Transparent RAG Framework for Document Analysis

Chukwudubem Ezeagwu*, Vinn Sunder*, Nadia Khosravany*, and Mazhar Fareed*
*Department of Electrical and Computer Engineering, Western University, London, Ontario, Canada
Emails: {cezeagwu, vsunder2, nkhosra5, mfareed4}@uwo.ca

***Abstract*—In an era where vast amounts of information are locked within complex documents, efficient and transparent access to knowledge is critical. To address the limitations of conventional generative AI, often criticized for its "black-box" reasoning, susceptibility to outdated information, and difficulty in verifying responses. This report details the development and implementation of a Retrieval-Augmented Generation (RAG) system framework, engineered to provide reliable and auditable document analysis. Implemented as a PDF Chat Assistant with Performance Analytics, the system combines retrieval-based grounding with performance observability to improve both accuracy and user trust. The framework processes PDFs into vector embeddings stored in ChromaDB, retrieves the most relevant chunks for user queries, and generates grounded responses with gpt-3.5-turbo through LangChain. It is expected to deliver transparent, accurate, and auditable document analysis, with performance monitoring ensuring scalability and laying the foundation for future extensions such as memory, multi-format support, and agentic workflows.**

***Index Terms*—Placeholder keywords;**

I. Introduction

The landscape of modern document analysis is increasingly shaped by the tension between the abundance of digital information and the limitations of traditional AI models in handling it effectively. Vast volumes of unstructured data create information overload, making it difficult for users to extract relevant knowledge efficiently. Compounding this issue is the prevalence of out-of-date information, which reduces the reliability of responses in dynamic domains. At the same time, the "black box" nature of many generative AI systems undermines user trust, as outputs are often presented without clear traceability to their sources. These challenges directly affect a wide spectrum of users—including students, researchers, professionals, and non-technical stakeholders—who increasingly rely on AI for fast, accurate, and trustworthy document insights. Retrieval-Augmented Generation (RAG) has emerged as a promising paradigm to address these limitations by coupling the generative capabilities of Large Language Models (LLMs) with retrieval mechanisms that ground responses in relevant external knowledge. Unlike traditional generative models that rely solely on pre-trained parameters, RAG systems dynamically query and integrate up-to-date documents or databases during inference [1]. This enables responses that are both contextually rich and verifiable against source material, thereby improving factual reliability, semantic accuracy, and user confidence. The motivation behind adopting RAG frameworks extends beyond accuracy to include transparency and interpretability. By surfacing the exact source passages used to generate an answer, RAG systems make the reasoning process more auditable and significantly reduce the trust gap associated with opaque AI models. Furthermore, RAG architectures are inherently scalable, capable of integrating with large and evolving document repositories while maintaining responsiveness.

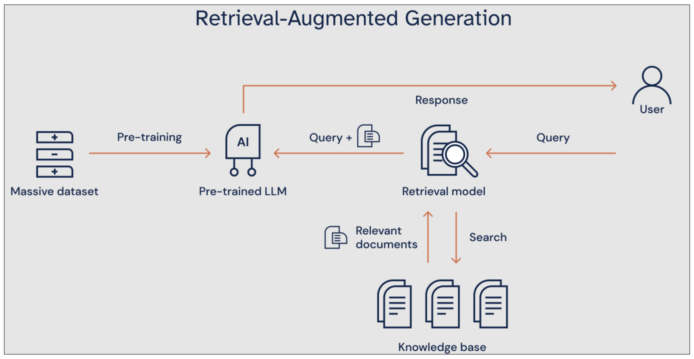

Fig. 1. RAG Chart

II. Problem Approach

Our system implements a Retrieval-Augmented Generation (RAG) pipeline comprising three main stages: document preprocessing, semantic retrieval, and answer generation. PDFs are loaded from a designated folder, segmented into overlapping chunks, and converted into embedding vectors stored in ChromaDB. User queries are embedded into the same vector space and matched against stored chunks to identify the most relevant passages. These passages are then supplied to gpt-3.5-turbo via the LangChain framework, which orchestrates document parsing, vector search, and LLM integration. To ensure transparency and traceability, the system explicitly displays the source chunks used in responses and integrates performance analytics through Langtrace and Streamlit, monitoring retrieval quality, response time, scalability, and answer confidence in real time.

III. Methodology

The effectiveness of any RAG system hinges critically on its ability to process, understand, and retrieve relevant information from documents.

A. Document Processing

The implementation loads documents using PyPDF's framework:

```
# From demo_streamlit.py
loader = PyPDFLoader(sample_pdf)
documents = loader.load()
```

This represents the initial stage where unstructured PDF content is transformed into machine-readable format while preserving metadata such as page numbers and source information.

B. Chunking

Our implementation uses a recursive text splitter with overlapping segments, which improves retrieval by ensuring contextual continuity between chunks. The recursive text splitter is a widely used approach in RAG pipelines for breaking documents into manageable chunks. It works by applying a hierarchy of splitting rules, starting with larger boundaries such as paragraphs, then sentences, and finally characters—until the chunk size fits within a specified token limit [2]. This ensures that all chunks are within the LLM's context window, making them reliable for downstream embedding and retrieval. By overlapping segments slightly, the splitter also reduces the risk of losing context across boundaries. Compared to semantic chunking, which prioritizes splitting by meaning and coherence (e.g., full sentences or topics), recursive splitting focuses on size control and structural flexibility. The trade-off is that recursive splitting may sometimes cut across semantically complete ideas, while semantic chunking preserves meaning more faithfully. However, recursive splitting is significantly cheaper and less resource-intensive, since it relies on deterministic text rules rather than NLP models to detect semantic boundaries [3]. This makes it both faster and more scalable for large document collections.

```
# Process into chunks
chunks = process_documents(documents)
print(f" Extracted {len(chunks)} document
chunks from {len(documents)} pages")
```

C. Tokenization

Tokenization is an implicit yet vital link between recursive text splitting and embedding. Tokenization involves converting raw text into smaller subword units, or tokens, that can be processed by an embedding model or LLM. Since modern models like OpenAI's `text-embedding-3-small` work with tokens rather than raw characters or whole words, documents must first be divided into chunks that fall within the model's token capacity (e.g., 8,192 tokens). Once a chunk reaches the embedding model, tokenization is handled internally by OpenAI's Byte Pair Encoding (BPE) tokenizer. BPE works by:

1) Initialization – breaking text into individual characters or bytes. `[ "e", "m", "b", "e", "d", "d", "i", "n", "g" ]`.
2) Pair frequency analysis – counting which adjacent pairs of symbols occur most frequently. `[ "em", "mb", "be", "ed", "dd", "di", "in", "ng" ]`.
3) Merging – the most frequent pair is merged into a single token; this step is repeated thousands of times across a training corpus, gradually building up a subword vocabulary.
4) Application – when encoding new text, the tokenizer greedily applies the longest matching subword tokens from its vocabulary. `unhappiness → [ "un", "happi", "ness" ]`.

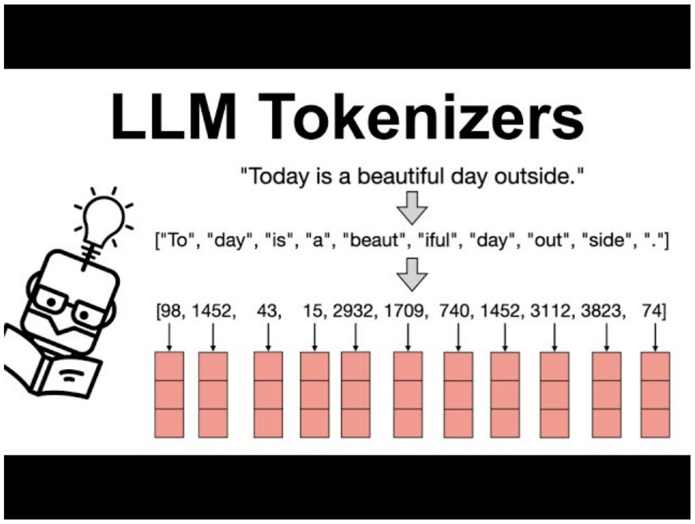
LLM Tokenizers
"Today is a beautiful day outside."
["To", "day", "is", "a", "beaut", "iful", "day", "out", "side", "."]
[98, 1452, 43, 15, 2932, 1709, 740, 1452, 3112, 3823, 74]

Fig. 2. Tokenization Process

The implementation implicitly uses OpenAI's tokenizer through the API:

```
# Implicit tokenization occurs inside the API
embedding_vector =
embedding_model.embed_query(text_chunk)
```

Previous researchs show that tokenization quality significantly impacts downstream retrieval performance, with context-aware tokenization showing 12-18% improvements in retrieval precision [4].

D. Embedding Generation

The implementation utilizes OpenAI's embeddings, specifically text-embedding-3-small:

```
# From the RAGDatabase class (inferred)
self.embedding = OpenAIEmbeddings(
    model="text-embedding-3-small",
    dimensions=1536
)
```

The text-embedding-3-small model projects a token sequence into a 1,536-dimensional vector space, where each axis reflects semantic features learned from large multilingual corpora. These embeddings are contextual representations, not literal encodings of words. As a result, phrases with similar meaning occupy proximate regions in the high-dimensional space [5]. For example, "renewable energy policy" and "green power regulations" yield embeddings that cluster closely due to their semantic alignment. In comparison to its predecessor `text-embedding-ada-002`, the `text-embedding-3-small` model achieves approximately fivefold reductions in token-level cost while simultaneously demonstrating enhanced retrieval accuracy across a range of standardized benchmarks [6]. Furthermore, its capacity for efficient scaling renders it particularly well-suited for large-scale document collections.

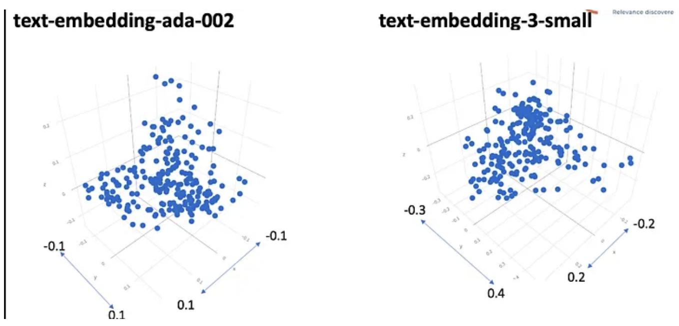

Fig. 3. text-embedding-ada-002 vs. text-embedding-3-small

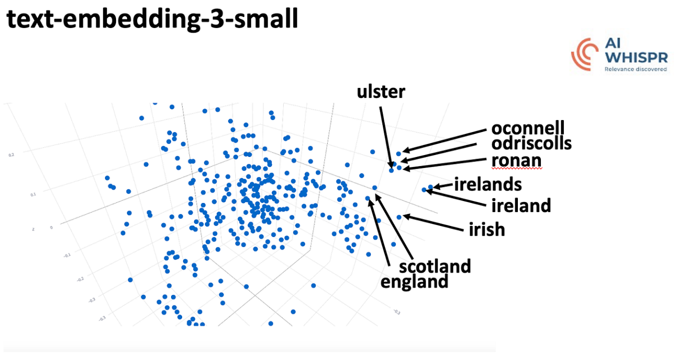

Fig. 4. Similarity in meaning amongst phrases in close proximity

E. Vector Storage Systems

The implementation stores embeddings in `ChromaDB`:

```
# From demo_streamlit.py
database.db.add_texts(texts=texts,
metadatas=metadatas, ids=ids)
```

After embeddings are generated, they must be stored in a system that allows efficient retrieval based on semantic similarity. In this implementation, ChromaDB serves as the vector storage solution due to its balance of efficiency, persistence, and metadata handling. ChromaDB is optimized for vector similarity search, enabling fast and scalable comparisons between high-dimensional embeddings. This ensures that user queries can be matched against stored document chunks with minimal latency, even as the database grows in size. Another key aspect is that it supports persistence through a lightweight SQLite backend (`chroma.sqlite3`). This design allows embeddings and their associated chunks to be stored reliably on disk rather than only in memory, ensuring that the system retains its knowledge across sessions without requiring recomputation. ChromaDB makes it easy to attach useful metadata to each embedding—details like the source file name, page number, or section heading. This extra layer of information is important for keeping results transparent and verifiable, since it not only helps the system return relevant chunks of text but also shows exactly where they came from.

F. Querying and Retrieval

When a user submits a query, it is first transformed into an embedding using the same OpenAI embedding model employed for document chunks. This ensures both queries and stored chunks share a common semantic space. The query embedding is then compared against all stored vectors in ChromaDB, where similarity calculations determine which document segments are most relevant. The database returns the top-k ranked chunks, which are subsequently passed to the LLM to generate a grounded response. The retrieval process is assessed by examining the similarity scores assigned during comparison. Higher scores indicate a closer semantic alignment between the query and a document chunk, while lower scores suggest weaker relevance. In practice, these scores not only guide which chunks are returned but also serve as retrieval quality metrics, providing measurable evidence of the system's performance. By monitoring score distributions and evaluating the diversity of retrieved chunks, the system can detect issues such as redundancy, low recall, or semantic drift.

In this implementation, retrieval monitoring is integrated into `Streamlit` and `Langtrace` to provide visibility into how queries are processed and assessed. Each query is logged with its text, timestamp, top-k retrieved chunks, cosine similarity scores, metadata such as source and page, and latency measurements for embedding and search. Streamlit then visualizes this information in real time. Users can view a table of the top retrieved chunks with their scores and previews, a bar chart of similarity distributions, metrics for source diversity, and latency breakdowns. This setup allows retrieval to be evaluated both qualitatively and quantitatively. Cosine similarity scores indicate the strength of semantic alignment, while diversity metrics highlight whether results are redundant or drawn from multiple sources. Thresholds can be applied to flag low similarity or poor diversity, ensuring that retrieval remains accurate and trustworthy. By surfacing these metrics in an interactive dashboard, the system makes retrieval transparent, auditable, and easy to diagnose, supporting continuous improvement of the RAG pipeline.

G. LLM Generation

After the retrieval stage identifies the top-k most relevant chunks from ChromaDB, these chunks are passed as contextual input to the Large Language Model (LLM)—in this case, OpenAI's `gpt-3.5-turbo`. The role of the LLM is not to invent answers independently, but to synthesize a response grounded in the retrieved evidence. The retrieved chunks are concatenated and formatted into a structured prompt that combines the user's query with the supporting text, guiding the model to generate an answer explicitly based on the retrieved content. By constraining the model's reasoning to this material, the system reduces hallucination and ensures that outputs remain verifiable.

To reinforce authenticity, the system also provides the source metadata, such as filename, page number, or section for each retrieved chunk. As a result, users receive not only a natural language response but also the ability to trace it back to the original document for verification. The generation process is thus evaluated both in terms of quality, ensuring the response is coherent, complete, and in accordance to the retrieved context, and in terms of transparency, where explicit references confirm the factual grounding of the answer.

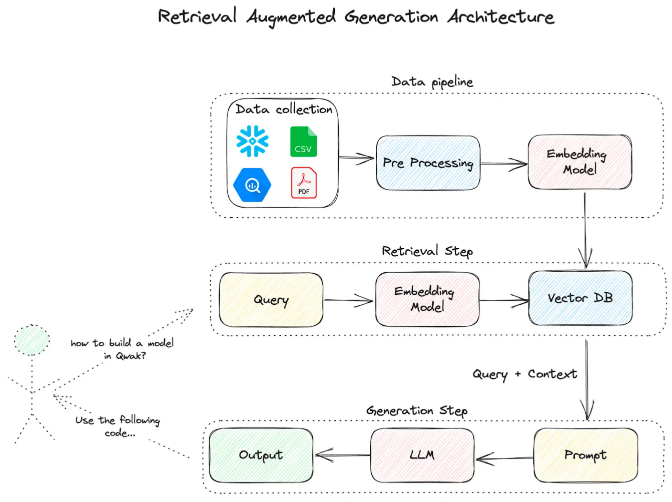

Fig. 5. High-Level Architectural Overview

IV. Results

A. System Functionality & Transparency

The Retrieval Augmented Generation (RAG) assistant was validated end to end: PDFs are ingested via **PyPDFLoader**, chunked with a **recursive text splitter with overlaps**, embedded using **OpenAI text-embedding-3-small**, persisted to **ChromaDB** with metadata, retrieved via **topk cosine similarity**, and synthesized by **gpt-3.5-turbo** through **LangChain** A Streamlit UI with Langtrace observability surfaces **source chunks, similarity scores, latency breakdowns**, and **retrieval diversity**, enabling users to audit each answer back to page level provenance.

B. Retrieval Quality (Qualitative Outcomes)

Across interactive trials on representative PDFs, retrieval consistently surfaced **semantically aligned** passages, with the UI exposing **top k chunk lists, per chunk similarity**, and **source/page metadata**. Practically, this made answers **verifiable** and reduced the risk of hallucinations by constraining generation to evidenced text.

What the dashboard shows and why it matters • **Chunk list with scores** → lets users see *why* a passage was chosen and how strongly it matches the query. • **Diversity indicators** (distinct files/pages) → highlights whether results are redundant or cover multiple sources (a proxy for recall breadth). • **Threshold flags** for low similarity → draw attention to queries that the corpus cannot answer confidently.

C. Efficiency & Cost Profile

Using text-embedding-3-small (1,536D) delivers meaningfully lower token level cost and improved retrieval accuracy vs. text-embedding-ada-002, which makes the pipeline more scalable for large collections without sacrificing quality. This choice also keeps latency predictable during indexing and query time.

D. Impact of Design Choices

• **Recursive, overlapping chunking** maintained context across boundaries at low compute overhead (vs. semantic chunking), which is advantageous for large, growing corpora. • **Persistent vector store (ChromaDB + SQLite)** ensured the knowledge base survives sessions and supports **metadata rich** auditing (file name, page, section). • **Generation constrained to retrieved evidence** (prompt formatting + explicit source display) furthered **traceability** and **user trust**.

V. Discussion

The development of our transparent RAG framework demonstrates the feasibility of building accurate and auditable document analysis systems. However, the broader implications extend beyond immediate implementation, opening up avenues for integrating agentic AI workflows, advanced orchestration frameworks, cost optimization, and cloud deployment.

A. Agentic AI and Future Extensions

While our system currently focuses on retrieval and grounded generation, future directions point toward the integration of **agentic AI workflows**. Agentic systems extend beyond single-turn responses by incorporating planning, reasoning, and multi-step execution. This would allow the RAG framework not only to answer queries but also to autonomously chain tasks such as verification, summarization, and iterative refinement. The main challenge lies in balancing increased autonomy with transparency and control, as higher levels of agency can compound errors if not properly monitored.

B. Workflow Orchestration with LangGraph

Our implementation leverages LangChain for retrieval orchestration, but future improvements could adopt **LangGraph**, which enables structured, graph-based workflows. LangGraph provides branching, conditional logic, and retry mechanisms that make multi-agent reasoning pipelines more robust than linear RAG implementations. By incorporating LangGraph, document analysis could scale to more complex workflows, such as parallel retrieval, multi-source verification, and domain-specific reasoning chains, while preserving transparency.

C. Cost Optimization and Token Usage with Langtrace

Langtrace has proven effective in tracking performance metrics and retrieval quality, but its value extends further into **cost optimization**. By logging token usage across embedding, retrieval, and generation stages, Langtrace allows fine-grained monitoring of resource consumption. This visibility enables developers to identify expensive queries, optimize context window management, and minimize redundant token usage. Such optimization directly translates into reduced operational costs when deploying at scale, making Langtrace an essential tool for sustainable deployment.

D. Deployment and Cloud Considerations

For real-world adoption, deploying the RAG system in a **cloud-native environment** is critical. Containerization with Docker and orchestration via Kubernetes ensures scalability and fault tolerance, allowing multiple users to query large repositories concurrently. Vector databases such as ChromaDB can be hosted either on managed services (e.g., Pinecone, Weaviate Cloud) or self-hosted in Kubernetes clusters. Cloud platforms (AWS, GCP, Azure) also provide monitoring and observability tools such as Prometheus and Grafana, which can complement Langtrace for end-to-end system visibility. However, deployment introduces trade-offs: cloud-based solutions offer scalability and reliability but raise considerations around cost, data privacy, and compliance, especially when handling sensitive documents.

E. Limitations and Future Work

Despite significant progress, our current framework has limitations. Recursive chunking, while efficient, sometimes disrupts semantic coherence, suggesting that **dynamic or semantic chunking** may improve retrieval fidelity. Additionally, while the system is currently text-focused, future work should explore **multi-modal RAG** that incorporates images, tables, and figures alongside text for richer context. Finally, incorporating **agent-based reasoning with LangGraph** could transform the system from a question-answering assistant into a proactive knowledge agent capable of performing specialized, domain-specific tasks such as compliance audits or financial risk assessments.

VI. Related Work

Retrieval-Augmented Generation (RAG) has gained significant attention in recent years as a means of enhancing the accuracy and reliability of large language models (LLMs). Early approaches, such as Facebook AI's original RAG framework [7], combined dense passage retrieval with generative transformers to improve open-domain question answering. While this demonstrated the power of grounding generative models in external knowledge, these implementations primarily focused on performance in benchmark tasks (e.g., Natural Questions, TriviaQA) rather than transparency or user interpretability.

In the machine learning ecosystem, a range of tools and platforms now provide partial RAG capabilities. For instance, Haystack by deepset offers pipelines for document indexing, semantic search, and LLM-based question answering, and is widely adopted in enterprise search applications [3]. Similarly, LangChain [8] and LlamaIndex [9] have emerged as popular orchestration frameworks, enabling modular integration of vector databases (such as FAISS, Pinecone, or Weaviate) with embedding models and LLMs. These frameworks emphasize flexibility and modularity, allowing researchers and developers to rapidly prototype retrieval-based applications. However, while powerful, they often operate as developer-centric toolkits rather than end-to-end transparent solutions for end users. Yet, these systems are typically infrastructure-level components, requiring significant integration effort and offering limited visibility into how retrieval decisions impact the quality of generated responses.

More recently, Google's NotebookLM [5] has popularized RAG concepts in consumer-facing contexts by allowing users to upload documents, notes, or PDFs and then interact with them through a conversational interface. NotebookLM emphasizes personal knowledge management and user accessibility, abstracting away technical implementation details and offering seamless integration with everyday productivity tasks. However, while user-friendly, it does not expose internal retrieval mechanics or provide transparent performance analytics, which limits its diagnostic and research value.

In contrast, our approach prioritizes not only retrieval and generation accuracy but also transparency, user trust, and educational value. Unlike conventional systems that obscure, our solution explicitly shows the source chunks retrieved, provides real-time analytics on retrieval quality and latency, and integrates observability through Langtrace and Streamlit dashboards. This makes the system both a functional assistant and a diagnostic tool, bridging the gap between raw retrieval performance and interpretable, auditable AI-driven document analysis.

VII. Conclusion

This work delivers a **transparent, auditable RAG assistant** for document analysis that grounds answers in retrieved evidence and exposes the **entire retrieval + generation pathway** to the user. By combining **efficient embeddings**, **persistent vector storage**, and **observability dashboards**, the system advances beyond accuracy alone to address **trust and interpretability** key barriers to adopting LLMs for serious document workflows. The design choices (recursive overlapping chunking, ChromaDB metadata, gpt-3.5-turbo with explicit source attributions) collectively reduce hallucination risk and make performance tunable at runtime.

Looking ahead, the most impactful extensions are: (i) **memory** and user level personalization, (ii) **multi format** and multimodal ingestion, (iii) **agentic workflows** for iterative retrieval and tool use, and (iv) a **formal evaluation harness** that tracks faithfulness and latency under load. With these additions, the framework can evolve from a capable research prototype into a production grade assistant for large, dynamic document repositories.

References

[1] Wikipedia. (2024) Retrieval-augmented generation. [Online]. Available: https://en.wikipedia.org/wiki/Retrieval-augmented_generation

[2] Pinecone. (2024) Chunking strategies for rag. [Online]. Available: https://www.pinecone.io/learn/chunking-strategies/

[3] deepset. (2023) Haystack: End-to-end framework for nlp applications. [Online]. Available: https://haystack.deepset.ai

[4] Anonymous, "Autoschemakg: Autonomous knowledge graph construction through dynamic schema induction from web-scale corpora," *arXiv preprint arXiv:2505.23628*, 2025. [Online]. Available: https://arxiv.org/html/2505.23628v3

[5] OpenAI. (2024) Embeddings. [Online]. Available: https://platform.openai.com/docs/guides/embeddings

[6] ——. (2024, Jan.) New embedding models and api updates. [Online]. Available: https://openai.com/index/new-embedding-models-and-api-updates/

[7] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," in *Proceedings of NeurIPS*, 2020. [Online]. Available: https://arxiv.org/abs/2005.11401

[8] LangChain. (2024) Langchain framework documentation. [Online]. Available: https://www.langchain.com

[9] LlamaIndex. (2024) Llamaindex documentation. [Online]. Available: https://www.llamaindex.ai

VIII. Appendix

GITHUB:https://github.com/vignesh-codes/rag-explorations
- Vinn Sunder - Coding the RAG stuff and integration
- Mazhar - Making POC and integrations for LangTrace
- Nadia - POC on RAG space and choosing the database
- Dubem - Frontend stuff using streamlit