# Comprehensive Development Documentation for RAGLens Document Processing Pipeline

## Project Overview

This document provides a detailed explanation of the RAGLens document processing pipeline implementation, focusing on the modular architecture designed for the hackathon project. The pipeline processes documents through extraction, splitting, embedding, indexing, and knowledge graph generation stages.

## Table of Contents

1. [Project Structure](#project-structure)
2. [Core Components](#core-components)
   * [Models](#models)
   * [Pipeline Components](#pipeline-components)
   * [Services](#services)
   * [Tasks](#tasks)
   * [API Layer](#api-layer)
3. [Graph Generation Module](#graph-generation-module)
4. [Testing Framework](#testing-framework)
5. [Development Guide](#development-guide)
6. [Original vs. New Implementation](#original-vs-new-implementation)
7. [Common Issues and Solutions](#common-issues-and-solutions)

## Project Structure

backend/  
├── config/ # Project configuration  
│ ├── \_\_init\_\_.py  
│ ├── asgi.py  
│ ├── settings.py # Django settings  
│ ├── urls.py # Main URL routing  
│ └── wsgi.py  
├── dochub/ # Document hub application  
│ ├── 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 # Testing command  
│ ├── migrations/ # Database migrations  
│ ├── models/ # Data models  
│ │ ├── \_\_init\_\_.py  
│ │ ├── document.py # Document model  
│ │ └── folder.py # Folder model  
│ ├── pipeline/ # Processing pipeline components  
│ │ ├── \_\_init\_\_.py  
│ │ ├── extractors/ # Text extraction  
│ │ │ ├── \_\_init\_\_.py  
│ │ │ ├── base.py # Base extractor interface  
│ │ │ └── docling\_extractor.py # Docling implementation  
│ │ ├── splitters/ # Text splitting  
│ │ │ ├── \_\_init\_\_.py  
│ │ │ ├── base.py # Base splitter interface  
│ │ │ └── langchain\_splitter.py# Langchain implementation  
│ │ ├── embeddings/ # Embedding generation  
│ │ │ ├── \_\_init\_\_.py  
│ │ │ ├── base.py # Base generator interface  
│ │ │ └── openai\_embeddings.py # OpenAI implementation  
│ │ ├── indexers/ # Vector indexing  
│ │ │ ├── \_\_init\_\_.py  
│ │ │ ├── base.py # Base indexer interface  
│ │ │ └── chroma\_indexer.py # ChromaDB implementation  
│ │ └── graphs/ # Knowledge graph generation  
│ │ ├── \_\_init\_\_.py  
│ │ ├── client.py # Neo4j client (from neo4j\_client.py)  
│ │ ├── generator.py # Graph generator (from graph\_generator.py)  
│ │ └── schema.py # Schema manager (from schema\_manager.py)  
│ ├── services/ # Business logic services  
│ │ ├── \_\_init\_\_.py  
│ │ ├── document\_service.py # Document processing orchestration  
│ │ └── search\_service.py # Document search functionality  
│ ├── signals.py # Signal handlers  
│ ├── tasks/ # Celery tasks  
│ │ ├── \_\_init\_\_.py  
│ │ └── document\_tasks.py # Document processing tasks  
│ ├── templates/ # HTML templates  
│ │ └── dochub/  
│ │ └── test\_dashboard.html # Pipeline test dashboard  
│ ├── utils/ # Utility functions  
│ │ ├── \_\_init\_\_.py  
│ │ ├── pipeline\_logger.py # Pipeline testing logger  
│ │ └── graph\_visualizer.py # Graph visualization utility  
│ ├── \_\_init\_\_.py  
│ ├── apps.py # App configuration  
│ └── views.py # Django views  
├── chatbot/ # Chatbot application (optional)  
├── utils/ # Shared utilities  
│ └── \_\_init\_\_.py  
├── media/ # Media files  
│ ├── Documents/ # Document storage  
│ └── pipeline\_tests/ # Pipeline test artifacts  
├── celery.py # Celery configuration  
├── manage.py # Django management script  
└── requirements.txt # Project dependencies

## Core Components

### Models

#### Document Model (dochub/models/document.py)

The Document model represents a file uploaded to the system:

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)

Key fields: - id: UUID for unique identification - name: Document name - file: Actual file storage - folder: Optional parent folder - status: Processing status (‘processing’, ‘ready’, ‘error’) - file\_type: MIME type or extension - size: File size in bytes

The document\_upload\_path function determines where files are stored, organizing them by folder structure.

#### Folder Model (dochub/models/folder.py)

The Folder model organizes documents in a hierarchical structure:

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)

Key fields: - id: UUID for unique identification - name: Folder name - parent: Optional parent folder (self-referential)

The build\_folder\_path function builds the file path for folders based on the hierarchy.

### Pipeline Components

The pipeline uses a modular architecture with clear interfaces for each component. This allows easy replacement of implementations.

#### Text Extraction (dochub/pipeline/extractors/)

Text extractors convert document files to plain text:

* base.py: Defines the TextExtractor interface
* docling\_extractor.py: Implements text extraction using Docling library

The DoclingExtractor extracts text from various file formats: - Plain text (.txt): Simple file reading - PDF (.pdf): Docling’s PDF processing with OCR - Other formats: Graceful fallback to simpler methods

#### Text Splitting (dochub/pipeline/splitters/)

Text splitters divide long documents into manageable chunks:

* base.py: Defines the TextSplitter interface
* langchain\_splitter.py: Implements text splitting using LangChain

The LangchainSplitter uses recursive character splitting with configurable chunk size and overlap.

#### Embedding Generation (dochub/pipeline/embeddings/)

Embedding generators create vector representations of text chunks:

* base.py: Defines the EmbeddingGenerator interface
* openai\_embeddings.py: Implements embedding generation using OpenAI’s API

The OpenAIEmbeddingGenerator uses models like “text-embedding-ada-002” to create high-dimensional vector representations.

#### Vector Indexing (dochub/pipeline/indexers/)

Vector indexers store and retrieve embeddings:

* base.py: Defines the VectorIndexer interface
* chroma\_indexer.py: Implements vector indexing using ChromaDB

The ChromaIndexer stores document chunks with their embeddings and metadata, enabling semantic search.

### Graph Generation Module (dochub/pipeline/graphs/)

This critical module creates knowledge graphs from document content:

#### Neo4j Client (dochub/pipeline/graphs/client.py)

The Neo4jClient class handles connections to the Neo4j graph database:

class Neo4jClient:  
 def \_\_init\_\_(self):  
 uri = settings.NEO4J\_URI  
 username = settings.NEO4J\_USERNAME  
 password = settings.NEO4J\_PASSWORD  
   
 self.driver = GraphDatabase.driver(uri, auth=(username, password))  
 self.\_initialize\_constraints()

Key functions: - create\_document\_node: Create a node for a document - create\_custom\_entity\_node: Create an entity node - create\_relationship\_safely: Create a relationship between entities - get\_document\_graph: Retrieve graph data for a document - get\_folder\_graph: Retrieve graph data for a folder

#### Graph Generator (dochub/pipeline/graphs/generator.py)

The GraphGenerator class extracts entities and relationships from document text:

class GraphGenerator:  
 def \_\_init\_\_(self, user\_id=None, schema\_data=None, llm\_provider="openai"):  
 self.neo4j\_client = Neo4jClient()  
 self.llm\_provider = llm\_provider  
   
 # Use provided schema or get from schema manager  
 self.schema\_manager = SchemaManager()  
   
 if schema\_data:  
 self.schema = schema\_data  
 elif user\_id:  
 self.schema = self.schema\_manager.get\_user\_schema(user\_id)  
 else:  
 self.schema = self.schema\_manager.get\_system\_default\_schema()

Key functions: - extract\_entities\_and\_relations\_openai: Extract entities and relationships using OpenAI - process\_document: Process a document to generate a knowledge graph - detect\_document\_type: Detect the type of document for better entity extraction

This class uses OpenAI’s models to extract structured information from unstructured text.

#### Schema Manager (dochub/pipeline/graphs/schema.py)

The SchemaManager class manages graph schemas for document processing:

class SchemaManager:  
 def \_\_init\_\_(self):  
 self.\_system\_default\_schema = self.\_load\_system\_default\_schema()  
   
 def get\_system\_default\_schema(self):  
 return self.\_system\_default\_schema  
   
 def get\_user\_schema(self, user\_id):  
 schema = GraphSchema.get\_user\_active\_schema(user\_id)  
   
 if schema:  
 return schema.schema\_data  
   
 return self.\_system\_default\_schema

Key functions: - \_load\_system\_default\_schema: Load the default schema - validate\_schema: Validate a schema’s structure - register\_missing\_relationship: Dynamically register a missing relationship type

The schema defines entity types (Person, Organization, etc.) and relationship types.

### Services

#### Document Service (dochub/services/document\_service.py)

The DocumentService orchestrates the document processing pipeline:

class DocumentService:  
 def \_\_init\_\_(self):  
 self.extractor = DoclingExtractor()  
 self.splitter = LangchainSplitter()  
 self.embedding\_generator = OpenAIEmbeddingGenerator()  
 self.indexer = ChromaIndexer()  
 self.graph\_generator = GraphGenerator()  
   
 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  
 metadata = {...} # Document metadata  
 self.indexer.index(chunks, embeddings, metadata)  
   
 # 5. Generate knowledge graph  
 graph\_result = self.graph\_generator.process\_document(...)  
   
 return {...} # Result information

This service handles the full pipeline from text extraction to graph generation.

### Tasks

#### Document Processing Task (dochub/tasks/document\_tasks.py)

Celery tasks for asynchronous document processing:

@shared\_task(bind=True, max\_retries=3)  
def process\_document\_task(self, document\_id):  
 try:  
 # Get document  
 with transaction.atomic():  
 document = Document.objects.select\_for\_update().get(id=document\_id)  
 document.status = 'processing'  
 document.save(update\_fields=['status'])  
   
 # Process document  
 service = DocumentService()  
 result = service.process\_document(document)  
   
 # Update status to ready  
 with transaction.atomic():  
 document = Document.objects.select\_for\_update().get(id=document\_id)  
 document.status = 'ready'  
 document.save(update\_fields=['status'])  
   
 return result  
   
 except Exception as e:  
 # Update status to error  
 with transaction.atomic():  
 document = Document.objects.select\_for\_update().get(id=document\_id)  
 document.status = 'error'  
 document.error\_message = str(e)  
 document.save(update\_fields=['status', 'error\_message'])  
   
 # Retry if appropriate  
 if self.request.retries < self.max\_retries:  
 raise self.retry(exc=e)  
   
 return {"error": str(e)}

Additional tasks: - cleanup\_processing\_documents: Fix documents stuck in processing state - reprocess\_failed\_document: Retry processing for failed documents - mock\_process\_document\_task: Simplified version for testing

### Signals

Signal handlers in dochub/signals.py automate document and folder operations:

@receiver(post\_save, sender=Document)  
def handle\_document\_post\_save(sender, instance, created, \*\*kwargs):  
 if created and instance.file:  
 # Update file metadata  
 instance.file\_type = ...  
 instance.size = ...  
   
 # Queue processing task  
 process\_document\_task.delay(str(instance.id))  
  
@receiver(post\_save, sender=Folder)  
def handle\_folder\_post\_save(sender, instance, created, \*\*kwargs):  
 if created:  
 # Create physical directory  
 path = instance.physical\_path  
 os.makedirs(path, exist\_ok=True)  
  
@receiver(pre\_delete, sender=Document)  
def handle\_document\_pre\_delete(sender, instance, \*\*kwargs):  
 if instance.file:  
 # Delete physical file  
 file\_path = instance.file.path  
 if os.path.exists(file\_path):  
 os.remove(file\_path)

These handlers ensure: - Document metadata is updated on creation - Processing tasks are queued automatically - Physical folders are created/deleted as needed - Physical files are deleted when documents are deleted

### API Layer

#### Serializers (dochub/api/serializers.py)

Serializers convert between Django models and JSON:

class DocumentSerializer(serializers.ModelSerializer):  
 url = serializers.SerializerMethodField()  
   
 class Meta:  
 model = Document  
 fields = ['id', 'name', 'file', 'url', 'folder', 'file\_type', 'size',   
 'status', 'error\_message', 'created\_at', 'updated\_at']  
 read\_only\_fields = ['file\_type', 'size', 'status', 'error\_message',   
 'created\_at', 'updated\_at']  
   
 def get\_url(self, obj):  
 if obj.file:  
 request = self.context.get('request')  
 if request:  
 return request.build\_absolute\_uri(obj.file.url)  
 return None

Key serializers: - FolderSerializer: For Folder model data - DocumentSerializer: For Document model data - BulkUploadSerializer: For uploading multiple files - BulkDeleteSerializer: For deleting multiple items

#### Views (dochub/api/views.py)

Views handle API requests:

class DocumentViewSet(viewsets.ModelViewSet):  
 serializer\_class = DocumentSerializer  
 parser\_classes = [parsers.MultiPartParser, parsers.FormParser]  
   
 def get\_queryset(self):  
 queryset = Document.objects.all()  
   
 # Filter by folder if specified  
 folder = self.request.query\_params.get('folder', None)  
 if folder:  
 if folder.lower() == 'null' or folder.lower() == 'root':  
 queryset = queryset.filter(folder\_\_isnull=True)  
 else:  
 queryset = queryset.filter(folder=folder)  
   
 return queryset  
   
 @action(detail=False, methods=['post'])  
 def bulk\_upload(self, request):  
 serializer = BulkUploadSerializer(data=request.data)  
 if serializer.is\_valid():  
 documents = serializer.save()  
   
 doc\_serializer = DocumentSerializer(  
 documents,   
 many=True,  
 context={'request': request}  
 )  
 return Response(  
 {"message": f"Successfully uploaded {len(documents)} files",   
 "documents": doc\_serializer.data},  
 status=status.HTTP\_201\_CREATED  
 )  
 return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)

Key views: - FolderViewSet: CRUD operations for folders - DocumentViewSet: CRUD operations for documents - BulkDeleteView: Bulk deletion of folders and documents - document\_graph, folder\_graph, entity\_graph: Graph data endpoints

#### URLs (dochub/api/urls.py)

URL patterns for API endpoints:

router = DefaultRouter()  
router.register('folders', FolderViewSet, basename='folder')  
router.register('documents', DocumentViewSet, basename='document')  
  
urlpatterns = [  
 path('', include(router.urls)),  
 path('bulk\_delete/', BulkDeleteView.as\_view(), name='bulk-delete'),  
 path('graph/document/<uuid:document\_id>/', document\_graph, name='document-graph'),  
 path('graph/folder/<uuid:folder\_id>/', folder\_graph, name='folder-graph'),  
 path('graph/entity/', entity\_graph, name='entity-graph'),  
 path('documents/status/<uuid:document\_id>/', document\_status, name='document-status'),  
 path('documents/status/', document\_status, name='documents-status'),  
]

## Testing Framework

### Pipeline Logger (dochub/utils/pipeline\_logger.py)

The PipelineLogger captures detailed information during pipeline testing:

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"""  
 def decorator(func):  
 @functools.wraps(func)  
 def wrapper(\*args, \*\*kwargs):  
 self.log(f"Starting step: {step\_name}")  
 start\_time = time.time()  
   
 try:  
 result = func(\*args, \*\*kwargs)  
   
 # Calculate step time  
 step\_time = time.time() - start\_time  
 self.step\_times[step\_name] = step\_time  
 self.log(f"Completed step: {step\_name} in {step\_time:.2f} seconds")  
   
 return result  
 except Exception as e:  
 self.log(f"Error in step {step\_name}: {str(e)}", logging.ERROR)  
 raise  
   
 return wrapper  
 return decorator

The logger provides methods to log and save: - Extracted text - Text chunks - Embeddings - OpenAI API requests and responses - Graph data

### Test Command (dochub/management/commands/test\_pipeline.py)

The test\_pipeline command runs the pipeline on a specific document:

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

Options: - --verbose: Enable verbose output - --save-artifacts: Save intermediate artifacts - --log-file: Specify a log file - --openai-model: Specify an OpenAI model for graph generation

### Dashboard (dochub/templates/dochub/test\_dashboard.html)

The test dashboard displays pipeline test results: - Document information - Performance metrics - Text extraction results - Chunk analysis - Embedding details - Knowledge graph visualization - OpenAI API calls and responses

## Development Guide

### Environment Setup

1. **Create a virtual environment**:

* python -m venv .venv  
  source .venv/bin/activate # On Windows: .venv\Scripts\activate

1. **Install dependencies**:

* pip install -r requirements.txt

1. **Set up environment variables**: Create a .env file with:

* # Django settings  
  SECRET\_KEY=your-secret-key  
  DEBUG=True  
    
  # Neo4j settings  
  NEO4J\_URI=bolt://localhost:7687  
  NEO4J\_USERNAME=neo4j  
  NEO4J\_PASSWORD=your-password  
    
  # OpenAI settings  
  OPENAI\_API\_KEY=your-openai-api-key  
    
  # Celery settings  
  CELERY\_BROKER\_URL=redis://localhost:6379/0  
  CELERY\_RESULT\_BACKEND=redis://localhost:6379/0

1. **Create necessary directories**:

* mkdir -p media/Documents  
  mkdir -p media/pipeline\_tests

1. **Apply migrations**:

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

### Testing the Pipeline

1. **Start required services**:

* # Start Redis for Celery  
  redis-server  
    
  # Start Neo4j  
  # (Use Neo4j Desktop or Docker)  
    
  # Start Celery worker  
  celery -A config worker --loglevel=info  
    
  # Start Django development server  
  python manage.py runserver

1. **Upload a document**:
   * Use the frontend interface or Django admin
   * Note the document ID for testing
2. **Run a basic test**:

* python manage.py test\_pipeline <document\_id>

1. **Run a detailed test with artifacts**:

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

1. **View the test dashboard**: Open http://localhost:8000/dochub/test-dashboard/<document\_id>/
2. **Test with different document types**:
   * PDF documents (text and scanned)
   * Word documents (.docx)
   * Plain text files (.txt)
   * Large documents

### Common Issues and Solutions

#### 1. OpenAI API Issues

**Problem**: API key errors or rate limits

**Solution**: - Check your API key is correctly set in .env - Consider using a different model (e.g., gpt-4o-mini instead of gpt-4o) - Implement retries with exponential backoff

#### 2. Neo4j Connection Issues

**Problem**: Cannot connect to Neo4j

**Solution**: - Ensure Neo4j is running - Check connection parameters in .env - Verify Neo4j allows connections from your application

#### 3. Document Processing Timeouts

**Problem**: Processing gets stuck

**Solution**: - Implement the cleanup\_processing\_documents task as a scheduled job - Add better error handling in each pipeline step - Implement progress tracking

#### 4. Text Extraction Issues

**Problem**: Poor text extraction quality

**Solution**: - Adjust OCR settings for PDFs - Implement file type-specific extraction methods - Add better error handling for corrupted files

#### 5. Large Documents

**Problem**: Memory issues with large documents

**Solution**: - Implement streaming processing - Process documents in smaller batches - Set reasonable limits on document size

## Original vs. New Implementation

### Original Codebase

The original codebase had:

1. **Complex architecture**:
   * Multiple apps (vault, cloud, assistant)
   * Authentication and authorization
   * Complex folder structure
2. **Neo4j graph generation**:
   * neo4j\_client.py: Neo4j database interaction
   * graph\_generator.py: Entity and relationship extraction
   * schema\_manager.py: Schema definition and management
3. **Processing pipeline**:
   * text\_extractor.py: Text extraction from documents
   * text\_splitter.py: Text chunking
   * embedding\_generator.py: Embedding generation
   * indexer.py: Vector database storage
4. **Celery integration**:
   * Asynchronous document processing
   * Background tasks

### New Implementation

The new implementation:

1. **Simplified architecture**:
   * Modular pipeline with clear interfaces
   * Removal of authentication (for hackathon simplicity)
   * Focus on document processing functionality
2. **Enhanced testability**:
   * Pipeline testing framework
   * Performance monitoring
   * Visual dashboard
3. **Improved error handling**:
   * Better logging
   * Graceful fallbacks
   * Status tracking
4. **Maintained core functionality**:
   * Full document processing pipeline
   * Knowledge graph generation
   * Vector search capabilities

### Key Migration Notes

1. **Graph Module**:
   * The graphs directory is migrated from original files
   * neo4j\_client.py → dochub/pipeline/graphs/client.py
   * graph\_generator.py → dochub/pipeline/graphs/generator.py
   * schema\_manager.py → dochub/pipeline/graphs/schema.py
2. **Processing Components**:
   * Each component now follows an interface-based design
   * Original implementations moved to concrete classes
3. **Settings**:
   * Environment variables for API keys and connection parameters
   * Configurable options for pipeline components
4. **Celery Tasks**:
   * Improved error handling
   * Retry mechanism
   * Status tracking

## Conclusion

This document processing pipeline provides a modular, extensible framework for: - Extracting text from various document formats - Splitting text into manageable chunks - Generating vector embeddings for semantic search - Building knowledge graphs from document content

The modular architecture allows for easy customization and extension, while the testing framework enables detailed performance analysis and optimization.

By following the development guide and testing procedures, you can ensure a robust implementation for your hackathon project.