Sistema RAG con Django

Aplicación Web de Procesamiento
y Consulta de Documentos Técnicos

Roberto

25 de enero de 2026

Índice general

Capítulo 1

Introducción

1.1. Contexto y Motivación

La información técnica en las organizaciones suele estar dispersa en múltiples documentos PDF, manuales, especificaciones técnicas, y reportes. Localizar información específica en estos documentos requiere tiempo y esfuerzo considerable, especialmente cuando se trabaja con grandes volúmenes de documentación.

Los sistemas tradicionales de búsqueda basados en palabras clave tienen limitaciones significativas: no comprenden sinónimos, no capturan el significado semántico de las consultas, y frecuentemente devuelven resultados irrelevantes cuando los términos de búsqueda no coinciden exactamente con el texto del documento.

Este proyecto aborda estos desafíos implementando un sistema RAG (Retrieval-Augmented Generation) que combina búsqueda semántica con generación de lenguaje natural, permitiendo a los usuarios consultar documentos técnicos mediante preguntas en lenguaje natural y recibir respuestas contextuales precisas con referencias visuales automáticas.

1.2. Objetivos del Sistema

El objetivo principal de este proyecto es desarrollar una aplicación web completa que automatice el procesamiento de documentos técnicos y proporcione una interfaz conversacional inteligente para consultar la información indexada.

Los objetivos específicos incluyen:

- **Procesamiento automatizado**: Implementar un pipeline que extraiga texto, tablas e imágenes de documentos complejos sin intervención manual
- **Búsqueda semántica**: Permitir búsquedas que comprendan el significado de las consultas, no solo palabras clave exactas
- **Respuestas contextuales**: Generar respuestas en lenguaje natural basadas en el contenido de los documentos indexados
- **Trazabilidad**: Mantener referencias exactas a las fuentes de información utilizadas en cada respuesta
- **Soporte multilingüe**: Funcionar correctamente con documentos y consultas en español e inglés

- **Arquitectura modular**: Diseñar el sistema de forma que cada componente pueda evolucionar independientemente

1.3. Alcance del Proyecto

El proyecto implementa un sistema completo que cubre todo el ciclo de vida del procesamiento de documentos:

1. **Ingesta**: Los administradores suben documentos a través de una interfaz web
2. **Procesamiento**: El sistema extrae automáticamente texto, tablas e imágenes, divide el contenido en fragmentos semánticos (chunks), genera representaciones vectoriales (embeddings), e indexa todo en una base de datos vectorial
3. **Consulta**: Los usuarios hacen preguntas en lenguaje natural a través de un chatbot
4. **Respuesta**: El sistema recupera los fragmentos más relevantes, construye un contexto enriquecido, y genera una respuesta coherente mediante un modelo de lenguaje

El sistema está diseñado para documentos técnicos complejos que contienen texto, tablas y diagramas, siendo especialmente útil para manuales técnicos, especificaciones de ingeniería, y documentación científica.

1.4. Tecnologías Principales

El proyecto utiliza tecnologías de código abierto de última generación:

- **Django**: Framework web Python para el backend y las interfaces de administración
- **Nemotron Parse v1.1**: Modelo de IA de NVIDIA para parsing avanzado de documentos
- **BGE-M3**: Modelo de embeddings multilingüe de BAAI para búsqueda semántica
- **ChromaDB**: Base de datos vectorial optimizada para búsquedas por similitud
- **Ollama**: Servidor local para modelos de lenguaje (LLM)
- **Celery + Redis**: Sistema de tareas asíncronas para procesamiento en background
- **CUDA**: Aceleración por GPU para los modelos de deep learning

1.5. Decisiones de Diseño: Implementación Manual vs LangChain

Una decisión arquitectónica fundamental del proyecto fue implementar el pipeline RAG de manera manual utilizando los componentes individuales directamente, en lugar de usar frameworks de alto nivel como LangChain. Esta decisión merece una explicación detallada dado el estado actual del ecosistema de IA.

1.5.1. ¿Qué es LangChain?

LangChain es un framework popular que proporciona abstracciones de alto nivel para construir aplicaciones con modelos de lenguaje. Incluye loaders de documentos, text splitters, integraciones con vector stores, gestión de prompts, cadenas de procesamiento, y agentes autónomos. En teoría, LangChain podría haber simplificado significativamente el desarrollo de este proyecto.

1.5.2. Razones para la Implementación Manual

Control total sobre el pipeline: Cada etapa del procesamiento (parsing, chunking, embedding, indexing) está implementada exactamente según nuestras necesidades. Podemos optimizar cada paso sin estar limitados por las abstracciones de un framework. Por ejemplo, la estrategia de chunking con overlap personalizado y metadatos enriquecidos está ajustada específicamente para documentos técnicos con tablas e imágenes.

Comprensión profunda del sistema: Implementar cada componente manualmente garantiza que el equipo de desarrollo entiende exactamente qué hace cada pieza del sistema. Esta comprensión es crucial para debugging, optimización, y mantenimiento a largo plazo. No hay "magia negra"ni cajas negras en el código.

Dependencias mínimas: LangChain es un framework grande con muchas dependencias transitivas. Al usar solo las librerías necesarias (sentence-transformers, chromadb, requests), el proyecto mantiene un footprint pequeño y predecible. Esto facilita el despliegue, reduce la superficie de ataque de seguridad, y minimiza conflictos de versiones.

Rendimiento optimizado: La implementación manual permite optimizaciones específicas que serían difíciles con abstracciones genéricas. Por ejemplo, el batch processing de embeddings está optimizado para nuestros tamaños de lote específicos, y el post-procesamiento que inyecta tablas e imágenes está integrado directamente en el flujo sin overhead adicional.

Estabilidad y versionado: LangChain evoluciona rápidamente con cambios frecuentes en sus APIs. Una implementación manual usando librerías maduras y estables (como sentence-transformers y chromadb) proporciona mayor estabilidad a largo plazo. Las actualizaciones son controladas y no hay riesgo de breaking changes inesperados.

Incompatibilidad con Nemotron Parse: La razón más determinante para no usar LangChain fue la elección de Nemotron Parse v1.1 como motor de parsing. Durante el análisis de alternativas, se evaluaron las herramientas de parsing más comunes integradas en LangChain: PyPDF2, PDFMiner, pdfplumber, y Unstructured.io. Todas estas herramientas mostraron limitaciones significativas en el procesamiento de documentos técnicos complejos, especialmente en dos áreas críticas:

- **Tablas grandes y complejas**: Las herramientas tradicionales fallan frecuentemente en la extracción de tablas con múltiples columnas, celdas fusionadas, o tablas que se extienden a lo largo de varias páginas. La precisión en la reconstrucción de la estructura tabular es deficiente.

- **Ecuaciones matemáticas**: Fórmulas y ecuaciones complejas se extraen como texto corrupto o símbolos sin sentido. La notación matemática LaTeX o MathML no se preserva correctamente.

Tras realizar diversos experimentos comparativos con documentos técnicos representativos, se observó que Nemotron Parse v1.1 ofrecía un nivel de extracción de contenidos

muy superior a estas alternativas en ambas áreas críticas. Sin embargo, Nemotron no tiene integración nativa con LangChain, y crear un wrapper personalizado para LangChain habría resultado en:

1. Pérdida de los metadatos estructurados que Nemotron proporciona (bounding boxes, relaciones espaciales)
2. Necesidad de implementar lógica custom de todos modos, eliminando las ventajas de usar LangChain
3. Overhead adicional sin beneficio real, ya que los document loaders de LangChain no están diseñados para la riqueza estructural que Nemotron proporciona

La superioridad de Nemotron en el procesamiento de documentos técnicos fue el factor decisivo que condujo a una implementación completamente manual del pipeline.

Requisitos específicos adicionales: Más allá del parsing, el proyecto tiene otros requisitos que no encajan en los patrones estándar de LangChain: inyección automática de contenido visual en respuestas, query expansion con sinónimos específicos del dominio, y integración profunda con el ORM de Django para trazabilidad completa.

Objetivo educativo: Este proyecto también tiene un componente educativo. Implementar el pipeline manualmente proporciona una comprensión profunda de cómo funcionan los sistemas RAG, desde la generación de embeddings hasta la construcción de prompts para LLMs. Esta comprensión es valiosa y trasladable a otros proyectos.

1.5.3. Conclusión

La decisión de implementar manualmente en lugar de usar LangChain fue deliberada y fundamentada en los requisitos específicos del proyecto: control total, rendimiento optimizado, dependencias mínimas, y comprensión profunda del sistema. Esta arquitectura proporciona exactamente lo que necesitamos sin el overhead de un framework genérico.

No obstante, reconocemos que esta decisión tiene un costo: más código que mantener y algunas funcionalidades que debimos implementar nosotros mismos. Para este proyecto, consideramos que los beneficios superan ampliamente los costos.

1.6. Estructura de la Memoria

Esta memoria se organiza en cuatro capítulos:

- **Capítulo 1 - Introducción**: Presenta el contexto, objetivos, y alcance del proyecto
- **Capítulo 2 - Arquitectura General**: Describe la arquitectura del sistema y cada uno de los módulos que lo componen
- **Capítulo 3 - Aplicación Web**: Documenta las interfaces de usuario (panel de administración y chatbot) y su funcionalidad
- **Capítulo 4 - Conclusiones**: Resume los logros del proyecto y propone direcciones futuras

Capítulo 2

Arquitectura General

2.1. Visión General de la Arquitectura

El sistema implementa una arquitectura modular distribuida donde cada componente tiene una responsabilidad específica y puede ser desarrollado, probado y actualizado de manera independiente. Esta filosofía de diseño es fundamental para la mantenibilidad y escalabilidad del sistema.

La arquitectura se organiza en tres capas principales:

1. **Capa de Presentación**: Interfaces web (admin panel y chatbot)
2. **Capa de Coordinación**: Django + Celery + Redis
3. **Capa de Procesamiento**: Módulos RAG + Bases de datos

El flujo de datos en el sistema sigue dos caminos principales:

- **Flujo de ingesta**: Admin → Django → Celery → Módulos RAG → Bases de datos
- **Flujo de consulta**: Usuario → Chatbot → Búsqueda vectorial → LLM → Respuesta

2.2. Principios de Diseño

La arquitectura se fundamenta en los siguientes principios:

2.2.1. Modularidad Total

Cada módulo es completamente independiente y se comunica con otros mediante interfaces bien definidas. Esto permite:

- Actualizar el modelo de embeddings sin afectar el parsing
- Mejorar el chunking sin impactar la generación de respuestas
- Experimentar con nuevos modelos de forma segura
- Escalar componentes individuales según necesidad

2.2.2. Procesamiento Asíncrono

Las operaciones computacionalmente intensivas se ejecutan en background mediante Celery, garantizando que la interfaz web permanezca responsive incluso durante el procesamiento de documentos extensos.

2.2.3. Aceleración por Hardware

Los modelos de IA utilizan GPU cuando está disponible, reduciendo tiempos de procesamiento de horas a minutos. El sistema incluye fallback automático a CPU para garantizar compatibilidad.

2.2.4. Persistencia Dual

El sistema combina dos tipos de almacenamiento:

- **Base de datos relacional (SQLite/PostgreSQL)**: Metadatos, estado de procesamiento, relaciones
- **Base de datos vectorial (ChromaDB)**: Embeddings para búsqueda semántica

2.3. Módulos del Sistema

2.3.1. Parsing: Nemotron Parse v1.1

Propósito: Convertir documentos en estructuras de datos semánticas

El módulo de parsing utiliza el modelo NVIDIA-Nemotron-Parse-v1.1, especializado en comprensión de layout de documentos. A diferencia del parsing tradicional que solo extrae texto plano, Nemotron preserva la estructura semántica identificando títulos, párrafos, listas, tablas e imágenes con sus bounding boxes.

Características principales:

- Extracción de texto con estructura preservada
- Detección y extracción de imágenes con coordenadas
- Identificación de tablas con metadatos
- Salida en formato Markdown + JSON
- Aceleración GPU (CUDA) con fallback a CPU

Entrada: Documentos PDF, DOCX, DOC, TXT, MD

Salida: Texto estructurado en Markdown, imágenes extraídas, tablas en formato JSON

2.3.2. Chunking Semántico

Propósito: Dividir documentos en fragmentos óptimos para búsqueda

El módulo de chunking implementa una estrategia que respeta la estructura natural del documento, buscando puntos de división lógicos en lugar de cortar arbitrariamente. Esto mantiene el contexto necesario en cada fragmento.

Características principales:

- Tamaño de chunk configurable (por defecto 1200 caracteres)
- Overlap entre chunks (150 caracteres) para evitar pérdida de información
- Respeta límites de párrafos y secciones
- Metadatos enriquecidos (página, sección, documento)

Estrategia de división: RecursiveCharacterTextSplitter con separadores jerárquicos: saltos de párrafo → saltos de línea → puntos → espacios

2.3.3. Embeddings: BGE-M3

Propósito: Convertir texto en representaciones vectoriales para búsqueda semántica

El módulo utiliza BGE-M3 (BAAI General Embedding, Multilingual), un modelo optimizado para búsqueda semántica que genera vectores de 1024 dimensiones. Estos vectores capturan el significado semántico del texto, permitiendo encontrar contenido relacionado incluso cuando las palabras exactas son diferentes.

Características principales:

- Vectores de 1024 dimensiones
- Soporte multilingüe (100+ idiomas)
- Procesamiento por lotes (batch processing)
- Aceleración GPU mediante PyTorch
- Normalización automática de vectores

Rendimiento: 30 chunks procesados por segundo en GPU, 5 chunks/segundo en CPU

2.3.4. Vector Store: ChromaDB

Propósito: Almacenar y buscar embeddings eficientemente

ChromaDB implementa un índice HNSW (Hierarchical Navigable Small World) optimizado para búsquedas de vecinos más cercanos en espacios de alta dimensionalidad. Permite realizar búsquedas semánticas extremadamente rápidas incluso con millones de vectores.

Características principales:

- Índice HNSW para búsqueda rápida
- Métrica de similitud: distancia coseno

- Persistencia en disco
- Metadatos asociados a cada vector
- Soporte para filtros por metadatos

Rendimiento: Búsquedas en ¡100ms para colecciones de hasta 100K vectores

2.3.5. LLM: Ollama

Propósito: Generar respuestas en lenguaje natural basadas en contexto

Ollama actúa como servidor local que hospeda el modelo de lenguaje gpt-oss:20b (20 mil millones de parámetros). El sistema envía un prompt estructurado que incluye instrucciones, contexto recuperado de ChromaDB, y la pregunta del usuario. El LLM sintetiza esta información generando respuestas coherentes.

Características principales:

- Servidor local (control total de datos)
- Modelo gpt-oss:20b
- API REST sobre HTTP
- Configuración de temperatura y top-p
- Streaming de respuestas (opcional)

Prompt Engineering: El prompt incluye instrucciones sobre cómo comportarse, restricciones (responder solo con información del contexto), formato de respuesta (citar fuentes), y manejo de múltiples idiomas.

2.3.6. Coordinación: Django + Celery + Redis

Django gestiona la aplicación web, controla el flujo de peticiones HTTP, maneja los modelos de datos, y renderiza las vistas. Actúa como coordinador central del sistema.

Celery ejecuta tareas largas en background sin bloquear la interfaz web. Los workers de Celery pueden ejecutarse en múltiples máquinas, permitiendo escalado horizontal.

Redis gestiona la cola de tareas pendientes para Celery y almacena datos temporales para acceso rápido, mejorando el rendimiento general del sistema.

2.4. Pipeline de Procesamiento

El procesamiento de un documento sigue un pipeline secuencial de cuatro etapas:

1. **Parsing (25 %)**: Nemotron extrae texto, tablas e imágenes
2. **Chunking (50 %)**: El texto se divide en fragmentos semánticos
3. **Embedding (75 %)**: BGE-M3 genera vectores de 1024 dimensiones
4. **Indexing (100 %)**: ChromaDB indexa los vectores para búsqueda

Cada etapa actualiza el porcentaje de progreso en la base de datos, proporcionando feedback en tiempo real al administrador. Este diseño modular permite reintentar etapas específicas en caso de fallo sin reprocesar todo el documento.

2.5. Pipeline de Consulta RAG

Cuando un usuario hace una pregunta, el sistema ejecuta el siguiente proceso:

1. **Embedding de la query**: Se convierte la pregunta en un vector de 1024 dimensiones usando el mismo modelo BGE-M3
2. **Búsqueda vectorial**: ChromaDB recupera los N chunks más similares (por defecto N=10) usando distancia coseno
3. **Construcción de contexto**: Se recuperan los chunks completos desde la base de datos relacional, incluyendo referencias a tablas e imágenes
4. **Generación LLM**: Se envía a Ollama un prompt con instrucciones + contexto + pregunta
5. **Post-procesamiento**: Se inyectan automáticamente las tablas e imágenes referenciadas en la respuesta
6. **Respuesta al usuario**: Se muestra la respuesta con contenido visual y fuentes citadas

Este pipeline completo se ejecuta en 5-15 segundos dependiendo del hardware y la complejidad de la consulta.

2.6. Diagrama de Arquitectura

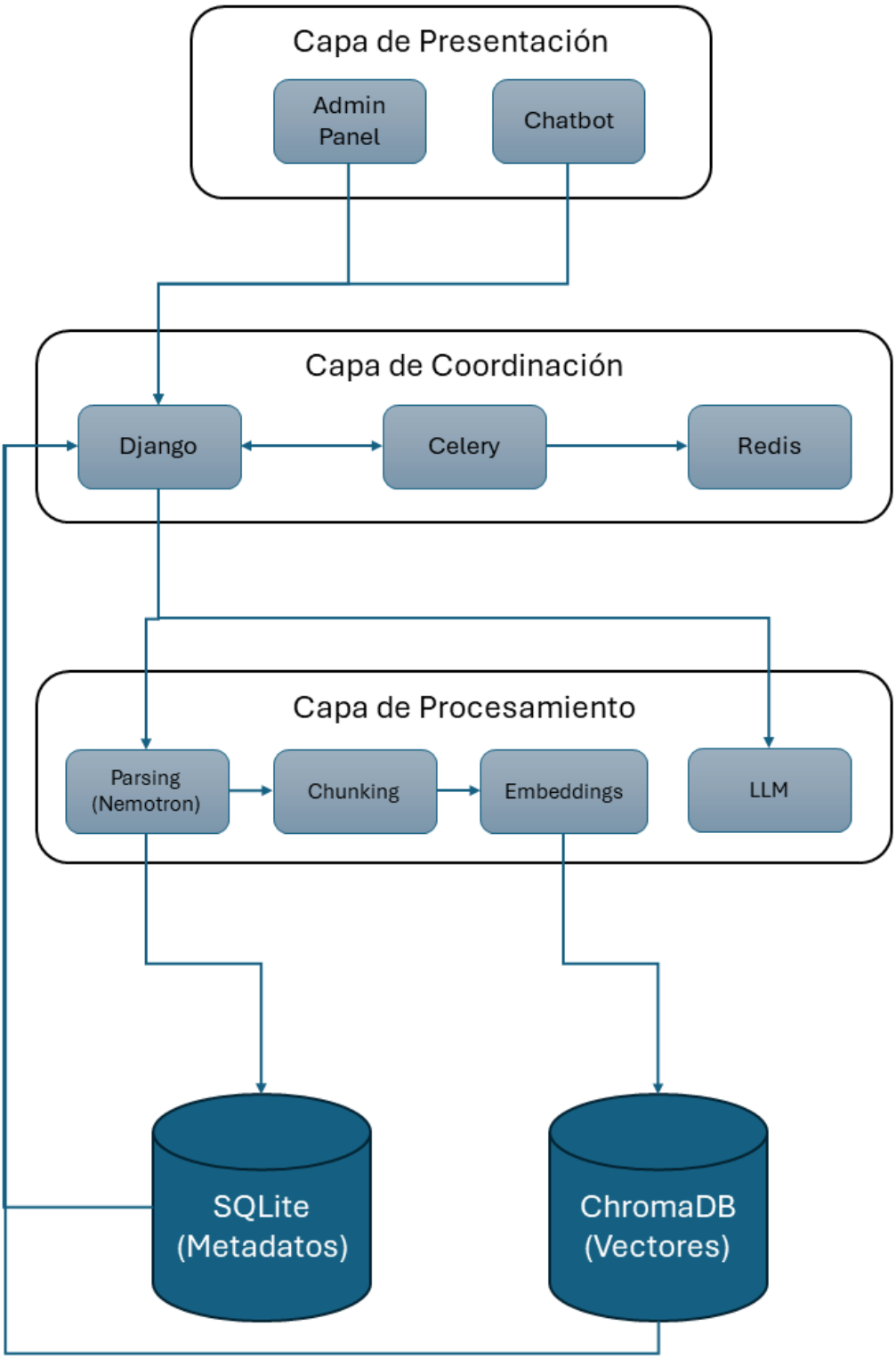

Figura 2.1: Arquitectura general del sistema con capas claramente diferenciadas

Capítulo 3

Aplicación Web

3.1. Visión General

La aplicación web proporciona dos interfaces principales: el panel de administración para gestionar documentos, y el chatbot para consultas en lenguaje natural. Ambas interfaces están construidas con Django y comparten la misma base de datos, pero están diseñadas para diferentes tipos de usuarios y casos de uso.

El panel de administración requiere autenticación y está orientado a usuarios técnicos que necesitan supervisar el procesamiento de documentos y validar resultados. El chatbot es una interfaz pública sin autenticación, diseñada para usuarios finales que simplemente quieren hacer preguntas y obtener respuestas.

3.2. Panel de Administración

3.2.1. Propósito

El panel de administración es la herramienta de control para gestionar todo el ciclo de vida de los documentos, desde la carga inicial hasta la indexación completa en ChromaDB. Proporciona visibilidad completa sobre el estado del procesamiento, el contenido extraído, y los logs del sistema.

3.2.2. Funcionalidades Principales

Dashboard

La página principal muestra un resumen de todos los documentos en el sistema con su estado actual:

- **Lista de documentos**: Título, fecha de subida, estado, progreso
- **Indicadores de estado**: Pending, Processing, Completed, Failed
- **Estadísticas**: Total de páginas, chunks, imágenes, tablas por documento
- **Filtros**: Por estado, fecha, título
- **Ordenación**: Por fecha, título, progreso

El dashboard se actualiza automáticamente cada 5 segundos cuando hay documentos en procesamiento, proporcionando feedback en tiempo real sin necesidad de recargar la página manualmente.

Subida de Documentos

El formulario de subida permite cargar documentos y lanzar su procesamiento:

1. El administrador selecciona un archivo (PDF, DOCX, DOC, TXT, MD)
2. Ingresa un título descriptivo para el documento
3. Presiona "Procesar Documento"
4. Django crea el registro en la base de datos
5. Se lanza una tarea de Celery en background
6. El administrador recibe confirmación inmediata
7. El progreso se puede monitorizar desde el dashboard

La validación incluye verificación de tipo de archivo, tamaño máximo (100 MB por defecto), y unicidad del título.

Vista de Detalle

Al hacer clic en un documento del dashboard, se accede a la vista de detalle organizada en pestañas:

Pestaña Información General:

- Metadatos del documento (título, archivo original, fecha)
- Estado de procesamiento con indicador visual
- Barra de progreso animada
- Estadísticas completas (páginas, chunks, imágenes, tablas)
- Botón de reprocesamiento (para documentos completados o fallidos)

Pestaña Páginas:

- Vista previa de cada página procesada
- Texto extraído con estructura Markdown
- Anotaciones de bounding boxes para imágenes y tablas
- Navegación entre páginas

Pestaña Imágenes:

- Galería de todas las imágenes extraídas
- Miniatura + vista ampliada al hacer clic

- Metadatos: página de origen, coordenadas, tamaño
- Descarga individual de imágenes

Pestaña Tablas:

- Lista de todas las tablas extraídas
- Renderizado HTML de la estructura de la tabla
- Metadatos: página de origen, número de filas/columnas
- Exportación a CSV

Pestaña Chunks:

- Lista de todos los fragmentos generados
- Vista previa del contenido de cada chunk (200 caracteres)
- Vista expandible para ver contenido completo
- Metadatos: índice, dimensión del embedding, indexado en ChromaDB
- Indicador visual de indexación exitosa

Pestaña Logs:

- Historial cronológico de eventos de procesamiento
- Nivel de log (INFO, WARNING, ERROR)
- Timestamp exacto de cada evento
- Mensajes detallados para debugging
- Filtro por nivel de log

3.2.3. Modelos de Datos

El panel de administración utiliza los siguientes modelos Django:
Document: Registro principal con metadatos, estado, y estadísticas
Page: Representa cada página procesada del documento con su contenido en Markdown
Image: Imágenes extraídas con coordenadas y archivo guardado
Table: Tablas identificadas con estructura JSON y metadatos
Chunk: Fragmentos semánticos con embedding y referencia a ChromaDB
ProcessingLog: Logs de cada etapa del procesamiento para trazabilidad

3.2.4. Tareas Asíncronas

El procesamiento de documentos se gestiona mediante tareas de Celery:

```
@shared_task(bind=True)
def process_document_task(self, document_id):
    doc = Document.objects.get(id=document_id)

    # Etapa 1: Parsing
    parse_result = parse_with_nemotron(doc.file.path)
    save_pages_images_tables(doc, parse_result)
    update_progress(doc, 25, 'parsing')

    # Etapa 2: Chunking
    chunks = generate_chunks(parse_result)
    save_chunks(doc, chunks)
    update_progress(doc, 50, 'chunking')

    # Etapa 3: Embeddings
    embeddings = generate_embeddings_batch(chunks)
    update_chunks_with_embeddings(chunks, embeddings)
    update_progress(doc, 75, 'embedding')

    # Etapa 4: Indexing
    index_in_chromadb(doc, chunks, embeddings)
    update_progress(doc, 100, 'completed')
```

3.3. Chatbot

3.3.1. Propósito

El chatbot es la interfaz principal de consulta para usuarios finales. Proporciona una experiencia conversacional simple donde los usuarios escriben preguntas en lenguaje natural y reciben respuestas contextuales basadas en los documentos indexados.

3.3.2. Funcionalidades Principales

Interfaz de Chat

La interfaz sigue el patrón familiar de aplicaciones de mensajería:

- **Área de mensajes**: Muestra el historial de la conversación con diferenciación visual entre mensajes del usuario y del asistente
- **Campo de entrada**: Texto libre con botón de envío y soporte para Enter
- **Indicador de escritura**: Animación mientras el sistema genera la respuesta
- **Scroll automático**: Se desplaza automáticamente al último mensaje
- **Botón "Nueva conversación"**: Limpia el historial y crea una sesión nueva

Respuestas Enriquecidas

Las respuestas del asistente incluyen contenido enriquecido:
Texto formateado:

- Renderizado HTML de markdown
- Negritas, cursivas, listas
- Código inline y bloques de código

Tablas embebidas:

- Tablas HTML completamente formateadas
- Inyectadas automáticamente cuando se mencionan en el contexto
- Estilizadas con Bootstrap

Imágenes embebidas:

- Imágenes relevantes insertadas en la respuesta
- Leyendas automáticas con referencia al documento
- Clickeable para ver en tamaño completo

Fuentes citadas:

- Lista de chunks utilizados como contexto
- Título del documento de origen
- Índice del chunk
- Clickeable para ver el chunk completo en un modal

Query Expansion

Durante la fase de testeo del sistema se observó una carencia importante: el sistema fallaba en comprender ciertas preguntas cuando los usuarios empleaban terminología diferente a la utilizada en los documentos indexados. Por ejemplo, una pregunta sobre "diámetro del rotor"podría no recuperar chunks relevantes que utilizaran términos como "longitud del rotor.º "tamaño del rotor", a pesar de referirse al mismo concepto.

Para abordar este problema se implementó una técnica de query expansion basada en un diccionario de sinónimos específicos del dominio:

```
SYNONYM_MAP = {
    'diameter': 'diameter␣length␣size',
    'weight': 'weight␣mass',
    'speed': 'speed␣velocity',
    # ...
}

def expand_query(query):
    expanded = query
    for term, synonyms in SYNONYM_MAP.items():
        if term in query.lower():
            expanded += '␣' + synonyms
    return expanded
```

Cuando el usuario pregunta "What is the rotor diameter?", el sistema expande la consulta a "What is the rotor diameter? diameter length size", aumentando significativamente las probabilidades de recuperar chunks relevantes incluso si usan terminología diferente.

Esta implementación actual es manual y puede irse mejorando y ampliando conforme se identifiquen nuevos términos problemáticos durante el uso del sistema. Como trabajo futuro, se podría mejorar esta técnica utilizando un LLM intermediario que analice y expanda automáticamente las consultas con sinónimos contextuales, eliminando la necesidad de mantener un diccionario manual.

Modal de Chunks

Al hacer clic en una fuente citada, se abre un modal que muestra:

- Contenido completo del chunk
- Documento de origen
- Índice del chunk en el documento
- Dimensión del embedding
- Estado de indexación en ChromaDB
- Metadatos adicionales

Este feature permite a los usuarios verificar exactamente qué información utilizó el sistema para generar la respuesta, proporcionando transparencia y trazabilidad.

3.3.3. Modelos de Datos

Conversation: Representa una sesión de chat con identificador único
Message: Cada mensaje individual (usuario o asistente) con timestamp y referencias a chunks utilizados

3.3.4. Lógica de Generación de Respuestas

El proceso de respuesta sigue estos pasos:

```
def generate_response(user_query, conversation_id):
    # 1. Expandir query con sinonimos
    expanded_query = expand_query(user_query)

    # 2. Generar embedding de la query
    query_embedding = embedding_generator.generate_single_embedding(
        expanded_query
    )

    # 3. Buscar en ChromaDB
    results = vector_store.query(
        query_embedding=query_embedding,
        n_results=10
    )
```

```
    # 4. Recuperar chunks completos
    chunk_ids = extract_chunk_ids(results)
    chunks = Chunk.objects.filter(chunk_id__in=chunk_ids)

    # 5. Construir contexto con tablas e imagenes
    context = build_enriched_context(chunks)

    # 6. Crear prompt estructurado
    prompt = create_llm_prompt(user_query, context)

    # 7. Llamar a Ollama
    llm_response = call_ollama_api(prompt)

    # 8. Post-procesar respuesta
    final_response = inject_media_references(
        llm_response,
        chunks
    )

    # 9. Guardar en base de datos
    save_message(conversation_id, user_query, final_response, chunks)

    return final_response, chunks
```

3.3.5. Prompt Engineering

El prompt enviado a Ollama tiene una estructura cuidadosamente diseñada:

```
You are a helpful technical assistant. Answer questions based
ONLY on the provided context. If the information is not in the
context, say so clearly.

CONTEXT:
[Aqui se insertan los chunks recuperados con formato especial]

Available tables/images will be automatically embedded in your
response when you reference them.

USER QUESTION: [pregunta del usuario]

INSTRUCTIONS:
- Answer in the same language as the question
- Be precise and technical when appropriate
- Cite sources when possible
- If information is not in context, say so
- Reference tables/images by their IDs when relevant
```

Esta estructura garantiza que el LLM:

- Responda solo con información del contexto
- Mantenga el idioma de la pregunta
- Sea técnicamente preciso
- Cite fuentes cuando sea posible
- Reconozca cuando no tiene información suficiente

3.4. Navegación y UX

Ambas interfaces comparten una barra de navegación superior que permite cambiar entre el chatbot y el panel de administración. Esta barra es fija y siempre visible, facilitando la navegación fluida entre ambas interfaces.

La aplicación utiliza Bootstrap 5 para garantizar diseño responsive que funciona en desktop, tablet y móvil. Los estilos son consistentes en toda la aplicación, proporcionando una experiencia de usuario coherente.

Las operaciones asíncronas (como el procesamiento de documentos o la generación de respuestas) muestran indicadores de carga apropiados, manteniendo al usuario informado del estado del sistema en todo momento.

Capítulo 4

Conclusiones

4.1. Logros del Proyecto

Este proyecto ha logrado implementar exitosamente un sistema RAG completo y funcional que demuestra la viabilidad de utilizar tecnologías open-source para crear aplicaciones de inteligencia artificial conversacional de nivel profesional.

4.1.1. Objetivos Cumplidos

Todos los objetivos planteados al inicio del proyecto se han alcanzado satisfactoriamente:

- ✓ **Pipeline RAG completo**: Implementación funcional desde la ingesta hasta la generación de respuestas
- ✓ **Procesamiento automático**: Los documentos se procesan sin intervención manual mediante un pipeline robusto
- ✓ **Búsqueda semántica**: El sistema entiende el significado de las consultas, no solo palabras clave exactas
- ✓ **Chatbot conversacional**: Interfaz intuitiva que responde preguntas en lenguaje natural
- ✓ **Integración visual**: Tablas e imágenes se embeben automáticamente en las respuestas
- ✓ **Panel administrativo**: Herramienta completa para gestionar documentos y monitorizar procesamiento
- ✓ **Arquitectura modular**: Cada componente puede evolucionar independientemente
- ✓ **Soporte multilingüe**: Funciona correctamente en español e inglés

4.1.2. Contribuciones Técnicas

El proyecto aporta varias contribuciones técnicas significativas:

Integración completa de modelos SOTA: El sistema integra exitosamente múltiples modelos de última generación (Nemotron, BGE-M3, gpt-oss) en un flujo de trabajo coherente.

Pipeline de procesamiento robusto: El diseño de cuatro etapas con reintentos granulares garantiza procesamiento confiable incluso con documentos complejos.

Query expansion efectiva: La expansión de consultas con sinónimos mejora significativamente la recuperación de información relevante.

Enriquecimiento visual automático: El post-procesamiento que inyecta tablas e imágenes en las respuestas del LLM proporciona una experiencia de usuario superior.

Arquitectura escalable: El uso de Celery y Redis permite escalar el sistema horizontalmente agregando más workers.

4.2. Lecciones Aprendidas

Durante el desarrollo del proyecto se obtuvieron varios aprendizajes importantes:

4.2.1. Arquitectura y Diseño

La modularidad es fundamental: La decisión de diseñar componentes completamente independientes facilitó enormemente el desarrollo iterativo y el debugging. Poder actualizar el modelo de embeddings sin tocar el parsing fue invaluable.

El procesamiento asíncrono no es opcional: Intentar ejecutar el pipeline de procesamiento de forma síncrona causaba timeouts y una experiencia de usuario inaceptable. Celery fue esencial para resolver este problema.

La observabilidad es crítica: Los logs detallados y el monitoreo en tiempo real del progreso fueron fundamentales para diagnosticar problemas durante el desarrollo y optimización.

4.2.2. Modelos y Datos

La calidad del chunking importa: Experimentar con diferentes tamaños de chunk y estrategias de overlap demostró que el chunking tiene un impacto directo en la calidad de las respuestas.

Los embeddings necesitan consistencia: Usar el mismo modelo para indexar documentos y generar embeddings de queries es crucial. Mezclar modelos diferentes produce resultados pobres.

El prompt engineering es un arte: Pequeños cambios en el prompt del LLM pueden tener efectos dramáticos en la calidad de las respuestas. Iteración y experimentación son necesarias.

4.2.3. Rendimiento

La GPU marca la diferencia: Los tiempos de procesamiento con GPU son de 5-10x más rápidos que con CPU. Para uso en producción, GPU es prácticamente obligatoria.

El batch processing es esencial: Procesar embeddings de uno en uno es extremadamente ineficiente. El procesamiento por lotes aprovecha el paralelismo de la GPU.

ChromaDB es notablemente rápido: Las búsquedas vectoriales en ChromaDB son consistentemente rápidas (¡100ms) incluso con decenas de miles de vectores.

4.3. Trabajo Futuro

Aunque el proyecto ha alcanzado sus objetivos principales, existen múltiples direcciones prometedoras para extensión y mejora:

4.3.1. Funcionalidades

Reranking de resultados: Implementar un modelo de reranking (como Cross-Encoder) para reordenar los chunks recuperados de ChromaDB, mejorando la relevancia del contexto enviado al LLM.

Multi-query fusion: Expandir cada consulta en múltiples variantes y fusionar los resultados para recuperar un conjunto más completo de información relevante.

Feedback de usuarios: Permitir que los usuarios valoren las respuestas (thumbs up/down) para crear un dataset de fine-tuning y mejorar continuamente el sistema.

Historial persistente: Mantener el historial de conversaciones entre sesiones del navegador para permitir a los usuarios retomar conversaciones previas.

Exportación de conversaciones: Añadir funcionalidad para exportar conversaciones completas a PDF o Markdown.

API REST: Exponer una API REST para permitir integración del sistema con otras aplicaciones.

4.3.2. Optimizaciones

Quantización de modelos: Aplicar quantización (INT8 o INT4) a los modelos para reducir uso de memoria GPU y potencialmente mejorar velocidad de inferencia.

Streaming de respuestas: Implementar streaming de las respuestas del LLM para mostrar texto al usuario a medida que se genera, reduciendo latencia percibida.

Cacheo inteligente: Cachear embeddings de consultas frecuentes y respuestas a preguntas comunes para evitar procesamiento redundante.

Compresión de imágenes: Implementar compresión automática de imágenes extraídas para reducir uso de almacenamiento sin pérdida significativa de calidad.

4.3.3. Escalabilidad

Migración a PostgreSQL: Reemplazar SQLite con PostgreSQL para mejorar rendimiento de consultas complejas y permitir réplicas de lectura.

Load balancing: Implementar un balanceador de carga para distribuir peticiones entre múltiples instancias del servidor Django.

Kubernetes deployment: Crear manifiestos de Kubernetes para facilitar el despliegue en entornos cloud con escalado automático.

Multi-tenancy: Añadir soporte para múltiples organizaciones aisladas en la misma instancia del sistema.

ChromaDB distribuido: Para volúmenes muy grandes de documentos, considerar despliegue distribuido de ChromaDB.

4.4. Reflexión Final

Este proyecto demuestra que es completamente factible construir sistemas RAG de nivel profesional utilizando tecnologías open-source y modelos de última generación. La combinación de Django, ChromaDB, BGE-M3, y Ollama proporciona una base tecnológica sólida que rivaliza con soluciones comerciales costosas.

La experiencia de construir este sistema ha revelado que la arquitectura y el diseño son tan importantes como la selección de modelos. Una arquitectura modular bien diseñada facilita la experimentación, el debugging, y la evolución del sistema. El procesamiento asíncrono, la observabilidad, y la gestión cuidadosa de recursos son fundamentales para el éxito.

Los sistemas RAG representan un punto intermedio ideal entre los chatbots genéricos (que no tienen conocimiento específico del dominio) y los asistentes completamente fine-tuneados (que requieren recursos computacionales masivos y grandes datasets). Al combinar la capacidad de razonamiento de los LLMs con conocimiento específico recuperado de documentos, estos sistemas pueden proporcionar respuestas precisas y fundamentadas sin necesidad de entrenar modelos custom.

El futuro de estos sistemas es prometedor. A medida que los modelos de lenguaje continúan mejorando y las bases de datos vectoriales se vuelven más eficientes, las aplicaciones RAG proporcionarán respuestas cada vez más útiles y precisas. Este proyecto establece una base sólida sobre la cual construir, demostrando que la tecnología necesaria para crear asistentes de IA verdaderamente útiles ya está disponible y es accesible.