergeIds]);

      merged += mergeIds.length;
    } catch (e) {
      errors++;
    }
  }

  return { merged, errors };
}
```

---

4. Cold Tier Implementation

4.1 Iceberg Table Schema

```
CREATE TABLE cortex_archives (
  tenant_id STRING,
  record_type STRING,
  record_id STRING,
  data STRING,            -- Compressed JSON
  archived_at TIMESTAMP,
  original_created_at TIMESTAMP,
  checksum STRING
)
PARTITIONED BY (tenant_id, date(archived_at), record_type)
LOCATION 's3://cortex-cold-archive/iceberg/'
TBLPROPERTIES (
  'table_type' = 'ICEBERG',
  'format' = 'parquet',
  'write.parquet.compression-codec' = 'snappy'
);
```

4.2 Archive Process

```
// cold-tier.service.ts
```

```
class ColdTierService {
  async archiveNodes(
    tenantId: string,
    nodeIds: string[]
  ): Promise<{ archived: number; sizeBytes: number }> {
    // Fetch nodes to archive
    const nodes = await executeStatement(`
      SELECT * FROM cortex_graph_nodes
      WHERE tenant_id = $1 AND id = ANY($2)
    `, [tenantId, nodeIds]);

    if (!nodes.rows.length) return { archived: 0, sizeBytes: 0 };

    // Prepare Iceberg records
    const records = nodes.rows.map(node => ({
      tenant_id: tenantId,
      record_type: 'graph_node',
      record_id: node.id,
      data: gzip(JSON.stringify(node)),
      archived_at: new Date().toISOString(),
      original_created_at: node.created_at,
      checksum: sha256(JSON.stringify(node))
    }));

    // Write to S3 via Iceberg
    const s3Key = `iceberg/${tenantId}/${new Date().toISOString().split('T')[0]}/nodes_${Date.

    await this.writeParquet(s3Key, records);

    // Track in metadata table
    const sizeBytes = records.reduce((sum, r) => sum + r.data.length, 0);

    await executeStatement(`
      INSERT INTO cortex_cold_archives
      (tenant_id, original_tier, original_table_name, archive_reason, s3_key,
       iceberg_table_name, record_count, size_bytes, checksum)
      VALUES ($1, 'warm', 'cortex_graph_nodes', 'age', $2, 'cortex_archives', $3, $4, $5)
    `, [tenantId, s3Key, nodeIds.length, sizeBytes, sha256(JSON.stringify(nodeIds))]);

    // Mark nodes as archived
    await executeStatement(`
      UPDATE cortex_graph_nodes
      SET status = 'archived', archived_at = NOW()
      WHERE id = ANY($1)
    `, [nodeIds]);

    return { archived: nodeIds.length, sizeBytes };
  }
  async retrieveFromCold(
    tenantId: string,
    recordIds: string[]
  ): Promise<any[]> {
    // Query Athena for archived records
    const query = `
      SELECT record_id, data
      FROM cortex_archives
      WHERE tenant_id = '${tenantId}'
        AND record_id IN (${recordIds.map(id => `'${id}'`).join(',')})
    `;

    const result = await this.athena.startQueryExecution({
      QueryString: query,
      ResultConfiguration: { OutputLocation: `s3://cortex-athena-results/${tenantId}/` }
    });

    // Wait for results
    const records = await this.waitForResults(result.QueryExecutionId);

    // Decompress and return
    return records.map(r => ({
      id: r.record_id,
      ...JSON.parse(gunzip(r.data))
    }));
  }
}
```

4.3 Stub Nodes - The Zero-Copy Innovation

The Problem: Tenants have 50TB+ in existing data lakes. Moving it is expensive and creates compliance issues.

The Solution: Stub Nodes - metadata pointers that enable graph queries over external data without copying it.

```
// Stub Node - metadata pointer to external content
interface StubNode {
  id: string;
  tenantId: string;
  nodeType: 'stub';

  // What this stub represents
  label: string;              // "Maintenance Log 2024.csv"
  description?: string;

  // Where the actual content lives
  externalSource: {
    mountId: string;          // Reference to Zero-Copy mount
    uri: string;              // "s3://bucket/logs/maintenance_2024.csv"
    format: 'csv' | 'json' | 'parquet' | 'pdf' | 'docx';
    sizeBytes: number;
    lastModified: Date;
  };

  // Partial metadata extracted during scan
  extractedMetadata: {
    columns?: string[];       // For tabular data
    pageCount?: number;       // For documents
    dateRange?: { start: Date; end: Date };
    entityMentions?: string[];
  };

  // Graph connections (these enable traversal without fetching content)
  connectedTo: string[];      // IDs of related nodes in the warm tier
}

// Fetch content ONLY when graph traversal determines it's needed
async function fetchViaStubNode(stubId: string, range?: ContentRange): Promise<Buffer> {
  const stub = await db.getStubNode(stubId);
  const mount = await db.getMount(stub.externalSource.mountId);

  // Generate signed URL for specific content range
  const signedUrl = await generateSignedUrl(mount, stub.externalSource.uri, range);

  // Fetch only what's needed (e.g., pages 47-48, not entire 500-page PDF)
  return await fetchWithRange(signedUrl, range);
}
```

4.4 Zero-Copy Mount Implementation

```
interface ZeroCopyMountConfig {
  snowflake?: {
    account: string;
    warehouse: string;
    database: string;
    schema: string;
    role?: string;
  };
  databricks?: {
    workspaceUrl: string;
    catalog: string;
    schema: string;
  };
  s3?: {
    bucket: string;
    prefix: string;
    region: string;
  };
}

class ZeroCopyMountService {
  async scanMount(mountId: string): Promise<ZeroCopyScanResult> {
    const mount = await this.getMount(mountId);

    let objects: Array<{ key: string; size: number; lastModified: Date }> = [];

    switch (mount.source_type) {
      case 'snowflake':
        objects = await this.scanSnowflake(mount.connection_config);
        break;
      case 's3':
        objects = await this.scanS3(mount.connection_config);
        break;
      case 'databricks':
        objects = await this.scanDatabricks(mount.connection_config);
        break;
    }

    // Index objects as graph nodes
    let nodesCreated = 0;
    for (const obj of objects) {
      const exists = await this.nodeExistsForObject(mount.tenant_id, mountId, obj.key);
      if (!exists) {
        await this.createNodeForObject(mount.tenant_id, mountId, obj);
        nodesCreated++;
      }
    }

    // Update mount stats
    await executeStatement(`
      UPDATE cortex_zero_copy_mounts
      SET status = 'active',
          last_scan_at = NOW(),
          object_count = $2,
          total_size_bytes = $3,
          indexed_node_count = indexed_node_count + $4
      WHERE id = $1
    `, [mountId, objects.length, objects.reduce((s, o) => s + o.size, 0), nodesCreated]);

    return {
      objectsScanned: objects.length,
      objectsIndexed: nodesCreated,
      nodesCreated,
      errorCount: 0,
      scannedAt: new Date()
    };
  }

  private async scanSnowflake(config: any): Promise<any[]> {
    // Use Snowflake connector to list tables/views
    const connection = await snowflake.createConnection(config);

    const result = await connection.execute({
      sqlText: `
        SELECT TABLE_NAME, ROW_COUNT, BYTES
        FROM INFORMATION_SCHEMA.TABLES
        WHERE TABLE_SCHEMA = '${config.schema}'
      `
    });

    return result.map(row => ({
      key: `${config.database}.${config.schema}.${row.TABLE_NAME}`,
      size: row.BYTES || 0,
      lastModified: new Date()
    }));
  }
}
```

---

5. Tier Coordinator Service

5.1 Core Orchestration Logic

```
// tier-coordinator.service.ts

class TierCoordinatorService {
  async orchestrateDataFlow(tenantId: string): Promise<DataFlowResult> {
    const config = await this.getConfig(tenantId);
    const results: DataFlowResult = {
      hotToWarm: { promoted: 0, errors: 0 },
      warmToCold: { archived: 0, errors: 0 },
      coldToWarm: { retrieved: 0, errors: 0 }
    };

    // 1. Hot → Warm promotion (for expired TTLs)
    if (config.enableAutoPromotion) {
      results.hotToWarm = await this.promoteHotToWarm(tenantId);
    }

    // 2. Warm → Cold archival (for aged data)
    if (config.enableAutoArchival) {
      results.warmToCold = await this.archiveWarmToCold(tenantId);
    }

    // 3. Record metrics
    await this.recordDataFlowMetrics(tenantId, results);

    return results;
  }

  async promoteHotToWarm(tenantId: string): Promise<{ promoted: number; errors: number }> {
    // Get expired session contexts from Redis
    const expiredKeys = await this.redis.keys(`${tenantId}:session:*:context`);

    let promoted = 0;
    let errors = 0;

    for (const key of expiredKeys) {
      const ttl = await this.redis.ttl(key);

      // If TTL is low, promote to warm tier
      if (ttl < 300) { // Less than 5 minutes remaining
        try {
          const data = await this.redis.get(key);
          if (data) {
            const session = JSON.parse(data);

            // Extract entities and create graph nodes
            await this.warmTier.ingestSession(tenantId, session);

            promoted++;
          }
        } catch (e) {
          errors++;
        }
      }
    }

    return { promoted, errors };
  }

  async archiveWarmToCold(tenantId: string): Promise<{ archived: number; errors: number }> {
    const config = await this.getConfig(tenantId);

    // Find nodes older than retention period (excluding evergreen)
    const nodesToArchive = await executeStatement(`
      SELECT id FROM cortex_graph_nodes
      WHERE tenant_id = $1
        AND status = 'active'
        AND is_evergreen = false
        AND node_type NOT IN ($2)
        AND created_at < NOW() - INTERVAL '1 day' * $3
      LIMIT 1000
    `, [tenantId, config.evergreenNodeTypes, config.warm.retentionDays]);

    if (!nodesToArchive.rows.length) {
      return { archived: 0, errors: 0 };
    }

    const nodeIds = nodesToArchive.rows.map(r => r.id);
    return await this.coldTier.archiveNodes(tenantId, nodeIds);
  }
}
```

5.2 GDPR Erasure Cascade

```
async processGdprErasure(requestId: string): Promise<void> {
  const request = await this.getErasureRequest(requestId);

  await this.updateRequestStatus(requestId, 'processing');

  try {
    // 1. Hot Tier - Immediate deletion
    await this.hotTier.deleteAllForUser(request.tenantId, request.userId);
    await this.updateTierStatus(requestId, 'hot', 'completed');

    // 2. Warm Tier - Anonymize or delete
    if (request.scopeType === 'user') {
      await executeStatement(`
        UPDATE cortex_graph_nodes
        SET status = 'deleted',
            properties = '{}',
            label = 'REDACTED'
        WHERE tenant_id = $1
          AND properties->>'created_by' = $2
      `, [request.tenantId, request.userId]);
    } else {
      // Tenant-wide deletion
      await executeStatement(`
        UPDATE cortex_graph_nodes
        SET status = 'deleted'
        WHERE tenant_id = $1
      `, [request.tenantId]);
    }
    await this.updateTierStatus(requestId, 'warm', 'completed');

    // 3. Cold Tier - Write tombstone records
    await this.coldTier.writeTombstones(request.tenantId, request.userId);
    await this.updateTierStatus(requestId, 'cold', 'completed');

    await this.updateRequestStatus(requestId, 'completed');
  } catch (error) {
    await this.updateRequestStatus(requestId, 'failed', error.message);
    throw error;
  }
}
```

---

6. Database Schema

6.1 Core Tables

See migration file: `V2026_01_23_002__cortex_memory_system.sql`

Key tables:

Table	Purpose	RLS Enabled
`cortex_config`	Per-tenant configuration	
`cortex_graph_nodes`	Knowledge graph nodes	
`cortex_graph_edges`	Node relationships	
`cortex_graph_documents`	Source documents	
`cortex_cold_archives`	Archive metadata	
`cortex_zero_copy_mounts`	External data sources	
`cortex_data_flow_metrics`	Flow statistics	
`cortex_tier_health`	Health snapshots	
`cortex_tier_alerts`	Threshold alerts	
`cortex_housekeeping_tasks`	Maintenance schedules	
`cortex_gdpr_erasure_requests`	Deletion tracking	
`cortex_conflicting_facts`	Contradiction detection	

6.2 Index Strategy

```sql
-- Vector similarity (IVFFlat for pgvector)
CREATE INDEX idx_cortex_graph_nodes_embedding
ON cortex_graph_nodes USING ivfflat (embedding vector_cosine_ops)
WITH (lists = 100);

-- Tenant + status lookups
CREATE INDEX idx_cortex_graph_nodes_status
ON cortex_graph_nodes(tenant_id, status);

-- Graph traversal support
CREATE INDEX idx_cortex_graph_edges_source ON cortex_graph_edges(source_node_id);
CREATE INDEX idx_cortex_graph_edges_target ON cortex_graph_edges(target_node_id);
```

```sql
-- Unresolved conflicts
CREATE INDEX idx_cortex_conflicting_facts_unresolved
ON cortex_conflicting_facts(tenant_id)
WHERE resolved_at IS NULL;
```

---

7. API Implementation

7.1 Lambda Handler Structure

```typescript
// lambda/admin/cortex.ts

export const handler = async (event: APIGatewayProxyEvent): Promise<APIGatewayProxyResult> =>
  const path = event.path.replace(/^\/api\/admin\/cortex/, '');
  const method = event.httpMethod;
  const tenantId = getTenantId(event);

  // Set RLS context
  await executeStatement(`SET app.current_tenant_id = '${tenantId}'`, []);

  // Route to handlers
  switch (true) {
    case path === '/overview' && method === 'GET':
      return getOverview(tenantId);
    case path === '/config' && method === 'GET':
      return getConfig(tenantId);
    case path === '/config' && method === 'PUT':
      return updateConfig(tenantId, JSON.parse(event.body));
    case path === '/health' && method === 'GET':
      return getTierHealth(tenantId);
    case path === '/health/check' && method === 'POST':
      return checkTierHealth(tenantId);
    // ... more routes
  }
};
```

7.2 Response Format

All API responses follow this structure:

```typescript
interface ApiResponse<T> {
  success: boolean;
  data?: T;
  error?: {
    code: string;
    message: string;
  };
  meta?: {
    timestamp: string;
    requestId: string;
  };
}
```

---

8. Migration Guide

8.1 Phase 1: Dual-Write Mode

Enable writing to both old and new systems:

```
async function dualWriteMemory(tenantId: string, userId: string, memory: Memory): Promise<void
  // Write to legacy table
  await legacyMemoryService.store(tenantId, userId, memory);

  // Write to Cortex hot tier
  await hotTierService.setSessionContext(tenantId, userId, {
    ...memory,
    conversationId: memory.sessionId
  });
}
```

8.2 Phase 2: Backfill Historical Data

```
-- Migrate existing memories to Warm tier
INSERT INTO cortex_graph_nodes (tenant_id, node_type, label, properties, embedding, created_at
SELECT
  tenant_id,
  'fact' as node_type,
  content as label,
  jsonb_build_object('legacy_id', id, 'store_id', store_id) as properties,
  embedding,
  created_at
FROM memories
WHERE NOT EXISTS (
  SELECT 1 FROM cortex_graph_nodes cgn
  WHERE cgn.properties->>'legacy_id' = memories.id::text
);
```

8.3 Phase 3: Read Fallback

```
async function getMemory(tenantId: string, userId: string): Promise<Memory> {
  // Try hot tier first
  const hot = await hotTierService.getSessionContext(tenantId, userId);
  if (hot) return hot;

  // Fall back to warm tier
  const warm = await warmTierService.searchByUser(tenantId, userId);
  if (warm.length) {
    // Promote to hot tier
    await hotTierService.setSessionContext(tenantId, userId, warm[0]);
    return warm[0];
  }

  // Fall back to legacy
  return legacyMemoryService.get(tenantId, userId);
}
```

8.4 Phase 4: Cut-Over

Disable legacy writes, enable legacy archival to Cold tier.

8.5 Phase 5: Deprecate Legacy

Remove legacy code paths after 30-day monitoring period.

---

9. Testing Strategy

9.1 Unit Tests

```
describe('TierCoordinatorService', () => {
  it('should promote expired hot tier data to warm', async () => {
    // Arrange
    await hotTier.setSessionContext('tenant1', 'user1', mockSession, 1);
    await sleep(2000); // Let TTL expire

    // Act
    const result = await tierCoordinator.promoteHotToWarm('tenant1');

    // Assert
    expect(result.promoted).toBe(1);
    const warmNode = await warmTier.getLatestForUser('tenant1', 'user1');
    expect(warmNode).toBeDefined();
  });

  it('should archive old warm tier data to cold', async () => {
    // Arrange
    await warmTier.createNode('tenant1', {
      ...mockNode,
      createdAt: new Date(Date.now() - 100 * 24 * 60 * 60 * 1000) // 100 days ago
    });

    // Act
    const result = await tierCoordinator.archiveWarmToCold('tenant1');

    // Assert
    expect(result.archived).toBe(1);
  });
});
```

9.2 Integration Tests

```
describe('Cortex E2E', () => {
  it('should handle full data lifecycle', async () => {
    // 1. Store in hot tier
    await api.post('/api/cortex/session', { userId: 'u1', context: {...} });

    // 2. Verify hot tier read
    const hot = await api.get('/api/cortex/session/u1');
    expect(hot.status).toBe(200);

    // 3. Trigger promotion
    await api.post('/api/admin/cortex/housekeeping/trigger', { taskType: 'archive_promotion' }

    // 4. Verify warm tier has data
    const warm = await api.get('/api/admin/cortex/graph/explore?search=u1');
    expect(warm.data.nodes.length).toBeGreaterThan(0);

    // 5. GDPR erasure
    await api.post('/api/admin/cortex/gdpr/erasure', { targetUserId: 'u1', scopeType: 'user' }

    // 6. Verify deletion
    const deleted = await api.get('/api/admin/cortex/graph/explore?search=u1');
    expect(deleted.data.nodes.length).toBe(0);
  });
});
```

---

10. Cortex v2.0 Implementation

10.1 Service Architecture

All v2.0 services follow consistent patterns:

File Locations:

```
packages/infrastructure/lambda/shared/services/cortex/
  golden-rules.service.ts      # Override system + Chain of Custody
  stub-nodes.service.ts        # Zero-copy pointers
  telemetry.service.ts         # MQTT/OPC UA injection
  entrance-exam.service.ts     # Curator verification
  graph-expansion.service.ts   # Twilight Dreaming v2
  model-migration.service.ts   # One-click model swap
  tier-coordinator.service.ts  # Core tier orchestration
```

API Handler:

packages/infrastructure/lambda/admin/cortex-v2.ts

Database Migration:

packages/infrastructure/migrations/V2026_01_23_003__cortex_v2_features.sql

10.2 Golden Rules Service

```
import { GoldenRulesService } from './cortex/golden-rules.service';

const service = new GoldenRulesService(db);

// Create a rule
const rule = await service.createRule({
  tenantId,
  ruleType: 'force_override',
  condition: 'max pressure Pump 302',
  override: 'The maximum pressure for Pump 302 is 100 PSI.',
  reason: 'Verified by Chief Engineer',
}, userId);

// Check for matches during retrieval
const match = await service.checkMatch(tenantId, 'What is the max pressure for Pump 302?');
if (match) {
  return match.override; // Skip further retrieval
}
```

10.3 Stub Nodes Service

```
import { StubNodesService } from './cortex/stub-nodes.service';

const service = new StubNodesService(db);

// Scan a mount and create stub nodes
const scanResult = await service.scanMount(mountId, tenantId);
// { created: 150, updated: 23, errors: [] }

// Fetch specific content range
const response = await service.fetchContent({
  tenantId,
  stubNodeId,
  range: { type: 'pages', start: 47, end: 48 }, // Only pages 47-48
  ttlSeconds: 3600,
});
// Returns signed URL for range-based fetch
```

10.4 Telemetry Service

```
import { TelemetryService } from './cortex/telemetry.service';
```

```
const service = new TelemetryService(db, redis);

// Create feed
const feed = await service.createFeed({
  tenantId,
  name: 'pump_302_sensors',
  protocol: 'opc_ua',
  endpoint: 'opc.tcp://plc.factory.local:4840',
  nodeIds: ['ns=2;s=Pump302.Pressure', 'ns=2;s=Pump302.Temperature'],
  pollIntervalMs: 1000,
  contextInjection: true,
});

// Get data for context injection
const snapshots = await service.getContextInjectionData(tenantId);
// Inject into AI context window
```

10.5 Entrance Exam Service

```
import { EntranceExamService } from './cortex/entrance-exam.service';

const service = new EntranceExamService(db);

// Generate exam for a domain
const exam = await service.generateExam({
  tenantId,
  domainId: 'hydraulics',
  domainPath: 'Engineering > Hydraulics',
  questionCount: 10,
  passingScore: 80,
});

// SME completes exam, corrections create Golden Rules
const result = await service.completeExam(examId, tenantId, userId);
// { passed: true, score: 90, goldenRulesCreated: ['rule-123'] }
```

10.6 Graph Expansion Service

```
import { GraphExpansionService } from './cortex/graph-expansion.service';

const service = new GraphExpansionService(db);

// Create and run expansion task
const task = await service.createTask({
  tenantId,
  taskType: 'infer_links',
  targetScope: 'domain',
});
```

```
const result = await service.runTask(task.id, tenantId);

// Review and approve inferred links
const pendingLinks = await service.getPendingLinks(tenantId);
await service.approveLink(linkId, tenantId, userId);
```

10.7 Model Migration Service

```
import { ModelMigrationService } from './cortex/model-migration.service';

const service = new ModelMigrationService(db);

// Initiate migration
const migration = await service.initiateMigration({
  tenantId,
  targetModel: { provider: 'meta', modelId: 'llama-3-70b-instruct' },
});

// Validate and test
const validation = await service.validateMigration(migration.id, tenantId);
const testResults = await service.runTests(migration.id, tenantId);

// Execute (or rollback)
await service.executeMigration(migration.id, tenantId);
// await service.rollbackMigration(migration.id, tenantId);
```

---

11. Performance Optimization

10.1 Redis Optimization

- **Pipeline batch operations**: Group related reads/writes
- **Use SCAN over KEYS**: Avoid blocking on large keyspaces
- **Compress large values**: Gzip values > 10KB

10.2 Neptune Optimization

- **Index frequently traversed edges**: Create composite indexes
- **Use path limiting**: Always set `times(N)` in repeat steps
- **Cache hot subgraphs**: Materialize frequently-accessed paths

10.3 pgvector Optimization

- **Tune IVFFlat lists**: Set `lists = sqrt(rows)` as baseline
- **Use HNSW for large datasets**: Better recall at scale
- **Reduce dimensions**: Consider PCA from 4096 → 1536

10.4 S3/Iceberg Optimization

- **Partition by tenant + date**: Prune scans effectively

- **Use Snappy compression**: Best speed/ratio balance
- **Compact small files**: Merge files < 128MB

---

12. The Sovereign Cortex Moats: Technical Deep Dive

The Cortex Memory System creates six interlocking competitive moats. This section provides the engineering details behind each.

12.1 Semantic Structure (Data Gravity 2.0)

The Problem with Vector RAG:

```
Traditional RAG: document → chunk → embed → similarity search
Result: "Pump 302" and "500 PSI" appear in same chunk (co-occurrence)
```

The Cortex Approach:

```
Cortex: document → extract entities → extract relationships → graph storage
Result: Pump_302 --(feeds)--> Valve_B --(pressure_limit)--> 500_PSI
```

Implementation:

```
// graph-rag.service.ts - Knowledge extraction
async extractKnowledge(tenantId: string, documentId: string, content: string) {
  // Extract triples via LLM
  const triples = await this.extractTriples(content, config);

  // Convert to typed entities and relationships
  for (const triple of triples) {
    const subjectEntity = {
      id: crypto.randomUUID(),
      tenantId,
      type: this.inferEntityType(triple.subject), // EQUIPMENT, PERSON, LOCATION, etc.
      name: triple.subject,
      properties: {},
      sourceDocumentIds: [documentId],
      confidence: triple.confidence,
    };

    const relationship = {
      sourceEntityId: subjectEntity.id,
      targetEntityId: objectEntity.id,
      type: this.inferRelationshipType(triple.predicate), // feeds, limits, contains
      description: triple.predicate,
      weight: triple.confidence,
    };
  }
}
```

Why Structure is Sticky: - Graph nodes are tenant-specific UUIDs (not portable) - Relationship types are learned from tenant data (not transferable) - Edge weights are calibrated through usage (not reproducible)

Database Schema:

```sql
CREATE TABLE cortex_graph_nodes (
  id UUID PRIMARY KEY,
  tenant_id UUID NOT NULL,
  node_type VARCHAR(50) NOT NULL,  -- equipment, person, process, etc.
  label VARCHAR(500) NOT NULL,
  properties JSONB DEFAULT '{}',
  embedding vector(4096),  -- For hybrid search
  source_document_ids UUID[] DEFAULT '{}',
  confidence DECIMAL(3,2),
  CONSTRAINT tenant_isolation CHECK (tenant_id = app.current_tenant_id)
);

CREATE TABLE cortex_graph_edges (
  id UUID PRIMARY KEY,
  tenant_id UUID NOT NULL,
  source_node_id UUID REFERENCES cortex_graph_nodes(id),
  target_node_id UUID REFERENCES cortex_graph_nodes(id),
  edge_type VARCHAR(100) NOT NULL,  -- feeds, contains, limits, requires
  weight DECIMAL(5,4) DEFAULT 1.0,
  properties JSONB DEFAULT '{}'
);
```

12.2 Chain of Custody (The Trust Ledger)

The Audit Problem: Standard AI systems cannot prove provenance. When asked "why did you say X?", they can only regenerate an explanation.

The Cortex Solution: Every critical fact is cryptographically signed during ingestion.

Implementation:

```typescript
// golden-rules.service.ts - Chain of Custody
async createChainOfCustody(entry: ChainOfCustodyEntry): Promise<ChainOfCustodyEntry> {
  // Generate verification hash
  const contentHash = crypto
    .createHash('sha256')
    .update(JSON.stringify({
      factId: entry.factId,
      originalContent: entry.originalContent,
      verifiedContent: entry.verifiedContent,
      verifierId: entry.verifierId,
      timestamp: entry.verificationTimestamp,
    }))
    .digest('hex');

  const result = await this.db.query(
    `INSERT INTO cortex_chain_of_custody (
      fact_id, tenant_id, original_content, verified_content,
      verifier_id, verifier_name, verifier_role, verification_type,
      verification_hash
    ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9)
    RETURNING *`,
    [entry.factId, entry.tenantId, entry.originalContent,
     entry.verifiedContent, entry.verifierId, entry.verifierName,
     entry.verifierRole, entry.verificationType, contentHash]
  );

  return this.mapRowToEntry(result.rows[0]);
}

async verifyChainOfCustody(factId: string, tenantId: string): Promise<boolean> {
  const entry = await this.getChainOfCustody(factId, tenantId);

  // Recompute hash and compare
  const expectedHash = crypto
    .createHash('sha256')
    .update(JSON.stringify({
      factId: entry.factId,
      originalContent: entry.originalContent,
      verifiedContent: entry.verifiedContent,
      verifierId: entry.verifierId,
      timestamp: entry.verificationTimestamp,
    }))
    .digest('hex');

  return entry.verificationHash === expectedHash;
}
```

Database Schema:

```
CREATE TABLE cortex_chain_of_custody (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  fact_id UUID NOT NULL,
  tenant_id UUID NOT NULL,
  original_content TEXT NOT NULL,
  verified_content TEXT NOT NULL,
  verifier_id UUID NOT NULL,
  verifier_name VARCHAR(255) NOT NULL,
  verifier_role VARCHAR(100) NOT NULL,
  verification_type verification_type NOT NULL,
  verification_timestamp TIMESTAMPTZ DEFAULT NOW(),
  verification_hash VARCHAR(64) NOT NULL,  -- SHA-256
  previous_hash VARCHAR(64),  -- For chain linking
  metadata JSONB DEFAULT '{}'
);
```

12.3 Tribal Delta (Heuristic Lock-in)

The Knowledge Gap: Foundation models know textbook answers. They don't know: - "In Mexico City, filters clog faster due to humidity" - "Bob prefers Verdana 11pt for all reports" - "The Friday checklist includes a step the manual forgot"

The Golden Rules System:

```
// golden-rules.service.ts
async createGoldenRule(rule: Omit<GoldenRule, 'id' | 'createdAt'>): Promise<GoldenRule> {
  const result = await this.db.query(
    `INSERT INTO cortex_golden_rules (
      tenant_id, domain_path, original_statement, corrected_statement,
      reason, severity, source_node_id, created_by, priority
    ) VALUES ($1, $2, $3, $4, $5, $6, $7, $8, $9)
    RETURNING *`,
    [rule.tenantId, rule.domainPath, rule.originalStatement,
     rule.correctedStatement, rule.reason, rule.severity,
     rule.sourceNodeId, rule.createdBy, rule.priority || 50]
  );

  return this.mapRowToRule(result.rows[0]);
}

async checkGoldenRules(tenantId: string, statement: string): Promise<GoldenRule | null> {
  // Find matching rule by semantic similarity or exact match
  const result = await this.db.query(
    `SELECT * FROM cortex_golden_rules
     WHERE tenant_id = $1
       AND is_active = true
       AND (
         original_statement ILIKE '%' || $2 || '%'
         OR $2 ILIKE '%' || original_statement || '%'
       )
     ORDER BY priority DESC, created_at DESC
     LIMIT 1`,
    [tenantId, statement]
  );

  return result.rows[0] ? this.mapRowToRule(result.rows[0]) : null;
}
```

Integration with Query Flow:

```
// In the Retrieval Dance
async processQuery(query: string, tenantId: string) {
  // Step 1: Get base AI response
  const baseResponse = await this.getModelResponse(query);

  // Step 2: Check for Golden Rule overrides
  const goldenRule = await this.goldenRulesService.checkGoldenRules(
    tenantId,
    baseResponse
  );

  if (goldenRule) {
    // Override with tenant-specific knowledge
    return {
      response: goldenRule.correctedStatement,
      source: 'golden_rule',
      ruleId: goldenRule.id,
      reason: goldenRule.reason,
    };
  }

  return { response: baseResponse, source: 'model' };
}
```

12.4 Sovereignty (Vendor Arbitrage)

The Lock-in Fear: Enterprises worry about building on a single AI provider that might raise prices or degrade.

The Intelligence Compiler: RADIANT treats the Cortex (data structure) as the permanent asset and models as swappable CPUs.

```
// model-migration.service.ts
async initiateMigration(request: MigrationRequest): Promise<ModelMigration> {
  // Validate target model is supported
  const targetConfig = this.getModelConfig(request.targetModel);
  if (!targetConfig) {
    throw new Error(`Unsupported target model: ${request.targetModel.modelId}`);
  }

  // Get current model config
  const currentModel = await this.getCurrentModel(request.tenantId);

  // Create migration record
  const migration = await this.db.query(
    `INSERT INTO cortex_model_migrations (
      tenant_id, source_model, target_model, status
    ) VALUES ($1, $2, $3, 'pending')
    RETURNING *`,
    [request.tenantId, JSON.stringify(currentModel),
     JSON.stringify(request.targetModel)]
  );

  return this.mapRowToMigration(migration.rows[0]);
}

async runTests(migrationId: string, tenantId: string): Promise<TestResults> {
  // Run test suite against new model
  const tests = [
    { type: 'accuracy', weight: 0.4 },
    { type: 'latency', weight: 0.2 },
    { type: 'cost', weight: 0.2 },
    { type: 'safety', weight: 0.2 },
  ];

  const results = await Promise.all(
    tests.map(t => this.runTestType(migrationId, t.type))
  );

  // Calculate weighted score
  const score = tests.reduce((acc, test, i) =>
    acc + (results[i].passed ? test.weight : 0), 0
  );

  return { tests: results, overallScore: score, passed: score >= 0.8 };
}
```

Key Insight: The Cortex stores: - Graph relationships (tenant-owned, not portable) - Golden Rules (tenant-specific overrides) - Chain of Custody (verification history)

None of this is tied to a specific model. Swap from Claude to GPT to Llama—the Cortex remains.

12.5 Entropy Reversal (Data Hygiene)

The Entropy Problem: Traditional systems accumulate contradictions: - Manual v2024 says "30 days" - Manual v2026 says "15 days" - Both are indexed. Which is correct?

Twilight Dreaming Solution:

```
// graph-expansion.service.ts
async findDuplicates(task: GraphExpansionTask): Promise<InferredLink[]> {
  // Find nodes with high embedding similarity
  const candidates = await this.db.query(
    `SELECT a.id as id_a, b.id as id_b,
            1 - (a.embedding <=> b.embedding) as similarity
```

```
      FROM cortex_graph_nodes a
      JOIN cortex_graph_nodes b ON a.tenant_id = b.tenant_id
      WHERE a.tenant_id = $1
        AND a.id < b.id  -- Avoid duplicates
        AND a.node_type = b.node_type
        AND 1 - (a.embedding <=> b.embedding) > 0.95
      ORDER BY similarity DESC
      LIMIT 100`,
     [task.tenantId]
  );

  return candidates.rows.map(row => ({
    sourceNodeId: row.id_a,
    targetNodeId: row.id_b,
    edgeType: 'duplicate_of',
    confidence: row.similarity,
    evidence: { method: 'embedding_similarity' },
  }));
}

async resolveConflicts(tenantId: string): Promise<void> {
  // Find conflicting facts
  const conflicts = await this.db.query(
    `SELECT * FROM cortex_conflicting_facts
     WHERE tenant_id = $1 AND status = 'pending'`,
    [tenantId]
  );

  for (const conflict of conflicts.rows) {
    // Apply resolution rules:
    // 1. Newer document supersedes older
    // 2. Higher-confidence source wins
    // 3. Golden Rule overrides both
    const resolution = await this.determineResolution(conflict);
    await this.applyResolution(conflict.id, resolution);
  }
}
```

Housekeeping Schedule:

```
const HOUSEKEEPING_TASKS = [
  { type: 'ttl_enforcement', frequency: 'hourly' },
  { type: 'archive_promotion', frequency: 'nightly' },
  { type: 'deduplication', frequency: 'nightly' },
  { type: 'graph_expansion', frequency: 'weekly' },
  { type: 'conflict_resolution', frequency: 'nightly' },
  { type: 'iceberg_compaction', frequency: 'nightly' },
  { type: 'index_optimization', frequency: 'weekly' },
];
```

12.6 Mentorship Equity (Sunk Cost)

The Engagement Problem: Traditional AI training is tedious data entry. SMEs disengage.

The Curator Quiz (Entrance Exam):

```
// entrance-exam.service.ts
async generateExam(request: ExamGenerationRequest): Promise<EntranceExam> {
  // Get facts from the domain
  const facts = await this.db.query(
    `SELECT * FROM cortex_graph_nodes
     WHERE tenant_id = $1
       AND properties->>'domain_path' LIKE $2 || '%'
       AND confidence < 0.9  -- Focus on uncertain facts
     ORDER BY confidence ASC
     LIMIT $3`,
    [request.tenantId, request.domainPath, request.questionCount]
  );

  // Generate quiz questions from facts
  const questions = facts.rows.map(fact => ({
    factId: fact.id,
    statement: this.formatAsQuestion(fact),
    expectedAnswer: fact.label,
    confidence: fact.confidence,
  }));

  return this.createExam({
    tenantId: request.tenantId,
    domainId: request.domainId,
    domainPath: request.domainPath,
    questions,
    passingScore: request.passingScore || 80,
  });
}

async processResults(examId: string, answers: ExamAnswer[]): Promise<ExamResults> {
  for (const answer of answers) {
    if (answer.isCorrect) {
      // Increase fact confidence + create Chain of Custody
      await this.db.query(
        `UPDATE cortex_graph_nodes SET confidence = LEAST(confidence + 0.1, 1.0)
         WHERE id = $1`,
        [answer.factId]
      );

      await this.goldenRulesService.createChainOfCustody({
        factId: answer.factId,
        tenantId: exam.tenantId,
        verifierId: exam.examineeId,
        verificationType: 'exam_verification',
      });
    } else {
      // Create Golden Rule from correction
      await this.goldenRulesService.createGoldenRule({
        tenantId: exam.tenantId,
        originalStatement: answer.originalStatement,
        correctedStatement: answer.correction,
        reason: 'SME correction during Entrance Exam',
        sourceNodeId: answer.factId,
        createdBy: exam.examineeId,
      });
    }
  }
}
```

Psychological Lock-in Metrics:

```
-- Track SME investment per tenant
SELECT
  tenant_id,
  COUNT(DISTINCT examinee_id) as sme_count,
  SUM(duration_minutes) as total_hours,
  COUNT(*) as exams_completed,
  SUM(CASE WHEN score >= passing_score THEN 1 ELSE 0 END) as passed
FROM cortex_entrance_exams
WHERE status = 'completed'
GROUP BY tenant_id;
```

---

13. Implementation Gap Analysis

Moat	Implementation Status	Gap	Notes
Semantic Structure	Fully Implemented	None	-
Chain of Custody	Fully Implemented	None	-
Tribal Delta	Fully Implemented	None	-
Sovereignty	Fully Implemented	None	-
Entropy Reversal	Fully Implemented	None	Hybrid 3-tier resolution (basic/LLM/human)

Moat	Implementation Status	Gap	Notes
Mentorship Equity	Fully Implemented	None	-
Zero-Copy Index	Fully Implemented	None	-

Hybrid Conflict Resolution (Entropy Reversal)

The conflict resolution system uses a 3-tier approach:

```
// Usage
const service = new GraphExpansionService(db, modelRouter);
const result = await service.resolveConflicts(tenantId);
// Returns: { resolved: 47, escalated: 3 }

// Manual resolution for escalated conflicts
await service.resolveConflictManually(
  conflictId,
  tenantId,
  userId,
  'MERGED',
  'Combined both sources for complete picture',
  'The filter replacement interval is 15 days in humid climates, 30 days otherwise'
);

// Get statistics
const stats = await service.getConflictStats(tenantId);
// { pending: 0, resolved: 47, escalated: 3, byTier: { basic: 42, llm: 5, human: 3 } }
```

Tier Distribution: - **Tier 1 (Basic Rules)**: ~95% - Date comparison, content length, similarity - **Tier 2 (LLM)**: ~4% - Semantic reasoning for numeric/contextual conflicts - **Tier 3 (Human)**: ~1% - Authoritative source conflicts, low-confidence LLM results

---

Document Version: 5.0.0
For operational procedures, see CORTEX-MEMORY-ADMIN-GUIDE.md