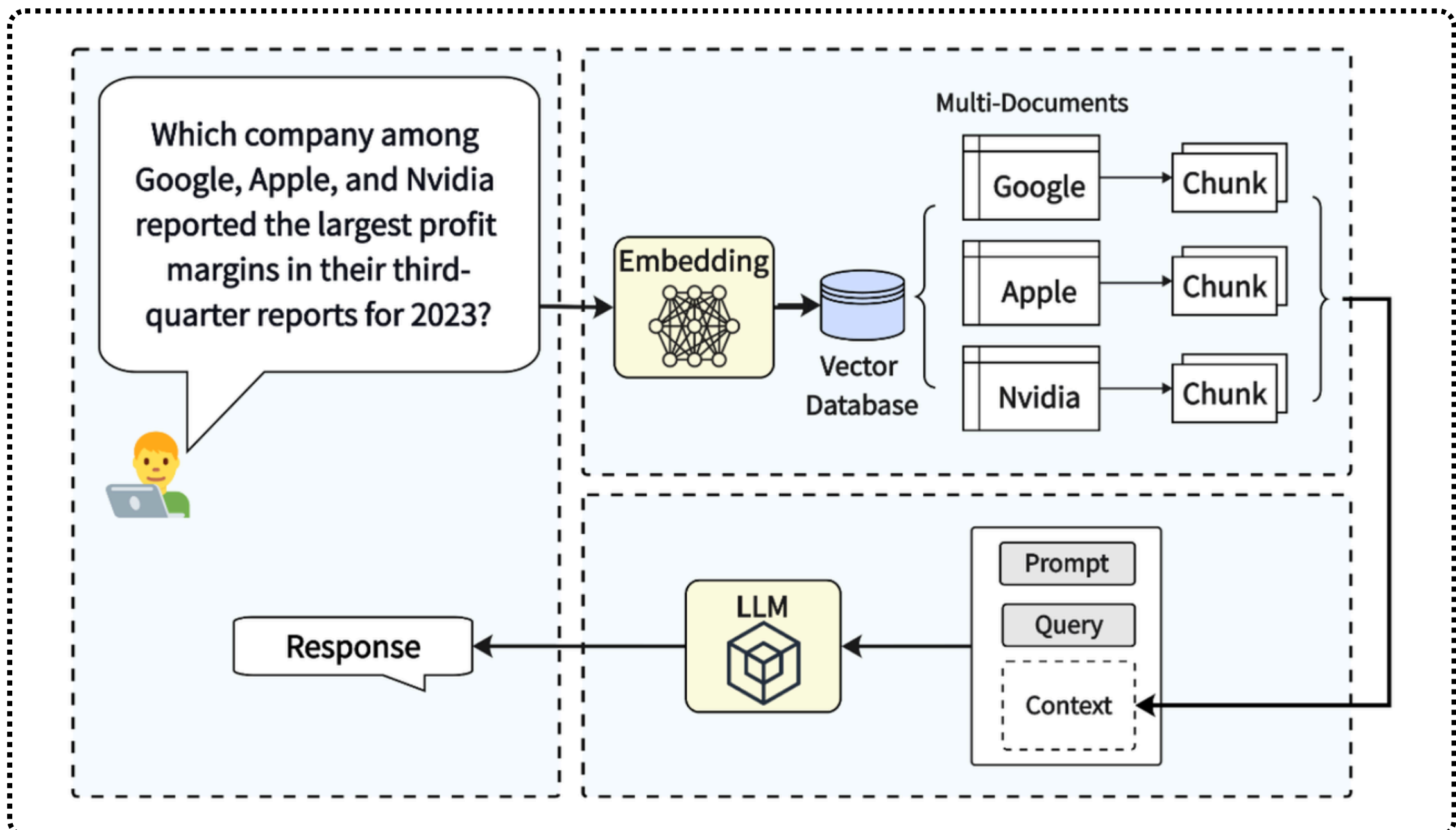
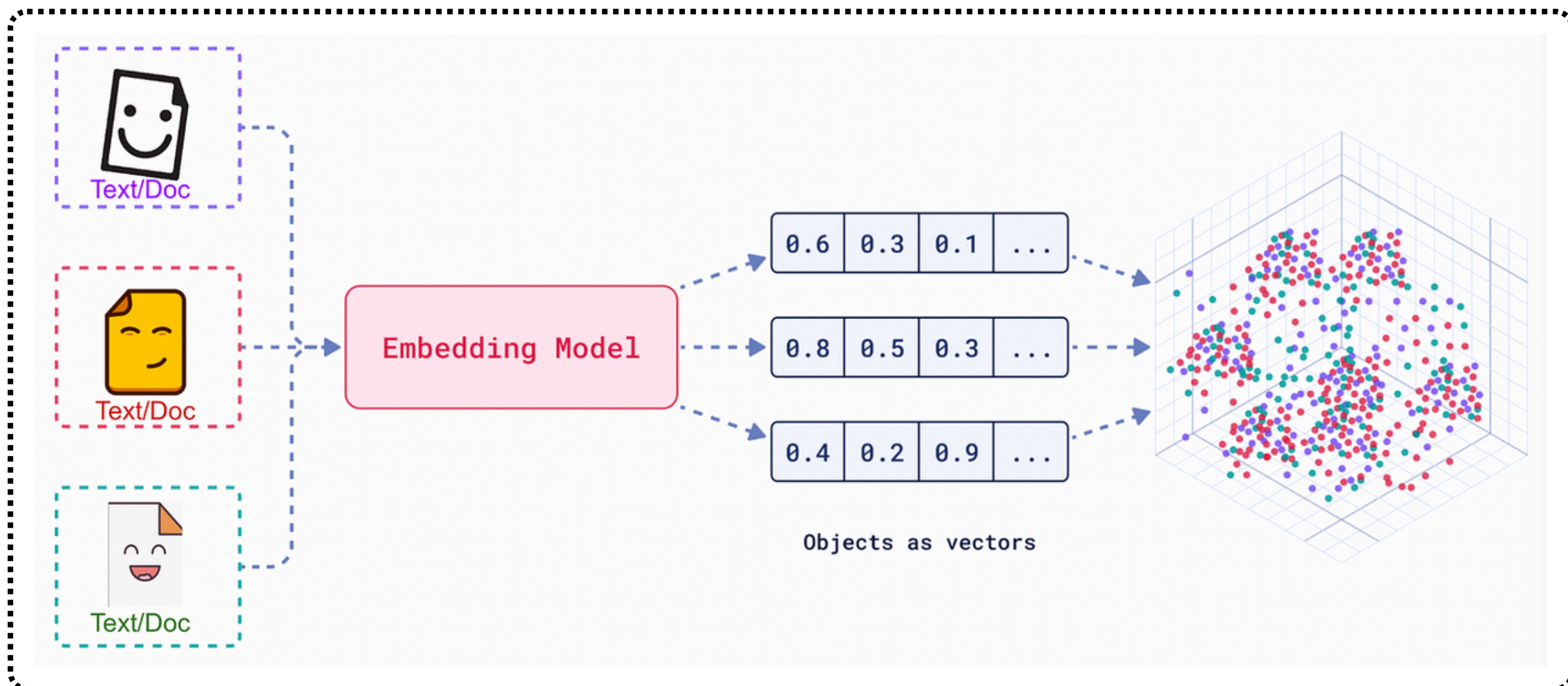


Mastering RAG

Embedding in RAG



What are Embeddings?

Embeddings are dense vector representations of text that capture semantic meanings, relationships, and contextual nuances. These numerical vectors allow machines to compare, retrieve, and process text efficiently in high-dimensional space.

In RAG, embeddings are used to:

- Retrieve relevant documents based on query similarity.
- Improve response accuracy by feeding relevant data into the generative model.
- Enhance contextual understanding by maintaining semantic similarity between queries and retrieved chunks.



Types of Embeddings in RAG

1. Word-Level Embeddings

Example Models: Word2Vec, GloVe, FastText

Word-level embeddings assign fixed vector representations to individual words. These embeddings are useful for basic similarity tasks but lack contextual awareness.

2. Sentence-Level Embeddings

Example Models: Sentence-BERT (SBERT), Universal Sentence Encoder (USE)

Sentence embeddings capture the meaning of entire sentences, making them ideal for document retrieval and RAG applications where context matters.

3. Document-Level Embeddings

Example Models: Doc2Vec, Longformer, Hierarchical Transformers

Document embeddings represent entire documents as vectors, suitable for retrieving large passages or articles in knowledge-intensive RAG applications.



4. Contextual Embeddings

Example Models: BERT, RoBERTa, GPT-based models
Contextual embeddings generate different vector representations based on word usage in context, making them highly effective for NLP tasks, including RAG-based retrieval.

Implementing Embeddings in RAG

Popular NLP libraries like Hugging Face's transformers, sentence-transformers, and OpenAI's API can be used to generate embeddings.

Example: Using Sentence-BERT for Embeddings

```
from sentence_transformers import SentenceTransformer

# Load a pre-trained embedding model
model = SentenceTransformer('all-MiniLM-L6-v2')

# Generate embeddings for text
text = "What are embeddings in RAG?"
embedding = model.encode(text)

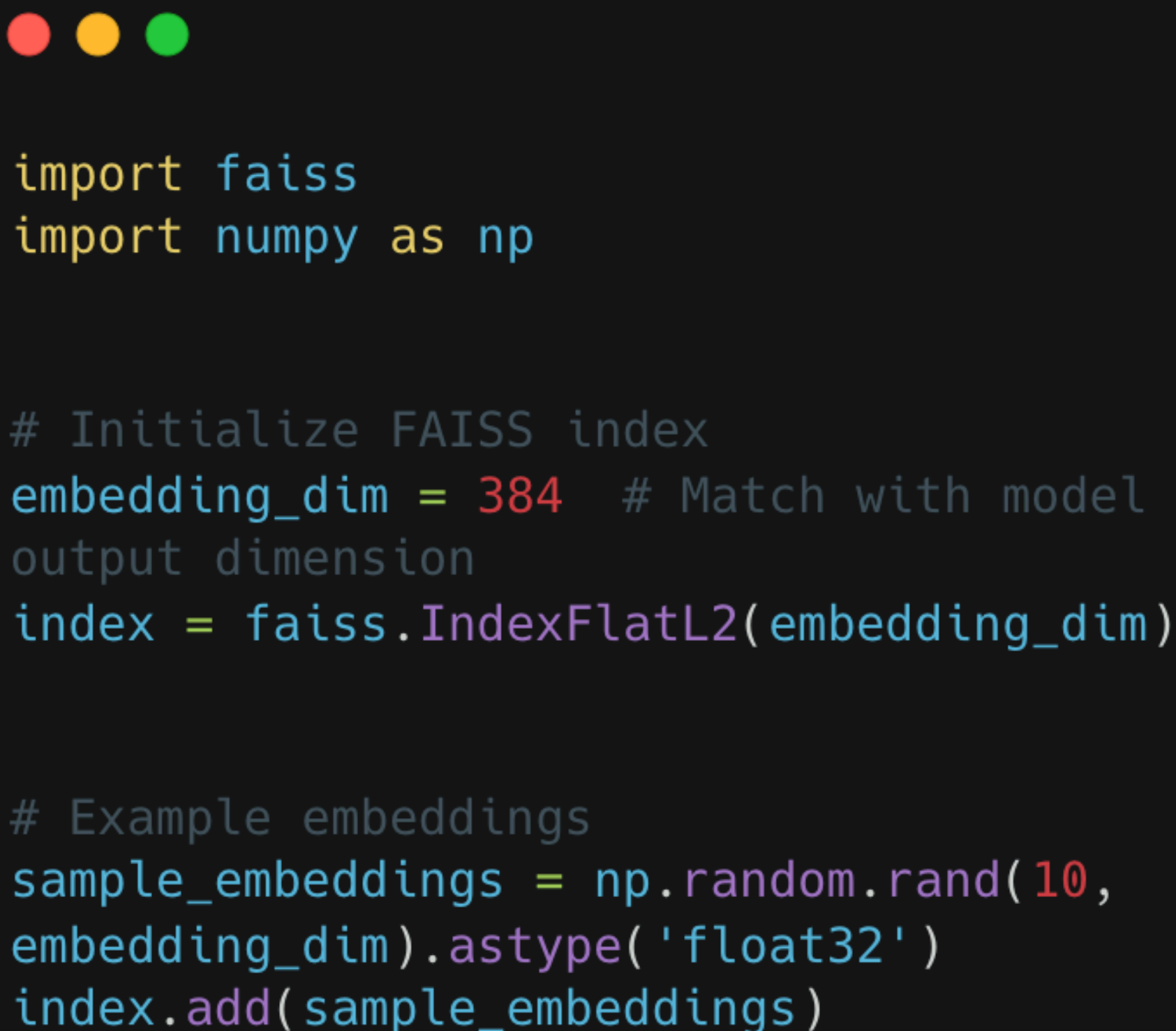
print(embedding.shape) # Output: (384,) vector
representation
```



2. Storing Embeddings

Vector databases such as FAISS, Pinecone, Weaviate, ChromaDB allow efficient storage and retrieval of embeddings.

Example: Storing Embeddings with FAISS



```
import faiss
import numpy as np

# Initialize FAISS index
embedding_dim = 384 # Match with model
output dimension
index = faiss.IndexFlatL2(embedding_dim)


# Example embeddings
sample_embeddings = np.random.rand(10,
embedding_dim).astype('float32')
index.add(sample_embeddings)
```



3. Retrieving Similar Chunks

When a user inputs a query, its embedding is computed and compared against stored embeddings to retrieve the most relevant information.

Example: Searching in FAISS



```
query_embedding = np.random.rand(1,  
embedding_dim).astype('float32')  
distance, indices =  
index.search(query_embedding, k=3)  
print("Top retrieved document indices:",  
indices)
```



Best Practices for Using Embeddings in RAG

- **Choose the Right Embedding Model** – Use domain-specific models if applicable (e.g., SciBERT for scientific literature).
- **Optimize Chunk Size** – Ensure the text chunks are neither too small (losing context) nor too large (causing dilution of information).
- **Leverage Vector Databases** – Use FAISS, Pinecone, or ChromaDB for scalable and efficient retrieval.
- **Fine-tune Embeddings** – Adapt embedding models for specific tasks to improve relevance and accuracy.
- **Evaluate Embedding Performance** – Use similarity metrics (cosine similarity, L2 distance) to assess retrieval effectiveness.