Advanced RAG Patterns Assessment

Overview

This document assesses the implementation status of three advanced RAG patterns in the Unitasa platform:

1. **Agentic RAG**
2. **Context-Aware RAG**
3. **Re-Ranking RAG**

---

1. Agentic RAG

☑ IMPLEMENTED (Partial)

What is Agentic RAG?

Agentic RAG uses autonomous agents that can:

- Make decisions about when and how to retrieve information
- Use tools and take actions based on retrieved context
- Reason about multiple retrieval steps
- Adapt retrieval strategy based on query complexity

Implementation Status in Your System

Location: `app/agents/conversational_agent.py`

☑ Implemented Features:

1. Intent-Based Routing

```
# app/agents/conversational_agent.py
def _detect_intent(self, message: str) -> str:
    """Detect user intent from message"""
    intent_patterns = {
        "crm_inquiry": ["crm", "customer relationship", ...],
        "integration_help": ["integrate", "connect", ...],
        "assessment_help": ["assessment", "questions", ...],
        "pricing_inquiry": ["price", "cost", ...],
    }
    # Routes to appropriate knowledge source
```

2. Multi-Source Knowledge Retrieval

```
class ConversationalAgent(BaseAgent):
    def __init__(self):
```

```python
        self.rag_chain = get_confidence_rag_chain()  # Vector RAG
        self.crm_kb = get_crm_knowledge_base()       # Structured KB

    async def get_rag_response(self, query: str):
        # Agent decides which knowledge source to use
        if self._is_common_question(query):
            return self.crm_kb.find_answer(query)  # Fast path
        else:
            return await self.rag_chain.ainvoke({"question": query})
```

3. Context Accumulation Across Turns

```python
# Maintains conversation state
self.conversation_contexts = {
    session_id: {
        "message_count": 5,
        "identified_crm": "hubspot",
        "qualification_score": 75.0,
        "pain_points": ["manual_processes"],
        "intent_history": ["crm_inquiry", "integration_help"]
    }
}
```

4. Action-Taking Based on Context

```python
def _should_request_handoff(self, context, response):
    """Agent decides to escalate to human"""
    if context.get("qualification_score", 0) >= 80:
        return True  # High-value lead action

    if "technical" in context.get("intent_history", []):
        return True  # Complex query action
```

5. Qualification Signal Extraction

```python
def _analyze_qualification_signals(self, message, context):
    """Agent extracts business intelligence from conversation"""
    # Identifies CRM systems
    # Detects pain points
    # Scores business maturity
    # Calculates qualification score
    return qualification_updates
```

❌ Missing Agentic Features:

1. Tool Use / Function Calling

- No explicit tool definitions for the agent
- Cannot call external APIs or databases dynamically
- No ReAct (Reasoning + Acting) loop

2. Multi-Step Reasoning

- No chain-of-thought prompting
- No iterative refinement of queries
- No self-reflection on retrieved results

3. Query Decomposition

- Cannot break complex queries into sub-queries
- No hierarchical question answering

Use Cases Where Agentic RAG Would Help

Use Case 1: Complex CRM Comparison Queries

Current Limitation:

```
User: "Compare HubSpot and Salesforce for a 50-person B2B SaaS company
       with $5M ARR, considering integration complexity, cost, and
       our existing tech stack (Stripe, Intercom, Slack)"
```

What Happens Now: Single RAG query, may miss nuances

With Full Agentic RAG:

```
# Agent would decompose into sub-tasks:
1. Retrieve HubSpot features and pricing
2. Retrieve Salesforce features and pricing
3. Check integration compatibility with Stripe, Intercom, Slack
4. Calculate TCO for 50-person team
5. Synthesize comparison based on all factors
```

Use Case 2: Dynamic CRM Health Checks

Current Limitation: Static knowledge base

With Agentic RAG:

```
# Agent could:
1. Query user's actual CRM via API
2. Retrieve best practices from knowledge base
3. Compare actual vs. best practices
4. Generate personalized recommendations
```

Use Case 3: Guided Troubleshooting

Current Limitation: One-shot answers

With Agentic RAG:

```
# Agent could:
1. Ask clarifying questions
2. Retrieve relevant docs based on answers
3. Test hypotheses
4. Iterate until solution found
```

Implementation Roadmap for Full Agentic RAG

Phase 1: Tool Integration (2-3 weeks)

```python
from langchain.agents import Tool, AgentExecutor
from langchain.agents.react.base import ReActDocstoreAgent

tools = [
    Tool(
        name="CRM_Knowledge_Search",
        func=lambda q: rag_chain.invoke({"question": q}),
        description="Search CRM integration knowledge base"
    ),
    Tool(
        name="CRM_API_Query",
        func=lambda params: query_crm_api(params),
        description="Query user's CRM system directly"
    ),
    Tool(
        name="Calculate_ROI",
        func=lambda data: calculate_roi(data),
        description="Calculate ROI for CRM investment"
    )
]

agent = AgentExecutor.from_agent_and_tools(
    agent=ReActDocstoreAgent.from_llm_and_tools(llm, tools),
    tools=tools,
    verbose=True
)
```

Phase 2: Query Decomposition (1-2 weeks)

```python
def decompose_query(complex_query: str) -> List[str]:
    """Break complex query into sub-queries"""
```

```
    prompt = f"""
    Break this complex question into 3-5 simpler sub-questions:
    {complex_query}

    Sub-questions:
    """
    return llm.invoke(prompt).split('\n')

# Then retrieve for each sub-query and synthesize
```

Phase 3: Self-Reflection (1 week)

```python
def reflect_on_answer(query: str, answer: str, docs: List[Document]):
    """Agent evaluates its own answer"""
    reflection_prompt = f"""
    Query: {query}
    Answer: {answer}
    Sources: {docs}

    Is this answer complete and accurate?
    What's missing?
    Should I retrieve more information?
    """
    return llm.invoke(reflection_prompt)
```

---

2. Context-Aware RAG

☑ IMPLEMENTED (Strong)

What is Context-Aware RAG?

Context-Aware RAG maintains and uses conversation history, user profile, and session state to:

- Understand follow-up questions
- Resolve pronouns and references
- Personalize responses based on user context
- Track conversation flow

Implementation Status in Your System

Location: Multiple files

☑ Implemented Features:

1. Conversation History Tracking

```
```

```python
# app/core/chat_service.py
def _get_conversation_history(self, session_id: int, limit: int = 20):
    """Get conversation history for a session"""
    messages = self.db.query(ChatMessage).filter(
        ChatMessage.session_id == session_id
    ).order_by(ChatMessage.timestamp.asc()).limit(limit).all()

    return [
        {
            "role": "user" if msg.sender == "user" else "assistant",
            "content": msg.content,
            "timestamp": msg.timestamp.isoformat(),
            "intent": msg.intent
        }
        for msg in messages
    ]
```

2. Session Context Management

```python
# app/agents/conversational_agent.py
context = {
    "session_id": session_id,
    "started_at": datetime.utcnow().isoformat(),
    "message_count": 5,
    "identified_crm": "hubspot",           # User's CRM
    "qualification_score": 75.0,           # Lead quality
    "intent_history": [...],               # Intent tracking
    "pain_points": ["manual_processes"],   # Extracted needs
    "crm_interest_level": "high"          # Engagement level
}
```

3. Context Injection into Prompts

```python
# app/agents/conversational_agent.py
def _format_context(self, context, conversation_history):
    """Format conversation context for prompt"""
    context_parts = [
        f"Session: {context.get('message_count', 0)} messages",
        f"Identified CRM: {context['identified_crm']}",
        f"Qualification Score: {score:.1f}/100",
        f"Pain Points: {', '.join(pain_points)}",
        "Recent conversation:",
        # Last 4 messages included
    ]
    return "\n".join(context_parts)
```

4. System Prompt with Context Variables

```
system_prompt = """
You are a helpful AI assistant for Unitasa...

Current conversation context: {context}
User's detected intent: {intent}
CRM interest level: {crm_interest}
"""
```

5. Lead Profile Enrichment

```python
# app/core/chat_service.py
async def _update_lead_qualification(self, lead_id, analytics):
    """Update lead profile based on conversation"""
    lead = self.db.query(Lead).filter(Lead.id == lead_id).first()

    # Update CRM system
    if identified_crm and not lead.current_crm_system:
        lead.current_crm_system = identified_crm

    # Update pain points
    combined_pain_points = list(set(
        existing_pain_points + new_pain_points
    ))

    # Update segment
    if chat_score >= 71:
        lead.readiness_segment = "hot"
```

6. Persistent Session Storage

```python
# app/models/chat_session.py
class ChatSession(Base):
    session_id = Column(String, unique=True)
    qualification_score = Column(Float, default=0.0)
    crm_interest_level = Column(String)
    identified_crm = Column(String)
    pain_points = Column(JSON)
    # ... stored in database
```

☑ Strong Implementation Score: 9/10

Your Context-Aware RAG is **very well implemented**. You have:

- ☑ Multi-turn conversation tracking
- ☑ Session state management
- ☑ User profile enrichment

- ☑ Context injection into prompts
- ☑ Persistent storage
- ☑ Intent tracking across turns

🔧 Minor Improvements Possible:

1. Conversation Summarization

```
# For very long conversations (>20 messages)
def summarize_old_context(messages: List[Dict]) -> str:
    """Summarize old messages to save context window"""
    if len(messages) > 20:
        old_messages = messages[:-10]  # All but last 10
        summary_prompt = f"Summarize this conversation: {old_messages}"
        summary = llm.invoke(summary_prompt)
        return summary + "\n\nRecent messages:\n" + format(messages[-10:])
```

2. Coreference Resolution

```
# Handle pronouns better
User: "Tell me about HubSpot"
Agent: "HubSpot is a CRM..."
User: "How much does it cost?"  # "it" = HubSpot

# Current: Works via conversation history
# Better: Explicit entity tracking
context["current_topic"] = "hubspot"
```

3. User Preference Learning

```
# Track user preferences over time
user_profile = {
    "preferred_communication_style": "technical",  # vs. business
    "detail_level": "comprehensive",  # vs. brief
    "topics_of_interest": ["integration", "automation"],
    "decision_stage": "evaluation"  # vs. awareness, purchase
}
```

Use Cases Where Context-Aware RAG Excels

Use Case 1: Follow-up Questions ☑ WORKING

```
User: "Tell me about HubSpot integration"
Agent: [Provides HubSpot info]
```

```
User: "How long does it take?"  # "it" = HubSpot integration
Agent: [Correctly understands context]
```

Use Case 2: Progressive Qualification ☑ WORKING

```
Turn 1: User mentions "Salesforce" → context.identified_crm = "salesforce"
Turn 2: User mentions "manual processes" → context.pain_points.append(...)
Turn 3: User mentions "enterprise" → context.qualification_score += 20
Turn 4: Agent offers enterprise demo (context-aware action)
```

Use Case 3: Personalized Recommendations ☑ WORKING

```python
# Agent knows user's CRM and pain points
if context["identified_crm"] == "hubspot":
    # Recommend HubSpot-specific integrations
if "data_silos" in context["pain_points"]:
    # Emphasize data unification features
```

---

3. Re-Ranking RAG

❌ NOT IMPLEMENTED

What is Re-Ranking RAG?

Re-Ranking RAG adds a second-stage ranking after initial retrieval:

1. **First stage**: Fast retrieval (semantic search, BM25) - recall-focused
2. **Second stage**: Precise reranking (cross-encoder) - precision-focused

This two-stage approach balances speed and accuracy.

Why Re-Ranking Matters

Problem with Current System:

```python
# Current: Single-stage retrieval
docs = vectorstore.similarity_search(query, k=5)
# These 5 docs go directly to LLM
```

Issues:

- Bi-encoder (used in embeddings) optimizes for recall, not precision
- May include marginally relevant documents
- No fine-grained relevance scoring

- Order matters for LLM context

With Re-Ranking:

```
# Stage 1: Fast retrieval (recall)
candidate_docs = vectorstore.similarity_search(query, k=20)

# Stage 2: Precise reranking (precision)
reranked_docs = cross_encoder.rerank(query, candidate_docs, top_k=5)

# Top 5 are much more relevant
```

Implementation Status: ❌ NOT FOUND

Searched in:

- `app/rag/advanced_retrievers.py` - No reranking
- `app/rag/lcel_chains.py` - No reranking stage
- `app/rag/confidence_scorer.py` - Scores but doesn't rerank

Current Retrieval Flow:

```
Query → Embed → Vector Search (k=4-5) → LLM
```

Missing:

```
Query → Embed → Vector Search (k=20) → Rerank → Top 5 → LLM
```

Use Cases Where Re-Ranking Would Help

Use Case 1: Ambiguous Queries

Query: "integration setup"

Current (No Reranking):

```
Retrieved docs (by cosine similarity):
1. "HubSpot integration setup" (0.82 similarity)
2. "Salesforce integration overview" (0.80 similarity)
3. "API integration best practices" (0.79 similarity)
4. "Zapier integration guide" (0.78 similarity)
5. "Integration security considerations" (0.77 similarity)
```

With Reranking:

```
After cross-encoder reranking:
1. "HubSpot integration setup" (0.95 relevance) ☑ Much better
2. "Salesforce integration setup" (0.93 relevance) ☑ More specific
3. "Integration setup checklist" (0.89 relevance) ☑ More relevant
4. "API integration best practices" (0.65 relevance)
5. "Integration security" (0.58 relevance)
```

Use Case 2: Long Documents

Problem: Bi-encoders struggle with long documents

Current: May retrieve chunks that mention keywords but aren't truly relevant

With Reranking: Cross-encoder reads full query + document, better understanding

Use Case 3: Nuanced Questions

Query: "What's the difference between OAuth2 and API key authentication for CRM integration?"

Current: May retrieve docs about OAuth2 OR API keys separately

With Reranking: Prioritizes docs that compare both methods

Implementation Guide for Re-Ranking

Option 1: Cross-Encoder Reranking (Recommended)

Step 1: Install Dependencies

```
pip install sentence-transformers
```

Step 2: Create Reranker Class

```python
# app/rag/reranker.py
from sentence_transformers import CrossEncoder
from typing import List, Tuple
from langchain_core.documents import Document

class CrossEncoderReranker:
    """Rerank documents using cross-encoder model"""

    def __init__(self, model_name: str = "cross-encoder/ms-marco-MiniLM-L-12-v2"):
        self.model = CrossEncoder(model_name)

    def rerank(
        self,
        query: str,
        documents: List[Document],
```

```
        top_k: int = 5
    ) -> List[Tuple[Document, float]]:
        """Rerank documents by relevance to query"""

        # Prepare pairs for cross-encoder
        pairs = [[query, doc.page_content] for doc in documents]

        # Get relevance scores
        scores = self.model.predict(pairs)

        # Sort by score (descending)
        doc_score_pairs = list(zip(documents, scores))
        doc_score_pairs.sort(key=lambda x: x[1], reverse=True)

        # Return top_k
        return doc_score_pairs[:top_k]
```

Step 3: Integrate into Retrieval Chain

```
# app/rag/advanced_retrievers.py
class ReRankingRetriever(BaseRetriever):
    """Retriever with reranking stage"""

    def __init__(self, vectorstore: VectorStore, reranker: CrossEncoderReranker):
        super().__init__()
        self.vectorstore = vectorstore
        self.reranker = reranker

    def _get_relevant_documents(
        self, query: str, *, run_manager: CallbackManagerForRetrieverRun
    ) -> List[Document]:
        """Retrieve and rerank documents"""

        # Stage 1: Retrieve more candidates (recall)
        candidate_docs = self.vectorstore.similarity_search(query, k=20)

        # Stage 2: Rerank for precision
        reranked_docs_with_scores = self.reranker.rerank(
            query,
            candidate_docs,
            top_k=5
        )

        # Extract documents (optionally store scores in metadata)
        reranked_docs = []
        for doc, score in reranked_docs_with_scores:
            doc.metadata['rerank_score'] = float(score)
            reranked_docs.append(doc)

        return reranked_docs
```

Step 4: Update LCEL Chain

```
# app/rag/lcel_chains.py
class ConfidenceRAGChain:
    def __init__(self):
        # Add reranker
        from app.rag.reranker import CrossEncoderReranker
        self.reranker = CrossEncoderReranker()

        # Use reranking retriever
        self.retriever = ReRankingRetriever(
            vectorstore=get_vector_store(),
            reranker=self.reranker
        )
```

Step 5: Monitor Reranking Impact

```
# app/rag/monitoring.py
@dataclass
class RAGQueryMetrics:
    # Add reranking metrics
    reranking_time: float = 0.0
    avg_rerank_score: float = 0.0
    score_improvement: float = 0.0  # vs. initial retrieval
```

Option 2: LLM-based Reranking (More Expensive)

```
class LLMReranker:
    """Rerank using LLM to score relevance"""

    def __init__(self, llm):
        self.llm = llm

    def rerank(self, query: str, documents: List[Document], top_k: int = 5):
        """Score each document with LLM"""
        scored_docs = []

        for doc in documents:
            prompt = f"""
            Query: {query}
            Document: {doc.page_content[:500]}

            Rate relevance (0-10):
            """
            score = float(self.llm.invoke(prompt))
            scored_docs.append((doc, score))

        # Sort and return top_k
```

```
        scored_docs.sort(key=lambda x: x[1], reverse=True)
        return scored_docs[:top_k]
```

Option 3: Cohere Rerank API (Easiest)

```python
import cohere

class CohereReranker:
    """Use Cohere's rerank API"""

    def __init__(self, api_key: str):
        self.client = cohere.Client(api_key)

    def rerank(self, query: str, documents: List[Document], top_k: int = 5):
        """Rerank using Cohere API"""

        # Prepare documents
        docs_text = [doc.page_content for doc in documents]

        # Call Cohere rerank
        results = self.client.rerank(
            query=query,
            documents=docs_text,
            top_n=top_k,
            model="rerank-english-v2.0"
        )

        # Map back to Document objects
        reranked_docs = []
        for result in results:
            doc = documents[result.index]
            doc.metadata['rerank_score'] = result.relevance_score
            reranked_docs.append(doc)

        return reranked_docs
```

Performance Impact of Reranking

Latency:

- Cross-encoder: +50-100ms (local model)
- Cohere API: +100-200ms (API call)
- LLM-based: +500-1000ms (expensive)

Accuracy Improvement:

- Typical: +10-20% in answer quality
- Complex queries: +30-40% improvement
- Simple queries: Minimal difference

Cost:

- Cross-encoder: Free (local model)
- Cohere: ~$0.002 per 1000 documents
- LLM-based: ~$0.01-0.05 per query

Recommended Implementation Priority

Phase 1: Cross-Encoder Reranking (HIGH PRIORITY)

- **Effort**: 1-2 days
- **Impact**: High (10-20% quality improvement)
- **Cost**: Free (local model)
- **Use**: Default for all queries

Phase 2: Adaptive Reranking (MEDIUM PRIORITY)

- **Effort**: 1 day
- **Impact**: Medium (cost optimization)
- **Logic**: Only rerank when confidence is low

```python
if initial_confidence < 0.7:
    docs = reranker.rerank(query, candidate_docs)
else:
    docs = candidate_docs[:5]  # Skip reranking
```

Phase 3: Hybrid Reranking (LOW PRIORITY)

- **Effort**: 2-3 days
- **Impact**: Medium (best of both worlds)
- **Logic**: Combine multiple reranking signals

```python
final_score = (
    0.4 * cross_encoder_score +
    0.3 * bm25_score +
    0.2 * embedding_similarity +
    0.1 * source_authority
)
```

Summary Scorecard

Pattern	Status	Score	Priority	Effort
Agentic RAG	◍ Partial	6/10	Medium	4-6 weeks
Context-Aware RAG	☑ Strong	9/10	Low	1 week (polish)

Pattern	Status	Score	Priority	Effort
Re-Ranking RAG	❌ Missing	0/10	**HIGH**	1-2 days

Detailed Breakdown

1. Agentic RAG: 6/10 (Partial Implementation)

What You Have ✅:

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

What's Missing ❌:

- Tool use / function calling
- Multi-step reasoning (ReAct loop)
- Query decomposition
- Self-reflection
- Iterative refinement

Impact of Missing Features:

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

Recommendation:

- **Priority**: Medium (nice-to-have for complex queries)
- **Quick Win**: Add query decomposition (1 week)
- **Full Implementation**: 4-6 weeks for complete agentic system

---

2. Context-Aware RAG: 9/10 (Strong Implementation)

What You Have ✅:

- Conversation history tracking (20 messages)
- Session state management
- User profile enrichment
- Context injection into prompts
- Persistent storage (database)
- Intent tracking across turns
- Lead qualification accumulation

What's Missing 🔧:

- Conversation summarization (for very long chats)
- Explicit coreference resolution
- User preference learning

Impact of Missing Features:

- Minor: Long conversations may exceed context window
- Minor: Pronoun resolution relies on LLM (works but not optimal)

Recommendation:

- **Priority**: Low (already excellent)
- **Polish**: Add conversation summarization (1 week)
- **Enhancement**: User preference tracking (1 week)

---

3. Re-Ranking RAG: 0/10 (Not Implemented)

What You Have ✅:

- Nothing (no reranking stage)

What's Missing ❌:

- Cross-encoder reranking
- Second-stage precision ranking
- Relevance score refinement

Impact of Missing Features:

- **HIGH**: Suboptimal document selection
- **HIGH**: Lower answer quality for ambiguous queries
- **MEDIUM**: Wasted LLM context on marginally relevant docs

Recommendation:

- **Priority**: **HIGH** (biggest ROI for effort)
- **Quick Win**: Cross-encoder reranking (1-2 days)
- **Expected Impact**: +10-20% answer quality improvement

Implementation Roadmap

Week 1: Re-Ranking RAG (HIGH PRIORITY)

Effort: 1-2 days **Impact**: High

```
# Day 1: Implement cross-encoder reranker
from sentence_transformers import CrossEncoder
```

```
class CrossEncoderReranker:
    def __init__(self):
        self.model = CrossEncoder('cross-encoder/ms-marco-MiniLM-L-12-v2')

    def rerank(self, query, docs, top_k=5):
        pairs = [[query, doc.page_content] for doc in docs]
        scores = self.model.predict(pairs)
        # Sort and return top_k
        return sorted(zip(docs, scores), key=lambda x: x[1], reverse=True)[:top_k]

# Day 2: Integrate into retrieval chain
class ReRankingRetriever(BaseRetriever):
    def _get_relevant_documents(self, query):
        candidates = self.vectorstore.similarity_search(query, k=20)
        reranked = self.reranker.rerank(query, candidates, top_k=5)
        return [doc for doc, score in reranked]
```

Testing:

```
# Compare with/without reranking
queries = [
    "How to integrate HubSpot?",
    "What's the difference between OAuth2 and API key?",
    "CRM setup for enterprise"
]

for query in queries:
    # Without reranking
    docs_no_rerank = vectorstore.similarity_search(query, k=5)

    # With reranking
    docs_reranked = reranking_retriever.get_relevant_documents(query)

    # Compare quality
    print(f"Query: {query}")
    print(f"No rerank: {[d.metadata['source'] for d in docs_no_rerank]}")
    print(f"Reranked: {[d.metadata['source'] for d in docs_reranked]}")
```

Week 2-3: Context-Aware Polish (LOW PRIORITY)

Effort: 1 week **Impact**: Medium

```
# Conversation summarization
def summarize_conversation(messages):
    if len(messages) > 20:
        old_messages = messages[:-10]
        summary = llm.invoke(f"Summarize: {old_messages}")
        return summary + "\n\nRecent:\n" + format(messages[-10:])
    return format(messages)
```

```
# User preference tracking
class UserPreferenceTracker:
    def learn_preferences(self, user_id, interactions):
        # Analyze communication style
        # Track topics of interest
        # Identify decision stage
        pass
```

Week 4-9: Agentic RAG Enhancement (MEDIUM PRIORITY)

Effort: 4-6 weeks **Impact**: High (for complex queries)

Phase 1: Query Decomposition (Week 4)

```
def decompose_complex_query(query):
    if is_complex(query):
        sub_queries = llm.invoke(f"Break into sub-questions: {query}")
        results = [rag_chain.invoke(q) for q in sub_queries]
        return synthesize(results)
    return rag_chain.invoke(query)
```

Phase 2: Tool Integration (Week 5-6)

```
from langchain.agents import Tool, AgentExecutor

tools = [
    Tool(name="RAG_Search", func=rag_chain.invoke),
    Tool(name="CRM_API", func=query_crm_api),
    Tool(name="Calculate_ROI", func=calculate_roi)
]

agent = AgentExecutor.from_agent_and_tools(
    agent=ReActAgent.from_llm_and_tools(llm, tools),
    tools=tools
)
```

Phase 3: Self-Reflection (Week 7-8)

```
def reflect_and_refine(query, answer, docs):
    reflection = llm.invoke(f"""
    Query: {query}
    Answer: {answer}

    Is this complete? What's missing?
    """)
```

```
    if "incomplete" in reflection.lower():
        # Retrieve more information
        additional_docs = retrieve_more(query, reflection)
        # Regenerate answer
        return generate_improved_answer(query, docs + additional_docs)

    return answer
```

Expected ROI by Implementation

Re-Ranking RAG (Week 1)

- **Effort**: 1-2 days
- **Cost**: $0 (local model)
- **Latency**: +50-100ms
- **Quality Improvement**: +10-20%
- **ROI**: ★★★★★ (Excellent)

Context-Aware Polish (Week 2-3)

- **Effort**: 1 week
- **Cost**: Minimal
- **Latency**: Negligible
- **Quality Improvement**: +5%
- **ROI**: ★★★ (Good)

Agentic RAG (Week 4-9)

- **Effort**: 4-6 weeks
- **Cost**: Higher LLM usage
- **Latency**: +500-1000ms
- **Quality Improvement**: +20-30% (complex queries only)
- **ROI**: ★★★ (Good for specific use cases)

Conclusion

Current State

Your RAG system is **production-ready** with:

- ☑ Excellent context-aware capabilities (9/10)
- ◍ Partial agentic features (6/10)
- ✕ Missing reranking (0/10)

Recommended Action Plan

Immediate (This Week):

1. ☑ Implement cross-encoder reranking (1-2 days, huge ROI)
2. ☑ Test reranking on sample queries
3. ☑ Monitor quality improvement

Short-term (Next Month):

1. Polish context-aware features (conversation summarization)
2. Add query decomposition for complex queries
3. Implement adaptive reranking (skip for simple queries)

Long-term (Next Quarter):

1. Full agentic RAG with tool use
2. Self-reflection and iterative refinement
3. Multi-step reasoning capabilities

Final Assessment

Overall RAG Maturity: 7.5/10

You have a **strong foundation** with excellent context-awareness. The **highest ROI improvement** is adding reranking (1-2 days for +10-20% quality). Agentic features are nice-to-have but not critical for your current use cases.

Priority Order:

1. 🔴 **Re-Ranking** (1-2 days, high impact)
2. 🟡 **Context Polish** (1 week, medium impact)
3. 🟢 **Agentic Enhancement** (4-6 weeks, specific use cases)