

# **Sistema RAG con Django**

Aplicación Web de Procesamiento  
y Consulta de Documentos Técnicos

Roberto

4 de enero de 2026

# Índice general

<b>1. Introducción</b>	<b>4</b>
1.1. Resumen Ejecutivo . . . . .	4
1.1.1. Características Principales . . . . .	4
1.2. Objetivos del Sistema . . . . .	4
1.3. Stack Tecnológico . . . . .	5
<b>2. Arquitectura del Sistema</b>	<b>6</b>
2.1. Visión General . . . . .	6
2.2. Flujo de Procesamiento de Documentos . . . . .	6
2.3. Flujo de Consulta (Chatbot) . . . . .	6
<b>3. Módulos del Sistema</b>	<b>10</b>
3.1. Admin Panel . . . . .	10
3.1.1. Funcionalidad . . . . .	10
3.1.2. Modelos de Datos . . . . .	10
3.1.3. Tareas Celery . . . . .	11
3.2. Chatbot . . . . .	11
3.2.1. Funcionalidad . . . . .	11
3.2.2. Modelos de Datos . . . . .	12
3.2.3. Lógica de Respuesta . . . . .	12
3.3. Parsing (Nemotron) . . . . .	13
3.3.1. Características . . . . .	13
3.3.2. Configuración . . . . .	13
3.4. Chunking Semántico . . . . .	13
3.4.1. Estrategia . . . . .	13
3.4.2. Implementación . . . . .	13
3.5. Embeddings (BGE-M3) . . . . .	14
3.5.1. Modelo . . . . .	14
3.5.2. Implementación . . . . .	14
3.6. Vector Store (ChromaDB) . . . . .	15
3.6.1. Configuración . . . . .	15
3.7. LLM (Ollama) . . . . .	15
3.7.1. Integración . . . . .	15
3.7.2. Prompt Engineering . . . . .	15
<b>4. Despliegue y Configuración</b>	<b>17</b>
4.1. Requisitos del Sistema . . . . .	17
4.1.1. Hardware . . . . .	17
4.1.2. Software . . . . .	17

4.2. Instalación . . . . .	17
4.2.1. 1. Clonar Repositorio . . . . .	17
4.2.2. 2. Crear Entorno Virtual . . . . .	17
4.2.3. 3. Instalar Dependencias . . . . .	18
4.2.4. 4. Configurar Base de Datos . . . . .	18
4.2.5. 5. Iniciar Redis . . . . .	18
4.2.6. 6. Iniciar Celery Worker . . . . .	18
4.2.7. 7. Iniciar Ollama . . . . .	18
4.2.8. 8. Iniciar Django . . . . .	18
4.3. Configuración . . . . .	18
4.3.1. Settings de Django . . . . .	18
<b>5. Uso del Sistema</b>	<b>20</b>
5.1. Panel de Administración . . . . .	20
5.1.1. Acceso . . . . .	20
5.1.2. Subir Documento . . . . .	20
5.1.3. Explorar Contenido . . . . .	20
5.2. Chatbot . . . . .	20
5.2.1. Acceso . . . . .	20
5.2.2. Realizar Consultas . . . . .	21
5.2.3. Nueva Conversación . . . . .	21
<b>6. Diagramas de Secuencia</b>	<b>22</b>
6.1. Procesamiento de Documento . . . . .	22
6.2. Consulta en Chatbot . . . . .	22
<b>7. Rendimiento y Optimización</b>	<b>24</b>
7.1. Métricas de Rendimiento . . . . .	24
7.2. Optimizaciones Implementadas . . . . .	24
7.3. Recomendaciones . . . . .	24
<b>8. Resolución de Problemas</b>	<b>26</b>
8.1. Problemas Comunes . . . . .	26
8.1.1. ChromaDB no encuentra documentos . . . . .	26
8.1.2. Celery task fails . . . . .	26
8.1.3. Ollama timeout . . . . .	26
8.1.4. Out of GPU memory . . . . .	27
<b>9. Conclusiones y Trabajo Futuro</b>	<b>28</b>
9.1. Logros del Proyecto . . . . .	28
9.2. Trabajo Futuro . . . . .	28
9.2.1. Funcionalidades . . . . .	28
9.2.2. Optimizaciones . . . . .	29
9.2.3. Escalabilidad . . . . .	29
9.3. Reflexión Final . . . . .	29

<b>A. Apéndices</b>	<b>30</b>
A.1. Comandos Útiles . . . . .	30
A.1.1. Django . . . . .	30
A.1.2. Celery . . . . .	30
A.1.3. Redis . . . . .	30
A.2. Estructura de Directorios . . . . .	31
A.3. Referencias . . . . .	31

# Capítulo 1

## Introducción

### 1.1. Resumen Ejecutivo

Este proyecto implementa un sistema completo de **Retrieval-Augmented Generation (RAG)** utilizando Django como framework web. El sistema permite procesar documentos técnicos complejos, extraer información estructurada, y proporcionar una interfaz de chatbot inteligente para consultar la información mediante procesamiento de lenguaje natural.

#### 1.1.1. Características Principales

- **Procesamiento de documentos:** Parsing automático con Nemotron Parse v1.1
- **Extracción inteligente:** Imágenes, tablas, y texto estructurado
- **Chunking semántico:** División contextual del contenido
- **Embeddings de alto rendimiento:** BGE-M3 con GPU (CUDA)
- **Búsqueda vectorial:** ChromaDB para retrieval eficiente
- **Chatbot conversacional:** Respuestas basadas en contexto con Ollama
- **Procesamiento asíncrono:** Celery + Redis para tareas en background
- **Interfaz administrativa:** Panel completo de gestión

### 1.2. Objetivos del Sistema

1. Automatizar el procesamiento de documentos técnicos complejos
2. Proporcionar acceso rápido a información mediante búsqueda semántica
3. Generar respuestas contextuales en lenguaje natural
4. Mantener trazabilidad de fuentes y referencias
5. Soportar múltiples idiomas de forma transparente

### 1.3. Stack Tecnológico

Componente	Tecnología	Versión
Framework Web	Django	5.0
Base de Datos	SQLite	3.x
Task Queue	Celery	5.x
Message Broker	Redis	Latest
Parsing	Nemotron Parse	v1.1
Embeddings	BGE-M3	
Vector DB	ChromaDB	0.5.23
LLM	Ollama	gpt-oss:20b
GPU Support	CUDA	PyTorch
Frontend	Bootstrap	5.3

Cuadro 1.1: Stack tecnológico del proyecto

# **Capítulo 2**

## **Arquitectura del Sistema**

### **2.1. Visión General**

El sistema está compuesto por dos aplicaciones Django principales y varios módulos de procesamiento independientes.

### **2.2. Flujo de Procesamiento de Documentos**

### **2.3. Flujo de Consulta (Chatbot)**

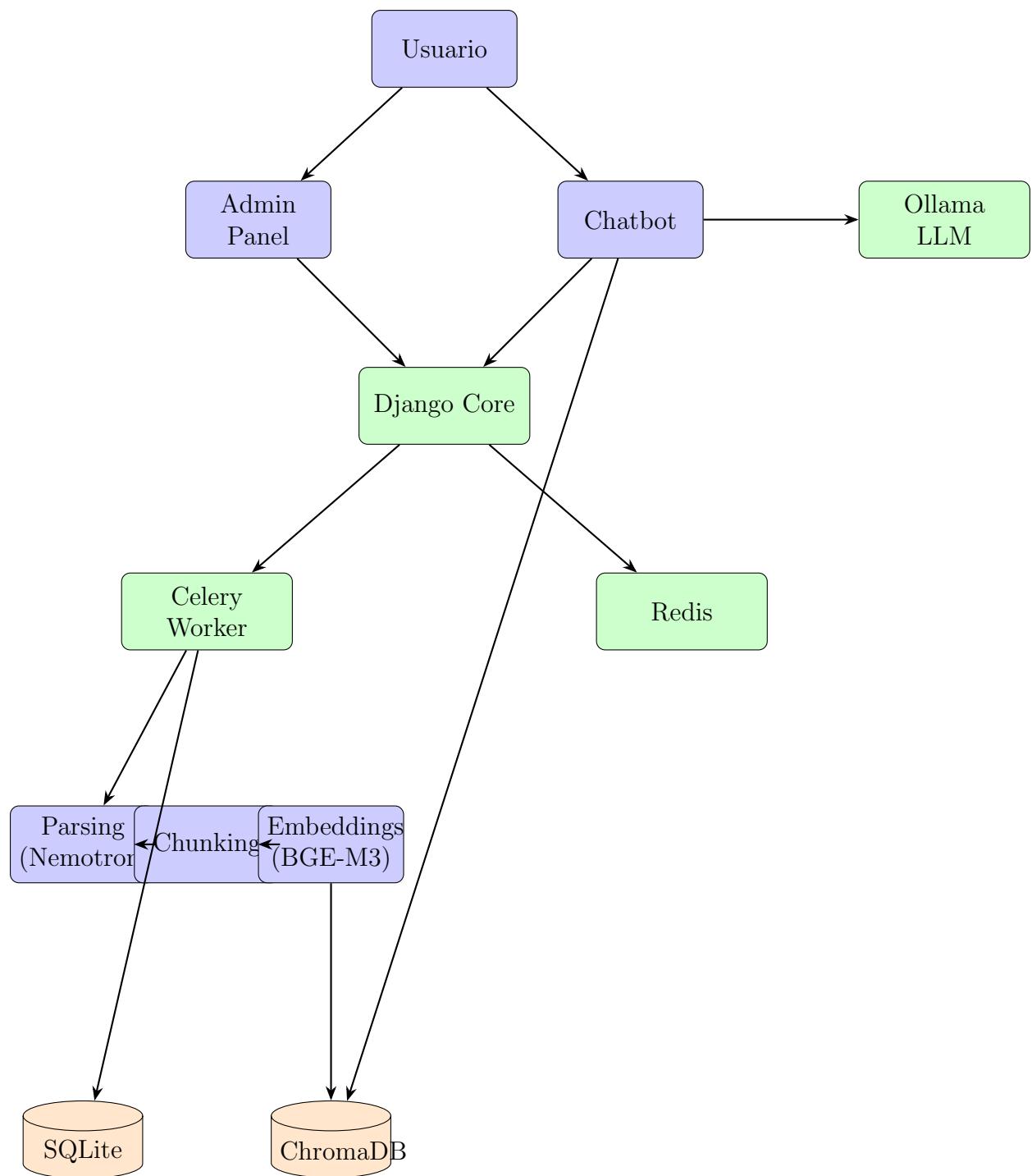


Figura 2.1: Arquitectura general del sistema

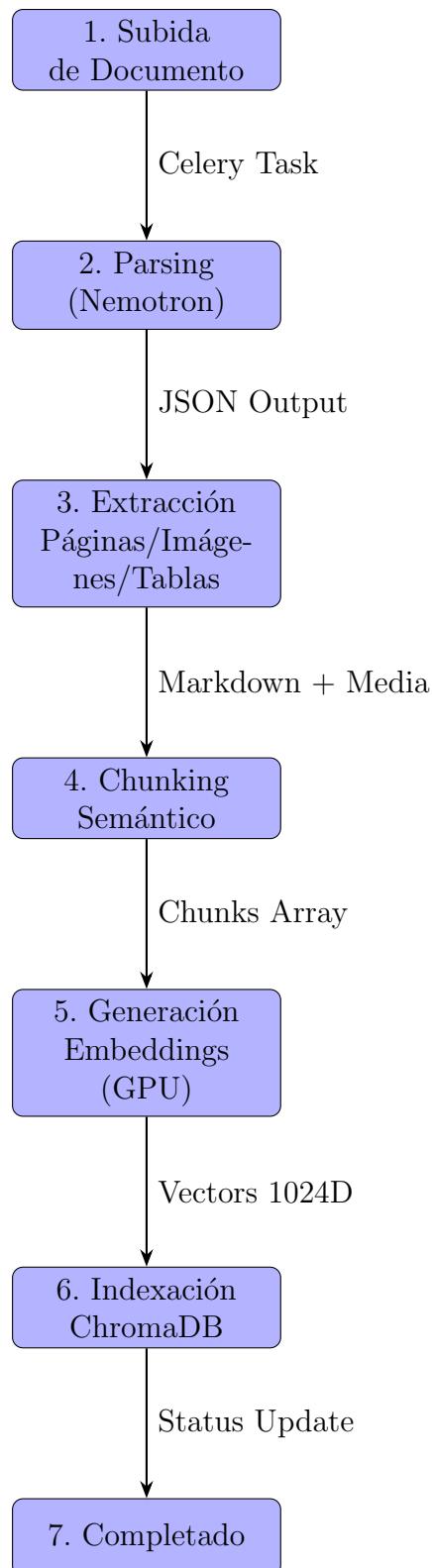


Figura 2.2: Pipeline de procesamiento de documentos

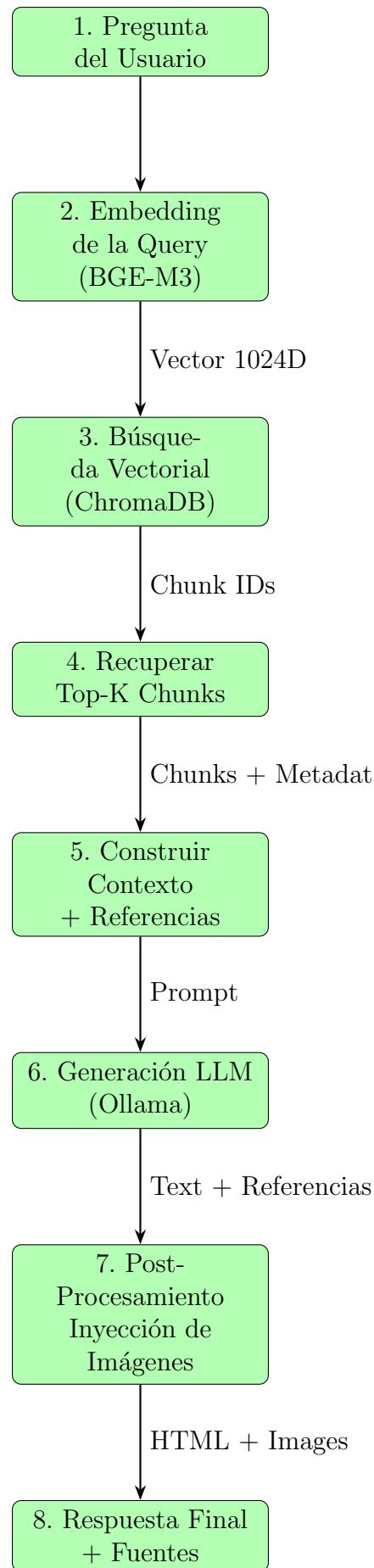


Figura 2.3: Pipeline de consulta del chatbot

# Capítulo 3

## Módulos del Sistema

### 3.1. Admin Panel

#### 3.1.1. Funcionalidad

El panel de administración proporciona una interfaz completa para gestionar el ciclo de vida de los documentos:

- **Subida de documentos:** Soporta PDF, DOCX, DOC, TXT, MD
- **Monitoreo de progreso:** 4 etapas con porcentaje en tiempo real
- **Visualización de contenido:** Páginas anotadas, imágenes extraídas, tablas
- **Gestión de chunks:** Exploración de fragmentos indexados
- **Logs detallados:** Trazabilidad completa del procesamiento
- **Reprocesamiento:** Reindexar documentos con nuevos parámetros

#### 3.1.2. Modelos de Datos

Listing 3.1: Modelo Document

```
class Document(models.Model):  
    title = models.CharField(max_length=500)  
    original_filename = models.CharField(max_length=500)  
    file = models.FileField(upload_to='documents/\%Y/\%m/\%d/')  
    status = models.CharField(max_length=20, choices=STATUS_CHOICES)  
    progress_percentage = models.IntegerField(default=0)  
  
    # Procesamiento  
    parsing_completed = models.BooleanField(default=False)  
    chunking_completed = models.BooleanField(default=False)  
    embedding_completed = models.BooleanField(default=False)  
    indexing_completed = models.BooleanField(default=False)  
  
    # Estadísticas  
    total_pages = models.IntegerField(default=0)  
    total_chunks = models.IntegerField(default=0)  
    total_images = models.IntegerField(default=0)  
    total_tables = models.IntegerField(default=0)
```

Listing 3.2: Modelo Chunk

```

class Chunk(models.Model):
    document = models.ForeignKey(Document, on_delete=models.CASCADE)
    chunk_id = models.CharField(max_length=100)
    content = models.TextField()
    chunk_index = models.IntegerField()
    metadata = models.JSONField(default=dict)

    # Embedding
    embedding_vector = models.JSONField(null=True, blank=True)
    embedding_dimension = models.IntegerField(null=True, blank=True)

    # ChromaDB
    chromadb_id = models.CharField(max_length=255)
    indexed_in_chromadb = models.BooleanField(default=False)

```

### 3.1.3. Tareas Celery

Listing 3.3: Tarea de procesamiento

```

@shared_task(bind=True)
def process_document_task(self, document_id):
    # Procesa documento completo en background
    doc = Document.objects.get(id=document_id)

    # 1. Parsing (25 porciento)
    parse_result = parse_document(doc.file.path)
    update_progress(doc, 25, 'parsing')

    # 2. Chunking (50 porciento)
    chunks = chunk_document(parse_result)
    update_progress(doc, 50, 'chunking')

    # 3. Embeddings (75 porciento)
    embeddings = generate_embeddings(chunks)
    update_progress(doc, 75, 'embedding')

    # 4. Indexing (100 porciento)
    index_to_chromadb(chunks, embeddings)
    update_progress(doc, 100, 'completed')

```

## 3.2. Chatbot

### 3.2.1. Funcionalidad

Interfaz conversacional pública que permite:

- **Consultas en lenguaje natural:** En español o inglés
- **Respuestas contextuales:** Basadas en los documentos indexados
- **Referencias visuales:** Tablas e imágenes embebidas automáticamente
- **Historial de conversación:** Sesiones independientes por usuario

- **Fuentes citadas:** Muestra los chunks utilizados

### 3.2.2. Modelos de Datos

Listing 3.4: Modelo Conversation

```
class Conversation(models.Model):
    session_id = models.CharField(max_length=100, unique=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

class Message(models.Model):
    MESSAGE_TYPES = [
        ('user', 'Usuario'),
        ('assistant', 'Asistente'),
    ]
    conversation = models.ForeignKey(Conversation)
    message_type = models.CharField(max_length=10, choices=MESSAGE_TYPES)
    content = models.TextField()
    retrieved_chunks = models.ManyToManyField('admin_panel.Chunk')
    created_at = models.DateTimeField(auto_now_add=True)
```

### 3.2.3. Lógica de Respuesta

Listing 3.5: Generación de respuesta

```
def generate_response(query):
    # 1. Generar embedding de la query
    query_embedding = embedding_generator.generate_single_embedding(
        query)

    # 2. Buscar en ChromaDB
    results = vector_store.query(
        query_embedding=query_embedding,
        n_results=5
    )

    # 3. Recuperar chunks
    chunk_ids = [r['metadata']['chunk_id'] for r in results['results']]
    chunks = Chunk.objects.filter(chunk_id__in=chunk_ids)

    # 4. Construir contexto
    context = build_context(chunks)

    # 5. Generar respuesta con LLM
    response = call_ollama(query, context)

    # 6. Post-procesar (inyectar imágenes)
    response = inject_media_references(response, chunks)

    return response, chunks
```

### 3.3. Parsing (Nemotron)

#### 3.3.1. Características

Utiliza el modelo nvidia/NVIDIA-Nemotron-Parse-v1.1 para:

- Extracción de texto con estructura preservada
- Detección y extracción de imágenes con bounding boxes
- Identificación de tablas con metadatos
- Salida en formato Markdown + JSON

#### 3.3.2. Configuración

Listing 3.6: Configuración de Nemotron

```
RAG_CONFIG = {
    'PARSING': {
        'MODEL': 'nvidia/NVIDIA-Nemotron-Parse-v1.1',
        'DEVICE': 'cuda' if torch.cuda.is_available() else 'cpu',
        'BATCH_SIZE': 4,
        'MAX_LENGTH': 512,
    }
}
```

### 3.4. Chunking Semántico

#### 3.4.1. Estrategia

Implementa chunking basado en contexto semántico:

- **Tamaño de chunk:** 1200 caracteres
- **Overlap:** 150 caracteres
- **Preservación de estructura:** Respeta párrafos y secciones
- **Metadatos:** Página, sección, documento, índice

#### 3.4.2. Implementación

Listing 3.7: Chunking semántico

```
class DocumentChunker:
    def __init__(self, chunk_size=1200, chunk_overlap=150):
        self.chunk_size = chunk_size
        self.chunk_overlap = chunk_overlap

    def chunk_document(self, text, metadata=None):
        chunks = []
        splitter = RecursiveCharacterTextSplitter(
```

```

        chunk_size=self.chunk_size,
        chunk_overlap=self.chunk_overlap,
        separators=["\n\n", "\n", ".\u2022", "\u2022", ""]
    )

    for i, chunk_text in enumerate(splitter.split_text(text)):
        chunk = {
            'chunk_id': f'chunk_{i:04d}',
            'content': chunk_text,
            'chunk_index': i,
            'metadata': metadata or {}
        }
        chunks.append(chunk)

    return chunks

```

## 3.5. Embeddings (BGE-M3)

### 3.5.1. Modelo

Utiliza BAAI/bge-m3 de SentenceTransformers:

- **Dimensiones:** 1024
- **Backend:** sentence-transformers
- **Dispositivo:** CUDA (GPU)
- **Multilingüe:** Español, inglés, y más de 100 idiomas

### 3.5.2. Implementación

Listing 3.8: Generador de embeddings

```

class EmbeddingGenerator:
    def __init__(self, model_name='BAAI/bge-m3'):
        self.device = 'cuda' if torch.cuda.is_available() else 'cpu'
        self.model = SentenceTransformer(model_name, device=self.device)
        self.dimension = 1024

    def generate_embeddings(self, texts):
        # Genera embeddings para multiples textos
        embeddings = self.model.encode(
            texts,
            convert_to_numpy=True,
            show_progress_bar=True
        )
        return embeddings

    def generate_single_embedding(self, text):
        # Genera embedding para un solo texto
        embedding = self.model.encode(text, convert_to_numpy=True)
        return embedding.tolist()

```

## 3.6. Vector Store (ChromaDB)

### 3.6.1. Configuración

Listing 3.9: Inicialización de ChromaDB

```
class VectorStore:
    def __init__(self, collection_name='rag_documents',
                 persist_directory='./chroma_db'):
        self.client = chromadb.PersistentClient(path=persist_directory)
        self.collection = self.client.get_or_create_collection(
            name=collection_name,
            metadata={"hnsw:space": "cosine"})
    )

    def add_documents(self, chunks, embeddings, metadatas, ids):
        # Aniade documentos a la colección
        self.collection.add(
            embeddings=embeddings,
            documents=chunks,
            metadatas=metadatas,
            ids=ids
        )

    def query(self, query_embedding, n_results=5):
        # Busca documentos similares
        results = self.collection.query(
            query_embeddings=[query_embedding],
            n_results=n_results,
            include=['documents', 'metadatas', 'distances']
        )
        return results
```

## 3.7. LLM (Ollama)

### 3.7.1. Integración

Utiliza Ollama como servidor LLM local:

- **Modelo:** gpt-oss:20b
- **Puerto:** 11434
- **API:** HTTP REST
- **Timeout:** 60 segundos

### 3.7.2. Prompt Engineering

Listing 3.10: Prompt del sistema

```
You are an expert assistant that answers questions based
on technical documents.
```

\*\*CRITICAL: Respond in the SAME LANGUAGE as the user's question.\*\*

Document context:  
{context}

User's question: {query}

Instructions:

- Answer clearly and concisely
- Use ONLY the information provided in the context
- If the information is not in the context, state it clearly
- Cite the document when relevant
- IMPORTANT: If you mention a table or image, include its EXACT reference (example: TABLA\_15 or IMAGEN\_8)

# Capítulo 4

## Despliegue y Configuración

### 4.1. Requisitos del Sistema

#### 4.1.1. Hardware

Componente	Especificación Recomendada
CPU	8+ cores
RAM	16 GB mínimo, 32 GB recomendado
GPU	NVIDIA con 12+ GB VRAM (RTX 3060+)
Almacenamiento	50 GB SSD

Cuadro 4.1: Requisitos de hardware

#### 4.1.2. Software

- Python 3.12+
- CUDA 12.0+ (para GPU)
- Redis Server
- Ollama (Docker o instalación local)

### 4.2. Instalación

#### 4.2.1. 1. Clonar Repositorio

```
git clone <repository_url>
cd WebApp
```

#### 4.2.2. 2. Crear Entorno Virtual

```
python -m venv venv
source venv/bin/activate  # Linux/Mac
venv\Scripts\activate     # Windows
```

### 4.2.3. 3. Instalar Dependencias

```
pip install -r requirements.txt
```

### 4.2.4. 4. Configurar Base de Datos

```
python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser
```

### 4.2.5. 5. Iniciar Redis

```
redis-server
```

### 4.2.6. 6. Iniciar Celery Worker

```
celery -A rag_project worker -l info --pool=solo
```

### 4.2.7. 7. Iniciar Ollama

```
# Docker
docker run -d -p 11434:11434 --gpus all ollama/ollama
docker exec -it <container_id> ollama pull gpt-oss:20b

# O instalacion local
ollama serve
ollama pull gpt-oss:20b
```

### 4.2.8. 8. Iniciar Django

```
python manage.py runserver
```

## 4.3. Configuración

### 4.3.1. Settings de Django

Listing 4.1: rag-project/settings.py

```
# RAG Configuration
RAG_CONFIG = {
    'PARSING': {
        'MODEL': 'nvidia/NVIDIA-Nemotron-Parse-v1.1',
        'DEVICE': 'cuda',
    },
    'CHUNKING': {
        'CHUNK_SIZE': 1200,
```

```
'CHUNK_OVERLAP': 150,
'STRATEGY': 'semantic',
},
'EMBEDDINGS': {
    'MODEL': 'BAAI/bge-m3',
    'DIMENSION': 1024,
    'DEVICE': 'cuda',
},
'VECTOR_STORE': {
    'TYPE': 'chromadb',
    'COLLECTION_NAME': 'rag_documents',
    'PERSIST_DIRECTORY': os.path.join(BASE_DIR, 'chroma_db'),
}
}

# Ollama Configuration
OLLAMA_CONFIG = {
    'URL': 'http://localhost:11434',
    'MODEL': 'gpt-oss:20b',
    'TEMPERATURE': 0.7,
    'TOP_P': 0.9,
    'TIMEOUT': 60,
}

# Celery Configuration
CELERY_BROKER_URL = 'redis://localhost:6379/0'
CELERY_RESULT_BACKEND = 'redis://localhost:6379/0'
```

# **Capítulo 5**

## **Uso del Sistema**

### **5.1. Panel de Administración**

#### **5.1.1. Acceso**

1. Navegar a <http://localhost:8000/admin-panel/>
2. Iniciar sesión con credenciales de administrador

#### **5.1.2. Subir Documento**

1. Click en ”Subir Documento”
2. Seleccionar archivo (PDF, DOCX, DOC, TXT, MD)
3. Ingresar título
4. Click en ”Procesar Documento”
5. Monitorear progreso en dashboard

#### **5.1.3. Explorar Contenido**

1. Click en documento en el dashboard
2. Ver páginas anotadas, imágenes, tablas
3. Explorar chunks generados
4. Revisar logs de procesamiento

### **5.2. Chatbot**

#### **5.2.1. Acceso**

1. Navegar a <http://localhost:8000/>
2. No requiere autenticación

### 5.2.2. Realizar Consultas

1. Escribir pregunta en el campo de texto
2. Presionar Enter o click en “Enviar”
3. Esperar respuesta (10-30 segundos primera vez)
4. Ver respuesta con tablas/imágenes embebidas
5. Revisar fuentes consultadas al final

### 5.2.3. Nueva Conversación

1. Click en ”Nueva conversación”
2. Se crea una nueva sesión independiente

# Capítulo 6

## Diagramas de Secuencia

### 6.1. Procesamiento de Documento

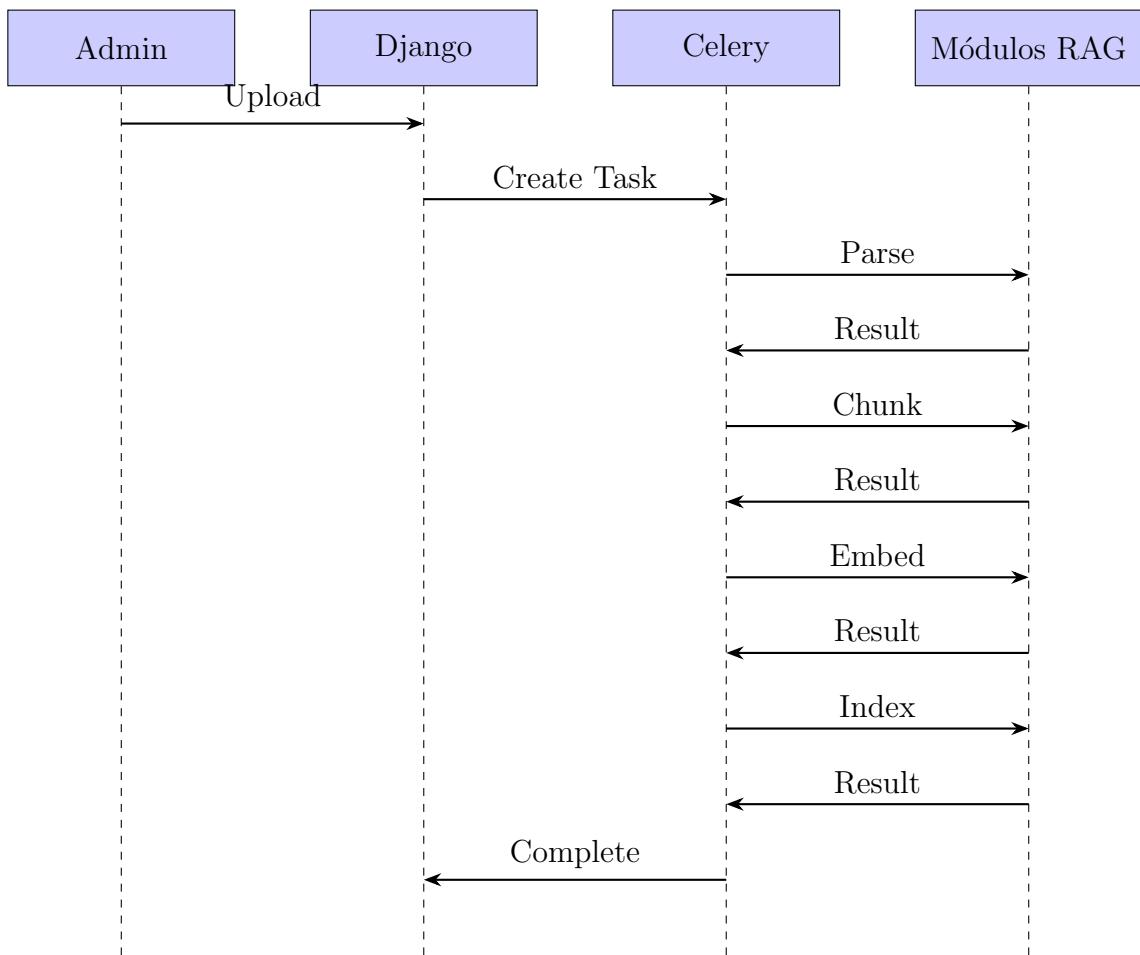


Figura 6.1: Diagrama de secuencia: Procesamiento de documento

### 6.2. Consulta en Chatbot

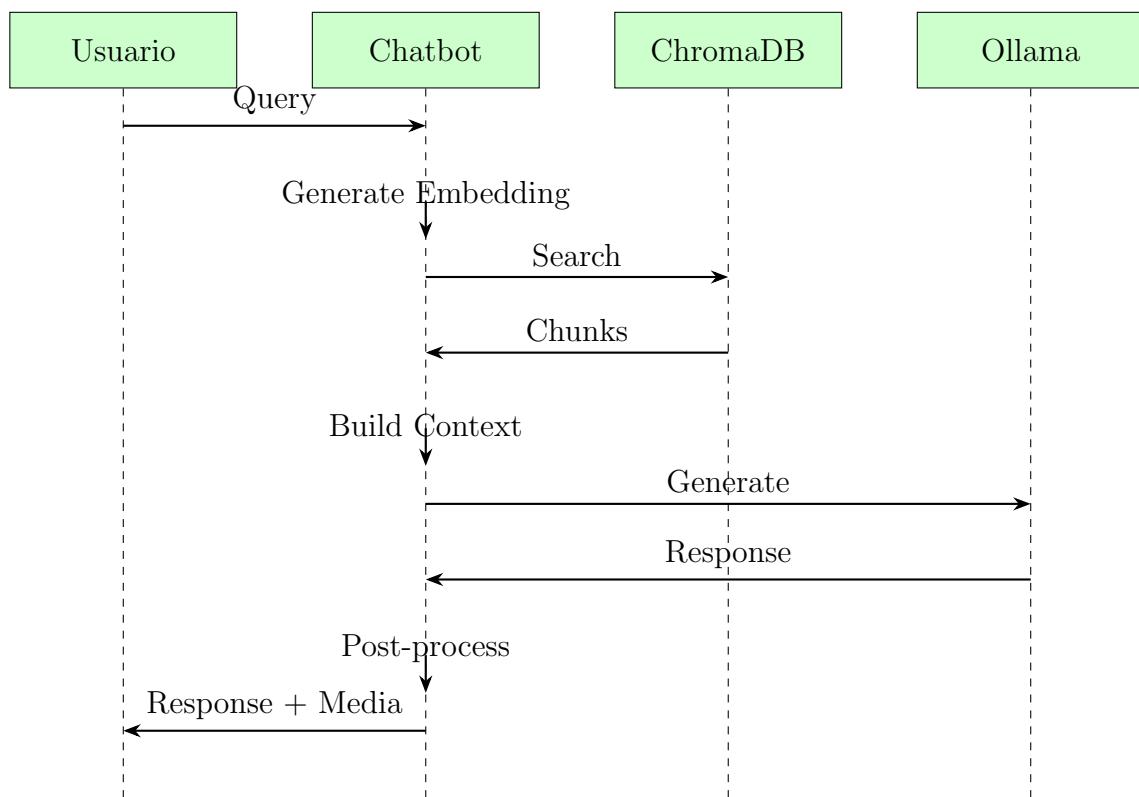


Figura 6.2: Diagrama de secuencia: Consulta en chatbot

# Capítulo 7

## Rendimiento y Optimización

### 7.1. Métricas de Rendimiento

Operación	Tiempo Promedio	Notas
Parsing (10 páginas)	2-3 minutos	GPU acelerado
Chunking	5-10 segundos	
Embedding generation	10-30 segundos	Batch de 30 chunks
ChromaDB indexing	1-2 segundos	
Query embedding	0.5-1 segundo	
Vector search	0.1-0.5 segundos	
LLM generation	10-30 segundos	Primera llamada
LLM generation	3-5 segundos	Llamadas siguientes

Cuadro 7.1: Métricas de rendimiento

### 7.2. Optimizaciones Implementadas

- **GPU acceleration:** CUDA para embeddings y parsing
- **Batch processing:** Embeddings generados en lotes
- **Async processing:** Celery para tareas largas
- **Caching:** Redis para resultados intermedios
- **Lazy loading:** Imágenes cargadas bajo demanda
- **Connection pooling:** ChromaDB persistent client

### 7.3. Recomendaciones

1. **Pre-cargar modelos:** Warm-up de BGE-M3 y Ollama al inicio
2. **Monitoreo:** Usar Flower para Celery tasks
3. **Logs:** Configurar logging estructurado

4. **Backups:** Respaldar SQLite y ChromaDB periódicamente
5. **Escalabilidad:** Considerar PostgreSQL para producción

# Capítulo 8

## Resolución de Problemas

### 8.1. Problemas Comunes

#### 8.1.1. ChromaDB no encuentra documentos

**Síntoma:** Query returns empty results

**Solución:**

1. Verificar que documentos están indexados: `collection.count()`
2. Revisar dimensión de embeddings (debe ser 1024)
3. Comprobar nombres de colección coinciden
4. Verificar `persist_directory` es correcto

#### 8.1.2. Celery task fails

**Síntoma:** Processing stuck o failed

**Solución:**

1. Revisar logs de Celery worker
2. Verificar Redis está corriendo: `redis-cli ping`
3. Comprobar permisos de archivos
4. Revisar memoria GPU disponible

#### 8.1.3. Ollama timeout

**Síntoma:** LLM no responde o timeout

**Solución:**

1. Verificar Ollama está corriendo: `curl http://localhost:11434`
2. Aumentar timeout en settings
3. Revisar logs de Ollama
4. Considerar modelo más pequeño si GPU limitada

### 8.1.4. Out of GPU memory

**Síntoma:** CUDA out of memory error

**Solución:**

1. Reducir batch size de embeddings
2. Cerrar otros procesos usando GPU
3. Usar modelo más pequeño
4. Considerar CPU fallback

# Capítulo 9

## Conclusiones y Trabajo Futuro

### 9.1. Logros del Proyecto

- ✓ Pipeline RAG completo funcional
- ✓ Procesamiento automatico de documentos
- ✓ Busqueda vectorial eficiente
- ✓ Chatbot conversacional multilingue
- ✓ Integracion de tablas e imagenes
- ✓ Interface administrativa completa
- ✓ Procesamiento asincrono robusto

### 9.2. Trabajo Futuro

#### 9.2.1. Funcionalidades

- Reranking de resultados
- Multi-query fusion
- Feedback de usuarios sobre respuestas
- Exportar conversaciones a PDF
- API REST para integración externa
- Autenticación de usuarios para chatbot
- Dashboard de analytics

### 9.2.2. Optimizaciones

- Quantización de modelos
- Cacheo de embeddings frecuentes
- Streaming de respuestas LLM
- Pre-fetching de imágenes
- Compresión de imágenes

### 9.2.3. Escalabilidad

- Migración a PostgreSQL
- Implementar load balancing
- Kubernetes deployment
- Multi-tenancy support
- Distributed ChromaDB

## 9.3. Reflexión Final

Este proyecto demuestra la viabilidad de implementar un sistema RAG completo utilizando tecnologías open-source y modelos de última generación. La combinación de Django, ChromaDB, y Ollama proporciona una base sólida para aplicaciones de IA conversacional basadas en documentos.

La arquitectura modular permite fácil extensión y personalización, mientras que el procesamiento asíncrono garantiza una experiencia de usuario fluida incluso con documentos grandes.

# Apéndice A

## Apéndices

### A.1. Comandos Útiles

#### A.1.1. Django

```
# Crear migraciones
python manage.py makemigrations

# Aplicar migraciones
python manage.py migrate

# Crear superusuario
python manage.py createsuperuser

# Shell interactivo
python manage.py shell

# Colectar archivos estaticos
python manage.py collectstatic
```

#### A.1.2. Celery

```
# Iniciar worker
celery -A rag_project worker -l info --pool=solo

# Monitorear con Flower
celery -A rag_project flower

# Purge all tasks
celery -A rag_project purge
```

#### A.1.3. Redis

```
# Iniciar servidor
redis-server

# CLI
redis-cli
```

```
# Ping
redis-cli ping

# Flush all
redis-cli FLUSHALL
```

## A.2. Estructura de Directorios

```
WebApp/
    admin_panel/           # App de administracion
        models.py
        views.py
        tasks.py
        urls.py
        templates/
    chatbot/               # App de chatbot
        models.py
        views.py
        urls.py
        templates/
    rag_project/          # Configuracion Django
        settings.py
        urls.py
        celery.py
    tools/                 # Modulos RAG
        document_chunker.py
        embedding_generator.py
        vector_store.py
        reranker.py
    media/                 # Archivos subidos
        documents/
        extracted_images/
        extracted_tables/
    chroma_db/             # Base de datos vectorial
    templates/             # Templates globales
    manage.py
    requirements.txt
```

## A.3. Referencias

- Django Documentation: <https://docs.djangoproject.com/>
  - ChromaDB Documentation: <https://docs.trychroma.com/>
  - Sentence Transformers: <https://www.sbert.net/>
  - Ollama Documentation: <https://ollama.ai/>
  - Celery Documentation: <https://docs.celeryproject.org/>
  - BGE-M3 Paper: <https://huggingface.co/BAAI/bge-m3>
  - Nemotron Parse: <https://huggingface.co/nvidia/NVIDIA-Nemotron-Parse-v1>.
- 1