HRGenie: An Agentic System for Offer Letter Generation

📄 Project Overview

HRGenie is a sophisticated agentic system designed to automate the generation of formal offer letters. The system intelligently combines employee-specific metadata with relevant HR policies, using a Retrieval-Augmented Generation (RAG) pipeline to ensure contextual accuracy. For maximum reliability and to mitigate potential LLM failures, it incorporates a robust fallback mechanism using deterministic Jinja2 templating. The entire system is built on a modular architecture, with a Python-based core, a FastAPI backend, and a Streamlit frontend, facilitating a clean, maintainable, and scalable solution.

PROJECT LINK (deployed on hugging face):
https://huggingface.co/spaces/hackarag/CoverRAG

✨ Key Features at a Glance

- **Intelligent Document Chunking:** A custom strategy for parsing HR policies and handling tabular data effectively.
- **Vectorized Knowledge Base:** A Qdrant vector database stores embeddings of HR policies, forming a searchable and context-rich knowledge base.
- **Context-Aware Retrieval:** A retriever component fetches the most relevant policy documents based on an employee's query, ensuring the LLM has the necessary information.
- **Hybrid Generation:** Utilizes a generative LLM for dynamic, RAG-based content creation, with a graceful failover to a static, deterministic Jinja2 template.
- **Modular Architecture:** The system's design separates concerns across different components (ingestion, retrieval, generation, API, UI), promoting flexibility and maintainability.

📁 Directory Structure & Core Components

The project is structured logically to separate data, core logic, utilities, and user interfaces, reflecting a professional software development approach.

```
project-root/
├── data/                          # 📦 All input/output data
│   ├── raw_pdfs/                  # HR policies and sample letters
│   │   ├── HR Leave Policy.pdf
```

```
│   │   ├── HR Travel Policy.pdf
│   │   └── HR Offer Letter.pdf
│   ├── docs_chunks/             # Chunked JSONs from PDFs
│   ├── embeddings/              # Embeddings (raw)
│   ├── qdrant_ready_embeddings/ # Qdrant-compatible embeddings
│   ├── employee_list.csv        # Source employee metadata
│   ├── employee_list.json       # Converted JSON
│   ├── wfo_policy.json          # Mapping of team to WFO policy
│   ├── generated_letters/       # Markdown/Plaintext outputs
│   └── offer_letters/           # Final offer letter PDFs
│
├── backend/                     # 🧠 Core logic + model pipeline
│   ├── ingest/                  # Chunking, embedding, upload
│   │   ├── chunk_and_embed.py
│   │   └── upload_qdrant.py
│   ├── retriever.py             # Qdrant-based retriever
│   ├── generate_offer_letter.py # RAG-based generation (LLM + retriever)
│   ├── generate_offer_withoutrag.py # LLM-only generator (no retrieval)
│   └── generate_offer_letter_nollm.py # Jinja2 fallback generator
│
├── utils/                       # 🔧 Shared helpers/utilities
│   ├── load_employee_metadata.py
│   └── save_offer_letter_pdf.py
│
├── templates/                   # 📄 Jinja2 fallback templates
│   └── offer_template.txt
│
├── frontend/                    # 🎛️ UI layer
│   ├── app.py                   # Streamlit UI
│   └── static_ui/               # (Optional) HTML-based UI
│       └── index.html
│
├── api/                         # 🌐 REST API
│   └── api_server.py            # FastAPI backend server
│
├── logs/
│   └── chunking.log             # Chunking log
│
├── requirements.txt             # Python dependencies
└── README.md                    # Project documentation
```

⚙️ Detailed Workflow and Technical Highlights

Step 1: Intelligent Document Ingestion and Chunking

The project's foundation is built on its ability to parse and structure unstructured PDF documents. The `chunk_elements()` function uses the `unstructured` library but enhances it with custom logic to address the common challenge of handling tabular data.

- **Core Partitioning Logic:** The script begins by partitioning the PDF using `unstructured.partition_pdf` with the `infer_table_structure=True` parameter. This is a critical first step as it instructs the parser to correctly identify and extract tables as distinct `Table` elements, rather than treating them as a continuous stream of text.
- **Specialized Table Handling:** The custom chunking logic is implemented to handle these `Table` elements specifically. When a `Table` element is encountered, the script finalizes any preceding text into a separate chunk. It then creates a new, dedicated chunk for the table itself, ensuring that its structured information is preserved and not merged with surrounding paragraphs. This prevents critical data from being lost or misinterpreted by the LLM.
- **Heuristic for Title Association:** The code includes a heuristic check to identify and disregard short headings or captions that may appear directly before a table. This prevents these short text snippets from mistakenly being treated as standalone section titles, ensuring they remain semantically linked to the table they describe.

Step 2: Embedding Generation and Qdrant Storage

After chunking, the text must be converted into a searchable format. This process is designed to be robust and production-ready.

- **Embedding Model & Dimensionality:** Each chunk of text is transformed into a high-dimensional vector using **OpenAI's `text-embedding-3-small`** model. The chosen dimensionality of `1536` is explicitly configured in the Qdrant collection, which is the standard size for this model.
- **Qdrant-Ready Data Format:** The `embed_and_upload.py` script takes the generated embeddings and structures them for Qdrant ingestion. Each data point is formatted as a `PointStruct` with three essential components:
    1. **`id`**: A unique identifier generated using `UUID5` derived from the chunk's ID. This method ensures that the same chunk will always produce the same ID, making

the upload process **idempotent** and preventing duplicate entries on subsequent runs.
2. **`vector`**: The 1536-dimensional embedding vector representing the text chunk.
3. **`payload`**: The original metadata, including the `section_title`, `chunk_type`, and `source` document. This metadata is invaluable for advanced filtering and for providing context in the LLM prompt.

- **Collection Setup & Distance Metric:** The script connects to a locally hosted Qdrant instance. It programmatically creates a collection named `policy_chunks` if it does not exist, configured with a `COSINE` distance metric. This metric is the standard choice for text embeddings as it measures the angular similarity between vectors, which effectively represents semantic relatedness.
- **Efficient Bulk Upload:** The `client.upsert()` method is utilized to perform an efficient bulk upload of all the points, optimizing the population of the vector database.

Step 3: Retrieval-Augmented Generation (RAG)

The RAG pipeline is the core of the system's "agentic" capabilities, allowing it to generate contextual and accurate offer letters.

- **Function: `load_employee_metadata()`:** This utility is responsible for fetching employee-specific details (name, team, band, salary, etc.) from `employee_list.json`. It includes robust error handling to raise descriptive exceptions if the file or a specific employee is not found, and it normalizes keys for data consistency.
- **Function: `retrieve_relevant_chunks()`:** This is the **retriever component**. It takes a user query, generates an embedding for it, and then performs a similarity search against the `policy_chunks` collection in Qdrant. It retrieves the `top_k` most relevant text chunks, which are then passed to the LLM as external context.
- **Function: `generate_offer_letter()`:** This is the primary generation script. It orchestrates the RAG process by combining the employee metadata with the text chunks retrieved from Qdrant. These are injected into a highly specific, few-shot prompt that instructs the LLM (`gpt-4o-mini`) to act as a "professional HR assistant." The use of a low temperature (`0.3`) encourages a precise and non-creative response, crucial for generating a formal and accurate document.

Step 4: Jinja2 Fallback (No-LLM) Setup

A key strength of this system is its ability to function reliably without an LLM. This is handled by a well-designed fallback mechanism.

- **Function: `generate_offer_letter_jinja()`:** This is the **no-LLM function**. It completely bypasses the LLM and instead uses the **Jinja2 template engine** to render

`offer_template.txt`. This approach serves as a deterministic and highly reliable **guardrail** against LLM failures.
- **Deterministic Policy Application:** Instead of relying on an LLM to interpret policies, this function uses explicit, hardcoded Python dictionaries (`TITLE_BY_TEAM`, `WFO_POLICY_BY_TEAM`) for policy mapping. It looks up the correct job title and policy details based on the employee's team, ensuring a predictable and consistent output every time.
- **Custom Jinja Filters:** A custom Jinja filter named `comma` is implemented to format salary numbers, a professional touch that improves the readability of the final document.
- **Graceful Failure:** The `generate_offer_letter.py` script is wrapped in a `try...except` block. This ensures that if the LLM call fails for any reason (API error, timeout, or a validation failure), the system seamlessly falls back to the Jinja2-based generator, guaranteeing that an offer letter can always be produced.

Step 5: API and Frontend Integration

The final steps involve exposing the system's functionality to the end-user.

- **`api_server.py`:** A FastAPI server handles requests from the frontend. It uses a Pydantic `BaseModel` for robust request body validation and a simple `use_jinja` flag to dynamically route the request to either the LLM-based or the Jinja2-based generator.
- **`app.py`:** A Streamlit application serves as the user-friendly interface. It features a toggle button to switch between the LLM and no-LLM generation modes. It displays the generated offer letter and allows for a PDF download using the `save_offer_letter_pdf.py` utility, which handles Unicode fonts and text formatting for a professional-looking document.

💻 Tech Stack & Python Libraries

This project is built using a combination of powerful Python libraries and frameworks, selected for their effectiveness in building scalable and reliable RAG systems.

Core Libraries

- **`unstructured`**: For document parsing, chunking, and metadata extraction.
- **`qdrant-client`**: The official Python client for interacting with the Qdrant vector database.
- **`openai`**: The Python library for interfacing with OpenAI's LLMs and embedding models.
- **`jinja2`**: The templating engine used for the deterministic no-LLM fallback.

Web Frameworks & UI

- `fastapi`: The high-performance web framework for building the API server.
- `uvicorn`: The ASGI server that runs the FastAPI application.
- `streamlit`: The framework used to create the user-friendly web application for HR personnel.
- `pydantic`: Used for data validation and settings management, integrated with FastAPI.

Utilities & Other

- `python-dotenv`: For managing environment variables securely.
- `tqdm`: A progress bar for displaying the progress of loops, useful during embedding generation and data ingestion.
- `fpdf2`: A library for generating PDF documents from the final offer letter text.
- `datetime`: A built-in Python library for handling dates, used to timestamp the offer letters.
- `langchain`: Provides a framework and abstractions for building the RAG pipeline.

—---------

For more descriptive knowledge go through the README file.