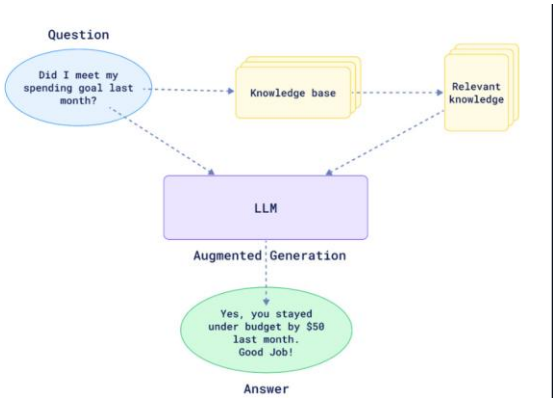
RAG

Retrieval-Augmented Generation (RAG) is an AI architecture that enhances a Large Language Model (LLM) by connecting it to an **external knowledge source**.
Instead of relying only on the LLM's internal training data, RAG retrieves relevant information *at query time* and injects it into the prompt before generating an answer.

This makes the system to provide:

- More accurate answers
- Stay up to date with relevant source
- Domain specific

Limitation of LLMs	RAG Solutions
LLMs hallucinate missing facts	Retrieves real documents as grounding
Models have outdated knowledge	Pulls updated knowledge from DB/API
Cannot store all domain-specific data	Uses external vector DB as memory
Expensive retraining	Only update knowledge base → cheap
Limit on context window	Retrieve only relevant pieces



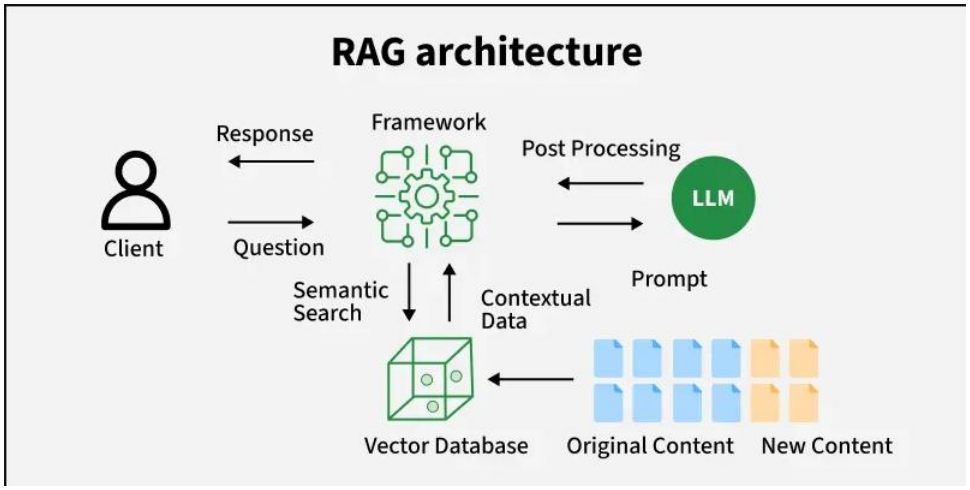
RAG example from Qdrant

RAG Architecture

A RAG architecture includes the **retriever** and the **generator** as their main components.

The **retriever** is the search component of a RAG system. When a user asks a question, the retriever converts it into an embedding and performs semantic search inside a vector database. It then identifies and returns the most relevant chunks of information from the available documents. This ensures that the system finds accurate, up-to-date, and meaningful context that will be used to answer the query.

The **generator** is the Large Language Model (LLM) that creates the final response. It takes the user's question together with the retrieved context and generates a coherent, grounded answer. Because it relies on real retrieved data rather than only its internal knowledge, the generator produces responses that are more factual, reliable, and tailored to the user's query.



RAG Architecture Flow

1. Client Sends a Question

The process begins when the **client** submits a query.
This question is sent to the **RAG framework**, which starts the entire retrieval + generation pipeline.

2. Framework Performs Semantic Search

Once the question is received:

- The framework converts the question into an **embedding** (a vector representation).
- This embedding is sent to the **vector database**.

- The vector database stores embeddings of all existing knowledge (Original Content + New Content).

The system performs **semantic similarity search**, meaning:

It does not search using keywords, Instead it finds chunks of text that are *closest in meaning* to the query in vector space.

This step retrieves the most relevant **contextual data** from the knowledge base.

3. Vector Database Returns Contextual Data

The vector database returns:

- The **top-k most relevant text chunks**,
- Metadata or document references,
- Sections of Original Content and New Content that match the query.

These retrieved items act as the **evidence** the LLM will rely on.

4. Framework Builds a Prompt for the LLM

The framework now:

- Combines the **user question**
- With the **retrieved contextual data**

This forms an **augmented prompt**. This is the "retrieval-augmented" part of RAG as the LLM doesn't answer based on guessing; it answers based on retrieved facts.

5. LLM Generates a Grounded Response

The augmented prompt is sent to the **LLM**.

The LLM processes:

- The question
- The supporting evidence (retrieved chunks)

And produces a **factual, grounded answer**.

Because the LLM sees real data in the prompt, hallucinations are significantly reduced.

6. Framework Applies Post-Processing

Before sending the answer back:

The framework may perform optional operations:

- Formatting the answer
- Filtering or validating content
- Adding citations or sources
- Summarizing long outputs

This step ensures the final message is clean and user-friendly.

7. Final Response Returned to Client

The processed output is returned to the client as the **final grounded answer**.

RAG Architecture Concepts

1. Classical RAG

Classical RAG represents the simplest form of retrieval-augmented generation. The system retrieves the top-K text chunks from a vector database and directly feeds them into the LLM to generate an answer. This way is fast and easy to implement, but it lacks deeper reasoning, document reranking, and noise reduction. As a result, its accuracy heavily depends on how good the retrieval step is. Naïve RAG works best for stable knowledge bases where content is clean, consistent, and not too large.

2. Retrieve-and-Rerank RAG

Retrieve-and-Rerank improves on classical RAG by adding a **reranking step** after initial retrieval. The retriever first pulls a large set of potentially relevant chunks (e.g., top-50). Then a reranker model often a cross-encoder evaluates and scores each chunk for relevance. Only the highest-scoring chunks are passed to the LLM. This significantly improves precision and reduces irrelevant or noisy context. It is especially effective in domains like legal search, customer support, and compliance, where accurate retrieval is crucial.

3. Multimodal RAG

Multimodal RAG extends traditional RAG beyond text to include **images, audio, video, diagrams, and structured documents**. It uses multimodal embedding models (to encode different data types into a unified vector space. The generator is also multimodal, allowing the system to reason over visual and textual information together. This variant is essential for applications like document understanding, vision-language tasks, medical imaging workflows, and multimedia retrieval.

4. Graph RAG

Graph RAG builds a **knowledge graph** from documents by treating chunks as nodes and linking them through semantic relationships (edges). Retrieval is performed not only based on vector similarity but also graph connectivity, enabling structured reasoning. The LLM receives context enriched by related nodes, paths, and hierarchical relationships. This is ideal for complex knowledge domains such as scientific literature, enterprise knowledge graphs, technical documentation, and research papers where relationships matter as much as content.

5. Hybrid RAG

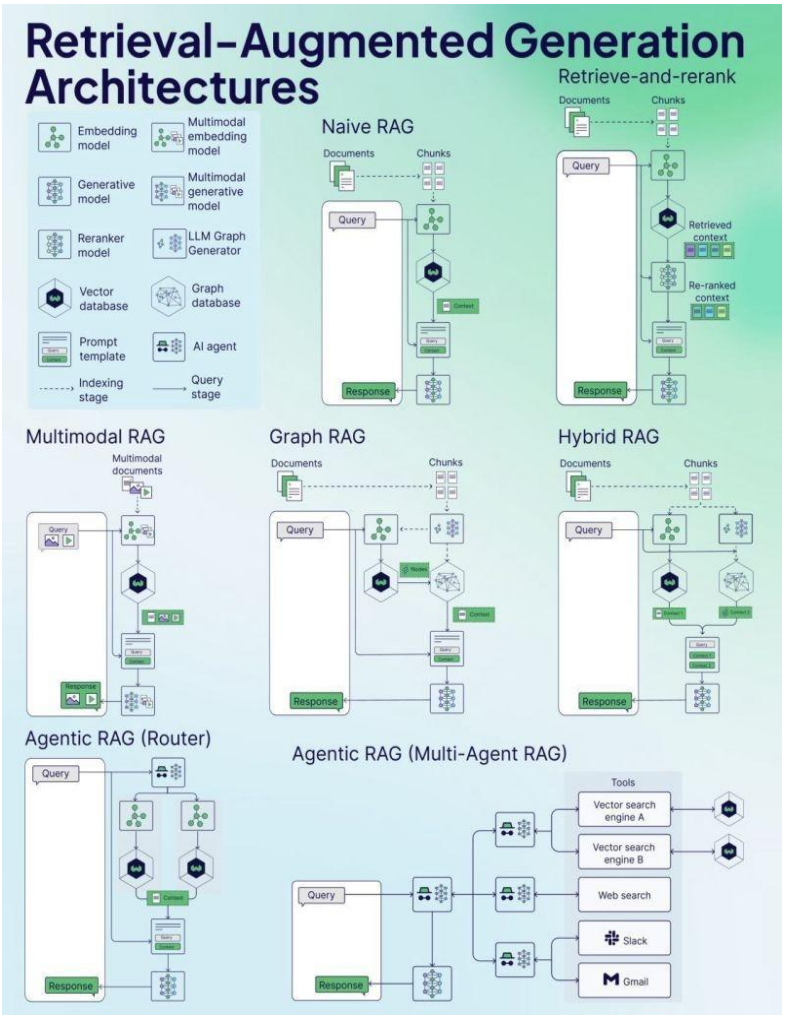
Hybrid RAG combines **dense retrieval** (vector embeddings) with **sparse retrieval** (keyword-based search like BM25). Dense retrieval excels at semantic similarity, while sparse retrieval is strong at exact term matching. Merging these approaches creates a more robust system capable of handling a wide variety of query types from conceptual questions to keyword-heavy searches. This leads to higher recall, especially in domains with jargon, abbreviations, or long-tail terminology.

6. Agentic RAG (Router Model)

In Agentic RAG with routing, an LLM-based agent intelligently analyzes user intent and decides **which retriever, tool, or database** is most appropriate for the query. The agent may route financial questions to a finance retriever, technical questions to documentation retrievers, and so on. This model is widely used in enterprise workflows where knowledge comes from many heterogeneous sources—APIs, search engines, internal databases, and file stores.

7. Agentic RAG (Multi-Agent Model)

Multi-Agent RAG involves several specialized agents working in parallel. Each agent performs a distinct retrieval or reasoning task—querying search engines, interacting with email, extracting structured data, or retrieving from various knowledge bases. Their outputs are then merged and synthesized by a coordinator agent or LLM. This approach functions like an **AI operating system**, where multiple tools collaborate to produce a comprehensive final answer. It is extremely powerful for automation, research tasks, enterprise copilots, and complex decision-making systems.



RAG Architecture Concepts from weaviate

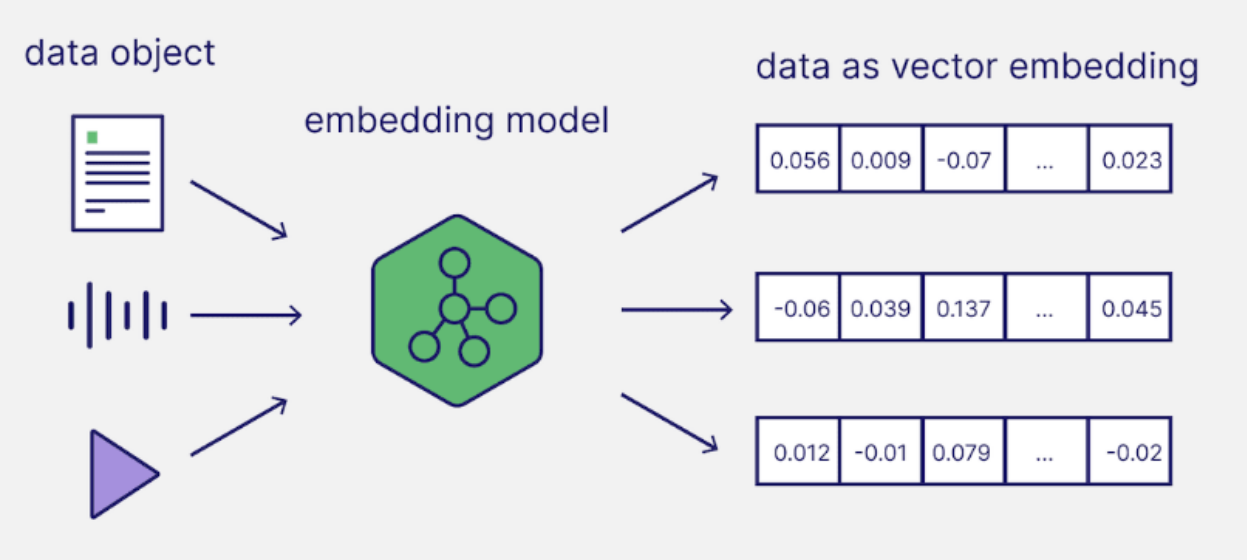
Classical Vs Multimodal RAG

Classical RAG is a retrieval-augmented generation approach that works exclusively with text: it embeds text documents, retrieves semantically similar text chunks from a vector database, and uses a language model to generate answers based solely on textual context. This makes it simple, fast, and cost-effective for knowledge bases, policy documents, chatbots, and other text-heavy applications. In contrast, Multimodal RAG expands this pipeline to support multiple data types such as images, charts, tables, scanned PDFs, audio, and video frames by using multimodal embeddings and multimodal LLMs capable of reasoning across different formats. As a result, it can interpret diagrams, extract insight from visual elements, analyze dashboards, and retrieve information across various modalities in one unified search space. While Classical RAG is easier to implement and cheaper to operate, Multimodal RAG is more powerful and better suited for real-world enterprise data, where information often exists beyond plain text.

Embedding models and vectors

Embeddings are numerical representations of words or phrases in a high-dimensional vector space. These representations map discrete objects (such as words, sentences, or images) into a continuous latent space, capturing their relationship. They are a fundamental component in the field of Natural Language Processing (NLP) and machine learning.

Embeddings position words into a vector in a high-dimensional latent space so that words with similar meanings are closer together. This helps ML models understand and process text more effectively. For example, the vector for “apple” would be closer to “fruit” than to “car”.



How embedding models work:

1. **Vectorization** — The model encodes each input string as a high-dimensional vector.
2. **Similarity scoring** — Vectors are compared using mathematical metrics to measure how closely related the underlying texts are.

Similarity metrics

Several metrics are commonly used to compare embeddings:

- **Cosine similarity** — measures the angle between two vectors.
- **Euclidean distance** — measures the straight-line distance between points.
- **Dot product** — measures how much one vector projects onto another.

Types of embedding models

1. Word embedding models

These models aims to convert words into numerical vectors that capture semantic meanings and relationships between different words. For example:

- **Word2vec:** This model learns word embeddings by predicting a word based on its context or predicting context based on a word.
  GloVe (**Global vectors for vector representation)**: uses word co-occurrence statistics from a large corpus to create embeddings.

2. Audio embedding models

These models aims to convert speech data into embeddings which are useful for tasks like speech recognition etc. For example:

- **VGGish:** An embedding model based on CNN used for audio particularly music and speech.
- **Wav2vec:** This model generates embeddings for raw speech audio which is effective for speech to text tasks.

3. Sentence or document embedding models

These models aims to convert sentences or documents into numerical vectors representing entire sentences or documents rather than just individual words.

- **Doc2vec:** This model is an extension of Word2vec that generates embeddings for whole documents by considering the context of the words in the document.
- **InferSent:** This is a sentence encoder that learns to map sentences into embeddings for various tasks.

4. Image embedding models

These models represent images as vectors, enabling tasks like image recognition and retrieval.

- **CNN (Convolutional neural networks):** Machine learning models like ResNet and VGG extract features from images and are used to generate image classification and recognition embeddings.
- **CLIP (Contrastive language image pre training):** This model connects images and textual descriptions by generating embeddings for both and aligning them in the same vector space.

Dense VS Sparse Embeddings

Feature	Dense Embeddings	Sparse Embeddings
Definition	Vectors where most values are non-zero. Generated by Neural Networks (e.g., BERT, GPT).	Vectors where most values are zero. Generated by statistical methods (BM25) or learned models (SPLADE).
Dimensionality	Fixed and relatively low (e.g., 768, 1536, 3072).	Extremely high, typically equal to vocabulary size (e.g., 30,000+).
Semantic Capability	Captures deep meaning, context, and synonyms.	Captures exact keyword presence. No inherent semantic understanding.
Use Case	Semantic Search, Question Answering, Multimodal retrieval.	Keyword search, exact matching of error codes, IDs, Part numbers.
Storage	Compact, fixed size per document.	Requires specialized compression (inverted index) due to high dimensionality.

Multimodal Embeddings

These models are trained on massive datasets of image-caption pairs to learn a "Shared Embedding Space".

In a shared space, the vector for an image of a hydraulic pump and the vector for the text "a photo of a hydraulic pump" are mathematically close (high cosine similarity). This enables powerful cross-modal capabilities:

1. **Text-to-Image Retrieval:** A user queries "wiring diagram for pump," and the system retrieves the actual image of the diagram because the embedding of the query text matches the embedding of the image.
2. **Image-to-Text Retrieval:** Using an image as a query to find relevant textual descriptions or manuals.

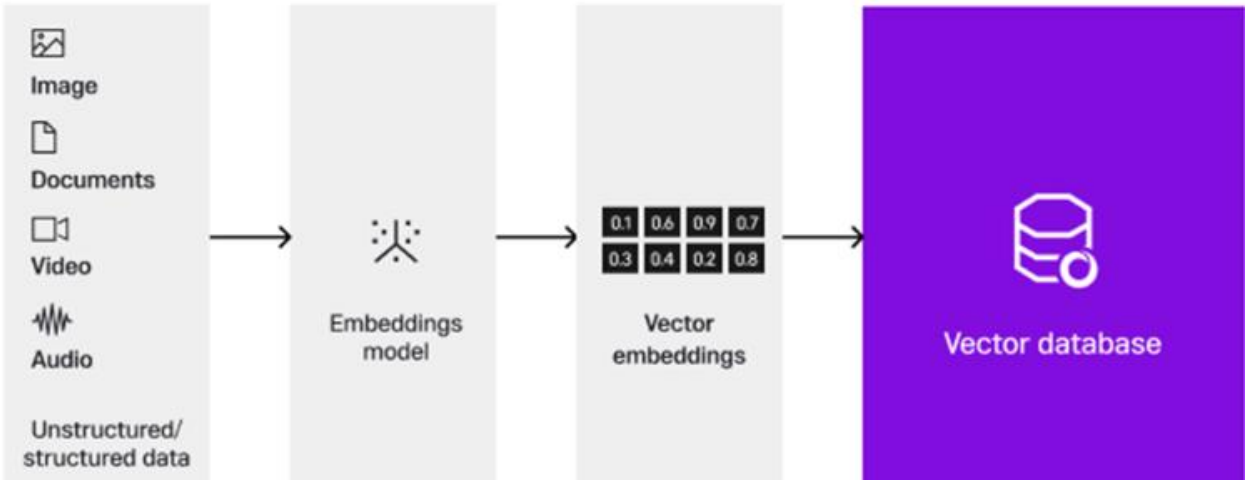
Implementing this requires a dual-path pipeline where text chunks run through a text encoder (like OpenAI's text-embedding-3) and images run through a vision encoder (like CLIP ViT-B/32), both stored in aligned or separate collections within MilvusCka

Vector Database Fundamentals

Vector databases are specialized database systems designed to manage, store, and retrieve high-dimensional data, typically represented as vectors. These vectors are numerical representations of complex data points, such as images, text, or audio. Vector databases are specialized database systems designed to manage, store, and retrieve high-dimensional data, typically represented as vectors. These vectors are numerical representations of complex data points, such as images, text, or audio.

Types of vector databases

Vector databases can be categorized based on the data types they handle, the indexing techniques they use, the storage models they implement, and the architectures they are based upon.



How a vector database works

Architecture

The architecture of vector databases includes the following:

- Storage layer: This layer is responsible for storing vector data. It may use traditional database storage mechanisms but is optimized for vector storage.
- Indexing layer: Here, vectors are indexed to allow efficient querying. This layer uses algorithms and data structures suited for high-dimensional data.
- Query processing layer: This layer handles the processing of queries. It interprets queries, accesses the appropriate index, and retrieves the relevant vectors.
- Similarity computation: This is an integral part of the query processing layer, where the similarity between the query vector and the database vectors is computed.

Data representation

Traditional databases handle structured data (tables, rows and columns). Vector databases manage unstructured or semi-structured data represented as arrays of numbers

Indexing and search

Because vectors can have hundreds or thousands of dimensions, exact search across all vectors is slow. Vector databases therefore use specialized indexing techniques. Graph-based indexes such as *Hierarchical Navigable Small World* (HNSW) provide high recall and fast search in high dimensions. Tree-based indexes (KD-trees or R-trees) work for low dimensions, while hashing-based methods use locality-sensitive hashing for rapid retrieval.

Indexing Algorithms: HNSW vs. IVFFlat

The index is the core component that makes vector search fast. The two most prominent algorithms supported by databases like Milvus and PGVector are HNSW and IVFFlat.

Hierarchical Navigable Small World (HNSW)

HNSW is currently considered the state-of-the-art algorithm for in-memory vector search. It relies on a multi-layer graph structure.

Mechanism:

It builds a hierarchy of graphs. The top layers are sparse (analogous to an express highway or a skip list), allowing the search algorithm to quickly traverse across the vector space to the

general neighborhood of the query. The lower layers are dense, allowing for fine-grained traversal to find the exact nearest neighbors.

Pros: Extremely fast query speed, high recall (accuracy), and robustness to diverse data distributions.

Cons: High memory usage (it stores the full graph structure in RAM) and slower build times compared to IVF.

Best For: Applications requiring low-latency real-time search (like the Chatbot in the assignment) where memory is available.

Inverted File Flat (IVFFlat)

IVFFlat is a quantization-based index.

Mechanism:

It utilizes K-Means clustering to partition the vector space into N regions (Voronoi cells). During a search, the system first identifies which cluster centroid is closest to the query vector and then only searches the vectors within that cluster

Pros: Much lower memory footprint (especially when combined with Product Quantization to compress vectors), faster build times.

Cons: Lower recall (accuracy) than HNSW. If the correct answer lies just outside the checked clusters (the "edge problem"), it will be missed.

Best For: Massive datasets (millions or billions of vectors) where memory is a constraint.

References

1. Qdrant. *What is RAG in AI?* https://qdrant.tech/articles/what-is-rag-in-ai/
2. GeeksforGeeks. *What is Retrieval-Augmented Generation (RAG)?* https://www.geeksforgeeks.org/nlp/what-is-retrieval-augmented-generation-rag/
3. GeeksforGeeks. *RAG Architecture in NLP* https://www.geeksforgeeks.org/nlp/rag-architecture/
4. NVIDIA Technical Blog. *An Easy Introduction to Multimodal Retrieval-Augmented Generation (RAG)*
5. LangChain Documentation. *Text Embedding Integrations* https://docs.langchain.com/oss/python/integrations/text_embedding
6. Nexla AI Infrastructure Blog. *Vector Databases: Tutorial, Best Practices & Examples* https://nexla.com/ai-infrastructure/vector-databases/
7. GeeksforGeeks. *What is a Vector Database?* https://www.geeksforgeeks.org/data-science/what-is-a-vector-database/