Contract Intelligence API Design Document

1. Architectural Overview

The Contract Intelligence API is built on a standard three-tier, containerized architecture designed for scalability, persistence, and external service integration. This architecture separates the application logic from the data layer and external LLM services, ensuring maintainability and adherence to modern best practices.

Architecture Diagram

The system comprises three primary components: the Client, the Web/API Tier, and the Data/External Services Tier.

1. **Client/User:** Interacts with the API primarily through Swagger UI, cURL, or other HTTP clients.
2. **Web/API Tier (Django/DRF Container):**
   - Handles all HTTP requests (GET, POST).
   - Contains the core business logic (`api/views.py`) for routing and data processing.
   - Hosts the utility functions (`api/utils.py`) responsible for text extraction and LLM communication.
   - Mediates all interaction between the PostgreSQL database and the OpenAI LLM.
3. **Data & External Services Tier:**
   - **PostgreSQL Database:** Provides persistent storage for all contract metadata and extracted text.
   - **OpenAI LLM:** External service used for the heavy-lifting of complex tasks (Extraction, Auditing, RAG).

2. Data Model

The core data entity is the `Document` , stored in the PostgreSQL database. This model ensures that all necessary metadata and the complete text content of the contract are persisted.

Field Name	Data Type	Description	Rationale
id	Integer (Primary Key)	Unique identifier generated upon successful ingestion.	Used as the `doc_id` for all downstream API calls (`/extract` , `/audit` , `/ask`).
title	String (255 chars)	Original filename of the uploaded PDF (e.g., `msa1.pdf`).	Facilitates user identification and logging.

file	FileField	Stores the original PDF file reference (though only text is actively used).	Retains the source document for archival purposes.
extracted_text	TextField	**Full, plain-text content** extracted from the PDF upon ingestion.	This is the source content for all LLM operations (Extraction, RAG, Auditing).
uploaded_at	DateTime	Timestamp of ingestion.	For logging and data management.

3. Chunking Rationale (RAG Approach)

The Retrieval-Augmented Generation (RAG) feature (`/api/ask/`) must answer questions based on the document content.

- **Choice: Direct Text Slicing** with a large context window (`gpt-3.5-turbo-16k`).
- **Mechanism:** Instead of using a complex vector database for embedding and retrieval, the system relies on the LLM's extended context capability.
  - The complete `extracted_text` is loaded from the database.
  - The text is sliced to fit within the model's context limits (e.g., `doc.extracted_text[:20000]`) to create the `CONTEXT` block in the user prompt.
  - The prompt instructs the model to answer the `QUESTION` **ONLY** based on this provided `CONTEXT` .
- **Trade-off Rationale:** This approach was chosen for its rapid development, simplicity, and low operational overhead. It is effective for shorter to medium-length contracts (which fit entirely within the 16k token limit), but it sacrifices the ability to perform deep, semantic retrieval across vast, multi-document libraries.

4. Fallback Behavior and Reliability

The system is designed with a **Rule Engine Fallback** to ensure that critical extraction and auditing services remain functional and return a predictable payload even if the external LLM is inaccessible (e.g., due to quota exhaustion, network issues, or service downtime).

- **Mechanism:** A global boolean flag, `FALLBACK_ENABLED` , is defined in `api/views.py` .
- **Fallback Logic:**

```
if FALLBACK_ENABLED:
    # Return deterministic, hardcoded data immediately
    return Response({"parties": ["Fallback Inc.", "Contracting Party"], ...}, statu
else:
    # Proceed with LLM API call
    # ...
```

- **Value:** This mechanism allows the service to deliver a predictable, structured response—albeit one with placeholder data—to downstream systems, preventing critical application failure and providing time for operational teams to restore LLM service availability.

5. Security Notes

Security measures focus on protecting sensitive credentials and controlling access.

1. **Credential Management:** The `OPENAI_API_KEY` and PostgreSQL database credentials are managed exclusively through **environment variables** loaded via the project's `.env` file. These secrets are never hardcoded into the application source code.
2. **Container Isolation:** Using Docker Compose ensures the services operate in isolated networks, minimizing the external attack surface. The database is only accessible by the `web` container.
3. **Input Validation:** File ingestion (`/ingest`) is restricted to the PDF file format, and subsequent LLM endpoints enforce validation on the `document_id` and the JSON input to prevent malformed requests.
4. **Least Privilege:** The PostgreSQL user is configured to have minimal required permissions on the