

Clase 5 – RAG Foundations

Embeddings + Ingesta + Chunking + Vector DB

Bootcamp AI Engineer · Código Facilito

Hoy construimos el pipeline completo:

doc → chunks → embeddings → vector DB → retrieve



¿Dónde estamos?

Lo que ya tenemos:

core/llm_client.py – cliente LLM con logging y trazabilidad (Clase 1)

core/tokenlab.py – control de tokens, costos y latencia (Clase 2)

prompting/promptkit.py – plantillas y chains versionables (Clase 3)

contracts/structured_output.py – outputs JSON con Pydantic (Clase 4)

Lo que construiremos hoy:

rag/embeddings.py

rag/ingestion.py

rag/vectorstore.py



Embeddings: texto como vectores

Un embedding convierte texto en un vector numérico de N dimensiones.

Textos con significado similar → vectores cercanos en el espacio.

"El gato duerme en el sofá" → [0.12, -0.34, 0.87, ...]

"Un felino descansa en el mueble" → [0.11, -0.33, 0.85, ...]

"Python es un lenguaje de programación" → [-0.45, 0.72, 0.01, ...]

Los primeros dos están «cerca». El tercero está «lejos».

Esa distancia es lo que usamos para buscar.



De archivos a texto limpio

Los documentos del mundo real vienen sucios:

PDFs con tablas rotas y headers repetidos

Markdown con frontmatter YAML

TXT con encodings extraños

HTML con basura de navegación

Nuestro trabajo: extraer texto limpio + metadatos útiles.

Basura entra → basura sale.

La calidad del RAG empieza aquí.



RAG en 60 segundos

Retrieval-Augmented Generation = darle al LLM contexto que no tiene.

Sin RAG: el modelo alucina o responde con información genérica.

Con RAG: el modelo responde con TUS documentos como fuente.

Flujo de indexación:

Documentos → Chunks → Embeddings → Vector DB

Flujo de consulta:

Query → Embedding → Búsqueda → Contexto → LLM → Respuesta

No es magia. Es un pipeline de ingeniería.



rag/embeddings.py – Parte 1

Generando embeddings con OpenAI

```
from openai import OpenAI
import numpy as np

client = OpenAI()

def get_embedding(
    text: str,
    model: str = "text-embedding-3-small"
) -> list[float]:
    """Genera el embedding de un texto."""
    text = text.replace("\n", " ").strip()
    response = client.embeddings.create(
        input=text, model=model
    )
    return response.data[0].embedding

# Probemos
emb = get_embedding("El gato duerme en el sofá")
print(f"Dimensiones: {len(emb)}")    # 1536
print(f"Primeros 5: {emb[:5]}")
```

Modelo recomendado: text-embedding-3-small (1536 dims, barato, bueno)



rag/embeddings.py — Parte 2

Similitud coseno: midiendo cercanía

```
def cosine_similarity(a: list[float], b: list[float]) -> float:  
    """Calcula similitud coseno entre dos vectores."""  
    a, b = np.array(a), np.array(b)  
    return float(np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b)))  
  
# Demo  
emb1 = get_embedding("El gato duerme en el sofá")  
emb2 = get_embedding("Un felino descansa en el mueble")  
emb3 = get_embedding("Python es un lenguaje de programación")  
  
print(f"Gato vs Felino: {cosine_similarity(emb1, emb2):.4f}") # ~0.92  
print(f"Gato vs Python: {cosine_similarity(emb1, emb3):.4f}") # ~0.45
```

Más cercano a 1.0 = más similares semánticamente.

rag/embeddings.py – Parte 3

Batch embeddings y costos

```
def get_embeddings_batch(
    texts: list[str],
    model: str = "text-embedding-3-small"
) -> list[list[float]]:
    """Genera embeddings en batch (más eficiente y barato)."""
    cleaned = [t.replace("\n", " ").strip() for t in texts]
    response = client.embeddings.create(input=cleaned, model=model)
    return [item.embedding for item in response.data]

# Costos aproximados (text-embedding-3-small):
# $0.02 por 1M tokens
# Un documento de 10 páginas ~ 3,000 tokens ~ $0.00006
# 10,000 documentos ~ $0.60
```

Siempre usar batch. Nunca un loop de llamadas individuales.



Modelos de embeddings: OpenAI vs Open Source

Modelo	Dims	Costo	Latencia	Idiomas
text-embedding-3-small	1536	\$0.02/1M tok	~100ms (API)	Multilingüe
text-embedding-3-large	3072	\$0.13/1M tok	~150ms (API)	Multilingüe
jina-embeddings-v2-base-en	768	Gratis (local)	~50ms (GPU)	Inglés
BAAI/bge-m3	1024	Gratis (local)	~80ms (GPU)	Multilingüe
all-MiniLM-L6-v2	384	Gratis (local)	~20ms (GPU)	Inglés

Embeddings Open Source (HuggingFace)

Alternativa local y gratuita

```
from transformers import AutoModel

embedding_model = AutoModel.from_pretrained(
    'jinaai/jina-embeddings-v2-base-en',
    trust_remote_code=True
)

user_query = "user query"
query_embeddings = embedding_model.encode(user_query).tolist()

# Criterio de decisión:
# Privacidad total o cero costos → local
# Multilingüe fácil, sin importar costo → OpenAI
# Producción: mide calidad de retrieval con TUS datos
```

En producción, mide calidad de retrieval con TUS datos antes de decidir.

rag/ingestion.py — Parte 1: Loaders

Document dataclass + loader de TXT

```
from pathlib import Path
from dataclasses import dataclass, field
from datetime import datetime

@dataclass
class Document:
    content: str
    metadata: dict = field(default_factory=dict)
    doc_id: str = ""

def load_txt(path: Path) -> Document:
    text = path.read_text(encoding="utf-8")
    return Document(
        content=text,
        metadata={"source": str(path), "type": "txt",
                  "loaded_at": datetime.now().isoformat()},
        doc_id=path.stem,
    )
```

Un Document = contenido + metadatos. Así de simple.



rag/ingestion.py – Parte 2: PDF

Loader de PDF con pypdf

```
from pypdf import PdfReader

def load_pdf(path: Path) -> Document:
    reader = PdfReader(str(path))
    pages = []
    for i, page in enumerate(reader.pages):
        text = page.extract_text() or ""
        pages.append(text)

    return Document(
        content="\n\n".join(pages),
        metadata={
            "source": str(path),
            "type": "pdf",
            "num_pages": len(reader.pages),
            "loaded_at": datetime.now().isoformat(),
        },
        doc_id=path.stem,
    )
```

pypdf es liviano y sin dependencias pesadas. Para tablas: pdfplumber o unstructured.



rag/ingestion.py – Parte 3: Markdown

Loader de Markdown

```
def load_markdown(path: Path) -> Document:
    text = path.read_text(encoding="utf-8")

    # Remover frontmatter YAML si existe
    if text.startswith("---"):
        parts = text.split("---", 2)
        if len(parts) >= 3:
            text = parts[2].strip()

    return Document(
        content=text,
        metadata={
            "source": str(path),
            "type": "markdown",
            "loaded_at": datetime.now().isoformat(),
        },
        doc_id=path.stem,
    )
```



rag/ingestion.py – Parte 4

Router de ingesta: orquestando loaders

```
LOADERS = {
    ".txt": load_txt,
    ".pdf": load_pdf,
    ".md": load_markdown,
}

def load_document(path: str | Path) -> Document:
    path = Path(path)
    if not path.exists():
        raise FileNotFoundError(f"No existe: {path}")
    loader = LOADERS.get(path.suffix.lower())
    if loader is None:
        raise ValueError(f"Formato no soportado: {path.suffix}")
    return loader(path)

def load_directory(dir_path: str | Path) -> list[Document]:
    dir_path = Path(dir_path)
    docs = []
    for file in sorted(dir_path.rglob("*")):
        if file.suffix.lower() in LOADERS:
            docs.append(load_document(file))
    print(f"Cargados {len(docs)} documentos desde {dir_path}")
    return docs
```



¿Por qué chunking?

Un documento de 50 páginas tiene ~15,000 tokens.

El embedding de todo el documento es un promedio difuso → pierde especificidad.

Solución: partir en chunks más pequeños.

Cada chunk captura una idea o sección concreta.

El reto: ¿dónde cortar?

Muy grande → pierde precisión en la búsqueda

Muy pequeño → pierde contexto

Sin overlap → pierde continuidad entre chunks

Sweet spot típico: 500-1000 tokens con 10-20% de overlap.



rag/ingestion.py – Chunking básico

Chunking por caracteres con overlap

```
@dataclass
class Chunk:
    content: str
    metadata: dict = field(default_factory=dict)
    chunk_id: str = ""

def chunk_text(doc: Document, chunk_size: int = 1000,
              chunk_overlap: int = 200) -> list[Chunk]:
    text = doc.content
    chunks, start, idx = [], 0, 0
    while start < len(text):
        end = start + chunk_size
        chunk_text = text[start:end]
        chunks.append(Chunk(
            content=chunk_text,
            metadata={**doc.metadata, "chunk_index": idx},
            chunk_id=f"{doc.doc_id}_chunk_{idx}",
        ))
        start += chunk_size - chunk_overlap
        idx += 1
    return chunks
```



rag/ingestion.py — Chunking mejorado

Respetando párrafos: no cortar a mitad de oración

```
def chunk_by_paragraphs(doc: Document, max_chunk_size: int = 1000,
                       separator: str = "\n\n") -> list[Chunk]:
    paragraphs = doc.content.split(separator)
    chunks, current, idx = [], "", 0

    for para in paragraphs:
        para = para.strip()
        if not para:
            continue
        if len(current) + len(para) + 2 > max_chunk_size and current:
            chunks.append(Chunk(
                content=current.strip(),
                metadata={**doc.metadata, "chunk_index": idx},
                chunk_id=f"{doc.doc_id}_chunk_{idx}",
            ))
            current, idx = "", idx + 1
        current += para + "\n\n"

    if current.strip():
        chunks.append(Chunk(
            content=current.strip(),
            metadata={**doc.metadata, "chunk_index": idx},
            chunk_id=f"{doc.doc_id}_chunk_{idx}",
        ))
return chunks
```



Metadatos: el arma secreta del RAG

Los metadatos no son opcionales. Son filtros de búsqueda.

```
chunk.metadata = {  
    "source": "manual_operaciones.pdf",  
    "type": "pdf",  
    "chunk_index": 3,  
    "num_pages": 45,  
    "loaded_at": "2025-02-25T10:30:00",  
    # Agrega lo que necesites:  
    "department": "ingeniería",  
    "version": "2.1",  
    "language": "es",  
}  
  
# Despues podrás hacer:  
# "Busca solo en documentos de ingeniería"  
# "Solo docs actualizados este mes"  
  
# Sin metadatos → solo búsqueda semántica  
# Con metadatos → semántica + filtros = precisión real
```



Vector Stores: ¿qué son?

Un vector store almacena embeddings + metadatos y permite buscar por similitud.

Opciones populares:

ChromaDB — ligero, embedded, perfecto para prototipos y producción ligera

FAISS (Facebook) — ultra rápido, sin metadatos nativos, más bajo nivel

Pinecone — managed, escala bien, cuesta dinero

Weaviate — open source, features avanzados, más complejo

Hoy usaremos ChromaDB: cero infraestructura, se instala con pip.

```
pip install chromadb
```

rag/vectorstore.py – Parte 1

ChromaDB: creando la colección

```
import chromadb
from rag.embeddings import get_embeddings_batch

def create_vectorstore(
    collection_name: str = "documents",
    persist_dir: str = "./chroma_db",
) -> chromadb.Collection:

    """Crea o conecta a una colección de ChromaDB."""
    client = chromadb.PersistentClient(path=persist_dir)
    collection = client.get_or_create_collection(
        name=collection_name,
        metadata={"hnsw:space": "cosine"},
    )
    print(f"Colección '{collection_name}': {collection.count()} docs")
    return collection
```

PersistentClient = datos sobreviven entre ejecuciones. cosine = similitud coseno.



rag/vectorstore.py — Parte 2

Indexando chunks en ChromaDB

```
def index_chunks(
    collection: chromadb.Collection,
    chunks: list,
    batch_size: int = 50,
) -> int:
    total = 0
    for i in range(0, len(chunks), batch_size):
        batch = chunks[i : i + batch_size]
        texts = [c.content for c in batch]
        ids = [c.chunk_id for c in batch]
        metadatas = [c.metadata for c in batch]

        embeddings = get_embeddings_batch(texts)

        collection.upsert(
            ids=ids,
            embeddings=embeddings,
            documents=texts,
            metadatas=metadatas,
        )
        total += len(batch)
        print(f"  Indexados {total}/{len(chunks)} chunks")
    return total
```



rag/vectorstore.py – Parte 3

Búsqueda: el momento de la verdad

```
@dataclass
class SearchResult:
    content: str
    metadata: dict
    score: float
    chunk_id: str

def search(collection, query: str, n_results: int = 5,
          where: dict | None = None) -> list[SearchResult]:
    query_embedding = get_embedding(query)
    results = collection.query(
        query_embeddings=[query_embedding],
        n_results=n_results,
        where=where,
        include=["documents", "metadatas", "distances"],
    )
    return [
        SearchResult(
            content=results["documents"][0][i],
            metadata=results["metadatas"][0][i],
            score=1 - results["distances"][0][i],
            chunk_id=results["ids"][0][i],
        )
        for i in range(len(results["ids"][0]))
    ]
```



El poder de where

Búsqueda con filtros de metadatos

```
# Buscar solo en PDFs
results = search(collection, "proceso de ensamblaje",
                 where={"type": "pdf"})

# Buscar solo en documentos de ingeniería
results = search(collection, "tolerancias de piezas",
                 where={"department": "ingeniería"})

# Filtros compuestos
results = search(collection, "procedimiento de seguridad",
                 where={"$and": [
                     {"type": "pdf"},
                     {"department": "operaciones"}]})

# Sin filtros: buscas en todo.
# Con filtros: buscas solo donde tiene sentido.
# En producción esto es crítico.
```

No quieres que un query de RH devuelva docs de finanzas.



Pipeline end-to-end

Juntando todo: de cero a RAG funcional

```
from rag.ingestion import load_directory, chunk_by_paragraphs
from rag.vectorstore import create_vectorstore, index_chunks, search

# 1. Cargar documentos
docs = load_directory("./data/empresa/")

# 2. Hacer chunking
all_chunks = []
for doc in docs:
    chunks = chunk_by_paragraphs(doc, max_chunk_size=800)
    all_chunks.extend(chunks)
print(f"Total: {len(all_chunks)} chunks de {len(docs)} documentos")

# 3. Indexar en vector store
collection = create_vectorstore("mi_empresa")
index_chunks(collection, all_chunks)

# 4. Buscar
results = search(collection, "¿Cuál es el proceso de onboarding?")
for r in results[:3]:
    print(f"[{r.score:.3f}] {r.content[:100]}...")
```

De cero a RAG funcional en ~50 líneas útiles.



RAG + LLM = respuestas con fundamento

Conectando el retrieve con el LLM

```
from core.llm_client import call_llm

def rag_query(collection, question: str) -> str:
    # Retrieve
    results = search(collection, question, n_results=3)

    # Construir contexto
    context = "\n\n---\n\n".join([
        f"[Fuente: {r.metadata['source']}]\n{r.content}"
        for r in results
    ])

    # Generate
    prompt = f"""Responde basándote ÚNICAMENTE en el contexto.
    Si no encuentras la respuesta, di "No tengo información".

    CONTEXTO:
    {context}

    PREGUNTA: {question}"""

    return call_llm(prompt)

answer = rag_query(collection, "¿Cuántos días de vacaciones tengo?")
```



Errores comunes en RAG

1. Chunks demasiado pequeños

El LLM no tiene suficiente contexto. Solución: chunks de 500-1000 chars.

2. Chunks demasiado grandes

El embedding es difuso y la búsqueda pierde precisión. No pasar de 1500 chars.

3. No usar metadatos

Todo se mezcla y no puedes filtrar. Siempre incluir source, type, etc.

4. Ignorar el prompt de generación

El LLM inventa cosas. Ser explícito: "basándote ÚNICAMENTE en el contexto".

5. No validar la ingesta

El PDF se cargó con basura. Revisar samples ANTES de indexar.



Tu repo tras Clase 5

Estructura del proyecto

```
 proyecto/
    |--- core/
    |     |--- llm_client.py      ← Clase 1
    |     |--- tokenlab.py       ← Clase 2
    |--- prompting/
    |     |--- promptkit.py      ← Clase 3
    |--- contracts/
    |     |--- structured_output.py   ← Clase 4
    |--- rag/
    |     |--- embeddings.py
    |     |--- ingestion.py
    |     |--- vectorstore.py
    |--- data/
    |     |--- empresa/
    |--- chroma_db/
```

← HOY
get_embedding, batch, cosine
loaders, chunking, Document
create, index, search

tus documentos de prueba
vector store persistido



Tarea y preview Clase 6

Tarea:

Indexar al menos 5 documentos reales (empresa, universidad o proyecto)

Probar 10 queries distintas y guardar los resultados

Identificar al menos 2 queries donde el retrieve falle

Próxima clase — RAG Avanzado:

Búsqueda híbrida (BM25 + vector)

Multi-query: reformular la pregunta para mejorar recall

Reranking: reordenar resultados con un segundo modelo

Compresión de contexto: extraer solo lo relevante antes del LLM

