**Skill Matrix App**

The Skill Matrix Filler App is a Python Flask-based web application designed to automate the process of extracting data from resumes, answering questions about resumes and job templates, and dynamically filling skill/qualification matrices. It leverages advanced technologies like Large Language Models (LLMs) and vector databases (ChromaDB) to provide intelligent document processing capabilities.

The application is structured into a main application file (app.py), core logic modules (core/), utility functions (utils/), and static frontend assets (static/).

**1. app.py (The Central Orchestrator)**

* **Purpose:** app.py is the **heart of the application**. It's the Flask web server that handles all incoming requests from the web browser, orchestrates the workflow by calling functions from other modules, and sends responses back to the user interface. It acts as the "traffic controller" for the entire system.
* **Key Responsibilities:**
  + **Web Server:** Initializes and runs the Flask web server, listening for user interactions via HTTP requests.
  + **Routing:** Defines the different URLs (endpoints) that the web browser can access (e.g., /, /upload\_and\_ingest, /fill\_template, /ask, /download\_filled\_resume).
  + **Centralized Logging:** Configures the application-wide logging system, directing detailed logs to app.log for troubleshooting and important messages to the console for real-time monitoring.
  + **File Management:** Manages the temporary storage of uploaded resumes and templates (uploads/) and the saving of generated documents (filled\_docs/). It ensures unique file naming using timestamps (e.g., resume\_basename\_YYYYMMDD\_HHMMSS.ext) to prevent conflicts and stores them directly in the uploads/ folder (no nested subfolders per upload).
  + **Session Management (via ChromaDB):** Relies solely on ChromaDB to store and retrieve all session-specific data (structured resume JSON, document text chunks, file paths, template schema), ensuring data persistence and scalability. The application does not use an in-memory session\_store.
  + **Workflow Orchestration:**
    - **Upload & Ingest (/upload\_and\_ingest):** Receives files, saves them, initiates resume parsing, and then triggers the embedding and storage of all relevant text and the template schema into ChromaDB.
    - **Get Structured Data (/get\_structured\_resume\_data):** Fetches the LLM-parsed structured resume data from ChromaDB for display on the frontend. Uses ChromaDB's $and operator for precise filtering.
    - **Fill Template (/fill\_template):** Retrieves necessary data (structured resume, template schema, original template path) from ChromaDB, determines the type of template (skill matrix vs. generic), calls the appropriate template processing function, and provides a download link for the filled document.
    - **Ask Q&amp;A (/ask):** Retrieves relevant document chunks and structured data from ChromaDB, constructs a comprehensive context, sends it to the LLM for a refined answer, and returns the response.
    - **Download Filled Resume (/download\_filled\_resume):** Serves the generated Word document to the user's browser.
* **Key Design Aspects:** This module is the "face" of the application, demonstrating the user's interaction points and the overall flow of automation. Its robustness and efficient data handling are key to a smooth user experience.

**2. core/ Module (The Brains of the Operation)**

The core/ directory contains the specialized Python modules that perform the intelligent processing tasks. core/\_\_init\_\_.py is an empty file that simply designates core as a Python package, allowing its internal modules to be imported relatively.

**2a. core/chroma\_service.py (The Memory)**

* **Purpose:** This module acts as the interface to the **ChromaDB vector database**. It manages how documents are stored, retrieved, and organized, forming the application's long-term memory for processed data.
* **Key Responsibilities:**
  + **Persistent Client:** Initializes ChromaDB in persistent mode, meaning all stored data is saved to disk (chroma\_db\_data/) and remains available even after the application restarts.
  + **Custom Embedding Function:** It explicitly tells ChromaDB to use the get\_embedding function from llama\_runner.py (which uses all-MiniLM-L6-v2 via SentenceTransformer) for generating embeddings. This ensures consistency between how text is vectorized for storage and how user queries are vectorized for retrieval.
  + **Collection Management:** Provides functions to get\_or\_create\_collection (ensuring data is organized in logical groups, like by session) and clear\_collection.
  + **Document Addition (add\_documents\_to\_chroma):** Takes text content, metadata (like session\_id, source\_type - e.g., 'resume\_text\_chunk', 'structured\_resume\_data', 'template\_text\_chunk', 'template\_schema'), and unique IDs, then converts them into embeddings and stores them in the specified ChromaDB collection.
  + **Querying (query\_chroma):** Executes a similarity search query against a specified ChromaDB collection, optionally filtering by metadata. It correctly handles complex filter syntax passed from app.py (e.g., using $and, $or operators).
* **Key Design Aspects:** This module underpins the "intelligent search" (Retrieval-Augmented Generation - RAG) capability. It allows the LLM to access specific, relevant information from past documents, preventing hallucination and ensuring factual accuracy. Its persistence means processed data isn't lost.

**2b. core/llama\_runner.py (The Conversational AI Engine)**

* **Purpose:** This module is the direct interface to the **Large Language Model (LLM)**, specifically Mistral via Ollama. It handles all text generation and embedding generation using the LLM.
* **Key Responsibilities:**
  + **Ollama Integration:** Configures the connection to the local Ollama server and specifies the quantized LLM model to use (mistral:7b-instruct-v0.2-q4\_K\_M) for text generation. This leverages Ollama's local optimization and potential GPU offloading for improved performance.
  + **Embedding Model:** Loads the all-MiniLM-L6-v2 SentenceTransformer model, which is used to convert text into numerical vector embeddings for ChromaDB operations.
  + **Startup Validation:** Checks if the Ollama server is accessible and if the embedding model loads successfully, ensuring the core AI components are ready.
  + **get\_embedding Function:** Provides a consistent method for other modules (core/chroma\_service.py) to obtain vector embeddings for any given text.
  + **run\_llama\_prompt Function:**
    - **LLM Interaction:** Sends formatted prompts (with context) to the Ollama server and retrieves the generated text responses.
    - **Robust Prompting:** Structures the conversation with system and user roles, providing clear instructions for the LLM to be accurate, concise, and **strictly adhere to the provided context, avoiding outside knowledge or hallucination**.
    - **Parameter Control:** Allows tuning LLM behavior with temperature (controlling creativity/determinism), top\_p (sampling diversity), and max\_new\_tokens (response length).
* **Key Design Aspects:** This module is where the "intelligence" comes from. It's responsible for the LLM's ability to understand complex requests, extract structured data, and generate coherent, context-aware responses. Its local deployment via Ollama is a key cost and privacy advantage.

**2c. core/resume\_parser.py (The Resume Interpreter)**

* **Purpose:** This module specializes in taking raw resume text and transforming it into a structured, machine-readable JSON format using the power of the LLM.
* **Key Responsibilities:**
  + **Text Extraction:** Calls extract\_text\_from\_file (utils.file\_utils) to get raw text content from various resume file formats (DOCX, PDF, DOC).
  + **Context Management:** Truncates very long resumes to fit within the LLM's context window, ensuring efficient processing.
  + **parse\_resume\_to\_structured\_data Function:**
    - **Programmatic Name Pre-Extraction:** **This is a critical robustness feature.** It performs a very targeted, deterministic LLM call (low temperature, short response) on just the first few lines of the resume to extract the full\_name. This significantly minimizes the chances of name hallucination.
    - **Dynamic Name Injection:** The pre-extracted name is then *injected* directly into the JSON schema within the main LLM prompt, acting as a fixed constraint for the LLM during the full resume parsing process.
    - **Robust JSON Extraction Prompt:** Provides precise instructions to the LLM to output only a specific JSON schema, handling missing fields gracefully, and **explicitly demanding accuracy from the resume text without inference or guessing**.
    - **Post-Processing Override:** Includes a final safeguard to ensure the full\_name in the ultimate output JSON is the pre-extracted name.
    - **JSON Validation:** Parses the LLM's raw response into a JSON object, with error handling for malformed JSON.
* **Key Design Aspects:** This module automates a traditionally manual and error-prone process (resume data entry), enabling downstream automation for skill matrix filling and internal talent management. The robust name extraction is vital for data integrity.

**2d. core/template\_analyzer.py (The Template Architect) - NEW MODULE**

* **Purpose:** This new module is designed to **dynamically understand the structure and fields of any blank template** by leveraging the LLM, translating that understanding into a machine-readable JSON schema.
* **Key Responsibilities:**
  + **Text Extraction:** Obtains the raw text content of the template file.
  + **Context Management:** Truncates long template texts to fit LLM context windows.
  + **analyze\_template\_structure Function:**
    - **LLM-Powered Structure Inference:** Sends the raw template text to the LLM with a highly specific prompt. This prompt guides the LLM to identify logical sections, expected fields, tables, and determine if sections/tables are repeatable.
    - **Schema Output:** The LLM is instructed to output this inferred structure as a valid JSON schema, including field names, types, column descriptions for tables, and is\_repeatable flags.
    - **Robust Prompting:** Uses strict guidelines to prevent hallucination and ensure accurate structural inference based *only* on the provided template text.
    - **JSON Validation:** Parses and validates the LLM's JSON output for the schema.
* **Key Design Aspects:** This module is a crucial step towards true "any template" dynamic filling. It allows the system to adapt to new template layouts without hardcoding, by creating a semantic understanding of the template's structure.

**2e. core/template\_processor.py (The Document Filler)**

* **Purpose:** Fills Word document templates with data, using LLM-generated instructions and (for generic templates) the LLM-inferred template\_schema.
* **Key Responsibilities:**
  + **DOCX Manipulation Helpers:** Contains low-level helper functions (\_clear\_paragraph\_content, \_add\_text\_to\_paragraph\_smart, \_clear\_cell\_content, \_add\_text\_to\_cell\_smart) to precisely insert/replace text in Word document paragraphs and table cells, handling line breaks and bullet points.
  + **fill\_dynamic\_template Function:**
    - **Generic Filling:** Designed to fill *any* arbitrary Word template.
    - **Schema-Guided LLM Prompting:** Retrieves **both** the structured resume data **and** the template\_schema (from core/template\_analyzer.py). It then uses a sophisticated prompt to instruct the LLM to generate filling instructions, **explicitly mapping the resume data to the template\_schema's inferred structure**.
    - **Dynamic Application (Current Limitation):** It applies the LLM's JSON instructions to fill paragraphs and *pre-existing* table cells. **Important Note:** While the LLM can *instruct* about repeating sections (e.g., provide a list of 5 job entries), the current python-docx integration **does not automatically create new table rows or complex structures if the template doesn't have enough pre-defined empty slots.** This would require highly advanced python-docx manipulation or dedicated templating libraries.
  + **fill\_skill\_matrix\_template Function:**
    - **Specific Filling:** Optimized for the Skill-Qualification\_Matrix\_Template.docx format.
    - **Dynamic Row Start Detection:** Uses re.match(r'^\d+\.', first\_cell\_text) to dynamically find the starting row of qualifications (e.g., "1.", "2.") in the table, making it adaptable to slight header changes.
    - **Highly Robust Prompts for Responses:** For each qualification, it makes targeted LLM calls for "Vendor / Consultant Response" and "Customer / Project References". These prompts demand strict adherence to resume data and explicitly state "MUST state 'N/A'" if information is not found, preventing hallucination.
    - **Direct Cell Insertion:** Inserts the LLM's responses directly into the correct table cells.
* **Key Design Aspects:** This module automates the generation of key outputs like skill matrices, significantly reducing manual effort and ensuring consistency in responses. The new dynamic capabilities push towards handling diverse template formats.

**3. utils/ Module (The Toolset)**

The utils/ directory contains helper functions used across various modules. utils/\_\_init\_\_.py is an empty file that simply designates utils as a Python package.

**3a. utils/file\_utils.py (The Document Reader)**

* **Purpose:** Provides fundamental utilities for interacting with various document files and processing their raw text content.
* **Key Responsibilities:**
  + **extract\_text\_from\_file:** A versatile function that extracts plain text from common document formats: PDF (using PyMuPDF), DOCX (using python-docx), and old DOC files (using mammoth). It includes robust error handling for unsupported formats or extraction failures.
  + **chunk\_text:** Splits large blocks of text (like entire resumes or templates) into smaller, overlapping "chunks". These chunks are essential for efficient embedding and retrieval from ChromaDB, as well as for fitting within LLM context windows.
  + **get\_combined\_context (for debugging/logging):** A simple helper to get raw text for debugging, not directly used in the LLM's main RAG context, which is dynamically built from ChromaDB.
* **Key Design Aspects:** This module ensures the system can ingest information from a variety of common document formats, which is foundational for any document automation solution. Its chunking capabilities optimize data for downstream AI processing.