# DocHub Developer Documentation

## Table of Contents

1. [Introduction](#introduction)
2. [System Architecture](#system-architecture)
3. [Data Models](#data-models)
4. [Signal System](#signal-system)
5. [API Endpoints](#api-endpoints)
6. [Development Environment Setup](#development-environment-setup)
7. [Testing](#testing)
8. [Troubleshooting](#troubleshooting)

## Introduction

DocHub is a document management system with knowledge graph capabilities. The system allows users to:

* Organize documents in folders
* Upload and manage documents
* Process documents to extract text and entities
* Build and query knowledge graphs
* Search document content semantically

This documentation provides comprehensive information for developers working on the DocHub project.

## System Architecture

### Overview

DocHub is built with Django and uses a modern tech stack:

* **Backend**: Django REST Framework
* **Database**: SQLite (can be configured for PostgreSQL in production)
* **File Storage**: Django’s file storage system
* **Task Queue**: Celery with Redis
* **Vector Storage**: ChromaDB
* **Graph Database**: Neo4j
* **Text Processing**: OpenAI and LangChain

### File Structure

backend/  
├── config/ # Project configuration  
│ ├── \_\_init\_\_.py  
│ ├── asgi.py  
│ ├── celery.py # Celery configuration  
│ ├── settings.py # Django settings  
│ ├── urls.py # Main URL routing  
│ └── wsgi.py  
├── dochub/ # Document management app  
│ ├── api/ # API layer  
│ │ ├── \_\_init\_\_.py  
│ │ ├── serializers.py # Data serializers  
│ │ ├── urls.py # API endpoints  
│ │ └── views.py # API views  
│ ├── management/ # Custom management commands  
│ │ └── commands/  
│ │ └── test\_pipeline.py # Pipeline testing command  
│ ├── models/ # Data models  
│ │ ├── \_\_init\_\_.py  
│ │ ├── document.py # Document model  
│ │ └── folder.py # Folder model  
│ ├── pipeline/ # Document processing pipeline  
│ │ ├── extractors/ # Text extraction  
│ │ │ ├── base.py # Base extractor interface  
│ │ │ └── docling\_extractor.py # Implementation  
│ │ ├── splitters/ # Text splitting  
│ │ │ ├── base.py # Base splitter interface  
│ │ │ └── langchain\_splitter.py # Implementation  
│ │ ├── embeddings/ # Embedding generation  
│ │ │ ├── base.py # Base generator interface  
│ │ │ └── openai\_embeddings.py # Implementation  
│ │ ├── indexers/ # Vector indexing  
│ │ │ ├── base.py # Base indexer interface  
│ │ │ └── chroma\_indexer.py # Implementation  
│ │ └── graphs/ # Knowledge graph generation  
│ │ ├── client.py # Neo4j client  
│ │ ├── generator.py # Graph generator  
│ │ └── schema.py # Schema manager  
│ ├── services/ # Business logic services  
│ │ ├── \_\_init\_\_.py  
│ │ ├── document\_service.py # Document processing service  
│ │ └── search\_service.py # Search functionality  
│ ├── tasks/ # Celery tasks  
│ │ ├── \_\_init\_\_.py  
│ │ └── document\_tasks.py # Document processing tasks  
│ ├── templates/ # HTML templates  
│ │ └── debug/  
│ │ └── test\_dashboard.html # Pipeline test dashboard  
│ ├── utils/ # Utility functions  
│ │ ├── pipeline\_logger.py # Pipeline testing logger  
│ │ └── graph\_visualizer.py # Graph visualization utility  
│ ├── \_\_init\_\_.py  
│ ├── apps.py # App configuration  
│ ├── signals.py # Signal handlers  
│ ├── urls.py # URL routing  
│ └── views.py # Django views  
├── chatbot/ # Chatbot app with RAG capabilities  
├── media/ # Media storage  
│ ├── Documents/ # Root folder for all documents  
│ └── pipeline\_tests/ # Pipeline test artifacts  
└── utils/ # Shared utilities

### Directory Structure

All documents and folders are stored in a single directory structure under /media/Documents/. The system maintains a hierarchical structure:

media/  
└── Documents/ # Root directory  
 ├── file1.pdf # Root-level document  
 ├── ProjectA/ # Folder  
 │ ├── document1.pdf # Document in folder  
 │ └── document2.docx # Another document  
 └── ProjectB/ # Another folder  
 ├── requirements.txt # Document  
 └── Subproject/ # Nested folder  
 └── design.pdf # Document in nested folder

This structure allows for organizing documents in a logical hierarchy while maintaining a clean physical structure.

## Data Models

DocHub has two primary models:

### Folder Model

class Folder(models.Model):  
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 name = models.CharField(max\_length=255)  
 parent = models.ForeignKey('self', on\_delete=models.CASCADE,   
 null=True, blank=True, related\_name='subfolders')  
 created\_at = models.DateTimeField(auto\_now\_add=True)  
 updated\_at = models.DateTimeField(auto\_now=True)

Features: - UUIDs for IDs - Self-referential for folder hierarchy - Timestamps for creation and updates

### Document Model

class Document(models.Model):  
 STATUS\_CHOICES = (  
 ('processing', 'Processing'),  
 ('ready', 'Ready'),  
 ('error', 'Error'),  
 )  
   
 id = models.UUIDField(primary\_key=True, default=uuid.uuid4, editable=False)  
 name = models.CharField(max\_length=255)  
 file = models.FileField(upload\_to=document\_upload\_path)  
 folder = models.ForeignKey(Folder, on\_delete=models.CASCADE,   
 null=True, blank=True, related\_name='documents')  
 file\_type = models.CharField(max\_length=100, blank=True)  
 size = models.PositiveIntegerField(default=0)  
 status = models.CharField(max\_length=20, choices=STATUS\_CHOICES, default='processing')  
 error\_message = models.TextField(blank=True, null=True)  
 created\_at = models.DateTimeField(auto\_now\_add=True)  
 updated\_at = models.DateTimeField(auto\_now=True)

Features: - UUIDs for IDs - File attachment with custom upload path - Folder association - Processing status tracking - File metadata (type, size) - Error handling - Timestamps for creation and updates

### Path Building Functions

The system uses special functions to build file paths:

def build\_folder\_path(folder):  
 """  
 Recursively build the folder path based on hierarchy.  
 Returns a path like: Documents/ParentFolder/ChildFolder/  
 """  
 parts = ["Documents"] # Start with Documents as root  
 current = folder  
 while current:  
 parts.append(current.name)  
 current = current.parent  
 return os.path.join(\*parts)  
  
def document\_upload\_path(instance, filename):  
 """  
 Define upload path for documents.  
 Returns a path like: Documents/FolderHierarchy/filename  
 """  
 if instance.folder:  
 folder\_path = build\_folder\_path(instance.folder)  
 return os.path.join(folder\_path, filename)  
 else:  
 return os.path.join("Documents", filename)

## Signal System

Django signals are used extensively to handle various operations automatically:

### Document Signals

#### post\_save (Document)

Triggered when a document is created or updated: 1. Updates file metadata (file\_type, size) 2. Sets status to ‘processing’ 3. Triggers the document processing task asynchronously

@receiver(post\_save, sender=Document)  
def handle\_document\_post\_save(sender, instance, created, \*\*kwargs):  
 if created and instance.file:  
 # Update metadata  
 # Queue processing task  
 process\_document\_task.delay(str(instance.id))

#### pre\_delete (Document)

Triggered before a document is deleted: 1. Deletes the physical file 2. Cleans up empty directories

@receiver(pre\_delete, sender=Document)  
def handle\_document\_pre\_delete(sender, instance, \*\*kwargs):  
 if instance.file:  
 # Delete physical file  
 # Clean up empty directories

### Folder Signals

#### post\_save (Folder)

Triggered when a folder is created or updated: 1. Creates the physical directory structure

@receiver(post\_save, sender=Folder)  
def handle\_folder\_post\_save(sender, instance, created, \*\*kwargs):  
 if created:  
 # Create physical directory  
 folder\_physical\_path = os.path.join(settings.MEDIA\_ROOT, build\_folder\_path(instance))  
 os.makedirs(folder\_physical\_path, exist\_ok=True)

#### pre\_save (Folder)

Triggered before a folder is saved: 1. Handles folder renaming by moving the physical directory

@receiver(pre\_save, sender=Folder)  
def handle\_folder\_pre\_save(sender, instance, \*\*kwargs):  
 if not instance.pk:  
 return # Skip for new folders  
   
 # Get old folder  
 # If name changed, rename physical directory

### Startup Signal

The app’s ready() method creates initial structures:

def ready(self):  
 import dochub.signals  
   
 # Create initial Documents directory  
 # Create root folder if it doesn't exist

## API Endpoints

DocHub exposes the following RESTful API endpoints:

### Folder Operations

| Method | Endpoint | Description |
| --- | --- | --- |
| GET | /api/dochub/folders/ | List all folders (can filter by parent) |
| POST | /api/dochub/folders/ | Create a new folder |
| GET | /api/dochub/folders/{id}/ | Get a specific folder’s details |
| PUT | /api/dochub/folders/{id}/ | Update a folder |
| DELETE | /api/dochub/folders/{id}/ | Delete a folder |
| GET | /api/dochub/folders/{id}/documents/ | List documents in a folder |
| GET | /api/dochub/folders/{id}/subfolders/ | List subfolders of a folder |

#### Example: Create Folder

Request:

curl -X POST http://localhost:8000/api/dochub/folders/ \  
 -H "Content-Type: application/json" \  
 -d '{"name": "ProjectA", "parent": "uuid-of-parent-folder"}'

Response:

{  
 "id": "550e8400-e29b-41d4-a716-446655440000",  
 "name": "ProjectA",  
 "parent": "uuid-of-parent-folder",  
 "created\_at": "2025-03-11T12:34:56Z",  
 "updated\_at": "2025-03-11T12:34:56Z",  
 "document\_count": 0,  
 "subfolder\_count": 0  
}

### Document Operations

| Method | Endpoint | Description |
| --- | --- | --- |
| GET | /api/dochub/documents/ | List all documents (can filter by folder) |
| POST | /api/dochub/documents/ | Upload a document |
| GET | /api/dochub/documents/{id}/ | Get a specific document’s details |
| PUT | /api/dochub/documents/{id}/ | Update a document |
| DELETE | /api/dochub/documents/{id}/ | Delete a document |
| POST | /api/dochub/documents/bulk\_upload/ | Upload multiple documents |

#### Example: Upload Document

Request:

curl -X POST http://localhost:8000/api/dochub/documents/ \  
 -F "name=requirements.txt" \  
 -F "file=@/path/to/local/requirements.txt" \  
 -F "folder=uuid-of-folder"

Response:

{  
 "id": "550e8400-e29b-41d4-a716-446655440001",  
 "name": "requirements.txt",  
 "file": "/media/Documents/ProjectA/requirements.txt",  
 "url": "http://localhost:8000/media/Documents/ProjectA/requirements.txt",  
 "folder": "uuid-of-folder",  
 "file\_type": "txt",  
 "size": 1024,  
 "status": "processing",  
 "created\_at": "2025-03-11T12:45:30Z",  
 "updated\_at": "2025-03-11T12:45:30Z"  
}

### Bulk Operations

| Method | Endpoint | Description |
| --- | --- | --- |
| POST | /api/dochub/documents/bulk\_upload/ | Upload multiple documents |
| POST | /api/dochub/bulk\_delete/ | Delete multiple folders and documents |

#### Example: Bulk Upload

Request:

curl -X POST http://localhost:8000/api/dochub/documents/bulk\_upload/ \  
 -F "files=@/path/to/file1.pdf" \  
 -F "files=@/path/to/file2.docx" \  
 -F "folder=uuid-of-folder"

#### Example: Bulk Delete

Request:

curl -X POST http://localhost:8000/api/dochub/bulk\_delete/ \  
 -H "Content-Type: application/json" \  
 -d '{  
 "folder\_ids": ["uuid1", "uuid2"],  
 "document\_ids": ["uuid3", "uuid4"]  
 }'

### Graph Operations

| Method | Endpoint | Description |
| --- | --- | --- |
| GET | /api/dochub/graph/document/{id}/ | Get knowledge graph for a document |
| GET | /api/dochub/graph/folder/{id}/ | Get knowledge graph for a folder |
| GET | /api/dochub/graph/entity/ | Get knowledge graph for an entity |

## Development Environment Setup

### Prerequisites

* Python 3.10 or newer
* Node.js 16+ and npm (for frontend)
* Redis server (for Celery)
* Neo4j (optional, for knowledge graph functionality)
* UV package manager (for dependency management)

### Installation Steps

1. **Clone the repository:**

* git clone https://github.com/your-org/AIHackathon-RAGLens.git  
  cd AIHackathon-RAGLens

1. **Install UV package manager:**

* # Install UV  
  curl -fsSL https://github.com/astral-sh/uv/releases/download/0.1.19/uv-installer.sh | bash

1. **Create and activate a virtual environment:**

* cd backend  
  uv venv  
  source .venv/bin/activate # On Windows: .venv\Scripts\activate

1. **Install dependencies:**

* uv pip install -r requirements.txt

1. **Create required directories:**

* mkdir -p media/Documents  
  chmod -R 755 media

1. **Apply migrations:**

* python manage.py makemigrations dochub  
  python manage.py makemigrations chatbot  
  python manage.py migrate

1. **Start Redis server:**

* # Install Redis if needed  
  sudo apt-get install redis-server # Ubuntu/Debian  
    
  # Start Redis  
  sudo service redis-server start # Ubuntu/Debian  
  # or  
  redis-server # Direct command

1. **Start the development server:**

* python manage.py runserver

1. **In a new terminal, start the Celery worker:**

* cd path/to/backend  
  source .venv/bin/activate  
  celery -A config worker --loglevel=info

### Configuration

The system uses environment variables for configuration. Create a .env file in the backend directory:

# Django settings  
SECRET\_KEY=your-secret-key  
DEBUG=True  
ALLOWED\_HOSTS=localhost,127.0.0.1  
  
# Neo4j settings (optional)  
NEO4J\_URI=bolt://localhost:7687  
NEO4J\_USERNAME=neo4j  
NEO4J\_PASSWORD=password  
  
# OpenAI API key (required for document processing)  
OPENAI\_API\_KEY=your-openai-api-key  
  
# Celery settings  
CELERY\_BROKER\_URL=redis://localhost:6379/0  
CELERY\_RESULT\_BACKEND=redis://localhost:6379/0

## Testing

### API Testing with cURL

Test the folder creation:

curl -X POST http://localhost:8000/api/dochub/folders/ \  
 -H "Content-Type: application/json" \  
 -d '{"name": "TestFolder", "parent": null}'

Test document upload:

curl -X POST http://localhost:8000/api/dochub/documents/ \  
 -F "name=test.pdf" \  
 -F "file=@/path/to/test.pdf" \  
 -F "folder=folder-uuid-from-previous-step"

### Pipeline Testing Framework

The system includes a comprehensive pipeline testing framework:

#### Pipeline Logger

The PipelineLogger class in dochub/utils/pipeline\_logger.py captures detailed information during pipeline execution:

class PipelineLogger:  
 def \_\_init\_\_(self, document\_id, log\_level=logging.DEBUG, save\_artifacts=True):  
 self.document\_id = str(document\_id)  
 self.log\_level = log\_level  
 self.save\_artifacts = save\_artifacts  
 self.metrics = {}  
 self.step\_times = {}  
   
 if self.save\_artifacts:  
 self.artifact\_dir = Path(settings.MEDIA\_ROOT) / 'pipeline\_tests' / self.document\_id  
 os.makedirs(self.artifact\_dir, exist\_ok=True)  
   
 def log\_step(self, step\_name):  
 """Decorator to log and time a pipeline step"""  
 # Implementation details...

Key features: - Step timing and performance metrics - Artifact saving for detailed analysis - Detailed logging of each processing step - OpenAI API request/response tracking

#### Test Command

The custom management command test\_pipeline in dochub/management/commands/test\_pipeline.py allows running the pipeline with detailed logging:

# Basic test  
python manage.py test\_pipeline <document\_id>  
  
# Detailed test with artifact saving  
python manage.py test\_pipeline <document\_id> --verbose --save-artifacts  
  
# Specify OpenAI model for graph generation  
python manage.py test\_pipeline <document\_id> --openai-model=gpt-4o-mini

This command provides real-time feedback on the pipeline’s performance and can save intermediate artifacts for analysis.

#### Test Dashboard

The test dashboard at /dochub/test-dashboard/<document\_id>/ displays detailed pipeline results:

* Document information and metadata
* Step-by-step performance metrics
* Text extraction results and statistics
* Chunk information with token counts
* Embedding visualizations
* Knowledge graph visualization
* API calls and responses

#### Using the Testing Framework

To test a document through the pipeline:

1. Upload a document using the API or frontend
2. Note the document UUID
3. Run the test command:

* python manage.py test\_pipeline <document-uuid> --verbose --save-artifacts

1. View the results in the dashboard:

* http://localhost:8000/dochub/test-dashboard/<document-uuid>/

### Automated Unit Testing

Run the Django test suite:

python manage.py test dochub

For testing specific components:

# Test extractors  
python manage.py test dochub.tests.test\_extractors  
  
# Test the entire pipeline  
python manage.py test dochub.tests.test\_pipeline

## Troubleshooting

### Common Issues

#### Media Directory Issues

**Problem**: Files aren’t being saved correctly or are inaccessible.

**Solution**: 1. Check directory permissions: bash chmod -R 755 media 2. Verify the MEDIA\_ROOT setting in settings.py points to the correct location. 3. Ensure the directory structure exists: bash mkdir -p media/Documents mkdir -p media/pipeline\_tests

#### Celery Not Processing Tasks

**Problem**: Documents stay in ‘processing’ status.

**Solution**: 1. Verify Redis is running: bash redis-cli ping # Should return PONG 2. Check Celery worker logs for errors. 3. Restart the Celery worker: bash celery -A config worker --loglevel=info 4. Run the cleanup task to fix stuck documents: python from dochub.tasks.document\_tasks import cleanup\_processing\_documents cleanup\_processing\_documents.delay()

#### Pipeline Component Failures

**Problem**: Document processing fails in a specific pipeline stage.

**Solutions by component**:

1. **Text Extraction Issues**:
   * Check file format support
   * Verify file is not corrupted or password protected
   * For PDFs, check if OCR is needed for scanned documents
   * Try reprocessing using the test command:
   * python manage.py test\_pipeline <document-id> --verbose
2. **Embedding Generation Issues**:
   * Verify OpenAI API key is valid and has sufficient quota
   * Check for rate limiting errors in logs
   * Try with a smaller chunk size or different model:
   * # In openai\_embeddings.py  
     EMBEDDING\_MODEL = "text-embedding-ada-002" # Change if needed
3. **ChromaDB Indexing Issues**:
   * Check that ChromaDB is properly initialized
   * Verify persistence directory exists and is writable
   * If database is corrupted, you may need to reset it:
   * # Back up first!  
     rm -rf backend/chroma\_db/\*
4. **Graph Generation Issues**:
   * Check Neo4j connection parameters
   * Verify Neo4j is running and accessible
   * Check OpenAI API errors in logs
   * Try with a simpler model like gpt-3.5-turbo if rate limited

#### Database Migration Errors

**Problem**: Migrations fail with errors.

**Solution**: 1. Delete any conflicting migration files if they exist. 2. Reset the migrations if needed (backup your data first): bash python manage.py migrate dochub zero rm dochub/migrations/0\*.py python manage.py makemigrations dochub python manage.py migrate dochub

### Debug Tools

#### Pipeline Logger

When troubleshooting pipeline issues, use the pipeline logger to capture detailed information:

from dochub.utils.pipeline\_logger import PipelineLogger  
  
logger = PipelineLogger(document\_id='your-doc-id', save\_artifacts=True)  
  
@logger.log\_step("extraction")  
def test\_extraction(file\_path):  
 # Your extraction code here  
 pass

#### Test Dashboard

The test dashboard provides visual insights into pipeline failures:

http://localhost:8000/dochub/test-dashboard/<document-id>/

#### Manual Step Testing

You can test individual pipeline components manually:

from django.core.management import call\_command  
  
# Test with verbose output and artifact saving  
call\_command('test\_pipeline', 'document-id', verbose=True, save\_artifacts=True)

### Logs

Check the following logs for troubleshooting:

1. **Django Server Logs**: Output from the Django development server
2. **Celery Worker Logs**: Output from the Celery worker process
3. **Pipeline Test Logs**: Saved in media/pipeline\_tests/<document\_id>/
4. **debug.log**: Application-wide logging in the backend directory

For real-time logging, run the server and worker with increased verbosity:

# Django with debug output  
python manage.py runserver --verbosity 2  
  
# Celery with debug output  
celery -A config worker --loglevel=debug

## Document Processing Flow

### Overview

When a document is uploaded:

1. The post\_save signal in signals.py triggers process\_document\_task
2. The task orchestrates the document processing pipeline:
   * Extracts text from the document
   * Splits text into manageable chunks
   * Generates embeddings for each chunk
   * Indexes chunks and embeddings in ChromaDB
   * Extracts entities and relationships
   * Builds a knowledge graph in Neo4j
3. The document status is updated to ‘ready’ upon successful processing

### Pipeline Architecture

The document processing pipeline is implemented with a modular architecture following clear interfaces:

dochub/  
├── pipeline/ # Processing pipeline components  
│ ├── extractors/ # Text extraction  
│ │ ├── base.py # Base extractor interface  
│ │ └── docling\_extractor.py # Implementation  
│ ├── splitters/ # Text splitting  
│ │ ├── base.py # Base splitter interface  
│ │ └── langchain\_splitter.py # Implementation  
│ ├── embeddings/ # Embedding generation  
│ │ ├── base.py # Base generator interface  
│ │ └── openai\_embeddings.py # Implementation  
│ ├── indexers/ # Vector indexing  
│ │ ├── base.py # Base indexer interface  
│ │ └── chroma\_indexer.py # Implementation  
│ └── graphs/ # Knowledge graph generation  
│ ├── client.py # Neo4j client  
│ ├── generator.py # Graph generator  
│ └── schema.py # Schema manager  
├── services/ # Business logic services  
│ └── document\_service.py # Pipeline orchestration  
└── tasks/ # Celery tasks  
 └── document\_tasks.py # Asynchronous processing

### Pipeline Components

#### 1. Text Extraction (extractors/)

The DoclingExtractor implements the TextExtractor interface to extract text from various file formats: - Plain text (.txt): Simple file reading with encoding fallbacks - PDF (.pdf): Uses Docling with OCR capability, fallback to pdfminer - DOCX (.docx): Uses docx2txt library - Robust error handling with multiple fallback mechanisms

#### 2. Text Splitting (splitters/)

The LangchainSplitter implements the TextSplitter interface to divide long documents into manageable chunks: - Uses recursive character splitting - Configurable chunk size and overlap - Preserves semantic meaning where possible

#### 3. Embedding Generation (embeddings/)

The OpenAIEmbeddingGenerator implements the EmbeddingGenerator interface: - Uses OpenAI’s “text-embedding-ada-002” model - Creates high-dimensional vector representations of text chunks - Handles batch processing for efficiency

#### 4. Vector Indexing (indexers/)

The ChromaIndexer implements the VectorIndexer interface: - Stores document chunks with their embeddings - Attaches metadata for context - Enables semantic search capabilities

#### 5. Graph Generation (graphs/)

The knowledge graph module extracts structured information: - Neo4jClient: Handles database connection and graph operations - SchemaManager: Defines entity and relationship types - GraphGenerator: Extracts entities and relationships from text using LLMs

### Orchestration

The DocumentService in services/document\_service.py orchestrates the pipeline:

def process\_document(self, document):  
 # 1. Extract text  
 text = self.extractor.extract(document.file.path)  
   
 # 2. Split text into chunks  
 chunks = self.splitter.split(text)  
   
 # 3. Generate embeddings  
 embeddings = self.embedding\_generator.generate(chunks)  
   
 # 4. Index chunks and embeddings  
 metadata = {...} # Document metadata  
 self.indexer.index(chunks, embeddings, metadata)  
   
 # 5. Generate knowledge graph  
 graph\_result = self.graph\_generator.process\_document(...)  
   
 return {...} # Result information

### Asynchronous Processing

The process\_document\_task in tasks/document\_tasks.py handles asynchronous execution: - Runs the pipeline in a Celery worker - Manages document status updates - Includes error handling and retry mechanism - Prevents orphaned documents with cleanup\_processing\_documents task

### Error Handling

The pipeline implements robust error handling: - Each component has specific error handling for its domain - The document service catches and logs exceptions - The Celery task updates document status and stores error messages - A cleanup task finds and fixes documents stuck in processing

### Testing and Debugging

The system includes tools for testing and debugging the pipeline: - test\_pipeline management command for running the pipeline on a specific document - PipelineLogger for capturing detailed metrics and artifacts - Test dashboard for visualizing pipeline results and performance

## Additional Resources

* [Django Documentation](https://docs.djangoproject.com/)
* [Django REST Framework Documentation](https://www.django-rest-framework.org/)
* [Celery Documentation](https://docs.celeryq.dev/)
* [ChromaDB Documentation](https://docs.trychroma.com/)
* [Neo4j Python Driver Documentation](https://neo4j.com/docs/python-manual/current/)