Comment créer une architecture complète multiprojet (LLM + SD + Whisper + RAG)

> Created 7 janvier 2025
>
> Author : SA

Avec **ma configuration locale** :

- **Debian 12**
- **Ryzen 7**
- **RTX 3060 12 Go**
- **32 Go RAM**

Super — voilà **une architecture multiprojet complète** (prête à déployer localement sur Debian 12, RTX 3060 12 Go, Ryzen, 32 Go RAM) qui regroupe :

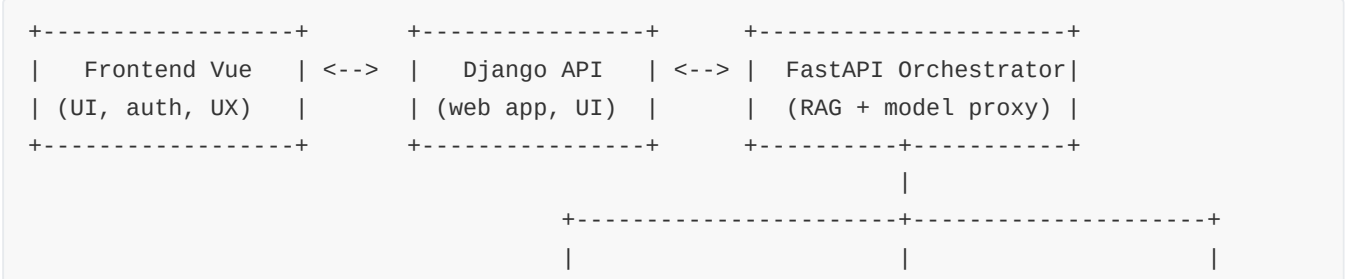
- **LLM local** (Qwen2.5-Coder / Mistral / Llama)
- **Stable Diffusion** (images)
- **Whisper / transcription audio**
- **RAG (PDF/documents → embeddings → vecteur DB → LLM)**
- **APIs / workers / frontend (Django + Vue)**
- **Monitoring, sécurité, backups**

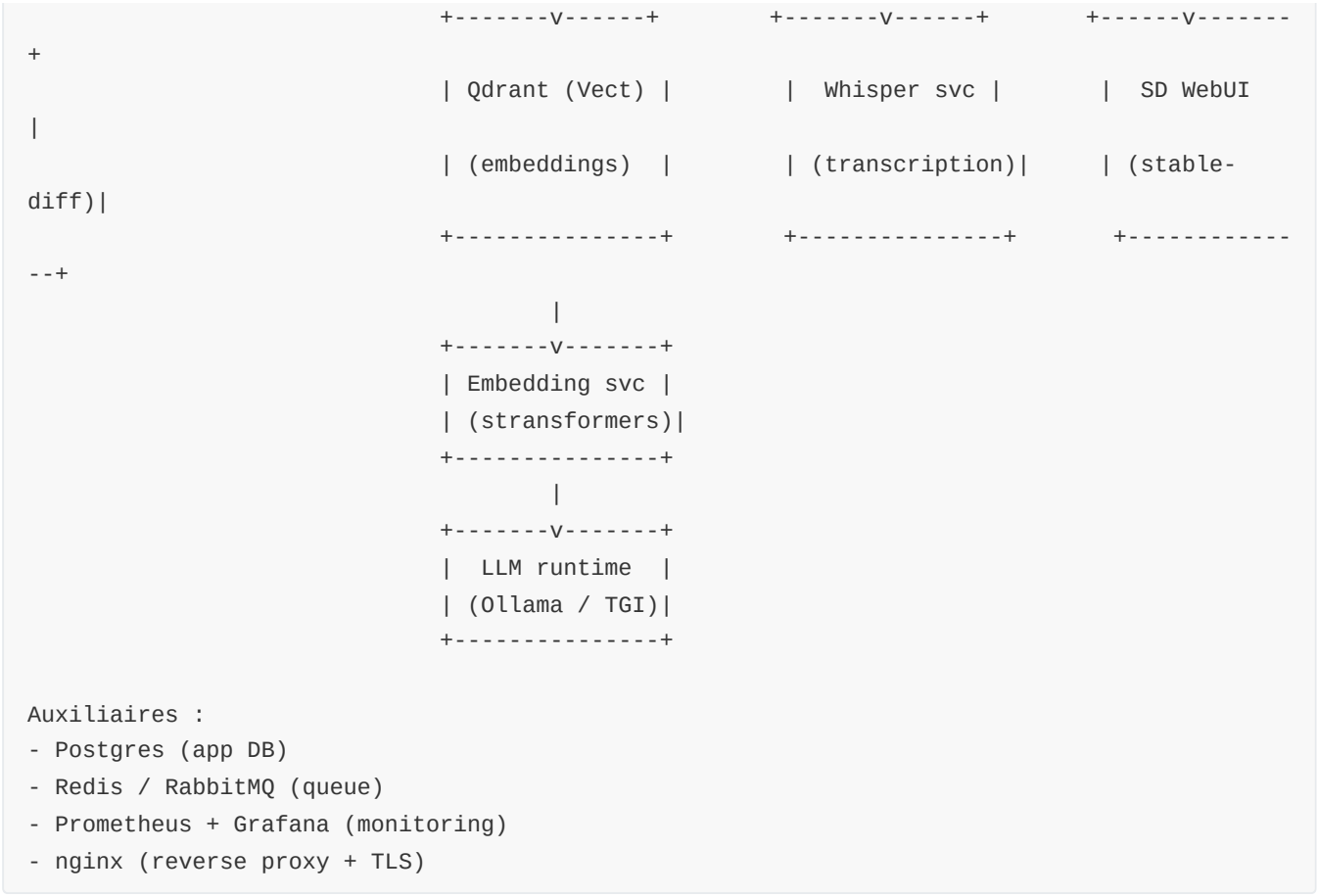
Je fournis :

1. Schéma d'architecture (conceptuel)
2. Description des composants et rôles
3. Docker Compose d'exemple (minimal & réaliste pour ta machine)
4. Considérations GPU / ressources (important pour RTX 3060 12 Go)
5. Exemples d'API orchestration (upload PDF → RAG → question → réponse)
6. Checklist déploiement & sécurité

---

1) Schéma d'architecture (vue d'ensemble)





2) Composants et responsabilités

- **Frontend Vue.js** : UI, upload PDF, chat, génération images, affichage résultats. Communique avec Django (auth, UI) et direct avec FastAPI pour IA.
- **Django** : gestion utilisateurs, permissions, interface admin, stockage métadonnées (Postgres). Expose endpoints frontend (auth, projets).
- **FastAPI Orchestrator** : service léger qui orchestre : extraction PDF → embeddings → indexation → query RAG → appel LLM (ou SD/Whisper). Peut être le point d'entrée pour appels IA.
- **Worker (Celery/RQ)** : tâches longues (indexation PDF, génération images, transcriptions). Utilise Redis/RabbitMQ.
- **Embeddings service** : microservice (Python) qui calcule embeddings via `sentence-transformers` (peut utiliser CPU/GPU selon modèle).
- **Vecteur DB (Qdrant)** : stocke vecteurs, supporte recherche ANN. Simple à containeriser.
- **LLM runtime** : choix entre
  - **Ollama installé sur host** (facile, CLI) — recommandé si tu veux gérer modèles localement ; **OU**
  - **Text-Generation-Inference (TGI)** docker (Hugging Face TGI) — serveur géré, nécessite CUDA docker runtime.
- **Stable Diffusion (SD WebUI)** : container (automatic1111) ou diffusers server. Grosse consommation VRAM pour big models.
- **Whisper service** : `faster-whisper` ou `whisperx` en microservice pour transcription audio.

- **Monitoring** : `prometheus`, `grafana`, `nvidia-smi exporter` pour garder un œil sur GPU/Mémoire/Temp.
- **Postgres** : metadata (users, docs, jobs).
- **nginx** : reverse proxy, TLS termination.

---

3) Docker Compose (exemple minimal — adapté à 1 machine)

> Remarque : ta RTX3060 est un **unique GPU 12 Go**. SD + LLM en même temps va souvent dépasser VRAM. Je fournis le compose mais tu devras **ne pas démarrer SD et LLM simultanément** ou configurer `CUDA_VISIBLE_DEVICES` pour dispatcher si tu as d'autres GPU.

```
# docker-compose.yml (version minimale, adaptée à ta machine)
version: "3.8"
services:

  postgres:
    image: postgres:15
    environment:
      POSTGRES_USER: aiuser
      POSTGRES_PASSWORD: secret
      POSTGRES_DB: airdb
    volumes:
      - pg_data:/var/lib/postgresql/data
    restart: unless-stopped

  redis:
    image: redis:7
    restart: unless-stopped

  qdrant:
    image: qdrant/qdrant:latest
    ports:
      - "6333:6333"
    volumes:
      - qdrant_storage:/qdrant/storage
    environment:
      - QDRANT__SERVICE__GRPC_PORT=6334
    restart: unless-stopped

  django:
    build: ./backend  # Dockerfile for Django
    command: gunicorn backend.wsgi:application --bind 0.0.0.0:8000 --workers 3
    volumes:
      - ./backend:/app
    env_file: .env
    ports:
      - "8000:8000"
    depends_on:
```

```yaml
    - postgres
    - redis
  restart: unless-stopped

fastapi:
  build: ./orchestrator
  volumes:
    - ./orchestrator:/app
  env_file: .env
  ports:
    - "8100:8100"
  depends_on:
    - qdrant
    - redis
  restart: unless-stopped

worker:
  build: ./worker
  command: celery -A tasks worker --loglevel=info
  env_file: .env
  depends_on:
    - redis
    - postgres
  restart: unless-stopped

embeddings:
  build: ./embeddings
  env_file: .env
  volumes:
    - ./models/embeddings:/models
  restart: unless-stopped

tgi:   # optionnel : text-generation-inference server (GPU)
  image: ghcr.io/huggingface/text-generation-inference:latest
  runtime: nvidia
  environment:
    - CUDA_VISIBLE_DEVICES=0
    - MODEL_ID=Qwen2.5-coder-7b   # local mount or HF repo
  volumes:
    - ./models/llm:/models/llm
  ports:
    - "8080:8080"
  deploy:
    resources:
      limits:
        devices:
          - driver: nvidia
            count: 1
            capabilities: [gpu]
  restart: unless-stopped

sd-webui:
  image: ghcr.io/automatic1111/stable-diffusion-webui:latest
```

```
    runtime: nvidia
    environment:
      - NVIDIA_VISIBLE_DEVICES=0
    volumes:
      - ./models/sd:/stable-diffusion/models
      - ./sd-config:/config
    ports:
      - "7860:7860"
    restart: unless-stopped

  whisper:
    build: ./whisper
    env_file: .env
    ports:
      - "8001:8001"
    restart: unless-stopped

  nginx:
    image: nginx:stable
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - ./nginx/conf.d:/etc/nginx/conf.d
      - ./certs:/etc/ssl/certs
    depends_on:
      - django
      - fastapi
    restart: unless-stopped

volumes:
  pg_data:
  qdrant_storage:
  models_llm:
  models_sd:
```

Notes :

- `runtime: nvidia` ou `deploy.resources` peut varier selon ta version Docker / Nvidia Container Toolkit.
- `tgi` et `sd-webui` utilisent tous les deux la même GPU — **ne les exécute pas ensemble** sur une 3060 sauf si tu gardes modèles très légers ou limites VRAM.
- Tu peux démarrer **TGI (LLM)** pour les sessions de code, puis arrêter TGI et démarrer SD lorsque tu veux générer images.

---

4) Considérations GPU / performance (critique pour RTX 3060 12 Go)

- **VRAM limitée (12 Go)** : un modèle Qwen2.5-Coder 7B quantifié en Q4 tient, SD XL ou SD 1.5 + large VAE peut nécessiter 6–9 Go. MAIS avoir les deux simultanés risque d'OOM.

- **Stratégies** :
  - **Exécuter les services gourmands séquentiellement** (start/stop via systemctl/docker-compose).
  - Utiliser **offload CPU/RAM** pour LLM (ex : accelerate + offload) — réduit VRAM mais impacte perf.
  - **Limiter batchsize / threads** des services.
  - Pour embeddings, utilise des modèles légers (all-MiniLM) sur CPU si besoin.
- **Swap & RAM** : configure un fichier swap (16–32 Go) pour éviter OOM critiques lors d'offload (faible perf mais évite crash).
- **Thermal / PSU** : surveille temp GPU; 3060 peut chauffer en longues sessions.

---

5) Exemple d'orchestration : upload PDF → RAG → question → réponse

Workflow résumé

1. Upload PDF → FastAPI reçoit → stocke fichier → push job indexation worker
2. Worker : extrait texte (PyMuPDF), chunking, calcule embeddings (sentence-transformers), stocke vecteurs dans Qdrant + metadata dans Postgres.
3. Frontend : user pose question → FastAPI récupère top-k chunks depuis Qdrant → construit `prompt = system + context + user question` → envoie au LLM server (TGI / Ollama) → retourne réponse.

Exemple code (FastAPI skeleton)

```
# orchestrator/app.py
from fastapi import FastAPI, UploadFile, File
from qdrant_client import QdrantClient
from sentence_transformers import SentenceTransformer
import requests, uuid, os

app = FastAPI()
qdrant = QdrantClient(url="http://qdrant:6333")
embed_model = SentenceTransformer("all-MiniLM-L6-v2")  # CPU ok

@app.post("/upload_pdf")
async def upload_pdf(file: UploadFile = File(...)):
    path = f"/data/{uuid.uuid4()}.pdf"
    with open(path, "wb") as f:
        f.write(await file.read())
    # push to worker (via redis/celery) - pseudo
    # celery_app.send_task("tasks.index_pdf", args=[path])
    return {"status": "queued"}

@app.post("/ask")
async def ask(question: str):
    # 1) query qdrant for context
    hits = qdrant.search(
        collection_name="pdf_docs",
```

```
        query_vector=embed_model.encode(question).tolist(),
        limit=4
    )
    context = "\n\n".join([h.payload["text"] for h in hits])
    prompt = f"Contexte:\n{context}\n\nQuestion: {question}\nRéponds de manière concise."
    # 2) call LLM (TGI HTTP)
    r = requests.post("http://tgi:8080/generate",
                      json={"inputs": prompt, "parameters":{"max_new_tokens":256}})
    return r.json()
```

---

6) Monitoring, logs & observabilité

- **Prometheus** : exporter node + cAdvisor + nvidia-dcgm exporter (GPU metrics).
- **Grafana** : dashboard GPU/CPU/mem, queue length, request latencies.
- **ELK / Loki** : logs centralisés si souhaité.
- **Healthchecks** : endpoints `/health` sur FastAPI / Django.

---

7) Sécurité & bonnes pratiques

- **Ne pas exposer les services modèles** (TGI / SD) directement sur Internet — mettre derrière Django / FastAPI + auth.
- **TLS** (nginx / certbot) pour tout trafic public.
- **Auth API keys** pour endpoints IA.
- **Limits** pour éviter abus (rate limit) — Nginx + Django Rest Framework throttle.
- **Backups** : dump Postgres (cron + rota), Qdrant snapshot planifié.
- **Sauvegarder modèles** (./models) hors container (volume) pour restauration.

---

8) Exemple de Dockerfile (service embeddings simple)

```
# embeddings/Dockerfile
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8200"]
```

`requirements.txt` minimal :

```
fastapi
uvicorn[standard]
sentence-transformers
torch  # si GPU, installer with CUDA build; sinon CPU
qdrant-client
```

---

9) Checklist déploiement (ordre conseillé)

1. Mettre à jour Debian + installer drivers NVIDIA + nvidia-docker.
2. Préparer dossiers `./models/llm`, `./models/sd`, `./models/embeddings`. Télécharger modèles offline.
3. Lancer infra basique : Postgres, Redis, Qdrant (docker-compose up -d).
4. Déployer Django (DB migrations) → runserver/gunicorn.
5. Déployer FastAPI + worker. Tester endpoints health.
6. Déployer LLM (ollama host OR tgi docker) ; tester inference.
7. Déployer embeddings service → run sample embedding → add into qdrant.
8. Déployer Whisper, SD seulement si besoin et **pas en même temps que LLM si VRAM saturée**.
9. Configure nginx + TLS, monitoring.

---

10) Limitations & recommandations pratiques

- **Sur une seule 3060 12 Go**, attendre de **ne pas exécuter simultanément** LLM lourd + SD lourd. Planifie les jobs.
- Pour productivité, envisage **un petit second GPU** (ou machine secondaire) pour dédier SD ou LLM.
- **Test local** d'abord : charge un seul modèle (Qwen2.5-Coder 7B quantifié) et vérifie tokens/s.
- Prends l'habitude d'utiliser **quantized models (Q4_K_M)** pour tenir en 12 Go.

Si tu veux, je peux **générer pour toi** (immédiatement) :

- un **docker-compose complet et prêt à exécuter** (basé sur le YAML ci-dessus) avec dossiers/volumes et un script `setup.sh` pour Debian 12 (install drivers, docker, nvidia-toolkit), **OU**
- un **template FastAPI + worker** complet (fichiers Python) pour l'orchestrator RAG, incluant extraction PDF + embedding + qdrant calls, prêt à coller et lancer sur ta machine.

Dis-moi lequel tu veux que je crée tout de suite — je te le fournis prêt à copier/coller.

Très bonne question — il existe plusieurs modèles **open source / "free-weight"** (ou quasi) qui peuvent être très utiles pour du développement Python / Django + Vue.js, ou pour faire de la génération ou l'analyse de documents (PDF, etc.). Voici une sélection + quelques recommandations, avec leurs points forts + limites.

🌐 Quelques modèles open-source pertinents

Voici une liste de modèles que tu peux utiliser **localement** ou via des API open source, selon tes besoins (code, texte, document, résumé, etc.) :

Modèle	Points forts / usages pertinents
DeepSeek-R1 / DeepSeek-Coder	Très bon pour le code (génération, refactoring) selon des comparatifs. (bugbazdev.com)
Qwen / Qwen-Coder / Qwen-2.5-Coder	Série très forte pour le code : Qwen2.5-Coder (ex : 7B) est optimisée pour les tâches de codage. (Koyeb) Qwen3-Coder est aussi citée parmi les meilleurs. (KDnuggets) Le GitHub "Awesome-LLM" liste aussi Qwen3-Coder. (GitHub) Très bon pour des workflows de dev (Python, JS, etc.).
Mistral 7B / Mistral Small / Mixtral	Mistral est très efficace, bon compromis perf / taille. Selon dev: "Mistral 7B … rivalise dans les tâches de code" (DEV Community) La version "Codestral" (Mistral) est explicitement conçue pour du code : **Codestral-22B**, **Mamba-Codestral-7B**. (Mike Slinn)
GPT-J (6B)	Très "classique", bien pour des tâches de texte, du résumé, du traitement de document, ou des agents simples. (Wikipédia) Moins bon pour des tâches très complexes de codage comparé aux LLM plus récents / spécialisés, mais reste solide.
BLOOM	Modèle très large et multilingue. Peut servir pour du texte, du résumé, du traitement de documents PDF, des tâches de génération / compréhension, mais moins optimisé "spécial code". (Wikipédia)
TinyLlama	Modèle plus petit (1.1B) : très utile si tu veux faire des tâches plus légères, des résumés, ou des analyses sur des petits morceaux de texte, sans avoir un GPU ultra puissant. (arXiv)

✅ Modèles "plus pertinents" selon tes cas d'usage

Voici selon ton scénario **dev Python / Django + Vue.js + Postgres** + éventuellement "travailler sur des PDF" :

1. **Code (génération, suggestions, refactoring)**
   - *Qwen2.5-Coder (7B ou 14B)* : très bon pour du code, utilisation locale / dev.
   - *DeepSeek-Coder V2* : pour des tâches de codage plus avancées, refactoring, tests.
   - *Codestral (Mistral)* : très bon pour un modèle "code pur" optimisé.
2. **Traitement de documents (PDF, résumé, extraction)**
   - *BLOOM* : bon pour des tâches textuelles / de summarization / compréhension de documents.
   - *GPT-J* : plus léger, simple à déployer, utile pour des agents "analyser du PDF + extraire du texte".
   - *Mistral 7B* : peut faire du texte + résumé, raisonnable à faire tourner localement selon ta machine.
3. **Usage mixte (code + texte)**
   - *Qwen3-Coder* : si tu veux un modèle "tout-en-un" codage + conversation + documentation.

- *DeepSeek-R1* : bon “équilibre”, selon les versions (coder / général).

⚠️ Points de vigilance

- Même des modèles “open source” peuvent avoir des **licences spécifiques** : bien vérifier (Apache, licence “open model”, usage commercial, etc.).
- La qualité dépend fortement du **quantization** (4-bit, 8-bit) si tu fais du “local inference” avec des ressources limitées.
- Pour des tâches de lecture PDF + RAG (retrieval-augmented generation), il faudra souvent coupler un LLM + un vecteur / index (genre Chroma, FAISS) + un parser PDF (PyPDF2, pdfminer, etc.).
- Le modèle ne “sait pas toujours tout” : pour des PDF très techniques, la performance peut varier, et le LLM n’est pas un “OCR intelligent” mais un modèle de langage.

Si tu veux, je peux te donner une **sélection de 3 LLMs open source optimisés spécifiquement pour du dev Python / Django + doc PDF** (donc ce que *toi* tu utilises) — et te donner comment les intégrer dans ton app (via LangChain ou un agent). Veux-tu que je fasse ça ?

Parfait, je vais te construire **une stack complète**, prête pour une machine **Debian 12 + RTX 3060 12 Go + Django + QwenCoder + RAG PDF**, 100% locale sans cloud.

Cette stack comprend :
✔ Environnement Python 3.11
✔ Django API pour interroger le modèle
✔ Qwen2.5-Coder 7B local via Ollama ou ExllamaV2
✔ Extraction PDF locale (PyMuPDF)
✔ RAG complet : parse PDF → embeddings → index FAISS → requêtes
✔ Endpoint d’API JSON pour utiliser ton assistant IA dans ton frontend (Vue.js)

🧱 1) Pré-requis système Debian 12

Assure-toi que ton système est à jour :

```
sudo apt update && sudo apt upgrade -y
```

Installer les dépendances GPU (CUDA + drivers)

Debian 12 + RTX 3060 = install propre via nvidia-driver précompilé :

```
sudo apt install -y linux-headers-$(uname -r)
sudo apt install -y nvidia-driver firmware-misc-nonfree
```

Redémarre :

```
sudo reboot
```

Vérifie :

```
nvidia-smi
```

Tu dois voir **RTX 3060**.

🧰 2) Installer Ollama (moteur LLM local)

Ollama simplifie tout : téléchargement, quantization, exécution.

```
curl -fsSL https://ollama.com/install.sh | sh
```

🤖 3) Télécharger Qwen2.5-Coder (7B ou 14B)

La 3060 est parfaite pour **7B en Q4**.

```
ollama pull qwen2.5-coder
```

Tu peux tester :

```
ollama run qwen2.5-coder
```

📦 4) Créer projet Django pour l'API

```
mkdir ai_server && cd ai_server
python3 -m venv venv
source venv/bin/activate
pip install django djangorestframework pymupdf chromadb langchain faiss-cpu pydantic
requests
```

Créer le projet :

```
django-admin startproject backend
cd backend
python manage.py startapp rag
```

📄 5) Extraction PDF locale (PyMuPDF)

Dans `rag/pdf_utils.py` :

```python
import fitz

def extract_pdf_text(path):
    doc = fitz.open(path)
    text = ""
    for page in doc:
        text += page.get_text()
    return text
```

Simple, rapide, local.

🧠 6) Créer un index RAG (Chroma + FAISS)

Dans `rag/rag_engine.py` :

```python
import chromadb
from chromadb.config import Settings
from langchain.embeddings import HuggingFaceEmbeddings

chroma_client = chromadb.Client(Settings(chroma_db_impl="duckdb+parquet",
persist_directory="./chroma"))

embedder = HuggingFaceEmbeddings(
    model_name="sentence-transformers/all-MiniLM-L6-v2"
)

def create_index(doc_id, text):
    collection = chroma_client.get_or_create_collection("pdf_docs")
    chunks = split_text(text)
    embeddings = embedder.embed_documents(chunks)

    for i, chunk in enumerate(chunks):
        collection.add(
            ids=[f"{doc_id}-{i}"],
            documents=[chunk],
            embeddings=[embeddings[i]]
        )
    chroma_client.persist()

def query_rag(question):
    collection = chroma_client.get_collection("pdf_docs")
    results = collection.query(query_texts=[question], n_results=4)
    return results["documents"][0]

def split_text(text, max_len=700):
    words = text.split()
    out, buf = [], []
    for w in words:
        buf.append(w)
        if len(buf) >= max_len:
```

```
            out.append(" ".join(buf))
            buf = []
    if buf:
        out.append(" ".join(buf))
    return out
```

🤖 7) Endpoint Django pour interroger Qwen2.5-Coder + RAG

Dans `rag/views.py` :

```
import subprocess
import json
from rest_framework.response import Response
from rest_framework.decorators import api_view
from .rag_engine import query_rag
from .pdf_utils import extract_pdf_text, create_index

def ask_ollama(model, prompt):
    process = subprocess.Popen(
        ["ollama", "run", model],
        stdin=subprocess.PIPE,
        stdout=subprocess.PIPE,
        text=True
    )
    out, _ = process.communicate(prompt)
    return out

@api_view(['POST'])
def upload_pdf(request):
    pdf = request.FILES.get("file")
    if not pdf:
        return Response({"error": "no file"}, status=400)

    with open("uploaded.pdf", "wb") as f:
        for chunk in pdf.chunks():
            f.write(chunk)

    text = extract_pdf_text("uploaded.pdf")
    create_index("doc1", text)

    return Response({"status": "indexed"})

@api_view(['POST'])
def ask(request):
    question = request.data.get("question")
    context_chunks = query_rag(question)
    context = "\n\n".join(context_chunks)

    prompt = f"""
```

```
Tu es un assistant technique.
Voici les informations trouvées dans le document :

{context}

Maintenant réponds à la question :
{question}
"""
    answer = ask_ollama("qwen2.5-coder", prompt)
    return Response({"answer": answer})
```

🛣️ 8) Ajouter les routes Django

Dans `backend/urls.py` :

```
from django.contrib import admin
from django.urls import path
from rag.views import upload_pdf, ask

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/upload_pdf/', upload_pdf),
    path('api/ask/', ask),
]
```

🚀 9) Démarrer le serveur Django

```
python manage.py runserver 0.0.0.0:8000
```

Ton stack est maintenant opérationnel :

✔ Upload PDF → indexation

`POST /api/upload_pdf/` avec le fichier PDF

✔ Pose une question → réponse Qwen2.5 + RAG

```
POST /api/ask/
{
  "question": "Quels sont les articles clés du PDF ?"
}
```

🧩 10) Intégration Vue.js (simple)

Un service API :

```
async function ask(question) {
  const res = await fetch("http://localhost:8000/api/ask/", {
    method: "POST",
    headers: {"Content-Type": "application/json"},
    body: JSON.stringify({ question })
  });
  return res.json();
}
```

Et tu affiches `answer`.

🎯 Stack complète installée :

Composant	Technologie
LLM local	Qwen2.5-Coder 7B (Ollama)
Extraction PDF	PyMuPDF
Embeddings	SentenceTransformers
Index RAG	ChromaDB + DuckDB + FAISS
API Backend	Django REST Framework
Frontend	Vue.js

Si tu veux, je peux t'ajouter :
✅ un **Dockerfile complet**
✅ un **script Bash d'installation automatique Debian 12**
✅ un schéma d'architecture
Ou convertir ton stack en **FastAPI** si tu préfères.

Voici **un plan complet, concret et immédiatement exploitable** pour intégrer l'IA dans **tes deux plateformes SaaS** :

1. **SaaS Conciergerie**
2. **SaaS Logistique / Workflow / Traçabilité (scanner, documents, etc.)**

...en tirant profit de **ta stack locale** (Debian 12 + IA + n8n + Django/Vue/DRF + Docker).

Les services IA proposés sont **réalisables en local** avec les modèles que tu peux faire tourner (Qwen, Mistral, DeepSeek, Whisper, SD, RAG PDF).

🧠 1) IA pour ta plateforme Conciergerie (SaaS)

Cette plateforme concerne souvent :

- gestion de clients
- demandes d'intervention
- réservations
- facturation
- relation client
- tâches récurrentes
- communications avec prestataires

Voici les services IA les plus puissants pour ce domaine :

1.1 Chat IA connecté à la base Django (RAG SQL)

🔹 Un chatbot qui répond sur :

- état d'une commande
- solde client
- historique d'interventions
- disponibilité des prestataires
- statut d'une facture
- planning / calendrier

🛠️ Tech :

- Qwen2.5 7B → parfait pour questions opérationnelles
- RAG Postgres : convertir requêtes en SQL + vérifier/exécuter
- Retour JSON analysé puis formaté pour l'utilisateur

🎁 Valeur pour client :

- Accès immédiat à leur information sans support humain
- Autonomie et gain de temps

1.2 Génération automatique de documents officiels

🔹 Factures
🔹 Devis
🔹 Contrats de prestation
🔹 Comptes rendus
🔹 Rapports mensuels

🛠️ Tech :

- Django template + IA → générer le texte
- n8n → automatiser le flux (PDF → signature → envoi mail)

🎁 Valeur :

- Plus aucun document manuel
- Qualité homogène
- Envoi automatisé

1.3 Assistant Smart Workflow

Exemples :

- “Crée un nouveau workflow pour nettoyage Airbnb avec check-in > nettoyage > check-out”
- “Ajoute une tâche de rappel 24h avant la prestation”
- “Génère checklist selon type de logement”

🎁 Valeur :

- Les clients peuvent créer des process sans rien connaître de technique

---

1.4 Analyse intelligente des emails entrants

Cas d’usage :

- Email envoyé par un client → IA extrait :
  - adresse
  - date souhaitée
  - type de service
  - urgence
  - pièces jointes PDF (contrats, info logement)

🛠️ Tech :

- Whisper (si message vocal)
- LLM extraction JSON
- Auto-création de ticket via API Django

🎁 Valeur :

- Pas de support manuel → réduction de charge humaine

1.5 Assistant voyageurs / propriétaires

Tu peux proposer comme fonctionnalité PREMIUM :

- chatbot pour répondre automatiquement aux voyageurs
- traduction automatique
- assistant qui rédige messages Airbnb
- suggestion prix / optimisation planning
- analyse des commentaires voyageurs pour scoring

🎁 Valeur énorme :

- Réduit 70% du travail récurrent concierge / gestionnaire

🚚 2) IA pour plateforme Logistique / Scanning / Workflow / Traçabilité

Cas d'usage typiques :

- suivi colis
- scan QR/Barcode
- suivi entrepôt / picking
- mouvements stock
- contrôles qualité
- documents transporteurs

C'est un domaine où l'IA locale apporte **un gain massif**.

2.1 Chat IA connecté à la BDD Logistique

Le client peut demander :

- "Où en est le colis X ?"
- "Quel est le délai moyen du transporteur TNT ?"
- "Quelles livraisons sont en retard aujourd'hui ?"
- "Synthèse entrepôt Semaine 12"
- "Quel chauffeur a le meilleur taux de ponctualité ?"

🛠️ Tech :

- RAG SQL
- Vérification SQL + exécution contrôlée

🎁 Valeur :

- Remplace les requêtes SQL personnalisées
- Support niveau 1 automatisé

2.2 Analyse IA des documents PDF scannés (poids lourds !)

Par exemple :

- bons de livraison
- preuves de dépôt (POD)
- factures transporteurs
- documents CMR
- feuilles de route
- certificats de conformité

L'IA extrait :

- date
- expéditeur
- destinataire
- poids
- anomalies
- signature
- numéro de tracking
- montant
- TVA

🛠️ Tech :

- OCR (Tesseract / PaddleOCR)
- LLM extraction → JSON structure
- Intégration DB via API DRF

🎁 Valeur :

- Remplacement de la saisie manuelle (énorme gain)

2.3 Détection anomalies logistiques

Sur chaque scan ou workflow :

- colis manquant
- doublon
- erreur d'adresse
- rupture chaîne froid
- retard vs ETA
- incohérence poids / dimensions
- colis non scanné dans une zone

- panne récurrente d'un équipement

🛠️ Tech :

- règles + IA classification
- LLM pour générer message clair

🎁 Valeur :

- Réduction erreurs opérationnelles (gros ROI)

2.4 Génération automatique des rapports logistiques

Rapports instantanés :

- performance journalière
- performance chauffeurs
- anomalies journalières
- comparatifs entre périodes
- temps moyen entre étapes workflow
- taux de réussite scanning

🛠️ Tech :

- IA = texte + résumé
- Graphiques = Matplotlib/Plotly
- PDFs = générés automatiquement
- n8n → envoi automatique aux clients

🎁 Valeur :

- Moins d'analystes nécessaires
- Meilleure qualité reporting

2.5 Assistant visuel (via Stable Diffusion ou CLIP)

- analyse photos de dégâts colis
- reconnaissance type colis
- classification d'emballage
- comparaison état avant/après
- détection conformité (ex. étiquette lisible?)

🎁 Valeur :

- Déclare automatiquement sinistres ou anomalies

2.6 Assistant vocal chauffeurs / opérateurs

Avec Whisper tu peux créer :

- assistant vocal embarqué (Android)
- dictée de rapports
- suivi main libre
- déclaration incident en conduite

🎁 Valeur :

- Amélioration workflow terrain
- Moins de clics, plus de vitesse

🔄 3) n8n → Automatisation IA avancée (connecté à Django & IA)

Tu peux créer des **workflows automatisés** :
(ce que les clients adorent)

Exemple workflow :

"Quand un PDF est uploadé → OCR → LLM → JSON → API Django insert → email client confirm."

Autres automatisations :

- si livraison terminée → générer compte rendu
- si anomalie détectée → message Slack/Teams
- si chauffeur dépasse temps → alerte automatisée
- si facture reçue → extraction TVA → insertion compta
- si client pose question → Chat IA répond → log CRM

🎁 Valeur :
Tu apportes **une plateforme low-code AI automation** clef en main.

🔥 4) Services IA PREMIUM que tu peux vendre

Conciergerie

✔ Chat IA connecté base client
✔ Analyse messages voyageurs
✔ Génération automatique d'annonces et descriptions
✔ Analyse satisfaction clients
✔ Optimisation planning & pricing

Logistique

✔ OCR + extraction PDF automatisée
✔ Détection anomalie/retard
✔ Chat IA connecté au workflow
✔ Assistant chauffeurs vocal
✔ Prediction livraison ETA
✔ Génération rapports automatiques
✔ Analyse qualité opérationnelle

🎯 5) Ton avantage : INTÉGRATION FULL LOCAL

Tu peux vendre :

✔ Sécurité totale (pas de cloud)

✔ Données sensibles non exposées (logistique/crm)

✔ Performance GPU locale

✔ Solutions adaptées métier

✔ SLA garanti (ta plateforme + ta stack IA)

✔ Coût inférieur aux API OpenAI

C'est un argument massif pour les entreprises.

🧱 6) Je peux maintenant te fournir :

📌 Une **architecture complète pour ces plateformes (IA + Django + Vue + n8n)**
📌 Un **plan technique d'intégration IA dans Django (DRF)**
📌 Les **endpoints API IA** prêts à coller
📌 Un **workflow n8n complet**
📌 Un **schéma RAG SQL**
📌 Les **modèles Django** pour stocker documents + embeddings
📌 Le **docker-compose AI + SaaS unifié**

Dis-moi ce que tu veux en premier.