MULTIMODAL RETRIEVAL - AUGMENTED GENERATION SYSTEM

By Denys Yuvzhenko

Presented in Partial Fulfilment of the Requirements for the Degree

Master of Software Engineering

American University Kyiv

2025

APPROVED BY:

Viktor Putrenko, Dr. Habil., Prof

Table of contents

Abstract

This study presents an asynchronous, web-based Retrieval-Augmented Generation (RAG) system that integrates multimodal inputs (text, images, tables) to enhance information retrieval and generation in various contexts. The system is developed in Python and hosted on AWS, combining Chroma DB for vector storage and Anthropic's Claude-3-Haiku LLM model accessible via AWS Bedrock. By leveraging modern cloud capabilities, the solution scales efficiently and handles diverse data modalities in real time.

Through systematic experiments, this project highlights the effectiveness of multimodal embedding techniques for refining retrieval accuracy and providing context-aware responses. The architecture's modular design supports seamless feature integration, making it adaptable for different use cases such as customer support, educational tools, and content creation. Key findings emphasize the role of vector databases in dynamic information updates and confirm that large language models, when appropriately curated and grounded, can produce high-quality, relevant outputs.

Chapter 1

1.1 Introduction

Today's AI-driven solutions are rapidly reshaping how we find, process, and interpret information across diverse fields—ranging from business intelligence and customer support to research and beyond [1][2]. Key to these advancements is the emergence of powerful techniques like Retrieval-Augmented Generation (RAG), which seamlessly merges real-time data retrieval with generative capabilities to produce context-rich outputs.

Central to this project is the design and implementation of a multimodal RAG [2] system. By leveraging a vector database for efficient data storage and retrieval, coupled with an advanced LLM for generation, the system can handle various data formats (text, images, tables) asynchronously. This approach offers scalable, on-demand insights applicable to everything from market analysis and automated content creation to domain-specific research.

Beyond its technical merits, the system highlights important considerations around security, data privacy, and responsible AI practices. These factors are becoming increasingly critical as organizations integrate AI solutions into their workflows, whether to streamline customer service, refine decision-making processes, or create adaptive learning tools.

Overall, this diploma project underscores the transformative potential of integrating modern AI architectures into real-world operations. By showcasing flexible data handling, robust retrieval, and domain-agnostic generation, it offers a blueprint for harnessing AI's capabilities across industries—paving the way for more responsive, intelligent applications that drive innovation and efficiency at scale.

1.2 Overview of Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) is an approach that combines a knowledge retrieval mechanism with a generative model to produce contextually accurate and up-to-date outputs [3]. Instead of relying solely on a pre-trained model's internal parameters, RAG pulls relevant data from external sources—often stored in vector databases—before generating a response. This setup helps deliver more precise, domain-focused information for tasks as varied as customer support, data analysis, market research, and content generation.

In practice, RAG typically operates in two main phases: retrieval and generation [4]. The retrieval component searches through indexed material (e.g., documents, FAQs, or multimodal data stored in a vector database like Chroma DB) to identify the best matching context. The generation component then uses that context to craft a response with greater relevance than a standalone Large Language Model (LLM) might achieve. This architecture makes RAG systems highly adaptable, enabling them to serve unique needs across various industries—such as real-time market trend analysis, complex report generation, or specialized research tasks.

By integrating retrieval and generation in a unified pipeline, RAG solutions can provide responsive, high-fidelity answers while reducing hallucinations and inaccuracies. As companies and organizations continue to explore AI-driven workflows, RAG offers a powerful framework for building applications that harness large language models—like Claude-3-Haiku—while maintaining a sharp focus on the most relevant data.

1.3 Multimodal Approaches in Modern AI Systems

Multimodal AI systems integrate various data types—such as text, images, audio, or even sensor readings—to deliver richer, more context-aware responses [5][6]. Instead of processing a single input channel, these systems correlate multiple

sources of information to improve accuracy, enhance user experiences, and unlock use cases spanning industry, research, and consumer applications.

By examining details from different modalities, AI models can capture nuances that a text-only or image-only system might miss. For example, a support bot might leverage text transcripts alongside product images to better understand and troubleshoot user queries, while a market analysis tool could fuse social media sentiment (text) with satellite imagery of store traffic to gain a competitive edge. In research contexts, combining scientific papers (text) with lab instrument readings (numeric data) or microscopy images (visual data) can accelerate discoveries and help validate hypotheses more efficiently.

Multimodal setups often involve specialized pipelines for each data format. After preprocessing, features are merged through a unified embedding space or a transformer-based architecture that can handle multiple input types concurrently. This enables systems to dynamically adapt to diverse data sources, delivering responses that closely align with real-world scenarios. As the field evolves, the integration of multimodal techniques with Retrieval-Augmented Generation (RAG) further refines content accuracy and extends application reach—ensuring that high-quality, tailored information is available to users across a wide array of domains and platforms.

A crucial aspect of building these systems is storing and managing multimodal data in vector databases. In practice, each data type (e.g., text, image, audio) is transformed into a high-dimensional numeric embedding before being indexed. A database such as Chroma DB then uses similarity search methods (often approximate nearest neighbor techniques) to quickly retrieve the most relevant embeddings. For instance, an e-commerce platform might store both product descriptions (text) and images (visual data) as separate but interlinked embeddings, allowing a combined search approach that yields more precise results when generating recommendations.

Once the relevant embeddings are identified, the Retrieval-Augmented Generation (RAG) pipeline fuses them into a unified context for a generative model. The model, aware of these multiple modalities, can produce outputs that account for textual cues, visual patterns, or even audio signals. This leads to more holistic and accurate responses, whether the system is addressing complex customer queries, supporting data-driven decision-making, or facilitating advanced research tasks. By bridging different data types under one search and generation framework, multimodal AI systems enable truly comprehensive insights and solutions in real time.

1.4 State of the Art in RAG

This area of work and research is highly new. In fact, we can only talk about a few years of intensive work an in this area. Therefore, with a non-zero probability, despite the fact that I have used the latest technologies and the most modern approaches, by the time you read this text, the rapid development of technology may make this work outdated.
Retrieval-Augmented Generation (RAG) represents a cutting-edge paradigm in natural language processing (NLP) [8] and information retrieval. It enhances traditional generative models by integrating an external knowledge source—usually stored in a vector database—to provide up-to-date, context-specific answers. This approach not only mitigates the problem of "model hallucination" (where the model fabricates facts) but also allows highly accurate, domain-specific responses [16].

1.4.1 Early and Foundational Work

Open-Domain Question Answering (QA): Early RAG-like techniques emerged in QA tasks, where a language model would consult an indexed set of documents

to find relevant passages before generating an answer. Systems such as DrQA (Facebook AI Research) and REALM (Google) experimented with retrieving textual chunks to serve as context for generation 111.

Dense Passage Retrieval [4] (DPR): This approach leverages dual encoders—one for the query and one for the documents—to create vector embeddings that can be compared efficiently. DPR was instrumental in showing how retrieval significantly improves end-to-end QA performance 222.

1.4.2 Modern RAG Architectures

Contemporary RAG systems rely heavily on transformer-based architectures—like GPT or Claude—combined with advanced retrieval mechanisms. Below I collected are some widely adopted designs and best practices:

Retriever-Generator Pipeline. This is one of the most commonly implemented workflows, where the system is split into two distinct modules—a retriever and a generator.

Retriever: Focuses on scanning a vector database (such as Chroma DB, Milvus, or Pinecone) to find semantically relevant embeddings. Rather than just text, these embeddings may represent a wide array of data modalities, including paragraphs from PDF documents, images extracted from product catalogs, or even short audio transcriptions. The retriever's primary responsibility is to convert the user query into a vector and efficiently retrieve the top-k matches via Approximate Nearest Neighbor (ANN) search.

Generator: Once the retriever returns the most relevant items, the generator (often a transformer-based Large Language Model like GPT-4 [9], Claude [10], or a Llama-derivative) synthesizes a coherent answer that references the retrieved context. This architectural split ensures the LLM has direct, up-to-date

knowledge from the vector store, minimizing the chance of hallucinations and maximizing factual relevance [16].

Iterative RAG (Re-Ranking or Rerun Retrieval). Iterative RAG pipelines build on the basic retriever-generator model but introduce additional steps to refine or re-rank results.

Multiple Iterations: In this design, the system retrieves an initial batch of top-k documents or embeddings, produces a partial (or draft) response, and then uses that partial response to refine the query. This might involve clarifying ambiguous terms, adding extra keywords, or removing less relevant context. The retriever is invoked again, potentially returning a different set of more precise results that better align with the refined query.

Re-Ranking: Another technique is to apply a second module—often a cross-encoder—for re-ranking. The cross-encoder will compare each candidate document with the query more thoroughly, improving the precision of the retrieved set. Such re-ranking is especially beneficial when tasks demand high accuracy, such as medical or legal queries where small factual mistakes can be detrimental.

Hybrid Indexing. While many RAG systems rely purely on dense embeddings for semantic similarity, some adopt a "hybrid" approach that blends sparse (keyword-based) indexing—like BM25—with dense (vector-based) retrieval.

Sparse + Dense: Keyword-based search is adept at matching exact terminology or acronyms, while vector-based retrieval excels at capturing semantic relationships and synonyms. A hybrid index can yield strong coverage for both literal matches and conceptual or paraphrased queries. Users who rely on specific domain terms (such as medical codes, legal phrases, or brand names) benefit from the sparse component, while the dense vectors handle broader, more context-oriented matching.

In summary, modern RAG solutions are increasingly sophisticated, incorporating multiple layers of retrieval refinement and indexing strategies to ensure precise, context-driven generation. The Retriever-Generator Pipeline forms the core of most systems, but techniques such as iterative retrieval (with re-ranking) and hybrid indexing open the door to higher accuracy, domain specificity, and improved tolerance of ambiguous queries. These architectures highlight how pairing advanced transform-based language models with well-structured retrieval mechanisms can deliver highly relevant, up-to-date responses in diverse environments—from e-commerce recommendation engines to deeply specialized medical knowledge bases.

1.4.3 Advanced Techniques for Multimodal RAG

Despite the fact that this direction is one of the newest, I took the liberty of highlighting the following two techniques that allow to achieve the most relevant and accurate answers on large databases with mixed content types:

Multimodal Embeddings: Instead of limiting the vector database to text embeddings, advanced RAG systems store embeddings for images, videos, or audio. Approaches such as CLIP (by OpenAI) or BLIP (Salesforce Research) enable semantically meaningful vector representations of images, which can then be retrieved similarly to text 333.

Cross-Modal Retrieval: Allows queries in one modality (e.g., text) to retrieve related content in another modality (e.g., images). This opens up a variety of use cases like product search, visual QA, and more.

RAG has reshaped how we view language models, demonstrating that external context retrieval can significantly boost factual accuracy and domain specialization. The move toward multimodality, iterative refinement, and dynamic knowledge updates promises even more robust and versatile RAG

systems. As LLMs continue to scale and vector databases mature, RAG will likely remain the leading paradigm for applications demanding both creative generation and factual grounding.

1.5 Large Language Models (LLMs) and Their Role in RAG

Large Language Models (LLMs) form the backbone of Retrieval-Augmented Generation (RAG) by turning retrieved context into coherent and contextually accurate answers. These models, trained on massive text corpora, can generate human-like language, summarize complex information, or even infer connections across topics. When combined with real-time retrieval from external data sources, LLMs are significantly more powerful and reliable than when operating solely on their internal parameters.

Key Characteristics of LLMs in RAG:

Pre-training on Vast Corpora. LLMs such as GPT-4 [9] or Anthropic Claude [10] undergo pre-training on billions of tokens collected from diverse sources, including books, websites, and user-generated content. This stage equips the model with a broad understanding of language structure and semantic relationships. By absorbing patterns from this extensive corpus, LLMs develop a generalized capability to respond to prompts on various topics.

This extensive linguistic and factual grounding means that LLMs can be paired with different types of domain-specific data during the retrieval phase, allowing them to handle a wide range of queries—from technical troubleshooting to in-depth market analysis.

When new context is presented via retrieval, the model leverages its base knowledge while integrating fresh, external content to respond with increased precision.

Because the model already "knows" a great deal about language and real-world concepts, augmenting it with real-time or specialized data can yield remarkably accurate results without having to train a custom model from scratch.

Context Window Constraints. Despite their extensive pre-training, LLMs maintain a finite context window, often measured in tokens, that limits how much text they can "see" or process in a single pass. For instance, a model may only accept a few thousand tokens of input before it cannot ingest further material.

Retrieving only the most pertinent data from a vector database is essential to ensure the final prompt stays within the model's context window. By filtering out extraneous or repetitive material, RAG systems can present a curated set of chunks or passages that fit into this constrained input space.

This mechanism helps avoid cutting off essential details mid-prompt, which could confuse the model or compromise the fidelity of the final answer.

Architects of RAG solutions must plan for robust chunking and summarization strategies, as well as top-k or top-n retrieval logic, to keep the input text concise yet maximally informative.

Enhanced Accuracy and Reduced Hallucinations. LLMs, by their nature, can sometimes produce fabricated or off-topic statements—commonly referred to as "hallucinations." Such occurrences arise when the model draws upon probabilistic patterns rather than referencing verified facts.

Providing curated data from a vector store effectively grounds the model's generation process. Because the LLM is referencing real, contextually relevant text, it becomes significantly less prone to inventing details that are not reflected in the source material.

In knowledge-intensive domains (e.g., medical or legal fields), limiting the scope of possible responses to high-quality, domain-vetted information can drastically reduce factual errors.

This grounding not only bolsters correctness but also builds user trust, as the system can reference and cite specific passages or metadata that informed its final answer.

Flexibility and Domain Adaptation. LLMs offer a high degree of adaptability when supplemented with domain-specific training or fine-tuning. By ingesting corpora in specialized fields—like finance, law, or healthcare—an LLM can refine its approach to technical language, conventions, and idiomatic usage within that domain.

Developers can use incremental fine-tuning strategies on LLMs for specialized tasks, retaining the general benefits of the model's broad pre-training while honing its responses in narrower areas of expertise.

Reinforcement Learning from Human Feedback (RLHF) provides an additional layer of alignment, ensuring that generated content adheres to professional standards and expert-reviewed guidelines.

When integrated with retrieval, a model that has already been partially fine-tuned on relevant topics will better synthesize newly retrieved context, achieving more coherent, domain-specific outputs.

Workflow in a RAG System:

1. User Query: The user provides text prompts or multimodal data (e.g., images).
2. Retrieval Stage: A vector database (e.g., Chroma DB) identifies top-k embeddings that match the user's query.
3. Context Injection: These embeddings, typically chunks of text or encoded visuals, are passed into the LLM's context window.
4. Generation: The LLM synthesizes a response that references the retrieved context, aiming for concise, accurate, and up-to-date information.

By grounding LLMs in context retrieved from vector databases, RAG frameworks harness the best of both worlds: broad language fluency from massive pre-training and precise, domain-specific knowledge from external data sources. This synergy addresses critical limitations of standalone generative models and opens the door to reliable, versatile applications across business, research, and beyond.

1.6 Multimodal Embeddings

Multimodal embeddings extend the concept of text-only vector representations to include a variety of data formats—images, audio, video, and even sensor data. By converting different data types into a shared vector space, AI systems can perform cross-modal retrieval and reasoning, creating new possibilities in applications such as visual search, audio-based information retrieval, and video summarization.

1.6.1 Foundations of Multimodal Embeddings

Unified Embedding Space. Systems like CLIP (Contrastive Language–Image Pre-training) map text and images into a common vector space. A similar approach can be applied to audio or video, aligning different modalities so that semantically similar content appears close in high-dimensional space 2.

Benefit: Enables cross-modal queries (e.g., retrieving an image based on textual descriptions) and consistent retrieval pipelines.

Encoder Architectures. Text Encoders: Typically use transformer-based models (e.g., BERT, GPT) or sentence transformers designed for semantic search.

Image Encoders: Often rely on convolutional neural networks (CNNs) or vision transformers (ViT) to extract high-level features.

Audio Encoders: May involve spectrogram-based CNNs or recurrent architectures to capture temporal patterns.

Best Practice: Ensure each encoder is well-calibrated so that embeddings are comparable across modalities (e.g., by normalizing vector norms or using a shared embedding dimension).

Training Strategies. Contrastive Learning: Pairing relevant data (e.g., image–caption pairs) as positives, while treating mismatched pairs as negatives, encourages the model to learn robust multimodal features.

Self-Supervised Approaches: Systems like multimodal autoencoders or masked prediction tasks can learn shared representations without explicit labels.

1.6.2 Incorporating Multimodal Embeddings in RAG

Each modality generates its own embeddings, but all embeddings reside in a single vector database (e.g., Chroma DB).

Example: Product catalog data could include text descriptions, user reviews, and product images—all indexed for quick retrieval in the RAG pipeline.

Cross-Modal Retrieval. A user might provide a textual query (“Find images of running shoes for marathon training”), triggering a search that returns relevant image embeddings.

Conversely, an image or audio sample could query related text entries or knowledge bases for contextual information.

By unifying text, images, audio, and other data formats into a cohesive vector space, multimodal embeddings open the door to holistic, cross-domain capabilities [7] in RAG systems. As industries increasingly rely on varied data sources, the ability to seamlessly integrate multimodal information promises more robust, accurate, and user-centric AI solutions.

1.7 Vector Databases for Real-Time Retrieval

Vector databases play a central role in RAG pipelines by efficiently handling the high-dimensional embeddings needed for semantic search [11].

Unlike traditional relational databases that primarily index structured data, vector databases specialize in similarity-based queries—finding the nearest vectors to a given query embedding in real-time. This capability is particularly valuable when dealing with both text and multimodal data (e.g., images, audio).

Key Concepts and Functionality:

High-Dimensional Indexing. Approximate Nearest Neighbor (ANN): Algorithms like HNSW (Hierarchical Navigable Small World) and IVF (Inverted File) enable sub-millisecond searches even across millions or billions of embeddings.

Metric Spaces: Data points are typically compared using cosine similarity or Euclidean distance.

Scalability and Sharding. Horizontal Scaling: Distributing embeddings across multiple nodes (sharding) allows systems to handle massive datasets without bottlenecks.

Load Balancing: Ensures even query distribution and reduces single-point failures.

Best Practice: Implementation of dynamic sharding strategies that can rebalance data when new nodes are added or existing ones become overloaded.

Real-Time Updates. Online Insertion/Deletion: In many RAG scenarios, new data arrives continually (e.g., fresh documents, updated product images). Vector databases must handle these real-time inserts.

Index Reconstruction: Some databases allow partial index rebuilding to accommodate frequent updates, balancing retrieval speed with up-to-date results.

Hybrid Indexing. Sparse + Dense: Combining keyword-based search (like BM25) with dense vector search supports both exact matches and semantic similarity.

Integration vector DB in a RAG Pipeline:

1. Embedding Generation: Text, images, or other data are converted into vector representations via encoders.
2. Storage and Indexing: These embeddings are uploaded to a vector database, which builds indexes to accelerate nearest-neighbor queries.
3. Query: An input query is embedded and passed to the vector database to retrieve the most relevant items (top-k).
4. Response Generation: A Large Language Model (LLM) uses the returned items as context to produce an answer or generate content.

By relying on specialized data structures, vector databases deliver fast, scalable, and semantically robust search - a cornerstone of effective RAG implementations. Their real-time retrieval capabilities ensure that generated content remains relevant, accurate, and adaptive to changing data landscapes.

Chapter 2

2.1 High level architecture and designing RAG system

The Retrieval-Augmented Generation (RAG) System is designed to seamlessly integrate real-time data retrieval with powerful language modeling capabilities, enabling the production of contextually relevant and multimodal responses for a variety of domains. At its core, this system combines a vector database for storing and retrieving multimodal embeddings (text, images, etc.) with a Large Language

Model (LLM) to enrich user queries with up-to-date and domain-specific information.

Much like an integrated platform in an educational environment, the RAG System functions as a robust, scalable solution that prioritizes flexibility, accuracy, and simplicity for both administrators and end users. By leveraging modern AI techniques, the system can handle diverse data (e.g., PDFs, images, tables) and deliver intelligent responses tailored to the user's query. This architecture focuses on real-time retrieval, transparent communication with AWS-managed services, and a well-organized API layer that supports synchronous or asynchronous usage.

2.1.1 User Interface

To be honest, I planned to make a more traditional, user-friendly interface that would meet modern industry standards, but I didn't have enough time for the frontend, so I had to limit myself to the default FastAPI [12] interface. The user-facing component of the RAG System is a lightweight FastAPI application that offers a straightforward, intuitive interface to submit and manage queries. Administrators and advanced users can interact with the system through various endpoints—either synchronously or by delegating tasks to a worker Lambda function for asynchronous processing. This design not only improves accessibility and ease of management but also enhances the user experience by automatically distributing workload for resource-intensive queries.

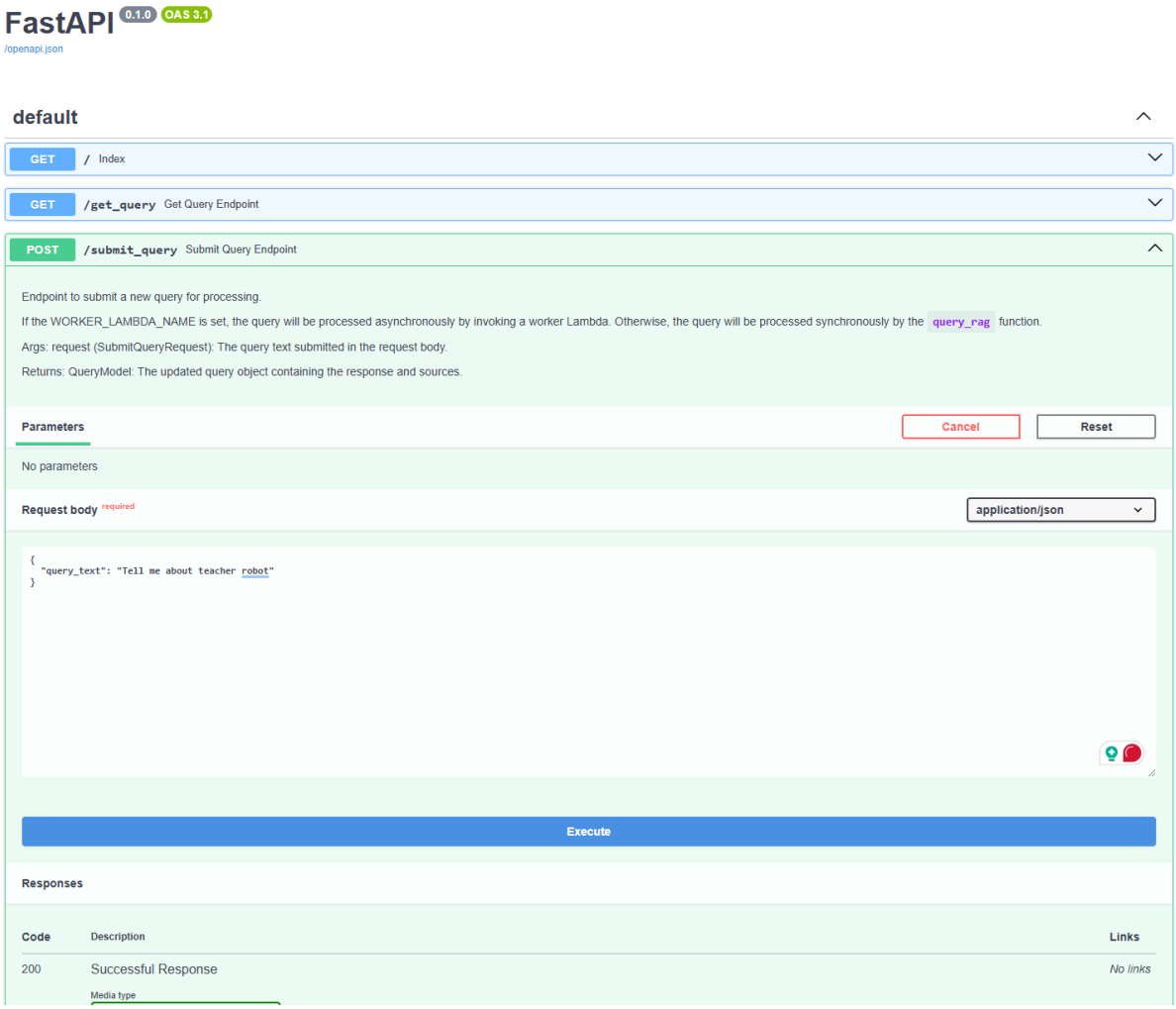

Figure 2.1.1 System interface

2.1.2 Backend and Core Logic

At the heart of the system is the Python-based backend, that coordinates the entire Retrieval-Augmented Generation workflow by interfacing with multiple specialized modules. The process begins with document ingestion, where PDFs, images, and table data are harvested, divided into smaller chunks, and then embedded using a Bedrock-based model specifically tasked with generating vector representations. During this phase, the system extracts textual content, table-like structures, and any relevant images from PDF sources, relying on carefully designed chunk-splitting logic to ensure each portion remains within manageable size limits. These chunks, once converted into floating-point vectors,

are written to the Chroma database, a vector store designed for rapid similarity searches. By continually updating Chroma DB in this fashion, the system ensures that any query can access the most recent semantic representations, thus reducing the risk of outdated references and enabling context-rich retrieval.

Once a user query arrives, the backend draws upon the LLM integration layer, connecting to a large language model (such as Anthropic Claude) through AWS Bedrock. Before the model is called, the backend retrieves top-k relevant embeddings from Chroma DB by comparing the user's query embedding to the stored vectors; these retrieved chunks are then merged into a context prompt designed to guide the LLM toward an answer that remains faithful to the underlying data. The goal here is twofold: to minimize hallucinations and to maintain high fidelity to the documents ingested earlier. By injecting these top-k chunks directly into the prompt, the LLM receives immediate, domain-specific knowledge that significantly boosts the coherence and accuracy of the generated text.

To keep track of every incoming request and subsequent answer, the backend leverages a DynamoDB table that logs essential query details, including query text, source metadata, and the model's generated response. This design ensures the system can handle a spike in usage by distributing state management across a serverless, NoSQL data store rather than relying on a single node or a traditional relational database. Scalability becomes straightforward under high-traffic scenarios, as DynamoDB scales seamlessly when writes and reads intensify, and each query record is associated with a unique key to allow instantaneous lookups.

For operations that demand significant processing time, the backend employs an asynchronous architecture. When a query is likely to involve substantial parsing, large volumes of data, or extensive generation time, the system transfers control to a worker Lambda function. This worker remains responsible for finalizing RAG tasks—retrieving embeddings from Chroma DB, constructing the LLM prompt, generating an answer, and recording the output back in DynamoDB.

Because this worker executes independently of the front-end API, the user-facing Lambda returns an immediate acknowledgment, thus sparing the user from lengthy waits and preventing potential timeouts. By separating heavy-duty tasks from synchronous API calls, the backend remains responsive even under large loads, and errors in one long-running query do not impede others.

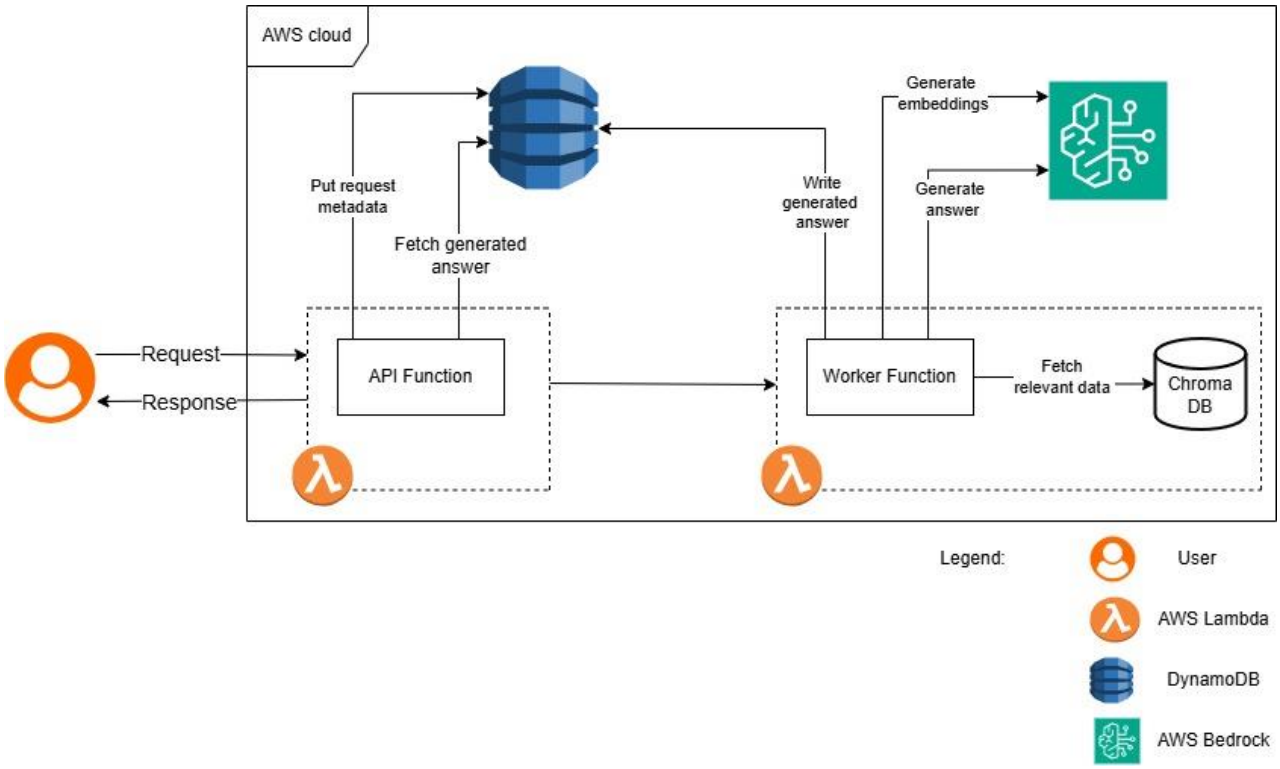

Figure 2.1.2 Backend architecture

2.1.3 Deployment and Infrastructure

The RAG System is containerized and deployed via AWS CDK (Cloud Development Kit) [14]. Through CDK, core components are defined as code, making them easy to maintain, version-control, and replicate:

- Lambda Functions: Two Docker-based Lambda functions—one for API handling (FastAPI) and one for background (worker) tasks.
- DynamoDB: Houses query metadata, enabling quick lookups of query status and results.

- Chroma Vector DB: Maintains vector embeddings for text, tables, and images.
- Amazon Bedrock: Provides the language model (Claude) and embedding services.
- IAM Roles: Enforce least-privilege access to ensure a secure environment.

Through this setup, the RAG System provides an adaptable, cloud-native platform for real-time information retrieval, content generation, and robust data storage. Administrators can easily configure environment variables (e.g., DynamoDB table names, Lambda handlers) and scale individual components to match evolving demands, ensuring efficiency and reliability in various real-world scenarios.

2.1.4 Technical Specifications

Name	Description
Frontend (FastAPI)	A lightweight HTTP service exposing endpoints for user query submission and retrieval. Designed for easy integration with any web or mobile client.
Backend (Python)	Core business logic for ingestion, embedding, retrieval, and orchestrating RAG workflows. Handles PDF parsing, chunk splitting, and updates to Chroma DB.
AWS Lambda	A background service that performs asynchronous processing of complex or large-scale queries. Invoked by the API Lambda or other triggers when tasks require extended execution time.
Chroma Vector DB	Stores embeddings for text, images, and table data. Facilitates quick similarity searches to deliver relevant context to the LLM.

LLM Integration (AWS Bedrock)	Interfaces with Anthropic Claude (or similar LLM) to generate context-aware responses. Embeddings also provided by Bedrock [13] for consistent vector representations.
DynamoDB (Query Metadata)	NoSQL database for logging query status, storing final answers, and providing a persistent record of user interactions.
AWS CDK Infrastructure	Automates deployment of Lambdas, DynamoDB table, IAM roles, and function URLs. Ensures a repeatable, code-driven approach to infrastructure management.
Docker Containerization	Both the API and worker components are packaged into Docker images, streamlining deployment and ensuring consistent environments (e.g., shared libraries, Python dependencies).
Security and Permissions	IAM policies and environment variables restrict data access to required resources. Optionally extended with VPC integration, encryption at rest, or custom authentication schemes for production.

Table 2.1.4 Technical Specification

2.2 Data Ingestion and Workflow

This chapter focuses on two essential processes in our RAG System: converting PDF files with multimodal content into the vector database, and the end-to-end data flow when a user submits a query and receives a context-enriched answer.

2.2.1 Converting PDF Files with Multimodal Info into the Vector DB

Modern organizations often deal with a variety of data types—text, images, tables, etc.—within a single PDF document. Our RAG pipeline addresses this

challenge by breaking each file into appropriately embedded chunks before storing them in Chroma DB. Below is an overview of this ingestion flow:

1. **File Detection and Reading.** The system checks a source directory (e.g., src/data/source/) for any PDF files. Each file discovered is opened by PyMuPDF (fitz), which iterates through its pages.
2. **Chunk Splitting and Metadata Tagging.** After extraction, text-based content is split into smaller parts to ensure each segment remains within suitable lengths for embedding. Throughout this process, metadata such as file source, page number, and content type (text, image, or table) is retained. A unique identifier is assigned so each chunk can be traced back to its PDF origin.
3. **Vector Embedding.** Each chunk (text or Base64-encoded image) is passed to a Bedrock-based embedding function—defined in get_embedding_function.py - which converts it into a floating-point vector. The pipeline can quickly handle semantic searches by storing both the chunk and its vector.
4. **Database Persistence.** Finally, the newly created or modified chunks, along with their embeddings, enter Chroma DB. Before adding each chunk, the system checks its unique ID to avoid duplicates. With this ingestion complete, the system is primed for RAG workflows, ensuring that text, images, and tables are immediately accessible for similarity searching and context generation.

Key moments:

- Comprehensive Extraction: Captures text, images, and basic table data in one pass.
- Automatic Chunking: Smaller text pieces improve both retrieval and model context.
- Consistent Metadata: Each extracted piece is coupled with source info, enabling accurate traceability.

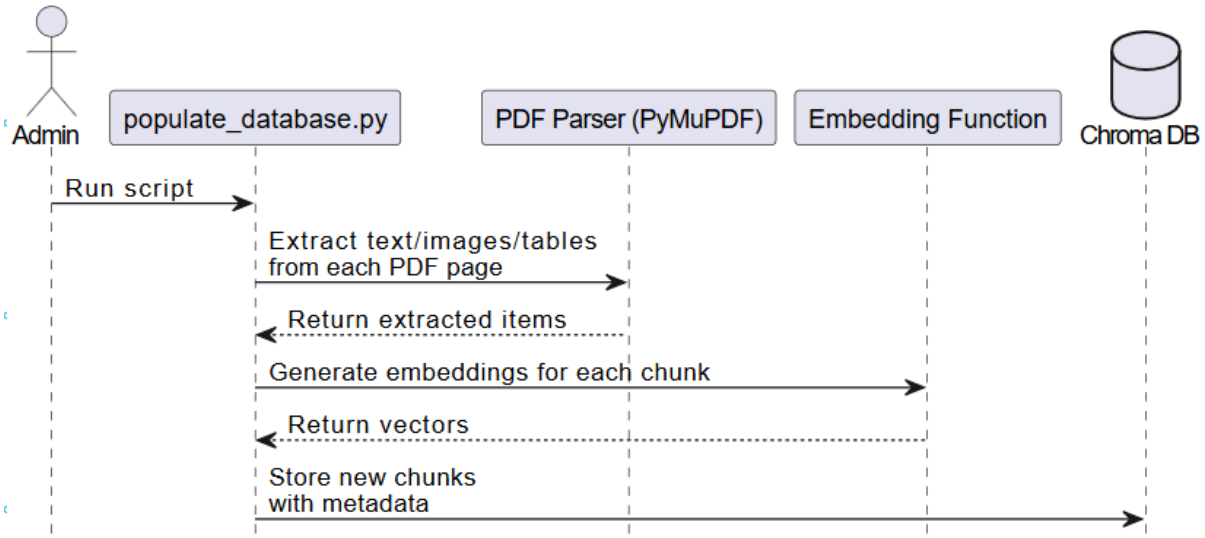

Figure 2.2.1 Converting PDF files into Vector DB

2.2.2 End-to-End Data Flow: From User Request to Context-Aware Response

Once the PDF ingestion and embedding steps are complete, users can query the system to retrieve relevant information. Below is the typical flow that starts with a user request and concludes with a context-enhanced answer:

1. **User Submits Query.** The user sends a query (e.g., “Show me info about your services”) to the FastAPI endpoint. A QueryModel object is created, storing essential fields like query_id, query_text, and create_time, and is immediately placed into DynamoDB with a status of “in progress.”
2. **Retrieval Step (Vector Database Lookup).** The query text is embedded using the Bedrock [13] embeddings model, then a similarity search is performed in Chroma DB to locate the top-k relevant chunks (text, tables, images). These chunks might include PDF excerpts, base64-encoded images, or table data pertinent to the user’s query.
3. **Prompt Assembly and Model Invocation.** The retrieved chunks are grouped by type (text, table, image) and combined with the user’s question to form a prompt template. This prompt is sent to the LLM (Anthropic

Claude or a similar model via AWS Bedrock), which generates a context-aware answer.

4. **Response Creation and Storage.** The model's response is added to the QueryModel record in DynamoDB, along with IDs of the documents that were used as context. The query's status is updated to "complete," indicating that a valid answer is ready.
5. **User Retrieval of Answer.** The user or client application can retrieve the completed answer by querying DynamoDB or calling the FastAPI endpoint (GET /get_query?query_id=...). The returned data includes the AI-generated answer plus any relevant source metadata.

One additional idea for this project, not yet implemented due to limited time, involves a separate caching service that would deliver results for very similar queries directly from a database. This could lessen the load on the LLM but presents its own challenges.

2.3 Database Structure

The RAG System operates with two primary data storage mechanisms, each fulfilling distinct roles:

1. Chroma Vector Database. Stores high-dimensional embeddings derived from text, images, and tables for rapid similarity search.
   Facilitates real-time retrieval of context relevant to user queries.
2. DynamoDB (Metadata and Query State Management). Houses metadata such as query states, final answers, and references to relevant source chunks.
   Ensures persistence and fast lookups for user queries, enabling both synchronous and asynchronous processing.

By separating vector storage from query and state metadata, the system can efficiently handle large-scale retrieval tasks while maintaining a minimal, consistent record of query progress and results.

2.3.1 Chroma Vector Database

Chroma DB is designed specifically for storing and searching large sets of embeddings (high-dimensional vectors). In the RAG System, every chunk of extracted text, table data, or base64-encoded image content is converted into vectors using a Bedrock-based embedding function. These vectors are then persisted in a Chroma index to enable efficient Approximate Nearest Neighbor (ANN) lookups.

Each chunk inserted into Chroma DB typically consists of:

- Embedding Vector: A numeric array representing the semantic content of the chunk.
- Page Content: Original extracted text or encoded media (e.g., base64-encoded image).
- Metadata Dictionary: Key-value pairs capturing:
    - source: Filename or document source identifier (e.g., my_document.pdf).
    - page: Page number or location reference in the original file.
    - type: Classification of the chunk (e.g., text, table, image).
    - id: Unique identifier for the chunk (e.g., a hash of the chunk's content)

Example (Conceptual):

```
{
  "embedding": [0.234, -0.091, ... , 0.778],
```

```
  "page_content": "Base64 Encoded Data or Extracted Text",
  "metadata": {
    "source": "data/source/document.pdf",
    "page": 5,
    "type": "image",
    "id": "my_document.pdf:5:image:123456789"
  }
}
```


Although Chroma DB doesn't enforce a rigid "schema" in the same way a relational database does, it provides a structured approach to indexing embeddings. By default, it maintains:

- ID Field: A unique document identifier in the vector store.
- Embedding Field: Stores the vector representation for each document chunk.
- Metadata Field: An opaque JSON-like structure for any additional data relevant to retrieval.

2.3.2 DynamoDB for Query Metadata

While Chroma DB is optimized for vector similarity search, DynamoDB (NoSQL) is used to track user queries, record AI-generated answers, and maintain the overall workflow state. This separation ensures that the system can quickly serve query metadata—such as whether a query is complete—without scanning a large embedding store. The **primary key** is a query_id, representing a unique user query or session.

Attribute	Type	Description
query_id	String	Primary key for the query.
create_time	Number	Unix timestamp indicating when the query was created.
query_text	String	Original user input text.
answer_text	String	AI-generated answer.
sources	List	List of relevant chunk IDs from Chroma DB.
is_complete	Boolean	Indicates whether query processing is finished (true/false).

Table 2.3.2 DynamoDB schema.

Chapter 3

Over the course of developing and refining this Retrieval-Augmented Generation (RAG) System, several technical and conceptual hurdles emerged—each yielding valuable insights into the synergy of Large Language Models (LLMs), vector databases, and advanced cloud-based infrastructures. This chapter examines the principal challenges encountered, the strategies employed to resolve them, and the broader lessons gleaned regarding AI-driven architectures.

3.1 Challenges faced during the work process

Here is a list of challenges I faced during my work that is far from complete:

Model Updates and Ecosystem Shifts.

Challenge: The rapid iteration of LLM releases (e.g., new versions of Claude or GPT) often led to shifting performance benchmarks and compatibility issues. Models trained on older corpora sometimes struggled to integrate newly ingested domain-specific knowledge.

Impact: Ensuring consistent results across multiple model updates required careful planning—particularly around prompt engineering and the RAG pipeline's reliance on the model's context window [15].

Hallucination and Factual Consistency.

Challenge: Even the most advanced LLMs may generate plausible-sounding but factually inaccurate responses. RAG architectures mitigate this by providing external context; however, balancing LLM creativity with factual precision remains non-trivial.

Impact: Emphasizing a robust retrieval step and explicit references to embedded content reduced hallucinations. Nonetheless, the risk of spurious text lingered when the provided context was incomplete or ambiguously phrased.

Prompt Engineering Complexity.

Challenge: Constructing prompts that effectively guide the LLM required extensive experimentation, especially when dealing with multimodal inputs (images, tables, text) and domain-specific jargon.

Impact: Achieving clarity and consistency in the generated answers demanded iterative refinement of prompt templates, hierarchical structuring of retrieved content, and consideration of the LLM's token constraints.

Indexing Multimodal Data.

Challenge: Handling PDFs containing both textual and visual elements required a unified approach to embedding and storage. Splitting tables, images (base64-encoded), and text into coherent chunks while preserving metadata proved intricate.

Impact: Ensuring the consistency of chunk identifiers (e.g., source, page, type) was crucial for transparent retrieval. Any mismatch risked disjointed context or confusion in RAG prompts.

Asynchronous Query Handling.

Challenge: Claude's model endpoints via Bedrock typically operate on a request-response basis. However, certain queries demanded more extensive context or additional checks, making straightforward synchronous calls suboptimal.

Impact: Designing a Worker Lambda that asynchronously engaged Claude allowed for better load management but introduced complexity in tracking query states and responses in DynamoDB.

Billing and Usage Patterns

Challenge: Fine-grained cost visibility was essential. Since each invocation to Claude, even for short requests, contributes to operational expenses, usage patterns needed close monitoring.

Impact: Adopting caching strategies—such as memoizing repeated content or re-ranker steps—helped mitigate costs. Logging token usage and query frequencies provided data for optimization.

Unfortunately, this strategy could not be implemented due to lack of time, but this approach should definitely be used in future systems.

Security and Policy Configuration.

Challenge: Bedrock endpoints require specific IAM roles and policy structures. In a multi-account setting, cross-account permissions and resource-based policies added complexity.

Impact: A rigorous DevOps pipeline, combined with Infrastructure as Code (CDK) [14], ensured all security policies remained consistent and verifiable, thus preventing unauthorized invocations and data leakage.

3.2 Lessons Learned

The Power of Retrieval-Augmented Generation. Through implementing RAG workflows, it became apparent that factual consistency and adaptability significantly increase when external data is integrated at query time. Even highly capable LLMs benefit from a well-curated knowledge source:

Data Centricity: The system's overall fidelity hinges on the quality and organization of the embeddings. Proper chunking, metadata labeling, and re-ranking strategies enhance retrieval success rates.

Contextual Grounding: By grounding the LLM in real-time documents, hallucinations are minimized, and domain-specific queries yield more accurate results. This approach underscores the fundamental synergy between advanced NLP models and robust data retrieval.

Importance of Multimodal Embeddings

The integration of text, images, and table data illuminated the necessity of carefully handling different modalities:

Enhanced Use Cases: Combining textual and visual data supports a breadth of scenarios, from technical manuals with diagrams to e-commerce platforms with product images.

Complexity in Chunking: Handling base64-encoded images and table extraction required specialized logic to keep them distinct yet linked to textual context. This ensures the LLM can reference visual or tabular data accurately.

Iterative Prompt Engineering and Model Tuning

Prompt design is more of an art than a science:

Context Windows: With limited token windows, balancing the volume of retrieved text with clarity in the prompt demanded iterative experimentation. Summaries or smaller retrieval sets occasionally yielded more coherent answers.

Instruction Clarity: Encouraging the LLM to rely solely on provided context (e.g., "Answer only from these documents…") mitigated off-topic or hallucinatory outputs. Overly broad instructions tended to produce meandering results.

Resilience and Scalability with Serverless Microservices

The dual-Lambda approach - API and Worker - proved pivotal for a scalable, event-driven pipeline:

Non-Blocking Concurrency: Complex queries that consume significant compute resources do not stall user-facing endpoints, since asynchronous calls allow immediate user acknowledgments and parallel processing in the Worker Lambda.

IaC Consistency: Leveraging AWS CDK ensured that changes to infrastructure could be tracked in source control, enabling reproducible deployments across environments.

The challenges confronted during the development of this RAG System—ranging from LLM adaptation and vector indexing intricacies to robust asynchronous orchestration—have yielded significant insights. These lessons, rooted in both practical experimentation and a deep personal engagement with the topic, shape a roadmap for further enhancements. Ultimately, the confluence of Large Language Models, intelligent retrieval, and scalable cloud infrastructure heralds a new frontier in AI applications, one where factual grounding, multimodal understanding, and on-demand adaptability converge to transform how information is accessed and consumed.

In sum, the project underscored the necessity of interdisciplinary thinking—combining the rigor of software engineering, the depth of modern NLP research, and the architectural elegance of serverless cloud design. My eagerness to expand

upon these foundations is unwavering, as each challenge fosters a more robust, visionary perspective on the role of AI in orchestrating dynamic, context-aware intelligence across diverse domains.

Bibliography

1. Brown, T., Mann, B., et al. (2020). Language Models are Few-Shot Learners. arXiv:2005.14165.

2. Lewis, P., Perez, E., et al. (2021). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401.

3. Mialon, G., et al. (2023). Augmenting LLMs with External Knowledge: A Survey of Retrieval-Augmented Generation (RAG). arXiv:2306.13931.

4. Karpukhin, V., Oguz, B., et al. (2020). Dense Passage Retrieval for Open-Domain Question Answering. arXiv:2004.04906.

5. Radford, A., Kim, J. W., et al. (2021). Learning Transferable Visual Models from Natural Language Supervision (CLIP). arXiv:2103.00020.

6. Li, J., et al. (2022). BLIP: Bootstrapping Language-Image Pre-training for Unified Vision-Language Understanding and Generation. arXiv:2201.12086.

7. Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. arXiv:1908.10084.

8. Lewis, P., Perez, E., et al. (2021). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. arXiv:2005.11401.

9. OpenAI. (2023). GPT-4 Technical Report. arXiv:2303.08774.

10. Anthropic. Claude Documentation. Retrieved from: https://docs.anthropic.com/en/docs/welcome

11. Johnson, J., et al. (2019). Billion-scale similarity search with GPUs. IEEE Transactions on Big Data. arXiv:1702.08734

12. FastAPI https://fastapi.tiangolo.com/

13. AWS. Bedrock AI Service Documentation. Retrieved from: https://aws.amazon.com/documentation-overview/bedrock/

14. AWS. CDK documentation. Retrieved from: https://aws.amazon.com/cdk/

15. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention is all you need.* Advances in Neural Information Processing Systems, 30. https://arxiv.org/abs/1706.03762

16. Wolf, T., Debut, L., et al. (2020). Transformers: State-of-the-Art Natural Language Processing. arXiv:1910.03771.