Table of Contents

RAG Implementation Analysis: Unitasa Marketing Intelligence Platform

Executive Summary

This document provides a comprehensive analysis of the Retrieval-Augmented Generation (RAG) patterns and design decisions implemented in the Unitasa platform. The system demonstrates a sophisticated multi-layered RAG architecture with advanced retrieval strategies, confidence scoring, and intelligent LLM routing.

Current Status: Agentic RAG v0.8 (Partial Implementation)

- ☑ Intent-based routing, Multi-source knowledge, Context accumulation, Action-taking, Qualification signal extraction
- ❌ Tool use / function calling, Multi-step reasoning (ReAct loop), Query decomposition, Self-reflection, Iterative refinement

---

1. Dense Vector Retrieval with ChromaDB

Concept Identification

Pattern: Dense semantic retrieval using embeddings and vector similarity search

Implementation Details

Location: `app/rag/vectorstore_manager.py`

Key Components:

- `VectorStoreManager` class manages ChromaDB operations
- OpenAI embeddings (`text-embedding-ada-002` or similar)
- Persistent storage in `./chroma_db` directory
- Collection-based organization (`marketing_knowledge` default)

How It Works:

```
# Vector store initialization
vectorstore = Chroma(
```

```
    collection_name="marketing_knowledge",
    embedding_function=OpenAIEmbeddings(),
    persist_directory="./chroma_db"
)

# Similarity search with scoring
docs_with_scores = vectorstore.similarity_search_with_score(
    query="CRM integration best practices",
    k=4
)
```

Design Motivation ("Why")

Problem Solved:

- Semantic understanding of user queries beyond keyword matching
- Efficient retrieval from large knowledge bases
- Persistent storage of marketing domain knowledge

Why This Approach:

- **Dense retrieval** captures semantic meaning better than keyword search
- **ChromaDB** provides lightweight, embedded vector database (no separate server needed)
- **OpenAI embeddings** offer high-quality semantic representations
- **Persistence** ensures knowledge base survives restarts

Trade-offs:

- ☑ Excellent semantic understanding
- ☑ Fast similarity search with HNSW indexing
- ☑ Easy deployment (embedded database)
- ⚠ Embedding costs for large document sets
- ⚠ Limited to semantic similarity (may miss exact keyword matches)

Integration Points

Data Flow:

1. **Ingestion** → Documents chunked → Embedded → Stored in ChromaDB
2. **Query** → User question embedded → Similarity search → Top-k documents retrieved
3. **Generation** → Retrieved docs + query → LLM → Response

Connected Components:

- `app/rag/ingestion.py` - Feeds documents into vector store
- `app/rag/advanced_retrievers.py` - Uses vector store for retrieval
- `app/rag/lcel_chains.py` - Orchestrates retrieval + generation

Best Practices Applied

☑ **Metadata enrichment** - Documents tagged with source, timestamp, chunk info ☑ **Persistence** - Vector store persisted to disk for durability ☑ **Async operations** - Non-blocking database operations ☑ **Collection stats** - Monitoring document count and health

Areas for Improvement

1. **Hybrid search** - Combine with BM25 for better keyword matching (partially implemented in EnsembleRetriever)
2. **Embedding caching** - Cache embeddings to reduce API costs
3. **Index optimization** - Tune HNSW parameters for your data distribution
4. **Multi-tenancy** - Separate collections per customer/use case

---

2. Document Chunking Strategy

Concept Identification

Pattern: Recursive character text splitting with overlap

Implementation Details

Location: `app/rag/ingestion.py`

Configuration:

```
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,        # Characters per chunk
    chunk_overlap=200,      # Overlap between chunks
    separators=["\n\n", "\n", " ", ""],  # Split hierarchy
    length_function=len
)
```

Supported Formats:

- PDF (via `PyPDFLoader`)
- Markdown (via `UnstructuredMarkdownLoader`)
- Text files (via `TextLoader`)
- HTML (via `BSHTMLLoader`)
- JSON (custom parsing)
- Web URLs (via `WebBaseLoader`)

Design Motivation ("Why")

Problem Solved:

- Long documents exceed LLM context windows
- Need to maintain semantic coherence in chunks
- Balance between chunk size and retrieval precision

Why This Approach:

- **1000 character chunks** - Optimal balance for context vs. precision
- **200 character overlap** - Prevents information loss at boundaries
- **Hierarchical separators** - Respects document structure (paragraphs → sentences → words)
- **Recursive splitting** - Ensures chunks don't exceed size limits

Trade-offs:

- ☑ Maintains context across chunk boundaries
- ☑ Respects natural document structure
- ☑ Flexible for various document types
- ⚠ Overlap increases storage requirements by ~20%
- ⚠ Fixed chunk size may split mid-concept

Integration Points

Data Flow:

```
Raw Document → Loader → Text Splitter → Chunks → Metadata Addition → Vector Store
```

Metadata Enrichment:

```
chunk.metadata.update({
    "chunk_id": f"chunk_{i}",
    "chunk_index": i,
    "total_chunks": len(chunks),
    "ingestion_timestamp": datetime.utcnow().isoformat()
})
```

Best Practices Applied

☑ **Semantic splitting** - Respects paragraph and sentence boundaries ☑ **Overlap strategy** - Prevents context loss ☑ **Metadata tracking** - Each chunk knows its position ☑ **Multi-format support** - Handles diverse document types

Areas for Improvement

1. **Semantic chunking** - Use embeddings to detect topic boundaries
2. **Dynamic chunk sizing** - Adjust based on content density
3. **Table/code preservation** - Special handling for structured content
4. **Language-aware splitting** - Different strategies for different languages

---

3. Multi-Query Retrieval

Concept Identification

Pattern: Query expansion and result fusion

Implementation Details

Location: `app/rag/advanced_retrievers.py` - `MultiQueryRetriever`

How It Works:

```
# Generate query variations
query_variations = [
    "How to integrate CRM?",
    "What is the process to integrate CRM?",  # Rephrased
    "integrate CRM",  # Keyword focus
]

# Retrieve for each variation
for query_var in query_variations:
    docs = vectorstore.similarity_search(query_var, k=3)
    all_docs.extend(docs)

# Deduplicate and return top results
return deduplicated_docs[:5]
```

Query Expansion Strategy:

- Rephrase "how" → "what is the process"
- Rephrase "what" → "explain"
- Extract first 3 words (keyword focus)
- Extract last 3 words (keyword focus)

Design Motivation ("Why")

Problem Solved:

- Single query formulation may miss relevant documents
- Users phrase questions differently than document content
- Improves recall without sacrificing precision

Why This Approach:

- **Query diversity** - Captures different phrasings of same intent
- **Deduplication** - Prevents redundant results
- **Fusion strategy** - Combines results from multiple searches

Trade-offs:

- ☑ Improved recall (finds more relevant docs)
- ☑ Robust to query phrasing
- ☑ Simple implementation
- ⚠ 3-5x retrieval cost (multiple searches)

- ⚠ Basic query expansion (could use LLM for better variations)

Integration Points

Used in `app/rag/lcel_chains.py` as an alternative retrieval strategy:

```
retriever = get_advanced_retriever(strategy="multi_query")
```

Best Practices Applied

☑ **Deduplication** - Content-based hashing prevents duplicates ☑ **Result limiting** - Returns top 5 to control context size

Areas for Improvement

1. **LLM-powered expansion** - Use LLM to generate better query variations
2. **Reciprocal Rank Fusion** - Better result merging algorithm
3. **Adaptive expansion** - Expand more for ambiguous queries
4. **Query analysis** - Detect when expansion is beneficial

---

4. Ensemble Retrieval (Hybrid Search)

Concept Identification

Pattern: Combining dense (semantic) and sparse (keyword) retrieval

Implementation Details

Location: `app/rag/advanced_retrievers.py` - `EnsembleRetriever`

How It Works:

```
# Semantic search (dense vectors)
semantic_docs = vectorstore.similarity_search(query, k=5)

# Keyword search (BM25)
bm25_docs = BM25Retriever.get_relevant_documents(query)

# Combine and deduplicate
all_docs = semantic_docs + bm25_docs
unique_docs = deduplicate(all_docs)[:5]
```

BM25 Component:

- Traditional TF-IDF based ranking
- Excellent for exact keyword matches
- Complements semantic search

Design Motivation ("Why")

Problem Solved:

- Dense retrieval misses exact keyword matches
- Sparse retrieval misses semantic similarity
- Best of both worlds approach

Why This Approach:

- **Complementary strengths** - Semantic + keyword matching
- **Improved coverage** - Catches both conceptual and literal matches
- **Industry standard** - Proven effective in production systems

Trade-offs:

- ☑ Best recall and precision balance
- ☑ Handles both semantic and keyword queries
- ☑ Robust to different query types
- ⚠ Requires maintaining BM25 index
- ⚠ More complex than single retrieval method
- ⚠ Need to tune fusion weights

Integration Points

Default retrieval strategy in the system:

```
retriever = get_advanced_retriever(strategy="ensemble")  # Default
```

Best Practices Applied

☑ **Hybrid approach** - Industry best practice for RAG ☑ **Deduplication** - Prevents redundant results from both methods

Areas for Improvement

1. **Weighted fusion** - Assign weights to semantic vs. keyword results
2. **Reciprocal Rank Fusion (RRF)** - Better merging algorithm
3. **Dynamic weighting** - Adjust weights based on query type
4. **Cross-encoder reranking** - Final reranking stage for precision

---

5. Contextual Compression

Concept Identification

Pattern: Post-retrieval document compression to keep only relevant content

Implementation Details

Location: `app/rag/advanced_retrievers.py` - `ContextualCompressionRetriever`

How It Works:

```
# Retrieve more documents than needed
raw_docs = vectorstore.similarity_search(query, k=10)

# Compress each document
for doc in raw_docs:
    # Extract sentences containing query terms
    query_terms = set(query.lower().split())
    relevant_sentences = [
        sentence for sentence in doc.split('.')
        if any(term in sentence.lower() for term in query_terms)
    ]
    compressed_content = '. '.join(relevant_sentences)

# Return top compressed documents
return compressed_docs[:5]
```

Design Motivation ("Why")

Problem Solved:

- Retrieved chunks contain irrelevant information
- LLM context window is limited and expensive
- Reduce noise in the context

Why This Approach:

- **Focused context** - Only relevant sentences passed to LLM
- **Cost reduction** - Fewer tokens = lower API costs
- **Improved accuracy** - Less noise = better answers

Trade-offs:

- ☑ Reduces context size by 50-70%
- ☑ Lowers LLM API costs
- ☑ Improves answer relevance
- ⚠ May lose important context
- ⚠ Simple keyword matching (could use LLM for better compression)
- ⚠ Additional processing overhead

Integration Points

Available as alternative retrieval strategy:

```
retriever = get_advanced_retriever(strategy="contextual")
```

Best Practices Applied

☑ **Over-retrieval then compress** - Retrieve 10, return 5 compressed ☑ **Minimum length check** - Ensures compressed content is meaningful

Areas for Improvement

1. **LLM-based compression** - Use small LLM to extract relevant content
2. **Extractive summarization** - More sophisticated than sentence filtering
3. **Relevance scoring** - Score each sentence for relevance
4. **Preserve context** - Keep surrounding sentences for coherence

---

6. Confidence Scoring System

Concept Identification

Pattern: Multi-factor confidence assessment for RAG responses

Implementation Details

Location: `app/rag/confidence_scorer.py` - `RAGConfidenceScorer`

Confidence Factors (weighted):

```
confidence_weights = {
    'relevance_score': 0.3,        # Query-document overlap
    'document_count': 0.2,         # Number of sources
    'semantic_similarity': 0.25,   # Conceptual alignment
    'source_authority': 0.15,      # Source credibility
    'answer_consistency': 0.1      # Response-document consistency
}
```

Calculation Example:

```
confidence_score = (
    0.85 * 0.3 +  # High relevance
    0.9 * 0.2 +   # 4 documents found
    0.75 * 0.25 + # Good semantic match
    0.8 * 0.15 +  # Authoritative sources
    0.7 * 0.1     # Consistent answer
) = 0.815  # "high" confidence
```

Confidence Levels:

- **High** (≥0.8): Strong evidence, multiple sources
- **Medium** (0.6-0.8): Reasonable coverage
- **Low** (0.4-0.6): Limited sources or weak relevance

- **Very Low** (<0.4): Insufficient information

Design Motivation ("Why")

Problem Solved:

- RAG systems can hallucinate or provide low-quality answers
- Users need to know when to trust AI responses
- System needs to know when to escalate to humans

Why This Approach:

- **Multi-factor assessment** - More robust than single metric
- **Weighted scoring** - Prioritizes most important factors
- **Actionable thresholds** - Different thresholds for different use cases

Trade-offs:

- ☑ Transparent confidence assessment
- ☑ Enables conditional logic (e.g., human handoff)
- ☑ Improves user trust
- ⚠ Heuristic-based (not ML-based)
- ⚠ Weights may need tuning per domain

Integration Points

Used in LCEL Chain:

```
# app/rag/lcel_chains.py
chain = (
    RunnablePassthrough.assign(
        confidence=self._calculate_confidence  # Confidence scoring
    )
    | prompt
    | llm
    | StrOutputParser()
)
```

Threshold-Based Actions:

```
# app/agents/conversational_agent.py
if confidence_score < 0.5:
    # Low confidence - request human handoff
    state["requires_handoff"] = True
```

Best Practices Applied

☑ **Multi-dimensional scoring** - Considers multiple quality factors ☑ **Source authority tracking** - Weights credible sources higher ☑ **Explainable scores** - Provides breakdown of factors ☑ **Use-case specific thresholds** - Different thresholds for different scenarios

Areas for Improvement

1. **ML-based scoring** - Train model to predict answer quality
2. **User feedback loop** - Learn from user ratings
3. **Embedding-based similarity** - Use actual cosine similarity instead of heuristics
4. **Calibration** - Ensure confidence scores match actual accuracy

---

7. LCEL Chain Orchestration

Concept Identification

Pattern: LangChain Expression Language (LCEL) for composable RAG pipelines

Implementation Details

Location: `app/rag/lcel_chains.py` - `ConfidenceRAGChain`

Chain Architecture:

```
chain = (
    # Step 1: Retrieve documents and calculate confidence
    RunnablePassthrough.assign(
        context=self._format_context,      # Retrieve & format docs
        confidence=self._calculate_confidence  # Score confidence
    )
    # Step 2: Generate response with prompt
    | prompt
    # Step 3: LLM generation
    | self.llm
    # Step 4: Parse output
    | StrOutputParser()
    # Step 5: Add metadata
    | RunnableLambda(self._add_metadata)
)
```

Context Formatting:

```
def _format_context(self, inputs: Dict[str, Any]) -> str:
    docs = self.retriever.get_relevant_documents(inputs["question"])

    context_parts = []
    for i, doc in enumerate(docs, 1):
        source = doc.metadata.get('source', f'doc_{i}')
        context_parts.append(f"[Source {i}: {source}]\n{doc.page_content}")
```

```
    return "\n\n".join(context_parts)
```

Design Motivation ("Why")

Problem Solved:

- Need composable, maintainable RAG pipelines
- Want to add steps (monitoring, scoring) without breaking flow
- Enable async execution for performance

Why This Approach:

- **LCEL syntax** - Declarative, readable pipeline definition
- **Composability** - Easy to add/remove steps
- **Async support** - Non-blocking execution
- **Type safety** - Better IDE support and error catching

Trade-offs:

- ☑ Clean, maintainable code
- ☑ Easy to extend with new steps
- ☑ Built-in async support
- ☑ Streaming support (if needed)
- ⚠ Learning curve for LCEL syntax
- ⚠ Debugging can be challenging

Integration Points

Used by Conversational Agent:

```
# app/agents/conversational_agent.py
self.rag_chain = get_confidence_rag_chain()

async def get_rag_response(self, query: str):
    return await self.rag_chain.ainvoke({"question": query})
```

Monitoring Integration:

```
# Automatic monitoring in chain
await record_rag_query(
    query=inputs["question"],
    response=result["answer"],
    confidence=result.get("confidence", 0.0),
    execution_time=elapsed_time,
    success=True
)
```

Best Practices Applied

☑ **Citation tracking** - Sources included in context ☑ **Metadata enrichment** - Timestamp, chain type added ☑ **Error handling** - Try-catch with monitoring ☑ **Async execution** - Non-blocking operations

Areas for Improvement

1. **Streaming responses** - Stream tokens as they're generated
2. **Caching** - Cache frequent queries
3. **Parallel retrieval** - Retrieve from multiple sources simultaneously
4. **Fallback chains** - Alternative chains if primary fails

---

8. Structured Knowledge Base (CRM Domain)

Concept Identification

Pattern: Hybrid RAG with structured knowledge + vector search

Implementation Details

Location: `app/core/crm_knowledge_base.py` - `CRMKnowledgeBase`

Structured Data:

```
crm_data = {
    "hubspot": {
        "name": "HubSpot CRM",
        "integration_complexity": "easy",
        "setup_time_minutes": 10,
        "oauth2_supported": True,
        "key_features": [...],
        "best_for": "Marketing teams focused on inbound strategies"
    },
    # ... more CRMs
}
```

Query Methods:

- `get_crm_info(crm_name)` - Direct lookup
- `find_answer(question)` - Keyword matching
- `get_crm_comparison(crm_list)` - Structured comparison
- `get_setup_checklist(crm_name)` - Procedural knowledge

Design Motivation ("Why")

Problem Solved:

- Frequently asked questions need instant, accurate answers
- Structured data (pricing, features) better stored in code than vectors

- Reduce latency and cost for common queries

Why This Approach:

- **Hybrid strategy** - Structured data for facts, RAG for complex queries
- **Zero latency** - No embedding or LLM call for direct lookups
- **Guaranteed accuracy** - Structured data doesn't hallucinate
- **Easy updates** - Change code vs. re-embedding documents

Trade-offs:

- ☑ Instant responses for common queries
- ☑ 100% accuracy for structured data
- ☑ No API costs for lookups
- ☑ Easy to maintain and update
- ⚠ Limited to predefined knowledge
- ⚠ Doesn't scale to large knowledge bases
- ⚠ Requires manual curation

Integration Points

Used by Conversational Agent:

```
# app/agents/conversational_agent.py
self.crm_kb = get_crm_knowledge_base()

# Try structured KB first, fallback to RAG
answer = self.crm_kb.find_answer(query)
if not answer:
    answer = await self.rag_chain.ainvoke({"question": query})
```

Best Practices Applied

☑ **Hybrid approach** - Structured + unstructured knowledge ☑ **Fast path optimization** - Direct lookup before expensive RAG ☑ **Domain-specific** - Tailored to CRM integration domain

Areas for Improvement

1. **Database storage** - Move to DB for easier updates
2. **Admin UI** - Allow non-technical updates
3. **Versioning** - Track changes to knowledge base
4. **A/B testing** - Compare structured vs. RAG answers

---

9. Intelligent LLM Routing

Concept Identification

Pattern: Multi-provider LLM routing with cost optimization and fallback

Implementation Details

Location: `app/llm/router.py` - `LLMRouter`

Provider Hierarchy:

```
fallback_order = [
    LLMProvider.GROK_2,           # Primary: Innovation + real-time
    LLMProvider.CLAUDE_SONNET,    # Secondary: Balance + safety
    LLMProvider.GPT_4,            # Tertiary: Complex reasoning
    LLMProvider.GPT_3_5_TURBO     # Fallback: Speed + cost
]
```

Routing Logic:

```
def _make_routing_decision(self, task_analysis):
    for provider in fallback_order:
        # Check API key availability
        if not config.api_key:
            continue

        # Check rate limits
        if not self._check_rate_limit(provider):
            continue

        # Select this provider
        return RoutingDecision(provider, reason, cost, latency)
```

Task Analysis Factors:

- Complexity score (length, keywords)
- Content type (code, math, research)
- Budget constraints
- Speed requirements

Design Motivation ("Why")

Problem Solved:

- Different tasks need different LLM capabilities
- Cost optimization across providers
- Resilience through fallback strategies
- Rate limit management

Why This Approach:

- **Cost optimization** - Route simple tasks to cheaper models
- **Capability matching** - Complex tasks to powerful models

- **Resilience** - Automatic fallback if provider unavailable
- **Rate limit handling** - Prevents API throttling

Trade-offs:

- ☑ Optimized cost per query
- ☑ High availability through fallbacks
- ☑ Automatic rate limit management
- ☑ Flexible provider configuration
- ⚠ Complexity in routing logic
- ⚠ Need to maintain multiple API keys
- ⚠ Inconsistent response quality across providers

Integration Points

Used Throughout System:

```
# app/agents/conversational_agent.py
llm = get_optimal_llm("Provide helpful conversational responses")

# app/rag/lcel_chains.py
llm = get_optimal_llm("Generate comprehensive answers with citations")
```

Monitoring:

```
logger.info(
    "LLM routing decision",
    provider=decision.provider.value,
    reason=decision.reason,
    estimated_cost=decision.estimated_cost
)
```

Best Practices Applied

☑ **Graceful degradation** - Fallback to available providers ☑ **Rate limit tracking** - Prevents API throttling
☑ **Cost awareness** - Tracks estimated costs ☑ **Observability** - Logs routing decisions

Areas for Improvement

1. **ML-based routing** - Learn optimal routing from performance data
2. **A/B testing** - Compare provider performance
3. **Cost tracking** - Actual cost monitoring and budgets
4. **Quality scoring** - Track response quality per provider
5. **Dynamic pricing** - Adjust routing based on real-time pricing

---

10. RAG Monitoring and Observability

Concept Identification

Pattern: Comprehensive monitoring of RAG system performance

Implementation Details

Location: `app/rag/monitoring.py` - `RAGMonitor`

Metrics Tracked:

```
@dataclass
class RAGQueryMetrics:
    query: str
    response: str
    confidence: float
    execution_time: float
    success: bool
    error: Optional[str]
    retriever_type: str
    num_docs_retrieved: int
    llm_provider: str
```

Performance Stats:

- Total queries
- Success rate
- Average execution time
- Average confidence score
- Min/max latency
- Error distribution

Health Status:

```
if success_rate >= 0.95 and avg_time <= 3.0 and confidence >= 0.7:
    status = "healthy"
elif success_rate >= 0.85 and avg_time <= 5.0:
    status = "degraded"
else:
    status = "unhealthy"
```

Design Motivation ("Why")

Problem Solved:

- Need visibility into RAG system performance
- Detect degradation before users complain
- Optimize based on real usage patterns
- Debug issues with specific queries

Why This Approach:

- **Rolling window** - Keep last 1000 queries for recent trends
- **Multi-dimensional metrics** - Latency, confidence, success rate
- **Health scoring** - Automated health assessment
- **Error tracking** - Categorize and count errors

Trade-offs:

- ☑ Real-time performance visibility
- ☑ Automated health checks
- ☑ Historical trend analysis
- ☑ Error categorization
- ⚠ Memory overhead for metrics storage
- ⚠ No persistent storage (lost on restart)

Integration Points

Automatic Recording:

```
# app/rag/lcel_chains.py
await record_rag_query(
    query=inputs["question"],
    response=result["answer"],
    confidence=result.get("confidence", 0.0),
    execution_time=elapsed_time,
    success=True
)
```

Health Checks:

```
# Can be exposed via API endpoint
health = get_rag_health_status()
# {"status": "healthy", "success_rate": 0.97, ...}
```

Best Practices Applied

☑ **Structured logging** - Using structlog for rich logs ☑ **Rolling window** - Prevents unbounded memory growth ☑ **Health scoring** - Automated alerting thresholds ☑ **Error categorization** - Track error types

Areas for Improvement

1. **Persistent storage** - Store metrics in database
2. **Alerting** - Send alerts when health degrades
3. **Dashboards** - Grafana/Datadog integration
4. **Query analysis** - Identify problematic query patterns
5. **Cost tracking** - Monitor API costs per query

11. Agent Orchestration with RAG

Concept Identification

Pattern: Conversational agent with RAG-augmented responses

Implementation Details

Location: `app/agents/conversational_agent.py` - `ConversationalAgent`

Agent Architecture:

```
class ConversationalAgent(BaseAgent):
    def __init__(self):
        self.llm = get_optimal_llm("conversational responses")
        self.rag_chain = get_confidence_rag_chain()
        self.crm_kb = get_crm_knowledge_base()
        self.conversation_contexts = {}  # Session tracking
```

Decision Flow:

```
User Query
    ↓
Intent Detection (crm_inquiry, integration_help, etc.)
    ↓
Qualification Scoring (CRM mentions, pain points, budget signals)
    ↓
Knowledge Retrieval
    ├→ Structured KB (fast path for common questions)
    └→ RAG Chain (complex queries)
    ↓
Response Generation (with context and qualification data)
    ↓
Handoff Decision (high-value leads → human agent)
```

Context Tracking:

```
context = {
    "session_id": session_id,
    "message_count": 5,
    "identified_crm": "hubspot",
    "qualification_score": 75.0,
    "pain_points": ["manual_processes", "data_silos"],
    "crm_interest_level": "high"
}
```

Design Motivation ("Why")

Problem Solved:

- Need intelligent, context-aware conversations
- Qualify leads while providing value
- Know when to escalate to humans
- Maintain conversation state across messages

Why This Approach:

- **Hybrid knowledge** - Structured + RAG for best of both
- **Intent detection** - Route to appropriate knowledge source
- **Qualification tracking** - Build lead profile during conversation
- **Handoff logic** - Escalate high-value leads automatically

Trade-offs:

- ☑ Intelligent, context-aware responses
- ☑ Automatic lead qualification
- ☑ Seamless human handoff
- ☑ Multi-turn conversation support
- ⚠ Complex state management
- ⚠ Memory overhead for session tracking
- ⚠ Potential for context drift in long conversations

Integration Points

Chat Service Integration:

```
# app/core/chat_service.py
response_data = await process_chat_message(
    session_id=session_id,
    user_message=content,
    conversation_history=history
)

# Update lead qualification
if session.lead_id:
    await self._update_lead_qualification(
        session.lead_id,
        response_data["analytics"]
    )
```

API Endpoint:

```
# app/api/v1/chat.py
@router.post("/message")
async def send_message(session_id: str, message: str):
```

```
    result = await chat_service.process_message(session_id, message)
    return result
```

Best Practices Applied

☑ **Stateful conversations** - Track context across messages ☑ **Intent classification** - Route to appropriate handlers ☑ **Lead qualification** - Extract business signals ☑ **Graceful handoff** - Escalate when appropriate ☑ **Analytics tracking** - Monitor conversation quality

Areas for Improvement

1. **Memory management** - Summarize old conversations
2. **Multi-modal support** - Handle images, documents
3. **Proactive suggestions** - Suggest next questions
4. **Sentiment analysis** - Detect frustration, urgency
5. **Personalization** - Adapt tone to user preferences

---

12. Agentic RAG with Tool Use and Reasoning

Concept Identification

Pattern: Agentic RAG with tool use, multi-step reasoning, and dynamic knowledge sources

Implementation Details

Current Status: Partial Implementation (Agentic RAG v0.8) **Missing Features**: Tool use / function calling, Multi-step reasoning (ReAct loop), Query decomposition, Self-reflection, Iterative refinement

Current Capabilities:

- Intent-based routing
- Multi-source knowledge (structured + RAG)
- Context accumulation
- Action-taking (handoff decisions)
- Qualification signal extraction

Architecture Overview:

```
class AgenticRAGAgent(BaseAgent):
    def __init__(self):
        self.llm = get_optimal_llm("agentic reasoning")
        self.rag_chain = get_confidence_rag_chain()
        self.crm_kb = get_crm_knowledge_base()
        self.tools = []  # ❌ MISSING: Tool registry
        self.reasoning_engine = None  # ❌ MISSING: ReAct loop
        self.memory = ConversationMemory()

    async def process_query(self, query: str, context: Dict):
        # Step 1: Intent detection and routing
```

```
        intent = await self._detect_intent(query)

        # Step 2: Knowledge retrieval (current implementation)
        knowledge = await self._gather_knowledge(query, intent)

        # Step 3: Reasoning and tool use (MISSING)
        # ❌ No multi-step reasoning
        # ❌ No tool selection
        # ❌ No query decomposition
        # ❌ No self-reflection

        # Step 4: Response generation
        response = await self._generate_response(knowledge, context)

        # Step 5: Action taking (partial)
        actions = await self._determine_actions(response, context)

        return {
            "response": response,
            "actions": actions,
            "confidence": knowledge.get("confidence", 0.0)
        }
```

Missing Components:

1. **Tool Use / Function Calling**
    - No tool registry or tool calling capabilities
    - Cannot execute external functions (APIs, calculations, data processing)
    - Limited to pre-defined knowledge sources
2. **Multi-step Reasoning (ReAct Loop)**
    - No iterative reasoning process
    - Cannot break down complex queries into sub-tasks
    - No "think → act → observe → reflect" cycle
3. **Query Decomposition**
    - Cannot split complex queries into simpler sub-queries
    - No ability to handle multi-part questions
    - Limited to single-turn reasoning
4. **Self-reflection**
    - No ability to evaluate own responses
    - Cannot identify knowledge gaps
    - No iterative refinement of answers
5. **Iterative Refinement**
    - Cannot improve answers based on feedback

    - No ability to ask clarifying questions
    - Limited to single-pass generation

Design Motivation ("Why")

Problem Solved:

- Current system is reactive, not proactive
- Cannot handle complex multi-step queries
- Limited to predefined knowledge and actions
- No dynamic tool selection or adaptation

Why Agentic RAG Matters:

- **Tool Use**: Access external APIs, perform calculations, manipulate data
- **Multi-step Reasoning**: Break down complex problems into manageable steps
- **Query Decomposition**: Handle multi-part questions effectively
- **Self-reflection**: Improve answer quality through iteration
- **Iterative Refinement**: Learn from interactions and feedback

Impact of Missing Features:

- ❌ Cannot handle complex multi-step queries
- ❌ No dynamic tool selection
- ❌ Limited to predefined knowledge sources
- ❌ Cannot verify or refine answers
- ❌ Reduced automation capabilities

Integration Points

Current Integration:

```
# app/agents/conversational_agent.py (current)
class ConversationalAgent(BaseAgent):
    def __init__(self):
        self.rag_chain = get_confidence_rag_chain()
        self.crm_kb = get_crm_knowledge_base()
        # Missing: tool registry, reasoning engine
```

Future Integration (Proposed):

```
# app/agents/agentic_rag_agent.py (proposed)
class AgenticRAGAgent(BaseAgent):
    def __init__(self):
        self.tools = [
            Tool(name="crm_lookup", func=self._crm_lookup),
            Tool(name="calculate_roi", func=self._calculate_roi),
            Tool(name="search_web", func=self._search_web),
            Tool(name="analyze_competitor", func=self._analyze_competitor)
```

```
        ]
        self.reasoning_engine = ReActAgent(
            llm=get_optimal_llm("reasoning"),
            tools=self.tools,
            memory=self.memory
        )
```

Best Practices Applied

☑ **Intent-based routing** - Routes queries to appropriate handlers ☑ **Context accumulation** - Maintains conversation state ☑ **Qualification extraction** - Identifies business signals ☑ **Action-taking logic** - Determines when to handoff

Areas for Improvement

1. **Tool Implementation**
    - Add tool registry with function calling
    - Implement common business tools (CRM lookup, ROI calculator, web search)
    - Add tool selection and execution logic
2. **Reasoning Engine**
    - Implement ReAct loop for multi-step reasoning
    - Add query decomposition capabilities
    - Enable self-reflection and iterative refinement
3. **Memory and Context**
    - Implement persistent memory across sessions
    - Add context summarization for long conversations
    - Enable cross-session learning
4. **Evaluation and Monitoring**
    - Track tool usage and success rates
    - Monitor reasoning quality and effectiveness
    - Add feedback loops for continuous improvement
5. **Safety and Reliability**
    - Add tool execution safety checks
    - Implement reasoning bounds and timeouts
    - Add human oversight for critical actions

Summary of RAG Design Decisions

Architecture Philosophy

Layered Approach:

1. **Storage Layer** - ChromaDB for vector storage
2. **Retrieval Layer** - Multiple strategies (dense, hybrid, compressed)
3. **Scoring Layer** - Confidence assessment
4. **Generation Layer** - LLM routing and response generation
5. **Orchestration Layer** - LCEL chains and agent logic
6. **Monitoring Layer** - Performance tracking and health checks

Key Design Principles

1. **Hybrid Knowledge** - Combine structured data with vector search
2. **Multi-Strategy Retrieval** - Ensemble, multi-query, compression
3. **Confidence-Driven** - Score and act on confidence levels
4. **Cost-Optimized** - Intelligent LLM routing
5. **Observable** - Comprehensive monitoring and logging
6. **Resilient** - Fallback strategies at every layer

Trade-off Analysis

Aspect	Choice	Benefit	Cost
Vector DB	ChromaDB (embedded)	Easy deployment, no separate server	Limited scalability vs. Pinecone/Weaviate
Embeddings	OpenAI	High quality, proven	API costs, vendor lock-in
Chunking	1000 chars + 200 overlap	Good balance	20% storage overhead
Retrieval	Ensemble (default)	Best recall+precision	More complex, slower
LLM Routing	Multi-provider fallback	Cost optimization, resilience	Complexity, inconsistency
Monitoring	In-memory metrics	Real-time, simple	Lost on restart
Knowledge	Hybrid (structured + RAG)	Fast + flexible	Maintenance overhead

Performance Characteristics

Typical Query Flow:

```
User Query (0ms)
    ↓
Intent Detection (5-10ms)
    ↓
Structured KB Lookup (1-2ms) OR
    ↓
Vector Search (50-100ms)
    ├─ Embedding generation (30-50ms)
    └─ Similarity search (20-50ms)
```

```
    ↓
Confidence Scoring (5-10ms)
    ↓
LLM Generation (1000-3000ms)
    ↓
Response (Total: 1-3 seconds)
```

Bottlenecks:

1. **LLM generation** - 70-90% of total time
2. **Embedding generation** - 10-20% of total time
3. **Vector search** - 5-10% of total time

Cost Analysis

Per Query Costs (estimated):

- Embedding generation: $0.0001 (100 tokens)
- Vector storage: Negligible (local)
- LLM generation: $0.002-0.02 (varies by provider)
- **Total**: ~$0.002-0.02 per query

Optimization Strategies:

1. Cache embeddings for common queries
2. Use cheaper models for simple queries (GPT-3.5)
3. Structured KB for frequent questions (zero cost)
4. Batch embedding generation

Concrete Example: End-to-End Flow

Scenario: User asks "How do I integrate HubSpot with Unitasa?"

Step 1: Intent Detection

```
# app/agents/conversational_agent.py
detected_intent = "integration_help"  # Keywords: integrate, hubspot
```

Step 2: Knowledge Retrieval

Option A: Structured KB (Fast Path)

```
# app/core/crm_knowledge_base.py
crm_info = crm_kb.get_crm_info("hubspot")
# Returns: {
#   "name": "HubSpot CRM",
#   "setup_time_minutes": 10,
```

```
#   "oauth2_supported": True,
#   ...
# }
```

Option B: RAG Chain (Complex Queries)

```
# app/rag/lcel_chains.py

# 1. Embed query
query_embedding = embed("How do I integrate HubSpot?")

# 2. Vector search
docs = vectorstore.similarity_search(query_embedding, k=4)
# Returns: [
#   Document("HubSpot integration guide..."),
#   Document("OAuth2 setup for HubSpot..."),
#   Document("Field mapping best practices..."),
#   Document("Common HubSpot integration issues...")
# ]

# 3. Calculate confidence
confidence = confidence_scorer.score_response(query, docs, "")
# Returns: 0.85 (high confidence)

# 4. Format context with citations
context = """
[Source 1: hubspot_integration_guide.md]
HubSpot integration guide...

[Source 2: oauth2_setup.md]
OAuth2 setup for HubSpot...
"""

# 5. Generate response
prompt = f"""Context: {context}
Question: {query}
Answer:"""

response = llm.invoke(prompt)
```

Step 3: Qualification Scoring

```
# app/agents/conversational_agent.py
qualification_updates = {
    "identified_crm": "hubspot",  # Detected from query
    "qualification_score": 65.0,  # +15 for CRM identification
    "crm_interest_level": "medium"
}
```

Step 4: Response Generation

```
response = """
Great question! Integrating HubSpot with Unitasa is straightforward and takes
about 10 minutes.

Here's how:
1. Navigate to Settings > Integrations in your Unitasa dashboard
2. Click "Connect HubSpot"
3. Authorize via OAuth2 (you'll be redirected to HubSpot)
4. Configure field mapping for contacts and deals
5. Test the connection with a sample sync

HubSpot supports bi-directional sync, so changes in either system will be
reflected automatically.

Would you like me to walk you through the field mapping configuration?

[Sources: HubSpot Integration Guide, OAuth2 Setup Documentation]
"""
```

Step 5: Handoff Decision

```
# Qualification score: 65 (medium)
# No handoff needed (threshold: 80 for high-value)
requires_handoff = False
```

Step 6: Monitoring

```
# app/rag/monitoring.py
await record_rag_query(
    query="How do I integrate HubSpot?",
    response=response,
    confidence=0.85,
    execution_time=1.2,  # seconds
    success=True,
    retriever_type="ensemble",
    llm_provider="grok_2"
)
```

Recommendations for Production

High Priority

1. **Persistent Monitoring**

- Store metrics in PostgreSQL or TimescaleDB
- Set up Grafana dashboards
- Configure alerts for degraded performance

2. **Embedding Caching**
   - Cache query embeddings (Redis)
   - Reduce API costs by 50-70%
   - Implement TTL-based invalidation
3. **Reranking Stage**
   - Add cross-encoder reranking after retrieval
   - Improves precision significantly
   - Models: `cross-encoder/ms-marco-MiniLM-L-12-v2`
4. **Evaluation Framework**
   - Create test set of query-answer pairs
   - Track metrics: accuracy, relevance, latency
   - A/B test retrieval strategies

Medium Priority

5. **Semantic Chunking**
   - Use embeddings to detect topic boundaries
   - Better than fixed-size chunks
   - Library: `semantic-text-splitter`
6. **Query Understanding**
   - Detect query type (factual, procedural, comparison)
   - Route to specialized retrievers
   - Use small classifier model
7. **Feedback Loop**
   - Collect user ratings (👍/👎)
   - Fine-tune confidence scorer
   - Identify knowledge gaps
8. **Multi-tenancy**
   - Separate collections per customer
   - Namespace-based isolation
   - Customer-specific knowledge bases

Low Priority

9. **Advanced Compression**

    - LLM-based document compression
    - Extractive summarization
    - Reduces context size by 70%+

10. **Streaming Responses**
    - Stream LLM tokens as generated
    - Better user experience
    - Perceived latency reduction

Conclusion

Your RAG implementation demonstrates **production-grade architecture** with:

☑ **Multiple retrieval strategies** for different use cases ☑ **Confidence scoring** for quality assurance ☑ **Intelligent LLM routing** for cost optimization ☑ **Comprehensive monitoring** for observability ☑ **Hybrid knowledge** combining structured and unstructured data ☑ **Agent orchestration** for conversational AI

Agentic RAG Status: v0.8 (Partial Implementation)

- ☑ Intent-based routing, Multi-source knowledge, Context accumulation, Action-taking, Qualification signal extraction
- ❌ Tool use / function calling, Multi-step reasoning (ReAct loop), Query decomposition, Self-reflection, Iterative refinement

The system is well-architected for a marketing intelligence platform, with clear separation of concerns and extensibility built in. The main areas for improvement are around **evaluation**, **caching**, **reranking**, and **agentic capabilities** to further optimize quality and cost.

Overall Assessment: 8.5/10 - Strong foundation with clear paths for optimization. Agentic RAG implementation at v0.8 requires completion of tool use, reasoning loops, and iterative refinement for full automation capabilities.