IACADEMY

DE FUNDAMENTOS A ARQUITECTURA DE IA

MÓDULO 26

Modelos open-source en produccion

Avanzado

iacedemy.com — 2026

MÓDULO 26

Modelos open-source en produccion

Nivel: Avanzado

Autor: David Moya

Publicación: Mayo 2026

Plataforma: iacedemy.com

El panorama open-source 2026

El ecosistema de modelos open-source ha madurado enormemente. Ya no es "una alternativa barata a GPT-4". Modelos como Qwen 3.5 27B, Llama 4 y Phi-4 compiten directamente con APIs comerciales en la mayoria de tareas, con la ventaja de que corren en tu propia infraestructura.

El cambio clave fue la convergencia de tres factores: modelos cada vez mejores publicados con licencias permisivas, herramientas de serving eficientes (vLLM), y hardware GPU mas accesible (Hetzner, Lambda). Hoy puedes tener un endpoint LLM con rendimiento comparable a Claude Haiku por 170 EUR/mes en un servidor dedicado.

Las ventajas del self-hosting son claras:

- **Soberania de datos:** Ningun dato sale de tu infraestructura. Obligatorio para sectores regulados (salud, finanzas, defensa).
- **Coste predecible:** Pagas por servidor, no por token. Si procesas mas de ~50K tokens/dia, self-hosting es mas barato.
- **Latencia:** Sin viaje a un data center externo. Ideal para aplicaciones interactivas.
- **Personalizacion:** Puedes fine-tunear, cuantizar, y adaptar el modelo a tu caso exacto.
- **Sin limites de rate:** Escala con tu hardware, no con quotas de un proveedor.

Las desventajas tambien existen: mantenimiento de infraestructura, actualizaciones de modelos, y que los mejores modelos (GPT-4o, Claude Opus) siguen sin tener equivalente open-source para tareas de razonamiento complejo.

Qwen, Llama, Phi, Mistral: como elegir

No todos los modelos open-source son iguales. Cada familia tiene fortalezas y debilidades. Aqui va una comparativa practica basada en benchmarks reales de produccion, no de leaderboards academicos.

MODELO	PARAMS	FORTALEZA	DEBILIDAD	LICENCIA	IDIOMAS
Qwen 3.5 27B	27B	Mejor en espanol, coding, instrucciones complejas	VRAM alta para 27B	Apache 2.0	Multi (100+)
Llama 4 Scout	17B activos (MoE)	Razonamiento, contexto largo (10M tokens)	Llama License (restricciones)	Llama License	Multi
Phi-4	14B	Eficiencia, STEM, razonamiento	Peor en generacion creativa	MIT	EN principal
Mistral Large 2	123B	Razonamiento avanzado, multilingue	Pesado, requiere multi-GPU	Mistral Research	Multi
DeepSeek V3	671B (37B activos)	Coding, matematicas	MoE complejo de servir	MIT	EN/ZH

Recomendacion por caso de uso

- **Chatbot general en espanol:** Qwen 3.5 27B. Sin discusion el mejor para espanol.
- **Clasificacion y extraccion:** Phi-4 14B. Rapido, eficiente, suficiente para tareas estructuradas.
- **Codigo:** Qwen 3.5 27B o DeepSeek Coder V3.
- **Edge/movil:** Phi-4 3.8B o Qwen 3.5 3B.
- **Produccion con presupuesto minimo:** Qwen 3.5 7B en 4-bit. Cabe en 6GB VRAM.

vLLM: de cero a servir en 30 minutos

vLLM es el servidor de inferencia de referencia. Usa PagedAttention para gestionar memoria de forma optima, soporta batching continuo, y expone una API 100% compatible con OpenAI. Eso significa que cualquier codigo que use el SDK de OpenAI funciona sin cambios apuntando a tu servidor vLLM.

Instalacion y arranque

```
# Instalar vLLM
pip install vllm

# Servir Qwen 3.5 27B en 4-bit
vllm serve Qwen/Qwen2.5-27B-Instruct-AWQ \
  --host 0.0.0.0 \
  --port 8000 \
  --max-model-len 8192 \
  --gpu-memory-utilization 0.90 \
  --quantization awq \
  --tensor-parallel-size 1
```

Eso es todo. En 2-3 minutos tienes un endpoint OpenAI-compatible corriendo en tu GPU.

Usar desde tu codigo

```
from openai import OpenAI

# Apuntar al servidor vLLM local
client = OpenAI(
    base_url="http://localhost:8000/v1",
    api_key="not-needed"  # vLLM no requiere auth por defecto
)

response = client.chat.completions.create(
    model="Qwen/Qwen2.5-27B-Instruct-AWQ",
    messages=[
        {"role": "system", "content": "Eres un asistente tecnico."},
        {"role": "user", "content": "Explica que es PagedAttention en 2 frases."}
    ],
    temperature=0.7,
    max_tokens=256
)

print(response.choices[0].message.content)
```

Configuracion para produccion

```
# docker-compose.yml para vLLM en produccion
version: "3.8"
services:
  vllm:
    image: vllm/vllm-openai:latest
    runtime: nvidia
    ports:
      - "8000:8000"
    volumes:
      - /data/models:/models
    environment:
      - NVIDIA_VISIBLE_DEVICES=all
    command: >
      --model /models/Qwen2.5-27B-Instruct-AWQ
      --host 0.0.0.0
      --port 8000
      --max-model-len 8192
      --gpu-memory-utilization 0.90
      --quantization awq
      --max-num-seqs 32
      --enable-prefix-caching
    deploy:
      resources:
        reservations:
          devices:
            - driver: nvidia
              count: 1
              capabilities: [gpu]
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

Ollama para desarrollo local

Ollama es la herramienta perfecta para desarrollo local. Un solo binario que gestiona descarga, cuantizacion y serving de modelos. No es ideal para produccion (no tiene batching eficiente), pero para iterar rapido es imbatible.

Uso basico

```
# Instalar (macOS/Linux)
curl -fsSL https://ollama.com/install.sh | sh

# Descargar y correr un modelo
ollama run qwen2.5:7b

# Servir como API (se activa automaticamente)
curl http://localhost:11434/api/chat -d '{
  "model": "qwen2.5:7b",
  "messages": [{"role": "user", "content": "Hola"}],
  "stream": false
}'

# Listar modelos instalados
ollama list

# Crear un modelo custom con Modelfile
cat > Modelfile << 'EOF'
FROM qwen2.5:7b
SYSTEM "Eres un experto en ciberseguridad. Responde en espanol, de forma concisa."
PARAMETER temperature 0.3
PARAMETER num_ctx 4096
EOF

ollama create mi-modelo-ciber -f Modelfile
ollama run mi-modelo-ciber
```

Ollama con OpenAI SDK

Ollama tambien expone un endpoint compatible con OpenAI en el puerto 11434:

```
from openai import OpenAI

client = OpenAI(
    base_url="http://localhost:11434/v1",
    api_key="ollama"
)

# Mismo codigo que usarias con la API de OpenAI
response = client.chat.completions.create(
    model="qwen2.5:7b",
    messages=[{"role": "user", "content": "Que es un buffer overflow?"}]
)
print(response.choices[0].message.content)
```

Cuantizacion: GPTQ, AWQ, GGUF

Los modelos en precision completa (FP16) ocupan aproximadamente 2 bytes por parametro. Un modelo de 27B necesita ~54GB. Cuantizacion reduce la precision de los pesos (de 16-bit a 4-bit o 8-bit) para que quepa en menos VRAM con perdida minima de calidad.

Comparativa de formatos

FORMATO	FRAMEWORK	BITS	VRAM (27B)	CALIDAD VS FP16	VELOCIDAD
FP16 (sin cuantizar)	Cualquiera	16	~54GB	100%	Baseline
GPTQ	vLLM, TGI	4	~16GB	97-99%	Rapido (GPU)
AWQ	vLLM, TGI	4	~16GB	97-99%	Mas rapido (GPU)
GGUF Q4_K_M	llama.cpp, Ollama	4 (mixed)	~16GB	96-98%	CPU viable
GGUF Q8_0	llama.cpp, Ollama	8	~30GB	99%+	CPU viable
GGUF Q2_K	llama.cpp, Ollama	2	~9GB	85-92%	CPU viable

Recomendacion practica

- **Produccion con GPU:** AWQ 4-bit con vLLM. Mejor rendimiento en GPU NVIDIA.
- **Desarrollo local con GPU:** GPTQ o AWQ en vLLM/Ollama.
- **Sin GPU (CPU only):** GGUF Q4_K_M en llama.cpp/Ollama. Funciona en Mac M1+ con Metal.
- **Maxima calidad:** GGUF Q8_0 o FP16 si tienes la VRAM.

```
# Descargar un modelo AWQ ya cuantizado de HuggingFace
# La mayoria de modelos populares ya tienen versiones AWQ disponibles
pip install huggingface_hub
huggingface-cli download Qwen/Qwen2.5-27B-Instruct-AWQ --local-dir /data/models/qwen27b-awq

# Convertir un modelo a GGUF (para Ollama/llama.cpp)
pip install llama-cpp-python
python -m llama_cpp.convert_hf_to_gguf /path/to/model --outfile model.gguf --outtype q4_k_m
```

Hardware y benchmarks reales

Los benchmarks de marketing son mentira. Aqui van numeros reales medidos en produccion con vLLM sirviendo Qwen 3.5 27B AWQ 4-bit, prompt de 500 tokens, generando 200 tokens.

HARDWARE	VRAM	COSTE/MES	TOK/S (1 USER)	TOK/S (8 CONCURRENT)	LATENCIA P95
RTX 4090	24GB	~1.800 EUR (compra)	35	28 per user	~6s
L40S (Hetzner GEX44)	48GB	~170 EUR/mes	42	35 per user	~5s
A100 40GB	40GB	~2.20 EUR/h cloud	55	45 per user	~3.5s
A100 80GB	80GB	~3.50 EUR/h cloud	58	50 per user	~3s
Mac M3 Max 64GB	Shared	~4.000 EUR (compra)	18	N/A	~12s

Break-even: self-hosted vs API

A que volumen compensa self-hosting frente a usar Claude Haiku o GPT-4o-mini via API?

```
# Calculo break-even
# API (Claude Haiku): ~0.25 USD / 1M input tokens + 1.25 USD / 1M output tokens
# Promedio: ~0.75 USD / 1M tokens totales

# Self-hosted Hetzner GEX44: 170 EUR/mes fijo
# Capacidad: ~42 tok/s = ~108M tokens/mes (24/7)

# Break-even:
# 170 EUR / 0.75 USD per 1M tokens = ~226M tokens
# Pero capacidad real con concurrencia y idle time: ~50M tokens/mes util

# Conclusion: a partir de ~50K tokens/dia (1.5M/mes), self-hosting empieza a ser viable
# A 200K tokens/dia (6M/mes), self-hosting cuesta ~4x menos que API
```

El punto de inflexion real esta en ~50.000 tokens por dia. Por debajo, usa APIs. Por encima, el servidor dedicado se paga solo en 2-3 meses.

Patrones de produccion

Tener un modelo corriendo no es produccion. Produccion requiere health checks, fallbacks, logging y capacidad de actualizar sin downtime.

Patron 1: Fallback cascade

```
import httpx
from typing import Optional

MODELS = [
    {"url": "http://vllm-primary:8000/v1", "model": "qwen27b-awq", "timeout": 30},
    {"url": "http://vllm-fallback:8000/v1", "model": "qwen7b-awq", "timeout": 15},
    {"url": "https://api.anthropic.com/v1", "model": "claude-haiku", "timeout": 10},
]

async def generate_with_fallback(messages: list, max_tokens: int = 256) -> Optional[str]:
    """Intenta modelos en cascada hasta que uno responda."""
    for model_config in MODELS:
        try:
            async with httpx.AsyncClient(timeout=model_config["timeout"]) as client:
                response = await client.post(
                    f"{model_config['url']}/chat/completions",
                    json={
                        "model": model_config["model"],
                        "messages": messages,
                        "max_tokens": max_tokens,
                    }
                )
                response.raise_for_status()
                return response.json()["choices"][0]["message"]["content"]
        except (httpx.TimeoutException, httpx.HTTPStatusError) as e:
            print(f"Fallback: {model_config['model']} fallo ({e}), intentando siguiente")
            continue
    return None  # Todos fallaron
```

Patron 2: Blue-green deployment

Cuando actualizas un modelo (nueva version, nuevo fine-tune), no puedes tener downtime. El patron blue-green resuelve esto:

```
# Nginx config para blue-green con health check
upstream vllm_blue {
    server vllm-blue:8000;
}

upstream vllm_green {
    server vllm-green:8000;
}

# Variable que controla cual esta activo
map $uri $backend {
    default vllm_blue;  # Cambiar a vllm_green al deployar
}

server {
    listen 80;

    location /v1/ {
        proxy_pass http://$backend;
        proxy_set_header Host $host;
        proxy_read_timeout 60s;
    }

    location /health {
        proxy_pass http://$backend/health;
    }
}
```

El flujo es: levantar green con el nuevo modelo, verificar health check, cambiar el upstream en nginx (reload sin downtime), y apagar blue cuando confirmas que green funciona.

Patron 3: Circuit breaker

```
import time
from dataclasses import dataclass

@dataclass
class CircuitBreaker:
    failure_threshold: int = 5
    recovery_timeout: int = 60
    failures: int = 0
    last_failure: float = 0
    state: str = "closed"  # closed, open, half-open

    def can_execute(self) -> bool:
        if self.state == "closed":
            return True
        if self.state == "open":
            if time.time() - self.last_failure > self.recovery_timeout:
                self.state = "half-open"
                return True
            return False
        return True  # half-open: permit one request

    def record_success(self):
        self.failures = 0
        self.state = "closed"

    def record_failure(self):
        self.failures += 1
        self.last_failure = time.time()
        if self.failures >= self.failure_threshold:
            self.state = "open"

# Uso con vLLM
breaker = CircuitBreaker(failure_threshold=3, recovery_timeout=30)

async def safe_generate(messages):
    if not breaker.can_execute():
        return await fallback_api(messages)  # Ir directo al fallback
    try:
        result = await vllm_generate(messages)
        breaker.record_success()
        return result
    except Exception:
        breaker.record_failure()
        return await fallback_api(messages)
```

Licencias y compliance

No todos los modelos "open-source" son iguales legalmente. Antes de usar un modelo en produccion, verifica la licencia.

MODELO	LICENCIA	USO COMERCIAL	RESTRICCIONES
Qwen 3.5	Apache 2.0	Si, sin limites	Ninguna
Phi-4	MIT	Si, sin limites	Ninguna
Mistral (small/large)	Apache 2.0	Si, sin limites	Ninguna
Llama 4	Llama License	Si (con condiciones)	700M MAU limit, no competir con Meta
DeepSeek V3	MIT	Si, sin limites	Ninguna
Gemma 2	Gemma License	Si (con condiciones)	No generar contenido para menores, otros

> **Consejo legal**
>
> Si vas a usar modelos en produccion para un negocio, usa modelos Apache 2.0 o MIT. Te ahorras problemas legales. Qwen y Phi son las opciones mas seguras juridicamente. Llama tiene restricciones que pueden afectarte si creces (700M MAU limit).

Compliance con regulacion EU (AI Act)

El AI Act europeo clasifica los sistemas de IA por riesgo. Si usas modelos open-source en produccion dentro de la UE, ten en cuenta:

- Documentar que modelo usas, que version, y para que tarea
- Mantener logs de inferencia durante el periodo regulatorio (90 dias minimo)
- Si el uso es "alto riesgo" (salud, legal, RRHH), necesitas evaluacion de conformidad
- Self-hosting facilita el compliance: tienes control total sobre los datos y el modelo

Ejercicio practico

Ejercicio: Despliega un modelo con vLLM y benchmarkea

Objetivo: Poner en marcha un servidor vLLM con Qwen 7B y medir rendimiento real.

Paso 1: Setup

Si tienes GPU NVIDIA, instala vLLM directamente. Si no, usa Google Colab (T4 gratis) o Vast.ai (~0.20 EUR/h por una RTX 3090).

```
# Opcion A: Local con GPU
pip install vllm
vllm serve Qwen/Qwen2.5-7B-Instruct-AWQ --port 8000 --quantization awq

# Opcion B: Ollama (Mac/Linux, sin GPU NVIDIA)
ollama pull qwen2.5:7b
# API disponible en http://localhost:11434
```

Paso 2: Benchmark basico

```
import time
import httpx

async def benchmark(n_requests: int = 20):
    url = "http://localhost:8000/v1/chat/completions"
    payload = {
        "model": "Qwen/Qwen2.5-7B-Instruct-AWQ",
        "messages": [{"role": "user", "content": "Escribe 3 ventajas de Python"}],
        "max_tokens": 150
    }

    latencies = []
    async with httpx.AsyncClient(timeout=60) as client:
        for i in range(n_requests):
            start = time.time()
            resp = await client.post(url, json=payload)
            latencies.append(time.time() - start)

    print(f"Requests: {n_requests}")
    print(f"Latencia media: {sum(latencies)/len(latencies):.2f}s")
    print(f"Latencia P95: {sorted(latencies)[int(0.95*len(latencies))]:.2f}s")
    print(f"Throughput: {n_requests/sum(latencies):.1f} req/s")

import asyncio
asyncio.run(benchmark())
```

Paso 3: Comparar con API

Ejecuta el mismo benchmark contra la API de OpenAI o Anthropic (con la misma tarea). Compara latencia, throughput y coste por request.

Paso 4: Documentar

Crea una tabla comparativa con: modelo, cuantizacion, hardware, latencia media, throughput, coste estimado mensual para 10K requests/dia.

Entregable: Screenshot del servidor corriendo + tabla comparativa self-hosted vs API con tu hardware real.

Conclusiones clave

Key takeaways del M26

1. Qwen 3.5 27B es el mejor modelo open-source para espanol y coding en 2026. Apache 2.0, sin restricciones.
2. vLLM es el estandar para produccion: API compatible con OpenAI, batching eficiente, y PagedAttention.
3. Ollama es perfecto para desarrollo local pero NO para produccion (sin batching, sin multi-GPU).
4. AWQ 4-bit es la cuantizacion optima para GPU: 97-99% de calidad con 1/4 de VRAM.
5. Break-even self-hosted vs API: ~50K tokens/dia. Por debajo, usa APIs. Por encima, self-hosting gana.
6. Siempre implementa fallback cascade y circuit breakers. El modelo local se cae, ten un plan B.
7. Usa SOLO modelos Apache 2.0 o MIT en produccion comercial. Llama License tiene restricciones ocultas.