Olivier Dahan

Contact: [odahan@gmail.com](mailto:odahan@gmail.com)

Web: [www.e-naxos.com/blog](http://www.e-naxos.com/blog)

Consulting, assistance, développement .NET



PIPELINE IA LOCAL EN C# (LLM/RAG)

# Introduction

**Série « IA & C# pour l’entreprise »**

**Le présent document accompagne la vidéo « RAG Local en C# » ainsi que les fichiers sources des exemples. Le tout étant publié sur GitHub et formant un ensemble non séparable.**

**Lire la licence qui accompagne l’ensemble des fichiers, à la racine du dépôt GitHub, fichier Licence.txt.**

**Ce document reprend les informations données dans la vidéo afin de rendre plus aisée la mise en place d’une copie de l’environnement démontré. Il s’agit donc juste d’un support en complément de la vidéo et non d’un cours indépendant.**

**Comme le précise la licence, aucune garantie de fonctionnement n’est offerte, les informations ne sont fournies qu’à titre éducatif et ne peuvent être ni copiée ni utilisées sans l’accord de l’auteur. Aucun support n’est offert avec la vidéo ni avec les fichiers présents sur GitHub dont le présent document. Si vous constatez des erreurs ou coquilles vous pouvez transmettre cette information à l’auteur.**

**En cas de doute consultez toujours la documentation officielle du produit concerné (Ollama, Qdrant…).**

# Pré-requis

1. **.NET SDK 9** installé
2. dotnet –info  
   → Vérifie que la ligne Version: indique **9.x**.
3. **PowerShell** (celui de Windows suffit).
4. **curl** disponible (intégré à Windows 11).
5. **Docker Desktop installé et à jour**

# Etape 1 - LLM local

Objectif : un seul Ollama natif (pas de conteneur), + modèle léger et rapide.

1. **Installer Ollama** (natif) – si déjà fait et OK, garder tel quel. (Ollama.com)
2. **Attention** en fin d’install une UI est lancée, ne pas l’utilisée elle charge d’autres modèles, la fermer et continuer la procédure.
3. **Charger un modèle (llama3, qwen2.5 ou 3, mistral… page** [**https://ollama.com/library**](https://ollama.com/library) **)** :
4. Commande « ollama pull llama3 » pour télécharger le modèle « llama3 » par exemple.
5. Commande « ollama list » pour voir la liste des modèles installés
6. Lancer « **ollama run llama3** » pour lancer le modèle llama3 dans la console (ou choisir un autre modèle)
7. Taper « /bye » et Envoi pour sortir du mode conversationnel.
8. **Test de santé** (API locale) syntaxe pour CMD et non PowerShell (en une seule ligne !) :   
   curl.exe http://localhost:11434/api/generate -H "Content-Type: application/json" -d "{\"model\":\"llama3\",\"prompt\":\"Bonjour en une phrase.\",\"stream\":false}"

Attendu : JSON, exemple réel de réponse reçue :   
{"model":"llama3","created\_at":"2025-09-11T20:14:12.4521694Z","response":"\"Bonjour, comment allez-vous?\" (Hello, how are you?)","done":true,"done\_reason":"stop","context":[128006,882,128007,271,82681,665,6316,17571,13,128009,128006,78191,128007,271,1,82681,11,4068,12584,89,45325,7673,320,9906,11,1268,527,499,10380],"total\_duration":2650390600,"load\_duration":2415473500,"prompt\_eval\_count":15,"prompt\_eval\_duration":97176400,"eval\_count":16,"eval\_duration":137740700}

Attention : llama3 en natif n’utilise pas le GPU il est optimisé pour le CPU. La commande **nvidia-smi** permet de vérifier l’utilisation de la VRAM avant et après l’installation (et le lancement de llama3). D’autres modèles peuvent être optimisés autrement.

Ollama se lance au démarrage de Windows.   
Pour vérifier ou activer / désactiver : *Windows settings / applications / démarrage et chercher Ollama.*

Les modèles sont chargés à la demande lors 1er accès d’API. Par défaut ils sont déchargés au bout de 5mn. On peut changer ces durées si nécessaire. Le 1er appel à l’API a donc une latence pour le chargement du modèle et si la 2de demande intervient après le délai (5mn par défaut) il y a aura à nouveau latence.

Voici une commande **curl** qui permet de faire un warm-up et de conserver le model 1h (plus si des requêtes interviennent entre temps ce qui remet le compteur à zéro) :  
curl.exe http://localhost:11434/api/generate -H "Content-Type: application/json" -d "{\"model\":\"llama3\",\"prompt\":\"warmup\",\"stream\":false,\"keep\_alive\":\"1h\"}"

Si Ollama ou Windows sont arrêtés, il faut recommencer car ce n’est pas persistant. On peut mettre 24h au lieu d’une heure si pas de problème de VRAM. On peut aussi créer un Bat et l’ajouter au démarrage de Windows.

Prompt pour tester rapidement si un modèle sait raisonner correctement, s’il maîtrise le français et s’il peut écrire un mail même avec aucun contexte (c’est super court, pas scientifique, mais cela permet de comparer rapidement les sorties de plusieurs modèles sous Ollama par exemple) :

**Rédige un email de réponse à un client qui signale un retard de livraison. Explique brièvement les raisons du délai, propose une solution et termine de façon polie et professionnelle.**

# Etape 2 - Base vectorielle

Objectif : Qdrant isolé, persistant et simple à remettre d’aplomb.

1. **Installer Docker Desktop** (si absent).  
   <https://docs.docker.com/desktop/setup/install/windows-install/>
2. **Lancer Docker desktop et signup**
3. **Démarrer Qdrant** avec un **volume nommé** (Windows-friendly) :  
     
   docker run -d --name qdrant -p 6333:6333 -p 6334:6334 -v qdrant\_data:/qdrant/storage qdrant/qdrant:latest
4. **Test de santé** :
   * Ouvrir dans le browser http://localhost:6333/ → JSON d’info Qdrant.  
     Réponse réelle (exemple) :   
     {"title":"qdrant - vector search engine","version":"1.15.4","commit":"20db14f87c861f3958ad50382cf0b69396e40c10"}
   * Ou utiliser la commande console :
   * curl http://localhost:6333/

Note versioning (optionnel) : On peut “geler” l’image après test avec la commande PowerShell :  
  
docker inspect qdrant | Select-String -Pattern "Image"  
  
Puis retirer plus tard via son **digest** si on veut une reproductibilité stricte. Si la commande retourne une version « *latest* » il faudra passer par une autre commande pour obtenir le digest (le sha de la version réellement utilisée) :  
  
docker inspect quay.io/qdrant/qdrant:latest | Select-String -Pattern "Digest"  
  
La sortie donnera un digest (exemple `sha256:abcd1234...`).  
  
Il faut conserver ce digest quelque part.   
Pour réutiliser plus tard la même install avec l’exacte même version on utilisera une commande de ce type :   
  
docker pull [quay.io/qdrant/qdrant@sha256:abcd1234](mailto:quay.io/qdrant/qdrant@sha256:abcd1234)...

On remarque l’utilisation de l’arobase qui indique le digest et non une version.

# Etape 3 - Embeddings — service dédié (Docker, CPU-only)

Objectif : découpler complètement les embeddings du LLM. Zéro Python installé **sur l’hôte** (tout est dans le conteneur). Modèle : **all-MiniLM-L6-v2** (dimension **384**), robuste et rapide.

1. **Créer le serveur d’embedding** (API /embed) :

Crée un dossier **embedding-server** avec ces 2 fichiers.

embedding-server**/Dockerfile**

FROM python:3.10-slim

RUN apt-get update && apt-get install -y --no-install-recommends \

build-essential curl ca-certificates && rm -rf /var/lib/apt/lists/\*

RUN pip install --no-cache-dir flask sentence-transformers

WORKDIR /app

COPY server.py .

ENV PORT=80

EXPOSE 80

CMD ["python", "server.py"]

Cela va permettre de créer un conteneur Docker prêt à exécuter l’application Flask qui utilise `sentence-transformers` pour faire du NLP, tout en restant aussi légère que possible. Flask est un micro framework web en python qui va servir à démarrer un serveur qui exposera le service d’embeddings. Si vous ne comprenez pas Python vous pouvez demander à une IA de vous expliquer ligne par ligne ce DockerFile vous verrez c’est vraiment simple. Cela permet d’emballer le serveur d’embeddings avec un serveur web dans un conteneur, et ensuite de ne plus avoir à dealer avec python. Il existe d’autres moyens, moins simples, et même une librairie .NET permettant de choisir le moteur de calcul des embeddings, mais cela crée des dépendances externes non Microsoft en plus et ici j’ai préféré une solution fiable à 100% et surtout 100% autonome sans dépendance.

embedding-server**/server.py**

import os

from torch import set\_num\_threads, set\_num\_interop\_threads

set\_num\_threads(int(os.getenv("TCH\_THREADS", "6")))

set\_num\_interop\_threads(2)

from flask import Flask, request, jsonify

from sentence\_transformers import SentenceTransformer

MODEL\_NAME = "sentence-transformers/all-MiniLM-L6-v2"

app = Flask(\_\_name\_\_)

model = SentenceTransformer(MODEL\_NAME)

@app.route("/embed", methods=["POST"])

def embed():

data = request.get\_json(force=True)

text = data.get("text", "")

if not isinstance(text, str) or not text.strip():

return jsonify({"error": "Field 'text' must be a non-empty string"}), 400

vec = model.encode(text).tolist() # 384 floats

return jsonify({"model": MODEL\_NAME, "embedding": vec})

@app.route("/ready", methods=["GET"])

def ready():

return jsonify({"status": "ok", "model": MODEL\_NAME})

if \_\_name\_\_ == "\_\_main\_\_":

app.run(host="0.0.0.0", port=80)

Ce fichier crée un petit service web qui, lorsqu’on lui envoie un texte, retourne le vecteur d’embedding correspondant. C’est le cœur de l’« embedding‑server » que l’on construit et exécute ensuite avec Docker. On note au départ la préparation de PyTorch qui fixe le nombre de threads à 6 et limite aussi le nombre de threads pour les opérations inter-processus. Je vous conseille de laisser comme ça ou d’adapter selon la puissance de votre machine, mais ne laissez pas sans limite, des appels rapprochés d’embeddings pourraient saturer la machine en consommant tous les threads jusqu’au thread Windows. Pas d’écran vert, mais une machine qui se fige complètement. C’est ce qui m’ait arrivé la première fois et que ces lignes permettent d’éviter !

1. **Build & run** (fabrique le container à partir des fichiers d’instructions) :
2. cd embedding-server  
   docker build -t local-embed .
3. puis :

docker run -d --name local-embed -p 5000:80 ^

--cpus="6.0" --memory="6g" --memory-swap="6g" ^

-e OMP\_NUM\_THREADS=6 -e MKL\_NUM\_THREADS=6 -e OPENBLAS\_NUM\_THREADS=6 -e NUMEXPR\_NUM\_THREADS=6 -e TCH\_THREADS=6 ^

local-embed

1. **Tests de santé** :
2. curl <http://localhost:5000/ready>  
   réponse réelle exemple : page d’info *dont {"model":"sentence-transformers/all-MiniLM-L6-v2","status":"ok"}*
3. Powershell test :  
   curl.exe --% -X POST "http://localhost:5000/embed" -H "Content-Type: application/json" -d "{\"text\":\"La Révolution française a transformé la société.\"}"  
     
   Réponse réelle (exemple) :

{"embedding":[-0.02790573053061962,0.011535847559571266,-0.022395676001906395,-0.08052609115839005,0.003917482681572437,0.04993533715605736,-0.028205454349517822,0.017972296103835106,0.049399010837078094,0.002972088987007737,0.036203786730766296,0.055330049246549606,0.014727460220456123,-0.02313794195652008,-0.0222336258739233,-0.08712176978588104,-0.07771416753530502,0.07159534841775894,-0.021875916048884392,0.024243880063295364,-0.023579824715852737,-0.060482367873191833,-0.05075937882065773,0.05053355544805527,0.007410505786538124,-0.014118862338364124,-0.009665355086326599,-0.05519307032227516,-0.007685751188546419,-0.020028382539749146,-0.015148676931858063,0.07936578243970871,-0.07941603660583496,-0.010385987348854542,-… … … … … … … … 0.0025942842476069927,0.004793552216142416,0.0384165421128273,-0.03875252231955528,-0.02453637309372425,-0.001908507663756609,0.009254657663404942,0.07677903026342392,-0.10427062958478928],"model":"sentence-transformers/all-MiniLM-L6-v2"}

Attendu : JSON avec "embedding": [ ... 384 valeurs ... ].

# Etape 4 - RAG minimal en .NET (sans SK) — validation bout-en-bout

Objectif : éprouver la tuyauterie **avant** d’ajouter éventuellement Semantic Kernel (le diagnostic est plus simple en évitant toute librairie et en restant en http).

1. **Créer un projet console** :
2. dotnet new console -n **TestSetup** -f net9.0
3. cd **TestSetup**
4. **Ouvrir le projet TestSetup**
5. **Exécuter** : dotnet run

On doit voir à la console :

* + **Dimension** détectée ≈ **384**
  + **Création**/détection de la collection
  + **Ingestion** de 2–3 textes
  + **Recherche** topK avec scores
  + **Réponse** de llama3 basée sur le contexte

À ce stade, on a l’assurance que **chaque brique** fonctionne et que **les dimensions** concordent.

**Sortie réelle exemple :**

=== RAG local minimal (UUID) : Embeddings -> Qdrant -> Ollama ===

[Check] Embeddings OK

[Check] Qdrant OK

[Check] Ollama OK

[Embed] Dimension détectée = 384

[Qdrant] Collection 'docs\_demo' déjà existante.

[Qdrant] Ingesté 3 document(s).

[TopK] Passages retenus :

- (score 0,470) Qdrant est une base de données vectorielle performante et simple à déployer.

- (score 0,395) Le RAG combine une base vectorielle et un modèle génératif.

- (score 0,116) .NET 9 apporte des améliorations de performance et du tooling modernisé.

[LLM] Réponse exemple :

*Le RAG est un combinatoire qui réunit une base de données vectorielle (Qdrant) avec un modèle génératif. Le rôle de Qdrant dans ce contexte est donc de fournir la base de données vectorielle performante et simple à déployer, qui servira de fondement au RAG pour stocker et traiter les données.*

=== Terminé ===

# Etape 5 - Reranking

Pour améliorer les résultats RAG, après la recherche QDrant on peut effectuer une étape de reranking qui consiste à affiner les résultats retournés pour une meilleure pertinence et ne conserver que les passages de texte vraiment pertinents (et diminuer la taille du contexte accessoirement).

Petit service local qui tournera dans un conteneur Docker :

Dockerfile

FROM python:3.10-slim

# Certificats TLS pour HTTPS (Hugging Face), nettoyage après install

RUN apt-get update \

&& apt-get install -y --no-install-recommends ca-certificates \

&& rm -rf /var/lib/apt/lists/\*

RUN pip install --no-cache-dir flask sentence-transformers

WORKDIR /app

COPY server\_rerank.py .

ENV RERANK\_MODEL=BAAI/bge-reranker-v2-m3

ENV PORT=80

# (Optionnel) Cache HF persistant si tu montes un volume

# ENV HF\_HOME=/root/.cache/huggingface

CMD ["python", "server\_rerank.py"]

-------- server\_rerank.py ------

import os

from flask import Flask, request, jsonify

from sentence\_transformers import CrossEncoder

app = Flask(\_\_name\_\_)

MODEL\_NAME = os.getenv("RERANK\_MODEL", "BAAI/bge-reranker-v2-m3")

# Chargement au démarrage (simple). Si tu préfères un chargement lazy,

# on peut déplacer ce code dans la route /rerank ou /ready.

cross = CrossEncoder(MODEL\_NAME)

@app.route("/ready", methods=["GET"])

def ready():

return jsonify({"status": "ok", "model": MODEL\_NAME})

@app.route("/rerank", methods=["POST"])

def rerank():

data = request.get\_json(force=True)

query = data.get("query", "")

cands = data.get("candidates", [])

top\_n = int(data.get("top\_n", 5))

if not query or not cands:

return jsonify({"error": "missing query or candidates"}), 400

pairs = [(query, c) for c in cands]

scores = cross.predict(pairs).tolist()

ranked = sorted(zip(cands, scores), key=lambda x: x[1], reverse=True)[:top\_n]

return jsonify({

"model": MODEL\_NAME,

"items": [{"text": t, "score": float(s)} for t, s in ranked]

})

if \_\_name\_\_ == "\_\_main\_\_":

port = int(os.getenv("PORT", "80"))

# 0.0.0.0 indispensable pour exposer le port à l’hôte Docker

app.run(host="0.0.0.0", port=port)

**--Build et run—**

docker build -t local-rerank .

docker rm -f local-rerank 2>NUL

docker run -d --name local-rerank -p 5001:80 ^

-e RERANK\_MODEL="BAAI/bge-reranker-v2-m3" ^

-e PORT=80 ^

local-rerank

--test—

curl.exe <http://localhost:5001/ready>

réponse réelle exemple : {"model":"BAAI/bge-reranker-v2-m3","status":"ok"}

code **skMinimalRagRerank**

C’est une modification de **skMinimalRag** avec l’ajout du Reranking : système de choix par ligne de commande + menu console + nouvelle fonction utile et ses records, + liste de Docs beaucoup plus longue (environ 40 lignes au lieu de 3) + séquence de reranking après recherche sur QDrant.

Les affichages montrent l’effet très net du reranking.

code **skMinimalRagRerankChunk**

* Ajout du traitement de docx/pdf -> chunk -> qdrant
* Prompt ouvert pour interroger les docs

Se base sur le code précédent qui lui-même se basait sur le précédent, etc. Seul le 1er exemple sans SK est différent volontairement puisque pas de SK du tout. Pour l’instant SK est juste ajouté mais pas utilisé complètement, loin s’en faut, c’est une progression lente par étape voulue. SK sert juste de glue on aurait pu faire sans. Et dans le futur je vous montrerai comment utiliser son successeur, Agent Framework (juste annoncé en preview au moment où je corrige ce texte).

L’étape suivante sera de faire traiter les chunks par SK ou AF au lieu de le faire à la main dans le code. Mais ici je reste sur l’idée d’un Pipeline local complet interrogeable en http. Le code exemple ne sert qu’à valider le pipelne, rien de plus.

Les packages suivants permettent de lire les PDF et les Docx (et Extensions.AI est la base de SK et de AF).

dotnet add package PdfPig

dotnet add package DocumentFormat.OpenXml

Microsoft.Extensions.AI

Avant run : lancement des serveurs dans Docker et vérification par curl :

* Qdrant  
  curl http://localhost:6333/  
  réponse :   
  {"title":"qdrant - vector search engine","version":"1.15.4","commit":"20db14f87c861f3958ad50382cf0b69396e40c10"}
* Server d’embeddings  
  curl <http://localhost:5000/ready>  
  Réponse :  
  {"model":"sentence-transformers/all-MiniLM-L6-v2","status":"ok"}
* Serveur de Reranking  
  curl <http://localhost:5001/ready>  
  Réponse :  
  {"model":"BAAI/bge-reranker-v2-m3","status":"ok"}

La limitation des threads du serveur python embeddings a été corrigé (pris en compte plus haut), mais il faut relancer qdrant lui aussi avec des limites cela évitera de figer votre machine. Livre à vous de modifier les limites si vous comprenez les commandes : (powershell)

Pour les embeds :   
docker rm -f local-embed 2>$null; docker run -d --name local-embed -p 5000:80 --cpus 6 --memory 6g --memory-swap 6g -e OMP\_NUM\_THREADS=6 -e MKL\_NUM\_THREADS=6 -e OPENBLAS\_NUM\_THREADS=6 -e NUMEXPR\_NUM\_THREADS=6 local-embed

Vérification :   
Invoke-WebRequest http://localhost:5000/ready | Select-Object -ExpandProperty Content

Pour Qdrant :

docker rm -f qdrant 2>$null; docker run -d --name qdrant --cpus 2 --memory 2g --memory-swap 2g -p 6333:6333 -p 6334:6334 -v qdrant\_data:/qdrant/storage qdrant/qdrant:latest

Vérification :

Invoke-WebRequest http://localhost:6333/ | Select-Object -ExpandProperty Content

# Etape 6 - Dépannage ciblé (les vrais cas qui coincent)

* **Pare-feu Windows** : si un port est bloqué, créer une règle entrante pour 11434, 5000, 6333, 6334.
* **Conflit de ports** : modifie le -p côté docker run **et** l’URL dans le code.
* **Qdrant “dimension mismatch”** : la collection doit avoir **384** (MiniLM). Supprimer la collection et la recréer si elle a été créée avec une autre dimension.
* **Ollama ne répond pas** : fermer toute UI “chat” Ollama gardant le modèle en RAM, puis relancer. Vérifier ollama list.
* **Embeddings lents** : c’est du CPU-only ; pour accélérer, on pourrait passer plus tard à un runtime GPU (DirectML/CUDA) ou à un modèle d’embedding plus puissant (et plus lourd).
* **Français moyen** : Si la qualité du Français généré par le LLM choisi n’est pas parfait, changer de modèle ou ajouter des consignes de style dans le message “system” (ex. “registre soutenu, évite les anglicismes, répondre en 6–8 lignes max.”).

FIN !



Olivier Dahan

Contact: [odahan@gmail.com](mailto:odahan@gmail.com)

Web: [www.e-naxos.com/blog](http://www.e-naxos.com/blog)

Consulting, assistance, développement .NET