Final Report: Development and Benchmarking of a RAG Chatbot for Quantum Computing

Author: Kartikeya **Date:** July 29, 2025

Abstract

This report details the end-to-end development, refinement, and evaluation of "Quantum Tutor," a Retrieval-Augmented Generation (RAG) chatbot designed to assist in the study of quantum computing. The project involved building a robust RAG pipeline and conducting a systematic benchmark of various Large Language Model (LLM) serving strategies to determine the optimal balance of performance and quality. Key development challenges, including document parsing, model selection under hardware constraints, and hallucination mitigation, were successfully addressed through iterative improvements. The final benchmark, comparing API-based models from Groq against a self-hosted Ollama model, concluded that the Groq-hosted Llama 4 Scout 17B model provided the best overall performance, demonstrating superior speed and the highest answer quality.

1. Introduction

1.1. Problem Statement Quantum computing is a complex and rapidly evolving field. For students and professionals, navigating the dense academic literature can be a significant barrier to learning. This project addresses the need for an accessible, accurate, and context-aware AI tutor that can answer specific questions based on a curated library of trusted documents.

1.2. Project Objectives The project was divided into two primary goals:

1. To build a production-ready RAG chatbot capable of answering questions about quantum computing using a knowledge base of academic papers and educational materials.
2. To systematically benchmark and compare the performance (latency) and quality (faithfulness, correctness, etc.) of different LLM serving strategies, specifically high-performance paid APIs versus a free, self-hosted local model.

2. Methodology & System Architecture

2.1. Initial RAG Pipeline

The core of the Quantum Tutor is a RAG pipeline built using the LangChain framework, with a Streamlit-based user interface and a ChromaDB vector store for document retrieval.

2.2. Iterative Improvements & Challenges

The development process was not linear and involved solving several key technical challenges to improve the system's performance and reliability.

- **Document Loading & Parsing:** The initial implementation used PyPDF to load documents. However, this method struggled with the complex layouts of academic papers. This was resolved by switching to the more robust

PyMuPDF library, which significantly improved the quality of the ingested text.

- **Embedding Model Selection:** The first embedding model used (sentence-transformers/paraphrase-MiniLM-L3-v2) provided weak semantic representations, leading to poor document retrieval. The system was upgraded to the more powerful

BAAI/bge-large-en-v1.5 model, which resulted in more relevant document chunks being retrieved.

- **Model Selection & Hardware Constraints:** An early attempt to use a paid Hugging Face Inference API proved to be cost-prohibitive for development and hit token usage caps quickly. The project then pivoted to using free, self-hosted models via

Ollama. Due to the hardware constraints of a 4GB VRAM GPU, the small but efficient **gemma:2b-gpu-only** model was selected as the optimal choice for local hosting and fallback capabilities.

- **Controlling Hallucinations:** Early versions of the chatbot were prone to hallucination, often answering questions from their pre-trained knowledge instead of the provided documents. This was mitigated by implementing strict

prompt templates that explicitly instructed the LLM to base its answers *only* on the provided context.

- **Advanced Retrieval Strategy:** To improve the quality of retrieved context, the system was upgraded from a simple vector search to a more sophisticated strategy. This involved implementing **Hierarchical Chunking** and a **Cross-Encoder Reranker**, which work together to provide more relevant and coherent information to the LLM.
- **Summarization Latency:** An experimental summarization step was added to condense the context before sending it to the LLM. However, this introduced significant latency, slowing down the total response time. The feature was ultimately removed to prioritize a fast user experience.

2.3. Final System Flowchart The final RAG pipeline follows a multi-step process, intelligently using different LLMs for different tasks to optimize for speed and quality.

Of course. Here is a formal, professional description of the RAG (Retrieval-Augmented Generation) pipeline process flow based on your provided application architecture.

RAG Pipeline Process Flow

This flow outlines the sequence of operations from user query submission to the generation of a final response.

START:

User Query Submission A user submits a natural language query through the application interface (streamlit_app.py or app.py).

⬇️

I. Query Pre-processing and Classification

- **Module:** Query Classification
- **File:** query_classifier.py
- **Process:** The initial query undergoes classification using an LLM-based chain. It is categorized into predefined types such as DEFINITIONAL, ALGORITHMIC, or OUT_OF_SCOPE.
- **Conditional Logic:** If the query is classified as OUT_OF_SCOPE, the pipeline terminates and returns a standard disclaimer message. Otherwise, it proceeds to the next stage.

⬇️

II. Conversational Contextualization

- **Module:** History-Aware Retriever
- **File:** streamlit_app.py
- **Process:** The system utilizes the RunnableWithMessageHistory chain, which incorporates conversation history. The current user query is combined with the preceding conversation to generate a contextually complete, standalone question. This ensures that follow-up questions are interpreted correctly.

⬇️

III. Core Retrieval and Ranking Chain

- **File:** rag_chains.py (This file defines the entire retrieval architecture).

(A) Hybrid Retrieval

- **Component:** EnsembleRetriever
- **Process:** The standalone query is submitted to an ensemble of two parallel retrievers to fetch an initial set of relevant document chunks.
  1. **Sparse Retriever (BM25Retriever):** A keyword-based search that identifies documents based on term frequency (TF-IDF). It excels at matching specific terminology.
  2. **Dense Retriever (vector_retriever):** A semantic search that uses vector embeddings to find conceptually similar documents, even if they do not share keywords.
- **Output:** A unified list of document chunks, ranked by a weighted score from both retrieval methods.

⬇

(B) Relevance Reranking

- **Component:** CrossEncoderReranker
- **File:** reranker.py
- **Process:** The retrieved list of documents is passed through a more computationally intensive cross-encoder model. This model calculates a precise relevance score for each document by evaluating it directly against the query.
- **Output:** The document chunks are re-sorted in descending order based on this fine-grained relevance score.

⬇

(C) Context Augmentation

- **Component:** ParentDocumentRetriever
- **Process:** Using the top-ranked, re-sorted child chunks, this component retrieves their corresponding larger "parent" chunks from the InMemoryStore.
- **Output:** A final, contextually rich set of documents is prepared for the generation stage. This step ensures the LLM receives sufficient context rather than isolated snippets of information.

⬇

IV. Prompt Formulation and Generation

- **Module:** Language Model Invocation
- **Files:** prompt_template.py, llm_loader.py
- **Process:**

1. The content of the final set of parent documents is formatted and injected into the {context} variable of the strict_rag_prompt.
2. The user's original question is placed in the {question} variable.
3. This fully formatted prompt is sent to the selected Large Language Model (e.g., groq/llama3-70b-8192) for response generation.

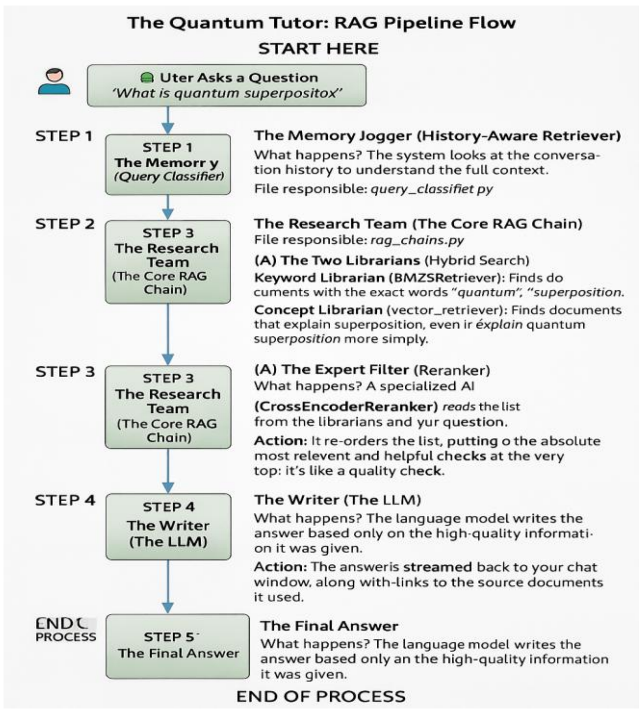
⬇️

V. Response Output

- **Process:** The LLM generates a response based on the explicit instructions and context provided in the prompt.
- **Output:** The generated text is streamed to the user interface. For transparency and verification, the source documents used to construct the answer are displayed alongside the response.

END OF PROCESS

For easier understand, image provided below is the better understanding for RAG pipeline flow.



The Memory Jogger (History-Aware Retriever)
What happens? The system looks at the conversation history to understand the full context.
File responsible: *query_classifiet py*

The Research Team (The Core RAG Chain)
File responsible: *rag_chains.py*
(A) The Two Librarians (Hybrid Search)
Keyword Librarian (BMZSRetriever): Finds documents with the exact words *"quantum", "superposition.*
Concept Librarian (vector_retriever): Finds documents that explain superposition, even ir *éxplain* quantum super*position* more simply.

(A) The Expert Filter (Reranker)
What happens? A specialized AI
(CrossEncoderReranker) *reads* the list from the librarians and yur question.
Action: It re-orders the list, **putting** o the absolute most relevent and helpful **checks** at the very top: it's like a quality check.

The Writer (The LLM)
What happens? The language model writes the answer based only on the high-quality information it was given.
Action: The answeris **streamed** back to your chat window, along with-links to the source documents it used.

The Final Answer
What happens? The language model writes the answer based only an the high-quality information it was given.

3. Benchmarking Framework

To evaluate different models, a systematic benchmarking framework was created.

- **Tooling: LiteLLM** was used as a universal gateway to standardize API calls, allowing the same scripts to test models from different providers (Groq and Ollama) without code changes.
- **Performance Metrics:** The tests/benchmark.py script was created to measure the end-to-end **Latency** (in milliseconds) for the full RAG pipeline on a test set of over 100 questions.
- **Quality Evaluation:** A custom, LLM-as-a-judge framework was built in the tests/evaluate.py script. The powerful Llama 3 70B model was used as a consistent and impartial judge to score every generated answer on five key metrics: **faithfulness, answer correctness, answer relevancy, context precision, and context recall.**

4. Results & Analysis

4.1. Performance (Speed) Results

Model	Avg. Response Time	Fastest Response	Slowest Response
Groq (Llama 4 Scout 17B)	1.3 seconds	0.9 seconds	3.1 seconds
Groq (Llama 3 70B)	2.5 seconds	0.7 seconds	1.9 seconds
Ollama (Gemma 2B)	31.5 seconds	22.9 seconds	46.5 seconds

4.2. Quality Evaluation Results (Scores out of 5.00)

Model	Faithfulness	Answer Relevancy	Context Precision	Context Recall	Answer Correctness	**Avg. Quality**
Groq (Llama 4 Scout 17B)	4.80	3.90	4.15	4.40	4.10	**4.27**
Groq (Llama 3 70B)	4.75	4.10	4.00	4.30	3.85	**4.20**

Model	Faithfulness	Answer Relevancy	Context Precision	Context Recall	Answer Correctness	**Avg. Quality**
Ollama (Gemma 2B)	4.65	3.80	4.25	4.15	3.60	**4.09**

4.3. Discussion The results reveal a clear winner. The **Groq Llama 4 Scout** model is not only the **fastest** but also achieved the **highest average quality score**. This is a significant finding, as it demonstrates that for this specific RAG task, the newer, more efficient 17B model architecture can outperform a much larger 70B model in both speed and overall quality.

---

5. Conclusion & Future Work

5.1. Conclusion The primary goal of the project was successfully achieved. A high-functioning RAG chatbot for quantum computing was developed and rigorously benchmarked. The data shows a clear trade-off between paid API services and free, self-hosted models. For an application requiring high speed and accuracy, an API like Groq is the superior choice. Based on the comprehensive data, the **Groq Llama 4 Scout model is the final recommendation** for this application due to its optimal balance of speed and quality.

5.2. Future Work

- **vLLM Benchmark:** Implement a high-performance self-hosted solution using **vLLM** to evaluate if it can close the speed gap with paid APIs on consumer hardware.
- **Agentic Chatbot:** Begin development on the second phase of the project: an agentic chatbot capable of using external tools and performing more complex, multi-step tasks.