LO ESENCIAL · **Módulo 4 · Inferencia local**

Del runtime de escritorio al servidor con concurrencia. Medir antes de opinar.

4.1 · Runtime de un usuario contra servidor con concurrencia

Inferencia local

Objetivos

- Elegir runtime según usuarios simultáneos
- Explicar batching continuo y caché de prefijos

Dos familias de software, dos problemas distintos

Todo lo que sirve un modelo en local cae en una de dos familias. Confundirlas es la causa número uno de «funcionaba en la demo y se cayó en la oficina».

	Runtime de escritorio	Servidor de inferencia
Para quién	**un** usuario, o pocos en secuencia	**varios** usuarios a la vez
Cómo atiende	una petición tras otra	*batching continuo*: mezcla peticiones el mismo paso
Caché de prefijos	parcial o ninguna	sí: reutiliza el prefill de instrucciones documentos comunes
Formatos	GGUF cuantizado; corre en CPU, Metal y CUDA	pesos en FP16/BF16 o cuantizacion propias; **CUDA** en la práctica
Instalación	un comando	contenedor, drivers, memoria reservad
Ejemplos `[VOLÁTIL — 2026-09]`	llama.cpp y sus envoltorios (Ollama, LM Studio…)	vLLM, SGLang, TGI
Perfil	A y B	B (justo) y C

Las dos exponen una **API compatible con OpenAI** (`/v1/chat/completions`). Esa es la interfaz neutral del curso: **el asistente habla con esa API y no sabe qué hay detrás.** Cambiar de runtime a servidor —o de modelo— no toca el código del asistente.

Qué hace el batching continuo

Un runtime de escritorio procesa la petición 1 completa, luego la 2. Con cinco usuarios, el quinto espera a los otros cuatro: TTFT de decenas de segundos.

Un servidor con batching continuo mete los tokens de las cinco peticiones **en el mismo paso** de la GPU. Como generar un token es leer los pesos una vez (módulo 1), leerlos para cinco secuencias a la vez cuesta casi lo mismo que para una. El resultado: cinco usuarios con TTFT parecido al de uno. Eso —no un modelo mejor— es lo que compra el cliente cuando pasa de B a C.

Caché de prefijos

El asistente manda siempre las mismas instrucciones (2 000 tokens) y, en una sesión, el mismo documento. Sin caché, cada petición paga ese prefill. Con caché, el servidor reconoce el prefijo idéntico y salta directo a la parte nueva. En agentes de diez pasos, es la diferencia entre usable e inusable (módulo 1.2).

> **PROCEDIMIENTO · Laboratorio 4.1 · levantar los dos y compararlos**

```
cd ~/labs/el-tornillo && mkdir -p labs/04-01 && cd labs/04-01
# 1) runtime de escritorio (perfil A/B) — un comando, API en :8080
llama-server -m <modelo>.gguf -c 8192 --port 8080
# 2) servidor con concurrencia (perfil B/C, CUDA) — contenedor, API en :8000
docker run --gpus all -p 8000:8000 <imagen-del-servidor> --model <modelo> --max-model-len
    8192
```

```
# 3) la misma petición a los dos
for p in 8080 8000; do curl -s localhost:$p/v1/chat/completions -H 'content-type:
    application/json' \
  -d '{"model":"local","messages":[{"role":"user","content":"Diga hola en una línea."}]}'
    | jq -r '.choices[0].message.content'; done
```

Lo que debe ver: la misma respuesta (o equivalente) por los dos puertos. El asistente no distingue. La comparación con 1 y 5 usuarios es la lección 4.4.
Entregable: los dos comandos de arranque en `labs/04-01/arranque.md` con la versión y la fecha. Commit: «Capa 4.1: dos formas de servir».

Qué se degrada en cada perfil

A — Solo runtime de escritorio; un usuario a la vez; lote nocturno

B — Runtime cómodo; servidor con concurrencia **cabe** con modelos pequeños y contexto corto

C — Servidor con concurrencia y caché de prefijos; 10+ usuarios

Cuándo NO usar esto **Runtime de un usuario contra servidor con concurrencia**

- **No use un runtime de escritorio como servidor** para más de dos usuarios: se serializa.
- **No instale un servidor con concurrencia en un portátil sin GPU**: no corre, o corre peor que el runtime.
- **No ate el asistente a la API propia de un runtime** (endpoints no estándar): en el momento de cambiar, se reescribe.

PRÁCTICA · **Práctica**

1. Cinco usuarios mandan a la vez una pregunta que tarda 4 s sola. ¿TTFT del quinto en un runtime de escritorio? ¿Y en un servidor con batching, aproximadamente?
2. ¿Qué parte del contexto del asistente conviene poner **primero** para aprovechar la caché de prefijos? ¿Por qué?

Referencias

1. llama.cpp — `llama-server` y API compatible: https://github.com/ggml-org/llama.cpp

2. vLLM — documentación: https://docs.vllm.ai/ · SGLang: https://docs.sglang.ai/ `[VOLÁTIL — verificado: 2026-09]`

3. Yu, G.-I. et al. (2022). *Orca: A Distributed Serving System for Transformer-Based Generative Models*. OSDI — el origen del batching continuo.

4.2 · Cuantización: el costo real en calidad

> Objetivos
>
> - Medir la pérdida con un set fijo
> - Elegir el nivel de cuantización por tarea

Medir la pérdida, no suponerla

En la lección 0.3 se dijo que la cuantización cuesta calidad «en tareas finas». Aquí se mide con la tarea de la ferretería: **extraer campos de facturas**. Un método, tres cuantizaciones, un número por cada una.

El set fijo

30 facturas de proveedor (reales, anonimizadas) con sus campos anotados a mano: número, fecha, proveedor, NIT, subtotal, IVA, total. `labs/04-02/set/` con un `esperado.json`.

La métrica es **exactitud por campo**: el campo vale 1 si es idéntico al anotado, 0 si no. No hay «casi».

> PROCEDIMIENTO · **Laboratorio 4.2**

```
cd ~/labs/el-tornillo && mkdir -p labs/04-02 && cd labs/04-02
for q in Q8_0 Q6_K Q4_K_M; do
  llama-server -m modelo-${q}.gguf -c 8192 --port 8080 &  PID=$!; sleep 15
  python3 extraer.py --set set/ --salida resultados-${q}.json      # misma instrucción,
    mismo esquema
  kill $PID; sleep 3
done
python3 - <<'PY'
import json
esp = json.load(open("set/esperado.json"))
for q in ["Q8_0","Q6_K","Q4_K_M"]:
```

```
    res = json.load(open(f"resultados-{q}.json"))
    campos = ["numero","fecha","proveedor","nit","subtotal","iva","total"]
    ok = sum(res[f][c] == esp[f][c] for f in esp for c in campos)
    print(f"{q:7s} exactitud por campo: {ok/(len(esp)*len(campos))*100:5.1f} %")
PY
```

Lo que debe ver: una tabla con tres porcentajes. Un resultado típico `[VOLÁTIL — verificado: 2026-09]` con un modelo 8B en esta tarea: Q8 ≈ 96–98 %, Q6 ≈ 95–97 %, Q4 ≈ 90–94 %. **Los números de usted serán otros**: por eso se mide.

Y con esa tabla se toma la decisión, no con un foro:

Si…	Entonces
Q4 pierde menos de 1 punto	Q4: cabe más contexto y más usuarios
Q4 pierde 3–6 puntos	Q6 o Q8, aunque quepan menos usuarios; o un modelo más pequeño en Q8
Q8 tampoco llega al objetivo	el problema no es la cuantización: es el prompt, el esquema o el OCR (módulos 5 y 7)

Entregable: `resultados-*.json` y una tabla en `labs/04-02/decision.md`. Commit: «Capa 4.2: cuantización medida».

Qué se degrada en cada perfil

A — Solo Q4 (y Q6 con paciencia); la medición tarda horas, pero es la misma

B — Q6/Q8 en modelos medianos; se mide en minutos

C — Q8 o F16 en modelos medianos; se puede medir contra la referencia sin cuantizar

Cuándo NO usar esto **Cuantización: el costo real en calidad**

- No mida con 5 facturas: el ruido es mayor que la diferencia. Mínimo 30; mejor 100.
- No compare cuantizaciones de **modelos distintos**: eso es otra pregunta (¿cuál modelo?), con el mismo método.
- No cuantice por debajo de Q4 para tareas de cifras.

PRÁCTICA · **Práctica**

1. Con su tabla, ¿cuántas facturas al mes leería mal cada cuantización si el cliente procesa 400? Póngale un costo en pesos a corregir cada una.
2. ¿Qué cambia si la tarea es «resumir un contrato» en vez de «extraer cifras»?

Referencias

1. Dettmers, T. y Zettlemoyer, L. (2023). *The case for 4-bit precision: k-bit Inference Scaling Laws.* ICML — arXiv:2212.09720.

2. llama.cpp — tipos de cuantización y `llama-perplexity`: https://github.com/ggml-org/llama.cpp

3. Módulo 12 de este curso — la misma medición, convertida en eval permanente.

4.3 · Medir: tokens/s, TTFT y prefill

> Objetivos
>
> - Construir un banco de pruebas reproducible
> - Leer los tres números de una corrida

Un banco de pruebas que cualquiera pueda repetir

Tres números por corrida, en un JSON, con fecha, versión del runtime, modelo, cuantización y hardware. Sin ese JSON no hubo medición: hubo una impresión.

Número	Qué es	Cómo se obtiene con la API estándar
TTFT	ms hasta el primer token	tiempo desde el envío hasta el primer `data:` con contenido (streaming)
tokens/s de decode	velocidad de escritura	tokens de la respuesta ÷ (tiempo total − TTFT)
tokens/s de prefill	velocidad de lectura	tokens de entrada ÷ TTFT (aproximación válida si la entrada es larga)

> PROCEDIMIENTO · **Laboratorio 4.3** · `bench.sh`

```
cd ~/labs/el-tornillo && mkdir -p labs/04-03 && cd labs/04-03
cat > bench.py <<'PY'
import time, json, sys, requests, argparse, datetime, platform
ap = argparse.ArgumentParser(); ap.add_argument("--url", default="http://localhost:8080/v1
    /chat/completions")
ap.add_argument("--entrada", default="entrada-6k.txt"); ap.add_argument("--modelo",
    default="local")
ap.add_argument("--salida", default="bench.json"); a = ap.parse_args()
texto = open(a.entrada).read()
```

```
t0 = time.perf_counter(); ttft = None; n = 0
with requests.post(a.url, json={"model": a.modelo, "stream": True, "max_tokens": 200,
      "messages": [{"role":"user","content": texto + "\n\nResuma en 150 palabras."}]},
     stream=True) as r:
    for linea in r.iter_lines():
        if not linea.startswith(b"data:") or b"[DONE]" in linea: continue
        d = json.loads(linea[5:])
        if d["choices"][0]["delta"].get("content"):
            if ttft is None: ttft = time.perf_counter() - t0
            n += 1
total = time.perf_counter() - t0
res = {"fecha": datetime.datetime.now().isoformat(timespec="seconds"), "host": platform.
     node(),
       "modelo": a.modelo, "entrada_chars": len(texto), "ttft_s": round(ttft,3),
       "decode_tok_s": round(n/(total-ttft),1), "tokens_salida": n, "total_s": round(total
     ,2)}
json.dump(res, open(a.salida,"w"), indent=2); print(json.dumps(res, ensure_ascii=False))
PY
python3 bench.py --entrada entrada-6k.txt --salida bench-$(date +%F).json
```

Lo que debe ver: un JSON con `ttft_s` y `decode_tok_s`. Se corre **tres veces** y se guarda la mediana. Se repite cada vez que cambie algo: modelo, cuantización, runtime, driver, contexto.

Entregable: `bench.py`, tres JSON, y `labs/04-03/historial.md` con una fila por corrida. Commit: «Capa 4.3: banco de pruebas».

Qué se degrada en cada perfil

A — 5–15 tok/s

B — 30–80 tok/s

C — 80–200 tok/s

`[VOLÁTIL — verificado: 2026-09]` — cifras de referencia; las suyas van al historial.

Cuándo NO usar esto **Medir: tokens/s, TTFT y prefill**

- No mida con la GPU compartida con otra cosa (el navegador con video, otro modelo cargado).
- No compare TTFT de entradas de distinto tamaño.
- No publique un número sin su JSON.

PRÁCTICA · Práctica

1. Corra el banco con `--max_tokens 50` y con 500. ¿Qué número cambia y cuál no? ¿Por qué?
2. Diseñe el historial para que un cambio de driver se note como una regresión.

Referencias

1. OpenAI — referencia de la API de chat (formato de streaming que los runtimes locales imitan): https://platform.openai.com/docs/api-reference/chat
2. Módulo 12 de este curso — canarios nocturnos que corren este banco solos.

4.4 · Por qué «funciona en mi máquina» muere con cinco usuarios

> Objetivos
>
> - Hacer una prueba de carga con cinco usuarios
> - Diagnosticar colas y saturación

El experimento que hay que hacer antes de vender

Un asistente que responde en 3 segundos a **una** persona puede tardar 40 con **cinco**. El cliente no compra «funciona»; compra «funciona el martes a las 10, cuando los tres están usándolo». Este laboratorio lo simula.

PROCEDIMIENTO · **Laboratorio 4.4 · prueba de carga**

```
# carga.py — N usuarios simultáneos contra la API; reporta TTFT de cada uno
import asyncio, time, json, httpx, sys
URL = sys.argv[1] if len(sys.argv) > 1 else "http://localhost:8080/v1/chat/completions"
N   = int(sys.argv[2]) if len(sys.argv) > 2 else 5
doc = open("entrada-3k.txt").read()

async def usuario(i, cliente):
    t0 = time.perf_counter(); ttft = None
```

```
    async with cliente.stream("POST", URL, json={"model":"local","stream":True,"max_tokens
    ":120,
           "messages":[{"role":"user","content": doc + f"\n\nPregunta {i}: ¿cuál es el
    total?"}]}) as r:
        async for linea in r.aiter_lines():
            if linea.startswith("data:") and '"content"' in linea and ttft is None:
                ttft = time.perf_counter() - t0
    return i, round(ttft, 2), round(time.perf_counter() - t0, 2)

async def main():
    async with httpx.AsyncClient(timeout=300) as c:
        res = await asyncio.gather(*[usuario(i, c) for i in range(N)])
    for i, ttft, total in res: print(f"usuario {i}: TTFT {ttft:6.2f} s  total {total:6.2f}
     s")
    print("TTFT máximo:", max(r[1] for r in res), "s")
asyncio.run(main())

python3 carga.py http://localhost:8080/v1/chat/completions 1
python3 carga.py http://localhost:8080/v1/chat/completions 5
python3 carga.py http://localhost:8000/v1/chat/completions 5    # el servidor con
    concurrencia
```

Lo que debe ver `[VOLÁTIL — verificado: 2026-09]`, orden de magnitud en perfil B con un 8B Q4:

	1 usuario	5 usuarios
Runtime de escritorio	TTFT ~3 s	usuario 4: ~15 s; usuario 5: **~19 s** (cola)
Servidor con batching	TTFT ~3 s	todos entre 3,5 y 5 s

Diagnóstico: ¿cola o saturación?

Síntoma	Causa	Qué se hace
TTFT crece **linealmente** con los usuarios	cola: se atiende uno por uno	servidor con batching (4.1)
TTFT crece y el decode **baja** para todos	saturación: la GPU no da más	menos contexto, modelo más pequeño, más GPU
errores de memoria con 5	la caché de contexto × 5 no cabe	reducir `max-model-len` o cuantizar la caché

Lo que se le dice al cliente

«Con el equipo de fase 2, cinco personas a la vez esperan menos de 5 segundos la primera palabra. Con más de diez, hay que pasar a fase 3.» Esa frase sale de esta tabla y va en la propuesta (módulo 14). Sin la tabla, es una promesa.

Entregable: `carga.py` y `labs/04-04/carga.md` con las tres tablas. Commit: «Capa 4: el asistente aguanta cinco».

Qué se degrada en cada perfil

A — Un usuario. La prueba de 5 sirve para **demostrarle al cliente** por qué hace falta la fase 2

B — 3–5 usuarios con batching y contexto corto

C — 10+

Cuándo NO usar esto **Por qué «funciona en mi máquina» muere con cinco usuarios**

- No pruebe carga contra el equipo del cliente en horario laboral.
- No extrapole de 5 a 50 usuarios: a partir de cierto punto la memoria, no el cómputo, es el techo.

PRÁCTICA · Práctica

1. Con sus tablas, ¿a cuántos usuarios simultáneos se compromete por contrato? ¿Con qué margen?
2. ¿Qué cuota por usuario (peticiones/minuto) evitaría que uno solo sature a los demás? Módulo 11.

Referencias

1. Kwon, W. et al. (2023). PagedAttention — arXiv:2309.06180.
2. httpx — cliente HTTP asíncrono: https://www.python-httpx.org/
3. Módulo 11 de este curso — cuotas por usuario y trazas atribuidas.