**tu es expert en deep learning, html, codage d application, je souhaite intégré une IA dans mon logiciel pour qu elle puisse faire le suivi de dossier , faire des rappels a l utilisateur de ces taches a faire quotidienne, conseiller l utilisateur**

**ChatGPT a dit :**

Réflexion durant 7 secondes

Pour intégrer un vrai « assistant IA » dans ton logiciel capable :

* de suivre l’état des dossiers,
* de rappeler chaque jour les tâches à faire,
* et de conseiller l’utilisateur,

voici une feuille de route technique et fonctionnelle :

**1. Architecture générale**

1. **Base de données :**
   * Table **dossiers** (statut, échéances, métadonnées).
   * Table **tâches** liées aux dossiers (description, date d’échéance, « rappel envoyé »).
   * Table **utilisateurs** (paramètres de notification, préférences).
2. **Backend (API)**
   * **Langage** : Python (Flask / FastAPI) ou Java (Spring Boot).
   * **Routes principales** :
     + GET /api/dossiers : liste et statut.
     + POST /api/dossiers/:id/taches : ajouter/mettre à jour une tâche.
     + GET /api/taches/rappels?date=YYYY-MM-DD : tâches à rappeler aujourd’hui.
     + POST /api/ia/conseil : envoie de contexte / question, renvoie une réponse de l’IA.
3. **Service de planification (Scheduler)**
   * **Celery** (Python) ou **Quartz** (Java) pour tâches périodiques.
   * Tous les matins (cron à 8h) :
     + Interroger GET /api/taches/rappels pour la date du jour.
     + Envoyer notifications (email / in-app).
   * Possibilité de rappels push / SMS via Twilio ou Firebase Cloud Messaging.
4. **Module IA**
   * **API OpenAI** (GPT-4/4o) ou modèle hébergé (e.g. Llama 2) pour le conseil
     + **Pipeline RAG** (Retrieval-Augmented Generation) :
       1. Extrait des infos du dossier (métadonnées, historique).
       2. Recherche dans la base (ou vector store) les cas similaires.
       3. Envoie le contexte + question à l’API IA pour un conseil contextualisé.
   * **Fine-tuning** possible si tu as beaucoup de cas d’usage spécifiques (scripts, réponses types).
5. **Frontend**
   * **Composant “Suivi de dossier”** : tableau listant statuts et prochains jalons.
   * **Composant “Rappels du jour”** : affiche les tâches à faire aujourd’hui.
   * **Composant “Conseil IA”** : zone de chat/question où l’utilisateur pose ses questions.

**2. Exemple de flux “rappel quotidien”**

1. **Celery :** tâche planifiée à 08:00 → appelle GET /api/taches/rappels?date=2025-07-08.
2. **Backend** récupère les tâches dont date\_echeance = 2025-07-08 et rappel\_envoye = false.
3. **Backend** envoie un email ou un push in-app (via WebSocket) à l’utilisateur.
4. **Backend** marque rappel\_envoye = true pour ne pas renvoyer.
5. **Frontend** reçoit l’événement push et affiche une notification.

**3. Exemple de code minimal (FastAPI + Celery + OpenAI)**

python

CopierModifier

# backend/app.py

from fastapi import FastAPI

from pydantic import BaseModel

import datetime, databases

import openai

app = FastAPI()

db = databases.Database("postgresql://user:pass@localhost/db")

openai.api\_key = "TA\_CLE\_OPENAI"

class Question(BaseModel):

dossier\_id: int

message: str

@app.get("/api/taches/rappels")

async def get\_rappels(date: str):

query = "SELECT \* FROM taches WHERE date\_echeance = :date AND rappel\_envoye = false"

return await db.fetch\_all(query, values={"date": date})

@app.post("/api/comptabilite/objectif-ca")

async def get\_objectif\_ca(date: str):

# Exemple simplifié

result = await db.fetch\_one("SELECT objectif\_ca FROM objectifs WHERE date = :d", {"d": date})

return {"objectif\_ca": result["objectif\_ca"] if result else None}

@app.post("/api/ia/conseil")

async def conseil(data: Question):

# Récupérer contexte du dossier

dossier = await db.fetch\_one("SELECT \* FROM dossiers WHERE id = :id", {"id": data.dossier\_id})

# Optionnel : recherche de cas similaires, injection du contexte

prompt = f"Contexte dossier: {dossier['description']}.\nUtilisateur: {data.message}\nConseils :"

resp = openai.ChatCompletion.create(

model="gpt-4o-mini",

messages=[{"role": "system", "content": "Tu es un assistant pro."},

{"role": "user", "content": prompt}]

)

return {"conseil": resp.choices[0].message.content}

python

CopierModifier

# worker/tasks.py

from celery import Celery

from datetime import date

import requests

celery = Celery(...)

@celery.on\_after\_configure.connect

def setup\_periodic\_tasks(sender, \*\*kwargs):

# Chaque jour à 8h

sender.add\_periodic\_task(

crontab(hour=8, minute=0),

send\_daily\_reminders.s()

)

@celery.task

def send\_daily\_reminders():

today = date.today().isoformat()

r = requests.get(f"http://localhost:8000/api/taches/rappels?date={today}")

for t in r.json():

# Envoi de notification (mail, push…)

notify\_user(t["user\_id"], f"N’oubliez pas : {t['description']} à faire aujourd’hui.")

# Marquer comme envoyé

requests.post(f"http://localhost:8000/api/taches/{t['id']}/mark\_sent")

**4. Conseils de mise en œuvre**

* **Sécurité & permissions** : authentification (JWT/OAuth2) pour les routes IA, dossiers et rappels.
* **UI/UX** : un bandeau ou toast pour les rappels ; zone de chat enrichie (historique, réponses sauvegardées).
* **Monitoring** : logs Celery, taux d’erreur IA, temps de réponse.
* **Montée en charge** : mettre en cache les conseils pour des questions fréquentes.

Avec cette base, tu pourras :

1. Construire un **suivi automatique** des dossiers et rappels.
2. Offrir un **assistant conversationnel** personnalisé pour conseiller l’utilisateur sur ses cas.
3. Ajuster et **affiner** les modèles IA selon tes besoins métier.

N’hésite pas si tu veux un exemple plus poussé sur un point (front React, workflow RAG, intégration de notifications…).

**Vous avez dit :**

dans quoi il faut que je code ca ?

**ChatGPT a dit :**

Réflexion durant quelques secondes

Pour ce projet hybride IA + application métier, je te conseille de séparer clairement deux couches :

**1. La couche « moteur IA »**

**Langage :**

* **Python** est quasi-incontournable pour le Deep Learning et l’intégration de modèles (PyTorch, TensorFlow, scikit-learn),
* et pour appeler sans friction l’API OpenAI (ou tout autre service) via leur SDK officiel.

**Frameworks / modules clés :**

* **FastAPI** ou **Flask** pour exposer des endpoints REST/JSON (ex. : /api/ia/conseil, /api/ia/rappels),
* **Celery** + **Redis** (ou RabbitMQ) pour la planification et l’envoi de rappels périodiques,
* **SQLAlchemy** ou **Tortoise ORM** pour gérer la BDD des dossiers, tâches et états de rappel.

**Exemple de répartition :**

* ia\_service.py (FastAPI) reçoit la question, interroge la BDD et fait un RAG + appel OpenAI.
* tasks.py (Celery) planifie un « send\_daily\_reminders » chaque matin.

**2. La couche « application »**

**Backend métier**

* Toujours en **Python**, tu peux regrouper tes APIs gestion de dossiers, facturation, etc. dans le même service FastAPI, ou scinder en micro-services distincts.
* Si tu es plus à l’aise en **Java**, **Spring Boot** offre un écosystème robuste, mais tu perdras la souplesse native Python/IA.

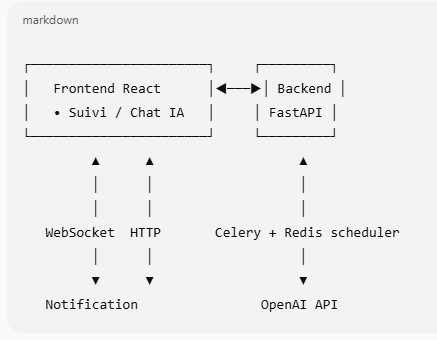
**Frontend**

* **Web** :
  + **React** ou **Vue.js** pour construire des composants dynamiques (suivi de dossier, rappels, chat IA).
  + **Next.js** si tu veux du server-side rendering + API routes intégrées.
* **Desktop** (si ton logiciel est une application lourde) :
  + **Electron** (wrapper Chromium + Node.js) pour réutiliser tes composants Web.
  + **PyQt** ou **Tkinter** en Python, mais l’UX est moins moderne.

**3. Exemple d’architecture**

markdown

CopierModifier



* **WebSocket** pour push instantané de rappels in-app.
* **Celery** déclenche chaque matin (08:00) un job qui :
  1. lit la BDD des tâches à faire aujourd’hui,
  2. envoie emails/push/WebSocket,
  3. marque rappel\_envoye = true.

**Résumé**

* **Code IA / Deep Learning** → **Python** (FastAPI, Celery, PyTorch/TensorFlow)
* **API métier & rappels** → **mêmes services FastAPI** ou micro-services en Python
* **Frontend** → **React/Next.js** (ou Electron si tu veux une app desktop hybride)

Ça te donne un écosystème cohérent, où la partie IA reste en Python (librairies, SDK), et le front en JS/TS pour l’interaction utilisateur.