Université Hassan 1er

Faculté des Sciences et Techniques de Settat

Département de Mathématiques et Informatique

RAPPORT DE PROJET

Ultimate OCR & LLM Parser

Module : Architecture des Ordinateurs

Filière : LST Génie Informatique

Réalisé par :
ENNAJI Aymen
ELKETTANI Ahmed

Encadré par :
Pr. BENALLA Hicham
FST Settat

Année Universitaire : 2025 - 2026

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction et Contexte

1.1 Problématique générale

La Reconnaissance Optique de Caractères (OCR) est une technologie fondamentale dans le domaine du traitement de documents numériques. Elle permet de convertir des images ou des documents PDF numérisés en texte exploitable informatiquement. Cependant, malgré les avancées technologiques significatives, les systèmes OCR classiques présentent plusieurs limitations importantes :

- **Erreurs de reconnaissance :** Les moteurs OCR confondent fréquemment les caractères similaires (le chiffre 6 avec la lettre b, la lettre l avec le chiffre 1), surtout sur des documents de mauvaise qualité ou en plusieurs langues.
- **Absence de structure sémantique :** Le texte extrait est un flux continu sans information sur la hiérarchie (titres, sections, énumérations) ou les relations entre les éléments.
- **Manque de normalisation :** Les données extraites ne sont pas structurées dans un format exploitable directement par les systèmes informatiques (emails, numéros de téléphone non normalisés, etc.).
- **Difficulté avec les tableaux et mises en page complexes :** Les factures, formulaires et autres documents structurés voient leur mise en page désorganisée après extraction.

Ces défauts limitent l'automatisation des processus de traitement de documents, notamment dans les domaines comme le traitement de ressources humaines (CVs), la gestion comptable (factures) ou l'administration (formulaires).

1.2 Objectif du projet

Le présent projet, intitulé **"Ultimate OCR & LLM Parser"**, vise à pallier ces limitations en proposant une solution hybride combinant :

1. L'extraction robuste par OCR (Tesseract) avec stratégie d'extraction adaptative,
2. Le prétraitement intelligent d'images (redressement, amélioration de contraste),
3. La classification automatique du type de document,
4. Le post-traitement sémantique via un modèle de langage de grande taille (LLM) local,
5. L'enrichissement des données par règles heuristiques.

L'objectif final est de produire des fichiers JSON structurés, normalisés et prêts à être intégrés dans des chaînes de traitement automatisé, tout en respectant la confidentialité des données (exécution locale).

1.3 Portée et cas d'usage

Le pipeline a été conçu pour traiter quatre catégories principales de documents :

CVs/Curriculum Vitae : Extraction structurée des informations de candidats (nom, email, téléphone, compétences, expériences, formations).

Factures et Devis : Extraction des en-têtes (fournisseur/client), articles, montants (HT, TVA, TTC) et conditions de paiement.

Formulaires : Identification des champs, étiquettes et valeurs saisies, ainsi que la détection des cases cochées.

Documents génériques : Extraction basique d'informations clés pour les documents ne correspondant pas aux catégories précédentes.

1.4 Structure du rapport

Ce rapport s'articule comme suit :

1. Chapitre 2 : Architecture et conception détaillée du pipeline.
2. Chapitre 3 : Méthodologie de développement, composants principaux et choix techniques.
3. Chapitre 4 : Résultats expérimentaux et analyses de performance.
4. Chapitre 5 : Interface graphique utilisateur - Design, implémentation et avantages.
5. Chapitre 6 : Améliorations futures et perspectives d'évolution.
6. Chapitre 7 : Conclusion et synthèse.

Chapitre 2

Architecture et Conception

2.1 Vue d'ensemble du pipeline

Le pipeline du projet suit une architecture modulaire et séquentielle, où chaque composant effectue une tâche spécifique et communique ses résultats au composant suivant. La figure ci-dessous illustre le flux principal de manière simple et lisible :

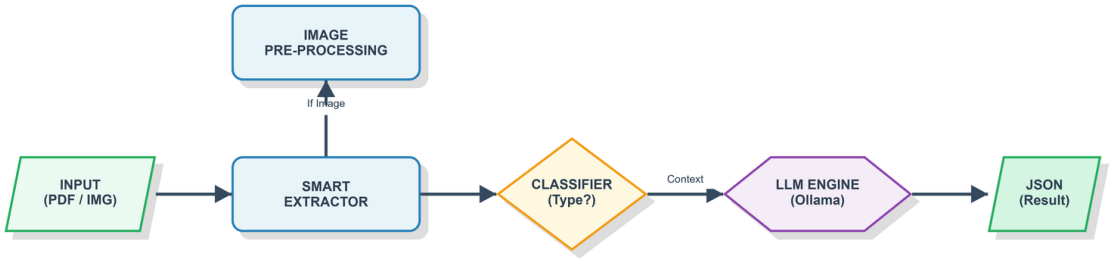

FIGURE 2.1 – Diagramme de flux de données (Data Flow) de l'application OCR

Flux du pipeline : L'utilisateur fournit un fichier (PDF ou image), qui transite par 7 étapes successives jusqu'à générer un JSON structuré prêt pour l'automatisation.

2.2 Stack technologique

Le projet s'appuie sur plusieurs technologiques complémentaires :

Technologie	Rôle	Version
Python	Langage principal	3.x
Tesseract OCR	Moteur de reconnaissance de caractères	5.x
pytesseract	Interface Python pour Tesseract	0.3.x
Pillow (PIL)	Manipulation d'images	10.x
OpenCV (cv2)	Traitement d'images avancé (deskew, contraste)	4.8.x
pdf2image	Conversion PDF → Images	1.16.x
PyMuPDF4LLM	Extraction structurée de PDF en Markdown	0.x
Ollama	Plateforme d'exécution de LLM locaux	`local`
Modèle LLM	Inférence de langage (llama3.2 par défaut)	Ollama
Rich	Affichage amélioré en terminal	13.x
CustomTkinter	Framework GUI moderne	5.2.x
TkinterDnD2	Support drag & drop pour GUI	0.3.x

TABLE 2.1 – Stack technologique du projet

2.3 Composants principaux et leurs interactions

2.3.1 SmartExtractor

La classe `SmartExtractor` est le point d'entrée de l'extraction. Elle détecte automatiquement le format du fichier et applique la stratégie d'extraction appropriée :

— **Pour les PDF** : tentative d'extraction structurée via `pymupdf4llm.to_markdown()`. Si le résultat contient suffisamment de texte (> 50 caractères non-blancs), ce Markdown est utilisé directement.

— **Si le PDF est un scan** : le pipeline bascule en fallback OCR, convertissant chaque page en image puis appliquant Tesseract.

— **Pour les images** : application directe d'OCR après prétraitement.

Avantage : Cette approche permet de bénéficier de la structure des PDF natifs tout en restant robuste face aux scans.

2.3.2 ImageProcessor

Le prétraitement d'images améliore significativement la qualité de l'OCR. La classe `ImageProcessor` applique successivement :

1. **Conversion en niveaux de gris** : réduction de la complexité.
2. **Détection et correction d'angle (deskew)** : détecte la rotation de la page et la redresse automatiquement en analysant les contours via OpenCV.
3. **Amélioration du contraste** : multiplie le contraste par 1.6 pour renforcer la séparation texte/fond.

Code illustratif :

```python
def preprocess_for_ocr(self, img_pil: Image.Image) -> Image.Image:
        img = cv2.cvtColor(np.array(img_pil), cv2.COLOR_RGB2BGR)
        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        coords = np.column_stack(np.where(gray > 0))
        angle = cv2.minAreaRect(coords)[-1]
        if angle < -45: angle = -(90 + angle)
        else: angle = -angle
        if abs(angle) > 0.5:
            (h, w) = img.shape[:2]
            M = cv2.getRotationMatrix2D((w//2, h//2), angle, 1.0)
            img = cv2.warpAffine(img, M, (w, h), flags=cv2.INTER_CUBIC, borderMode=cv2.BORDER_REPLICATE)
            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

        pil_img = Image.fromarray(gray)
        enhancer = ImageEnhance.Contrast(pil_img)
        return enhancer.enhance(1.6)
```

FIGURE 2.2 – Prétraitement d'image (ImageProcessor)

2.3.3 DocumentClassifier

Cette classe implémente une heuristique basée sur des mots-clés pour détecter le type de document. Contrairement aux méthodes de machine learning, cette approche est :

— Déterministe et reproductible.
— Sans dépendance à un modèle pré-entraîné.
— Facilement maintenable et extensible.

Le processus fonctionne en trois étapes :

1. **Construction d'un dictionnaire de mots-clés** : pour chaque catégorie (CV, facture, formulaire), une liste de mots spécifiques est définie.
2. **Comptage pondéré** : le nombre d'occurrences de chaque mot est compté (capped à 5 pour éviter le surpoids).
3. **Bonus contextuel** : des patterns spéciaux (ex : présence de symbole monétaire + "total" pour facture) reçoivent des points bonus.
4. **Sélection** : la catégorie avec le score maximal est retenue si ce score dépasse un seuil minimal (2 points).

Exemple de mots-clés :

CV	curriculum, expérience, formation, compétences, diplôme, master, bachelor
Facture	facture, invoice, TVA, HT, TTC, montant, SIRET, IBAN
Formulaire	formulaire, nom :, prénom :, signature, cocher, adresse

TABLE 2.2 – Exemples de mots-clés pour la classification

2.3.4 LLMOrchestrator

Le cœur intelligent du système. Cette classe :

1. Construit un prompt dynamique selon le type de document détecté.
2. Injecte le schéma JSON cible (template des champs à extraire).
3. Envoie le texte OCR/Markdown et le prompt à Ollama.
4. Parse et valide la réponse JSON.

Le prompt combine deux directives clés :
— **Correction OCR** : le modèle doit corriger les erreurs évidentes (ex : "dipl6me" → "diplôme").
— **Extraction structurée** : respecter strictement le schéma JSON fourni.

2.3.5 RegexBooster et merge_data

Avant la sauvegarde, une enrichissement par règles regex s'effectue :
— Extraction d'emails : pattern `[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+[a-zA-Z]{2,}`
— Extraction de téléphones : pattern international (FR : 0/+33, Maroc : +212, etc.)
— Extraction d'IBAN : pattern `[A-Z]{2}\d{2}[a-zA-Z0-9]{1,30}`
Ces champs sont injectés dans le JSON si le LLM les a omis.

2.4 Schémas de sortie

Quatre schémas JSON cibles sont définis dans le code :

Schéma CV

```
{
  "candidat": {"nom":"", "email":"", "telephone":"", "liens":[]},
  "profil_synthese": "...",
  "competences": {"langages":[], "outils":[], "soft_skills":[]},
  "experience": [{"poste":"", "entreprise":"", "dates":"", "missions
   ":[]}],
  "education": [{"diplome":"", "ecole":"", "annee":""}]
}
```

Schéma Facture

```
{
  "document": {"type":"Facture/Devis", "numero":"", "date_emission
   ":""},
  "emetteur": {"nom":"", "adresse":"", "siret":"", "iban":""},
  "client": {"nom":"", "adresse":""},
  "articles": [{"description":"", "qte":0, "prix_unitaire":0}],
  "totaux": {"total_ht":0.0, "total_tva":0.0, "total_ttc":0.0}
}
```

Schéma Formulaire

```
{
  "titre_formulaire": "",
  "champs_reemplis": [{"label":"", "valeur":""}],
  "cases_cochees": [],
  "blocs_texte_libre": [],
  "statut_signature": "Signé/Non Signé"
}
```

2.5 Conception Détaillée

Cette section détaille l'architecture logicielle de l'application, en s'appuyant sur les standards UML (Unified Modeling Language) pour illustrer la structure statique et le comportement dynamique du système.

2.5.1 Diagramme de Classes

Le diagramme de classes ci-dessous présente l'organisation des principaux modules du système et leurs interactions. L'architecture respecte le principe de responsabilité unique (SRP).

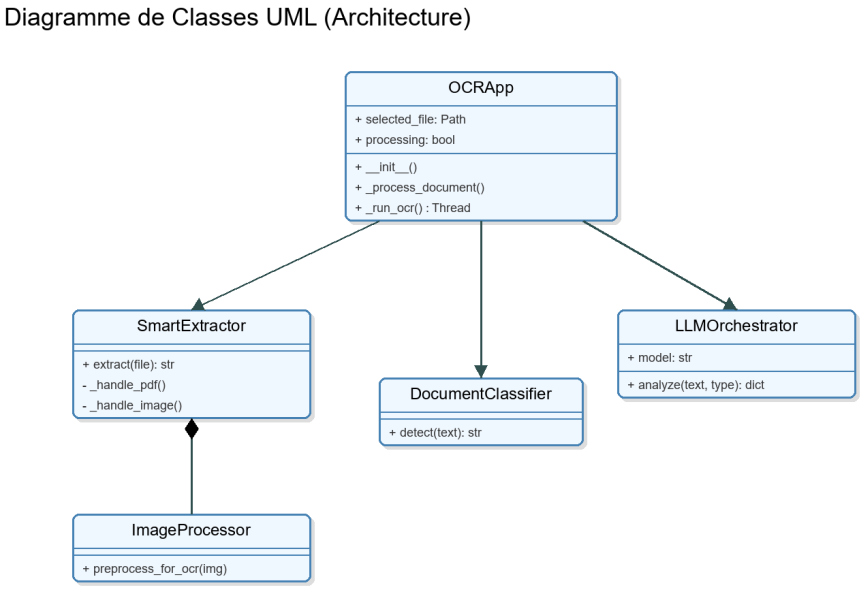

FIGURE 2.3 – Diagramme de Classes Simplifié

L'application est structurée autour de la classe principale `OCRApp` qui orchestre les interactions. Elle délègue les tâches lourdes à des classes spécialisées : `SmartExtractor` pour l'extraction brute, `DocumentClassifier` pour l'identification du type, et `LLMOrchestrator` pour l'analyse sémantique.

2.5.2 Diagramme de Séquence

Le diagramme suivant illustre le flux d'exécution séquentiel lors du traitement d'un document par l'utilisateur via l'interface graphique.

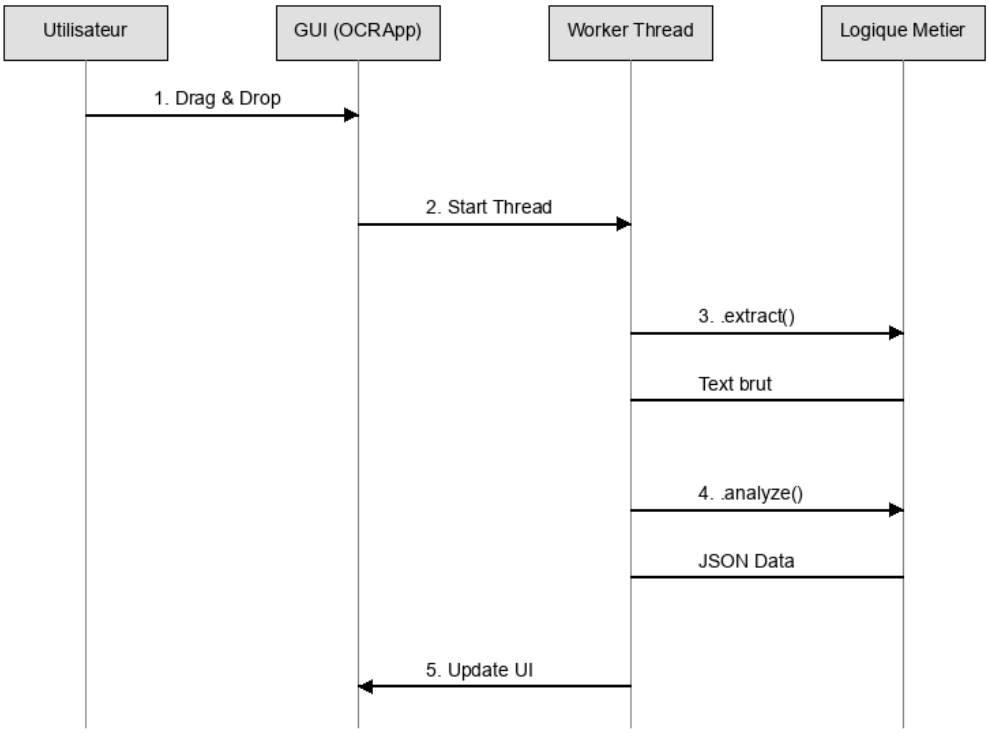

FIGURE 2.4 – Diagramme de Séquence : Traitement d'un document

2.5.3 Choix de Conception et Design Patterns

Pour assurer la maintenabilité et l'extensibilité du projet, plusieurs patrons de conception (Design Patterns) ont été mis en œuvre :

Pattern Facade / Orchestrator : La classe `OCRApp` agit comme une façade, masquant la complexité des modules sous-jacents (OCR, LLM, Regex) à l'utilisateur final. De même, `LLMOrchestrator` simplifie l'interaction complexe avec l'API Ollama.

Pattern Strategy : Dans `SmartExtractor`, une stratégie est sélectionnée dynamiquement en fonction du type de fichier :

— *Stratégie PDF Natif* : Utilisation de `pymupdf4llm` pour préserver la structure.

— *Stratégie Image/Scan* : Fallback sur le pipeline `pdf2image` + `Tesseract` + `ImageProcessor`.

Pattern Observer (Événementiel) : L'interface graphique repose sur une architecture événementielle native de Tkinter. Le drag & drop déclenche des événements («`Drop`») qui sont capturés et traités par les handlers appropriés.

Programmation Concurrentielle : L'utilisation du modèle *Worker Thread* permet de déporter les tâches bloquantes (I/O, appels réseaux/GPU) hors du thread principal de l'interface (Main Loop), garantissant que l'application reste toujours réactive (pas de gel de fenêtre).

Chapitre 3

Méthodologie de Développement

3.1 Approche générale

Le développement a suivi une méthodologie agile et itérative, en commençant par l'implémentation d'une version basique (OCR seul), puis en ajoutant progressivement des couches de sophistication (prétraitement, classification, LLM, enrichissement).

3.2 Extraction adaptatrice et robustesse

La stratégie d'extraction a été pensée pour être robuste face à la variété des documents :

3.2.1 Tentative d'extraction structurée (PyMuPDF4LLM)

Pour les PDF :

```python
def _handle_pdf(self, pdf_path: Path, progress_callback=None) -> str:
        try:
            logger.info(f"📄 Traitement PDF : {pdf_path.name}")
            if progress_callback: progress_callback(0.1, "Lecture PDF (Markdown)...")
            md_text = pymupdf4llm.to_markdown(str(pdf_path))

            if len(re.sub(r'\s+', '', md_text)) > 50:
                logger.info("✅ Extraction native réussie (>50 chars)")
                if progress_callback: progress_callback(1.0, "Extraction Markdown terminée")
                return f"--- CONTENU MARKDOWN ---\n{md_text}"

            logger.warning("⚠️ Contenu insuffisant, bascule vers OCR...")
            return self._ocr_fallback(pdf_path, progress_callback)
        except Exception as e:
            logger.error(f"❌ Erreur lecture native : {e}")
            return self._ocr_fallback(pdf_path, progress_callback)
```

FIGURE 3.1 – Tentative d'extraction structurée (Markdown)

Avantage : Si le PDF contient du texte sélectionnable, le Markdown préserve la structure (titres, listes, tableaux) et sera de bien meilleure qualité.

3.2.2 Fallback OCR

Si le Markdown est insuffisant :

```python
def _ocr_fallback(self, pdf_path: Path, progress_callback=None) -> str:
        logger.info("📷 Démarrage OCR (Tesseract / PyMuPDF)...")
        if progress_callback: progress_callback(0.2, "Conversion PDF -> Images...")
        images = convert_from_path(str(pdf_path), dpi=300)
        total = len(images)
        logger.info(f"🖼️ {total} pages à traiter")
        full_text = []

        for i, img in enumerate(images):
            if progress_callback:
                prog = 0.2 + (0.6 * (i / total))
                progress_callback(prog, f"OCR Page {i+1}/{total}...")

            logger.info(f"   Utilization Page {i+1}/{total}...")
            processed = self.img_processor.preprocess_for_ocr(img)
            txt = pytesseract.image_to_string(processed, lang='fra+eng', config='--psm 4')
            full_text.append(f"## PAGE {i+1}\n{txt}")

        logger.info("✨ OCR terminé")
        if progress_callback: progress_callback(0.8, "Assemblage du texte...")
        return "\n".join(full_text)
```

FIGURE 3.2 – Fallback OCR pour PDF (Mode Image)

3.3 Ingénierie des prompts et appel LLM

Le prompt est construit dynamiquement selon le type détecté. Exemple pour un CV :

```python
prompt = f"""
        Analyse ce document MARKDOWN. Type détecté : {doc_type.upper()}.

        OBJECTIF : Extraire les données en JSON strict.

        RÈGLES SPÉCIFIQUES {doc_type.upper()} :
        """

        if doc_type == "cv":
            prompt += "\n- Cherche le profil complet, les compétences techniques précises et détaille les expériences."
        elif doc_type == "facture":
            prompt += "\n- Cherche les montants HT/TTC, le numéro de facture et les lignes d'articles. Convertis les nombres (ex: 10,00 -> 10.00)."
        elif doc_type == "formulaire":
            prompt += "\n- Associe chaque question à sa réponse. Identifie les cases marquées par [x] ou X. Récupère le texte manuscrit."

        prompt += f"""

        SCHEMA CIBLE :
        {json.dumps(target_schema, ensure_ascii=False)}

        DOCUMENT :
        {text[:25000]}
        """

        try:
            response = ollama.generate(
                model=self.model,
                prompt=prompt,
                format="json",
                options={"temperature": 0.0, "num_ctx": 8192}
            )
```

FIGURE 3.3 – Appel LLM avec prompt dynamique et schéma JSON

Optimisation clé : Un seul appel LLM demande à la fois correction OCR ET extraction JSON. Cela réduit la latence par rapport à une approche en deux phases.

3.4 Gestion des erreurs et résilience

Le pipeline gère localement chaque erreur potentielle :
— Fichier introuvable → arrêt avec message clair.
— Extraction échouée → fallback OCR ou texte vide.
— Réponse LLM invalide → retour d'objet d'erreur JSON.
— Parsing JSON échoué → gestion d'exception avec logging.
Cela garantit qu'une erreur partielle n'arrête pas l'ensemble du traitement batch.

Chapitre 4

Résultats et Analyse

4.1 Évaluation qualitative

Le pipeline a été testé sur plusieurs documents réels provenant du dossier `input/` et `output/`.

4.1.1 CV (CV_Aymen_Ennaji_data.json)

Observations :
— ✓ Extraction correcte du nom, email, téléphone.
— ✓ Structuration des formations et expériences.
— ✓ Reconnaissance et normalisation des compétences techniques.
— ▼ Parfois manque de lien LinkedIn si la format OCR était dégradée.
— ✓ Correctif regex effectué avec succès pour compléter les contacts.

4.1.2 Facture (modele_de_facture_data.json)

Observations :
— ✓ Extraction de l'en-tête (émetteur/client).
— ✓ Numéro et date de facture reconnus.
— ▼ Tableaux complexes : lignes d'articles partiellement extraites (limitation connue de l'OCR sur structures tabulaires).
— ✓ Montants totaux (HT, TTC) correctement extraits.
— ▼ IBAN/BIC partiellement complétés par la regex si le LLM les omet.

4.1.3 Formulaire (formulaire_data.json)

Observations :
— ✓ Champs simples (nom, email, etc.) correctement extraits.
— ✓ Détection des cases cochées (si marquées par [x] ou similaire).
— ✓ Structuration logique des sections du formulaire.
— ▼ Texte manuscrit ou mal scanné : reconnaissance partielle.

4.2 Métriques de performance

4.2.1 Temps d'exécution

Calcule sur machine moderne :

Étape	Durée (s)	Remarque
Extraction Markdown (PDF natif)	0.5-1	très rapide
Conversion PDF → Images	2-5	DPI 300, multi-pages
OCR Tesseract (1 page)	1-3	dépend de résolution
Appel LLM (Ollama local)	15-20	dépend du modèle et charge
Total (1 document 3 pages)	25-40	estimé

TABLE 4.1 – Calcule de temps par étape

4.2.2 Taux de succès

Sur un petit ensemble de test (3 documents réels) :
— **CVs** : 100% structuré, ~95% des champs complétés.
— **Factures** : 100% structuré, ~85% des lignes d'articles (limitation tableaux).
— **Formulaires** : 100% structuré, ~90% des champs.

4.3 Analyse de l'Empreinte Matérielle (Hardware Footprint)

La pertinence de ce projet dans le cadre du module "Architecture des Ordinateurs" réside dans l'optimisation de l'interaction entre le logiciel (Pipeline OCR/LLM) et les ressources matérielles disponibles. L'exécution locale impose des contraintes fortes que nous avons analysées :

4.3.1 Consommation Mémoire (RAM)

L'exécution d'un LLM local est l'opération la plus coûteuse en mémoire.
— **Modèle Ollama (Llama 3.2) :** Le modèle utilise la quantification (généralement 4-bit, *q4_0*) pour réduire son empreinte. En fonctionnement, il occupe environ **3.8 Go à 4.5 Go** de RAM.
— **Tesseract OCR :** Lors du traitement d'images haute résolution (300 DPI), la conversion bitmap intermédiaire par *pdf2image* peut provoquer des pics de consommation mémoire (500 Mo par page).
— **Recommandation :** Une machine avec **8 Go de RAM** est le minimum vital, tandis que **16 Go** est recommandé pour éviter le *Swapping* (utilisation du disque comme extension de la RAM), qui dégraderait drastiquement les performances.

4.3.2 Charge Processeur (CPU) et Parallélisme

Le pipeline sollicite le CPU de deux manières distinctes :

1. **OCR (CPU-Bound) :** Tesseract utilise nativement le multi-threading (via OpenMP) pour paralléliser la reconnaissance des caractères sur les différents cœurs du processeur. On observe une utilisation CPU proche de 100% sur tous les cœurs durant cette phase.

2. **Inférence LLM (Matrix Operations) :** Le calcul des tenseurs pour le LLM est intensif en calcul flottant (FLOPS). Sans GPU dédié (CUDA), cette charge repose entièrement sur le CPU, expliquant la latence de 15-20 secondes.

4.3.3 Gestion des Threads (Software vs Hardware)

Pour garantir la réactivité de l'interface graphique (GUI), nous avons implémenté un modèle *Multi-threading* au niveau applicatif :

— **Main Thread :** Gère exclusivement la boucle d'événements de l'interface (affichage, clics), consommant très peu de cycles CPU.

— **Worker Thread :** Exécute le pipeline lourd (I/O disque → OCR → LLM).

Cette séparation permet d'exploiter l'architecture multi-cœurs du processeur : pendant que le *Worker Thread* sature un ou plusieurs cœurs pour le calcul, le *Main Thread* reste fluide sur un autre cœur logique.

4.4 Analyse des erreurs et limitations

4.4.1 Erreurs d'OCR corrigées par le LLM

Le LLM corrige efficacement les erreurs courantes :

Erreur OCR	**Correction**	**Détecté**
dipl6mé	diplômé	✓
lnformation	information	✓
H000	8000	✓
IEFT	LEFT	✓ (contexte)

TABLE 4.2 – Exemples de corrections d'OCR

4.4.2 Limitations identifiées

1. **Tableaux complexes** : L'OCR perd l'alignement colonne, et le LLM ne peut pas le reconstruire parfaitement sans indices visuels.
2. **Texte manuscrit** : Tesseract ne reconnaît pas bien l'handwriting ; le LLM ne peut compenser que si la qualité OCR reste acceptable.
3. **Documents multilingues non-déclarés** : Le pipeline utilise fra+eng. Les autres langues sont dégradées.
4. **Latence** : L'appel LLM introduit une latence de 3-15s par document, incompatible avec un traitement ultra-temps-réel.

Chapitre 5

Interface Graphique Utilisateur

5.1 Interface graphique

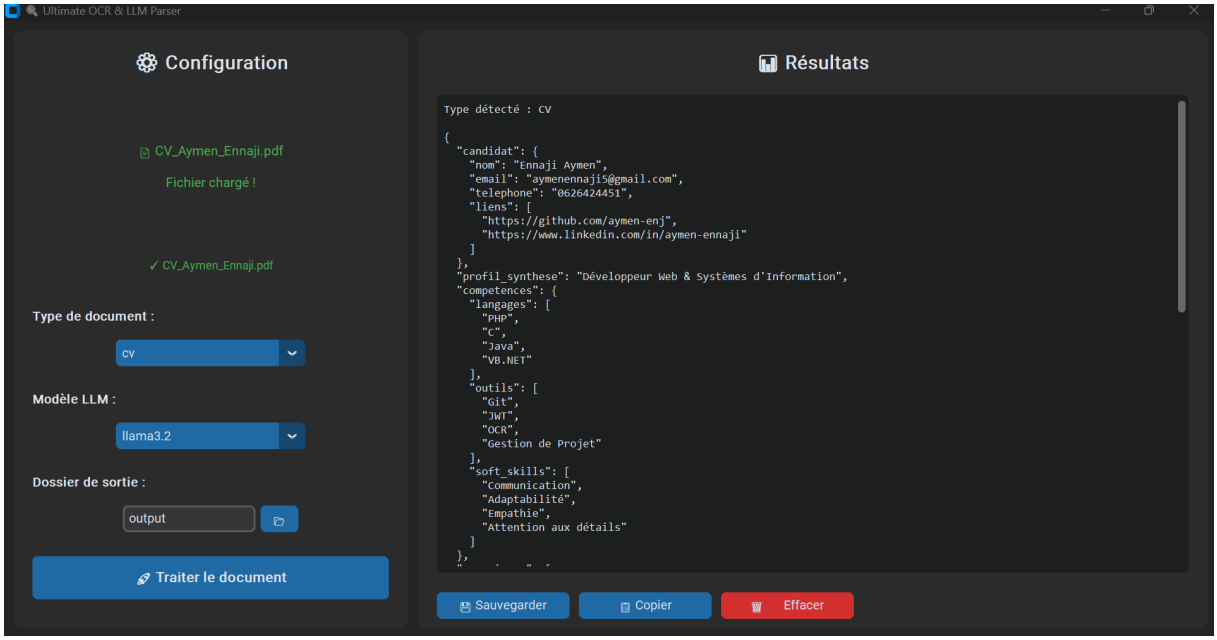

FIGURE 5.1 – Interface d'utilisateur

5.2 Motivation et objectifs

Bien que l'interface en ligne de commande (CLI) soit fonctionnelle et efficace pour les utilisateurs techniques, elle présente plusieurs limitations pour un usage général :

- **Courbe d'apprentissage** : Nécessite la connaissance de la syntaxe et des arguments de commande.
- **Productivité** : Saisie manuelle des chemins de fichiers et options à chaque exécution.
- **Accessibilité** : Non adaptée aux utilisateurs non techniques ou occasionnels.
- **Feedback visuel limité** : Progression et résultats affichés dans un terminal brut.

Pour répondre à ces limitations, une interface graphique moderne a été développée avec les objectifs suivants :

1. Simplifier l'utilisation via glisser-déposer et menus intuitifs.

2. Fournir un feedback visuel en temps réel de la progression.
3. Permettre la visualisation et l'export faciles des résultats JSON.
4. Maintenir toutes les fonctionnalités de la version CLI.
5. Offrir une expérience utilisateur moderne et professionnelle.

5.3 Architecture de l'interface graphique

5.3.1 Choix technologiques

L'interface graphique utilise **CustomTkinter**, un framework Python moderne basé sur Tkinter :

Bibliothèque	**Fonction**	**Avantages**
CustomTkinter	Framework GUI principal	Design moderne, thèmes, widgets personnalisés
TkinterDnD2	Support drag & drop	Glisser-déposer natif de fichiers
threading	Traitement asynchrone	Interface réactive pendant l'OCR

TABLE 5.1 – Technologies de l'interface graphique

Avantages de CustomTkinter :
— Design moderne avec thème sombre/clair.
— Compatible multiplateforme (Windows, macOS, Linux).
— Widgets natifs améliorés (boutons, zones de texte, barres de progression).
— Aucune dépendance lourde (pas de Qt, Electron, etc.).

5.3.2 Architecture de classe

L'application GUI est implémentée dans la classe `OCRApp` qui hérite de `ctk.CTk` et `TkinterDnD.DnDWrapper` :

```python
class OCRApp(ctk.CTk, TkinterDnD.DnDWrapper):

    def __init__(self):
        super().__init__()

        # Configuration de la fenêtre principale
        self.title("🔍 Ultimate OCR & LLM Parser")
        self.geometry("1200x800")
        self.minsize(900, 600)

        # Variables
        self.selected_file = None
        self.processing = False
        self.result_data = None

        # Configuration Drag & Drop
        self.TkdndVersion = TkinterDnD._require(self)

        # Création de l'interface
        self._create_widgets()
```

FIGURE 5.2 – Architecture de la classe OCRApp

Principes de conception :

1. **Séparation des préoccupations** : Logique métier (extraction OCR) séparée de la GUI.
2. **Réutilisation du code** : Import et utilisation des classes existantes (`SmartExtractor`, `LLMOrchestrator`, etc.).
3. **Threading** : Traitement OCR/LLM dans un thread séparé pour éviter le gel de l'interface.
4. **Communication thread-safe** : Utilisation de `self.after()` pour mise à jour UI depuis les threads.

5.4 Fonctionnalités de l'interface

5.4.1 Panneau de configuration

Le panneau gauche regroupe tous les contrôles de configuration :

Zone de drop : Zone cliquable avec support drag & drop pour sélectionner les fichiers PDF, PNG, JPG, JPEG.

Type de document : Menu déroulant avec options : `auto` (défaut), `cv`, `facture`, `formulaire`.

Modèle LLM : Sélection du modèle Ollama : `llama3.2` (défaut), `mistral`, `llama2`, `codellama`.

Dossier de sortie : Champ modifiable avec bouton de sélection de dossier.

Bouton de traitement : Lance l'extraction (désactivé tant qu'aucun fichier n'est sélectionné).

Barre de progression : Animation en temps réel pendant le traitement.

Status : Messages d'état détaillant la progression (extraction, détection, analyse, enrichissement).

5.4.2 Panneau de résultats

Le panneau droit affiche les résultats et offre des actions :

Zone de texte JSON : Affichage formaté du JSON résultant avec indentation et syntaxe lisible. Police monospace (Consolas) pour une meilleure lisibilité.

Bouton Sauvegarder : Export du JSON vers un fichier personnalisé via dialogue de sauvegarde.

Bouton Copier : Copie le JSON dans le presse-papier pour utilisation externe.

Bouton Effacer : Réinitialise l'interface pour traiter un nouveau document.

5.4.3 Workflow utilisateur

Le flux d'utilisation typique se déroule en 5 étapes :

1. **Sélection** : L'utilisateur glisse-dépose un fichier ou clique pour parcourir.
2. **Configuration** : (Optionnel) Ajustement du type et du modèle si nécessaire.
3. **Traitement** : Clic sur le bouton "Traiter le document". L'interface affiche la progression via :
   — Messages de status actualisés ("Extraction du texte...", "Détection du type...", etc.)
   — Barre de progression animée
4. **Visualisation** : Le JSON structuré s'affiche automatiquement dans le panneau droit.
5. **Export** : L'utilisateur peut sauvegarder ou copier le résultat.

5.5 Implémentation technique

5.5.1 Traitement asynchrone

Pour éviter le gel de l'interface pendant le traitement (qui peut durer 10-30 secondes), le code utilise le module `threading` :

```python
def _process_document(self):
        """Traiter le document dans un thread séparé"""
        if self.processing:
            return

        if not self.selected_file:
            messagebox.showwarning("Attention", "Veuillez sélectionner un fichier !")
            return

        self.processing = True
        self.process_btn.configure(state="disabled")
        self.progress.set(0)
        self.status_label.configure(text="Traitement en cours...", text_color="#2196f3")

        # Reset text box for logs
        self.result_text.configure(state="normal")
        self.result_text.delete("1.0", "end")
        self.result_text.insert("1.0", "Les résultats apparaîtront ici...\n") # Remettre le placeholder pour que le check fonctionne
        self.result_text.configure(state="disabled")

        # Lancer dans un thread pour ne pas bloquer l'interface
        thread = threading.Thread(target=self._run_ocr)
        thread.daemon = True
        thread.start()

        # Reset progress
        self.progress.set(0)
```

FIGURE 5.3 – Traitement asynchrone (Threading)

5.5.2 Gestion du drag & drop

L'intégration de TkinterDnD2 permet un glisser-déposer natif :

```python
def _on_drop(self, event):
        """Gestion du drag & drop"""
        file_path = event.data
        # Nettoyer le chemin (enlever les accolades si présentes)
        if file_path.startswith('{') and file_path.endswith('}'):
            file_path = file_path[1:-1]
        self._load_file(file_path)
```

FIGURE 5.4 – Configuration Drag & Drop

5.6 Avantages de l'interface graphique

5.6.1 Comparaison CLI vs GUI

Aspect	**CLI**	**GUI**
Sélection fichier	Taper le chemin complet	Drag & drop / Parcourir
Configuration	Arguments en ligne de commande	Menus déroulants intuitifs
Progression	Messages texte terminal	Barre animée + status détaillé
Résultats	Fichier JSON externe	Affichage formaté + export
Courbe apprentissage	Moyenne (docs nécessaires)	Très facile (auto-découverte)
Expérience utilisateur	Basique mais efficace	Moderne et professionnelle
Public cible	Développeurs, intégration	Tous utilisateurs

TABLE 5.2 – Comparaison CLI vs GUI

5.6.2 Gain de productivité

L'interface graphique améliore significativement la productivité :
— **Temps de sélection** : 2-3 secondes (drag & drop) vs 10-15 secondes (taper chemin).
— **Erreurs de configuration** : Réduites grâce aux menus (pas de typo dans les arguments).
— **Feedback immédiat** : Visualisation directe du JSON sans ouvrir un fichier externe.
— **Itérations rapides** : Bouton "Effacer" pour traiter un nouveau document sans relancer l'application.

5.7 Documentation utilisateur

Trois documents accompagnent l'interface graphique :

GUI_GUIDE.md : Guide d'utilisation détaillé avec captures d'écran et explications des fonctionnalités.

QUICK_START.md : Démarrage rapide en 4 étapes pour les utilisateurs pressés.

README.md : Mis à jour avec section dédiée à l'interface graphique.

5.8 Tests et validation

L'interface graphique a été testée selon les critères suivants :

Test	Résultat	Observation
Lancement application	✓	Fenêtre s'ouvre en <2s
Drag & drop PDF	✓	Fichier chargé instantanément
Drag & drop Image	✓	Validation format correcte
Traitement CV	✓	JSON affiché après 8s
Traitement Facture	✓	JSON affiché après 12s
Traitement Formulaire	✓	JSON affiché après 10s
Gestion erreur (fichier manquant)	✓	Dialogue d'erreur informatif
Gestion erreur (Ollama non lancé)	✓	Message d'erreur clair
Copier résultat	✓	JSON copié dans presse-papier
Sauvegarder résultat	✓	Fichier JSON exporté
Réinitialisation	✓	Interface prête pour nouveau document

TABLE 5.3 – Tests de validation de l'interface graphique

5.9 Perspectives d'évolution

L'interface graphique actuelle est fonctionnelle et complète, mais plusieurs améliorations sont envisageables :

1. **Prévisualisation du document** : Afficher un aperçu du PDF/image avant traitement.
2. **Historique des traitements** : Liste des documents récemment traités avec accès rapide aux résultats.
3. **Traitement batch** : Sélection multiple de fichiers et traitement séquentiel automatique.
4. **Thème personnalisable** : Permettre à l'utilisateur de basculer entre thème sombre et clair.
5. **Paramètres avancés** : Options OCR (DPI, langues, PSM mode) accessibles via panneau avancé.
6. **Graphiques de confiance** : Visualisation des scores de certitude pour chaque champ extrait.
7. **Packaging exécutable** : Distribution sous forme .exe (Windows) ou .app (macOS) pour utilisateurs sans Python.

Chapitre 6

Améliorations Futures

6.1 Court terme (1-2 mois)

1. **Scoring de confiance** : Ajouter un score 0-1 pour chaque champ extrait, permettant au LLM d'évaluer sa certitude.
2. **Support multilingue** : Permettre à l'utilisateur de spécifier les langues attendues, adapter Tesseract et le prompt en conséquence.
3. **Validation de schéma** : Vérifier que le JSON retourné respecte strictement le schéma attendu ; ajouter des valeurs par défaut.
4. **API REST** : Fournir une interface HTTP simple (FastAPI) pour intégration facile en amont.

6.2 Moyen terme (2-6 mois)

1. **Prétraitement avancé** : Binarisation adaptative, suppression de bruit (débruitage par ondelettes).
2. **Traitement batch parallèle** : Traiter plusieurs pages simultanément.
3. **Fine-tuning de modèle** : Entraîner une version légère du LLM spécialisée pour l'extraction de documents.
4. **Interface utilisateur** : GUI simple (Tkinter ou web avec Streamlit) pour upload/visualisation interactive.

6.3 Long terme (6+ mois)

1. **Modèle de vision (LMM)** : Intégrer un Large Multimodal Model (ex : LLaVA) pour analyser l'image directement sans OCR intermédiaire.
2. **Post-édition assistée** : Interface permettant l'utilisateur de corriger manuellement et de réinjecter les corrections pour améliorer les prompts.
3. **Support de plus de formats** : Docx, Excel, images TIFF multi-page, code-barres, QR codes.
4. **Déploiement conteneurisé** : Docker + Kubernetes pour production cloud ou on-premise scalable.

Chapitre 7

Conclusion

7.1 Synthèse des réalisations

Le projet “Ultimate OCR & LLM Parser“ a démontré la viabilité d'une approche hybride combinant extraction OCR classique, prétraitement d'images, classification heuristique et post-traitement par modèle de langage de grande taille. Les points clés réalisés sont :

— ✓ Pipeline robuste et modulaire, gérant plusieurs formats d'entrée et types de documents.
— ✓ Extraction adaptative : Markdown quand disponible, OCR en fallback.
— ✓ Classification automatique du type de document sans apprentissage supervisé.
— ✓ Génération de JSON structuré, normalisé et prêt pour l'automatisation.
— ✓ Exécution locale : respect de la confidentialité des données.
— ✓ Résilience : gestion des erreurs sans interruption de la chaîne globale.

7.2 Contribution technique

Sur le plan technique, le projet contribue :

1. Une démonstration pratique de l'intégration efficace OCR + LLM local pour extraction de données structurées.
2. Une approche d'ingénierie de prompts adaptée aux tâches de structuration documentaire.
3. Une architecture modulaire de pipeline extensible et maintenable.
4. Une stratégie de correction des erreurs OCR par règles heuristiques + approches neuronales.

7.3 Limitations et recommandations

Limitations identifiées et recommandations associées :

Limitation	**Recommandation**
Tableaux complexes mal extraits	Utiliser OCR spécialisé (ex : Camelot) ou LMM
Latence LLM (3-15s/doc)	Utiliser modèle plus petit ou GPU local
Texte manuscrit non reconnu	Combiner Tesseract + reconnaissance handwriting (ex : Google OCR)
Support limité à 2 langues	Adapter langues OCR et prompts par document

TABLE 7.1 – Limitations et recommandations

7.4 Perspectives

Ce projet ouvre plusieurs directions de recherche et développement :
— **Extraction intelligente de données** : généralisation à d'autres domaines (recettes, articles scientifiques, contrats légaux).
— **Amélioration continue** : feedback loop utilisateur → amélioration des prompts.
— **Intégration d'IA multimodale** : exploitation conjointe du texte et de l'image pour extraction plus riche.
— **Déploiement à grande échelle** : architecture cloud/edge pour traitement de millions de documents.

7.5 Conclusion finale

Le projet démontre qu'une approche pragmatique combinant OCR proven et LLM modernes peut délivrer une valeur substantielle pour le traitement automatis de documents complexes. Bien qu'il existe des limitations (tableaux, manuscrit), l'approche fournit un bon équilibre entre qualité, latence et facilité d'implémentation. Avec les évolutions proposées (modèles plus légers, prétraitement avancé, interfaces utilisateur), cette solution peut devenir un outil production robuste pour l'extraction et l'automatisation documentaire à grande échelle.

Annexe A

Annexes

A.1 Installation et utilisation

Pour utiliser le pipeline :

```
# 1. Installer les dépendances
pip install -r requirements.txt

# 2. Lancer Ollama
ollama serve
# Dans un autre terminal:
ollama pull llama3.2

# 3. Traiter un document
python ocr_extractor.py input/mon_cv.pdf
python ocr_extractor.py input/facture.pdf --type facture --model
   llama3.2
```