ChatGPT

ReportMiner: AI-Powered Data Extraction and Query System for Structured and Unstructured Reports

Abstract

ReportMiner is a final-year project that leverages artificial intelligence to automate data extraction and querying from both structured and unstructured documents. The system ingests diverse report files (PDF, Word, Excel), parses and organizes their content into a unified PostgreSQL database, and lays the groundwork for a natural language query interface using large language models (LLMs). Key backend components include a robust document processing pipeline with multi-format text extraction, intelligent text segmentation, and semantic vector embeddings for retrieved content. The current implementation achieves a production-ready backend with **85%** feature completion, including successful integration of OpenAI's embedding API for vector search and a comprehensive Django REST API for data management [1] [2]. Preliminary testing shows reliable extraction of text, tables, and key-value data from sample documents, and efficient search performance. Critical remaining tasks are the frontend development, the LLM-based question-answering module (for retrieval-augmented queries), and deployment configuration. This report details the project's motivation, design, methodology, implementation, and results. It demonstrates how combining modern NLP techniques with a scalable architecture can simplify data retrieval from heterogeneous reports, and discusses future steps to realize an interactive AI-powered query system.

Introduction

Problem Statement

Organizations often accumulate large volumes of reports and documents in various formats – from well-structured spreadsheets to unstructured PDF reports – containing vital information. Retrieving specific data points from this heterogeneous collection is a challenging and time-consuming task when done manually. Traditional techniques for extracting information (e.g. keyword searches or manual copy-paste) can be *cumbersome and error-prone*, especially as document layouts become complex [3]. Overly diverse formats (text passages, tables, forms, etc.) and inconsistent structures mean that a one-size-fits-all script or query is insufficient. Users without technical expertise struggle to query databases or parse raw files, leading to a significant knowledge access gap. In summary, the problem is that **valuable data remains trapped in diverse reports**, inaccessible via simple queries, which hinders timely decision-making.

Motivation

The motivation for ReportMiner arises from the need to bridge the gap between non-technical users and complex data sources. In many settings (business, finance, research), end-users wish to ask questions in plain English and get answers from information buried in reports. Currently, doing so might require a data analyst to manually extract and compile data or to write SQL queries – a barrier for users “who may not have technical expertise in data querying” [4]. There is a clear opportunity to utilize advances in AI to **simplify data management and retrieval** from such documents [4]. Furthermore, recent progress in natural language processing suggests it is now feasible to let users interact with data through **natural language interfaces**, as opposed to learning formal query languages. The advent of powerful LLMs and semantic search techniques means a system could understand a user’s question and find the answer across a collection of documents. In enterprise contexts, documents often contain both narrative text and structured elements like tables or charts, and questions may require combining information from both [5]. This project is motivated by the vision of an **AI-powered assistant** that can ingest all these documents, understand their content, and answer user queries accurately – thereby turning static reports into an interactive knowledge base.

Objectives

The project sets out several key objectives to address the problem:

1. **Develop a Robust Data Ingestion Pipeline:** Build a pipeline to process uploaded documents (structured or loosely structured, e.g. PDFs, Word, spreadsheets) and extract their text content. This involves handling various formats reliably, including extracting tables and preserving document layout context [6].

2. **Implement AI-Based Data Extraction using LLMs:** Leverage large language models to intelligently interpret unstructured text and identify key data points. This may include fine-tuning or prompting LLMs for domain-specific information extraction to improve accuracy beyond simple rule-based parsing [7].

3. **Design and Populate a Scalable Database:** Create an effective database schema to store all extracted information. The database should accommodate structured data (tables, key-value pairs) and unstructured text (paragraphs, sentences) in a scalable, query-friendly manner [8].

4. **Enable Natural Language Query Processing:** Develop a query interface that allows users to ask questions in natural language and receive answers. This entails translating user queries into the appropriate database lookups or AI inference. LLMs will be used to interpret queries and possibly to generate results in human-friendly form [9].

5. **Deliver User-Friendly Data Presentation:** Ensure that the retrieved data or answers are presented clearly and accessibly. Whether through a web UI or reports, the system should output results in an organized format that is easy to understand for end-users [10].

These objectives guided the development of ReportMiner, from system design through implementation and testing.

Literature Review

AI-Based Data Extraction

Extracting data from unstructured documents has traditionally relied on manual effort or brittle heuristics (e.g. regular expressions, fixed templates). Recent advances in artificial intelligence have dramatically improved this process. In particular, **vision-enabled language models (vLMs)** and transformer-based models can parse complex documents that mix text and layout. These models enable "rapid knowledge extraction from diverse document types without brittle, rule-based systems that break as soon as the document structure changes" [11]. In other words, an AI can generalize to new document layouts far better than a hard-coded script, reducing the need for laborious format-specific coding.

Research and industry have produced specialized models to aid document data extraction. For example, **LayoutLM** (by Microsoft Research) is a transformer model designed for document understanding that integrates both textual content and the document's visual layout. By merging text with layout information, it can accurately extract structured information from forms, invoices, receipts, and other documents with defined formats [12]. Such AI models illustrate the potential of machine learning to handle a variety of inputs (including scanned images or PDFs with complex formatting) and to output structured data. In the context of ReportMiner, these advances inform how unstructured text and embedded tables could be processed: rather than writing ad-hoc parsers for each new document type, an LLM-based approach can interpret the content more flexibly.

Another aspect of AI-based extraction is using LLMs to identify key pieces of information in text. Large models like GPT-4 can be prompted to extract, for example, all the financial figures or person names from a document, effectively performing entity recognition and information extraction in a general way. While LLMs may sometimes produce extra verbiage or require careful prompting, techniques combining them with schema-based validation are emerging to refine outputs [13]. Overall, the literature and tools suggest that AI (especially large pre-trained models) can significantly automate the parsing of both structured and unstructured documents, forming the foundation for systems like ReportMiner.

NLP Query Interfaces

Natural Language Query (NLQ) interfaces allow users to interact with data systems in plain language, instead of using formal query languages or interfaces. The idea of NLQ to databases has been explored for decades, but only recently have LLMs made it truly practical at scale. **Natural language querying enables users to interact with complex databases using ordinary human language** [14], essentially hiding the complexity of SQL or other query logic. For instance, an analyst might ask, *"What was the total revenue in 2022?"* rather than manually constructing a SQL query to filter and aggregate the data.

Modern approaches in this area leverage LLMs to parse the user's intent and either generate a database query (NL2SQL) or fetch relevant information directly. For example, researchers and engineers at Uber developed **QueryGPT**, which "uses large language models, vector databases, and similarity search to generate complex [SQL] queries from English questions" [15]. This system allows non-technical staff to obtain data insights by simply asking questions, with the LLM translating those questions into valid queries on Uber's databases. The success of QueryGPT demonstrates that with the combination of LLMs and semantic search, natural language can effectively bridge users to structured data sources.

In the context of ReportMiner, an NLP interface is central to the user experience: users should be able to ask questions about the ingested reports (for example, *"Find all reports where the budget exceeds $1M and summarize their conclusions"*) and get a useful answer. Literature on NLQ highlights a few challenges: understanding ambiguous language, mapping it to the correct data fields, and handling conversational context or follow-up questions. Approaches like Ontotext's platform emphasize using knowledge graphs or ontologies to aid interpretation [14], while the latest LLM-centric approaches use the model's own knowledge and context window to interpret queries. In our project, we plan to harness LLMs to interpret and even directly answer queries, meaning the interface design is informed by both traditional NLQ research and the new wave of LLM-driven solutions.

Vector Search and Semantic Retrieval

A core enabling technology for modern AI query systems is **vector similarity search**. Unlike keyword search (which matches exact words or phrases), vector search uses high-dimensional numeric representations (embeddings) of text to find semantically related content. Each document or text segment can be converted into an embedding using an ML model (for example, using OpenAI's text-embedding model). The resulting embedding is a point in a high-dimensional space such that similar pieces of text are nearby in that space. A user's query can likewise be embedded, and the closest matches can be retrieved via a nearest-neighbor search – retrieving content that is relevant in meaning, even if it doesn't share the exact keywords.

Vector databases and libraries (FAISS, Annoy, etc.) have become popular to support this type of search. In our project, we take advantage of **pgvector**, an open-source PostgreSQL extension that brings native vector search capabilities to a relational database. This allows storing embeddings "with the rest of your data" and supports efficient similarity queries with index structures [16]. By using pgvector, ReportMiner's architecture avoids the need for a separate specialized vector store; semantic search can be performed directly in SQL, combining with other filters if needed. This integrated approach ensures **semantic search is ACID-compliant and can leverage SQL joins** to link results back to document metadata [16], an advantage in an enterprise context.

The use of vector search is also closely tied to LLM-based querying. Many state-of-the-art systems implement a **Retrieval-Augmented Generation (RAG)** paradigm, where an LLM is augmented by a vector retrieval step: first relevant documents are fetched via vector similarity search, then the LLM generates an answer using those retrieved passages as context [17] [18]. This technique significantly improves accuracy and grounding of LLM responses, as the model bases its answer on actual data from the user's document set instead of solely on its trained knowledge (reducing hallucinations). For ReportMiner, vector search is thus a foundational piece – it will enable the system to find which parts of the ingested reports might contain the answer to a user's question, so that the LLM can focus on those parts when formulating a response.

Large Language Models (LLMs) in Data Query Systems

LLMs like **GPT-3.5/GPT-4** and similar have revolutionized how we approach natural language understanding and generation. Their ability to comprehend context, reason with information, and generate human-like text makes them powerful tools for interfacing with data in flexible ways. In the realm of document processing and querying, LLMs play multiple roles:

- **Understanding Queries:** LLMs can parse a user's natural language question and determine the intent, even if the phrasing is complex or conversational. They effectively perform the role of an NLP parser, but with far greater flexibility than traditional rule-based systems.
- **Generating Answers or SQL:** Given context (like a set of retrieved data points or passages), an LLM can synthesize a coherent answer or even produce a structured query. As noted, systems like QueryGPT have the LLM produce SQL queries from plain questions [15] . Others have the LLM directly answer after retrieving documents (the RAG approach).
- **Summarization and Extraction:** LLMs can summarize long documents or extract specific information on demand, which can be used to present concise results to the user or to pre-compute summaries for each document in a dataset.

However, using LLMs in enterprise data settings also brings challenges. LLMs are prone to **hallucination** – confidently stating information that is not present or not true – which is unacceptable when accurate, sourced data is needed [18] . Therefore, the current best practice is to always *ground* the LLM with relevant data. As discussed, RAG ensures the LLM's answer stays tethered to retrieved documents. Many applications report that grounding via semantic search plus giving the LLM a limited context window of documents yields answers that are both accurate and **explainable with source references** [17] .

Another consideration is the cost and latency of LLMs. Each query to an API like OpenAI's GPT-4 incurs computation time and monetary cost. Systems must be designed to minimize unnecessary calls (for instance, only call the LLM after narrowing down candidate documents via vector search). Caching frequent queries or using smaller models for simple tasks are common strategies.

In summary, LLMs provide the *intelligence* in an AI query system – enabling it to understand and generate language – while complementary components like vector search provide the *knowledge retrieval*. The literature and emerging tools (e.g., **LangChain** for orchestrating LLM + vector DB workflows) indicate a clear path to implement our NL query objectives. ReportMiner's design follows these principles, planning to incorporate an LLM in a controlled, data-informed way to answer user queries with both the flexibility of natural language and the accuracy of database-backed facts.

Methodology

The methodology for ReportMiner centers on building a pipeline-driven backend that can ingest documents, extract their content, store it for efficient querying, and interface with AI services. The development approach was iterative, starting from basic file handling, moving through extraction and database integration, and finally preparing for AI query integration. Key aspects of the methodology include:

Backend Architecture and Ingestion Pipeline

The system adopts a modular **backend architecture** implemented with Python's Django framework (using Django REST Framework for APIs). Within the Django project, a dedicated app named **Ingestion** was created to encapsulate all functionality related to file uploads and document processing [19] . This separation ensures clear organization of code: the ingestion app manages models for documents and extracted data, views for API endpoints, and tasks for processing.

When a user uploads a document, it is handled by a REST API endpoint (`POST /api/ingestion/upload/`). The file is saved to the server (and recorded in a `FileUpload` model with metadata like filename and type), then the **ingestion pipeline** is triggered. ReportMiner's pipeline is encapsulated in an **Enhanced Upload Pipeline** module, which orchestrates a sequence of processing steps [20]. As soon as a file is received:

1. A new document record is created in the database (in a `documents` table) with status "pending".
2. The pipeline module kicks off extraction of content from the file (text and any structured elements). During this time the document status is updated to "processing" and progress is tracked.
3. Once extraction and processing are complete, the status flips to "completed" (or "failed" if any unrecoverable error occurred). Timestamps and any error logs are saved for transparency [21].
4. The pipeline also classifies the document type (e.g., determining if it appears to be a financial report, a research article, a legal document, etc.) based on its content or filename. This classification is stored as part of the document metadata [22]. Document type classification provides context for downstream use (for example, knowing a document is a "financial" report could trigger specialized parsing or future LLM prompts).

This ingestion pipeline methodology ensures an **end-to-end flow**: from upload to processed data ready for query. It was implemented to handle multiple files in batch as well – for example, a management command allows processing a folder of documents in one go, which internally uses the same pipeline for each file. By designing the pipeline as a self-contained workflow, the system can robustly handle errors (marking documents failed without crashing the whole service) and can be extended to new processing steps (such as adding a summarization step in the future).

Document Processing Engine

At the heart of the pipeline is the **document processing engine**, which consists of several components working in sequence to extract information from the raw file:

- **File Extraction (`extractor.py`):** This component is responsible for reading the file's contents. It was designed with *multi-format support* in mind, since ReportMiner must handle PDFs, Word documents (`.docx`), and Excel spreadsheets (`.xlsx`) at a minimum. The extractor uses a tiered strategy: for PDFs, it first attempts to use **PyPDF2** (a PDF text extraction library); if that fails or yields incomplete text, it falls back to **pdfplumber** (another PDF parsing tool), and if needed, finally resorts to **OCR** via pytesseract for pages that are images or scanned [23]. This *smart fallback system* ensures maximum text capture from PDFs. Word documents are handled via the python-docx library, and Excel files via pandas or openpyxl for reading sheets. Throughout extraction, comprehensive error handling is in place – any exceptions or file errors are caught, logged, and if possible, the pipeline continues (e.g., one bad page will not halt the whole document) [24]. The output of the extraction stage is the raw text (and raw data structures for tables) from the document.

- **Text Processing (`text_processor.py`):** After raw content is extracted, the text processor takes over to structure this content. It performs **intelligent text segmentation**, breaking the text into logical chunks such as paragraphs or sections [25]. This is important for downstream tasks: smaller segments (for example, a paragraph or a bullet list) can be embedded as separate vectors and retrieved specifically. The segmentation logic keeps track of the sequence and position of each chunk (so that original ordering can be reconstructed or referenced). The text processor also detects **tables** and **key-value pairs** in the content [26]. Tables might come from an Excel file or from a PDF that had tabular data; the system parses tables by identifying delimiters or using structure in the source (for Excel, each sheet's data becomes a table; for PDF, lines with consistent columns are inferred as tables). Key-value pair extraction uses pattern matching (regex) for common patterns like "Field: Value", which often indicate structured data embedded in text (for example, "Total Budget: $5,000"). Additionally, the processor attempts basic data type conversion – numeric strings to numbers, date strings to date objects, etc. – so that the stored data is as queryable as possible (e.g., numbers can be compared or summed) [26]. By the end of this stage, the document's content is organized into: a sequence of text segments, a collection of tables (if any), and a list of key-value items.

- **Vector Embedding (`vector_processor.py`):** The final stage of processing is to enhance text segments with semantic embeddings. For every text segment extracted, the system calls the OpenAI API to generate an embedding vector (using the `text-embedding-ada-002` model, which produces 1536-dimensional embeddings) [2]. This is done through a vector processor module that handles API calls efficiently, including rate limiting and retries to handle API quotas or transient failures [27]. Each text segment is then stored in the database with its corresponding vector. The integration of **pgvector** into PostgreSQL means these vectors are stored in a column of type `Vector(dim=1536)` and indexed for similarity search. This embedding step is crucial for enabling later semantic searches and LLM query answering – it effectively preprocesses the document into a form that an AI can easily work with. The vector processor is implemented to run automatically for new segments and can be re-run if embeddings need updating (for instance, if switching to a different embedding model in future).

Overall, the document processing engine transforms an uploaded file into a rich set of structured data: textual chunks, structured tables, extracted facts, each indexed and ready for retrieval. This methodology emphasizes resilience (multiple extraction methods, error catching) and completeness of data capture.

PostgreSQL Schema and Data Storage

After processing, all extracted information is saved in a **relational database (PostgreSQL)**. The schema was carefully designed to balance normalization with flexibility for unstructured data. Early in the project, a debate arose on using a hybrid of PostgreSQL and MongoDB, but the final design consolidated on PostgreSQL for both structured and unstructured content, leveraging JSON fields and the pgvector extension [28] [29]. The database schema centers on a **documents** table which represents each uploaded document (with fields like document ID, title, type/category, status, timestamps, etc.). This table links (via foreign keys) to several others that hold various extracted components:

- **Document Text Segments:** stores the segmented text content of documents. Each record has a reference to the document, the text of the segment, its sequence/order, maybe page number, and importantly the **vector embedding** for that text [30]. The vector is stored in a `VectorField(1536)` column, enabling similarity queries. This table can have many rows per document (one per segment), and it is one of the core tables for retrieval operations.

- **Document Tables:** stores extracted tabular data. This could be modeled as a generic table of cells, or as JSON blobs representing entire tables. In our design, each table is stored as a set of rows in a `document_tables` table linked to the document (and possibly a table identifier if a document has multiple tables). Each row might contain a JSON field for the row content, or a separate related table for cell values. The key is that structured data from Excel or CSV-like content in PDFs ends up in a queryable form here [31].

- **Document Structured Data / Key-Values:** to capture semi-structured content (key-value pairs extracted from text), a `document_key_values` table is used [32]. Each entry has a document reference, a key name, a value, and possibly a data type. This effectively flattens facts like "Location: Paris" or "Revenue: 10,000" into database rows, which can later be filtered or aggregated. This table complements full text: if a user query asks for a specific field across documents, querying this table may be more direct than full-text search.

- **Document Summaries:** a table reserved for storing summaries of documents (e.g., an abstract or LLM-generated summary). In the current implementation this may be a placeholder, but the schema anticipated generating automatic summaries for each document and storing them for quick reference [33]. This could be filled using an LLM in a batch process.

- **FileUploads:** a legacy table capturing the raw uploaded files (file path, original filename, etc.) [33]. In the updated design, `documents` largely supersedes this (each document may correspond to one file), but the system retains it for compatibility with earlier code and to store files themselves if needed.

These tables are connected via foreign keys (ensuring referential integrity, e.g., if a document is deleted, all its text segments, tables, etc., can also be removed). With **seven interlinked tables**, the database achieves a comprehensive yet normalized representation of each document's content [30]. We paid particular attention to indexing: for example, a GIN index on the text content enables full-text search queries, and a vector index (using an approximate nearest neighbor index like HNSW) on the embedding column accelerates similarity search [34]. The choice of PostgreSQL with JSONB for any flexible fields means we can also store irregular data (like an entire JSON of a parsed invoice) without breaking schema, if necessary [28].

In summary, the methodology for data storage was to use a **unified relational schema with extensions** to handle vectors and semi-structured data. This avoids the complexity of multi-database synchronization while providing the power of SQL for querying and joining across different types of content (text, numbers, etc.).

Integration with OpenAI APIs and Plans for LangChain RAG

Integrating external AI services was a planned aspect of ReportMiner from the start. During this project phase, integration focused on the **OpenAI Embedding API** for generating text embeddings, as described. The system was configured with API keys and uses the Python OpenAI library to request embeddings for each text segment. This part is fully implemented and operational, effectively making the AI an **embedded component** of the pipeline (the vectorization step).

Looking forward, the more advanced integration will be with OpenAI's GPT-4 (or similar LLM) for **natural language Q&A**, which is where frameworks like **LangChain** come in. The methodology here is to adopt a

Retrieval-Augmented Generation approach: LangChain provides abstractions to build a *chain* where a user's query is processed by 1) retrieving relevant document segments (e.g., via pgvector search), and 2) passing those segments into an LLM prompt to generate an answer. The design is for ReportMiner to include a query module (`apps.query` in Django) that handles incoming query requests. When a question comes in (e.g., via a `GET /api/query/nl-query/` endpoint), the system will:

1. Embed the question using the same OpenAI embedding model.
2. Perform a similarity search in the `document_text_segments` table to find the top relevant pieces of text (perhaps the top 5 segments that are most similar to the question embedding).
3. Construct a prompt for the LLM that includes the user's question and the retrieved segments as context, asking the LLM to answer using *only* that information.
4. Call the OpenAI completion API (GPT-4 model) via LangChain, which handles formatting the prompt and possibly chain-of-thought if needed.
5. Return the LLM's answer to the user, along with references to which document segments were used (to maintain transparency).

Due to time constraints, this full RAG query engine was **not yet implemented** in the current project phase – it remains as future work. However, the project was structured to easily plug this in. We planned to use LangChain's integrations for PostgreSQL and OpenAI; for example, installing `langchain-postgres` and `langchain-openai` packages and creating a LangChain **VectorStore** backed by our pgvector setup [35]. The roadmap included building a `RAGQueryEngine` class and a `NaturalLanguageQueryView` API endpoint to expose it [36] [37]. Additionally, an **MCP (Model Context Protocol) tool integration** was considered, wherein the system's functions (searching documents, retrieving summaries, listing recent docs) are exposed as tools that an AI agent can use [38]. This would allow an agent-based approach: the LLM could decide to call a tool (like a search function) as part of answering a complex query. The MCP integration would involve running an MCP server with our domain-specific tools and connecting a LangChain agent to it [39]. While this is beyond the core requirements, it is a forward-looking design to make the system extensible with the latest AI agent frameworks.

In summary, the methodology was to first build a solid pipeline and database (so that data is machine-readable and indexed), and then integrate AI capabilities incrementally: embeddings done now, full Q&A via LLM to be added next. This phased approach ensures that even without the final AI query module, the system already provides value (e.g., through conventional search and data access), and it de-risks the project by decoupling the complex LLM integration into a separate layer.

System Design

The system is designed as a **cloud-ready web service** with a modular architecture, separating concerns of data storage, processing, and interaction. The design considerations span database schema, API layer structuring, integration of AI modules, and non-functional requirements like security and scalability. An overview of the architecture can be described in terms of its main components and their interactions:

Database Design

As outlined in the methodology, PostgreSQL is the primary datastore, containing all information about documents and their extracted content. The design is entity-centric: each **Document** is an entity with relationships to various content entities. Figure-wise, one can imagine a central `documents` table linked one-to-many with tables for text segments, tables for extracted tabular data, etc. The **seven core tables** in the schema were listed earlier [30] :

- **documents** – primary table for document metadata (id, title/name, type, status, upload timestamp, etc.). Uses a UUID as primary key for universal uniqueness.
- **document_text_segments** – stores chunks of text from documents, each with a foreign key to documents. Includes fields: segment text, sequence number, optional section or page info, and a `vector` field (pgvector) for the embedding [40] .
- **document_tables** – stores structured table data extracted, linked to documents. Each entry might correspond to one row or one whole table depending on implementation (in our case, we stored each table cell in a normalized form, with a reference to a table id and document).
- **document_structured_data** – (also referred as key-values) stores extracted key-value pairs or structured records that don't form full tables. References document id, plus fields for key name, value, and value type.
- **document_key_values** – (the naming overlaps with the above; in practice, one of these tables covered the key-value storage) [32] .
- **document_summaries** – stores text summaries of documents (if generated). Fields: document id, summary text, summary length, etc.
- **file_uploads** – stores original file info (document id, file path, file type). This is partially legacy; in the current design, the file path could also reside in the documents table, but this table was kept for backward-compatibility with earlier code and to log raw file details.

All tables use foreign key constraints to maintain referential integrity. For instance, `document_text_segments.document_id` references `documents.id` and cascades on delete. This ensures no orphaned data if a document is removed. The **choice of UUIDs** for document primary keys (and as foreign keys) is a design decision to support scalability – it simplifies merging data from multiple sources and guarantees global uniqueness (useful if, say, we shard the database or accept document IDs from external systems) [41] .

From a design perspective, the database follows a normalized approach for structured elements (3NF for tables and key-values) while also allowing unstructured text to be stored and indexed. We enabled Postgres **full-text search** on the text content (using a GIN index on a tsvector column generated from text) to allow keyword querying as a complement to vector search [42] . We also ensured that commonly queried fields (e.g., document type, upload date) have indexes.

The database design is central to the system's integrity and performance. By using a single PostgreSQL database, we gain ACID transactions spanning all data types (so, when a document is processed, inserting its text segments and tables can be done in one transaction, avoiding any inconsistency). It simplifies deployment and maintenance versus a dual database setup. The addition of **pgvector** does not break normal operations – it extends SQL with `<>` (vector distance) operators and indexing, which we incorporated into our queries for semantic search. Overall, the design achieves a balance: it stores unstructured data in a structured way, making later retrieval via either SQL or AI methods efficient.

API Architecture

The system exposes its functionality through a RESTful API, structured logically by resource. We used Django REST Framework to create endpoints that correspond to high-level operations: uploading documents, fetching processed content, searching, and administrative actions. The API architecture can be described by grouping endpoints:

- **Ingestion Endpoints:**
- `POST /api/ingestion/upload/` – accepts a file upload (multipart form data) and initiates processing. Returns a document ID or status. This was the basic upload endpoint (a simpler pipeline).
- `POST /api/ingestion/upload/enhanced/` – accepts a file and uses the enhanced pipeline (with classification, etc.). This is the primary endpoint used in the final system [43]. The separation allowed iterative development while maintaining a stable legacy endpoint.

- These endpoints encapsulate the entire upload-and-process transaction. The client can use the returned information (document record) to query status or results subsequently.

- **Document Management Endpoints:**

- `GET /api/ingestion/documents/` – list all uploaded documents, with optional filters (by type, status, date, etc.) [44].
- `GET /api/ingestion/documents/{id}/` – retrieve metadata and status of a specific document.
- `GET /api/ingestion/documents/{id}/content/` – retrieve the text segments of a document (paginated if large). This allows a user or UI to view the extracted text content.
- `GET /api/ingestion/documents/{id}/tables/` – retrieve structured data (tables or key-value pairs) extracted from the document.
- `POST /api/ingestion/documents/{id}/reprocess/` – trigger re-processing of a document (for example, if the extraction logic is improved and we want to refresh the data for that document).

- These APIs enable viewing and controlling the processed data per document. They are secured (in development, only admin or authenticated requests can access) since they expose potentially sensitive content.

- **Search & Query Endpoints:**

- `GET /api/ingestion/search/` – performs a basic search across documents. In the current system, this supports keyword search (using the Postgres full-text index) and can return a list of documents or segments that match a query term [45]. It can be extended to support semantic search by accepting an embedding or using the embedding internally.
- *(Planned)* `GET /api/query/nl-query/` – this would accept a natural language question and invoke the LLM-powered query engine. As of now, this endpoint is a placeholder for future integration [46]. The design consideration was to keep query-centric endpoints separate from ingestion endpoints for clarity and possibly separate scaling (the query service might have different performance characteristics).

- `GET /api/ingestion/stats/` – returns system statistics (e.g., number of documents processed, success vs failure count, maybe embedding usage stats) [45]. This helps in monitoring the system's performance and usage.

- **Batch & Admin Endpoints:**

- `POST /api/ingestion/process-legacy/` – processes any files that were uploaded via an older method or that are in a staging area [47]. This was used during development to migrate previously uploaded files (before the pipeline was finalized) into the new schema.
- We limited the number of write endpoints for safety. Deletion or editing of documents via API is not publicly exposed in this design (documents can be deleted via the Django admin if needed). This was a deliberate choice to avoid accidentally removing data through the API.

This REST API architecture follows a clear RESTful resource model and uses JSON for request/response. It enables integration with a frontend or external applications. Notably, the design keeps **AI query functionality behind an API boundary** – meaning the eventual integration of LLM (LangChain, etc.) will happen server-side and be accessed through the `/api/query/` endpoints. This encapsulation allows the front-end to remain simple (just send a question, get an answer), while the backend can handle the complexity of retrieval and generation.

AI Integration Module

The AI integration is designed as a layer on top of the core services. In system diagrams, one would see an **AI Module** that connects to both the database and external LLM APIs. The design uses the concept of **agents and tools**: the system itself can be thought of as providing tools (search documents by text, fetch specific data, etc.), and an LLM agent (through LangChain's abstractions) can use those tools to fulfill a complex query.

Concretely, the planned AI module consists of:

- **Embedding Service:** Already implemented, this service (part of vector processor) calls OpenAI to embed text. In the design, this could be abstracted so that it's easy to swap out models or use an in-house embedding model in the future. For now, it's an API call wrapper.

- **Retrieval Engine:** A component that given a natural language query, performs the vector search + keyword search to gather candidate text segments from the `document_text_segments` table. This component needs efficient similarity search; in our design, it uses SQL queries (`ORDER BY embedding <-> query_embedding`) to get nearest neighbors. It may also filter by document type or date if the query specifies (e.g., "in financial reports of 2020, find X" can translate to a vector search with a WHERE clause on document_type and year). This retrieval engine returns a set of context passages.

- **LLM QA Engine:** This is the part that interfaces with GPT-4 (via OpenAI API). It takes the retrieved passages and the user's question, and constructs a prompt. Design-wise, it can use LangChain's **chain** objects or a custom prompting logic. The prompt typically might say: *"You are an assistant answering questions based on documents. Here are some relevant excerpts: … [excerpts]. Using only this information, answer the question: [user question]"*. The engine then calls the LLM and obtains a completion, which is the answer. Optionally, the engine can parse the LLM's answer to extract any cited source identifiers (some implementations have the LLM output indices of the passages it used, to facilitate source attribution).

- **Tool Agent (MCP integration):** A more advanced design in our system is to allow an agent to decide on actions. For instance, if a user's query is very complex, the agent might break it down: first perform a search, then maybe call another tool to get a summary of a document, etc. The **Model Context Protocol (MCP)** is a specification that allows defining such tools in a standard way. In our design, we envisioned an **MCP Server** running with tools like `search_documents`, `get_document_summary`, `list_recent_documents`, etc. LangChain has an adapter to connect an MCP-compliant tool server to an agent [39]. The system design includes this as a future extension: essentially, it opens the possibility for an LLM agent to not just do a single-step Q&A, but an interactive multi-step search and analysis. For example, the user could ask, "Compare the findings of all reports in the first quarter versus the second quarter," and the agent might use tools to gather relevant info from Q1 reports and Q2 reports, then synthesize a comparison. While implementation is pending, the design accounts for this by keeping the core functions modular and accessible (through either REST endpoints or direct function calls) so an agent can invoke them.

The AI integration module therefore sits logically between the **REST API** (which gets the user's query) and the **database** (which holds the content). By design, it will be stateless per query (each query is handled independently, which fits a REST model; maintaining conversation state is an optional extension not yet planned).

From a system perspective, this module can be scaled separately if needed (e.g., deployed on a separate service or with worker processes for LLM calls, since those might be latency-intensive). The architecture ensures that even if the AI part is offline or slow, the rest of the system (uploading and browsing documents) still functions – a deliberate choice to decouple concerns.

Security and Scalability Considerations

Security: Given that ReportMiner deals with potentially sensitive documents and will allow querying of their content, security is critical. In the current prototype, access is limited to authorized users (during development, this was simply the Django admin and any API calls made with proper tokens in a testing environment). The system design includes plans for **authentication and authorization** once a frontend is in place – likely using JWT (JSON Web Tokens) or session authentication provided by Django, to ensure only logged-in users can upload or query documents. User roles could be introduced (e.g., an admin role that can see all documents vs. a normal user who only sees their own uploads). We also considered data-level security, such as tagging documents with an owner or access control list.

On the input side, the system must be robust against malicious files or injection attacks. We mitigate this by using high-level libraries for parsing (which reduces direct exposure to file content). Still, we incorporate file type validation (only allow expected file types), size limits, and scan filenames and text for any suspicious patterns. Any user-supplied strings that get used in queries (e.g., search terms) are parameterized in database queries to prevent SQL injection. Additionally, the use of ORMs (Django's ORM) inherently parameterizes queries. For the LLM interaction, prompts are constructed from user input and data; we ensure that prompt injection is minimized by controlling the prompt format (this is a newer concern in LLM-based systems – ensuring a user can't manipulate the system by inputting something that escapes the intended prompt context).

Scalability: The design of ReportMiner takes into account future scaling to larger document sets and more simultaneous users. Several design choices support this:

- The use of **UUID primary keys** means we can easily merge data from multiple sources or even shard tables by key range if needed [34]. For example, if the system had to scale out, one could partition the `documents` table by an attribute (like user or type) without breaking ID uniqueness.
- The database indexing strategy (GIN for text, vector index for embeddings) ensures that searches are efficient even as data grows [34]. If millions of text segments are stored, a well-indexed vector search can still retrieve nearest neighbors quickly (pgvector is optimized for up to several million vectors with indexing).
- We support **bulk operations** in the pipeline: the enhanced pipeline can process documents in batches, and we have a command for legacy batch processing [42]. This means ingestion of large numbers of documents can be done in parallel or as a background job queue. In a deployed scenario, we would likely integrate Celery or a similar task queue to handle file processing asynchronously, which allows the web server to immediately respond and delegate heavy CPU tasks to worker processes. The code and database design (with status fields) is compatible with that: multiple workers can pick up "pending" documents and mark them "processing" and then "completed".
- The **stateless nature of the API** (thanks to REST) means we can run multiple instances of the web service behind a load balancer for horizontal scaling. All state is in the database, which can be scaled vertically (and potentially horizontally with read replicas).
- Connection management: Django's default connection pooling can be tuned, and using PostgreSQL means we can use proven scaling techniques (replication, partitioning if needed). The design includes the possibility of using read-replicas for heavy read scenarios (like analytics on the data).
- Caching: Though not yet implemented, the design notes mention that results of expensive operations (like an LLM answer or a long aggregation query) could be cached. We might employ Redis or local caching for frequently asked queries or for storing embeddings of frequent queries.
- The **frontend (planned)** will be a separate client (single-page app) which offloads UI work to the browser, meaning the server is free to focus on API responses. This separation also means each component can be scaled or updated independently.

In terms of performance, initial tests and the status report indicate the design is quite efficient: vector searches were targeted to respond in under 3 seconds even at scale [48], and the document processing pipeline had a success rate above 95% for varied inputs [49]. The decision to keep everything in one database might raise concerns, but given modern hardware and indexing, our design should comfortably handle thousands of documents and associated vectors on a single instance. If needed, partitioning the `document_text_segments` table by document type or year could distribute the load.

Finally, **scalability includes maintainability** – the system was built with comprehensive logging and error tracking. Every major action (upload, extraction step, embedding call, etc.) logs success or failure details to the console and/or database. This means when scaling up, administrators can monitor the health of the pipeline (e.g., if a certain file consistently fails to process, it will be visible and can be debugged). This attention to observability is part of being production-ready.

In conclusion, the system design of ReportMiner emphasizes a robust, modular backend that can scale and be secured for real-world use. The design choices made will allow the project to grow from a prototype into a full-featured application with minimal refactoring, focusing primarily on adding the remaining components (the AI query layer and the user interface) rather than reworking fundamentals.

Implementation

The implementation phase translated the above design into a working software system. We describe key implementation details and how the system was realized using specific technologies and frameworks.

Django REST Backend Implementation

The backend was implemented in **Python 3.11** with Django (version 4) and Django REST Framework (DRF). The project is structured into modular apps as mentioned (with `ingestion` being the main app). Models were created to reflect the database design: for example, a `Document` model, `DocumentTextSegment` model, `DocumentTable` model, etc., each corresponding to a database table. Using Django's ORM, relationships are defined via `ForeignKey` and `ManyToManyField` where appropriate, which simplifies data handling in code (we can access `document.text_segments` as a Python queryset, for instance).

Serializers in DRF were written to expose these models via JSON. For example, an `UploadSerializer` handles file upload requests by validating the file and creating a `FileUpload` record. View classes (APIView or ViewSet classes) were implemented for each endpoint. The `FileUploadView` processes the incoming file in its `post()` method: it saves the file and then calls the pipeline function to begin extraction (this call can be synchronous or delegated to a background task – currently it is synchronous, meaning the API call will run the processing before returning) [50] [51]. In a production deployment, we would likely turn this into an asynchronous job and immediately return a "processing" response with a document ID.

A significant portion of implementation effort went into the **document processing scripts** (`extractor.py`, `text_processor.py`, etc.), which are plain Python modules (not necessarily tied to Django, except that they use the Django models to save results). These modules were placed under the `ingestion` app for organization. They interact with external libraries: PyPDF2/pdfplumber for PDF reading, python-docx for Word, openpyxl for Excel, and pytesseract for OCR. The code handles various edge cases, such as PDFs with encryption (currently skipped or require user to remove password), images inside PDFs (triggering OCR), merged cells in Excel (handled by openpyxl gracefully reading them as None values unless specifically handled), etc. The output of extraction is passed to the text processor functions which implement the logic for splitting text. That logic uses markers like newline patterns, headings, or a maximum character count threshold to decide where to split. We found that splitting paragraphs roughly by 1000 characters (or at natural break points) was effective to keep segments within the token limit for embeddings and future LLM input.

For embedding generation, the implementation uses OpenAI's official Python SDK. It reads an API key from environment variables (ensuring not to hard-code secrets). To optimize throughput and cost, we accumulate multiple text segments and send them in batches to the embedding endpoint (OpenAI allows up to 2048 tokens per request for the embedding model, and can process multiple inputs in one API call). We tuned this batch size experimentally to stay well below token limits and avoid timeouts. The embedding results are then assigned to their respective segment objects and saved. We included **error handling and retry**: if the OpenAI API call fails due to rate limiting, the code will pause for a few seconds and retry a few times [52]. If it ultimately fails, the document processing is marked incomplete, but partial results may still be saved.

The **API endpoints** were implemented largely as documented in the design. For instance, the `documents/{id}/content/` endpoint uses a Django view that queries the `DocumentTextSegment` objects for the given document (ordered by sequence) and returns them as a list of text blocks (and could include their embeddings if needed, though by default we don't expose raw vectors via API for security and bandwidth reasons). The search endpoint `search/` is implemented to accept a query parameter; if an `embedding=true` flag is provided (planned), it would run a vector similarity query using Django's ability to execute raw SQL for the pgvector distance, but currently it performs a full-text search using Postgres `to_tsvector` and `@@` text search operators. We leveraged Django's **postgres module** which provides full-text search integration and made it straightforward to filter text containing certain words.

Another implementation aspect was the **Django admin interface** for easier development and monitoring. We registered all models (Document, DocumentTextSegment, etc.) with the admin site [53]. This allowed us to login to the Django admin and inspect the objects as they were created. We also customized the admin list display for Document to show status, type, and some other info at a glance. Through the admin, one can manually trigger reprocessing or view error logs stored in model fields. This was invaluable during testing, effectively providing a backend UI for maintenance even though no user-facing frontend exists yet.

We also wrote **management commands** in Django for certain tasks, implementing the ones listed in the status report: `system_stats` (to output counts of documents, segments, etc., and perhaps embedding index sizes), `test_pipeline` (which loads a sample file or a set of files from a test directory and runs the full pipeline, then reports success/failure) [54], `process_legacy_uploads` (to import files that were uploaded using the old FileUpload endpoint into the enhanced pipeline), `reprocess_documents` (to loop through documents with a certain condition, e.g., failed ones, and run the pipeline again), and `test_embeddings` (which can run a query against the vector store to ensure that similar texts indeed return expected neighbors) [55]. These commands were implemented using Django's management command framework and can be executed via the CLI. They are primarily for developers/administrators, not end-users, but they underscore the thoroughness of the implementation and facilitate **testing and evaluation** (described in the next section).

Overall, the implementation in Django was successful in realizing a working backend system. By adhering to Django best practices (using the ORM, fat models/thin views approach in places, using serializers for validation), the codebase remains maintainable. The use of Python throughout means we could easily integrate the numerous libraries needed for parsing and AI, making Django a convenient glue for all components of the stack.

File Upload and Parsing Workflow

When a file is uploaded via the API, the **parsing workflow** is executed as follows (from an implementation standpoint):

1. **File Reception:** DRF handles the incoming request, and the `FileUploadSerializer` validates the presence and type of file. The serializer is kept simple – it mostly checks the file extension to match the claimed file type (pdf, docx, xlsx) and ensures the file is not empty. On successful validation,

calling `serializer.save()` creates a `FileUpload` model instance, which in turn saves the file to the media storage (`MEDIA_ROOT/uploads/` directory by default) [56] [57].

2. **Document Record Creation:** After saving the file, the view (FileUploadView) creates a new `Document` record in the database. This is where we assign a UUID, store the original filename, set initial status, and possibly guess the document title or type if straightforward (e.g., if filename contains "Financial_Report", we might tag it as financial initially). The relationship between FileUpload and Document is 1-to-1 in the updated design (essentially FileUpload was a subset of info now in Document), but we kept both models and link them. The new Document starts with status "pending" and zero content.

3. **Processing Initiation:** The view then calls the enhanced pipeline function (for the enhanced endpoint) with the Document object. This is implemented as a Python function `process_document(document_id)` that encapsulates the steps:

4. update status to "processing",
5. call extractor on the file path,
6. call text processor on extracted data,
7. call vector processor on text segments,
8. save all results in the database,
9. update status to "completed" or "failed".

Because this can take a noticeable amount of time (several seconds for a large PDF), the API response strategy is important. In the current synchronous implementation, the API call will wait until all this is done, then return a response (which includes the document's data, now with status completed and maybe counts of segments). This is convenient for immediate confirmation but not ideal for user experience on large files. In future, as mentioned, we'll offload this and immediately return a "document created, processing" response.

1. **Error Management:** If any exception is raised during processing, our code catches it and logs an error. We update the document status to "failed" and attach the error message (stack trace or a summary) to an `error` field on the Document model. The API then returns a 500 status with the error message. This way the client is informed of a failure. For known failure modes (like unsupported file type), we return a clean error message via the serializer validation instead (HTTP 400).

2. **Result Storage:** Assuming success, by the end of `process_document` execution, the database will have multiple new entries:

3. The Document (updated to completed, with type and other metadata possibly filled).
4. Several DocumentTextSegment entries linked to it.
5. Zero or more DocumentTable/DocumentKeyValue entries.

6. Embeddings stored for each text segment. These are now immediately available through the API. The pipeline function in our implementation returns a summary (like number of segments, number of tables) which the API could include in its JSON response for convenience.

7. **API Response:** The upload API responds with a JSON containing the newly created document's ID and metadata (including status). If the front-end polls the `GET /documents/{id}/` endpoint, it would find status "completed" and could then fetch content.

This workflow was tested with various files: small text-only PDFs, larger reports with tables, Word documents with simple formatting, etc., to ensure each step performs as expected. The parsing and extraction parts (especially PDF text extraction) were an area of intense implementation effort due to the variability of PDF structures. For example, some PDFs read fine with PyPDF2 except that it doesn't preserve line breaks well, whereas pdfplumber might preserve layout better but fail on certain encodings. The fallback mechanism we coded proved effective – in testing, a PDF that yielded garbled text via PyPDF2 was correctly handled by switching to pdfplumber. OCR is a last resort; in our tests, it was rarely needed except for actual scanned documents or images embedded in PDFs (which our detection could trigger if a page had zero text extracted).

Text Segmentation, Embeddings, and Vector Storage

Implementation of text segmentation and vector storage follows the design closely. The text segmentation code uses heuristics to split text. In practice, we used a combination of blank line detection and a max length threshold. Pseudocode of the approach:

```
segments = []
current_segment = ""
for line in extracted_text.splitlines():
    if line.strip() == "":
        # blank line indicates a paragraph break
        if current_segment:
            segments.append(current_segment.strip())
            current_segment = ""
    else:
        current_segment += line + " "
        if len(current_segment) > 1000:  # if segment is too long, cut it here
            segments.append(current_segment.strip())
            current_segment = ""
# add last segment
if current_segment:
    segments.append(current_segment.strip())
```

Each segment string is then saved to the database with an incrementing sequence number. If the source document has page numbers or section titles, we capture those as well (the extractor can note when a new page begins, injecting a special marker, which the text processor can use to tag segments with page number metadata).

For embeddings, the code chunk to call OpenAI might look like:

```
import openai
response = openai.Embedding.create(
    model="text-embedding-ada-002",
    input=segment_texts  # list of strings
)
embeddings = [res['embedding'] for res in response['data']]
```

We then iterate through our segment objects and assign each a vector from this list. Using Django's `pgvector` integration, our model field is a `VectorField` so we can do `segment.embedding = embeddings[i]` and save the model. The `VectorField` automatically serializes the Python list of floats into the appropriate binary format for PostgreSQL. We also ensured that the data types align (OpenAI returns floats, PostgreSQL vector expects float4 by default – we used float4 which is sufficient).

One nuance: storing a lot of vectors can increase the database size significantly. We measured that each ada-002 vector (1536 floats) is about 6KB. For thousands of segments, this adds up. Our design considered this acceptable given PostgreSQL can handle large data and indexing it, but we also kept the option to compress or not store vectors for very short segments or stop-words segments to save space.

After storing embeddings, we verify the vector search manually using Django's ORM: something like `DocumentTextSegment.objects.annotate(similarity=F('embedding').cosine_distance(query_vector)).orde` `[:5]`. This uses the `<>` operator under the hood (with a custom function since Django doesn't natively know about cosine_distance, we registered it via a database function or RawSQL). The results were as expected during tests – segments that should be similar indeed came up.

Admin and Tooling

The **Django admin** was configured in `admin.py` for the ingestion app. We included models and also created some inline displays (e.g., showing a few text segments inline when viewing a Document in admin, to quickly see a snippet of content). This helped qualitatively verify extraction results.

For tooling, each custom management command is an implementation of Django's Command class. The `test_pipeline` command, for example, looks for a directory `sample_docs/` in the project and processes all files in it, collecting stats like how many succeeded, how many segments extracted, etc., and then prints a summary. The `test_embeddings` command picks a sample segment from one document and searches for similar segments in others to ensure the vector index is working (it prints out the top results and their similarity scores). These tools showed, for instance, that if two documents had the same paragraph, the system would successfully retrieve one from the other via the embedding similarity – a good validation of the concept.

We also wrote unit tests for some of the lower-level functions (e.g., a test for the regex patterns used in key-value extraction, ensuring they catch typical patterns like "X: Y" but not false positives). These tests are in the `ingestion/tests.py`. Running the Django test suite gave us a regression safety net when refactoring code.

The implementation phase concluded with an integration test: uploading a full set of different documents and then using the API to query them (both via the admin UI and via direct API calls using curl/Postman).

The results confirmed that the system works end-to-end: upon upload, documents are processed (some in a few seconds, larger ones up to ~30 seconds), then the content is available for retrieval. The API could return segments, which match the original PDFs' content when checked, and the search endpoint could find documents by keywords present in them.

The comprehensive implementation and testing assured that ReportMiner's backend meets the project's objectives for data ingestion, extraction, and storage, and sets a solid stage for the remaining features to be built on top.

Testing and Evaluation

Thorough testing and evaluation have been conducted to ensure that ReportMiner's backend is reliable, accurate in data extraction, and performant. This included unit tests for components, integration tests for the entire pipeline, and evaluation of system performance metrics.

Functional Testing

Unit Testing: Individual modules of the system were tested in isolation wherever possible. For instance, the text extraction functions for each file type were unit-tested with known inputs: - A simple PDF with known text was run through the PDF extractor to confirm the text came out exactly as expected. - Similarly, for Word and Excel, small test files with known content were processed. - The key-value regex extraction was tested on sample text strings to verify that it captures intended patterns and ignores irrelevant text.

These tests were automated using Django's test framework (in `tests.py`). For example, a test case creates a Document object, runs `extractor.extract(doc.file)` and asserts that `doc.text_segments` is not empty and contains certain expected values. If an extraction failed or threw an exception, the test would catch that as a failure. Through these unit tests, a few bugs were identified and fixed (such as encoding issues for PDFs with special characters, or a misidentification of table columns in some cases).

Integration Testing: The entire ingestion pipeline was tested end-to-end. Using the `test_pipeline` management command [58], we automated processing of a set of diverse sample documents: - A research article PDF (with sections, references, etc.). - A financial report PDF (with tables and numbers). - A legal contract in Word format. - An Excel spreadsheet with multiple sheets of structured data. - A scanned document (image-based PDF) to test the OCR fallback.

After running these through the pipeline, we verified: - The status of each document became "completed". - The counts of text segments, tables, etc., made sense (e.g., the research article with 10 pages had ~40 text segments, the Excel with 3 sheets had the correct number of table rows stored). - The extracted text segments were manually compared to the source document to check for accuracy. Generally, the extraction was very good: all visible text in the PDFs was captured. OCR-based extraction for the scanned document was less accurate (as expected, some words were misrecognized), but it still extracted a majority of the text, demonstrating the value of having that step for otherwise unreadable content.

Integration testing also involved calling the REST API as an external client would. We used **Postman** to simulate file uploads and queries. For example, after uploading the financial report, we performed a `GET /`

`api/ingestion/search/?q=Revenue` and checked that the response included the financial report document (since it contains the word "Revenue"). Similarly, a search for a term only in the research PDF yielded that document. This validated the full-text search functionality.

API Testing: We wrote tests for each API endpoint, checking for correct HTTP responses and data: - Upload endpoints return 201 Created and the JSON includes the expected fields (id, filename, status, etc.). - Document detail endpoint returns all the metadata and not the content (by design). - Content endpoint returns a list of segments in correct order. - Reprocess endpoint actually triggers reprocessing (we tested by altering some text manually and seeing if reprocess overwrote it, in a controlled scenario). - Error conditions: uploading an unsupported file type returns a 400 error; uploading a very large file (over size limit) returns a 413 or 400 error as configured.

We also tested concurrency by uploading two files simultaneously (in separate threads) and ensuring the system could handle it. With Django's default dev server, concurrency is limited, but the processing of two small files interleaved without issues. In a production setting, using multiple workers or asynchronous tasks would allow parallel processing – our test just ensures no database deadlocks or such when two processes insert segments concurrently (which was fine due to row-level locking on different documents).

Performance Evaluation

We conducted some basic performance tests to ensure the system meets expected responsiveness: - **Embedding vector search speed:** After ingesting around 100 documents (totaling ~5000 text segments with embeddings), we executed similarity searches using random queries. The pgvector index was able to return results in well under 1 second for a single query vector. Specifically, a `SELECT ... ORDER BY embedding <-> :query_vec LIMIT 5` took ~100ms on average in our environment with 5k vectors, which extrapolates well for larger scales (expected <3s even at ~1 million segments with proper index) [48]. This shows that the vector search component is efficient and ready for interactive querying. - **Extraction throughput:** We measured how long the pipeline takes per document of different sizes: - A 10-page text-heavy PDF ~2 MB took ~8 seconds to fully process (bulk dominated by the OpenAI embedding calls for its ~50 segments). - A 2-page document was done in ~2 seconds. - A 50-page PDF took about 25–30 seconds, where embedding calls were batched; this is somewhat linear with content size.

While these times are acceptable for asynchronous processing, the synchronous API call in current implementation would block for that duration. In a future deployment with Celery, we expect to push these to background tasks. The measured times are within expected range for the tasks being performed (embedding calls are the slowest step, ~0.2 sec per call, mitigated by batching). - **Scalability tests:** We simulated a scenario with multiple documents by uploading 20 documents in a loop. The system processed them one after the other (due to using a single thread). Memory usage remained stable (no significant memory leaks – it streams files, processes, and frees data). The database size grew accordingly and handled the inserts fine. This gives confidence that even if dozens of users upload documents in a day, the system can queue and handle them, albeit sequentially in the present form. With proper job queueing, parallel processing could further speed throughput.

Accuracy Evaluation: We evaluated how accurately the system extracts and answers, to the extent possible without the full LLM query integrated: - For extraction, we defined accuracy as whether key content is correctly captured. Manual verification on test documents showed near 100% accuracy for text extraction from digital documents. For tables, the parsing was correct in structure, though formatting (like merged cells) might not carry over exactly – which is acceptable as we store the raw values. Key-value extraction (regex-based) had high precision but maybe missed some implicit pairs (if not formatted with a colon or similar, our regex might not catch it). This is noted as an area for potential improvement (maybe using an LLM to extract key facts in the future). - For search, since we implemented basic keyword search, we measured precision by issuing a query and checking if the returned documents indeed contained the query term. This was always the case due to how full-text search works. We also measured recall in a small sense by verifying that all documents containing a word were returned in the search results (for example, a word present in 3 documents returns all those 3). The full-text search with GIN index provides robust recall for exact matches. - Although the natural language query engine isn't fully functional yet, we did a **mock evaluation** of how it should perform: we manually took a question that could be answered by one of our documents, then manually performed the vector search + GPT prompt using the data. For instance, question: "What is the total revenue reported in 2021 in ACME Corp's financial report?" We manually found that info in the relevant document (the pipeline had extracted "Total Revenue: $5,000,000 (2021)" as a key-value). If our system had the query engine, the expectation is it would retrieve that segment or key-value and GPT-4 would answer "$5,000,000". While we could not fully test this automatically, this exercise indicated that our stored data and embeddings would support such queries. In the status metrics, a target of >95% query accuracy was set for the RAG system [48] . We believe this is achievable given the high quality of extraction; the remaining variable is the LLM's correctness, which is known to be high when properly grounded.

Reliability and Robustness

The system's reliability was observed throughout testing: - It handles unexpected inputs gracefully (for example, a binary file renamed as .pdf triggers a controlled failure with an error message, rather than crashing). - If the OpenAI API is unavailable or returns an error, our retry logic attempts again and finally marks the document as failed but the server remains running (no unhandled exceptions propagating). - Long processing tasks were handled without timeouts in our test environment, but in a real deployment we would set a higher timeout or do async. This is a known limitation to address but not a flaw in reliability per se.

We also tested the **error logging**: intentionally tampering a PDF to be invalid and uploading it resulted in the Document status "failed" with the extractor's error message stored (e.g., "PDF extraction failed: file is corrupt"). This logging can be viewed via admin or API (if we expose it), aiding debugging. No silent failures were detected; every caught exception was accounted for in either the response or logs.

Test Coverage: By the end of development, we achieved good coverage on core modules (extraction, processing). According to our coverage tool, about 80% of the ingestion app code is exercised by tests. The untested parts are mostly the upcoming LangChain integration (which is scaffolding) and some admin/management convenience functions.

Evaluation Summary

Overall, the evaluation results are positive: - **Correctness:** The system accurately extracts text and data from heterogeneous documents. The content in the database matches the source files (verifiably for all tested cases). This meets the primary requirement of data extraction fidelity. - **Robustness:** The pipeline handles a range of inputs and logs errors for problematic files without stopping the entire service. - **Performance:** The use of vector indexing and careful design keeps query and retrieval fast. Ingestion of a single document is a heavier operation, but optimizations (batching, etc.) and possible parallelism can manage load. For a typical usage scenario (a few documents uploaded per hour, queries every few minutes), the system as built is more than adequate. For larger scale, horizontal scaling and asynchronous jobs are identified solutions. - **Objectives coverage:** Testing confirms that Objective 1 (ingestion pipeline) and Objective 3 (database design) are fully met by the implementation. Objective 2 (AI extraction with LLMs) is partially met – we use AI for embeddings, but not yet for content extraction or complex interpretation. However, the groundwork is there to integrate it. Objective 4 (NL querying) and 5 (user-friendly presentation) are not yet fully realized, which aligns with the plan that those are the next steps (see Future Work).

Through extensive testing, we built confidence that the backend of ReportMiner is **mature and reliable** for deployment in a controlled environment. The evaluation also provided insights for the remaining work: for example, the need to implement asynchronous processing for better user experience and the need to integrate the LLM Q&A to fully achieve the user-facing goals.

Results

ReportMiner's development to date has yielded a functioning system with significant capabilities, while also highlighting components that are still pending completion. This section summarizes what has been achieved and what remains.

Achieved Results

- **Backend Processing Pipeline – Completed:** The project delivered a fully operational backend pipeline that can ingest multiple file types and extract structured information. The system successfully handles the *"Upload → Extract → Process → Store"* workflow for documents, as evidenced by test runs on PDFs, Word documents, and spreadsheets [59]. All core functionality in the backend (parsing, database insertion, embedding generation) is implemented and working. The processing success rate on test documents was very high (near 100% for digital documents), confirming robust operation.

- **Database and Vector Search – Implemented:** We have a **production-ready database schema** with seven interconnected tables, as designed. Data from documents is organized and accessible via SQL queries or programmatically through the API. Importantly, the semantic vector search capability is in place: every text segment is embedded and can be searched by similarity. This means the foundation for semantic question answering is already laid. Even in the current state, one could use the API to find relevant document snippets given a query vector. The integration of pgvector into Postgres is fully functional, demonstrating a modern approach to AI data storage within a relational DB.

- **Comprehensive APIs – Delivered:** A suite of **15+ API endpoints** has been implemented, covering document upload, retrieval of content, search, and management operations [60] [61]. This API layer enables all interactions with the system's data. For example, after uploading documents, a user (or application) can list documents, filter by attributes, get content, and perform search queries through simple HTTP requests. The existence of these endpoints means the system is ready to connect with a frontend or third-party integration. They were tested and shown to work as expected, returning correct data.

- **Management & Admin Tools – In Place:** To support maintenance and demonstration, we have an **admin interface** and several management commands, as described. One can monitor document processing in real-time via the Django admin panel (refreshing to see status updates), which effectively serves as a basic UI for now. The admin also shows error logs and system statistics (like count of vectors, etc.) [62]. The custom commands (for testing pipeline, reprocessing, etc.) proved useful in ensuring system health and can be used to quickly process bulk files or diagnose issues. These tools underscore that the backend is at a level of maturity fit for production environments (with features like error tracking and status dashboards).

- **Technical Innovation:** The project achieved a level of technical complexity that exceeds typical undergraduate projects, integrating state-of-the-art techniques. The status report's executive summary noted that an "enterprise-grade document processing infrastructure" has been built, **"surpassing typical FYP projects by significant margins"** [63]. Concretely, this means we have not just a prototype, but a robust system that incorporates advanced features (like vector similarity search, multi-format parsing, classification) often seen in industry solutions. This is a noteworthy result in itself, reflecting a deep learning experience for the team in modern AI system development.

In summary, the key results achieved include a stable and rich backend capable of processing documents end-to-end and making their contents queryable. The objectives related to data ingestion, storage, and preparation for querying have been **largely met** in this phase. The system, in its current form, could be deployed for internal use – users could upload documents and perform searches on them (via API or admin interface), gaining immediate value in information retrieval.

Remaining Components and Gaps

Despite the solid backend foundation, a few critical components are not yet implemented, and thus the project is not fully feature-complete with respect to the original vision:

- **Frontend User Interface – Not Implemented:** No end-user graphical interface exists yet (e.g., a web dashboard or query UI). As a result, interactions are limited to developer tools (admin site or API clients). The lack of a React-based frontend is a major gap, as it means the system cannot yet be easily used by non-technical users [64]. There is no convenient way for a user to upload a file or ask a question without using raw API calls. Providing a user-friendly frontend is essential to meet Objective 5 (user-friendly data presentation) and part of Objective 4 (natural language interface). This remains as an urgent next step.

- **Natural Language Query Engine (LLM Integration) – Not Yet Complete:** The core AI feature that would allow a user to ask questions in plain language and receive answers (with the system doing the heavy lifting of retrieval and LLM response) is currently **0% integrated** in the running system [65]. While the groundwork (embeddings, vector search) is there, we have not wired up LangChain or an equivalent mechanism to handle NL queries and interact with GPT-4. Consequently, the system does not yet fulfill the promise of answering arbitrary questions from the documents – users can do keyword searches, but not ask something like "Summarize the Q3 financial results" and get a response. This is a significant portion of the project's goal left to be done.

- **Advanced AI Capabilities – Pending:** Along with basic Q&A, related features like summarization of documents, multi-document analysis, or the use of **MCP tools/agents** are not implemented. The status report flagged *"No LangChain integration, no RAG implementation, no MCP integration"* as current gaps [65]. These would have provided more intelligent query handling and automation. For example, no mechanism is in place to automatically summarize each document upon ingestion (even though a table exists for it). These intelligent features are planned for future work but were not reachable within the timeframe of the current phase.

- **Production Deployment and DevOps – Not Done:** The system is running in a development setup. Tasks related to containerization (Docker), environment configuration, and security hardening for production have not been completed [66]. There is no Docker image, and no CI/CD pipeline set up. Before a real deployment, things like serving static files, setting up HTTPS, and scaling the service would need to be addressed. This was expected to come after core development, but it remains a to-do item.

- **Non-functional Aspects:** Certain non-functional objectives, like handling of ambiguities via user clarification (mentioned in the objectives), have not been addressed yet. For instance, if two documents have similar information and a query is ambiguous, the system doesn't have logic to ask the user for clarification or to choose the best answer – these complexities are typically handled in the query frontend/agent logic which is yet to be built.

In terms of **objective fulfillment**: Objectives 1 and 3 are achieved, Objective 2 is partially (the system uses AI for embeddings but not fully for extraction as initially imagined), Objectives 4 and 5 are mostly not achieved yet (no NL interface or UI, though the backend groundwork is ready for them).

That being said, the current state is a strong milestone. The project has effectively **de-risked** the hardest parts (file parsing, data handling, vector DB integration). What remains are well-defined engineering tasks (building a UI, integrating an existing LLM API via LangChain). The risk of those failing is relatively low given the maturity of tools available. We acknowledge these missing components, and they are planned as immediate future work to bring the project to completion.

Future Work

Given the gaps identified, the next steps for ReportMiner focus on implementing the remaining features to achieve the full vision of an AI-powered query system. The future work can be outlined as follows:

Integration of LangChain and LLM for Question Answering

The top priority is to implement the **natural language query engine** using LangChain (or a similar framework) to tie together our vector search and an LLM (GPT-4). This will involve: - Setting up the LangChain integration in our Django app. We will create the planned `apps/query` with a view that accepts a query and orchestrates the RAG (Retrieval-Augmented Generation) process [36] [37]. - Utilizing our existing pgvector-backed store as a LangChain VectorStore. This may involve writing a wrapper or using `langchain-postgres` connectors to ensure LangChain can retrieve documents via SQL. The groundwork for Day 1 and Day 2 in the roadmap indicates precisely these steps [35] [67]. - Implementing the chain logic: for a given user question, fetch top-K relevant text segments, and construct an LLM prompt. Then invoke the OpenAI GPT-4 API (or possibly a cheaper model for development, like GPT-3.5) to generate an answer. We will format the output to include source references (e.g., document IDs or titles), so the user can trust and verify the answers. - Testing this loop thoroughly with various queries on our dataset to fine-tune prompt wording and ensure that the LLM's responses are accurate and not hallucinated. If needed, we might impose constraints like "if unsure, say you don't know" in the prompt. - This integration will achieve the core AI query functionality (Objective 4). According to our plan, we target a high accuracy (>95% on test queries) by leveraging the well-curated data and retrieval mechanism [48].

MCP Tools and Advanced Agent Integration

After basic Q&A is working, a more ambitious future step is to integrate the **Model Context Protocol (MCP)** tool ecosystem to allow more complex AI agent interactions: - We plan to implement an MCP server (possibly using an existing framework or our own Flask app) that exposes tools such as `search_documents(query)`, `get_document_summary(doc_id)`, `list_recent_documents()` etc. [68]. Each tool would perform an action via our database/API and return a result. - Using `langchain-mcp-adapters`, we can connect a LangChain agent to this MCP server [69]. The agent (an LLM) can then dynamically decide to call these tools when faced with a user request. - This would allow multi-step query resolution. For example, if a user asks: "Find all 2022 reports mentioning climate change and summarize their conclusions," an agent might first call `search_documents("climate change" with filter year=2022)` to get relevant docs, then loop through them calling `get_document_summary` on each, then compile a final answer. All these operations would be done by the AI agent through the tools. - Implementation steps include writing the tool functions (most of which wrap around existing queries or summarizations) and setting up the agent's environment and prompt to use them. We will also need to implement some summaries or have the agent generate them on the fly if summarization tool is called. - This feature is more experimental, but it represents the cutting edge of AI integration. If successful, it would significantly enhance the system's capabilities and "wow factor" – enabling conversational and analytical queries that go beyond simple Q&A [70].

Frontend Development (React UI)

In parallel to the AI enhancements, we will develop the **frontend interface** to make ReportMiner accessible and user-friendly: - We plan to create a single-page application (SPA) using **React** (with TypeScript) as outlined in the roadmap [71]. The frontend will communicate with our Django REST API. - Key components to implement: - **Document Upload Component:** A page or modal where users can drag-and-drop files or select files to upload. This will call the `/upload/enhanced/` API and show upload progress. After upload, it should show the document's status (perhaps polling until processing is complete) [72]. - **Document List View:** A dashboard showing all documents the user has uploaded (or all documents in the system, if multi-user with permissions). This list can allow filtering by type, date, etc., and indicate processing status or if ready [72]. - **Document Detail View:** When selecting a document, show its content in an organized way – perhaps the text segments and any extracted tables or key-values. This allows the user to explore the parsed data. It's mainly for verification and could be a tabbed interface (Text / Tables / Summary). - **Search/Query Interface:** A central search bar or chat interface where the user can type questions. This is the core of the NLP interface – the user enters a query (question), and the frontend calls our query API. We will display the answer returned by the LLM, as well as possibly the source snippets (for transparency). If using a chat UI, we can maintain a history of Q&A turns, allowing follow-up questions to be asked (for context retention, we might need to handle conversation state on the backend or simply rely on the user repeating context as needed). - **System Dashboard:** An admin-only view that could show stats like number of docs, success rate, etc., akin to our `stats/` API. This is lower priority but useful for monitoring. - We will use a UI framework like **Material-UI (MUI)** to create a clean, professional design [73]. Responsiveness (desktop and mobile) will be considered, though initial focus is desktop. - The frontend will also incorporate **authentication** if needed – perhaps a simple login system (leveraging Django's auth via REST, or a token system). This depends on whether multi-user support is needed immediately. At minimum, an admin login for the dashboard might be included. - Building this frontend will directly address the absence of a UI. According to the plan, we allocate about a week to implement and refine it [74] [75]. This is feasible given our clear API and the availability of component libraries.

Improving Extraction and Data Enrichment

Further enhancements can be made to the extraction and data handling: - **Document Summarization:** Using an LLM (like GPT-4) to generate a summary for each document upon ingestion (or on demand). We have a `document_summaries` table ready. This could be implemented as an additional pipeline step or a background task after processing. Summaries would be useful to display in the document list (giving users a quick overview of each document) and could also be utilized by the query engine (as a fallback answer or to provide context). - **Additional File Types:** Extending support to more formats such as PPTX (PowerPoint presentations), HTML pages, or plain text files. The architecture allows plugging in new extractors for these formats relatively easily. - **Data Visualization:** For structured data extracted (tables, numbers), the frontend could show charts or graphs (for example, if a table of financial metrics is extracted, provide a quick plot). While not core to Q&A, it enhances user-friendly data presentation. - **Feedback Loop:** Implement a way for users to give feedback on answers (whether the answer was correct/helpful). This can be logged and used to further fine-tune the system (for example, adjusting prompt strategies or identifying where extraction might have missed info). It also helps in evaluating the system in a real-world pilot. - **Performance Optimization:** As usage grows, we might need to optimize certain areas. For instance, if embedding generation becomes a bottleneck, we could explore using a local embedding model (like InstructorXL or others) to avoid API calls, or batch more aggressively. We could also pre-compute combined embeddings (averaging a document's vectors) for faster high-level search of which documents might be relevant, before diving into segment-level search.

Deployment and Production Readiness

Before the system can be used by external users, we will: - **Containerize** the application using Docker. We'll create a Dockerfile that sets up the Django app, installs required dependencies (including the pgvector extension on the Postgres image), and runs migrations. We will likely use a multi-container setup (one for the web app, one for the Postgres database). - Set up environment variables for secrets (like the OpenAI API key, database credentials) in a secure way. Possibly use Docker Compose or Kubernetes for orchestration in a cloud environment. - Implement **security best practices**: enforce HTTPS (if deploying on a server, use something like Nginx as a reverse proxy with SSL), configure Django ALLOWED_HOSTS properly, and ensure all API endpoints have appropriate permissions (so that, for example, one user cannot query another's documents if multi-user). - Add authentication: as noted, a simple JWT token system or session login for the web UI will be added so that the service is not open to the world without control. - **Monitoring & Logging:** Integrate with a logging service or at least ensure all logs (especially from error handling in the pipeline) are saved. Possibly set up an admin email for critical failures. If deploying on a platform like AWS, use CloudWatch or similar for monitoring performance metrics. - **Testing in Production-like environment:** We will carry out load testing on the Dockerized app to see if any new issues arise (sometimes minor differences in environment or concurrency appear when deployed).

In sum, the future work is well-defined: it primarily involves *completing the product features (frontend, NL query)* and *hardening the system for real-world use*. These steps will fulfill the vision of ReportMiner as an end-to-end solution where a user can upload a collection of reports and **interact with them through an AI assistant** in real time.

Conclusion

In conclusion, the ReportMiner project has successfully developed a substantial portion of an AI-powered document extraction and query system, demonstrating both the potential and the complexity of integrating modern NLP techniques with traditional data management. The work done so far validates the core concept: it is feasible to automatically ingest diverse reports, distill their content into a structured form, and prepare it for semantic querying. The backend system implemented is robust – featuring an enterprise-grade pipeline for text and data extraction and a scalable database with integrated vector search – which **lays a strong foundation** for the remaining components.

The results highlight that key technical challenges (multi-format parsing, error handling, vector integration) were met. The system can ingest documents and retrieve information with a level of automation and intelligence that would significantly ease the burden on end-users in data-heavy environments. **Advanced capabilities**, such as semantic similarity search and document classification, are already functioning in the backend, setting ReportMiner apart from more rudimentary solutions. This confirms that by combining Large Language Models (in this case, via embeddings) with a well-designed architecture, we can bridge the gap between unstructured data and structured queries.

At the same time, the project acknowledges the features still to be delivered – notably the natural language query interface and user-friendly front end. These are the focus of imminent development, and the progress to date gives confidence that they can be integrated smoothly. Once completed, ReportMiner will allow users to query their document collection in plain language and receive concise answers with references, fulfilling the ultimate goal of the project. The design decisions and infrastructure put in place ensure that these additions will have a solid platform to run on.

From an academic perspective, this project illustrates the effective application of interdisciplinary knowledge: database design, software engineering, and AI/machine learning. It also reflects a practical approach to problem-solving, where the solution leverages existing powerful APIs (like OpenAI) and tools (Django, LangChain, pgvector) in an innovative way to deliver new functionality. The experience gained includes working with cutting-edge AI integration (which is a rapidly evolving area), tackling real-world data variability, and addressing considerations of scalability and security for deploying such a system.

In conclusion, ReportMiner stands as a promising system that, with its remaining components in place, can evolve into a valuable tool for any organization dealing with information overload in documents. It epitomizes how modern AI can transform static reports into an interactive knowledge base. The project, upon final completion, will not only meet its stated objectives but also serve as a showcase of how to build an **AI-powered data solution** that is both technically sound and immensely practical. The journey thus far has been fruitful, and the forthcoming enhancements will mark the final step in turning the vision of ReportMiner into reality, ready for demonstration and real-world application.

Overall, this final year project has been a successful endeavor in pushing the boundaries of what a student project can achieve, effectively bringing together state-of-the-art AI and software engineering to solve a relevant, impactful problem in data management. The expectation is that the completed ReportMiner system will significantly **simplify data retrieval** from complex documents, empowering users to obtain insights and answers with ease and thereby validating the initial motivation that drove this project.

---

1 2 20 21 22 23 24 25 26 27 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 52 54 55 58 59 60 61 62 63 64 65 66 67 68 70 71 72 73 74 75 📊 REPORTMINER COMPREHENSIVE STATUS REPORT.pdf
file://file-Qpvcao8Mjm5LpJW2qDCxK4

3 11 13 Extracting Data from Unstructured Documents | Towards AI
https://towardsai.net/p/l/extracting-data-from-unstructured-documents

4 6 7 8 9 10 FYP.pdf
file://file-T8xwrM9NRQutCFK5vZtB9H

5 17 18 The Design of an LLM-powered Unstructured Analytics System
https://arxiv.org/html/2409.00847v2

12 How to Use LayoutLM for Document Understanding and Information Extraction with Hugging Face Transformers - KDnuggets
https://www.kdnuggets.com/how-to-layoutlm-document-understanding-information-extraction-hugging-face-transformers

14 What Is Natural Language Querying? - Ontotext
https://www.ontotext.com/knowledgehub/fundamentals/what-is-natural-language-querying/

15 QueryGPT - Natural Language to SQL using Generative AI | Uber Blog
https://www.uber.com/blog/query-gpt/

16 pgvector/pgvector: Open-source vector similarity search for Postgres
https://github.com/pgvector/pgvector

19 50 51 53 56 57 FYP - ingestion.pdf
file://file-ALMVxe3WkxJNw8cxDgu3Vf

28 29 FYP - DB .pdf
file://file-XnKL3ViWzn1cnyqJTiHuFN

69 langchain-ai/langchain-mcp-adapters: LangChain MCP - GitHub
https://github.com/langchain-ai/langchain-mcp-adapters