PDF Question-Answering Agent using LangChain & Gemini API: Retrieval-Augmented Generation for Enterprise Knowledge Access

I. Executive Summary: Project Mandate and Solution Overview

1.1 Project Title & Objective

Project Title: PDF Question-Answering Agent using LangChain & Gemini API: Retrieval-Augmented Generation for Enterprise Knowledge Access.

Core Objective: The primary objective of this project is the design and deployment of a scalable, reliable Retrieval-Augmented Generation (RAG) system. This system must be capable of ingesting and indexing vast amounts of unstructured content contained within Portable Document Format (PDF) files. Through this indexing process, the agent will transform static documentation into an interactive, conversational knowledge base capable of generating contextually accurate, verifiable answers to complex user queries.

1.2 Business Justification for RAG Implementation

The ubiquity of the PDF format means that critical internal, domain-specific information—such as policy manuals, legal contracts, research reports, and technical

specifications—is often siloed within these files. This situation presents a significant challenge: organizational knowledge is trapped as 'dark data,' making systematic querying and synthesis time-consuming and inefficient. The core problem this agent solves is the elimination of manual searches through large documents, providing immediate, accurate access to specialized knowledge.

This is where the application of Artificial Intelligence becomes imperative, specifically through the RAG framework. Standard Large Language Models (LLMs) are restricted by their static pre-training data and consequently lack the domain expertise required for specialized applications.[1] RAG provides the necessary **factual grounding** by injecting real-time, domain-specific context derived directly from the indexed PDF corpus. This mechanism drastically mitigates the critical risk of **hallucinations**, ensuring responses are accurate and relevant.[1]

Architecturally, the selection of RAG over model fine-tuning represents a strategic operational decision favoring agility and economic efficiency. Fine-tuning a foundation model requires substantial computational resources and necessitates costly, full retraining cycles whenever the source data is updated. The RAG architecture bypasses these high **retraining costs** by maintaining a separate, dynamic knowledge base.[2] This methodology shifts the operational cost model from massive upfront capital expenditure (CapEx) associated with model training toward more scalable operational expenditure (OpEx) for API usage and vector database maintenance. This approach offers superior agility, which is vital in dynamic enterprise environments where internal documentation changes frequently.

II. Strategic Justification and Core Concepts

2.1 The Challenge of Knowledge Silos and LLM Factual Drift

Large Language Models (LLMs) possess powerful linguistic and reasoning capabilities but are inherently limited by their knowledge cut-off date. In specialized domains, reliance on pre-trained LLM knowledge inevitably leads to factual drift, irrelevance, and generic outputs.[3] Enterprise knowledge bases, particularly when structured in complex PDF formats, contain proprietary and specialized information that is simply inaccessible to general-purpose LLMs.[2] This knowledge silo represents a major hurdle for organizations seeking to leverage AI for decision support.

2.2 Introduction to Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is the definitive architectural solution to these limitations. RAG integrates a generative LLM with an external information retrieval system, typically a vector database containing domain data.[2] When a user query is submitted, RAG retrieves relevant text segments (context) from the external index based on semantic similarity, which is then used to augment the prompt sent to the LLM.[2]

The mechanism works by retrieving the top relevant passages, which are subsequently prepended to the user query, creating an "augmented prompt" that guides the LLM's response generation.[2] This process is transformative because it enables **grounded generation**.[3] This ensures that the generated answer is verifiable and directly attributable to the content within the source documents, leading to increased user trust and overall system utility.[1] RAG effectively allows LLMs to use domain-specific and up-to-date information that was never part of their initial training data.[1]

III. Technology Stack and Component Analysis

The project stack is built on a foundation of Python, utilizing LangChain for orchestration and the Google Gemini API for core AI services.

3.1 Orchestration Layer: LangChain Framework

LangChain is selected as the architectural glue for this project, serving as the essential orchestration layer.[4] Its primary function is to manage the complex, multi-step RAG pipeline, encompassing everything from initial document loading and text segmentation to vector store interaction and the final LLM call.[6] LangChain provides standardized, modular interfaces (abstractions) for connecting various components—such as different document loaders, embedding models, vector stores, and retrievers—which facilitates rapid prototyping and the easy swapping of components as the system scales.[7] This modularity is key for continuous optimization.

3.2 Generative and Embedding Models: Google Gemini API

The Google Gemini API provides the necessary high-performance models for both vectorization and generation.

- **Generative Model (LLM):** A capable model from the Gemini family, such as gemini-pro or the more efficient gemini-flash, is employed as the core generator. The model is chosen for its strong reasoning capabilities and its efficiency in handling the long context windows that result from combining a user query with multiple retrieved context chunks.[4]
- **Embedding Model:** A dedicated Gemini embedding model (e.g., text-embedding-004) is utilized to transform both the incoming document chunks and the real-time user queries into high-dimensional vector representations.[9]

Using the same technology vendor (Google) for both the generative and embedding models is a deliberate architectural choice. This ensures optimal semantic alignment between the indexed documents and the search query vectors, minimizing potential integration overhead and maximizing the likelihood of high-quality retrieval.[8]

3.3 Vector Storage Mechanisms: Chroma / FAISS Analysis

The vector store acts as the searchable knowledge base, efficiently indexing the high-dimensional vector representations of document chunks.[11]

- **FAISS (Facebook AI Similarity Search):** FAISS is recognized for its optimized index management and search efficiency, translating directly into superior retrieval speed. Experimental evidence indicates that FAISS exhibits slightly higher performance in both context precision and recall, as well as being faster in similarity search compared to Chroma.[12] This makes FAISS a strong candidate for low-latency, high-volume production systems.
- **Chroma:** Chroma is an excellent choice for local development and prototyping due to its simplicity and native Python integration.[12]

The overall system architecture must manage the fundamental trade-off between deployment complexity and operational speed. The system's true performance bottleneck in a production environment is frequently the **retrieval latency**, which is the inherent extra delay introduced by the search step that non-RAG LLMs avoid.[14] Therefore, mitigating this degradation is critical. The strategic decision is to use Chroma for the initial Minimum Viable Product (MVP)

phase for rapid implementation, followed by a mandated transition to FAISS or a scalable, managed enterprise vector database (like Pgvector or Zilliz Cloud [4]) in the long-term roadmap to ensure scalability, responsiveness, and consistent retrieval speed.[13]

Table III.3: Comparative Analysis of Vector Store Candidates

Feature	**Chroma**	**FAISS**	**Strategic Rationale**
Speed (Indexing/Search)	Good	Superior (Faster search time observed) [13]	Critical for optimizing system latency [15]
Scalability (Enterprise)	Limited (Local focus)	High (Optimized C++ index structure)	Preferred for high-volume, low-latency production
Retrieval Accuracy	Good	Slightly Superior in precision/recall [13]	Directly impacts Groundedness and answer quality
Setup Complexity	Low (Native Python)	High (C++ dependency, index management)	Chroma is ideal for rapid prototyping

3.4 Development Ecosystem and Utilities

The core implementation language is Python, leveraged by the LangChain libraries. Streamlit is recommended for building the initial user interface (UI).[16] Streamlit enables the rapid creation of an interactive web application where users can upload documents, submit queries, and receive real-time answers, facilitating efficient demonstration and testing.[5] For document parsing, LangChain's abstraction layer permits the integration of specialized PDF parsers to handle initial unstructured data extraction, although future phases will explore using the Gemini API directly for more effective handling of complex layouts.[7]

IV. System Architecture and Data Flow Blueprint

The RAG system operates through two primary flows: the **Ingestion Pipeline** (for data setup) and the **Retrieval & Generation Pipeline** (for real-time query answering).

4.1 High-Level Architecture Block Diagram

The system architecture is organized into modular components, chained together and orchestrated by the LangChain framework:

User Input (PDF File) ⟶ Document Loader ⟶ Text Splitter/Chunker ⟶ Embedding Model ⟶ Vector Database (Chroma/FAISS)

User Query ⟶ Query Embedding Model ⟶ Vector Database (Search/Retrieval) ⟶ Retriever (Top K Chunks) ⟶ Prompt Augmentation ⟶ Gemini LLM (Generator) ⟶ Final Answer (+ Citations) ⟶ Streamlit UI

4.2 Detailed Ingestion Pipeline Workflow (Indexing)

The ingestion pipeline is a crucial, one-time process (or a periodic update cycle) that transforms raw documents into the searchable vector index.

1. **User Uploads PDF:** The pipeline is triggered by the user uploading one or more PDF documents via the Streamlit interface.[5]
2. **Document Loading and Text Extraction (Loader):** LangChain's document loader abstraction reads the file. The loader is responsible for parsing the complex, unstructured PDF format, extracting all readable text content, and handling embedded metadata.
3. **Document Splitting and Chunking (Splitter):** The extracted raw text is segmented into smaller, contextually relevant blocks, known as chunks. The strategy must avoid arbitrary splitting of semantic units, which could fragment critical information.[18] Recursive or **paragraph-based chunking** is preferred for structured PDFs as it aligns with the natural flow of ideas.[18] Typical chunk sizes are set to optimize retrieval effectiveness and fit

within the Gemini model's context window, generally ranging from 500 to 1,000 tokens with appropriate overlap to maintain context continuity.[5]

4. **Embedding Generation:** Each isolated text chunk is transmitted to the **Gemini Embedding API**. The model encodes the text into a fixed-length, high-dimensional vector. This vector captures the chunk's semantic meaning, enabling mathematical comparison and search based on concept rather than exact keywords.[11]
5. **Vector Database Persistence:** The generated vectors are indexed and stored within the selected Vector Database (Chroma/FAISS). Crucially, the original text chunk and its associated metadata (e.g., source file name, page number) are stored alongside the vector. This metadata is essential for citation and verification during the generation phase.[11]

4.3 Detailed Retrieval and Generation Pipeline Workflow (Query Time)

This process executes in real-time upon receiving a user request, demanding low latency.

1. **User Asks a Question:** A natural language query is submitted through the application interface.
2. **Query Vectorization:** The system immediately vectorizes the user query using the **exact same Gemini Embedding Model** employed during the ingestion process.[10] Consistency in the embedding model ensures that the query vector exists within the same semantic space as the document vectors, maximizing search precision.
3. **Vector Search and Retrieval (Retriever):** The query vector is passed to the Vector Database to perform a high-speed similarity search. The system uses metrics like Cosine Similarity to identify the most relevant document vectors.[10] The **Retriever** component, configured via LangChain, extracts the Top-K (e.g., K=5) semantically closest chunks, which are the most relevant segments of the source PDF content.[5]
4. **Prompt Augmentation:** LangChain orchestrates the construction of the final, augmented prompt. This template is critically important; it combines system instructions (e.g., defining the agent's persona and objective to answer accurately), the totality of the retrieved context chunks (the grounding material), and the original user question.
5. **LLM Generation:** The augmented prompt is submitted to the Gemini Chat Model. The model synthesizes a final, coherent, and factual answer, strictly adhering to the instruction to base its response **solely** on the provided context.[1]
6. **Response Output:** The system returns the generated response to the user via the Streamlit UI, including mandatory source citations (file name and page number) derived from the chunk metadata.[1]

The effectiveness of the final generated answer is fundamentally tied to the quality of the retrieval step. The success of the pipeline is governed by a direct causal loop: if the chunking

is executed poorly—resulting in semantic fragmentation or incomplete context—the retrieval step will fetch insufficient or misleading information. The Large Language Model, despite its advanced reasoning capabilities, cannot produce a grounded answer if the retrieved context is flawed, following the principle of Garbage In, Garbage Out (GIGO).[5] Therefore, persistent engineering focus must be applied to optimizing the Text Splitter/Chunker component and adopting structure-aware techniques like recursive splitting.[18]

V. Foundational Concepts: Deep Dive

5.1 The Crucial Role of Chunking: Context Preservation

Chunking is the architectural mechanism for transforming large, unstructured documents into discrete, semantically meaningful units that can be vectorized and searched efficiently.[5] The primary challenge in chunking is balancing context preservation with token limitations. A chunk that is too expansive may dilute the specific relevance of the information, confusing the LLM. Conversely, an overly small chunk (e.g., a single sentence) often strips away the necessary explanatory context required for the LLM to synthesize a complete answer.[18]

For structured PDFs, which contain logical breaks, tables, and sections, simple fixed-length chunking is often suboptimal due to its tendency to arbitrarily split logical units. Advanced strategies, such as **page-level chunking** (which one evaluation found to offer the highest average accuracy and consistency) [20] or **paragraph-based chunking**, are prioritized. These methods naturally align with the human-readable structure of the text, ensuring each resulting chunk encapsulates a complete idea or topic, providing a "Richer Context" for the retrieval system.[18]

5.2 The Mathematics of Embeddings: Semantic Search

Embeddings are dense, fixed-length vectors—lists of floating-point numbers—that serve as the mathematical representation of the semantic meaning of a piece of text.[10] The importance of embeddings lies in their ability to facilitate semantic search. Traditional keyword-based search is rigid, failing when users employ synonyms or different phrasing. Embeddings

overcome this by mapping texts with similar underlying concepts or ideas closely together in a high-dimensional vector space. The search mechanism thus becomes centered on meaning rather than mere lexical matching.[10]

During the query phase, the similarity between the query vector and the document vectors is quantified using mathematical metrics, typically **Cosine Similarity**, which measures the angle between the vectors, or the Dot Product.[10] These metrics determine the degree of relevance, linking the user's abstract question to the document's specific content.

5.3 Vector Search Operations: Nearest Neighbor Requirements

Vector search is the rapid identification of the "Nearest Neighbors" to the query vector within the database. These neighbors are the document chunks that are mathematically closest in the vector space and, therefore, the most semantically relevant.[11] Since RAG operates in real-time, retrieving results in milliseconds is essential. This necessity mandates that the vector database implements highly optimized search algorithms, specifically **Approximate Nearest Neighbor (ANN)** techniques, such as HNSW or IVFFlat, to efficiently scale searches across indices containing millions of vectors. The optimization of this retrieval step is paramount for achieving acceptable system latency.[4]

5.4 LangChain's Role in Orchestration

LangChain defines the workflow by managing the sequential steps, or "chains," that link the various components of the RAG architecture.[6] The standard Q&A process is managed via a RetrievalQA Chain or a similar structured sequence, which manages the clean flow of data from the Retriever output directly to the LLM input.[5] Furthermore, LangChain supports the development of sophisticated, next-generation architectures, such as **Agentic RAG**.[22] This approach utilizes an **Agentic Router** with decision-making capabilities to dynamically determine the most effective retrieval path. For instance, the router can choose between a vector search for internal PDFs or a general web search tool, based on an analysis of the user query's specific context.[22]

VI. System Features and Design Specifications

The system blueprint mandates several core features necessary for an effective and reliable enterprise knowledge agent.

6.1 Multi-PDF and Multi-Document Support

The design is engineered to support the ingestion and indexing of multiple documents concurrently.[5] The Vector Database will store embeddings from numerous uploaded PDFs, either in a single consolidated index or within dedicated, segregated indices. Critically, the Retrieval & Generation Pipeline allows a single user query to perform a comprehensive search across the entire corpus of indexed knowledge, enabling the synthesis of complex answers derived from multiple source documents.

6.2 Advanced Context Management: Conversational Memory

To move beyond stateless, single-turn query answering, the system integrates **Conversational Memory**. LangChain components are leveraged to retain the history of the current interaction. Before processing a new query, the system utilizes the Gemini LLM to rewrite the incoming question, seamlessly integrating the previously established conversational context. For example, a follow-up query such as "What were the limitations of X?" is contextually augmented to clarify that "X" refers to a topic discussed in the preceding turn, enhancing the fluidity and utility of the interaction.[23]

6.3 Verification and Transparency: Source Citation and Grounded Responses

A cornerstone of enterprise AI adoption is trust, which is built on verifiability. The system employs stringent prompt engineering, instructing the Gemini LLM to generate responses that are **strictly grounded** in the provided retrieved context. The metadata associated with each retrieved chunk (including the file name and page number [19]) is passed to the LLM. The LLM is explicitly tasked with synthesizing and returning these **source citations** alongside the final answer, providing users with the necessary transparency to cross-check information and

confirm the factual basis of the output.[1]

6.4 File-Based Retrieval and Metadata Filtering

For situations where users need to target their query to specific documents, the system supports file-based retrieval. This is implemented by leveraging the document title stored in the chunk metadata. The Retriever is configured to apply a **metadata filter** during the similarity search, ensuring that only vectors where the metadata field matches the specified document title are searched. This technique drastically improves search precision and eliminates noise from unrelated documents.

VII. Expected Performance, Evaluation, and Constraints

To ensure the system is ready for production, performance must be evaluated across multiple dimensions: retrieval effectiveness, response quality, and underlying system efficiency.

7.1 Key Performance Indicators (KPIs)

Evaluation requires specialized metrics that measure the quality of the answer relative to the retrieval performance and the infrastructure cost.[15]

Table VII.1: RAG System Evaluation Metrics

Metric Category	Specific Metric	Description	Target Range
Response Quality	Groundedness	Measures if the response is entirely supported by the retrieved context	$> 90\%$

		(Anti-hallucination) [15]	
Response Quality	Factual Accuracy	Measures correspondence between the generated answer and the ground truth in the source [15]	$> 85\%$
Retrieval Efficacy	Recall@K	Proportion of relevant chunks retrieved in the Top K results [24]	K=5, Recall $> 75\%$
System Performance	Overall Latency	Total end-to-end response time (Query $\rightarrow$ Answer) [14]	Target < 4.0 seconds
System Performance	Cost/Query	Total token consumption and API cost per query [15]	Must be logged for OpEx control

7.2 Predicted Observations on Retrieval Quality

The system is expected to deliver high accuracy for queries that require direct fact lookup or simple extraction, particularly when the relevant context chunks are highly focused. However, performance variability is anticipated for complex, multi-hop queries that necessitate synthesizing and reasoning over facts scattered across numerous, non-contiguous chunks or different documents. This limitation points directly to the need for advanced retrieval techniques to maintain high consistency.

7.3 Limitations of Basic RAG Implementation

While powerful, the initial RAG implementation possesses inherent limitations that must be addressed in future phases. Firstly, the system's performance is heavily dependent on the quality of the external data and the ingestion/indexing strategy.[14] If the initial PDF parsing fails to accurately extract tables or preserve document structure, the subsequent RAG output will be compromised.

Secondly, RAG systems inherently incur increased complexity and latency due to the mandatory retrieval step.[14] This retrieval latency must be aggressively managed through architectural choices (e.g., FAISS) and algorithmic enhancements (e.g., reranking).

Finally, the MVP system, relying predominantly on text extraction, currently lacks the ability to interpret crucial information conveyed through visual elements—such as diagrams, organizational charts, or complex data tables embedded within the PDF.[25] Furthermore, a significant operational challenge for internal data is the difficulty and cost associated with generating the necessary labeled ground-truth datasets for rigorous evaluation. This gap necessitates the exploration of alternative methods, such as synthetic data generation or using LLM-as-a-Judge frameworks for performance monitoring.[24]

A critical operational requirement that informs system design is the management of costs. Given that both the LLM generation and the embedding steps utilize the Google Gemini API on a pay-per-use (token consumption) model, the overall cost per query is an essential metric.[15] High traffic, combined with frequent, resource-intensive monitoring calls (such as utilizing the LLM-as-a-Judge for ongoing performance checks), can lead to high operational expenditure (OpEx). Consequently, the system must integrate a robust logging layer to track and optimize the **Cost/Query** alongside traditional accuracy and latency metrics.

VIII. Roadmap and Strategic Enhancements

The system development is structured into phases, with the initial deployment focusing on the core text-based RAG workflow. Phase II and beyond prioritize sophisticated retrieval and the utilization of the Gemini API's advanced capabilities.

8.1 Phase II Enhancements: Advanced Retrieval Techniques

To address the limitations of basic Top-K similarity search, the following advanced retrieval strategies are planned:

1. **Query Expansion and Rewriting:** For queries that are vague or overly brief, the system will employ the LLM to generate multiple, refined versions of the original query. These diverse queries are then used to perform parallel retrieval, significantly improving the chances of surfacing comprehensive and highly relevant context.[27]
2. **Reranking Implementation:** A **Cross-Encoder Reranker** will be implemented immediately following the initial retrieval of the Top-K chunks. This smaller, dedicated model scores the pairwise relevance of the retrieved text chunks against the original query. This step refines the ranking, filtering out less relevant or noisy documents that were retrieved solely due to proximity in the vector space, ensuring that only the truly most pertinent context is passed to the resource-intensive Gemini LLM.[21] This process increases accuracy and boosts the Groundedness score.
3. **Hybrid Search:** The system will move toward a hybrid search model, combining the semantic strengths of vector search with the precision of traditional keyword search (e.g., integrating PostgreSQL text search). This combination is essential for handling complex technical queries that require simultaneous matching of abstract concepts and exact terminology (e.g., model numbers or specific names).[21]

8.2 Leveraging Gemini's Capabilities: Multi-Modal RAG for Visual PDF Content

The most significant strategic enhancement is the shift to **Multimodal RAG**. For complex enterprise documents, charts, tables, and diagrams often hold vital information that is invisible to text-only extraction.[25] The architecture must evolve to utilize the Gemini model's native vision capabilities for processing PDFs.[17]

Instead of relying on conventional document parsers for visual data, Gemini will be used to analyze and interpret images, diagrams, and complex tables within the PDF, generating detailed textual descriptions, summaries, or structured output.[23] These AI-generated descriptions of visual content are then chunked, vectorized, and indexed alongside the document text.[23] This approach allows users to ask grounded questions based on visual context, such as "Explain the Q3 revenue chart," successfully retrieving the necessary text description to synthesize a factual answer. Prioritizing this multimodal integration unlocks the full value of complex PDFs, transforming the system into a competitive, high-value enterprise asset.

8.3 Model Optimization and Fine-Tuning

Continuous improvement will involve systematic model optimization:

- **Advanced Embeddings:** A comparison of the current Gemini embedding model against newer or domain-specialized embeddings will determine if higher retrieval precision can be achieved.
- **Targeted Fine-Tuning:** While RAG avoids extensive foundational model retraining, limited fine-tuning (e.g., using LoRA techniques) may be explored to improve the LLM's adherence to specific desired output formats, ensuring consistency in citation style or structured data extraction, without compromising the core RAG grounding mechanism.[2]

8.4 UI and Deployment Improvements

The initial Streamlit UI serves as an excellent prototyping environment. However, long-term deployment mandates a transition to a robust, scalable cloud infrastructure (e.g., utilizing a high-performance framework like FastAPI deployed on Google Cloud Run) to manage high concurrency and ensure enterprise-level uptime. Furthermore, an essential step in maturing the system is the integration of RAG Ops—establishing continuous monitoring, tracking performance metrics (as defined in Section VII), and implementing user feedback mechanisms ("Was this answer helpful?") to create a vital telemetry data pipeline for ongoing optimization.[3]

IX. Final Summary and Project Synthesis

The deployment of the **PDF Question-Answering Agent using LangChain & Gemini API** establishes a sophisticated and highly functional knowledge retrieval system. By meticulously implementing the Retrieval-Augmented Generation (RAG) architecture, the project successfully harnesses the powerful orchestration capabilities of LangChain and the advanced generative and embedding services of the Google Gemini API. The system ensures that all generated answers are rigorously **factually grounded** in the source PDF content, dramatically increasing information accuracy and user confidence while minimizing the risk of LLM hallucination. The architectural blueprint is modular, enabling efficient scaling and component replacement. Coupled with a strategic roadmap that emphasizes advanced retrieval techniques (reranking, hybrid search) and the crucial integration of Gemini’s multimodal vision capabilities, this agent is positioned as a scalable, high-accuracy

foundation for transformative enterprise knowledge access.

Works cited

1. Retrieval-augmented generation - Wikipedia, accessed on November 13, 2025, https://en.wikipedia.org/wiki/Retrieval-augmented_generation
2. What is RAG (Retrieval Augmented Generation)? - IBM, accessed on November 13, 2025, https://www.ibm.com/think/topics/retrieval-augmented-generation
3. What is Retrieval-Augmented Generation (RAG)? - Google Cloud, accessed on November 13, 2025, https://cloud.google.com/use-cases/retrieval-augmented-generation
4. Build RAG Chatbot with LangChain, pgvector, Google Vertex AI Gemini 2.0 Pro, and Nomic Nomic Embed - Zilliz, accessed on November 13, 2025, https://zilliz.com/tutorials/rag/langchain-and-pgvector-and-google-vertex-ai-gemini-2.0-pro-and-nomic-nomic-embed
5. Analyzing Multiple PDFs Using a Generative RAG System and Chat GPT - YouTube, accessed on November 13, 2025, https://www.youtube.com/watch?v=qXR-9hd3_8U
6. Build a RAG agent with LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/langchain/rag
7. Build a semantic search engine with LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/langchain/knowledge-base
8. ChatGoogleGenerativeAI - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/integrations/chat/Generative_ai
9. Embeddings | Gemini API - Google AI for Developers, accessed on November 13, 2025, https://ai.google.dev/gemini-api/docs/embeddings
10. Embedding models - Docs by LangChain, accessed on November 13, 2025, https://docs.langchain.com/oss/python/integrations/text_embedding
11. RAG Diagram Guide: Visual Architecture of Retrieval-Augmented Generation - Latenode, accessed on November 13, 2025, https://latenode.com/blog/ai-frameworks-technical-infrastructure/rag-retrieval-augmented-generation/rag-diagram-guide-visual-architecture-of-retrieval-augmented-generation
12. accessed on November 13, 2025, https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb#:~:text=Therefore%2C%20vector%20store%20choice%20emerges,in%20context%20precision%20and%20recall.
13. Comparing RAG Part 2: Vector Stores; FAISS vs Chroma | by Stepkurniawan - Medium, accessed on November 13, 2025, https://medium.com/@stepkurniawan/comparing-faiss-with-chroma-vector-stores-0953e1e619eb
14. 15 Pros & Cons of Retrieval Augmented Generation (RAG) [2025] - DigitalDefynd, accessed on November 13, 2025, https://digitaldefynd.com/IQ/pros-cons-of-retrieval-augmented-generation/
15. Assess performance: Metrics that matter - Azure Databricks | Microsoft Learn,

accessed on November 13, 2025, https://learn.microsoft.com/en-us/azure/databricks/generative-ai/tutorials/ai-cookbook/evaluate-assess-performance
16. EONVERSE AI Intern – Screening Challenge.pdf
17. Better PDF OCR in RAG: Gemini 2.0 as a LangChain Loader - YouTube, accessed on November 13, 2025, https://www.youtube.com/watch?v=OmwbKbr5gVo
18. 11 Chunking Strategies for RAG — Simplified & Visualized | by Mastering LLM (Large Language Model), accessed on November 13, 2025, https://masteringllm.medium.com/11-chunking-strategies-for-rag-simplified-visualized-df0dbec8e373
19. Design and Develop a RAG Solution - Azure Architecture Center | Microsoft Learn, accessed on November 13, 2025, https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/rag/rag-solution-design-and-evaluation-guide
20. Finding the Best Chunking Strategy for Accurate AI Responses | NVIDIA Technical Blog, accessed on November 13, 2025, https://developer.nvidia.com/blog/finding-the-best-chunking-strategy-for-accurate-ai-responses/
21. 9 advanced RAG techniques to know & how to implement them - Meilisearch, accessed on November 13, 2025, https://www.meilisearch.com/blog/rag-techniques
22. Implementing Agentic RAG using Langchain and Gemini 2.0 - Reddit, accessed on November 13, 2025, https://www.reddit.com/r/Rag/comments/1i2320e/implementing_agentic_rag_using_langchain_and/
23. Building a Multimodal RAG System for PDF Files with Langchain4j, Gemini, and Hugging Face | by Marouane Lasmak | Medium, accessed on November 13, 2025, https://medium.com/@marouane.lasmak/building-a-multimodal-rag-system-for-pdf-files-with-langchain4j-gemini-and-hugging-face-399be16250cf
24. How do you evaluate RAG performance and monitor at scale? (PM perspective) - Reddit, accessed on November 13, 2025, https://www.reddit.com/r/Rag/comments/1n7em4z/how_do_you_evaluate_rag_performance_and_monitor/
25. Document understanding | Gemini API - Google AI for Developers, accessed on November 13, 2025, https://ai.google.dev/gemini-api/docs/document-processing
26. Seven Failure Points When Engineering a Retrieval Augmented Generation System - arXiv, accessed on November 13, 2025, https://arxiv.org/html/2401.05856v1
27. Improving RAG with Query expansion & reranking models | by Akash A Desai - Medium, accessed on November 13, 2025, https://aksdesai1998.medium.com/improving-rag-with-query-expansion-reranking-models-31d252856580