# ALBERT: Agent Learning through Building Embedded RAG Tools - A Practical Exploration of LangChain Development

Ruben Parra Montenegro
*Department of Computer Science*
*Oakland University*
Rochester, Michigan
parramontenegro@oakland.edu

***Abstract*—This paper presents Albert, a command-line AI agent built using LangChain framework with Retrieval-Augmented Generation (RAG) capabilities. The system addresses the challenge of creating an accessible, extensible agent platform that integrates document processing, web scraping, vector storage, and conversational memory while maintaining security through sandboxing. Albert demonstrates how open-source tools can be combined to build a functional agent using only free-tier API services. The implementation showcases practical solutions to common challenges in agent development including tool integration, prompt engineering, and semantic search. Key results include successful PDF-to-markdown conversion, functional RAG implementation achieving semantic retrieval from converted documents, and a modular architecture enabling easy tool extension.**

## I. Introduction

The rapid advancement of Large Language Models (LLMs) has created opportunities for building intelligent agents capable of complex task execution. However, developing practical agent systems remains challenging, particularly for developers new to frameworks like LangChain. This project addresses the need for accessible, educational examples of agent development by creating Albert—a functional CLI agent that demonstrates core concepts including tool calling, memory management, and retrieval-augmented generation.

The primary motivation was to understand how modern AI agents function at a fundamental level. Rather than using pre-built solutions, this project required implementing the full pipeline: from tool design and vector embeddings to prompt engineering and security considerations. The goal was to create a system that is simultaneously educational (for learning agent architecture) and practical (capable of real document processing tasks).

Albert tackles several key challenges: (1) integrating multiple specialized tools (web search, scraping, PDF processing) into a cohesive agent, (2) implementing RAG to enable question-answering over user documents, (3) maintaining conversation context across interactions, and (4) ensuring security through proper sandboxing. The system is designed to work entirely with free-tier APIs, making it accessible for experimentation and learning.

## II. Related Work

### A. Agent Frameworks

LangChain [1] has emerged as a leading framework for building LLM-powered applications, providing abstractions for tool calling, memory management, and chain composition. AutoGPT [2] demonstrates fully autonomous agent behavior but requires more computational resources. Albert differs by prioritizing simplicity and educational value over autonomy.

### B. Document Processing

Traditional PDF extraction tools like PyPDF2 and pdfminer struggle with complex layouts. Docling [3] represents a newer approach, using deep learning to preserve document structure including tables and formatting. Albert leverages Docling specifically for its ability to maintain semantic structure during conversion to markdown.

### C. Retrieval-Augmented Generation

RAG systems [4] combine parametric knowledge (LLM) with non-parametric knowledge (retrieved documents). ChromaDB [5] provides an efficient local vector store, while Sentence-Transformers [6] offers open-source embedding models. Albert implements a standard RAG pipeline but focuses on user document processing rather than general knowledge retrieval.

### D. Web Scraping Agents

Playwright [7] enables browser automation for JavaScript-heavy sites, surpassing tools like BeautifulSoup for dynamic content. Combined with search APIs like SerpAPI, agents can autonomously research topics. Albert integrates both components but maintains human oversight rather than full autonomy.

## III. Data

### A. User Documents

The primary data consists of user-provided PDF documents that are converted to markdown format. Test cases included technical resumes. No external datasets were required as the system processes user-uploaded content.

### B. Preprocessing

PDF documents undergo the following pipeline:

1) **Conversion**: Docling extracts text, tables, and structure from PDFs while preserving semantic relationships
2) **Chunking**: RecursiveCharacterTextSplitter divides markdown into 1000-character chunks with 200-character overlap to maintain context
3) **Embedding**: Sentence-Transformers (all-MiniLM-L6-v2 model) generates 384-dimensional dense vectors for each chunk
4) **Storage**: ChromaDB indexes embeddings with metadata (source filename, chunk number)

### C. Sandboxing

All file operations occur within a configurable sandbox directory to prevent unauthorized file system access. Path traversal attacks are mitigated through `os.path.basename()`.

## IV. Methods

### A. Agent Architecture

Albert uses LangChain's `create_agent` function to build a ReAct-style [8] agent. The architecture consists of:

- **LLM**: OpenRouter API provides access to multiple models (Developed with Nemotron Nano)
- **Tools**: Dynamically discovered functions exposed to the agent
- **Memory**: LangGraph's InMemorySaver maintains conversation context
- **Checkpointer**: Enables state persistence (currently session-based)

The system uses a JSON configuration file (`config.json`) for centralized management of model parameters, API endpoints, directory paths, and system prompts. This approach enables easy experimentation with different models and configurations without code modification. Key configuration elements include:

- **Model Settings**: Model identifier, temperature (0.7), and token limits (2000)
- **Directory Paths**: Sandbox root, PDF input/output directories, and vector store location
- **System Prompt**: Explicit instructions defining agent capabilities and tool usage guidelines, including directives to prioritize vector search for document-related queries

### B. Tool Implementation

Custom tools were implemented as Python functions with docstrings serving as descriptions for the LLM:

**Web Search & Scraping**: `Google_search` queries SerpAPI and returns formatted results with URLs. `Web_scrape` uses Playwright to extract clean text from webpages.

**File Management**: `Write_to_file` creates files within the sandbox.

**PDF Processing**: `PDF_converter_to_MD` converts PDFs using Docling, preserving tables and structure. `file_name_extractor` lists available files.

**Vector Store Operations**: `Add_to_vector_store` chunks and embeds documents. `Search_vector_store` performs semantic similarity search. `List_vector_store` shows indexed documents.

### C. RAG Pipeline

The retrieval system uses the following workflow:

1) User query is embedded using the same model as document chunks
2) ChromaDB performs cosine similarity search
3) Top-k chunks (default k=3) are retrieved
4) Results are formatted with source attribution and returned to the agent

### D. Prompt Engineering

System prompts explicitly instruct the agent when to use each tool. Early testing revealed that without clear guidance, smaller models fail to use vector search appropriately. The final prompt includes:

- Listed capabilities
- Explicit instructions for tool selection
- Instructions on when to use RAG vs. web search

## V. Experiments & Results

### A. Model Selection

The system was developed and tested using Nvidia's Nemotron Nano 9B model via OpenRouter's free tier. While fast and accessible, testing revealed challenges with tool-calling reliability. The model occasionally produced malformed JSON during complex tool invocations, resulting in HTTP 400 errors. Based on development experience, approximately 20% of multi-step tasks required retry attempts due to JSON formatting issues. This highlighted the importance of robust error handling and the tradeoff between model accessibility (free tier) and tool-calling reliability.

### B. RAG Performance

**Qualitative Analysis**:
*Example Query*: "What is my work experience?"
*Retrieved Chunks*:

```
Result 1 (from resume.md):
## Process Manufacturing Engineer
SOMEWHERE LLC | Sep 2021 - Dec 2024
- Designed automated systems...
```

The system successfully retrieved relevant employment history with correct temporal ordering and detail preservation. Table structures from original PDFs were maintained in markdown format.

### C. Tool Integration

**Success Cases**:

- Converting PDFs to markdown and automatically adding to vector store
- Searching web, scraping results, and saving summaries to files

- Answering questions about converted documents using RAG
- Creating files within the sandbox with content generated by the agent

**Limitations**:

- Agent sometimes asks clarifying questions instead of proceeding with available information
- multi-turn tool calls sometimes lead to confusion without explicit prompts
- Web scraping fails on ocassion
- Have trouble putting markdown to vector store, agent likes to start fresh with pdf.

## VI. Conclusion

This project successfully demonstrates a functional AI agent combining LangChain, RAG, and multiple specialized tools. Key accomplishments include:

- Implemented complete RAG pipeline with document conversion, chunking, embedding, and semantic retrieval
- Built modular tool architecture enabling easy extension
- Achieved reliable tool calling with appropriate model selection
- Maintained security through proper sandboxing and input validation
- Created accessible system using only free-tier APIs(Can use paid openrouter models for better performance)

### A. Limitations

- Session-only memory (no persistence across restarts)
- Limited to text/PDF documents (no image or audio processing)
- Relies on docstring quality for tool descriptions
- No multi-agent collaboration or complex planning
- Rate limits on free-tier APIs restrict usage(Can use paid openrouter models to avoid this)

### B. Future Work

- Allowing file creation anywhere within sandbox, including subdirectories
- Describe features for better user experience
- File-based checkpointers for conversation persistence
- Model Context Protocol (MCP) server integration
- Dynamic model switching (fast vs. expert modes)
- Permission system for fine-grained action control
- Autonomous web navigation using HTML/CSS analysis
- Additional LangChain features (LCEL, custom chains)

### C. Lessons Learned

Building Albert from the ground up provided critical insights into agent development. Prior to this project, I had no understanding of how LangChain agents orchestrated tool calls, how vector databases indexed and retrieved information, how embedding models transformed text into semantic vectors, how RAG pipelines connected these components, or how prompt engineering influenced agent behavior. The project required learning each element and understanding their integration:

- **Agent Architecture and Tool Orchestration**: How LangChain coordinates tool calls and manages conversational memory
- **RAG Pipelines**: Building retrieval systems with vector embeddings using ChromaDB and Sentence-Transformers
- **Custom Tool Design**: Creating function calling interfaces that LLMs can reliably invoke
- **Model Limitations**: Smaller models struggle with complex tool calls and structured output generation
- **Prompt Engineering Impact**: How system prompts and docstrings directly influence agent behavior and tool usage
- **Security Basics**: Implementing path sanitization, sandboxing, and input validation to prevent unauthorized access
- **PDF Data Extraction**: Using Docling to preserve tables and formatting during document conversion
- **Debugging Agents**: Sometimes explicit instruction through prompts is required to teach proper tool usage
- **API Rate Limits**: Understanding providers: OpenRouter (50/day)

The overarching lesson: building functional agents requires understanding the complete pipeline rather than isolated components. Prompts, tools, memory, embeddings, and model selection must work together cohesively—failure in any single element compromises the entire system.

## References


[1] LangChain, "LangChain Documentation," 2024. [Online]. Available: https://python.langchain.com/

[2] Significant Gravitas, "AutoGPT," GitHub repository, 2023. [Online]. Available: https://github.com/Significant-Gravitas/AutoGPT

[3] IBM Research, "Docling: Document Processing Library," 2024. [Online]. Available: https://github.com/DS4SD/docling

[4] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," NeurIPS, 2020.

[5] Chroma, "Chroma: The AI-native open-source embedding database," 2024. [Online]. Available: https://www.trychroma.com/

[6] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," EMNLP, 2019.

[7] Microsoft, "Playwright Documentation," 2024. [Online]. Available: https://playwright.dev/

[8] S. Yao et al., "ReAct: Synergizing Reasoning and Acting in Language Models," ICLR, 2023.