Design Brief - ArXiv RAG Pipeline

Project: CS6300 Assignment 5 - ArXiv RAG Pipeline
Date: October 2024
Repository: GitHub Repository

PEAS Analysis

Performance Measure

- Successfully fetching academic papers from arXiv API
- Correctly populating ChromaDB with paper embeddings
- Accurate context retrieval for user queries
- Natural conversational responses using retrieved context
- End-to-end success: user query → retrieval → LLM response

Tool-Level Success Metrics

- **ArXiv Fetching**: 12+ papers retrieved with abstracts and full text
- **Vector DB Population**: All papers encoded and stored with metadata
- **RAG Retrieval**: Top-5 relevant papers retrieved for each query
- **Chat Management**: Conversation history maintained across sessions

Environment

- ArXiv API for academic paper discovery
- ChromaDB vector database for paper storage
- LM Studio local LLM server
- User research questions and meeting preparation needs
- Local file system for chat history and reports

Actuators (Tools)

1. **ArXiv Fetcher** - Searches and retrieves papers with PDF parsing
2. **Vector DB Populator** - Encodes and stores papers in ChromaDB
3. **RAG Engine** - Retrieves relevant context and manages caching
4. **Chat Manager** - Maintains conversation history and sessions

Sensors

- ArXiv API responses for paper metadata
- PDF parsing for full paper text
- Sentence transformers for text embeddings
- LLM responses via LM Studio API
- File system for chat history persistence

Environment Properties Analysis

Observable vs. Partially Observable: - **Partially Observable** - The agent cannot observe all available papers on arXiv, user's actual meeting context, complete paper content, or future paper releases - **Fully Observable** - The agent has complete observability of retrieved paper content, vector database contents, conversation history, and all tool inputs/outputs

Deterministic vs. Stochastic: - **Stochastic** - LLM responses vary, ArXiv search results can vary, paper availability may change, vector similarity scores can vary slightly - **Deterministic** - ArXiv API responses, vector encoding operations, file system operations, and tool execution follow consistent logic

Episodic vs. Sequential: - **Sequential** - Tools build on each other, conversation history influences future responses, context retrieval depends on previous database population, chat sessions maintain state across interactions

Static vs. Dynamic: - **Dynamic** - New papers are constantly added to arXiv, user research interests evolve, vector database grows, conversation context accumulates - **Static** - ArXiv API interface, ChromaDB storage structure, tool interfaces, and local file system structure remain constant

Discrete vs. Continuous: - **Discrete** - All actions and states are discrete: tool calls, paper retrieval operations, success/failure states, conversation turns, vector similarity scores, chat sessions

Tool Specifications

Tool 1: ArXiv Fetcher

Name: `arxiv_fetcher`

Description: Searches arXiv API for academic papers on specified topics, parses Atom XML responses, and extracts full paper content including abstracts and PDF text. Handles pagination and rate limiting while providing structured paper data for vector database population.

Inputs: - `topic` (string): Research topic to search for (e.g., 'neural networks', 'machine learning') - `max_papers` (int, optional): Maximum number of papers to retrieve (default: 12)

Outputs: - List of dictionaries containing: `id`, `title`, `abstract`, `authors`, `categories`, `text`, `pdf_url`, `published`, `source`

Error Handling: - **API Errors**: ArXiv API unavailability, rate limiting, malformed responses - **PDF Processing Errors**: PDF download failures, parsing errors, text extraction issues - **Content Errors**: Empty abstracts, inaccessible PDFs, encoding problems - **Network Errors**: Connection timeouts, SSL certificate issues, download failures

Tool 2: Vector DB Populator

Name: `vector_db_populator`

Description: Initializes ChromaDB collections, encodes paper text using sentence-transformers, and populates vector database with embeddings and metadata. Supports similarity search with metadata filtering and provides abstraction for different vector database backends.

Inputs: - `texts` (list): List of paper texts to encode and store - `metadata` (list): List of metadata dictionaries for each paper - `collection_name` (string): Name for the ChromaDB collection

Outputs: - Boolean success indicator, collection statistics, encoder initialization confirmation

Error Handling: - **Database Errors**: ChromaDB connection failures, collection creation errors - **Encoding Errors**: Sentence transformer failures, text preprocessing issues - **Metadata Errors**: Invalid metadata format, type conversion failures - **Storage Errors**: Disk space issues, permission problems, corruption

Tool 3: RAG Engine

Name: `rag_engine`

Description: Retrieves top-k relevant papers for user queries, implements context caching for performance, builds prompts with retrieved context and conversation history, and manages context window to prevent LLM overflow. Provides intelligent truncation and dynamic prompt sizing.

Inputs: - `user_query` (string): User's question or request - `conversation_history` (list, optional): Previous conversation messages - `top_k` (int, optional): Number of relevant papers to retrieve (default: 5)

Outputs: - Tuple containing: `response` (string), `retrieved_context` (list), context truncation warnings if needed

Error Handling: - **Retrieval Errors**: Vector database failures, similarity search errors - **Context Errors**: Context window overflow, prompt truncation issues - **LLM Errors**: API failures, response parsing errors, timeout issues - **Cache Errors**: Cache corruption, TTL expiration, memory issues

Tool 4: Chat Manager

Name: `chat_manager`

Description: Creates and manages chat sessions, maintains conversation history across interactions, saves chat sessions to JSON files, and loads previous conversations for context. Provides session persistence and conversation state management.

Inputs: - `session_id` (string, optional): Existing session ID or None for new session - `topic` (string): Topic for the chat session - `message` (dict): Message to add with role and content

Outputs: - Session object with: `session_id`, `conversation_history`, `session_metadata`, `file_path`

Error Handling: - **Session Errors**: Session creation failures, ID conflicts - **File Errors**: JSON serialization failures, file write errors - **History Errors**: Message format validation, conversation corruption - **Storage Errors**: Directory creation failures, permission issues

Agent Architecture

Framework Choice

- **Custom Python Implementation**: Built with smolagents-compatible structure for tool integration
- Provides clean separation between population and conversation phases
- Supports both scripted orchestration and model-driven responses
- Minimal overhead for rapid prototyping and testing

Model Configuration

- **Model**: `qwen/qwen3-4b-2507` (local deployment via LM Studio)
- **Endpoint**: `http://localhost:1234/v1` (local LLM server)
- **Rationale**: Chosen for cost efficiency, privacy, and rapid iteration without cloud credits
- **API Compatibility**: OpenAI-compatible interface for seamless integration
- **Context Management**: Intelligent truncation and dynamic prompt sizing

Two-Flow Architecture

Flow 1: Population Pipeline

```
User Topic → ArXiv Fetcher → Papers (title, abstract, PDF text) →
Vector DB Populator → Encode with sentence-transformers →
Store in ChromaDB → Ready for queries
```

Flow 2: Conversation Pipeline

```
User Question → RAG Engine (retrieve top-k papers) →
Build prompt (context + history) → LLM Studio API →
Response with citations → Chat Manager (save) → Continue conversation
```

Context Management

- **Intelligent Truncation**: 1000 characters per message limit

- **Reduced Context Window**: 2 messages maximum in conversation history
- **Dynamic Prompt Sizing**: Estimates total prompt length and truncates proactively
- **Overflow Prevention**: Graceful handling of context window overflow errors
- **Cache Management**: Context caching for improved performance

Orchestration Strategy

- **Sequential Tool Execution**: Tools execute in predetermined order for each flow
- **State Management**: Each tool reads and modifies the current system state
- **Flow Separation**: Population and conversation phases are independent
- **Error Handling**: Failed operations are logged but don't stop the pipeline
- **Session Persistence**: Chat history maintained across interactions

Key Design Decisions

- **Local LLM**: Chosen for cost efficiency, privacy, and rapid iteration
- **Two-Phase Architecture**: Clear separation between data preparation and usage
- **Context-Aware Responses**: RAG retrieval ensures relevant, cited responses
- **Conversation Continuity**: Chat history enables natural multi-turn interactions
- **Intelligent Caching**: Context caching improves performance and reduces API calls
- **Graceful Degradation**: System continues working even if individual components fail

Evaluation Plan

Test Strategy

The evaluation plan uses a unified test runner with multiple scenarios to validate both individual tool performance and integrated RAG pipeline behavior. Tests are designed to validate the two-flow architecture: population pipeline and conversation pipeline.

Test Framework

- **Unified Test Runner**: `scripts/test_runner.py` with extensible scenario support
- **Multiple Scenarios**: Poignant prompts, conversation flow, and smart person simulation

- **Comprehensive Metrics**: Success rates, response times, context retrieval accuracy
- **Error Handling**: Context overflow management and graceful degradation

Test Scenarios

1. Poignant Prompts Test Purpose: Test focused, direct questions on complex topics - **Queries**: 10 targeted questions about algorithms, results, datasets, limitations - **Success Criteria**: >80% query success rate, relevant context retrieval - **Command**: `python3 scripts/test_runner.py --scenario poignant --topic "neural networks" --max-papers 12`

2. Conversation Test Purpose: Test conversational flow that builds understanding - **Queries**: 8 follow-up questions that build on previous responses - **Success Criteria**: Conversation coherence, context accumulation, history management - **Command**: `python3 scripts/test_runner.py --scenario conversation --topic "neural networks" --max-papers 12`

3. Smart Person Test Purpose: Test meeting preparation with technical jargon - **Queries**: 8 questions about buzzwords, methodologies, performance metrics - **Success Criteria**: Technical terminology usage, impressive metrics citation - **Command**: `python3 scripts/test_runner.py --scenario smart-person --topic "deep learning" --max-papers 12`

Success Metrics

- **Query Success Rate**: >80% for all scenarios
- **Response Time**: <20 seconds per query on average
- **Context Retrieval**: Top-5 relevant papers retrieved per query
- **Conversation Coherence**: Natural flow across multiple turns
- **Cache Utilization**: Efficient context caching and reuse
- **Context Management**: 100% recovery from overflow errors

Expected Behaviors

- All scenarios complete within 3 minutes total
- Success rate >80% for valid research queries
- Graceful degradation when individual tools fail
- Consistent response format with source citations
- Intelligent context management preventing overflow errors

This design brief provides the complete foundation for the ArXiv RAG Pipeline, covering PEAS analysis, tool specifications, agent architecture, and evaluation plan as required for the assignment.