

I. Introduction & Project Goal

This document outlines the requirements for a project to build an AI-powered recruitment scanning agent. The primary goal of this project is to automate the initial stages of candidate screening by:

- Processing a Job Description (JD) provided as a PDF file to extract key details like Position Title and Job ID.
- Processing locally stored digital CVs (PDF and DOCX formats) in an efficient, memory-optimal batch manner.
- Extracting key candidate information from CVs using Google's **Gemini 1.5 Flash** model via its API.
- Matching candidate profiles against the provided Job Description (JD) using the **Gemini 1.5 Flash** API.
- Generating a match score, ranking category, and detailed reasoning for each candidate's match.
- Populating a structured candidate profile with all extracted and generated information into a designated Notion database, while skipping the creation of duplicate entries.

I. Scope of Work

The outsourced developer/company will be responsible for developing a command-line interface (CLI) application that performs the following sequential operations:

1. CLI Initial Guidance & Pre-run Checks:

- When the application is first run or executed with a specific help command (e.g., --help), it should display clear instructions to the user on:
 - The purpose of the script.
 - Required CLI arguments (e.g., CV folder path, JD PDF file path, Notion Database ID) and their expected formats.
 - How to set up the necessary configurations in the .env file (API keys, default Notion Database ID if not provided via CLI).
- Before starting the main processing, the script should perform pre-run checks for:
 - Presence and validity of required CLI arguments.
 - Existence of the .env file and essential API keys/configurations.
 - Existence and readability of the specified CV folder and JD PDF file.
- If any pre-run check fails, the script should output a specific, user-friendly error message indicating the problem and how to rectify it, then exit gracefully.

2. Initialization:

- Load necessary configurations, including API keys for Google Gemini and Notion, the Notion Database ID, and input file paths (CV folder, JD PDF file) via CLI arguments and/or a .env configuration file. CLI arguments should take precedence if both are provided for a given setting (e.g., Notion DB ID).
- Initialize API clients for Gemini and Notion.

3. Job Description (JD) Processing:

- Accept a single Job Description (JD) provided as a **PDF file** path via a CLI argument.
- Extract all text content from the JD PDF. If the JD PDF is unreadable or password-protected, the script should report an error and terminate gracefully.
- The primary approach for extracting "Position Title" and "Job ID" from the JD text should be via a targeted Gemini API call (using the **Gemini 1.5 Flash** model) on the extracted JD text. Rule-based methods may be proposed by the developer as an alternative *only if* the provided sample JDs exhibit a highly consistent and simple structure amenable to such rules, and this approach is demonstrably simpler and equally effective. The final method for JD parsing requires client discussion and approval. If critical JD information (Position Title, Job ID) cannot be extracted, the script should notify the user and may either halt or proceed with CV matching without this specific JD context (this behavior to be clarified based on client preference during development).

4. CV Processing (Batch Operation with Progress Indication & Memory Optimization):

- The CLI must display **loading indicators with a percentage bar** (or similar visual progress feedback, e.g., using rich or tqdm) showing the progress of iterating through and processing CV files (e.g., 'Processing CV X of Y: [CV_FILENAME] [| | | ----] 50%').
- The application must process CVs in a **batch manner**, efficiently handling all supported files (PDF, DOCX) within the specified folder.
- Processing should be **memory-optimal**, ensuring the script does not consume excessive memory that could lead to crashes when dealing with a moderate batch of CVs (e.g., 20-50 files) or reasonably large individual files. This includes processing files one by one where feasible, closing file handles promptly, and avoiding loading entire file contents into memory simultaneously if not strictly necessary for the parsing library.
- For each CV:
 - **Text Extraction:** Extract raw text content from the CV file. Handle potential empty or malformed files gracefully.
 - **AI Processing (via Gemini 1.5 Flash API):**
 - Send the extracted CV text and the full JD text (and its extracted details like Position Title and Job ID for context if helpful to the prompt) to the Gemini 1.5 Flash API.
 - Instruct Gemini (via the prompt outlined in Section 5) for: Candidate profile information (Full Name, Email, Contact Number, Skills, Experience Summary), Match Score, Ranking Category, and a detailed **Reasoning Statement** for the match.
 - Receive and parse the structured JSON response from Gemini.
 - **Notion Database Integration:**
 - **Duplicate Prevention:** Before creating a new entry, perform a check in the Notion database to see if a candidate with the same extracted email already exists for the current Job ID (JD). If such an entry is found, the script should log that a potential duplicate was identified for that specific Job ID and **skip creating a new entry**.

- If no duplicate is found, create a new Notion entry (page) for the processed candidate in the specified database.
 - Populate the Notion properties (as defined in Section 4) with the information extracted from the CV by Gemini, the AI-generated matching details (score, category, reasoning), the extracted JD details (Position Title, Job ID), and process-related data (e.g., CV filename, processing date).
 - **Handling Missing Data from Gemini:** If Gemini returns "N/A" for optional fields (e.g., contact_number, email if not found) or if a field is legitimately missing, the corresponding Notion property should be left blank (if the property type allows) or populated with "N/A" if a string is required and makes sense. This behavior should be consistent.
 - **Error Handling for Individual CVs:** During batch processing, if an error occurs while processing a single CV (e.g., corrupted file, unparseable content by Gemini, specific API error for that CV after retries if applicable), the script should log the error to the console, clearly identify the problematic CV file, and then **continue processing the next CV** in the folder. It should not terminate the entire batch process for an individual file error.
 - Log basic success/failure messages (or duplicate skipped messages) to the CLI for each CV processed.
5. **Completion & Summary Report:**
- Upon completion of processing all files in the folder, output a summary report to the CLI, including:
 - Total number of files found in the CV folder.
 - Number of supported CV files attempted for processing.
 - Number of CVs successfully processed and added to Notion.
 - Number of CVs skipped due to being potential duplicates.
 - Number of CVs that failed processing (with a reference to check console logs for specific errors).
 - The Notion Database URL or a reminder of the Database ID used for output.

II. Technical Specifications & Tech Stack

The application must be developed using the following technologies:

- **Programming Language:** Python (latest version 3.13.3)
- **CLI Framework:** Typer (preferred, for its ease of use and integration with rich) or Click.
- **CLI Progress Indicators:** A library like rich (integrated with Typer) or tqdm for displaying progress bars.
- **AI Model:** Google Gemini 1.5 Flash
- **API Client Libraries:**
 - **Google Gemini API:** google-generativeai Python library.
 - **Notion API:** notion-client official Python SDK.

- **Document Parsing Libraries:**
 - For PDF files (both CVs and JD): pdfplumber (recommended) or PyPDF2.
 - For DOCX files (CVs): python-docx.
- **Environment & Dependency Management:**
 - Standard Python venv for virtual environments.
 - A requirements.txt file listing all project dependencies.
- **Configuration Management:**
 - Use of a .env file (leveraging the python-dotenv library) to manage API keys, the Notion Database ID, and other configurable parameters.
- **Coding Standards:** Adherence to PEP 8 guidelines and general best practices for Python development for readability and maintainability.
- **Memory Management:** Code should be written with memory efficiency in mind, especially during file processing loops.

III. Notion Database Structure

The client will pre-configure a Notion Database. The developer will be provided with the **Database ID**. The script must populate the following properties for each candidate:

Property Name	Notion Type	Data Source / Purpose
Candidate Name	Title	Candidate's full name (primary identifier). Extracted by Gemini.
Email	Email	Candidate's primary email address. Extracted by Gemini.
Contact Number	Phone	Candidate's primary phone number. Extracted by Gemini.
Skills	Multi-select	List of key skills. Extracted by Gemini. API should create new options if they don't exist.
Experience Summary	Text	Concise summary of work experience. Extracted/Generated by Gemini.
CV Filename	Text	Name of the original CV file processed. Set by script.
Position Title (JD)	Text	Position title extracted from the JD. Set by script (via Gemini/rules).
Job ID (JD)	Text	Job ID extracted from the JD. Set by script (via Gemini/rules).
JD Match Score	Number	Numerical score (0-100) from Gemini.

Ranking Category	Select	E.g., "High Fit," "Medium Fit," "Low Fit." Options pre-defined by Client. Set by Gemini.
AI Ranking Reason	Text	Detailed explanation & reasoning for the ranking from Gemini.
Processing Date	Date	Date/time of processing. Set by script.
Status	Select	Initial status (e.g., "New - AI Processed"). Options pre-defined by Client. Set by script.

(The client will define the options for 'Select' type properties like Ranking Category and Status in Notion beforehand).

IV. Gemini API Prompt Guidance

The core AI processing relies on effective prompting of the **Gemini 1.5 Flash** model. The developer will implement a prompt structure capable of extracting the required CV information and performing the JD matching, including the **reasoning stage**. The following is a recommended starting point, which must be adapted to also facilitate the extraction of "Position Title" and "Job ID" from the JD text (this might be best handled by a focused instruction within the main prompt or a separate, small pre-processing call to Gemini for the JD text before the main CV-JD matching prompt).

- **Main Prompt (for CV analysis and CV-JD matching with Gemini 1.5 Flash):**

Plaintext

SYSTEM: You are an expert AI Recruitment Analyzer. Your task is to meticulously parse the provided Candidate CV and Job Description (JD). First, extract key information from the CV. Then, analyze the CV against the JD to determine a match score, assign a ranking category, and provide a concise but thorough ****reasoning**** for your assessment. Finally, return all information as a single, valid JSON object.

USER:

Please process the following Candidate CV and Job Description.

****Candidate CV Text:****

[CV_TEXT_PLACEHOLDER]

****Job Description (JD) Text (Key details like Position Title:**

"[EXTRACTED_JD_TITLE_PLACEHOLDER]" and Job ID: "[EXTRACTED_JD_ID_PLACEHOLDER]" are provided for context):**

[JD_FULL_TEXT_PLACEHOLDER]

****Instructions & Output Format:****

Based on the CV and JD provided above, perform the following:

1. ****Extract Candidate Profile Information from the CV:****

* `full_name`: Extract the candidate's full name. If not clearly identifiable, use "N/A".

* `email`: Extract the candidate's primary email address. If multiple are present, choose the most professional or primary one. If none is found, use "N/A".

* `contact_number`: Extract the candidate's primary phone number. If none is found, use "N/A".

* `skills`: Extract a list of key skills (technical skills, software proficiency, soft skills, languages, certifications, etc.) mentioned in the CV. Return these as an array of strings. If no specific skills section is found, try to infer key skills from the experience section.

* `experience_summary`: Provide a concise summary of the candidate's work experience. This should ideally include their most recent or most relevant roles, the companies they worked for, the duration of these roles (if specified), and a brief highlight of their key responsibilities or achievements in those roles. Aim for a summary that gives a good overview of their career trajectory and expertise. If the experience section is sparse, summarize what is available.

2. ****Perform CV-JD Matching, Scoring, and Ranking:****

* `match_score`: Evaluate the candidate's overall suitability for the role described in the JD. Consider the alignment of their skills, the relevance and duration of their experience, and their educational background against the requirements and preferences stated in the JD. Provide a numerical score between 0 and 100, where 100 represents a perfect match.

* `ranking_category`: Based on your `match_score` and overall qualitative assessment, assign a ranking category from the following options: "High Fit", "Medium Fit", "Low Fit".

* "High Fit": Strong alignment with most key requirements, likely a good candidate for an interview (typically scores 80-100).

* "Medium Fit": Aligns with several requirements but may have some gaps or less direct experience (typically scores 55-79).

* "Low Fit": Significant misalignment with key requirements (typically scores below 55).

* `ranking_reason`: Provide a detailed yet concise (2-4 sentences) ****reasoning statement**** for the assigned `match_score` and `ranking_category`. This reason should clearly articulate the AI's thought process, highlighting the most critical factors from the CV that align with the JD, as well as any significant gaps or areas where the candidate does not meet the JD requirements.

****Required Output Format (for CV-JD matching):****

Return your complete analysis as a single, valid JSON object. Do not include any explanatory text or headers outside of this JSON object.

```
```json
```

```
{
```

```
 "full_name": "string_or_N/A",
```

```

"email": "string_or_N/A",
"contact_number": "string_or_N/A",
"skills": ["string_skill1", "string_skill2", "..."],
"experience_summary": "string_summary_of_experience",
"match_score": integer_0_to_100,
"ranking_category": "string_High Fit_or_Medium Fit_or_Low Fit",
"ranking_reason": "string_detailed_explanation_and_reasoning"
}

```

Please ensure all string values within the JSON are properly escaped if necessary.

## V. Deliverables

1. **Python Application:** A fully functional Python script (or set of organized modules) executable via the CLI as per the specified functionality, including CLI loading indicators and initial guidance.
2. **Dependency File:** A requirements.txt file listing all necessary Python libraries and their versions.
3. **Configuration Template:** A .env.example file indicating all necessary environment variables.
4. **Source Code:** Well-commented and organized Python source code adhering to specified coding standards, committed regularly to the client-provided Git repository.
5. **Unit Test Suite:** Python unit tests (e.g., using unittest or pytest frameworks) covering critical functions and modules, including logic for text extraction, API interactions (mocked), and data parsing.
6. **Setup and Usage Instructions:** A comprehensive README.md file explaining:
  - How to set up the Python virtual environment and install dependencies.
  - How to configure the .env file with API keys and Notion details.
  - How the CLI's initial guidance system works.
  - Clear instructions on how to run the CLI script with example commands and expected arguments.
  - Instructions on how to run the unit tests.

## VI. Acceptance Criteria (for Project Completion)

The project will be considered complete when the following criteria are met:

1. The application can be successfully executed via the CLI, accepting parameters for the CVs folder, JD PDF file, and Notion Database ID.
2. The CLI displays user-friendly **initial setup/guidance instructions** and **loading indicators or a progress bar** during the CV processing loop.
3. Text is successfully extracted from the provided PDF JD file.
4. "Position Title" and "Job ID" are correctly extracted from the JD content and stored as distinct properties in each relevant Notion candidate entry.
5. The application correctly extracts text from sample digital PDF and DOCX CV files provided by the client.
6. The application successfully interacts with the Google **Gemini 1.5 Flash** API, sending the CV and

JD data, and receiving a structured JSON response as per the defined format, including the **reasoning** for rankings.

7. The application successfully interacts with the Notion API, creating new entries in the client-specified Notion database (skipping entries identified as duplicates per the defined logic).
8. All properties in the Notion database (as listed in Section 4) are correctly populated with data from the Gemini API response and script-generated information.
9. The application demonstrates **batch processing** of at least 5-10 diverse sample CVs against 1-2 sample JDs provided by the client, with accurate data population in Notion and **without excessive memory consumption** (i.e., operates efficiently on a standard developer machine without crashing or significant slowdown due to memory).
10. The application adheres to **general Python coding best practices** (e.g., PEP 8 guidelines, clear variable names, logical code structure, modular design where appropriate) and implements **security best practices** for handling API keys and sensitive data (e.g., exclusively via environment variables, no hardcoding of secrets).
11. **Unit tests** for critical functions (e.g., file processing, data transformation, core logic of API call preparation) are provided, achieve reasonable coverage (e.g., >70% for tested modules where applicable for a PROJECT), and pass successfully.
12. The application handles basic errors gracefully (e.g., file not found, API connection issue, individual CV processing error) by logging an error message to the console and continuing with the next CV where possible, or exiting cleanly for critical setup errors.
13. All code and documentation are committed to the client-provided Git repository daily.
14. The project is delivered within the specified timeline.

## VII. Assumptions

- The Client will provide valid API keys for the Google Gemini service (specifically for use with **Gemini 1.5 Flash**) and a Notion integration token with appropriate permissions.
- The Client will provide the Notion Database ID for an existing, pre-configured database matching the property structure outlined in Section 4.
- The Client will provide a set of sample digital (not scanned image-based) CVs (in PDF and DOCX format) and 1-2 sample Job Descriptions (as PDF files) for development and testing purposes.
- The Client will provide access to a **designated Git repository** (e.g., GitHub, GitLab) where all project code and documentation will be committed and managed. The developer will be required to work within this provided repository.
- The Client understands that API responses, capabilities, and specific model version identifiers for **Gemini 1.5 Flash** are subject to Google's offerings and potential updates. The developer will use the most appropriate and available version of **Gemini 1.5 Flash** at the time of development.
- This PROJECT focuses on functionality with digitally created documents. Complex OCR for scanned, image-based, or handwritten CVs is explicitly out of scope.
- The primary objective is functional validation. While coding standards, unit tests, and best practices are required, the depth and breadth of these for a 14-working-day PROJECT will be focused on core functionality rather than exhaustive production-level hardening.



## VIII. Intellectual Property (IP) Ownership

All original source code, scripts, documentation, and any other intellectual property created or developed during this project shall be the sole and exclusive property of the Client, [Your Name/Company Name]. The developer/company agrees not to reuse, resell, redistribute, fork, or claim any ownership rights over the project deliverables, methodologies, or specific implementations developed for this project, outside the scope of fulfilling this agreement. All rights, title, and interest in and to the work product will be irrevocably assigned to the Client upon completion and final payment.

## IX. Timeline & Reporting

- The timeline for the development and delivery of this project is strictly **14 working days** from the project commencement date.
- The developer/company is required to provide a status update to the Client via email or a designated communication channel at the **End of Day (EOD) every working day**. This update should include tasks completed, progress against the timeline, any challenges encountered, and the plan for the next working day.
- All code must be committed to the client-provided Git repository daily.

## X. Appendix: Sequential Diagram

vNEX9TaIPPEVIO81zFyDxwke0y1Xq2UH23h1FEItG6UB\_JahXkFQnui2ZaX\_0mYX-  
OjdYYVchuvQSQsZnXRnHbKxwmamJc5avv3lX7GZujBHGW62TVv2R5Gt7cYq464EWp5W7U2  
9ms9bmaYYtilmrwavnF3tRJ8iwlxSxASIWvidhjDyfgUSWiPjNMKY6MLwiFsibNPIxt1RgWDEKvN  
plk6W0rboJjt85dEmCmVzDeDemFbMVaOhfblyC5cp9WZHkqo9fJN2yamalnX8aNrhnCf\_ZLNO  
1sdu\_8GdEUGHEaju\_hpve7VbMFfcVkhNe8\_K6OqvESrsqLRgTxJS5uab0zU-  
vSyg1GspzzuXd2ui24sqcJR4YtjQWr5gqnSVAessJRbheiDcaH7ZQR50BBpFU39Ns4deb630DL0  
BnT7oO\_FSNTx0Htg6l6T6qTeg1xDngqoQb\_-RzOj\_4t9KxaWIVtli6Q3mgrLk99xhFP-  
8qyUUBuohFQjrDY7PTk5zEG\_w4N3T-  
OS4e\_6q33\_ZPzk77532zujryhscvuqddo97J2cvX5\_2Ts57\_fNH3\_ue-z3unvbPXr\_sn5-  
f9F6\_6vX7\_cf\_ATrC3ws

