RAG Integration Implementation Plan

Complete Guide to Integrating all-rag-strategies into local-ai-packaged

Executive Summary

This document provides a comprehensive implementation plan to complete the integration of the **all-rag-strategies** advanced RAG system into your **local-ai-packaged** project. Based on the analysis of your current codebase, this plan addresses the critical gaps between your architectural documentation and the actual implementation.

Current Status

- ✅ **Complete:** Supabase infrastructure, n8n orchestration, Open WebUI interface
- ❌ **Missing:** RAG schema initialization, ingestion service integration, database connection standardization
- ⚠️ **Partial:** Documentation describes features not yet implemented

Critical Integration Gaps

1. Database Schema Initialization

Problem: The RAG schema from `all-rag-strategies/implementation/sql/schema.sql` is not being applied to the Supabase PostgreSQL database during startup.

Impact: Without the schema, the RAG system cannot store documents, chunks, or embeddings, rendering the entire retrieval system non-functional.

Current State:

- Your `start_services.py` script starts Supabase but never initializes the RAG schema
- The schema file exists but is not referenced in the startup process
- No database health checks or schema validation

Solution: Implement schema initialization through multiple approaches:

1. Copy schema.sql to Supabase init directory before startup
2. Apply schema via runtime execution after database is ready
3. Add database health checks and verification

2. Ingestion Service Missing

Problem: There is no ingestion service defined in the main `docker-compose.yml` file.

Impact: No automated way to populate the knowledge base with documents through the Docker infrastructure.

Current State:

- Ingestion service only exists in separate `docker-compose-ars.yml`
- Not integrated into the unified service stack
- No document processing pipeline in main architecture

Solution: Add ingestion service to main docker-compose with on-demand profile execution.

3. RAG Agent Service Missing

Problem: The RAG agent service that provides advanced retrieval capabilities is not part of the main service stack.

Impact: Sophisticated RAG strategies documented in your architecture are not accessible.

Current State:

- RAG agent exists only in `docker-compose-ars.yml`
- Separated from the main local-ai-packaged ecosystem
- No unified access to advanced RAG capabilities

Solution: Add RAG agent service to main docker-compose (optional, depending on architecture preference).

4. Database Connection Mismatch

Problem: Different services use incompatible database connection strings.

Impact: Services cannot communicate with the unified database, breaking the integrated architecture.

Current State:

- ARS services use: `postgresql://raguser:ragpass123@postgres:5432/postgres`
- Supabase services use: `postgresql://postgres:${POSTGRES_PASSWORD}@db:5432/postgres`
- Hostname mismatch (`postgres` vs `db`)
- Credential mismatch

Solution: Standardize all services to use the Supabase database configuration.

Implementation Plan

Phase 1: Enhanced Startup Script

Modifications to start_services.py

New Functions to Add

1. prepare_rag_schema()

```python
def prepare_rag_schema():
    """
    Prepare RAG database schema for initialization.
    Copies schema.sql to Supabase init directory.
    """
    schema_source = "all-rag-strategies/implementation/sql/schema.sql"
    init_dir = "supabase/docker/volumes/db/init"
    schema_dest = os.path.join(init_dir, "02-rag-schema.sql")

    os.makedirs(init_dir, exist_ok=True)
    shutil.copyfile(schema_source, schema_dest)
```

2. wait_for_database()

```python
def wait_for_database():
    """
    Wait for Supabase database to be fully ready.
    Checks database health via pg_isready.
    """
    max_attempts = 30
    for attempt in range(max_attempts):
        result = subprocess.run(
            ["docker", "exec", "supabase-db", "pg_isready", "-U", "postgres"],
            capture_output=True
        )
        if result.returncode == 0:
            return True
        time.sleep(2)
    return False
```

3. initialize_rag_schema_runtime()

```python
def initialize_rag_schema_runtime():
    """
    Apply RAG schema to running database as fallback.
    Uses docker exec and psql to apply schema.
    """
    with open("all-rag-strategies/implementation/sql/schema.sql") as f:
        schema_sql = f.read()

    process = subprocess.Popen(
        ["docker", "exec", "-i", "supabase-db",
         "psql", "-U", "postgres", "-d", "postgres"],
        stdin=subprocess.PIPE,
        text=True
    )
    process.communicate(input=schema_sql)
```

Updated Startup Sequence

The enhanced startup flow:

1. Clone/update Supabase repository
2. Prepare Supabase environment (.env copy)
3. **NEW:** Prepare RAG schema (copy to init directory)
4. Configure SearXNG
5. Stop existing containers
6. Start Supabase services
7. **NEW:** Wait for database to be ready
8. **NEW:** Verify/apply RAG schema
9. Start local AI services

Command Line Options

Add new flag to skip schema initialization:

```
python start_services.py --profile cpu --skip-schema-init
```

Phase 2: Docker Compose Integration

Service Definitions to Add

RAG Ingestion Service

Add to `docker-compose.yml`:

```
services:
  # ... existing services ...

  rag-ingestion:
    build:
      context: ./all-rag-strategies/implementation
      dockerfile: Dockerfile.rag
    container_name: rag_ingestion
    profiles:
      - ingestion
    depends_on:
      db:
        condition: service_healthy
    environment:
      - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@db:5432/postgres
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - LLM_CHOICE=${LLM_CHOICE:-gpt-4o-mini}
      - EMBEDDING_MODEL=${EMBEDDING_MODEL:-text-embedding-3-small}
    volumes:
      - ./all-rag-strategies/implementation/documents:/app/documents
      - ./all-rag-strategies/implementation:/app
      - ./.env:/app/.env
    command: ["uv", "run", "python", "ingestion/ingest.py", "--documents", "/app/document
    restart: "no"
```

Key Configuration Points:

- **Profile:** Uses `ingestion` profile for on-demand execution
- **Database:** Connects to Supabase `db` service (not separate postgres)
- **Volumes:** Mounts documents directory and implementation code
- **Environment:** Uses shared .env file for consistency
- **Restart:** Set to "no" since ingestion is a one-time operation

RAG Agent Service (Optional)

```
  rag-agent:
    build:
      context: ./all-rag-strategies/implementation
      dockerfile: Dockerfile.rag
    container_name: rag_agent
    restart: unless-stopped
    depends_on:
      db:
        condition: service_healthy
    environment:
      - DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@db:5432/postgres
      - OPENAI_API_KEY=${OPENAI_API_KEY}
      - LLM_CHOICE=${LLM_CHOICE:-gpt-4o-mini}
    volumes:
      - ./all-rag-strategies/implementation:/app
      - ./.env:/app/.env
    command: ["uv", "run", "python", "rag_agent.py"]
    expose:
      - 8000/tcp
```

Note: This service is optional because RAG functionality is primarily accessed through n8n workflows.

Dockerfile Creation

Create `all-rag-strategies/implementation/Dockerfile.rag`:

```
FROM python:3.11-slim

WORKDIR /app

# Install uv for fast Python package management
RUN pip install uv

# Copy requirements and install dependencies
COPY requirements.txt .
RUN uv pip install --system -r requirements.txt

# Copy application code
COPY . .

# Default command
CMD ["uv", "run", "python", "rag_agent.py"]
```

Phase 3: Environment Configuration

Required Environment Variables

Ensure your `.env` file contains:

```
# Database Configuration
POSTGRES_PASSWORD=your-super-secret-and-long-postgres-password
POSTGRES_HOST=db
POSTGRES_PORT=5432
POSTGRES_DB=postgres
DATABASE_URL=postgresql://postgres:${POSTGRES_PASSWORD}@db:5432/postgres

# AI Provider Configuration
OPENAI_API_KEY=sk-your-openai-api-key
LLM_CHOICE=gpt-4o-mini
EMBEDDING_MODEL=text-embedding-3-small

# Optional Alternative Providers
# ANTHROPIC_API_KEY=
# GEMINI_API_KEY=

# n8n Configuration
N8N_ENCRYPTION_KEY=your-encryption-key
N8N_USER_MANAGEMENT_JWT_SECRET=your-jwt-secret

# Supabase Configuration
JWT_SECRET=your-super-secret-jwt-token-with-at-least-32-characters-long
ANON_KEY=your-anon-key
SERVICE_ROLE_KEY=your-service-role-key
```

Database URL Standardization

Critical: All services must use the same database connection format:

✅ **Correct:**

```
postgresql://postgres:${POSTGRES_PASSWORD}@db:5432/postgres
```

❌ **Incorrect (from docker-compose-ars.yml):**

```
postgresql://raguser:ragpass123@postgres:5432/postgres
```

The hostname must be `db` (the Supabase database service name), not `postgres`.

Phase 4: Integration Architecture

System Architecture Diagram

```
┌─────────────────────────────────────────────────────────────────────┐
│                      Local AI Stack (Unified)                       │
├─────────────────────────────────────────────────────────────────────┤
│                                                                     │
│  ┌──────────────┐       ┌──────────────┐       ┌──────────────┐    │
│  │  Open WebUI  │──────▶│     n8n      │──────▶│   Supabase   │    │
│  │              │       │              │       │   Studio     │    │
│  └──────────────┘       └──────┬───────┘       └──────────────┘    │
│                                │                                    │
│                                ▼                                    │
│                       ┌────────────────┐                            │
│                       │  RAG Workflow  │                            │
│                       │   (Template)   │                            │
│                       └───────┬────────┘                            │
│                               │                                     │
│                               ▼                                     │
│  ┌──────────────────────────────────────────────────────────────┐  │
│  │         Supabase PostgreSQL + pgvector Extension             │  │
│  │                                                              │  │
│  │  ┌──────────────┐  ┌──────────────┐  ┌──────────────────┐   │  │
│  │  │  documents   │  │  chunks      │  │ embeddings_idx   │   │  │
│  │  │    table     │  │  table       │  │   (vector)       │   │  │
│  │  └──────────────┘  └──────────────┘  └──────────────────┘   │  │
│  │                                                              │  │
│  └──────────────────────────────▲───────────────────────────────┘  │
│                                 │                                   │
│              ┌──────────────────┴──────────────────┐                │
│              │                                     │                │
│  ┌───────────┴──────────────┐       ┌──────────────┴─────┐         │
│  │  rag-ingestion           │       │  rag-agent         │ (optional)│
│  │  (on-demand via          │       │ (CLI/API)          │         │
│  │   --profile)             │       │                    │         │
│  └──────────────────────────┘       └────────────────────┘         │
│                                                                        │
└──────────────────────────────────────────────────────────────────────────────────────┘
```

Data Flow

1. **Document Ingestion:**
   - User places documents in `all-rag-strategies/implementation/documents/`
   - Runs: `docker compose -p localai --profile ingestion up`
   - Ingestion service processes documents, creates chunks, generates embeddings
   - Data stored in Supabase PostgreSQL
2. **Query Processing:**
   - User submits query via Open WebUI
   - Open WebUI forwards to n8n via pipe script
   - n8n executes RAG workflow
   - Workflow queries database for relevant chunks
   - LLM generates response with retrieved context
   - Response returned to user

Phase 5: Testing & Verification

Step 1: Verify Schema Initialization

After starting services, check if RAG schema was applied:

```
# Connect to database
docker exec -it supabase-db psql -U postgres -d postgres

# List tables (should see documents, chunks, etc.)
\dt

# Check for pgvector extension
\dx

# Exit
\q
```

Expected output:

```
                   List of relations
 Schema |        Name        | Type  |  Owner
--------+--------------------+-------+----------
 public | chunks             | table | postgres
 public | documents          | table | postgres
```

Step 2: Test Ingestion

```
# Place test documents in documents directory
mkdir -p all-rag-strategies/implementation/documents
echo "Test document content" &gt; all-rag-strategies/implementation/documents/test.txt

# Run ingestion
docker compose -p localai --profile cpu --profile ingestion up rag-ingestion

# Check logs
docker logs rag_ingestion

# Verify in database
docker exec -it supabase-db psql -U postgres -d postgres -c "SELECT COUNT(*) FROM documer
```

Step 3: Test RAG Query

Via n8n workflow:

1. Access n8n at http://localhost:8001
2. Import RAG workflow template
3. Test with sample query
4. Verify response includes relevant context

Via CLI (if rag-agent service added):

```
docker exec -it rag_agent python rag_agent.py
```

Troubleshooting Guide

Issue 1: Schema Not Applied

Symptoms:

- Tables don't exist in database
- Queries fail with "relation does not exist"

Diagnosis:

```
docker exec -it supabase-db psql -U postgres -d postgres -c "\dt"
```

Solutions:

1. Check if schema file exists: `ls all-rag-strategies/implementation/sql/schema.sql`
2. Manually apply schema:

```
docker exec -i supabase-db psql -U postgres -d postgres &lt; all-rag-strategies/imple
```

3. Re-run enhanced start script: `python start_services_enhanced.py --profile cpu`

Issue 2: Ingestion Service Fails

Symptoms:

- Container exits immediately
- No documents processed

Diagnosis:

```
docker logs rag_ingestion
```

Common Causes:

1. **Database not ready:** Wait longer or check db health
2. **Missing API key:** Verify OPENAI_API_KEY in .env
3. **Wrong DATABASE_URL:** Check connection string format
4. **No documents:** Ensure documents directory has files

Solutions:

```
# Check database connection
docker exec -it rag_ingestion psql $DATABASE_URL -c "SELECT 1;"

# Verify environment variables
docker exec -it rag_ingestion env | grep DATABASE_URL

# Check documents directory
docker exec -it rag_ingestion ls -la /app/documents
```

Issue 3: Database Connection Errors

Symptoms:

- "could not connect to server"
- "connection refused"

Diagnosis:

```
# Check if db container is running
docker ps | grep supabase-db

# Test connection from host
docker exec -it supabase-db pg_isready -U postgres
```

Solutions:

1. Ensure Supabase is fully started: wait 30 seconds after startup
2. Check network connectivity:
   ```
   docker network inspect localai_default
   ```
3. Verify DATABASE_URL uses `db` hostname (not `localhost` or `postgres`)

Issue 4: n8n Can't Access RAG Data

Symptoms:

- Workflow executes but returns no results
- Database queries timeout

Solutions:

1. Verify n8n can reach database:
   ```
   docker exec -it n8n ping db
   ```
2. Check n8n environment variables include DATABASE_URL
3. Update n8n workflow credentials with correct connection string
4. Test query directly in database to verify data exists

Migration Checklist

Use this checklist to track your implementation progress:

Pre-Implementation

- [ ] Back up current working configuration
- [ ] Review all attached files and understand current architecture
- [ ] Identify any custom modifications to preserve
- [ ] Document current service URLs and access points

Implementation Steps

- [ ] Replace `start_services.py` with enhanced version
- [ ] Add `rag-ingestion` service to `docker-compose.yml`
- [ ] (Optional) Add `rag-agent` service to `docker-compose.yml`
- [ ] Create `Dockerfile.rag` in `all-rag-strategies/implementation/`
- [ ] Update `.env` file with required variables
- [ ] Verify DATABASE_URL format in all service definitions

Testing

- [ ] Stop all existing containers
- [ ] Run enhanced startup script
- [ ] Verify Supabase database is healthy
- [ ] Confirm RAG schema is applied
- [ ] Test document ingestion
- [ ] Verify data appears in database
- [ ] Test RAG query via n8n workflow
- [ ] Check all services are accessible

Documentation

- [ ] Update project README with new startup instructions
- [ ] Document ingestion process for team
- [ ] Create runbook for troubleshooting
- [ ] Update architecture diagrams (CLAUDE.md)

Next Steps After Implementation

1. Optimize Configuration

- Tune embedding model for your use case
- Adjust chunk sizes for optimal retrieval
- Configure LLM parameters (temperature, max tokens)

2. Add Documents

- Organize documents by category in subdirectories
- Create ingestion schedule for regular updates
- Implement document versioning strategy

3. Enhance n8n Workflows

- Create specialized workflows for different query types
- Add conversation memory and context
- Implement query refinement and feedback loops

4. Monitor Performance

- Set up logging and metrics collection
- Track query latency and accuracy
- Monitor database size and performance

5. Security Hardening

- Implement authentication for services
- Use secrets management for API keys
- Enable SSL/TLS for production deployments

Conclusion

This implementation plan provides a complete roadmap to bridge the gap between your documented architecture and actual implementation. The key insight is that your foundation is solid—all the pieces exist, they just need to be connected properly.

Summary of Changes:

1. **Enhanced startup script** adds RAG schema initialization
2. **Docker compose additions** integrate ingestion and agent services
3. **Standardized configuration** ensures all services use the same database
4. **Testing procedures** verify complete integration
5. **Troubleshooting guide** helps resolve common issues

After completing these changes, your local-ai-packaged project will have a fully functional, production-ready RAG system that matches your architectural vision.

References

- Original AI Analysis Response (provided in query)
- CONTEXT.md - Integration architecture documentation
- PROMPT.md - Integration implementation notes
- start_services.py - Current startup script
- docker-compose.yml - Main service definitions
- docker-compose-ars.yml - RAG service reference
- rag_agent.py - RAG CLI implementation