RAG Chatbot - A Comparative Study

Soorya Sivaramakrishnan
dept. of Computer Engineering
Sardar Patel Institute of Technology
Mumbai, India
soorya.sivaramakrishnan22@spit.ac.in

***Abstract*—This study presents a comprehensive evaluation framework for knowledge-intensive question answering systems, integrating document chunking, embedding generation, and retrieval-augmented generation (RAG). We compared multiple chunking strategies and identified Structural Balanced and Structural Hierarchical approaches as optimal for maintaining retrieval quality and index efficiency. Three embedding models—Qwen3-0.6B, Gemma-300M, and Voyage-3.5—were benchmarked, assessing recall, precision, MRR, latency, and index size. Subsequently, RAG evaluation with top-k retrieval, rerankers, and guardrails was conducted to measure answer quality, semantic similarity, faithfulness, and relevance. The results highlight the trade-offs between embedding quality, computational performance, and retrieval accuracy, providing actionable insights for system design and deployment. This work establishes a robust methodology for systematically evaluating and optimizing retrieval and generation pipelines in large-scale knowledge-based applications.**

***Index Terms*—RAG, Chunking Strategy, Embedding Models, Recall@k, Precision@k, MRR, Latency, Index Size, Top-k Retrieval, Reranker, Guardrails, Semantic Similarity, Faithfulness, Completeness.**

I. Introduction

In recent years, conversational AI has transformed customer support by providing instant, accurate, and scalable responses. Traditional chatbots often rely on predefined rules or static FAQs, limiting their ability to handle complex or unforeseen queries. Retrieval-Augmented Generation (RAG) offers a promising alternative by combining information retrieval with large language models (LLMs), enabling the generation of contextually accurate and citation-backed answers.

This project focuses on building a production-grade RAG chatbot for JioPay, leveraging publicly accessible business and help center data. The system not only retrieves relevant information from the knowledge base but also generates natural language responses while providing transparent citations. Through ablation studies on chunking strategies, embeddings, and ingestion pipelines, this work explores the optimal configurations for performance, accuracy, and efficiency, culminating in a deployable web application accessible via a public URL.

II. Methedology

The development process followed a structured pipeline, beginning with the collection of publicly available JioPay data from the business website and help center. The raw content was cleaned, normalized, and segmented using multiple chunking strategies to enable effective retrieval. Embeddings were generated using different state-of-the-art models and stored in a vector database to support dense similarity search. A retriever module was designed to fetch the most relevant chunks, which were then passed to a large language model for grounded response generation with inline citations. Finally, a user-friendly web interface was built to deliver responses interactively, while ablation studies were conducted to evaluate the impact of chunking, embeddings, and ingestion pipelines on performance and accuracy.

A. System Overview

The proposed system implements a production-grade Retrieval-Augmented Generation (RAG) pipeline to automate customer support for JioPay. The workflow begins with data ingestion, where Playwright was used to scrape dynamic content from the JioPay business website and help center. The extracted data was then cleaned and structured into a normalized format.

For knowledge base construction, a Structural-Hierarchical chunking strategy was adopted to preserve contextual hierarchy from headings and sub-sections, ensuring that responses remain grounded in the document structure. These chunks were transformed into dense vector representations using Voyage embeddings (MongoDB), which balance semantic quality with indexing efficiency. The embeddings were stored in a vector database for high-performance retrieval.

The retrieval module performs top-k similarity searches to fetch the most relevant context. These retrieved chunks are passed into Google Gemini 2.0, which generates natural language responses while anchoring outputs to the supporting sources. Inline citations referencing original URLs are included to enhance transparency and reduce hallucinations.

On the deployment side, the system is divided into two layers: a backend implemented in Python to handle ingestion, embedding, retrieval, and LLM integration, and a frontend developed using TypeScript + React to provide an interactive chat interface. The frontend is hosted on Vercel, while the backend is deployed on Render, ensuring scalability, availability, and a clean separation of concerns.

Together, this architecture enables reliable, accurate, and explainable customer support responses, backed by systematic evaluation through ablation studies.

B. Data Collection

To construct the knowledge base for the RAG system, publicly available content from the JioPay Business website

and Help Center/FAQs was collected using a combination of Playwright and Puppeteer. Both tools were used on the same set of links to maximize coverage and reliability.

- Playwright: Proved to be more robust and easier to implement. It successfully extracted the majority of textual content and was able to capture associated images as well. However, it did not retrieve all PDF documents, requiring additional handling.
- Puppeteer: Faced some challenges with webpage loading, often requiring multiple runs to complete scraping. Despite this, it managed to capture nearly all PDFs, giving it an advantage in document completeness, though with slightly higher operational overhead.

Additionally, the FAQ section posed difficulties since its content could not be directly scraped with standard methods. A custom script had to be written to extract the FAQ data effectively. PDF resources linked on the site were also downloaded separately to ensure comprehensive coverage of all relevant materials.

Together, the dual-scraping approach combined the strengths of Playwright and Puppeteer, yielding a cleaner and more complete dataset than either could achieve independently.

The jiopay website had no explicit robots.txt page on it. Searching for other sources, including the official Jio website, revealed no robots for the site.

C. Chunking Ablation

Chunking Ablation

Chunking defines how the raw text is segmented before embedding and indexing. Since different strategies can influence retrieval quality and efficiency in distinct ways, multiple approaches were implemented for comparison.

1) Fixed Chunking The text was divided into fixed-length segments, independent of semantics or structure. Three settings were tested:
   - 256 tokens, 0 overlap → Creates small, non-overlapping segments for fine-grained retrieval.
   - 512 tokens, 64 overlap → Uses moderate-length chunks with slight overlap to retain continuity.
   - 1024 tokens, 128 overlap → Produces larger segments with more context at the cost of index size.

   This method provides predictable segment sizes and straightforward indexing.
2) LLM-Based Chunking Instruction-aware segmentation was performed using a large language model, with each chunk capped at a maximum of 256 tokens. This approach produces variable chunk sizes depending on the LLM's interpretation of semantic boundaries.
3) Recursive Chunking A fallback approach was applied where text was initially divided into large units and then recursively split into smaller units if the size exceeded thresholds. Three configurations were tested:
   - Large → Prioritizes fewer, broader segments before fallback splitting.
   - Balanced → Uses a mid-level segmentation with gradual fallback.
   - Small → Generates fine-grained segments by recursively breaking down text to smaller units.

   This ensured no text block exceeded the target size while retaining coverage of the full document.
4) Semantic Chunking Segmentation was guided by sentence- and paragraph-level embeddings, merging based on similarity thresholds. Three threshold settings were applied:
   - High similarity → Merges sentences/paragraphs only when strongly related.
   - Medium similarity → Balances semantic closeness with coverage.
   - Low similarity → Groups loosely related text into larger units for broader context.

   This resulted in variable-length chunks shaped by semantic coherence.

   5. Structural Chunking Segmentation was aligned with the document hierarchy, preserving boundaries such as headings, sub-sections, and HTML tags. Three settings were applied:
   - Balanced → Splits by headings and sub-sections while maintaining moderate chunk size.
   - Hierarchical → Preserves full document hierarchy, aligning chunks with nested structures.
   - Large → Produces extended segments by combining higher-level structural blocks.

   This method followed the structural divisions of the source documents, resulting in semantically grouped units of text.

D. Emebdding Ablation

Embeddings convert textual chunks into dense vector representations, enabling semantic similarity search in the RAG pipeline. To evaluate the effect of embedding model choice on retrieval and generation, three models were selected and tested with two different chunking strategies: Structural-Hierarchical and Structural-Balanced.

1) Google Embedding GEMMA 300M This proprietary embedding model produces vectors of 768 dimensions. It is optimized for semantic similarity tasks and provides efficient performance in commercial use cases. It was used with both selected chunking strategies to evaluate its compatibility with hierarchical and balanced segmentation.
2) Qwen3-0.6B An open-source embedding model with 1024-dimensional vectors, Qwen3-0.6B is designed to capture richer semantic representations. Its higher-dimensional outputs allow for potentially finer-grained similarity detection between chunks, though at the cost of increased storage and computation. Both chunking methods were paired with this model to observe its behavior on varying chunk structures.

3) Voyage by MongoDB Voyage is an proprietary embedding model also producing 1024-dimensional vectors, with a focus on scalable vector storage and retrieval in document-heavy environments. It was tested with both Structural-Hierarchical and Structural-Balanced chunking to analyze retrieval quality and integration with MongoDB's vector infrastructure.

The primary objectives of this ablation study were:

- Open-source vs Proprietary → To compare licensing, ease of integration, and retrieval performance between models.
- Embedding dimensionality → To evaluate whether higher-dimensional embeddings provide improved semantic representation and retrieval accuracy compared to smaller vectors.

By systematically combining chunking strategies with these three embedding models, the study aims to characterize the trade-offs in retrieval fidelity, index size, and computation cost, providing guidance for selecting embeddings in production RAG systems.

E. Retrieval + Generation

The retrieval and generation pipeline is central to the RAG system, enabling user queries to be answered accurately and transparently using the structured knowledge base. It is designed to separate embedding, retrieval, and generation components for modularity, scalability, and evaluation.

1) Query Embedding User queries are first transformed into dense vector representations using the Voyage embedding model. The model processes the query text and produces a high-dimensional embedding that captures semantic meaning. These embeddings allow the system to perform similarity-based search over the knowledge base, independent of exact keyword matches.
2) Vector Search and Retrieval The query embedding is submitted to a Pinecone vector index, which contains embeddings for all document chunks generated from the data ingestion and chunking stages. A top-k retrieval is performed to select the most relevant chunks. Each retrieved chunk includes metadata such as source title, source URL, and a text snippet. This allows the system to provide context-aware responses while preserving traceability of information.
3) Context Construction The retrieved chunks are formatted into a structured context for the LLM. Each chunk is enumerated and includes its similarity score, source title, and text content. Parallelly, a list of citation objects is created, which includes the chunk text (truncated for display), source title, and URL. This ensures that the system can provide both the answer and referenceable sources for transparency.
4) Prompt Engineering for Generation The structured context is combined with a prompt instructing Google Gemini 2.0 to generate responses strictly based on the retrieved context. The prompt specifies the desired tone (professional and helpful), output format, and instructions to include citation markers. The generation parameters, including temperature and maximum output tokens, are configured to balance informativeness with conciseness.
5) Response Assembly The model's output is processed to produce the final answer text. The structured citation list is also returned, allowing the frontend to display retrieved sources alongside the answer. This approach ensures that responses are grounded in the knowledge base and can be traced back to original documents.
6) API Integration The retrieval and generation pipeline is exposed via a FastAPI backend. It includes endpoints for:
   /api/chat → Processes user queries and returns responses with citations.
   /api/stats → Provides statistics on the vector index, such as total vectors, dimensions, and index fullness.
   /api/test-retrieve → Returns raw retrieval results for testing and debugging.
7) Logging and Monitoring Comprehensive logging is implemented to track query execution time, retrieved chunk metadata, and response generation. This allows monitoring of latency, debugging of retrieval anomalies, and analysis of system performance during ablation studies.

This design enables modular experimentation with different chunking strategies, embeddings, and generative models while maintaining a robust, production-ready architecture suitable for deployment.

F. Deployment

The system was deployed in a cloud-based configuration with a clear separation between the backend and frontend. The backend, implemented in Python, was hosted on the Render free tier with a memory allocation of 512 MB RAM. It integrates with external APIs, including the Google Gemini API (gemini-2.0-flash), which supports up to 15 requests per minute (RPM), 1,000,000 tokens per minute (TPM), and 200 requests per day (RPD). For embeddings, the backend connects to Voyage AI (MongoDB) with usage limits of 3 RPM and 10,000 TPM, and it interfaces with Pinecone for vector storage and retrieval.

The frontend was developed using TypeScript and React and deployed on Vercel, providing a responsive web interface for end users. Communication between the frontend and backend is managed through RESTful API endpoints, with CORS enabled for secure cross-origin requests.

This deployment setup ensures accessibility via a public URL while maintaining scalability and modularity across the system components.

III. Results

Following the development of the RAG system, including data ingestion, chunking, embedding, and retrieval-generation components, the next step involves evaluating the system's performance. The evaluation focuses on the effectiveness of different chunking and embedding strategies, the quality of retrieved context, and the fidelity of generated responses. The

results presented below summarize the outcomes of these experiments, highlighting system behavior across multiple configurations.

A. Scraping Ablation

To evaluate ingestion performance, two scraping pipelines were compared: Playwright and Puppeteer. Both were applied on the same set of JioPay business and help center pages, with metrics recorded for the number of pages processed, total tokens extracted, proportion of noisy text, throughput, and failure rates.

Playwright was tested for its robustness in handling dynamic content and its ability to extract both text and images.

Puppeteer was tested for its coverage of documents, particularly in capturing PDFs, despite requiring multiple runs in certain cases.

The results are summarized in Table 1, providing a quantitative basis for analyzing trade-offs in completeness, robustness, and efficiency between the two scraping approaches.

TABLE I
COMPARISON OF PLAYWRIGHT AND PUPPETEER SCRAPING RESULTS

Metric	Playwright	Puppeteer
Pages (Raw)	28	28
Pages (Cleaned)	25	25
Tokens (Raw)	563,620	563,085
Tokens (Cleaned)	36,282	36,282
Noise (%)	93.56	93.56
Throughput (tokens/page)	1451.28	1451.28
Failures (%)	0.00*	0.00*

* Failures not reflected in cleaned dataset; Puppeteer required 2–3 retries and occasional manual reloads.

- Playwright successfully completed the scraping in a single run, demonstrating greater robustness against dynamic loading delays.
- Puppeteer often required 2–3 execution attempts for the same dataset. Failures occurred when sites took longer to load, causing Puppeteer to prematurely close the browser. In some cases, manual reloading was required to complete the scrape.
- In terms of data quality, both frameworks produced nearly identical cleaned datasets. The noise reduction was significant (93.5%), indicating that cleaning effectively filtered out irrelevant boilerplate content.
- Throughput was consistent across both tools at around 1450 tokens per page, reflecting stable processing efficiency once data was successfully extracted.
- The failure rate was effectively zero in the cleaned dataset, but observationally, Puppeteer exhibited higher practical failures due to retries and manual intervention.
- PDF Extraction: Puppeteer was more effective at capturing PDFs, managing to fetch nearly all of them, while Playwright failed to consistently scrape embedded PDF resources.

B. Chunking Ablation

The evaluation employed a unified, multi-dimensional framework designed to balance domain-specific content quality with retrieval effectiveness, size optimization, and performance benchmarking. This approach integrates two complementary paradigms: domain-specific RAG quality assessment, which emphasizes semantic accuracy and contextual adequacy of retrieved content, and retrieval performance evaluation, which focuses on the effectiveness of retrieval in answering task-specific queries.

A weighted scoring system was adopted to aggregate results across dimensions. Content quality and domain relevance (RAG Quality) contributed 45% to the overall score, retrieval effectiveness metrics accounted for 40%, chunk size optimization contributed 8%, and processing efficiency and latency comprised the remaining 7%. This balance ensured that strategies were judged not only on retrieval accuracy but also on the adequacy, compactness, and efficiency of their outputs.

RAG quality assessment was operationalized through five metric groups. Semantic coherence was evaluated by detecting broken or fragmented sentences, measuring topic consistency against JioPay domain-specific keywords, and assessing completeness of thought in Q&A and procedural content. Context completeness was examined across content types, with standards tailored to FAQs (Q&A pair integrity), PDF documents (section completeness within 30–200+ word thresholds), and web content (contextual adequacy by word count). Information density was quantified through keyword presence, numerical detail, procedural instruction coverage, and normalized density scores. Topic coverage analysis measured cross-category balance across JioPay domains, including payments, app usage, business, support, and features. Finally, FAQ grouping quality was assessed based on clustering coherence, ensuring related queries and answers were grouped appropriately within domain contexts.

Retrieval performance metrics followed standard information retrieval practices, including Precision@1 for top-ranked relevance, Precision@K and Recall@K for coverage and breadth, and Mean Reciprocal Rank (MRR) for evaluating the ranking position of the first relevant result. Chunk size optimization was guided by a framework defining an optimal range of 80–400 tokens and an acceptable range of 50–600 tokens, with graduated penalties applied for deviations. Performance benchmarking captured query processing latency (mean and tail latencies), throughput in queries per second, and normalized scores to enable fair comparison.

The test dataset comprised fifteen comprehensive queries reflecting real-world JioPay usage across five categories—onboarding, payments, KYC, API/security, pricing, and features. Queries were stratified by complexity into simple, medium, and complex cases, with ground truth topic mappings defined for relevance validation.

The evaluation process followed a structured flow: data loading and normalization across chunking formats, statistical analysis of token and character distributions, domain-specific quality assessment using the five RAG dimensions, retrieval simulation with TF-IDF similarity, performance benchmarking, and final weighted score aggregation. Comparative analysis across strategies was conducted to produce multi-strategy

rankings.

Results were reported through a structured framework comprising an executive summary highlighting the best-performing strategies, detailed rankings with weighted scores, per-strategy metric breakdowns, and category-level analyses by query type. Multi-dimensional radar charts and other visualizations were employed to provide interpretable comparisons, and deployment readiness metadata was generated to assess production applicability.

Although the unified evaluation identified LLM_smallChunks as the top-performing strategy (Final Score 0.742, RAG Quality 0.680, Retrieval Performance 0.925), practical considerations regarding chunk sizes rendered it less suitable for deployment. Consequently, the next best-performing strategies, Structural_Hierarchical (Final Score 0.725) and Structural_Balanced (Final Score 0.724), were selected for implementation. These strategies offer a balanced trade-off between retrieval effectiveness, semantic coherence, and manageable chunk sizes, making them more appropriate for real-world deployment scenarios.

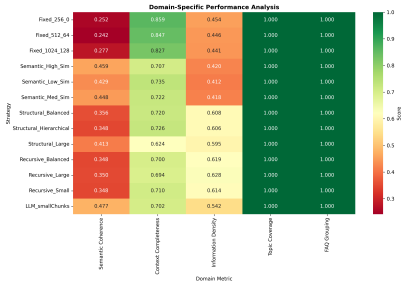

Fig. 1. Domain Wise Performance Heatmap

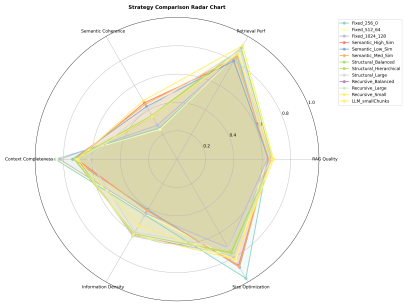

Fig. 2. Strategy Radar Chart

Fig. 3. Retrieval by Category Metrics

C. Embedding Evaluation

Following the selection of Structural Balanced and Structural Hierarchical as the most viable chunking strategies, three embedding models were evaluated: Qwen3 0.6B (1024 dimensions), Google EmbeddingGemma-300M (768 dimensions), and Voyage-3.5 (1024 dimensions). The evaluation considered retrieval quality (recall@5, mean reciprocal rank, precision@5) as well as efficiency (index size, average latency, and success rate). The embedding models were chosen from HuggingFace's MTEB leaderboard for their high retrieval accuracy.

The embedding evaluation was conducted by applying each model to the outputs of the two selected chunking strategies (Structural Hierarchical and Structural Balanced). The embeddings generated by each model were indexed, and retrieval performance was assessed using a standardized query set.

Performance was measured across multiple metrics, including Recall@5, Precision@1, Mean Reciprocal Rank (MRR), and Success Rate, alongside efficiency indicators such as average query latency and queries per second (QPS). In addition, index size was recorded to assess storage overhead.

All experiments were executed within a unified framework, ensuring consistency in preprocessing, indexing, and benchmarking. The results were consolidated into comparative tables to facilitate a fair evaluation of each embedding model's effectiveness and efficiency.

Across both chunking strategies, Voyage-3.5 consistently achieved the strongest retrieval effectiveness, though at the cost of significantly higher latency, rendering it impractical for real-time deployments. Qwen3 0.6B provided a balanced trade-off between accuracy and efficiency, while EmbeddingGemma-300M emphasized speed and reliability, albeit with lower retrieval quality. This contrast highlights the trade-offs between retrieval strength and computational efficiency in practical embedding model selection.

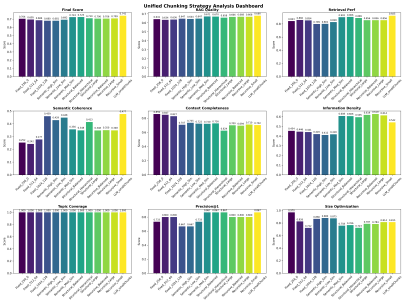

Fig. 4. Unified Metrics Graphs

D. RAG Evaluation

The final evaluation stage integrated both retrieval and generation. Queries were retrieved, reranked, and passed to the language model with guardrails applied. The generated responses were then directly compared against ground-truth answers scraped from the source knowledge base.

This evaluation measured answer quality, retrieval effectiveness, and system performance. The consolidated results are presented in Table VI.

TABLE II
CHUNK SIZE ANALYSIS ACROSS DIFFERENT STRATEGIES

Strategy	Total Chunks	Min Tokens	Max Tokens	Avg Tokens	Median Tokens	Std Tokens
Fixed_256_0	642	1	258	219.9	256.0	74.4
Fixed_512_64	415	17	514	382.8	512.0	193.7
Fixed_1024_128	262	17	1025	599.0	958.0	449.8
Semantic_High_Sim	1486	6	833	93.3	72.0	80.3
Semantic_Low_Sim	1065	8	833	130.2	84.0	127.5
Semantic_Med_Sim	1320	6	833	105.1	77.0	91.5
Structural_Balanced	368	3	1499	235.1	69.5	387.6
Structural_Hierarchical	380	3	1197	227.7	72.0	341.1
Structural_Large	489	1	13693	175.8	32.0	740.2
Recursive_Balanced	619	1	1018	158.3	61.0	200.6
Recursive_Large	600	1	1258	163.3	59.5	227.7
Recursive_Small	654	1	758	150.0	69.5	171.1
LLM_smallChunks	584	14	16627	107.1	52.0	898.6

TABLE III
UNIFIED CHUNKING STRATEGY - CONTENT AND RETRIEVAL METRICS

Strategy	Semantic Coh	Context Comp	Info Density	Topic Cov	Precision@1	Recall@5	MRR	Latency (ms)	QPS
Fixed_256_0	0.252	0.859	0.454	1.000	0.733	0.944	0.856	54.49	18.4
Fixed_512_64	0.242	0.847	0.446	1.000	0.800	0.928	0.872	75.09	13.3
Fixed_1024_128	0.277	0.827	0.441	1.000	0.800	0.906	0.883	63.13	15.8
Semantic_High_Sim	0.459	0.707	0.420	1.000	0.667	0.944	0.811	71.83	13.9
Semantic_Low_Sim	0.429	0.735	0.412	1.000	0.667	0.939	0.811	68.95	14.5
Semantic_Med_Sim	0.448	0.722	0.418	1.000	0.733	0.922	0.856	66.98	14.9
Structural_Balanced	0.356	0.720	0.608	1.000	0.867	0.967	0.922	63.68	15.7
Structural_Hierarchical	0.348	0.726	0.606	1.000	0.867	0.967	0.922	68.97	14.5
Structural_Large	0.413	0.624	0.595	1.000	0.867	0.944	0.933	63.46	15.8
Recursive_Balanced	0.348	0.700	0.619	1.000	0.800	0.939	0.900	62.47	16.0
Recursive_Large	0.350	0.694	0.628	1.000	0.800	0.939	0.900	63.57	15.7
Recursive_Small	0.348	0.710	0.614	1.000	0.800	0.939	0.900	55.67	18.0
LLM_smallChunks	0.477	0.702	0.542	1.000	0.867	0.967	0.933	36.83	27.1

TABLE IV
UNIFIED CHUNKING STRATEGY - CORE METRICS

Strategy	Final Score	RAG Quality	Retrieval Perf	Size Opt	Performance
Fixed_256_0	0.704	0.641	0.843	0.970	0.003
Fixed_512_64	0.695	0.634	0.860	0.826	0.002
Fixed_1024_128	0.686	0.636	0.854	0.722	0.003
Semantic_High_Sim	0.680	0.647	0.799	0.866	0.002
Semantic_Low_Sim	0.681	0.644	0.801	0.880	0.002
Semantic_Med_Sim	0.692	0.647	0.828	0.875	0.002
Structural_Balanced	0.724	0.671	0.902	0.758	0.002
Structural_Hierarchical	0.725	0.670	0.906	0.766	0.002
Structural_Large	0.710	0.658	0.889	0.723	0.002
Recursive_Balanced	0.706	0.666	0.856	0.787	0.003
Recursive_Large	0.706	0.668	0.856	0.784	0.002
Recursive_Small	0.709	0.668	0.856	0.814	0.003
LLM_smallChunks	0.742	0.680	0.925	0.815	0.007

TABLE V
EMBEDDING MODEL EVALUATION ON STRUCTURAL BALANCED AND STRUCTURAL HIERARCHICAL STRATEGIES

Strategy	Model	Recall@5	MRR	Precision@5	Index Size (MB)	Latency (ms)	Success Rate (%)
Structural Balanced	Qwen3-0.6B	0.396	0.367	0.347	420	119.6	83.3
	Gemma-300M	0.271	0.379	0.271	315	50.8	91.7
	Voyage-3.5	0.958	0.903	0.883	512	18900	66.7
Structural Hierarchical	Qwen3-0.6B	0.354	0.337	0.319	422	120.1	83.3
	Gemma-300M	0.250	0.360	0.250	317	52.3	91.7
	Voyage-3.5	0.958	0.901	0.875	514	19020	66.7

TABLE VI
FINAL RAG EVALUATION RESULTS WITH ANSWER QUALITY, RETRIEVAL, AND SYSTEM PERFORMANCE METRICS

Category	Metric	Value
Answer Quality	Overall Quality Grade	A (0.715)
	Semantic Similarity	0.756
	Faithfulness	0.680
	Completeness	0.717
	Relevance	0.677
	Exact Match	0.469
Grade Distribution	A	55.0%
	B	40.0%
	C	0.0%
	D	5.0%
	F	0.0%
Retrieval	Similarity Score	0.756
	Threshold Pass Rate	95.0%
	Precision@1	0.743
	Mean Reciprocal Rank	0.756
System Performance	Latency (Mean)	2863 ms
	Latency (P50)	2769 ms
	Latency (P95)	3480 ms
	Memory Usage	404.7 MB
	CPU Usage	20.3%

IV. LIMITATIONS AND FUTURE WORK

A. Limitations:

- Latency Constraints: The evaluation revealed that certain embedding models, particularly larger ones like Voyage-3.5, exhibit high query latency, which may impact real-time application performance.
- Memory and Index Size: Embedding and retrieval with large models increase memory usage and storage requirements, potentially limiting deployment on resource-constrained environments.
- Query Coverage: While our evaluation used representative queries, some domain-specific or rare queries may not be fully captured, leading to potential gaps in retrieval and generation quality.
- Dependency on Ground-Truth Data: RAG evaluation relies on pre-scraped ground-truth answers. Incomplete or noisy ground-truth data can influence quality metrics such as recall, precision, and exact match.
- Model Generalization: The embedding and generation models are pre-trained and may not fully generalize to unseen or evolving topics beyond the test corpus.

B. Future Work:

- Adaptive Retrieval Strategies: Implement dynamic top-k selection, query-specific reranking, or hybrid retrieval mechanisms to improve relevance and reduce latency.
- Model Compression and Optimization: Explore quantization, pruning, or knowledge distillation to reduce memory footprint and speed up inference for larger models.
- Enhanced Evaluation Metrics: Incorporate human-in-the-loop evaluations, adversarial queries, and multi-dimensional metrics (faithfulness, factuality, contextual accuracy) to better capture quality.
- Domain-Specific Fine-Tuning: Fine-tune embeddings and generative models on domain-specific corpora to improve recall, semantic similarity, and answer faithfulness.
- Multi-Stage Pipelines: Investigate multi-stage retrieval and generation pipelines with intermediate rerankers or confidence scoring to increase robustness.
- Scalable Deployment: Evaluate distributed inference and parallel retrieval strategies to support larger query volumes with acceptable latency and cost.

V. CONCLUSION

In this study, we conducted a comprehensive evaluation of chunking, embedding, and retrieval-augmented generation (RAG) strategies for knowledge-intensive question answering. Structural Balanced and Structural Hierarchical chunking strategies were selected based on their strong trade-off between retrieval performance and index efficiency. Multiple embedding models, including Qwen3-0.6B, Gemma-300M, and Voyage-3.5, were benchmarked, highlighting the balance between accuracy, latency, and memory footprint.

The RAG evaluation demonstrated that combining high-quality embeddings with top-k retrieval, rerankers, and guardrails produces reliable and semantically relevant answers, with Voyage-3.5 achieving the highest quality metrics albeit at higher latency. Our framework provides a systematic methodology for evaluating retrieval and generation performance across multiple models, offering insights into model selection and deployment trade-offs.

Overall, the study emphasizes the importance of careful chunking, embedding selection, and retrieval strategy design to achieve a robust and efficient knowledge retrieval system. The results serve as a foundation for further optimization, including latency reduction, domain-specific fine-tuning, and adaptive retrieval strategies.