Cahier des charges : Configuration RAG Preprocessing avec système Fallback

1. Introduction et contexte

Ce cahier des charges définit les spécifications techniques pour un système de configuration flexible et robuste destiné au prétraitement de documents dans le cadre d'un pipeline RAG (Retrieval-Augmented Generation). L'objectif principal est de créer une architecture modulaire permettant de traiter tous types de fichiers avec des stratégies d'optimisation configurables et des mécanismes de fallback automatiques [1] [2] [3].

Le système doit supporter une variété de formats de documents tout en offrant la possibilité de choisir entre différents modes d'optimisation : vitesse maximale, optimisation mémoire, compromis équilibré, ou qualité maximale [4] [5] [6].

2. Objectifs du système

2.1 Objectifs fonctionnels

Le système doit permettre de :

- **Traiter automatiquement** tous les types de fichiers supportés avec le meilleur parser disponible
- **Gérer les échecs** avec des stratégies de fallback automatiques par type de fichier
- **Optimiser les performances** selon le mode choisi (speed, memory, compromise, quality, custom)
- **Configurer facilement** l'enchaînement des bibliothèques via un fichier YAML
- **Monitorer les performances** avec des métriques détaillées (temps, mémoire, taux de succès)

2.2 Objectifs techniques

- Architecture modulaire et extensible permettant l'ajout facile de nouvelles bibliothèques
- Gestion robuste des erreurs avec retry et fallback automatiques
- Optimisation mémoire pour le traitement de grands volumes de documents
- Support multilingue notamment pour le français et l'anglais
- Logging détaillé pour le debugging et l'analyse

3. Extensions de fichiers supportées

Le système doit prendre en charge les catégories de fichiers suivantes [7] [8] [9] :

3.1 Documents PDF

- **Extensions** : `.pdf`
- **Caractéristiques** : Support des PDFs natifs et scannés, extraction de tableaux, images et formules
- **Complexité** : Élevée (nécessite OCR pour les scans)

3.2 Documents Microsoft Office

- **Extensions** : `.docx`, `.doc`, `.pptx`, `.ppt`, `.xlsx`, `.xls`
- **Caractéristiques** : Préservation de la structure, extraction de métadonnées
- **Complexité** : Moyenne

3.3 Documents Google/LibreOffice

- **Extensions** : `.odt`, `.odp`, `.ods`
- **Caractéristiques** : Formats open source, structure XML
- **Complexité** : Moyenne

3.4 Images

- **Extensions** : `.png`, `.jpg`, `.jpeg`, `.tiff`, `.bmp`, `.heic`
- **Caractéristiques** : Nécessite OCR systématique, preprocessing possible
- **Complexité** : Élevée

3.5 Contenu Web

- **Extensions** : `.html`, `.htm`
- **Caractéristiques** : Extraction texte avec préservation structure sémantique
- **Complexité** : Faible à moyenne

3.6 Markdown

- **Extensions** : `.md`
- **Caractéristiques** : Format texte structuré, parsing rapide
- **Complexité** : Faible

4. Modes d'optimisation globaux

Le système propose cinq modes d'optimisation configurables [10] [11] [12] :

4.1 Mode Speed (Vitesse)

Priorité : Temps de traitement minimal

Caractéristiques :

- Parsers légers et rapides (PyPDF, pdfplumber, PyMuPDF)
- OCR désactivé par défaut
- Chunking fixe simple (500-1000 caractères)
- Post-processing minimal
- Batch processing parallèle agressif

Cas d'usage : Indexation rapide de grands volumes, prototypage, analyse préliminaire

Performances attendues :

- PDFs : 30-50 fichiers/seconde
- Office : 50-100 fichiers/seconde

4.2 Mode Memory (Mémoire)

Priorité : Consommation mémoire minimale

Caractéristiques :

- Traitement par chunks avec streaming [13] [14] [15]
- Utilisation de générateurs Python au lieu de listes
- Memory-mapped files pour fichiers > 10MB
- Garbage collection forcé après chaque document
- Batch size réduit (3-5 documents simultanés)

Cas d'usage : Serveurs à ressources limitées, traitement de très gros fichiers, environnements contraints

Performances attendues :

- Empreinte mémoire : < 2GB même pour gros volumes
- Vitesse : -30% par rapport au mode speed

4.3 Mode Compromise (Équilibre)

Priorité : Équilibre qualité/performance

Caractéristiques :

- Parsers polyvalents (Marker, Unstructured) [2] [7] [16]
- OCR conditionnel (détection automatique du besoin)
- Chunking récursif ou sémantique léger

- Post-processing sélectif
- Batch size modéré (10 documents)

Cas d'usage : Usage général en production, pipeline RAG standard

Performances attendues :

- PDFs : 10-20 fichiers/seconde
- Qualité d'extraction : 85-90%

4.4 Mode Quality (Qualité)

Priorité : Extraction de qualité maximale

Caractéristiques :

- Parsers ML avancés (Docling, Surya, Marker)[2] [17] [18]
- OCR systématique multi-moteurs avec vote majoritaire
- Chunking sémantique avancé[3] [19] [20]
- Post-processing complet (correction erreurs, normalisation)
- Extraction tables, formules, images avec compréhension

Cas d'usage : Documents techniques complexes, extraction données critiques, corpus académique

Performances attendues :

- PDFs : 0.5-2 fichiers/seconde (selon complexité)
- Qualité d'extraction : 95-98%

4.5 Mode Custom (Personnalisé)

Priorité : Configuration manuelle fine

Caractéristiques :

- Tous les paramètres configurables individuellement
- Mix possible de stratégies par type de fichier
- Optimisations ciblées selon le cas d'usage

Cas d'usage : Besoins spécifiques, cas d'usage hybrides

5. Bibliothèques de parsing disponibles

5.1 Parsers PDF

Le tableau suivant compare les principales bibliothèques pour le parsing de PDF [2] [21] [22] [23] :

Bibliothèque	Vitesse	Qualité	Mémoire	Licence	Cas d'usage optimal
PyPDF/PyPDF2	⋆⋆⋆⋆⋆	⋆⋆	⋆⋆⋆⋆⋆	MIT	PDFs simples, texte basique
pdfplumber	⋆⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	MIT	Extraction tableaux
PyMuPDF	⋆⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆⋆⋆	AGPL	Usage général, rapide
Marker	⋆⋆⋆	⋆⋆⋆⋆⋆	⋆⋆⋆	GPL	Documents complexes
Docling	⋆⋆	⋆⋆⋆⋆⋆	⋆⋆	MIT	Compréhension avancée
Unstructured	⋆⋆⋆	⋆⋆⋆⋆	⋆⋆⋆	Apache 2.0	Pipeline complet

Recommandations par mode :

- **Speed** : PyMuPDF (vitesse exceptionnelle, bonne qualité)
- **Memory** : PyPDF (footprint minimal)
- **Compromise** : Marker ou Unstructured
- **Quality** : Docling avec Marker en fallback

5.2 Moteurs OCR

Comparaison des moteurs OCR pour l'extraction de texte depuis images et PDFs scannés [24] [25] [26] [27] :

Moteur	Vitesse	Qualité	Mémoire	Langues	GPU	Licence
Tesseract	⋆⋆⋆	⋆⋆⋆	⋆⋆⋆	100+	Non	Apache 2.0
EasyOCR	⋆⋆⋆⋆	⋆⋆⋆⋆	⋆⋆	80+	Oui	Apache 2.0
PaddleOCR	⋆⋆⋆⋆	⋆⋆⋆⋆⋆	⋆⋆⋆	80+	Oui	Apache 2.0
RapidOCR	⋆⋆⋆⋆⋆	⋆⋆⋆	⋆⋆⋆⋆	Limité	Non	MIT
Surya	⋆⋆⋆	⋆⋆⋆⋆⋆	⋆⋆	90+	Oui	Apache 2.0

Points clés OCR :

- **Tesseract** : Standard de l'industrie, excellent support français, nécessite preprocessing pour qualité optimale
- **EasyOCR** : Très bon sur textes manuscrits et images bruitées
- **PaddleOCR** : Excellent pour documents structurés et textes asiatiques
- **RapidOCR** : Le plus rapide, bon pour traitement de masse
- **Surya** : Meilleure compréhension de layout, intégré dans Marker

5.3 Parsers Office

Bibliothèque	Formats	Vitesse	Installation
python-docx	.docx	⋆⋆⋆⋆	71MB
python-pptx	.pptx	⋆⋆⋆⋆	71MB
openpyxl	.xlsx	⋆⋆⋆	71MB
Unstructured	Tous	⋆⋆⋆	146MB

5.4 Parsers Images

Preprocessing :

- PIL/Pillow : Standard, rapide, léger
- OpenCV : Preprocessing avancé (débruitage, correction perspective)

Extraction texte : Via moteurs OCR (voir section 5.2)

5.5 Parsers HTML

Bibliothèque	Vitesse	Qualité	Usage
BeautifulSoup4	⋆⋆⋆⋆	⋆⋆⋆	Parsing général
html2text	⋆⋆⋆⋆⋆	⋆⋆	Conversion MD rapide
Unstructured	⋆⋆⋆	⋆⋆⋆⋆	Préservation structure

5.6 Parsers Markdown

Bibliothèque	Vitesse	Features
markdown	⋆⋆⋆⋆⋆	Extensions standard
mistune	⋆⋆⋆⋆⋆	Très rapide, conforme CommonMark

6. Architecture du fichier de configuration YAML

6.1 Structure générale

Le fichier de configuration YAML doit suivre la structure hiérarchique suivante [28] [29] [30] [31] :

```
# Mode d'optimisation global
global_optimization_mode: compromise

# Processeurs par type de fichier
file_processors:
  pdf: { ... }
  office: { ... }
  images: { ... }
```

```
  html: { ... }
  markdown: { ... }

# Stratégie de chunking
chunking: { ... }

# Optimisations mémoire
memory_optimization: { ... }

# Surcharges par mode
mode_configurations: { ... }

# Gestion erreurs et logging
error_handling: { ... }
logging: { ... }

# Configuration output
output: { ... }
```

6.2 Configuration des processeurs

Pour chaque type de fichier, la configuration suit le pattern :

```
type_fichier:
  extensions: [".ext1", ".ext2"]

  # Pipeline primaire
  primary:
    library: nom_bibliothèque
    config:
      param1: valeur1
      param2: valeur2

  # Pipelines fallback (ordre d'essai)
  fallback:
    - library: nom_fallback_1
      config: { ... }
    - library: nom_fallback_2
      config: { ... }

  # OCR si applicable
  ocr:
    enabled: true/false/auto
    engines:
      primary: moteur_principal
      fallback: [moteur2, moteur3]
    config_moteur1: { ... }
```

6.3 Configuration du chunking

Quatre stratégies de chunking sont supportées [3] [19] [20] [32] :

Fixed : Chunks de taille fixe

- Paramètres : `chunk_size`, `chunk_overlap`, `unit` (characters/tokens)
- Usage : Mode speed, documents homogènes

Recursive : Découpage hiérarchique sur séparateurs

- Paramètres : `chunk_size`, `chunk_overlap`, `separators`
- Usage : Mode compromise, préservation structure

Semantic : Chunking basé sur la similarité sémantique

- Paramètres : `min_chunk_size`, `max_chunk_size`, `similarity_threshold`
- Usage : Mode quality, compréhension contextuelle

Adaptive : Configuration par type de document

- Paramètres : règles spécifiques par catégorie
- Usage : Mode custom, cas d'usage mixtes

6.4 Optimisations mémoire

Stratégies configurables [13] [14] [33] [15] :

- **use_generators** : Générateurs vs listes pour itération
- **stream_processing** : Traitement par flux pour gros fichiers
- **force_gc** : Garbage collection forcé après chaque document
- **mmap_large_files** : Memory-mapped files pour fichiers > seuil
- **batch_size** : Nombre de documents en mémoire simultanément

Seuils :

- **max_file_size_mb** : Taille au-delà de laquelle streaming obligatoire
- **max_memory_usage_mb** : Limite mémoire globale du processus

6.5 Surcharges par mode

Chaque mode peut surcharger la configuration par défaut :

```
mode_configurations:
  speed:
    pdf:
      primary: pymupdf
      ocr: false
    chunking:
      strategy: fixed
```

```
      chunk_size: 500

  quality:
    pdf:
      primary: docling
      ocr: always
    chunking:
      strategy: semantic
```

6.6 Gestion des erreurs

Configuration de la résilience du système :

- **max_retries** : Nombre de tentatives avant fallback
- **retry_delay_seconds** : Délai entre tentatives
- **fallback_on_error** : Activer mécanisme fallback
- **continue_on_error** : Continuer traitement batch si erreur

6.7 Logging et métriques

Système de logging configurable :

- **level** : DEBUG | INFO | WARNING | ERROR
- **log_file** : Chemin fichier log
- **log_to_console** : Affichage console
- **track_metrics** : Liste des métriques à tracer
  - processing_time
  - memory_usage
  - success_rate
  - library_used

7. Implémentation Python

7.1 Architecture logicielle

L'implémentation Python doit suivre une architecture orientée objet avec les composants suivants :

Classes principales :

- `RAGPreprocessingConfig` : Gestion de la configuration YAML
- `DocumentProcessor` : Orchestration du traitement
- `LibraryAdapter` : Interface abstraite pour chaque bibliothèque
- `FallbackManager` : Gestion de la stratégie de fallback
- `MetricsCollector` : Collection et agrégation des métriques

Modules secondaires :

- `parsers/` : Adaptateurs pour chaque bibliothèque
- `chunking/` : Implémentations des stratégies de chunking
- `ocr/` : Wrappers pour moteurs OCR
- `memory/` : Utilitaires d'optimisation mémoire

7.2 Pattern de fallback

Le mécanisme de fallback doit suivre le pattern Chain of Responsibility :

1. Tentative avec le processeur primaire
2. Si échec ou exception : itération sur liste fallback
3. Pour chaque fallback : tentative avec retry configurable
4. Si tous échouent : logging erreur et retour None
5. Collection des métriques à chaque étape

7.3 Gestion de la mémoire

Techniques d'optimisation à implémenter :

Streaming : Traitement par chunks pour fichiers > seuil

```
def process_large_file(file_path, chunk_size=1024*1024):
    with open(file_path, 'rb') as f:
        while chunk := f.read(chunk_size):
            yield process_chunk(chunk)
```

Générateurs : Éviter chargement complet en mémoire

```
def process_batch(files):
    for file in files:
        result = process_file(file)
        yield result
        gc.collect()  # Force cleanup
```

Memory-mapped files : Pour accès rapide sans chargement complet

```
import mmap
with open(file_path, 'r') as f:
    with mmap.mmap(f.fileno(), 0, access=mmap.ACCESS_READ) as m:
        return process_mmap(m)
```

7.4 Interface des adaptateurs

Chaque bibliothèque doit implémenter l'interface :

```
class LibraryAdapter(ABC):
    @abstractmethod
    def parse(self, file_path: Path, config: Dict) -&gt; Dict[str, Any]:
        pass

    @abstractmethod
    def supports(self, file_extension: str) -&gt; bool:
        pass

    @abstractmethod
    def get_metrics(self) -&gt; ProcessingMetrics:
        pass
```

8. Stratégies de fallback par type de fichier

8.1 PDF

Pipeline recommandé :

1. **Primary** : Marker (qualité excellente, gère complexité)
2. **Fallback 1** : PyMuPDF (rapide, fiable, bonne qualité)
3. **Fallback 2** : Unstructured hi_res (versatile)
4. **Fallback 3** : pdfplumber (spécialiste tableaux)
5. **Last resort** : PyPDF (extraction texte basique)

OCR si nécessaire :

- Détection automatique (texte natif vs scanné)
- Pipeline : Tesseract → PaddleOCR → EasyOCR

8.2 Office

Pipeline recommandé :

1. **Primary** : Unstructured (support universel)
2. **Fallback par format** :
    - .docx → python-docx
    - .pptx → python-pptx
    - .xlsx → openpyxl

8.3 Images

Pipeline recommandé :

1. **Preprocessing** : Pillow (redimensionnement, normalisation)
2. **OCR primary** : PaddleOCR (qualité/vitesse)
3. **OCR fallback 1** : Tesseract (robustesse)
4. **OCR fallback 2** : EasyOCR (manuscrit)

8.4 HTML

Pipeline recommandé :

1. **Primary** : BeautifulSoup4 avec lxml parser
2. **Fallback** : html2text (conversion MD)

8.5 Markdown

Pipeline recommandé :

1. **Primary** : mistune (rapide, conforme)
2. **Fallback** : markdown standard library

9. Métriques et monitoring

9.1 Métriques par document

Pour chaque document traité, collecter :

- **processing_time** : Temps total de traitement (ms)
- **memory_peak** : Pic de consommation mémoire (MB)
- **library_used** : Bibliothèque ayant réussi
- **fallback_count** : Nombre de fallbacks nécessaires
- **chunk_count** : Nombre de chunks générés
- **file_size** : Taille du fichier source (bytes)
- **status** : SUCCESS | FALLBACK_USED | FAILED

9.2 Métriques agrégées

Au niveau du batch :

- **throughput** : Documents/seconde
- **success_rate** : Taux de réussite (%)
- **avg_processing_time** : Temps moyen par document
- **avg_memory_usage** : Consommation mémoire moyenne

- **library_distribution** : Répartition des bibliothèques utilisées

9.3 Alertes

Définir des seuils pour alertes :

- Taux d'échec > 5%
- Temps de traitement > 10x la moyenne
- Consommation mémoire > seuil configuré
- Taux de fallback > 30%

10. Tests et validation

10.1 Dataset de test

Constituer un dataset de test représentatif :

- **PDFs natifs** : 50 documents variés
- **PDFs scannés** : 30 documents (qualité variable)
- **Office** : 40 documents (Word, Excel, PowerPoint)
- **Images** : 30 images avec texte
- **HTML** : 20 pages web
- **Markdown** : 20 fichiers

Total : 190 documents test

10.2 Tests unitaires

Couvrir :

- Chargement configuration YAML
- Application des modes d'optimisation
- Mécanisme de fallback
- Adaptateurs de bibliothèques
- Stratégies de chunking

10.3 Tests d'intégration

Valider :

- Traitement end-to-end par type de fichier
- Gestion des erreurs et fallbacks
- Collection des métriques
- Performance selon le mode

10.4 Benchmarks de performance

Mesurer pour chaque mode :

Mode	Throughput (docs/s)	Mémoire (MB)	Qualité (%)
Speed	Cible : 30-50	< 500	75-80
Memory	Cible : 10-20	< 200	80-85
Compromise	Cible : 10-20	< 1000	85-90
Quality	Cible : 1-5	< 2000	95-98

11. Déploiement et maintenance

11.1 Prérequis système

Python : >= 3.9

Dépendances principales :

- PyYAML pour configuration
- Bibliothèques de parsing selon configuration
- memory_profiler pour monitoring

Ressources recommandées :

- **Mode Speed** : 2 CPU, 2GB RAM
- **Mode Memory** : 1 CPU, 512MB RAM
- **Mode Compromise** : 2 CPU, 4GB RAM
- **Mode Quality** : 4 CPU, 8GB RAM, GPU optionnel

11.2 Installation

```
pip install rag-preprocessing
# ou
pip install -r requirements.txt
```

Installation optionnelle des dépendances lourdes :

```
# OCR engines
pip install tesseract easyocr paddleocr

# Advanced parsers
pip install marker-pdf docling unstructured
```

11.3 Configuration initiale

1. Copier le template YAML
2. Ajuster selon le cas d'usage
3. Tester avec échantillon
4. Itérer sur la configuration

11.4 Monitoring en production

- Logs rotatifs avec retention 30 jours
- Métriques exportées vers Prometheus/Grafana
- Alertes sur anomalies
- Dashboard de suivi des performances

11.5 Maintenance

Mises à jour bibliothèques : Mensuel

Revue configuration : Trimestriel

Optimisation : En continu selon métriques

Documentation : Maintenir à jour

12. Évolutions futures

12.1 Court terme (3-6 mois)

- Support formats additionnels (RTF, ePub)
- Amélioration détection qualité OCR
- Chunking avec overlap dynamique
- Parallélisation GPU pour OCR

12.2 Moyen terme (6-12 mois)

- Chunking agentic avec LLM
- Auto-tuning des paramètres par ML
- Support multi-GPU
- API REST pour service distant

12.3 Long terme (12+ mois)

- Support vidéos et audio (transcription)
- Détection automatique du meilleur mode
- Federated learning pour amélioration continue
- Intégration VLM pour compréhension visuelle

13. Conclusion

Ce cahier des charges définit un système complet et flexible pour le prétraitement de documents dans un pipeline RAG. L'architecture proposée permet :

- **Flexibilité** : Configuration YAML complète sans modification code
- **Robustesse** : Mécanismes de fallback automatiques
- **Performance** : Modes d'optimisation adaptés aux contraintes
- **Extensibilité** : Architecture modulaire pour ajout facile de nouvelles bibliothèques
- **Monitoring** : Métriques détaillées pour optimisation continue

Le système peut être adapté à différents cas d'usage en ajustant simplement la configuration, rendant son déploiement rapide et sa maintenance aisée.

Les performances attendues varient significativement selon le mode choisi, permettant de prioriser soit la vitesse (30-50 docs/s), soit la qualité (95-98% précision), soit un compromis équilibré.

La stratégie de fallback multiniveau garantit un taux de réussite élevé même sur des documents complexes ou corrompus, avec une dégradation gracieuse de la qualité plutôt qu'un échec complet.

[34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [69] [70] [71] [72] [73] [74] [75] [76] [77]

⁂

1. https://www.pingcap.com/article/how-to-optimize-rag-pipelines-for-maximum-efficiency/
2. https://www.xzjhi.com/en/ai-develope-tools-series-2-open-source-doucment-parsing.html
3. https://agenta.ai/blog/the-ultimate-guide-for-chunking-strategies
4. https://www.dhiwise.com/post/build-rag-pipeline-guide
5. https://encord.com/blog/rag-pipelines/
6. https://www.deepset.ai/blog/preprocessing-rag
7. https://nanonets.com/blog/document-parsing/
8. https://docs.databricks.com/aws/en/generative-ai/tutorials/ai-cookbook/quality-data-pipeline-rag
9. https://unstructured.io/blog/level-up-your-genai-apps-essential-data-preprocessing-for-any-rag-system
10. https://arxiv.org/html/2412.10543v1
11. https://arxiv.org/html/2505.08445v1
12. https://www.linkedin.com/posts/mandarkarhade_rag-hyper-parameter-optimization-activity-7328477458985340928-PitM

13. https://www.reddit.com/r/pythontips/comments/149qlts/some_quick_and_useful_python_memory_optimization/
14. https://martinheinz.dev/blog/68
15. https://www.honeybadger.io/blog/reducing-your-python-apps-memory-footprint/
16. https://unstructured.io/blog/chunking-for-rag-best-practices
17. https://kevinhu.io/notes/how-marker-works/
18. https://arxiv.org/html/2408.09869v5
19. https://gpt-trainer.com/blog/rag+chunking+strategy
20. https://weaviate.io/blog/chunking-strategies-for-rag
21. https://arxiv.org/html/2410.09871v1
22. https://www.f22labs.com/blogs/5-best-document-parsers-in-2025-tested/
23. https://pypi.org/project/marker-pdf/0.3.2/
24. https://arivelm.com/tesseract-vs-easyocr/
25. https://ironsoftware.com/csharp/ocr/blog/ocr-tools/easyocr-vs-tesseract/
26. https://www.koncile.ai/en/ressources/10-open-source-ocr-tools-you-should-know-about
27. https://modal.com/blog/8-top-open-source-ocr-models-compared
28. https://pathway.com/developers/templates/configure-yaml/
29. https://www.reddit.com/r/LangChain/comments/1glq66t/customizing_llm_templates_with_yaml_configuration/
30. https://github.com/RUC-NLPIR/FlashRAG
31. https://pathway.com/developers/templates/yaml-snippets/rag-configuration-examples/
32. https://community.databricks.com/t5/technical-blog/the-ultimate-guide-to-chunking-strategies-for-rag-applications/ba-p/113089
33. https://dev.to/pragativerma18/understanding-pythons-garbage-collection-and-memory-optimization-4mi2
34. https://intuitionlabs.ai/articles/ai-ocr-models-pdf-structured-text-comparison
35. https://chamomile.ai/reliable-rag-with-data-preprocessing/
36. https://stackoverflow.blog/2024/12/27/breaking-up-is-hard-to-do-chunking-in-rag-applications/
37. https://www.reddit.com/r/LangChain/comments/1c1ksv6/best_libraryframework_for_parsing_pdf_documents/
38. https://github.com/docling-project/docling-parse
39. https://www.youtube.com/watch?v=IWhP5lvY_oc
40. https://www.mycelium-of-knowledge.org/unlocking-content-converting-pdfs-to-markdown-with-marker-pdf/
41. https://huggingface.co/learn/cookbook/en/rag_with_unstructured_data
42. https://dev.to/aairom/using-docling-parse-3pk5
43. https://unstructured.io/blog/understanding-what-matters-for-llm-ingestion-and-preprocessing
44. https://github.com/docling-project/docling
45. https://noblefilt.com/marker/
46. https://github.com/Unstructured-IO/unstructured
47. https://docling-project.github.io/docling/
48. https://www.youtube.com/watch?v=mdLBr9IMmgI
49. https://unstructured.readthedocs.io/en/main/best_practices/strategies.html
50. https://indaydream.com/docling-application-tutorial/

51. https://github.com/datalab-to/marker
52. https://github.com/Unstructured-IO/community
53. https://www.youtube.com/watch?v=5UZafoHDopI
54. https://www.youtube.com/watch?v=ROqpVzEzRIQ
55. https://github.com/App-vNext/Polly/blob/main/docs/strategies/fallback.md/
56. https://www.pollydocs.org/strategies/fallback.html
57. https://atalupadhyay.wordpress.com/2025/08/07/document-intelligence-guide-to-docling-for-ai-ready-data-processing/
58. https://arxiv.org/html/2407.04577v2
59. https://www.reddit.com/r/Rag/comments/1glq2fm/easily_customize_llm_pipelines_with_yaml/
60. https://rasa.com/docs/rasa/fallback-handoff/
61. https://www.digitalocean.com/community/tutorials/langchain-llm-fallback-gradient-ai
62. https://github.com/mftnakrsu/Comparison-of-OCR
63. https://microsoft.github.io/graphrag/config/yaml/
64. https://stackoverflow.com/questions/79465250/how-do-i-use-the-new-resilience-strategy-in-polly-core-to-a-fallback-strategy-wi
65. https://www.reddit.com/r/learnpython/comments/pserq9/tesseract_or_easyocr_which_one_is_better_and_why/
66. https://marker-inc-korea.github.io/AutoRAG/tutorial.html
67. https://docs.pega.com/bundle/customer-decision-hub/page/customer-decision-hub/cdh-portal/campaign-fallback-strategy.html
68. https://pugixml.org/benchmark.html
69. https://dev.to/nhirschfeld/i-benchmarked-4-python-text-extraction-libraries-2025-4e7j
70. https://peerdh.com/blogs/programming-insights/comparing-xml-parsing-libraries-for-performance-benchmarks
71. https://arxiv.org/html/2412.11854v1
72. https://colinchjava.github.io/2023-10-26/09-59-06-565538-comparison-of-java-dom-parser-with-other-xml-parsing-libraries/
73. https://dev.to/aaravjoshi/python-memory-optimization-9-practical-techniques-for-large-scale-applications-5fac
74. https://discuss.python.org/t/optimizing-memory-usage-for-large-csv-processing-in-python-3-12/98287
75. https://research.aimultiple.com/hybrid-rag/
76. https://www.reddit.com/r/haskell/comments/xc936e/performance_comparison_of_popular_parser_libraries/
77. https://stackoverflow.com/questions/3021264/python-tips-for-memory-optimization