Manual Unificado — Instalación, Configuración y Uso de una Solución LLM Local

Este manual reúne y organiza la información de:
• Guia_LLM_Local_PASO_A_PASO.docx
• Guia_LLM_Local.docx
para ofrecer un único documento claro, completo y ejecutable.

Índice

1. Visión general y arquitectura

2. Preparación del entorno (Windows 11 / macOS / Linux)

3. Instalación de Ollama y descarga de modelos (Opción A)

4. Opción A — API local con FastAPI (sin RAG)

5. Opción A — Variante con RAG (Chroma + embeddings locales)

6. Opción B — API con OpenAI (requiere API key)

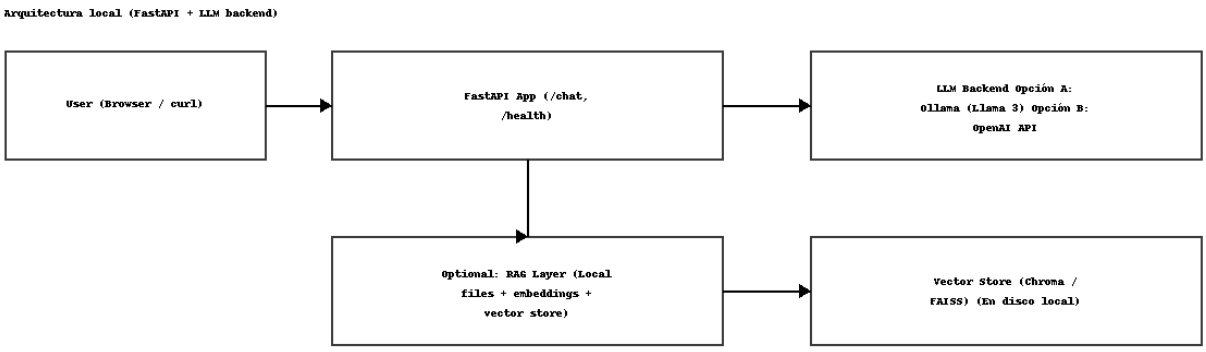
7. Dataset de ejemplo y estructura de proyecto

8. Verificación (smoke tests)

9. Troubleshooting y buenas prácticas

1) Visión general y arquitectura

El objetivo es ejecutar localmente una API REST que use un modelo de lenguaje (LLM) para responder preguntas. Se presentan dos rutas: (A) 100% local con Ollama y (B) vía OpenAI API. Adicionalmente, se muestra cómo agregar RAG con un vector store local (Chroma) para mejorar la calidad con tus documentos.



2) Preparación del entorno

Requisitos: Python 3.10+ (ideal 3.11), pip, (opcional) Git. Se recomienda usar un entorno virtual (venv).

Windows 11 (PowerShell):

```
winget source update
winget install -e --id Git.Git
winget install -e --id Python.Python.3.11
# Cierra y reabre PowerShell
python --version
pip --version

# Carpeta y venv
mkdir C:\llm-local && cd C:\llm-local
python -m venv .venv
.\.venv\Scripts\Activate.ps1
python -m pip install --upgrade pip
```

macOS:

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install python git
python3 --version
pip3 --version
```

```
mkdir ~/llm-local && cd ~/llm-local
python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
```

Linux (Ubuntu/Debian):

```
sudo apt update
sudo apt install -y python3 python3-pip python3-venv git curl
python3 --version
pip3 --version
mkdir ~/llm-local && cd ~/llm-local
python3 -m venv .venv
source .venv/bin/activate
python -m pip install --upgrade pip
```

3) Instalar Ollama y descargar modelos (Opción A)

Descarga Ollama e instala según tu SO. Luego descarga los modelos necesarios.

```
# Windows/macOS: https://ollama.com/download
# macOS con brew:
brew install --cask ollama
# Linux:
curl -fsSL https://ollama.com/install.sh | sh

# Verificar e iniciar (si fuese necesario):
ollama --version
# Descargar modelos (LLM y embeddings para RAG):
ollama pull llama3
ollama pull nomic-embed-text
```

4) Opción A — API local con FastAPI (sin RAG)

Crea los siguientes archivos en tu carpeta de proyecto (p.ej., C:\llm-local):

requirements.txt:

```
fastapi
uvicorn
pydantic
```

```
requests
```

app.py:

```
from fastapi import FastAPI
from pydantic import BaseModel
import requests

app = FastAPI()

class Query(BaseModel):
    q: str

@app.get("/health")
def health():
    return {"status": "ok"}

@app.post("/chat")
def chat(q: Query):
    resp = requests.post("http://localhost:11434/api/generate", json={
        "model": "llama3",
        "prompt": q.q,
        "stream": False
    })
    data = resp.json()
    return {"answer": data.get("response", "(sin respuesta)")}
```

Instalar y ejecutar:

```
pip install -r requirements.txt
uvicorn app:app --host 0.0.0.0 --port 8080
```

Pruebas (otra terminal):

```
curl http://localhost:8080/health
curl -X POST http://localhost:8080/chat -H "Content-Type: application/json" -d
"{"q":"Hola, ¿quién eres?"}"

$body = @{ q = Hola, Quien eres?" } | ConvertTo-Json

$r = Invoke-RestMethod -Uri "http://localhost:8080/chat" -Method POST -Body
$body -ContentType 'application/json; charset=utf-8'
```

```
$r.answer
```

5) Opción A — Variante con RAG (Chroma + embeddings locales)

Estructura sugerida: coloca tus .txt en ./data/. Se usarán embeddings locales con Ollama y se indexará en Chroma.

requirements_rag.txt:

```
langchain
chromadb
tiktoken
requests
fastapi
uvicorn
pydantic
```

index.py (ingesta + embeddings locales):

```
import glob, os, requests, chromadb

def embed(text: str):
    r = requests.post("http://localhost:11434/api/embeddings", json={
        "model": "nomic-embed-text",
        "prompt": text
    })
    r.raise_for_status()
    return r.json()["embedding"]

def main():
    chroma = chromadb.Client()
    coll = chroma.get_or_create_collection("docs")
    files = glob.glob("./data/*.txt")
    if not files:
        print("No se encontraron archivos en ./data/.")
    for path in files:
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
            content = f.read()
        vec = embed(content)
        coll.add(documents=[content], embeddings=[vec],
ids=[os.path.basename(path)])
        print(f"Indexado: {os.path.basename(path)}")
    print("Indexado OK.")
```

```
if __name__ == "__main__":
    main()
```

app_rag.py (API con recuperación + generación):

```
from fastapi import FastAPI
from pydantic import BaseModel
import requests, chromadb

app = FastAPI()
coll = chromadb.Client().get_or_create_collection("docs")

class Query(BaseModel):
    q: str

def embed(text: str):
    r = requests.post("http://localhost:11434/api/embeddings", json={
        "model": "nomic-embed-text",
        "prompt": text
    })
    r.raise_for_status()
    return r.json()["embedding"]

@app.get("/health")
def health():
    return {"status": "ok"}

@app.post("/chat")
def chat(q: Query):
    qvec = embed(q.q)
    hits = coll.query(query_embeddings=[qvec], n_results=3)
    context = "\n".join(hits.get("documents", [[""]])[0])
    prompt = f"Usa el contexto para responder con
precisión:\n{context}\n\nPregunta: {q.q}"
    r = requests.post("http://localhost:11434/api/generate", json={
        "model": "llama3",
        "prompt": prompt,
        "stream": False
    })
    r.raise_for_status()
    return {"answer": r.json().get("response", "(sin respuesta)")}
```

Instalar, indexar y ejecutar:

```
pip install -r requirements_rag.txt
python index.py
uvicorn app_rag:app --host 0.0.0.0 --port 8080
```

Pruebas:

```
curl http://localhost:8080/health
curl -X POST http://localhost:8080/chat -H "Content-Type: application/json" -d
"{"q":"¿Qué dice la política?"}"
```

```
$body = @{ q = "¿Qué dice la política?" } | ConvertTo-Json$r = Invoke-
RestMethod -Uri "http://localhost:8080/chat" -Method POST -Body $body -
ContentType 'application/json; charset=utf-8'
```

```
$r.answer
```

6) Opción B — API con OpenAI (requiere API key)

Crea el archivo .env con tu clave y la aplicación FastAPI que llama a OpenAI.

.env:

```
OPENAI_API_KEY=sk-xxxxx
```

requirements_openai.txt:

```
fastapi
uvicorn
pydantic
openai
python-dotenv
```

app_openai.py:

```
import os
from fastapi import FastAPI
from pydantic import BaseModel
from dotenv import load_dotenv
from openai import OpenAI
```

```
load_dotenv()
client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

app = FastAPI()

class Query(BaseModel):
    q: str

@app.get("/health")
def health():
    return {"status": "ok"}

@app.post("/chat")
def chat(q: Query):
    resp = client.chat.completions.create(
        model="gpt-4o-mini",
        messages=[{"role": "user", "content": q.q}]
    )
    return {"answer": resp.choices[0].message.content}
```

Instalar y ejecutar:

```
pip install -r requirements_openai.txt
uvicorn app_openai:app --host 0.0.0.0 --port 8080
```

Prueba:

```
curl -X POST http://localhost:8080/chat -H "Content-Type: application/json" -d
"{"q":"Resume RAG en 3 puntos"}"

$body = @{ q = " Resume RAG en 3 puntos" } | ConvertTo-Json

$r = Invoke-RestMethod -Uri "http://localhost:8080/chat" -Method POST -Body
$body -ContentType 'application/json; charset=utf-8'

$r.answer
```

7) Dataset de ejemplo y estructura de proyecto

Estructura sugerida del proyecto (cuando usas RAG):

```
llm-local/
  data/
```

```
    faq_llm.txt
    policy_llm.txt
    howto_llm.txt
  requirements.txt
  app.py
  requirements_rag.txt
  index.py
  app_rag.py
  requirements_openai.txt
  app_openai.py
  .env.example
```

Puedes usar los archivos de ejemplo del ZIP que te compartí para no escribir todo a mano.

8) Verificación (smoke tests)

```
# Ver API sin RAG (Ollama)
curl http://localhost:8080/health
curl -X POST http://localhost:8080/chat -H "Content-Type: application/json" -d
"{"q":"Hola"}"

# Ver API con RAG (Ollama + Chroma)
curl http://localhost:8080/health
curl -X POST http://localhost:8080/chat -H "Content-Type: application/json" -d
"{"q":"¿Qué dice la política?"}"

# Ver API con OpenAI
curl -X POST http://localhost:8080/chat -H "Content-Type: application/json" -d
"{"q":"Resume RAG"}"
```

9) Troubleshooting y buenas prácticas

• Ollama no responde → inicia la app o `ollama serve`; verifica `ollama --version`.

• Modelos faltan → `ollama pull llama3` y `ollama pull nomic-embed-text`.

• Puerto 8080 ocupado → cambiar a `--port 8081` en uvicorn.

• Venv no activa en Windows → `Set-ExecutionPolicy RemoteSigned -Scope CurrentUser` y reintenta activar.

• OpenAI 401/SSL → revisa `.env`, conexión y que la clave sea válida.

• Calidad RAG baja → aplica chunking por tokens, agrega metadatos (fuente/título/fecha), ajusta tamaño/overlap.

10) Mejora de RAG — Chunking por tokens con tiktoken

Dividir documentos en fragmentos solapados (“chunks”) mejora la recuperación y precisión. Ajusta el tamaño y el solapamiento según tu dominio (300–600 tokens y 40–100 de solape son buenos puntos de partida).

Instalación:

pip install tiktoken

Función de chunking (añádela a tu pipeline de ingesta):

```
import tiktoken

def chunk_text(text: str, chunk_size_tokens=400, chunk_overlap_tokens=60,
encoding_name="cl100k_base"):
    enc = tiktoken.get_encoding(encoding_name)
    tokens = enc.encode(text)
    chunks = []
    i = 0
    step = max(1, chunk_size_tokens - chunk_overlap_tokens)
    while i < len(tokens):
        window = tokens[i:i+chunk_size_tokens]
        chunks.append(enc.decode(window))
        i += step
    return chunks
```

Integración en index.py (versión con chunking):

```
import glob, os, requests, chromadb, tiktoken

def chunk_text(text: str, chunk_size_tokens=400, chunk_overlap_tokens=60,
encoding_name="cl100k_base"):
    enc = tiktoken.get_encoding(encoding_name)
    tokens = enc.encode(text)
    chunks, i = [], 0
    step = max(1, chunk_size_tokens - chunk_overlap_tokens)
    while i < len(tokens):
        window = tokens[i:i+chunk_size_tokens]
        chunks.append(enc.decode(window))
        i += step
```

```
    return chunks

def embed(text: str):
    r = requests.post("http://localhost:11434/api/embeddings", json={
        "model": "nomic-embed-text",
        "prompt": text
    })
    r.raise_for_status()
    return r.json()["embedding"]

def main():
    chroma = chromadb.Client()
    coll = chroma.get_or_create_collection("docs")
    files = glob.glob("./data/*.txt")
    if not files:
        print("No se encontraron archivos en ./data/.")
    for path in files:
        with open(path, "r", encoding="utf-8", errors="ignore") as f:
            content = f.read()
        parts = chunk_text(content, 400, 60)
        batch_docs, batch_embs, batch_ids = [], [], []
        for idx, part in enumerate(parts):
            vec = embed(part)
            batch_docs.append(part)
            batch_embs.append(vec)
            batch_ids.append(f"{os.path.basename(path)}#{idx:04d}")
        if batch_docs:
            coll.add(documents=batch_docs, embeddings=batch_embs, ids=batch_ids,
metadatas=[{"file": os.path.basename(path)}]*len(batch_docs))
            print(f"Indexado {len(batch_docs)} chunks de {os.path.basename(path)}")
    print("Indexado OK (con chunking).")

if __name__ == "__main__":
    main()
```

Recomendaciones: registra metadatos (archivo, sección, fecha); normaliza texto (remover HTML/ruido); evalúa tamaños diferentes y mide recall/precision con un set de preguntas de validación.

11) Benchmark local — latencia, tokens/s y uso

A continuación, un script de referencia para medir tiempos de respuesta (latencia p50/p95) y rendimiento aproximado en tokens/s. Úsalo para **Opción A (Ollama)** y **Opción B (OpenAI)**.

Instalación:

pip install tiktoken requests statistics python-dotenv openai

Script de benchmark (benchmark.py):

```
import time, statistics, requests, os
from typing import List
import tiktoken
from dotenv import load_dotenv
from openai import OpenAI

def count_tokens(text: str, enc_name="cl100k_base"):
    enc = tiktoken.get_encoding(enc_name)
    return len(enc.encode(text))

def bench_local_ollama(prompt: str, runs=5):
    latencies, toks = [], []
    for _ in range(runs):
        t0 = time.time()
        r = requests.post("http://localhost:11434/api/generate", json={
            "model": "llama3",
            "prompt": prompt,
            "stream": False
        })
        t1 = time.time()
        r.raise_for_status()
        resp = r.json().get("response","")
        lat = t1 - t0
        tok_out = count_tokens(resp)
        latencies.append(lat)
        toks.append(tok_out / lat if lat > 0 else 0)
    return latencies, toks

def bench_openai(prompt: str, runs=5):
    load_dotenv()
    client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
```

```
    latencies, toks = [], []
    for _ in range(runs):
        t0 = time.time()
        resp = client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role":"user","content":prompt}]
        )
        t1 = time.time()
        out = resp.choices[0].message.content
        lat = t1 - t0
        tok_out = count_tokens(out)
        latencies.append(lat)
        toks.append(tok_out / lat if lat > 0 else 0)
    return latencies, toks

def summarize(label: str, latencies: List[float], toksps: List[float]):
    print(f"=== {label} ===")
    print(f"runs={len(latencies)}")
    print("latency (s): min={:.3f} p50={:.3f} p95={:.3f} max={:.3f}".format(
        min(latencies), statistics.median(latencies),
        sorted(latencies)[int(0.95*len(latencies))-1], max(latencies)))
    print("tokens/s:    min={:.1f} p50={:.1f} p95={:.1f} max={:.1f}".format(
        min(toksps), statistics.median(toksps),
        sorted(toksps)[int(0.95*len(toksps))-1], max(toksps)))

if __name__ == "__main__":
    prompt = "Explica RAG en 3 puntos concisos."
    # Opción A (Ollama local)
    try:
        lat, tps = bench_local_ollama(prompt, runs=5)
        summarize("Ollama (llama3)", lat, tps)
    except Exception as e:
        print("Ollama bench error:", e)

    # Opción B (OpenAI)
    try:
        lat, tps = bench_openai(prompt, runs=5)
        summarize("OpenAI (gpt-4o-mini)", lat, tps)
    except Exception as e:
        print("OpenAI bench error:", e)
```

Ejecución:

1) Asegúrate de que la API local (Ollama) responde (ollama serve; modelo descargado)
2) (Opcional) Crea .env con OPENAI_API_KEY para la prueba OpenAI
python benchmark.py

Interpretación: compara latencia p50/p95 y tokens/s. Ajusta parámetros (prompt, tamaño de salida, hardware) y considera hacer 20–30 ejecuciones para mayor estabilidad.