

Embeddings

Turning text (and other data) into vectors for semantic search, clustering, classification, and RAG.

Last reviewed 28 Sep 2026 ·

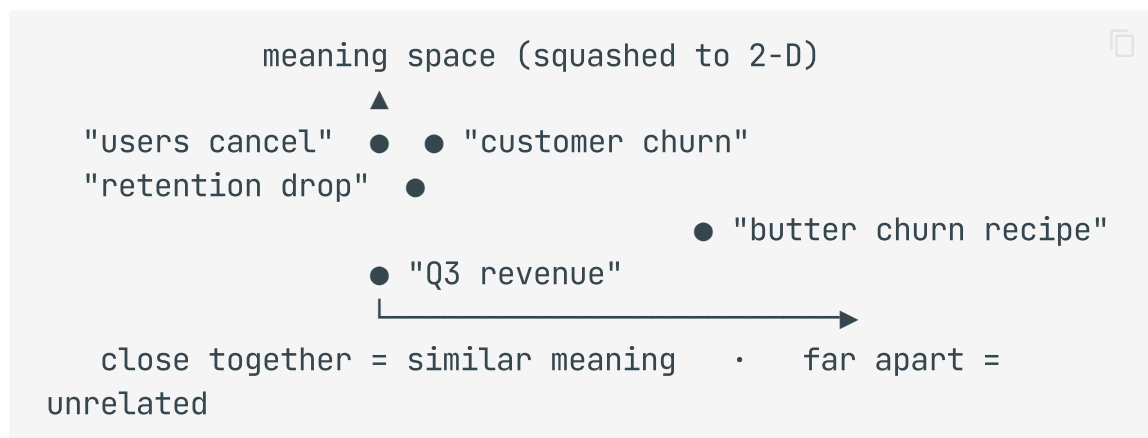
Prerequisites: [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) ·

Related: [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) · [Vector Databases](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) · [Glossary](https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/) (https://sarangambekar1997.github.io/data-engineering-handbook/99-reference/glossary/)

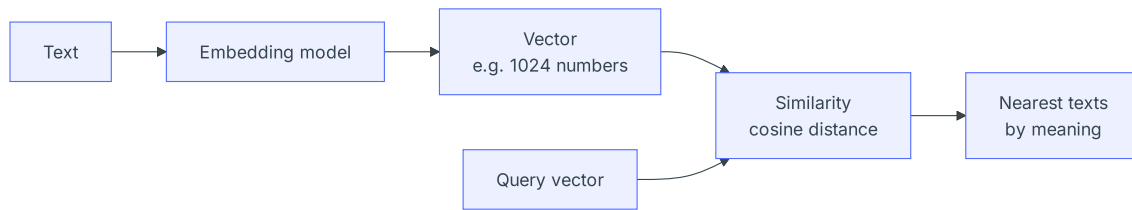
Overview

Challenge: Keyword matching compares words, not meaning. A search for "customer churn" misses a document titled "Why users cancel their subscriptions", although it is exactly what was needed. SQL `LIKE` and full-text search share this limitation.

Solution: an embedding model converts text into a vector of numbers (for example, 1,024 dimensions) such that texts with *similar meaning* are close together. "Customer churn" and "users cancelling subscriptions" map to nearby vectors, while "customer churn" and "butter churn" do not. Finding similar content then becomes a distance calculation.



Relevance to data engineering: embeddings underpin RAG and semantic search, and support record deduplication, classification without model training, and clustering. Generating and storing them at scale — batching, incremental updates, versioning — is a data pipeline problem.



On this page

Basic - [What Are Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#what-are-embeddings) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#what-are-embeddings) - [Generating Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#generating-embeddings) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#generating-embeddings) - [Cosine Similarity](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#cosine-similarity) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#cosine-similarity)

Intermediate - [Choosing an Embedding Model](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#choosing-an-embedding-model) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#choosing-an-embedding-model) - [Embedding in Batches](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#embedding-in-batches) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#embedding-in-batches) - [Semantic Search Without a Vector DB](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#semantic-search-without-a-vector-db) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#semantic-search-without-a-vector-db) - [Chunking Text for Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#chunking-text-for-embeddings) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#chunking-text-for-embeddings)

Advanced - [Use Cases Beyond RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#use-cases-beyond-rag) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#use-cases-beyond-rag) - [Fine-Tuning Embeddings](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#fine-tuning-embeddings) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#fine-tuning-embeddings) - [Storing Embeddings at Scale](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#storing-embeddings-at-scale) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#storing-embeddings-at-scale) - [Embedding Pipelines in Production](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#embedding-pipelines-in-production) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#embedding-pipelines-in-production)

Reference - [Common Pitfalls](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#common-pitfalls) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#common-pitfalls) - [Cheat Sheet](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#cheat-sheet) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#cheat-sheet) - [Interview Questions](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#interview-questions) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#interview-questions) - [Further Reading](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#further-reading) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/embeddings/#further-reading)

What Are Embeddings

An embedding is a vector (list of floats) that represents the semantic meaning of a piece of text. Texts with similar meanings have vectors that are close together in space.

```
"How many orders were placed today?"      → [0.021, -0.143,
0.087, ...] (1536 dims)
"What is today's order count?"             → [0.019, -0.141,
0.091, ...] (very close)
"The capital of France is Paris"           → [-0.432, 0.201,
-0.055, ...] (far away)
```

What embeddings capture: - Semantic similarity (not just keyword overlap) - "dog" and "canine" are close; "dog" and "cat" are closer than "dog" and "table" - Context and intent, not just surface form

Where embeddings are used: - **RAG:** find relevant documents to give to an LLM -

Semantic search: search by meaning, not keywords - **Deduplication:** find near-duplicate records - **Clustering:** group similar items - **Classification:** classify text by proximity to class examples - **Anomaly detection:** flag records far from normal

Generating Embeddings

```
# OpenAI
from openai import OpenAI
import numpy as np

client = OpenAI()

def embed(texts: list[str], model="text-embedding-3-small")
    -> np.ndarray:
    response = client.embeddings.create(
        input=texts,
        model=model
    )
    return np.array([r.embedding for r in response.data])

# Single text
vec = embed(["What is Apache Kafka?"])[0]
print(f"Dimensions: {len(vec)}")    # 1536 for text-
embedding-3-small

# Multiple texts at once (up to 2048 in one call)
vecs = embed([
    "Apache Kafka is a distributed event streaming
```

```
platform",
    "A cloud data warehouse stores structured data for
analytics",
    "SQL transformations turn raw tables into analytics-
ready models",
])
print(f"Shape: {vecs.shape}")    # (3, 1536)
```

```
# Anthropic (via Voyage AI – Anthropic's embedding partner)
# pip install voyageai
import voyageai

vo = voyageai.Client()    # reads VOYAGE_API_KEY

result = vo.embed(
    ["Apache Kafka is a distributed event streaming
platform"],
    model="voyage-4",
    input_type="document"    # use input_type="query" for
search queries
)
vec = result.embeddings[0]
print(len(vec))    # 1024
```

```
# Sentence Transformers – free, runs locally
# pip install sentence-transformers
from sentence_transformers import SentenceTransformer

model = SentenceTransformer("all-MiniLM-L6-v2")    # 384
dims, fast
vecs = model.encode([
    "Apache Kafka is a distributed event streaming
platform",
    "A cloud data warehouse stores structured data for
analytics",
])
print(vecs.shape)    # (2, 384)
```

Cosine Similarity

The most common way to measure how similar two embeddings are.

```
import numpy as np

def cosine_similarity(a: np.ndarray, b: np.ndarray) -> float:
    return np.dot(a, b) / (np.linalg.norm(a) *
                           np.linalg.norm(b))

# Range: -1 (opposite) to 1 (identical)
# In practice the scale depends on the model, so there is
# no universal threshold: choose one
# from labelled pairs of your own data

query    = embed(["What is Kafka?"])[0]
doc1     = embed(["Apache Kafka is a distributed event
streaming platform"])[0]
doc2     = embed(["The Eiffel Tower is in Paris"])[0]

print(cosine_similarity(query, doc1))    # higher: the
related pair
print(cosine_similarity(query, doc2))    # lower: the
unrelated pair (the exact values depend on the model)
```

```
# Batch similarity: query vs many documents
def top_k_similar(query_vec: np.ndarray, doc_vecs:
np.ndarray, k: int = 5) -> list[int]:
    # Normalize all vectors
    q = query_vec / np.linalg.norm(query_vec)
    D = doc_vecs / np.linalg.norm(doc_vecs, axis=1,
keepdims=True)
    scores = D @ q    # dot product = cosine similarity for
normalized vectors
    return np.argsort(scores)[::-1][:k].tolist()
```

Choosing an Embedding Model

Model	Dims	Speed	Cost	Best for
<code>text-embedding-3-small</code> (OpenAI)	1536	Fast	Low	General purpose, RAG
<code>text-embedding-3-large</code> (OpenAI)	3072	Medium	Medium	High-accuracy retrieval
<code>voyage-4</code> (Voyage AI)	1024 (256, 512, 2048 selectable)	Fast	Low	General, Anthropic ecosystem
<code>voyage-4-large</code> (Voyage AI)	1024 (256, 512, 2048 selectable)	Medium	Medium	High-accuracy retrieval
<code>all-MiniLM-L6-v2</code> (local)	384	Very fast	Free	Prototyping, offline
<code>bge-large-en</code> (local)	1024	Medium	Free	Production on-prem

Embedding models are released often: Voyage's current generation is 4, and `voyage-3` and `voyage-3-large` are its previous one. Others to compare include Cohere, Gemini, open-weight models like BGE, E5, and Nomic). Compare on the [MTEB leaderboard](https://huggingface.co/spaces/mteb/leaderboard) (<https://huggingface.co/spaces/mteb/leaderboard>), then test on *your* data — leaderboard rank doesn't guarantee the best retrieval for your domain.

Tips: - Start with `text-embedding-3-small` — it's fast and good enough for most RAG - Only upgrade to `large` if retrieval quality is measurably worse - Check the dimension limit of your vector store: pgvector indexes a `vector` column of up to 2,000 dimensions (`halfvec`: 4,000), so `text-embedding-3-large`'s 3072 needs the `dimensions` parameter or `halfvec` - Local models (sentence-transformers) are great for prototyping and cost-sensitive workloads - Normalize vectors before storing — speeds up dot-product search

Embedding in Batches

```
from openai import OpenAI
import numpy as np
import time
```



```

client = OpenAI()

def embed_large_dataset(texts: list[str], batch_size: int =
500) -> np.ndarray:
    """Embed a large list of texts efficiently in
    batches."""
    all_embeddings = []

    for i in range(0, len(texts), batch_size):
        batch = texts[i:i + batch_size]
        # Remove newlines – they can hurt embedding quality
        batch = [t.replace("\n", " ") for t in batch]

        response = client.embeddings.create(
            input=batch,
            model="text-embedding-3-small"
        )
        batch_embeddings = [r.embedding for r in
response.data]
        all_embeddings.extend(batch_embeddings)

        if i % 5000 == 0:
            print(f"Embedded {i}/{len(texts)}")
            time.sleep(0.1) # gentle rate limiting

    return np.array(all_embeddings)

```

Semantic Search Without a Vector DB

For small datasets (< 100k), skip the vector DB and use numpy directly.

```

import numpy as np
import json
from openai import OpenAI

client = OpenAI()

# Build an in-memory index
documents = [

```

```

    "Kafka is used for real-time event streaming",
    "SQL transformations turn raw tables into analytics-
ready models",
    "Airflow orchestrates data pipelines as DAGs",
    "A cloud data warehouse separates storage from
compute",
    "Delta Lake adds ACID transactions to data lakes",
]

print("Building embeddings index...")
doc_embeddings = embed_large_dataset(documents)
doc_embeddings = doc_embeddings /
np.linalg.norm(doc_embeddings, axis=1, keepdims=True)

# Save to disk
np.save("embeddings.npy", doc_embeddings)
with open("documents.json", "w") as f:
    json.dump(documents, f)

# Search
def search(query: str, k: int = 3) -> list[dict]:
    q_vec = embed([query])[0]
    q_vec = q_vec / np.linalg.norm(q_vec)
    scores = doc_embeddings @ q_vec
    top_k = np.argsort(scores)[::-1][:k]
    return [{"text": documents[i], "score":
float(scores[i])} for i in top_k]

results = search("how do I schedule a data pipeline?")
for r in results:
    print(f"{r['score']:.3f} {r['text']}")
# 0.821 Airflow orchestrates data pipelines as DAGs
# 0.612 SQL transformations turn raw tables into
analytics-ready models
# 0.543 Kafka is used for real-time event streaming

```

Chunking Text for Embeddings

Long documents must be split into chunks before embedding. The chunk strategy dramatically affects retrieval quality.

```

from typing import Generator

def chunk_by_tokens(text: str, max_tokens: int = 512,
overlap: int = 50) -> list[str]:
    """
    Split text into overlapping chunks.
    overlap: tokens shared between adjacent chunks –
    preserves context at boundaries.
    """
    words = text.split()
    chunks = []
    start = 0
    while start < len(words):
        end = start + max_tokens
        chunk = " ".join(words[start:end])
        chunks.append(chunk)
        start += max_tokens - overlap
    return chunks

def chunk_by_paragraph(text: str, max_chars: int = 1000) ->
list[str]:
    """Split on paragraph breaks, merge short
    paragraphs."""
    paragraphs = [p.strip() for p in text.split("\n\n") if
p.strip()]
    chunks = []
    current = ""
    for para in paragraphs:
        if len(current) + len(para) < max_chars:
            current += "\n\n" + para if current else para
        else:
            if current:
                chunks.append(current)
            current = para
    if current:
        chunks.append(current)
    return chunks

def chunk_markdown_by_section(text: str) -> list[dict]:
    """Split markdown on ## headings, preserving section
    title."""
    import re

```

```

sections = re.split(r'\n(?:##\s)', text)
return [
    {"title": s.split("\n")[0].strip("# "), "content":
s}
    for s in sections if s.strip()
]

```

Chunking rules: - 256–512 tokens is a good starting point for most RAG - Overlap of 10–20% helps context at chunk boundaries - Never split mid-sentence - Add document metadata (title, source URL, section) to each chunk before embedding - Prefer semantic splits (paragraphs, sections) over fixed-size splits

Use Cases Beyond RAG

Semantic deduplication

```

def find_duplicates(texts: list[str], threshold: float =
0.95) -> list[tuple[int, int, float]]:
    vecs = embed(texts)
    vecs = vecs / np.linalg.norm(vecs, axis=1,
keepdims=True)
    sims = vecs @ vecs.T
    dupes = []
    for i in range(len(texts)):
        for j in range(i + 1, len(texts)):
            if sims[i, j] > threshold:
                dupes.append((i, j, float(sims[i, j])))
    return dupes

```

Zero-shot classification via embeddings

```

def classify(text: str, labels: list[str]) -> str:
    """Classify text by finding the closest label
embedding."""
    text_vec = embed([text])[0]
    label_vecs = embed(labels)
    sims = [cosine_similarity(text_vec, lv) for lv in
label_vecs]
    return labels[np.argmax(sims)]

```

```
classify("pipeline failed: connection timeout", ["error",  
"warning", "info"])  
# → "error"
```

Clustering

```
from sklearn.cluster import KMeans  
  
vecs = embed(log_messages)  
kmeans = KMeans(n_clusters=10, random_state=42)  
labels = kmeans.fit_predict(vecs)  
  
# Each cluster = a class of log messages  
for cluster_id in range(10):  
    examples = [log_messages[i] for i, l in  
enumerate(labels) if l == cluster_id][:3]  
    print(f"\nCluster {cluster_id}:")  
    for e in examples:  
        print(f"  {e}")
```

Fine-Tuning Embeddings

When off-the-shelf embeddings don't capture your domain well (e.g., proprietary table names, internal jargon).

```
# Using sentence-transformers with custom training data  
from sentence_transformers import SentenceTransformer,  
InputExample, losses  
from torch.utils.data import DataLoader  
  
# Training pairs: (query, positive_doc, negative_doc)  
train_examples = [  
    InputExample(texts=["orders table", "fct_orders  
contains one row per order", "dim_customer contains  
customer data"]),  
    InputExample(texts=["daily revenue", "SELECT  
SUM(amount) FROM orders GROUP BY date", "SELECT * FROM  
customers"]),
```

```

]

model = SentenceTransformer("all-MiniLM-L6-v2")
loader = DataLoader(train_examples, shuffle=True,
batch_size=16)
loss = losses.TripletLoss(model)

model.fit(train_objectives=[(loader, loss)], epochs=3,
warmup_steps=100)
model.save("./my_domain_embeddings")

```

Storing Embeddings at Scale

For large datasets (> 100k vectors), persist embeddings rather than recomputing.

```

# Store in Parquet (simplest)
import pandas as pd
import numpy as np

def save_embeddings(texts: list[str], embeddings:
np.ndarray, path: str):
    df = pd.DataFrame({
        "text": texts,
        "embedding": [e.tolist() for e in embeddings]
    })
    df.to_parquet(path, index=False)

def load_embeddings(path: str) -> tuple[list[str],
np.ndarray]:
    df = pd.read_parquet(path)
    texts = df["text"].tolist()
    vecs = np.array(df["embedding"].tolist())
    return texts, vecs

```

```

# Store in PostgreSQL with pgvector
# pip install psycopg2-binary pgvector
import psycopg2
from pgvector.psycopg2 import register_vector

```

```

conn =
psycopg2.connect("postgresql://user:pass@localhost/db")

# The extension must exist before register_vector runs: it
# looks the type up and raises
# ProgrammingError("vector type not found in the database")
otherwise
with conn.cursor() as cur:
    cur.execute("CREATE EXTENSION IF NOT EXISTS vector")
conn.commit()
register_vector(conn)

with conn.cursor() as cur:
    cur.execute("""
        CREATE TABLE IF NOT EXISTS embeddings (
            id          SERIAL PRIMARY KEY,
            source       TEXT,
            content      TEXT,
            embedding     vector(1536)
        )
    """)
    # HNSW gives a better speed-recall trade-off than
    # IVFFlat, at the cost of a slower build and more memory.
    # An IVFFlat index must be created after the table
    # holds data, so that it can learn its lists
    cur.execute("CREATE INDEX IF NOT EXISTS emb_idx ON
embeddings USING hnsw (embedding vector_cosine_ops)")
    conn.commit()

# Insert
def upsert_embedding(cur, source: str, content: str, vec:
list[float]):
    cur.execute(
        "INSERT INTO embeddings (source, content,
embedding) VALUES (%s, %s, %s)",
        (source, content, vec)
    )

# Search
def search_pgvector(cur, query_vec: list[float], k: int =
5) -> list[dict]:
    cur.execute(

```

```

        "SELECT source, content, 1 - (embedding <=>
%s::vector) AS score FROM embeddings ORDER BY embedding <=>
%s::vector LIMIT %s",
        (query_vec, query_vec, k)
    )
    return [{"source": r[0], "content": r[1], "score":
r[2]} for r in cur.fetchall()]

```

Embedding Pipelines in Production

```

# Incremental embedding pipeline – only embed new/changed documents
import hashlib
import json

def content_hash(text: str) -> str:
    return hashlib.md5(text.encode()).hexdigest()

class IncrementalEmbeddingPipeline:
    def __init__(self, store):
        self.store = store # dict or DB that maps hash →
embedding

    def embed_new(self, documents: list[dict]) ->
list[dict]:
        """Only embed documents whose content has
changed."""
        to_embed = []
        from_cache = []

        for doc in documents:
            h = content_hash(doc["content"])
            if h in self.store:
                from_cache.append(**doc, "embedding":
self.store[h], "cached": True)}
            else:
                to_embed.append(**doc, "hash": h)}

        if to_embed:
            texts = [d["content"] for d in to_embed]

```

```

        vecs = embed_large_dataset(texts)
        for doc, vec in zip(to_embed, vecs):
            self.store[doc["hash"]] = vec.tolist()
            from_cache.append(**doc, "embedding":
vec.tolist(), "cached": False})

        print(f"Embedded: {len(to_embed)} new,
{len(from_cache) - len(to_embed)} cached")
        return from_cache

```

Common Pitfalls

Pitfall	Symptom	Fix
Mixing embeddings from different models or versions in one index	Similarity scores become meaningless; search quality collapses	Store the model name and version with every vector; re-embed everything when you switch models
Embedding whole documents	Relevant passages drowned out; retrieval returns vague matches	Chunk into passages (a few hundred tokens) with some overlap; embed chunks
Chunks cut mid-sentence or mid-table	Retrieved text lacks context and misleads the LLM	Split on structure (headings, paragraphs, rows); prepend the title or section to each chunk
Query and documents embedded differently	Lower recall with models that expect input types	Use the model's <code>input_type</code> / query vs document prefixes where supported
Re-embedding the full corpus every run	Slow and expensive pipelines	Hash content; embed only new or changed chunks
Comparing unnormalized vectors with dot product	Rankings skewed by vector length	Normalize (or use cosine similarity) consistently
Choosing a model from a leaderboard alone	Worse retrieval on your domain than expected	Build a small labelled query set and measure recall@k on your own data
Using embeddings for exact lookups (IDs, error codes, SKUs)	"Similar" results instead of the exact match	Hybrid search: combine keyword (BM25) with vector search

Pitfall	Symptom	Fix
Storing vectors as JSON text	Huge storage, slow loads	Native vector types (pgvector, vector DBs) or float32 arrays in Parquet

Cheat Sheet

Task	Code
OpenAI	<code>client.embeddings.create(model="text-embedding-3-small", input=texts) → [d.embedding for d in r.data]</code>
Voyage (recommended with Claude)	<code>vo.embed(texts, model="voyage-4", input_type="document") → .embeddings</code>
Local	<code>SentenceTransformer("all-MiniLM-L6-v2").encode(texts, normalize_embeddings=True)</code>
Cosine similarity	<code>a @ b / (np.linalg.norm(a) * np.linalg.norm(b))</code>
Top-k over a matrix (normalized)	<code>scores = M @ q; idx = np.argsort(-scores)[:k]</code>
Shrink dimensions (Matryoshka models)	OpenAI <code>dimensions=512</code> · Voyage <code>output_dimension=512</code>
Content hash for incremental updates	<code>hashlib.sha256(text.encode()).hexdigest()</code>

Similarity metrics: cosine (direction only — the default for text) · dot product (equals cosine for normalized vectors, fastest) · Euclidean / L2 (distance; used by some indexes)

Rules of thumb: chunks of ~200–800 tokens with 10–20% overlap · normalize vectors · keep the model name and dimensions in metadata · batch API calls (hundreds of texts per request) · evaluate with recall@k on real queries

Storage options: NumPy/Parquet (under ~100k vectors, offline) · pgvector (already on Postgres) · a dedicated vector DB (millions of vectors, filtering, low latency) · lakehouse tables with vector search (Databricks, Snowflake)

Interview Questions

Q: What is an embedding, and how is it different from a keyword index? A: An embedding is a dense vector produced by a neural model that captures meaning, so semantically similar texts have nearby vectors even when they share no words. A keyword index (like BM25) matches exact terms and is great for IDs, names, and rare terms, but misses paraphrases. In practice they're complementary, which is why production search often uses both (hybrid search).

Q: Why do we chunk documents before embedding them, and how do you choose a chunk size? A: A single vector can only summarize so much. Embedding a 30-page document produces a blurry average that matches many queries weakly, and whatever you retrieve must also fit in the LLM's prompt. Chunking into passages makes each vector specific. Size is a trade-off: small chunks are precise but lose context; large chunks keep context but dilute relevance. Start around a few hundred tokens with some overlap, split on natural boundaries, and tune using retrieval metrics on real queries.

Q: What is cosine similarity and why is it used for text embeddings? A: It's the cosine of the angle between two vectors — 1 for the same direction, 0 for unrelated, negative for opposite. It compares direction rather than magnitude, and for text embeddings direction carries the meaning. If vectors are normalized to length 1, cosine similarity equals the dot product, which is cheaper to compute — which is why many systems normalize at write time.

Q: How would you build a pipeline that keeps embeddings up to date for a changing document store? A: Treat it as incremental ETL: detect new, changed, and deleted documents (CDC or content hashes), re-chunk only the changed ones, embed in batches with retries and rate limiting, and upsert vectors keyed by a stable chunk ID along with metadata (source ID, hash, model version, timestamps). Deletes must remove the vectors too. Record the embedding model version so that a model change triggers a full, versioned re-embed — ideally into a new index you swap in once it's ready.

Q: How do you evaluate an embedding model for your use case? A: Build a labelled set of realistic queries paired with the documents that should be retrieved, then measure retrieval metrics — recall@k, MRR, or nDCG — for each candidate model and chunking strategy. Also weigh cost per million tokens, latency, vector size (storage and search cost), and whether the model can run where your data is allowed to go.

Further Reading

- [OpenAI embeddings guide](https://developers.openai.com/api/docs/guides/embeddings) (<https://developers.openai.com/api/docs/guides/embeddings>)
- [Voyage AI documentation](https://docs.voyageai.com/) (<https://docs.voyageai.com/>) and [Anthropic's embeddings guide](https://platform.claude.com/docs/en/build-with-claude/embeddings) (<https://platform.claude.com/docs/en/build-with-claude/embeddings>)

- [Sentence Transformers](https://sbert.net/) (https://sbert.net/) — local embedding models and fine-tuning
 - [MTEB leaderboard](https://huggingface.co/spaces/mteb/leaderboard) (https://huggingface.co/spaces/mteb/leaderboard) — benchmark results across many tasks
 - [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) and [Vector Databases](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/vector-databases/) — where embeddings are used next
-

Previous: [LLM APIs](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/llm-apis/) ·

Next: [RAG](https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) (https://sarangambekar1997.github.io/data-engineering-handbook/07-ai/rag/) · **Back to:** [Index](https://sarangambekar1997.github.io/data-engineering-handbook/) (https://sarangambekar1997.github.io/data-engineering-handbook/)

 September 28, 2026