Facsímil 04

La caja de herramientas

Capítulo 06

Cloud frente a local: privacidad, latencia y coste

686f6c61 · 2026

Capítulo 06: Cloud frente a local: privacidad, latencia y coste

La decisión que no cabe en un eslogan

Después de montar un modelo local, aparece una tentación muy humana: convertirlo en bandera. “Local es privado”. “Cloud es escalable”. “Local es barato”. “Cloud es caro”. Ninguna de esas frases aguanta mucho si la llevas a producción.

Venimos del capítulo 05, donde vimos que un modelo local es una pila: pesos, runtime, memoria, API, configuración y evaluación. Ahora comparamos esa pila con una API gestionada o una plataforma cloud. No para escoger por ideología, sino para responder una pregunta de ingeniería: **dónde debe ejecutarse esta inferencia, con estos datos, este volumen, esta latencia, este presupuesto y este nivel de operación**.

La idea central es esta: **local y cloud no son bandos; son posiciones distintas dentro de una arquitectura**.

Estado del arte con fecha de corte

Fecha de corte: 10 de junio de 2026.
Fuentes consultadas ese día: documentación oficial de controles de datos y precios de OpenAI, Anthropic, Google Cloud Vertex AI y Amazon Bedrock; documentación oficial de latencia; y artículos académicos sobre latencia de cola y edge computing.

Lo estable es el método: mirar frontera de confianza, retención, región, latencia, coste total, elasticidad, operación y plan de salida. Lo cambiante son precios, modelos disponibles, regiones, controles de retención, límites de API, multiplicadores por residencia, descuentos y disponibilidad de hardware.

FUENTE	QUÉ APORTA	CÓMO USARLA
OpenAI data controls.[1]	Explica uso de datos, retención por endpoint y controles como retención cero o monitorización modificada cuando aplica.	Para no decir “API” sin revisar qué se guarda y durante cuánto tiempo.
OpenAI models API.[2]	Devuelve modelos disponibles para una clave y metadatos básicos como `id`, `created` y `owned_by`.	Para no usar identificadores copiados de ejemplos viejos.
OpenAI pricing.[3]	Precios por tokens, cache, batch y modalidades.	Para calcular coste por flujo real, no por intuición.
OpenAI latency optimization.[4]	Criterios para reducir latencia de aplicaciones con modelos.	Para separar tokens, streaming, modelo y arquitectura.
Anthropic data retention.[5]	Diferencia arreglos de retención, alcance de ZDR y funciones que necesitan almacenamiento.	Para preguntar qué modo contractual tienes, no qué marca usas.
Anthropic models API.[6]	Lista modelos disponibles con paginación y fechas de creación.	Para separar “familia Claude” de identificador concreto de API.
Anthropic pricing.[7]	Precios por entrada, salida, cache y geografía.	Para ver que cache y residencia también cambian coste.
Gemini models.[8]	Lista modelos, modalidades y patrones de versión estable, preview, latest y experimental.	Para fijar versiones estables y no depender de alias que cambian.
Vertex AI data governance.[9]	Controles de gobernanza, retención y condiciones de servicios generativos.	Para tratar cloud como contrato de datos, no solo endpoint.
Vertex AI pricing.[10]	Precios por modelo, tokens, herramientas y modalidades.	Para revisar token, imagen, vídeo, grounding y extras.
Amazon Bedrock data protection.[11]	Modelo de responsabilidad compartida, cifrado, IAM, logs y separación con proveedores de modelos.	Para entender qué controla AWS y qué sigues controlando tú.
Amazon Bedrock pricing.[12]	Precios por modelo y modo de inferencia.	Para comparar marketplaces, regiones y modalidades.

FUENTE	QUÉ APORTA	CÓMO USARLA
OpenRouter models API.[13]	Expone modelos de muchos proveedores con `context_length`, modalidades, precios, parámetros soportados y enlaces a endpoints.	Para construir inventario técnico de modelos cloud comparables.
OpenRouter routing.[14]	Permite controlar proveedores, preferencias, rutas y sustitución cuando un proveedor no encaja.	Para entender un gateway como capa de selección, no como “un modelo más”.
Ollama Cloud y API.[15]	Mantiene herramientas locales mientras ejecuta modelos grandes en el servicio cloud de Ollama; la API local usa `localhost:11434/api`, cloud `https://ollama.com/api` y compatibilidad parcial OpenAI en `/v1`.	Para distinguir “uso Ollama” de “la inferencia está en mi máquina”.
vLLM, llama.cpp, TGI y SGLang.[16]	Runtimes para servir modelos propios con APIs compatibles, batching, plantillas de chat, GPU/CPU y opciones de producción.	Para montar servidor local con parámetros, observabilidad y contrato, no solo “ejecutar un modelo”.
Alquiler de GPU.[17]	Permite alquilar VM o instancia con acelerador, pagar por tiempo, reservar capacidad o usar capacidad con descuento bajo condiciones.	Para separar coste de API de coste de infraestructura propia en cloud.
GPU clouds y VM GPU.[18]	Documentan familias GPU, billing, almacenamiento, pods dedicados, contenedores y disponibilidad.	Para entender que alquilar GPU incluye storage, imagen, datos, arranque y operación.
The Tail at Scale.[19]	Explica por qué p95 y p99 importan más que el promedio en servicios interactivos.	Para medir latencia como experiencia real, no media bonita.
Edge Computing: Vision and Challenges.[20]	Sitúa edge/local como forma de acercar cómputo a datos y usuarios.	Para entender local como ubicación arquitectónica, no como capricho.

La revisión del 10 de junio refuerza que cloud frente a local no se decide solo con precio por millón de tokens. OpenAI separa controles de datos, optimización de coste, optimización de latencia y selección de modelos; Anthropic documenta retención, residencia, uso/coste y límites de tasa; AWS Bedrock presenta la protección de datos desde responsabilidad compartida; y Vertex AI publica capacidades, release notes y SLA por servicio.[21]

La pregunta de ingeniería queda así: ¿puedo medir coste, latencia, retención, región, límites, errores y salida por proveedor con la misma rúbrica? Si no puedes, todavía no estás comparando local y cloud; estás comparando sensaciones. Un diseño serio incluye inventario de modelos, contrato de datos, presupuesto por flujo, p95/p99, plan de degradación, logs filtrados y salida de emergencia si un proveedor cambia modelo, precio o disponibilidad.

Qué no significa “privado”

Privado no significa “no usa internet”. Un portátil con un modelo local puede tener logs, copias de seguridad, extensiones, puertos abiertos, carpetas sincronizadas y usuarios con permisos amplios. Cloud no significa automáticamente “lo ve todo el proveedor”: puede haber contratos, controles de retención, cifrado, regiones, IAM, redes privadas y auditoría. La pregunta correcta no es si algo suena local o remoto; es **dónde existe texto claro, quién puede acceder, cuánto tiempo queda guardado y qué contrato lo gobierna**.

Tampoco conviene confundir privacidad con cumplimiento. Puedes proteger datos en local y aun así incumplir una política interna por falta de trazabilidad. Puedes usar cloud y cumplir mejor porque tienes auditoría, control de accesos y residencia definida. Depende del caso.

Y privacidad tampoco equivale a calidad. El modelo local más controlado puede no servir para la tarea. El modelo cloud más potente puede no encajar con los datos. La decisión empieza por la frontera de confianza, pero no termina ahí.

La frontera de confianza

Antes de hablar de coste, dibuja por dónde viajan los datos.

LUGAR DONDE EXISTE EL DATO	PREGUNTA QUE DEBES HACER	SEÑAL DE CONTROL
Navegador o app cliente	¿El usuario escribe datos sensibles?	Minimización antes de enviar.
Backend propio	¿Qué se loguea antes de llamar al modelo?	Logs filtrados, cifrado y permisos.
Proveedor cloud	¿Qué retiene el endpoint usado?	Contrato, región, DPA/BAA cuando aplique, controles de retención.
Runtime local	¿Quién puede leer disco, memoria, logs y puerto?	Usuario dedicado, permisos, `localhost`, rutas controladas.
Herramientas conectadas	¿El modelo llama sistemas externos?	Scopes mínimos, auditoría y validación de salida.
Caché o vector store	¿Se guardan prompts, fragmentos o embeddings?	TTL, borrado, cifrado y separación por cliente.

La frontera no es un punto; es una cadena. Si haces RAG local pero subes la respuesta completa a una API cloud para “mejorarla”, la frontera cambió. Si usas cloud pero anonimizas antes, haces hashing de identificadores y evitas enviar documentos completos, también cambió.

Latencia: no mires solo el promedio

La latencia total de una llamada no es un número único: es la suma de las etapas por las que pasa la petición. Esta descomposición es una identidad de contabilidad de tiempo (el total es la suma de sus partes), útil para localizar dónde duele:

$$L_{\text{total}} = L_{\text{red}} + L_{\text{cola}} + L_{\text{prefill}}(T_{\text{entrada}}) + L_{\text{decode}}(T_{\text{salida}}) + L_{\text{herramientas}} + L_{\text{postproceso}}$$

TÉRMINO	QUÉ SIGNIFICA	QUÉ CAMBIA LOCAL/CLOUD
L_{red}	Viaje entre cliente, backend y modelo.	Local puede reducirlo; cloud depende de región y red.
L_{cola}	Espera antes de ejecutar.	Cloud puede absorber picos; local se satura antes.
L_{prefill}	Procesar tokens de entrada.	Crece con contexto. RAG y documentos largos lo disparan.
L_{decode}	Generar tokens de salida.	Depende de modelo, hardware, cuantización y runtime.
$L_{\text{herramientas}}$	Consultas a bases de datos, APIs o buscadores.	A veces domina más que el modelo.
$L_{\text{postproceso}}$	Validar JSON, guardar, renderizar o reintentar.	Suele olvidarse en demos.

Dos de estos términos no son intuición de ingeniería, sino que tienen una base teórica que conviene conocer.

La cola obedece a la teoría de colas. La Ley de Little relaciona el número medio de peticiones en el sistema L, la tasa de llegada λ y el tiempo medio que cada una pasa en el sistema W:[22]

$$L = \lambda \cdot W$$

La consecuencia práctica: cuando la tasa de llegada se acerca a la capacidad de servicio, la espera en cola se dispara de forma no lineal. Por eso un servidor al 90 por ciento de utilización ya sufre colas largas, y cloud, al poder escalar réplicas, absorbe picos que saturarían a un servidor local fijo.

El modelo sigue las dos fases de la inferencia autoregresiva. L_{prefill} procesa todo el prompt de entrada de una vez; L_{decode} genera la salida token a token reutilizando la KV cache.[23] La latencia de la parte de modelo se expresa con las dos métricas estándar del serving, TTFT (*time to first token*, dominada por el prefill) y TPOT (*time per output token*, dominada por el decode):

$$L_{\text{modelo}} = \text{TTFT} + (N_{\text{salida}} - 1) \cdot \text{TPOT}$$

Esta es la descomposición que usan los sistemas de serving modernos para optimizar latencia, hasta el punto de separar prefill y decode en máquinas distintas porque tienen perfiles de cómputo opuestos.[24]

El promedio engaña. Si una app tiene 1000 usuarios y el 5 por ciento sufre esperas largas, ese 5 por ciento puede ser el que abandona. Dean y Barroso explican por qué la cola de latencias importa en sistemas a escala: cuando una petición depende de muchos componentes, la probabilidad de que al menos uno vaya lento crece, y por eso hay que mirar percentiles altos, no la media.[25] No basta con que “normalmente vaya bien”.

MÉTRICA	QUÉ MIDE	DECISIÓN
p50	Experiencia típica.	Sirve para ver sensación normal.
p95	Casos lentos frecuentes.	Útil para producto.
p99	Cola larga.	Útil para flujos críticos o muchos usuarios.
TTFT	Tiempo hasta ver el primer token.	Mejora percepción si usas streaming.
tokens/s	Velocidad de salida.	Importa en respuestas largas.
timeouts	Peticiones que no terminan a tiempo.	Marcan límites de arquitectura.

Local puede ganar si el usuario y los datos están cerca del modelo y el modelo cabe bien. Cloud puede ganar si necesita hardware optimizado, batch, escalado, modelos grandes, alta concurrencia o regiones específicas. La única forma seria de decidir es medir el mismo caso en ambas rutas.

Coste: token barato no significa sistema barato

Igual que en el capítulo 4, lo que decide no es el precio por token, sino el coste total de propiedad: la suma de todo lo que pagas por servir la función al mes.[26] En cloud, ese coste mensual para N peticiones es:

$$C_{\text{cloud}} = N \cdot \left(\frac{T_{\text{entrada}}}{10^6} P_{\text{entrada}} + \frac{T_{\text{salida}}}{10^6} P_{\text{salida}} \right) + C_{\text{cache}} + C_{\text{herramientas}} + C_{\text{almacenamiento}} + C_{\text{observabilidad}}$$

En local, el coste no escala con las peticiones sino con la capacidad: es un coste mayormente fijo, donde el hardware se amortiza a lo largo de su vida útil. La cuenta mínima es:

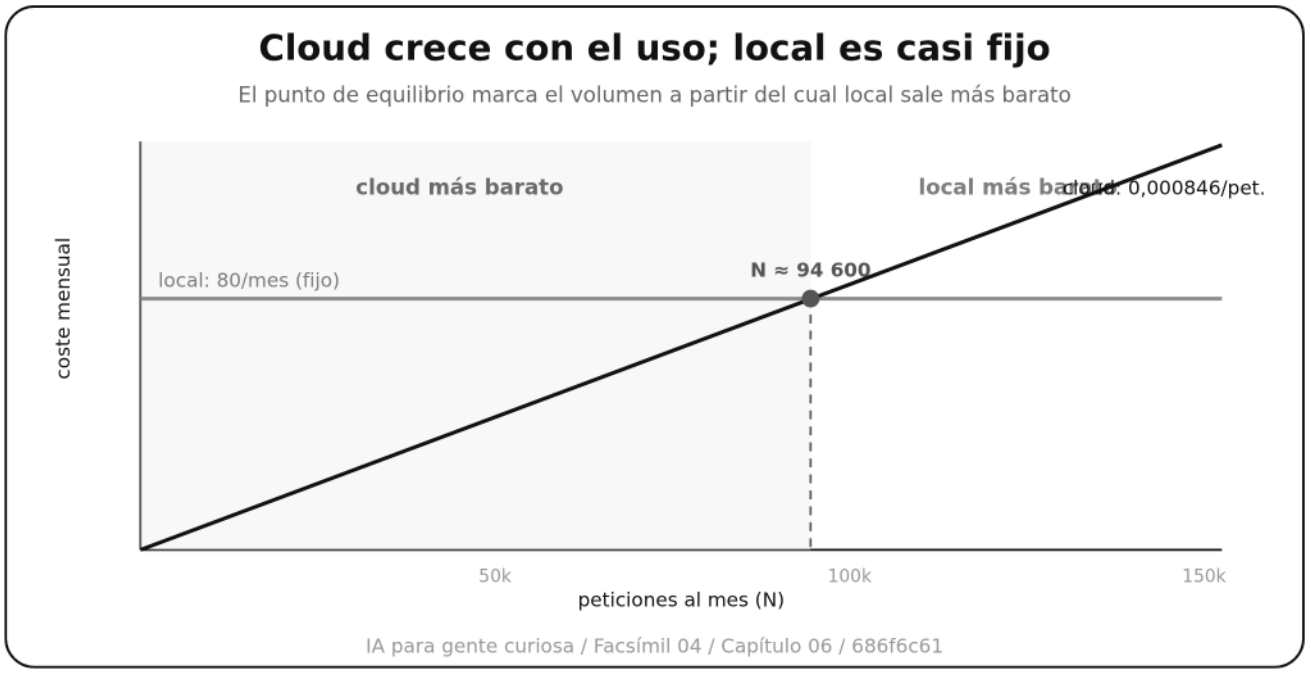
$$C_{\text{local}} = \frac{C_{\text{hardware}}}{M_{\text{amortizacion}}} + C_{\text{energia}} + C_{\text{operacion}} + C_{\text{mantenimiento}} + C_{\text{fallos}}$$

COSTE	CLOUD	LOCAL
Tokens	Visible y variable.	No se paga por token, pero sí por capacidad.
Hardware	Incluido en precio o instancia.	Compra, alquiler o servidor propio.
Picos	Elasticidad bajo demanda.	Necesitas sobredimensionar o aceptar cola.
Mantenimiento	Delegado en gran parte.	Drivers, runtime, modelos, disco, seguridad, monitorización.
Fallos	SLA, regiones, límites y dependencia externa.	Tú operas la pila.
Privacidad	Contratos y controles del proveedor.	Control físico/lógico, pero también responsabilidad propia.
Cambio de modelo	Más fácil si la API lo ofrece.	Depende de pesos, runtime y hardware disponible.

Como cloud crece con N y local es casi fijo, existe un **punto de equilibrio**: el volumen a partir del cual local sale más barato. Es el clásico análisis coste-volumen (break-even) de la contabilidad de gestión, donde el umbral es el coste fijo dividido entre el margen de contribución por unidad.[27] Se obtiene igualando ambas cuentas y despejando N:

$$N_{\text{equilibrio}} = \frac{C_{\text{local mensual}}}{\frac{T_{\text{entrada}}}{10^6} P_{\text{entrada}} + \frac{T_{\text{salida}}}{10^6} P_{\text{salida}}}$$

Pongamos números, los mismos que mide la práctica de este capítulo: un servidor local que cuesta 80 al mes y un coste cloud de 0,000846 por petición. El equilibrio está en $N_{\text{equilibrio}} = 80/0{,}000846 \approx 94\,600$ peticiones al mes. Por debajo de ese volumen, cloud es más barato; por encima, el coste fijo local se reparte entre tantas peticiones que gana. Por eso para 1200 peticiones al mes cloud es claramente más barato (estás muy por debajo del equilibrio), y solo cuando hablas de decenas de miles o millones de peticiones homogéneas empieza a compensar la infraestructura propia. Ese número no decide solo (faltan privacidad, latencia y operación), pero baja la conversación a tierra.



Tres decisiones que se confunden

Cuando alguien dice “lo hacemos local o cloud”, en realidad hay tres decisiones mezcladas:

DECISIÓN	OPCIONES	QUÉ PREGUNTA RESPONDE
Dónde corre la generación	API cloud, servidor propio, portátil, edge, híbrido.	¿Dónde se ejecuta el modelo que genera texto?
Dónde viven los datos	Base propia, documentos locales, vector store cloud, almacenamiento regional.	¿Dónde están los documentos antes y después de inferir?
Dónde se opera el producto	App local, backend propio, cloud gestionado, marketplace.	¿Quién escala, observa, actualiza y responde cuando falla?

Puedes tener generación cloud con datos minimizados localmente. Puedes tener generación local con logs sincronizados a una nube corporativa. Puedes tener embeddings locales y generación cloud. Puedes tener RAG cloud y clasificación local. Lo profesional es nombrar la arquitectura exacta.

Estrategias reales de despliegue

No hay dos caminos; hay una familia de estrategias. La pregunta no es “¿cloud o local?”, sino **qué capa quieres delegar y qué capa quieres controlar**.

ESTRATEGIA	QUÉ CONTROLAS	QUÉ DELEGAS	CUÁNDO ENCAJA	CUIDADO PRINCIPAL
API directa a un laboratorio	Prompt, contrato de salida, observabilidad de tu app.	Modelo, servidor, escalado, optimizaciones de inferencia.	Necesitas calidad alta, velocidad de desarrollo y modelos actuales.	Dependencia de precios, límites, región y política de datos.
Plataforma cloud gestionada	Región, IAM, redes, trazabilidad cloud, despliegue corporativo.	Servir el modelo y actualizar infraestructura base.	Empresa con cloud ya gobernada, auditoría y compras centralizadas.	El contrato cloud no elimina tu responsabilidad de diseño.
Gateway de modelos	Un endpoint común, selección de modelo, fallback, comparación rápida.	Relación con muchos proveedores y normalización parcial de APIs.	Quieres probar modelos o tener plan B sin reescribir toda la app.	No todos los parámetros significan lo mismo en todos los modelos.
Ollama Cloud	Herramienta local, CLI/API Ollama, cambio suave desde modelos pequeños.	Ejecución de modelos cloud de Ollama cuando no caben en tu equipo.	Quieres seguir usando flujo Ollama con modelos más grandes.	“Uso Ollama” no siempre significa “el cálculo ocurre localmente”.
Servidor propio de inferencia	Runtime, modelo exacto, hardware, red, logs, versionado y costes fijos.	Poco: tú operas casi todo.	Volumen estable, datos cerca, requisitos offline o control fino.	Operación, capacidad, actualizaciones y degradación bajo carga.
Híbrida por flujo	Qué parte va local, qué parte va cloud, qué se cachea y qué se deriva.	Solo las piezas elegidas.	La mayoría de productos reales con distintas sensibilidades y costes.	Sin reglas explícitas se convierte en una mezcla difícil de depurar.

OpenRouter entra en la tercera categoría: no es un modelo, es un **router/gateway** con una API compatible en la que eliges modelos de distintos proveedores. Su endpoint de modelos publica campos como `id`, `context_length`, modalidades, precios y parámetros soportados. Eso sirve para hacer inventario, pero no sustituye tu evaluación: dos modelos con la misma ventana de contexto pueden comportarse distinto con JSON, herramientas, español, razonamiento o latencia.

Ollama Cloud es otra cosa. Ollama puede seguir pareciendo local desde tu terminal, pero los modelos marcados como cloud se ejecutan en Ollama Cloud para poder usar modelos que no caben en tu GPU. Esto es cómodo para probar, pero cambia la frontera de confianza y el coste: la interfaz local no garantiza inferencia local.

Cómo saber qué modelos cloud tienes de verdad

Nunca elijas un modelo copiando un nombre de una entrada antigua, una captura o una demo. El identificador de modelo es una dependencia de producción. Tiene versión, fecha, capacidades, precio, modalidad, límites y, a veces, política de retirada.

PROVEEDOR O GATEWAY	CÓMO INVENTARIARLO	QUÉ MIRAR ANTES DE USARLO
OpenAI	`GET https://api.openai.com/v1/models` con tu clave.	`id`, familia, endpoint soportado, precio actual, entrada multimodal, herramientas, structured outputs y modelo recomendado para tu tarea.
Anthropic	`GET https://api.anthropic.com/v1/models` con `anthropic-version`.	`id`, `display_name`, fecha de creación, ventana de contexto, coste, soporte de tools y modo de pensamiento si aplica.
Gemini API	Página de modelos y API de listado cuando trabajas con clave.	Si el nombre es estable, preview, latest o experimental; modalidades, herramientas, contexto, rate limits y fecha de retirada.
Bedrock o Vertex AI	Catálogo de modelos dentro de la región y cuenta.	Modelo disponible por región, precio por modalidad, IAM, logging, residencia y quotas.
OpenRouter	`GET https://openrouter.ai/api/v1/models`.	`id`, proveedor, `context_length`, `pricing`, `supported_parameters`, modalidades y endpoints concretos.
Ollama local	`GET http://localhost:11434/api/tags` o `ollama list`.	Modelo descargado, tamaño, cuantización, fecha, template y si responde bien a tu contrato.
Ollama Cloud	Catálogo de modelos cloud y base URL `https://ollama.com/api`.	Si el modelo se ejecuta cloud, cuenta/API key, precio, límites y qué datos salen de tu máquina.
Servidor local OpenAI-compatible	`GET http://host:puerto/v1/models` si el runtime lo expone.	Nombre servido, plantilla de chat, límites de contexto, dtype, cuantización y parámetros aceptados.

Un inventario mínimo debería quedar así, aunque lo guardes en una hoja o en JSON:

CAMPO	EJEMPLO	POR QUÉ IMPORTA
`provider`	`openai`, `anthropic`, `openrouter`, `local-vllm`	Te dice quién opera la inferencia.
`model_id`	`gpt-...`, `claude-...`, `meta-llama/...`	Es la dependencia exacta de código.
`endpoint`	`/v1/chat/completions`, `/v1/responses`, `/api/chat`	No todos los modelos sirven en todos los endpoints.
`context_tokens`	`128000`, `1000000`, `4096`	Define cuánto texto puedes meter sin partir.
`input_price` y `output_price`	USD por millón de tokens	El coste de salida suele ser más alto que el de entrada.
`modalities`	texto, imagen, audio, embeddings	Evita elegir texto para un problema multimodal.
`tools_json_schema`	sí/no/parcial	Afecta agentes, validación y salidas estructuradas.
`retention_region`	política y región	Afecta cumplimiento y arquitectura.
`version_policy`	estable, preview, latest	Afecta reproducibilidad.
`checked_at`	`2026-06-10`	Hace explícito cuándo era verdad.

Comandos de inventario, no de producción:

```
# OpenAI: modelos accesibles por tu clave
curl https://api.openai.com/v1/models \
  -H "Authorization: Bearer $OPENAI_API_KEY"

# Anthropic: modelos accesibles por tu workspace
curl https://api.anthropic.com/v1/models \
  -H "x-api-key: $ANTHROPIC_API_KEY" \
  -H "anthropic-version: 2023-06-01"

# OpenRouter: modelos, contexto, precios y parámetros soportados
curl https://openrouter.ai/api/v1/models \
  -H "Authorization: Bearer $OPENROUTER_API_KEY"

# Ollama local: modelos descargados en tu máquina
curl http://localhost:11434/api/tags

# Servidor local compatible con OpenAI, si lo expone
curl http://localhost:8000/v1/models \
  -H "Authorization: Bearer $LOCAL_API_KEY"
```

Lo importante es no mezclar “modelo que existe”, “modelo que mi cuenta puede usar”, “modelo que mi endpoint acepta” y “modelo que mi evaluación aprueba”. Son cuatro filtros distintos.

Servir inferencia local en serio

Bajar un GGUF, escribir `ollama run` y ver una respuesta es una prueba de vida. Un servidor de inferencia es otra cosa: es una pieza de infraestructura que debe cargar pesos, reservar memoria, aceptar peticiones, encolar trabajo, generar tokens, devolver errores comprensibles, medir latencia y sobrevivir a usos repetidos.

La memoria que reserva un servidor de inferencia no es el tamaño del fichero, sino el mismo presupuesto que vimos en el capítulo 5, ahora con el batch como variable de primer orden:

$$\text{memoria}_{\text{total}} \approx \text{pesos}_{\text{cuantizados}} + \text{KV cache}(L, T, B) + \text{runtime} + \text{margen}$$

Donde L son capas, T tokens de contexto, B peticiones simultáneas y la KV cache guarda claves y valores de atención para no recalcular todo en cada token. Por eso un modelo puede “caber” con una conversación corta y romperse cuando subes contexto, batch o concurrencia. En MoE tampoco basta mirar parámetros totales: importan parámetros activados por token, memoria de pesos, comunicación entre GPUs y eficiencia del runtime.

CAPA	DECISIÓN TÉCNICA	QUÉ MIRAR
Hardware	CPU, GPU, VRAM, RAM, disco NVMe, red.	VRAM útil, ancho de banda, drivers, consumo, refrigeración y margen.
Artefacto	GGUF, safetensors, AWQ/GPTQ/FP8/BF16, revisión exacta.	Licencia, checksum, tokenizer, chat template y versión fija.
Runtime	Ollama, llama.cpp, vLLM, SGLang, TGI, LM Studio.	Batching, KV cache, cuantización, multi-GPU, tools, JSON y compatibilidad OpenAI.
Configuración	`max_model_len`, dtype, batch, concurrencia, tensor parallel, cache.	Que el límite declarado sea sostenible con tus usuarios reales.
API	`/v1/chat/completions`, `/v1/embeddings`, `/api/chat`, streaming.	Contrato estable, errores, timeouts y parámetros aceptados.
Entrada	Plantilla de chat, system prompt, roles, documentos, imágenes.	Si la plantilla es incorrecta, el modelo parece peor de lo que es.
Operación	proceso, reinicio, logs, métricas, health checks, despliegue.	p50/p95/p99, TTFT, tokens/s, cola, VRAM, errores de JSON y coste eléctrico.
Acceso	bind de red, autenticación, TLS, rate limits, CORS.	No publiques `0.0.0.0` sin proxy, clave, límites y logs útiles.
Evolución	canary, rollback, evals, cambio de modelo.	Cambiar cuantización o template puede cambiar respuestas aunque el nombre parezca igual.

Runtimes habituales:

RUNTIME	MEJOR PARA	PUNTOS TÉCNICOS
Ollama	Desarrollo local, demos, herramientas personales, API sencilla.	Muy cómodo; distingue local de cloud cuando uses modelos cloud.
LM Studio	Exploración visual, pruebas con modelos descargados, endpoint local.	Bueno para aprender; no lo confundas con una plataforma multiusuario.
llama.cpp `llama-server`	GGUF, CPU/edge, GPU modesta, despliegues ligeros.	Expone servidor HTTP compatible, opciones de host/puerto, GPU offload y endpoints de chat/embeddings.
vLLM	Alto throughput en GPU, servicio multiusuario, OpenAI-compatible.	Continuous batching, KV cache eficiente, tensor parallel, cuantización y `--api-key`.
SGLang	Baja latencia, alto throughput, modelos grandes/multimodales.	Runtime optimizado, OpenAI API, RadixAttention, prefix caching y gateway.
Hugging Face TGI	Servir modelos HF con API REST y Messages API.	Streaming, tensor parallel, Prometheus/Grafana, despliegue cloud o propio.

Un arranque local mínimo con vLLM no debería terminar en “funciona”. Debería fijar modelo, dtype, contexto, nombre servido y clave:

```
python -m vllm.entrypoints.openai.api_server \
  --model Qwen/Qwen3-8B-Instruct \
  --served-model-name local-qwen3-8b \
  --host 127.0.0.1 \
  --port 8000 \
  --dtype auto \
  --max-model-len 8192 \
  --gpu-memory-utilization 0.86 \
  --api-key "$LOCAL_API_KEY"
```

Y una prueba de vida técnica debería medir contrato, streaming y coste aproximado de tokens, no solo leer una respuesta bonita:

```
URL="http://127.0.0.1:8000/v1"

curl "$URL/chat/completions" \
  -H "Authorization: Bearer $LOCAL_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{
    "model": "local-qwen3-8b",
    "messages": [
      {"role": "system", "content": "Devuelve solo JSON valido."},
      {"role": "user", "content": "Clasifica: acceso al campus virtual."}
    ],
    "temperature": 0.1,
    "max_tokens": 160,
    "stream": false
  }'
```

Si esa llamada falla, no has “fallado con IA”; has descubierto una capa concreta: modelo no cargado, plantilla incorrecta, falta de VRAM, endpoint mal expuesto, clave ausente, límite de contexto, JSON inestable o timeout. Eso es mucho más útil que una demo.

Alquilar GPUs para inferencia

Entre una API gestionada y una GPU en tu mesa existe una opción muy usada: **alquilar GPU en cloud y servir tú el modelo**. Es cloud en infraestructura, pero local en responsabilidad técnica: eliges artefacto, runtime, contenedor, plantilla, métricas, límites y política de despliegue.

Hay tres formas habituales:

FORMA	QUÉ ALQUILAS	QUÉ CONTROLAS	CUÁNDO ENCAJA	RIESGO PRINCIPAL
VM GPU clásica	Una máquina con GPU, CPU, RAM, disco y red.	Sistema operativo, Docker, runtime, modelo, API y logs.	Servicio propio estable, pruebas serias, fine-tuning ligero o inferencia dedicada.	Pagas mientras está asignada, aunque esté esperando peticiones.
Pod GPU especializado	Una instancia GPU más directa, a menudo con plantillas o contenedor propio.	Imagen, volumen, librerías, runtime y arranque.	Probar modelos abiertos, levantar endpoints rápidos, trabajos por horas.	Disponibilidad por región/GPU y coste de almacenamiento persistente.
Serverless GPU o endpoint dedicado	Workers que arrancan bajo demanda o endpoint gestionado con GPU detrás.	Menos infraestructura; normalmente controlas imagen y escalado.	Tráfico irregular, demos, colas batch, picos previsibles.	Cold start, límites de ejecución, cola y menor control fino.

AWS EC2, Google Compute Engine y Azure permiten VMs con familias aceleradas por GPU. Runpod y proveedores similares simplifican el alquiler de pods con GPU y contenedores propios. Esto no sustituye a Bedrock, Vertex AI u OpenAI: es otra capa. En vez de comprar tokens de un modelo servido por otro, alquilas capacidad y corres tu pila de inferencia.

Alquilar una GPU se factura por hora, pero el coste real incluye las horas ociosas y los extras de operación. El coste mensual es:

$$C_{\text{gpu}} = H_{\text{activa}} P_{\text{hora}} + H_{\text{idle}} P_{\text{hora}} + C_{\text{volumen}} + C_{\text{egress}} + C_{\text{imagenes}} + C_{\text{operacion}}$$

Lo que de verdad importa para comparar con cloud es convertir ese precio por hora en **coste por token**, y ahí la utilización lo cambia todo. Si una GPU genera R tokens por segundo pero solo está ocupada una fracción U del tiempo, el coste por millón de tokens generados es:

$$C_{1M} \approx \frac{P_{\text{hora}}}{3600 \cdot R_{\text{tokens/s}} \cdot U} \cdot 10^6$$

Donde $R_{\text{tokens/s}}$ es la velocidad útil del servidor y U es la utilización real. Con números: una GPU a 2,50 por hora que genera 40 tokens/s sale a $C_{1M} = 2{,}50/(3600 \cdot 40 \cdot 0{,}5) \cdot 10^6 \approx 34{,}7$ por millón de tokens si está ocupada la mitad del tiempo ($U = 0{,}5$). Si solo está ocupada el 10 por ciento ($U = 0{,}1$), ese mismo hardware cuesta **173,6** por millón: cinco veces más. Este es el punto que más se olvida: una GPU barata por hora puede ser carísima por token si está casi vacía. Por eso el alquiler de GPU solo bate al precio por token de una API cuando consigues mantener una utilización alta y sostenida.

Para inferencia, la GPU no se elige por nombre bonito:

NECESIDAD	GPU O ACELERADOR TÍPICO	POR QUÉ
Modelo 7B-13B cuantizado, baja concurrencia	L4, A10, RTX 4090/5090, RTX 6000, T4 si aceptas menor margen.	Suele bastar para pilotos, herramientas internas y modelos pequeños.
Modelo 30B-70B, contexto mayor o más usuarios	A100 80GB, H100, H200, B200 o multi-GPU.	Más VRAM, ancho de banda y margen para KV cache.
Inferencia muy optimizada en AWS	Inf2 u otros aceleradores específicos.	Pueden ser eficientes, pero exigen stack y compilación propios.
Tráfico irregular	Serverless GPU o workers autoscalados.	Pagas menos espera, pero introduces arranque en frío.
Tráfico estable	GPU dedicada con reserva o compromiso.	Mejora disponibilidad y coste si sabes que la usarás muchas horas.

El tamaño de modelo no basta. Para servir bien necesitas medir:

SEÑAL	QUÉ MIDE	DECISIÓN
VRAM libre tras cargar pesos	Margen para KV cache y batch.	Si queda poco margen, baja contexto, cuantización o concurrencia.
TTFT	Tiempo hasta primer token.	Si es alto, revisa cola, prefill, cold start o modelo demasiado grande.
tokens/s por petición	Velocidad de generación.	Compara runtime, cuantización y GPU.
throughput agregado	Tokens/s o peticiones/s con concurrencia.	Sirve para decidir batch, réplicas y autoscaling.
utilización de GPU	Porcentaje real de uso.	Si es baja, estás pagando idle; si es alta, aparecerá cola.
errores por memoria	Peticiones que fallan por VRAM/contexto.	Define límites de entrada y concurrencia.
tiempo de arranque	Descargar imagen, montar volumen, cargar pesos.	Crítico en serverless y pods efímeros.

Checklist mínimo antes de usar GPU alquilada para una API de inferencia:

- Imagen Docker reproducible con CUDA, runtime, versión de Python y dependencias fijadas.

- Modelo y tokenizer en volumen persistente o cache precalentada; no descargar 80 GB en cada arranque.
- Health check que no diga “vivo” hasta haber cargado el modelo.
- Warmup con una petición corta para inicializar kernels, plantilla y cache.
- Límite de contexto y `max_tokens` por endpoint, no solo por buena voluntad del cliente.
- Autenticación delante del runtime, aunque sea interno.
- Métricas de p50, p95, p99, TTFT, tokens/s, cola, VRAM y errores.
- Política de apagado: deallocated/delete cuando no se usa; “stopped” no siempre significa coste cero según proveedor.
- Plan de fallback: otro modelo, otra región, API gestionada o cola de espera.

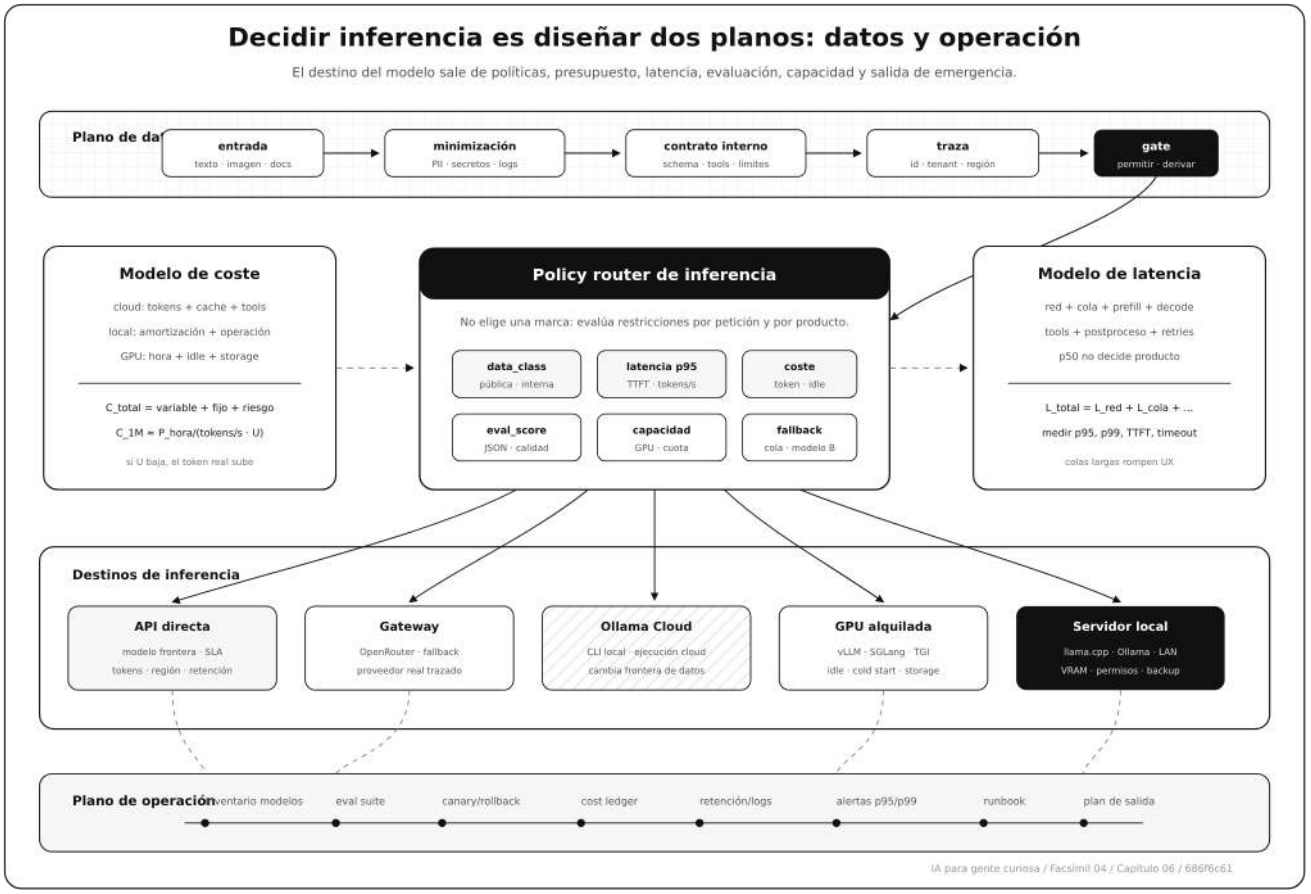
La regla práctica: si vas a usar una GPU alquilada como API, trátala como producto en producción desde el minuto uno. Si solo la enciendes para experimentar, trátala como laboratorio caro y pon alarma de apagado.

Cuándo elegir cada ruta

SITUACIÓN	RUTA PROBABLE	POR QUÉ
Prototipo rápido con usuarios internos	Cloud o LM Studio local.	Aprendes rápido sin comprar infraestructura.
Datos sensibles y flujo simple	Local o cloud con controles contractuales fuertes.	La decisión depende de política, no de marca.
Mucha concurrencia variable	Cloud.	La elasticidad suele compensar.
Volumen estable y tarea acotada	Local/propio puede competir.	Puedes amortizar hardware y optimizar.
Necesitas modelo frontera	Cloud.	Los mejores modelos no suelen estar todos como pesos descargables.
Offline, aula, demo, entorno cerrado	Local.	Funciona sin depender de red externa.
Latencia de milisegundos cerca del usuario	Local/edge si el modelo cabe.	La distancia de red importa.
Cumplimiento con región definida	Cloud regional o local controlado.	Se decide por residencia, auditoría y contrato.
Comparar muchos modelos sin reescribir clientes	Gateway como OpenRouter.	Normaliza entrada y permite inventariar precios, contexto y parámetros.
Usar herramientas Ollama con modelos que no caben	Ollama Cloud.	Mantienes flujo Ollama pero cambias la ubicación real de inferencia.
Servir modelos abiertos con control técnico	GPU alquilada con vLLM, SGLang, TGI o llama.cpp.	Controlas artefacto, runtime y API sin comprar hardware.
Tráfico con picos y largas pausas	Serverless GPU o endpoint autoscalado.	Puede reducir idle, a cambio de cold start y menos control fino.

La ruta híbrida es común: clasificación local, RAG con datos propios, generación cloud para casos difíciles, cache de respuestas frecuentes y fallback local si la API externa no está disponible. Híbrido no significa improvisado; significa que cada pieza tiene una razón.

Mapa visual de decisión



En el día a día

Imagina una universidad que quiere clasificar incidencias de estudiantes. Si el texto contiene solo categorías generales, una API cloud puede dar calidad alta y mantenimiento bajo. Si el texto incluye expedientes, datos médicos o información contractual, quizá convenga anonimizar, resumir localmente o ejecutar todo en un entorno controlado.

Imagina una asesoría que analiza miles de facturas al mes. Si el volumen es bajo, una API potente evita comprar hardware. Si el volumen es constante, el modelo local puede ahorrar coste. Pero si cada error de extracción cuesta una llamada humana, el coste real no está en tokens: está en fallos.

Imagina una app de escritorio que debe funcionar en una fábrica sin red estable. Local gana por disponibilidad. Pero si el modelo debe razonar sobre documentos complejos y actualizados, quizá necesites sincronizar, cachear o derivar algunos casos a cloud. La arquitectura buena suele ser menos épica y más concreta.

Por qué debería importarte

Porque la elección local/cloud afecta a producto, privacidad, presupuesto, operación y experiencia de usuario. No es una decisión que pueda tomar solo compras, solo legal o solo ingeniería. Hay que poner a todos mirando la misma tabla.

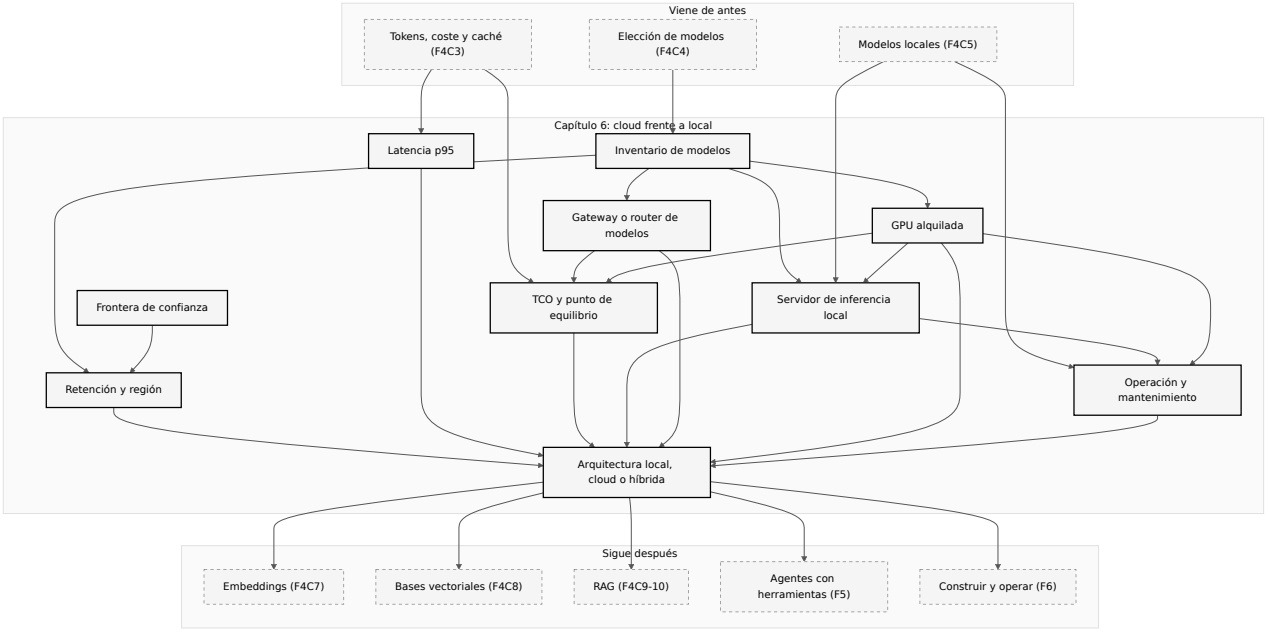
También importa porque los próximos capítulos del facsímil se apoyan en esta decisión. Los embeddings, las bases vectoriales, el RAG y las herramientas de datos pueden correr local, cloud o híbrido. Si no sabes elegir ubicación, cada capítulo posterior se convierte en una colección de piezas sin arquitectura.

Dónde solía tropezar yo

ERROR	POR QUÉ ES UN ERROR	ANTÍDOTO
Decir “local” como sinónimo de seguro	Local también tiene logs, backups, puertos y permisos.	Dibujar la frontera de confianza completa.
Decir “cloud” como sinónimo de caro	Pocas peticiones o modelos frontera pueden ser más baratos por API que operar hardware.	Calcular coste mensual, no coste emocional.
Comparar latencia media	El promedio oculta esperas largas.	Medir p50, p95, p99 y timeouts.
Olvidar el coste de operación local	Drivers, actualizaciones, disco, monitorización y guardias cuestan tiempo.	Incluir horas humanas y mantenimiento en el TCO.
No mirar la retención por endpoint	Distintas funciones pueden guardar estado de forma distinta.	Revisar documentación y contrato antes de enviar datos reales.
Construir sin plan de salida	Cambiar de proveedor o runtime tarde puede doler mucho.	Mantener pruebas comunes y contrato de API propio.
Creer que un gateway es un modelo	OpenRouter, por ejemplo, puede enrutar a modelos y proveedores distintos.	Guardar modelo, proveedor, endpoint, precio, fecha y parámetros soportados.
Confundir Ollama con inferencia local siempre	Ollama Cloud mantiene la experiencia de herramienta local, pero ejecuta modelos cloud.	Mirar si el modelo está descargado localmente o se ejecuta en cloud.
Montar servidor local sin contrato operativo	Una respuesta en terminal no mide colas, contexto, JSON, p95 ni reinicios.	Definir runtime, API, clave, métricas, límites, salud y rollback.
Alquilar GPU y olvidar el idle	Una GPU por hora puede salir cara si espera vacía.	Calcular coste por token útil con utilización real.
Confundir stopped con deallocated	En algunos clouds parar una VM no libera todo el coste asignado.	Revisar estado facturable, discos, IPs, snapshots y volúmenes.

Cómo encaja todo

Este mapa conecta la decisión cloud/local con lo que ya vimos y con lo que viene. Fíjate en que no sale de “gusto por herramientas”, sino de restricciones medibles.


Viene de antes
Tokens, coste y caché (F4C3)
Elección de modelos (F4C4)
Modelos locales (F4C5)
Capítulo 6: cloud frente a local
Latencia p95
Inventario de modelos
Gateway o router de modelos
GPU alquilada
Frontera de confianza
TCO y punto de equilibrio
Servidor de inferencia local
Retención y región
Operación y mantenimiento
Arquitectura local, cloud o híbrida
Sigue después
Embeddings (F4C7)
Bases vectoriales (F4C8)
RAG (F4C9-10)
Agentes con herramientas (F5)
Construir y operar (F6)

Vocabulario aprendido

TÉRMINO	DEFINICIÓN
Frontera de confianza	Punto o cadena donde los datos pasan a otro dominio técnico o contractual.
TCO	Coste total de propiedad, incluyendo uso, infraestructura, operación y mantenimiento.
Latencia p95	Tiempo por debajo del cual termina el 95 por ciento de peticiones.
TTFT	Tiempo hasta recibir el primer token.
Throughput	Capacidad de procesar peticiones o tokens por unidad de tiempo.
Región	Ubicación geográfica donde se procesa o guarda una carga.
Retención	Tiempo durante el que se conservan datos, logs o estado.
Residencia de datos	Restricción sobre dónde deben vivir o procesarse los datos.
Capacidad elástica	Capacidad de escalar recursos bajo demanda.
Punto de equilibrio	Volumen donde coste local y cloud se igualan bajo hipótesis dadas.
Gateway de modelos	Capa que ofrece un endpoint común y enruta peticiones a distintos proveedores o modelos.
Servidor de inferencia	Servicio que carga pesos, gestiona memoria, cola peticiones y expone una API para generar salidas.
GPU alquilada	Acelerador en cloud que pagas por tiempo para ejecutar tu propio runtime o contenedor.
KV cache	Memoria usada para guardar claves y valores de atención ya calculados durante la generación.
Modelo servido	Nombre de modelo que expone tu API; puede ser distinto del repositorio o archivo interno.

Antes de pasar página

- [ ] ¿Puedo explicar por qué local no significa automáticamente privado? (Si no, vuelve a «Qué no significa “privado”».)

- [ ] ¿Puedo explicar por qué cloud no significa automáticamente caro? (Si no, vuelve a «Coste: token barato no significa sistema barato».)
- [ ] ¿Sé dibujar la frontera de confianza de un flujo completo? (Si no, vuelve a «La frontera de confianza».)
- [ ] ¿Sé inventariar modelos disponibles por endpoint y no por memoria? (Si no, vuelve a «Cómo saber qué modelos cloud tienes de verdad».)
- [ ] ¿Sé distinguir API directa, plataforma cloud, gateway, Ollama Cloud y servidor propio? (Si no, vuelve a «Estrategias reales de despliegue».)
- [ ] ¿Sé calcular coste cloud con entrada, salida, cache y extras? (Si no, vuelve a «Coste: token barato no significa sistema barato».)
- [ ] ¿Sé calcular coste local incluyendo hardware, energía y operación? (Si no, vuelve a «Servir inferencia local en serio».)
- [ ] ¿Sé calcular coste de GPU alquilada incluyendo idle, storage, red y operación? (Si no, vuelve a «Alquilar GPUs para inferencia».)
- [ ] ¿Estoy midiendo p95 y no solo promedio? (Si no, vuelve a «Latencia: no mires solo el promedio».)
- [ ] ¿Sé qué datos se retienen y durante cuánto tiempo en el endpoint elegido? (Si no, vuelve a «La frontera de confianza».)
- [ ] ¿Sé qué runtime, contexto, cuantización y nombre servido tiene mi servidor local? (Si no, vuelve a «Servir inferencia local en serio».)
- [ ] ¿Sé comparar local y cloud con el mismo caso y los mismos criterios? (Si no, repasa «Cuándo elegir cada ruta».)

En resumen

IDEA FUERZA	DETALLE
Local y cloud son posiciones arquitectónicas.	No son bandos; responden a restricciones distintas.
Privacidad exige dibujar la frontera de confianza.	Hay que saber dónde existe el dato, quién accede y cuánto se retiene.
La latencia útil se mide en p95 y p99.	El promedio no captura la experiencia de usuarios lentos.
El coste real es TCO.	Tokens, hardware, operación, mantenimiento, cache y herramientas cuentan.
Un gateway no es un modelo.	OpenRouter u otros routers pueden cambiar proveedor, parámetros y precio detrás de un endpoint común.
Ollama Cloud no equivale a inferencia local.	Conserva la experiencia Ollama, pero cambia ubicación, coste y frontera de datos.
Servir local en serio requiere infraestructura.	Modelo, runtime, KV cache, API, claves, métricas, colas, límites y rollback forman parte del sistema.
Alquilar GPU es cloud con responsabilidad propia.	Pagas tiempo, storage e idle; tú sirves, mides, actualizas y apagas.
La ruta híbrida suele ser la más realista.	Minimiza datos localmente, escala cloud cuando aporta y mantiene una eval común.

Para saber más

Amazon Web Services. (2026). *Amazon Bedrock pricing*. https://aws.amazon.com/bedrock/pricing/

Amazon Web Services. (2026). *Data protection in Amazon Bedrock*. https://docs.aws.amazon.com/bedrock/latest/userguide/data-protection.html

Amazon Web Services. (2026). *Amazon EC2 Pricing*. https://aws.amazon.com/ec2/pricing/

Amazon Web Services. (2026). *Specifications for Amazon EC2 accelerated computing instances*. https://docs.aws.amazon.com/ec2/latest/instancetypes/ac.html

Anthropic. (2026). *API and data retention*. https://platform.claude.com/docs/en/manage-claude/api-and-data-retention

Anthropic. (2026). *List Models*. https://platform.claude.com/docs/en/api/models/list

Anthropic. (2026). *Pricing*. https://platform.claude.com/docs/en/about-claude/pricing

Dean, J. y Barroso, L. A. (2013). *The Tail at Scale*. *Communications of the ACM, 56*(2), 74-80. https://doi.org/10.1145/2408776.2408794

Google. (2026). *Gemini API models*. https://ai.google.dev/gemini-api/docs/models

Google Cloud. (2026). *About GPU instances*. https://docs.cloud.google.com/compute/docs/gpus/about-gpus

Google Cloud. (2026). *GPU machine types*. https://docs.cloud.google.com/compute/docs/gpus

Google Cloud. (2026). *Vertex AI and zero data retention*. https://cloud.google.com/vertex-ai/generative-ai/docs/data-governance

Google Cloud. (2026). *Vertex AI pricing*. https://cloud.google.com/vertex-ai/generative-ai/pricing

Hugging Face. (2026). *Text Generation Inference: HTTP API Reference*. https://huggingface.co/docs/text-generation-inference/reference/api_reference

llama.cpp. (2026). *llama-server*. https://www.mintlify.com/ggml-org/llama.cpp/api/tools/llama-server

Microsoft Azure. (2026). *Linux Virtual Machines Pricing*. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/

Microsoft Azure. (2026). *Virtual machine sizes overview*. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/overview

Ollama. (2026). *API introduction*. https://docs.ollama.com/api/introduction

Ollama. (2026). *Cloud*. https://docs.ollama.com/cloud

Ollama. (2026). *OpenAI compatibility*. https://docs.ollama.com/api/openai-compatibility

OpenAI. (2026). *Data controls in the OpenAI platform*. https://developers.openai.com/api/docs/guides/your-data

OpenAI. (2026). *Latency optimization*. https://developers.openai.com/api/docs/guides/latency-optimization

OpenAI. (2026). *List models*. https://developers.openai.com/api/reference/resources/models/methods/list

OpenAI. (2026). *Pricing*. https://developers.openai.com/api/docs/pricing

OpenRouter. (2026). *List all models and their properties*. https://openrouter.ai/docs/api/api-reference/models/get-models

OpenRouter. (2026). *Provider routing*. https://openrouter.ai/docs/guides/routing/provider-selection

Runpod. (2026). *Cloud GPU Instances for AI Workloads*. https://www.runpod.io/product/cloud-gpus

Runpod. (2026). *Pods pricing*. https://docs.runpod.io/pods/pricing

Shi, W., Cao, J., Zhang, Q., Li, Y. y Xu, L. (2016). *Edge Computing: Vision and Challenges*. *IEEE Internet of Things Journal, 3*(5), 637-646. https://doi.org/10.1109/JIOT.2016.2579198

SGLang. (2026). *Welcome to SGLang*. https://docs.sglang.io/index.html

vLLM. (2026). *OpenAI-Compatible Server*. https://docs.vllm.ai/en/stable/serving/openai_compatible_server/

Notas

1. OpenAI. (2026). *Data controls in the OpenAI platform*. https://developers.openai.com/api/docs/guides/your-data. Consultado el 10 de junio de 2026. ↩
2. OpenAI. (2026). *List models*. https://developers.openai.com/api/reference/resources/models/methods/list. Consultado el 10 de junio de 2026. ↩
3. OpenAI. (2026). *Pricing*. https://developers.openai.com/api/docs/pricing. Consultado el 10 de junio de 2026. ↩
4. OpenAI. (2026). *Latency optimization*. https://developers.openai.com/api/docs/guides/latency-optimization. Consultado el 10 de junio de 2026. ↩
5. Anthropic. (2026). *API and data retention*. https://platform.claude.com/docs/en/manage-claude/api-and-data-retention. Consultado el 10 de junio de 2026. ↩
6. Anthropic. (2026). *List Models*. https://platform.claude.com/docs/en/api/models/list. Consultado el 10 de junio de 2026. ↩
7. Anthropic. (2026). *Pricing*. https://platform.claude.com/docs/en/about-claude/pricing. Consultado el 10 de junio de 2026. ↩

8. Google. (2026). *Gemini API models*. https://ai.google.dev/gemini-api/docs/models. Consultado el 10 de junio de 2026. ↩

9. Google Cloud. (2026). *Vertex AI and zero data retention*. https://cloud.google.com/vertex-ai/generative-ai/docs/data-governance. Consultado el 10 de junio de 2026. ↩

.0. Google Cloud. (2026). *Vertex AI pricing*. https://cloud.google.com/vertex-ai/generative-ai/pricing. Consultado el 10 de junio de 2026. ↩

.1. Amazon Web Services. (2026). *Data protection in Amazon Bedrock*. https://docs.aws.amazon.com/bedrock/latest/userguide/data-protection.html. Consultado el 10 de junio de 2026. ↩

.2. Amazon Web Services. (2026). *Amazon Bedrock pricing*. https://aws.amazon.com/bedrock/pricing/. Consultado el 10 de junio de 2026. ↩

.3. OpenRouter. (2026). *List all models and their properties*. https://openrouter.ai/docs/api/api-reference/models/get-models. Consultado el 10 de junio de 2026. ↩

.4. OpenRouter. (2026). *Provider routing*. https://openrouter.ai/docs/guides/routing/provider-selection. Consultado el 10 de junio de 2026. ↩

.5. Ollama. (2026). *Cloud*. https://docs.ollama.com/cloud. Consultado el 10 de junio de 2026. Véase también *API introduction*: https://docs.ollama.com/api/introduction y *OpenAI compatibility*: https://docs.ollama.com/api/openai-compatibility. ↩

.6. vLLM. (2026). *OpenAI-Compatible Server*. https://docs.vllm.ai/en/stable/serving/openai_compatible_server/. Consultado el 10 de junio de 2026. llama.cpp. (2026). *llama-server*. https://www.mintlify.com/ggml-org/llama.cpp/api/tools/llama-server. Hugging Face. (2026). *TGI HTTP API Reference*. https://huggingface.co/docs/text-generation-inference/reference/api_reference. SGLang. (2026). *Welcome to SGLang*. https://docs.sglang.io/index.html. ↩

.7. AWS. (2026). *Amazon EC2 Pricing*. https://aws.amazon.com/ec2/pricing/. Consultado el 10 de junio de 2026. AWS. (2026). *Specifications for Amazon EC2 accelerated computing instances*. https://docs.aws.amazon.com/ec2/latest/instancetypes/ac.html. Google Cloud. (2026). *GPU machine types*. https://docs.cloud.google.com/compute/docs/gpus. Google Cloud. (2026). *About GPU instances*. https://docs.cloud.google.com/compute/docs/gpus/about-gpus. ↩

.8. Microsoft Azure. (2026). *Virtual machine sizes overview*. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/overview. Microsoft Azure. (2026). *Linux Virtual Machines Pricing*. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/. Runpod. (2026). *Pods pricing*. https://docs.runpod.io/pods/pricing. Runpod. (2026). *Cloud GPU Instances for AI Workloads*. https://www.runpod.io/product/cloud-gpus. Consultado el 10 de junio de 2026. ↩

.9. Jeffrey Dean y Luiz André Barroso. (2013). *The Tail at Scale*. *Communications of the ACM, 56*(2), 74-80. https://doi.org/10.1145/2408776.2408794. ↩

20. Weisong Shi y otros (2016). *Edge Computing: Vision and Challenges*. *IEEE Internet of Things Journal, 3*(5), 637-646. https://doi.org/10.1109/JIOT.2016.2579198. ↩

21. Fuentes de esta revisión: OpenAI. (2026). *Cost optimization*. https://developers.openai.com/api/docs/guides/cost-optimization. OpenAI. (2026). *Model selection*. https://developers.openai.com/api/docs/guides/model-selection. Anthropic. (2026). *Usage and Cost Admin API*. https://platform.claude.com/docs/en/manage-claude/usage-cost-api. Anthropic. (2026). *Data residency*. https://platform.claude.com/docs/en/manage-claude/data-residency. Amazon Web Services. (2026). *Data protection in Amazon Bedrock*. https://docs.aws.amazon.com/bedrock/latest/userguide/data-protection.html. Google Cloud. (2026). *Vertex AI Generative AI release notes*. https://docs.cloud.google.com/vertex-ai/generative-ai/docs/release-notes. Google Cloud. (2026). *Vertex AI Generative AI SLA*. https://cloud.google.com/vertex-ai/generative-ai/sla. Todas consultadas el 10 de junio de 2026. ↩

22. John D. C. Little. (1961). *A Proof for the Queuing Formula: L = λW*. Operations Research, 9(3), 383-387. https://doi.org/10.1287/opre.9.3.383. ↩

23. Woosuk Kwon y otros (2023). *Efficient Memory Management for Large Language Model Serving with PagedAttention*. SOSP. https://arxiv.org/abs/2309.06180. ↩

24. Yinmin Zhong y otros (2024). *DistServe: Disaggregating Prefill and Decoding for Goodput-optimized Large Language Model Serving*. OSDI 2024. https://arxiv.org/abs/2401.09670. ↩

25. Jeffrey Dean y Luiz André Barroso. (2013). *The Tail at Scale*. Communications of the ACM, 56(2), 74-80. https://doi.org/10.1145/2408776.2408794. ↩

26. Lisa M. Ellram. (1995). *Total Cost of Ownership: An Analysis Approach for Purchasing*. International Journal of Physical Distribution & Logistics Management, 25(8), 4-23. https://doi.org/10.1108/09600039510099928. ↩

27. Charles T. Horngren, Srikant M. Datar y Madhav V. Rajan. (2015). *Cost Accounting: A Managerial Emphasis* (15.ª ed.). Pearson. Capítulo sobre análisis coste-volumen-beneficio. ↩