Rapport d'Analyse du Projet : Job Application Assistant

Maxim Quénel | Pierre Louis Brun | Mame Alé Seye | Denis Bereziuc

https://github.com/Maxim-Quenel/job_app_assistant_matching_CV

28 décembre 2025

Résumé

Ce document présente une analyse détaillée du projet d'assistant de candidature automatisé, couvrant son contexte, son architecture, ses choix techniques, le traitement des données et les perspectives d'évolution.

Table des matières

1 Contexte du Projet

Le projet Job Application Assistant est une application web locale conçue pour automatiser et optimiser le processus de recherche d'emploi. Il vise à réduire la charge cognitive et temporelle liée à la personnalisation des candidatures en :

— Récupérant automatiquement des offres d'emploi ciblées.

— Analysant et résumant ces offres pour en extraire les points clés.

— Restructurant le CV du candidat pour l'anonymiser et le standardiser.

— Calculant un score de pertinence (matching) entre le CV et chaque offre pour prioriser les meilleures opportunités.

2 Architecture Technique

L'application repose sur une architecture monolithique modulaire, orchestrée par un serveur web léger.

2.1 Composants Principaux

Backend (Serveur) : Développé en Python avec le framework Flask. Il expose des endpoints API (de `/api/step1` à `/api/step6`) pour déclencher les différentes étapes du pipeline.

Frontend (Interface) : Pages HTML servies par Flask (`render_template`), interagissant avec l'API via JavaScript pour lancer les tâches et afficher la progression.

Task Management : Utilisation du module `threading` natif de Python pour exécuter les tâches longues (IA, scraping) en arrière-plan sans bloquer l'interface utilisateur.

Persistance des Données : Système de fichiers simple (CSV et TXT) stockés dans un dossier local `data/`. L'absence de base de données relationnelle complexe simplifie le déploiement et le débogage.

2.2 Flux de Contrôle

Le flux est séquentiel mais découpé en étapes indépendantes, permettant à l'utilisateur de valider chaque phase intermédiaire :

$$\text{Scraping} \rightarrow \text{Job Rewrite} \rightarrow \text{CV Convert} \rightarrow \text{CV Rewrite} \rightarrow \text{Matching}$$

3 Choix de Conception

— **Approche "Local-First" et Confidentialité :** Le choix d'utiliser des modèles d'IA exécutés localement via la librairie transformers (plutôt que des API cloud payantes) garantit la confidentialité des données sensibles (CVs) et l'indépendance vis-à-vis des coûts récurrents.

— **Modularité des Services :** Chaque fonction métier est isolée dans un module dédié (ex : `services/scraper.py`, `services/job_rewriter.py`). Cela facilite la maintenance, les tests unitaires et le remplacement potentiel d'un composant.

— **État basé sur des fichiers :** L'utilisation de fichiers tels que `jobs_raw.csv`, `jobs_rewritten.csv` ou `cv_synthesized.txt` permet une transparence totale. L'utilisateur peut ouvrir ces fichiers pour vérifier ou corriger les données manuellement.

4 Traitement des Données (Data Pipeline)

Le pipeline de données transforme des informations non structurées en scores exploitables :

Ingestion (Étape 1)

— Les offres sont scrapées ou saisies par texte brut.

— Stockage initial dans le fichier `jobs_raw.csv`.

Normalisation (Étape 2 & 4)

— **Offres :** Le LLM nettoie et structure les descriptions (compétences, missions, infos clés) dans une nouvelle colonne "Resume_IA".

— **CV :** Conversion PDF vers TXT, suivie d'une réécriture par LLM pour standardiser le format et anonymiser les données personnelles.

Matching (Étape 5 & 6)

— Construction de paires (Texte du CV, Texte de l'Offre).

— Les offres "enrichies" combinent le poste, l'entreprise et le résumé IA pour donner un contexte maximal au modèle de matching.

5 Choix du Fournisseur LLM & Modèles

Le projet utilise des modèles Open Source performants exécutés via Hugging Face Transformers.

5.1 Modèle Génératif (Mise en forme)

— **Modèle :** Qwen/Qwen2.5-1.5B-Instruct

— **Justification :** Il s'agit d'un "Small Language Model" (SLM) très performant pour sa taille (1.5 milliard de paramètres). Il est suffisamment léger pour tourner sur des GPU grand public tout en offrant une excellente capacité de suivi d'instructions.

— **Utilisation :** Réécriture des CVs et des offres d'emploi avec une température faible (0.1/0.2) pour assurer la stabilité.

5.2 Modèles de Matching (Scoring)

Le projet implémente une stratégie de matching à deux niveaux :

5.2.1 Approche Classique (Bi-Encoder)

- **Modèle :** BAAI/bge-m3
- **Type :** Sentence Transformer.
- **Fonctionnement :** Conversion indépendante du CV et des offres en vecteurs. Similarité calculée par cosinus.
- **Rôle :** Premier filtre rapide (Étape 5).

5.2.2 Approche Avancée (Cross-Encoder/Reranking)

- **Modèle :** BAAI/bge-reranker-v2-m3
- **Type :** Cross-Encoder.
- **Fonctionnement :** Le modèle analyse le CV et l'offre ensemble pour estimer une similarité sémantique fine.
- **Rôle :** Affinement final plus précis (Étape 6), produisant un score de pertinence entre 0 et 100%.

6 Évaluation

Actuellement, l'évaluation du système est qualitative et empirique : le système affiche les "Top Matches" et l'utilisateur juge de la pertinence en consultant les liens ou les résumés générés.

7 Améliorations Futures Recommandées

7.1 Technique

- **File de Tâches Robuste :** Remplacer threading par une solution comme Celery ou RQ (avec Redis) pour gérer les échecs et les tentatives de relance.
- **Base de Données :** Migrer des fichiers CSV vers SQLite pour permettre des requêtes complexes (filtrage par date, score, entreprise).
- **Optimisation Inférence :** Implémenter la quantification (4-bit via bitsandbytes) pour accélérer le chargement des modèles et réduire la consommation de VRAM.

7.2 Fonctionnel

- **Feedback Loop :** Ajouter un système de notation ("Pouce haut/bas") sur les résultats pour affiner les futurs modèles selon les préférences de l'utilisateur.
- **Génération de Lettre de Motivation :** Ajouter une étape utilisant le modèle Qwen pour rédiger une lettre personnalisée basée sur le CV et l'offre sélectionnée.
- **Interface de Filtrage :** Permettre le filtrage des résultats finaux par score minimum ou mots-clés (ex : exclusion de certains types de contrats).

7.3 Qualité du Code

— **Gestion des Erreurs :** Renforcer les blocs try/except autour des appels réseaux et chargements de modèles.

— **Configuration :** Centraliser les constantes (modèles, chemins) dans un fichier de configuration (`config.py`) ou des variables d'environnement.

Documentation Technique : Architecture du Projet

8 Objectif

Le projet est une application web Flask conçue pour automatiser le processus de recherche d'emploi. Elle orchestre un flux complet allant de la récupération des offres à l'analyse de pertinence via l'Intelligence Artificielle (LLM). L'interface propose un parcours utilisateur en 6 étapes distinctes, incluant la visualisation des logs en temps réel et des prévisualisations de données.

9 Structure du Dépôt

L'organisation des fichiers suit une logique modulaire :

- `app.py` : Point d'entrée de l'application Flask. Gère les routes API et l'orchestration des étapes via des threads.
- `services/` : Contient toute la logique métier (scraping, parsing JSON, réécriture LLM, algorithmes de matching).
- `utils/` : Utilitaires transverses, notamment le logger applicatif pour l'interface utilisateur.
- `templates/index.html` : Interface web unique composée de 6 cartes interactives et d'un panneau de logs.
- `static/` : Ressources frontend (CSS et JS) gérant les interactions, le polling et les prévisualisations.
- `data/` : Dossier de stockage pour les fichiers intermédiaires (CSV, TXT) et les résultats finaux.
- `msedgedriver.exe` : Driver Selenium nécessaire pour le pilotage du navigateur Edge.

10 Flux Fonctionnel (Pipeline)

Le traitement des données suit un pipeline séquentiel :

Étape 1 : Collecte des Offres

- **Méthodes :**
  - Scraping : Automatisation via `services/scraper.py` (cible : HelloWork).
  - Texte Brut : Importation manuelle via `services/raw_job_parser.py`.
- **Sortie :** `data/jobs_raw.csv`

Étape 2 : Réécriture des Offres (Enrichissement)

- **Traitement :** Utilisation du modèle Qwen2.5-1.5B-Instruct via `services/job_rewriter.py`.
- **Action :** Résumé et structuration des offres.
- **Sortie :** `data/jobs_rewritten.csv` (ajout de la colonne Resume_IA).

Étape 3 : Conversion du CV

- **Traitement :** Extraction du texte depuis le PDF via pdfplumber et nettoyage avec `services/cv_converter.py`.
- **Sortie :** `data/cv_converted.txt`

Étape 4 : Synthèse du CV

- **Traitement :** Réécriture et normalisation du profil candidat par le LLM via `services/cv_rewrite`
- **Sortie :** `data/cv_synthesized.txt`

Étape 5 : Matching Sémantique

- **Traitement :** Calcul de similarité cosinus sur les embeddings générés par le modèle BAAI/bge-m3 (`services/matcher.py`).
- **Sortie :** `data/final_matches.csv` (Score 0-100).

Étape 6 : Cross-Matching (Reranking)

- **Traitement :** Affinement de la pertinence via un Cross-Encoder BAAI/bge-reranker-v2-m3 (`services/cross_encoder_matcher.py`).
- **Sortie :** `data/final_matches_cross.csv`

11 Backend (Flask)

Le fichier `app.py` centralise la logique serveur.

11.1 Routes Principales

- `GET /` : Chargement de l'application principale.
- `GET /api/logs` : Récupération de l'état des tâches et des logs pour l'UI.
- `GET /api/preview/step1..step6` : Affichage d'échantillons de données pour validation.
- `GET /api/files/<filename>` : Téléchargement des fichiers générés dans `data/`.
- `POST /api/step1` à `/api/step6` : Déclencheurs des traitements pour chaque étape.
- `POST /api/step3/upload` : Endpoint dédié à l'upload du CV PDF.

11.2 Orchestration

- **Fonction** `run_task()` : Exécute chaque étape dans un thread séparé pour éviter de bloquer le serveur Flask pendant les traitements longs (IA, scraping).
- **Logger** (`utils/logger.py`) : Centralise le statut des opérations et alimente l'interface en temps réel.

12 Services (Logique Métier)

Chaque script dans le dossier `services/` a une responsabilité unique :

— `scraper.py` : Pilotage de Selenium (Edge) pour extraire liens, missions et profils.

— `raw_job_parser.py` : Conversion de texte brut (copier-coller) en JSON structuré via Qwen.

— `job_rewriter.py` : Génération de résumés structurés pour les offres d'emploi.

— `cv_converter.py` : Extraction brute PDF vers Texte et nettoyage préliminaire.

— `cv_rewriter.py` : Création d'une synthèse standardisée du CV via LLM.

— `matcher.py` : Matching rapide (Bi-Encoder) via embeddings et similarité cosinus.

— `cross_encoder_matcher.py` : Matching de précision (Cross-Encoder) pour le reclassement final.

13 Frontend

L'interface est gérée par `templates/index.html` et `static/js/main.js`.

— **Structure :** 6 cartes correspondant aux étapes du pipeline + un terminal de logs.

— **Mécanique :** Polling sur `/api/logs` toutes les secondes pour mettre à jour la progression.

— **Visualisation :** Les appels à `/api/preview/stepX` permettent à l'utilisateur de vérifier les résultats intermédiaires sans quitter l'interface.

— **Automatisation :** L'upload du PDF déclenche automatiquement l'étape de conversion (Step 3).

14 Données et Artefacts

Tous les fichiers sont stockés localement dans le dossier `data/` :

— `jobs_raw.csv` : Données brutes (issues du scraping ou du parsing texte).

— `jobs_rewritten.csv` : Données enrichies avec les résumés IA.

— `cv_converted.txt` : Extraction brute du CV.

— `cv_synthesized.txt` : Profil structuré et anonymisé.

— `final_matches.csv` : Résultats du premier filtre sémantique.

— `final_matches_cross.csv` : Classement final par pertinence (Reranker).

15 Dépendances Clés

L'application repose sur la stack technique suivante :

— Backend : Flask

— Manipulation de données : Pandas (I/O CSV)

— Scraping : Selenium + Edge Driver

— Traitement PDF : pdfplumber

— IA & NLP : Transformers, Torch (pour Qwen2.5), Sentence-Transformers, Scikit-learn