**System Development**

In this project we employed Retrieval-Augmented Generation (RAG) a process that improves the output of large language models (LLMs) by using as context to its generated output external knowledge bases. The framework follows a preprocessed data (i.e cleaned data) which is embedded into vector representations for efficient similarity searches using cosine similarity and stored in a cloud-based vector database like Pinecone. A generative AI model, such as Meta’s LLaMA or Google’s Gemini, is integrated with a vector database retriever to implement a Retrieval-Augmented Generation (RAG) approach, enabling user to get fitness and diet recommendation.

**- Data Collection** -

**- Data Preprocessing** -

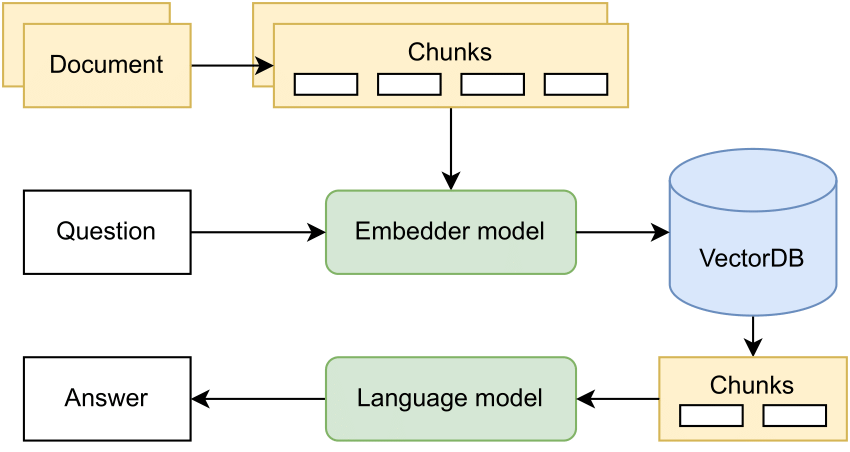
**- Data Embedding**: After Preprocessing, we will then use a embedding model to convert data into vector embedding which the model has access to and can add to its context, also for optimal similarity search within document using cosine similarity. Then finally upload the data to cloud based vector database (Pinecone).

**- Model Design:** Use a generative AI model (e.g., Google’s Gemini GPT-based models) they are open source free to use models, we set up an agent like system with the LLM to function as the fitness recommender. The LLMs are decoder only transformer model which is a variation of the popular transformer model released by Google. Implement the RAG approach by connecting the vector database retriever with cosine similarity search for reliable source-backed responses.

**- Deployment:** We deployed our work to cloud services to ensure accessibility for users anytime and anywhere. For the front-end application, we utilized Vercel, which provided seamless deployment, automatic scaling, and a global content delivery network (CDN) for optimal performance. For the back-end, we chose Render, leveraging its automated builds, scalability, and efficient handling of API requests. This setup enabled a smooth and responsive user experience while maintaining reliability and ease of maintenance

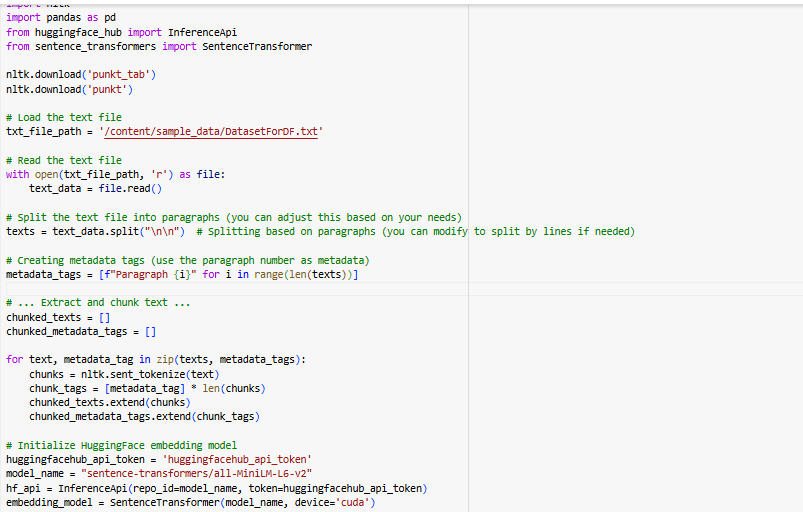
**Framework**

The framework enhances the intelligence of a fitness and diet app by combining retrieval-based search with generative AI for personalized recommendations. In this setup, Pinecone serves as the vector database, efficiently storing and retrieving relevant fitness chunked, nutritional information, and user-specific data. The Gemini model acts as the core AI, generating insightful responses based on retrieved content, ensuring accuracy and contextual relevance. To create high-quality vector representations, the Sentence Transformer model is used for embeddings, converting user queries and database entries into meaningful vector spaces. This architecture enables real-time, context-aware recommendations, such as personalized meal plans and exercise routines, by retrieving the most relevant data before generating responses, thereby improving accuracy, scalability, and user engagement.

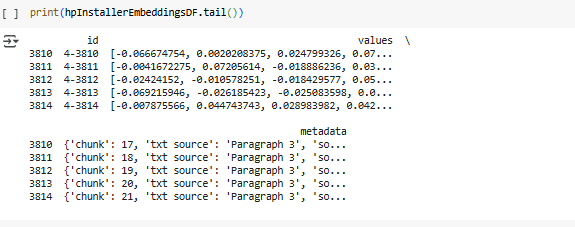


**Data Embedding**

Embedding data refers to the process of converting text, words, or other symbolic data into dense numerical vectors within a continuous high-dimensional space. In the context of natural language processing (NLP), these embeddings represent semantic and syntactic relationships between words or phrases as distances and directions in this vector space, meaning words with similar meanings or usage patterns will be located closer together. For instance, in a well-trained embedding space, the vector for "happy" might be closer to "joyful" than to "sad," reflecting their semantic relationships. Modern embedding techniques like Word2Vec, GloVe, or those used in transformer models like BERT and NLLB go beyond simple one-hot encoding by capturing rich contextual information through methods such as predicting words from their context (skip-gram) or predicting context from words (continuous bag of words). These embeddings serve as the foundation for many NLP tasks because they allow machine learning models to process text mathematically - the vectors can be added, subtracted, and measured for similarity, enabling operations like finding analogies or measuring semantic similarity between words or documents. For this system we will make use of **sentence-transformers/all-MiniLM-L6-v2** from huggingface we will then upsert the vectors to pinecone database.

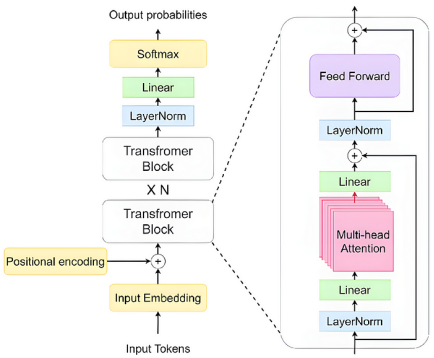


The code above shows the process of converting text to vector and we also showed the embeddings output below and we have saved it in a pandas dataframe ready to the upsert into a vector db.



Model design

The architecture of the Gemini model builds upon the principles of a decoder-only transformer architecture, optimized for multimodal inputs and tasks. In a decoder-only model, the architecture consists primarily of a stack of transformer decoder layers, each featuring self-attention and feedforward components. This design allows the model to autoregressively predict the next token or element based on the input sequence, making it highly effective for tasks like text generation and sequential data processing. The Gemini model extends this architecture to accommodate multimodal data by integrating specialized input embeddings and cross-modal attention mechanisms. These components enable the model to process diverse data types—such as audio, images, videos, and text—within the same framework

****

Our application leverages **LangChain** to build a Retrieval-Augmented Generation (RAG) system for a personalized diet and fitness assistant. To enable intelligent conversational retrieval, we implemented LangChain's retrieval and history-aware mechanisms by utilizing `**create\_history\_aware\_retriever`** and **`create\_retrieval\_chain**`. The memory system, powered by **`ConversationBufferMemory**`, allows our chatbot to maintain context by remembering past interactions, ensuring a seamless and contextual conversation. We employed Pinecone as the vector database, storing embeddings generated using the Hugging Face Sentence Transformer model (`all-MiniLM-L6-v2`) to efficiently retrieve relevant diet and fitness information. The core AI model, Google's Gemini, is integrated via `**ChatGoogleGenerativeAI`**, enabling context-aware responses with personalized recommendations. To structure responses, we designed a custom prompt template, instructing the model to provide actionable and precise recommendations based on a user's query, chat history, and retrieved documents. The chatbot's response generation pipeline is built with LangChain's document combination chain, `create\_stuff\_documents\_chain`, which integrates retrieved information with user queries for more insightful answers. Additionally, we implemented a Flask API with CORS support, allowing seamless interaction with the system via a `/chat` endpoint, where user messages are processed using the `generate\_response` function. This function ensures clean and well-structured outputs, filtering out unnecessary characters and long dashes. To maintain persistent session-based conversations, we use **`ChatMessageHistory`** to track interactions, ensuring continuity in responses. Overall, by combining LangChain’s powerful retrieval and memory capabilities with Pinecone for vector storage and Gemini for generation, our application delivers highly personalized, context-aware diet and fitness recommendations in real time.

**Deployment**

To make our diet and fitness chatbot easily accessible, we deployed the backend on Render and built a user-friendly chat interface using React with the Chatbot Kit library, deploying it to Vercel for a seamless frontend experience. The backend, developed in Flask, was set up on Render to handle API requests efficiently, ensuring our chatbot could retrieve and generate responses in real time. We configured the Flask app to expose an endpoint (/chat) where users' messages are processed, and responses are generated using our LangChain-based Retrieval-Augmented Generation (RAG) pipeline powered by Pinecone for vector search and Google’s Gemini for response generation. To deploy on Render, we created a render.yaml configuration file, specifying the environment variables, dependencies, and runtime settings necessary for smooth operation. Our frontend chat interface was built using React and the Chatbot Kit library, which provided a structured way to design interactive chat components with custom styling and conversation flow. The React app was designed to send user queries to the Render backend, receive responses, and display them in a clean, intuitive chat UI. We utilized Axios to handle API requests between the frontend and backend, ensuring efficient communication. The chatbot UI was further enhanced with features like user message history display, loading indicators, and smooth animations for a better experience. Once the React app was complete, we deployed it to Vercel, leveraging its fast global CDN for quick response times and seamless updates. By combining Render’s backend hosting capabilities with Vercel’s optimized frontend deployment, we created a fully functional, scalable, and responsive chatbot system that delivers real-time personalized fitness and diet recommendations to users anywhere, anytime.

**Backend Technologies:**

Python – Core language for backend development.

Flask – Lightweight web framework for handling API requests.

LangChain – Framework for building the Retrieval-Augmented Generation (RAG) pipeline.

Pinecone – Vector database for storing and retrieving embeddings.

Google Gemini – Large language model for generating responses.

Sentence Transformers (Hugging Face) – Used for text embeddings (all-MiniLM-L6-v2).

Render – Cloud platform for deploying the backend API.

OpenAI Embeddings (optional) – Alternative embedding model for vector representation.

**Frontend Technologies:**

React.js – Frontend framework for building the chatbot UI.

Chatbot Kit – React library for designing chatbot interfaces.

Axios – HTTP client for handling API requests between the frontend and backend.

Vercel – Deployment platform for the React frontend.

**Other Tools & Services:**

dotenv – Used to manage environment variables.

Flask-CORS – Enables cross-origin resource sharing between frontend and backend.

ReGex (re module) – Used for cleaning chatbot responses.

Git & GitHub – Version control and collaboration.

Render.yaml – Configuration file for Render deployment.